

A cluster of several mushrooms of various sizes and shapes, all glowing with a soft, ethereal white light. The mushrooms are set against a solid black background, which makes their luminous forms stand out. The lighting creates a subtle gradient, with the mushrooms appearing brighter in the center and darker towards the edges. The overall effect is dreamlike and mysterious.

DiGi-Cream

# 20 Rules For OOP In Delphi

by Marco Cantù

Most Delphi programmers use their development environment as they would use Visual Basic *[Editor throws his hands up in horror at the mere thought!]*, without realising and taking advantage of the power they have at their hands. Delphi is based on an object oriented architecture, which does not only impact the VCL structure but also each and every Delphi application.

In this article I don't want to cover the theory of OOP, but just suggest some simple rules which might help you improve the structure of your programs. These rules of thumb should be considered as suggestions, to be applied or not depending on the actual type of application you are building. My suggestion is simply to keep them in mind.

The key principle I want to underline is encapsulation. We want to create flexible and robust classes, which will allow us to change the implementation later on without affecting the rest of the program. This is not the only criterion for good OOP, but it represents the foundation, so if I actually over-stress it in this article I have some good reasons to do so.

Finally, to underline the fact that these principles should be used in our daily work by all of us Delphi programmers, I'm going to focus mainly on the development of forms, even if some of the rules equally apply to the development of components. Those who write components must consider OOP and classes as a central element. Those who use components at times forget about OOP: this article can be considered as a reminder.

## Part 1: A Form Is A Class

Programmers usually treat forms as objects, while in fact they are classes. The difference is that you can have multiple form objects based on the same form class. The confusing thing is that Delphi

creates a default global object for every form class you define. This is certainly handy for newcomers, but can turn into a bad habit.

### Rule 1: One Class, One Unit

Always remember that the `private` and `protected` portions of a class are hidden only to classes and procedures in other units. Therefore, if you want to have an effective encapsulation you should use a different unit for every class. For simple classes, inheriting one from the other, you can actually use a shared unit, but only if the number of classes is limited: Don't place a 20-classes complex hierarchy in a single unit, even if Borland does it in the VCL source code...

If you think about forms, Delphi follows the 'one class, one unit' principle by default, which is certainly handy. When adding non-form classes to a project, create new separate units.

### Rule 2: Name Components

It is very important to give a meaningful name to each form and each unit. Unluckily the two names must be different, although I tend to use similar names for the two, such as `AboutForm` and `About.pas`.

It's important to use descriptive names for components too. The most common notation is to use a few lower case initial letters for the class type, followed by the role of the component, as in `btnAdd` or `editName`. There are actually many similar notations following this style and there is really no reason to say any one of them is best, it's up to your personal taste.

### Rule 3: Name Events

It is even more important to give proper names to event handling methods. If you name the components properly, the default name of `Button1Click`, for example, becomes `btnAddClick`. Although we can guess what the method does from the button name, I think it is

way better to use a name describing the effect of the method, not the attached component. For example, the `OnClick` event of the `btnAdd` button can be named `AddToList`. This makes the code more readable, particularly when you call the event handler from another method of the class, and helps developers attach the same method to multiple events or to different components, although I have to say that using `Actions` is currently my preferred choice for non-trivial programs.

### Rule 4: Use Form Methods

If forms are classes their code is collected in methods. Besides the event handlers, which play a special role but can still be called as other methods, it is often useful to add custom methods to form classes. You can add methods performing actions and accessing to the status of the form. It is much better to add a public method to a form than to let other forms operate on its components directly.

### Rule 5: Add Form Constructors

A secondary form created at runtime can provide other specific constructors beside the default one (inherited from the `TComponent` class). If you don't need compatibility with versions of Delphi prior to 4, my suggestion is to overload the `Create` method, adding the required initialisation parameters. Listing 1 gives an example.

### Rule 6: Avoid Global Variables

Global variables (that is, variables declared in the `interface` portion of a unit) should be avoided. Here are a few suggestions to help you do this.

If you need extra data storage for a form, add some `private` fields to it. In this case each form instance will have its own copy of the data.



You might use unit variables (declared in the implementation portion of the unit) for data shared among multiple instances of the form class.

If you need data shared among forms of different types, you can share them by placing the data in the main form, or in a global object, and use methods or properties to access the data.

### **Rule 7: Never Use Form1 In TForm1**

You should never refer to a specific object in a method of the class of that object. In other words, never refer to `Form1` in a method of the `TForm1` class. If you need to refer to the current object, use the `self` keyword. Keep in mind that most of the time this is not needed, as you can refer directly to methods and data of the current object.

If you don't follow this rule, you'll get into trouble when you create multiple instances of the form.

### **Rule 8: Seldom Use Form1 In Other Forms**

Even in the code of other forms, try to avoid direct references to global objects, such as `Form1`. It is much better to declare local variables or `private` fields to refer to other forms.

For example, the main form of a program can have a `private` field referring to a dialog box. Obviously this rule becomes essential if you plan creating multiple instances of the secondary form. You can keep a list in a field of the main form, or simply use the `Forms` array of the global `Screen` object.

### **Rule 9: Remove Form1**

Actually, my suggestion is to remove the global form object which is automatically added by Delphi to the program. This is possible only if you disable the automatic creation of that form (again added by Delphi), something which I suggest you should get rid of anyway.

I think that removing the global form object is very useful for Delphi newcomers, who then won't get confused between the class

```
public
  constructor Create (Text: string); reintroduce; overload;
  constructor TFormDialog.Create(Text: string);
begin
  inherited Create (Application);
  Edit1.Text := Text;
end;
```

#### ► Listing 1

```
private
  function GetText: String;
  procedure SetText(const Value: String);
public
  property Text: String
    read GetText write SetText;
function TFormDialog.GetText: String;
begin
  Result := Edit1.Text;
end;
procedure TFormDialog.SetText(const Value: String);
begin
  Edit1.Text := Value;
end;
```

#### ► Listing 2: You can add a property to a form to expose a property of a component.

and the global object anymore. In fact, after the global object has been removed, any reference to it will result in an error.

### **Rule 10: Add Form Properties**

As I've already mentioned, when you need data for a form, add a `private` field. If you need to access this data from other classes, then add properties to the form. With this approach you will be able to change the code of the form and its data (including its user interface) without having to change the code of other forms or classes.

You should also use properties or methods to initialise a secondary form or dialog box, and to read its final state. The initialisation can also be performed using a constructor, as I have already described.

### **Rule 11: Expose Components Properties**

When you need to access the status of another form, you should not refer directly to its components. This would bind the code of other forms or classes to the user interface, which is one of the portions of an application subject to most changes. Rather, declare a form property mapped to the component property: this is accomplished with a `Get` method that reads the component status and a `Set` method that writes it.

Suppose you now change the user interface, replacing the component with another one. All you have to do is fix the `Get` and `Set` methods related with the property, you won't have to check and modify the source code of all the forms and classes which might refer to that component. You can see an example in Listing 2.

### **Rule 12: Array Properties**

If you need to handle a series of values in a form, you can declare an array property. In case this is an important information for the form you can make it also the default array property of the form, so that you can directly access its value by writing `SpecialForm[3]`.

Listing 3 shows how you can expose the items of a listbox as the default array property of the form hosting it.

### **Rule 13: Use Side-Effects In Properties**

Remember that one of the advantages of using properties instead of accessing global data is that you can cause side-effects when writing (or reading) the value of a property.

For example, you can draw directly on the form surface, set the values of multiple properties, call special methods, change the status of multiple components at once, or fire an event, if available.

```

type
  TFormDialog = class(TForm)
  private
    ListItems: TListBox;
    function GetItems(Index: Integer): string;
    procedure SetItems(Index: Integer; const Value: string);
  public
    property Items[Index: Integer]: string read GetItems write SetItems; default;
  end;
function TFormDialog.GetItems(Index: Integer): string;
begin
  if Index >= ListItems.Items.Count then
    raise Exception.Create('TFormDialog: Out of Range');
  Result := ListItems.Items [Index];
end;
procedure TFormDialog.SetItems(Index: Integer; const Value: string);
begin
  if Index >= ListItems.Items.Count then
    raise Exception.Create('TFormDialog: Out of Range');
  ListItems.Items [Index] := Value;
end;

```

► Listing 3: The definition of a default array property in a form.

```

procedure TComponent.SetReference(Enable: Boolean);
var
  Field: ^TComponent;
begin
  if FOwner <> nil then begin
    Field := FOwner.FieldAddress(FName);
    if Field <> nil then
      if Enable then
        Field^ := Self
      else
        Field^ := nil;
  end;
end;

```

► Listing 4: The VCL code used to hook a component to its reference in the owner form.

### Rule 14: Hide Components

Too often I hear OOP purists complaining because Delphi forms include the list of the components in the published section, an approach that doesn't conform to the principle of encapsulation. They are actually pointing out an important issue, but most of them seem to be unaware that the solution is at hand without rewriting Delphi or changing the language.

The component references which Delphi adds to a form can be moved to the private portion, so that they won't be accessible by other forms. This way you can make compulsory the use of properties mapped to the components (see *Rule 11* above) to access their status.

If Delphi places all the components in the published section, this is because of the way these fields are bound to the components created from the .DFM file. When you set a component's name the VCL automatically attaches the component object to its reference in the form. This is possible only if the reference is published, because

Delphi uses RTTI and TObject methods to perform this.

If you want to understand the details, refer to Listing 4, which has the code of the SetReference method of the TComponent class, which is called by InsertComponent, RemoveComponent and SetName.

Once you know this, you realise that by moving the component references from the published to the private section you lose this automatic behaviour. To fix the problem, simply make it manual, by adding the following code for each component in the OnCreate event handler of the form:

```

Edit1 := FindComponent('Edit1')
as TEdit;

```

The second operation you have to do is register the component classes in the system, so that their RTTI information is included in the compiled program and made available to the system. This is needed only once for every component class, and only if you move all the component references of this type to the private section. You can add

this call even if it is not required, as an extra call to the RegisterClasses method is harmless. The RegisterClasses method is usually added to the initialization section of the unit hosting the form:

```
RegisterClasses([TEdit]);
```

### Rule 15: The OOP Form Wizard

Repeating the two operations above for every component of every form is certainly boring and time consuming. To avoid this excessive burden, I've written a simple wizard which generates the lines of code to add to the program in a small window. You'll need to do two simple copy and paste operations for each form.

The wizard doesn't automatically place the source code in the proper location: I'm working to fix this and you can check my website ([www.marcocantu.com](http://www.marcocantu.com)) for an updated version.

### Part 2: Inheritance

After a first set of rules devoted to classes, and particularly form classes, here comes another short list of suggestions and tips related to inheritance and visual form inheritance.

### Rule 16: Visual Form Inheritance

This is a powerful mechanism, if used properly. From my experience, its value grows with the size of the project. In a complex program you can use the hierarchical relationship among forms to operate on groups of forms with polymorphism.

Visual form inheritance allows you to share the common behaviour of multiple forms: you can have common methods, properties, event handlers, components, component properties, component event handlers, and so on.

### Rule 17: Limit Protected Data

When building a hierarchy of classes, some programmers tend to use mainly protected fields, as private fields are not accessible by subclasses. I won't say this is always wrong, but it is certainly

against encapsulation. The implementation of protected data is shared among all inherited forms, and you might have to update all of them in case the original definition of the data changes.

Notice that if you follow the rule of hiding components (*Rule 14*) the inherited forms can't possibly access the private components of the base class. In an inherited form, code such as `Edit1.Text := ''`; will not be compiled anymore. I can see this might not be terribly handy, but at least in theory it should be regarded as a positive thing, not negative. If you feel this is too much of a concession to encapsulation, declare the component references in the protected section of the base form.

### Rule 18: Protected Access Methods

It is much better, instead, to keep the component references in the private section and add access functions to their properties to the base class. If these access functions are used only internally and are not part of the class interface, you should declare them as protected. For example, the `GetText` and `SetText` form methods described in *Rule 11* can become protected and we could access the edit text by calling:

```
SetText('');
```

Actually, as the method was mapped to a property, we can simply write:

```
Text := '';
```

### Rule 19: Protected Virtual Methods

Another key point to have a flexible hierarchy is to declare virtual

methods you can call from the external classes to obtain polymorphism. If this is a common approach, it is less frequent to see protected virtual methods, called by other public methods. This is an important technique, as you can customise the virtual method in a derived class, modifying the behaviour of the objects.

### Rule 20: Virtual Methods For Properties

Even property access methods can be declared as virtual, so that a derived class can change the behaviour of the property without having to redefine it. This approach is seldom used by the VCL but is very flexible and powerful. To accomplish this, simply declare as virtual the `Get` and `Set` methods of *Rule 11*. The base form will have the code of Listing 5.

In the inherited form you can now override the virtual method `SetText`, to add some extra behaviour:

```
procedure TFormInherit.SetText(  
    const Value: String);  
begin  
    inherited SetText (Value);  
    if Value = '' then  
        Button1.Enabled := False;  
end;
```

### The Code

All the code fragments in this article can be found in the `OopDemo` example project, included on this month's disk. You should check in particular the secondary form (in the `frm2` unit) and the derived one (in the `inher` unit). Notice that in order to use, at the same time, a custom constructor with initialisation code and the private component references, it is necessary to set the `OldCreateOrder` property of

the form. Otherwise the initialisation code in the form constructor (which uses the components) will be executed before the `OnCreate` method of the form, which connects the references to the actual components.

On the disk you'll also find the compiled package of a first draft version of the OOP Form Wizard, but you should (hopefully) be able to find a more complete version on my website.

### Conclusion

Programming in Delphi according to good OOP principles is far from obvious, as some of the rules I've listed highlight. I don't think that you should consider all of my rules compulsory, as some of them might stretch your patience. The rules should be applied in the proper context, and become more and more important as the size of the application grows, along with the number of programmers working on it. Even for smaller programs, however, keeping in mind the OOP principles underlying my rules (encapsulation before all others) can really help.

There are certainly many other rules of thumb you can come up with, as I haven't tried to get into memory handling and RTTI issues, which are so complex to deserve specific articles.

My conclusion is that following the rules I've highlighted has a cost, in terms of extra code: it is the price you have to pay to obtain a more flexible and robust program. It is the price of object oriented programming. Let's hope that future Delphi versions help us reduce that price.

---

Marco Cantù is the author of the *Mastering Delphi* series, *Delphi Developer's Handbook*, and of the free online book *Essential Pascal*. He teaches classes on Delphi foundations and advanced topics. Check his website at [www.marcocantu.com](http://www.marcocantu.com) for more information. You can reach him on his public newsgroups: see the website for details.

► Listing 5: A form with properties implemented with virtual methods.

```
type  
    TFormDialog = class(TForm)  
        procedure FormCreate(Sender: TObject);  
    private  
        Edit1: TEdit;  
    protected  
        function GetText: String; virtual;  
        procedure SetText(const Value: String); virtual;  
    public  
        constructor Create (Text: string); reintroduce; overload;  
        property Text: String read GetText write SetText;  
    end;
```



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Introduction

**October 1, 1999: Book is on Delphi 5 Companion CD.  
Source Code available. [Examples list added.](#)**



**Book Cover. Apollo, the god worshipped at Delphi,  
in an Italian 17th century fresco.**

The first few editions of Mastering Delphi, the best selling Delphi book I've written, provided an introduction to the Pascal language in Delphi. Due to space constraints and because many Delphi programmers look for more advanced information, in the latest edition this material was completely omitted. To overcome the absence of this information, I've started putting together this online book, titled **Essential Pascal**.

This is a detailed book on Pascal, which for the moment will be available for free on my web site (I really don't know what will happen next, I might even find a publisher). This is a work in progress, and any feedback is welcome. The first complete version of this book, dated July '99, has been published on the Delphi 5 Companion CD.

## Copyright

The text and the source code of this book is copyrighted by Marco Cantù. Of course you can use the programs and adapt them to

your own needs, only you are not allowed to use them in books, training material, and other copyrighted formats. Feel free to link your site with this one, but please do not duplicate the material as it is very subject to frequent changes and updates.

## The Book Structure

The following is the current structure of the book:

- **Chapter 1: Pascal History**
- **Chapter 2: Coding in Pascal**
- **Chapter 3: Types, Variables, and Constants**
- **Chapter 4: User-Defined Data Types**
- **Chapter 5: Statements**
- **Chapter 6: Procedures and Functions**
- **Chapter 7: Handling Strings**
- **Chapter 8: Memory** (and Dynamic Arrays)
- **Chapter 9: Windows Programming**
- **Chapter 10: Variants**
- **Chapter 11: Programs and Units**
- **Appendix A: Glossary of terms**
- **Appendix B: Examples**

## Source Code

The source code of all the examples mentioned in the book is available. The code has the same Copyright as the book: Feel free to use it at will but don't publish it on other documents or site. Links back to this site are welcome.

Download the source code in a single zip file, [EPasCode.zip](#) (only 26 KB in size) and check out the [list of the examples](#).

## Feedback

Please let me know of any errors you find, but also of topics not clear enough for a beginner. I'll be able to devote time to the project depending also on the feedback I receive. Let me know also which other topics (not covered in Mastering Delphi 4) you'd like to see here. Again, hook onto the newsgroup, listed on my web site, and look for the books section, or mail to [marco@marcocantu.com](mailto:marco@marcocantu.com) (putting **Essential Pascal** in the subject (and your request or comment in the text).

## Acknowledgements

If I'm publishing a book on the web for free, I think this is mainly due to Bruce Eckel's experience with Thinking in Java. I'm a friend of Bruce and think he really did a great job with that book and few others.

As I mentioned the project to people at Borland I got a lot of positive feedback as well. And of course I must thank the company for making first the Turbo Pascal series of compilers and now the Delphi series of visual IDEs.

I'm starting to get some precious feedback. The first readers who helped improving this material quite a lot are Charles Wood and Wyatt Wong. Mark Greenhaw helped with some editing the text. Rafael Barranco-Droege offered a lot of technical corrections and language editing. Thanks.

## Author

Marco Cantù lives in Piacenza, Italy. After writing C++ and Object Windows Library books and articles, he delved into Delphi programming. He is the author of the Mastering Delphi book series, published by Sybex, as well as the advanced Delphi Developers Handbook. He writes articles for many magazines, including The Delphi Magazine, speaks at Delphi and Borland conferences around the world, and teaches Delphi classes at basic and advanced levels.

You can find more details about Marco and his work on his web site, [www.marcocantu.com](http://www.marcocantu.com).

© Copyright Marco Cantù, Wintech Italia Srl 1995-2000





- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 1 Pascal History

The Object Pascal programming language we use in Delphi wasn't invented in 1995 along with the Borland visual development environment. It was simply extended from the Object Pascal language already in use in the Borland Pascal products. But Borland didn't invent Pascal, it only helped make it very popular and extended it a little...

This chapter will contain some historical background on the Pascal language and its evolution. For the moment it contains only very short summaries.

### Wirth's Pascal

The Pascal language was originally designed in 1971 by Niklaus Wirth, professor at the Polytechnic of Zurich, Switzerland. Pascal was designed as a simplified version for educational purposes of the language Algol, which dates from 1960.

When Pascal was designed, many programming languages existed, but few were in widespread use: FORTRAN, C, Assembler, COBOL. The key idea of the new language was order, managed through a strong concept of data type, and requiring declarations and structured program controls. The language was also designed to be a teaching tool for students of programming classes.

### Turbo Pascal

Borland's world-famous Pascal compiler, called Turbo Pascal, was introduced in 1983, implementing "Pascal User Manual and Report" by Jensen and Wirth. The Turbo Pascal compiler has been one of the best-selling series of compilers of all time, and made the language particularly popular on the PC platform, thanks to its balance of simplicity and power.

Turbo Pascal introduced an Integrated Development Environment (IDE) where you could edit the code (in a WordStar compatible editor), run the compiler, see the errors, and jump back to the lines containing those errors. It sounds trivial now, but previously you had to quit the editor, return to DOS; run the command-line compiler, write down the error lines, open the editor and jump there.

Moreover Borland sold Turbo Pascal for 49 dollars, where Microsoft's Pascal compiler was sold for a few hundred. Turbo Pascal's many years of success contributed to Microsoft's eventual cancellation of its Pascal compiler product.

### Delphi's Pascal

After 9 versions of Turbo and Borland Pascal compilers, which gradually extended the language, Borland released Delphi in 1995, turning Pascal into a visual programming language.

Delphi extends the Pascal language in a number of ways, including many object-oriented extensions which are different from other flavors of Object Pascal, including those in the Borland Pascal with Objects compiler.

## **Next Chapter: Coding in Pascal**



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 2 Coding in Pascal

Before we move on to the subject of writing Pascal language statements, it is important to highlight a couple of elements of Pascal coding style. The question I'm addressing here is this: Besides the syntax rules, how should you write code? There isn't a single answer to this question, since personal taste can dictate different styles. However, there are some principles you need to know regarding comments, uppercase, spaces, and the so-called pretty-printing. In general, the goal of any coding style is clarity. The style and formatting decisions you make are a form of shorthand, indicating the purpose of a given piece of code. An essential tool for clarity is consistency-whatever style you choose, be sure to follow it throughout a project.

### Comments

In Pascal, comments are enclosed in either braces or parentheses followed by a star. Delphi also accepts the C++ style comments, which can span to the end of the line:

```
{this is a comment}
(* this is another comment *)
// this is a comment up to the end of the line
```

The first form is shorter and more commonly used. The second form was often preferred in Europe because many European keyboards lack the brace symbol. The third form of comments has been borrowed from C++ and is available only in the 32-bit versions of Delphi. Comments up to the end of the line are very helpful for short comments and for commenting out a line of code.

In the listings of the book I'll try to mark comments as italic (and keywords in bold), to be consistent with the default Delphi syntax highlighting.

Having three different forms of comments can be helpful for making nested comments. If you want to comment out several lines of source code to disable them, and these lines contain some real comments, you cannot use the same comment identifier:

```
{ ... code
{comment, creating problems}
... code }
```

With a second comment identifier, you can write the following code, which is correct:

```
{ ... code
```

```
//this comment is OK
... code }
```

Note that if the open brace or parenthesis-star is followed by the dollar sign (\$), it becomes a compiler directive, as in {\$X+}.

Actually, compiler directives are still comments. For example, {\$X+ This is a comment} is legal. It's both a valid directive and a comment, although sane programmers will probably tend to separate directives and comments.

## Use of Uppercase

The Pascal compiler (unlike those in other languages) ignores the case (capitalization) of characters. Therefore, the identifiers Myname, MyName, myname, myName, and MYNAME are all exactly equivalent. On the whole, this is definitely a positive, since in case-sensitive languages, many syntax errors are caused by incorrect capitalization.

Note: There is only one exception to the case-insensitive rule of Pascal: the Register procedure of a components' package must start with the uppercase R, because of a C++Builder compatibility issue.

There are a couple of subtle drawbacks, however. First, you must be aware that these identifiers really are the same, so you must avoid using them as different elements. Second, you should try to be consistent in the use of uppercase letters, to improve the readability of the code.

A consistent use of case isn't enforced by the compiler, but it is a good habit to get into. A common approach is to capitalize only the first letter of each identifier. When an identifier is made up of several consecutive words (you cannot insert a space in an identifier), every first letter of a word should be capitalized:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

Other elements completely ignored by the compiler are the spaces, new lines, and tabs you add to the source code. All these elements are collectively known as white space. White space is used only to improve code readability; it does not affect the compilation.

Unlike BASIC, Pascal allows you to write a statement on several lines of code, splitting a long instruction on two or more lines. The drawback (at least for many BASIC programmers) of allowing statements on more than one line is that you have to remember to add a semicolon to indicate the end of a statement, or more precisely, to separate a statement from the next one. Notice that the only restriction in splitting programming statements on different lines is that a string literal may not span several lines.

Again, there are no fixed rules on the use of spaces and multiple-line statements, just some rules of thumb:

- The Delphi editor has a vertical line you can place after 60 or 70 characters. If you use this line and try to avoid surpassing this limit, your source code will look better when you print it on paper. Otherwise long lines may get broken at any position, even in the middle of a word, when you print them.
- When a function or procedure has several parameters, it is common practice to place the parameters on different lines.
- You can leave a line completely white (blank) before a comment or to divide a long piece of code in smaller portions. Even this simple idea can improve the readability of the code, both on screen and when you print it.
- Use spaces to separate the parameters of a function call, and maybe even a space before the initial open parenthesis. Also keep operands of an expression separated. I know that some programmers will disagree with these ideas, but I insist: Spaces are free; you don't pay for them. (OK, I know that they use up disk space and modem connection time when you upload or download a file, but this is less and less relevant, nowadays.)

## Pretty-Printing

The last suggestion on the use of white spaces relates to the typical Pascal language-formatting style, known as pretty-printing. This rule is simple: Each time you need to write a compound statement, indent it two spaces to the right of the rest of the current statement. A compound statement inside another compound statement is indented four spaces, and so on:

```

if ... then
    statement;

if ... then
begin
    statement1;
    statement2;
end;

if ... then
begin
    if ... then
        statement1;
        statement2;
    end;
end;

```

The above formatting is based on pretty-printing, but programmers have different interpretations of this general rule. Some programmers indent the begin and end statements to the level of the inner code, some of them indent begin and end and then indent the internal code once more, other programmers put the begin in the line of the if condition. This is mostly a matter of personal taste.

A similar indented format is often used for lists of variables or data types, and to continue a statement from the previous line:

```

type
    Letters = set of Char;
var
    Name: string;
begin
    { long comment and long statement, going on in the
      following line and indented two spaces }
    MessageDlg ('This is a message',
        mtInformation, [mbOk], 0);

```

Of course, any such convention is just a suggestion to make the code more readable to other programmers, and it is completely ignored by the compiler. I've tried to use this rule consistently in all of the samples and code fragments in this book. Delphi source code, manuals, and Help examples use a similar formatting style.

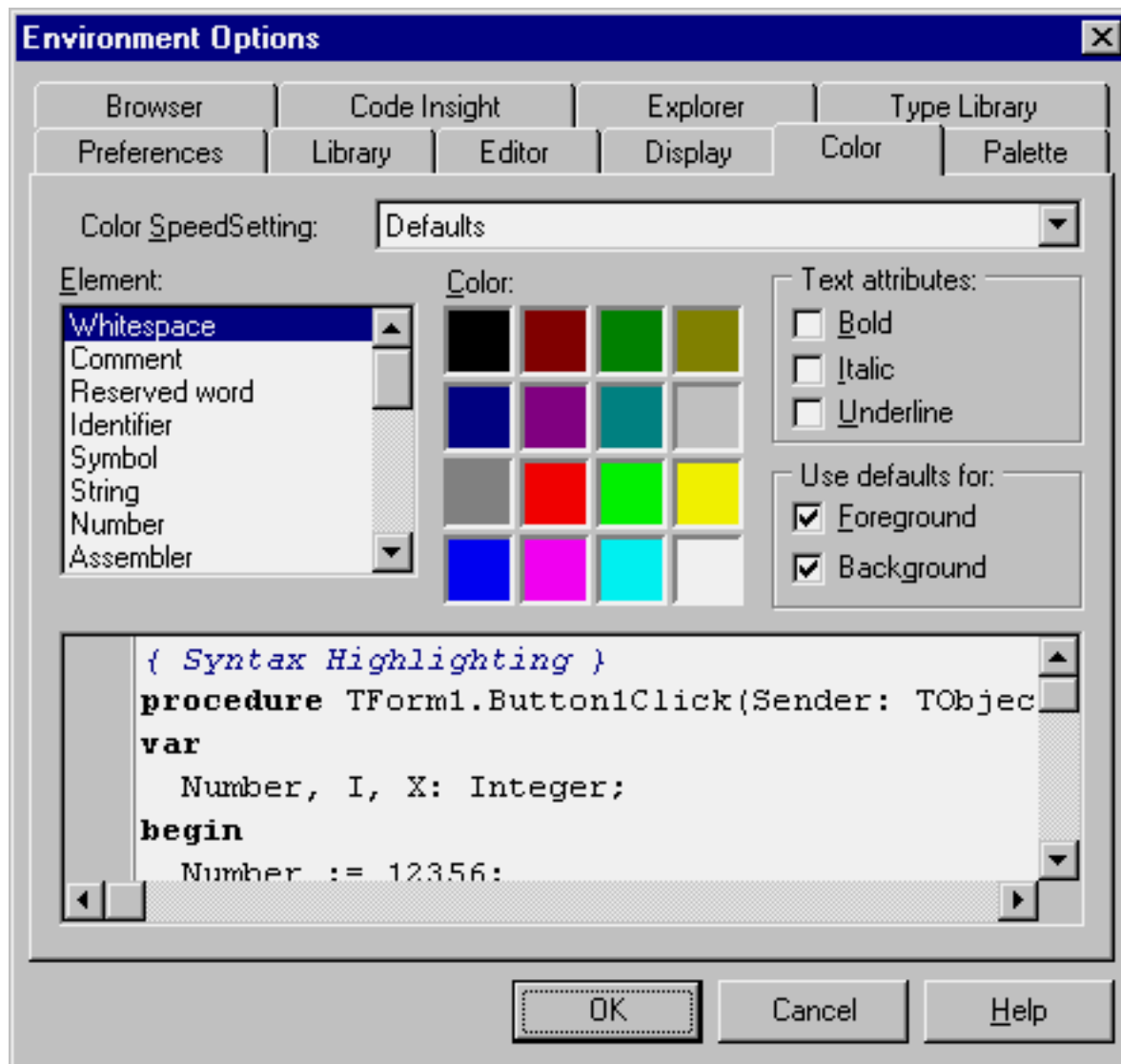
## Syntax Highlighting

To make it easier to read and write Pascal code, the Delphi editor has a feature called color syntax highlighting. Depending on the meaning in Pascal of the words you type in the editor, they are displayed using different colors. By default, keywords are in bold, strings and comments are in color (and often in italic), and so on.

Reserved words, comments, and strings are probably the three elements that benefit most from this feature. You can see at a glance a misspelled keyword, a string not properly terminated, and the length of a multiple-line comment.

You can easily customize the syntax highlight settings using the Editor Colors page of the Environment Options dialog box (see Figure 2.1). If you work by yourself, choose the colors you like. If you work closely with other programmers, you should all agree on a standard color scheme. I find that working on a computer with a different syntax coloring than the one I am used to is really difficult.

**FIGURE 2.1:** The dialog box used to set the color syntax highlighting.



Note: In this book I've tried to apply a sort of syntax highlighting to the source code listings. I hope this actually makes them more readable.

## Using Code Templates

Delphi 3 introduced a new feature related to source code editing. Because when writing Pascal language statements you often repeat the same sequence of keywords, Borland has provided a new feature called Code Templates. A code template is simply a piece of code related with a shorthand. You type the shorthand, then press Ctrl+J, and the full piece of code appears. For example, if you type `arrayd`, and then press Ctrl+J, the Delphi editor will expand your text into:

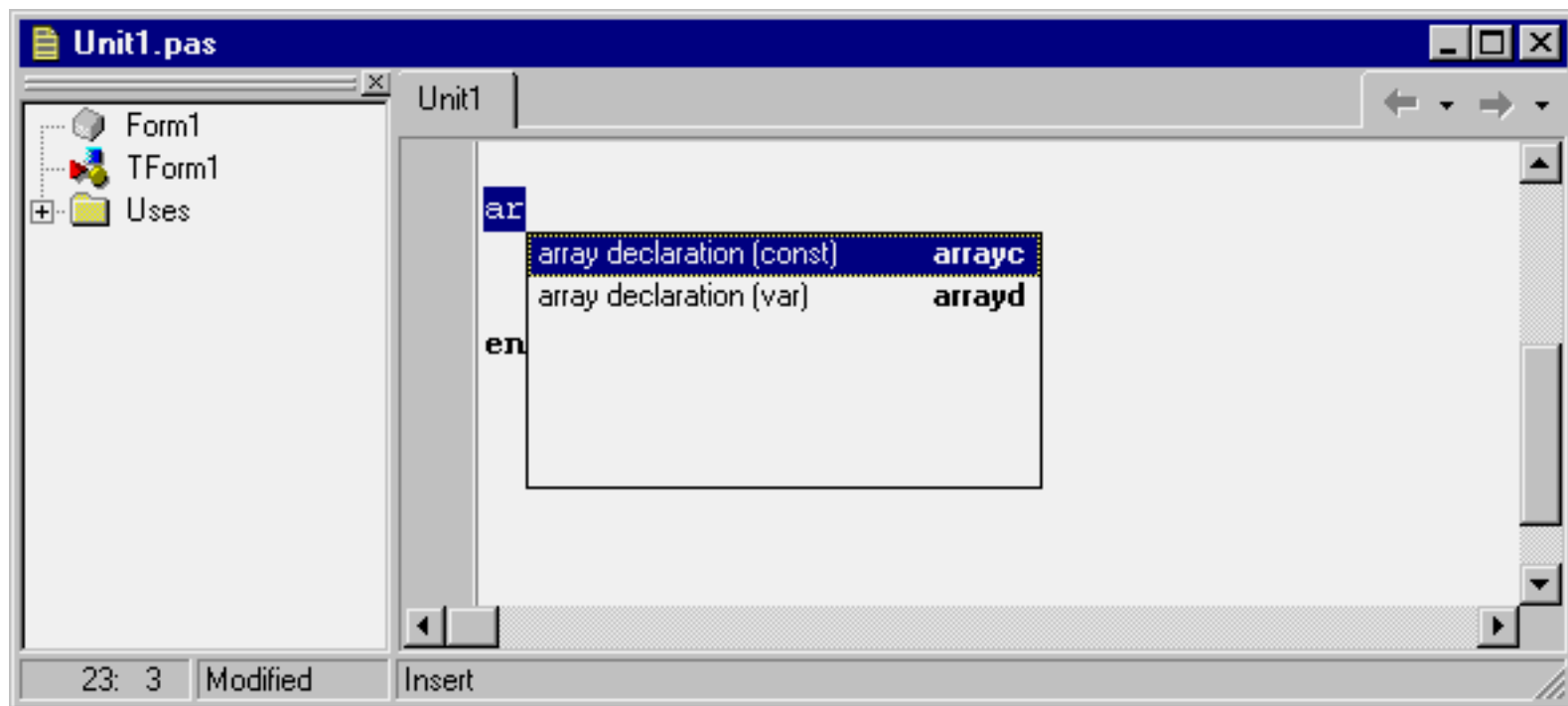
```
array [0..] of ;
```

Since the predefined code templates usually include several versions of the same construct, the shortcut generally terminates with a



letter indicating which of the versions you are interested in. However, you can also type only the initial part of the shortcut. For example, if you type ar and then press Ctrl+J, the editor will display a local menu with a list of the available choices with a short description, as you can see in Figure 2.2.

**Figure 2.2: Code Templates selection**



You can fully customize the code templates by modifying the existing ones or adding your own common code pieces. If you do this, keep in mind that the text of a code template generally includes the '|' character to indicate where the cursor should jump to after the operation, that is, where you start typing to complete the template with custom code.

## Language Statements

Once you have defined some identifiers, you can use them in statements and in the expressions that are part of some statements. Pascal offers several statements and expressions. Let's look at keywords, expressions, and operators first.

## Keywords

Keywords are all the Object Pascal reserved identifiers, which have a role in the language. Delphi's help distinguishes between reserved words and directives: Reserved words cannot be used as identifiers, while directives should not be used as such, even if the compiler will accept them. In practice, you should not use any keywords as an identifier.

In Table 2.1 you can see a complete list of the identifiers having a specific role in the Object Pascal language (in Delphi 4), including keywords and other reserved words.

**Table 2.1: Keywords and other reserved words in the Object Pascal language**

Keyword	Role
absolute	directive (variables)
abstract	directive (method)

and	operator (boolean)
array	type
as	operator (RTTI)
asm	statement
assembler	backward compatibility (asm)
at	statement (exceptions)
automated	access specifier (class)
begin	block marker
case	statement
cdecl	function calling convention
class	type
const	declaration or directive (parameters)
constructor	special method
contains	operator (set)
default	directive (property)
destructor	special method
dispid	dispinterface specifier
dispinterface	type
div	operator
do	statement
downto	statement (for)
dynamic	directive (method)
else	statement (if or case)
end	block marker
except	statement (exceptions)
export	backward compatibility (class)
exports	declaration
external	directive (functions)
far	backward compatibility (class)
file	type
finalization	unit structure
finally	statement (exceptions)
for	statement
forward	function directive
function	declaration
goto	statement
if	statement
implementation	unit structure
implements	directive (property)
in	operator (set) - project structure
index	directive (dispinterface)
inherited	statement
initialization	unit structure
inline	backward compatibility (see asm)
interface	type

is	operator (RTTI)
label	declaration
library	program structure
message	directive (method)
mod	operator (math)
name	directive (function)
near	backward compatibility (class)
nil	value
nodefault	directive (property)
not	operator (boolean)
object	backward compatibility (class)
of	statement (case)
on	statement (exceptions)
or	operator (boolean)
out	directive (parameters)
overload	function directive
override	function directive
package	program structure (package)
packed	directive (record)
pascal	function calling convention
private	access specifier (class)
procedure	declaration
program	program structure
property	declaration
protected	access specifier (class)
public	access specifier (class)
published	access specifier (class)
raise	statement (exceptions)
read	property specifier
readonly	dispatch interface specifier
record	type
register	function calling convention
reintroduce	function directive
repeat	statement
requires	program structure (package)
resident	directive (functions)
resourcestring	type
safecall	function calling convention
set	type
shl	operator (math)
shr	operator (math)
stdcall	function calling convention
stored	directive (property)
string	type
then	statement (if)

threadvar	declaration
to	statement (for)
try	statement (exceptions)
type	declaration
unit	unit structure
until	statement
uses	unit structure
var	declaration
virtual	directive (method)
while	statement
with	statement
write	property specifier
writeln	dispatch interface specifier
xor	operator (boolean)

## Expressions and Operators

There isn't a general rule for building expressions, since they mainly depend on the operators being used, and Pascal has a number of operators. There are logical, arithmetic, Boolean, relational, and set operators, plus some others. Expressions can be used to determine the value to assign to a variable, to compute the parameter of a function or procedure, or to test for a condition. Expressions can include function calls, too. Every time you are performing an operation on the value of an identifier, rather than using an identifier by itself, that is an expression.

Expressions are common to most programming languages. An expression is any valid combination of constants, variables, literal values, operators, and function results. Expressions can also be passed to value parameters of procedures and functions, but not always to reference parameters (which require a value you can assign to).

## Operators and Precedence

If you have ever written a program in your life, you already know what an expression is. Here, I'll highlight specific elements of Pascal operators. You can see a list of the operators of the language, grouped by precedence, in Table 2.1.

Contrary to most other programming languages, the `and` and `or` operators have precedence compared to the relational one. So if you write `a < b and c < d`, the compiler will try to do the `and` operation first, resulting in a compiler error. For this reason you should enclose each of the `<` expression in parentheses: `(a < b) and (c < d)`.

Some of the common operators have different meanings with different data types. For example, the `+` operator can be used to add two numbers, concatenate two strings, make the union of two sets, and even add an offset to a `PChar` pointer. However, you cannot add two characters, as is possible in C.

Another strange operator is `div`. In Pascal, you can divide any two numbers (real or integers) with the `/` operator, and you'll invariably get a real-number result. If you need to divide two integers and want an integer result, use the `div` operator instead.

**Table 2.2: Pascal Language Operators, Grouped by Precedence**

### Unary Operators (Highest Precedence)

@	Address of the variable or function (returns a pointer)
not	Boolean or bitwise not
<b>Multiplicative and Bitwise Operators</b>	
*	Arithmetic multiplication or set intersection
/	Floating-point division
div	Integer division
mod	Modulus (the remainder of integer division)
as	Allows a type-checked type conversion among at runtime (part of the RTTI support)
and	Boolean or bitwise and
shl	Bitwise left shift
shr	Bitwise right shift
<b>Additive Operators</b>	
+	Arithmetic addition, set union, string concatenation, pointer offset addition
-	Arithmetic subtraction, set difference, pointer offset subtraction
or	Boolean or bitwise or
xor	Boolean or bitwise exclusive or
<b>Relational and Comparison Operators (Lowest Precedence)</b>	
=	Test whether equal
<>	Test whether not equal
<	Test whether less than
>	Test whether greater than
<=	Test whether less than or equal to, or a subset of a set
>=	Test whether greater than or equal to, or a superset of a set
in	Test whether the item is a member of the set
is	Test whether object is type-compatible (another RTTI operator)

## Set Operators

The set operators include union (+), difference (-), intersection (\*), membership test (in), plus some relational operators. To add an element to a set, you can make the union of the set with another one that has only the element you need. Here's a Delphi example related to font styles:

```
Style := Style + [fsBold];
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

As an alternative, you can use the standard Include and Exclude procedures, which are much more efficient (but cannot be used with component properties of the set type, because they require an l-value parameter):

```
Include (Style, fsBold);
```

## **Conclusion**

Now that we know the basic layout of a Pascal program we are ready to start understanding its meaning in detail. We'll start by exploring the definition of predefined and user defined data types, then we'll move along to the use of the keywords to form programming statements.

## **Next Chapter: Types, Variables, and Constants**





- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 3 Types, Variables, and Constants

The original Pascal language was based on some simple notions, which have now become quite common in programming languages. The first is the notion of data type. The type determines the values a variable can have, and the operations that can be performed on it. The concept of type is stronger in Pascal than in C, where the arithmetic data types are almost interchangeable, and much stronger than in the original versions of BASIC, which had no similar concept.

### Variables

Pascal requires all variables to be declared before they are used. Every time you declare a variable, you must specify a data type. Here are some sample variable declarations:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

The `var` keyword can be used in several places in the code, such as at the beginning of the code of a function or procedure, to declare variables local to the routine, or inside a unit to declare global variables. After the `var` keyword comes a list of variable names, followed by a colon and the name of the data type. You can write more than one variable name on a single line, as in the last statement above.

Once you have defined a variable of a given type, you can perform on it only the operations supported by its data type. For example, you can use the Boolean value in a test and the integer value in a numerical expression. You cannot mix Booleans and integers (as you can with the C language).

Using simple assignments, we can write the following code:

```
Value := 10;
IsCorrect := True;
```

But the next statement is not correct, because the two variables have different data types:

```
Value := IsCorrect; // error
```

If you try to compile this code, Delphi issues a compiler error with this description: Incompatible types: 'Integer' and 'Boolean'. Usually, errors like this are programming errors, because it does not make sense to assign a True or False value to a variable of the Integer data type. You should not blame Delphi for these errors. It only warns you that there is something wrong in the code.

Of course, it is often possible to convert the value of a variable from one type into a different type. In some cases, this conversion is automatic, but usually you need to call a specific system function that changes the internal representation of the data.

In Delphi you can assign an initial value to a global variable while you declare it. For example, you can write:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

This initialization technique works only for global variables, not for variables declared inside the scope of a procedure or method.

## Constants

Pascal also allows the declaration of constants to name values that do not change during program execution. To declare a constant you don't need to specify a data type, but only assign an initial value. The compiler will look at the value and automatically use its proper data type. Here are some sample declarations:

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantù';
```

Delphi determines the constant's data type based on its value. In the example above, the Thousand constant is assumed to be of type `SmallInt`, the smallest integral type which can hold it. If you want to tell Delphi to use a specific type you can simply add the type name in the declaration, as in:

```
const
  Thousand: Integer = 1000;
```

When you declare a constant, the compiler can choose whether to assign a memory location to the constant, and save its value there, or to duplicate the actual value each time the constant is used. This second approach makes sense particularly for simple constants.

Note: The 16-bit version of Delphi allows you to change the value of a typed constant at run-time, as if it was a variable. The 32-bit version still permits this behavior for backward compatibility when you enable the `$J` compiler directive, or use the corresponding Assignable typed constants check box of the Compiler page of the Project Options dialog box. Although this is the default, you are strongly advised not to use this trick as a general programming technique. Assigning a new value to a constant disables all the compiler optimizations on constants. In such a case, simply declare a variable, instead.

## Resource String Constants

When you define a string constant, instead of writing:

```
const
  AuthorName = 'Marco Cantù';
```

starting with Delphi 3 you can write the following:

```
resourcestring
  AuthorName = 'Marco Cantù';
```

In both cases you are defining a constant; that is, a value you don't change during program execution. The difference is only in the implementation. A string constant defined with the `resourcestring` directive is stored in the resources of the program, in a string table.

To see this capability in action, you can look at the `ResStr` example, which has a button with the following code:

```
resourcestring
  AuthorName = 'Marco Cantù';
  BookName = 'Essential Pascal';

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage (BookName + #13 + AuthorName);
end;
```

The output of the two strings appears on separate lines because the strings are separated by the newline character (indicated by its numerical value in the `#13` character-type constant).

The interesting aspect of this program is that if you examine it with a resource explorer (there is one available among the examples that ship with Delphi) you'll see the new strings in the resources. This means that the strings are not part of the compiled code but stored in a separate area of the executable file (the EXE file).

Note: In short, the advantage of resources is in an efficient memory handling performed by Windows and in the possibility of localizing a program (translating the strings to a different language) without having to modify its source code.

## Data Types

In Pascal there are several predefined data types, which can be divided into three groups: ordinal types, real types, and strings. We'll discuss ordinal and real types in the following sections, while strings are covered later in this chapter. In this section I'll also introduce some types defined by the Delphi libraries (not predefined by the compiler), which can be considered predefined types.

Delphi also includes a non-typed data type, called variant, and discussed in Chapter 10 of this book. Strangely enough a variant is a type without proper type-checking. It was introduced in Delphi 2 to handle OLE Automation.

## Ordinal Types

Ordinal types are based on the concept of order or sequence. Not only can you compare two values to see which is higher, but you can also ask for the value following or preceding a given value or compute the lowest or highest possible value.

The three most important predefined ordinal types are Integer, Boolean, and Char (character). However, there are a number of other related types that have the same meaning but a different internal representation and range of values. The following Table 3.1 lists the ordinal data types used for representing numbers.

**Table 3.1: Ordinal data types for numbers**

Size	Signed Range	Unsigned Range
8 bits	ShortInt -128 to 127	Byte 0 to 255
16 bits	SmallInt -32768 to 32767	Word 0 to 65,535
32 bits	LongInt -2,147,483,648 to 2,147,483,647	LongWord (since Delphi 4) 0 to 4,294,967,295
64 bits	Int64	
16/32 bits	Integer	Cardinal

As you can see, these types correspond to different representations of numbers, depending on the number of bits used to express the value, and the presence or absence of a sign bit. Signed values can be positive or negative, but have a smaller range of values, because one less bit is available for the value itself. You can refer to the Range example, discussed in the next section, for the actual range of values of each type.

The last group (marked as 16/32) indicates values having a different representation in the 16-bit and 32-bit versions of Delphi. Integer and Cardinal are frequently used, because they correspond to the native representation of numbers in the CPU.

## Integral Types in Delphi 4

In Delphi 3, the 32-bit unsigned numbers indicated by the Cardinal type were actually 31-bit values, with a range up to 2 gigabytes. Delphi 4 introduced a new unsigned numeric type, LongWord, which uses a truly 32-bit value up to 4 gigabytes. The Cardinal type is now an alias of the new LongWord type. LongWord permits 2GB more data to be addressed by an unsigned number, as mentioned above. Moreover, it corresponds to the native representation of numbers in the CPU.

Another new type introduced in Delphi 4 is the Int64 type, which represents integer numbers with up to 18 digits. This new type is fully supported by some of the ordinal type routines (such as High and Low), numeric routines (such as Inc and Dec), and string-conversion routines (such as IntToStr). For the opposite conversion, from a string to a number, there are two new specific functions: StrToInt64 and StrToInt64Def.

## Boolean

Boolean values other than the Boolean type are seldom used. Some Boolean values with specific representations are required by Windows API functions. The types are ByteBool, WordBool, and LongBool.

In Delphi 3 for compatibility with Visual Basic and OLE automation, the data types ByteBool, WordBool, and LongBool were modified to represent the value True with -1, while the value False is still 0. The Boolean data type remains unchanged (True is 1, False is 0). If you've used explicit typecasts in your Delphi 2 code, porting the code to later versions of Delphi might result in errors.

## Characters

Finally there are two different representation for characters: ANSISChar and WideChar. The first type represents 8-bit characters, corresponding to the ANSI character set traditionally used by Windows; the second represents 16-bit characters, corresponding to the new Unicode characters supported by Windows NT, and only partially by Windows 95 and 98. Most of the time you'll simply use the Char type, which in Delphi 3 corresponds to ANSISChar. Keep in mind, anyway, that the first 256 Unicode characters correspond exactly to the ANSI characters.

Constant characters can be represented with their symbolic notation, as in 'k', or with a numeric notation, as in #78. The latter can also be expressed using the Chr function, as in Chr (78). The opposite conversion can be done with the Ord function.

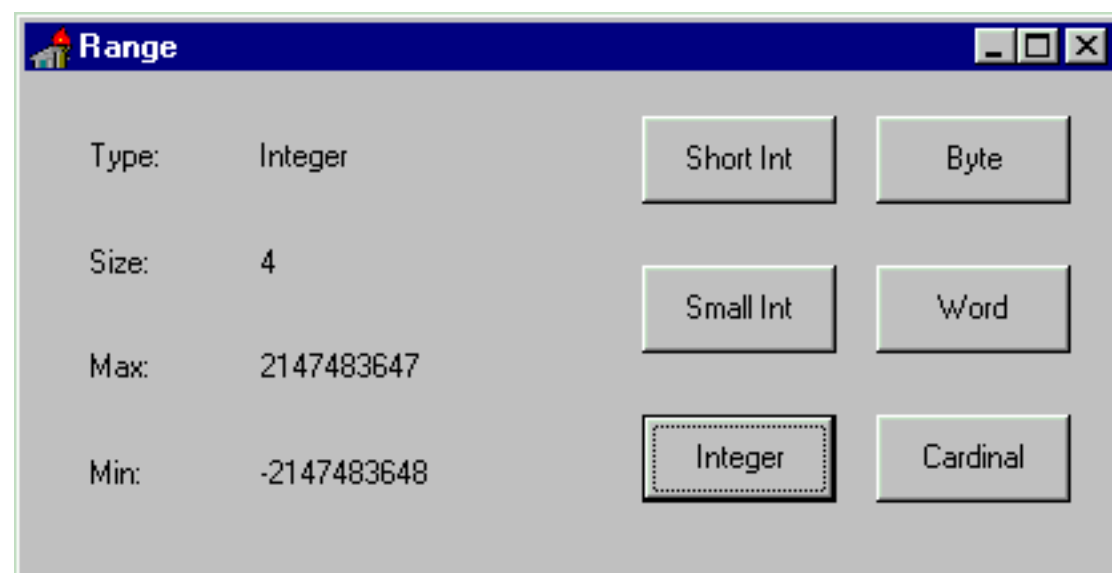
It is generally better to use the symbolic notation when indicating letters, digits, or symbols. When referring to special characters, instead, you'll generally use the numeric notation. The following list includes some of the most commonly used special characters:

- #9 tabulator
- #10 newline
- #13 carriage return (enter key)

## The Range Example

To give you an idea of the different ranges of some of the ordinal types, I've written a simple Delphi program named Range. Some results are shown in Figure 3.1.

**FIGURE 3.1:** The Range example displays some information about ordinal data types (Integers in this case).



The Range program is based on a simple form, which has six buttons (each named after an ordinal data type) and some labels for categories of information, as you can see in Figure 3.1. Some of the labels are used to hold static text, others to show the information about the type each time one of the buttons is pressed.

Every time you press one of the buttons on the right, the program updates the labels with the output. Different labels show the data type, number of bytes used, and the maximum and minimum values the data type can store. Each button has its own OnClick event-response method because the code used to compute the three values is slightly different from button to button. For example, here is the source code of the OnClick event for the Integer button (BtnInteger):

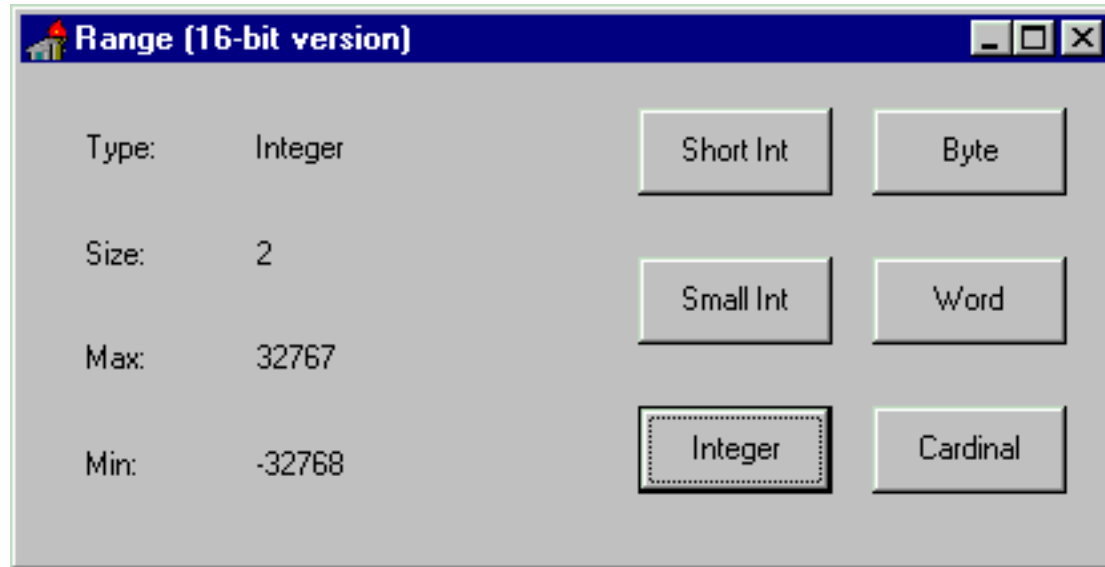
```
procedure TFormRange.BtnIntegerClick(Sender: TObject);
begin
  LabelType.Caption := 'Integer';
  LabelSize.Caption := IntToStr (SizeOf (Integer));
  LabelMax.Caption := IntToStr (High (Integer));
  LabelMin.Caption := IntToStr (Low (Integer));
end;
```

If you have some experience with Delphi programming, you can examine the source code of the program to understand how it works.

For beginners, it's enough to note the use of three functions: `SizeOf`, `High`, and `Low`. The results of the last two functions are ordinals of the same kind (in this case, integers), and the result of the `SizeOf` function is always an integer. The return value of each of these functions is first translated into strings using the `IntToStr` function, then copied to the captions of the three labels.

The methods associated with the other buttons are very similar to the one above. The only real difference is in the data type passed as a parameter to the various functions. Figure 3.2 shows the result of executing this same program under Windows 95 after it has been recompiled with the 16-bit version of Delphi. Comparing Figure 3.1 with Figure 3.2, you can see the difference between the 16-bit and 32-bit Integer data types.

**FIGURE 3.2:** The output of the 16-bit version of the Range example, again showing information about integers.



The size of the Integer type varies depending on the CPU and operating system you are using. In 16-bit Windows, an Integer variable is two bytes wide. In 32-bit Windows, an Integer is four bytes wide. For this reason, when you recompile the Range example, you get a different output.

The two different representations of the Integer type are not a problem, as long as your program doesn't make any assumptions about the size of integers. If you happen to save an Integer to a file using one version and retrieve it with another, though, you're going to have some trouble. In this situation, you should choose a platform-independent data type (such as `LongInt` or `SmallInt`). For mathematical computation or generic code, your best bet is to stick with the standard integral representation for the specific platform--that is, use the Integer type--because this is what the CPU likes best. The Integer type should be your first choice when handling integer numbers. Use a different representation only when there is a compelling reason to do so.

## Ordinal Types Routines

There are some system routines (routines defined in the Pascal language and in the Delphi system unit) that work on ordinal types. They are shown in Table 3.2. C++ programmers should notice that the two versions of the `Inc` procedure, with one or two parameters, correspond to the `++` and `+=` operators (the same holds for the `Dec` procedure).

**Table 3.2:** System Routines for Ordinal Types

Routine	Purpose
<code>Dec</code>	Decrements the variable passed as parameter, by one or by the value of the optional second parameter.
<code>Inc</code>	Increments the variable passed as parameter, by one or by the specified value.



Odd	Returns True if the argument is an odd number.
Pred	Returns the value before the argument in the order determined by the data type, the predecessor.
Succ	Returns the value after the argument, the successor.
Ord	Returns a number indicating the order of the argument within the set of values of the data type.
Low	Returns the lowest value in the range of the ordinal type passed as its parameter.
High	Returns the highest value in the range of the ordinal data type.

Notice that some of these routines, when applied to constants, are automatically evaluated by the compiler and replaced by their value. For example if you call High(X) where X is defined as an Integer, the compiler can simply replace the expression with the highest possible value of the Integer data type.

## Real Types

Real types represent floating-point numbers in various formats. The smallest storage size is given by Single numbers, which are implemented with a 4-byte value. Then there are Double floating-point numbers, implemented with 8 bytes, and Extended numbers, implemented with 10 bytes. These are all floating-point data types with different precision, which correspond to the IEEE standard floating-point representations, and are directly supported by the CPU numeric coprocessor, for maximum speed.

In Delphi 2 and Delphi 3 the Real type had the same definition as in the 16-bit version; it was a 48-bit type. But its usage was deprecated by Borland, which suggested that you use the Single, Double, and Extended types instead. The reason for their suggestion is that the old 6-byte format is neither supported by the Intel CPU nor listed among the official IEEE real types. To completely overcome the problem, Delphi 4 modifies the definition of the Real type to represent a standard 8-byte (64-bit) floating-point number.

In addition to the advantage of using a standard definition, this change allows components to publish properties based on the Real type, something Delphi 3 did not allow. Among the disadvantages there might be compatibility problems. If necessary, you can overcome the possibility of incompatibility by sticking to the Delphi 2 and 3 definition of the type; do this by using the following compiler option:

```
{ $REALCOMPATIBILITY ON }
```

There are also two strange data types: Comp describes very big integers using 8 bytes (which can hold numbers with 18 decimal digits); and Currency (not available in 16-bit Delphi) indicates a fixed-point decimal value with four decimal digits, and the same 64-bit representation as the Comp type. As the name implies, the Currency data type has been added to handle very precise monetary values, with four decimal places.

We cannot build a program similar to the Range example with real data types, because we cannot use the High and Low functions or the Ord function on real-type variables. Real types represent (in theory) an infinite set of numbers; ordinal types represent a fixed set of values.

Note: Let me explain this better. when you have the integer 23 you can determine which is the following value. Integers are finite (they have a determined range and they have an order). Floating point numbers are infinite even within a small range, and have no order: in fact, how many values are there between 23 and 24? And which number follows 23.46? It is 23.47, 23.461, or 23.4601? That's really hard to know!

For this reason, it makes sense to ask for the ordinal position of the character w in the range of the Char data type, but it makes no sense at all to ask the same question about 7143.1562 in the range of a floating-point data type. Although you can indeed know whether one real number has a higher value than another, it makes no sense to ask how many real numbers exist before a given number (this is the meaning of the Ord function).

Real types have a limited role in the user interface portion of the code (the Windows side), but they are fully supported by Delphi, including the database side. The support of IEEE standard floating-point types makes the Object Pascal language completely appropriate for the wide range of programs that require numerical computations. If you are interested in this aspect, you can look at the arithmetic functions provided by Delphi in the system unit (see the Delphi Help for more details).

**Note:** Delphi also has a Math unit that defines advanced mathematical routines, covering trigonometric functions (such as the ArcCosh function), finance (such as the InterestPayment function), and statistics (such as the MeanAndStdDev procedure). There are a number of these routines, some of which sound quite strange to me, such as the MomentSkewKurtosis procedure (I'll let you find out what this is).

## Date and Time

Delphi uses real types also to handle date and time information. To be more precise Delphi defines a specific TDateTime data type. This is a floating-point type, because the type must be wide enough to store years, months, days, hours, minutes, and seconds, down to millisecond resolution in a single variable. Dates are stored as the number of days since 1899-12-30 (with negative values indicating dates before 1899) in the integer part of the TDateTime value. Times are stored as fractions of a day in the decimal part of the value.

TDateTime is not a predefined type the compiler understands, but it is defined in the system unit as:

```
type
  TDateTime = type Double;
```

Using the TDateTime type is quite easy, because Delphi includes a number of functions that operate on this type. You can find a list of these functions in Table 3.3.

**Table 3.3: System Routines for the TDateTime Type**

Routine	Description
Now	Returns the current date and time into a single TDateTime value.
Date	Returns only the current date.
Time	Returns only the current time.
DateTimeToStr	Converts a date and time value into a string, using default formatting; to have more control on the conversion use the FormatDateTime function instead.
DateTimeToString	Copies the date and time values into a string buffer, with default formatting.
DateToStr	Converts the date portion of a TDateTime value into a string.
TimeToStr	Converts the time portion of a TDateTime value into a string.
FormatDateTime	Formats a date and time using the specified format; you can specify which values you want to see and which format to use, providing a complex format string.
StrToDateTime	Converts a string with date and time information to a TDateTime value, raising an exception in case of an error in the format of the string.
StrToDate	Converts a string with a date value into the TDateTime format.
StrToTime	Converts a string with a time value into the TDateTime format.
DayOfWeek	Returns the number corresponding to the day of the week of the TDateTime value passed as parameter.

DecodeDate	Retrieves the year, month, and day values from a date value.
DecodeTime	Retrieves out of a time value.
EncodeDate	Turns year, month, and day values into a TDateTime value.
EncodeTime	Turns hour, minute, second, and millisecond values into a TDateTime value.

To show you how to use this data type and some of its related routines, I've built a simple example, named `TimeNow`. The main form of this example has a `Button` and a `Listbox` component. When the program starts it automatically computes and displays the current time and date. Every time the button is pressed, the program shows the time elapsed since the program started.

Here is the code related to the `OnCreate` event of the form:

```

procedure TFormTimeNow.FormCreate(Sender: TObject);
begin
    StartTime := Now;
    ListBox1.Items.Add (TimeToStr (StartTime));
    ListBox1.Items.Add (DateToStr (StartTime));
    ListBox1.Items.Add ('Press button for elapsed time');
end;

```

The first statement is a call to the `Now` function, which returns the current date and time. This value is stored in the `StartTime` variable, declared as a global variable as follows:

```

var
    FormTimeNow: TFormTimeNow;
    StartTime: TDateTime;

```

I've added only the second declaration, since the first is provided by Delphi. By default, it is the following:

```

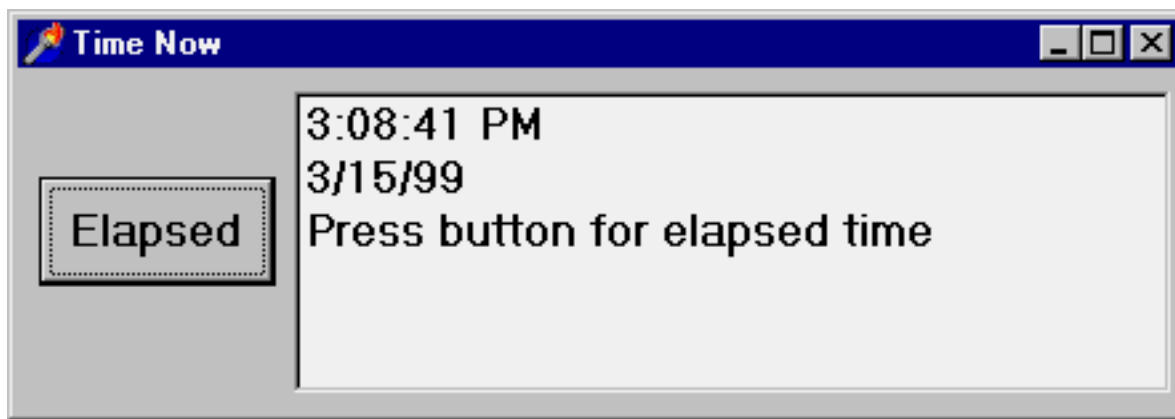
var
    Form1: TForm1;

```

Changing the name of the form, this declaration is automatically updated. Using global variables is actually not the best approach: It should be better to use a private field of the form class, a topic related to object-oriented programming and discussed in *Mastering Delphi 4*.

The next three statements add three items to the `Listbox` component on the left of the form, with the result you can see in *Figure 3.3*. The first line contains the time portion of the `TDateTime` value converted into a string, the second the date portion of the same value. At the end the code adds a simple reminder.

**FIGURE 3.3: The output of the `TimeNow` example at startup.**



This third string is replaced by the program when the user clicks on the Elapsed button:

```

procedure TFormTimeNow.ButtonElapsedClick(Sender: TObject);
var
    StopTime: TDateTime;
begin
    StopTime := Now;
    ListBox1.Items [2] := FormatDateTime ('hh:nn:ss',
        StopTime - StartTime);
end;

```

This code retrieves the new time and computes the difference from the time value stored when the program started. Because we need to use a value that we computed in a different event handler, we had to store it in a global variable. There are actually better alternatives, based on classes.

**Note:** The code that replaces the current value of the third string uses the index 2. The reason is that the items of a list box are zero-based: the first item is number 0, the second number 1, and the third number 2. More on this as we cover arrays.

Besides calling `TimeToStr` and `DateToStr` you can use the more powerful `FormatDateTime` function, as I've done in the last method above (see the Delphi Help file for details on the formatting parameters). Notice also that time and date values are transformed into strings depending on Windows international settings. Delphi reads these values from the system, and copies them to a number of global constants declared in the `SysUtils` unit. Some of them are:

```

DateSeparator: Char;
ShortDateFormat: string;
LongDateFormat: string;
TimeSeparator: Char;
TimeAMString: string;
TimePMString: string;
ShortTimeFormat: string;
LongTimeFormat: string;
ShortMonthNames: array [1..12] of string;
LongMonthNames: array [1..12] of string;
ShortDayNames: array [1..7] of string;
LongDayNames: array [1..7] of string;

```

More global constants relate to currency and floating-point number formatting. You can find the complete list in the Delphi Help file under the topic Currency and date/time formatting variables.

**Note:** Delphi includes a `DateTimePicker` component, which provides a sophisticated way to input a date, selecting it from a

calendar.

## Specific Windows Types

The predefined data types we have seen so far are part of the Pascal language. Delphi also includes other data types defined by Windows. These data types are not an integral part of the language, but they are part of the Windows libraries. Windows types include new default types (such as `DWORD` or `UINT`), many records (or structures), several pointer types, and so on.

Among Windows data types, the most important type is represented by handles, discussed in Chapter 9.

## Typecasting and Type Conversions

As we have seen, you cannot assign a variable to another one of a different type. In case you need to do this, there are two choices. The first choice is typecasting, which uses a simple functional notation, with the name of the destination data type:

```
var
  N: Integer;
  C: Char;
  B: Boolean;
begin
  N := Integer ('X');
  C := Char (N);
  B := Boolean (0);
```

You can typecast between data types having the same size. It is usually safe to typecast between ordinal types, or between real types, but you can also typecast between pointer types (and also objects) as long as you know what you are doing.

Casting, however, is generally a dangerous programming practice, because it allows you to access a value as if it represented something else. Since the internal representations of data types generally do not match, you risk hard-to-track errors. For this reason, you should generally avoid typecasting.

The second choice is to use a type-conversion routine. The routines for the various types of conversions are summarized in Table 3.4. Some of these routines work on the data types that we'll discuss in the following sections. Notice that the table doesn't include routines for special types (such as `TDateTime` or `variant`) or routines specifically intended for formatting, like the powerful `Format` and `FormatFloat` routines.

**Table 3.4: System Routines for Type Conversion**

Routine	Description
<code>Chr</code>	Converts an ordinal number into an ANSI character.
<code>Ord</code>	Converts an ordinal-type value into the number indicating its order.
<code>Round</code>	Converts a real-type value into an Integer-type value, rounding its value.
<code>Trunc</code>	Converts a real-type value into an Integer-type value, truncating its value.
<code>Int</code>	Returns the Integer part of the floating-point value argument.
<code>IntToStr</code>	Converts a number into a string.
<code>IntToHex</code>	Converts a number into a string with its hexadecimal representation.

StrToInt	Converts a string into a number, raising an exception if the string does not represent a valid integer.
StrToIntDef	Converts a string into a number, using a default value if the string is not correct.
Val	Converts a string into a number (traditional Turbo Pascal routine, available for compatibility).
Str	Converts a number into a string, using formatting parameters (traditional Turbo Pascal routine, available for compatibility).
StrPas	Converts a null-terminated string into a Pascal-style string. This conversion is automatically done for AnsiStrings in 32-bit Delphi. (See the section on strings later in this chapter.)
StrPCopy	Copies a Pascal-style string into a null-terminated string. This conversion is done with a simple PChar cast in 32-bit Delphi. (See the section on strings later in this chapter.)
StrPLCopy	Copies a portion of a Pascal-style string into a null-terminated string.
FloatToDecimal	Converts a floating-point value to record including its decimal representation (exponent, digits, sign).
FloatToStr	Converts the floating-point value to its string representation using default formatting.
FloatToStrF	Converts the floating-point value to its string representation using the specified formatting.
FloatToText	Copies the floating-point value to a string buffer, using the specified formatting.
FloatToTextFmt	As the previous routine, copies the floating-point value to a string buffer, using the specified formatting.
StrToFloat	Converts the given Pascal string to a floating-point value.
TextToFloat	Converts the given null-terminated string to a floating-point value.

**NEW** Note: In recent versions of Delphi's Pascal compiler, the Round function is based on the FPU processor of the CPU. This processor adopts the so-called "Banker's Rounding", which rounds middle values (as 5.5 or 6.5) up and down depending whether they follow an odd or an even number.

## Conclusion

In this chapter we've explored the basic notion of type in Pascal. But the language has another very important feature: It allows programmers to define new custom data types, called user-defined data types. This is the topic of the next chapter.

## Next Chapter: User-Defined Data Types



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 4 User-Defined Data Types

Along with the notion of type, one of the great ideas introduced by the Pascal language is the ability to define new data types in a program. Programmers can define their own data types by means of type constructors, such as subrange types, array types, record types, enumerated types, pointer types, and set types. The most important user-defined data type is the class, which is part of the object-oriented extensions of Object Pascal, not covered in this book.

If you think that type constructors are common in many programming languages, you are right, but Pascal was the first language to introduce the idea in a formal and very precise way. There are still few languages with so many mechanisms to define new types.

### Named and Unnamed Types

These types can be given a name for later use or applied to a variable directly. When you give a name to a type, you must provide a specific section in the code, such as the following:

```
type
  // subrange definition
  Uppercase = 'A'..'Z';

  // array definition
  Temperatures = array [1..24] of Integer;

  // record definition
  Date = record
    Month: Byte;
    Day: Byte;
    Year: Integer;
end;

  // enumerated type definition
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);

  // set definition
  Letters = set of Char;
```

Similar type-definition constructs can be used directly to define a variable without an explicit type name, as in the following code:

```

var
  DecemberTemperature: array [1..31] of Byte;
  ColorCode: array [Red..Violet] of Word;
  Palette: set of Colors;

```

Note: In general, you should avoid using unnamed types as in the code above, because you cannot pass them as parameters to routines or declare other variables of the same type. The type compatibility rules of Pascal, in fact, are based on type names, not on the actual definition of the types. Two variables of two identical types are still not compatible, unless their types have exactly the same name, and unnamed types are given internal names by the compiler. Get used to defining a data type each time you need a variable with a complicated structure, and you won't regret the time you've spent in it.

But what do these type definitions mean? I'll provide some descriptions for those who are not familiar with Pascal type constructs. I'll also try to underline the differences from the same constructs in other programming languages, so you might be interested in reading the following sections even if you are familiar with kind of type definitions exemplified above. Finally, I'll show some Delphi examples and introduce some tools that will allow you to access type information dynamically.

## Subrange Types

A subrange type defines a range of values within the range of another type (hence the name subrange). You can define a subrange of the Integer type, from 1 to 10 or from 100 to 1000, or you can define a subrange of the Char type, as in:

```

type
  Ten = 1..10;
  OverHundred = 100..1000;
  Uppercase = 'A'..'Z';

```

In the definition of a subrange, you don't need to specify the name of the base type. You just need to supply two constants of that type. The original type must be an ordinal type, and the resulting type will be another ordinal type.

When you have defined a subrange, you can legally assign it a value within that range. This code is valid:

```

var
  UppLetter: UpperCase;
begin
  UppLetter := 'F';

```

But this one is not:

```

var
  UppLetter: UpperCase;
begin
  UppLetter := 'e'; // compile-time error

```

Writing the code above results in a compile-time error, "Constant expression violates subrange bounds." If you write the following code instead:

```

var
  UppLetter: Uppercase;
  Letter: Char;
begin
  Letter := 'e';

```



```
UppLetter := Letter;
```

Delphi will compile it. At run-time, if you have enabled the Range Checking compiler option (in the Compiler page of the Project Options dialog box), you'll get a Range check error message.

**Note:** I suggest that you turn on this compiler option while you are developing a program, so it'll be more robust and easier to debug, as in case of errors you'll get an explicit message and not an undetermined behavior. You can eventually disable the option for the final build of the program, to make it a little faster. However, the difference is really small, and for this reason I suggest you to leave all these run-time checks turned on, even in a shipping program. The same holds true for other run-time checking options, such as overflow and stack checking.

## Enumerated Types

Enumerated types constitute another user-defined ordinal type. Instead of indicating a range of an existing type, in an enumeration you list all of the possible values for the type. In other words, an enumeration is a list of values. Here are some examples:

```
type
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);
  Suit = (Club, Diamond, Heart, Spade);
```

Each value in the list has an associated ordinality, starting with zero. When you apply the Ord function to a value of an enumerated type, you get this zero-based value. For example, Ord (Diamond) returns 1.

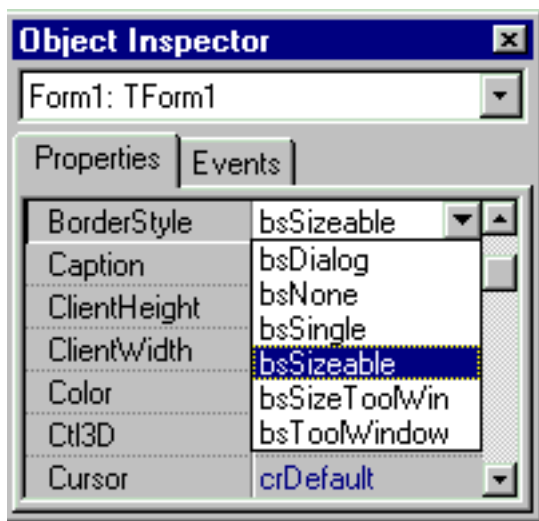
**Note:** Enumerated types can have different internal representations. By default, Delphi uses an 8-bit representation, unless there are more than 256 different values, in which case it uses the 16-bit representation. There is also a 32-bit representation, which might be useful for compatibility with C or C++ libraries. You can actually change the default behavior, asking for a larger representation, by using the SZ compiler directive.

The Delphi VCL (Visual Component Library) uses enumerated types in many places. For example, the style of the border of a form is defined as follows:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsSizeToolWin, bsToolWindow);
```

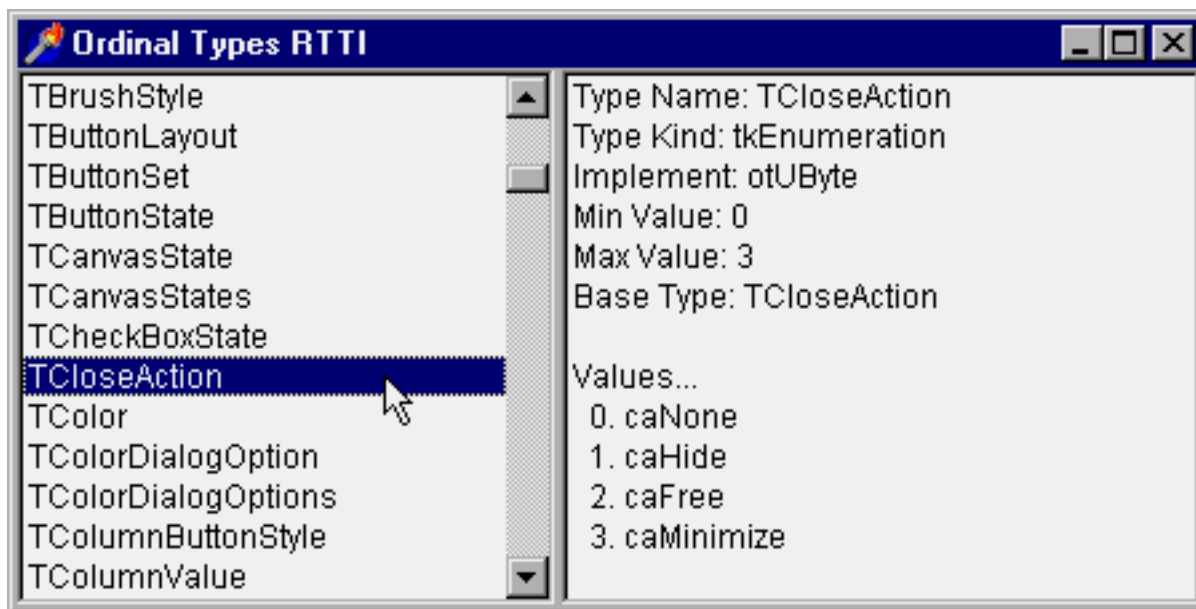
When the value of a property is an enumeration, you usually can choose from the list of values displayed in the Object Inspector, as shown in Figure 4.1.

**Figure 4.1: An enumerated type property in the Object Inspector**



The Delphi Help file generally lists the possible values of an enumeration. As an alternative you can use the OrdType program, available on [www.marcocantu.com](http://www.marcocantu.com), to see the list of the values of each Delphi enumeration, set, subrange, and any other ordinal type. You can see an example of the output of this program in Figure 4.2.

**Figure 4.2:** Detailed information about an enumerated type, as displayed by the OrdType program (available on my web site).



## Set Types

Set types indicate a group of values, where the list of available values is indicated by the ordinal type the set is based onto. These ordinal types are usually limited, and quite often represented by an enumeration or a subrange. If we take the subrange 1..3, the possible values of the set based on it include only 1, only 2, only 3, both 1 and 2, both 1 and 3, both 2 and 3, all the three values, or none of them.

A variable usually holds one of the possible values of the range of its type. A set-type variable, instead, can contain none, one, two, three, or more values of the range. It can even include all of the values. Here is an example of a set:

```
type
```

```
Letters = set of Uppercase;
```

Now I can define a variable of this type and assign to it some values of the original type. To indicate some values in a set, you write a comma-separated list, enclosed within square brackets. The following code shows the assignment to a variable of several values, a single value, and an empty value:

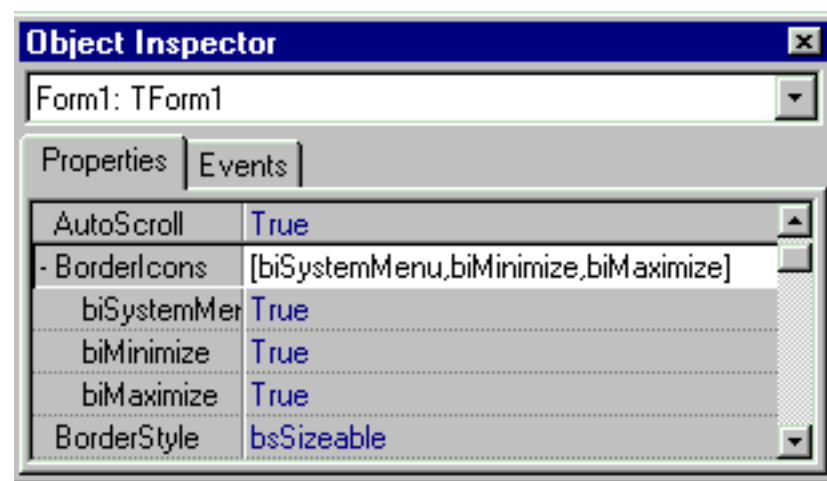
```
var
  Letters1, Letters2, Letters3: Letters;
begin
  Letters1 := ['A', 'B', 'C'];
  Letters2 := ['K'];
  Letters3 := [];
```

In Delphi, a set is generally used to indicate nonexclusive flags. For example, the following two lines of code (which are part of the Delphi library) declare an enumeration of possible icons for the border of a window and the corresponding set type:

```
type
  TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
  TBorderIcons = set of TBorderIcon;
```

In fact, a given window might have none of these icons, one of them, or more than one. When working with the Object Inspector (see Figure 4.3), you can provide the values of a set by expanding the selection (double-click on the property name or click on the plus sign on its left) and toggling on and off the presence of each value.

**Figure 4.3: A set-type property in the Object Inspector**



Another property based on a set type is the style of a font. Possible values indicate a bold, italic, underline, and strikethrough font. Of course the same font can be both italic and bold, have no attributes, or have them all. For this reason it is declared as a set. You can assign values to this set in the code of a program as follows:

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles
```

You can also operate on a set in many different ways, including adding two variables of the same set type (or, to be more precise, computing the union of the two set variables):

```
Font.Style := OldStyle + [fsUnderline]; // two sets
```

Again, you can use the OrdType examples included in the TOOLS directory of the book source code to see the list of possible values of many sets defined by the Delphi component library.

## Array Types

Array types define lists of a fixed number of elements of a specific type. You generally use an index within square brackets to access to one of the elements of the array. The square brackets are used also to specify the possible values of the index when the array is defined. For example, you can define a group of 24 integers with this code:

```
type
  DayTemperatures = array [1..24] of Integer;
```

In the array definition, you need to pass a subrange type within square brackets, or define a new specific subrange type using two constants of an ordinal type. This subrange specifies the valid indexes of the array. Since you specify both the upper and the lower index of the array, the indexes don't need to be zero-based, as is necessary in C, C++, Java, and other programming languages.

Since the array indexes are based on subranges, Delphi can check for their range as we've already seen. An invalid constant subrange results in a compile-time error; and an out-of-range index used at run-time results in a run-time error if the corresponding compiler option is enabled.

Using the array definition above, you can set the value of a DayTemp1 variable of the DayTemperatures type as follows:

```
type
  DayTemperatures = array [1..24] of Integer;

var
  DayTemp1: DayTemperatures;

procedure AssignTemp;
begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;
  DayTemp1 [25] := 67; // compile-time error
```

An array can have more than one dimension, as in the following examples:

```
type
  MonthTemps = array [1..24, 1..31] of Integer;
  YearTemps = array [1..24, 1..31, Jan..Dec] of Integer;
```

These two array types are built on the same core types. So you can declare them using the preceding data types, as in the following code:

```
type
  MonthTemps = array [1..31] of DayTemperatures;
  YearTemps = array [Jan..Dec] of MonthTemps;
```

This declaration inverts the order of the indexes as presented above, but it also allows assignment of whole blocks between variables. For example, the following statement copies January's temperatures to February:

```

var
    ThisYear: YearTemps;
begin
    ...
    ThisYear[Feb] := ThisYear[Jan];

```

You can also define a zero-based array, an array type with the lower bound set to zero. Generally, the use of more logical bounds is an advantage, since you don't need to use the index 2 to access the third item, and so on. Windows, however, uses invariably zero-based arrays (because it is based on the C language), and the Delphi component library tends to do the same.

If you need to work on an array, you can always test its bounds by using the standard `Low` and `High` functions, which return the lower and upper bounds. Using `Low` and `High` when operating on an array is highly recommended, especially in loops, since it makes the code independent of the range of the array. Later, you can change the declared range of the array indices, and the code that uses `Low` and `High` will still work. If you write a loop hard-coding the range of an array you'll have to update the code of the loop when the array size changes. `Low` and `High` make your code easier to maintain and more reliable.

Note: Incidentally, there is no run-time overhead for using `Low` and `High` with arrays. They are resolved at compile-time into constant expressions, not actual function calls. This compile-time resolution of expressions and function calls happens also for many other simple system functions.

Delphi uses arrays mainly in the form of array properties. We have already seen an example of such a property in the `TimeNow` example, to access the `Items` property of a `Listbox` component. I'll show you some more examples of array properties in the next chapter, when discussing Delphi loops.

Note: Delphi 4 introduced dynamic arrays into Object Pascal, that is arrays that can be resized at runtime allocating the proper amount of memory. Using dynamic arrays is easy, but in this discussion of Pascal I felt they were not an proper topic to cover. You can find a description of Delphi's dynamic arrays in Chapter 8.

## Record Types

Record types define fixed collections of items of different types. Each element, or field, has its own type. The definition of a record type lists all these fields, giving each a name you'll use later to access it.

Here is a small listing with the definition of a record type, the declaration of a variable of that type, and few statements using this variable:

```

type
    Date = record
        Year: Integer;
        Month: Byte;
        Day: Byte;
    end;

var
    Birthday: Date;

begin
    Birthday.Year := 1997;
    Birthday.Month := 2;
    Birthday.Day := 14;

```

Classes and objects can be considered an extension of the record type. Delphi libraries tend to use class types instead of record

types, but there are many record types defined by the Windows API.

Record types can also have a variant part; that is, multiple fields can be mapped to the same memory area, even if they have a different data type. (This corresponds to a union in the C language.) Alternatively, you can use these variant fields or groups of fields to access the same memory location within a record, but considering those values from different perspectives. The main uses of this type were to store similar but different data and to obtain an effect similar to that of typecasting (something less useful now that typecasting has been introduced also in Pascal). The use of variant record types has been largely replaced by object-oriented and other modern techniques, although Delphi uses them in some peculiar cases.

The use of a variant record type is not type-safe and is not a recommended programming practice, particularly for beginners. Expert programmers can indeed use variant record types, and the core of the Delphi libraries makes use of them. You won't need to tackle them until you are really a Delphi expert, anyway.

## Pointers

A pointer type defines a variable that holds the memory address of another variable of a given data type (or an undefined type). So a pointer variable indirectly refers to a value. The definition of a pointer type is not based on a specific keyword, but uses a special character instead. This special symbol is the caret (^):

```
type
  PointerToInt = ^Integer;
```

Once you have defined a pointer variable, you can assign to it the address of another variable of the same type, using the @ operator:

```
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  // change the value in two different ways
  X := 10;
  P^ := 20;
```

When you have a pointer P, with the expression P you refer to the address of the memory location the pointer is referring to, and with the expression P^ you refer to the actual content of that memory location. For this reason in the code fragment above P^ corresponds to X.

Instead of referring to an existing memory location, a pointer can refer to a new memory block dynamically allocated (on the heap memory area) with the New procedure. In this case, when you don't need the pointer any more, you'll also have to get rid of the memory you've dynamically allocated, by calling the Dispose procedure.

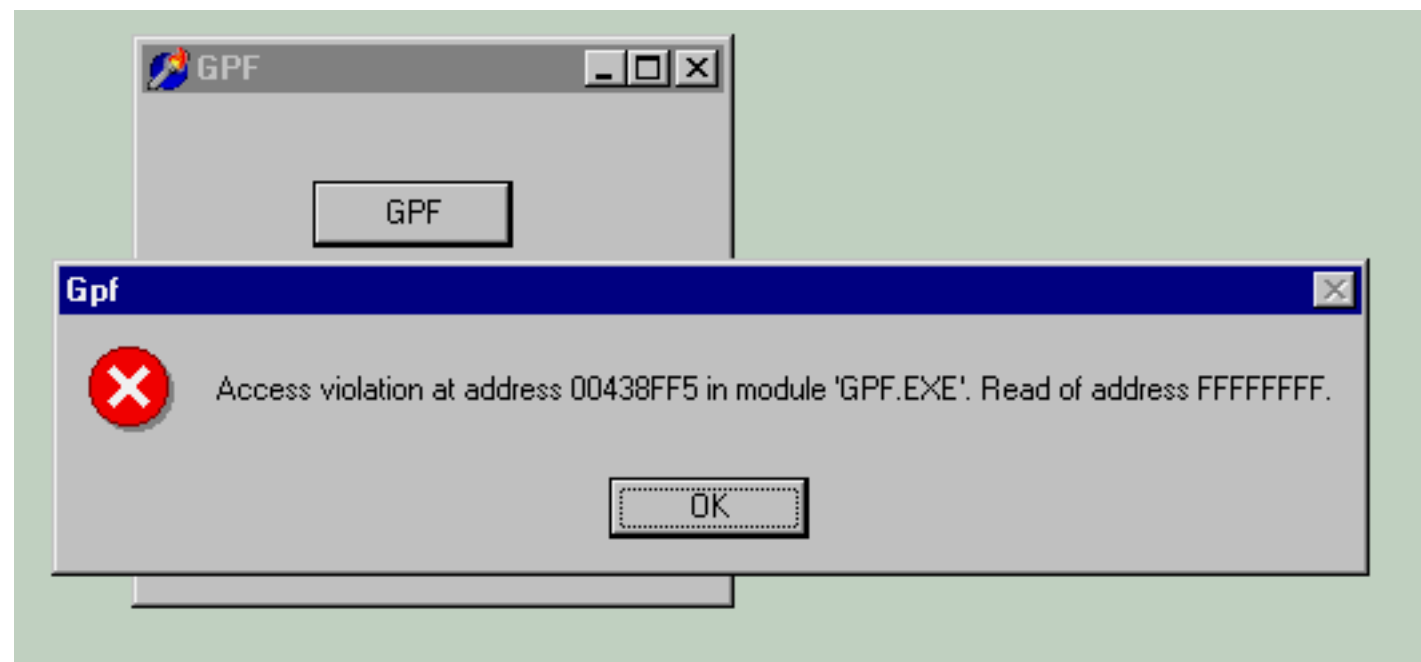
```
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  ShowMessage (IntToStr (P^));
  // termination
  Dispose (P);
end;
```

If a pointer has no value, you can assign the nil value to it. Then you can test whether a pointer is nil to see if it currently refers to a value. This is often used, because dereferencing an invalid pointer causes an access violation (also known as a general protection fault, GPF):

```
procedure TFormGPF.BtnGpfClick(Sender: TObject);
var
  P: ^Integer;
begin
  P := nil;
  ShowMessage (IntToStr (P^));
end;
```

You can see an example of the effect of this code by running the GPF example (or looking at the corresponding Figure 4.4). The example contains also the code fragments shown above.

**Figure 4.4: The system error resulting from the access to a nil pointer, from the GPF example.**



In the same program you can find an example of safe data access. In this second case the pointer is assigned to an existing local variable, and can be safely used, but I've added a safe-check anyway:

```
procedure TFormGPF.BtnSafeClick(Sender: TObject);
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  X := 100;
  if P <> nil then
    ShowMessage (IntToStr (P^));
end;
```

Delphi also defines a Pointer data type, which indicates untyped pointers (such as void\* in the C language). If you use an untyped pointer you should use GetMem instead of New. The GetMem procedure is required each time the size of the memory variable to allocate is not defined.

The fact that pointers are seldom necessary in Delphi is an interesting advantage of this environment. Nonetheless, understanding pointers is important for advanced programming and for a full understanding of the Delphi object model, which uses pointers "behind the scenes."

Note: Although you don't use pointers often in Delphi, you do frequently use a very similar construct—namely, references. Every object instance is really an implicit pointer or reference to its actual data. However, this is completely transparent to the programmer, who uses object variables just like any other data type.

## File Types

Another Pascal-specific type constructor is the file type. File types represent physical disk files, certainly a peculiarity of the Pascal language. You can define a new file data type as follows:

```
type
  IntFile = file of Integer;
```

Then you can open a physical file associated with this structure and write integer values to it or read the current values from the file.

**Author's Note:** Files-based examples were part of older editions of Mastering Delphi and I plan adding them here as well)

The use of files in Pascal is quite straightforward, but in Delphi there are also some components that are capable of storing or loading their contents to or from a file. There is some serialization support, in the form of streams, and there is also database support.

## Conclusion

This chapter discussing user-defined data types complete our coverage of Pascal type system. Now we are ready to look into the statements the language provides to operate on the variables we've defined.

## Next Chapter: Statements





- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 5 Statements

If the data types are one of the foundations of Pascal programming the other are statements. Statements of the programming language are based on keywords and other elements which allow you to indicate to a program a sequence of operations to perform. Statements are often enclosed in procedures or functions, as we'll see in the next chapter. Now we'll just focus on the basic types of commands you can use to create a program.

### Simple and Compound Statements

A Pascal statement is simple when it doesn't contain any other statements. Examples of simple statements are assignment statements and procedure calls. Simple statements are separated by a semicolon:

```
X := Y + Z; // assignment
Randomize; // procedure call
```

Usually, statements are part of a compound statement, marked by begin and end brackets. A compound statement can appear in place of a generic Pascal statement. Here is an example:

```
begin
  A := B;
  C := A * 2;
end;
```

The semicolon after the last statement before the end isn't required, as in the following:

```
begin
  A := B;
  C := A * 2
end;
```

Both versions are correct. The first version has a useless (but harmless) semicolon. This semicolon is, in fact, a null statement; that is, a statement with no code. Notice that, at times, null statements can be used inside loops or in other particular cases.

---

Note: Although these final semicolons serve no purpose, I tend to use them and suggest you do the same. Sometimes after you've written a couple of lines you might want to add one more statement. If the last semicolon is missing you should remember to add it, so it might be better to add it in the first place.

## Assignment Statements

Assignments in Pascal use the colon-equal operator, an odd notation for programmers who are used to other languages. The = operator, which is used for assignments in some other languages, in Pascal is used to test for equality.

Note: By using different symbols for an assignment and an equality test, the Pascal compiler (like the C compiler) can translate source code faster, because it doesn't need to examine the context in which the operator is used to determine its meaning. The use of different operators also makes the code easier for people to read.

## Conditional Statements

A conditional statement is used to execute either one of the statements it contains or none of them, depending on some test. There are two basic flavors of conditional statements: if statements and case statements.

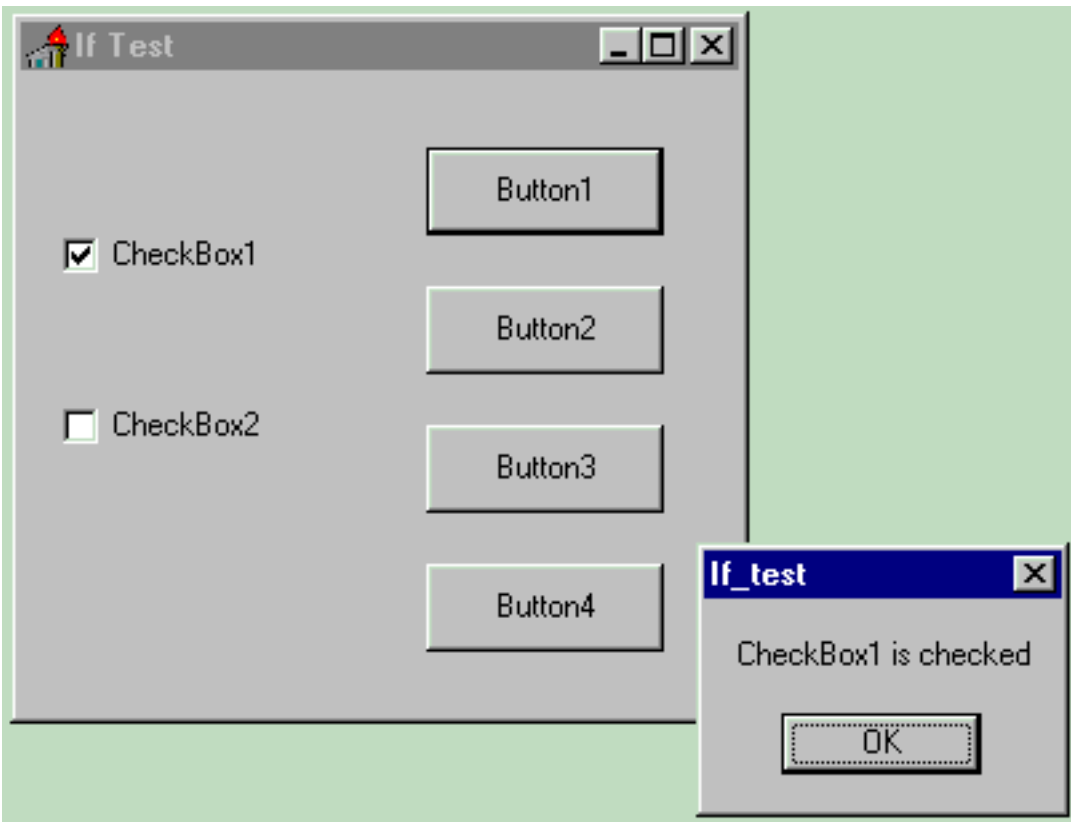
### If Statements

The if statement can be used to execute a statement only if a certain condition is met (if-then), or to choose between two different alternatives (if-then-else). The condition is described with a Boolean expression. A simple Delphi example will demonstrate how to write conditional statements. First create a new application, and put two check boxes and four buttons in the form. Do not change the names of buttons or check boxes, but double-click on each button to add a handler for its OnClick event. Here is a simple if statement for the first button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // simple if statement
    if CheckBox1.Checked then
        ShowMessage ('CheckBox1 is checked')
end;
```

When you click on the button, if the first check box has a check mark in it, the program will show a simple message (see Figure 5.1). I've used the ShowMessage function because it is the simplest Delphi function you can use to display a short message to the user.

**Figure 5.1: The message displayed by the IfTest example when you press the first button and the first check box is checked.**



If you click the button and nothing happens, it means the check box was not checked. In a case like this, it would probably be better to make this more explicit, as with the code for the second button, which uses an if-then-else statement:

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  // if-then-else statement
  if CheckBox2.Checked then
    ShowMessage ('CheckBox2 is checked')
  else
    ShowMessage ('CheckBox2 is NOT checked');
end;

```

Notice that you cannot have a semicolon after the first statement and before the else keyword, or the compiler will issue a syntax error. The if-then-else statement, in fact, is a single statement, so you cannot place a semicolon in the middle of it.

An if statement can be quite complex. The condition can be turned into a series of conditions (using the and, or and not Boolean operators), or the if statement can nest a second if statement. The last two buttons of the IfTest example demonstrate these cases:

```

procedure TForm1.Button3Click(Sender: TObject);
begin
  // statement with a double condition
  if CheckBox1.Checked and CheckBox2.Checked then
    ShowMessage ('Both check boxes are checked')
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  // compound if statement
  if CheckBox1.Checked then
    if CheckBox2.Checked then
      ShowMessage ('CheckBox1 and 2 are checked')
end;

```

```

else
  ShowMessage ('Only CheckBox1 is checked')
else
  ShowMessage (
    'Checkbox1 is not checked, who cares for Checkbox2?')
end;

```

Look at the code carefully and run the program to see if you understand everything. When you have doubts about a programming construct, writing a very simple program such as this can help you learn a lot. You can add more check boxes and increase the complexity of this small example, making any test you like.

## Case Statements

If your if statements become very complex, at times you can replace them with case statements. A case statement consists in an expression used to select a value, a list of possible values, or a range of values. These values are constants, and they must be unique and of an ordinal type. Eventually, there can be an else statement that is executed if none of the labels correspond to the value of the selector. Here are two simple examples:

```

case Number of
  1: Text := 'One';
  2: Text := 'Two';
  3: Text := 'Three';
end;

case MyChar of
  '+' : Text := 'Plus sign';
  '-' : Text := 'Minus sign';
  '*', '/': Text := 'Multiplication or division';
  '0'..'9': Text := 'Number';
  'a'..'z': Text := 'Lowercase character';
  'A'..'Z': Text := 'Uppercase character';
else
  Text := 'Unknown character';
end;

```

## Loops in Pascal

The Pascal language has the typical repetitive statements of most programming languages, including for, while, and repeat statements. Most of what these loops do will be familiar if you've used other programming languages, so I'll cover them only briefly.

### The For Loop

The for loop in Pascal is strictly based on a counter, which can be either increased or decreased each time the loop is executed. Here is a simple example of a for loop used to add the first ten numbers.

```

var
  K, I: Integer;
begin
  K := 0;
  for I := 1 to 10 do
    K := K + I;

```

This same for statement could have been written using a reverse counter:

```

var
  K, I: Integer;
begin
  K := 0;
  for I := 10 downto 1 do
    K := K + I;

```

The for loop in Pascal is less flexible than in other languages (it is not possible to specify an increment different than one), but it is simple and easy to understand. If you want to test for a more complex condition, or to provide a customized counter, you need to use a while or repeat statement, instead of a for loop.

Note: The counter of a for loop doesn't need to be a number. It can be a value of any ordinal type, such as a character or an enumerated type.

## While and Repeat Statements

The difference between the while-do loop and the repeat-until loop is that the code of the repeat statement is always executed at least once. You can easily understand why by looking at a simple example:

```

while (I <= 100) and (J <= 100) do
begin
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
end;

repeat
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);

```

If the initial value of I or J is greater than 100, the statements inside the repeat-until loop are executed once anyway.

The other key difference between these two loops is that the repeat-until loop has a reversed condition. The loop is executed as long as the condition is not met. When the condition is met, the loop terminates. This is the opposite from a while-do loop, which is executed while the condition is true. For this reason I had to reverse the condition in the code above to obtain a similar statement.

## An Example of Loops

To explore the details of loops, let's look at a small Delphi example. The Loops program highlights the difference between a loop with a fixed counter and a loop with an almost random counter. Start with a new blank project, place a list box and two buttons on the main form, and give the buttons a proper name (BtnFor and BtnWhile) by setting their Name property in the Object Inspector. You can also remove the word Btn from the Caption property (and eventually even add the & character to it to activate the following letter as a shortcut key). Here is a summary of the textual description of this form:

```

object Form1: TForm1
  Caption = 'Loops'
  object ListBox1: TListBox ...
  object BtnFor: TButton

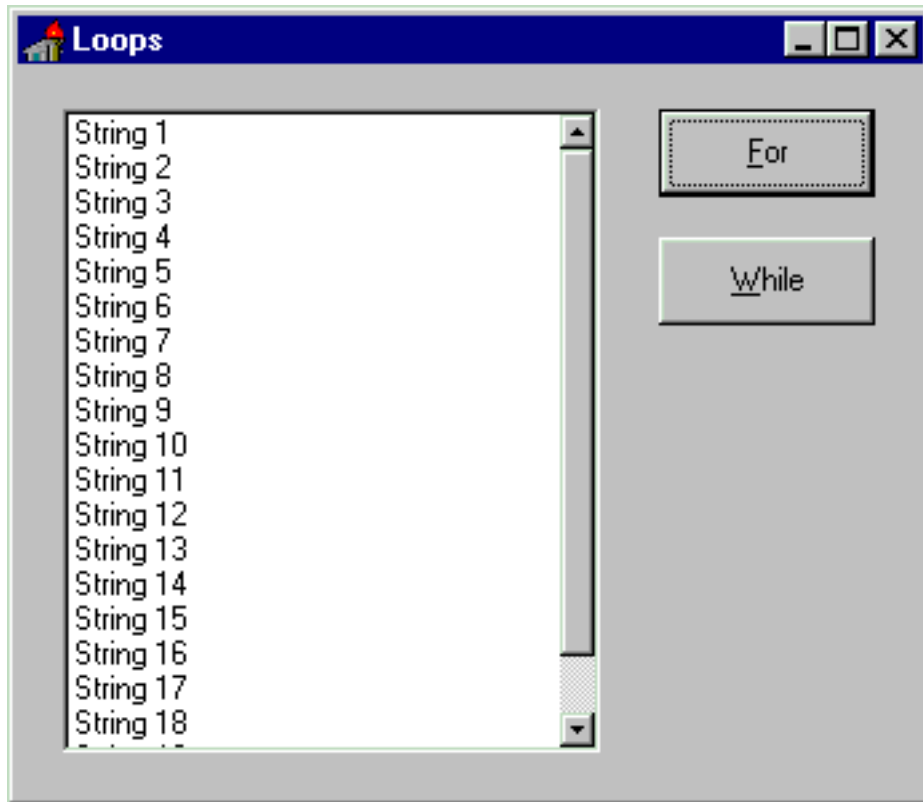
```

```

Caption = '&For'
OnClick = BtnForClick
end
object BtnWhile: TButton
Caption = '&While'
OnClick = BtnWhileClick
end
end
end

```

**Figure 5.2:** Each time you press the For button of the Loops example, the list box is filled with consecutive numbers.



Now we can add some code to the OnClick events of the two buttons. The first button has a simple for loop to display a list of numbers, as you can see in Figure 5.2. Before executing this loop, which adds a number of strings to the Items property of the list box, you need to clear the contents of the list box itself:

```

procedure TForm1.BtnForClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 1 to 20 do
    Listbox1.Items.Add ('String ' + IntToStr (I));
  end;

```

The code associated with the second button is slightly more complex. In this case, there is a while loop based on a counter, which is increased randomly. To accomplish this, I've called the Randomize procedure, which resets the random number generator, and the Random function with a range value of 100. The result of this function is a number between 0 and 99, chosen randomly. The series of random numbers control how many times the while loop is executed.

```

procedure TForm1.BtnWhileClick(Sender: TObject);
var

```

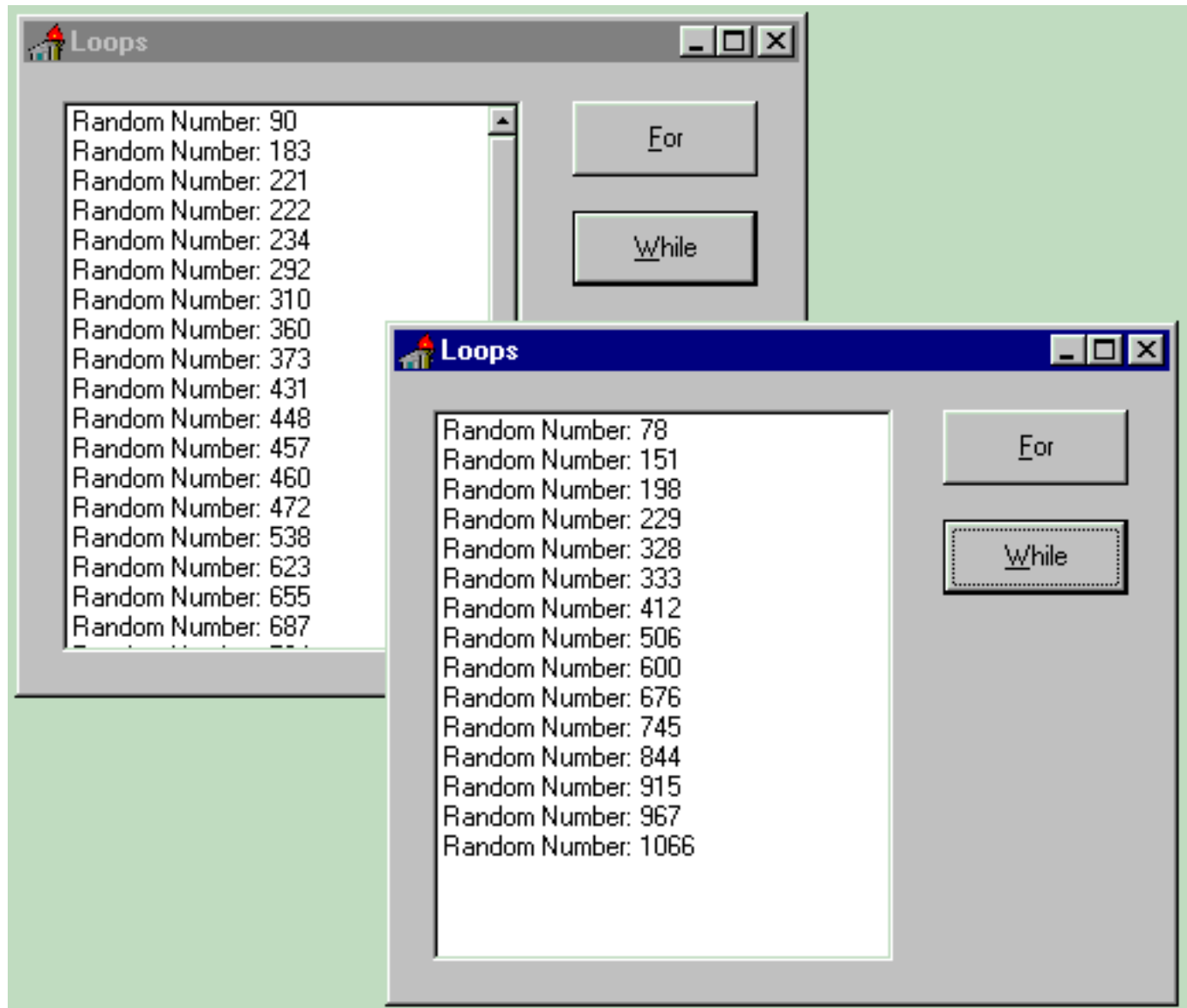
```

I: Integer;
begin
  ListBox1.Items.Clear;
  Randomize;
  I := 0;
  while I < 1000 do
  begin
    I := I + Random (100);
    Listbox1.Items.Add ('Random Number: ' + IntToStr (I));
  end;
end;

```

Each time you click the While button, the numbers are different, because they depend on the random-number generator. Figure 5.3 shows the results from two separate button-clicks. Notice that not only are the generated numbers different each time, but so is the number of items. That is, this while loop is executed a random number of times. If you press the While button several times in a row, you'll see that the list box has a different number of lines.

**Figure 5.3: The contents of the list box of the Loops example change each time you press the While button. Because the loop counter is incremented by a random value, every time you press the button the loop may execute a different number of times.**



Note: You can alter the standard flow of a loop's execution using the Break and Continue system procedures. The first interrupts the loop; the second is used to jump directly to the loop test or counter increment, continuing with the next iteration of the loop (unless the condition is zero or the counter has reached its highest value). Two more system procedures, Exit and Halt, let you immediately return from the current function or procedure or terminate the program.

## The With Statement

The last kind of Pascal statement I'll focus on is the with statement, which used to be peculiar to this programming language (although it has been recently introduced also in JavaScript and Visual Basic) and can be very useful in Delphi programming.

The with statement is nothing but shorthand. When you need to refer to a record type variable (or an object), instead of repeating its name every time, you can use a with statement. For example, while presenting the record type, I wrote this code:

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: Date;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

Using a with statement, I can improve the final part of this code, as follows:

```
begin
  with BirthDay do
  begin
    Year := 1995;
    Month := 2;
    Day := 14;
  end;
```

This approach can be used in Delphi programs to refer to components and other class types. For example, we can rewrite the final part of the last example, Loops, using a with statement to access the items of the list box:

```
procedure TForm1.WhileButtonClick(Sender: TObject);
var
  I: Integer;
begin
  with ListBox1.Items do
  begin
    Clear; // shortcut
    Randomize;
    I := 0;
    while I < 1000 do
    begin
      I := I + Random (100);
      // shortcut:
```



```

        Add ('Random Number: ' + IntToStr (I));
    end;
end;
end;

```

When you work with components or classes in general, the with statement allows you to skip writing some code, particularly for nested fields. For example, suppose that you need to change the Width and the Color of the drawing pen for a form. You can write the following code:

```

Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clRed;

```

But it is certainly easier to write this code:

```

with Form1.Canvas.Pen do
begin
    Width := 2;
    Color := clRed;
end;

```

When you are writing complex code, the with statement can be effective and spares you the declaration of some temporary variables, but it has a drawback. It can make the code less readable, particularly when you are working with different objects that have similar or corresponding properties.

A further drawback is that using the with statement can allow subtle logical errors in the code that the compiler will not detect. For example:

```

with Button1 do
begin
    Width := 200;
    Caption := 'New Caption';
    Color := clRed;
end;

```

This code changes the Caption and the Width of the button, but it affects the Color property of the form, not that of the button! The reason is that the TButton components don't have the Color property, and since the code is executed for a form object (we are writing a method of the form) this object is accessed by default. If we had instead written:

```

Button1.Width := 200;
Button1.Caption := 'New Caption';
Button1.Color := clRed; // error!

```

the compiler would have issued an error. In general, we can say that since the with statement introduces new identifiers in the current scope, we might hide existing identifiers, or wrongfully access another identifier in the same scope (as in the first version of this code fragment). Even considering this kind of drawback, I suggest you get used to with statements, because they can be really very handy, and at times even make the code more readable.

You should, however, avoid using multiple with statements, such as:

```

with ListBox1, Button1 do...

```

The code following this would probably be highly unreadable, because for each property defined in this block you would need to think about which component it refers to, depending on the respective properties and the order of the components in the with statement.

Note: Speaking of readability, Pascal has no `endif` or `endcase` statement. If an `if` statement has a begin-end block, then the end of the block marks the end of the statement. The `case` statement, instead, is always terminated by an `end`. All these end statements, often found one after the other, can make the code difficult to follow. Only by tracing the indentations can you see which statement a particular end refers to. A common way to solve this problem and make the code more readable is to add a comment after the end statement indicating its role, as in:

```
if ... then
  ...
end; // if
```

## Conclusion

This chapter has described how to code conditional statements and loops. Instead of writing long lists of such statements, programs are usually split in routines, procedures or functions. This is the topic of the next chapter, which introduces also some advanced elements of Pascal.

## Next Chapter: Procedures



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 6 Procedures and Functions

Another important idea emphasized by Pascal is the concept of the routine, basically a series of statements with a unique name, which can be activated many times by using their name. This way you avoid repeating the same statements over and over, and having a single version of the code you can easily modify it all over the program. From this point of view, you can think of routines as the basic code encapsulation mechanism. I'll get back to this topic with an example after I introduce the Pascal routines syntax.

### Pascal Procedures and Functions

In Pascal, a routine can assume two forms: a procedure and a function. In theory, a procedure is an operation you ask the computer to perform, a function is a computation returning a value. This difference is emphasized by the fact that a function has a result, a return value, while a procedure doesn't. Both types of routines can have multiple parameters, of given data types.

In practice, however, the difference between functions and procedures is very limited: you can call a function to perform some work and then skip the result (which might be an optional error code or something like that) or you can call a procedure which passes a result within its parameters (more on reference parameters later in this chapter).

Here are the definitions of a procedure and two versions of the same function, using a slightly different syntax:

```
procedure Hello;
begin
  ShowMessage ('Hello world!');
end;

function Double (Value: Integer) : Integer;
begin
  Double := Value * 2;
end;

// or, as an alternative
function Double2 (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

The use of Result instead of the function name to assign the return value of a function is becoming quite popular, and tends to make the code more readable, in my opinion.

Once these routines have been defined, you can call them one or more times. You call the procedure to make it perform its task, and call a function to compute the value:

```
procedure TForm1.Button1Click (Sender: TObject);
begin
```

```

Hello;
end;

procedure TForm1.Button2Click (Sender: TObject);
var
  X, Y: Integer;
begin
  X := Double (StrToInt (Edit1.Text));
  Y := Double (X);
  ShowMessage (IntToStr (Y));
end;

```

**Note:** For the moment don't care about the syntax of the two procedures above, which are actually methods. Simply place two buttons on a Delphi form, click on them at design time, and the Delphi IDE will generate the proper support code: Now you simply have to fill in the lines between begin and end. To compile the code above you need to add also an Edit control to the form.

Now we can get back to the encapsulation code concept I've introduced before. When you call the Double function, you don't need to know the algorithm used to implement it. If you later find out a better way to double numbers, you can easily change the code of the function, but the calling code will remain unchanged (although executing it will be faster!). The same principle can be applied to the Hello procedure: We can modify the program output by changing the code of this procedure, and the Button2Click method will automatically change its effect. Here is how we can change the code:

```

procedure Hello;
begin
  MessageDlg ('Hello world!', mtInformation, [mbOK]);
end;

```

**Tip:** When you call an existing Delphi function or procedure, or any VCL method, you should remember the number and type of the parameters. Delphi editor helps you by suggesting the parameters list of a function or procedure with a fly-by hint as soon as you type its name and the open parenthesis. This feature is called Code Parameters and is part of the Code Insight technology.

## Reference Parameters

Pascal routines allow parameter passing by value and by reference. Passing parameters by value is the default: the value is copied on the stack and the routine uses and manipulates the copy, not the original value.

Passing a parameter by reference means that its value is not copied onto the stack in the formal parameter of the routine (avoiding a copy often means that the program executes faster). Instead, the program refers to the original value, also in the code of the routine. This allows the procedure or function to change the value of the parameter. Parameter passing by reference is expressed by the var keyword.

This technique is available in most programming languages. It isn't present in C, but has been introduced in C++, where you use the & (pass by reference) symbol. In Visual Basic every parameter not specified as ByVal is passed by reference.

Here is an example of passing a parameter by reference using the var keyword:

```

procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;

```

In this case, the parameter is used both to pass a value to the procedure and to return a new value to the calling code. When you write:

```

var
  X: Integer;
begin
  X := 10;

```

```
DoubleTheValue (X);
```

the value of the X variable becomes 20, because the function uses a reference to the original memory location of X, affecting its initial value.

Passing parameters by reference makes sense for ordinal types, for old-fashioned strings, and for large records. Delphi objects, in fact, are invariably passed by value, because they are references themselves. For this reason passing an object by reference makes little sense (apart from very special cases), because it corresponds to passing a "reference to a reference."

Delphi long strings have a slightly different behavior: they behave as references, but if you change one of the string variables referring to the same string in memory, this is copied before updating it. A long string passed as a value parameter behaves as a reference only in terms of memory usage and speed of the operation. But if you modify the value of the string, the original value is not affected. On the contrary, if you pass the long string by reference, you can alter the original value.

Delphi 3 introduced a new kind of parameter, out. An out parameter has no initial value and is used only to return a value. These parameters should be used only for COM procedures and functions; in general, it is better to stick with the more efficient var parameters. Except for not having an initial value, out parameters behave like var parameters.

## Constant Parameters

As an alternative to reference parameters, you can use a const parameter. Since you cannot assign a new value to a constant parameter inside the routine, the compiler can optimize parameter passing. The compiler can choose an approach similar to reference parameters (or a const reference in C++ terms), but the behavior will remain similar to value parameters, because the original value won't be affected by the routine.

In fact, if you try to compile the following (silly) code, Delphi will issue an error:

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;      // compiler error
  Result := Value;
end;
```

## Open Array Parameters

Unlike C, a Pascal function or procedure always has a fixed number of parameters. However, there is a way to pass a varying number of parameters to a routine using an open array.

The basic definition of an open array parameter is that of a typed open array. This means you indicate the type of the parameter but do not know how many elements of that type the array is going to have. Here is an example of such a definition:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

Using High(A) we can get the size of the array. Notice also the use of the return value of the function, Result, to store temporary values. You can call this function by passing to it an array of Integer expressions:

```
X := Sum ([10, Y, 27*I]);
```

Given an array of Integers, of any size, you can pass it directly to a routine requiring an open array parameter or, instead, you can call the Slice function to pass only a portion of the array (as indicated by its second parameter). Here is an example, where the complete array is passed as

parameter:

```

var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);

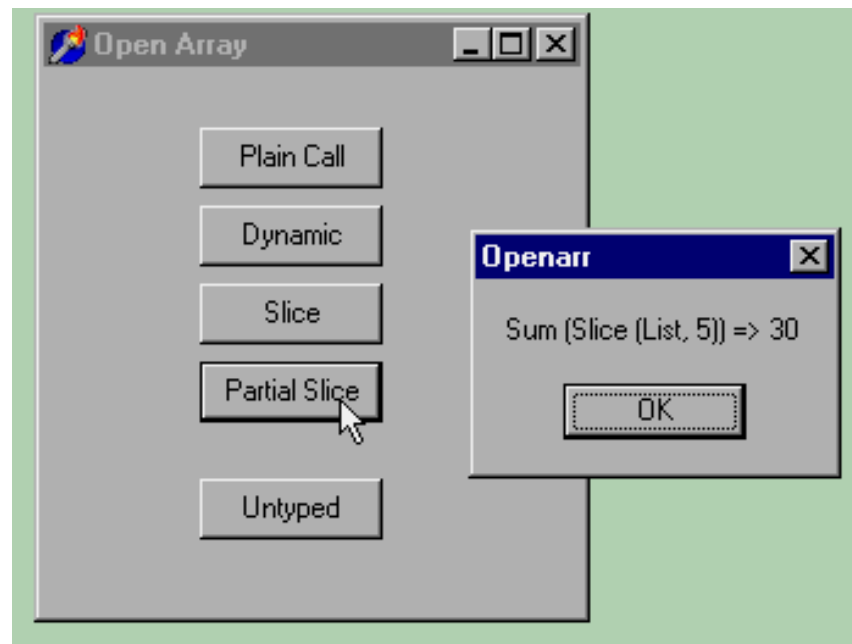
```

If you want to pass only a portion of the array to the Slice function, simply call it this way:

```
X := Sum (Slice (List, 5));
```

You can find all the code fragments presented in this section in the OpenArr example (see Figure 6.1, later on, for the form).

**Figure 6.1: The OpenArr example when the Partial Slice button is pressed**



Typed open arrays in Delphi 4 are fully compatible with dynamic arrays (introduced in Delphi 4 and covered in Chapter 8). Dynamic arrays use the same syntax as open arrays, with the difference that you can use a notation such as array of Integer to declare a variable, not just to pass a parameter.

## Type-Variant Open Array Parameters

Besides these typed open arrays, Delphi allows you to define type-variant or untyped open arrays. This special kind of array has an undefined number of values, which can be handy for passing parameters.

Technically, the construct array of const allows you to pass an array with an undefined number of elements of different types to a routine at once. For example, here is the definition of the Format function (we'll see how to use this function in [Chapter 7](#), covering strings):

```

function Format (const Format: string;
  const Args: array of const): string;

```

The second parameter is an open array, which gets an undefined number of values. In fact, you can call this function in the following ways:

```
N := 20;
S := 'Total: ';
Label1.Caption := Format ('Total: %d', [N]);
Label2.Caption := Format ('Int: %d, Float: %f', [N, 12.4]);
Label3.Caption := Format ('%s %d', [S, N * 2]);
```

Notice that you can pass a parameter as either a constant value, the value of a variable, or an expression. Declaring a function of this kind is simple, but how do you code it? How do you know the types of the parameters? The values of a type-variant open array parameter are compatible with the TVarRec type elements.

**Note:** Do not confuse the TVarRec record with the TVarData record used by the Variant type itself. These two structures have a different aim and are not compatible. Even the list of possible types is different, because TVarRec can hold Delphi data types, while TVarData can hold OLE data types.

The TVarRec record has the following structure:

```
type
  TVarRec = record
    case Byte of
      vtInteger:   (VInteger: Integer; VType: Byte);
      vtBoolean:   (VBoolean: Boolean);
      vtChar:      (VChar: Char);
      vtExtended:  (VExtended: PExtended);
      vtString:    (VString: PShortString);
      vtPointer:   (VPointer: Pointer);
      vtPChar:     (VPChar: PChar);
      vtObject:    (VObject: TObject);
      vtClass:     (VClass: TClass);
      vtWideChar:  (VWideChar: WideChar);
      vtPWideChar: (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:  (VCurrency: PCurrency);
      vtVariant:   (VVariant: PVariant);
      vtInterface: (VInterface: Pointer);
    end;
```

Each possible record has the VType field, although this is not easy to see at first because it is declared only once, along with the actual Integer-size data (generally a reference or a pointer).

Using this information we can actually write a function capable of operating on different data types. In the SumAll function example, I want to be able to sum values of different types, transforming strings to integers, characters to the corresponding order value, and adding 1 for True Boolean values. The code is based on a case statement, and is quite simple, although we have to dereference pointers quite often:

```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger: Result :=
        Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtChar:
        Result := Result + Ord (Args [I].VChar);
```

```

vtExtended:
  Result := Result + Args [I].VExtended^;
vtString, vtAnsiString:
  Result := Result + StrToIntDef ((Args [I].VString^), 0);
vtWideChar:
  Result := Result + Ord (Args [I].VWideChar);
vtCurrency:
  Result := Result + Args [I].VCurrency^;
end; // case
end;

```

end;

I've added this code to the OpenArr example, which calls the SumAll function when a given button is pressed:

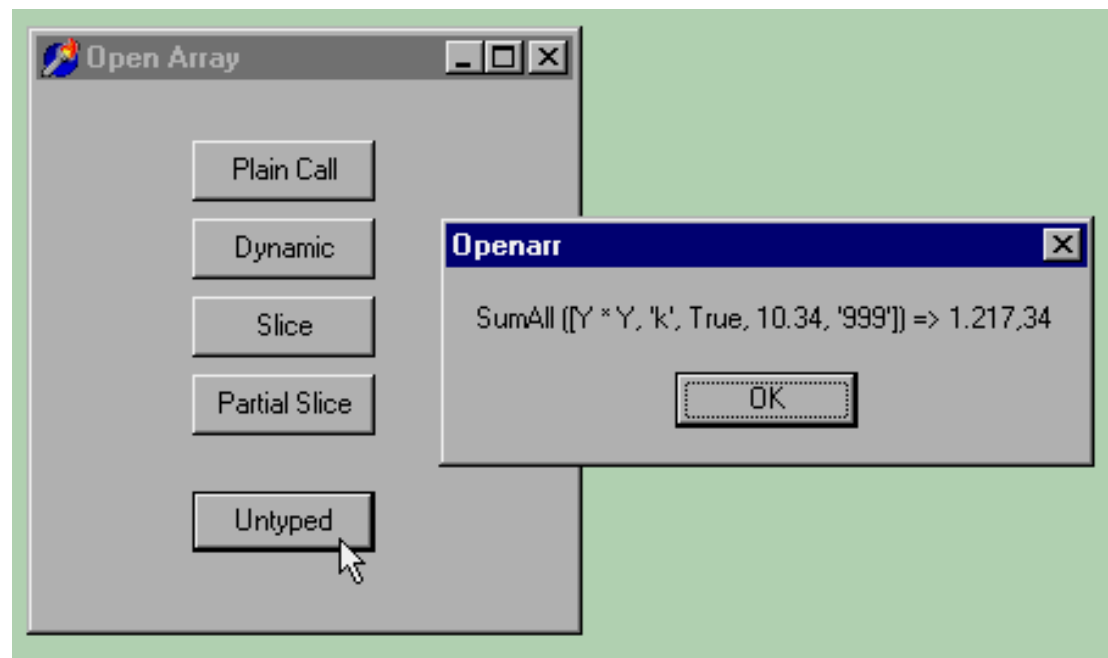
```

procedure TForm1.Button4Click(Sender: TObject);
var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34, '99999']);
  ShowMessage (Format (
    'SumAll ([Y*Y, ''k'', True, 10.34, ''99999'']) => %n', [X]));
end;

```

You can see the output of this call, and the form of the OpenArr example, in Figure 6.2.

**Figure 6.2: The form of the OpenArr example, with the message box displayed when the Untyped button is pressed.**



## Delphi Calling Conventions

The 32-bit version of Delphi has introduced a new approach to passing parameters, known as fastcall: Whenever possible, up to three parameters can be passed in CPU registers, making the function call much faster. The fast calling convention (used by default in Delphi 3) is indicated by the register keyword.

The problem is that this is the default convention, and functions using it are not compatible with Windows: the functions of the Win32 API must be declared using the stdcall calling convention, a mixture of the original Pascal calling convention of the Win16 API and the cdecl calling convention of the C language.



There is generally no reason not to use the new fast calling convention, unless you are making external Windows calls or defining Windows callback functions. We'll see an example using the stdcall convention before the end of this chapter. You can find a summary of Delphi calling conventions in the Calling conventions topic under Delphi help.

## What Is a Method?

If you have already worked with Delphi or read the manuals, you have probably heard about the term "method". A method is a special kind of function or procedure that is related to a data type, a class. In Delphi, every time we handle an event, we need to define a method, generally a procedure. In general, however, the term method is used to indicate both functions and procedures related to a class.

We have already seen a number of methods in the examples in this and the previous chapters. Here is an empty method automatically added by Delphi to the source code of a form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    {here goes your code}
end;
```

## Forward Declarations

When you need to use an identifier (of any kind), the compiler must have already seen some sort of declaration to know what the identifier refers to. For this reason, you usually provide a full declaration before using any routine. However, there are cases in which this is not possible. If procedure A calls procedure B, and procedure B calls procedure A, when you start writing the code, you will need to call a routine for which the compiler still hasn't seen a declaration.

If you want to declare the existence of a procedure or function with a certain name and given parameters, without providing its actual code, you can write the procedure or function followed by the forward keyword:

```
procedure Hello; forward;
```

Later on, the code should provide a full definition of the procedure, but this can be called even before it is fully defined. Here is a silly example, just to give you the idea:

```
procedure DoubleHello; forward;

procedure Hello;
begin
    if MessageDlg ('Do you want a double message?',
        mtConfirmation, [mbYes, mbNo], 0) = mrYes then
        DoubleHello
    else
        ShowMessage ('Hello');
end;

procedure DoubleHello;
begin
    Hello;
    Hello;
end;
```

This approach allows you to write mutual recursion: DoubleHello calls Hello, but Hello might call DoubleHello, too. Of course there must be a condition to terminate the recursion, to avoid a stack overflow. You can find this code, with some slight changes, in the DoubleH example.

Although a forward procedure declaration is not very common in Delphi, there is a similar case that is much more frequent. When you declare a procedure or function in the interface portion of a unit (more on units in the next chapter), it is considered a forward declaration, even if the forward keyword is not present. Actually you cannot write the body of a routine in the interface portion of a unit. At the same time, you must provide in the same unit the actual implementation of each routine you have declared.

The same holds for the declaration of a method inside a class type that was automatically generated by Delphi (as you added an event to a form or its components). The event handlers declared inside a TForm class are forward declarations: the code will be provided in the implementation portion of the unit. Here is an excerpt of the source code of an earlier example, with the declaration of the Button1Click method:

```
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;
```

## Procedural Types

Another unique feature of Object Pascal is the presence of procedural types. These are really an advanced language topic, which only a few Delphi programmers will use regularly. However, since we will discuss related topics in later chapters (specifically, method pointers, a technique heavily used by Delphi), it's worth a quick look at them here. If you are a novice programmer, you can skip this section for now, and come back to it when you feel ready.

In Pascal, there is the concept of procedural type (which is similar to the C language concept of function pointer). The declaration of a procedural type indicates the list of parameters and, in the case of a function, the return type. For example, you can declare a new procedural type, with an Integer parameter passed by reference, with this code:

```
type
  IntProc = procedure (var Num: Integer);
```

This procedural type is compatible with any routine having exactly the same parameters (or the same function signature, to use C jargon). Here is an example of a compatible routine:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

Note: In the 16-bit version of Delphi, routines must be declared using the far directive in order to be used as actual values of a procedural type.

Procedural types can be used for two different purposes: you can declare variables of a procedural type or pass a procedural type (that is, a function pointer) as parameter to another routine. Given the preceding type and procedure declarations, you can write this code:

```
var
  IP: IntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

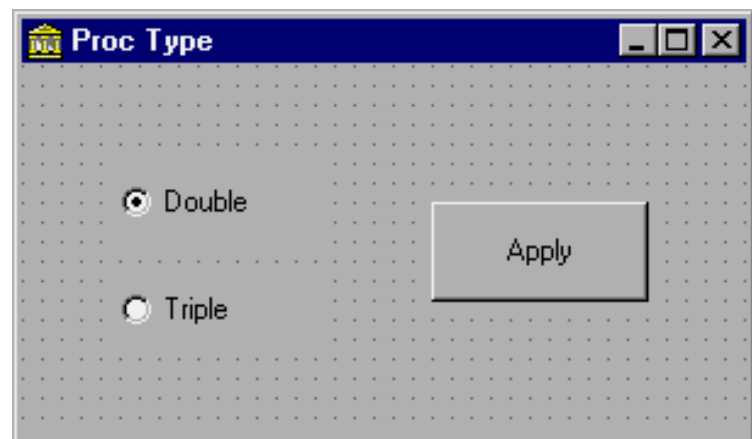
This code has the same effect as the following shorter version:

```
var
  X: Integer;
begin
  X := 5;
  DoubleTheValue (X);
end;
```

The first version is clearly more complex, so why should we use it? In some cases, being able to decide which function to call and actually calling it later on can be useful. It is possible to build a complex example showing this approach. However, I prefer to let you explore a fairly simple one, named ProcType. This example is more complex than those we have seen so far, to make the situation a little more realistic.

Simply create a blank project and place two radio buttons and a push button, as shown in Figure 6.3. This example is based on two procedures. One procedure is used to double the value of the parameter. This procedure is similar to the version I've already shown in this section. A second procedure is used to triple the value of the parameter, and therefore is named TripleTheValue:

**Figure 6.3: The form of the ProcType example.**



```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
  ShowMessage ('Value tripled: ' + IntToStr (Value));
end;
```

Both procedures display what is going on, to let us know that they have been called. This is a simple debugging feature you can use to test whether or when a certain portion of code is executed, instead of adding a breakpoint in it.

Each time a user presses the Apply button, one of the two procedures is executed, depending on the status of the radio buttons. In fact, when you have two radio buttons in a form, only one of them can be selected at a time. This code could have been implemented by testing the value of the radio buttons inside the code for the OnClick event of the Apply button. To demonstrate the use of procedural types, I've instead used a longer but interesting approach. Each time a user clicks on one of the two radio buttons, one of the procedures is stored in a variable:

```
procedure TForm1.DoubleRadioButtonClick(Sender: TObject);
begin
  IP := DoubleTheValue;
end;
```

When the user clicks on the push button, the procedure we have stored is executed:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  IP (X);
end;
```

To allow three different functions to access the IP and X variables, we need to make them visible to the whole form; they cannot be declared locally (inside one of the methods). A solution to this problem is to place these variables inside the form declaration:

```
type
  TForm1 = class(TForm)
    ...
  private
    { Private declarations }
```

```

IP: IntProc;
X: Integer;
end;

```

We will see exactly what this means in the next chapter, but for the moment, you need to modify the code generated by Delphi for the class type as indicated above, and add the definition of the procedural type I've shown before. To initialize these two variables with suitable values, we can handle the OnCreate event of the form (select this event in the Object Inspector after you have activated the form, or simply double-click on the form). I suggest you refer to the listing to study the details of the source code of this example.

You can see a practical example of the use of procedural types in Chapter 9, in the section A Windows Callback Function.

## Function Overloading

The idea of overloading is simple: The compiler allows you to define two functions or procedures using the same name, provided that the parameters are different. By checking the parameters, in fact, the compiler can determine which of the versions of the routine you want to call.

Consider this series of functions extracted from the Math unit of the VCL:

```

function Min (A,B: Integer): Integer; overload;
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;

```

When you call `Min (10, 20)`, the compiler easily determines that you're calling the first function of the group, so the return value will be an Integer.

The basic rules are two:

- Each version of the routine must be followed by the `overload` keyword.
- The differences must be in the number or type of the parameters, or both. The return type, instead, cannot be used to distinguish among two routines.

Here are three overloaded versions of a `ShowMsg` procedure I've added to the `OverDef` example (an application demonstrating overloading and default parameters):

```

procedure ShowMsg (str: string); overload;
begin
  MessageDlg (str, mtInformation, [mbOK], 0);
end;

procedure ShowMsg (FormatStr: string;
  Params: array of const); overload;
begin
  MessageDlg (Format (FormatStr, Params),
    mtInformation, [mbOK], 0);
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
  ShowMsg (IntToStr (I) + ' ' + Str);
end;

```

The three functions show a message box with a string, after optionally formatting the string in different ways. Here are the three calls of the program:

```

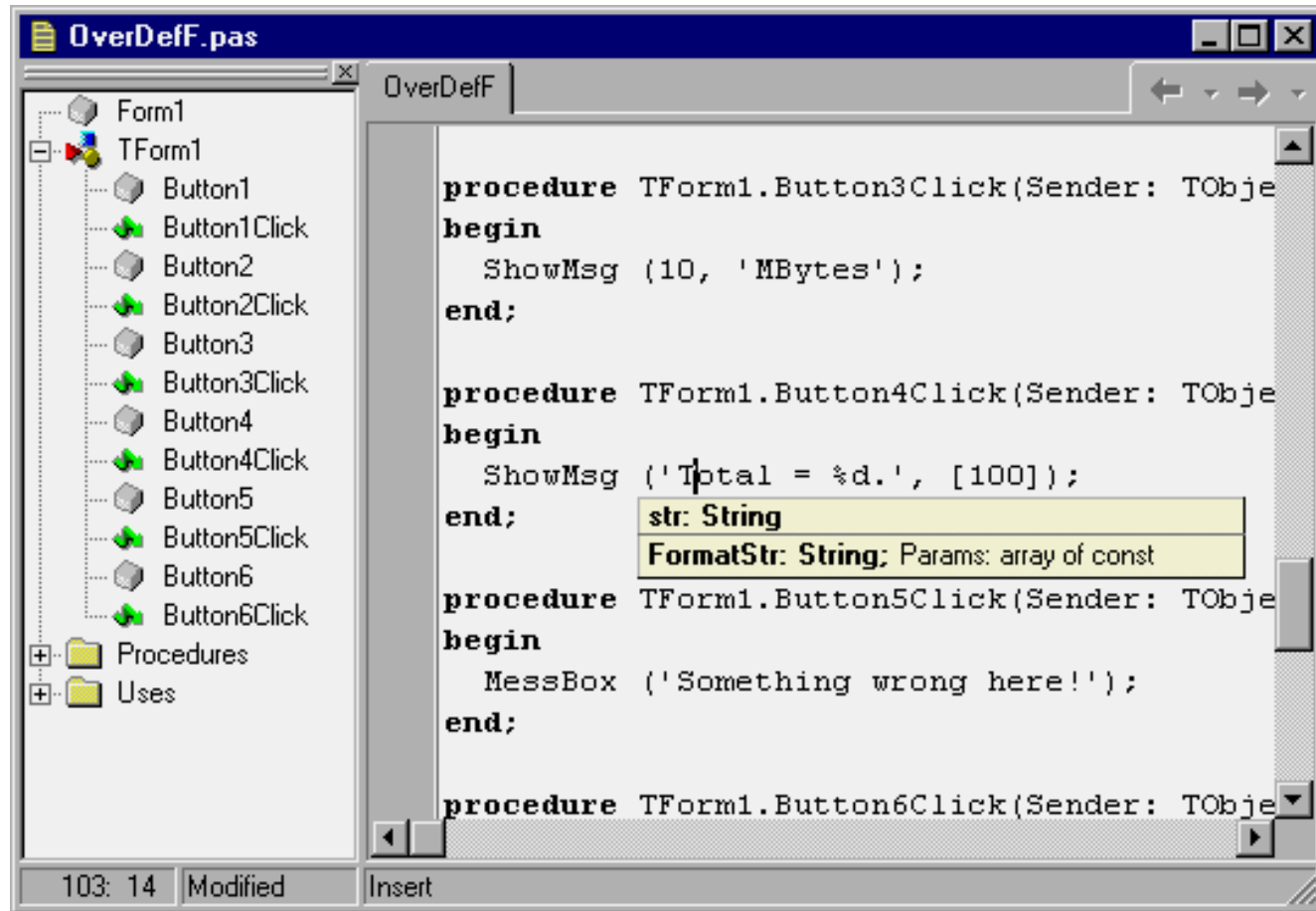
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);

```

```
ShowMsg (10, 'MBytes');
```

What surprised me in a positive way is that Delphi's Code Parameters technology works very nicely with overloaded procedures and functions. As you type the open parenthesis after the routine name, all the available alternatives are listed. As you enter the parameters, Delphi uses their type to determine which of the alternatives are still available. In Figure 6.4 you can see that after starting to type a constant string Delphi shows only the compatible versions (omitting the version of the ShowMsg procedure that has an integer as first parameter).

**Figure 6.4:** The multiple alternatives offered by Code Parameters for overloaded routines are filtered according to the parameters already available.



The fact that each version of an overloaded routine must be properly marked implies that you cannot overload an existing routine of the same unit that is not marked with the overload keyword. (The error message you get when you try is: "Previous declaration of '<name>' was not marked with the 'overload' directive.") However, you can overload a routine that was originally declared in a different unit. This is for compatibility with previous versions of Delphi, which allowed different units to reuse the same routine name. Notice, anyway, that this special case is not an extra feature of overloading, but an indication of the problems you can face.

For example, you can add to a unit the following code:

```
procedure MessageDlg (str: string); overload;
begin
  Dialogs.MessageDlg (str, mtInformation, [mbOK], 0);
end;
```

This code doesn't really overload the original MessageDlg routine. In fact if you write:

```
MessageDlg ('Hello');
```

you'll get a nice error message indicating that some of the parameters are missing. The only way to call the local version instead of the one of the VCL is to refer explicitly to the local unit, something that defeats the idea of overloading:

```
OverDefF.MessageDlg ('Hello');
```

## Default Parameters

A related new feature of Delphi 4 is that you can give a default value for the parameter of a function, and you can call the function with or without the parameter. Let me show an example. We can define the following encapsulation of the MessageBox method of the Application global object, which uses PChar instead of strings, providing two default parameters:

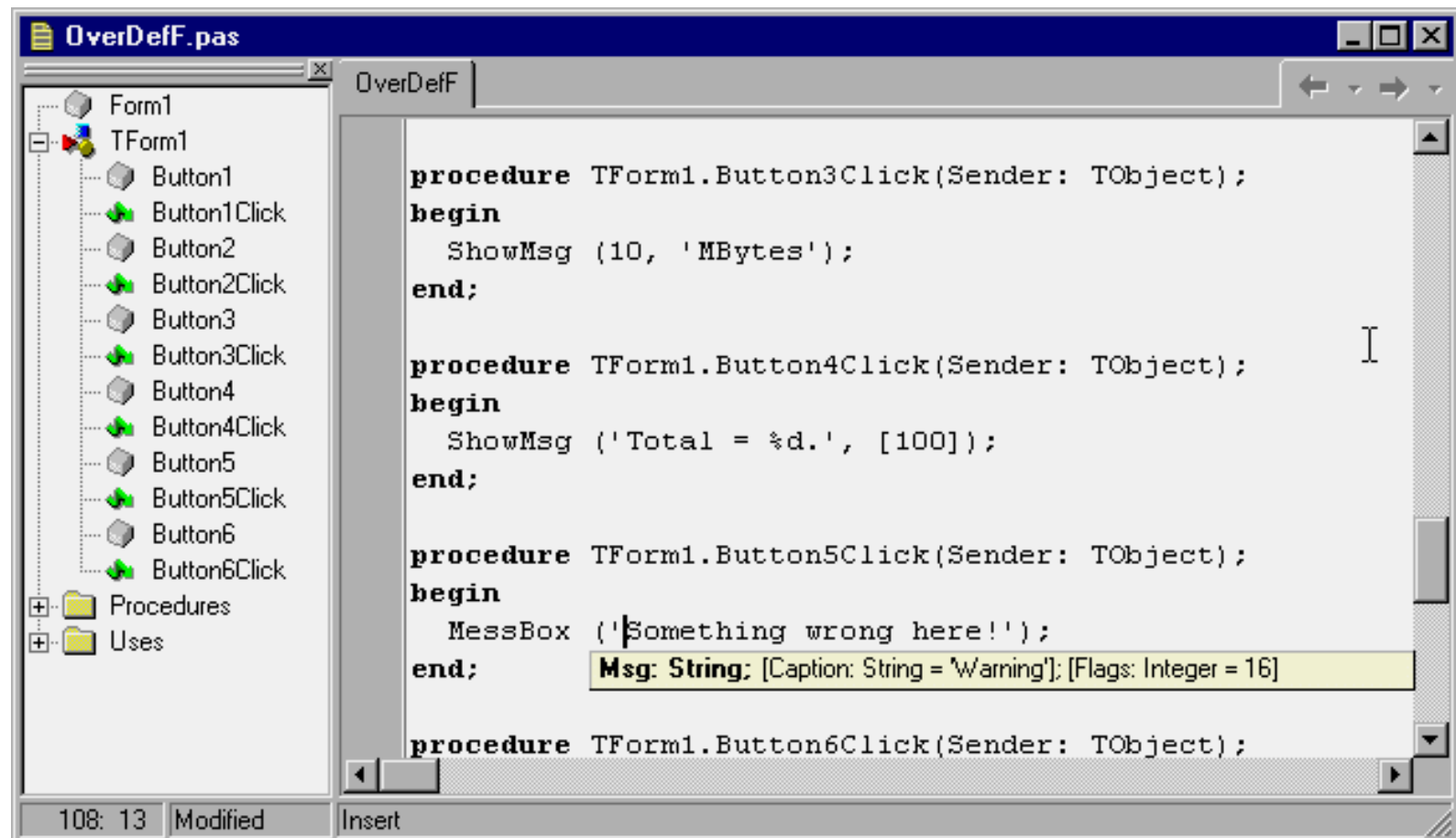
```
procedure MessBox (Msg: string;
  Caption: string = 'Warning';
  Flags: LongInt = mb_OK or mb_IconHand);
begin
  Application.MessageBox (PChar (Msg),
    PChar (Caption), Flags);
end;
```

With this definition, we can call the procedure in each of the following ways:

```
MessBox ('Something wrong here!');
MessBox ('Something wrong here!', 'Attention');
MessBox ('Hello', 'Message', mb_OK);
```

In Figure 6.5 you can see that Delphi's Code Parameters properly use a different style to indicate the parameters that have a default value, so you can easily determine which parameters can be omitted.

**Figure 6.5:** Delphi's Code Parameters mark out with square brackets the parameters that have default values; you can omit these in the call.



Notice that Delphi doesn't generate any special code to support default parameters; nor does it create multiple copies of the routines. The missing parameters are simply added by the compiler to the calling code.

There is one important restriction affecting the use of default parameters: You cannot "skip" parameters. For example, you can't pass the third parameter to the function after omitting the second one:

```
MessBox ('Hello', mb_OK); // error
```

This is the main rule for default parameters: In a call, you can only omit parameters starting from the last one. In other words, if you omit a parameter you must omit also the following ones.

There are a few other rules for default parameters as well:

- Parameters with default values must be at the end of the parameters list.
- Default values must be constants. Obviously, this limits the types you can use with default parameters. For example, a dynamic array or an interface type cannot have a default parameter other than nil; records cannot be used at all.
- Default parameters must be passed by value or as const. A reference (var) parameter cannot have a default value.

Using default parameters and overloading at the same time can cause quite a few problems, as the two features might conflict. For example, if I add to the previous example the following new version of the ShowMsg procedure:

```
procedure ShowMsg (Str: string; I: Integer = 0); overload;
begin
  MessageDlg (Str + ': ' + IntToStr (I),
    mtInformation, [mbOK], 0);
end;
```

then the compiler won't complain-this is a legal definition. However, the call:

```
ShowMsg ('Hello');
```

is flagged by the compiler as Ambiguous overloaded call to 'ShowMsg'. Notice that this error shows up in a line of code that compiled correctly before the new overloaded definition. In practice, we have no way to call the ShowMsg procedure with one string parameter, as the compiler doesn't know whether we want to call the version with only the string parameter or the one with the string parameter and the integer parameter with a default value. When it has a similar doubt, the compiler stops and asks the programmer to state his or her intentions more clearly.

## Conclusion

Writing procedure and functions is a key element of programming, although in Delphi you'll tend to write methods -- procedures and functions connected with classes and objects.

Instead of moving on to object-oriented features, however, the next few chapters give you some details on other Pascal programming elements, starting with strings.

## Next Chapter: Handling Strings



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 7 Handling Strings

String handling in Delphi is quite simple, but behind the scenes the situation is quite complex. Pascal has a traditional way of handling strings, Windows has its own way, borrowed from the C language, and 32-bit versions of Delphi include a powerful long string data type, which is the default string type in Delphi.

### Types of Strings

In Borland's Turbo Pascal and in 16-bit Delphi, the typical string type is a sequence of characters with a length byte at the beginning, indicating the current size of the string. Because the length is expressed by a single byte, it cannot exceed 255 characters, a very low value that creates many problems for string manipulation. Each string is defined with a fixed size (which by default is the maximum, 255), although you can declare shorter strings to save memory space.

A string type is similar to an array type. In fact, a string is almost an array of characters. This is demonstrated by the fact that you can access a specific string character using the `[]` notation.

To overcome the limits of traditional Pascal strings, the 32-bit versions of Delphi support long strings. There are actually three string types:

- The `ShortString` type corresponds to the typical Pascal strings, as described before. These strings have a limit of 255 characters and correspond to the strings in the 16-bit version of Delphi. Each element of a short string is of type `ANSIChar` (the standard character type).
- The `ANSIString` type corresponds to the new variable-length long strings. These strings are allocated dynamically, are reference counted, and use a copy-on-write technique. The size of these strings is almost unlimited (they can store up to two billion characters!). They are also based on the `ANSIChar` type.
- The `WideString` type is similar to the `ANSIString` type but is based on the `WideChar` type-it stores Unicode characters.

### Using Long Strings

If you simply use the string data type, you get either short strings or ANSI strings, depending on the value of the `$H` compiler directive. `$H+` (the default) stands for long strings (the `ANSIString` type), which is what is used by the components of the Delphi library.

Delphi long strings are based on a reference-counting mechanism, which keeps track of how many string variables are referring to the same string in memory. This reference-counting is used also to free the memory when a string isn't used anymore-that is, when the reference count reaches zero.



If you want to increase the size of a string in memory but there is something else in the adjacent memory, then the string cannot grow in the same memory location, and a full copy of the string must therefore be made in another location. When this situation occurs, Delphi's run-time support reallocates the string for you in a completely transparent way. You simply set the maximum size of the string with the `SetLength` procedure, effectively allocating the required amount of memory:

```
SetLength (String1, 200);
```

The `SetLength` procedure performs a memory request, not an actual memory allocation. It reserves the required memory space for future use, without actually using the memory. This technique is based on a feature of the Windows operating systems and is used by Delphi for all dynamic memory allocations. For example, when you request a very large array, its memory is reserved but not allocated.

Setting the length of a string is seldom necessary. The only case in which you must allocate memory for the long string using `SetLength` is when you have to pass the string as a parameter to an API function (after the proper typecast), as I'll show you shortly.

## Looking at Strings in Memory

To help you better understand the details of memory management for strings, I've written the simple `StrRef` example. In this program I declare two global strings: `Str1` and `Str2`. When the first of the two buttons is pressed, the program assigns a constant string to the first of the two variables and then assigns the second variable to the first:

```
Str1 := 'Hello';
Str2 := Str1;
```

Besides working on the strings, the program shows their internal status in a list box, using the following `StringStatus` function:

```
function StringStatus (const Str: string): string;
begin
  Result := 'Address: ' + IntToStr (Integer (Str)) +
    ', Length: ' + IntToStr (Length (Str)) +
    ', References: ' + IntToStr (PInteger (Integer (Str) - 8)^) +
    ', Value: ' + Str;
end;
```

It is vital in the `StringStatus` function to pass the string parameter as a `const` parameter. Passing this parameter by copying will cause the side effect of having one extra reference to the string while the function is being executed. By contrast, passing the parameter via a reference (`var`) or constant (`const`) parameter doesn't imply a further reference to the string. In this case I've used a `const` parameter, as the function is not supposed to modify the string.

To obtain the memory address of the string (useful to determine its actual identity and to see when two different strings refer to the same memory area), I've simply made a hard-coded typecast from the string type to the `Integer` type. Strings are references-in-practice, they're pointers: Their value holds the actual memory location of the string.

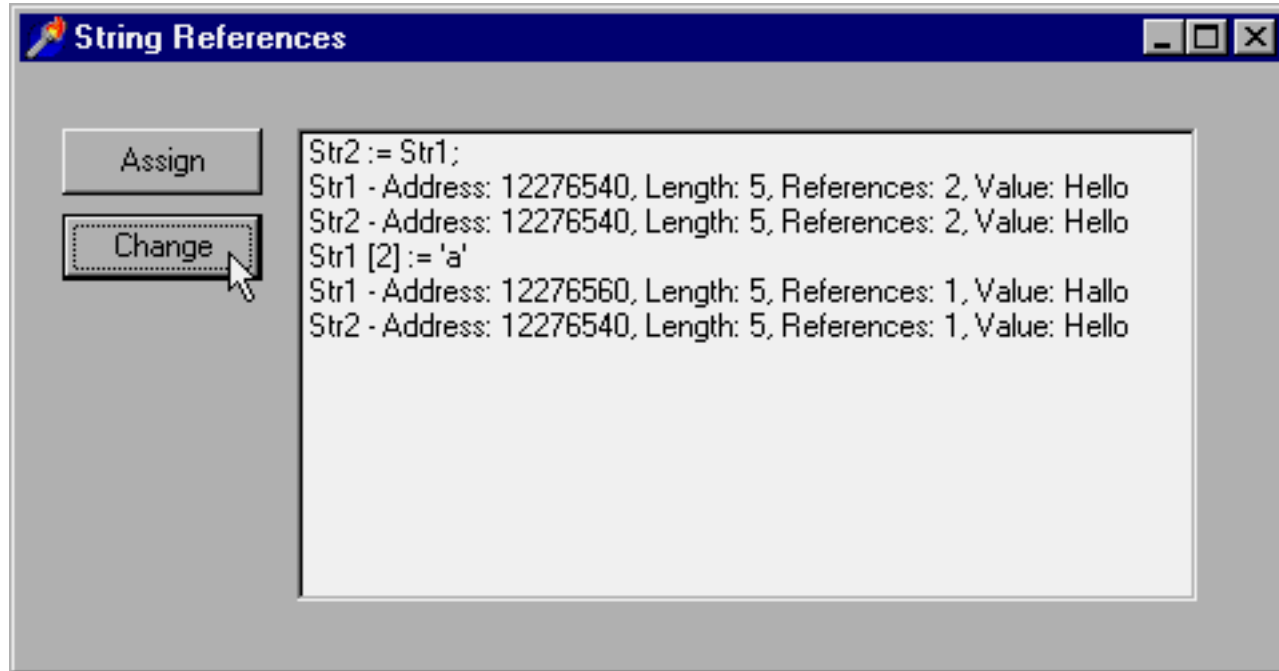
To extract the reference count, I've based the code on the little-known fact that the length and reference count are actually stored in the string, before the actual text and before the position the string variable points to. The (negative) offset is `-4` for the length of the string (a value you can extract more easily using the `Length` function) and `-8` for the reference count.

Keep in mind that this internal information about offsets might change in future versions of Delphi; there is also no guarantee that similar undocumented features will be maintained in the future.

By running this example, you should get two strings with the same content, the same memory location, and a reference count of 2,

as shown in the upper part of the list box of Figure 2.1. Now if you change the value of one of the two strings (it doesn't matter which one), the memory location of the updated string will change. This is the effect of the copy-on-write technique.

**Figure 7.1: The StrRef example shows the internal status of two strings, including the current reference count.**



We can actually produce this effect, shown in the second part of the list box of Figure 7.1, by writing the following code for the OnClick event handler of the second button:

```
procedure TFormStrRef.BtnChangeClick(Sender: TObject);
begin
  Str1 [2] := 'a';
  ListBox1.Items.Add ('Str1 [2] := ''a''');
  ListBox1.Items.Add ('Str1 - ' + StringStatus (Str1));
  ListBox1.Items.Add ('Str2 - ' + StringStatus (Str2));
end;
```

Notice that the code of the BtnChangeClick method can be executed only after the BtnAssignClick method. To enforce this, the program starts with the second button disabled (its Enabled property is set to False); it enables the button at the end of the first method. You can freely extend this example and use the StringStatus function to explore the behavior of long strings in many other circumstances.

## Delphi Strings and Windows PChars

Another important point in favor of using long strings is that they are null-terminated. This means that they are fully compatible with the C language null-terminated strings used by Windows. A null-terminated string is a sequence of characters followed by a byte that is set to zero (or null). This can be expressed in Delphi using a zero-based array of characters, the data type typically used to implement strings in the C language. This is the reason null-terminated character arrays are so common in the Windows API functions (which are based on the C language). Since Pascal's long strings are fully compatible with C null-terminated strings, you can simply use long strings and cast them to PChar when you need to pass a string to a Windows API function.

For example, to copy the caption of a form into a PChar string (using the API function GetWindowText) and then copy it into the Caption of the button, you can write the following code:

```

procedure TForm1.Button1Click (Sender: TObject);
var
    S1: String;
begin
    SetLength (S1, 100);
    GetWindowText (Handle, PChar (S1), Length (S1));
    Button1.Caption := S1;
end;

```

You can find this code in the LongStr example. Note that if you write this code but fail to allocate the memory for the string with SetLength, the program will probably crash. If you are using a PChar to pass a value (and not to receive one as in the code above), the code is even simpler, because there is no need to define a temporary string and initialize it. The following line of code passes the Caption property of a label as a parameter to an API function, simply by typecasting it to PChar:

```
SetWindowText (Handle, PChar (Label1.Caption));
```

When you need to cast a WideString to a Windows-compatible type, you have to use PWideChar instead of PChar for the conversion. Wide strings are often used for OLE and COM programs.

Having presented the nice picture, now I want to focus on the pitfalls. There are some problems that might arise when you convert a long string into a PChar. Essentially, the underlying problem is that after this conversion, you become responsible for the string and its contents, and Delphi won't help you anymore. Consider the following limited change to the first program code fragment above, Button1Click:

```

procedure TForm1.Button2Click(Sender: TObject);
var
    S1: String;
begin
    SetLength (S1, 100);
    GetWindowText (Handle, PChar (S1), Length (S1));
    S1 := S1 + ' is the title'; // this won't work
    Button1.Caption := S1;
end;

```

This program compiles, but when you run it, you are in for a surprise: The Caption of the button will have the original text of the window title, without the text of the constant string you have added to it. The problem is that when Windows writes to the string (within the GetWindowText API call), it doesn't set the length of the long Pascal string properly. Delphi still can use this string for output and can figure out when it ends by looking for the null terminator, but if you append further characters after the null terminator, they will be skipped altogether.

How can we fix this problem? The solution is to tell the system to convert the string returned by the GetWindowText API call back to a Pascal string. However, if you write the following code:

```
S1 := String (S1);
```

the system will ignore it, because converting a data type back into itself is a useless operation. To obtain the proper long Pascal string, you need to recast the string to a PChar and let Delphi convert it back again properly to a string:

```
S1 := String (PChar (S1));
```

Actually, you can skip the string conversion, because PChar-to-string conversions are automatic in Delphi. Here is the final code:

```

procedure TForm1.Button3Click(Sender: TObject);

```

```

var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := String (PChar (S1));
  S1 := S1 + ' is the title';
  Button3.Caption := S1;
end;

```

An alternative is to reset the length of the Delphi string, using the length of the PChar string, by writing:

```
SetLength (S1, StrLen (PChar (S1)));
```

You can find three versions of this code in the LongStr example, which has three buttons to execute them. However, if you just need to access the title of a form, you can simply use the Caption property of the form object itself. There is no need to write all this confusing code, which was intended only to demonstrate the string conversion problems. There are practical cases when you need to call Windows API functions, and then you have to consider this complex situation.

## Formatting Strings

Using the plus (+) operator and some of the conversion functions (such as IntToStr) you can indeed build complex strings out of existing values. However, there is a different approach to formatting numbers, currency values, and other strings into a final string. You can use the powerful Format function or one of its companion functions.

The Format function requires as parameters a string with the basic text and some placeholders (usually marked by the % symbol) and an array of values, one for each placeholder. For example, to format two numbers into a string you can write:

```
Format ('First %d, Second %d', [n1, n2]);
```

where n1 and n2 are two Integer values. The first placeholder is replaced by the first value, the second matches the second, and so on. If the output type of the placeholder (indicated by the letter after the % symbol) doesn't match the type of the corresponding parameter, a runtime error occurs. Having no compile-time type checking is actually the biggest drawback of using the Format function.

The Format function uses an open-array parameter (a parameter that can have an arbitrary number of values), something I'll discuss toward the end of this chapter. For the moment, though, notice only the array-like syntax of the list of values passed as the second parameter.

Besides using %d, you can use one of many other placeholders defined by this function and briefly listed in Table 7.1. These placeholders provide a default output for the given data type. However, you can use further format specifiers to alter the default output. A width specifier, for example, determines a fixed number of characters in the output, while a precision specifier indicates the number of decimal digits. For example,

```
Format ('%8d', [n1]);
```

converts the number n1 into an eight-character string, right-aligning the text (use the minus (-) symbol to specify left-justification) filling it with white spaces.

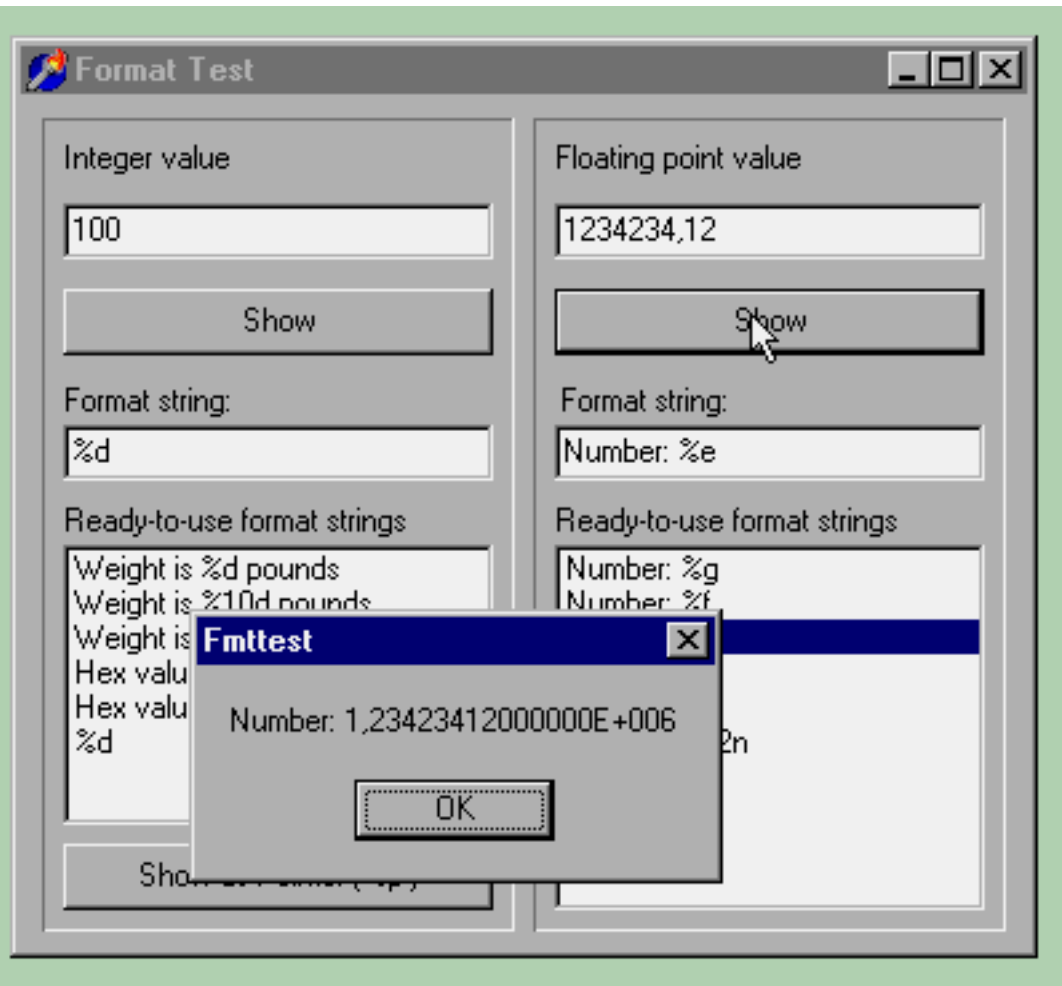
### Table 7.1: Type Specifiers for the Format Function

<b>TYPE SPECIFIER</b>	<b>DESCRIPTION</b>
d (decimal)	The corresponding integer value is converted to a string of decimal digits.
x (hexadecimal)	The corresponding integer value is converted to a string of hexadecimal digits.
p (pointer)	The corresponding pointer value is converted to a string expressed with hexadecimal digits.
s (string)	The corresponding string, character, or PChar value is copied to the output string.
e (exponential)	The corresponding floating-point value is converted to a string based on exponential notation.
f (floating point)	The corresponding floating-point value is converted to a string based on floating point notation.
g (general)	The corresponding floating-point value is converted to the shortest possible decimal string using either floating-point or exponential notation.
n (number)	The corresponding floating-point value is converted to a floating-point string but also uses thousands separators.
m (money)	The corresponding floating-point value is converted to a string representing a currency amount. The conversion is based on regional settings-see the Delphi Help file under Currency and date/time formatting variables.

The best way to see examples of these conversions is to experiment with format strings yourself. To make this easier I've written the `FmtTest` program, which allows a user to provide formatting strings for integer and floating-point numbers. As you can see in Figure 7.2, this program displays a form divided into two parts. The left part is for Integer numbers, the right part for floating-point numbers.

Each part has a first edit box with the numeric value you want to format to a string. Below the first edit box there is a button to perform the formatting operation and show the result in a message box. Then comes another edit box, where you can type a format string. As an alternative you can simply click on one of the lines of the `Listbox` component, below, to select a predefined formatting string. Every time you type a new formatting string, it is added to the corresponding list box (note that by closing the program you lose these new items).

**Figure 7.2: The output of a floating-point value from the `FmtTest` program**



The code of this example simply uses the text of the various controls to produce its output. This is one of the three methods connected with the Show buttons:

```

procedure TFormFmtTest.BtnIntClick(Sender: TObject);
begin
  ShowMessage (Format (EditFmtInt.Text,
    [StrToInt (EditInt.Text)]));
  // if the item is not there, add it
  if ListBoxInt.Items.IndexOf (EditFmtInt.Text) < 0 then
    ListBoxInt.Items.Add (EditFmtInt.Text);
end;

```

The code basically does the formatting operation using the text of the EditFmtInt edit box and the value of the EditInt control. If the format string is not already in the list box, it is then added to it. If the user instead clicks on an item in the list box, the code moves that value to the edit box:

```

procedure TFormFmtTest.ListBoxIntClick(Sender: TObject);
begin
  EditFmtInt.Text := ListBoxInt.Items [
    ListBoxInt.ItemIndex];
end;

```

## Conclusion

Strings are certainly a very common data type. Although you can safely use them in most cases without understanding how they work, this chapter should have made clear the exact behavior of strings, making it possible for you to use all the power of this data type.

Strings are handled in memory in a special dynamic way, as happens with dynamic arrays. This is the topic of the next chapter.

## **Next Chapter: Memory**



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 8 Memory

**Author's Note:** This chapter will cover memory handling, discuss the various memory areas, and introduce dynamic arrays. Temporarily only this last part is available.

### Delphi 4 Dynamic Arrays

Traditionally, the Pascal language has always had fixed-size arrays. When you declare a data type using the array construct, you have to specify the number of elements of the array. As expert programmers probably know, there were a few techniques you could use to implement dynamic arrays, typically using pointers and manually allocating and freeing the required memory.

Delphi 4 introduces a very simple implementation of dynamic arrays, modeling them after the dynamic long string type I've just covered. As long strings, dynamic arrays are dynamically allocated and reference counted, but they do not offer a copy-on-write technique. That's not a big problem, as you can deallocate an array by setting its variable to nil.

You can now simply declare an array without specifying the number of elements and then allocate it with a given size using the `SetLength` procedure. The same procedure can also be used to resize an array without losing its content. There are also other string-oriented procedures, such as the `Copy` function, that you can use on arrays.

Here is a small code excerpt, underscoring the fact that you must both declare and allocate memory for the array before you can start using it:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Array1: array of Integer;
begin
  Array1 [1] := 100; // error
  SetLength (Array1, 100);
  Array1 [99] := 100; // OK
  ...
end;
```

As you indicate only the number of elements of the array, the index invariably starts from 0. Generic arrays in Pascal account for a non-zero low bound and for non-integer indexes, two features that dynamic arrays don't support. To learn the status of a dynamic array, you can use the `Length`, `High`, and `Low` functions, as with any other array. For dynamic arrays, however, `Low` always returns 0, and `High` always returns the length minus one. This implies that for an empty array `High` returns -1 (which, when you think about it,



is a strange value, as it is lower than that returned by Low).

**Figure 8.1: The form of the DynArr example**



After this short introduction I can show you a simple example, called DynArr and shown in Figure 8.1. It is indeed simple because there is nothing very complex about dynamic arrays. I'll also use it to show a few possible errors programmers might make. The program declares two global arrays and initializes the first in the OnCreate handler:

```
var
  Array1, Array2: array of Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // allocate
  SetLength (Array1, 100);
end;
```

This sets all the values to zero. This initialization code makes it possible to start reading and writing values of the array right away, without any fear of memory errors. (Assuming, of course, that you don't try to access items beyond the upper bound of the array.) For an even better initialization, the program has a button that writes into each cell of the array:

```
procedure TForm1.btnFillClick(Sender: TObject);
var
  I: Integer;
begin
  for I := Low (Array1) to High (Array1) do
    Array1 [I] := I;
end;
```

The Grow button allows you to modify the size of the array without losing its contents. You can test this by using the Get value button after pressing the Grow button:

```
procedure TForm1.btnGrowClick(Sender: TObject);
begin
  // grow keeping existing values
  SetLength (Array1, 200);
end;

procedure TForm1.btnGetClick(Sender: TObject);
begin
  // extract
```

```

Caption := IntToStr (Array1 [99]);
end;

```

The only slightly complex code is in the `OnClick` event of the `Alias` button. The program copies one array to the other one with the `:=` operator, effectively creating an alias (a new variable referring to the same array in memory). At this point, however, if you modify one of the arrays, the other is affected as well, as they both refer to the same memory area:

```

procedure TForm1.btnAliasClick(Sender: TObject);
begin
    // alias
    Array2 := Array1;
    // change one (both change)
    Array2 [99] := 1000;
    // show the other
    Caption := IntToStr (Array1 [99]);

```

The `btnAliasClick` method does two more operations. The first is an equality test on the arrays. This tests not the actual elements of the structures but rather the memory areas the arrays refer to, checking whether the variables are two aliases of the same array in memory:

```

procedure TForm1.btnAliasClick(Sender: TObject);
begin
    ...
    if Array1 = Array2 then
        Beep;
    // truncate first array
    Array1 := Copy (Array2, 0, 10);
end;

```

The second is a call to the `Copy` function, which not only moves data from one array to the other, but also replaces the first array with a new one created by the function. The effect is that the `Array1` variable now refers to an array of 11 elements, so that pressing the `Get value` or `Set value` buttons produces a memory error and raises an exception (unless you have range-checking turned off, in which case the error remains but the exception is not displayed). The code of the `Fill` button continues to work fine even after this change, as the items of the array to modify are determined using its current bounds.

## Conclusion

This chapter temporarily covers only dynamic arrays, certainly an important element for memory management, but only a portion of the entire picture. More material will follow.

The memory structure described in this chapter is typical of Windows programming, a topic I'll introduce in the next chapter (without going to the full extent of using the `VCL`, though).

## Next Chapter: Windows Programming



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 9: Windows Programming

Delphi provides a complete encapsulation of the low-level Windows API using Object Pascal and the Visual Component Library (VCL), so it is rarely necessary to build Windows applications using plain Pascal and calling Windows API functions directly. Nonetheless, programmers who want to use some special techniques not supported by the VCL still have that option in Delphi. You would only want to take this approach for very special cases, such as the development of new Delphi components based on unusual API calls, and I don't want to cover the details. Instead, we'll look at a few elements of Delphi's interaction with the operating system and a couple of techniques that Delphi programmers can benefit from.

### Windows Handles

Among the data types introduced by Windows in Delphi, handles represent the most important group. The name of this data type is *THandle*, and the type is defined in the Windows unit as:

```
type
  THandle = LongWord;
```

Handle data types are implemented as numbers, but they are not used as such. In Windows, a handle is a reference to an internal data structure of the system. For example, when you work with a window (or a Delphi form), the system gives you a handle to the window. The system informs you that the window you are working with is window number 142, for example. From that point on, your application can ask the system to operate on window number 142—moving it, resizing it, reducing it to an icon, and so on. Many Windows API functions, in fact, have a handle as the first parameter. This doesn't apply only to functions operating on windows; other Windows API functions have as their first parameter a GDI handle, a menu handle, an instance handle, a bitmap handle, or one of the many other handle types.

In other words, a handle is an internal code you can use to refer to a specific element handled by the system, including a window, a bitmap, an icon, a memory block, a cursor, a font, a menu, and so on. In Delphi, you seldom need to use handles directly, since they are hidden inside forms, bitmaps, and other Delphi objects. They become useful when you want to call a Windows API function that is not supported by Delphi.

To complete this description, here is a simple example demonstrating Windows handles. The WHandle program has a simple form, containing just a button. In the code, I respond to the OnCreate event of the form and the OnClick event of the button, as indicated by the following textual definition of the main form:

```
object FormWHandle: TFormWHandle
  Caption = 'Window Handle'
  OnCreate = FormCreate
```

```

object BtnCallAPI: TButton
  Caption = 'Call API'
  OnClick = BtnCallAPIClick
end
end

```

As soon as the form is created, the program retrieves the handle of the window corresponding to the form, by accessing the `Handle` property of the form itself. We call `IntToStr` to convert the numeric value of the handle into a string, and we append that to the caption of the form, as you can see in Figure 9.1:

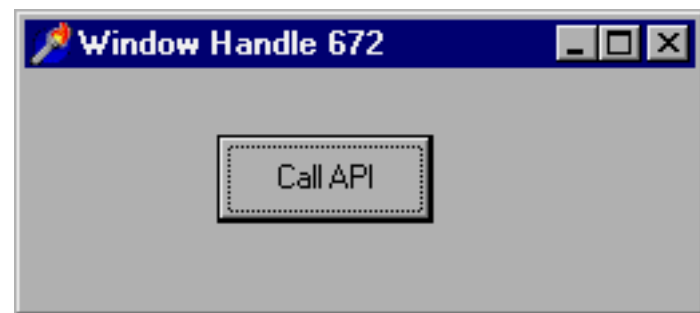
```

procedure TFormWHandle.FormCreate(Sender: TObject);
begin
  Caption := Caption + ' ' + IntToStr (Handle);
end;

```

Because `FormCreate` is a method of the form's class, it can access other properties and methods of the same class directly. Therefore, in this procedure we can simply refer to the `Caption` of the form and its `Handle` property directly.

**Figure 9.1: The WHandle example shows the handle of the form window. Every time you run this program you'll get a different value.**



If you run this program several times you'll generally get different values for the handle. This value, in fact, is determined by Windows and is sent back to the application. (Handles are never determined by the program, and they have no predefined values; they are determined by the system, which generates new values each time you run a program.)

When the user presses the button, the program simply calls a Windows API function, `SetWindowText`, which changes the text or caption of the window passed as the first parameter. To be more precise, the first parameter of this API function is the handle of the window we want to modify:

```

procedure TFormWHandle.BtnCallAPIClick(Sender: TObject);
begin
  SetWindowText (Handle, 'Hi');
end;

```

This code has the same effect as the previous event handler, which changed the text of the window by giving a new value to the `Caption` property of the form. In this case calling an API function makes no sense, because there is a simpler Delphi technique. Some API functions, however, have no correspondence in Delphi, as we'll see in more advanced examples later in the book.

## External Declarations

Another important element for Windows programming is represented by external declarations. Originally used to link the Pascal code to external functions that were written in assembly language, the external declaration is used in Windows programming to call a function from a DLL (a dynamic link library). In Delphi, there are a number of such declarations in the `Windows` unit:

```
// forward declaration
function LineTo (DC: HDC; X, Y: Integer): BOOL; stdcall;

// external declaration (instead of actual code)
function LineTo; external 'gdi32.dll' name 'LineTo';
```

This declaration means that the code of the function `LineTo` is stored in the `GDI32.DLL` dynamic library (one of the most important Windows system libraries) with the same name we are using in our code. Inside an external declaration, in fact, we can specify that our function refer to a function of a DLL that originally had a different name.

You seldom need to write declarations like the one just illustrated, since they are already listed in the Windows unit and many other Delphi system units. The only reason you might need to write this external declaration code is to call functions from a custom DLL, or to call undocumented Windows functions.

Note: In the 16-bit version of Delphi, the external declaration used the name of the library without the extension, and was followed by the name directive (as in the code above) or by an alternative index directive, followed by the ordinal number of the function inside the DLL. The change reflects a system change in the way libraries are accessed: Although Win32 still allows access to DLL functions by number, Microsoft has stated this won't be supported in the future. Notice also that the Windows unit replaces the `WinProcs` and `WinTypes` units of the 16-bit version of Delphi.

## A Windows Callback Function

We've seen in Chapter 6 that Object Pascal supports procedural types. A common use of procedural types is to provide callback functions to a Windows API function.

First of all, what is a callback function? The idea is that some API function performs a given action over a number of internal elements of the system, such as all of the windows of a certain kind. Such a function, also called an enumerated function, requires as a parameter the action to be performed on each of the elements, which is passed as a function or procedure compatible with a given procedural type. Windows uses callback functions in other circumstances, but we'll limit our study to this simple case.

Now consider the `EnumWindows` API function, which has the following prototype (copied from the Win32 Help file):

```
BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc, // address of callback function
    LPARAM lParam // application-defined value
);
```

Of course, this is the C language definition. We can look inside the Windows unit to retrieve the corresponding Pascal language definition:

```
function EnumWindows (
    lpEnumFunc: TFNWndEnumProc;
    lParam: LPARAM): BOOL; stdcall;
```

Consulting the help file, we find that the function passed as a parameter should be of the following type (again in C):

```
BOOL CALLBACK EnumWindowsProc (
    HWND hwnd, // handle of parent window
    LPARAM lParam // application-defined value
);
```

This corresponds to the following Delphi procedural type definition:

```
type
  EnumWindowsProc = function (Hwnd: THandle;
    Param: Pointer): Boolean; stdcall;
```

The first parameter is the handle of each main window in turn, while the second is the value we've passed when calling the EnumWindows function. Actually in Pascal the TFWndEnumProc type is not properly defined; it is simply a pointer. This means we need to provide a function with the proper parameters and then use it as a pointer, taking the address of the function instead of calling it. Unfortunately, this also means that the compiler will provide no help in case of an error in the type of one of the parameters.

Windows requires programmers to follow the stdcall calling convention every time we call a Windows API function or pass a callback function to the system. Delphi, by default, uses a different and more efficient calling convention, indicated by the register keyword.

Here is the definition of a proper compatible function, which reads the title of the window into a string, then adds it to a ListBox of a given form:

```
function GetTitle (Hwnd: THandle; Param: Pointer): Boolean; stdcall;
var
  Text: string;
begin
  SetLength (Text, 100);
  GetWindowText (Hwnd, PChar (Text), 100);
  FormCallBack.ListBox1.Items.Add (
    IntToStr (Hwnd) + ': ' + Text);
  Result := True;
end;
```

The form has a ListBox covering almost its whole area, along with a small panel on the top hosting a button. When the button is pressed, the EnumWindows API function is called, and the GetTitle function is passed as its parameter:

```
procedure TFormCallBack.BtnTitlesClick(Sender: TObject);
var
  EWProc: EnumWindowsProc;
begin
  ListBox1.Items.Clear;
  EWProc := GetTitle;
  EnumWindows (@EWProc, 0);
end;
```

I could have called the function without storing the value in a temporary procedural type variable first, but I wanted to make clear what is going on in this example. The effect of this program is actually quite interesting, as you can see in Figure 9.2. The Callback example shows a list of all the existing main windows running in the system. Most of them are hidden windows you usually never see (and many actually have no caption).

**Figure 9.2: The output of the Callback example, listing the current main windows (visible and hidden).**



## A Minimal Windows Program

To complete the coverage of Windows programming and the Pascal language, I want to show you a very simple but complete application built without using the VCL. The program simply takes the command-line parameter (stored by the system in the `cmdLine` global variable) and then extracts information from it with the `ParamCount` and `ParamStr` Pascal functions. The first of these functions returns the number of parameters; the second returns the parameter in a given position.

Although users seldom specify command-line parameters in a graphical user interface environment, the Windows command-line parameters are important to the system. For example, once you have defined an association between a file extension and an application, you can simply run a program by selecting an associated file. In practice, when you double-click on a file, Windows starts the associated program and passes the selected file as a command-line parameter.

Here is the complete source code of the project (a DPR file, not a PAS file):

```

program Strparam;

uses
  Windows;

begin
  // show the full string
  MessageBox (0, cmdLine,
    'StrParam Command Line', MB_OK);

  // show the first parameter
  if ParamCount > 0 then
    MessageBox (0, PChar (ParamStr (1)),

```

```

    '1st StrParam Parameter', MB_OK)
else
    MessageBox (0, PChar ('No parameters'),
        '1st StrParam Parameter', MB_OK);
end.

```

The output code uses the `MessageBox` API function, simply to avoid getting the entire VCL into the project. A pure Windows program as the one above, in fact, has the advantage of a very small memory footprint: The executable file of the program is about 16 Kbytes.

To provide a command-line parameter to this program, you can use Delphi's `Run > Parameters` menu command. Another technique is to open the Windows Explorer, locate the directory that contains the executable file of the program, and drag the file you want to run onto the executable file. The Windows Explorer will start the program using the name of the dropped file as a command-line parameter. Figure 9.3 shows both the Explorer and the corresponding output.

**Figure 9.3:** You can provide a command-line parameter to the `StrParam` example by dropping a file over the executable file in the Windows Explorer.



## Conclusion

In this chapter we've seen a low-level introduction to Windows programming, discussing handles and a very simple Windows program. For normal Windows programming tasks, you'll generally use the visual development support provided by Delphi and based on the VCL. But this is beyond the scope of this book, which is the Pascal language.

Next chapter covers variants, a very strange addition to Pascal type system, introduced to provide full OLE support.

## Next Chapter: Variants





- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 10 Variants

To provide full OLE support, the 32-bit version of Delphi includes the Variant data type. Here I want to discuss this data type from a general perspective. The Variant type, in fact, has a pervasive effect on the whole language, and the Delphi components library also uses them in ways not related to OLE programming.

### Variants Have No Type

In general, you can use variants to store any data type and perform numerous operations and type conversions. Notice that this goes against the general approach of the Pascal language and against good programming practices. A variant is type-checked and computed at run time. The compiler won't warn you of possible errors in the code, which can be caught only with extensive testing. On the whole, you can consider the code portions that use variants to be interpreted code, because, as with interpreted code, many operations cannot be resolved until run time. This affects in particular the speed of the code.

Now that I've warned you against the use of the Variant type, it is time to look at what it can do. Basically, once you've declared a variant variable such as the following:

```
var
  V: Variant;
```

you can assign to it values of several different types:

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

Once you have the variant value, you can copy it to any compatible-or incompatible-data type. If you assign a value to an incompatible data type, Delphi performs a conversion, if it can. Otherwise it issues a run-time error. In fact, a variant stores type information along with the data, allowing a number of run-time operations; these operations can be handy but are both slow and unsafe.

Consider the following example (called VariTest), which is an extension of the code above. I placed three edit boxes on a new form, added a couple of buttons, and then wrote the following code for the OnClick event of the first button:

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

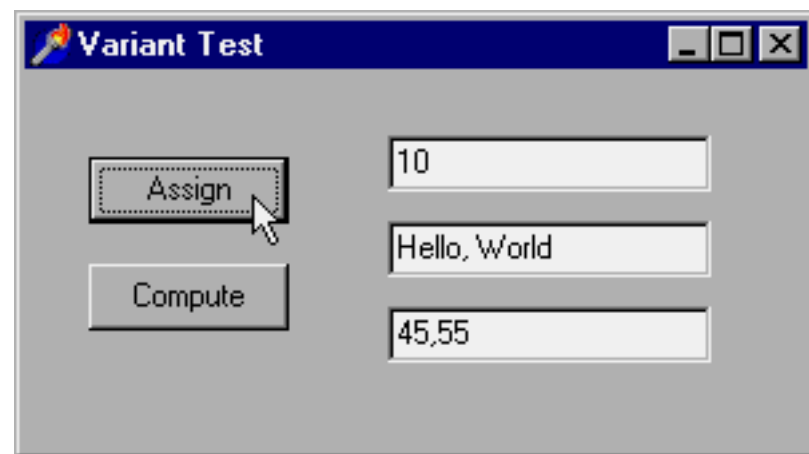
```

V: Variant;
begin
  V := 10;
  Edit1.Text := V;
  V := 'Hello, World';
  Edit2.Text := V;
  V := 45.55;
  Edit3.Text := V;
end;

```

Funny, isn't it? Besides assigning a variant holding a string to the Text property of an edit component, you can assign to the Text a variant holding an integer or a floating-point number. As you can see in Figure 10.1, everything works.

**Figure 10.1:** The output of the VariTest example after the Assign button has been pressed.



Even worse, you can use the variants to compute values, as you can see in the code related to the second button:

```

procedure TForm1.Button2Click(Sender: TObject);
var
  V: Variant;
  N: Integer;
begin
  V := Edit1.Text;
  N := Integer(V) * 2;
  V := N;
  Edit1.Text := V;
end;

```

Writing this kind of code is risky, to say the least. If the first edit box contains a number, everything works. If not, an exception is raised. Again, you can write similar code, but without a compelling reason to do so, you shouldn't use the Variant type; stick with the traditional Pascal data types and type-checking approach. In Delphi and in the VCL (Visual Component Library), variants are basically used for OLE support and for accessing database fields.

## Variants in Depth

Delphi includes a variant record type, `TVarData`, which has the same memory layout as the Variant type. You can use this to access the actual type of a variant. The `TVarData` structure includes the type of the Variant, indicated as `VType`, some reserved fields, and the actual value.

The possible values of the VType field correspond to the data types you can use in OLE automation, which are often called OLE types or variant types. Here is a complete alphabetical list of the available variant types:

- varArray
- varBoolean
- varByRef
- varCurrency
- varDate
- varDispatch
- varDouble
- varEmpty
- varError
- varInteger
- varNull
- varOleStr
- varSingle
- varSmallint
- varString
- varTypeMask
- varUnknown
- varVariant

You can find descriptions of these types in the Values in variants topic in the Delphi Help system.

There are also many functions for operating on variants that you can use to make specific type conversions or to ask for information about the type of a variant (see, for example, the VarType function). Most of these type conversion and assignment functions are actually called automatically when you write expressions using variants. Other variant support routines (look for the topic Variant support routines in the Help file) actually operate on variant arrays.

## Variants Are Slow!

Code that uses the Variant type is slow, not only when you convert data types, but also when you add two variant values holding an integer each. They are almost as slow as the interpreted code of Visual Basic! To compare the speed of an algorithm based on variants with that of the same code based on integers, you can look at the VSpeed example.

This program runs a loop, timing its speed and showing the status in a progress bar. Here is the first of the two very similar loops, based on integers and variants:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    time1, time2: TDateTime;
    n1, n2: Variant;
begin
    time1 := Now;
    n1 := 0;
    n2 := 0;
    ProgressBar1.Position := 0;
    while n1 < 5000000 do
    begin
        n2 := n2 + n1;
        Inc (n1);
        if (n1 mod 50000) = 0 then
        begin
            ProgressBar1.Position := n1 div 50000;
            Application.ProcessMessages;
        end;
    end;

```

```

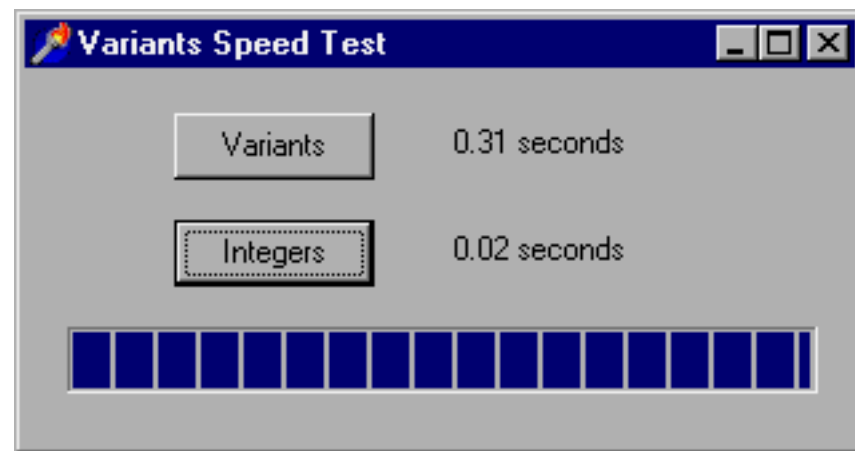
end;
// we must use the result
Total := n2;
time2 := Now;
Label1.Caption := FormatDateTime (
    'n:ss', Time2-Time1) + ' seconds';
end;

```

The timing code is worth looking at, because it's something you can easily adapt to any kind of performance test. As you can see, the program uses the `Now` function to get the current time and the `FormatDateTime` function to output the time difference, asking only for the minutes ("n") and the seconds ("ss") in the format string. As an alternative, you can use the Windows API's `GetTickCount` function, which returns a very precise indication of the milliseconds elapsed since the operating system was started.

In this example the speed difference is actually so great that you'll notice it even without a precise timing. Anyway, you can see the results for my own computer in Figure 10.2. The actual values depend on the computer you use to run this program, but the proportion won't change much.

**Figure 10.2: The different speeds of the same algorithm, based on integers and variants (the actual timing varies depending on the computer), as shown by the VSpeed example.**



## Conclusion

Variants are so different from traditional Pascal data types that I've decided to cover them in this short separate chapter. Although their role is in OLE programming, they can be handy to write quick and dirty programs without having even to think about data types. As we have seen, this affects performance by far.

Now that we have covered most of the language features, let me discuss the overall structure of a program and the modularization offered by units.

## Next Chapter: Program and Units



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Chapter 11 Program and Units

Delphi applications make intensive use of units, or program modules. Units, in fact, were the basis of the modularity in the language before classes were introduced. In a Delphi application, every form has a corresponding unit behind it. When you add a new form to a project (with the corresponding toolbar button or the File > New Form menu command), Delphi actually adds a new unit, which defines the class for the new form.

### Units

Although every form is defined in a unit, the reverse is not true. Units do not need to define forms; they can simply define and make available a collection of routines. By selecting the File > New menu command and then the Unit icon in the New page of the Object Repository, you add a new blank unit to the current project. This blank unit contains the following code, delimiting the sections a unit is divided into:

```
unit Unit1;

interface

implementation

end.
```

The concept of a unit is simple. A unit has a unique name corresponding to its filename, an interface section declaring what is visible to other units, and an implementation section with the real code and other hidden declarations. Finally, the unit can have an optional initialization section with some startup code, to be executed when the program is loaded into memory; it can also have an optional finalization section, to be executed on program termination.

The general structure of a unit, with all its possible sections, is the following:

```
unit unitName;

interface

// other units we need to refer to
uses
    A, B, C;

// exported type definition
type
```

```

newType = TypeDefinition;

// exported constants
const
  Zero = 0;

// global variables
var
  Total: Integer;

// list of exported functions and procedures
procedure MyProc;

implementation

uses
  D, E;

// hidden global variable
var
  PartialTotal: Integer;

// all the exported functions must be coded
procedure MyProc;
begin
  // ... code of procedure MyProc
end;

initialization
  // optional initialization part

finalization
  // optional clean-up code

end.

```

The `uses` clause at the beginning of the interface section indicates which other units we need to access in the interface portion of the unit. This includes the units that define the data types we refer to in the definition of other data types, such as the components used within a form we are defining.

The second `uses` clause, at the beginning of the implementation section, indicates more units we need to access only in the implementation code. When you need to refer to other units from the code of the routines and methods, you should add elements in this second `uses` clause instead of the first one. All the units you refer to must be present in the project directory or in a directory of the search path (you can set the search path for a project in the Directories/Conditionals page of the project's Options dialog box).

C++ programmers should be aware that the `uses` statement does not correspond to an `include` directive. The effect of a `uses` statement is to import just the precompiled interface portion of the units listed. The implementation portion of the unit is considered only when that unit is compiled. The units you refer to can be both in source code format (PAS) or compiled format (DCU), but the compilation must have taken place with the same version of the Delphi.

The interface of a unit can declare a number of different elements, including procedures, functions, global variables, and data types. In Delphi applications, the data types are probably used the most often. Delphi automatically places a new class data type in a unit each time you create a form. However, containing form definitions is certainly not the only use for units in Delphi. You can continue to have traditional units, with functions and procedures, and you can have units with classes that do not refer to forms or other visual elements.

## Units and Scope

In Pascal, units are the key to encapsulation and visibility, and they are probably even more important than the private and public keywords of a class. (In fact, as we'll see in the next chapter, the effect of the private keyword is related to the scope of the unit containing the class.) The scope of an identifier (such as a variable, procedure, function, or a data type) is the portion of the code in which the identifier is accessible. The basic rule is that an identifier is meaningful only within its scope—that is, only within the block in which it is declared. You cannot use an identifier outside its scope. Here are some examples.

- **Local variables:** If you declare a variable within the block defining a routine or a method, you cannot use this variable outside that procedure. The scope of the identifier spans the whole procedure, including nested routines (unless an identifier with the same name in the nested routine hides the outer definition). The memory for this variable is allocated on the stack when the program executes the routine defining it. As soon as the routine terminates, the memory on the stack is automatically released.
- **Global hidden variables:** If you declare an identifier in the implementation portion of a unit, you cannot use it outside the unit, but you can use it in any block and procedure defined within the unit. The memory for this variable is allocated as soon as the program starts and exists until it terminates. You can use the initialization section of the unit to provide a specific initial value.
- **Global variables:** If you declare an identifier in the interface portion of the unit, its scope extends to any other unit that uses the one declaring it. This variable uses memory and has the same lifetime as the previous group; the only difference is in its visibility.

Any declarations in the interface portion of a unit are accessible from any part of the program that includes the unit in its uses clause. Variables of form classes are declared in the same way, so that you can refer to a form (and its public fields, methods, properties, and components) from the code of any other form. Of course, it's poor programming practice to declare everything as global. Besides the obvious memory consumption problems, using global variables makes a program less easy to maintain and update. In short, you should use the smallest possible number of global variables.

## Units as Namespaces

The uses statement is the standard technique to access the scope of another unit. At that point you can access the definitions of the unit. But it might happen that two units you refer to declare the same identifier; that is, you might have two classes or two routines with the same name.

In this case you can simply use the unit name to prefix the name of the type or routine defined in the unit. For example, you can refer to the ComputeTotal procedure defined in the given Totals unit as Totals.ComputeTotal. This should not be required very often, as you are strongly advised against using the same name for two different things in a program.

However, if you look into the VCL library and the Windows files, you'll find that some Delphi functions have the same name as (but generally different parameters than) some Windows API functions available in Delphi itself. An example is the simple Beep procedure.

If you create a new Delphi program, add a button, and write the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Beep;
end;
```

then as soon as you press the button you'll hear a short sound. Now, move to the uses statement of the unit and change the code from this:

```
uses
    Windows, Messages, SysUtils, Classes, ...
```

to this very similar version (simply moving the SysUtils unit before the Windows unit):

#### uses

```
SysUtils, Windows, Messages, Classes, ...
```

If you now try to recompile this code, you'll get a compiler error: "Not enough actual parameters." The problem is that the Windows unit defines another Beep function with two parameters. Stated more generally, what happens in the definitions of the first units you include in the uses statement might be hidden by corresponding definitions of later units. The safe solution is actually quite simple:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    SysUtils.Beep;
end;
```

This code will compile regardless of the order of the units in the uses statements. There are few other name clashes in Delphi, simply because Delphi code is generally hosted by methods of classes. Having two methods with the same name in two different classes doesn't create any problem. The problems arise only with global routines.

## Units and Programs

A Delphi application consists of two kinds of source code files: one or more units and one program file. The units can be considered secondary files, which are referred to by the main part of the application, the program. In theory, this is true. In practice, the program file is usually an automatically generated file with a limited role. It simply needs to start up the program, running the main form. The code of the program file, or Delphi project file (DPR), can be edited either manually or by using the Project Manager and some of the Project Options related to the application object and the forms.

The structure of the program file is usually much simpler than the structure of the units. Here is the source code of a sample program file:

```
program Project1;

uses
    Forms,
    Unit1 in 'Unit1.PAS' {Form1DateForm};

begin
    Application.Initialize;
    Application.CreateForm (TForm1, Form1);
    Application.Run;
end.
```

As you can see, there is simply a uses section and the main code of the application, enclosed by the begin and end keywords. The program's uses statement is particularly important, because it is used to manage the compilation and linking of the application.

## Conclusion

At least for the moment, this chapter on the structure of a Pascal application written in Delphi or with one of the latest versions of Turbo Pascal, is the last of the book. Feel free to email me your comment and requests.

If after this introduction on the Pascal language you want to delve into the object-oriented elements of Object Pascal in Delphi, you can refer to my published book **Mastering Delphi 5** (Sybex, 1999). For more information on this and more advanced books of mine (and of other authors as well) you can refer to my web site, [www.marcocantu.com](http://www.marcocantu.com). The same site hosts updated versions of this



book, and its examples.

**[Back to the Cover Page](#)**



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Appendix A Glossary

This is a short glossary of technical terms used throughout the book. They might also be defined elsewhere in the text, but I've decided to collect them here anyway, to make it easier to find them.

### Heap (Memory)

The term Heap indicates a portion of the memory available to a program, also called dynamic memory area. The heap is the area in which the allocation and deallocation of memory happens in random order. This means that if you allocate three blocks of memory in sequence, they can be destroyed later on in any order. The heap manager takes care of all the details for you, so you simply ask for new memory with `GetMem` or by calling a constructor to create an object, and Delphi will return you a new memory block (optionally reusing memory blocks already discarded).

The heap is one of the three memory areas available to an application. The other two are the global area (this is where global variables live) and the stack. Contrary to the heap, global variables are allocated when the program starts and remain there until it terminates. For the stack see the specific entry in this glossary.

Delphi uses the heap for allocating the memory of each and every object, the text of the strings, for dynamic arrays, and for specific requests of dynamic memory (`GetMem`).

Windows allows an application to have up to 2 GigaBytes of address space, most of which can be used by the heap.

### Stack (Memory)

The term Stack indicates a portion of the memory available to a program, which is dynamic but is allocated and deallocated following specific order. The stack allocation is LIFO, Last In First Out. This means that the last memory object you've allocated will be the first to be deleted. Stack memory is typically used by routines (procedure, function, and method calls). When you call a routine, its parameters and return type are placed on the stack (unless you optimize the call, as Delphi does by default). Also the variables you declare within a routine (using a `var` block before the `begin` statement) are stored on the stack, so that when the routine terminates they'll be automatically removed (before getting back to the calling routine, in LIFO order).

The stack is one of the three memory areas available to an application. The other two are called global memory and heap. See the heap entry in this glossary..

Delphi uses the stack for routine parameters and return values (unless you use the default register calling convention), for local routine variables, for Windows API function calls, and so on.

Windows applications can reserve a large amount of memory for the stack. In Delphi you set this in the linker page of the project options, however, the default generally does it. If you receive a stack full error message this is probably because you have a function recursively calling itself forever, not because the stack space is too limited.

## **New requested terms**

- Dynamic
- Static
- Virtual
- memory leak
- painting
- literal
- array
- API
- class reference
- class method
- parent
- owner
- self



- [www.marcocantu.com](http://www.marcocantu.com)
- [Marco's Delphi Books](#)
- [Essential Pascal - Web Site](#)
- [Essential Pascal - Local Index](#)

## Marco Cantù's Essential Pascal

## Appendix B: Examples

This is a list of the examples which are part of Essential Pascal and available for download:

### Chapter 3

ResStr: resource strings  
Range: ordinal types ranges  
TimeNow: time manipulation

### Chapter 4

GPF: general protection faults with null pointers

### Chapter 5

IfTest: if statements  
Loops: for and while statements

### Chapter 6

OpenArr: open array parameters  
DoubleH: simple procedures  
ProcType: procedural types  
OverDef: overloading and default parameters

### Chapter 7

StrRef: strings reference counting  
LongStr: using long strings  
FmtTest: formatting examples

### Chapter 8

DynArr: dynamic arrays  
WHandle: Windows handles  
Callback: Windows callback functions  
StrParam: command line parameters

## **Chapter 10**

VariTest: simple variant operations  
VariSpeed: the speed of variants

# MARCO CANTÙ'S ESSENTIAL DELPHI

A Friendly Introductory  
Guide to Borland Delphi

<http://www.marcocantu.com/edelphi>



Copyright 1996-2002 Marco Cantù  
Revision 1.03 - April 13, 2002

# INTRODUCTION

**A**fter the successful publishing of the e-book Essential Pascal (available on my web site at the address <http://www.marcocantu.com/epascal>), I decided to follow up with an introduction to Delphi. Again most of the material you'll find here was in the first editions of my “printed” book Mastering Delphi, the best selling Delphi book I have written. Due to space constraints and because many Delphi programmers look for more advanced information, in the latest edition this material was completely omitted. To overcome the absence of this information, I have started putting together this second on-line book, titled *Essential Delphi*.

---

## Copyright

The text and the source code of this book are copyrighted by Marco Cantù. Of course, you can use the programs and adapt them to your own needs with no limitation, only you are not allowed to use them in books, training material, and other copyrighted formats without my permission (or in case you are using limited portions, referring to the original). Feel free to link your site with this one, but please do not duplicate the material (on your web site, on a CD) as it is subject to frequent changes and updates. Passing a copy to a friend, occasionally, is certainly something you can do if you do not modify it in any way.

You can print out this book both for personal use and for non-profit training (user-groups, schools, and universities are free to distribute a printed versions as long as they don't charge more than the printing costs and make it clear that this material is freely available, referring readers to the Essential Delphi web site (<http://www.marcocantu.com/edelphi>) for updates.

---

## Book Structure

The book structure is still under development, as the book evolves. This is the current structure:

**Chapter 1: A Form is a Window:**

**Chapter 2: Highlights of the Delphi Environment:**

**Chapter 3: The Object Repository and the Delphi Wizards:**

**Chapter 4: A Tour of the Basic Components**

**Chapter 5: Creating and Handling Menus [ some figures still missing ]**

**Chapter 6: Multimedia Fun [ all figures missing ]**

**Planned chapters:**

Chapter 7: Exploring Forms

Chapter 8: Delphi Database 101

Chapter 9: Reporting Basics

## Source Code

The source code of all the examples mentioned in the book is available on the book web site. The code has the same Copyright as the book: Feel free to use it at will but don't publish it on other documents or site. Links back to this site are welcome.

---

## Feedback

Please let me know of any errors you find (indicating revision number and page number), but also of topics not clear enough for a beginner. I'll be able to devote time to the project depending also on the feedback I receive. Let me know also which other topics (not covered in Mastering Delphi) you'd like to see here.

For reporting errors please use the books section of my newsgroup, as described on [www.marcocantu.com](http://www.marcocantu.com) or use my mailbox (which gets far too jammed) at [marco@marcocantu.com](mailto:marco@marcocantu.com).

---

## Acknowledgments

I have first started thinking about on-line publishing after Bruce Eckel's experience with *Thinking in Java*. I'm a friend of Bruce and think he really did a great job with that book and few others. After the overwhelming response of the "Essential Pascal" book, I started this new one and plan releasing the two as a printed book introducing Delphi (the only problem being to find a publisher).

---

## About the Author

Marco Cantù lives in Piacenza, Italy. After writing C++ and Object Windows Library books and articles, he delved into Delphi programming. He is the author of the *Mastering Delphi* book series, published by Sybex, as well as the advanced *Delphi Developers Handbook*. He writes articles for many magazines, including *The Delphi Magazine*, speaks at Delphi and Borland conferences around the world, and teaches Delphi classes at basic and advanced levels. More recently, he's specializing in XML technologies, still making most of his programming in Delphi. Of course, you can learn more details about Marco and his work by visiting his web site, [www.marcocantu.com](http://www.marcocantu.com).

---

## Donations

I'll probably set up an account on one of those donation/contribution systems, to let people who have enjoyed the book and learned from it, particularly if programming is their job (and not a hobby) and they do it for profit, contribute to its development. No extra material is offered to those donating to the book fund, only because I want to let anyone (particularly students and people leaving in poor countries) benefit from the availability of this material. Information will be available on the book web site.



## Table of Contents

<b>Marco Cantù's</b>	<b>1</b>
<b>Essential Delphi</b>	<b>1</b>
<b>Introduction</b>	<b>2</b>
Copyright	2
Book Structure	2
Source Code	3
Feedback	3
Acknowledgments	3
About the Author	3
Donations	3
<b>Chapter 1: A Form Is a Window</b>	<b>8</b>
Creating Your First Form	8
Adding a Title	9
Saving the Form	10
Using Components	10
Changing Properties	11
Responding to Events	13
Compiling and Running a Program	16
Changing Properties at Run-Time	18
Adding Code to the Program	18
A Two-Way Tool	20
Looking at the Source Code	20
The Textual Description of the Form	23
The Project File	25
Using Component Templates	26
What's Next	26
<b>Chapter 2: Highlights of the Delphi Environment</b>	<b>27</b>
Different Versions of Delphi	27
Asking for Help	27
Delphi Menus and Commands	28
The File Menu	29
The Edit Menu	30
The Search Menu	32
The View Menu	34
The Project Menu	35
The Run Menu	36
The Component Menu	37
The Database Menu	37
The Tools Menu	37
The Help Menu	38
The Delphi Toolbar	39
The Local Menus	39

Working with the Form Designer	40
The Component Palette	42
The Object Inspector	43
The Alignment Palette	44
Writing Code in the Editor	44
Using Editor Bookmarks	44
Code Insight	45
Code Completion	45
Code Templates	46
Code Parameter	46
Managing Projects	47
The Project Manager	47
Setting Project Options	48
Compiling a Project	48
Exploring a Compiled Program	49
The Integrated Debugger	49
The Object Browser	50
Additional Delphi Tools	50
The Files Produced by the System	50
What's Next	51
<b>Chapter three: The Object Repository and the Delphi Wizards</b>	<b>53</b>
The Object Repository	53
The New Page	54
The Forms Page	56
The Dialogs Page	57
The Data Modules Page	57
The Projects Page	57
Delphi Wizards	58
The Database Form Wizard	58
The Application Wizard	61
The Dialog Wizard	63
Customizing the Object Repository	63
Adding New Application Templates	63
The Empty Project Template	65
Adding New Form Templates to the Object Repository	65
The Object Repository Options	66
Installing new DLL Wizards	67
What's Next	67
<b>Chapter 4: A Tour of the Basic Components</b>	<b>69</b>
Windows Own Components	69
Clicking a Button	70
The Buttons Example	70
Clicking the Mouse Button	74
Adding Colored Text to a Form	74
The LabelCo Example	75
Dragging from One Component to Another	78
The Code for the Dragging Example	80

Accepting Input from the User	81
Handling the Input Focus	81
A Generic OnEnter Event Handler	83
Entering Numbers	84
Sophisticated Input Schemes	87
Creating a Simple Editor	88
The Font Dialog Box	89
Creating a Rich Editor	90
Making Choices	91
Grouping Radio Buttons	92
The Phrases1 Example	93
A List with Many Choices	95
The Form of the Phrases2 Example	96
Working with the List Boxes	98
Removing a Selected String from the Other List Box	98
Allowing Multiple Selections	102
The Third Version of the Phrases Example	102
Using a CheckListBox Component	105
Many Lists, Little Space	106
Choosing a Value in a Range	108
The Scroll Color Example	108
What's Next	110
<b>Chapter 5: Creating and Handling Menus</b>	<b>111</b>
The Structure of the Main Menu	111
Different Roles of Menu Items	112
Building a Menu with the Menu Designer	112
The Standard Structure of a Menu	113
Shortcut Keys and Hotkeys	113
Using the Predefined Menu Templates	114
Responding to Menu Commands	114
The Code Generated by the Menu Designer	115
The Code of the MenuOne Example	117
Modifying the Menu at Run-Time	118
Changing Menu Items at Run-Time	118
Disabling Menu Items and Hiding Pull-Down Menus	119
Using Radio Menu Items	120
Creating Menu Items Dynamically	122
Creating Menus and Menu Items Dynamically	125
Short and Long Menus	126
Graphical Menu Items	127
Customizing the Menu Check Mark	127
Bitmap Menu Items	129
Owner-Draw Menu Items	132
Customizing the System Menu	135
Building a Complete Menu	137
The File Menu	138
Short-Circuit Evaluation	139

The Paragraph Menu	142
The Font Menu	143
The Options Menu	145
Pop-Up Menus	146
An Automatic Local Menu	147
Modifying a Pop-Up Menu When It Is Activated	147
Handling Pop-Up Menus Manually	148
What's Next	150
<b>Chapter 6: Multimedia Fun</b>	<b>151</b>
Windows Default Sounds	151
Every Box Has a Beep	152
From Beeps to Music	154
The Media Player Component	155
Playing Sound Files	156
Running Videos	157
A Video in a Form	157
Working with a CD Drive	159
What's Next	161
<b>Epilogue</b>	<b>162</b>

# CHAPTER 1: A FORM IS A WINDOW

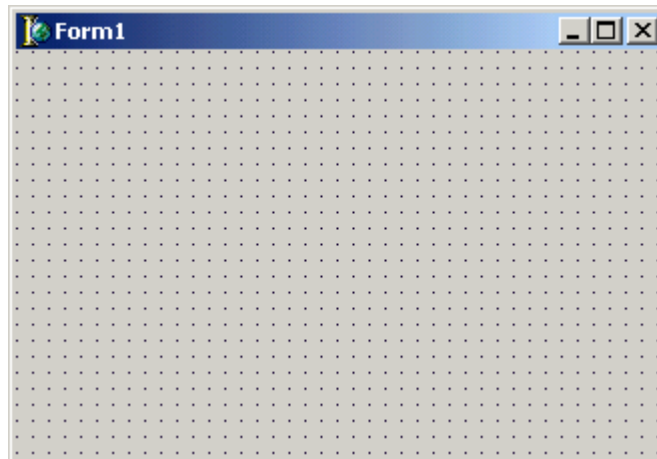
**W**indows applications are usually based on windows. So, how are we going to create our first window? We'll do it by using a form. As the first part of the title suggests, a form really is a window in disguise. There is no real difference between the two concepts, at least from a general point of view.

If you look closely, a form is always a window, but the reverse isn't always true. Some Delphi components are windows, too. A push button is a window. A list box is a window. To avoid confusion, I'll use the term *form* to indicate the main window of an application or a similar window and the term *window* in the broader sense.

---

## Creating Your First Form

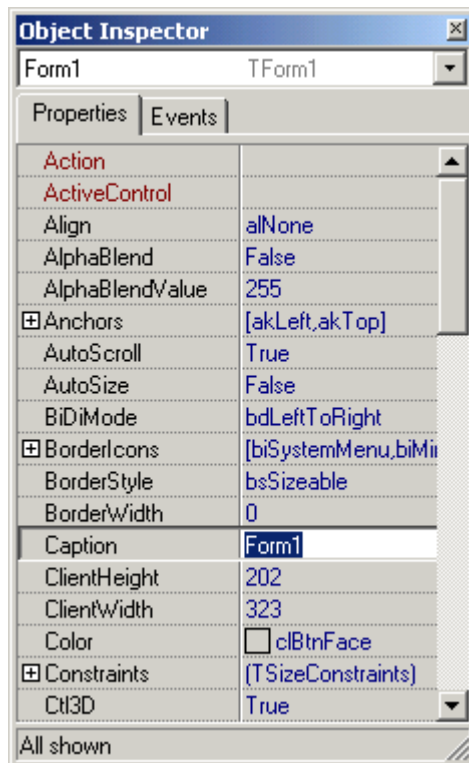
Even though you have probably already created at least some simple applications in Delphi, I'm going to show you the process again, to highlight some interesting points. Creating a form is one of the easiest operations in the system: you only need to open Delphi, and it will automatically create a new, empty form for you, as you can see in the figure below. That's all there is to it.



If you already have another project open, choose File | New | Application to close the old project (you may be prompted to save some of the files) and open a new blank project. Believe it or not, you already have a working application. You can run it, using the Run button on the toolbar or the Run | Run menu command, and it will result in a standard Windows program. Of course, this application won't be very useful, since it has a single empty window with no capabilities, but the default behavior of any Windows window.

## Adding a Title

Before we run the application, let's make a quick change. The title of the form is *Form1*. For a user, the title of the main window stands for the name of the application. Let's change *Form1* to something more meaningful. When you first open Delphi, the Object Inspector window should appear on the left side of the form (if it doesn't, open it by choosing View | Object Inspector or pressing the F11 key):



The Object Inspector shows the properties of the selected component. The window contains a tab control with two pages. The first page is labeled Properties. The other page is labeled Events and shows a list of events that can take place in the form or in the selected component.

The properties are listed in alphabetical order, so it's quite easy to find the ones you want to change (it is also possible to group them by category, as we'll see in the next chapter, but this feature is seldom used by Delphi developers). We can change the title of the form simply by changing the `Caption` property, which is selected by default. While you type a new caption, you can see the title of the form change. If you type *Hello*, the title of the form changes immediately. As an alternative, you can modify the internal name of the form by changing its `Name` property. If you have not entered a new caption, the new value of the `Name` property will be used for the `Caption` property, too.

Only a few of the properties of a component change while you type the new value. Most are applied when you finish the editing operation and press the Enter key (or move the input focus to a new property).

Although we haven't done much work, we have built a full-blown application, with a system menu and the default Minimize, Maximize, and Close buttons. You can resize the form by dragging its borders, move it by dragging its caption, maximize it to full-screen size, or minimize it. It works, but again, it's not very useful. If you look at the icon in the Taskbar, you'll see that something isn't right. Instead of showing the caption of the form as the icon caption, it shows the name of the project, something like *Project1*. We can fix this by giving a name to the project, which we'll do by saving it to disk with a new name.

---

## Saving the Form

Select the Save Project or Save Project As command from the File menu, and Delphi will ask you to give a name to the source code file associated with the form, and then to name the project file. Since the name of the project should match the caption of the form (*Hello*), I've named the form source file HELLOF.PAS, which stands for *Hello Form*. I've given the project file the name HELLO.DPR.

Unfortunately, we cannot use the same name for the project and the unit that defines the form; for each application, these items must have unique names. You can add the letter *F*, add *Form*, call every form unit *MainForm*, or choose any other naming convention you like. I tend to use a name similar to the project name, as simply calling it *Mainform* means you'll end up with a number of forms (in different projects) that all have the same name.

The name you give to the project file is used by default at run-time as the title of the application, displayed by Windows in the taskbar while the program is running. For this reason, if the name of the project matches the caption of the main form, it will also correspond to the name on the taskbar. You can also change the title of the application by using the Application page of the Project Options dialog box (choose Project | Options), or by writing a line of code to change the `Title` property of the Application global object.

---

## Using Components

Now it's time to start placing something useful in our Hello form. Forms can be thought of as component containers. Each form can host a number of components or controls. You can choose a component from the Components Palette above the form, in the Delphi window. There are four simple ways to place a component on a form. If you choose the Button component from the Standard page of the Components Palette, for example, you can do any of the following:

- Click on the component, move the mouse cursor to the form, press the left mouse button to set the upper-left corner of the button, and drag the mouse to set the button's size.
- Select the component as above, and then simply click on the form to place a button of the default height and width.
- Double-click on the icon in the Components Palette, and a component of that type will be added in the center of the form.
- Shift-click on the component icon, and place several components of the same kind in the form using one of the above procedures.

Our form will have only one button, so we'll center it in the form. You can do this by hand, with a little help from Delphi. When you choose View | Alignment Palette, a toolbox with alignment icons appears:



This toolbox makes a number of operations easy. It includes buttons to align controls or to center them in the form. Using the two buttons in the third column, you can place a component in the center of the form. Although we've placed the button in the center, as soon as you run the program, you can resize the form so that the button won't be in the center anymore. So the button is only in the center of the form at startup. Later on, we'll see how to make the button remain in the center after the form is resized, by adding some code. For now, our first priority is to change the button's label.

---

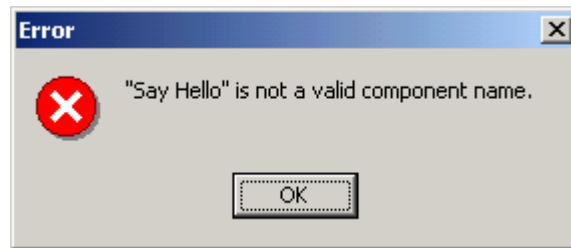
## Changing Properties

Like the form, the button has a `Caption` property that we can use to change its label (the text displayed inside it). As a better alternative, we can change the name of the button. The name is a kind of internal property, used only in the code of the program. However, as I mentioned earlier, if you change the name of a button before changing its caption, the `Caption` property will have the same text as the `Name` property. Changing the `Name` property is usually a good choice, and you should generally do this early in the development cycle, before you write much code.

It is quite common to define a naming convention for each type of component (usually the full name or a shorter version, such as "btn" for Button). If you use a different prefix for each type of component (as in "*ButtonHello*" or "*BtnHello*"), the combo box above the Object Inspector will list the components of the same kind in a group, because they are alphabetically sorted. If you instead use a suffix, naming the components "*HelloButton*" or "*HelloBtn*," components of the same kind will be in different positions on the list. In this second case, however, finding a particular component using the keyboard might be faster. In fact, when the Object Inspector is selected you can type a letter to jump to the first component whose name starts with that letter.

Besides setting a proper name for a component, you often need to change its `Caption` property. There are at least two reasons to have a caption different from the name. The first is that the name often follows a naming convention (as described in the note above) that you won't want to use in a caption. The second reason is that captions should be descriptive, and therefore they often use two or more words, as in my *Say hello* button. If you try to use this text as the `Name` property, however, Delphi will show an error message:

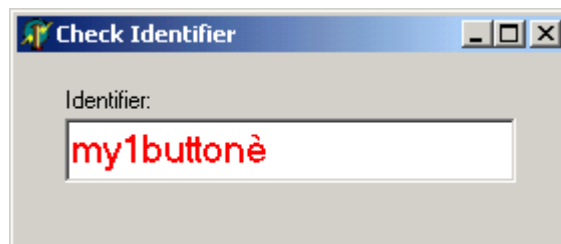




The name is an internal property, and it is used as the name of a variable referring to the component. Therefore, for the Name property, you must follow the rules for naming an identifier in the Pascal language:

- An identifier is a sequence of letters, digits, or underscore characters of any length (although only the first 63 characters are significant).
- The first character of an identifier cannot be a number; it must be a letter or the underscore character.
- No spaces are allowed in an identifier.
- Identifiers are not case-sensitive, but usually each word in an identifier begins with a capital letter, as in BtnHello. But btnhello, btnHello, and BTNHello refer to this same identifier.

You can use the `IsValidIdent` system function to check whether a given string is a valid identifier. The `CheckId` example calls this function while you type an identifier in its edit box, and changes the text color to indicate whether the string is valid (green) or not (red). The code of the example is quite simple, and you can look at it yourself on the disk. Try running this program to check any doubts about allowed component names.



Here is a summary of the changes we have made to the properties of the button and form. At times, I'll show you the structure of the form of the examples as it appears once it has been converted in a readable format (I'll describe how to convert a form into text later in this chapter). I won't show you the entire textual description of a form (which is often quite long), but rather only its key elements. I won't include the lines describing the position of the components, their sizes, or some less important default values. Here is the code:

```
object Form1: TForm1
  Caption = 'Hello'
  OnClick = FormClick
object BtnHello: TButton
  Caption = 'Say hello'
  OnClick = BtnHelloClick
end
```

end

This description shows some attributes of the components and the events they respond to. We will see the code for these events in the following sections. If you run this program now, you will see that the button works properly. In fact, if you click on it, it will be pushed, and when you release the mouse button, the on-screen button will be released. The only problem is that when you press the button, you might expect something to happen; but nothing does, because we haven't assigned any action to the mouse-click yet.

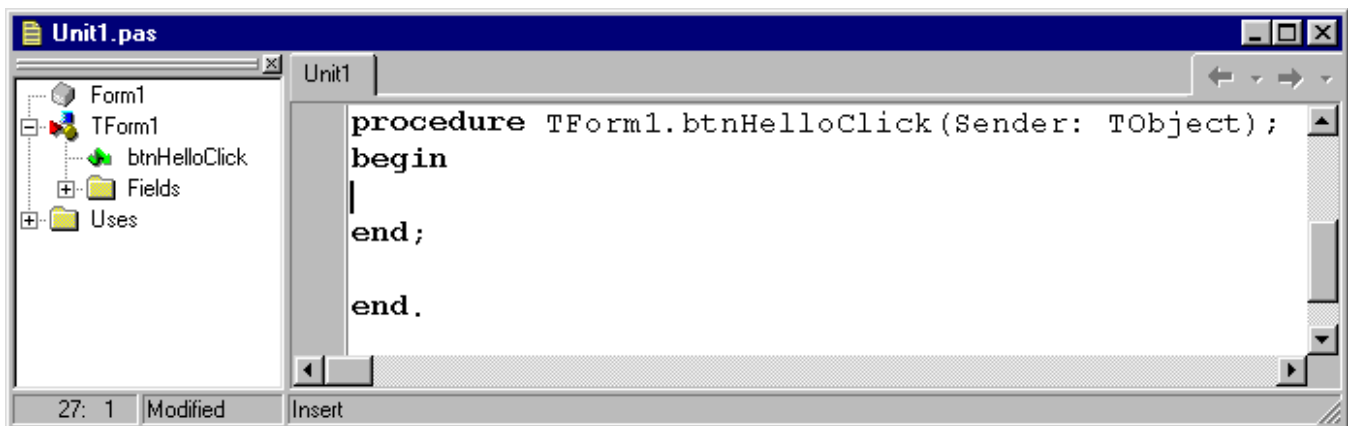
## Responding to Events

When you press the mouse button on a form or a component, Windows informs your application of the event by sending it a message. Delphi responds by receiving an event notification and calling the appropriate event-handler method. As a programmer, you can provide several of these methods, both for the form itself and for the components you have placed in it. Delphi defines a number of events for each kind of component. The list of events for a form is different from the list for a button, as you can easily see by clicking on these two components while the Events page is selected in the Object Inspector. Some events are common to both components.

There are several techniques you can use to define a handler for the `OnClick` event of the button:

- Select the button, either in the form or by using the Object Inspector's combo box (called the Object Selector), select the Events page, and double-click in the white area on the right side of the `OnClick` event. A new method name will appear, *BtnHelloClick*.
- Select the button, select the Events page, and enter the name of a new method in the white area on the right side of the `OnClick` event. Then press the Enter key to accept it.
- Double-click on the button, and Delphi will perform the default action for this component, which is to add a handler for the `OnClick` event. Other components have completely different default actions.

With any of these approaches, Delphi creates a procedure named `BtnHelloClick` (or the name you've provided) in the code of the form and opens the source code file in that position:



The default action for a button is to add a procedure to respond to the click event. Even if you are not sure of the effect of the default action of a component, you can still double-click on it. If you end up adding a new procedure

you don't need, just leave it empty. Empty method bodies generated by Delphi will be removed as soon as you save the file. In other words, if you don't put any code in them, they simply go away.

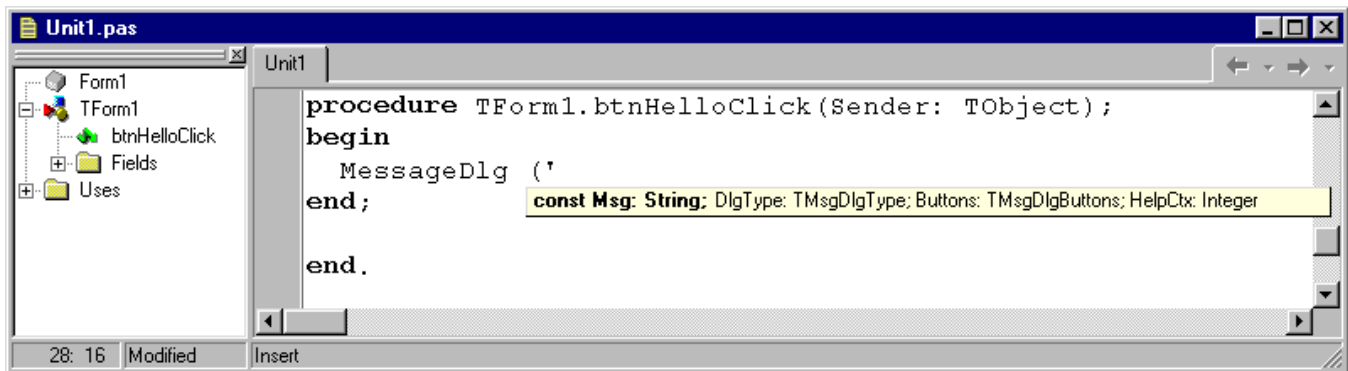
When you want to remove an event-response method you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event-handler that is still empty, consider adding a comment to it, so that it will not be removed.

Now we can start typing some instructions between the `begin` and `end` keywords that delimit the code of the procedure. Writing code is usually so simple that you don't need to be an expert in the language to start working with Delphi. (If you need to brush up your knowledge of Pascal you can refer to my online Essential Pascal book, while if you need derailed coverage of Object Pascal you can refer to my Mastering Delphi series.)

You should type only the line in the middle, but I've included the whole source code of the procedure to let you know where you need to add the new code in the editor:

```
procedure TForm1.BtnHelloClick(Sender: TObject);
begin
    MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
end;
```

The code is simple. There is only a call to a function, `MessageDlg`, to display a small message dialog box. The function has four parameters. Notice that as you type the open parenthesis, the Delphi editor will show you the list of parameters in a hint window, making it simpler to remember them.



If you need more information about the parameters of this function and their meanings, you can click on its name in the edit window and press F1. This brings up the Help information. Since this is the first code we are writing, here is a summary of that description (the rest of this book, however, generally does *not* duplicate the reference information available in Delphi's Help system, concentrating instead on examples that demonstrate the features of the language and environment):

- The first parameter of the `MessageDlg` function is the string you want to display: the message.

- The second parameter is the type of message box. You can choose `mtWarning`, `mtError`, `mtInformation`, or `mtConfirmation`. For each type of message, the corresponding caption is used and a proper icon is displayed at the side of the text.
- The third parameter is a set of values indicating the buttons you want to use. You can choose `mbYes`, `mbNo`, `mbOK`, `mbCancel`, or `mbHelp`. Since this is a set of values, you can have more than one of these values. Always use the proper set notation with square brackets (`[` and `]`) to denote the set, even if you have only one value, as in the line of the code above. (Essential Pascal discusses Pascal sets.)
- The fourth parameter is the help context, a number indicating which page of the Help system should be invoked if the user presses F1. Simply write 0 if the application has no help file, as in this case.

The function also has a return value, which I've just ignored, using it as if it were a procedure. In any case, it's important to know that the function returns an identifier of the button that the user clicked to close the message box. This is useful only if the message box has more than one button.

Programmers unfamiliar with the Pascal language, particularly those who use C/C++, might be confused by the distinction between a function and a procedure. In Pascal, there are two different keywords to define procedures and functions. The only difference between the two is that functions have a return value.

After you have written this line of code, you should be able to run the program. When you click on the button, you'll see the message box shown below.



Every time the user clicks on the push button in the form, a message is displayed. What if the mouse is pressed outside that area? Nothing happens. Of course, we can add some new code to handle this event. We only need to add an `OnClick` event to the form itself. To do this, move to the Events page of the Object Inspector and select the form. Then double-click at the right side of the `OnClick` event, and you'll end up in the proper position in the edit window. Now add a new call to the `MessageDlg` function, as in the following code:

```
procedure TForm1.FormClick(Sender: TObject);  
begin  
    MessageDlg ('You have clicked outside of the button',  
                mtWarning, [mbOK], 0);  
end;
```

With this new version of the program, if the user clicks on the button, the hello message is displayed, but if the user misses the button, a warning message appears. Notice that I've written the code on two lines, instead of one. The Pascal compiler completely ignores new lines, white spaces, tab spaces, and similar formatting characters. Program statements are separated by semicolons (`;`), not by new lines.

There is one case in which Delphi doesn't completely ignore line breaks: Strings cannot extend across multiple lines. In some cases, you can split a very long string into two different strings, written on two lines, and merge them by writing one after the other.

---

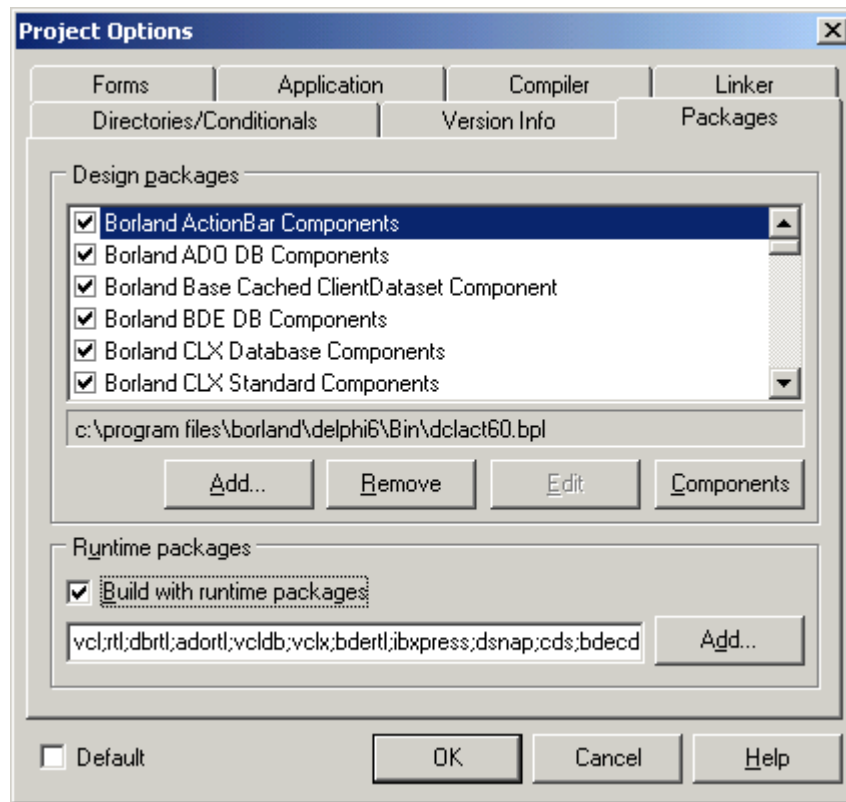
## Compiling and Running a Program

Before we make any further changes to our Hello program, let's stop for a moment to consider what happens when you run the application. When you click on the toolbar Run button or select Run | Run, Delphi does the following:

- 1: Compiles the Pascal source code file describing the form.
- 2: Compiles the project file.
- 3: Builds the executable (EXE) file, linking the proper libraries.
- 4: Runs the executable file, usually in debug mode.

In early versions of Delphi, the executable file you obtained was invariably a stand-alone program. Starting with version 3, Delphi allows you to link all the required libraries into the executable file, but you can also specify the use of separate run-time packages, making the executable file much smaller.

The key point is that when you ask Delphi to run your application, it compiles it into an executable file. You can easily run this file from the Windows Explorer or using the Run command on the Start button. Compiling this program as usual, linking all the required library code, produces an executable of about a couple of hundred Kb. By using run-time packages, this can shrink the executable to about 20 Kb. Simply select the Project | Options menu command, move to the Packages page, and select the check box *Build with runtime packages*:



*Packages* are dynamic link libraries containing Delphi components (the Visual Components Library). By using packages you can make an executable file much smaller. However, the program won't run unless the proper dynamic link libraries (such as `vc160.bpl`) are available on the computer where you want to run the program. The BPL extension stands for Borland Package Libraries; it is the extension used by Delphi (and C++Builder) packages, which are technically DLL files. Using this extension makes it easier to recognize them (and find them on a hard disk).

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the program built with run-time packages is much bigger than the space required by the bigger stand-alone executable file. For this reason the use of packages is not always recommended. The great advantage of Delphi over competing development tools is that you can easily choose whether to use the stand-alone executable or the small executable with run-time packages.

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable with that of a C or C++ program. Delphi compiled code runs much faster (at least 10 times faster) than the equivalent code in interpreted or *semi-compiled* tools.

Some users cannot believe that Delphi generates real executable code, because when you run a small program, its main window appears almost immediately, as happens in some interpreted environments. To see for yourself, try this: Open the Environment Options dialog box (using Tools | Options), move to the Preferences

page, and turn on the Show Compile Progress option. Now select Project | Build All. You'll see a dialog box with the compilation status. You'll find that this takes just a few seconds, or even less on a fast machine.

In the tradition of Borland's Turbo Pascal compilers, the Object Pascal compiler embedded in Delphi works very quickly. For a number of technical reasons, it is much faster than any C++ compiler. If you try using the new Borland C++ Builder development environment (which is very similar to Delphi) the compilation requires more time, particularly the first time you build an application. One reason for the higher speed of the Delphi compiler is that the language definition is simpler. Another is that the Pascal compilers and linkers have less work to do to include libraries or other compiled source files in a program, because of the structure of units.

---

## Changing Properties at Run-Time

Let's return to the Hello application. We now want to try to change some properties at run-time. For example, we might change the text of `HelloButton` from *Say hello* to *Say hello again* after the first time a user clicks on it. You may also need to widen the button, as the caption becomes longer. This is really simple. You only need to change the code of the `HelloButtonClick` procedure as follows:

```
procedure TForm1>HelloButtonClick(Sender: TObject);
begin
    MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
    btnHello.Caption := 'Say Hello Again';
end;
```

The Pascal language uses the `:=` operator to express an assignment and the `=` operator to test for equality. At the beginning, this can be confusing for programmers coming from other languages. For example in C and C++, the assignment operator is `=`, and the equality test is `==`. After a while, you'll get used to it. In the meantime, if you happen to use `=` instead of `:=`, you'll get an error message from the compiler.

A property such as `Caption` can be changed at run-time very easily, by using an assignment statement. Most properties can be changed at run-time, and some can be changed *only* at run-time. You can easily spot this last group: They are not listed in the Object Inspector, but they appear in the Help file for the component. Some of these run-time properties are defined as read-only, which means that you can access their value but cannot change it.

---

## Adding Code to the Program

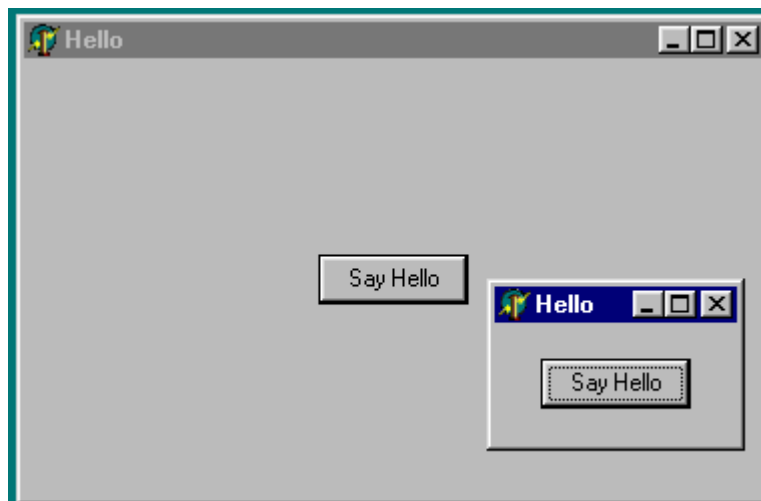
Our program is almost finished, but we still have a problem to solve, which will require some real coding. The button starts in the center of the form, but will not remain there when you resize the form. This problem can be solved in two radically different ways.

One solution is to change the border of the form to a thin frame, so that the form cannot be resized at run-time. Just move to the `BorderStyle` property of the form, and choose `bsSingle` instead of `bsSizeable` from the combo box. The other approach is to write some code to move the button to the center of the form each

time the form is resized, and that's what we'll do next. Although it might seem that most of your work in programming with Delphi is just a matter of selecting options and visual elements, there comes a time when you need to write code. As you become more expert, the percentage of the time spent writing code will generally increase.

When you want to add some code to a program, the first question you need to ask yourself is Where? In an event-driven environment, the code is always executed in response to an event. When a form is resized, an event takes place: `OnResize`. Select the form in the Object Inspector and double-click next to `OnResize` in the Events page. A new procedure is added to the source file of the form. Now you need to type some code in the editor, as follows:

```
procedure TForm1.FormResize(Sender: TObject);  
begin  
  BtnHello.Top := Form1.ClientHeight div 2 -  
    BtnHello.Height div 2;  
  BtnHello.Left := Form1.ClientWidth div 2 -  
    BtnHello.Width div 2;  
end;
```



To set the `Top` and `Left` properties of the button — that is, the position of its upper-left corner — the program computes the center of the frame, dividing the height and the width of the internal area or client area of the frame by 2, and then subtracts half the height or width of the button. Note also that if you use the `Height` and `Width` properties of the form, instead of the `ClientWidth` and `ClientHeight` properties, you will refer to the center of the whole window, including the caption at the top border. This final version of the example works quite well as you can see below.

This figure includes two versions of the form, with different sizes. By the way, this figure is a real snapshot of the screen. Once you have created a Windows application, you can run several copies of it at the same time by using the Explorer. By contrast, the Delphi environment can run only one copy of a program. When you run a program within Delphi, you start the integrated debugger, and it cannot debug two programs at the same time — not even two copies of the same program — unless you are using Windows NT/2000/XP.



---

## A Two-Way Tool

In the Hello example, we have written three small portions of code, to respond to three different events. Each portion of code was part of a different procedure (actually a method, as you'll learn reading Chapter 5). But where does the code we write end up? The source code of a form is written in a single Pascal language source file, the one we've named `HELLOF.PAS`. This file evolves and grows not only when you code the response of some events, but also as you add components to the form. The properties of these components are stored together with the properties of the form in a second file, named `HELLOF.DFM`.

Delphi can be defined as a two-way tool, since everything you do in the visual environment ends up in some code. Nothing is hidden away and inaccessible. You have the complete code, and although some of it might be fairly complex, you can edit everything. Of course, it is easier to use only the visual tools, at least until you are an expert Delphi programmer.

The term *two-way tool* also means that you are free to change the code that has been produced, and then go back to the visual tools. This is true as long as you follow some simple rules.

---

## Looking at the Source Code

Let's take a look at what Delphi has generated from our operations so far. Every action has an effect — in the Pascal code, in the code of the form, or in both. When you start a new, blank project, the empty form has some code associated with it, as in the following listing.

```
unit Unit1;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

The file, named `Unit1`, uses a number of units and defines a new data type (a class) and a new variable (an object of that class). The class is named `TForm1`, and it is derived from `TForm`. The object is `Form1`, of the new type `TForm1`.

*Units* are the modules into which a Pascal program is divided. When you start a new project, Delphi generates a program module and a unit that defines the main form. Each time you add a form to a Delphi program, you add a new unit. Units are then compiled separately and linked into the main program. By default, unit files have a .PAS extension and program files have a .DPR extension.

If you rename the files as suggested in the example, the code changes slightly, since the name of the unit must reflect the name of the file. If you name the file `Hellof.pas`, the code begins with

```
unit Hellof;
```

As soon as you start adding new components, the form class declaration in the source code changes. For example, when you add a button to the form, the portion of the source code defining the new data type becomes the following:

```
type
  TForm1 = class (TForm)
    Button1: TButton;
    ...
```

Now if you change the button's Name property (using the Object Inspector) to `BtnHello`, the code changes slightly again:

```
type
  TForm1 = class (TForm)
    BtnHello: TButton;
    ...
```

Setting properties other than the name has no effect in the source code. The properties of the form and its components are stored in a separate form description file (with a DFM extension).

Adding new event handlers has the biggest impact on the code. Each time you define a new handler for an event, a line is added to the data type definition of the form, an empty method body is added in the implementation part, and some information is stored in the form description file, too.

```
unit HelloForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class (TForm)
    btnHello: TButton;
    procedure btnHelloClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.btnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  btnHello.Caption := 'Say Hello Again';
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;

procedure TForm1.FormResize(Sender: TObject);
begin
  BtnHello.Top := Form1.ClientHeight div 2 -
    BtnHello.Height div 2;
  BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;

end.
```

It is worth noting that there is a single file for the whole code of the form, not just small fragments. Of course, the code is only a partial description of the form. The source code determines how the form and its components react to events. The form description (the DFM file) stores the values of the properties of the form and of its components. In general, source code defines the actions of the system, and form files define the initial state of the system.

---

## The Textual Description of the Form

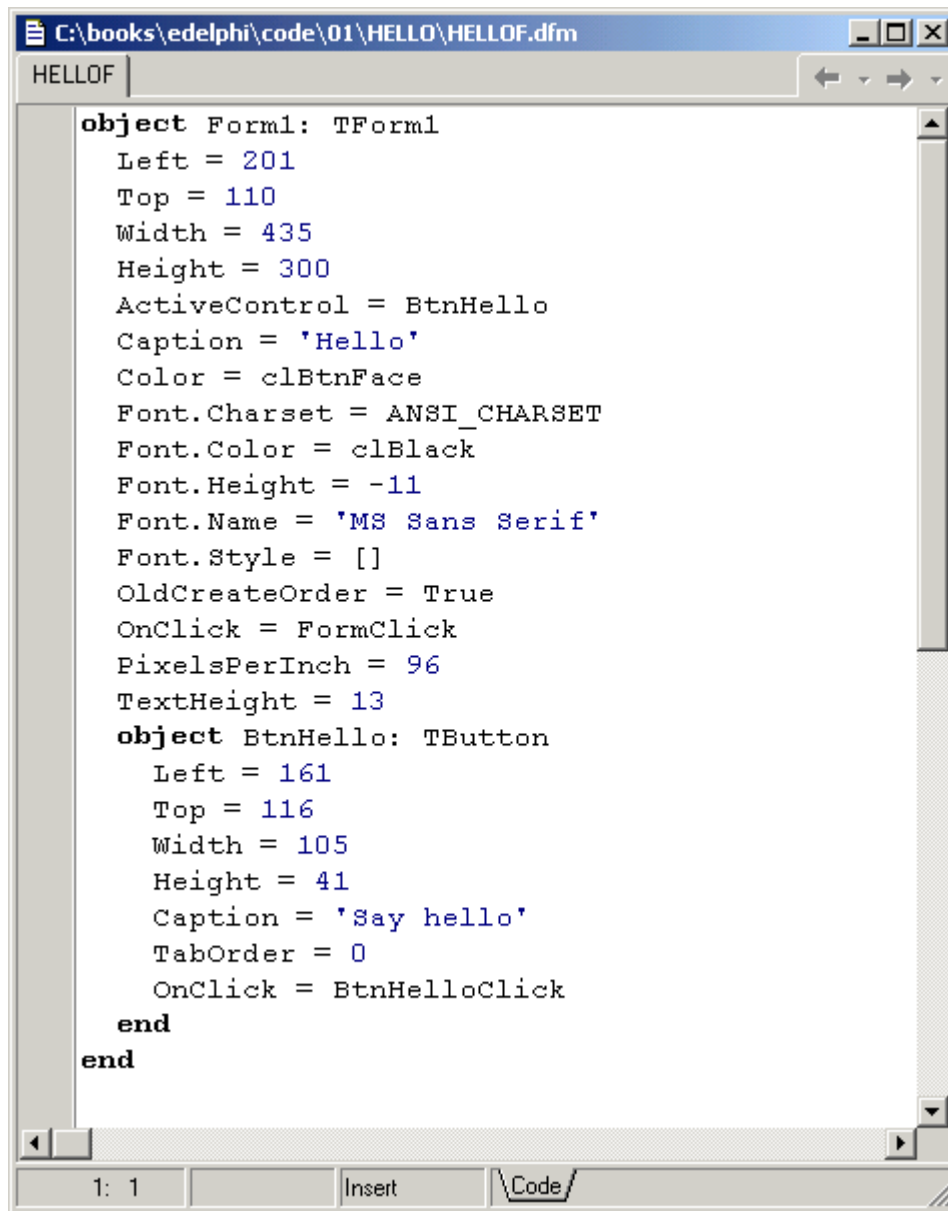
As I've just mentioned, along with the PAS file containing the source code, there is another file describing the form, its properties, its components, and the properties of the components. This is the DFM file, a binary or text file (this latter option has been introduced with Delphi 5). Whatever the format, if you load this file in the Delphi code editor, it will be converted into a textual description. This might give the false impression that the DFM file is indeed a text file, but this can be only if you've selected the corresponding option (available since Delphi 5).

You can open the textual description of a form simply by selecting the shortcut menu of the form designer (that is, right-clicking on the surface of the form at design-time) and selecting the View as Text command. This closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View as Form command of the local menu of the editor window. The alternative is to open the DFM file directly in the Delphi editor.

To understand what is stored in the DFM file, you can look at the next listing, which shows the textual description of the form of the first version of the Hello example. This is exactly the code you'll see if you give the View as Text command in the local menu of the form:

```
object Form1: TForm1
  Left = 235
  Top = 108
  Width = 430
  Height = 308
  Caption = 'Hello'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  OnCreate = FormCreate
  OnResize = FormResize
  PixelsPerInch = 96
  TextHeight = 13
  object btnHello: TButton
    Left = 165
    Top = 111
    Width = 75
    Height = 25
    Caption = 'Say Hello'
    TabOrder = 0
    OnClick = btnHelloClick
  end
end
```

You can compare this code with what I used before to indicate the key features and properties of the form and its components. As you can see in this listing, the textual description of a form contains a number of objects (in this case, two) at different levels. The `Form1` object contains the `BtnHello` object, as you can immediately see from the indentation of the text. Each object has a number of properties, and some methods connected to events (in this case, `OnClick`).

A screenshot of a Delphi IDE window titled 'HELLOF.dfm'. The window displays the textual description of a form and a button. The code is as follows:

```
object Form1: TForm1
  Left = 201
  Top = 110
  Width = 435
  Height = 300
  ActiveControl = BtnHello
  Caption = 'Hello'
  Color = clBtnFace
  Font.Charset = ANSI_CHARSET
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = True
  OnClick = FormClick
  PixelsPerInch = 96
  TextHeight = 13
  object BtnHello: TButton
    Left = 161
    Top = 116
    Width = 105
    Height = 41
    Caption = 'Say hello'
    TabOrder = 0
    OnClick = BtnHelloClick
  end
end
end
```

The window has a standard Windows-style title bar and a toolbar at the bottom with buttons for '1: 1', 'Insert', and '\Code/'. The text is displayed in a monospaced font with a light background and a vertical scrollbar on the right.

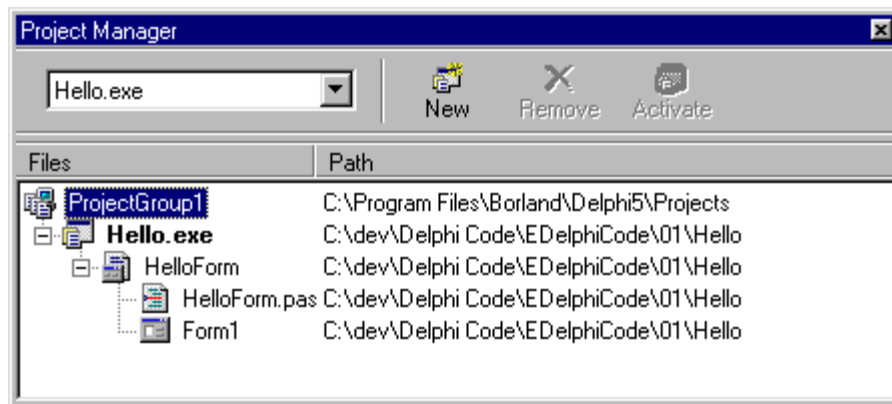
Once you've opened this file in Delphi, you can *edit* the textual description of the form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, this compilation will stop with an error message, and you'll need to correct the contents of your DFM file before you can reopen the form in the editor. For this reason, you shouldn't try to change the textual description of a form manually until you have a good knowledge of Delphi programming.

An expert programmer might choose to work on the text of a form for a number of reasons. For big projects, the textual description of the form is a powerful documenting tool, an important form of backup (in case someone plays with the form, you can understand what has gone wrong by comparing the two textual versions), and a good target for a version-control tool. For these reasons, Delphi also provides a DOS command-line tool, `CONVERT.EXE`, which can translate forms from the compiled version to the textual description and vice versa.

As we will see in the next chapter, the conversion is also applied when you cut or copy components from a form to the Clipboard.

## The Project File

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR). This file is built automatically, and you seldom need to change it, particularly for small programs. If you do need to change the behavior of a project, there are basically two ways to do so: You can use the Delphi Project Manager and set some project options, or you can manually edit the project file directly.



This project file is really a Pascal language source file, describing the overall structure of the program and its startup code:

```
program Hello;

uses
  Forms,
  Helloof in 'HelloForm.PAS' {Form1};

{$R *.RES}

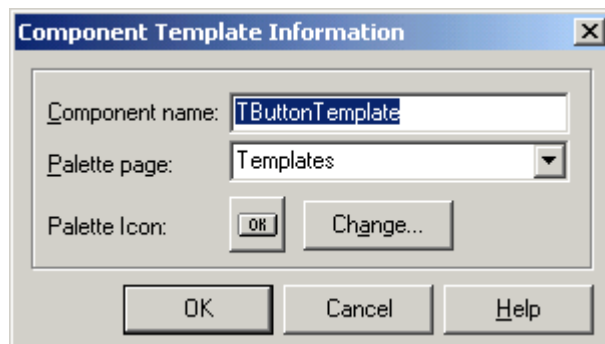
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

You can see this file with the View | Project Source menu command. As an alternative, you can click on the Select unit button of the toolbar or issue the equivalent menu command, View | Units. When you use one of these commands, Delphi shows a dialog box with the list of the source files of the project. You can choose the project file (named Hello in the example), or any other file you are interested in seeing.

## Using Component Templates

Suppose you want to create a brand new application, with a similar button and a similar event handler to the Hello program. It is possible to copy a component to the Clipboard, and then paste it into another form to create a perfect clone. However, doing so you copy only the properties of the component, and not the events associated with it.

Delphi allows you to copy one or more components, and install them as a new component template. This way, you also copy the code of the methods connected with the events of the component. Simply open the Hello example, or any other one, select the component you want to move to the template (or a group of components), and then select the Component | Create Component Template menu command. This opens the Component Template Information dialog box, shown below. Here you enter the name of the template, the page of the Component Palette where it should appear, and an icon.



---

## What's Next

In this chapter, we created a simple program, added a button to it, and handled some basic events, such as a click with the mouse or a resize operation. We also saw how to name files, projects, forms, and components, and how this affects the source code of the program. We looked at the source code of the simple programs we've built, although some of you might not be fluent enough in Object Pascal to understand the details.

Before we can look into more complex examples, we need to explore the Delphi development environment. This is the topics of the next chapter. The examples in this chapter should have shown you that Delphi is really easy to use. Now we'll start to look at the complex mechanisms behind the scenes that make this all possible. You'll see that Delphi is a very powerful tool, even though you can use it to write programs easily and quickly.

## CHAPTER 2: HIGHLIGHTS OF THE DELPHI ENVIRONMENT

In a visual programming tool such as Delphi, the role of the environment is certainly important, at times even more important than the programming language used by its compiler or interpreter. This is a good reason to spend some time reading this chapter.

This chapter won't discuss all of the features of Delphi or list all of its menu commands. Instead, it will give you the overall picture and help you to explore some of the environment traits that are not obvious, while suggesting some tips that may help you. You'll find more information about specific commands and operations throughout the book.

---

### Different Versions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single version of Delphi; there are three of them:

- The basic version (the "Standard" edition) is aimed at Delphi newcomers and casual programmers. It has all the features required to write programs in Delphi for Windows but (starting with Delphi 5) it has no support for any type of database programming.
- The second level (the "Professional" edition) is aimed at professional developers. It includes database support (with BDE and ODBC connectivity), limited Web support, and many more components.
- The full-blown Delphi (the "Enterprise" edition starting with Delphi 5, or the "Client/Server Suite" edition in previous versions) is aimed at developers building client/server applications. It includes also drivers for native Client/Server connection, full ADO support, MIDAS and Internet Express support, and (in Delphi) all of the new Snap technologies: BizSnap, WebSnap, and DataSnap.

Besides the different editions available, there are a number of ways to customize the Delphi environment. You can change the buttons of the toolbar, attach new commands to the Tools menu, hide some of the windows or elements, and resize and move all of them. In the screen illustrations throughout the book, I'll try to use a standard user interface (as it comes out of the box); however, I have my preferences, and I generally install many add-ons (written by third parties or by me), which might be reflected in some of the screen shots.

---

### Asking for Help

Now we can really start our tour. The first element of the environment we'll explore is the Help system. There are basically two ways to invoke the Help system: select the proper command in the Help pull-down menu, or choose an element of the Delphi interface or a token in the source code and press F1.



When you press F1, Delphi doesn't search for an exact match in the Help Search list. Instead, it tries to understand what you are asking. For example, if you press F1 when the text cursor is on the name of the `TButton1` component in the source code, the Delphi Help system automatically opens the description of the `TButton` class, since this is what you are probably looking for. This technique also works when you give the component a new name. Try naming the button `Foo`, then move the cursor to this word, press F1, and you'll still get the help for the `TButton` class. This means Delphi looks at the contextual meaning of the word for which you are asking help.

Note that there isn't just a single help file in Delphi. Most of the time, you'll invoke Delphi Help, but this file is complemented by an Object Pascal Help file, the Windows API Help, the Component Writer's Help, and many others (depending on your version of Delphi). These and other Help files have a common outline and a common search engine you can activate by pressing the Help Topics button while in the Help system. The Windows help engine dialog box that appears allows you to browse the contents of all of the help files in the group, search for a keyword in the index, or start the Find engine. The three capabilities are available in separate pages of the Help Topics dialog box.

You can find almost everything in the Help system, but you need to know what to search for. Usually this is obvious, but at times it is not. Spending some time just playing with the Help system will probably help you understand the structure of these files and learn how to find the information you need.

The Help files provide a lot of information, both for beginner and expert programmers, and they are especially valuable as a reference tool. They list all of the methods and properties for each component, the parameters of each method or function, and similar details, which are particularly important while you are writing code. Borland also distributes reference materials in the form of Adobe Acrobat files. These are electronic versions of the printed manuals that come in the Delphi box, so you can search them for a word, and you can also print the portions you are interested in (or even the whole file if you've got some spare paper).

The first version of Delphi included some Interactive Tutors in addition to the Help system. If you've never used Delphi (and if you have Delphi 1 installed), you might consider running these Tutors. They will guide you through Delphi's basic features and help you understand some of the terminology of the environment. Unluckily they were soon discontinued by Borland.

Besides the Delphi Help files, there are many sources of collections of tips and suggestions for Delphi programmers. Borland web sites provides some FAQs (Frequently Asked Questions) and a collection of Technical Information short papers (TI). You can find updates of both at the Borland Community Web site (<http://community.borland.com>). Besides these official Borland documents, you'll find many more tips in Borland newsgroups (and also on mine). Obviously the Web is a great source of information about Delphi itself and third party products. You can find a collection of my favorite Delphi Web pages in the Links portion of my web site (<http://www.marcocantu.com/links>).

---

## Delphi Menus and Commands

There are basically three ways to issue a command in the Delphi environment:

- Use the menu.

- Use the toolbar.
- Use one of the local menus activated by pressing the right mouse button.

The Delphi menus offer many commands. I won't bore you with a detailed description of the menu structure. For this type of information, you can refer to the printed documentation or the Help file. In the following sections, I'll present some suggestions on the use of some of the menu commands. Other suggestions will follow in the rest of the chapter.

## The File Menu

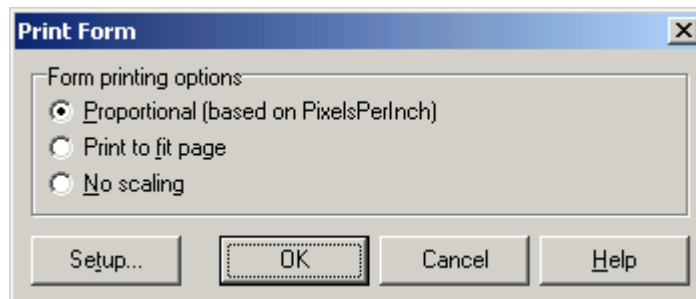
Our starting point is the File pull-down menu. The structure of this menu has kept changing from version to version of Delphi, with menu items for handling projects moving away, and specific commands to create new designer (for example data modules) coming and going. Still, this menu contains commands that operate on projects and commands that operate on source code files.

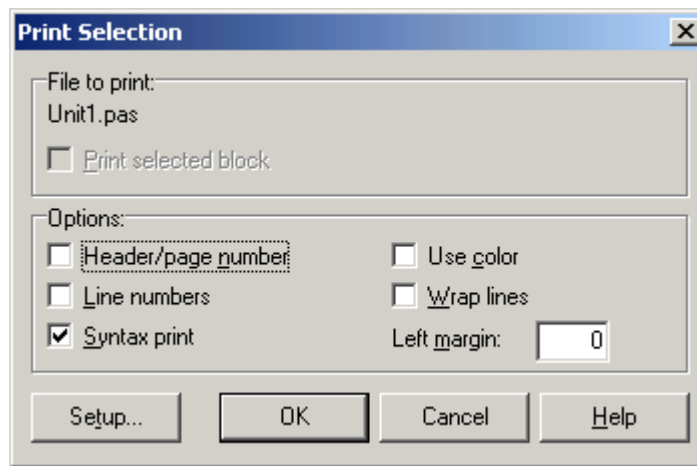
*[\*\*\* The File menu structure has changed in Delphi 6, with the File | New submenu, and the text here has not been updated accordingly]* Some commands can even be used to operate both on projects and on source code files. The commands related to projects are New, New Application, Open, Save Project As, Save All, Close All, Add to Project, and Remove from Project. Besides these, there is also a specific Project pull-down menu. The commands related to source code files are New, New Form, New Data Module, Open, Reopen, Save, Save As, Close, and Print. Most of these commands are very intuitive, but some require a little explanation.

Use the Reopen menu command to open projects or source code files you have worked on recently.

The New command actually opens the New Items dialog box, also called the Object Repository. This dialog box can be used to invoke Delphi Wizards and to create items such as new applications, forms that inherit from existing forms, threads, DLLs, Delphi components, and ActiveX controls. I'll cover the Object Repository's rich set of features in the next chapter.

Another peculiar command is Print. If you are editing source code and select this command, the printer will output the text with syntax highlighting as an option. If you are working on a form and select Print from the File menu, the printer will produce the graphical representation of the form. This is certainly nice, but it can be confusing, particularly if you are working on other Delphi windows. Fortunately, two different print options dialog boxes are displayed, so that you can check that the operation is correct.





## The Edit Menu

The Edit menu has some typical operations, such as Undo and Redo, and the Cut, Copy, and Paste commands, plus some specific commands for form or editor windows. The important thing to notice is that the standard features of the Edit menu (and the standard Ctrl+Z, Ctrl+X, Ctrl+C, and Ctrl+V keyboard shortcuts) work both with text and with form components. There are also some differences worth noting. For example, when you work with the editor, the first command of this pull-down menu is Undo; when you work with the form, it becomes Undelete. Unfortunately, the Form Designer has very limited Undo capabilities.

Of course, you can copy and paste some text in the editor, and you can also copy and paste components in one form, or from one form to another. You can even paste components to a different parent window of the same form, such as a panel or group box.

Besides using cut and paste commands, the Delphi editor allows you to move source code by selecting and dragging words, expressions, or lines. If you drag text while pressing the Ctrl key, it will be copied instead of moved.

## Copying and Pasting Components

What you might not have noticed is that you can also copy components from the form to the editor and vice versa. Delphi places components in the Clipboard along with their textual description. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component.

For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 56
  Top = 48
```

```

Width = 161
Height = 57
TabOrder = 0
Caption = 'Button1'
end

```

Now, if you change the name of the object, caption, or position, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```

object MyButton: TButton
  Left = 200
  Top = 200
  Width = 180
  Height = 60
  TabOrder = 0
  Caption = 'My Button'
  Font.Name = 'Arial'
end

```

Copying the above description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font. To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and change the position of the component so that it doesn't overlap a previous copy. You can see how Delphi modifies the properties of the component by copying it back to the editor. For example, this is what you get if you paste the text above in the form, and then copy it again into the editor:

```

object MyButton: TButton
  Left = 112
  Top = 128
  Width = 180
  Height = 60
  Caption = 'My Button'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 0
end

```

As you can see, some lines have been added automatically, to specify other properties of the font. Of course, if you write something completely wrong, such as this code:

```

object Button3: TButton
  Left = 100
  eight = 60
end

```

which has a spelling error (a missing 'H'), and try to paste it into a form, Delphi will show an error indicating what has gone wrong. You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

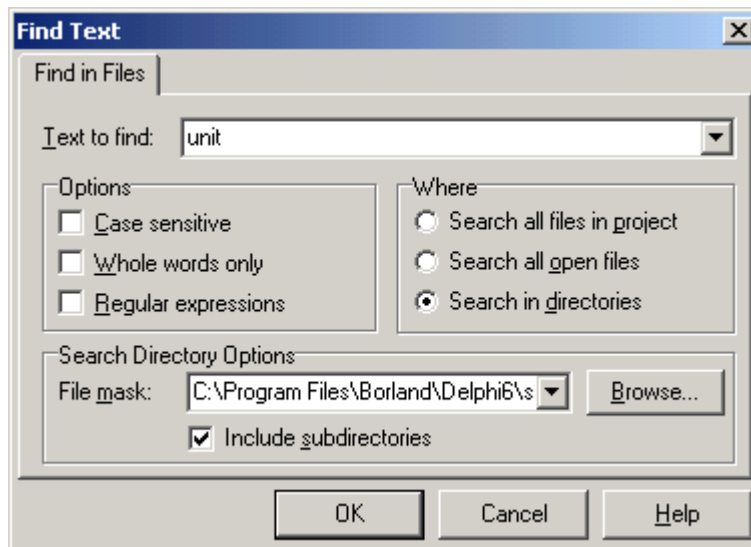
## More Edit Commands

Along with the typical commands found on most Edit menus in Windows applications, Delphi includes a number of commands that are mostly related to forms. The specific operations for forms can also be accessed through the form shortcut menu (the local menu you can invoke with the right mouse button) and will be covered later in the chapter.

One command not replicated in a form's local menu is Lock Controls, which is very useful for avoiding an accidental change to the position of a component in a form. For example, you might try to double-click on a component and actually end up moving it. Since there is no Undo operation on forms, protecting from similar errors by locking the controls after the form has been designed can be really useful.

## The Search Menu

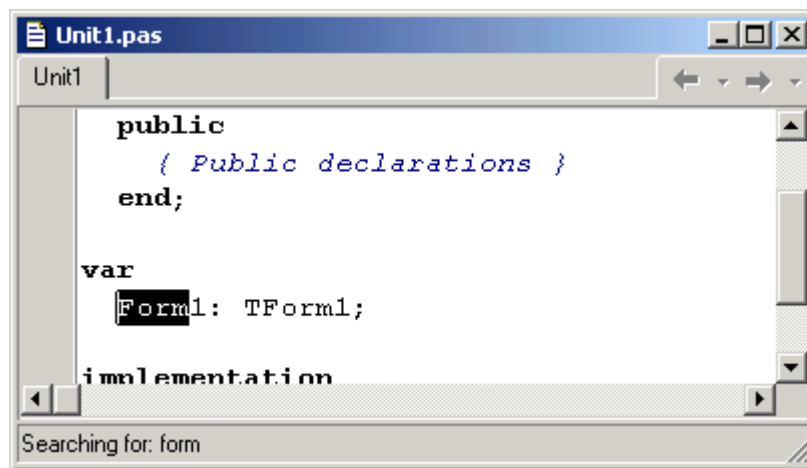
The Search menu has some standard commands, too, such as Search and Replace, and the Find in Files command (you can see its dialog box here):



The Find in Files command allows you to search for a string in all of the source code files of a project, all the open files, or all the files in a directory (optionally including its subdirectories), depending on the radio button you check. The result of the search will be displayed in the message area at the bottom of the editor window. You can select an entry to open the corresponding file and jump to the line containing the text.

You can use the Find in Files command to search for component, class, and type definitions in the VCL source code (if your version of Delphi includes it). This is an easy way to get detailed information on a component, although using Help is generally faster and simpler.

Other commands are not so simple to understand. The Incremental Search command is one of them. When you select this command, instead of showing a dialog box where you enter the text you want to find, Delphi moves to the editor. There, you can type the text you want to search for directly in the editor message area, as you can see here:



When you type the first letter, the editor will move to the first word starting with that letter. (But if your search text isn't found, the letters you typed won't even be displayed in the editor message area.) If that is not the word you are looking for, just keep typing; the cursor will continue to jump as you add letters. Although this command might look strange at first, it is very effective and extremely fast, particularly if you are typing and invoke it with a shortcut key (Ctrl+E if you are using the standard editor shortcuts).

The Go to Line Number command is quite intuitive. The Find Error command might seem strange at first. It is used to find a particular run-time error, not to search for a compiler error. When you are running a stand-alone program and you hit a very bad error, Delphi displays an internal address number (that is, the *logical* address of the compiled code). You can enter this value in the Find Error dialog box to have Delphi recompile the program, looking for the specific address. When it finds the address, Delphi shows the corresponding source code line. Often, however, the error is not in one of the lines of your code, but in a line of library or system code; in this (quite frequent) case the Find Error command cannot locate the offending line.

The last command on the Search menu, Browse Symbol, invokes the Object Browser, a tool you can use to explore all the symbols defined in a compiled program. To understand the output of the Object Browser, you need a good understanding of the Object Pascal language and of the Visual Component Library (VCL).

## The View Menu

The View pull-down menu combines the features you usually find in View and Window menus. There is no Window menu, because the Delphi environment is not an MDI application. Most of the View commands can

be used to display one of the windows of the Delphi environment, such as Project Manager, the Breakpoints list, or the Components command. Some of these windows are used during debugging; others when you are writing code. Most of these windows will be described later in this chapter.

It is possible to add a new item to the View menu, CPU Window, which can be used during debugging to view the Assembler code generated by the Delphi compiler, execute it step by step, and view the status of the CPU registers.

The commands on the second part of the View menu are important, which is why they are also available on the default toolbar. The Toggle Form/Unit (or F12) command is used to move between the form you are working on and its source code. If you use a source code window big enough to hold a reasonable amount of text, you'll use this command often. As an alternative, you can place the two windows (the editor and the form) so that a portion of the one below is always visible. With this arrangement, you can click on it with the mouse to move it to the front.

The New Edit Window command opens a second edit window. It is the only way to view two files side by side in Delphi, since the editor uses tabs to show the multiple files you can load. Once you have duplicated the edit window, you can make each one hold a different set of files, or view two portions of the same file.

The last two commands on the View menu can be used to hide the toolbar or the Components palette, although this is a good way to make Delphi look silly and uncomfortable. Working on forms without the Components palette is certainly not easy. If you remove both the toolbar and the Components palette, the Delphi main window is reduced to a bare menu.

---

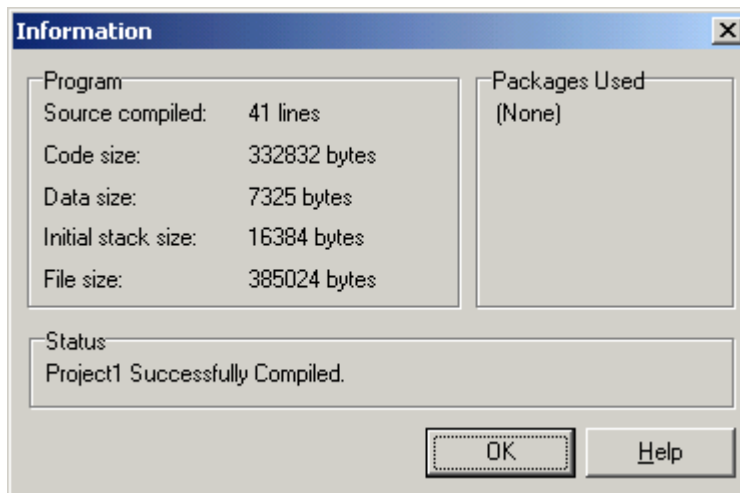
## The Project Menu

The next pull-down menu, Project, has commands to manage a project and compile it. Add to Project and Remove from Project are used to add forms or Pascal source code files to a program and to remove them from a project.

The Enterprise version of Delphi includes two more commands, Web Deploy Options and Web Deploy, which are not available in the other editions. These features are related with ActiveX and ActiveForms, some (now obsolete) Microsoft web technologies.

The Compile command builds or updates the application executable file, checking which source files have changed and recompiling them when needed. With Build All, you can ask Delphi to compile every source file of the project, even if it has not been changed since the last compilation. If you just want to know whether the syntax of the code you've written is correct, but you do not want to build the program, you can use the Syntax Check command.

The next Project command, Information, displays some details about the last compilation you've made. The following figure shows the information related to the compilation of the a short program:



The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU (Delphi Compiled Unit).

At the end of the Project menu comes the Options menu, used to set compiler and linker options, application object options, and so on. When you change the project options, you can check the Default box to indicate that the same set of options should be used for new projects. We will discuss project options again in this chapter and then throughout the book in the context of related topics.

---

## The Run Menu

The Run menu could have been named Debug as well. Most of its commands are related to debugging, including the Run command itself. When you run a program within the Delphi environment, you execute it under the integrated debugger (unless you disable the corresponding Environment option). The Run command and the corresponding toolbar icon are among the most commonly used commands, since Delphi automatically recompiles a program before running it — at least if the source code has changed. Simply hit F9 as a shortcut to compile and run a program.

The next command, Parameters, can be used to specify parameters to be passed on the command line to the program you are going to run, and to provide the name of an executable file when you want to debug a DLL (DLL debugging is another new Delphi 3 feature). The remaining commands are all used during debugging, to execute the program step by step, set breakpoints, inspect the values of variables and objects, and so on. Some of these debugging commands are also available directly in the editor local menu.



Delphi 3 has a couple of new commands related to ActiveX development (and not found in the “Standard” edition). The Register ActiveX Server and Unregister ActiveX Server menu commands basically add or remove the Windows Registry information about the ActiveX control defined by the current project.

---

## The Component Menu

The commands of the Component menu can be used to write components, add them to a package, or to install packages in Delphi. The New Component command invokes the simple Component Wizard. The three installation commands, Install Component, Import ActiveX Library, and Install Packages, can be used to add to the environment new Delphi components, packages, or ActiveX controls. Executing any of these commands adds the new components to the specified package and to the Components palette.

## Component Templates

We briefly used the Create Component Template menu item in the last chapter. When you issue this command after selecting one or more components in a form, it opens a dialog box where you specify the name of the new component template, a page on the Palette, and an icon. By default, the template name is the name of the first component you’ve selected followed by the word *template*. The default template icon is the icon of the first component you’ve selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Components palette (when Delphi displays the fly-by hint).

All the information about component templates is stored in a single file, DELPHI32.DCT, but there is apparently no way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand new form, edit it, and install it again as a component template *using the same name*. This way you can override the previous definition.

---

## The Database Menu

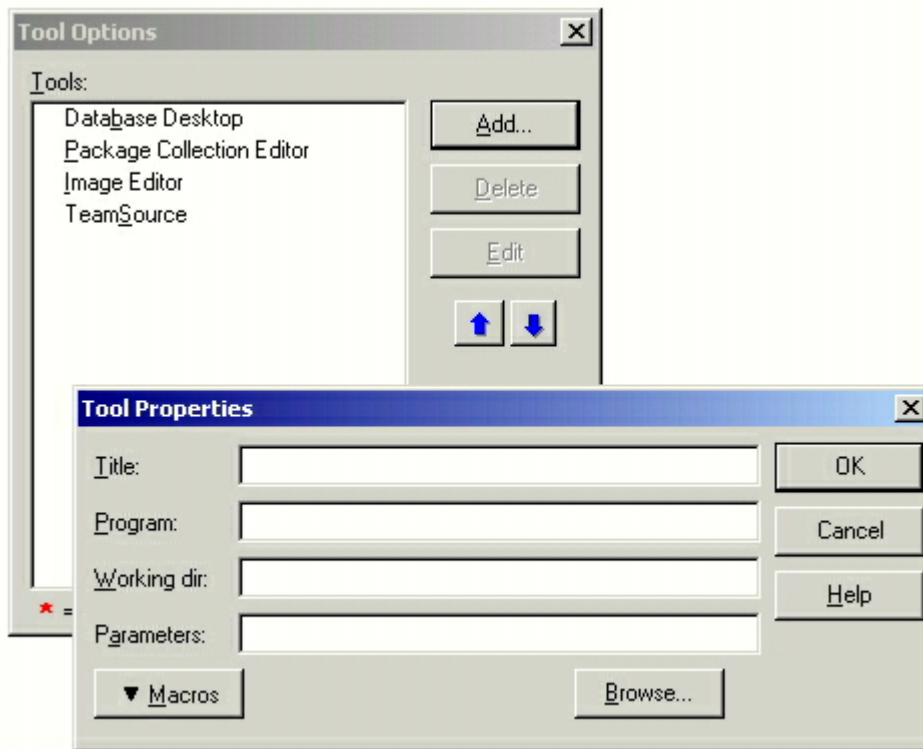
The Database menu collects the Delphi database-related tools, such as the Database Form Wizard and the Database Explorer. The Enterprise edition has the SQL Explorer instead of the Database Explorer (although the menu items is invariably called Database | Explore) and a menu item to start the SQL Monitor.

---

## The Tools Menu

The Tools menu simply lists a number of external programs and tools, just to make it easier to run them. You can use the Tools command to configure and add new external tools to the pull-down. Besides simply running a program, you can pass some parameters to it. Simple parameter lists can be passed directly on the command line, while complex ones can be built by clicking the Macros button in the lower part of the Tool Properties dialog box.

The Tools menu also includes a command to configure the Repository (discussed in the next chapter) and the Options command to configure the whole Delphi development environment. The Environment Options dialog box has many pages related to generic environment settings (the Preferences page), packages and library settings, many editor options (in the pages Editor, Display, and Colors), a page to configure the Components Palette, one for the object Browser, and one of the new Code Insight technology. I'll discuss many of these options when covering related features. You can see the dialog used to customize the Tools menu below.



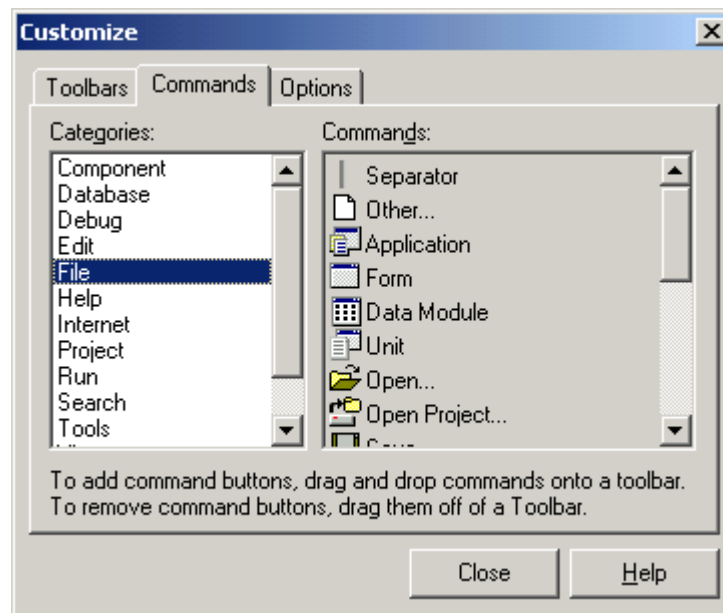
## The Help Menu

The Help menu can be used to get information about Delphi (Help | Help Topics) and also to display the Delphi About box. In this window, you can hold down the Alt key and type the letters VERSION to see the Delphi version and build number. Using other key combinations (as mentioned in the acknowledgments at the beginning of the book) you can see a list of the people involved in building Delphi. The Help menu was often populated by third-party Delphi Wizards (before the new ToolsApi made it harder to place wizards in what was their default location).

## The Delphi Toolbar

After you have used Delphi for a while, you'll realize that you use only a small subset of the available commands frequently. Some of these commands are probably already on the toolbar (Borland's name for a toolbar); some are not. If the commands you use a lot are not there, it's time to customize the toolbar so that it really helps you to use Delphi more efficiently.

An alternative to using the toolbar is to use shortcut keys. Although you must remember some key combinations to use them, shortcut keys let you invoke commands very quickly, particularly when you are writing code and your fingers are already on the keyboard.



You can easily resize the toolbar by dragging the thick line between it and the Components Palette. But the most important operations you can do with the toolbar are adding, removing, or replacing the icons using the Configure command of the toolbar local menu (simply press the right mouse button over it). This operation invokes the toolbar Editor (see above), one of the Delphi tools with the best user interface, at least in my opinion.

To add an icon to the toolbar, you simply need to find it under the proper category (corresponding to a pull-down menu), and drag it to the bar. In the same way, you can drag an icon away from the toolbar or simply move it to another location. During these operations, you can easily leave some space between groups of icons, to make them easier to remember and select.

## The Local Menus

Although Delphi has a good number of menu items, not all of the commands are available through the pull-down menus. At times, you need to use local menus for specific window areas. To activate a local menu, right-click over a window, or press Alt+F10. Even if you have other alternatives, using a local menu is usually

faster because you don't need to move the mouse up to the menu bar and select two levels of menus. It's also often easier, since all the local menu commands are related to the current window. Almost every window in Delphi (with the exclusion of dialog boxes) has its own local menu with related commands. I really suggest you get used to right-clicking on windows, because this is not only important in Delphi, but also has become a standard for most applications in Windows. Get used to it, and add local menu to the applications you build with Delphi, too.

---

## Working with the Form Designer

Designing forms is the core of visual development in the Delphi environment. Every component you place on a form and every property you set is stored in a file describing the form (a DFM file) and has some effect on the source code associated with the form (the PAS file).

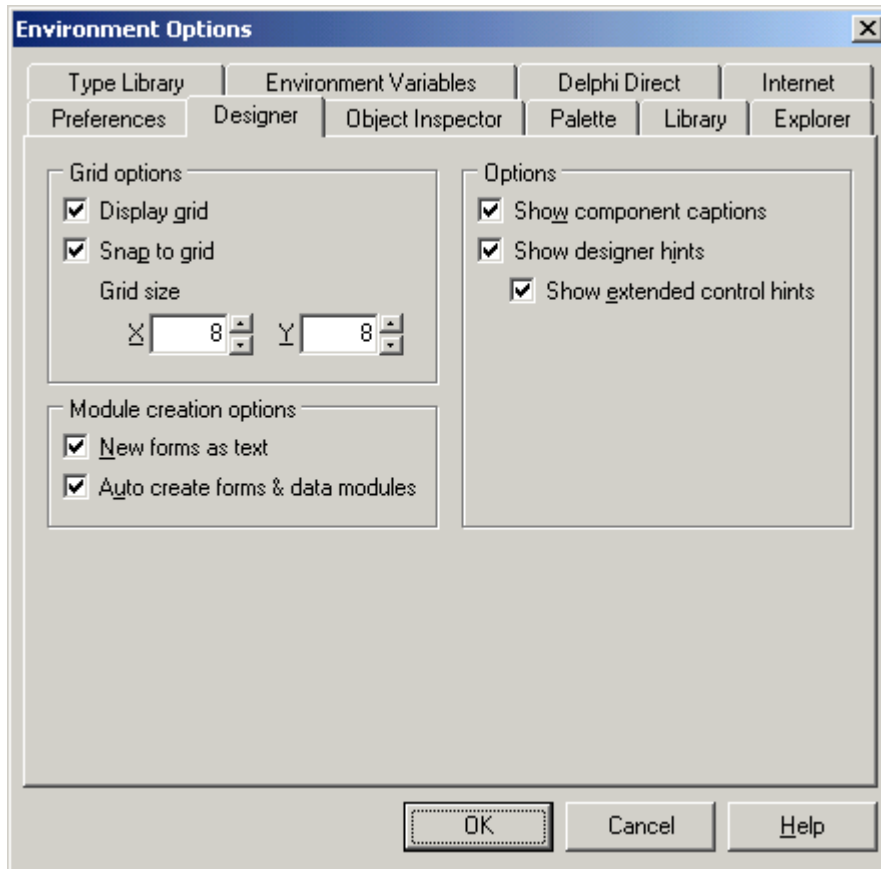
When you start a new, blank project, Delphi creates an empty form, and you can start working with it. You can also start with an existing form (using the various templates available), or add new forms to a project. A project (an application) can have any number of forms. Every time you work with a form at design-time, you are actually using Delphi's Form Designer. When you are working with a form, you can operate on its properties, on the properties of one of its components, or on those of several components at a time. To select the form or a component, you can simply click on it or use the Object Selector (the combo box in the Object Inspector), where you can always see the name and type of the selected item. You can select more than one component by Shift-clicking on the components, or by dragging a selection rectangle around the components on the form.

Even when a component covers the whole surface of the form, you can still select the form with the mouse. Just press and hold Shift while you click on the selected component. This will deselect the component and select the form by default. Using the keyboard, you can press Esc to select the parent of the current component.

While you are working on a form, the local menu has a number of useful features (some of which are also available in the Edit menu). You can use the Bring to Front and Send to Back commands to change the relative position of components of the same kind (you can never bring a graphical component in front of a component based on a window). In an inherited form, you can use the command Revert to Inherited to restore the properties of the selected component to the values of the parent form.

When you have selected more than one component, you can align or size them. Most of the options in the Alignment dialog box are also available in the Alignment palette (accessible through the View | Alignment Palette menu command). You can also open the Tab Order and Creation Order dialog boxes to set the tab order of the visual controls and the creation order of the non-visual controls. You can use the Add to Repository command to add the form you are working on to a list of forms available for use in other projects. Finally, you can use the View as Text command to close the form and open its textual description in the editor. A corresponding command in the editor local menu (View as Form) will reverse the situation.

Along with specific local menu commands, you can set some form options by using the Tools | Options command and choosing the Preferences page (up to Delphi 5) or the Designer page (from Delphi 6). This latter page is shown here:



The options related to forms refer to grid activation and size. The grid makes it easier to place components exactly where you want them on the form by “snapping” them to fixed positions and sizes. Without a grid, it is difficult to align two components manually (using the mouse).

There are two alternatives to using the mouse to set the position of a component: you can either set values for the Left and Top properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element’s position. (The Snap to Grid option works only for mouse operations.) Similarly, by pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. If you press Shift+Ctrl+an arrow key, instead, the component will be moved only at grid intervals.

Along with the commands described so far, a form’s local menu offers other commands when particular components are selected. In some cases, these menu commands correspond to component properties; others contain particularly useful commands. Table 2.1 lists the commands added to the local menu of a form when some of the components are selected (the TeeChart, Quick Report and Decision Cube components add too many commands to list here). Notice that in some cases these actions are also the default action of the component, the one automatically activated when you double-click on it in the Form Designer.

**Table 2.1: Local menu commands added when specific components are selected:** [\*\*\* not updated for recent versions of Delphi]

Menu Command	Components
Menu Designer	MainMenu, PopupMenu

Query Builder	Query (if the Visual Query Builder is available)
Fields Editor	Table, Query, StoredProc, ClientDataSet
Explore	Table, Query, StoredProc, Database
Define Parameters	Query, StoredProc
Database Editor	Database
Assign Local Data	ClientDataSet
UpdateSQL Editor	UpdateSQL
Execute	BatchMove
Columns Editor	DBGrid
Edit Report	Report
ImageList Editor	ImageList
New Page	Page Control
Next/Previous Page	Page Control, NoteBook, TabbedNotebook
Next/Previous Frame	Animate
Insert Object	OleContainer
New Button	ToolBar
New Separator	
Properties, About	All the ActiveX controls
Action Editor	WebDispatcher
ResponseEditor	QueryTableProducer, DataSetTableProducer

---

## The Component Palette

When you want to add a new component to the form you are working on, you can click on a component in one of the pages of the Component palette, and then click on the form to place the new component. On the form, you can press the left mouse button and drag the mouse to set the position and size of the component at once, or just click to let Delphi use a default size.

Each page of the palette has a number of components; each component has an icon and a name, which appears as a “fly-by” hint (just move the mouse on the icon and wait for a second). The hints show the official names of components, which I’ll use in this book. They are drawn from the names of the classes defining the component, without the initial *T* (for example, if the class is *TButton*, the name is *Button*).

If you need to place a number of components of the same kind into a form, shift-click on that component in the palette. Then, every time you click on the form, Delphi adds a new component of that kind. To stop this operation, simply click on the standard selector (the arrow icon) on the left side of the Component palette.

If you are temporarily using a mouse-less computer, you can add a component by using the View | Components List command. Select a component in the resulting list or type its name in the edit window, and then click on the Add to Form button.

Of course, you can completely rearrange the components in the various pages of the palette, adding new elements or just moving them from page to page: select Tools | Options and move to the Palette page. In this page of the dialog box, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page. It’s not a good idea to move components on the palette too often. If you do, you’ll probably waste time trying to locate them afterward.

When you have too many pages in the Component palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: rename the pages with shorter names, so that all the pages will fit on the screen. Obvious...once you've thought about it.

## The Object Inspector

When you are designing a form, you use the Object Inspector to set values of component or form properties. Its window lists the properties (or events) of the selected element and their values in two resizable columns. An Object Selector at the top of the Object Inspector indicates the current component and its data type; and you can use it to change the current selection. The Object Inspector doesn't list all of the properties of a component. It includes only the properties that can be set at design-time. As mentioned in Chapter 1, other properties are accessible only at run-time. To know about all the different properties of a component, refer to the Help files.

The right column of the Object Inspector allows only the editing appropriate for the data type of the property. Depending on the property, you will be able to insert a string or a number, choose from a list of options (indicated by an arrow), or invoke a specific editor (indicated by an ellipsis button). When a property allows only two values, such as True and False, you can toggle the value by double-clicking on it. If there are many values available, a double-click will select the next one in the list. If you double-click a number of times, all the values of the list will appear, but it is easier to select a multiple-choice value using the small combo box. For some properties, such as Color, you can enter a value, select an element from the list, or invoke a specific editor! Other properties, such as Font, can be customized either by expanding their sub-properties (indicated by a plus or minus sign next to the name) or by invoking an editor. In other cases, such as with string lists, the special editors are the only way to change a property.

The sub-property mechanism is available with sets and with classes. When you expand sub-properties, each of them has its own behavior in the Object Inspector, again depending on its data type.

You will use the Object Inspector often. It should always be visible when you are editing a form, but it can also be useful to look at the names of components and properties while you are writing code. For this reason, the Object Inspector's local menu has a Stay on Top command, which keeps the Object Inspector window in front of the Form Designer and the editor.

In Delphi 5 and 6 the features of the Object Inspector have been largely extended, with grouping and hiding or little-used properties, interface references, showing of read-only properties, and many other features too complex to discuss here.

## The Alignment Palette

The last tool related to form design is the Alignment palette. You can open this palette with the View menu's Alignment Palette command. As an alternative, you can choose the components you want to align, and then issue the Align command from the local menu of the form.

The Alignment palette features a number of commands to position the various controls, center them, space them equally, and so on. To see the effect of each button, simply move the mouse over the window and look at the fly-by hints. When I'm designing complex forms, I position the Alignment palette on the far side of the screen and make sure it always stays in sight by using the Stay on Top command of its local menu.

---

## Writing Code in the Editor

Once you have designed a form in Delphi, you usually need to write some code to respond to some of its events, as we did in Chapter 1. Every time you work on an event, Delphi opens the editor with the source file related to the form. You can easily jump back and forth between the Form Designer and the source code editor by clicking the Toggle Form Unit button on the toolbar, by clicking on the corresponding window, or by pressing the F12 function key.

The Delphi editor allows you to work on several source code files at once, using a “notebook with tabs” metaphor. Each page of the notebook corresponds to a different file. You can work on units related to forms, independent units of Pascal code, and project files; open the form description files in textual format; and even work on plain text files. You can jump from a page of the editor to the next by pressing the Ctrl+Tab keys (or Shift+Ctrl+Tab to move in the opposite direction).

When you work with the editor, you should probably expand its window so that you can see as many full lines of code as possible. A good approach is to size the editor so that it and the Object Inspector are the only windows that appear on the screen when you are writing code. By having the Object Inspector visible you can immediately see the names of the design-time properties of the components.

There are a number of environment options that affect the editor, mostly located in the Editor Options, Editor Display, and Editor Colors pages of the Environment Options dialog box. In the Preferences page, you can set the editor's Autosave feature. Saving the source code files each time you run the program can save the day when your program happens to crash the whole system (something not so rare as you might think). The other three pages of editor options can be used to set the default editor settings like keystroke mappings, syntax highlighting features, and font. Most of these options are fairly simple to understand.

The local menu of the edit window has some commands for debugging and others related to the editor itself, such as those to close the current page, open the file or unit under the cursor, view or hide the message pane below the window, and invoke the editor options discussed before.

---

## Using Editor Bookmarks

The Delphi editor also lets you set line bookmarks. When you are on a line of the editor, you can press Ctrl+Shift plus a number key from 0 to 9 to set a new bookmark, which then appears in the small gutter margin of the editor. Then you can use the Ctrl key plus the number key to jump back at that line of the editor. Pressing the Ctrl+Shift+*number* toggles the status of the bookmark, so you can use this combination again to remove it.

Bookmarks are quite useful when you have a long file and you are editing multiple methods at the same time, or to jump from the class definition to the definition of a method of the class.

Bookmarks have limitation that make them hardly usable. If you set again a given bookmark, the editor moves it. This might seem reasonable, but is actually a problem: If you create a new bookmark and happen to use the number of an existing one, by error, the older bookmark will be removed. Another odd behavior is that you



can add multiple bookmarks on the same line, but you'll only see the glyph of one of them. The real problem, however, is that bookmarks are not saved along with the file, nor restored when you reopen it. So they can be used only for a single editing session.

---

## Code Insight

Delphi editor has several features collectively known as Code Insight. The basic idea of this technology is to make it easier for both newcomers and experienced programmers to write code. There are four capabilities that Borland calls Code Insight:

- The Code Completion Wizard allows you to choose the property or method of an object simply by looking it up on a list, or by typing its initial letters. It also allows you to look for a proper value in an assignment statement.
- The Code Templates Wizard allows you to insert one of the predefined code templates, such as a complex statement with an inner begin-end block. You can also easily define new templates.
- The Code Parameter Wizard displays, in a hint or ToolTip window, the data type of a function's or method's parameters while you are typing them.
- The ToolTip Expression Evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression under the mouse cursor.

I'll cover the ToolTip Expression Evaluation later in this chapter, while introducing debugging features; in the next three sections I'll give you some more details on the other three Code Insight capabilities. You can enable and disable (or configure) each of these wizards in the Code Insight page of the Environment Options dialog box.

---

## Code Completion

There are two ways to activate this Wizard. You can simply type the name of an object, such as `Button1`, then add the dot, and wait:

```
| Button1.
```

Delphi will display a list of valid properties and methods you can apply to the object. The time you have to wait before the list is displayed depends on the Code Completion Delay option, which you can configure in the Code Insight page.

As an alternative you can type the initial portion of the property or method name, as in `Button1.Ca`, and then press `Ctrl+SpaceBar` to get the list immediately, but this time, the Wizard will try to guess which property or method you were looking for by looking at the characters you typed. You can also use this key combination in an assignment statement. If you type:

```
| x :=
```

and then press `Ctrl+SpaceBar`, Delphi will show you a list of possible objects, variables, or constants you can use at this point in the program (that is, in the current scope).

Notice that Delphi determines the elements to show in this list dynamically, by constantly parsing the code you write in the background. So if you add a new variable to a unit, it will show up in the list.

## Code Templates

Unlike the Code Completion Wizard, the Code Templates Wizard must be activated manually. You can do this by typing Ctrl+J to show a list of all of the templates.

More often, you'll first type a keyword, such as `if` or `array`, and then press Ctrl+J, to activate only the templates starting with those letters. For some keywords Borland has defined multiple templates, all starting with the keyword name (such as *ifA* and *ifB*). So if you press the keyword and then Ctrl+J, you'll get all the templates related to the keyword.

You can also use Code Templates Wizard simply to give a name to a common expression. For example, if you use the `MessageDlg` function often, you might want to enter a new Code Template called *mess*, type a description, and add then the following text:

```
MessageDlg ('|',  
            mtInformation, [mbOK], 0);
```

Now every time you need to create a message dialog box, you simply type *mess* and then Ctrl+J, and you get the full text. The vertical line (or pipe) character indicates the position in the source code where Delphi will move the cursor after pasting the text. You should choose the position where you want to start typing to edit the code generated by the template.

As this example demonstrates, Code Templates have no direct correspondence to language keywords, but are a more general mechanism. Code Templates are saved in the `DELPHI32.DCI` file, so it should be possible to copy this file to make your templates available on different machines. There seems to be no easy way to merge two Code Templates files, and there are no third-party tools yet to add more templates to a machine. Those enhancements will need to wait until Borland documents the internal structure of the `.DCI` file.

## Code Parameter

The third Code Insight technology I'll discuss here is Code Parameters, the one I was really hoping for and probably like best. Previously when I had to call an unfamiliar function I used to type the name, and then press F1 to jump to the Help system and see its parameters. Now I can simply type the function name, type the open (left) parenthesis, and the parameter names and types appear immediately on a fly-by hint window.

Notice in this figure that the first parameter appear in boldface type. After you type the first parameter and a comma, the second parameter will be set in bold, the same with the third, and so on. This is very useful for functions with many parameters, like some functions of the Window API. Try typing *CreateWindow*( and you'll understand what I mean.

Again, the Code Parameters Wizard works by parsing your code in the background. So if you write the following procedure at the beginning of the implementation section of a unit:

```
implementation  
  
procedure ShowInt (X: Integer);  
begin  
    MessageDlg (IntToStr (X),  
                mtInformation, [mbOK], 0);  
end;
```

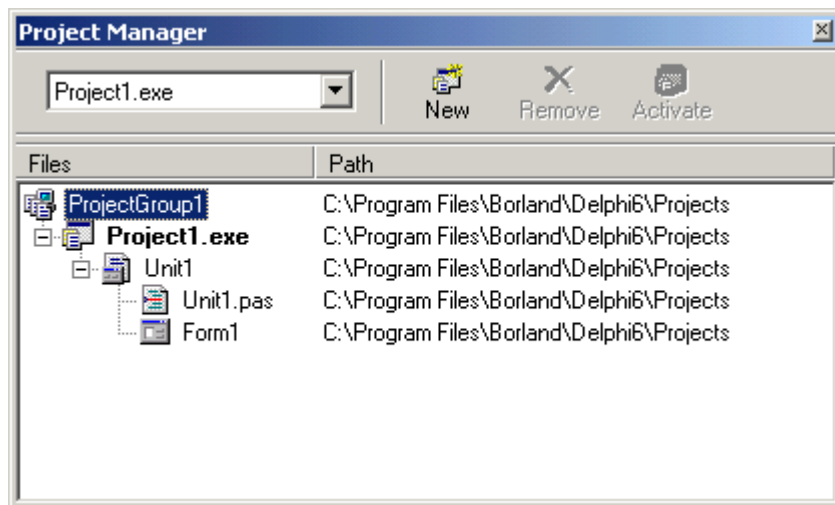
you'll have full information about its parameters when you type *ShowInt*( later.

# Managing Projects

In Delphi you also need to know how to manage project files. In Chapter 1, we saw that you can open a project file in the editor and edit the file. However, there are simpler ways to change some of the features of a project. For example, you can use the Project Manager window and Project Options.

## The Project Manager

When a project is loaded, you can choose the View | Project Manager command to open a project window. The window lists all of the forms and units that make up the current project. The Project Manager's local menu allows you to perform a number of operations on the project, such as adding new or existing files, removing files, viewing a source code file or a form, and adding the project to the repository. Most of these commands are also available in the toolbar of this window.



## Setting Project Options

From the Project Manager (or from the Project menu), you can invoke the Project Options dialog. The first page of Project Options, named Forms, lists the forms that should be created automatically at program startup (the default behavior) and the forms that are created manually by the program. You can easily move a form from one list to the other. The next page, Application, is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options, the other is to set or remove individual options in the source code with the `{$X+}` or `{$X-}` commands, where X is the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source code file, or even for just a few lines of code.

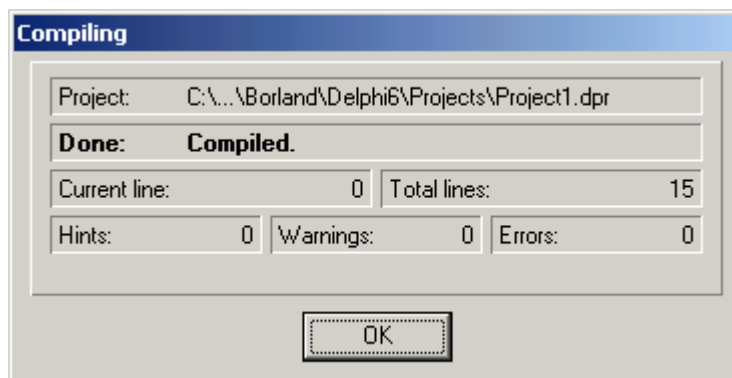
All of the Project Options are saved automatically with the project, but in a separate file with a DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options.

## Compiling a Project

There are several ways to compile a project. If you run it (by pressing F9 or clicking on the toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed. If you select `Compile | Build All`, instead, every file is compiled, even if it has not changed. This second command is seldom used, since Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options. In this case you have to use the `Build All` command to put the new options into effect.

The project lists the source code files that are part of it, and any related forms. This list is visible both in the project source and in the Project Manager, and is used to compile or rebuild a project. First, each source code file is turned into a Delphi compiled unit, a file with the same name as the Pascal source file and the DCU extension. For example, `Unit1.pas` is compiled into `Unit1.dcu`.

When the source code of the project itself is compiled, the compiled units that constitute the project are merged (or linked) into the executable file, together with code from the VCL library. You can better understand the compilation steps and follow what happens during this operation if you enable the `Show Compiler Progress` option. You'll find this option on the Preferences page of the Environment Options dialog box, under the `Compiling` heading. Although this slows down the compilation a little, the `Compile` window lets you see which source files are compiled each time (unless your computer is too fast; Delphi might compile several files per second on a fast PC).



---

# Exploring a Compiled Program

Delphi provides a number of tools you can use to explore a compiled program, including the debugger and the Object Browser.

---

## The Integrated Debugger

Delphi has an integrated debugger with a huge number of features. However, Borland also sells a more powerful stand-alone debugger, called Turbo Debugger. For nearly all of your debugging tasks, the integrated debugger works well enough, particularly if you activate the CPU view window. The stand-alone Turbo Debugger might be useful in a few special cases.

You don't need to do much to use the integrated debugger. In fact, each time you run a program from Delphi, it is executed by default in the debugger. This means that you can set a breakpoint to stop the program when it reaches a specific line of code. For example, open the Hello2 example we created in Chapter 1 and double-click on the button in the form to jump to the related code. Now set a breakpoint by clicking in the editor gutter margin, by choosing the Toggle Breakpoint command of the editor local menu, or by pressing F5.

The editor will highlight the line where you've placed the breakpoint, showing it in a different color. Now you can run the program as usual, but each time you press the button, the debugger will halt the program, showing you the corresponding line of code. You can execute this and the following lines one at a time (that is, step-by-step), look at the code of the functions called by the code, or continue running the program.

When a program is stopped, you can inspect its status in detail. Although there are many ways to inspect a value, the simplest approach is the ToolTip Expressions Evaluation. Simply move the mouse over the name of any variable, and you'll see its current value in a small hint window.

At times the ToolTip Expressions Evaluation seems not to work. This may happen if the optimizing compiler has removed some sections of generated code and placed variables in CPU registers. If you disable the compiler optimizations, you'll get more ToolTips.

---

## The Object Browser

Once you have compiled a program, you can run the Object Browser (available with the View | Browser menu command) to explore it, even if you are not running or debugging it. This tool allows you to see all of the classes defined by the program (or by the units used directly and indirectly by the program), all the global names and variables, and so on. For every class, the Object Browser shows the list of properties, methods, and variables — both local and inherited, private and public. The information displayed in the Object Browser may not mean much if you're still not familiar with the Object Pascal language used by Delphi.

---

## Additional Delphi Tools

Delphi provides many other programming tools. For example, the Menu Designer is a visual tool used to create the structure of a menu. There are also the various Wizards, used to generate the structure of an application or a new form. Other tools are stand-alone applications related to the development of a Windows application, such as the Image Editor and WinSight, a “spy” program that lets you see the Windows message flow.

There are several external database tools, such as the Database Desktop and the Database Explorer (some of which are not available in the lower-level editions of Delphi). A programmer can use other third-party tools to cover weak areas of Delphi. For example, you can use a full-blown resource editor (such as Borland’s Resource Workshop), or a tool to generate Help files more easily.

You can also install and use many additional Delphi add-on tools from third-party vendors. You’ll find demo versions of some of these tools on the companion CD, but there are many others available. Some of the add-on tools really complement Delphi nicely, and make you more productive.

---

## The Files Produced by the System

As you have seen, Delphi produces a number of files for each project, and you should know what they are and how they are named. There are basically two elements that have an impact on how files are named: the names you give to a project and its forms, and the predefined file extensions used by Delphi for the files you write and those generated by the system.

The great advantage of Delphi over other visual programming environments is that most of the source code files are plain ASCII text files. We explored Pascal source code, project code, and form description files at the end of Chapter 1. Now let’s take a minute to look at the structure of options and desktop files. Both types of files use a structure similar to Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of the HELLO.DOF file of the Hello2 example:

```
[Compiler]
A=1
B=0
...
[Linker]
MapFile=0
MinStackSize=16384
MaxStackSize=1048576
...
[Directories]
OutputDir=
SearchPath=
```

In Delphi the option files use the DOF extension, in Kylix the KOF extension (in Delphi 1 they used the OPT extension). These files have different contents and are not fully compatible.

The initial part of this file, which I've omitted, is a long list of compiler options. The same structure is used by the desktop files, which are usually much longer. It is worth looking at what is stored in these files to understand their role. In short, a desktop file (.DSK) lists Delphi windows, indicating their position and status. For example, this is the description of the main window:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

These are some of the sections related to other windows:

```
[ProjectManager]
[AlignmentPalette]
[PropertyInspector]
[Modules]
[formxxx]
[EditWindowxxx]
[Viewxxx]
```

Besides environment options and window positions, the desktop file contains a number of history lists (lists of files of a certain kind), and an indication of the current breakpoints, watches, active modules, closed modules, and forms.

---

## What's Next

This chapter presented an overview of the Delphi programming environment, including a number of tips and suggestions. Getting used to a new programming environment takes some time, particularly if it is a complex one. I could have devoted this entire book to detailing the Delphi programming environment, but I hope you'll agree that describing how to actually write programs is more useful and interesting.

A good way to learn about the Delphi environment is to use the Help system, where you can look up information about the environment elements, windows, and commands. Spend some time just browsing through the Help files. Of course, the best way to learn how the Delphi environment works is to use it to write programs. That's what Delphi is about. Now we can move on to an important feature of the Delphi environment we have only mentioned: the Object Repository and the Wizards.

# CHAPTER THREE: THE OBJECT REPOSITORY AND THE DELPHI WIZARDS

- Delphi's Object Repository
- Reusing existing applications and forms
- The Database Form Wizard
- Other Delphi Wizards
- Configuring the Object Repository

**W**hen you start working on a new application (or simply a new form), you have two choices. You can start from scratch with a blank application or form, or you can choose a predefined model from the Object Repository. If you decide to pick an existing model from the Object Repository, you have even more alternatives. You can make a copy of an existing item (called a template in Delphi 1), you can inherit from an existing item, or you can use one of the available Wizards.

A *Wizard* is a code generator. (Borland used to call them *Experts*, but in Delphi 3 the official name has been changed to Wizards, following Microsoft tradition.) Wizards ask you a number of questions, and use your answers to create some basic code, following predefined rules. You then start working on a project or a form that already has some code and components. Usually, the code generated by these tools can be compiled immediately, and it makes up the basic structure on which you build your program or form.

The purpose of this short chapter is simply to introduce you to the Object Repository and the Delphi Wizards, and to show you how easy they are to use. We won't study the code they generate, since that will be the topic of many examples in the book. From a programming standpoint, the Wizards are really useful. The pitfall is that you might be tempted to use them without trying to understand what they do. For this reason, in some examples I'll build the code manually instead of using the corresponding Wizard.

---

## The Object Repository

Delphi has several menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File menu, and also in other pull-down menus. What happens if you simply select File | New | Other (from Delphi 6, or File | New in earlier versions)? Delphi opens the Object Repository (also called the New dialog box). The Object Repository is used to create new elements of any kind: forms, applications, data modules, libraries, thread objects, components, automation objects, and more. The Object Repository dialog box has a number of pages:

- The *New* and *ActiveX* pages allow you to create many different types of new items. At times when you create a new item, Delphi asks you the name of a new class and few other things, in a sort of *mini-Wizard*.
- The “current project” page (actually you'll see the name of the project) allows you to inherit from a form or data module included in your current project. In this page you can create new



forms or data modules that inherit from those of the current project. The content of this page depends exclusively on the units included in the current project. Simply create a couple of forms and then return to this page, and you'll see that its contents have already changed.

- The *Forms*, *Dialogs*, and *Data Modules* pages allow you to create a new element of these kinds starting from an existing one or using a Wizard.
- The *Projects* page allows you to copy the files from an existing project stored in the Repository, or use the Application Wizard.
- The *Multitier*, *Business*, *WebSnap*, and *WebServices* pages provide a starting point for using some of the advanced features found only in the Enterprise edition (and mostly only starting with Delphi 6)

Use the radio buttons at the bottom of the Object Repository dialog box to indicate that you want to copy an existing item, inherit from it, or use it directly without making a copy.

The concept of inheritance in object-oriented programming is discussed in *Mastering Delphi*. In short, it is a way to add new capabilities to an existing form (or class in general) without making a full copy. This way, if you make a change in the original form (or class), the inherited form (or class) will be affected, too.

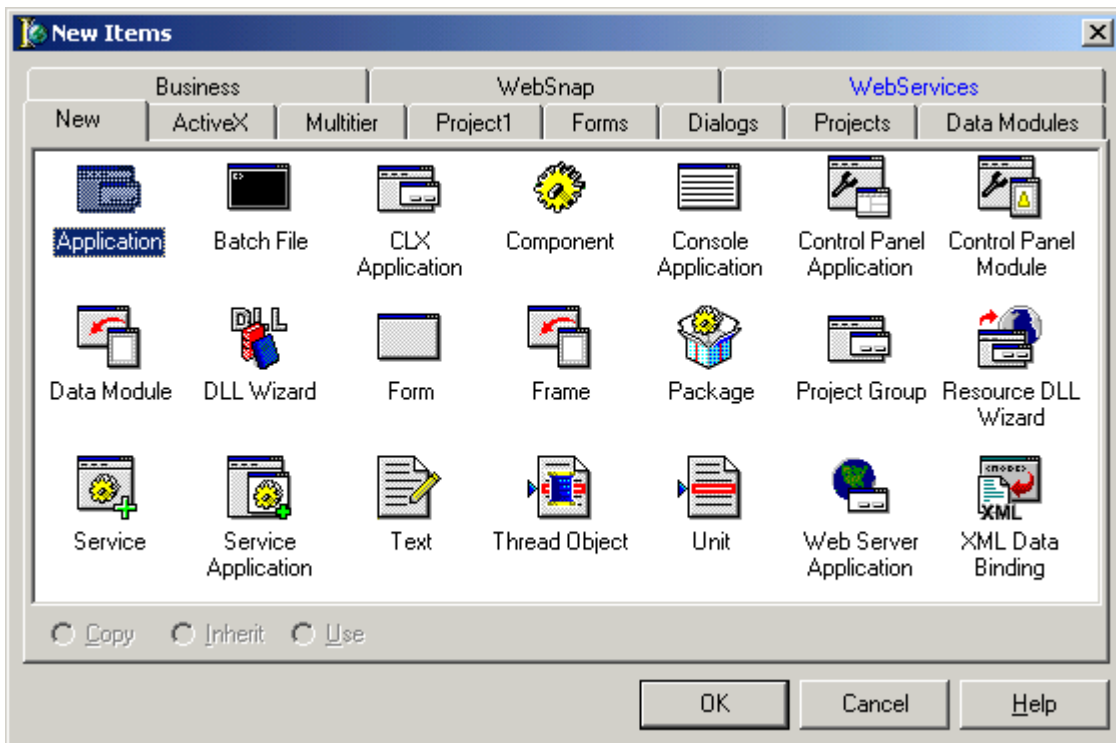
When you select a Wizard instead of a template, the only available radio button is *Copy*, meaning you'll end up with a new copy, the generated code. Keep in mind that I am discussing the pages of the Object Repository as they appear in Delphi 6 Enterprise, but different editions and versions of Delphi have fewer or different pages and items; and you can further customize the Repository, as we will see later on in this chapter.

The Object Repository has a local menu that allows you to sort items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, details). This last view is the only one that gives you the description, the author, and the date of the tool. This information is particularly important when looking at Wizards, projects, or forms you've added to the Repository.

---

## The New Page

The New page of the Object Repository (shown below) allows you to create several new items of the more commonly used kinds and is often an alternative to a direct menu command. Here is a list of the elements you can create from this page:



- *Application* creates a new blank project (the same as the command File | New | Application).
- *Batch File* opens a batch file you can include in a project group for automatic processing (to fully automate a complex build and deploy process).
- *CLX Application* creates a new blank project based on the CLX library (and on Qt), so that it will be fully portable to Kylix on the Linux platform (the same as the command File | New | CLX Application).
- *Component* creates a new Delphi component after you've completed the information requested by the simple Component Wizard. The same wizard can be activated with the Component | New Component menu command.
- *Console Application*
- *Control Panel Application*
- *Control Panel Module*
- *Data Module* creates a new blank data module (the same as the command File | New Data Module).
- *DLL Wizard* creates a simple DLL skeleton.
- *Form* creates a new blank form (the same as the command File | New Form).
- *Frame*
- *Package* creates a new Delphi component package. (You can also create a new package when creating a component.)

- *Project Group*
- *Resource DLL Wizard*
- *Service*
- *Service Application*
- *Text* opens a new ASCII text file in the Delphi editor.
- *Thread Object* creates a new thread object after asking you to fill in the New Thread Object dialog box. Multithreading in Windows is introduced in Chapter 25.
- *Unit* creates a new blank unit, a Pascal source file not connected with a form.
- *Web Server Application* allows you to create an ISAPI/NSAPI add-in DLL, a CGI stand-alone executable, a Win-CGI stand-alone executable, an Apache module, or a server application based on Delphi's Web Debugger. In each case Delphi creates a simple project based on a Web module (a special type of data module) instead of a form.
- *XML Data Binding*

[\*\*\*Many of the descriptions for newer entries are still missing]

As some of the features made available by the Object Repository are quite advanced, it doesn't make

---

## The Forms Page

This page lists predefined forms. Here is a short list of the predefined forms available in Delphi:

- *About box* is a simple About box.
- *Dual list box* is a form with two different list boxes, allowing a user to select a number of elements from one list and move them to the other list by pressing a button. Along with the components, this form contains a good amount of nontrivial Pascal code.
- *QuickReport Labels* creates a report form based on the QuickReport component.
- *QuickReport List* creates another form based on QuickReport, with a different layout.
- *QuickReport Master/Detail* is a third predefined report form with a more complex structure.
- *Tabbed pages* is a form based on the Windows PageControl.

Wizards can only be executed. The other forms, by contrast, can be used in multiple ways. You can add them into a project, inherit from them (in this case the original form will be automatically added to your project), or use them directly. When you use a form or inherit from one, take care not to make any change on the original forms in the Repository.

---

## The Dialogs Page

This page is similar to the previous one, but includes forms that are typically used as dialog boxes. Here is the list of its items:

- *Dialog with help* is available in two versions. One has the buttons on the right side of the form (*Vertical*), and the other has them in the lower portion (*Horizontal*), as you can see from the corresponding icons.
- *Dialog Wizard* is a simple Wizard capable of generating different kinds of dialog boxes with one or more pages, as we will see later in this chapter.
- *Password dialog* is a dialog box with a simple edit box for entering a password.
- *Reconcile Error Dialog* is used in MIDAS or DataSnap applications.
- *QuickReport Wizard* creates a Print Options dialog box for a report.
- *Standard dialog* is also available in two versions, again *Horizontal* and *Vertical*, with buttons in different positions.

---

## The Data Modules Page

You already know what a project and a form are, but what is a data module? It is a sort of form that never appears on screen at run-time and can be used to hold nonvisual components. It is mostly used to implement code related to database access. This page has only a data module at start-up, *Customer Data*. If you have several forms or applications accessing the same data tables and database queries, you can easily define new data modules and add them to the repository.

---

## The Projects Page

The last page we will look at in this introduction contains project schemes you can use as the starting point for your own application. These projects often include one or more forms. Here is the list of them:

- *Application Wizard* is another simple Wizard that allows you some limited choices for the file structure and other elements of an application.
- *MDI Application* defines the main elements of a Multiple Document Interface (MDI) program. It defines a main form for the MDI frame window, with a menu, a status bar, and a toolbar. It also defines a second form that can be used at run-time to create a number of child windows.
- *SDI Application* defines a main form with the standard attributes of a modern user interface, including a toolbar and a status bar, and also a typical About box.
- *Win2000 Logo Application* defines a sample application with most of the elements required by Microsoft for an application to get the Windows 2000 compatibility logo. This command basically creates an SDI application with a RichEdit component in it, and adds the code needed to make the application mail-enabled.

- *Win95 Logo Application* defines a sample application with most of the elements required by Microsoft for an application to get the Windows 95/98 compatibility logo. The application is similar to the one above.

When you select one of these projects, Delphi asks you to enter the name of an existing or a new directory. If you indicate a new directory, Delphi will automatically create it.

For example, we can create an SDI project based on the corresponding template. Then we can customize it, giving it a proper title, removing or adding new components, and writing some code. Some interesting components are already there, however, and there is even some ready-to-use code to make those components work properly. The menu and toolbar of the application, can be used to open some dialog boxes. File Open and File Save dialog boxes are wrapped up in components that are added to the form by the template; the About box is defined by the template as a second form.

In the simple SdiTemp example, I've decided to make just a few limited changes: I've entered a new title for the main form and some text for the labels for the About box (the property to use is Caption in both cases). The result is the application shown in [\*\*\*missing figure] and available in the source code.

---

## Delphi Wizards

Besides copying or using existing code, Delphi allows you to create a new form, application, or other code files, using a Wizard. Wizards (or Experts) allow you to enter a number of options and produce the code corresponding to your choices.

One of the most important predefined wizards is the Database Form Wizard, which you can activate using the Database | Form Wizard menu item or the icon in the Forms page of the Object Repository. There are also some other simple Wizards in Delphi, such as the Application Wizard and the Dialog Wizard. I've listed various other Wizards in the description of the pages of the Object Repository, in the last section.

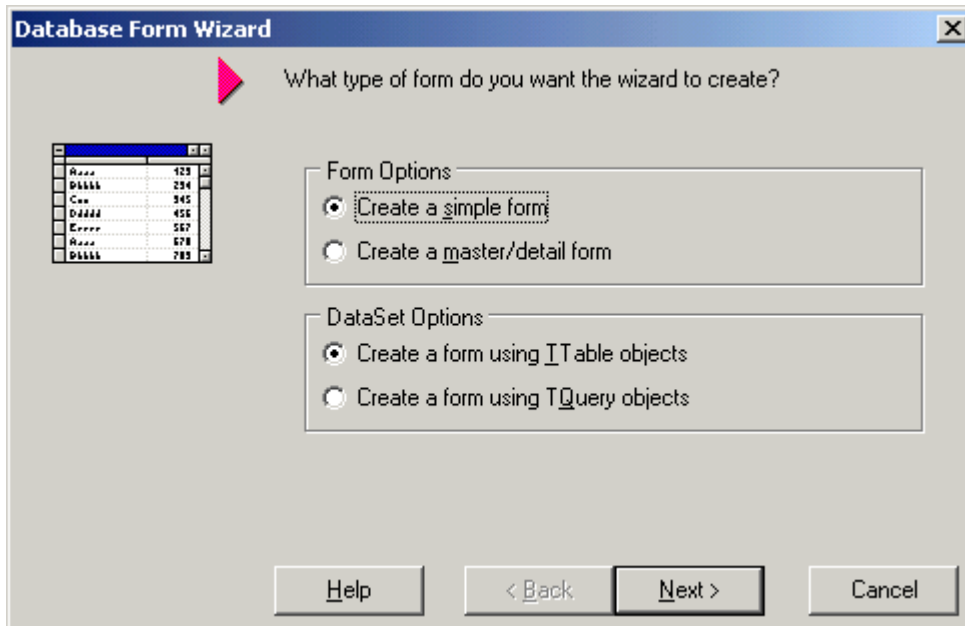
You can also buy add-on Wizards from third-party tool providers, download one from my web site, or even write your own Wizard. Add-on Wizards often show up in the Help menu, but it is possible to add new menu items in other Delphi pull-down menus or install Wizards in various pages of the Object Repository.

---

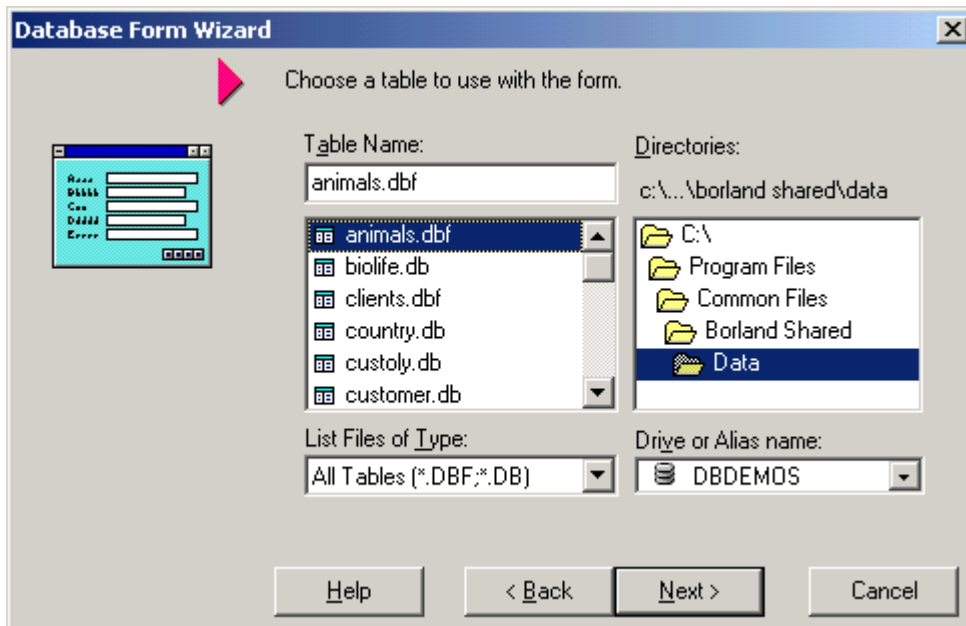
## The Database Form Wizard

In this section, I'll show you a quick example of the use of the Database Form Wizard, but I won't describe the application we build in detail. In this example, we'll build a database program using some of the data already available in Delphi. Note that you have to create a project first, and then start the Database Form Wizard. So you usually end up with two forms, unless you remove the original form from the project. Fortunately, one of the Wizard's options, displayed at the end, lets you select the new form generated by the Wizard as the main form.

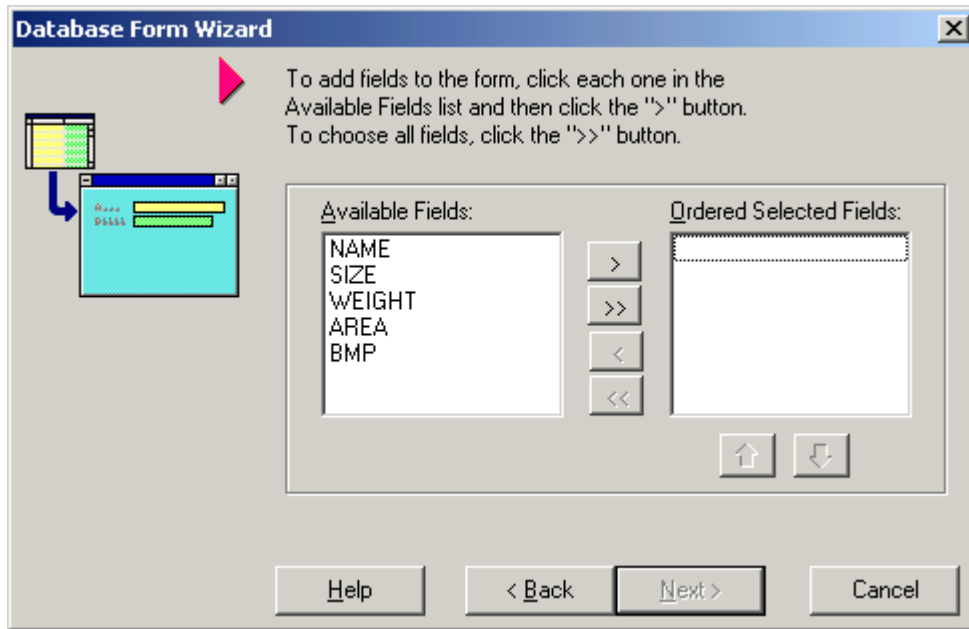
1. As soon as you start the Database Form Wizard, you will be presented with a number of choices, which depend on the options you choose at each step. The first page lets you choose between a simple or a master detail form, and between the use of tables or queries. Leave the selections as they are by default, and move on by clicking on the Next button.



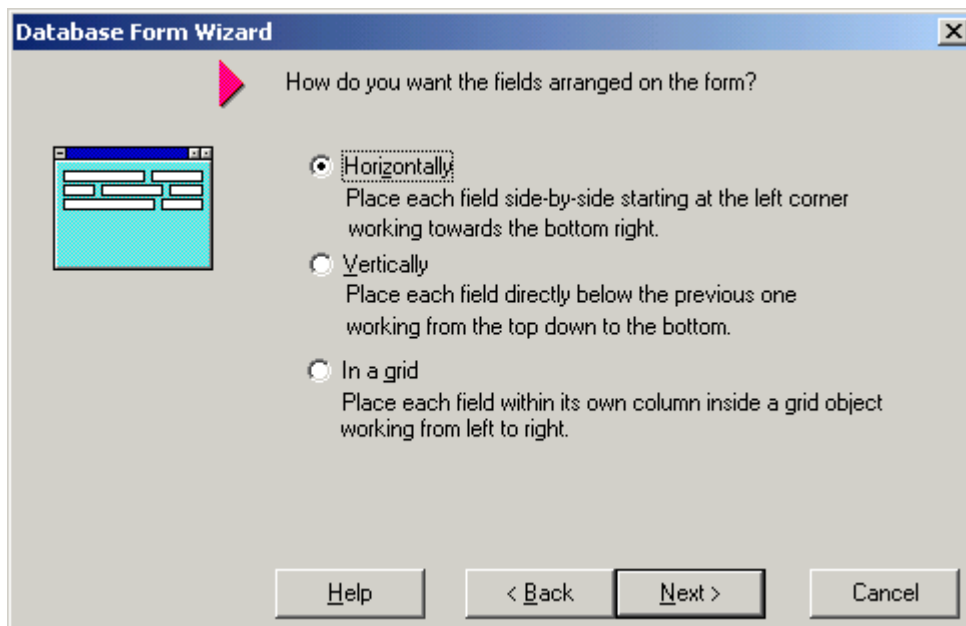
- 2. In the next page you can choose an existing database table to work on. In the Drive or Alias Name combo box, there should be a DBDEMOS alias. After you select this option, a number of Delphi demo database tables appear in the list. Choose the first, animals.dbf.



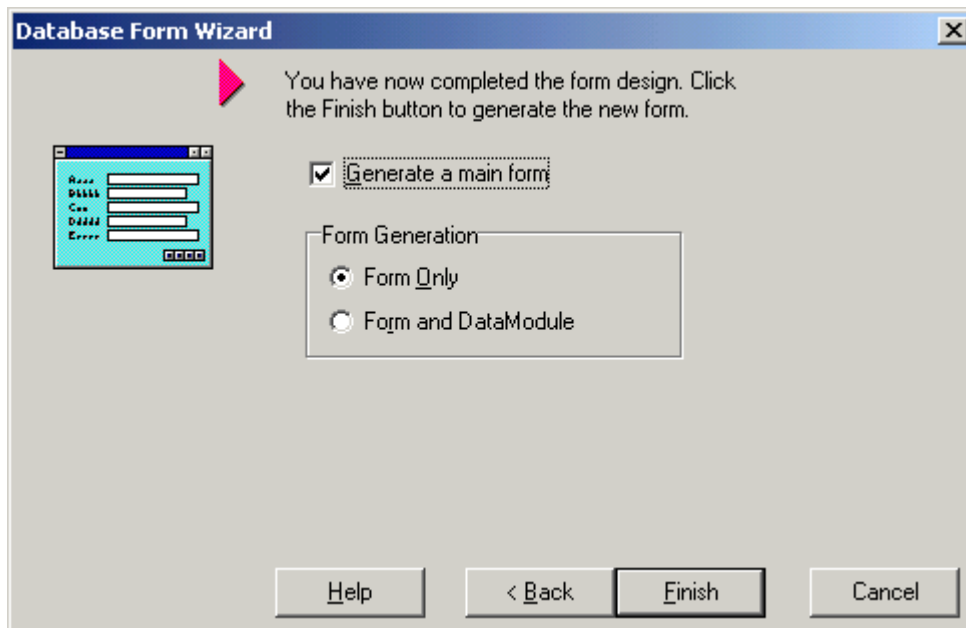
- 3. In the third page, you can choose the fields of the selected database table that you want to consider. To build this first example, choose all of the fields by clicking on the >> button.



- 4. On the next page, you can choose from various layouts. If you choose Vertical, the next page will ask you the position of the labels. The default option, Horizontal, might do.



- 5. The next page is the last. Leave the Generate a Main Form check box and the Form Only radio button selected, and click on the Finish button.



You can immediately compile and run the application. The result is a working database application, which allows you to navigate among many records using the buttons. This specific application (the DataExp example) even has a graphical field, displaying a bitmap with the current animal.

The output of the generated form is usually far from adequate. In this case, the image area is too small; at other times the positioning of the controls may not be satisfactory. Of course, you can easily move and resize the various components placed on the form at design-time.

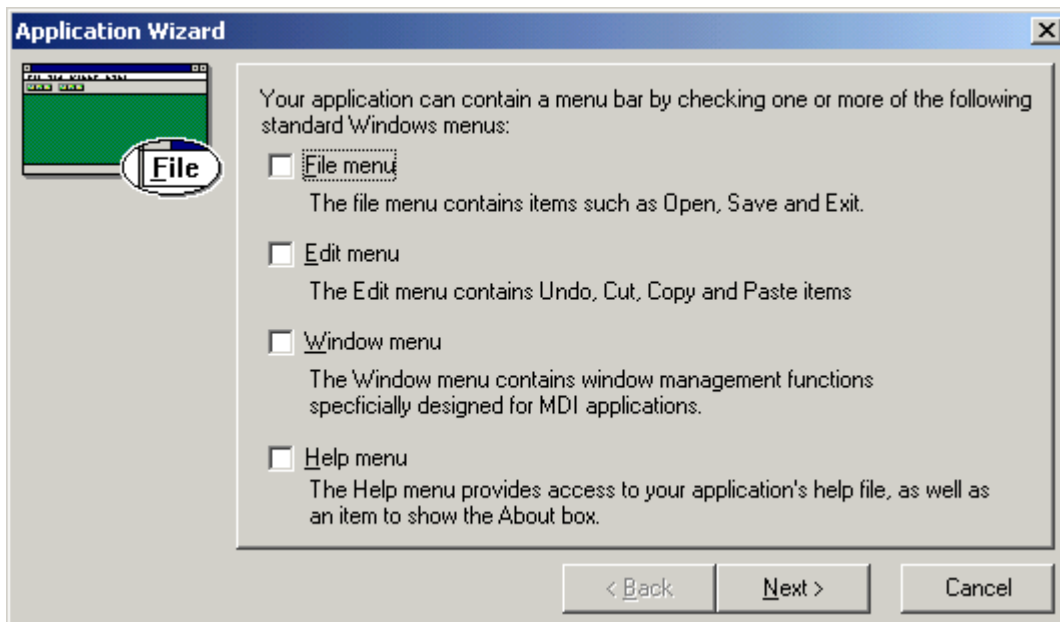
To make this task easier, you can select the Table component (the one in the upper-left corner of the form) and toggle its Active property to True. Then the data of the table's first record will be displayed at design-time. This is helpful because it allows you to see an example of the length of the field's text and the size of the image. Note that the Database Form Wizard generates almost no Pascal code, besides a line used to open the table when the program starts. The capabilities of the resulting programs stem from the power of the database-related components available in Delphi.

---

## The Application Wizard

Another interesting (although less powerful) tool is the Application Wizard. You can activate it from the Projects page of the Object Repository. The Application Wizard allows you to create the skeleton of a number of different kinds of applications, depending on the options you select.





The first page of this Wizard allows you to add some standard pull-down menus to the program: File, Edit, Window, and Help. If you select the File menu, the second page will ask you to enter the file extensions the program should consider. You should enter both a description of the file, such as *Text file (\*.txt)*, and the extension, *txt*. (You can input several extensions, each with its own description.) These values will be used by the default File Open and File Save dialog boxes that the Application Wizard will add to the program if you select the file support option.

Then, if you have selected any of the pull-down menus, the Application Wizard displays a nice visual tool you can use to build a toolbar. Unfortunately, this tool is not available as a separate editor inside Delphi. You simply select one of the pull-down menus, and a number of standard buttons corresponding to the typical menu items of this pull-down menu appear (but only if the menu has been selected on the first page of the Application Wizard).

To add a new button, select one of them in the graphical list box on the right and press the Insert button. The new toolbar button will be added at the position of the small triangular cursor. You can move this cursor by clicking on one of the elements already added to the toolbar. This cursor is also used to indicate a button you want to remove from the toolbar.

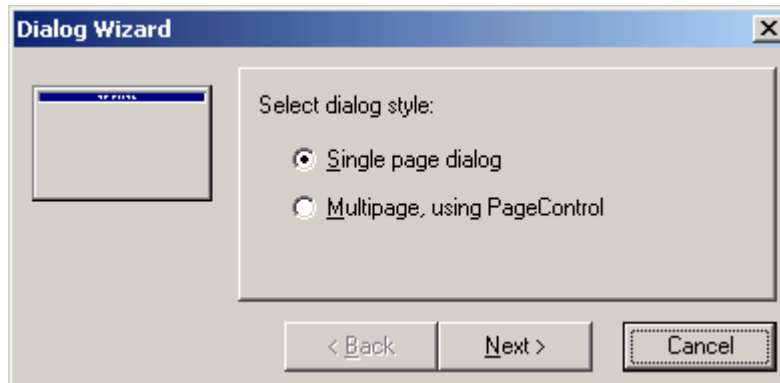
When the toolbar is finished, you can move to the last page. Here you can set several other options, such as choosing MDI support, adding a status bar, and enabling hints. You can also give a name to the new application and specify a directory for the source files. The name of the application can be long, but it cannot contain white spaces (it should be a valid Pascal identifier), and the directory for the application should be an existing directory. To place the project files in a new directory, choose the Browse button, enter the new path, and the dialog box will prompt you to create the new directory.

Although it is somewhat bare and it has room for improvement, the Delphi Application Wizard is much more useful than the predefined application templates for building the first version of an application. One of its biggest advantages is that you can define your own toolbar. Another advantage is that the Application Wizard generates more code (and more comments) than the corresponding templates do. The disadvantage of this Wizard is that it generates an application with a single form. Its MDI support is limited, because no child form is defined, and the generated application has no About box.

---

## The Dialog Wizard

Delphi's Dialog Wizard is a simple Wizard provided mostly as a demo, with its own source code. From the code of this Wizard, in theory you should be able to learn how to build other Wizards of your own. However, you can still use the Dialog Wizard as a tool to build two kinds of dialog boxes: simple dialog boxes and multiple-page dialog boxes based on the Windows PageControl component.



If you choose the simple dialog box, the Wizard will jump to the third page, where you can choose the button layout. If you choose the multiple-page dialog box, an intermediate page will appear to let you input the text of the various tabs. This Wizard is an alternative to the corresponding form templates of the Object Repository. Its advantage is that it allows you to input the names of the PageControl tabs directly.

---

## Customizing the Object Repository

Since writing a new wizard is far from simple, the typical way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the *New* and “current project” pages).

---

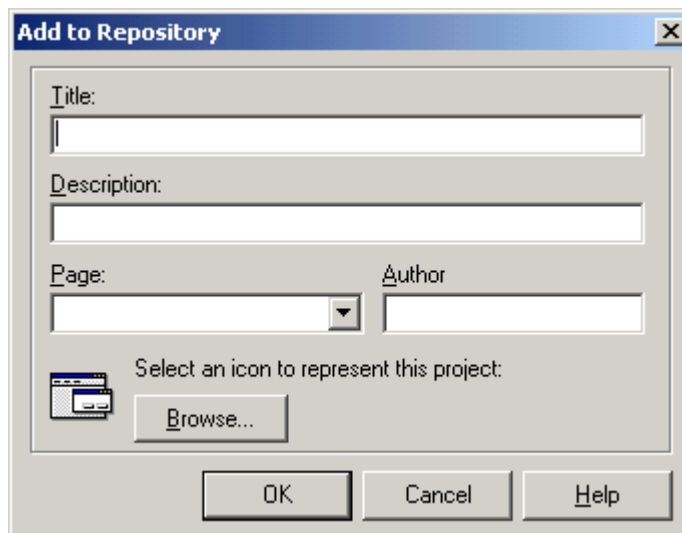
### Adding New Application Templates

Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of two or more similar programs, you can save the current status to a template, ready to use later on.

Although Borland calls everything you can put in the Object Repository an *object*, from an object-oriented perspective this is far from true. For this reason I call the schemes you can save to disk for later use *templates*. Application templates, in particular, do not relate to objects or classes in any way, but are copied to the directory of your new project. *Object Repository* sounds much better than *Browse Gallery* (the name used in Delphi 1), but besides the capability to activate form inheritance, there is not much object-oriented in this tool.

You can add a project to the Repository by using the Project | Add to Repository command, or by using the corresponding item of the local menu of the Project Manager window. As a very simple example, just to demonstrate the process, the following steps describe how you can save the slightly modified version of the default SDI template (shown earlier) as a template:

1. Open the modified SdiTemp example (or any other project you are working on).
2. Select the Project | Add to Repository menu command (or the Add Project to Repository command in the local menu of the Project Manager window).
3. In the Add to Repository dialog box, enter a title, a description for the new project template, and the name of the author. You can also choose an icon to indicate the new template or accept the default image. Finally, choose the page of the Repository where you wish to add the project.
- 4.: Click on OK, and the new template is added to the Delphi Object Repository.



Now, each time you open the Object Repository, it will include your custom template. If you later discover that the template is not useful any more, you can remove it. You can also use a project template to make a copy of an existing project so that you can continue to work on it after saving the original version.

However, there is a simpler way to accomplish this: copy the source files to a new directory and open the new project. If you do copy the source files, do *not* copy the DSK file, which indicates the position of the Windows on the screen. The DSK file holds a list of files open in the editor, using an absolute path. This means that as soon as you open the new project and start working on it, you may well end up editing the source code files of the original project and compiling the files of the new version (the project manager stores relative paths).

This will certainly surprise you when the changes you make in code or in forms seem to have no effect. Simply deleting the DSK file, or not copying it in the first place, avoids this problem.

## The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or wizards, this is not what you want. To solve this problem, you can add an Empty Project template to the Gallery.

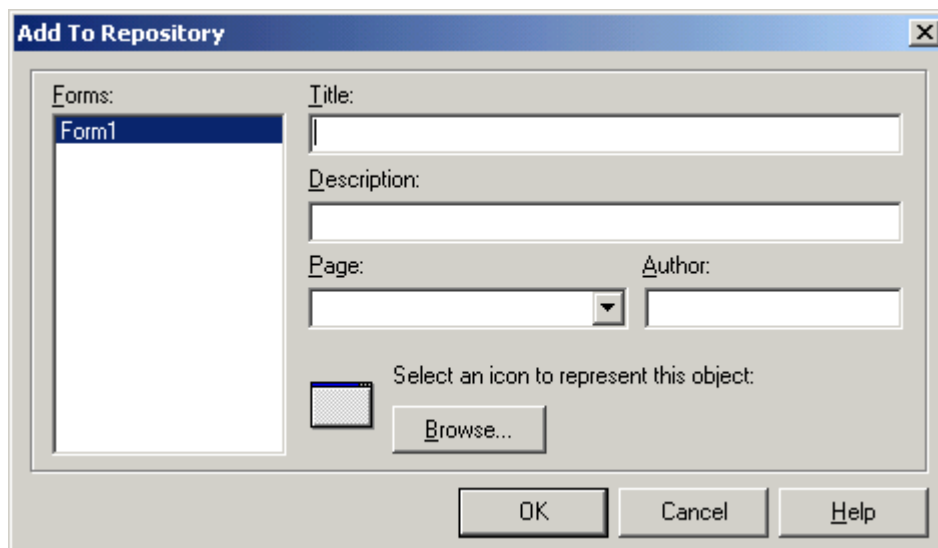
The steps required to accomplish this are simple:

- 1: Create a new project as usual.
- 2: Remove its only form from the project.
- 3: Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages. You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you need to use the File | Save Project As command to give a new name to the project, since saving the project automatically uses the default name in the template.

## Adding New Form Templates to the Object Repository

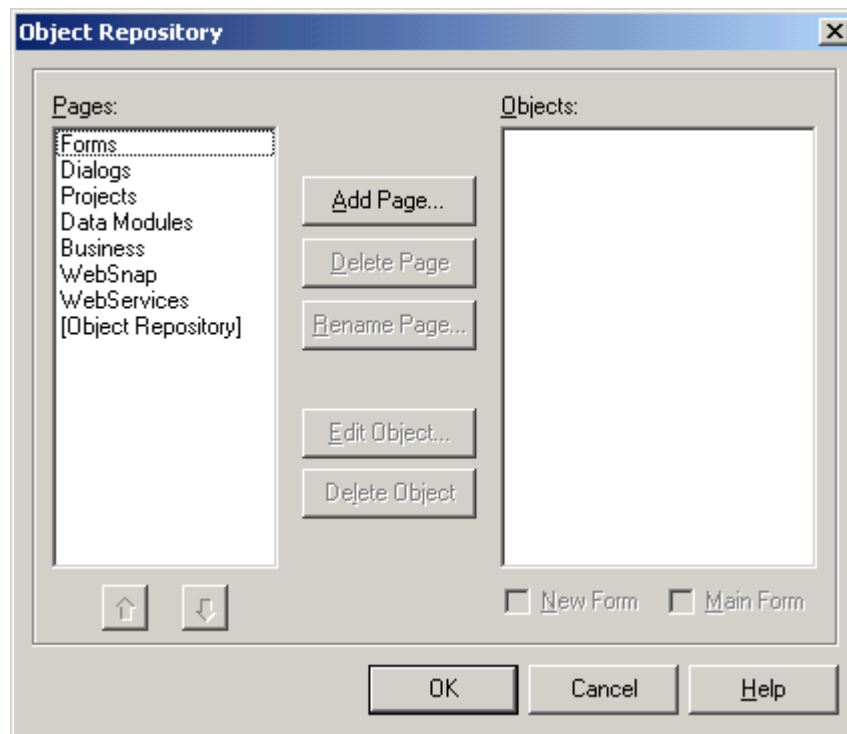
Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form you want to add, right-click on it, and select Add to Repository from the local menu. In the dialog box that appears (see below), you can choose which form of the current project should be added to the Repository, and set the title, description, author, page, and icon, as usual. Once you have set these elements and clicked on OK, the form is added to the proper page of the Object Repository.



This approach is suggested if you have a complex form to make available to other applications and other programmers. They will be able to use your form as is, make a copy of it, and inherit from it. For this reason, adding forms to the Repository is far more flexible than adding projects, which can only be copied as the starting point of a new application.

## The Object Repository Options

To further customize the Repository, you can use the Tools | Repository command to open the Object Repository dialog box. This dialog box is quite easy to use; on the left is the list of current Repository pages and on the right the list of items in each page, including both templates and Wizards. You can use this dialog to move repository items to different pages, to add new elements, or to delete existing ones.



You can use the three page-related buttons and the two buttons with arrows below the list of pages to arrange the structure of the Object Repository, adding new pages, renaming or deleting them, and changing their order. All these operations affect some of the tabs of the Object Repository itself (other tabs are fixed).

An important element of the Object Repository setup is the use of defaults:

- Use the *New Form* check box below the list of objects to designate the current form or Wizard as the default, to be used when a new form is created (File | New | Form). Only one object in the Object Repository can be used as a default; it is marked with a special symbol placed over its icon.

- The *Main Form* check box, instead, is used to indicate the form or Wizard used to create the main form of a new application (File | New | Application) when no special New Project is selected (see next bullet). The current Main form is indicated by a second special symbol.
- The New Project check box, available when you select a project object, can be used to mark the default project that Delphi uses when you issue the File | New | Application command. Also, the New Project is indicated by its own special symbol.

If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the OBJREPOS subdirectory of the Delphi main directory. At the same time, if you directly use a form or any other object without copying it, then you end up having some files of your project in this directory. It is important to realize how the repository works, because if you want to modify a project or an object saved in the repository, the best approach is to operate on the original files, without copying data back and forth to the Repository.

---

## Installing new DLL Wizards

Technically, new Wizards come in two different forms. Wizards may be part of components or packages, and in this case are installed the same way you install a component or a package. Other Wizards are distributed as stand-alone DLLs. In this case you should add the name of the DLL in the Windows Registry under the key: Software\Borland\Delphi x.0\Experts.

Simply add a new string key under this, choose a name you like (it doesn't really matter) and use as text the path and filename of the Wizard DLL. You can look at the entries already present under the Experts key to see how the path should be entered.

---

## What's Next

In this short chapter, we have seen how you can start the development of an application by using Delphi templates and Wizards. You can use one of the predefined application templates or form objects, start the Database Form Wizard, or use the other Wizards for a fast start with applications, forms, and other objects. However, I'll rarely use these templates and Wizards in the rest of the book. With the exception of the Database Form Wizard, these tools let you build only very simple applications, which you can often put together yourself in seconds when you are an experienced Delphi programmer. For beginners, Wizards and templates provide a quick start in code development. But you are not going to *remain* a beginner, are you?

So with the next chapter we'll start focusing on the use of the actual components and controls available in Delphi, to build actual applications, even if simple ones. Differently from other books of mine, which focus more on the foundations, this text proceeds mainly through practical examples.

Notice that the only reason there is no introduction to the use of the language and the core structure of the VCL, is that because you can find these topics covered in Essential Pascal (for the core language) and Mastering Delphi (for the OOP features and the VCL architecture).



## CHAPTER 4: A TOUR OF THE BASIC COMPONENTS

- Clicking a button or another component
- Adding colored text to a form
- Dragging from one component to another
- Accepting input from the user
- Creating a simple editor
- Making a choice with radio buttons and list boxes
- Allowing multiple selections
- Choosing a value in a range

Now that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the Visual Component Library, we are ready to delve into the central part of the book: the use of components. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

The system comes with a number of ready-to-use components. I will not describe every component in detail, examining each of its properties and methods. If you need this information, you can find it easily in the Help system. The aim of Part II of this book is to show you how to use some of the features offered by the Delphi predefined components to build applications. In fact, this chapter and those following will be based heavily on sample programs. These examples tend to be quite simple — although I've tried to make them meaningful — in order to focus on only a couple of features at a time.

I'll start by focusing on a number of basic components, such as buttons, labels, list boxes, edit fields, and other related controls. Some of the components discussed in this chapter are present in the Standard page of the Delphi Components palette; others are in different pages. I'm not going to describe all the components of the Standard page, either. My approach will be to discuss, in each chapter, logically related components, ignoring the order suggested by the pages of the Components palette.

---

### Windows Own Components

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window with a specific behavior, some properties, and some methods (although traditional C language code used to access the predefined components in Windows by sending and receiving messages). These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components.



Actually Microsoft's third step is its ActiveX controls (an extension of the older OCX technology), the natural successor of VBX controls. In Delphi you can use both ActiveX and native components, but if you look at the technology, Delphi components are really ahead of the ActiveX controls. It is enough to say that Delphi components use OOP to its full extent, while ActiveX controls do not fully implement the concept of inheritance. With the recent introduction of the .NET framework, Microsoft has finally followed Borland and Sun (with Java) with the adoption of a technology that treats components as classes and vice versa.

Windows 3 had six kinds of predefined controls, generally used inside dialog boxes. They were buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Windows 95 added a number of new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. These controls were already used by programmers, who had to re-implement them each time. Windows 95 developers could for the first time use the standard common controls provided by the system, and starting with Delphi 3 developers had the advantage of having corresponding easy-to-use components.

The standard system controls are the basic components of each Windows application, regardless of the programming language used to write it, and are very well known by every Windows user. Delphi literally wraps these Windows predefined controls in some of its basic components — including those discussed in this chapter.

Notice that CLX provides components similar to Windows common controls that are portable to Kylix. This is important as Linux, of course, hasn't got Microsoft's common controls!

---

## Clicking a Button

In the first chapter of this book, we built small applications based on a button. Clicking on that button caused a “*Hello*” message to appear on the screen. The only other operation that program performed was moving the button so that it always appeared in the middle of the form. Since then we've seen other examples that used buttons as a way to perform a given action (using their `OnClick` event).

Now we are going to build another form with several buttons and change some of their properties at run-time; in this example, clicking a button will usually change a property of another button. To build the Buttons program, I suggest you follow the instructions closely at first and then make any changes you want. Of course, you can read the description in the book and then work on the source files, if you prefer.

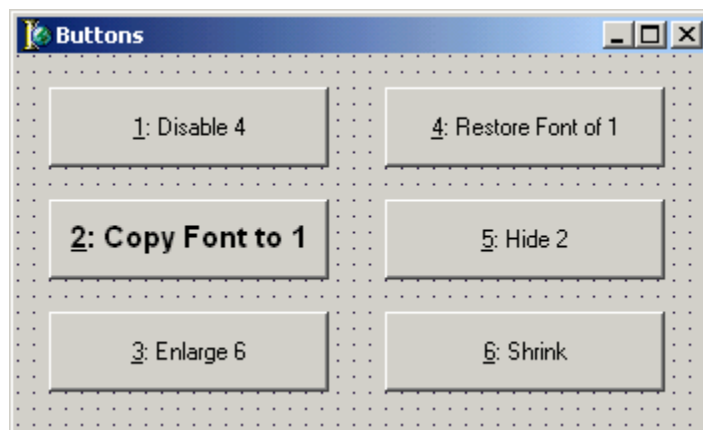
---

## The Buttons Example

First, open a new project and give it a name by saving it to disk. I've given the name `ButtonF.pas` to the unit describing the form and the name `Buttons.dpr` to the project. Now you can create a number of buttons, let's say six.

Instead of selecting the component, dragging it to the form, and then selecting it again to repeat the operation, you can take a shortcut. Simply select the component by clicking on the Components palette while holding down Shift. The component will remain selected, as indicated by a little border around it. Now you can create a number of instances of that component.

Even if you use the grid behind the form, you might need to use the Edit | Align command to arrange the buttons properly. Remember that to select all six buttons at a time, you can either drag a rectangle around them or select them in turn, holding down the Shift key. In this case, it's probably better to select a column of three buttons at a time and arrange them.



Now that we have a form with six buttons, we can start to set their properties. First of all, we can give the form a name (*ButtonsForm*) and a caption (*Buttons*). The next step is to set the text, or *Caption* property, of each button. Usually, a button's *Caption* describes the action performed when a user clicks on it. We want to follow this rule, adding the number of the button at the beginning of each caption. So if the first button disables button number four (which is the one on the same row), we can name it *1: Disable 4*. Following the same rule, we can create captions for the other buttons.

For a summary of the properties of the components of this form, you can refer to its textual description:

```

object ButtonsForm: TButtonsForm
  Caption = 'Buttons'
object Button1: TButton
  Caption = '&1: Disable 4'
 OnClick = Button1Click
end
object Button2: TButton
  Caption = '&2: Copy Font to 1'
  Font.Color = clBlack
  Font.Height = -15
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  ParentFont = False
  OnClick = Button2Click
end
object Button3: TButton
  Caption = '&3: Enlarge 6'

```

```
    OnClick = Button3Click
end
object Button4: TButton
    Caption = '&4: Restore Font of 1'
    OnClick = Button4Click
end
object Button5: TButton
    Caption = '&5: Hide 2'
    OnClick = Button5Click
end
object Button6: TButton
    Caption = '&6: Shrink'
    OnClick = Button6Click
end
end
```

Notice that every button has an underlined shortcut key, in this case the number of the button. Simply by placing an ampersand (&) character in front of each caption, as in '&1: Disable 4', we can create buttons that can be used with the keyboard. Just press a number below 7, and one of the buttons will be selected, although you won't see it pressed and released.

The final step, of course, is to write the code to provide the desired behavior. We want to handle the OnClick event of each button. The easiest code is that of Button2 and Button4. When you press Button2, the program copies the font of this button (which is different from the standard font of the other buttons) to Button1, and then disables itself:

```
procedure TButtonsForm.Button2Click(Sender: TObject);
begin
    Button1.Font := Button2.Font;
    Button2.Enabled := False;
end;
```

Pressing Button4 restores the original font of the button. Instead of copying the font directly, we can restore the font of the form, using the ParentFont property of the button. The event also enables Button2, so that it can be used again to change the font of Button1:

```
procedure TButtonsForm.Button4Click(Sender: TObject);
begin
    Button1.ParentFont := True;
    Button2.Enabled := True;
end;
```

To implement the Disable and Hide operations of Button1 and Button5, we might use a Boolean variable to store the current status. As an alternative, we can decide which operation to perform while checking the current status of the button — the status of the Enabled property. The two methods use two different approaches, as you can see in the following code:

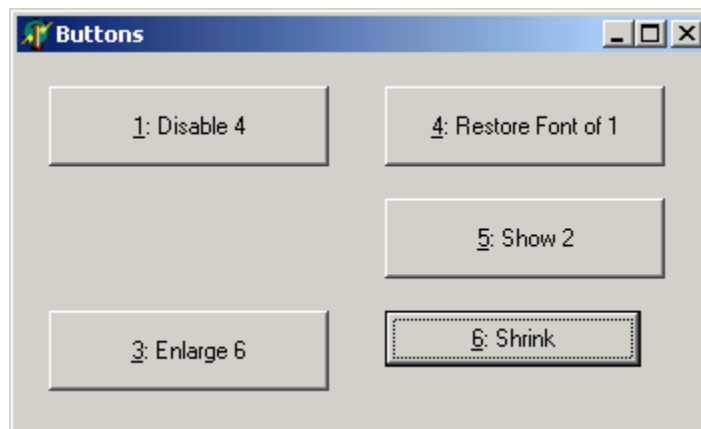
```
procedure TButtonsForm.Button1Click(Sender: TObject);
begin
    if not Button4.Enabled then
    begin
        Button4.Enabled := True;
        Button1.Caption := '&1: Disable 4';
    end
```

```

else
begin
    Button4.Enabled := False;
    Button1.Caption := '&1: Enable 4';
end;
end;

procedure TButtonsForm.Button5Click(Sender: TObject);
begin
    Button2.Visible := not Button2.Visible;
    if Button2.Visible then
        Button5.Caption := '&5: Hide 2'
    else
        Button5.Caption := '&5: Show 2';
end;

```



You can see the results of this code in the figure above. The last two buttons have *unconstrained* code. This means that you can shrink Button6 so much that it will eventually disappear completely:

```

procedure TButtonsForm.Button3Click(Sender: TObject);
begin
    Button6.Height := Button6.Height + 3;
    Button6.Width := Button6.Width + 3;
end;

procedure TButtonsForm.Button6Click(Sender: TObject);
begin
    Button6.Height := Button6.Height - 3;
    Button6.Width := Button6.Width - 3;
end;

```

It would have been quite easy, in any case, to check the current size of the button and prevent its reduction or enlargement by more than a certain value.

---

## Clicking the Mouse Button

Up to now we have based the examples on the `OnClick` event of buttons. Almost every component has a similar click event. But what exactly is a click? And how is it related to other events, such as `OnMouseDown` and `OnMouseUp`?

First, consider the click. At first sight you might think that to generate a click, a user has to press and release the left mouse button on the control. This is certainly true, but the situation is more complex. When the user clicks the left mouse button on a button component, the component is graphically pressed, too. However, if the user moves the cursor (holding down the left mouse button) outside the button surface, this button will be released. If the user now releases the left mouse button outside the button area, no effect — no click — takes place. On the other hand, if the user places the cursor back on the button, it will be *pressed* again, and when the mouse button is released, the click will occur. If this is not clear, experiment with a button; any button in any Windows application will do.

Now to the second question. In Windows, the behavior just described is typical of buttons, although Delphi has extended it to most components, as well as to forms. In any case, the system generates more basic events — one each time a mouse button is pressed, and another each time a button is released. In Delphi, these events are called `OnMouseDown` and `OnMouseUp`. Since the mouse has more than one button, these same events are generated when the user presses any mouse button.

You might want different actions to occur, depending on the mouse button. For this reason, these event handlers include a parameter indicating which button was pressed. These methods also include another parameter, indicating whether some special key (such as Shift or Ctrl) has been pressed and, finally, two more values indicating the *x* and *y* positions where the action took place. This is the method corresponding to this event for a form:

```
procedure TForm1.FormMouseDown (  
    Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);
```

Most of the time, we do not need such a detailed view, and handling the mouse-click event is probably more appropriate.

---

## Adding Colored Text to a Form

Now that you have played with buttons for a while, it's time to move to a new component, labels. Labels are just text, or comments, written in a form. Usually, the user doesn't interact with a label at all — or at least not directly. It doesn't make much sense to click on a label (although in Delphi this is technically possible). Keep in mind, however, that not all of the text you see in a form corresponds to a label. A form (and any other component) can simply output text on its surface, for example using the `TextOut` method.

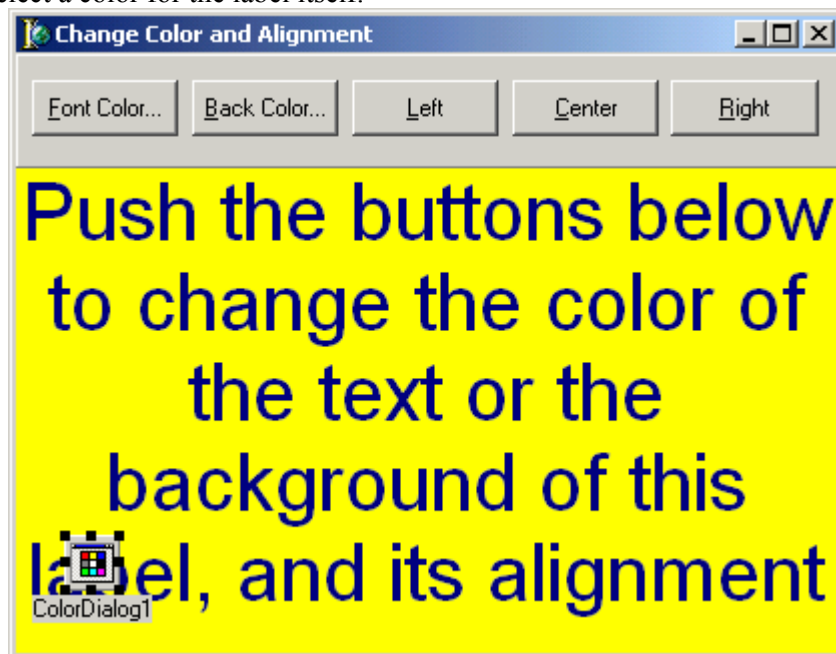
We use labels to provide descriptions of other components, particularly edit fields and list or combo boxes, because they have no title. If you open a dialog box in any Windows application, you'll probably see some text. These are *static controls* (in Windows terms) or *labels* (in Delphi terms).

Windows implements labels as windows of the static class. Delphi, instead, implements labels as non-windowed, graphical components. This is very important since it allows you to speed up form creation and save some Windows resources. However, Delphi also includes a new component that corresponds to the Windows label, the `StaticText` component. This component has similar properties and the same events, and Borland seems to have added it mainly for ActiveX support. We will use this component in the next example.

Besides using labels for descriptions, we can use instances of this component to improve and add some color to the user interface of our application. This is what we are going to do in the next example, `LabelCo`. The basic idea of this application is to test a couple of properties of the label component at run-time. Specifically, we want to alter the background color of the label, the color of its font, and the alignment of the text.

## The LabelCo Example

The first thing to do is to place a big label in the form and enter some text. Write something long. I suggest you set the `WordWrap` property to `True`, to have several lines of text, and the `AutoSize` property to `False`, to allow the label to be resized freely. It might also be a good idea to select a large font, to choose a color for the font, and to select a color for the label itself.



To change the font color, background color, and alignment properties of the label at run-time, we can use buttons. Instead of placing these buttons directly on the form, we can place a `Panel` component in the form, and then place the five buttons over (or actually inside) the panel. We need two to change the colors and three more to select the alignment — left, center, or right. Placing the buttons inside the panel, this last control will become the `Parent` component of the buttons: the coordinates of the buttons will be relative to the panel, so that moving the panel will move the buttons, hiding the Panel will hide the buttons, and so on.

The difference between the `Parent` and the `Owner` properties of a component is very important. The first indicates visual containment (like a button being inside a panel) while the latter indicates construction and destruction ownership (like a button being owned and destroyed by a form).

Making the label and the panel child components of the form allows us to align them (something you cannot do with buttons). Simply set the `Align` property of the Panel to `alTop`, and the `Align` property of the Label to `alClient`, and the two components will take up the full client area of the form. What's interesting is that when we resize the form, the two components will adjust their size and position correspondingly. Notice, by the way, that the `StaticText` component has no `Align` property.

The last component we have to place on the form is a `ColorDialog` component (you can find it in the Dialogs page of the Components palette). This component invokes the standard Windows Color dialog box. The resulting form is detailed in the following listing:

```

object ColorTextForm: TColorTextForm
  Caption = 'Change Color and Alignment'
  object Label1: TLabel
    Align = alClient
    Alignment = taCenter
    Caption = 'Push the buttons...' // omitted
    Color = clYellow
    Font.Color = clNavy
    Font.Height = -40
    Font.Name = 'Arial'
    WordWrap = True
  end
  object Panel1: TPanel
    Align = alTop
    object BtnFontColor: TButton...
    object BtnBackColor: TButton...
    object BtnLeft: TButton...
    object BtnCenter: TButton...
    object BtnRight: TButton...
  end
  object ColorDialog1: TColorDialog...
end

```

The two buttons used to change the color will display a dialog box, instead of performing an action directly. For this reason at the end of their `Caption` there is an ellipsis (...), which is the standard Windows convention for button and menu items to indicate the presence of a dialog box.

Now it's time to write some code. The click methods for the three alignment buttons are very simple. The program has to change the alignment of the label, as in:

```

procedure TColorTextForm.BtnLeftClick(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
end;

```

The other two methods should use the values `taCenter` and `taRightJustify` instead of `taLeftJustify`. You can find the names of these three choices in the `Alignment` property of the label, in the Object Inspector.

Writing code to change the color is a little more complex. In fact, we can provide a new value for the color, maybe choosing it from a list with a series of possible values. We might solve this problem, for example, by declaring an array of colors, entering a number of values, and then selecting a different element of the array each time. However, a more professional solution needs even less code: using the Windows standard dialog box to select a color.

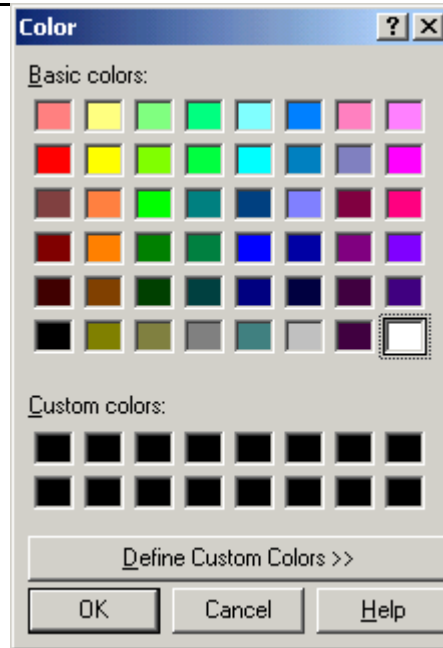
## The Standard Color Dialog Box

To use the standard Color dialog box, move to the Dialogs page of the Delphi Components palette, select the `ColorDialog` component, and place it anywhere on the form. The position has no effect, since at run-time this component is not visible inside the form. Now we can use the component, writing the following code:

```
procedure TColorTextForm.BtnFontColorClick(Sender: TObject);  
begin  
    ColorDialog1.Color := Label1.Font.Color;  
    if ColorDialog1.Execute then  
        Label1.Font.Color := ColorDialog1.Color;  
end;
```

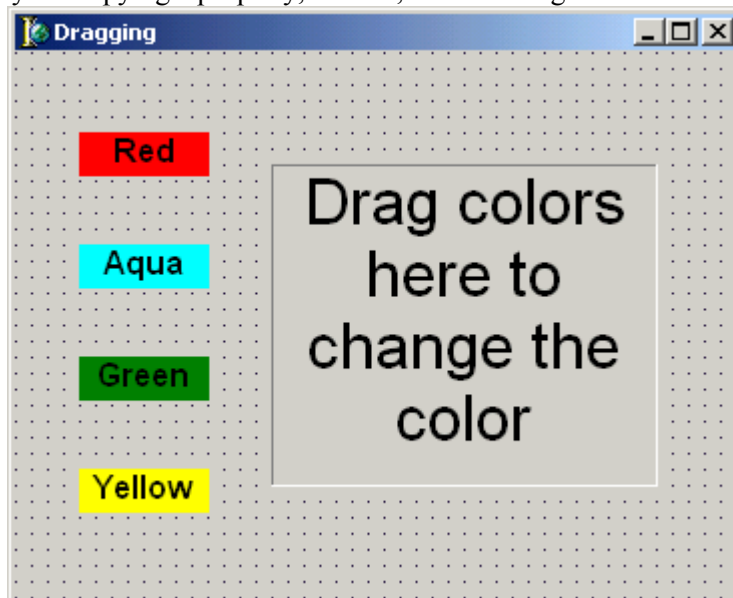
The three lines in the body of the procedure have the following meanings: with the first, we select the background color of the label as the initial color displayed by the dialog box; with the second, we run the dialog box; with the third, the color selected by the user in the dialog box is copied back to the label. We do this last operation only if the user closes the dialog box by pressing the OK button (in which case the `Execute` method returns `True`). If the user presses the Cancel button (and `Execute` returns `False`), we skip this last statement. To change the color of the label's text, we write similar code, referring this time to the `Label1.Font.Color` property. (The complete source code for the form `LabelCo` example is on the CD in the file `LABELF.PAS`.) You can see the Color dialog box in action in the figure below. This dialog box can also be expanded by clicking on the Define Custom Colors button.





## Dragging from One Component to Another

Before we try out other Delphi components, it's helpful to examine a particular technique: *dragging*. The dragging operation is quite simple and is increasingly common in Windows. In Windows you can drag files and programs from a folder to another, drop them on the desktop, or perform similar dragging operations on files and folders with the Windows Explorer. You usually perform this operation by pressing the mouse button on one component (or window) and releasing it on another component (or window). When this operation occurs, you can provide some code, usually for copying a property, a value, or something else to the destination component.



As an example, consider the form in the figure above. There are four color labels, with the name of each color as text, and a destination label, with some descriptive text. Actually the destination label is implemented with a `StaticText` component. This component has a special value for the `BorderStyle` property, `sbsSunken`, with a *lowered* effect. A similar capability is not available for plain labels. The aim of this example, named `Dragging`, is to be able to drag the color from one of the labels on the left to the static text, changing its color accordingly. The components have very simple properties, as the following textual description of the form summarizes:

```
object DraggingForm: TDraggingForm
  Caption = 'Dragging'
  Font.Color = clBlack
  Font.Height = -16
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  object LabelRed: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'Red'
    Color = clRed
    DragMode = dmAutomatic
  end
  object LabelAqua: TLabel...
  object LabelGreen: TLabel...
  object LabelYellow: TLabel...
  object StaticTarget: TStaticText
    Alignment = taCenter
    AutoSize = False
    BorderStyle = sbsSunken
    Caption = 'Drag colors here to change the color'
    Font.Height = -32
    OnDragDrop = StaticTargetDragDrop
    OnDragOver = StaticTargetDragOver
  end
end
```

After preparing the labels by supplying the proper values for the names and caption, as well as a corresponding color, you have to enable dragging. You can do this by selecting the value `dmAutomatic` for the `DragMode` property of the four labels on the left and responding to a couple of events in the destination label.

As an alternative to the automatic dragging mode, you might choose the manual dragging mode. This is based on the use of the `BeginDrag` and `EndDrag` methods. An alternative approach is to handle dragging manually, simply by providing a handler for events related to moving the mouse and pressing and releasing mouse buttons.

## The Code for the Dragging Example

The first event I want to consider is `OnDragOver`, which is called each time you are dragging and move the cursor over a component. This event indicates that the component accepts dragging. Usually, the event takes

place after a determination of whether the `Source` component (the one that originated the dragging operation) is of a specific type:

```
procedure TDraggingForm.StaticTargetDragOver(  
    Sender, Source: TObject; X, Y: Integer;  
    State: TDragState; var Accept: Boolean);  
begin  
    Accept := Source is TLabel;  
end;
```

This code accepts the dragging operation, activating the corresponding cursor, only if the `Source` object is really a `Label` component. Notice the use of the `is` dynamic type checking operator. The second method we have to write corresponds to the `OnDragDrop` event:

```
procedure TDraggingForm.StaticTargetDragDrop(  
    Sender, Source: TObject; X, Y: Integer);  
begin  
    StaticTarget.Color := (Source as TLabel).Color;  
end;
```

To read the value of the `Color` property from the `Source` object, we need to cast this object to the proper data type, in this case `TLabel`. We have to perform a type conversion — technically speaking, a type downcast (a typecast from a base class to a derived class, down through the hierarchy). A type downcast is not always safe. In fact, the idea behind this cast is that we receive the parameter `Source` of type `TObject`, which is really a label, and want to use it as a `TLabel` object, where `TLabel` is a class derived from `TObject`. However, in general, we face the risk of down-casting to `TLabel` an object that wasn't originally a label but, say, a button. When we start using the button as a label, we might have run-time errors.

In any case, when we use the `as` typecast, a type check is performed. Had the type of the `Source` object not been `TLabel`, an exception would have been raised. In this particular case, however, we haven't much to worry about. In fact, the `OnDragDrop` event is received only when the `Accept` parameter of the `OnDragOver` method is set to `True`, and we make this only if the `Source` object really is a `TLabel`.

---

## Accepting Input from the User

We have seen a number of ways a user can interact with the application we write using a mouse: mouse clicks, mouse dragging, and so on. What about the keyboard? We know that the user can use the keyboard instead of the mouse to select a button by pressing the key corresponding to the underlined letter of the caption (if any).

Aside from some particular cases, Windows can handle keyboard input directly. Defining handlers for keyboard-related events isn't a common operation, anyway. In fact, the system provides ready-to-use controls to build edit fields and a simple text editor. Delphi has several slightly different components in this area: `Edit`, `MaskEdit`, `Memo`, `RichText`, and the related data-aware controls. The two basic components are `Edit` and `Memo`.

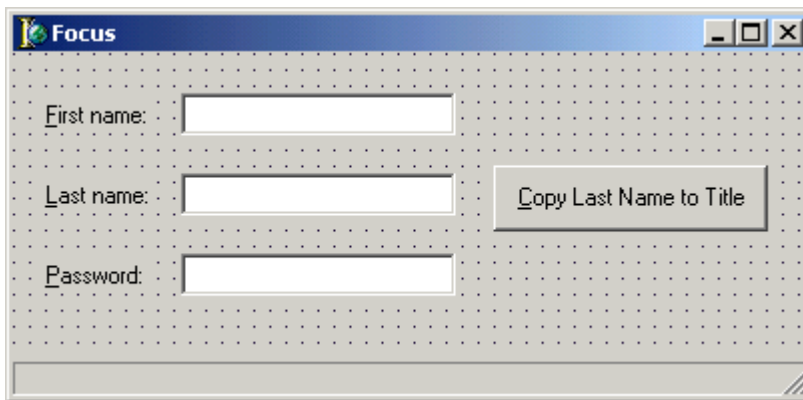
An `Edit` component allows a single line of text and has some specific properties, such as one that allows only a limited number of characters or one that shows a special password character instead of the actual text. A `Memo` component, as we will see in a while, can host several lines of text.

Our first example of the `Edit` component, named `Focus`, will demonstrate a feature common to many controls, the *input focus*. In Windows, it's fairly simple to determine which is the active main window: it is in front of the other windows, and the title bar is a different color. It is not as easy to determine which window (or component) has the input focus. If the user presses a key, which component is going to receive the corresponding

keyboard input message? It can be the active window, but it can also be one of its controls. Consider a form with several edit fields. Only one has the input focus at a given time. A user can move the input focus by using Tab or by clicking with the mouse on another component.

## Handling the Input Focus

What's important for our example is that each time a component receives or loses the input focus, it receives a corresponding event indicating that the user either has reached (`OnEnter`) or has left (`OnExit`) the component. So we can add some methods to the form to take control over the input focus and display this information in a label or a status bar.



Besides three edit boxes, the form has also some labels indicating the meaning of the three edit fields (*First name*, *Last name*, and *Password*). For the output of the status information I've used a specific Windows common control, the `StatusBar`, but using a label or a panel would have had a similar effect. In fact, you can use the `StatusBar` component as a single-line output tool, by setting its `SimplePanel` property to `True`. Here is a summary of the properties for this example:

```
object FocusForm: TFocusForm
  Caption = 'Focus'
  object Label1: TLabel
    Caption = '&First name:'
    FocusControl = EditFirstName
  end
  object Label2: TLabel
    Caption = '&Last name:'
    FocusControl = EditLastName
  end
  object Label3: TLabel
    Caption = '&Password:'
    FocusControl = EditPassword
  end
  object EditFirstName: TEdit
    TabOrder = 0
    OnEnter = EditFirstNameEnter
  end
  object EditLastName: TEdit
    TabOrder = 1
```

```
    OnEnter = EditLastNameEnter
end
object EditPassword: TEdit
    PasswordChar = '*'
    TabOrder = 2
    OnEnter = EditPasswordEnter
end
object ButtonCopy: TButton
    Caption = '&Copy Last Name to Title'
    TabOrder = 3
    OnClick = ButtonCopyClick
    OnEnter = ButtonCopyEnter
end
object StatusBar1: TStatusBar
    SimplePanel = True
end
end
```

As you can see, the form also contains a button we can use to copy the text of the `LastNameEdit` to the form's caption. This is just an example of how to work with text entered in an edit box. As you can see in the following code, before using the `Text` property of an edit box, it's a good idea to test whether the user has actually typed something or if the edit field is still empty:

```
procedure TFocusForm.ButtonCopyClick(Sender: TObject);
begin
    if EditLastName.Text <> '' then
        FocusForm.Caption := EditLastName.Text;
end;
```

Now we can move to the most interesting part of the program. We can write a comment in the status bar each time the focus is moved to a different control.

Displaying text in the status bar as the focus moves from control to control is a good way to guide the user through the steps of an application.

To accomplish this, we need four methods, one for each of the Edit components and one for the button, referring to the `OnEnter` event. Here is the code of one of the methods (the other three event handlers are very similar):

```
procedure TFocusForm.EditFirstNameEnter(Sender: TObject);
begin
    StatusBar1.SimpleText := 'Entering the first name...';
end;
```

You can test this program with the mouse or use the keyboard. If you press the `Tab` key, the input focus cycles among the Edit components and the button, without involving the labels. To have a proper sequence, you can change the `TabOrder` property of the windowed component. You can change this order either by entering a proper value for this property in the Object Inspector or (much better and easier) by using the Edit Tab Order dialog box, which can be called using the Tab Order command on the form's local menu. If you open this dialog box for the Focus example. Notice that the status bar is listed but you cannot actually move onto it using the `Tab` key.

A second way to select a component is to use a shortcut key. It is easy to place a shortcut key on the button, but how can you jump directly to an edit box? It isn't possible directly (the `Text` of the edit box changes

as a user types), but there is an indirect way. You can add the shortcut key — the ampersand (&) — to a label, then set the `FocusControl` property of the label to the corresponding Edit component.

Since Windows 95, the edit controls automatically have a local menu displayed when the user presses the right mouse button over them. Although you can easily customize such a menu in Delphi (as we will see in the next chapter), it is important to realize that this is standard behavior in the system.

---

## A Generic OnEnter Event Handler

The problem with this code is that we have to write four different `OnEnter` event handlers, copying four strings to the text of the `StatusBar` component. To add more edit boxes to the example, you would need to add more event handlers, copying the code over and over. And if you wanted to provide a slightly different output (for example, by changing the output of the `StatusBar` allowing for multiple panels), you would need to change the code many times.

The alternative solution is to write a single event handler for the `OnEnter` event of each edit box (and the button, too). We simply need to store the message for the status bar in a property, then refer to this property for the `Sender` object. A good technique is to use the `Hint` property, which is actually designed for providing descriptions to the user.

Simply store the proper messages in the `Hint` property of the edit boxes and of the button, then remove the current `OnEnter` event handler, and install this method for each of them:

```
procedure TForm1.GlobalEnter(Sender: TObject);
begin
    StatusBar1.SimpleText := (Sender as TControl).Hint;
end;
```

Notice you cannot write `Sender as TEdit` because the control might be a button as well. The solution is to typecast to a common ancestor class of `TButton` and `TEdit`, which defines the `Hint` property, as you can see in the code above.

---

## Entering Numbers

We saw in the previous example that it is very easy to use an Edit component to ask the user to input some text, although it must be limited to a single line. In general, it's quite common to ask users for numeric input, too. To accomplish this, you can use the `MaskEdit` component (in the Additional page of the Components palette) or simply use an Edit component and then convert the input string into an integer, using the standard Pascal `Val` procedure or the Delphi `IntToStr` function.

This sounds good, but what if the user types a letter when a number is expected? Of course, these conversion functions return an error code, so we can use it to test whether the user has really entered a number. The second question is, when can we perform this test? Maybe when the value of the edit box changes, when the component loses focus, or when the user clicks on a particular button, such as the OK button in a dialog box. As you'll see, not all of these techniques work well.

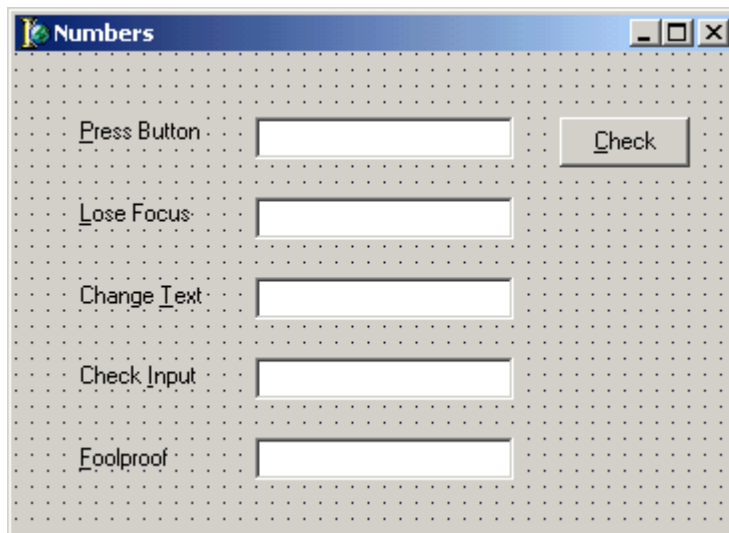
There is another, radically different, solution to the problem of allowing only numerical input in an edit box. You can look at the input stream to the edit box and stop any non-numerical input. This technique is not foolproof (a user can always paste some text into an edit box), but it works quite well and is easy to implement. Of course, you can improve it by combining it with one of the other techniques.

The next example, Numbers (see the form in the following figure), shows some of the techniques you can use to handle numerical input with an Edit component, so you can compare them easily. This example is meant as an exercise to discuss keyboard input and the input focus. To handle numerical input in an application you'll generally use specific components, as the SpinEdit or the UpDown controls available in Delphi. We will see an example of the use of another even more sophisticated control for keyboard input, the MaskEdit component.

In this example we're going to compare the effect of testing the input at different stages. First of all, build a form with five edit fields and five corresponding labels, describing in which occasion the corresponding Edit component checks the input. The form also has a button to check the contents of the first edit field. The contents of the first edit box are checked when the Check button is pressed. In the handler of the OnClick event of this button, the text is first converted into a number, using the Val procedure, which eventually returns an error code. Depending on the value of the code, a message is shown to the user:

```
procedure TNumbersForm.CheckButtonClick(Sender: TObject);  
var  
    Number, Code: Integer;  
begin  
    if Edit1.Text <> '' then  
        begin  
            Val (Edit1.Text, Number, Code);  
            if Code <> 0 then  
                begin  
                    Edit1.SetFocus;  
                    MessageDlg ('Not a number in the first edit',  
                                mtError, [mbOK], 0);  
                end  
            else  
                MessageDlg ('OK, the number in the first edit box is' +  
                            IntToStr (Number), mtInformation, [mbOK], 0);  
            end;  
        end;  
end;
```

If an error occurs, the application moves the focus back to the edit field before showing the error message to the user, thus inviting the user to correct the value. Of course, in this sample application a user can ignore this suggestion and move to another edit field.



The same kind of check is made on the second edit field when it loses the focus. In this case, the message is displayed automatically, but only if an error occurs. Why bother the user if everything is fine? The code here differs from that of the first edit field; it makes no reference to the `Edit2` component but always refers to the generic `Sender` control, making a safe typecast. To indicate the number of each button, I've used the `Tag` property, entering the number of the edit control.

As you can see in the following listing, instead of casting the `Sender` parameter to the `TEdit` class several times in the same method, it is better to do this operation once, saving the value in a local variable of the `TEdit` type. The type checking involved with these casts, in fact, is quite slow.

This method is a little more complex to write, but we will be able to use it again for a different component. Here is its code:

```
procedure TNumbersForm.Edit2Exit(Sender: TObject);  
var  
    Number, Code: Integer;  
    CurrEdit: TEdit;  
begin  
    CurrEdit := Sender as TEdit;  
    if CurrEdit.Text <> '' then  
        begin  
            Val(CurrEdit.Text, Number, Code);  
            if Code <> 0 then  
                begin  
                    CurrEdit.SetFocus;  
                    MessageDlg('The edit field number ' +  
                        IntToStr(CurrEdit.Tag) + ' does not have a valid number',  
                        mtError, [mbOK], 0);  
                end;  
            end;  
        end;  
end;
```



Since Delphi 2, this code produces a warning message (a hint) when compiled, because the `Number` variable is not used after a value has been assigned to it. To avoid these hints, you can ask the compiler to disable its generation inside a specific method using the `$HINTS` compiler directive. Simply write `{ $HINTS OFF }` before the method and `{ $HINTS ON }` after it, as I've done in the source code of this example.

The third Edit component makes a similar test each time its content changes (using the `OnChange` event). Although we have checked the input on different occasions — using different events — the three functions are very similar to each other. The idea is to check the string once the user has entered it.

For the fourth Edit component, I want to show you a completely different technique. We are going to make a check *before* the Edit even knows that a key has been pressed. The Edit component has an event, `OnKeyPress`, that corresponds to the action of the user. We can provide a method for this event and test whether the character is a number or the Backspace key (which has a numerical value of 8, so we can refer to it as the character `#8`). If not, we change the value of the key to the null character (`#0`), so that it won't be processed by the edit control, and produce a little warning sound:

```
procedure TNumbersForm.Edit4KeyPress(  
  Sender: TObject; var Key: Char);  
begin  
  // check if the key is a number or backspace  
  if not (Key in ['0'..'9', #8]) then  
    begin  
      Key := #0;  
      Beep;  
    end;  
end;
```

The fourth Edit component accepts only numbers for input, but it is not foolproof. A user can copy some text to the Clipboard and paste it into this Edit control with the Shift+Ins key combination (but not using Ctrl+V), avoiding any check. To solve this problem, we might think of adding a check for a change to the contents, as in the third edit field, or a check on the contents when the user leaves the edit field, as in the second component. This is the reason for the fifth, Foolproof edit field: it uses the `OnKeyPress` event of the fourth edit field, the `OnChange` method of the third, and the `OnExit` event of the second, thus requiring no new code.

To reuse an existing method for a new event, just select the Events page of the Object Inspector, move to the component, and instead of double-clicking to the left of the event name, select the button in the combo box at the right. A list of names of old methods compatible with the current event — having the same number of parameters — will be displayed.

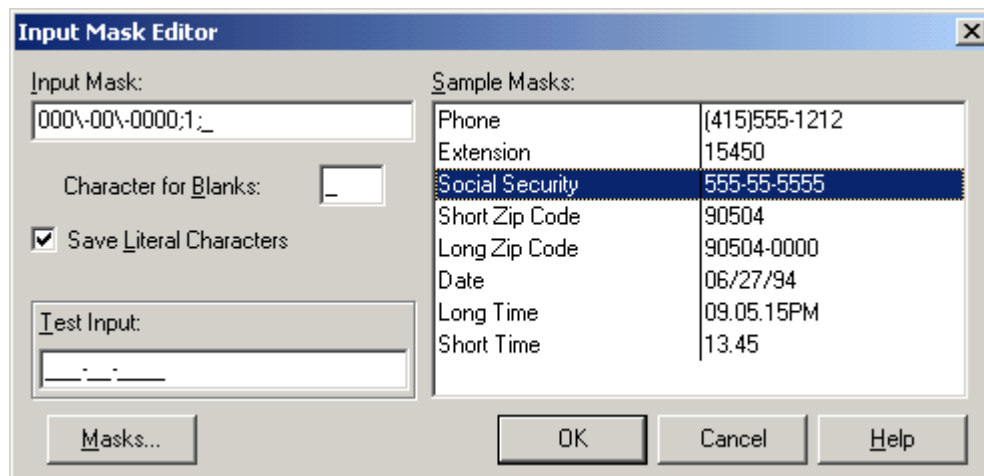
If you select the proper methods, the fifth component will combine the features of the third and the fourth. This is possible because in writing these methods, I took care to avoid any reference to the control to which they were related. The technique I used was to refer to the generic `Sender` parameter and cast it to the proper data type, which in this case was `TEdit`. As long as you connect a method of this kind to a component of the same kind, no problem should arise. Otherwise, you should make a number of type checks (using the `is` operator), which will probably make the code more complex to read. My suggestion is to share code only between controls of the same kind.

Notice also that to tell the user which edit box has incorrect text, I've added to each Edit component a value for the `Tag` property, as I mentioned before. Every edit box has a tag with its number, from 1 to 5.

## Sophisticated Input Schemes

In the last example, we saw how an Edit component can be customized for special input purposes. The components could really accept only numbers, but handling complex input schemes with a similar approach is not straightforward. For this reason, Borland has supplied a ready-to-use *masked edit component*, an edit component with an input mask stored in a string.

For example, to handle numbers of no more than five digits, we can set the `EditMask` property to `99999`. (The character 9 stands for *non-compulsory digit*; refer to the Delphi documentation for the meaning of the various characters and symbols in the edit mask.) I suggest that you don't enter a string directly in this property, but instead always open the associated editor by clicking on the small ellipses button. The Input Mask editor has a test window and includes sample masks for commonly used input values.



Notice that the Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to have the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated, by default, with semicolons).

To see more default input masks, you can press the Masks button, which allows you to open a mask file. The predefined files hold standard codes grouped by country. For example, if you open the Italian group, you can find the taxpayer number (or *fiscal code*, which is used like social security numbers in the U.S.). This code is a complex mix of letters and numbers (including the consonants representing name, birth date, area code, and more), as its mask demonstrates:

```
LLLLLL00L00L000L
```

In this kind of code, L stands for a letter and 0 for a number. While you can look these up in the Help file, there is a summary of these codes in the following Mask1 example.

The form of this example includes a MaskEdit and an Edit component. The Edit is used to change the `EditMask` property of the first one at run-time. To accomplish this, I've just written a couple of lines of code to copy the text of the property into the edit box at the beginning (the `OnCreate` event) and reverse the action each time the plain edit box changes (`Edit1Change`):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Edit1.Text := MaskEdit1.EditMask;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
    MaskEdit1.EditMask := Edit1.Text;
end;
```

The form also has a list box with the description of the most important codes used to build the mask.

---

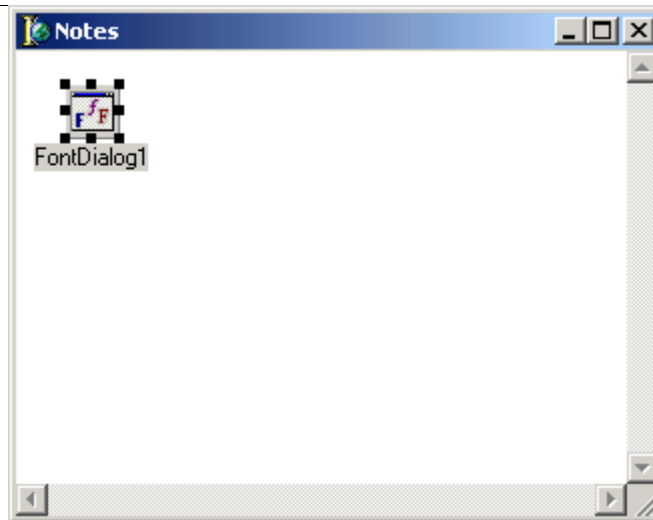
## Creating a Simple Editor

Edit components can handle a limited amount of text, and only on a single line. If you need to accept longer text input, you should use the Memo component. A Memo component is like an Edit component, but it can span several lines, contain scroll bars to move through the text, and contain more text.

The easiest way to use a Memo is as a text editor, as you'll see in the next example, Notes. The idea is to implement an editor covering all of the window (or form) which contains it, to resemble Windows Notepad. The only other feature we will implement is to give the user the option of choosing the font for the editor.

Both parts are very easy to implement. Create a new project and place a Memo component on the form. Delete its text, remove the border, and set the `Alignment` property to `alClient`, so that it will always cover the whole client area — the internal surface — of the form. Also add both scroll bars, horizontal and vertical, selecting the value `ssBoth` for the memo's `ScrollBars` property. Here is the summary (and the design time image of the form):

```
object NotesForm: TNotesForm
    Caption = 'Notes'
    object Memo1: TMemo
        Align = alClient
        BorderStyle = bsNone
        Font.Height = -19
        Font.Name = 'Times New Roman'
        ScrollBars = ssBoth
        OnDbClick = Memo1DbClick
    end
    object FontDialog1: TFontDialog...
```



## The Font Dialog Box

The second portion of the program involves the font. In the same way we used the standard Color dialog box in a previous example, we can use the standard Font selection dialog box provided by Windows. Just move to the Dialogs page of the Components palette and select the FontDialog component. Place it anywhere on the form, and add the following code when the user double-clicks inside the Memo:

```
procedure TNotesForm.Memo1Db1Click(Sender: TObject);  
begin  
    FontDialog1.Font := Memo1.Font;  
    if FontDialog1.Execute then  
        Memo1.Font := FontDialog1.Font;  
end;
```

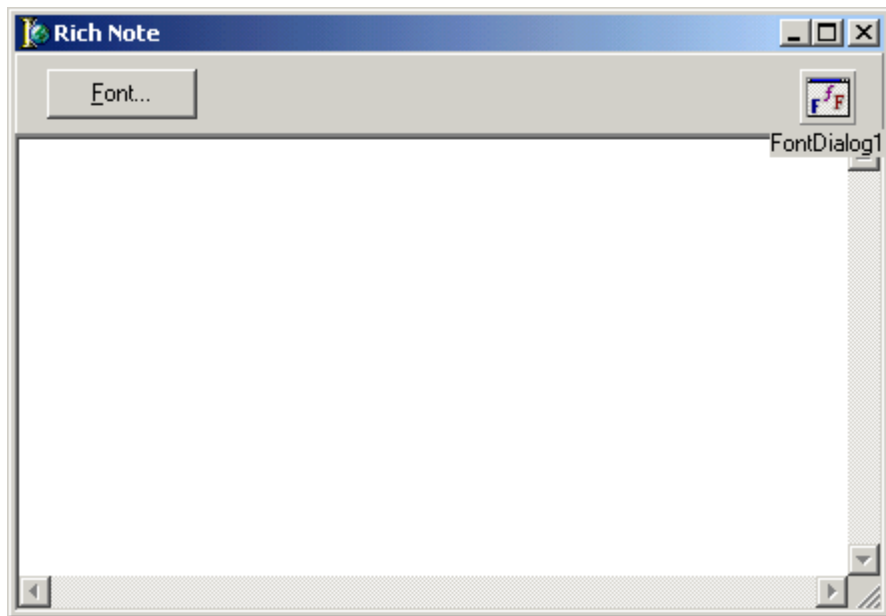
This code copies the current font to the corresponding property of the dialog component so it will be selected by default. Then it executes the dialog box. At the end, the `Font` property will contain the font the user selected. If the user presses the OK button, the third line of the above code copies the font back to the Memo.

This program is more powerful than it appears at first glance. For example, it allows copy and paste operations using the keyboard — this means you can copy text from your favorite word processor — and can handle the color of the font. Why not use it to place a big and colorful message on your screen?

## Creating a Rich Editor

Although you can choose a font in the Notes program, all of the text you have written will have the same font. Windows has a control that can handle the Rich Text Format (RTF). A Delphi component, RichEdit, encapsulates the behavior of this standard control.

You can find an example of a complete editor based on the RichEdit component among the examples that ship with Delphi. (The example is named RichEdit, too). Here, we'll only change the previous program slightly by replacing the Memo component with a RichEdit, and allow a user to change the font of the selected portion of the text, not the whole text.



The RichNote example has a RichEdit component filling its client area. However, the component has no double-click event, so I added a button to select the font and placed it in a panel aligned to the top of the form, making a very simple toolbar. Here is the textual description of some of the properties of the three components:

```
object RichEdit1: TRichEdit
  Align = alClient
  HideScrollBars = False
  ScrollBars = ssBoth
end
object Panel1: TPanel
  Align = alTop
  object Button1: TButton
    Caption = '&Font...'
  end
end
```

Notice the caption of the button, which has an ampersand for the shortcut key, and an ellipsis at the end to indicate that pressing it will open a dialog box. When the user clicks on the button, if some text is selected, the program shows the standard Font dialog box using the default font of the RichEdit component as the initial value. At the end, the selected font is copied to the attributes of the current selection. The `DefAttributes` and `SelAttributes` properties of the RichEdit component are not of the `TFont` type, but they are compatible, so we can use the `Assign` method to copy the value:

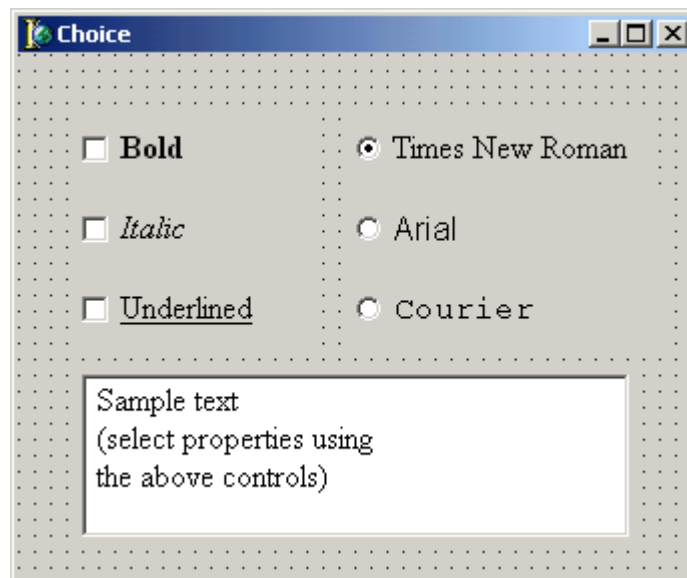
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign(RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign(FontDialog1.Font);
  end
  else
    ShowMessage ('No text selected');
end;
```

The RichEdit component has other attributes related to fonts and paragraph formatting. We will use this component in further examples of the book; however, the simple code above is enough to let users produce much more complex output than the Memo component allows.

## Making Choices

There are two standard Windows controls that allow the user to choose different options. The first is the *check box*, which corresponds to an option that can be selected freely (unless it has been disabled). The second control is the *radio button*, which corresponds to an exclusive selection. For example, if you see two radio buttons with the labels *A* and *B*, you can select *A* or select *B*, but not both of them at the same time. The other characteristic of a multiple choice is that you *must* check one of the radio buttons.

If the difference between check boxes and radio buttons is still not clear, an example might help you. The example has three check boxes to select the style *Bold*, *Italic*, or *Underlined*, and three radio buttons to choose a



font (*Times New Roman*, *Arial*, or *Courier*). There is also a memo field with some text to show the effect of the user selections immediately.

The difference between the use of the check boxes and the radio buttons should be obvious. The text might be bold and italic at the same time, but it cannot be Arial and Courier at once. A user must choose only one font (and cannot choose none) but can select each of the styles independently from the other two (including no style at all).

This program requires some simple code. Each time the user clicks on a check box or radio button, we have to create a corresponding action. For the text styles, we have to look at the `Check` property of the control and add or remove the corresponding element from the memo's `Font` property `Style` set:

```
procedure TForm1.CheckBoldClick(Sender: TObject);
begin
  if CheckBold.Checked then
    Memo1.Font.Style := Memo1.Font.Style + [fsBold]
  else
```

```
    Memo1.Font.Style := Memo1.Font.Style - [fsBold];  
end;
```

The other two check boxes have similar code for their `OnClick` events. The basic code for the radio buttons is even simpler since you cannot deselect a radio button by clicking on it:

```
procedure TForm1.RadioTimesClick(Sender: TObject);  
begin  
    Memo1.Font.Name := 'Times New Roman';  
end;
```

---

## Grouping Radio Buttons

Radio buttons represent exclusive choices. However, a form might contain several groups of radio buttons. Windows cannot determine by itself how the various radio buttons relate to each other. The solution, both in Windows and in Delphi, is to place the related radio buttons inside a container component. The standard Windows user interface uses a group box control to hold the radio buttons together, both functionally and visually. In Delphi, this control is implemented in the `GroupBox` component. However, Delphi has a second, similar component that can be used specifically for radio buttons: the `RadioGroup` component. A `RadioGroup` is a group box with some radio button inside it.

Using the radio group is probably easier than using the group box, but I'll use the more traditional approach to show you the code you can write to work with controls that have been placed inside another control. The `RadioGroup` component can automatically align its own internal radio buttons, and you can easily add new choices at run-time. You can see the differences between the two approaches in the next example.

The rules for building a group box with radio buttons are very simple. Place the `GroupBox` component in the form, then place the radio buttons in the group box. The `GroupBox` component contains other controls and is one of the container components used most often, together with the `Panel` component. If you disable or hide the group box, all the controls inside it will be disabled or hidden.

You can continue handling the individual radio buttons, but you might as well navigate through the array of controls owned by the group box. As discussed in the last chapter, the name of this property referring to this array of controls is `Controls`. Another property, `ControlCount`, holds the number of elements. These two properties can be accessed only at run-time.

---

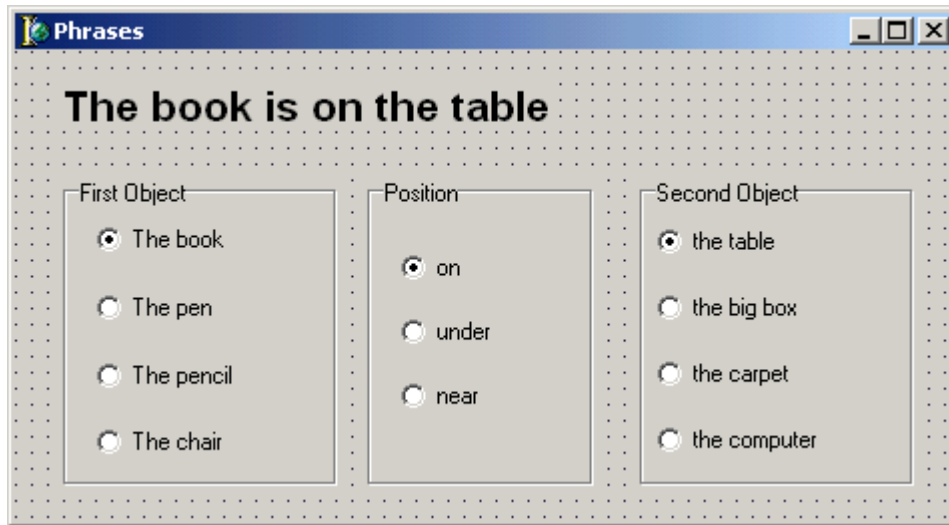
## The Phrases1 Example

If you've ever tried to learn a foreign language, you probably spent some time repeating the same silly and useless phrases over and over. Probably the most typical, when you learn English, is the infamous "*The book is on the table.*" To demonstrate radio buttons, the `Phrases1` example creates a tool to build such phrases by choosing among different available options.

This form is quite complex. If you rebuild it, remember that you must place the `GroupBox` components first and the radio buttons later. After doing this, you have to enter a proper caption for each element. The last selection is based on a radio group component, instead of a group box holding some radio buttons (as you can see in the textual description of the form below). In this case you create the options by entering a list of values in the `Items` property.

Remember that you also need to add a label, select a large font for it, and enter text corresponding to the radio buttons that are checked at design-time. This is an important point: When you place some radio buttons in a

form or in a group box, remember to check one of the elements at design-time. One radio button in each group should always be checked, and the `ItemIndex` property of the radio group, indicating the current selection, should have a proper value.



Here is the textual description of the form with a summary of this information:

```

object Form1: TForm1
  Caption = 'Phrases'
object Label1: TLabel
  Width = 243
  Caption = 'The book is on the table'
  Font.Height = -21
  Font.Name = 'Arial'
  Font.Style = [fsBold]
end
object GroupBox1: TGroupBox
  Caption = 'First Object'
object RadioBook: TRadioButton
  Caption = 'The book'
  Checked = True
  OnClick = ChangeText
end
object RadioPen: TRadioButton
  Caption = 'The pen'
  OnClick = ChangeText
end
object RadioPencil: TRadioButton...
object RadioChair: TRadioButton...
end
object GroupBox2: TGroupBox
  Caption = 'Position'
object RadioOn: TRadioButton
  Caption = 'on'
  Checked = True
  OnClick = ChangeText
end

```



```

    object RadioUnder: TRadioButton...
    object RadioNear: TRadioButton...
end
object RadioGroup1: TRadioGroup
  Caption = 'Second Object'
  Items.Strings = (
    'the table'
    'the big box'
    'the carpet'
    'the computer')
  OnClick = ChangeText
end
end
end

```

Now we have to write some code so that when the user clicks on the radio buttons, the phrase changes accordingly. There are different ways to do this. One is to follow the same approach as in the last example, providing a method for each button's `OnClick` event. Then we need to store the various portions of the phrase in some of the form's variables, change the portion corresponding to that button, and rebuild the whole phrase.

An alternative solution is to write a single method that looks at which buttons are currently checked and builds the corresponding phrase. This single method must be connected to the `OnClick` event of every radio button and of the `RadioGroup` component, a task we can easily accomplish. Select each of the radio buttons on the form (clicking on each one while you hold down the Shift key) and enter the name of the method in the Object Inspector. Since the method used to compute the new phrase doesn't refer to a specific control, you might name it yourself, simply entering a name in the second column of the Object Inspector next to the `OnClick` event. Here is the code of this single complex method:

```

procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: string;
  I: integer;
begin
  {look at which radio button is selected
  and add its text to the phrase}
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Phrase := (GroupBox1.Controls[I] as TRadioButton).Caption;

  {add the verb and blank spaces}
  Phrase := Phrase + ' is ';

  {repeat the operation on the second group box}
  for I := 0 to GroupBox2.ControlCount - 1 do
    with GroupBox2.Controls[I] as TRadioButton do
      if Checked then
        Phrase := Phrase + Caption;

  {retrieve the radio group selection, and display
  the result in the label}
  Label1.Caption := Phrase + ' ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex];
end;

```

The `ChangeText` method starts looking at which of the first group of radio buttons is selected, then moves on to adding a verb and the proper spaces between words. To determine which control in a group box is checked, the procedure scans these controls in a `for` loop. The `for` loop ranges from 0 to the number of controls

minus 1, because the `Controls` array is zero-based, and tests whether the `Checked` property of the radio button is `True`. A cast is required to perform this operation — we cannot use the `Checked` property on a generic control. When the checked radio button has been found, the program simply copies its caption to the string. At this point, the `for` loop might terminate, but since only one radio button is checked at a time, it is safe to let it reach its natural end — testing all the elements. The same operation is repeated two times, but you can see that the second time a `with` statement is used to make the code shorter and more readable.

As you can see from the final portion of the method above, if you are using the `RadioGroup` component, the code is much simpler. This control, in fact, has an `ItemIndex` property indicating which radio button is selected and an `Items` property with a list of the text of the fake radio buttons. Overall, using a radio group is very similar to using a list box (as we will see in the next example), aside from the obvious difference in the user interface of the two components.

---

## A List with Many Choices

If you want to add many selections, radio buttons are not appropriate, unless you create a really big form. The usual number of radio buttons is no more than 5 or 6. Another problem is that although you can disable a radio button, the elements of a group are usually fixed. Only when using a radio group can you have some flexibility. For both of these problems, the solution is to use a list box. A list box can host a large number of choices in a small space, because it can contain a scroll bar to show on screen only a limited portion of the whole list. Another advantage of a list box is that you can easily add new items to it or remove some of the current items. List boxes are extremely flexible and powerful.

Another important feature is that by using the `Listbox` component, you can choose between allowing only a single selection, a behavior similar to a group of radio buttons, and allowing multiple selections, which is similar to a group of check boxes. The next version of this example will have a multiple-selection list box.

For the moment, let's focus on a single-selection list box. We might use a couple of these components to change the `Phrases1` example slightly. Instead of having a number of radio buttons to select the first and second objects of the phrase, we can use two list boxes. Besides allowing us to have a larger number of items, the advantage is that we can allow the user to insert new objects in the list and prevent selection of the same object twice, to avoid a phrase such as "The book is on the book." As you might imagine, this example is really much more complicated than the previous one and will require some fairly complex code.

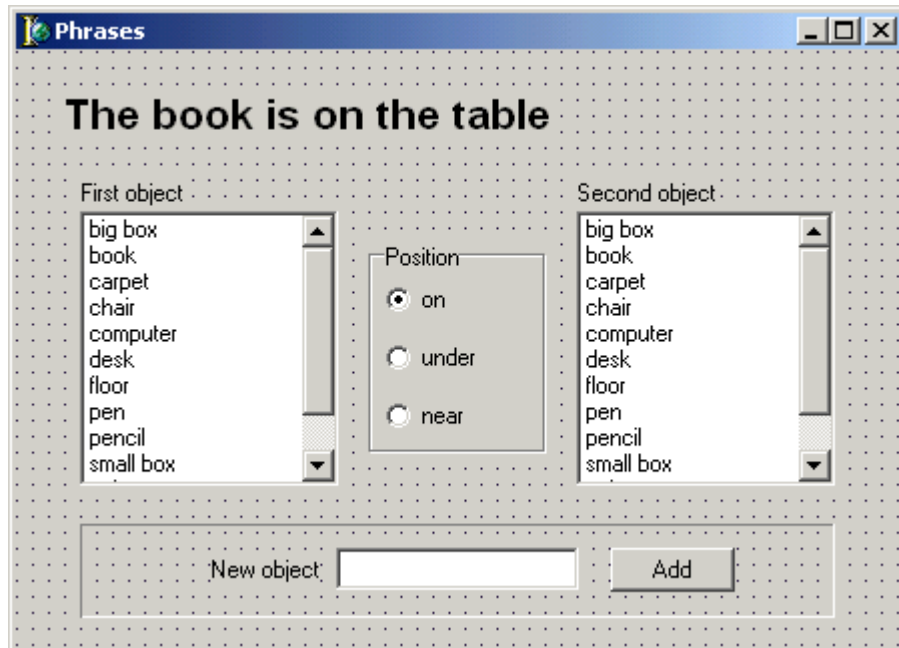
---

## The Form of the Phrases2 Example

As usual, the first step is to build a form. You can start with the form from the last example and remove the two group boxes on the sides and replace them with two list boxes. The radio buttons inside the group boxes will be deleted automatically. I've also replaced the central group box with a radio group. Actually, there's not much left from the previous example!

Now, add some strings to the `Items` property of both list boxes. For the example to work properly, the two list boxes should have the same strings; you can copy and paste them from the editor of the `Items` property

of one list box to the editor of the same property of the other component. To improve the usability of the program, you might sort the strings in the list boxes, setting their `Sorted` property to `True`. Remember also to add a couple of labels above the list boxes, to describe their contents.



In the lower part of the form, I've also added an edit field, with its label, and a button, and a bevel around them to group them visually (the bevel is just a graphical component, not a container). As we will see later, when a user presses the button, the text in the Edit control is added to both list boxes. This operation will take place only if the text of the edit box is not empty and the string is not already present in the list boxes.

Here is the textual description of the components of this updated form (which is really very different from the previous version):

```

object Form1: TForm1
  Caption = 'Phrases'
  OnCreate = FormCreate
object Label1: TLabel... // as in Phrases1
object Label2: TLabel
  Caption = 'First object'
end
object Label3: TLabel
  Caption = 'Second object'
end
object ListBox1: TListBox
  Items.Strings = (
    'big box'
    'book'
    'carpet'
    'chair'
    'computer'...)
  Sorted = True
  OnClick = ChangeText

```

```
end
object RadioGroup1: TRadioGroup
  Caption = 'Position'
  Items.Strings = (
    'on'
    'under'
    'near')
end
object ListBox2: TListBox... // identical to ListBox1
object Bevel1: TBevel...
object Label4: TLabel
  Caption = 'New object'
end
object EditNew: TEdit...
object ButtonAdd: TButton
  Caption = 'Add'
  OnClick = ButtonAddClick
end
end
```

---

## Working with the List Boxes

Once you have built this or a similar form, you can start writing some code. The first thing to do is to provide a new `ChangeText` procedure, connected with the `OnClick` event of the radio group and of the two list boxes. This procedure is simpler than in the previous example. In fact, to retrieve the selected text from the list box, you only need to get the number of the item selected (stored in the run-time property `ItemIndex`) and then retrieve the string at the corresponding position of the `Item` array, as the `Phrases1` program did.

Here is what the code for the procedure looks like initially (this is just a temporary version, different from the one in the source code):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: String;
begin
  Phrase := 'The ';
  Phrase := Phrase + ListBox1.Items [ListBox1.ItemIndex];
  Phrase := Phrase + ' is ';
  Phrase := Phrase + RadioGroup1.Items [RadioGroup1.ItemIndex];
  Phrase := Phrase + ' the ';
  Phrase := Phrase + ListBox2.Items [ListBox2.ItemIndex];
  Label1.Caption := Phrase;
end;
```

This program, however, won't work properly because, at the beginning, no item is selected in either list box. To solve this problem, we can add some code to the form's `OnCreate` event. In this code, we can look for the two default strings, book and table, and select them. You should do this operation in two steps. First, you need to look for the string's index in the array of strings, with the `IndexOf` method. Then you can use that value as the index of the currently selected item:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
```

```
N := ListBox1.Items.IndexOf ('book');
ListBox1.ItemIndex := N;
N := ListBox2.Items.IndexOf ('table');
ListBox2.ItemIndex := N;
end;
```

## Removing a Selected String from the Other List Box

Once this part of the program works, we have two more problems to solve: We must remove the selected string from the other list box (to avoid using the same term twice in a phrase), and we must write the code for the click event on the button.

The first problem is more complex, but I'll address it immediately since the solution of the second problem will be based partially on the code we write for the first one. Our aim is to delete from a list box the item currently selected in the other list box. This is easy to code. The problem is that once the selection changes, we have to restore the previous items, or our list boxes will rapidly become empty. A good solution is to store the two currently selected strings for the two list boxes in two private fields of the form, `String1` and `String2`:

```
type
  TForm1 = class(TForm)
    ...
  private
    String1, String2: String;
  end;
```

Now we have to change the code executed at startup and the code executed each time a new selection is made. In the `FormCreate` method, we need to store the initial value of the two strings and remove them from the other list box; the first string should be removed from the second list box, and vice versa. Since the `Delete` method of the `TStrings` class requires the index, we have to use the `IndexOf` function again to determine it:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
  String1 := 'book';
  String2 := 'table';

  {delete the selected string from the other list box
  to avoid a double selection}
  ListBox2.Items.Delete (ListBox2.Items.IndexOf (String1));
  ListBox1.Items.Delete (ListBox1.Items.IndexOf (String2));

  {select the two strings in their respective list boxes}
  N := ListBox1.Items.IndexOf (String1);
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf (String2);
  ListBox2.ItemIndex := N;
end;
```

The code to select the string should be executed after calling `Delete`, because removing an element before the one currently selected will alter the selection. The fact is that the selection is just a number referring to a string, not the reverse, as it should probably be. By the way, this doesn't depend on Delphi implementation but on the behavior of list boxes in Windows.

Things get complicated when a new item is selected in one of the list boxes. The `ChangeText` procedure has some new code at the beginning, executed only if the click took place on one of the list boxes (remember that the code is also associated with the group box). For each string, we have to check whether the selected item has changed and, in this case, add the previously selected string to the other list box and delete the new string. Here is the new version of the `ChangeText` method:

```
procedure TForm1.ChangeText(Sender: TObject);  
var  
    TmpStr: String;  
begin  
    // if a list box has changed  
    if Sender is TListBox then  
    begin  
        // get the text of the first string  
        TmpStr := ListBox1.Items [ListBox1.ItemIndex];  
        // if the first one has changed  
        if TmpStr <> String1 then  
        begin  
            // update the strings in ListBox2  
            {1.} ListBox2.Items.Add (String1);  
            {2.} ListBox2.Items.Delete (  
                ListBox2.Items.IndexOf (TmpStr));  
            {3.} ListBox2.ItemIndex :=  
                ListBox2.Items.IndexOf (String2);  
            {4.} String1 := TmpStr;  
        end;  
  
        // get the text of the second string  
        TmpStr := ListBox2.Items [ListBox2.ItemIndex];  
        // if the second one has changed  
        if TmpStr <> String2 then  
        begin  
            // update the strings in ListBox1  
            ListBox1.Items.Add (String2);  
            ListBox1.Items.Delete (ListBox1.Items.IndexOf (TmpStr));  
            ListBox1.ItemIndex := ListBox1.Items.IndexOf (String1);  
            String2 := TmpStr;  
        end;  
    end;  
  
    // build the phrase with the current strings  
    Label1.Caption := 'The ' + String1 + ' is ' +  
        RadioGroup1.Items [RadioGroup1.ItemIndex] +  
        ' the ' + String2;  
end;
```

What is the effect of the first part of this code? Here is a detailed description of the operations, referring to a new selection in the first list box. The procedure stores the selected element of the first list box in the

temporary string `TmpStr`. If this is different from the older selection, `String1`, four operations take place (refer to the numbers in the listing above):

1. The previously selected string, `String1`, is added to the other list box, `ListBox2`.
2. The new selection, `TmpStr`, is removed from the other list box.
3. The selected string of the other list box, `String2`, is reselected in case its position has been changed by the two preceding operations.
4. Once the two lists contain the correct elements, we can store the new value in `String1` and use it later on to build the phrase.

We perform the same steps for the other list box a few lines later. Notice that we don't need to access the list boxes again to build the phrase at the end of the `OnChange` method, since `String1` and `String2` already contain the values we need.

Implementing the `OnClick` event for the Add button is quite simple. The only precautions we have to take are to test whether there is actually some text in the edit box or if it is empty and to check whether the string is already present in one of the two list boxes. Checking only one of the list boxes will miss a correspondence between the text of the edit box and the item currently selected in the other list box.

To make this check, we can ask both `Listbox` components for the index of the new string; if it is not present in the list — if there is no match — the value `-1` will be returned. Otherwise, the `IndexOf` function returns the correct index, starting with `0` for the first element. In technical terms, we can say that the function returns the zero-based index of the element, or the error code `-1` if it is not found. Here is the code:

```

procedure TForm1.ButtonAddClick(Sender: TObject);
begin
    {if there is a string in the edit control and
     the string is not already present in one of the lists}
    if (EditNew.Text <> '') and
        (ListBox1.Items.IndexOf(EditNew.Text) < 0) and
        (ListBox2.Items.IndexOf(EditNew.Text) < 0) then
        begin
            {add the string to both list boxes}
            ListBox1.Items.Add (EditNew.Text);
            ListBox2.Items.Add (EditNew.Text);

            {reselects the current items properly}
            ListBox1.ItemIndex := ListBox1.Items.IndexOf (String1);
            ListBox2.ItemIndex := ListBox2.Items.IndexOf (String2);
        end
    else
        MessageDlg ('The edit control is empty or contains'
            + ' a string which is already present',
            mtError, [mbOK], 0);
    end;

```

In the final part of this method's code, we need to reselect the current item of each list box since the position of the selected item might change. This happens if the new item is inserted before the one that is currently selected — that is, if it has a lower sort order.

## Allowing Multiple Selections

A list box can allow the selection of either a single element or a number of elements. We make this choice in setting up a list box by specifying the value of its `Multiple` property. As the name implies, setting `Multiple` to `True` allows multiple selections. There are really two different kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case a user selects multiple items simply by clicking on them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items. This second choice is determined by the `ExtendedSelect` property.

While setting up a multiple-selection list box is very simple, the problems start to appear when you have to write the code. Accessing the selected item of a single-selection list box is simple. The `ItemIndex` property holds the index of the selected item, and the selected string can be retrieved with a simple expression:

```
ListBox2.Items[ListBox2.ItemIndex];
```

In a multiple-selection list box, on the other hand, we do not know how many items are selected, or even whether there is any item selected. In fact, a user can click on an item to select it, drag the cursor to select a number of consecutive items in the list, or click the mouse button on an item while holding down Ctrl key to toggle the selection of a single item without affecting the others. Using this last option, a user can even deselect all the items in a list box.

A program can retrieve information on the currently selected items by examining the `Selected` array. This array of `Boolean` values has the same number of entries as the list box. Each entry indicates whether the corresponding item is selected.

For example, to know how many items are selected in a list box, we need to scan the `Selected` array, usually with a `for` loop ranging from 0 to the number of items in the list minus one:

```
SelectCount := 0;
for ListItem := 0 to ListBox1.Items.Count - 1 do
  if ListBox1.Selected[ListItem] then
    Inc (SelectCount);
```

Actually, the `Listbox` component has an undocumented `SelCount` property, you can use to obtain exactly the information computed in the code above. We won't use this code in the next example, anyway, but a more complex version.

## The Third Version of the Phrases Example

With this information, we can build a new version of the Phrases example, allowing a user to select several items in the first list box. The only real difference between the form of this new version and that of the last one is the value of the `MultiSelect` property in the first list box is not set to `True`.

In addition, the label at the top of the form has been enlarged, enabling the `WordWrap` property and disabling the `AutoSize` property, to accommodate longer phrases. Since the example's code is complex enough, I've removed the portion used to delete from a list box the item selected in the other list box. In the case of multiple selections, this would have been really complicated.

The main problem we face is building the different phrases correctly. The basic idea is to scan the `Selected` array each time and add each of the selected objects to the phrase. However, we need to place an "and" before the name of the last object, omitting the comma if there are only two. Moreover, we need to decide



between singular and plural (*is* or *are*) and provide some default text if no element is selected. As you can see from the table below, building these phrases is not simple. In fact, if we store the phrase “*The book and the computer,*” when we need to add a third item, we must go back and change it.

Items Selected	SelectCount	Phrase
{none}	0	Nothing is
book	1	The book is
book, computer	2	The book and the computer are
book, computer, pen	3	The book, the computer, and the pen are
book, computer, pen, small box	4	The book, the computer, the pen, and the small box are

An alternative idea is to create two different phrases, one valid if no other elements will be added, the other prepared to host future objects (without the *and*). In the code, the `TmpStr1` string is the tentative final statement, while `TmpStr2` is the temporary string used to add a further element. At the end of the loop, `TmpStr1` holds the correct value. As you can see in the code below, in case only two items are selected we have to build the phrase in a slightly different way, removing the comma before the *and* conjunction.

Notice when scanning a sorted list box that the objects are always added to the resulting string in alphabetical order, not in the order in which they were selected. You can study how this idea has been implemented by looking at the new version of the `ChangeText` method, in the following code (and by studying the following table, which describes step-by-step how the strings are built):

```

procedure TForm1.ChangeText(Sender: TObject);
var
    Phrase, TmpStr1, TmpStr2: String;
    SelectCount, ListItem: Integer;
begin
    SelectCount := 0;

    {look at each item of the multiple selection list box}
    for ListItem := 0 to ListBox1.Items.Count - 1 do
        if ListBox1.Selected [ListItem] then
            begin
                {if the item is selected increase the count}
                Inc (SelectCount);
                if SelectCount = 1 then
                    begin
                        {store the string of the first selection}
                        TmpStr1 := ListBox1.Items.Strings [ListItem];
                        TmpStr2 := TmpStr1;
                    end
                else if SelectCount = 2 then
                    begin
                        {add the string of the second selection}
                        TmpStr1 := TmpStr1 + ' and the ' +
                            ListBox1.Items.Strings [ListItem];
                        TmpStr2 := TmpStr2 + ', the ' +
                            ListBox1.Items.Strings [ListItem];
                    end
                else // SelectCount > 2
                    begin

```

```

        {add the string of the further selection}
        TmpStr1 := TmpStr2 + ', and the ' +
            ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr2 + ', the ' +
            ListBox1.Items.Strings [ListItem];
    end;
end;

{build the first part of the phrase}
if SelectCount > 0 then
    Phrase := 'The ' + TmpStr1
else
    Phrase := 'Nothing';

if SelectCount <= 1 then
    Phrase := Phrase + ' is '
else
    Phrase := Phrase + ' are ';

{add the text of the radio button}
Phrase := Phrase +
    RadioGroup1.Items [RadioGroup1.ItemIndex];

{add the text of the second list box}
Phrase := Phrase + ' the ' +
    ListBox2.Items [ListBox2.ItemIndex];
Label1.Caption := Phrase;
end;

```

Items Selected	Steps	TmpStr1 (Tentative Final Statement)	TmpStr2 (Temporary Statement)
book	1	book	book
+ computer	2	book and the computer	book, the computer
+ pen	3	book, the computer, and the pen	book, the computer, the pen
+ small box	4	book, the computer, the pen, and the small box	book, the computer, the pen, the small box

The other procedures of the program change only slightly. The `FormCreate` method is simplified because we do not need to delete the selected item from the other list box. The `Add` method is simplified because both list boxes always have the same items and because the multiple-selection list box creates no problems with the selection if you add a new element.

An alternative solution to handle the status of multiple-selection list boxes is to look at the value of the `ItemIndex` property, which holds the number of the item of the list having the focus. If a user clicks on several items while holding down `Ctrl`, each time a click event takes place, you know which of the items have been selected or deselected — you can easily determine which of the two operations took place by looking at the value of the `Selected` array for that index. The problem is that if the user selects a number of elements by dragging the mouse, this method won't work. You need to intercept the dragging events, and this is considerably more complex than the technique described earlier.

## Using a CheckListBox Component

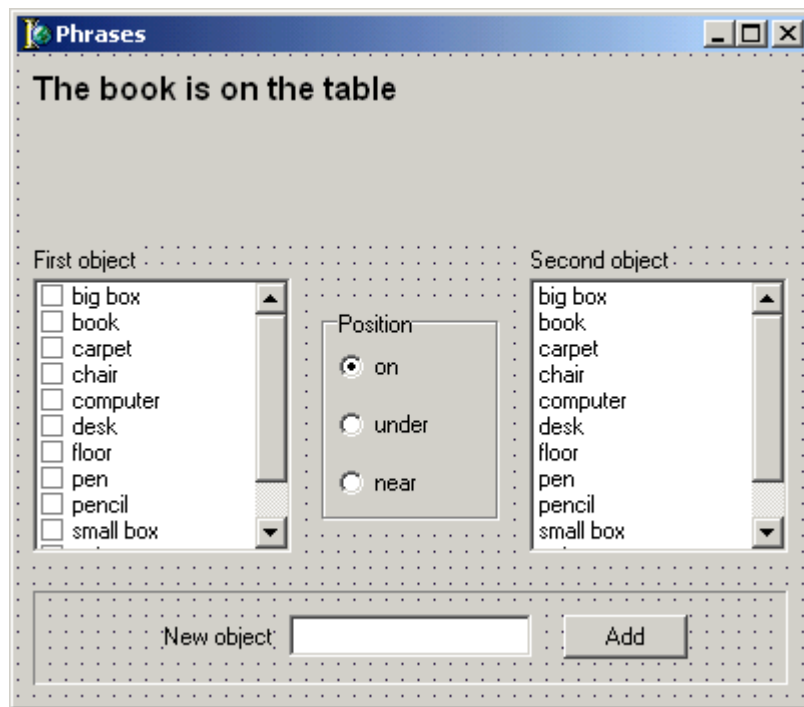
A further extension to the Phrases example is the use of the CheckListBox component, a component originally introduced by Borland in Delphi 3. This is basically a list box with a custom output (or an *owner-draw* list box, to use the proper technical term). Each item of the list is preceded by a check box. A user can select a single item of the list, but can also click on the check boxes to toggle their status.

If the component has the AllowGrayed property set to True, then each check box can be non-selected, grayed, or selected. Clicking on the check box alternates these three possible conditions. To check the current status of each item you can use the Checked and the State property. Both are array properties. The first, Checked, is a Boolean property you should use when AllowGrayed is set to False. The second, State, is a property of the TCheckBoxState data type:

```
type
```

```
TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed)
```

This property should be used when AllowGrayed is set to True, to distinguish among the three different states of each item. Apart from these properties, the specific user interface, and the new OnClickCheck event, this component behaves as a ListBox.



As I mentioned at the beginning of this section, to show you an example of the use of this component I've further updated the Phrases3 example, building the Phrases4 version. I've basically replaced the first multiple selection list box with the new CheckListBox component, set its Sorted property to True, and copied the Items. Then I've updated the code, replacing the ListBox1 object with the CheckListBox1 object, and replacing the Selected property of the first with the Checked property of the second in the ChangeText method. Here is an excerpt of the new version of this method:

```
for ListItem := 0 to CheckListBox1.Items.Count - 1 do
```

```
if CheckListBox1.Checked [ListItem] then
begin
  {if the item is selected increase the count}
  Inc (SelectCount);
  if SelectCount = 1 then
  begin
    {store the string of the first selection}
    TmpStr1 := CheckListBox1.Items.Strings [ListItem];
    TmpStr2 := TmpStr1;
  end
  else if SelectCount = 2 then
    ...
```

The important point of this program is that this component makes it more obvious to the user that the list box allows multiple selections. The plain list box, in fact, gives no clue of this fact.

---

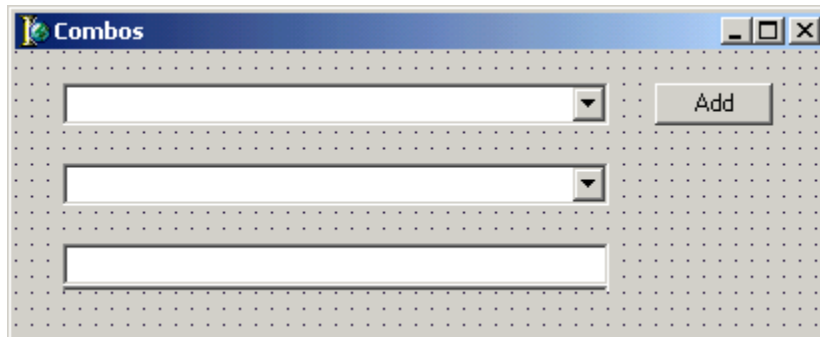
## Many Lists, Little Space

List boxes take up a lot of screen space, and they offer a fixed selection. That is, a user can choose only among the items in the list box and cannot make any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control. A combo box is similar to an edit box, and you can often enter some text in it. It is also similar to a list box, with a drop-down arrow that displays a list box. Even the name of the control suggests that it is a combination of two other controls, an Edit and a ListBox. However, the behavior of a ComboBox component might change a lot, depending on the value of its `Style` property. Here is a short description of the various styles:

- The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request.
- The `csDropDownList` style defines a combo box that does not allow editing. By pressing a key, the user selects the first word starting with that letter in the list.
- The `csSimple` style defines a combo box that always displays the list box below it. This version of the control allows direct editing.
- The `csOwnerDrawFixed` and `csOwnerDrawVariable` styles define combo boxes based on an owner-draw list — that is, a list containing graphics determined by the program rather than simple strings.

To see the difference between the first three types, you can run the Combos example, which I'll describe in a moment. As you can appreciate by testing the program, Combos displays three combo boxes having three different styles: drop-down, drop-down list, and simple.



This program is very simple. Each combo box has the same basic strings—the names of more than 20 different animals. The first combo box contains an *Add* button. If the user presses the button, any text entered in the combo box is added to its list, provided it is not already present. This is the code associated with the `OnClick` event of the button:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  with ComboBox1 do
    if (Text <> '') and (Items.IndexOf (Text) < 0) then
      Items.Add (Text);
end;
```

You can use the second combo box to experiment with the automatic lookup technique. If you press a key, the first of the names in the list starting with that letter will be selected. By pressing the up arrow key and the down arrow key, you can further navigate in the list without opening it. This navigation technique of using initial letters and arrows can be used with each of the combo boxes.

The third combo box is a variation of the first. Instead of adding the new element when the *Add* button is pressed, that action is performed when the user presses the Enter key. To test for this event, we can write a method for the combo box's `OnKeyPress` event and check whether the key is the Enter key, which has the numeric code 13. The remaining statements are similar to those of the button's `OnClick` event:

```
procedure TForm1.ComboBox3KeyPress(
  Sender: TObject; var Key: Char);
begin
  {if the user presses the Enter key}
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

Delphi's `DateTimePicker` component has a user interface similar to that of a `ComboBox`.

## Choosing a Value in a Range

The last basic component I want to explore in this chapter is the scroll bar. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. Notice, however, that when a scroll bar is associated with another component, it is really a portion of that component — one of its properties — and there is little relationship to the `ScrollBar` component itself. Forms having a scroll bar have no `ScrollBar` component. A portion of their border is used to display that graphical element.

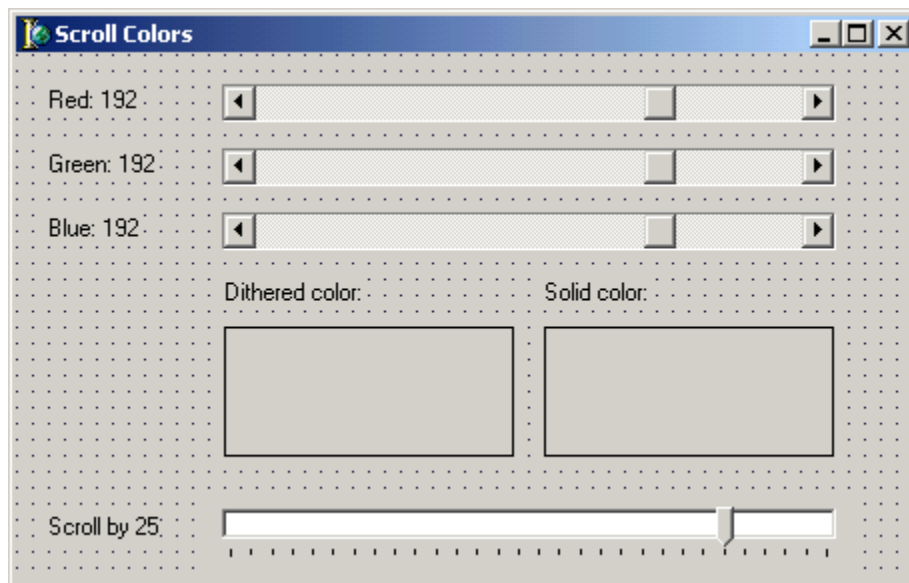
Direct usage of the `ScrollBar` component is quite rare, especially with the `TrackBar` Windows common control. However, there are cases in which it can play a role. The typical example is to allow a user to choose a numerical value in a large range (since a `TrackBar` is generally used for smaller ranges).

Most Windows programming books describe scroll bars using the example of selecting a color, and this book is no exception. But if you've seen a typical Windows example, you'll notice something very interesting: using Delphi, you can build this example in about one-fourth the time and writing a minimal amount of code.

### The Scroll Color Example

The `ScrollC` example — the name stands for scroll color — has a simple form with three scroll bars and three corresponding labels, a track bar with its own label, and some shape components to show the current color. Each scroll bar refers to one of the three fundamental colors, which in Windows are red, green, and blue (RGB). Each label displays the name of the corresponding color and the current value.

Scroll bars have a number of peculiar properties. You can use `Min` and `Max` to determine the range of possible values; `Position` holds the current position; and the `LargeChange` and `SmallChange` properties indicate the increment caused by clicking on the bar or on the arrow at the end of the bar, respectively.



In the ScrollC example, the value of each bar ranges from 0 to 255. The range is determined by the fact that each color is a DWORD with the lower three bytes representing the Red, Green, and Blue values (which is how colors are represented in Windows and in the VCL). The initial value of 192 has been chosen for the position because with settings of 192 for red, 192 for green, and 192 for blue, you get the typical light gray, which is the default value for the color of the form and of the shapes. Here is the textual description of one of these three ScrollBar components:

```
object ScrollBarRed: TScrollBar
  LargeChange = 25
  Max = 255
  Position = 192
  OnScroll = ScrollBarRedScroll
end
```

The TrackBar component has similar properties (Min, Max, Position):

```
object TrackBar1: TTrackBar
  Max = 30
  Min = 1
  Orientation = trHorizontal
  Frequency = 1
  Position = 25
  TickMarks = tmBottomRight
  TickStyle = tsAuto
  OnChange = TrackBar1Change
end
```

This control is used, in this example, to set the LargeChange property of the three scrollbars, with the following code:

```
procedure TFormScroll.TrackBar1Change(Sender: TObject);
begin
  LabelScroll.Caption := 'Scroll by ' +
    IntToStr(TrackBar1.Position);
  ScrollBarGreen.LargeChange := TrackBar1.Position;
  ScrollBarRed.LargeChange := TrackBar1.Position;
  ScrollBarBlue.LargeChange := TrackBar1.Position;
end;
```

When one of the scroll bars changes (the OnScroll event), the program has to update the corresponding label and the color of the shapes. The first of these shapes is used to show the color as it is determined by the three RGB values of the scroll bars. Assigning the color to the Color property of the brush used to fill the surface of the shape, we obtain a dithered color, an approximation of the real tint made with the colors available on the video adapter. The same color is assigned to the Color property of the pen of the second shape, resulting in the closest approximation of the requested color. Pens, in fact, do not use dithering, but rather the closest *pure* color. You can see the difference by running this example, although the effect might change depending on your video adapter.

If you browse through the code of the program, notice also that there is a third shape component used to mimic the border of the second shape. The real border of this shape, in fact, is enlarged to fill its whole surface, using a very wide pen. This way we use the color of the pen — the wide border — to actually fill the shape. Here is the code corresponding to one of the scroll bars:

```
procedure TFormScroll.ScrollBarRedScroll(Sender: TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  LabelRed.Caption := 'Red: ' + IntToStr(ScrollPos);
  Shape1.Brush.Color := RGB (ScrollBarRed.Position,
```

```
        ScrollBarGreen.Position, ScrollBarBlue.Position);  
    Shape2.Pen.Color := RGB (ScrollBarRed.Position,  
        ScrollBarGreen.Position, ScrollBarBlue.Position);  
end;
```

You need to copy this code once for each scroll bar and correct the name of the label and its output text. The second and third statements always remain the same. They are based on a Windows function, `RGB`, which takes three values in the range 0–255 and creates a 32-bit value with the code of the corresponding color.

It is interesting to note that the `OnScroll` event has three parameters: the sender, the kind of event (`ScrollCode`), and the final position of the thumb (`ScrollPos`). This type of event can be used for very precise control of the user's actions. The `ScrollCode` parameter indicates if the user is dragging the thumb (`scTrack`, `scPosition`, or `scEndScroll`), has clicked on one of the two final arrows (`scLineUp` or `scLineDown`), has clicked on the bar in one of the two directions (`scPageUp` or `scPageDown`), or is trying to scroll out of the range (`scTop` or `scBottom`).

---

## What's Next

In this chapter, we have started to explore some of the basic components available in Delphi. These components correspond to the standard Windows controls and some of the Windows common controls, and are extremely common in applications (with the exception of the stand-alone scroll bars). Of course, when you start adding more advanced Delphi components to an application, you can easily build more complex and colorful user interfaces and more powerful programs.

The next chapter is devoted to a specific and important topics: the use of menus. After that, we'll see more about forms, have a light excursus on multimedia, and wrap up the book with some simple specific techniques.



## CHAPTER 5: CREATING AND HANDLING MENUS

- The structure of a menu
- Using menu templates
- Checking, disabling, and modifying menus at run-time
- Creating menu items at run-time
- A custom menu check mark
- Bitmap menu items and owner-draw menu items
- The system menu
- Pop-up menus

The sample programs we have built so far have lacked one of the most important user-interface elements of any Windows application: the menu bar. Although our forms have each had a system menu, its use has been very limited. In practical applications, however, the menu bar is a central element in the development of a program. While the user can click and sometimes drag the mouse to select options, most complex tasks usually involve menu commands. Consider the applications you use and the number of menu commands you issue in those programs (including those invoked by a shortcut key, such as Ctrl+C, which is equivalent to the Edit | Copy command in most applications).

Menus are so important that almost any real Windows application has at least one. In fact, an application can also have several menus that change at run-time (more on this later), various local menus (usually activated with a right mouse click), and even a customized system menu.

The Borland programmers who created Delphi considered menus so important that they have placed the corresponding components in the Standard page of the Components palette.

---

### The Structure of the Main Menu

Before looking at the use of menus in Delphi, let me recap some general information about menus and their structure. Usually, a menu has two levels. A menu bar, appearing below the title of the window, contains the names of the pull-down menus, each of which in turn contains a number of items. However, the menu structure is very flexible. It is possible to place a menu item directly in the menu bar and to place a *second level* pull-down menu inside another pull-down menu.

You should avoid placing commands directly on the menu bar, because users tend to select the elements of the menu bar to explore the structure of the menu. They do not expect to issue a command this way. If, for some reason, you really need to place a command in the menu bar, at least place the standard exclamation mark after it. Using an exclamation mark is a standard hint, but most users have never seen this “convention,” so it’s best to avoid the whole situation altogether and simply have a pull-down menu with a single menu item. A typical example is a Help menu with a single About menu item.

Putting a pull-down menu inside another pull-down menu — a second-level pull-down — is far more common, and Windows in this case provides a default visual clue, a small triangular glyph at the right of the menu. Many applications use this technique, because the system makes heavy use of multilevel menus (consider

the Programs menu of the Start button). However, keep in mind that selecting a menu item in a second-level pull-down takes more time and can become tedious.

Many times, instead of having a second-level pull-down, you can simply group a number of options in the original pull-down and place two separator bars, one before and one after the group. You can see an exaggerated multilevel menu in the Levels example in the source code. Since this is a demonstration of what you should try to *avoid*, I won't list the structure of the menu here.

---

## Different Roles of Menu Items

Now let's turn our attention to menu items, regardless of their position in the menu structure. There are three fundamental kinds of menu items:

- *Commands* are menu items used to execute an action. They have no special visual clue.
- *State-setters* are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active. In this case, selecting the command produces the opposite action.
- *Dialog menu items* are menu items that cause a dialog box to appear. The real difference between these and the other menu items is that a user should be able to explore the possible effects of the corresponding dialog box and eventually abort it by choosing the Cancel button. These commands should have a visual clue, consisting of an ellipsis (three dots) after the text.

Besides the traditional state-setters with a check mark, you can also have radio menu items with a bullet check mark. These menu items represent alternative selections, just as `RadioButton` components do, and simply checking one of them disables the other elements of the group. We'll explore radio menu items in an example later on.

---

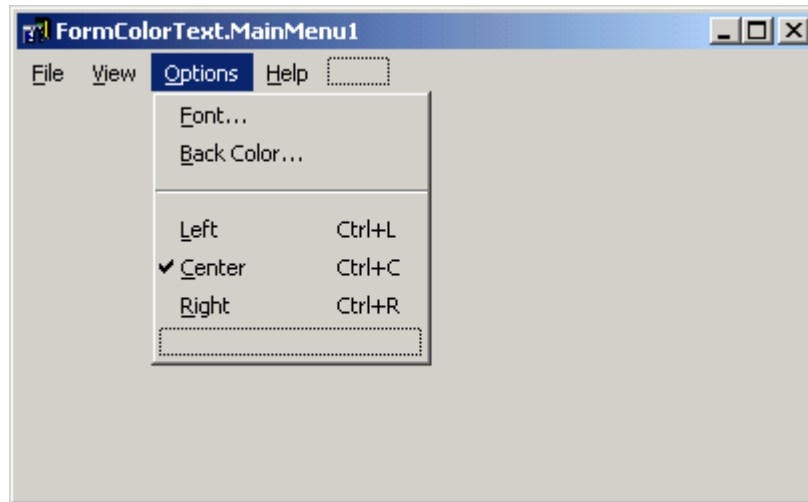
## Building a Menu with the Menu Designer

Delphi includes a special editor for menus, the Menu Designer. To invoke this tool, place a `MainMenu` component on a form and double-click on it. Don't worry too much about the position of the menu component on the form, since it doesn't affect the result; the menu is always placed properly, below the form's caption.

To be more precise, the form displays, below its caption, the menu indicated in its `Menu` property, which is set by default as soon as you create the first main menu component of the form. If the form has more than one main menu component, this property should be set manually and can be changed both at design-time and at run-time.

The Menu Designer is really powerful: It allows you to create a menu simply by writing the text of the commands, to move the items or pull-down menus by dragging them around, and to set the properties of the items easily. It is also very flexible, allowing you to place a command directly in the menu bar (this happens each time you do not write any element in the corresponding pull-down menu) or to create second-level pull-down menus.

To accomplish this, select the Create Submenu command on the Menu Designer's local menu (the local menu invoked with the right mouse button).



Another very important feature available through the Menu Designer is the ability to create a menu from a template. You can easily define new templates of your own. Simply create a menu, and use the Save As Template command on the local menu to add it to the list. This makes sense particularly if you need to have a similar menu in two applications or in two different forms of the same application.

---

## The Standard Structure of a Menu

If you've used Windows applications for some time, you have certainly noticed that the structure of an application's menu is not an invention of its programmers. There are a number of standard Windows guidelines describing how to arrange the commands in a menu. You can infer most of these rules by looking at the menus of some of the best-selling applications.

An application's menu bar should start with a File pull-down, followed by Edit, View, and then some commands specific to the application. The final part of the sequence includes Options, Tools, and Window (in MDI, or Multiple Document Interface, applications) and always terminates with Help. Each of these pull-down menus has a standard layout, although the actual items depend on the application. The File menu, for example, usually has commands such as New, Open, Save, Save As, Print, Print Setup, and Exit.

---

## Shortcut Keys and Hotkeys

A common feature of menu items is that they contain an underlined letter, generally called a *hotkey*. This letter, which is often the first letter of the text, can be used to select the menu using the keyboard. Pressing Alt plus the underlined key selects the corresponding pull-down menu. By pressing another underlined key on that menu, you issue a command.

Of course, each element of the menu bar must have a different underlined character. The same is true for the menu items on a specific pull-down menu. (Obviously, menu items on different pull-down menus can have

the same underlined letter.) To indicate the underlined key, you simply place an ampersand (&) before it, as in `Save &As . . .` or `&File`. In these examples, the underlined keys would be A for `Save As` and F for `File`.

Menu items have another standard feature: shortcut keys. When you see the shorthand description of a key, or key combination, beside a menu item, it means you can press those keys to give that command. Although giving menu commands with the mouse is easier, it tends to be somewhat slow, particularly for keyboard-intensive applications, since you have to move one of your hands from the keyboard to the mouse. Pressing Alt and the underlined letter might be faster, but it still requires two operations. Using a shortcut key usually involves pressing a special key and another key at the same time (such as Ctrl+C). Windows doesn't even display the corresponding pull-down menu, so this results in a faster internal operation, too.

In Delphi, associating a shortcut key with a menu item (pull-down menus cannot have a shortcut key) is very easy. You simply select a value for the `Shortcut` property, choosing one of the standard combinations: Ctrl or Shift plus almost any key.

You might even add shortcut keys to a program without adding a real menu. For example, you can create a pop-up menu, connect it to a form (by setting the `PopupMenu` property of the form), set the `Visible` property of all of its items to `False`, and add the proper shortcut keys; a user will never see the menu, but the shortcuts (documented in your Help system, of course) will work. If this is not clear, you can look at the `HShort` example (the name stands for "Hidden Shortcut") in the book code.

---

## Using the Predefined Menu Templates

To let you start developing an application's menu following the standard guidelines, Delphi contains some predefined menu templates. The templates include two different File pull-down menus, an Edit menu (including OLE commands), a Window menu, and two Help menus. There is also a complete MDI menu bar template, which has the same four menu categories.

Using these standard templates brings you some advantages. First of all, it is faster to reuse an existing menu than to build one from scratch. Second, the menu template follows the standard Windows guidelines for naming menu commands, for using the proper shortcuts, and so on. Of course, using these menus makes sense in a file-based application. But if the program you are writing doesn't handle files, has no editing capabilities, and is not MDI, you'll end up using only the template Help pull-down menu.

---

## Responding to Menu Commands

To build the `MenuOne` example, the first example with a menu, we will extend the `LabelCo` example of the last chapter. The new version of the form has been extended with a `MainMenu` component. This menu bar has four pull-down menus: the File pull-down with only the Exit option, the View pull-down with only the Toolbar menu item, the Options menu with various options, and the Help menu with the About menu item.

To add the separator in the Options pull-down menu, simply insert a hyphen as the text of the command. Do not change the `Break` property. (Except for rare situations, the `Break` property will make a mess of your menu. You are better off forgetting that this property even exists.)

Of course, the `Break` property has its uses, or it would not have been added to the component. You can better understand it if I use different names for its possible values, `NewLine` or `NewColumn`. If an item on the menu bar has the `mbMenuBarBreak` (or `NewLine`) value, this item will be displayed in a second or subsequent line. If a menu item has the `mbMenuBreak` (or `NewColumn`) value, this item will be added to a second or subsequent column of the pull-down. Neither of these features are used very often.

## The Code Generated by the Menu Designer

Once you have built this menu, take a look at the list of components displayed by the Object Inspector, or open the DFM file with the textual description of the form, which will also contain a textual description of the menu structure. Here is the portion of the textual description of the form related to the menu and its items:

```
object MainMenu1: TMainMenu
  object File1: TMenuItem
    Caption = '&File'
    object Exit1: TMenuItem
      Caption = 'E&xit'
      OnClick = Exit1Click
    end
  end
  object View1: TMenuItem
    Caption = '&View'
    object Toolbar1: TMenuItem
      Caption = '&Toolbar'
      Checked = True
      OnClick = Toolbar1Click
    end
  end
  object Options1: TMenuItem
    Caption = '&Options'
    object Font1: TMenuItem
      Caption = '&Font...'
      OnClick = Font1Click
    end
    object BackColor1: TMenuItem
      Caption = '&Back Color...'
      OnClick = BackColor1Click
    end
  end
  object N1: TMenuItem
    Caption = '-'
  end
  object Left1: TMenuItem
    Caption = '&Left'
    ShortCut = 16460 // stands for Ctrl+L
    OnClick = Left1Click
  end
  object Center1: TMenuItem
    Caption = '&Center'
    Checked = True
    ShortCut = 16451 // stands for Ctrl+C
```

```

        OnClick = Center1Click
    end
    object Right1: TMenuItem
        Caption = '&Right'
        ShortCut = 16466 // stands for Ctrl+R
        OnClick = Right1Click
    end
end
object Help1: TMenuItem
    Caption = '&Help'
    object About1: TMenuItem
        Caption = '&About Menu One...'
        OnClick = About1Click
    end
end
end
end

```

As you can see in the listing above, there is a specific component for each menu item, one for each pull-down menu, and, surprisingly, even one for each separator. Delphi builds the names of these components automatically when you insert the menu item's label. The rules are simple:

- Any blank or special character (including ampersands and hyphens) is removed.
- If there are no characters left, the letter N is added.
- A number is always added at the end of the name (1 if this is the first menu item with this name, a higher number if not).

All of these new components are listed in the Object Inspector, and you can select them directly or navigate among them by opening the Menu Designer and selecting menu items visually. Actually each of these items is also listed as a component in the class definition of the form:

```

type
    TFormColorText = class(TForm)
        MainMenu1: TMainMenu;
        Options1: TMenuItem;
        Font1: TMenuItem;
        BackColor1: TMenuItem;
        N1: TMenuItem;
        Left1: TMenuItem;
        Center1: TMenuItem;
        Right1: TMenuItem;
        Help1: TMenuItem;
        About1: TMenuItem;
        File1: TMenuItem;
        Exit1: TMenuItem;
        View1: TMenuItem;
        Toolbar1: TMenuItem;
        ...
    end;

```

If there are menu items your code will not refer to (such as the separators), you can actually delete the fields declaring these objects (as `N1` above). They will be created anyway, since they are listed in the form definition file, but you won't be able to access them easily from within the source code of the form. Since the objects are created anyway, you won't save much memory, though (only the space for the reference inside the form class), but removing useless statements may improve code readability.

To respond to menu commands, you should define a method for the `OnClick` event of each menu item. The `OnClick` event of the pull-down menus is used only in special cases — for example, to check whether the menu items below should be disabled. The `OnClick` event of the separators is totally useless, because it will never be activated.

Once you have defined the main menu of a form and it is displayed below the caption, you can add a new method for the `OnClick` event of a menu command simply by selecting it in the menu bar. If a handler is already present, Delphi will show you the corresponding portion of the source code; otherwise, a new method will be added to the form.

## The Code of the MenuOne Example

The code of the MenuOne example is very simple, and is similar to that of the LabelCo example it extends. The `OnClick` event of the menu command for the background color and of the corresponding toolbar button are connected to the same method:

```
object BtnBackColor: TButton
  Caption = '&Back Color...'
  OnClick = BackColor1Click
end
```

This method has the same code as the earlier version. The menu command and button related to the font are both connected to the `Font1Click` method, which this time displays the font selection dialog box, not the color selection dialog box. Notice that the code is quite compact, because it uses a `with` statement:

```
procedure TFormColorText.Font1Click(Sender: TObject);
begin
  with FontDialog1 do
  begin
    Font := Label1.Font;
    if Execute then
      Label1.Font := Font;
  end;
end;
```

The other three menu items of the Options pull-down menu (and the last three buttons of the toolbar) have basically the same code as the LabelCo example: the code of their `OnClick` event handlers simply set the `Alignment` property of the label. This works fine, but the resulting application doesn't follow the standard user-interface guidelines. Each time you have a series of choices in a menu, the selected choice should have a check mark beside it.

To accomplish this, you need to create two different operations. First, you have to place a check mark near the default choice, Center, changing the value of the menu item's `Check` property in the Object Inspector. Second, you should correct the code so that each time the selection changes, the check mark is properly set:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Center1.Checked := False;
  Right1.Checked := False;
end;
```

The other two methods are similar. You can simply copy the source code of the last three statements, paste this text twice into the other two methods, and correct the values of the three `Checked` properties so that each time one of them is set to `True`. Removing the check marks from the other items of the group can be handled in more efficient ways, but the real solution is to use radio menu items, instead of check marks. We'll look at this technique later on.

The View | Toolbar menu item is a typical item with a check mark, set when the toolbar is visible. Here is its code:

```
procedure TFormColorText.Toolbar1Click(Sender: TObject);
begin
  Panel1.Visible := not Panel1.Visible;
  Toolbar1.Checked := Panel1.Visible;
end;
```

The program toggles the status of the `Visible` property of the panel, then sets the check mark — the `Checked` property — of the menu item accordingly. You should only remember to set the initial value of this property, to match the initial status of the panel. The Help | About and File | Exit commands simply show a message box or call the `Close` method of the form, respectively. They are so simple I won't show their source code here. (You can find it in the `MenuOneF.PAS` file among the source code.)

---

## Modifying the Menu at Run-Time

You can perform a number of operations in Windows to change the structure of a menu at run-time. We'll start by examining in detail the operations you can do on a single menu item, then move to pull-down menus, build a flexible main menu, and change the default bitmap for the check mark.

---

## Changing Menu Items at Run-Time

It is important to note that menu items can change at run-time. For example, when a menu command cannot or should not be selected, it is usually grayed. In this case, the user has no way to issue that command. You shouldn't generally hide a menu item when it is not available, but simply disable it. This standard technique lets users know that the command is currently not available. Otherwise, they might think the menu command was somewhere else in the menu structure, and keep looking for it.

Another visual change is the use of the check mark, which applications can toggle on and off easily, as we've seen in the last example. At times, to implement a state-setter menu item, you can change the text of the



menu item altogether, which might result in an easier interface. For example, suppose an application has a Show Toolbar command. If you select it, a toolbar will appear and a check mark will be added to the item. This means that if you again select the Show Toolbar command, the toolbar will disappear: The command you issue has the opposite effect as its name. To avoid this problem, you might use two different captions for the two states of the menu item, such as *Show Toolbar* and *Hide Toolbar*.

When the user can select more than two choices, it is better to use multiple menu items with a check mark, or even better the new radio menu item user interface, which is becoming the standard approach.

Three properties are commonly used to modify a menu item:

- We used the `Checked` property in the example above to add or remove a check mark beside the menu item.
- The `Enabled` property can be used to gray a menu item so that it cannot be selected by a user (but it remains visible).
- The last property of this group is the `Caption`, the text of the menu item, which can be modified to reflect the actual effect of a command, as discussed above.

I'll demonstrate the use of these properties by extending the MenuOne example (the name of the new project is MenuOne2) with new menu items. I've added two new menu items to the View pull-down menu (Hide Label and Fixed Font) and two new menu items in the Options pull-down menu (Fixed View and Disable Help), plus a couple of new separators.

The Hide Label menu item demonstrates the use of different captions for an item depending on the status of the program. Here is its `OnClick` event handler:

```
procedure TFormColorText.HideLabel1Click(Sender: TObject);
begin
  Label1.Visible := not Label1.Visible;
  if Label1.Visible then
    HideLabel1.Caption := 'Hide &Label'
  else
    HideLabel1.Caption := 'Show &Label'
end;
```

---

## Disabling Menu Items and Hiding Pull-Down Menus

The other three new menu items of the MenuOne2 example are used to disable or hide other menu items or pull-down menus. The View | Fixed Font command is used to disable the Options | Font menu item:

```
procedure TFormColorText.FixedFont1Click(Sender: TObject);
begin
  ToggleCheck (FixedFont1);
  Font1.Enabled := not Font1.Enabled;
end;
```

This method calls the custom `ToggleCheck` procedure I've written as a shortcut to the code for toggling the check mark (something all these three final menu commands do). Here is its simple code:

```
procedure ToggleCheck (Item: TMenuItem);
begin
  Item.Checked := not Item.Checked;
end;
```

I've already advised you against hiding menu items (by turning off their `Visible` property), because users will probably try to find them in a different pull-down menu. Menu items should generally be disabled when you want to prevent users from calling them. Hiding entire pull-down menus, however, is a common practice. In many applications the pull-down menus you see reflect the window you are working on. Technically, it is possible to disable a pull-down menu as well, but this is not very common. The last two menu items I've added to the program do these two operations, as you can see in their corresponding `OnClick` event handlers:

```

procedure TFormColorText.FixedView1Click(Sender: TObject);
begin
    ToggleCheck (FixedView1);
    View1.Visible := not View1.Visible;
end;

procedure TFormColorText.DisableHelp1Click(Sender: TObject);
begin
    ToggleCheck (DisableHelp1);
    Help1.Enabled := not Help1.Enabled;
end;

```

The View pull-down menu has been removed, and the Help pull-down menu is grayed.

---

## Using Radio Menu Items

In addition to using check marks, in Windows you can use radio menu items. These provide not only a different user interface, but also different behavior (basically simpler code, since the system does some of the work for us). A notable example of this new user-interface feature is the View menu of the Windows Explorer.

In Delphi, simply set the `RadioItem` property of a `MenuItem` component to `True` and you get the new check mark for the item. If you set this property for several consecutive menu items and set their `GroupIndex` property to the same value, they'll use the new mark and behave as radio buttons. This means that only one of the menu items in the group will be selected at a time. Instead of having to deselect all other items manually, as you did in the first version of the `MenuOne` example, now you can simply select the proper menu item, and the rest is automatic.

Here is how you can implement this feature, in the third version of the `MenuOne` example (which also has another feature I'll discuss in the next section). The following listing shows the updated textual description of this group of menu items:

```

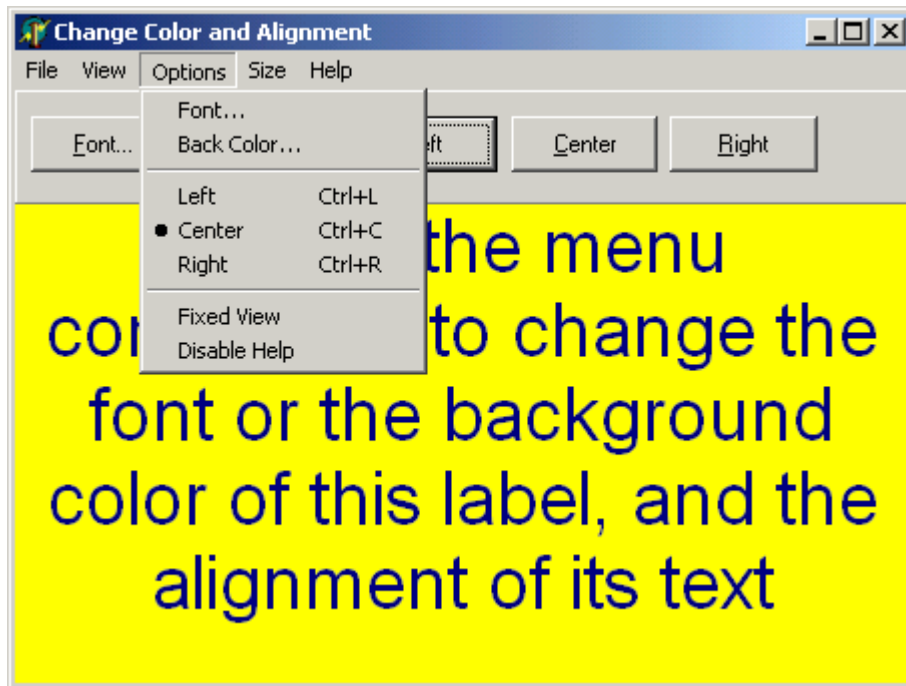
object Options1: TMenuItem
    Caption = '&Options'
    ...
object Left1: TMenuItem
    Caption = '&Left'
    GroupIndex = 1
    RadioItem = True
    OnClick = Left1Click
end
object Center1: TMenuItem
    Caption = '&Center'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = Center1Click
end

```

```

object Right1: TMenuItem
  Caption = '&Right'
  GroupIndex = 1
  RadioItem = True
  OnClick = Right1Click
end

```



This code will work as is, but we can actually simplify it. Here is the new version of the `Right1Click` method:

```

procedure TFormColorText.Right1Click(Sender: TObject);
begin
  Label1.Alignment := taRightJustify;
  // Left1.Checked := False; // now useless
  // Center1.Checked := False; // now useless
  Right1.Checked := True;
end;

```

I've simply commented out the two useless statements, instead of deleting them, to let you see the differences from the older version.

## Creating Menu Items Dynamically

The run-time changes on menu items and pull-down menus we've seen so far were all based on the direct manipulation of some properties. These components, however, also have some interesting methods, such as `Insert` and `Remove`, that you can use to make further changes.

The basic idea is that each object of the `TMenuItem` class — which Delphi uses for both menu items and pull-down menus — contains a list of menu items. Each of these items has the same structure, in a kind of

recursive way. A pull-down menu has a list of submenus, and each submenu has a list of submenus, each with its own list of submenus, and so on.

The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`, which contains the number of subitems. Adding new menu items (or entire pull-down menus) to a menu is fairly easy. Slightly more complex is the handling of the commands related to the new menu items. Basically, you need to write a specific message-response method in your code (without any help from the Delphi environment), and then assign it to the new menu item by setting its `OnClick` property. As an alternative, you can have a single method used for several `OnClick` events and use its `Sender` parameter to determine which menu command the user issued.

All these features are demonstrated by the `MenuOne3` example. As soon as you start this program, it creates a new pull-down with menu items used to change the size of the font of the big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you. I could have created the pull-down at design-time, and then added the menu items dynamically, but I prefer showing you the complete code, so that you can apply it to other cases.

To create a new menu item (or pull-down) you simply call the `Create` constructor of the `TMenuItem` class:

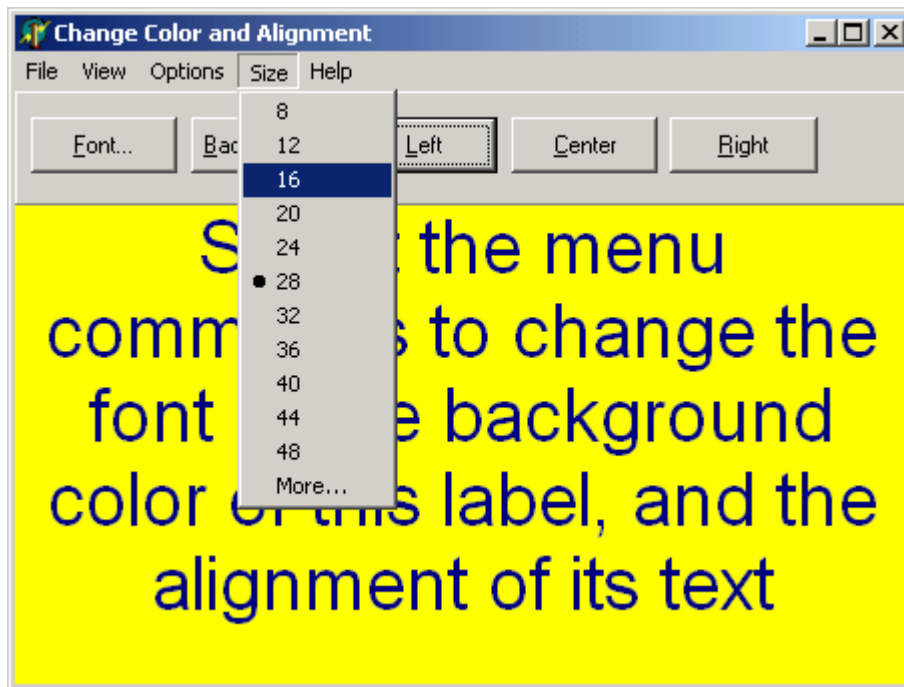
```
var
  PullDown: TMenuItem;
begin
  PullDown := TMenuItem.Create (self);
```

Then you can simply set its `Caption` and other properties, and finally insert it in the proper parent menu. The new pull-down should be inserted in `Items` of the `MainMenu1` component. You can calculate the position, knowing the index is zero-based, or you can ask the main menu component for the previous pull-down menu:

```
Position := MainMenu1.Items.IndexOf (Options1);
MainMenu1.Items.Insert (Position + 1, PullDown);
```

The menu items of this pull-down are created in a `while` loop (I don't use a `for` loop because I want to provide only one menu item for every four possible sizes: 8, 12, 16, and so on). The code to create each item is slightly more complex simply because I want to turn them into radio items, but the basic structure of the code is very simple:

```
var
  Item: TMenuItem;
  I: Integer;
begin
  ...
  I := 8;
  while I <= 48 do
  begin
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
```



To insert an item at the end I call the `Insert` method passing the number of items (`PullDown.Count`) as a parameter. As you can see, the program adds one extra item at the end of the menu, used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method, which shows the font selection dialog box:

```
Item := TMenuItem.Create (self);
Item.Caption := 'More...';
Item.OnClick := Font1Click;
PullDown.Insert (PullDown.Count, Item);
```

Also the `OnClick` events of the other menu items are connected with a method, but this time it is a method you have to define manually, by adding its declaration to the form class:

```
type
  TFormColorText = class(TForm)
    ...
  public
    procedure SizeItemClick(Sender: TObject);
  end;
```

The method should have the proper signature (or parameters list). Here is the code of the method, which is based on the `Sender` parameter:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
  end;
```

As you can see, this code doesn't set the proper check mark (or radio item mark) next to the selected item. The reason is that the user can select a new size by changing the font. For this reason, we can use a different approach, and handle the `OnClick` event of the pull-down menu. This event is activated just before showing the pull-down menu, so we can use it to set the proper check mark at that time:

```
procedure TFormColorText.SizeClick (Sender: TObject);
```

```

var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) =
        Label1.Font.Size then
        begin
          Items [I].Checked := True;
          Found := True;
          System.Break; // skip the rest of the loop
        end;
      if not Found then
        Items [Count - 1].Checked := True;
    end;
  end;
end;

```

This code scans the items of the pull-down menu we have activated (the Sender), skipping only the final one, and checks whether the caption matches the current Size of the font of the label. If no match is found, the program checks the last menu item, to indicate that a different size is active.

Of course we have to set this SizeClick event handler at run-time, when the pull-down menu is created. So we can finally look at the complete source code of the FormCreate method, which sums up all the features I have discussed in this section:

```

procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pulldown menu
  PullDown := TMenuItem.Create (self);
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
  Position := MainMenu1.Items.IndexOf (Options1);
  MainMenu1.Items.Insert (Position + 1, PullDown);

  // create menu items for various sizes
  I := 8;
  while I <= 48 do
  begin
    // create the new item
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
end;

```

```

end;

// add extra item at the end
Item := TMenuItem.Create (self);
Item.Caption := 'More...';
// make it a radio item
Item.GroupIndex := 1;
Item.RadioItem := True;
// handle click by showing the font dialog box
Item.OnClick := Font1Click;
PullDown.Insert (PullDown.Count, Item);
end;

```

---

## Creating Menus and Menu Items Dynamically

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the MenuOne3 example. As an alternative, you can also use some global functions available in the Menus unit:

```

function NewMenu(Owner: TComponent; const AName: string;
  Items: array of TMenuItem): TMainMenu;
function NewPopupMenu(Owner: TComponent;
  const AName: string; Alignment: TPopupAlignment;
  AutoPopup: Boolean; Items: array of TMenuItem):
  TPopupMenu;
function NewSubMenu(const ACaption: string;
  hCtx: Word; const AName: string;
  Items: array of TMenuItem): TMenuItem;
function NewItem(const ACaption: string;
  AShortcut: TShortcut; AChecked, AEnabled: Boolean;
  AOnClick: TNotifyEvent; hCtx: Word;
  const AName: string): TMenuItem;
function NewLine: TMenuItem;

```

The `NewMenu` and `NewPopupMenu` functions, in particular, should be used to create brand-new menus. Calling the constructors of the corresponding classes, in fact, doesn't always work properly.

---

## Short and Long Menus

If you don't like creating menu items dynamically, but still need to have a very flexible menu, there are a couple of good alternatives. You can create a large menu with all the items you need, then hide all the items and pull-down menus you do not want at the beginning. To add a new command you need only show it. This solution is a follow-up to what we have done up to now.

You can also create several menus, possibly with common elements, and exchange them as required. This approach is demonstrated in this section. A typical example of a form having two menus is one that uses two different sets of menus (long and short) for two different kinds of users (expert and inexperienced). This technique was common in major Windows applications for some years but has since been replaced by other approaches, such as letting each user redefine the whole structure of the menu.

The idea is simple and its implementation straightforward:

1. Prepare the full menu of the application, adding a menu item with the `Caption` 'Short'.
2. Add this menu to the Delphi menu template.
3. Place a second `MainMenu` component on the form, and copy its structure from the template.
4. In the second menu, remove the items corresponding to advanced features and change the `Caption` of the special item from 'Short' to 'Long'.
5. In the `Menu` property of the form, set the `MainMenu` component you want to use when the application starts, choosing one of the two available. Note that this operation has an effect on the form at design-time, too.
6. Write the code for the Short and Long commands so that when they are selected, the menu changes.

If you follow these steps, you'll end up with an application similar to `TwoMenus`, which can change its menu at run-time. The example has two different `MainMenu` components, with useless "dummy" menu items, plus the Short and Long commands.

The application does nothing apart from changing the main menu when the Short Menu or Long Menu items are selected. Here is the code for the Short Menu item:

```
procedure TForm1.ShortMenu1Click(Sender: TObject);
begin
  {activate short menu}
  Form1.Menu := MainMenu2;
end;
```

---

## Graphical Menu Items

Besides the run-time changes on a menu's structure I've listed so far, which are all directly available in Delphi, there are a number of operations you can perform on menus using the Windows API. In fact, there are several API functions referring to menus.

In particular, using Windows API functions for menus lets us add some graphics to them. We can customize the check mark, replace the strings with bitmaps, and even paint in the menu items.

---

## Customizing the Menu Check Mark

As I've just mentioned, there are a number of ways to customize a menu in Windows. In this section, I'm going to show you how you can customize the check mark used by a menu item, using two bitmaps of your own. This example, `NewCheck`, involves using bitmaps and calling a Windows API function.

First I should explain why we need two bitmaps, not just one. If you look at a menu item, it can have either a check mark or nothing. In general, however, Windows uses two different bitmaps for the checked and unchecked menu item. I've prepared two bitmaps, of 16 x 14 pixels, using the Delphi Image Editor. You can easily run this program from the Tools menu, but you can prepare the bitmaps with any editor, including Windows Paintbrush. The bitmaps should be stored in two BMP files in the same directory as the project.

The `NewCheck` example has a very simple form, with just two components, a `MainMenu` and a label:

```
object Form1: TForm1
```



```

Caption = 'New Check'
Menu = MainMenu1
OnCreate = FormCreate
OnDestroy = FormDestroy
object Label1: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'OFF'
    Font.Height = -96
    Font.Name = 'Arial'
    Font.Style = [fsBold]
end
object MainMenu1: TMainMenu
    object Command1: TMenuItem
        Caption = '&Command'
        OnClick = Command1Click
        object Toggle1: TMenuItem
            Caption = '&Toggle'
            OnClick = Toggle1Click
        end
    end
end
end

```

As you can see in the listing above, the menu item has a single command (Toggle), which will be used to change the text of the label from 'ON' to 'OFF' and change the check mark, too:

```

procedure TForm1.Toggle1Click(Sender: TObject);
begin
    Toggle1.Checked := not Toggle1.Checked;
    if Toggle1.Checked then
        Label1.Caption := 'ON'
    else
        Label1.Caption := 'OFF';
    end;

```

The most important portion of the code of this example is the call to the SetMenuItemBitmaps Windows API function:

```

function SetMenuItemBitmaps (Menu: HMenu;
    Position, Flags: Word;
    BitmapUnchecked, BitmapChecked: HBitmap): Bool;

```

This function has a number of parameters:

- The first parameter is the pull-down menu we refer to.
- The second parameter is the position of the menu item in that pull-down menu.
- The third parameter is a flag that determines how to interpret the previous parameter (Position).
- The last two parameters indicate the bitmaps that should be used.

Notice that this function changes the check mark bitmaps only for a specific menu item. Here is the code you can use in Delphi to call the function:

```

procedure TForm1.Command1Click(Sender: TObject);
begin
    SetMenuItemBitmaps (Command1.Handle,

```

```

    Toggle1.Command, MF_BYCOMMAND,
    Bmp2.Handle, Bmp1.Handle);
end;

```

This call uses two bitmap variables that are defined in the code and the names of some components (Command1 is the name of the pull-down, and Toggle1 is the name of the menu item). The code above shows that it is usually very easy to pass the handle of an element to a Windows function — just use its `Handle` property.

At first I thought this function could be called when the form was created, after the two bitmaps had been loaded from the file, but it cannot. Delphi changes the default Windows behavior somewhat, forcing the application to re-associate the bitmap with the menu items each time they are displayed. The solution I've found is to execute this call each time the pull-down menu is selected — that is, on the `OnClick` event of the pull-down.

The only thing left is to load the bitmaps. You need to add two fields of the `TBitmap` type to the form class, create an instance of the two objects, and then load the bitmaps from the two BMP files. This is done only once, when the form is created:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Bmp1 := TBitmap.Create;
    Bmp2 := TBitmap.Create;
    Bmp1.LoadFromFile ('ok.bmp');
    Bmp2.LoadFromFile ('no.bmp');
end;

```

The two bitmaps should also be destroyed when the program terminates (in the handler of the `OnDestroy` event of the form):

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    Bmp1.Free;
    Bmp2.Free;
end;

```

Notice that to run this program, you need to have the two BMP files in the same directory as the executable file. The bitmaps, in fact, are loaded at run-time and are not embedded by Delphi in the EXE file. As an alternative I could have used two non-visible `Image` components to hold the images.

You can indeed include a bitmap in the resources of an application and in its executable file in order to be able to ship the application in a single file. This process, however, is slightly more complex, so I've decided not to use it here.

## Bitmap Menu Items

Instead of placing a bitmap close to a menu item to indicate the status of its `Checked` property, as we've done in the previous section, you can actually replace the text of a menu item with a bitmap. In specific cases this can make an application easier to use.

BitMenu is a very simple program. I've put a shape in the middle of a form and added a menu to set the kind of shape (rectangle, rounded rectangle, or ellipse), and its color. Here is the textual description of the form and its menu:

```

object Form1: TForm1
  Caption = ' Bitmap Menu'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  OnResize = FormResize
object ShapeDemo: TShape... // default properties
object MainMenu1: TMainMenu
  object File1: TMenuItem...
    object Exit1: TMenuItem...
    end
  object Shape1: TMenuItem
    Caption = '&Shape'
    object Ellipse1: TMenuItem
      Caption = 'Ellipse'
      OnClick = Ellipse1Click
    end
    object RoundRec1: TMenuItem
      Caption = 'RoundRec'
      OnClick = RoundRec1Click
    end
    object Rectang1: TMenuItem
      Caption = 'Rectang'
      OnClick = Rectang1Click
    end
  end
  object Color1: TMenuItem
    Caption = '&Color'
    object Red1: TMenuItem
      Caption = 'Red'
      OnClick = Red1Click
    end
    object Green1: TMenuItem
      Caption = 'Green'
      OnClick = Green1Click
    end
    object Blue1: TMenuItem
      Caption = 'Blue'
      OnClick = Blue1Click
    end
  end
  object Help1: TMenuItem...
  object About1: TMenuItem...
end
end

```

I've listed the complete description of the menu items because their captions will play an important role in the code, as you'll see shortly. As a second step I've made the program work, by handling the various menu commands. Here are two examples from the two main pull-down menus:

```

procedure TForm1.Red1Click(Sender: TObject);
begin
  ShapeDemo.Brush.Color := clRed;
end;

procedure TForm1.Ellipse1Click(Sender: TObject);

```

```
begin
  ShapeDemo.Shape := stEllipse;
end;
```

I've also written a handler for the `OnResize` event of the form, to resize the shape depending on the actual size of the form. Instead of setting its four positional properties (`Left`, `Top`, `Width`, and `Height`) I've called the `SetBounds` method. This approach leads to faster code.

Now that we've written the code of the program, it is time to turn the menu items into bitmaps. To accomplish this I've prepared a bitmap file for each of the three shapes and one for each of the three base colors. Technically these bitmaps can have any size, but to make them look nice they should generally be wide and low (the typical size of a menu item).

Having prepared the six bitmap files, each named with the caption of its menu items (plus the `.BMP` extension), I've written a handler for the `OnCreate` event of the form, and replaced the text of the items of the fake standard menu with bitmaps. This can be accomplished by calling the `ModifyMenu` API function:

```
function ModifyMenu (hMnu: HMENU;
  uPosition, uFlags, uIDNewItem: UINT;
  lpNewItem: PChar): BOOL; stdcall;
```

The first parameter is the handle of the menu we are working on (typically a pull-down menu), the second is the position of the item we want to modify, the third is some menu flags indicating the effect of other parameters, the fourth is the identifier of the menu item (stored by Delphi in the `Command` property), and the last is the new string for the menu item. How can we use this to set a bitmap, instead? We simply pass the handle of a bitmap in the last parameter (instead of the string), and use the `mf_Bitmap` menu flag among the values of the `uFlags` parameter.

Now you are ready to look at the source code, which is based on two `for` loops. I wanted to make the code as generic as possible, so that adding a new kind of shape or color will be pretty straightforward:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
begin
  // load the bitmaps for the shapes
  for I := 0 to Shape1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Shape1.Items [I].Caption + '.bmp');
    ModifyMenu (Shape1.Handle, Shape1.Items [I].MenuIndex,
      mf_ByPosition or mf_Bitmap,
      Shape1.Items [I].Command, Pointer (Bmp.Handle));
    Shape1.Items [I].Tag := Integer (Bmp);
  end;

  // load the bitmaps for the colors
  for I := 0 to Color1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Color1.Items [I].Caption + '.bmp');
    ModifyMenu (Color1.Handle, Color1.Items [I].MenuIndex,
      mf_ByPosition or mf_Bitmap, Color1.Items [I].Command,
      Pointer (Bmp.Handle));
    Color1.Items [I].Tag := Integer (Bmp);
  end;
end;
```

```
end;
```

In the two calls to the `ModifyMenu` API function in the code above, we've used the `mf_ByPosition` flag in the third parameter, and passed the position of the item in the second parameter. However, we still need to pass the menu command as the fourth parameter, because this will be the command of the new menu item. If we don't, we'll lose the connection between the menu item and its Delphi event handler.

As you can see I save the reference to each bitmap object I create in the `Tag` property of the corresponding menu item. The only reason I have to save the bitmap objects is to be able to destroy them when the application terminates:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Shape1.Count - 1 do
    TBitmap (Shape1.Items [I].Tag).Free;
  for I := 0 to Color1.Count - 1 do
    TBitmap (Color1.Items [I].Tag).Free;
end;
```

As an alternative I could have saved the bitmap objects in an array (declared as a field of the form), and then destroyed each of the objects of the array at the end.

When you run this program, the Shape pull-down menu looks nice. The Color pull-down menu, however, is not working properly. As soon as you select it, in fact, the colors are displayed properly. But when you move over a menu item (as you can see by running the program) its colors are reversed by the menu item selection. Reversing the color you are asking for results in a very odd user interface: to select the color blue you have to click on an item that has temporarily turned to yellow!

Basically, you can use only black-and-white or gray-scaled bitmaps in a menu item. Color bitmaps create a lot of problems. If you want to obtain this effect, you should use an owner-draw menu item, as described in the next section.

---

## Owner-Draw Menu Items

In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

Actually in Delphi the situation is slightly more complex. The components can take care of painting themselves also in this case (as is the case for the `TBitBtn` class for bitmap buttons), and eventually activate corresponding events. If you don't think about the internal details, owner-draw techniques for list boxes and combo boxes require you simply to write the handler for a couple of events.

Delphi provides no support for owner-draw menu items, though. So in this case we have to use the standard Windows approach, and handle a couple of system messages in our form. Here is the definition of these methods in the source code of the `ODMenu` example, an extension of the `BitMenu` example discussed in the last section:

```
type
```

```

TForm1 = class(TForm)
...
public
  procedure WmMeasureItem (var Msg: TWmMeasureItem);
    message wm_MeasureItem;
  procedure WmDrawItem (var Msg: TWmDrawItem);
    message wm_DrawItem;
end;

```

The `wm_MeasureItem` message is sent by Windows once for each menu item when the pull-down menu is displayed to determine the size of each item. The `wm_DrawItem` message is sent when an item has to be repainted. This happens when Windows first displays the items, and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on.

In the `ODMenu` example I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've modified the `CreateForm` method to call the `ModifyMenu` API function passing the `mfOwnerDraw` menu flag. I also pass a code as the last parameter, to distinguish the various items. The other parameters are the same as in the previous version of the example:

```

procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ...
  // turn the menu items into owner-draw items
  for I := 0 to Color1.Count - 1 do
  begin
    ModifyMenu (Color1.Handle, Color1.Items [I].MenuIndex,
      mf_ByPosition or mf_OwnerDraw,
      Color1.Items [I].Command, Pointer (I));
  end;
end;

```

Now we have to write the code of the two owner-draw message handlers. The `WmMeasureItem` method receives as a parameter (in the corresponding `TWmMeasureItem` structure) a pointer to the `Windows MeasureItemStruct` structure (you can see the details of this structure in the `Windows API help file`). From this last structure we use the `CtlType` field, which stores the type of element we are measuring, to check whether this operation pertains to a menu. If it does, we use the `ItemWidth` and `ItemHeight` fields to set the width and height of the item. Since all items have the same size, the code is quite simple. The last thing we have to do is to provide a return value for the message, indicating we have handled it, in the `Result` field of the message structure. Here is the complete code:

```

procedure TForm1.WmMeasureItem (var Msg: TWmMeasureItem);
begin
  inherited;
  with Msg.MeasureItemStruct^ do
    if CtlType = odt_Menu then
    begin
      ItemWidth := 80;
      ItemHeight := 30;
      Msg.Result := 1; // we've handled it
    end;
end;

```

Drawing a menu item is slightly more complex, since to write Delphi code we have to create a `TCanvas` object, which encapsulates a Windows *device context* handle. Since we have to free this object to avoid memory leaks, I've used a `try-finally` statement to destroy it even if an error occurs:

```
Canvas1 := TCanvas.Create;
Canvas1.Handle := hDC;
try
  ...
finally
  Canvas1.Free;
end;
```

The code assigns to the `Handle` property of this `TCanvas` object the `hDC` field passed by the message in the `DrawItemStruct` structure (there is, again, a pointer to this structure passed in one of the fields of the `TWmDrawItem` parameter of the message-handler method). This `hDC` field passes to our code the handle to the device context (the painting area, the Canvas) of the pull-down menu we are going to paint.

Once we have set up the drawing mechanism, we can start with the actual code. First we have to check the state of the menu item, stored in the `ItemState` field. If this includes the `ods_Selected` flag (a condition we can test by checking the result of a bitwise and expression); then we have to paint the background of the menu item using the Windows system color for the highlighted items, `clHighlight`. Otherwise, we use the standard color for menus, `clMenu`.

Some of the Delphi constants for colors correspond to the Windows system color. These colors are not fixed, but reflect the current color setting made by the user.

Once we've assigned to the `Brush` of the canvas the background color, we can easily paint it by calling the `FillRect` method. This method has one single parameter, corresponding to the rectangle we have to erase. This information is available in the `rcItem` field of the `DrawItemStruct` structure. Notice, by the way, that if you don't limit your drawing area to this rectangle, you can paint over other menu items as well!

The second step is drawing the actual colored areas, which don't depend on the status of the menu item, since we want each color to show up properly even if the item is selected. By looking to the `ItemData` field, the code retrieves the code passed as the last parameter of the `ModifyMenu` function call, used to make the menu item owner-draw. Depending on the value of this field, the program sets a proper value for the `Color` of the `Brush` using a simple `case` statement. At this point we only have to paint the area, by calling the `Rectangle` function and passing, in the four parameters, a smaller area than the full surface of the menu item. In fact we need to reserve a border with the background color. In the example the border is 5 pixels wide.

Here, finally, is the complete source code of the `WmDrawItem` method:

```
procedure TForm1.WmDrawItem (var Msg: TWmDrawItem);
var
  Canvas1: TCanvas;
begin
  inherited;
  with Msg.DrawItemStruct^ do
    if CtlType = odt_Menu then
      begin
        // create a canvas for painting
        Canvas1 := TCanvas.Create;
        Canvas1.Handle := hDC;
        try
```

```

    // set the background color and draw it
    if (ods_Selected and ItemState <> 0) then
        Canvas1.Brush.Color := clHighlight
    else
        Canvas1.Brush.Color := clMenu;
    Canvas1.FillRect (rcItem);
    case ItemData of
        0: Canvas1.Brush.Color := clRed;
        1: Canvas1.Brush.Color := clLime;
        2: Canvas1.Brush.Color := clBlue;
    end;
    Canvas1.Rectangle (rcItem.Left + 5, rcItem.Top + 5,
        rcItem.Right - 10, rcItem.Bottom - 10);
    finally
        Canvas1.Free;
    end;
end;
end;
end;

```

I suggest you run this program along with the BitMenu program, to see the differences in the way the colored menu items are painted. Actually, looking at these two examples one might think of using the owner-draw technique also for menu items with black-and-white bitmaps.

---

## Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and for “quick-and-dirty” applications. Adding a single menu item to the system menu is straightforward:

```

AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_STRING, idSysAbout, '&About...');

```

The code fragment above (extracted from the SysMenu example) adds a separator and a new item to the system menu item. The `GetSystemMenu` API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The `AppendMenu` API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I've defined this identifier as:

```

const
    idSysAbout = 100;

```

In the SysMenu example, this code is executed in the `OnCreate` event handler, and it produces the new system menu. Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the `wm_Command` Windows message. This is handled internally by Delphi, which activates the `OnClick` event of the corresponding menu item component. The selection of system menu commands, instead, generates a `wm_SysCommand` message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.



We can intercept this command and check to see whether the command identifier (passed in the `CmdType` field of the `TWmSysCommand` parameter) of the menu item is our `idSysAbout`. Since there isn't a corresponding event in Delphi, we have to define a new message response method to the form class:

```
public
  procedure WMSysCommand (var Msg: TMessage);
  message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a `MainMenu` component to the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the `Menu` property of the form, removing the menu bar. This way we have a `MainMenu` component but nothing on the screen.

Now we can add some code to the `SysMenu` example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is pressed. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (self.Handle, FALSE),
        mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

This code uses the expression `self.Handle` to access the handle of the form. This is required because we are currently working on the `MainMenu1` component, as specified by the `with` statement.

The menu flag used in this case, `mf_Popup`, indicates that we are adding a pull-down menu. In this function call the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example we passed the identifier of the menu, instead). Since we are adding to the system menu items with sub-menus, the final structure of the system menu will have two levels.

The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms for two different things, the local menus and the secondary menus of the menu bar. Apparently, they've done this because these two elements are implemented with the same kind of internal windows; and the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course you can check for each menu item in the `WMSysCommand` method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the `OnClick` event of each item, as usual, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a `FindItem` method.

When we have found the menu item (and if we have found something), we can call its `Click` method (which invokes the `OnClick` handler). Here is the code I've added to the `WMSysCommand` method:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
  if Item <> nil then
    Item.Click;
```

In this code, The `CmdType` field of the message structure that is passed to the `WMSysCommand` procedure holds the command of the menu item being called.

You can also use a simple `if` or `case` statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as `sc_Close`, `sc_Minimize`, `sc_Maximize`, and so on. For more information, you can see the description of the `wm_SysCommand` message in the Windows API Help file, available in Delphi.

This application works but has one glitch. If you click the right mouse button over the TaskBar icon representing the application, you get a plain system menu (actually even different than the default one). The reason is that this system menu belongs to a different window, the window of the `Application` global object.

## Building a Complete Menu

Now that we know how to write a menu, disable and check menu items, and so on, we are ready to build the menu for a full-fledged application. Do you remember the `RichNote` example of the last chapter? It was a simple editor based on a `RichEdit` component. You could use it to write and change the font of the selected text, but that was all. Now we want to add a menu and implement a number of features, including a complete scheme for opening and saving the text files. In fact, we want to be able to ask the user to save any modified file before opening a new one, to avoid losing any changes. Sounds like a professional application, doesn't it?

First of all, we need to build the menu, following the standard. The main menu starts with two standard pull-down menus, `File` and `Edit`, with the typical menu items. Then there are two specific pull-down menus, `Font` and `Paragraph`, with menu items to set the text font and alignment. The last two pull-down menus, `Options` and `Help`, are *almost* standard: Their names are standard, but their menu items are not. The `Options` menu has commands to change the background color and to count the characters, and the `Help` menu has only the `About` menu item.

In this example, we want to implement most but not all of the commands of the menu, for example skipping the `Clipboard` commands. The following table shows the complete structure of the menu:

&File	&Edit	F&ont	&Paragraph	&Options	&Help
&New	Cu&t	&Times New Roman	&Left Aligned	&Backgroud Color	&About RichNote...
&Open...	&Copy	&Courier New	&Right Aligned	&Read Only	
&Save	&Paste	&Arial	&Centered	&Count chars...	
Save&As...		&Bold			
&Print...		&Italic			
E&xit		&Small			
		&Medium			
		&Large			
		More &Fonts...			

Having added a complete menu, we can now get rid of the simple panel and the font button of the RichNote example. The only visual component left in the form of the RichNot2 version will be the RichEdit component, which is aligned with the client area. Then there is the MainMenu component, and four components for standard dialog boxes (OpenDialog, SaveDialog, FontDialog, and ColorDialog).

## The File Menu

As I mentioned when we began working through the RichNot2 example, the most complex part of this program is implementing the commands of the File pull-down menu—New, Open, Save, and Save As. In each case, we need to track whether the current file has changed, saving the file only if it has. We should prompt the user to save the file each time the program creates a new file, loads an existing one, or terminates.

To accomplish this, I've added two fields and three methods to the class describing the form of the application:

```

private
    FileName: string;
    Modified: Boolean;
public
    function SaveChanges: Boolean;
    function Save: Boolean;
    function SaveAs: Boolean;
    
```

The FileName string and the Modified flag are set when the form is created and changed when a new file is loaded or the user renames a file with the Save As command. These two flags are initialized when the form is first created:

```

procedure TFormRichNote.FormCreate(Sender: TObject);
begin
    FileName := '';
    Modified := False;
end;
    
```

The value of the flag changes as soon as you type new characters in the RichEdit control (in its OnChange event handler):

```

procedure TFormRichNote.RichEdit1Change(Sender: TObject);
begin
    Modified := True;
end;
    
```

When a new file is created, the program checks whether the text has been modified. If so, it calls the `SaveChanges` function, which asks the user whether to save the changes, discard them, or skip the current operation:

```

procedure TFormRichNote.New1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
    begin
      RichEdit1.Text := '';
      Modified := False;
      FileName := '';
      Caption := 'RichNote - [Untitled]';
    end;
end;

```

If the creation of a new file is confirmed, some simple operations take place, including using *'Untitled'* instead of the file name in the form's caption.

---

## Short-Circuit Evaluation

The expression `if not Modified or SaveChanges then` requires some explanation. By default, Pascal performs what is called “short-circuit evaluation” of complex conditional expressions. The idea is simple: if the expression `not Modified` is true, we are sure that the whole expression is going to be true, and we don't need to evaluate the second expression. In this particular case, the second expression is a function call, and the function is called only if `Modified` is `True`. This behavior of `or` and `and` expressions can be changed by setting a Delphi compiler option called Complete Boolean Eval. You can find it on the Compiler page of the Project Options dialog box.

The message box displayed by the `SaveChanges` function has three options. If the user selects the Cancel button, the function returns `False`. If the user selects No, nothing happens (the file is not saved) and the function returns `True`, to indicate that although we haven't actually saved the file, the requested operation (such as creating a new file) can be accomplished. If the user selects Yes, the file is saved and the function returns `True`.

In the code of this function, notice in particular the call to the `MessageDlg` function used as the value of a case statement:

```

function TFormRichNote.SaveChanges: Boolean;
begin
  case MessageDlg (
    'The document ' + filename + ' has changed.' +
    #13#13 + 'Do you want to save the changes?',
    mtConfirmation, mbYesNoCancel, 0) of
    idYes:
      // call Save and return its result
      Result := Save;
    idNo:
      // don't save and continue
      Result := True;
    else // idCancel:
      // don't save and abort operation
      Result := False;
  end;

```

```
end;
```

In the MessageDlg call above, I've added explicit newline characters (#13) to improve the readability of the output. As an alternative to using a numeric character constant, you can call `Chr(13)`.

To actually save the file, another function is invoked: `Save`. This method saves the file if it already has a proper file name or asks the user to enter a name, calling the `SaveAs` functions. These are two more internal functions, not directly connected with menu items:

```
function TFormRichNote.Save: Boolean;
begin
  if Filename = '' then
    Result := SaveAs // ask for a file name
  else
    begin
      RichEdit1.Lines.SaveToFile (FileName);
      Modified := False;
      Result := True;
    end;
end;

function TFormRichNote.SaveAs: Boolean;
begin
  SaveDialog1.FileName := Filename;
  if SaveDialog1.Execute then
    begin
      Filename := SaveDialog1.FileName;
      Save;
      Caption := 'RichNote - ' + Filename;
      Result := True;
    end
  else
    Result := False;
end;
```

I use two functions to perform the `Save` and `SaveAs` operations (and do not call the corresponding menu handler directly) because I need a way to report a request to cancel the operation from the user. To avoid code duplication, the handlers of the `Save` and `SaveAs` menu items call the two functions too, although they ignore the return value:

```
procedure TFormRichNote.Save1Click(Sender: TObject);
begin
  if Modified then
    Save;
end;

procedure TFormRichNote.Saveas1Click(Sender: TObject);
begin
  SaveAs;
end;
```

Opening a file is much simpler. Before loading a new file, the program checks whether the current file has changed, asking the user to save it with the `SaveChanges` function, as before. The `Open1Click` method is based on the `OpenDialog` component, another default dialog box provided by Windows and supported by Delphi:

```

procedure TFormRichNote.Open1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
    if OpenFileDialog1.Execute then
      begin
        Filename := OpenFileDialog1.FileName;
        RichEdit1.Lines.LoadFromFile (FileName);
        Modified := False;
        Caption := 'RichNote - ' + FileName;
      end;
end;

```

The only other detail related to file operations is that both the `OpenDialog` and `SaveDialog` components of the `NotesForm` have a particular value for their `Filter` and `DefaultExt` properties, as you can see in the following fragment from the textual description of the form:

```

object OpenFileDialog1: TOpenDialog
  DefaultExt = 'rtf'
  FileEditStyle = fsEdit
  Filter = 'Rich Text File (*.rtf)|*.rtf|Any file (*.*)|*.*'
  Options = [ofHideReadOnly, ofPathMustExist, ofFileMustExist]
end

```

The string used for the `Filter` property (which should be written on a single line) contains four pairs of substrings, separated by the `|` symbol. Each pair has a description of the type of file that will appear in the File Open or File Save dialog box, and the filter to be applied to the files in the directory, such as `*.RTF`. To set the filters in Delphi, you can simply invoke the editor of this property, which displays a list with two columns.

The file-related methods above are also called from the `FormCloseQuery` method (the handler of the `OnCloseQuery` event), which is called each time the user tries to close the form, terminating the program. We can make this happen in various ways — by double-clicking on the system menu icon, selecting the system menu's Close command, pressing the `Alt+F4` keys, or calling the `Close` method in the code, as in the File | Exit menu command.

In `FormCloseQuery`, you can decide whether or not to actually close the application by setting the `CanClose` parameter, which is passed by reference. Again, if the current file has been modified, we call the `SaveChanges` function and use its return value. Again we can use the short-circuit evaluation technique:

```

procedure TFormRichNote.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not Modified or SaveChanges;
end;

```

The last menu item of the File menu is the Print command. Since the `RichEdit` component includes print capabilities and they are very simple to use, I've decided to implement it anyway. Here is the code, which actually produces a very nice printout:

```

procedure TFormRichNote.Print1Click(Sender: TObject);
begin
  RichEdit1.Print (FileName);
end;

```

## The Paragraph Menu

Compared to the File menu, the other pull-down menus of this example are simpler. The code of the Paragraph menu is based on some properties of its items. Here is their textual description (from the DFM file):

```

object Paragraph1: TMenuItem
  Caption = '&Paragraph'
  object LeftAligned1: TMenuItem
    Caption = '&Left Aligned'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
  object RightAligned1: TMenuItem
    Caption = '&Right Aligned'
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
  object Centered1: TMenuItem
    Caption = '&Centered'
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
end

```

As you can see, the program uses radio menu items, by giving to the three items the same value for the `GroupIndex` property and setting the `RadioItem` property to `True`. The menu items also share the same `RightAligned1Click` method for their `OnClick` event. Here is the code of the method, which is based on the correspondence between the position of the menu items in the pull-down (indicated by their `MenuItemIndex` property) and the order of the values of the `TAlignment` enumeration. It is a trick, but it works. Here is the code:

```

procedure TFormRichNote.RightAligned1Click(Sender: TObject);
begin
  RichEdit1.Paragraph.Alignment :=
    TAlignment ((Sender as TMenuItem).MenuItemIndex);
  (Sender as TMenuItem).Checked := True;
end;

```

First, this procedure sets the alignment of the current paragraph (the paragraph including the selected text or the editor cursor), then it checks the current menu item — the menu item that has activated the method (the `Sender` object). As you can see, this code relies on some controlled typecasts, based on the `as` keyword: this is what you have to do any time you want to write generic code (that is, to attach the same methods to events of different components).

Notice that setting the check mark for the current menu item is correct only until you change the selection or the current line in the text. For this reason, we can handle the `OnSelectionChange` event of the `RichEdit` component, and update the check mark of the Paragraph menu each time:

```

procedure TFormRichNote.RichEdit1SelectionChange(Sender: TObject);
begin
  Paragraph1.Items [Integer (RichEdit1.Paragraph.Alignment)].
    Checked := True;

```

```
| end;
```

## The Font Menu

The Font pull-down menu is built on the same concept, but it has two groups of radio items, plus items with a standard check mark. Each group, then, uses a different approach to handle the menu item selection with a single event response method. You can see the details of the properties of the menu items directly in the RichForm.DFM source code file (this listing was too long to reproduce here).

The code of the first group of menu items (used to select the font) is based on a simple trick: the name of the font to select corresponds to the Caption of the menu item, without the initial & character:

```
procedure TFormRichNote.TimesRoman1Click(Sender: TObject);
var
  FontName: string;
begin
  // get the font name and remove the &
  FontName := (Sender as TMenuItem).Caption;
  Delete (FontName, 1, 1);
  // change selected text font
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Name := FontName;
    (Sender as TMenuItem).Checked := True;
end;
```

This code acts on the current selection (using the SelAttributes property of the RichEdit1 component), as the RichNote example in Chapter 8 did. Notice that you can easily extend this code by adding new menu items consisting of the name of the font preceded by the & character. Then you'll necessarily have to set the same value for the GroupIndex property of the previous items (in this case 1). So if you simply set the RadioItem property to True and connect the OnClick event to the TimesRoman1Click method, they'll behave just like the existing font selection menu items.

The second group of items controls the selection of the bold and italic styles. This is accomplished by two similar but separate methods. Here is one of them:

```
procedure TFormRichNote.Bold1Click(Sender: TObject);
begin
  Bold1.Checked := not Bold1.Checked;
  if RichEdit1.SelLength > 0 then
    with RichEdit1.SelAttributes do
      if Bold1.Checked then
        Style := Style + [fsBold]
      else
        Style := Style - [fsBold];
end;
```

The last part of the menu has another group of radio menu items, used to set the size of the font. These menu items refer to a font size in the caption and have the same value stored in their Tag property. This makes the code very easy to write. Again, there is a single method for the three menu items, but you can add new items to this group with very little effort:

```
procedure TFormRichNote.Large1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Size := (Sender as TMenuItem).Tag;
```



```

(Sender as TMenuItem).Checked := True;
end;

```

The last item of this menu simply activates the Font dialog box. Notice that the font returned by this dialog box cannot be assigned directly to the `SelAttributes` property; we need to call the `Assign` method, instead:

```

procedure TFormRichNote.More1Click(Sender: TObject);
begin
    FontDialog1.Font := RichEdit1.Font;
    if FontDialog1.Execute then
        RichEdit1.SelAttributes.Assign (FontDialog1.Font);
        // update the check marks
        RichEdit1.SelectionChange (self);
end;

```

All the commands of this menu affect the status of the current selection, setting the check boxes and radio items properly. The method above, instead, calls the `RichEdit1.SelectionChange` method. This is the handler of the `OnSelectionChange` event of the `RichEdit` component.

This method scans some of the groups to determine which element has to be checked. As in other examples before, this code has been written so that you can easily extend it for new menu items (the listing also includes the code used to reset the paragraph alignment, discussed earlier):

```

procedure TFormRichNote.RichEdit1SelectionChange(Sender: TObject);
var
    FontName: string;
    I: Integer;
begin
    // check the font name radio menu item
    FontName := '&' + RichEdit1.SelAttributes.Name;
    for I := 0 to 2 do
        with Font1.Items [I] do
            if FontName = Caption then
                Checked := True;

    // check the bold and italic items
    Italic1.Checked :=
        fsItalic in RichEdit1.SelAttributes.Style;
    Bold1.Checked :=
        fsBold in RichEdit1.SelAttributes.Style;

    // check the font size
    for I := Small1.MenuIndex to Large1.MenuIndex do
        with Font1.Items [I] do
            if Tag = RichEdit1.SelAttributes.Size then
                Checked := True;

    // check the paragraph style
    Paragraph1.Items [Integer (RichEdit1.Paragraph.Alignment)].
        Checked := True;
end;

```

This method doesn't work perfectly. When you set a custom font, the selection doesn't change properly (because the current item remains selected). To fix it, we should remove any radio check mark when the value is not one of the possible selections, or add new menu items for this special case. The example is complex enough, so I think we can live with this minor inconvenience.

## The Options Menu

The last pull-down menu of the RichNot2 example is the Options menu. This menu has three unrelated commands used to customize the user interface, and to determine and display the length of the text. The first command displays a color selection dialog box, used to change the color of the background of the RichEdit component:

```
procedure TFormRichNote.BackColor1Click(Sender: TObject);
begin
    ColorDialog1.Color := RichEdit1.Color;
    if ColorDialog1.Execute then
        RichEdit1.Color := ColorDialog1.Color;
end;
```

The second command can be used to mark the text as read-only. In theory this property should be set depending on the status of the file on the disk. In the example, instead, the user can toggle the read-only attribute manually:

```
procedure TFormRichNote.ReadOnly1Click(Sender: TObject);
begin
    RichEdit1.ReadOnly := not RichEdit1.ReadOnly;
    ReadOnly1.Checked := not ReadOnly1.Checked;
end;
```

The last menu item activates a method to count the number of characters in the text and display the total in a message box. The core of the method is the call to the `GetTextLen` function of the RichEdit control. The number is extracted and formatted into an output string:

```
procedure TFormRichNote.Countchars1Click(Sender: TObject);
begin
    MessageDlg (Format (
        'The text has %d characters', [RichEdit1.GetTextLen]),
        mtInformation, [mbOK], 0);
end;
```

The handler of the `OnClick` event of this last item of the Options menu terminates the example. As mentioned at the beginning, this was a rather long and complex example, but its purpose was to show you the implementation of the menu commands of a real-world application. In particular, I explained in detail the File pull-down menu because this is something you'll probably need to handle in any file-related application.

Now we are ready to delve into another topic involving menus, the use of local menus activated by the right mouse button click.

---

## Pop-Up Menus

In Windows, it is common to see applications that have special local menus you activate by clicking the right mouse button. The menu that is displayed — a pop-up menu, in common Windows terminology — usually depends on the position of the mouse click. These menus tend to be easy to use since they group only the few commands related to the element that is currently selected. They are also usually faster to use than full-blown menus because you don't need to move the mouse up to the menu bar and then down again to go on working.

In Delphi, there are basically two ways to display pop-up menus, using the corresponding component. You can let Delphi handle them automatically or you can choose a manual technique. I'll explore both approaches, starting with the first, which is the simplest one.

To add a pop-up menu to a form, you need to perform a few simple operations. Create a `PopupMenu` component, add some menu items to it, and select the component as the value of the form's `PopupMenu` property. That's all. Of course, you should also add some handlers for the `OnClick` events of the local menu's various menu items, as you do with an ordinary menu.

---

## An Automatic Local Menu

To show you how to create a local menu, I've built an example that is an extension of the `Dragging` example. The new example is named `Local1`. I've added a first `PopupMenu` component to its form and connected it using the `PopupMenu` property of the form itself. Once this is done, running the program and clicking the right mouse button on the form displays the local menu. Then, I've added a second pop-up menu component, with two levels, to the form, and I've attached it to the `StaticText` component on the right, `LabelTarget`. To connect a local menu to a specific component, you simply need to set its `PopupMenu` property. The four methods related to the first group of commands of the `Colors` pull-down menu just select a color:

```
procedure TDraggingForm.Aqua1Click(Sender: TObject);
begin
    LabelTarget.Color := clAqua;
end;
```

The `Transparent` command selects the color of the parent form as the current color, setting the value of the `ParentColor` property to `True`.

The components of a form usually borrow some properties from the form. This is indicated by specific properties, such as `ParentColor` or `ParentFont`. When these properties are set to `True`, the current value of the component's property is ignored, and the value of the form is used instead. Usually, this is not a problem, because as soon as you set a property of the component (for example, the font), the corresponding property indicating the use of the parent attribute (`ParentFont`) is automatically set to `False`.

The last command of the pull-down menu, `User Defined`, presents the standard `Color Selection` dialog box to the user. The three commands of the pop-up menu's second pull-down change the alignment of the text of the big label and add a check mark near the current selection, deselecting the other two menu items. Here is one of the three methods:

```
procedure TDraggingForm.Center1Click(Sender: TObject);
begin
    LabelTarget.Alignment := taCenter;
    Left1.Checked := False;
    Center1.Checked := True;
    Right1.Checked := False;
end;
```

A pop-up menu, in fact, can use all the features of a main menu and can have checked, disabled, or hidden items, and more. I could have also used radio menu items for this pop-up menu, but this doesn't seem to be a very common approach.

## Modifying a Pop-Up Menu When It Is Activated

Why not use the same technique to display a check mark near the selected color? It is possible, but it's not a very good solution. In fact, there are six menu items to consider, and the color can also change when a user drags it from one of the labels on the left of the form. For this reason, and to show you another technique, I've followed a different approach.

Each time a pop-up menu is displayed, the `OnPopup` event is sent to your application. In the code of the corresponding method, you can place the check mark on the current selection of the color, independently from the action used to set it:

```
procedure TDraggingForm.PopupMenu2Popup(Sender: TObject);
var
  I: Integer;
begin
  // unchecks all menu items (not required for radio menu items)
  with Colors1 do
    for I := 0 to Count - 1 do
      Items[I].Checked := False;

  // checks the proper item
  case LabelTarget.Color of
    clRed: Red1.Checked := True;
    clAqua: Aqua1.Checked := True;
    clGreen: Green1.Checked := True;
    clYellow: Yellow1.Checked := True;
  else
    if LabelTarget.ParentColor then
      Transparent1.Checked := True
    else
      UserDefined1.Checked := True;
  end;
end;
```

This method's code requires some explanation. At the beginning, the menu items are all unchecked by using a `for` loop on the `Items` array of the `Colors1` menu. The advantage of this loop is that it operates on all the menu items, regardless of their number. Then the program uses a `case` statement to check the proper item.

## Handling Pop-Up Menus Manually

In the `Local1` example, we saw how to use automatic pop-up menus. As an alternative, you can set the `AutoPopup` property to `False` or not connect the pop-up menu to any component, and use the pop-up menu's `Popup` method to display it on the screen. This procedure requires two parameters: the *x* and *y* values of the position where the menu is going to be displayed. The problem is that you need to supply the *screen* coordinates of the point, not the client coordinates, which are the usual coordinates relative to the form's client area.

As an example, I've taken an existing application with a menu — the third version of the `MenuOne` example, described in this chapter — and added a peculiar pop-up menu. The idea is that there are two different pop-up menus, one to change the colors and the other to change the alignment of the text. Each time the user right-clicks on the caption, one of the two pop-up menus is displayed. In real applications, you'll probably have to decide which menu to display depending on the status of some variable. Here, I've followed a simple (and

arbitrary) rule: Each time the right mouse button is clicked, the pop-up menu changes. My aim is to show you how to do this in the simplest possible way.

The two pop-up menus are very simple, and correspond to actions already available in the main menu (and connected with the same event handlers). Actually, I've built these pop-up menus by copying the main menu to a menu template and then pasting from it. The only change I've made is to remove the shortcut keys.

When the user clicks the right mouse button over the label, which takes up the whole surface of the form, a method displays one of the two menus. Instead of using the `OnClick` event, I've trapped the `OnMouseDown` event of the label. This second event passes as parameters the coordinates of the mouse click.

These coordinates are relative to the label, so you have to convert them to screen coordinates by calling the `ClientToScreen` method of the label:

```
procedure TFormColorText.Label1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ClientPoint, ScreenPoint: TPoint;
begin
  if Button = mbRight then
    begin
      ClientPoint.X := X;
      ClientPoint.Y := Y;
      ScreenPoint := Label1.ClientToScreen (ClientPoint);
      Inc (ClickCount);
      if Odd (ClickCount) then
        PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
      else
        PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
    end;
  end;
```

In this procedure, you first have to check whether the right mouse button was clicked. The second step is to translate the coordinate of the position of the mouse click from client coordinates to screen coordinates. Screen coordinates are required by the `PopupMenu` component's `Popup` method.

The last thing we have to do is provide the proper check marks for the menu items of the second pop-up menu. A solution is to copy the current check marks of the main menu to the pop-up before displaying it:

```
if Odd (ClickCount) then
  PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
else
  begin
    {set the check marks as in the main menu}
    Left2.Checked := Left1.Checked;
    Center2.Checked := Center1.Checked;
    Right2.Checked := Right1.Checked;
    PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  end;
```

An alternative solution—the one I've actually implemented in the `Local2` example — is to set this check mark every time you set the check marks of the main menu, as in the following code:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Left2.Checked := True;
end;
```

In a more general application you might want to write a generic routine to apply the check marks more consistently, but in this example one of these two simple techniques will do.

---

## What's Next

In this chapter, we have seen how to create main menus and pop-up menus in Delphi. We've discussed the standard guidelines for the names of the pull-down menus and of the menu items, shortcut keys, check marks, graphical menus, local menus, and many other topics. You can explore in other directions, as well. For example, you can create a menu dynamically (at run-time) or copy portions of a menu to another menu, as in the SysMenu example.

The next step, however, is to explore a less common but very nice feature of Delphi programming, multimedia.

## CHAPTER 6: MULTIMEDIA FUN

- Windows default sounds
- From a beep to music
- The Media Player component
- Playing sounds and running videos
- Applications for audio CD drives

Among the many devices, the multimedia subsystem focuses sound cards or CD-ROM drives. Of course, besides being physically connected to your computer, these devices must be properly installed in Windows for your Delphi applications to access them.

Windows provides a specific API, known as the *Multimedia API*, to handle external devices such as video, MIDI, and CD drives. Delphi includes a corresponding Help file along with an easy-to-use component, the Media Player, to manipulate most multimedia devices. Before discussing this component, which is the main topic of the chapter, we'll look at some simpler ways to produce sound in Windows, beyond the simple beeps we have used previously.

---

### Windows Default Sounds

In the book's earlier examples, every time we wanted to notify the user of an error or a specific event, we called a Delphi system procedure (`Beep`) or a Windows API function (`MessageBeep`). The `Beep` procedure is defined in the Delphi run-time library as follows:

```
procedure Beep;  
begin  
  MessageBeep(0);  
end;
```

It simply passes the value 0 to the `MessageBeep` API function. Besides the values 0 and -1, both used to produce a beep with the internal speaker of the computer, the `MessageBeep` function can also accept other values, and play the corresponding sounds with your sound board. Here are the acceptable constants and the corresponding Windows sounds they produce (these are the sound names available in Control Panel):

<code>mb_IconAsterisk</code>	SystemAsterisk sound
<code>mb_IconExclamation</code>	SystemExclamation sound
<code>mb_IconHand</code>	SystemHand sound
<code>mb_IconQuestion</code>	SystemQuestion sound
<code>mb_Ok</code>	SystemDefault sound

You can change the association between system events and sound files using the Control Panel, which lists the sounds under the names shown in the right column above. These associations are stored in the Windows System Registry.

Notice that these constants are also the possible values of the `MessageBox` API function, encapsulated in the `MessageBox` method of the `TApplication` class. It is common to produce the corresponding sound when the message box is displayed. This feature is not directly available in the Delphi `MessageDlg` function (which displays a message box with a corresponding icon), but we can extend it easily by building a `SoundMessageDlg` function, as demonstrated by the following example.

## Every Box Has a Beep

To show you the capabilities of the `MessageBeep` API function, I've prepared a simple example, `Beeps`. The form of this example has a `RadioGroup` with some radio buttons from which the user can choose one of the five valid constants of the `MessageBeep` function. Here is the definition of the `RadioGroup`, from the textual description of the form:

```
object RadioGroup1: TRadioGroup
  Caption = 'Parameters'
  ItemIndex = 0
  Items.Strings = (
    'mb_IconAsterisk'
    'mb_IconExclamation'
    'mb_IconHand'
    'mb_IconQuestion'
    'mb_Ok')
end
```

The program plays the sound corresponding to the current selection when the user clicks on the `Beep Sound` button (one of the push buttons of the form). This button's `OnClick` event-handler first determines which radio button was selected, using a case statement, and then plays the corresponding sound:

```
procedure TForm1.BeepButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
    4: BeepConstant := mb_Ok;
  else
    BeepConstant := 0;
  end;
  MessageBeep (BeepConstant);
end;
```

The `else` clause of the case statement is provided mainly to prevent an annoying (but not dangerous) compiler warning. To compare the selected sound with the default beep sound, click on the second button of the column (labeled *Beep -1*), which has the following code:

```
procedure TForm1.BeepOneButtonClick(Sender: TObject);
begin
  MessageBeep (Cardinal (-1));
end;
```

You can also pass the corresponding `$FFFFFFFF` hexadecimal value to the `MessageBeep` function. There is actually no difference between the two approaches. To test whether a sound driver is installed in your



system (with or without a sound card, since it is possible to have a sound driver for the PC speaker), click on the first button (labeled *Test*), which uses a multimedia function, `WaveOutGetNumDevs`, to perform the test:

```
procedure TForm1.TestButtonClick(Sender: TObject);
begin
    if WaveOutGetNumDevs > 0 then
        SoundMessageDlg ('Sound is supported',
            mtInformation, [mbOk], 0)
    else
        SoundMessageDlg ('Sound is NOT supported',
            mtError, [mbOk], 0);
end;
```

To compile this function, you need to add the `MmSystem` unit to the `uses` clause. If your computer has no sound driver installed, you will hear only standard beeps, regardless of which sound is selected. The last two buttons have a similar aim: they both display a message box and play the corresponding sound.

The `OnClick` event handler of the Message Box button uses the traditional Windows approach. It calls the `MessageBeep` function and then the `MessageBox` method of the `Application` object soon afterward. The effect is that the sound is played when the message box is displayed. In fact (depending on the sound driver), playing a sound doesn't usually stop other Windows operations. Here is the code related to this fourth button:

```
procedure TForm1.BoxButtonClick(Sender: TObject);
var
    BeepConstant: Cardinal;
begin
    case RadioGroup1.ItemIndex of
        0: BeepConstant := mb_IconAsterisk;
        1: BeepConstant := mb_IconExclamation;
        2: BeepConstant := mb_IconHand;
        3: BeepConstant := mb_IconQuestion;
    else {including 4:}
        BeepConstant := mb_Ok;
    end;
    MessageBeep (BeepConstant);
    Application.MessageBox (
        PChar (RadioGroup1.Items [RadioGroup1.ItemIndex]),
        'Sound', BeepConstant);
end;
```

If you click on the last button, the program calls the `SoundMessageDlg` function, which is not an internal Delphi function. It's one I've added to the program, but you can use it in your applications. The only suggestion I have is to choose a shorter name if you want to use it frequently. `SoundMessageDlg` plays a sound, specified by its `AType` parameter, and then displays the Delphi standard message box:

```
function SoundMessageDlg (const Msg: string;
    AType: TMsgDlgType; AButtons: TMsgDlgButtons;
    HelpCtx: Longint): Integer;
var
    BeepConstant: Cardinal;
begin
    case AType of
        mtWarning: BeepConstant := mb_IconExclamation;
        mtError: BeepConstant := mb_IconHand;
        mtInformation: BeepConstant := mb_IconAsterisk;
        mtConfirmation: BeepConstant := mb_IconQuestion;
```

```

else
  BeepConstant := mb_Ok;
end;
MessageBeep(BeepConstant);
Result := MessageDlg (Msg, AType,
  AButtons, HelpCtx);
end;

procedure TForm1.MessDlgButtonClick(Sender: TObject);
var
  DlgType: TMsgDlgType;
begin
  case RadioGroup1.ItemIndex of
    0: DlgType := mtInformation;
    1: DlgType := mtWarning;
    2: DlgType := mtError;
    3: DlgType := mtConfirmation;
  else {including 4:}
    DlgType := mtCustom;
  end;
  SoundMessageDlg (
    RadioGroup1.Items [RadioGroup1.ItemIndex],
    DlgType, [mbOK], 0);
end;

```

SoundMessageDlg is a simple function, but your programs can really benefit from its use.

---

## From Beeps to Music

When you use the MessageBeep function, your choice of sounds is limited to the default system sounds. Another Windows API function, PlaySound, can be used to play a system sound, as well as any other waveform file (WAV). Again, I've built a simple example to show you this approach. The example is named ExtBeep (for Extended Beep) and has the simple form.

The form's list box shows the names of some system sounds and some WAV files, available in the current directory (that is, the directory containing ExtBeep.exe itself). When the user clicks on the Play button, the PlaySound function (defined in the MmSystem unit) is called:

```

procedure TForm1.PlayButtonClick(Sender: TObject);
begin
  PlaySound (PChar (Listbox1.Items [ListBox1.ItemIndex]),
    0, snd_Async);
end;

```

The first parameter is the name of the sound—either a system sound, a WAV file, or a specific sound resource (see the Win32 API Help file for details). The second parameter specifies where to look for a resource sound, and the third contains a series of flags, in this case indicating that the function should return immediately and let the sound play asynchronously. (An alternative value for this parameter is snd\_Sync. If you use this value, the function won't return until the sound has finished playing.) With asynchronous play, you can interrupt a long sound by calling the PlaySound function again, using nil for the first parameter:

```

procedure TForm1.StopButtonClick(Sender: TObject);
begin
  PlaySound (nil, 0, 0);
end;

```

```
end;
```

This is the code executed by the ExtBeep example when the user clicks on the Stop button. This button is particularly useful for stopping the repeated execution of the sound started by the Loop button, which calls PlaySound passing as its last parameter (snd\_Async or snd\_Loop).

The only other method of the example, FormCreate, selects the first item of the list box at startup, by setting its ItemIndex property to 0. This avoids run-time errors if the user clicks on the button before selecting an item from the list box. You can test this example by running it, and by adding the names of the other WAV files or system sounds (as listed in the registration database). I suggest you also test other values for the third parameter of the function (see the API help files for details).

---

## The Media Player Component

Now let's move back to Delphi and use the Media Player component. The Delphi TMediaPlayer class encapsulates most of the capabilities of the Windows Media Control Interface (MCI), a high-level interface for controlling internal and external media devices.

Perhaps the most important property of the TMediaPlayer component is DeviceType. Its value can be dtAutoSelect, indicating that the type of the device depends on the file extension of the current file (the FileName property). As an alternative, you can select a specific device type, such as dtAVIVideo, dtCDAudio, dtWaveAudio, and many others.

Once the device type (and eventually the file) have been selected, you can open the corresponding device (or set AutoOpen to True), and the buttons of the Media Player component will be enabled. The component has a number of buttons, not all of which are appropriate for each media type. There are actually three properties referring to the buttons: VisibleButtons, EnabledButtons, and ColoredButtons. The first determines which of the buttons are present in the control, the second determines which buttons are enabled, and the third determines which buttons have colored marks. By using the first two of these properties, you can permanently or temporarily hide or disable some of the buttons.

The component has several events. The OnClick event is unusual because it contains one parameter indicating which button was pressed and a second parameter you can use to disable the button's default action. The OnNotify event later tells the component whether the action generated by the button was successful. Another event, OnPostClick, is sent either when the action starts or when it ends, depending on the value of the Wait property. This property determines whether the operation on the device should be synchronous.

---

## Playing Sound Files

Our first example using the Media Player is very simple. The form of the MmSound example has some labels describing the current status, a button to select a new file, an OpenFileDialog component, and a Media Player component with the following settings:

```
object MediaPlayer1: TMediaPlayer
  VisibleButtons = [btPlay, btPause,
    btStop, btNext, btPrev]
  OnClick = MediaPlayer1Click
  OnNotify = MediaPlayer1Notify
```

**end**

When a user opens a new file, a wave table, or a MIDI file, the program enables the Media Player, and you can play the sound and use the other buttons, too:

```
procedure TForm1.NewButtonClick(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
        begin
            FileLabel.Caption := OpenFileDialog1.FileName;
            MediaPlayer1.FileName := OpenFileDialog1.FileName;
            MediaPlayer1.Open;
            MediaPlayer1.Notify := True;
        end;
    end;
```

Since I set the `Notify` property to `True`, the Media Player invokes the corresponding event handler, which outputs the information to a label:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
    case MediaPlayer1.NotifyValue of
        nvSuccessful : NotifLabel.Caption := 'Success';
        nvSuperseded : NotifLabel.Caption := 'Superseded';
        nvAborted    : NotifLabel.Caption := 'Aborted';
        nvFailure    : NotifLabel.Caption := 'Failure';
    end;
    MediaPlayer1.Notify := True;
end;
```

Notice that you need to set the `Notify` property to `True` every time the `OnNotify` event handler is called in order to receive further notifications. Another label is updated to display the requested command.

```
procedure TForm1.MediaPlayer1Click(Sender: TObject;
    Button: TMPBtnType; var DoDefault: Boolean);
begin
    case Button of
        btPlay: ActionLabel.Caption := 'Playing';
        btPause: ActionLabel.Caption := 'Paused';
        btStop: ActionLabel.Caption := 'Stopped';
        btNext: ActionLabel.Caption := 'Next';
        btPrev: ActionLabel.Caption := 'Previous';
    end;
end;
```

---

## Running Videos

So far, we have worked with sound only. Now it is time to move to another kind of media device: video. You indeed have a video device on your system, but to play video files (such as AVI files), you need a specific driver (directly available in Windows). If your computer can display videos, writing a Delphi application to do so is almost trivial: place a Media Player component in a form, select an AVI file in the `FileName` property, set the `AutoOpen` property to `True`, and run the program. As soon as you click on the Play button, the system opens a second window and shows the video in it.

Instead of playing the file in its own window, we can add a panel (or any other windowed component) to the form and use the name of this panel as the value of the Media Player's `Display` property. As an alternative,

we can set the `Display` and the `DisplayRect` properties to indicate which portions of the output window the video should cover.

Although it is possible to create a similar program writing no code at all, to do so I would have to know which AVI files reside on your computer, and specify the full path of one of them in the `FileName` property of the Media Player component. As an alternative, I've written a simple routine to open and start playing a file automatically. You only have to click on the panel (as the caption suggests).

```

procedure TForm1.Panel1Click(Sender: TObject);
begin
  if OpenFileDialog1.Execute then
    begin
      MediaPlayer1.FileName := OpenFileDialog1.FileName;
      MediaPlayer1.Open;
      MediaPlayer1.Perform (wm_LButtonDown, 0, $00090009);
      MediaPlayer1.Perform (wm_LButtonUp, 0, $00090009);
    end;
end;

```

After opening the Media Player, I could have called its `Play` method immediately to start it. But that would not have enabled and disabled the buttons properly. So I decided to simulate a click in position 9 on the *x*-axis and 9 on the *y*-axis of the Media Player window (instead of building the 32-bit value including both coordinates with a function, you can use the hexadecimal value directly, as in the code above). To avoid errors, I disabled all the buttons at design-time, until the simulated click takes place. I also automatically close the player when the application is closed (in the `OnClose` event handler).

---

## A Video in a Form

The Media Player component has some limits regarding the window it can use to produce the output. You can use many components, but not all of them. A strange thing you can try is to use the Media Player component itself as the video's output window. This works, but there are two problems. First, the Media Player component cannot be aligned, and it cannot be sized at will. If you try to use big buttons, their size will be reduced automatically at run-time. The second problem is that if you click on the Pause button, you'll see the button in front of the video, while the other buttons are still covered. (I suggest you try this approach, anyway, just for fun.)

One thing you cannot do easily is display the video in a form. In fact, although you cannot set the form as the value of the Media Player's `Display` property at design-time, you can set it at run-time. To try this, simply place a hidden Media Player component (set the `Visible` property to `False`) and an `OpenDialog` component in a form. Set a proper title and hint for the form itself, and enable the `ShowHints` property. Then write the following code to load, start, and stop the video when the user clicks on the form:

```

procedure TForm1.FormClick(Sender: TObject);
begin
  if MediaPlayer1.FileName = '' then
    if OpenFileDialog1.Execute then
      begin
        MediaPlayer1.FileName := OpenFileDialog1.FileName;
        MediaPlayer1.Open;
        Playing := False;
      end
    else
      exit; // stop if no file is selected

```

```
if Playing then
begin
  MediaPlayer1.Stop;
  Playing := False;
  Caption := 'MM Video (Stopped)';
  Hint := 'Click to play video';
end
else
begin
  MediaPlayer1.Display := self;
  MediaPlayer1.DisplayRect := ClientRect;
  MediaPlayer1.Play;
  Playing := True;
  Caption := 'MMV (Playing)';
  Hint := 'Click to stop video';
end;
end;
```

In this code, `Playing` is a private Boolean field of the form. Notice that the program shows the video using the full client area of the form. If the form is resized, you can simply enlarge the output rectangle accordingly:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  MediaPlayer1.DisplayRect := ClientRect;
end;
```

The best way to view a video is to use its original size, but with this program you can actually stretch it, and even change its proportions. Of course, the Media Player can also stop when it reaches the end of a file or when an error occurs. In both cases, we receive a notification event:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  Playing := False;
  Caption := 'MMV (Stopped)';
  Hint := 'Click to play video';
end;
```

---

## Working with a CD Drive

In addition to audio and video files, the MCI interface is generally used to operate external devices. There are many examples, but the most common MCI device connected to a PC is probably a CD-ROM drive. Most CD-ROM drives can also read audio CDs, sending the output to an external speaker or a sound card. You can use the MCI interface and the Media Player component to write applications that handle such a device. Basically, you need to set the `DeviceType` property to `dtCDAudio`, making sure no file is selected in the `FileName` property, and be ready with a CD player.

In fact, just by placing a Media Player component in a form, setting the above properties, and compiling and running the program, you end up with a fully functional audio CD player. When you start customizing the player, though, not everything is as simple as it seems at first glance. I've built an example using some more capabilities of this component and of Windows multimedia support related to audio CDs. The form of this

program has a couple of buttons, some labels to show the current status, a timer, and a SpinEdit component you can use to choose a track from the disk.

The idea is to use the labels to inform the user of the number of tracks on a disk, the current track, the current position within a track, and the length of the track, monitoring the current situation using the timer.

In general, if you can, use the `tfTMSF` value (Track, Minute, Second, Frame) for the `TimeFormat` property of the Media Player component to access positional properties (such as `Position` and `Length`). Extracting the values is not too complex if you use the proper functions of the `MmSystem` unit, such as the following:

```
CurrentTrack := Mci_TMSF_Track (MediaPlayer1.Position);
```

Here are the two functions that compute the values for the whole disk and for the current track:

```
procedure TForm1.CheckDisk;
var
  NTracks, NLen: Integer;
begin
  NTracks := MediaPlayer1.Tracks;
  NLen := MediaPlayer1.Length;
  DiskLabel.Caption := Format (
    'Tracks: %.2d, Length:%.2d:%.2d', [NTracks,
    Mci_TMSF_Minute (NLen), Mci_TMSF_Second (NLen)]);
  SpinEdit1.MaxValue := NTracks;
end;

procedure TForm1.CheckPosition;
var
  CurrentTrack, CurrentPos, TrackLen: Integer;
begin
  CurrentPos := MediaPlayer1.Position;
  CurPosLabel.Caption := Format ('Position: %.2d:%.2d',
    [Mci_TMSF_Minute (CurrentPos),
    Mci_TMSF_Second (CurrentPos)]);
  CurrentTrack := Mci_TMSF_Track (CurrentPos);
  TrackLen := MediaPlayer1.TrackLength [CurrentTrack];
  TrackNumberLabel.Caption := Format (
    'Current track: %.2d, Length:%.2d:%.2d', [CurrentTrack,
    Mci_MS_F_Minute (TrackLen), Mci_MS_F_Second (TrackLen)]);
end;
```

The code seems complex only because of the many conversions it makes. Notice in particular that the length of the current track (stored in the `TrackLength` property) is not measured using the default format, as the online help suggests, but with the MSF (Minute Second Frame) format.

The global values for the disk are computed only at startup and when the New CD button is clicked:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.TimeFormat := tfTMSF;
  MediaPlayer1.Open;
  CheckDisk;
  CheckPosition;
end;

procedure TForm1.NewButtonClick(Sender: TObject);
begin
  CheckDisk;
```

```
    CheckPosition;
end;
```

The values for the current track and position are computed this way each time the timer interval elapses, by calling the `CheckPosition` method. This is far from perfect, because if you want to play an audio CD while using other programs, a timer accessing the Media Player information often slows down the system too much. Of course, this mainly depends on your hardware. Besides telling the user what is going on, the form has the Media Player component to allow the user to start and stop playing, change tracks, and so on. The operations on this component activate and halt the timer:

```
procedure TForm1.MediaPlayer1PostClick(
  Sender: TObject; Button: TMPBtnType);
begin
  if MediaPlayer1.Mode = mpPlaying then
    Timer1.Enabled := True
  else
    Timer1.Enabled := False;
  CheckPosition;
end;
```

You can also use the Go button to jump to the track selected in the SpinEdit component, where the `MaxValue` property is set by the `CheckDisk` method. Here is the code I've written:

```
procedure TForm1.GoButtonClick(Sender: TObject);
var
  Playing: Boolean;
begin
  Playing := (MediaPlayer1.Mode = mpPlaying);
  if Playing then
    MediaPlayer1.Stop;
  MediaPlayer1.Position :=
    MediaPlayer1.TrackPosition[SpinEdit1.Value];
  CheckPosition;
  if Playing then
    MediaPlayer1.Play;
end;
```

A good extension to this program would be to connect it to a CD database with the title of each CD you own and the title of each track. (I would have done that if it hadn't been for the time it would have taken to enter the title and track of each of my disks.) Remember, anyway, that a similar program is already available in Windows.

---

## What's Next

In this chapter, we have seen how to add some audio and video capabilities to Delphi applications. We have seen how to add sound effects to respond to user actions. We have also seen examples of how to use the Media Player with sound files, video files, and an external device (an audio CD). With computers and CD-ROM players becoming faster every year, video is becoming an important feature of many applications. Don't underestimate this area of programming simply because you are writing *serious* business programs.



## EPILOGUE

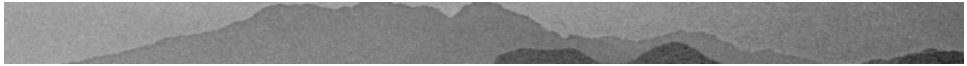
**D**elphi is a great programming environment. Now that you have learned about its core features, and seen the development of a number of programs, you might want to understand more about the language and fully master the internals of the VCL. This is where one of the books of the Mastering Delphi series I've written might help you. But you might also find more information in the other free e-books available on my web site.

As I mentioned in the Introduction, check the book page page, at

| <http://www.marcocantu.com/edelphi>

for corrections and updates; the site has also links to other Delphi sites, documentation, simple wizards and components, and hosts also a newsgroup, which is the preferred way to report problems about this book (in the related section). It is possible to read the newsgroup also directly on the web from my site, and also sign-up to a mailing list in which I'll announce future edition of this and other volumes.

Again, as discussed in the introduction donations are welcome, also in the form of code examples, chapters for this and other books... but also offering cash, buying a printed book, or attending one of the Delphi classes I set up.



# **MASTERING™ DELPHI™ 6**

---

Marco Cantù



San Francisco • Paris • Düsseldorf • Soest • London

Associate Publisher: Richard Mills  
Contracts and Licensing Manager: Kristine O'Callaghan  
Acquisitions Editor: Denise Santoro Lincoln  
Developmental Editors: Diane Lowery and Denise Santoro Lincoln  
Editor: Pete Gaughan  
Production Editor: Leslie E. H. Light  
Technical Editors: Danny Thorpe and Eddie Churchill  
Book Designer: Robin Kibby  
Graphic Illustrator: Tony Jonick  
Electronic Publishing Specialist: Kris Warrenburg, Cyan Design  
Proofreaders: Nanette Duffy, Amey Garber, Jennifer Greiman,  
Emily Hsuan, Laurie O'Connell, Nancy Riddiough  
Indexer: Ted Laux  
CD Coordinator: Christine Harris  
CD Technician: Kevin Ly  
Cover Designer: Design Site  
Cover Illustrator/Photographer: Sergie Loobkoff

Copyright © 2001 SYBEX Inc., 1151 Marina Village Parkway,  
Alameda, CA 94501. World rights reserved. The author created  
reusable code in this publication expressly for reuse by readers.  
Sybex grants readers limited permission to reuse the code found  
in this publication or its accompanying CD-ROM so long as the  
author is attributed in any application containing the reusable  
code and the code itself is never distributed, posted online by  
electronic transmission, sold, or commercially exploited as a  
stand-alone product. Aside from this specific exception concern-  
ing reusable code, no part of this publication may be stored in a  
retrieval system, transmitted, or reproduced in any way, including  
but not limited to photocopy, photograph, magnetic, or other  
record, without the prior agreement and written permission of  
the publisher.

Library of Congress Card Number: 2001088115  
ISBN: 0-7821-2874-2

SYBEX and the SYBEX logo are either registered trademarks or  
trademarks of SYBEX Inc. in the United States and/or other  
countries.

Mastering is a trademark of SYBEX Inc.  
Screen reproductions produced with Collage Complete.  
Collage Complete is a trademark of Inner Media Inc.

The CD interface was created using Macromedia Director,  
Copyright © 1994, 1997-1999 Macromedia Inc. For more infor-  
mation on Macromedia and Macromedia Director, visit  
<http://www.macromedia.com>.

TRADEMARKS: SYBEX has attempted throughout this book to  
distinguish proprietary trademarks from descriptive terms by fol-  
lowing the capitalization style used by the manufacturer.  
The author and publisher have made their best efforts to prepare  
this book, and the content is based upon final release software  
whenever possible. Portions of the manuscript may be based upon  
pre-release versions supplied by software manufacturer(s). The

author and the publisher make no representation or warranties of  
any kind with regard to the completeness or accuracy of the con-  
tents herein and accept no liability of any kind including but not  
limited to performance, merchantability, fitness for any particular  
purpose, or any losses or damages of any kind caused or alleged to  
be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

## Software License Agreement: Terms and Conditions

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the “Software”) to be used in connection with the book. SYBEX hereby grants to you a license to use the Software, subject to the terms that follow. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of SYBEX unless otherwise indicated and is protected by copyright to SYBEX or other copyright owner(s) as indicated in the media files (the “Owner(s)”). You are hereby granted a single-user license to use the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of SYBEX and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties (“End-User License”), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

### Reusable Code in This Book

The author created reusable code in this publication expressly for reuse for readers. Sybex grants readers permission to reuse for any purpose the code found in this publication or its accompanying CD-ROM so long as the author is attributed in any application containing the reusable code, and the code itself is never sold or commercially exploited as a stand-alone product.

### Software Support

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material but they are not supported by SYBEX. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate readme files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, SYBEX bears no responsibility. This notice concerning support for the Software is provided for your information only. SYBEX is not the agent or principal of the Owner(s), and SYBEX is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

## Warranty

SYBEX warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from SYBEX in any other form or media than that enclosed herein or posted to [www.sybex.com](http://www.sybex.com). If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

SYBEX Inc.  
Customer Service Department  
1151 Marina Village Parkway  
Alameda, CA 94501  
(510) 523-8233  
Fax: (510) 523-2373  
e-mail: [info@sybex.com](mailto:info@sybex.com)  
WEB: [HTTP://WWW.SYBEX.COM](http://WWW.SYBEX.COM)

After the 90-day period, you can obtain replacement media of identical format by sending us the defective disk, proof of purchase, and a check or money order for \$10, payable to SYBEX.

## Disclaimer

SYBEX makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will SYBEX, its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, SYBEX further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by SYBEX reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

## Shareware Distribution

This Software may contain various programs that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

## Copy Protection

The Software in whole or in part may or may not be copy-protected or encrypted. However, in all cases, reselling or redistributing these files without authorization is expressly forbidden except as specifically provided for by the Owner(s) therein.

*To Lella, the love of my life,  
and Benedetta, our love come to life.*

# ACKNOWLEDGMENTS

---

**T**his edition of *Mastering Delphi* marks the seventh year of the Delphi era, as it took Borland two years to release the latest incarnation of Delphi (along with its Linux twin, Kylix). As it has for many other programmers, Delphi has been my primary interest throughout these years; and writing, consulting, teaching, and speaking at conferences about Delphi have absorbed more and more of my time, leaving other languages and programming tools in the dust of my office. Because my work and my life are quite intertwined, many people have been involved in both, and I wish I had enough space and time to thank them all as they deserve. Instead, I'll just mention a few particular people and say a warm "Thank You" to the entire Delphi community (especially for the Spirit of Delphi 1999 Award I've been happy to share with Bob Swart).

The first official thanks are for the Borland programmers and managers who made Delphi possible and continue to improve it: Chuck Jazdzewski, Danny Thorpe, Eddie Churchill, Allen Bauer, Steve Todd, Mark Edington, Jim Tierney, Ravi Kumar, Jörg Weingarten, Anders Ohlsson, and all the others I have not had a chance to meet. I'd also like to give particular mention to my friends Ben Riga (the current Delphi product manager), John Kaster and David Intersimone (at Borland's Developer Relations), and others who have worked at Borland, including Charlie Calvert, Zack Urlocker and Nan Borreson.

The next thanks are for the Sybex editorial and production crew, many of whom I don't even know. Special thanks go to Pete Gaughan, Leslie Light, Denise Santoro Lincoln, and Diane Lowery; I'd also like to thank Richard Mills, Kristine O'Callaghan, and Kris Warrenburg.

This edition of *Mastering Delphi* has once again had an incredibly picky and detailed review from Delphi R&D team member Danny Thorpe. His highlights and comments in this and past editions have improved the book in all areas: technical content, accuracy, examples, and even readability. Thanks a lot. Previous editions also had special contributions: Tim Gooch worked on Part V for *Mastering Delphi 4*, and Giuseppe Madaffari contributed database material for the Delphi 5 edition. For this edition, Guy Smith-Ferrier rewrote the chapter on ADO, and Nando Dessena helped me with the InterBase chapter. Many improvements to the text and sample programs were suggested by technical reviewers of past editions (Juancarlo Añez, Ralph Friedman, Tim Gooch, and Alain Tadros) and in other reviews over the years by Bob Swart, Giuseppe Madaffari, and Steve Tendon.

Special thanks go to my friends Bruce Eckel, Andrea Provaglio, Norm McIntosh, Johanna and Phil of the BUG-UK, Ray Konopka, Mark Miller, Cary Jensen, Chris Frizelle of *The Delphi Magazine*, Foo Say How, John Howe, Mike Orriss, Chad “Kudzu” Hower, Dan Miser, Marco Miotti, and the entire D&D Team (Paolo, Andrea, Uberto, Nando, Giuseppe, and Mr. Coke). Also, a very big “Thank You” to all the attendees of my Delphi programming courses, seminars, and conferences in Italy, the United States, France, the United Kingdom, Singapore, the Netherlands, Germany, Sweden...

My biggest thanks go to my wife Lella who had to endure yet another many-months-long book-writing session and too many late nights (after spending the evenings with our daughter, Benedetta—I’ll thank her with a hug, as Daddy’s book looks quite boring to her). Many of our friends (and their kids) provided healthy breaks in the work: Sandro and Monica with Luca, Stefano and Elena, Marco and Laura with Matteo, Bianca, Luca and Elena with Tommaso, Chiara and Daniele with Leonardo, Laura, Vito and Marika with Sofia. Our parents, brothers, sisters, and their families were very supportive, too. It was nice to spend some of our free time with them and our six nephews—Matteo, Andrea, Giacomo, Stefano, Andrea, and Pietro.

Finally, I would like to thank all of the people, many of them unknown, who enjoy life and help to build a better world. If I never stop believing in the future and in peace, it is also because of them.

# INTRODUCTION

---

**T**he first time Zack Urlocker showed me a yet-to-be-released product code-named Delphi, I realized that it would change my work—and the work of many other software developers. I used to struggle with C++ libraries for Windows, and Delphi was and still is the best combination of object-oriented programming and visual programming for Windows.

Delphi 6 simply builds on this tradition and on the solid foundations of the VCL to deliver another astonishing and all-encompassing software development tool. Looking for database, client/server, multitier, intranet, or Internet solutions? Looking for control and power? Looking for fast productivity? With Delphi 6 and the plethora of techniques and tips presented in this book, you'll be able to accomplish all this.

## Six Versions and Counting

Some of the original Delphi features that attracted me were its form-based and object-oriented approach, its extremely fast compiler, its great database support, its close integration with Windows programming, and its component technology. But the most important element was the Object Pascal language, which is the foundation of everything else.

Delphi 2 was even better! Among its most important additions were these: the Multi-Record Object and the improved database grid, OLE Automation support and the variant data type, full Windows 95 support and integration, the long string data type, and Visual Form Inheritance. Delphi 3 added to this the code insight technology, DLL debugging support, component templates, the TeeChart, the Decision Cube, the WebBroker technology, component packages, ActiveForms, and an astonishing integration with COM, thanks to interfaces.

Delphi 4 gave us the AppBrowser editor, new Windows 98 features, improved OLE and COM support, extended database components, and many additions to the core VCL classes, including support for docking, constraining, and anchoring controls. Delphi 5 added to the picture many more improvements of the IDE (too many to list here), extended database support (with specific ADO and InterBase datasets), an improved version of MIDAS with Internet support, the TeamSource version-control tool, translation capabilities, the concept of frames, and new components.



Now Delphi 6 adds to all these features support for cross-platform development with the new Component Library for Cross-Platform (CLX), an extended run-time library, the new dbExpress database engine, Web services and exceptional XML support, a powerful Web development framework, more IDE enhancements, and a plethora of new components and classes, as you'll see in the following pages.

Delphi is a great tool, but it is also a complex programming environment that involves many elements. This book will help you master Delphi programming, including the Object Pascal language, Delphi components (both using the existing ones and developing your own), database and client/server support, the key elements of Windows and COM programming, and Internet and Web development.

You do not need in-depth knowledge of any of these topics to read this book, but you do need to know the basics of Pascal programming. Having some familiarity with Delphi will help you considerably, particularly after the introductory chapters. The book starts covering its topics in depth immediately; much of the introductory material from previous editions has been removed. Some of this material and an introduction to Pascal is available on the companion CD-ROM and on my Web site and can be a starting point if you are not confident with Delphi basics. Each new Delphi 6 feature is covered in the relevant chapters throughout the book.

## The Structure of the Book

The book is divided into four parts:

- Part I, “Foundations,” introduces new features of the Delphi 6 Integrated Development Environment (IDE) in Chapter 1, then moves to the Object Pascal language and to the run-time library (RTL) and Visual Component Library (VCL), providing both foundations and advanced tips.
- Part II, “Visual Programming,” covers standard components, Windows common controls, graphics, menus, dialogs, scrolling, docking, multipage controls, Multiple Document Interface, the Action List and Action Manager architectures, and many other topics. The focus is on both the VCL and CLX libraries. The final chapters discuss the development of custom components and the use of libraries and packages.
- Part III, “Database Programming,” covers plain database access, in-depth coverage of the data-aware controls, client/server programming, dbExpress, InterBase, ADO and dbGo, DataSnap (or MIDAS), and the development of custom data-aware controls and data sets.

- Part IV, “Beyond Delphi: Connecting with the World,” first discusses COM, OLE Automation, and COM+. Then it moves to Internet programming, covering TCP/IP sockets, Internet protocols and Indy, Web server-side extensions (with WebBroker and WebSnap), XML, and the development of Web services.

As this brief summary suggests, the book covers topics of interest to Delphi users at nearly all levels of programming expertise, from “advanced beginners” to component developers.

In this book, I’ve tried to skip reference material almost completely and focus instead on techniques for using Delphi effectively. Because Delphi provides extensive online documentation, to include lists of methods and properties of components in the book would not only be superfluous, it would also make it obsolete as soon as the software changes slightly. I suggest that you read this book with the Delphi Help files at hand, to have reference material readily available.

However, I’ve done my best to allow you to read the book away from a computer if you prefer. Screen images and the key portions of the listings should help in this direction. The book uses just a few conventions to make it more readable. All the source code elements, such as keywords, properties, classes, and functions, appear in this font, and code excerpts are formatted as they appear in the Delphi editor, with boldfaced keywords and italic comments and strings.

## Free Source Code on CD (and the Web)

This book focuses on examples. After the presentation of each concept or Delphi component, you’ll find a working program example (sometimes more than one) that demonstrates how the feature can be used. All told, there are about 300 examples presented in the book. These programs are directly available on the companion CD-ROM. The same material is also available on my Web site ([www.marcocantu.com](http://www.marcocantu.com)), where you’ll also find updates and examples from past editions. Inside the back cover of the book, you’ll find more information about the CD. Most of the examples are quite simple and focus on a single feature. More complex examples are often built step-by-step, with intermediate steps including partial solutions and incremental improvements.

### NOTE

Some of the database examples also require you to have the Delphi sample database DBDEMOS installed; it is part of the default Delphi installation. Others require the InterBase EMPLOYEE sample database.

Beside the source code files, the CD hosts the ready-to-use compiled programs. There is also an HTML version of the source code, with full syntax highlighting, along with a com-

plete cross-reference of keywords and identifiers (class, function, method, and property names, among others). The cross-reference is an HTML file, so you'll be able to use your browser to easily find all the programs that use a Delphi keyword or identifier you're looking for (not a full search engine, but close enough).

The directory structure of the sample code is quite simple. Basically, each chapter of the book has its own folder, with a subfolder for each example (e.g., 06\Borders). In the text, the examples are simply referenced by name (e.g., Borders).

---

**TIP**

To change an example, first copy it (or the entire `md6code` folder) to your hard disk, but before opening it remember to set the read-only flag to False (it is True by default on the read-only media)

---

**NOTE**

Be sure to read the source code archive's Readme file, which contains important information about using the software legally and effectively.

## How to Reach the Author

If you find any problems in the text or examples in this book, both the publisher and I would be happy to hear from you. Besides reporting errors and problems, please give us your unbiased opinion of the book and tell us which examples you found most useful and which you liked least. There are several ways you can provide this feedback:

- On the Sybex Web site ([www.sybex.com](http://www.sybex.com)), you'll find updates to the text or code as necessary. To comment on this book, click the Contact Sybex link and then choose Book Content Issues. This link displays a form where you can enter your comments.
- My own Web site ([www.marcocantu.com](http://www.marcocantu.com)) hosts further information about the book and about Delphi, where you might find answers to your questions. The site has news and tips, technical articles, free online books, white papers, Delphi links, and my collection of Delphi components and tools.
- I have also set up a newsgroup section specifically devoted to my books and to general Delphi Q&A. Refer to my Web site for a list of the newsgroup areas and for the instructions to subscribe to them. (In fact, these newsgroups are totally free but require a login password.) The newsgroups can also be accessed via a Web interface you can find on my site.
- Finally, you can reach me via e-mail at [marco@marcocantu.com](mailto:marco@marcocantu.com). For technical questions, please try using the newsgroups first, as you might get answers earlier and from multiple people. My mailbox is usually quite full and, regretfully, I cannot reply promptly to every request. (Please write to me in English or Italian.)



**PART I**

---

# Foundations

- **Chapter 1: The Delphi 6 IDE**
- **Chapter 2: The Object Pascal Language: Classes and Objects**
- **Chapter 3: The Object Pascal Language: Inheritance and Polymorphism**
- **Chapter 4: The Run-Time Library**
- **Chapter 5: Core Library Classes**

# The Delphi 6 IDE

- Object TreeView and Designer view
- The AppBrowser editor
- The code insight technology
- Designing forms
- The Project Manager
- Delphi files

In a visual programming tool such as Delphi, the role of the environment is at times even more important than the programming language. Delphi 6 provides many new features in its visual development environment, and this chapter covers them in detail. This chapter isn't a complete tutorial but mainly a collection of tips and suggestions aimed at the average Delphi user. In other words, it's not for newcomers. I'll be covering the new features of the Delphi 6 Integrated Development Environment (IDE) and some of the advanced and little-known features of previous versions as well, but in this chapter I won't provide a step-by-step introduction. Throughout this book, I'll assume you already know how to carry out the basic hands-on operations of the IDE, and all the chapters after this one focus on programming issues and techniques.

If you *are* a beginning programmer, don't be afraid. The Delphi Integrated Development Environment is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi CD) with a tutorial that introduces the development of Delphi applications. You can also find a step-by-step introduction to the Delphi IDE on my Web site, <http://www.marcocantu.com>. The short online book *Essential Delphi* is based on material from the first chapters of earlier editions of *Mastering Delphi*.

## Editions of Delphi 6

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

- The “Personal” edition is aimed at Delphi newcomers and casual programmers and has support for neither database programming nor any of the other advanced features of Delphi 6.
- The “Professional” edition is aimed at professional developers. It includes all the basic features, plus database programming support (including ADO support), basic Web server support (WebBroker), and some of the external tools. This book generally assumes you are working with at least the Professional edition.
- The “Enterprise” edition is aimed at developers building enterprise applications. It includes all the new XML and advanced Web services technologies, internationalization, three-tier architecture, and many other tools. Some chapters of this book cover features included only in Delphi Enterprise; these sections are specifically identified.

**NOTE**

In the past, some of the features of Delphi Enterprise have been available as an “up-sell” to owners of Delphi Professional. This might also happen for this version.

Besides the different editions available, there are ways to customize the Delphi environment. In the screen illustrations throughout the book, I’ve tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which might be reflected in some of the screen shots.

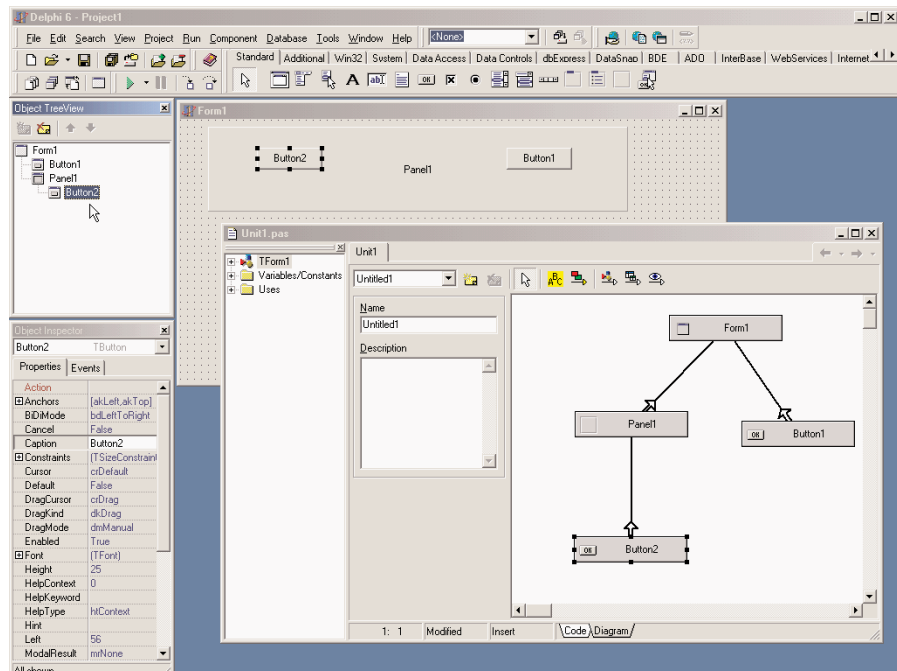
## The Delphi 6 IDE

The Delphi 6 IDE includes large and small changes that will really improve a programmer’s productivity. Among the key features are the introduction of the Object TreeView for every designer, an improved Object Inspector, extended code completion, and loadable views, including diagrams and HTML.

Most of the features are quite easy to grasp, but it’s worth examining them with some care so that you can start using Delphi 6 at its full potential right away. You can see an overall image of Delphi 6 IDE, highlighting some of the new features, in Figure 1.1.

**FIGURE 1.1:**

The Delphi 6 IDE: Notice the Object TreeView and the Diagram view.



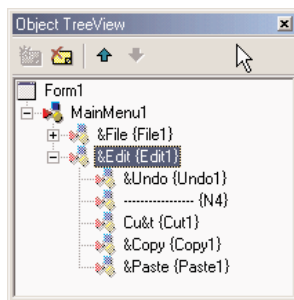


## The Object TreeView

Delphi 5 introduced a TreeView for data modules, where you could see the relations among nonvisual components, such as datasets, fields, actions, and so on. Delphi 6 extends the idea by providing an Object TreeView for every designer, including plain forms. The Object TreeView is placed by default above the Object Inspector; use the View > Object TreeView command in case it is hidden.

The Object TreeView shows all of the components and objects on the form in a tree, representing their relations. The most obvious is the parent/child relation: Place a panel on a form, a button inside it and one outside of the panel. The tree will show the two buttons, one under the form and the other under the panel, as in Figure 1.1. Notice that the TreeView is synchronized with the Object Inspector and Form Designer, so as you select an item and change the focus in any one of these three tools, the focus changes in the other two tools.

Besides parent/child, the Object TreeView shows also other relations, such as owner/owned, component/subobject, collection/item, plus various specific ones, including dataset/connection and data source/dataset relations. Here, you can see an example of the structure of a menu in the tree.



At times, the TreeView also displays “dummy” nodes, which do not correspond to an actual object but do correspond to a predefined one. As an example of this behavior, drop a Table component (from the BDE page) and you’ll see two grayed icons for the session and the alias. Technically, the Object TreeView uses gray icons for components that do not have design-time persistence. They are real components (at design time and at run time), but because they are default objects that are constructed at run time and have no persistent data that can be edited at design time, the Data Module Designer does not allow you to edit their properties. If you drop a Table on the form, you’ll also see items with a red question mark enclosed in a yellow circle next to them. This symbol indicates partially undefined items (there used to be a red square around those items in Delphi 5).

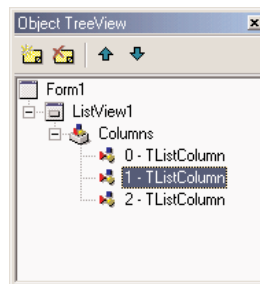


The Object TreeView supports multiple types of *dragging*:

- You can select a component from the palette (by clicking it, not actually dragging it), move the mouse over the tree, and click a component to drop it there. This allows you to drop a component in the proper container (form, panel, and others) regardless of the fact that its surface might be totally covered by other components, something that prevents you from dropping the component in the designer without first rearranging those components.
- You can drag components within the TreeView—for example, moving a component from one container to another—something that, with the Form Designer, you can do only with cut and paste techniques. Moving instead of cutting provides the advantage that if you have connections among components, these are not lost, as happens when you delete the component during the cut operation.
- You can drag components from the TreeView to the Diagram view, as we'll see later.

Right-clicking any element of the TreeView displays a shortcut menu similar to the component menu you get when the component is in a form (and in both cases, the shortcut menu may include items related to the custom component editors). You can even delete items from the tree.

The TreeView doubles also as a collection editor, as you can see here for the Columns property of a ListView control. In this case, you can not only rearrange and delete items, but also add new items to the collection.

**TIP**

You can print the contents of the Object TreeView for documentation purposes. Simply select the window and use the File > Print command, as there is no Print command in the shortcut menu.

## Loadable Views

Another important change has taken place in the Code Editor window. For any single file loaded in the IDE, the editor can now show multiple views, and these views can be defined programmatically and added to the system, then loaded for given files—hence the name *loadable* views.

The most frequently used view is the Diagram page, which was already available in Delphi 5 data modules, although it was less powerful. Another set of views is available in Web applications, including an HTML Script view, an HTML Result preview, and many others discussed in Chapter 22.

## The Diagram View

Along with the TreeView, another feature originally introduced in Delphi 5 Data Modules and now available for every designer is the Diagram view. This view shows dependencies among components, including parent/child relations, ownership, linked properties, and generic relations. For dataset components, it also supports master/detail relations and lookup connections. You can even add your comments in text blocks linked to specific components.

The Diagram is not built automatically. You must drag components from the TreeView to the diagram, which will automatically display the existing relations among the components you drop there. In Delphi 6, you can now select multiple items from the Object TreeView and drag them all at once to the Diagram page.

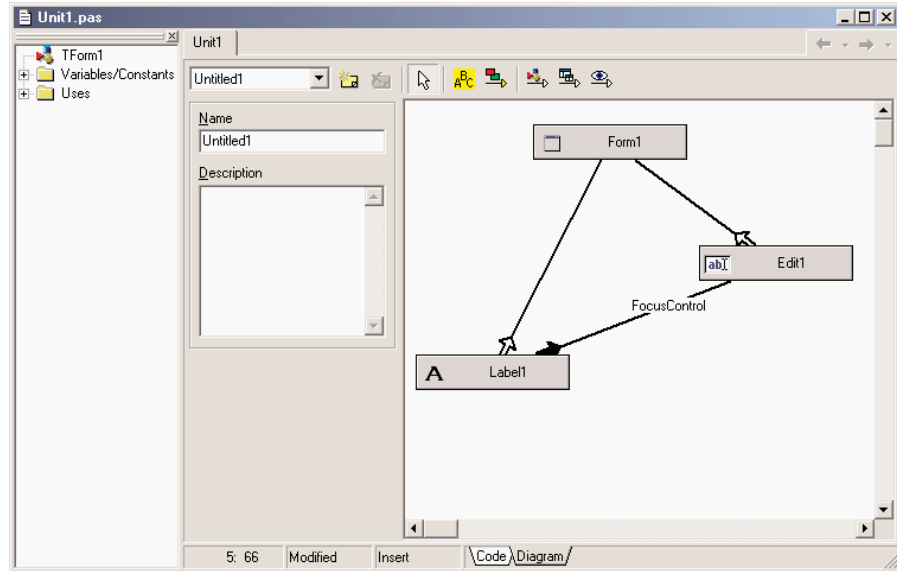
What's nice is that you can set properties by simply drawing arrows between the components. For example, after moving an edit and a label to Diagram, you can select the Property Connector icon, click the label, and drag the mouse cursor over the edit. When you release the mouse button, the Diagram will set up a property relation based on the `FocusControl` property, which is the only property of the label referring to an edit control. This situation is depicted in Figure 1.2.

As you can see, setting properties is *directional*: If you drag the property relation line from the edit to the label, you end up trying to use the label as the value of a property of the edit box. Because this isn't possible, you'll see an error message indicating the problem and offering to connect the components in the opposite way.

In Delphi 6, the Diagram view allows you to create multiple diagrams for each Delphi unit—that is, for each form or data module. Simply give a name to the diagram and possibly add a description, click the New Diagram button, prepare another diagram, and you'll be able to switch back and forth between diagrams using the combo box available in the toolbar of the Diagram view.

**FIGURE 1.2:**

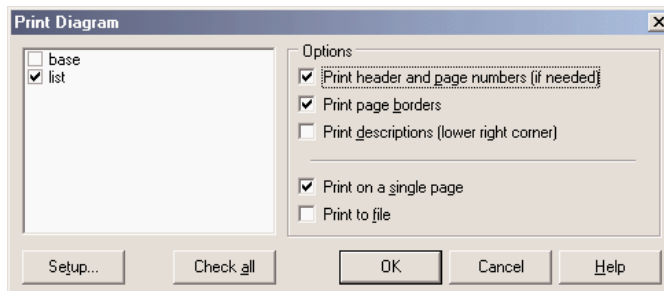
The Diagram view allows you to connect components using the Property connector.



Although you can use the Diagram view to set up relations, its main role is to document your design. For this reason, it is important to be able to print the content of this view. Using the standard File > Print command while the Diagram is active, Delphi prompts you for options, as you can see in Figure 1.3, allowing you to customize the output in many ways.

**FIGURE 1.3:**

The Print Options for the Diagram view



The information in the Data Diagram view is saved in a separate file, not as part of the DFM file. Delphi 5 used design-time information (DTI) files, which had a structure similar to INI files. Delphi 6 can still read the older .DTI format, but uses the new Delphi Diagram Portfolio format (.DDP). These files apparently use the DFM binary format (or a similar one), so they are not editable as text. All of these files are obviously useless at run time (it makes no sense to include them in the compilation of the executable file).

## An IDE for Two Libraries

Another very important change I just want to introduce here is the fact that Delphi 6, for the first time, allows you to use two different component libraries, VCL (Visual Components Library) and CLX (Component Library for Cross-Platform). When you create a new project, you simply choose which of the two libraries you want to use, starting with the File > New > Application command for a classic VCL-based Windows program and with the File > New > CLX Application command for a new CLX-based portable application.

Creating a new project or opening an existing one, the Component Palette is rearranged to show only the controls related to the current library (although most of them are actually shared). This topic is fully covered in Chapter 6, so I don't want to get into the details here; I'll just underline that you can use Delphi 6 to build applications you can compile right away for Linux using Kylix. The effect of this change on the IDE is really quite large, as many things “under the hood” had to be reengineered. Only programmers using the ToolsAPI and other advanced elements will notice all these internal differences, as they are mostly transparent to most users.

## Smaller Enhancements

Besides this important change and others I'll discuss in later sections, such as the update of the Object Inspector and of code completion, there are small (but still quite important) changes in the Delphi 6 IDE. Here is a list of these changes:

- There is a new Window menu in the IDE. This menu lists the open windows, something you could obtain in the past using the Alt+0 keys. This is really very handy, as windows often end up behind others and are hard to find. (Thanks, Borland, for listening to this and other simple but effective requests from users.)

### TIP

Two entries of the Main Window registry section of Delphi (under `\Software\Borland\Delphi\6.0` for the current user) allow you to hide this menu and disable its alphabetic sort order. This registry keys use strings (in place of Boolean values) where “-1” indicates true and “0” false.

- The File menu doesn't include specific items for creating new forms or applications. These commands have been increased in number and grouped under the File > New secondary menu. The Other command of this menu opens the New Item dialog box (the Object Repository) as the File > New command did in the past.
- The Component Palette local menu has a submenu listing all of the palette pages in alphabetic order. You can use it to change the active page, particularly when it is not visible on the screen.

**TIP**

The order of the entries in the Tabs submenu of the Component Palette local menu can be set in the same order as the palette itself, and not sorted alphabetically. This is accomplished by setting to "0" (false) the value of the Sort Palette Tabs Menu key of the Main Window registry section of Delphi (under `\Software\Borland\Delphi\6.0` for the current user).

- There is a new toolbar, the Internet toolbar, which is initially disabled. This toolbar supports WebSnap applications.

## Updated Environment Options Dialog Box

Quite a few small changes relate to the commonly used Environment Options dialog box. The pages of this dialog box have been rearranged, moving the Form Designer options from the Preferences page to the new Designer page. There are also a few new options and pages:

- The Preferences page of the Environment Options dialog box has a new check box that prevents Delphi windows from automatically docking with each other. This is a very welcome addition!
- A new page, Environment Variables, allows you to see system environment variables (such as the standard path names and OS settings) and set user-defined variables. The nice point is that you can use both system- and user-defined environment variables in each of the dialog boxes of the IDE—for example, you can avoid hard-coding commonly used path names, replacing them with a variable. In other words, the environment variables work similarly to the \$DELPHI variable, referring to Delphi's base directory, but can be defined by the user.
- Another new page is called Internet. In this page, you can choose the default file extensions used for HTML and XML files (mainly by the WebSnap framework) and also associate an external editor with each extension.

## Delphi Extreme Toys

At times, the Delphi team comes up with small enhancements of the IDE that aren't included in the product because they either aren't of general use or will require time to be improved in quality, user interface, or robustness. Some of these internal wizards and IDE extensions have now been made available, with the collective name of Delphi Extreme Toys, to registered Delphi 6 users. You should automatically get this add-on as you register your copy of the product (online or through a Borland office).

There isn't an official list of the content of the Extreme Toys, as Borland plans to keep extending them. The initial release includes an IDE-based search engine for seeking answers on Delphi across the Internet, a wizard for turning on and off specific compiler warnings,

and an “invokamatic” wizard for accelerating the creation of Web services. The Extreme Toys will, in essence, be *unofficial* wizards, code utilities, and components from the Delphi team—or useful stuff from various people.

## Recent IDE Additions

Delphi 5 provided a huge number of new features to the IDE. In case you’ve only used versions of Delphi prior to 5, or need to brush up on some useful added information, this is a short summary of the most important of the features introduced in Delphi 5.

### Saving the Desktop Settings

The Delphi IDE allows programmers to customize it in various ways—typically, opening many windows, arranging them, and docking them to each other. However, programmers often need to open one set of windows at design time and a different set at debug time. Similarly, programmers might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

For this reason, Delphi allows you to save a given arrangement of IDE windows (called a *desktop*) with a name and restore it easily. Also, you can make one of these groupings your default debugging setting, so that it will be restored automatically when you start the debugger. All these features are available in the Desktops toolbar. You can also work with desktop settings using the View > Desktops menu.

Desktop setting information is saved in DST files, which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its new property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78
```

```
[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1

[AlignmentPalette]
Create=1
Visible=0
...
```

Desktop settings override project settings. This helps eliminate the problem of moving a project between machines (or between developers) and having to rearrange the windows to your liking. Delphi 5 separates per-user and per-machine preferences from the project settings, to better support team development.

**TIP**

If you open Delphi and cannot see the form or other windows, I suggest you try checking (or deleting) the desktop settings. If the project desktop was last saved on a system running in a high-resolution video mode (or a multimonitor configuration) and opened on a different system with lower screen resolution or fewer monitors, some of the windows in the project might be located off-screen on the lower-resolution system. The simplest ways to fix that are either to load your own named desktop configuration after opening the project, thus overriding the project desktop settings, or just delete the DST file that came with the project files.

## The To-Do List

Another feature added in Delphi 5 was the to-do list. This is a list of tasks you still have to do to complete a project, a collection of notes for the programmer (or programmers, as this tool can be very handy in a team). While the idea is not new, the key concept of the to-do list in Delphi is that it works as a two-way tool.

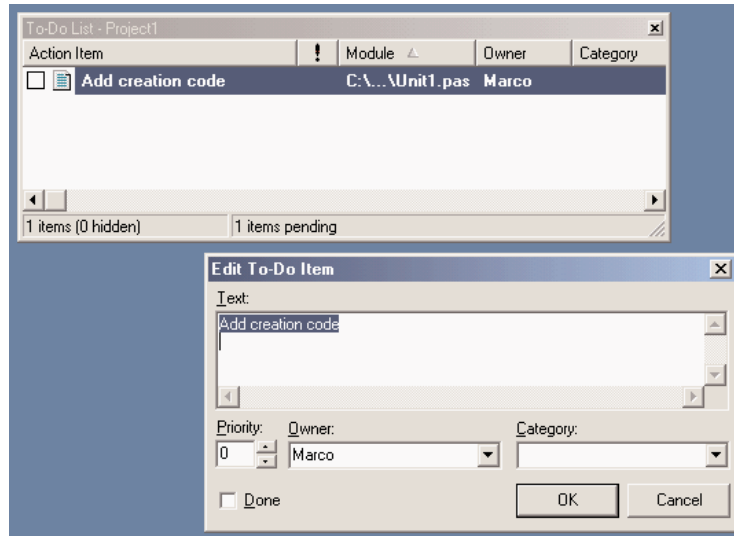
In fact, you can add or modify to-do items by adding special `TODO` comments to the source code of any file of a project; you'll then see the corresponding entries in the list. But you can also visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look like in the source code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    // TODO -oMarco: Add creation code
end;
```

The same item can be visually edited in the window shown in Figure 1.4.

**FIGURE 1.4:**

The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code.



The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the Ctrl+A key combination in the To-Do List window or right-click in the window and select Add from the shortcut menu. These items are saved in a special file with the .TODO extension.

You can use multiple options with a TODO comment. You can use -o (as in the code excerpt above) to indicate the owner, the programmer who entered the comment; the -c option to indicate a category; or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

Delphi treats everything after the colon, up to the end of line or the closing brace, depending on the type of comment, as the text of the to-do item. Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from TODO to DONE. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, which can automatically collect to-do information from the source code files as you type them, sort and filter them, and export them to the Clipboard as plain text or an HTML table.



## The AppBrowser Editor

The editor included with Delphi hasn't changed recently, but it has many features that many Delphi programmers don't know and use. It's worth briefly examining this tool. The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor, and you can also open multiple editor windows. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Shift+Ctrl+Tab to move in the opposite direction).

### TIP

In Delphi 6, you can drag-and-drop the tabs with the unit names in the upper portion of the editor to change their order, so that you can use a single Ctrl+Tab to move between the units you are mostly interested in. The local menu of the editor has also a Pages command, which lists all of the available pages in a submenu, a handy feature when many units are loaded.

Several options affect the editor, located in the new Editor Properties dialog box. You have to go to the Preferences page of the Environment Options dialog box, however, to set the editor's AutoSave feature, which saves the source code files each time you run the program (preventing data loss in case the program crashes badly).

I won't discuss the various settings of the editor, as they are quite intuitive and are described in the online Help. A tip to remember is that using the Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. You can also copy text instead of moving it, by pressing the Ctrl key while dragging.

## The Code Explorer

The Code Explorer window, which is generally docked on the side of the editor, simply lists all of the types, variables, and routines defined in a unit, plus other units appearing in uses statements. For complex types, such as classes, the Code Explorer can list detailed information, including a list of fields, properties, and methods. All the information is updated as soon as you start typing in the editor. You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration.

### TIP

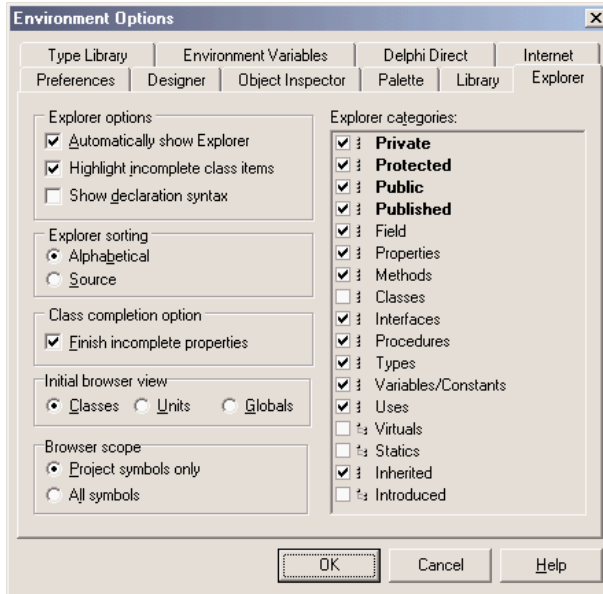
In Delphi 6 you can modify variables, properties, and method names directly in the Code Explorer.

While all that is quite obvious after you've used Delphi for a few minutes, some features of the Code Explorer are not so intuitive. One important point is that you have full control of the layout of the information, and you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer. Collapsing the tree can help you make your

selections more quickly. You can configure the Code Explorer by using the corresponding page of the Environment Options, as shown in Figure 1.5.

**FIGURE 1.5:**

You can configure the Code Explorer in the Environment Options dialog box.



Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view—it simply adds the node in the tree. For example, if you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer. On the contrary, the used units are listed as main nodes instead of being kept in the Uses folder. I generally disable the Types, Classes, and Variables selections.

Because each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, as this requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer, in fact, provides a very handy way of navigating the source code of a large unit. When you double-click a method in the Code Explorer, the focus moves to the definition in the class declaration (in the interface portion of the unit). You can use the Ctrl+Shift combination with the Up or Down arrow keys to jump from the definition of a method or procedure in the interface portion of a unit to its complete definition in the implementation portion (or back again).

**NOTE**

Some of the Explorer Categories shown in Figure 1.5 are used by the Project Explorer, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.

## Browsing in the Editor

Another feature of the AppBrowser editor is *Tooltip symbol insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the Visual Component Library (VCL).

**WARNING**

Although it may seem a good idea at first, you cannot use Tooltip symbol insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, in fact, the Tooltip won't appear.

The real bonus of this feature, however, is that you can turn it into a navigational aid. When you hold down the Ctrl key and move the mouse over the identifier, Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of Web browsers, and the pointer changes to a hand whenever it's positioned on the link.

For example, you can Ctrl+click the TLabel identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again as in a Web browser. You can also click the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

How can you jump directly to the VCL source code if it is not part of your project? The AppBrowser editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment.

## Class Completion

The third important feature of Delphi's AppBrowser editor is *class completion*, activated by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, as Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi also simplify life in a similar way for programmers who write a little extra code behind event handlers. The new code-generation feature, in fact, applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration:

```
public  
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }  
procedure TForm1.Hello(MessageText: string);  
begin  
end;
```

This is really handy, compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin...end` code for every method copied.

Class completion can also work the other way around. You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

## Code Insight

Besides the Code Explorer, class completion, and the navigational features, the Delphi editor supports the *code insight* technology. Collectively, the code insight techniques are based on a constant background parsing, both of the source code you write and of the source code of the system units your source code refers to.

Code insight comprises five capabilities: code completion, code templates, code parameters, Tooltip expression evaluation, and Tooltip symbol insight. This last feature was already covered in the section "Browsing in the Editor"; the other four will be discussed in the following subsections. You can enable, disable, and configure each of these features in the Code Insight page of the Editor Options dialog box.

## Code Completion

Code completion allows you to choose the property or method of an object simply by looking it up on a list or by typing its initial letters. To activate this list, you just type the name of an object, such as `Button1`, then add the dot, and wait. To force the display of the list, press `Ctrl+spacebar`; to remove it when you don't want it, press `Esc`. Code completion also lets you look for a proper value in an assignment statement.

In Delphi 6, as you start typing, the list filters its content according to the initial portion of the element you've inserted. The code completion list uses colors and shows more details to help you distinguish different items. Another new feature is that in the case of functions with parameters, parentheses are included in the generated code, and the parameters list hint is displayed immediately.

As you type `:=` after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click it to change the order of the items, sorting either by scope or by name, and you can also resize the window.

In Delphi 6, code completion also works in the interface section of a unit. If you press `Ctrl+spacebar` while the cursor is inside the class definition, you'll get a list of: virtual methods you can override (including abstract methods), the methods of implemented interfaces, the base class properties, and eventually system messages you can handle. Simply selecting one of them will add the proper method to the class declaration. In this particular case, the code completion list allows multiple selection.

### TIP

When the code you've written is incorrect, code insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific code insight errors in the Message pane (which must already be open; it doesn't open automatically to display compilation errors). To activate this feature, you need to set an undocumented registry entry, setting the string key `\Delphi\6.0\Compiling\ShowCodeInsightErrors` to the value '1'.

There are advanced features of Delphi 6 code completion that aren't easy to spot. One that I found particularly useful relates to the discovery of symbols in units not used by your project. As you invoke it (with `Ctrl+spacebar`) over a blank line, the list also includes symbols from common units (such as `Math`, `StrUtils`, and `DateUtils`) not already included in the `uses` statement of the current one. By selecting one of these *external* symbols, Delphi adds the unit to the `uses` statement for you. This feature (which doesn't work inside expressions) is driven by a customizable list of extra units, stored in the registry key `\Delphi\6.0\CodeCompletion\ExtraUnits`.

## Code Templates

Code templates allow you to insert one of the predefined code templates, such as a complex statement with an inner `begin...end` block. Code templates must be activated manually, by typing `Ctrl+J` to show a list of all of the templates. If you type a few letters (such as a keyword) before pressing `Ctrl+J`, Delphi will list only the templates starting with those letters.

You can add your own custom code templates, so that you can build your own shortcuts for commonly used blocks of code. For example, if you use the `MessageDlg` function often, you might want to add a template for it. In the Code Insight page of the Environment Options dialog box, click the Add button in the Code Template area, type in a new template name (for example, **mess**), type a description, and then add the following text to the template body in the Code memo control:

```
MessageDlg (' / ', mtInformation, [mbOK], 0);
```

Now every time you need to create a message dialog box, you simply type **mess** and then press `Ctrl+J`, and you get the full text. The vertical line (or pipe) character indicates the position within the source code where the cursor will be in the editor after expanding the template. You should choose the position where you want to start typing to complete the code generated by the template.

Although code templates might seem at first sight to correspond to language keywords, they are in fact a more general mechanism. They are saved in the `DELPHI32.DCI` file, so it should be possible to copy this file to make your templates available on different machines. Merging two code template files is not documented, though.

## Code Parameters

Code parameters display, in a hint or Tooltip window, the data type of a function's or method's parameters while you are typing it. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a pop-up hint window. To force the display of code parameters, you can press `Ctrl+Shift+spacebar`. As a further help, the current parameter appears in bold type.

## Tooltip Expression Evaluation

Tooltip expression evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor.

## More Editor Shortcut Keys

The editor has many more shortcut keys that depend on the editor style you've selected. Here are a few of the less-known shortcuts, most of which are useful:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a “gutter” margin on the side of the editor. To jump back to the bookmark, press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the facts that a new bookmark can override an existing one and that bookmarks are not persistent; they are lost when you close the file.
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is the one set by the Block Indent option in the **General** page of the **Editor Properties** dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R starts recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that might have little meaning once you've moved on to a different source code file. Editor macros are quite useful for performing multistep operations over and over again, such as reformatting source code or arranging data more legibly in source code.
- Holding down the Alt key, you can drag the mouse to select rectangular areas of the editor, not just consecutive lines and words.

## The Form Designer

Another Delphi window you'll interact with very often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, you can select a component directly with the mouse or through the Object Inspector, a handy feature when a control is behind another one or is very small. If one control covers another completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click the selected component. This will deselect the current component and select the form by default.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down `Ctrl`. Using arrow keys is particularly useful for fine-tuning an element's position. (The `Snap To Grid` option works only for mouse operations.) Similarly, by pressing the arrow keys while you hold down `Shift`, you can fine-tune the size of a component. (If you press `Shift+Ctrl` along with an arrow key, the component will be moved only at grid intervals.) Unfortunately, during these fine-tuning operations, the component hints with the position and size are not displayed.

To align multiple components or make them the same size, you can select several components and set the `Top`, `Left`, `Width`, or `Height` property for all of them at the same time. To select several components, you can click them with the mouse while holding down the `Shift` key, or, if all the components fall into a rectangular area, you can drag the mouse to “draw” a rectangle surrounding them. When you've selected multiple components, you can also set their relative position using the `Alignment` dialog box (with the `Align` command of the form's shortcut menu) or the `Alignment Palette` (accessible through the `View > Alignment Palette` menu command).

When you've finished designing a form, you can use the `Lock Controls` command of the `Edit` menu to avoid accidentally changing the position of a component in a form. This is particularly helpful, as `Undo` operations on forms are limited (only an `Undelete one`), but the setting is not persistent.

Among its other features, the `Form Designer` offers several `Tooltip` hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. `Delphi 6` offers extended hints, with details on the control position, size, tab order, and more. This is an addition to the `Show Component Captions` environment setting, which I keep active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the `Form Designer` by icons).
- As you move a component, the hint indicates the current position (the `Left` and `Top` properties).

Finally, you can save `DFM` (`Delphi Form Module`) files in plain text instead of the traditional binary resource format. You can toggle this option for an individual form with the `Form Designer's` shortcut menu, or you can set a default value for newly created forms in the



Designer page of the Environment Options dialog box. In the same page, you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box).

Having DFM files stored as text was a welcome addition in Delphi 5; it lets you operate more effectively with version-control systems. Programmers won't get a real advantage from this feature, as you could already open the binary DFM files in the Delphi editor with a specific command of the shortcut menu of the designer. Version-control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file.

In any case, note that if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFMs are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded faster).

---

**NOTE**

Text DFM files are more portable between versions of Delphi than their binary version. While an older version of Delphi might not accept a new property of a control in a DFM created by a newer version of Delphi, the older Delphis will still be able to read the rest of the text DFM file. If the newer version of Delphi adds a new data type, though, older Delphis will be unable to read the newer Delphi's binary DFMs at all. Even if this doesn't sound likely, remember that 64-bit operating systems are just around the corner. When in doubt, save in text DFM format. Also note that all versions of Delphi support text DFMs, using the command-line tool `Convert` in the `bin` directory.

## The Object Inspector in Delphi 6

Delphi 5 provided new features to the Object Inspector, and Delphi 6 includes even more additions to it. As this is a tool programmers use all the time, along with the editor and the Form Designer, its improvements are really significant.

The most important change in Delphi 6 is the ability of the Object Inspector to expand component references in-place. Properties referring to other components are now displayed in a different color and can be expanded by selecting the + symbol on the left, as it happens with internal subcomponents. You can then modify the properties of that other component without having to select it. See Figure 1.6 for an example.

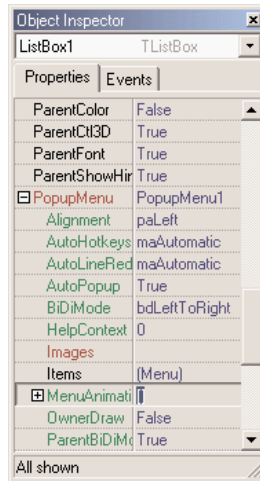
---

**NOTE**

This interface-expansion feature also supports subcomponents, as demonstrated by the new `LabeledEdit` control.

**FIGURE 1.6:**

A connected component (a pop-up menu) expanded in the Object Inspector while working on another component (a list box)

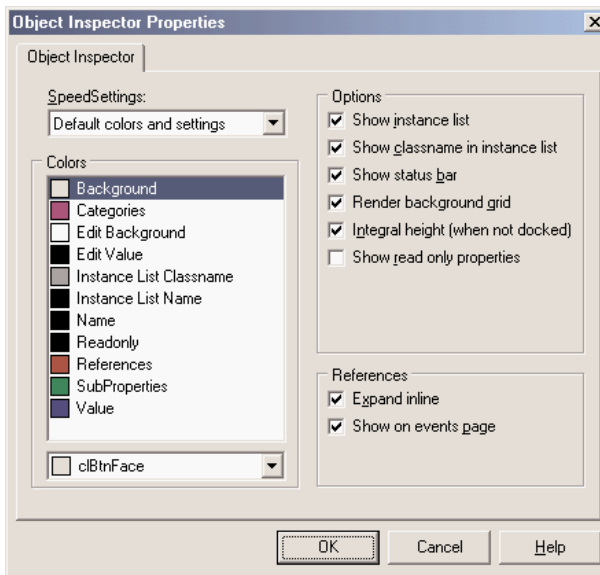
**TIP**

A related feature of the Object Inspector is the ability to select the component referenced by a property. To do this, double-click the property value with the left mouse button while keeping the Ctrl key pressed. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the MainMenu property of the form and Ctrl+double-clicking the value of this property. This selects the main menu indicated as the value of the property in the Object Inspector.

Here are some other relevant changes of the Object Inspector:

- The list at the top of the Object Inspector shows the type of the object and can be removed to save some space (and considering the presence of the Object TreeView).
- The properties that reference an object are now a different color and may be expanded without changing the selection.
- You can optionally also view read-only properties in the Object Inspector. Of course, they are grayed out.
- The Object Inspector has a new Properties dialog box (see Figure 1.7), which allows you to customize the colors of the various types of properties and the overall behavior of this window.

**FIGURE 1.7:**  
The new Object Inspector  
Properties dialog box



- Since Delphi 5, the drop-down list for a property can include graphical elements. This is used for properties such as Color and Cursor, and is particularly useful for the ImageIndex property of components connected with an ImageList.

#### NOTE

Interface properties can now be configured at design time using the Object Inspector. This makes use of the Interfaced Component Reference model introduced in Kylix/Delphi 6, where components may implement and hold references to interfaces as long as the interfaces are implemented by components. Interfaced Component References work like plain old component references, except that interface properties can be bound to any component that implements the necessary interface. Unlike component properties, interface properties are not limited to a specific component type (a class or its derived classes). When you click the drop-down list in the Object Inspector editor for an interface property, all components on the current form (and linked forms) that implement the interface are shown.

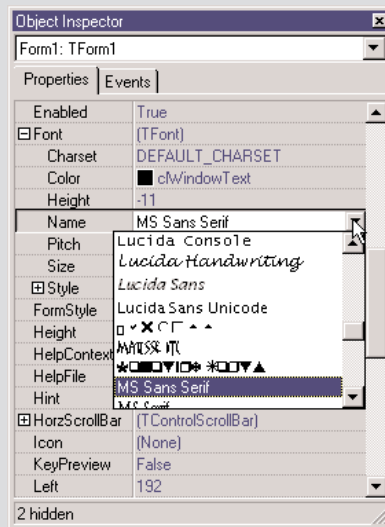
### Drop-Down Fonts in the Object Inspector

The Delphi Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting, corresponding to the Name subproperty of the Font property. This capability is actually built into Delphi, but it has been disabled because most computers have a large number of fonts installed and rendering

*Continued on next page*

them can really slow down the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplayFontNames` global variable of the new `VCLEditors` unit. I've done this in the `OiFontPk` package, which you can find among the program examples for this chapter on the companion CD-ROM.

Once this package is installed, you can move to the `Font` property of any component and use the graphical Name drop-down menu, as displayed here:



There is a second, more complex customization of the Object Inspector that I like and use frequently: a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations. You can find the package to install custom fonts in the Object Inspector on my Web site, [www.marcocantu.com](http://www.marcocantu.com).

## Property Categories

Delphi 5 also introduced the idea of property categories, activated by the `Arrange` option of the local menu of the Object Inspector. If you set it, properties won't be listed alphabetically but arranged by group, with each property possibly appearing in multiple groups.

Categories have the benefit of reducing the complexity of the Object Inspector. You can use the `View` submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories).

## Secrets of the Component Palette

The Component Palette is very simple to use, but there are a few things you might not know. There are four simple ways to place a component on a form:

- After selecting a control in the palette, click within the form to set the position for the control, and press-and-drag the mouse to size it.
- After selecting any component, simply click within the form to place it with the default height and width.
- Double-click the icon in the palette to add a component of that type in the center of the form.
- Shift-click the component icon to place several components of the same kind in the form. To stop this operation, simply click the standard selector (the arrow icon) on the left side of the Component Palette.

You can select the Properties command on the shortcut menu of the palette to completely rearrange the components in the various pages, possibly adding new elements or just moving them from page to page. In the resulting Properties page, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page.

### TIP

When you have too many pages in the Component Palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: Rename the pages with shorter names, so that all the pages will fit on the screen. Obvious—once you've thought about it.

The real undocumented feature of the Component Palette is the “hot-track” activation. By setting special keys of the Registry, you can simply select a page of the palette by moving over the tab, without any mouse click. The same feature can be applied to the component scrollers on both sides of the palette, which show up when a page has too many components. To activate this hidden feature, you must add an `Extras` key under `HKEY_CURRENT_USER\Software\Borland\Delphi\6.0`. Under this key enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string '1'.

## Defining Event Handlers

There are several techniques you can use to define a handler for an event of a component:

- Select the component, move to the Events page, and either double-click in the white area on the right side of the event or type a name in that area and press the Enter key.
- For many controls, you can double-click them to perform the default action, which is to add a handler for the `OnClick`, `OnChange`, or `OnCreate` events.

When you want to remove an event handler you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event handler that is still empty, consider adding a comment to it (even just the `//` characters), so that it will not be removed.

## Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate the component in the form. During this operation, Delphi duplicates all the properties, keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

It is also possible to copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source-code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'My Button'
  TabOrder = 0
  Font.Name = 'Arial'
end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

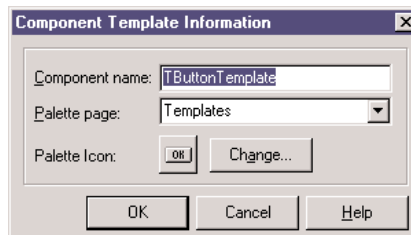
To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something completely wrong and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

## From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form, by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component ➤ Create Component Template menu command. This opens the Component Template Information dialog box, where you enter the name of the template, the page of the Component Palette where it should appear, and an icon.



By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

All the information about component templates is stored in a single file, `DELPHI32.DCT`, but there is apparently no way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

**TIP**

A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry `CCLibDir` under the key `\Software\Borland\Delphi\6.0\Component Templates`.

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the *frames* technology in Delphi does.

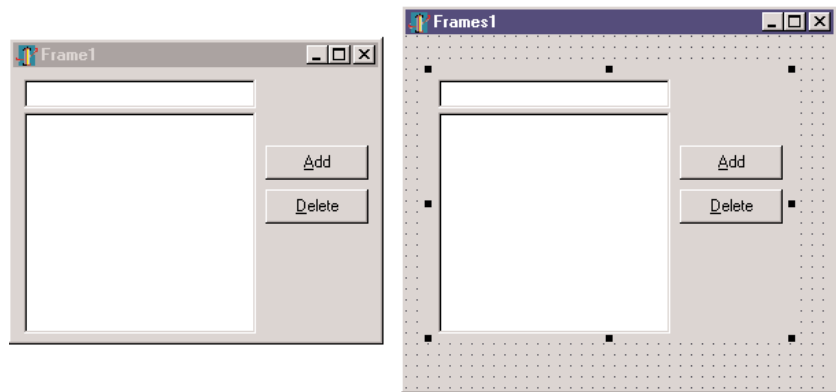
A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready, you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each of the instances of the frame.

You can see a simple example, called `Frames1`, in Figure 1.8 (its code is available on the companion CD). A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames. Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than component templates. Chapter 4 provides an in-depth look at VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful new technique.



**FIGURE 1.8:**

The Frames1 example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synchrony.



## Managing Projects

Delphi's multitarget Project Manager (View > Project Manager) works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files.

**TIP**

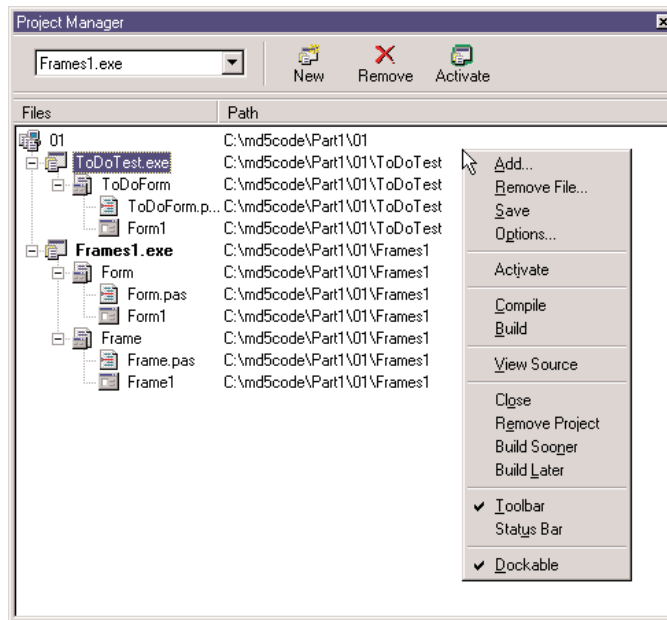
In Delphi 6, all open packages will show up as projects in the Project Manager view, even if they haven't been added to the project group.

In Figure 1.9, you can see the Project Manager with the project group for the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all of the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on it. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, or to open a unit.

Of all the projects in the group, only one is active, and this is the project you operate upon when you select a command such as Project > Compile. The Project pull-down of the main menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.) When you have multiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.

**FIGURE 1.9:**

Delphi's multitarget Project Manager



Among its advanced features, you can drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project.

The Project Manager automatically selects as the current project the one you are working with—for example, opening a file. You can easily see which project is selected and change it by using the combo box on the top of the form.

### TIP

Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu, and choose Resource File (\*.rc) as the file type. This resource file will be automatically bound to the project, even without a corresponding \$R directive.

Delphi saves the project groups with the new .BPG extension, which stands for Borland Project Group. This feature comes from C++Builder and from past Borland C++ compilers, a history that is clearly visible as you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment. Here is a simple example:

```
#-----
VERSION = BWS.01
#-----
!ifndef ROOT
ROOT = $(MAKEDIR)\..
```

```
!endif
#-----
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)\bin\dcc32.exe $**
BRCC = $(ROOT)\bin\brcc32.exe $**
#-----
PROJECTS = Project1.exe
#-----
default: $(PROJECTS)
#-----
Project1.exe: Project1.dpr
$(DCC)
```

## Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. What you can do instead is invoke the Project Options dialog from the Project Manager for each project. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{X+}` or `{X-}` commands, where you'd replace *X* with the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All project options are saved automatically with the project, but in a separate file with a .DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a CFG file, for command-line compilation. The two files have similar content but a different format: The *dcc* command-line compiler, in fact, cannot use .DOF files, but needs the .CFG format.

Another alternative for saving compiler options is to press Ctrl+O+O (press the O key twice while keeping Ctrl pressed). This inserts, at the top of the current unit, compiler directives that correspond to the current project options, as in the following listing:

```
{ $A+, B-, C+, D+, E-, F-, G+, H+, I+, J+, K-, L+, M-, N+, O+, P+, Q-, R-, S-, T-, U-, V+,
W-, X+, Y+, Z1 }

{ $MINSTACKSIZE $00004000 }
```

```
{$MAXSTACKSIZE $00100000}
```

```
{$IMAGEBASE $00400000}
```

```
{$APPTYPE GUI}
```

## Compiling and Building Projects

There are several ways to compile a project. If you run it (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed.

If you select **Compile > Build All** instead, every file is compiled, even if it has not changed. You should only need this second command infrequently, since Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options, in which case you have to use the **Build All** command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi compiled unit (DCU). (This step is performed only if the DCU file is not already up-to-date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the **Show Compiler Progress** option (in the **Preferences** page of the **Environment Options** dialog box).

### WARNING

Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for the cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows some strange error message, the first thing you should try is the **Build All** command to resynchronize the make feature with the current files on disk.

The **Compile** command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the **Open Project** toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't run unless the proper dynamic link libraries (such as `vc150.bp1`, which is quite large) are available on the computer where you want to run the program.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course, if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in Chapter 12.

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semicomplied" tools.

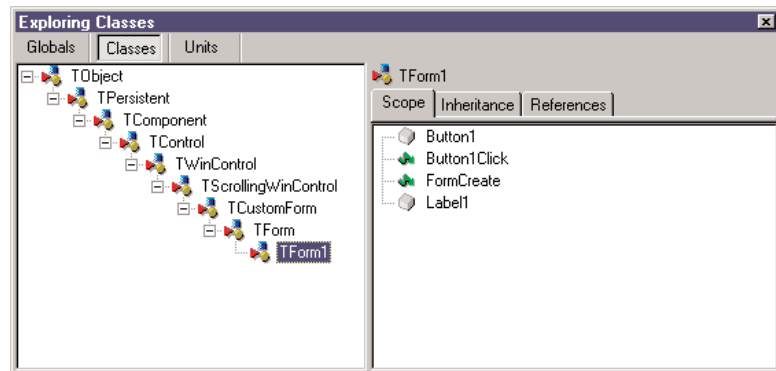
## Exploring a Project

Past versions of Delphi included an Object Browser, which you could use when a project was compiled to see a hierarchical structure of its classes and to look for its symbols and the source-code lines where they are referenced. Delphi now includes a similar but enhanced tool, called Project Explorer. Like the Code Explorer, it is updated automatically as you type, without recompiling the project.

The Project Explorer allows you to list Classes, Units, and Globals, and lets you choose whether to look only for symbols defined within your project or for those from both your project and VCL. You can see an example with only project symbols in Figure 1.10.

**FIGURE 1.10:**

The Project Explorer



You can change the settings of this Explorer and those of the Code Explorer in the Explorer page of the Environment Options or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the Explorer categories you see in this window are specific to the Project Explorer; others relate to both tools.

## Additional and External Delphi Tools

Besides the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (`PCE.exe`), and the Image Editor (`ImageEdit.exe`), are available from Tools menu of the IDE. In addition, the Enterprise edition has a link to the SQL Monitor (`SqlMon.exe`).

Other tools that are not directly accessible from the IDE include many command-line tools you can find in the `bin` directory of Delphi. For example, there is a command-line Delphi compiler (`DCC.exe`), a Borland resource compiler (`BRC32.exe` and `BRCC32.exe`), and an executable viewer (`TDump.exe`).

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools, mostly available in the `\Delphi6\bin` folder and in the Tools menu:

**Web App Debugger** (`WebAppDbg.exe`) is the debugging Web server introduced in Delphi 6. It is used to keep track of the requests sent to your applications and debug them. I'll discuss this tool in Chapter 21.

**XML Mapper** (`Xm1Mapper.exe`), again new in Delphi 6, is a tool for creating XML transformations to be applied to the format produced by the `ClientDataSet` component. More on this topic in Chapter 22.

**External Translation Manager** (`etm60.exe`) is the stand-alone version of the Integrated Translation Manager. This external tool can be given to external translators and is available for the first time in Delphi 6.

**Borland Registry Cleanup Utility** (`D6RegClean.exe`) helps you remove all of the Registry entries added by Delphi 6 to a computer.

**TeamSource** is an advanced version-control system provided with Delphi, starting with version 5. The tool is very similar to its past incarnation and is installed separately from Delphi.

**WinSight** (`Ws.exe`) is a Windows “message spy” program available in the `bin` directory.

**Database Explorer** can be activated from the Delphi IDE or as a stand-alone tool, using the `DBExp1or.exe` program of the `bin` directory.

**OpenHelp** (`oh.exe`) is the tool you can use to manage the structure of Delphi’s own Help files, integrating third-party files into the help system.

**Convert** (`Convert.exe`) is a command-line tool you can use to convert DFM files into the equivalent textual description and vice versa.

**Turbo Grep** (`Grep.exe`) is a command-line search utility, much faster than the embedded Find In Files mechanism but not so easy to use.

**Turbo Register Server** (`TRegSvr.exe`) is a tool you can use to register ActiveX libraries and COM servers. The source code of this tool is available under `\Demos\ActiveX\TRegSvr`.

**Resource Explorer** is a powerful resource viewer (but not a full-blown resource editor) you can find under `\Demos\ResXplor`.

**Resource Workshop** The Delphi 5 CD also includes a separate installation for Resource Workshop. This is an old 16-bit resource editor that can also manage Win32 resource files. It was formerly included in Borland C++ and Pascal compilers for Windows and was much better than the standard Microsoft resource editors then available. Although its user interface hasn’t been updated and it doesn’t handle long filenames, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files.

## The Files Produced by the System

Delphi produces various files for each project, and you should know what they are and how they are named. Basically, two elements have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you’ll find in the directory where a Delphi project resides. The table also shows when or under what circumstances these files are created and their importance for future compilations.

**TABLE 1.1:** Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to end-users.
.CAB	The Microsoft Cabinet compressed-file format used for Web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.
.DCP	Delphi Component Package: a file with symbol information for the code that was compiled into the package. It doesn't include compiled code, which is stored in DCU files.	Compilation	Required when you use packages. You'll distribute it only to other developers along with DPL files.
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DDP	The new Delphi Diagram Portfolio, used by the Diagram view of the editor (was .DTI in Delphi 5)	Development	No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.



**TABLE 1.1 continued:** Delphi Project File Extensions

<b>Extension</b>	<b>File Type and Description</b>	<b>Creation Time</b>	<b>Required to Compile?</b>
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
~DF	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN	Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).	Development (ITE)	Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.
.DLL	Dynamic Link Library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK	Delphi Package: the project source code file of a package.	Development	Yes.
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
~DP	Backup of the Delphi Project file (.DPR).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.

**TABLE 1.1 continued:** Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.DSM	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.EXE	Executable file: the Windows application you've produced.	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or .HTML, for Hypertext Markup Language: the file format used for Internet Web pages.	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
OCX	OLE Control Extension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
.PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.
~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.

**TABLE 1.1 continued:** Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.RPS	Translation Repository (part of the Integrated Translation Environment).	Development (ITE)	No. Required to manage the translations.
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
TODO	To-do list file, holding the items related to the entire project.	Development	No. This file hosts notes for the programmers.
.UDL	Microsoft Data Link.	Development	Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).

Besides the files generated during the development of a project in Delphi, there are many others generated and used by the IDE itself. In Table 1.2, I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

**TABLE 1.2:** Selected Delphi IDE Customization File Extensions

Extension	File Type
.DCI	Delphi code templates
.DRO	Delphi's Object Repository (The repository should be modified with the Tools ➤ Repository command.)
.DMT	Delphi menu templates
.DBI	Database Explorer information
.DEM	Delphi edit mask (files with country-specific formats for edit masks)
.DCT	Delphi component templates
.DST	Desktop settings file (one for each desktop setting you've defined)

## Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time covering their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depends on syntax highlighting, but it isn't saved with the file. It is worth noting that there is one single file for the whole code of the form, not just small code fragments.

---

**TIP**

In the listings in this book, I've matched the bold syntax highlighting of the editor for keywords and the italic for strings and comments.

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a .DFM extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is a binary and, in Delphi, can be saved either as a plain-text file or in the traditional Windows Resource format. You can set the default format you want to use for new projects in the **Designer** page of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command of a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can actually edit the textual description of a form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, compilation will stop with an error message and you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have good knowledge of Delphi programming.

---

**TIP**

In the book, I often show you excerpts of DFM files. In most of these excerpts, I only show the most relevant components or properties; generally, I have removed the positional properties, the binary values, and other lines providing little useful information.

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the **Project -> View Source** menu command.

Some of the other, less relevant files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF):

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

**NOTE**

A lot of information related to the status of the Delphi environment is saved in the Windows Registry, as well as in DSK and other files. I've already indicated a few special undocumented entries of the Registry you can use to activate specific features. You should explore the `HKEY_CURRENT_USER\Software\Borland\Delphi\6.0` section of the Registry to examine all the settings of the Delphi IDE (including all those you can modify with the Project Options and the Environment Options dialog boxes, as well as many others).

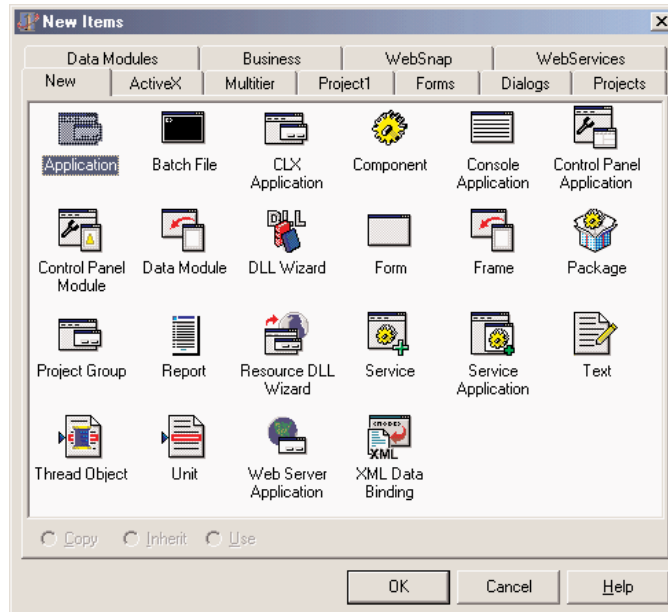
## The Object Repository

Delphi has menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File > New menu and in other pull-down menus. What happens if you simply select File > New > Other? Delphi opens the Object Repository, which is used to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New dialog box (shown in Figure 1.11) has several pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.

**FIGURE 1.11:**

The first page of the New dialog box, generally known as the “Object Repository”



### TIP

The Object Repository has a shortcut menu that allows you to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool, information that is particularly important when looking at wizards, projects, or forms that you've added to the Repository.

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and “current project” pages). Adding a new template to Delphi’s Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, ready to use later on. Simply use the Project ➤ Add To Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form that you want to add and select the Add To Repository command of its shortcut menu. Then indicate the title, description, author, page, and icon in its dialog box.

You might want to keep in mind that as you copy a project or form template to the repository and then copy it back to another directory, you are simply doing a copy and paste operation. This isn’t much different than copying the files manually.

## The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or Wizards, this is not what you want, however. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple:

1. Create a new project as usual.
2. Remove its only form from the project.
3. Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template’s files will be copied. There is also a disadvantage—you have to remember to use the File ➤ Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools ➤ Repository command. This opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing ones. You can even add new pages, rename or

delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File > New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File > New Application) when no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File > New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the OBJREPOS subdirectory of the Delphi main directory. At the same time, if you use a form or any other object directly without copying it, then you end up having some files of your project in this directory. It is important to realize how the Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files, without copying data back and forth to the Repository.

## Installing New DLL Wizards

Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they would be installed the same way you install a component or a package. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key `\Software\Borland\Delphi\6.0\Experts`. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.



## What's Next?

This chapter has presented an overview of the new and more advanced features of the Delphi 6 programming environment, including tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to start *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

Now we are ready to spend the next two chapters looking into the Object Pascal language and then proceed by studying the RTL and the class library included in Delphi 6.

# The Object Pascal Language: Classes and Objects

- The Pascal language
- New conditional compilation and hint directives
- Classes and objects
- The *Self* keyword
- Class methods and overloading
- Encapsulation: *private* and *public*
- Using properties
- Constructors
- Objects and memory

**M**ost modern programming languages support *object-oriented programming* (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding).

You can write Delphi applications even without knowing the details of Object Pascal. As you create a new form, add new components, and handle events, Delphi prepares most of the related code for you automatically. But knowing the details of the language and its implementation will help you to understand precisely what Delphi is doing and to master the language completely.

A single chapter doesn't allow space for a full introduction to the principles of object-oriented programming and the Object Pascal language. Instead, I will outline the key OOP features of the language and show how they relate to everyday Delphi programming. Even if you don't have a precise knowledge of OOP, the chapter will introduce each of the key concepts so that you won't need to refer to other sources.

## The Pascal Language

The Object Pascal language used by Delphi is an OOP extension of the classic Pascal language, which Borland pushed forward for many years with its Turbo Pascal compilers. The syntax of the Pascal language is known to be quite verbose and more readable than, for example, the C language. Its OOP extension follows the same approach, delivering the same power of the recent breed of OOP languages, from Java to C#.

In this chapter, I'll discuss only the object-oriented extensions of the Pascal language available in Delphi. However, I'll highlight recent additions Borland has done to the core language. These features have been introduced in Delphi 6 and are, at least partially, related to the Linux version of Delphi.

New Pascal features include the `$IF` and `$ELSEIF` directives for conditional compilation, the `$WARN` and `$MESSAGE` directives, and the `platform`, `library`, and `deprecated` hint directives. These topics are discussed in the following sections. Changes to the assembler (with new directives, support for MMX and Pentium Pro instructions, and many more features) are really beyond the scope of this book.

Other relatively minor changes in the language include a change in the default value for the `$WRITEABLECONST` compiler switch, which is now disabled. This option allows programs to modify the value of typed constants and should generally be left disabled, using variables instead of constants for modifiable values. Another change is the support for the `Int64` data type in variants. Finally, you can assign specific values to the elements of an enumeration (as in the C/C++ language), instead of using the default sequence of values.

## The New **\$IF** Compiler Directive

Delphi has always had a `$IFDEF` directive you could use to test whether a specific symbol was defined. (Delphi also has a `$IFNDEF` directive, with the opposite test.) This is used to obtain conditional compilation, as in

```
{IFDEF DEBUG}
  // executes only if the DEBUG directive is set
  ShowMessage ('Executing critical code');
{ENDIF}
```

By setting or not setting the `DEBUG` directive and recompiling, the extra line of code will be included or skipped by the compiler.

This code directive is powerful, but checking for multiple versions of Delphi and operating systems can force you to use multiple-nested `$IFDEF` directives, making the code totally unreadable. For this reason, Borland has introduced a new and more powerful directive for conditional compilation, `$IF`. Inside the directive you can use the `Defined` function to check whether a conditional symbol is defined, or use the `Declared` function to see whether a language constant is defined and use these constants within a constant Boolean expression. Here is some code that shows how to use a constant within the `$IF` directive (you can find this and other code excerpts of this and the next section in the `IfDirective` example on the companion CD):

```
const
  DebugControl = 2;

{IF Defined(DEBUG) and (DebugControl > 3)}
  ShowMessage ('Executing critical code');
{IFEND}
```

Notice that the statement is closed by a `$IFEND` and that you can also have an optional `$ELSE` branch. You can also concatenate conditions with the `$ELSEIF` directive, followed by another condition and evaluated only as an alternative to the `$IF` directive it refers to:

```
{IF one}
  ...
{$ELSEIF two}
  ...
{$ELSE}
  ...
{$IFEND}
```

Within the expressions of the `$IF` directive, you can use only untyped constants, which are really and invariably treated as constants by the compiler. You can follow the general rules of Pascal constant expressions. You can use all the language operators, the `and`, `or`, `xor`, and not Boolean operators, and mathematical ones including `div`, `mod`, `+`, `-`, `*`, `/`, `>` and `<`, to mention just a few common ones. You can also use predefined functions such as `SizeOf`, `High`, `Low`,

Prev, Succ, and others listed in the Delphi Help page “Constant expressions.” The expression can use constant symbols of any type, including floats and strings, so long as the expression itself ultimately evaluates to a True or False value.

**WARNING**

In these constant expressions, it is not possible to use type constants, which can be optionally modified in the code depending on the status of the writeable-typed constants directive (`$J` or `$WRITEABLECONST`). In any case, using constants you can modify is quite a bad idea in the first place.

Delphi provides a few predefined conditional symbols, including compiler version, the operating system, the GUI environment, and so on. I’ve listed the most important ones in Table 2.1. You can also use the `RTLVersion` constant defined in the System unit to test which version of Delphi (and its run-time library) you are compiling on. The predefined symbol `ConditionalExpressions` can be used to shield the new directives from older versions of Delphi:

```
{$IFDEF ConditionalExpressions}
  {$IF System.RTLVersion > 14.0}
    // do something
  {$IFEND}
{$ENDIF}
```

**TABLE 2.1:** Commonly Used Predefined Conditional Symbols

Symbol	Description
VER140	Compiling with Delphi 6, which is the 14.0 version of the Borland Pascal compiler; Delphi 5 used <code>VER130</code> , with lower numbers for past versions.
MSWINDOWS	Compiling on the Windows platform (new in Delphi 6).
LINUX	Compiling on the Linux platform. On Kylix, there are also the <code>LINUX32</code> , <code>POSIX</code> , and <code>ELF</code> predefined symbols.
WIN32	Compiling only on the 32-bit Windows platform. This symbol was introduced in Delphi 2 to distinguish from 16-bit Windows compilations (Delphi 1 defined the <code>WINDOWS</code> symbol). You should use <code>WIN32</code> only to mark code specifically for Win32, not Win16 or future Win64 platforms (for which the <code>WIN64</code> symbol has been reserved). Use <code>MSWINDOWS</code> , instead, to distinguish between Windows and other operating systems.
CONSOLE	Compiling a console application, and not a GUI one. This symbol is meaningful only under Windows, as all Linux applications are console applications.
BCB	Defined when the C++Builder IDE invokes the Pascal compiler.
<code>ConditionalExpressions</code>	Indicates that the <code>\$IF</code> directive is available. It is defined in Kylix and Delphi 6, but not in earlier versions.

**TIP**

I recommend using conditional compilation sparingly and only when it is really required. It is generally better, whenever possible, to write code that can adapt to different situations—for example, adding different versions of the same class (or different inherited classes) to the same program. Excessive use of conditional compilation makes a program hard to read and to debug.

**WARNING**

Remember to issue a Build All command when you change a conditional symbol or a constant, which can affect a conditional compilation; otherwise the affected units won't be recompiled unless their source code changes.

## New Hint Directives

Supporting multiple operating systems within the same source code base implies a number of compatibility issues. Besides a modified run-time library and a wholly new component library (discussed in Chapter 4, “The Run-Time Library,” and Chapter 5, “Core Library Classes”), Delphi 6 includes special directives Borland uses to mark special portions of the code. As they introduced the idea of custom warnings and messages (described in the previous section), they've added a few special predefined ones.

### The *platform* Directive

The first directive of this group is the `platform` directive, used to mark nonportable code. This directive can be used to mark procedures, variables, types, and almost any defined symbol. Borland uses `platform` in its libraries, so that when you use a platform-specific capability (for example, calling the `IncludeTrailingBackslash` function of the `SysUtils` unit), you'll receive a warning message, such as:

```
Symbol 'IncludeTrailingBackslash' is specific to a platform.
```

This warning is a hint for developers who plan to port their code between the Linux and Windows platforms, even in the future. In many cases, you'll be able to find an alternative approach that is fully platform independent. Check the help file (or eventually the library source code) for hints in this direction. In the case of the `IncludeTrailingBackslash` function, there is now a new version, called `IncludeTrailingDelimiter`, that is also portable to a Unix-based file system.

Of course you can use the `platform` directive to mark your code, for example, if you write a component or library that has platform-specific features. Here are a few examples:

```
var
  windowsversion: Integer = 2000 platform;
```

```
procedure Test; platform;
begin
  Beep;
end;

type
  TWinClass = class
    x: Integer;
  end platform;
```

The code fragments of this section are available, for your experiments, in the `IfDirective` example on the companion CD.

**NOTE**

The position of semicolons for hint directives can be quite confusing at first. The rule is that a hint directive must appear before the semicolon following the symbol it modifies. But a procedure, function, or unit header declaration can be followed only by reserved words, so its hint directive can appear following the semicolon. A type, variable, or constant declaration can be followed by another identifier, so the hint directive must come before the semicolon closing its declaration. Part of the rationale behind this is that the hint directives are not reserved words, so they can be used as the name of an identifier.

## The *deprecated* Directive

The `deprecated` directive works in a similar way to the `platform` directive; the only real differences are that it is used in a different context and produces a different compiler warning. The role of `deprecated` is to mark identifiers that are still part of the system for compatibility reasons, but either are going to be removed in the future or expose you to risks of incompatibility. This symbol is used sparingly in the Delphi library.

## The *library* Directive

The `library` directive works in a similar way to `deprecated` and `platform`; its role is to mark out code or components that are specific to a library (either VCL or CLX) and are not portable among them. However, apparently this symbol is never used within the Delphi library.

## The *\$WARN* Directive

The `$WARNINGS` directive (and the corresponding compiler option) allows you to turn off all the warning messages. Most programmers like to keep the messages on and tend to work with programs that compile with no hints and warnings. With the advent of the three hint directives discussed in the last section, however, there are programs specifically aimed for a platform, which cannot compile without compatibility warnings.

To overcome this situation, Delphi 6 introduces the `$WARN` directive, specifically aimed at disabling hint directives. As an example, you'll disable platform hints by writing this code:

```
{ $WARN SYMBOL_PLATFORM OFF }
```

The `$WARN` directive has five different parameters, related to the three hint directives, and can use the `ON` and `OFF` values for each:

- `SYMBOL_PLATFORM` and `UNIT_PLATFORM` can be used to disable the `platform` directive in the current unit or in the unit where the directive is specified. The warning, in fact, is issued while compiling the code that uses the symbol, not while compiling the code with the definition.
- `SYMBOL_LIBRARY` and `UNIT_LIBRARY` work on the `library` directive in the same manner as the `platform`-related parameters above.
- `SYMBOL_DEPRECATED` can be used to disable the `deprecated` directive.

## The `$MESSAGE` Directive

The compiler has now the ability to generate warnings in many different situations, so that the developer of a library or a portion of a program can let other programmers know of a given problem or risk in using a given feature, when the program can still legally compile. An extension to this idea is to let programmers insert custom warning messages in the code, with this syntax:

```
{ $MESSAGE 'Old version of the unit: consider using the updated version' }
```

Compiling this code will issue a hint message with the text provided. This feature can be used to indicate possible problems, suggest alternative approaches, mark unfinished code, and more. This is probably more reliable than using a `TODO` item (discussed in the preceding chapter), because a programmer might not open the To-Do List window but the compiler will remind him of the pending problem. However, it is the compiler that issues the message, so you'll see it even if the given portion of the code is not really used by the program because the linker will remove it from the executable file.

These type of free messages, like the hint directives, become very useful to let the developer of a component communicate with the programmers using it, warning of potential pitfalls.

# Introducing Classes and Objects

The cornerstone of the OOP extensions available in Object Pascal is represented by the `class` keyword, which is used inside type declarations. Classes define the blueprint of the



objects you create in Delphi. As the terms *class* and *object* are commonly used and often misused, let's be sure we agree on their definitions.

A *class* is a user-defined data type, which has a state (its representation) and some operations (its behavior). A class has some internal data and some methods, in the form of procedures or functions, and usually describes the generic characteristics and behavior of some similar objects.

An *object* is an instance of a class, or a variable of the data type defined by the class. Objects are *actual* entities. When the program runs, objects take up some memory for their internal representation. The relationship between object and class is the same as the one between variable and type.

To declare a new class data type in Object Pascal, with some local data fields and some methods, use the following syntax:

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

## NOTE

The convention in Delphi is to use the letter *T* as a prefix for the name of every class you write and every other type (*T* stands for *Type*). This is just a convention—to the compiler, *T* is just a letter like any other—but it is so common that following it will make your code easier to understand.

The following is a complete class definition, with two methods declared and not yet fully defined. The definition of these two methods (the `LeapYear` function and the `SetValue` procedure) must be present in the same unit of the class declaration and are written with this syntax:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils.pas
  Result := IsLeapYear (Year);
end;
```

The method names are prefixed with the class name (using the dot-notation), because a unit can hold multiple classes, possibly with methods having the same names. You can actually avoid retyping the method names and parameter list by using the class completion feature of the editor. Simply type or modify the class definition and press Ctrl+Shift+C while the cursor is within the class definition itself; this will allow Delphi to generate a skeleton of the definition of the methods, including the begin and end statements.

Once the class has been defined, we can create an object and use it as follows:

```
var
    ADay: TDate;
begin
    // create an object
    ADay := TDate.Create;
    // use the object
    ADay.SetValue (1, 1, 2000);
    if ADay.LeapYear then
        ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
    // destroy the object
    ADay.Free;
end;
```

Notice that `ADay.LeapYear` is an expression similar to `ADay.Year`, although the first is a function call and the second a direct data access. You can optionally add parentheses after the call of a function with no parameters. You can find the code snippets above in the source code of the `Date1` example; the only difference is that the program creates a date based on the year provided in an edit box.

## Classes, Objects, and Visual Programming

When I teach classes about OOP in Delphi, I always tell my students that regardless of how much OOP you know and how much you use it, Delphi forces you in the OOP direction. Even if you simply create a new application with a form and place a button over it to execute some code when the button is pressed, you are building an object-oriented application. In fact, the form is an object of a new class (by default `TForm1`, which inherits from the base `TForm` class provided by Borland), and the button is an instance of the `TButton` class, provided by Borland, as you can see in the following code snippet:

```
type
    TForm1 = class(TForm)
        Button1: TButton;
end;
```

Given these premises, it would be very hard to build a Delphi application without using classes and objects. Yes, I know it is technically possible, but I doubt it would make a lot of

sense. Not using objects and classes with Delphi would probably be more difficult than using them, as you have to give up all of the design-time tools for visual programming.

In any case, the real challenge is using OOP properly, something I'll try to teach you in this chapter (and in the rest of the book), along with an introduction to the key elements of the Object Pascal language.

## The *Self* Keyword

Methods are very similar to procedures and functions. The real difference is that methods have an implicit parameter, which is a reference to the current object. Within a method you can refer to this parameter—the current object—using the `Self` keyword. This extra hidden parameter is needed when you create several objects of the same class, so that each time you apply a method to one of the objects, the method will operate only on its own data and not affect sibling objects.

For example, in the `SetValue` method of the `TDate` class, listed earlier, we simply use `Month`, `Year`, and `Day` to refer to the fields of the current object, something you might express as

```
Self.Month := m;  
Self.Day := d;
```

This is actually how the Delphi compiler translates the code, *not* how you are supposed to write it. The `Self` keyword is a fundamental language construct used by the compiler, but at times it is used by programmers to resolve name conflicts and to make tricky code more readable.

### NOTE

The C++ and Java languages have a similar feature based on the keyword `this`.

All you really need to know about `Self` is that the technical implementation of a call to a method differs from that of a call to a generic subroutine. Methods have an extra hidden parameter, `Self`. Because all this happens behind the scenes, you do not need to know how `Self` works at this time.

If you look at the definition of the `TMethod` data type in the `System` unit, you'll see that it is a record with a `Code` field and a `Data` field. The first is a pointer to the function's address in memory; the second the value of the `Self` parameter to use when calling that function address. We'll discuss method pointers in Chapter 5.

## Overloaded Methods

Object Pascal supports overloaded functions and methods: you can have multiple methods with the same name, provided that the parameters are different. By checking the parameters, the compiler can determine which of the versions of the routine you want to call.

There are two basic rules:

- Each version of the method must be followed by the `overload` keyword.
- The differences must be in the number or type of the parameters or both. The return type cannot be used to distinguish between two methods.

Overloading can be applied to global functions and procedures and to methods of a class. As an example of overloading, I've added to the `TDate` class two different versions of the `SetValue` method:

```
type
  TDate = class
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    ...//the rest of the class declaration
  procedure TDate.SetValue (y, m, d: Integer);
  begin
    fDate := EncodeDate (y, m, d);
  end;

  procedure TDate.SetValue(NewDate: TDateTime);
  begin
    fDate := NewDate;
  end;
```

#### NOTE

In Delphi 6, the compiler has been enhanced to improve the resolution of overloaded methods, allowing the compilation of calls that were considered ambiguous. In particular, the compiler handles the difference between `AnsiString` and `WideString` types. The overload resolution also has better support for variant-type parameters (which will provide matches in case there is no exact match for another overloaded version) and interfaces (which are given precedence to object types). Finally, the compiler allows the `nil` value to match an interface-type parameter. Some of these improvements were already introduced in the Kylix compiler.

## Creating Components Dynamically

In Delphi, the `Self` keyword is often used when you need to refer to the current form explicitly in one of its methods. The typical example is the creation of a component at run time, where you must pass the owner of the component to its `Create` constructor and assign the same value to its `Parent` property. (The difference between `Owner` and `Parent` properties is discussed in the next chapter.) In both cases, you have to supply the current form as parameter or value, and the best way to do this is to use the `Self` keyword.

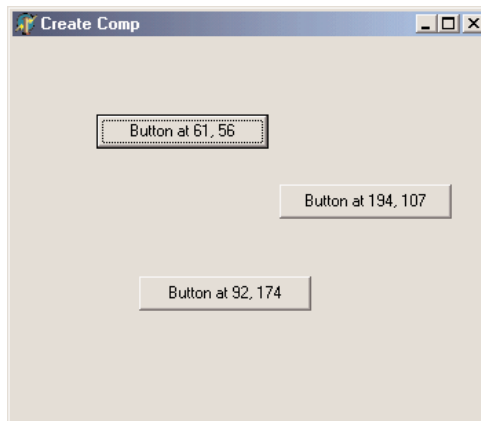
To demonstrate this kind of code, I've written the CreateC example (the name stands for *Create Component*) included on the companion CD. This program has a simple form with no components and a handler for its OnMouseDown event. I've used OnMouseDown because it receives as its parameter the position of the mouse click (unlike the OnClick event). I need this information to create a button component in that position. Here is the code of the method:

```
procedure TForm1.FormMouseDown (Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
var  
    Btn: TButton;  
begin  
    Btn := TButton.Create (Self);  
    Btn.Parent := Self;  
    Btn.Left := X;  
    Btn.Top := Y;  
    Btn.Width := Btn.Width + 50;  
    Btn.Caption := Format ('Button at %d, %d', [X, Y]);  
end;
```

The effect of this code is to create buttons at mouse-click positions, with a caption indicating the exact location, as you can see in Figure 2.1. In the code above, notice in particular the use of the Self keyword, as the parameter of the Create method and as the value of the Parent property. I'll discuss these two elements (ownership and the Parent property) in Chapter 5.

**FIGURE 2.1:**

The output of the CreateC example, which creates Button components at run time



It is very common to write code like the above method using a with statement, as in the following listing:

```
procedure TForm1.FormMouseDown (Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

```
begin
  with TButton.Create (Self) do
    begin
      Parent := Self;
      Left := X;
      Top := Y;
      Width := Width + 50;
      Caption := Format ('Button in %d, %d', [X, Y]);
    end;
  end;
```

**TIP**

When writing a procedure like the code you've just seen, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't make any practical difference, but if there are multiple instances of a form, using `Form1` would be an error. In fact, if the `Form1` variable refers to the first form of that type being created, by clicking in another form of the same type, the new button will always be displayed in the first form. Its `Owner` and `Parent` will be `Form1` and not the form the user has clicked. In general, referring to a particular instance of a class when the current object is required is bad OOP practice.

## Class Methods and Class Data

When you define a field in a class, you actually specify that the field should be added to each object of that class. Each instance has its own independent representation (referred to by the `Self` pointer). In some cases, however, it might be useful to have a field that is shared by all the objects of a class.

Other object-oriented programming languages have formal constructs to express this, while in Object Pascal we can simulate this feature using the encapsulation provided at the unit level. You can simply add a variable in the implementation portion of a unit, to obtain a class variable—a single memory location shared by all of the objects of a class.

If you need to access this value from outside the unit, you might use a method of the class. However, this forces you to apply this method to one of the instances of the class. An alternative solution is to declare a *class method*. A class method cannot access the data of any single object but can be applied to a class as a whole rather than to a particular instance.

To declare a class method in Object Pascal, you simply add the `class` keyword in front of it:

```
type
  MyClass = class
    class function ClassMeanValue: Integer;
```

The use of class methods is not very common in Object Pascal, because you can obtain the same effect by adding a procedure or function to a unit declaring a class. Object-oriented purists, however, will definitely prefer the use of a class method over a routine unrelated to a

class. For example, an OOP purist would add a class method for getting the current date to a `TDate` class instead of using a global function (also because some OOP languages, including Java, don't have the notion of global functions).

We'll see several class methods in the next chapter, when we'll examine the structure of the `TObject` class.

**TIP**

Contrary to other OOP languages, Delphi class methods can also be virtual, so they can be overridden and used to obtain polymorphism (a technique discussed later in this chapter).

## Encapsulation

A class can have any amount of data and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated*, inside the class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, as the methods can check whether the date is valid and refuse to modify the new value if it is not. Encapsulation is important because it allows the class writer to modify the internal representation in a future version.

The concept of encapsulation is often indicated by the idea of a “black box,” where you don't know about the internals: You only know how to interface with it or how to use it regardless of its internal structure. The “how to use” portion, called the *class interface*, allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Of course, you are supposed to use methods to access the data, which is shielded from unauthorized access. This is the object-oriented approach to a classical programming concept known as *information hiding*.

Delphi implements this class-based encapsulation but still supports the classic module-based encapsulation using the structure of units. Because the two are strictly related, let me recap the traditional approach first.

## Encapsulation and Units

A unit in Object Pascal is a secondary source-code file, with the main source-code file being represented by the project source code. Every unit has two main sections, called `interface` and `implementation`, as well as two optional ones for `initialization` and `finalization` code. I want to focus here on the information hiding implemented by units.

In short, every identifier (type, routine, variable, and so on) that you declare in the interface portion of a unit becomes visible to any other unit of the program, provided there is a `uses` statement referring back to the unit that defines the identifier. All the routines and methods you declare in the interface portion of the unit must later be fully defined in the implemented portion of the same unit. In the interface section of a unit, however, you cannot write any actual statements to execute.

On the other hand, any identifier you declare in the implementation portion of the unit is local to the unit and is not visible outside it. A unit can have local data, local support functions, and even local types that the rest of the program is not allowed to access. This provides a direct way to hide the implementation details of an abstraction from its users, so you can later change your code without affecting other units of the program (and without even having to notify the changes to other programmers writing those units).

When you write classes in a unit, you'll generally define them in the interface portion of a unit, but some special keywords allow you to hide portions of this class interface.

## Private, Protected, and Public

For class-based encapsulation, the Object Pascal language has three access specifiers: `private`, `protected`, and `public`. A fourth, `published`, controls RTTI and design time information and will be discussed in more detail in Chapter 5. Here are the three *classic* access specifiers:

- The `private` directive denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.
- The `protected` directive is used to indicate methods and fields with limited visibility. Only the current class and its subclasses can access `protected` elements. We'll discuss this keyword again in the “Protected Fields and Encapsulation” section.
- The `public` directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.

Generally, the fields of a class should be `private`; the methods are usually `public`. However, this is not always the case. Methods can be `private` or `protected` if they are needed only internally to perform some partial computation. Fields can be `protected` so that you can manipulate them in subclasses, but only if you are fairly sure that their type definition is not going to change. Access specifiers only restrict code outside your unit from accessing certain members of classes declared in the interface section of your unit. This means that if two classes are in the same unit, there is no protection for their `private` fields. Only by placing a class in the interface portion of a unit will you limit the visibility from classes and functions in other units to the `public` method and fields of the class.



As an example, consider this new version of the TDate class:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

In this version, the fields are now declared to be private, and there are some new methods. The first, `GetText`, is a function that returns a string with the date. You might think of adding other functions, such as `GetDay`, `GetMonth`, and `GetYear`, which simply return the corresponding private data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class you are implementing.

Another new method is the `Increase` procedure, which increases the date by one day. This is far from simple, because you need to consider the different lengths of the various months as well as leap and non-leap years. What I'll do to make it easier to write the code is change the internal implementation of the class to Delphi's `TDateTime` type for the internal implementation. The class definition will change to (the complete code will be in the next example, `DateProp`):

```
type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Notice that because the only change is in the private portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

**NOTE**

The `TDateTime` type is actually a floating-point number. The integral portion of the number indicates the date since 12/30/1899, the same base date used by OLE Automation and Microsoft applications. (Use negative values to express previous years.) The decimal portion indicates the time as a fraction. For example, a value of 3.75 stands for the second of January 1900, at 6:00 A.M. (three-quarters of a day). To add or subtract dates, you can add or subtract the number of days, which is much simpler than adding days with a day/month/year representation.

## Encapsulating with Properties

Properties are a very sound OOP mechanism, or a very well thought out application of the idea of encapsulation. Essentially, you have a name that completely hides its implementation details. This allows you to modify the class extensively without affecting the code using it. A good definition of properties is that of *virtual fields*. From the perspective of the user of the class that defines them, properties look exactly like fields, as you can generally read or write their value. For example, you can read the value of the `Caption` property of a button and assign it to the `Text` property of an edit box with the following code:

```
Edit1.Text := Button1.Caption;
```

This looks like we are reading and writing fields. However, properties can be directly mapped to data, as well as to access methods, for reading and writing the value. When properties are mapped to methods, the data they access can be part of the object or outside of it, and they can produce side effects, such as repainting a control after you change one of its values. Technically, a property is an identifier that is mapped to data or methods using a `read` and a `write` clause. For example, here is the definition of a `Month` property for a date class:

```
property Month: Integer read FMonth write SetMonth;
```

To access the value of the `Month` property, the program reads the value of the private field `FMonth`, while to change the property value it calls the method `SetMonth` (which must be defined inside the class, of course). Different combinations are possible (for example, we could also use a method to read the value or directly change a field in the `write` directive), but the use of a method to change the value of a property is very common. Here are two alternative definitions for the property, mapped to two access methods or mapped directly to data in both directions:

```
property Month: Integer read GetMonth write SetMonth;  
property Month: Integer read FMonth write FMonth;
```

**TIP**

When you write code that accesses a property, it is important to realize that a method might be called. The issue is that some of these methods take some time to execute; they can also produce side effects, often including a (slow) repainting of the component on the screen. Although side effects of properties are seldom documented, you should be aware that they exist, particularly when you are trying to optimize your code.

Often, the actual data and access methods are private (or protected) while the property is public. This means you must use the property to have access to those methods or data, a technique that provides both an extended and a simplified version of encapsulation. It is an *extended* encapsulation because not only can you change the representation of the data and its access functions, but you can also add or remove access functions without changing the calling code at all. A user only needs to recompile the program using the property.

## Class Completion for Properties

Properties provide a *simplified* encapsulation because when extra code is not required, you map the properties directly to fields, without writing tedious and useless access methods. And even when you want to write those methods, the IDE can use class completion (the Ctrl+Shift+C key combination) to generate the skeleton of the access methods of the properties for you. If you simply type in a class (say TMyClass),

```
property X: Integer;
```

and activate class completion, Delphi generates a SetX method for the property and adds the FX field to the class. The resulting code looks like this:

```
type
  TMyClass = class(TForm)
  private
    FX: Integer;
    procedure SetX(const Value: Integer);
  public
    property X: Integer read FX write SetX;
  end;

implementation

procedure TMyClass.SetX(const Value: Integer);
begin
  FX := Value;
end;
```

This really saves a lot of typing. You can even partially control how class completion generates Set and Get methods for the property. In fact, if you first type the property declaration including the read and write directives, as in

```
property X: Integer read GetX write SetX;
```

Class completion will generate the requested methods or add the field definition. If you want both the field and the methods, type in only the property name and its data type (as in the first example above), and let Delphi expand the declaration. At this point, fix the expanded declaration by replacing the FX field with a GetX method in the read portion, and invoke class completion a second time.

## Properties for the *TDate* Class

As an example, I've added properties for accessing the year, the month, and the day to an object of the *TDate* class discussed earlier. These properties are not mapped to specific fields, but they all map to the single *fDate* field storing the entire date information. This is the new definition of the class:

```
type
  TDate = class
  private
    fDate: TDateTime;
    procedure SetDay(const Value: Integer);
    procedure SetMonth(const Value: Integer);
    procedure SetYear(const Value: Integer);
    function GetDay: Integer;
    function GetMonth: Integer;
    function GetYear: Integer;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
  end;
```

Each of the Get and Set methods is easily implemented using functions available in the new *DateUtils* unit (discuss in more detail in Chapter 4). Here is the code for two of them (the others are very similar):

```
function TDate.GetYear: Integer;
begin
  Result := YearOf (fDate);
end;

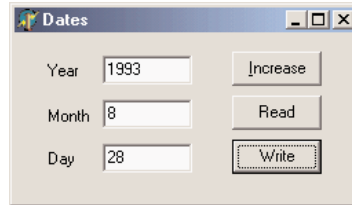
procedure TDate.SetYear(const Value: Integer);
begin
  fDate := RecodeYear (fDate, Value);
end;
```

The code for this class is available in the *DateProp* example. The program uses a secondary unit for the definition of the *TDate* class to enforce encapsulation and creates a single-date object stored in a form variable and kept in memory for the entire execution of the program. Using a standard approach, the object is created in the form *OnCreate* event handler and destroyed in the form *OnDestroy* event handler.

The form of the program (see Figure 2.2) has three edit boxes and buttons to copy the values of these edit boxes to and from the properties of the date object:

**FIGURE 2.2:**

The form of the DateProp example



```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
    EditYear.Text := IntToStr (TheDay.Year);
    EditMonth.Text := IntToStr (TheDay.Month);
    EditDay.Text := IntToStr (TheDay.Day);
end;
```

### WARNING

When writing the values, the program uses the `SetValue` method instead of setting each of the properties. In fact, assigning the month and the day separately can cause you trouble when the month is not valid for the current day. For example, the day is currently January 31, and you want to assign to it February 20. If you assign the month first, this part of the assignment will fail, as February 31 does not exist. If you assign the day first, the problem will arise when doing the reverse assignment. Due to the validity rules for dates, it is better to assign everything at once.

## Advanced Features of Properties

Properties have several advanced features I'll focus on in future chapters, specifically the introduction to the base classes of the library in Chapter 5 and writing custom Delphi components in Chapter 11, "Creating Components." This is a short summary of these more advanced features:

- The `write` directive of a property can be omitted, making it a *read-only* property. The compiler will issue an error if you try to change it. You can also omit the `read` directive and define a *write-only* property, but that doesn't make much sense and is used infrequently.
- The Delphi IDE gives special treatment to *design-time* properties, declared with the `published` access specifier and generally displayed in the Object Inspector for the selected component. More on the `published` keyword and its effect is in Chapter 5.
- The other properties, often called *run-time only* properties, are those declared with the `public` access specifier. These properties can be used in the program code.

- You can define *array-based* properties, which use the typical notation with square brackets to access an element of a list. The *string list-based* properties, such as the Lines of a list box, are a typical example of this group.
- Properties have special directives, including `stored` and `default`, which control the *component streaming system*, introduced in Chapter 5 and detailed in Chapter 11.

**NOTE**

You can usually assign a value to a property or read it, and you can even use properties in expressions, but you cannot always pass a property as a parameter to a procedure or method. This is because a property is not a memory location, so it cannot be used as a `var` parameter; it cannot be passed by reference.

## Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a class's field, you only need to change the code of some methods of that class and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.

Let me clarify this idea with an example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of one of the forms:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

This works but has two problems. First, the data is not connected to a specific instance of the form, but to the entire program. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

The second problem is that if you define the data as a global variable or as a public field of a form, you won't be able to modify its implementation in the future without affecting the

code that uses the data. For example, if you only have to read the current value from other forms, you can declare the data as private and provide a method to read the value:

```
type
  TForm1 = class(TForm)
  public
    function GetClicks: Integer;
  private
    nClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := nClicks;
end;
```

## Adding Properties to Forms

An even better solution is to add a property to the form. Every time you want to make some information of a form available to other forms, you should really use a property, for all the reasons discussed in the previous section. Simply change the field declaration of the form, shown in the preceding listing, by adding the keyword `property` in front of it and then press `Ctrl+Shift+C` to activate code completion. Delphi will automatically generate all of the extra code you need. In the form, you also need to handle the `OnClick` event, increasing the value of the property (and showing it in the form caption):

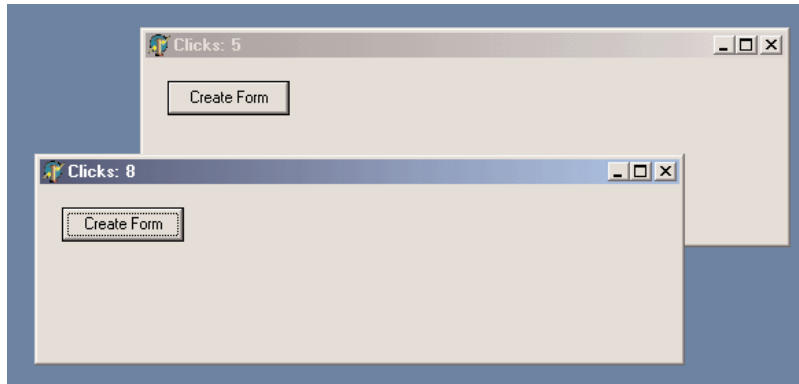
```
procedure TForm1.FormClick(Sender: TObject);
begin
  Inc (FClicks);
  Caption := 'Clicks: ' + IntToStr (FClicks);
end;
```

The complete code for this form class is available in the `FormProp` example and illustrated in Figure 2.3. The program can create multi-instances of the form (that is, multiple objects based on the same form class), each with its own click count. Clicking the `Create Form` button creates the secondary forms, using the following code:

```
procedure TForm1.btnCreateFormClick(Sender: TObject);
begin
  with TForm1.Create (Self) do
    Show;
end;
```

**FIGURE 2.3:**

Two forms of the FormProp example at run time

**NOTE**

Notice that adding a property to a form doesn't add to the list of the form properties in the Object Inspector.

In my opinion, properties should also be used in the form classes to encapsulate the access to the components of a form. For example, if you have a main form with a status bar used to display some information (and with the `SimplePanel` property set to `True`) and you want to modify the text from a secondary form, you might be tempted to write:

```
Form1.StatusBar1.SimpleText := 'new text';
```

This is a standard practice in Delphi, but it's not a good one, because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (replacing `StatusBar` with another control or activating multiple panels), you'll have to fix the code in many places. The alternative is to use a method or, even better, a property to hide the specific control. Simply type

```
property StatusText: string read GetText write SetText;
```

and press the `Ctrl+Shift+C` combination again, to let Delphi add the definition of both methods for reading and writing the property:

```
function TForm1.GetText: string;  
begin  
    Result := StatusBar1.SimpleText;  
end;
```

```
procedure TForm1.SetText(const Value: string);  
begin  
    StatusBar1.SimpleText := Value;  
end;
```



In the other forms of the program, you can simply refer to the `StatusText` property of the form, and if the user interface changes, only the `Set` and `Get` methods of the property are affected.

**NOTE**

See Chapter 5 for a detailed discussion of how you can avoid having published form fields for components, which will improve encapsulation. But don't rush there: the description requires a good knowledge of Delphi, and the technique discussed has a few drawbacks!

## Constructors

As I've mentioned, to allocate the memory for the object, we call the `Create` method. This is a *constructor*, a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on. The default `TObject.Create` constructor initializes all the data of the new instance to zero.

If you want your instance data to start out with a nonzero value, then you need to write a custom constructor to do that. The new constructor can be called `Create`, or it can have any other name; use the `constructor` keyword in front of it. Notice that you don't need to call `TObject.Create`: it is Delphi that allocates the memory for the new object, not the class constructor. All you have to do is to initialize the class base.

If you create objects without initializing them, calling methods later may result in odd behavior or even a run-time error. A consistent use of constructors to initialize objects' data is an important *preventive* technique to avoid these errors in the first place. For example, we must call the `SetValue` procedure of the `TDate` class after we've created the object. As an alternative, we can provide a customized constructor, which creates the object and gives it an initial value.

Although you can use any name for a constructor, you should stick to the standard name, `Create`. If you use a name other than `Create`, the `Create` constructor of the base `TObject` class will still be available, but a programmer calling this default constructor might bypass the initialization code you've provided because they don't recognize the name.

By defining a `Create` constructor with some parameters, you replace the default definition with a new one and make its use compulsory. For example, after you define

```
type
  TDate = class
  public
    constructor Create (y, m, d: Integer);
```

```
constructor TDate.Create (y, m, d: Integer);  
begin  
    fDate := EncodeDate (y, m, d);  
end;
```

you'll be able to call this constructor and not the standard Create:

```
var  
    ADay: TDate;  
begin  
    // Error, does not compile:  
    ADay := TDate.Create;  
    // OK:  
    ADay := TDate.Create (1, 1, 2000);
```

The rules for writing constructors for custom components are different, as we'll see in Chapter 11. In short, when you inherit from TComponent, you should override the default Create constructor with one parameter and avoid disabling it.

## Overloaded Constructors

*Overloading* is particularly relevant for constructors, because we can add to a class multiple constructors and call them all Create, which makes them easy to remember.

### NOTE

Historically, overloading was added to C++ to allow the use of multiple constructors that have the same name (the name of the class). In Object Pascal, this feature was considered unnecessary because multiple constructors can have different specific names. The increased integration of Delphi with C++Builder has motivated Borland to make this feature available in both languages, starting with Delphi 4. Technically, when C++Builder constructs an instance of a Delphi VCL class, it looks for a Delphi constructor named `Create` and nothing but `Create`. If the Delphi class has constructors by other names, they cannot be used from C++Builder code. Therefore, when creating classes and components you intend to share with C++Builder programmers, you should be careful to name all your constructors `Create` and distinguish between them by their parameter lists (using `overload`). Delphi does not require this, but it is required for C++Builder to use your Delphi classes.

As an example, I've added to the class two separate `Create` constructors: one with no parameters, which hides the default constructor, and one with the initialization values. The constructor with no parameter uses as the default value today's date:

```
type  
    TDate = class  
    public  
        constructor Create; overload;  
        constructor Create (y, m, d: Integer); overload;
```

```
constructor TDate.Create (y, m, d: Integer);  
begin  
    fDate := EncodeDate (y, m, d);  
end;  
  
constructor TDate.Create;  
begin  
    fDate := Date;  
end;
```

Having these two constructors makes it possible to define a new TDate object in two different ways:

```
var  
    Day1, Day2: TDate;  
begin  
    Day1 := TDate.Create (2001, 12, 25);  
    Day2 := TDate.Create; // today
```

See the section “The Complete *TDate* Class” later in this chapter for the DateView example, which includes the code of these constructors.

## Destructors

In the same way that a class can have a custom constructor, it can have a custom destructor, a method declared with the `destructor` keyword and called `Destroy`, which can perform some resource cleanup before an object is destroyed. Just as a constructor call allocates memory for the object, a destructor call frees the memory.

We can write code for a destructor, generally overriding the default `Destroy` destructor, to let the object execute some code before it is destroyed. Destructors are needed only for objects that acquire resources in their constructors or during their lifetime. In your code, of course, you don't have to handle memory de-allocation—this is something Delphi does for you.

`Destroy` is a virtual destructor of the `TObject` class. Most of the classes that require custom clean-up code when the objects are destroyed override this virtual method. The reason you should never define a new destructor is that objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor of the specific class (virtual methods will be discussed later in this chapter).

## ***Free (and nil)***

Free is a method of the TObject class, inherited by all other classes. The Free method basically checks whether the current object (Self) is not nil before calling the Destroy virtual destructor. Here is its pseudocode (the actual Delphi code is written in assembler):

```
procedure TObject.Free;  
begin  
  if Self <> nil then  
    Destroy;  
end;
```

By looking at this code, you can see that calling Free doesn't set the object to nil automatically; this is something you should do yourself! The reason is that the object doesn't know which variables may be referring to it, so it has no way to set them all to nil.

### **NOTE**

Automatically setting an object to nil is not possible. You might have several references to the same object, and Delphi doesn't track them. At the same time, within a method (such as Free) we can operate on the object, but we know nothing about the object reference—the memory address of the variable we've used to call the method. In other words, inside the Free method or any other method of a class, we know the memory address of the object (Self), but we don't know the memory location of the variable referring to the object.

Delphi 5 introduced a FreeAndNil procedure you can use to free an object and set its reference to nil at the same time. Simply call

```
FreeAndNil (Obj1)
```

instead of writing

```
Obj1.Free;  
Obj1 := nil;
```

The FreeAndNil procedure knows about the object reference, passed as a parameter, and can act on it. Here is Delphi code for FreeAndNil:

```
procedure FreeAndNil(var Obj);  
var  
  P: TObject;  
begin  
  P := TObject(Obj);  
  // clear the reference before destroying the object  
  TObject(Obj) := nil;  
  P.Free;  
end;
```

### **NOTE**

There's more on this topic in the section "Destroying Objects Only Once" later in this chapter.

## The Complete *TDate* Class

In the initial portion of this chapter, I've shown you bits and pieces of the source code for different versions of a *TDate* class. In Listing 2.1 is the complete interface portion of the unit that defines the *TDate* class.



**Listing 2.1:** The *TDate* class, from the *ViewDate* example

```
unit Dates;

interface

type
  TDate = class
  private
    fDate: TDateTime;
    procedure SetDay(const Value: Integer);
    procedure SetMonth(const Value: Integer);
    procedure SetYear(const Value: Integer);
    function GetDay: Integer;
    function GetMonth: Integer;
    function GetYear: Integer;
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string;
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
  end;

implementation

uses
  SysUtils, DateUtils;

procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

function TDate.LeapYear: Boolean;
begin
  Result := IsInLeapYear(fDate);
end;
```

```
procedure TDate.Increase (NumberOfDays: Integer = 1);  
begin  
    fDate := fDate + NumberOfDays;  
end;  
  
function TDate.GetText: string;  
begin  
    GetText := DateToStr (fDate);  
end;  
  
procedure TDate.Decrease (NumberOfDays: Integer = 1);  
begin  
    fDate := fDate - NumberOfDays;  
end;  
  
constructor TDate.Create (y, m, d: Integer);  
begin  
    fDate := EncodeDate (y, m, d);  
end;  
  
constructor TDate.Create;  
begin  
    fDate := Date;  
end;  
  
procedure TDate.SetValue(NewDate: TDateTime);  
begin  
    fDate := NewDate;  
end;  
  
procedure TDate.SetDay(const Value: Integer);  
begin  
    fDate := RecodeDay (fDate, Value);  
end;  
  
procedure TDate.SetMonth(const Value: Integer);  
begin  
    fDate := RecodeMonth (fDate, Value);  
end;  
  
procedure TDate.SetYear(const Value: Integer);  
begin  
    fDate := RecodeYear (fDate, Value);  
end;  
  
function TDate.GetDay: Integer;  
begin  
    Result := DayOf (fDate);  
end;
```

```

function TDate.GetMonth: Integer;
begin
    Result := MonthOf (fDate);
end;

function TDate.GetYear: Integer;
begin
    Result := YearOf (fDate);
end;

end.

```

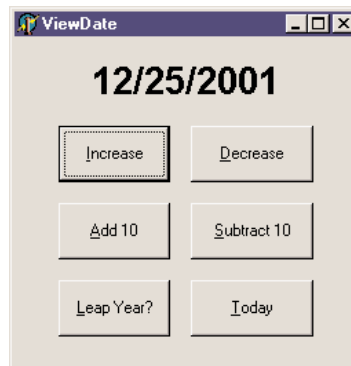
The aim of the Increase and Decrease methods, which have a default value for their parameter, is quite easy to understand. If called with no parameter, they change the value of the date to the next or previous day. If a NumberOfDays parameter is part of the call, they add or subtract that number.

GetText returns a string with the formatted date, using the DateToStr function.

The form of the example I've built to show you how to use the TDate class, as illustrated in Figure 2.4, has a caption to display a date and six buttons, which can be used to modify the date. To make the label component look nice, I've given it a big font, made it as wide as the form, set its Alignment property to taCenter, and set its AutoSize property to False.

**FIGURE 2.4:**

The output of the ViewDate example at startup



The startup code of this program is in the OnCreate event handler. In the corresponding method, we create an instance of the TDate class, initialize this object, and then show its textual description in the Caption of the label.

```

procedure TDateForm.FormCreate(Sender: TObject);
begin
    TheDay := TDate.Create (2001, 12, 25);
    LabelDate.Caption := TheDay.GetText;
end;

```

TheDay is a private field of the class of the form, TDateForm. (By the way, the name for the form class is automatically chosen by Delphi when we change the Name property of the form to DateForm.) The object is then destroyed along with the form:

```
procedure TDateForm.FormDestroy(Sender: TObject);  
begin  
    TheDay.Free;  
end;
```

When the user clicks one of the six buttons, we need to apply the corresponding method to the TheDay object and then display the new value of the date in the label:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);  
begin  
    TheDay.SetValue (Date);  
    LabelDate.Caption := TheDay.GetText;  
end;
```

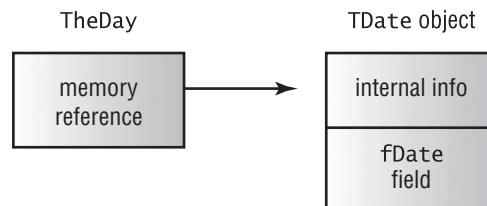
Notice that in this code we reuse an existing object, assigning a new date to it. We could also create a new object and assign it to the existing TheDate variable, but this can lead to confusing situations, as explained in the next section.

## Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Object Pascal, instead, is based on an *object reference model*. The idea is that a variable of a class type, such as the TheDay variable in the preceding ViewDate example, does not hold the value of the object. Rather, it contains a reference, or a *pointer*, to indicate the memory location where the object has been stored. You can see this structure depicted in Figure 2.5.

**FIGURE 2.5:**

A representation of the structure of an object in memory, with a variable referring to it



The only problem with this approach is that when you declare a variable, you don't create an object in memory; you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of the components you place on a form are built automatically by Delphi.



You've seen how to create an instance of an object by applying a constructor to its class. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the `Free` method. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch. The object reference model has many consequences on assigning object and on managing memory, as we'll see in the next two sections.

## Assigning Objects

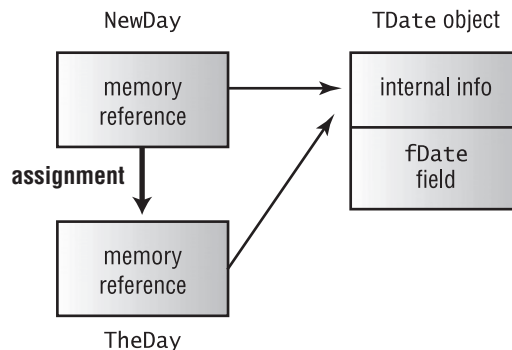
If a variable holding an object only contains a reference to the object in memory, what happens if you copy the value of that variable? Suppose we write the `BtnTodayClick` method of the `ViewDate` example in the following way:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);  
var  
    NewDay: TDate;  
begin  
    NewDay := TDate.Create;  
    TheDay := NewDay;  
    LabelDate.Caption := TheDay.GetText;  
end;
```

This code copies the memory address of the `NewDay` object to the `TheDay` variable (as shown in Figure 2.6); it doesn't copy the data of an object into the other. In this particular circumstance, this is not a very good approach, as we keep allocating memory for a new object every time the button is pressed, but we never release the memory of the object the `TheDay` variable was previously pointing to. This specific issue can be solved by freeing the old object, as in the following code (which is also simplified, without the use of an explicit variable for the newly created object):

**FIGURE 2.6:**

A representation of the operation of assigning an object reference to another one. This is different from copying the actual content of an object to another.



```
procedure TDateForm.BtnTodayClick(Sender: TObject);  
begin  
  TheDay.Free;  
  TheDay := TDate.Create;
```

The important thing to keep in mind is that, when you assign an object to another object, Delphi copies the reference to the object in memory to the new object/reference. You should not consider this a negative: In many cases, being able to define a variable referring to an existing object can be a plus. For example, you can store the object returned by calling a function or accessing a property and use it in subsequent statements, as this code snippet indicates:

```
var  
  ADay: TDate;  
begin  
  ADay: UserInformation.GetBirthDate;  
  // use a ADay
```

The same happens if you pass an object as a parameter to a function: You don't create a new object, but you refer to the same one in two different places of the code. For example, by writing this procedure and calling it as follows, you'll modify the `Caption` property of the `Button1` object, not of a copy of its data in memory (which would be totally useless):

```
procedure CaptionPlus (Button: TButton);  
begin  
  Button.Caption := Button.Caption + '+';  
end;  
  
// call...  
CaptionPlus (Button1)
```

What if you really want to change the data inside an existing object, so that it matches the data of another object? You have to copy each field of the object, which is possible only if they are all public, or provide a specific method to copy the internal data. Some classes of the VCL have an `Assign` method, which does this copy operation. To be more precise, most of the VCL classes inheriting from `TPersistent`, but not inheriting from `TComponent`, have the `Assign` method. Other `TComponent`-derived classes have this method but raise an exception when it is called.

In the `DateCopy` example, slightly modified from the `ViewDate` program, I've added an `Assign` method to the `TDate` class, and I've called it from the `Today` button, with the following code:

```
procedure TDate.Assign (Source: TDate);  
begin  
  fDate := Source.fDate;  
end;
```

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Create;
  TheDay.Assign(NewDay);
  LabelDate.Caption := TheDay.GetText;
  NewDay.Free;
end;
```

## Objects and Memory

Memory management in Delphi is subject to three rules: Every object must be created before it can be used; every object must be destroyed after it has been used; and every object must be destroyed only once. Whether you have to do these operations in your code, or you can let Delphi handle memory management for you, depends on the model you choose among the different approaches provided by Delphi.

Delphi supports three types of memory management for dynamic elements (that is, elements not in the stack and the global memory area):

- Every time you create an object explicitly, in the code of your application, you should also free it. If you fail to do so, the memory used by that object won't be released for other objects until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form) becomes responsible for destroying all the objects it owns. In other words, when you free the form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it. This is the standard behavior of the components you create at design time by placing them on a form or data module.
- When you allocate memory for strings, dynamic arrays, and objects referenced by interface variables (discussed in Chapter 3), Delphi automatically frees the memory when the reference goes out of scope. You don't need to free a string: when it becomes unreachable, its memory is released.

### Destroying Objects Only Once

Another problem is that if you call the `Destroy` destructor of an object twice, you get an error. If you remember to set the object to `nil`, you can call `Free` twice with no problem.

**NOTE**

You might wonder why you can safely call `Free` if the object reference is `nil`, but you can't call `Destroy`. The reason is that `Free` is a known method at a given memory location, whereas the virtual function `Destroy` is determined at run time by looking at the type of the object, a very dangerous operation if the object doesn't exist any more.

To sum things up, here are a couple of guidelines:

- Always call `Free` to destroy objects, instead of calling the `Destroy` destructor.
- Use `FreeAndNil`, or set object references to `nil` after calling `Free`, unless the reference is going out of scope immediately afterward.

In general, you can also check whether an object is `nil` by using the `Assigned` function. So the following two statements are equivalent, at least in most cases:

```
if Assigned (ADate) then ...  
if ADate <> nil then ...
```

Notice that these statements test only whether the pointer is not `nil`; they do not check whether it is a valid pointer. If you write the following code, the test will be satisfied, and you'll get an error on the line with the call to the method of the object:

```
ToDestroy.Free;  
if ToDestroy <> nil then  
    ToDestroy.DoSomething;
```

It is important to realize that calling `Free` doesn't set the object to `nil`.

## What's Next?

In this chapter, we have discussed the foundations of object-oriented programming (OOP) in Object Pascal. We have considered the definition of classes, the use of methods, encapsulation, and memory management, but also some more advanced concepts such as properties and the dynamic creation of components.

This is certainly a lot of information if you are a newcomer, but if you are fluent in another OOP language or if you've already used past versions of Delphi, you should be able to apply the topics covered in this chapter to your programming.

The next chapter continues on the same line, highlighting inheritance in particular, along with virtual functions and interfaces. It also includes a discussion on exception handling and

class references, so that at the end you'll have a complete overview of the language. At that point, you'll be ready to start focusing on the libraries the compiler relies on, and we'll get back to see how properties are used by Delphi and its IDE (in Chapter 5). Other chapters will provide further information on applying the OOP concepts to Delphi programming. You'll find OOP tips throughout the entire book, but particularly in Chapter 11, devoted to writing custom Delphi components.

# The Object Pascal Language: Inheritance and Polymorphism

- Inheritance
- Virtual methods
- Polymorphism
- Type-safe down-casting (run-time type information)
- Interfaces
- Working with exceptions
- Class references

**A**fter the introduction to classes and objects we've seen over the last chapter, let's move on to another key element of the language, *inheritance*. Deriving a class from an existing one is the real revolutionary idea of object-oriented programming, and it goes along with polymorphism, virtual functions, abstract functions, and many other topics discussed in this chapter.

We'll focus also on interfaces, another intriguing idea of the most recent OOP languages, and we'll cover a few more elements of Object Pascal, such as exception handling and class references. Together with the previous chapter, this will provide an almost complete roundup of the language.

## Inheriting from Existing Types

We often need to use a slightly different version of an existing class that we have written or that someone has given to us. For example, you might need to add a new method or slightly change an existing one. You can do this easily by modifying the original code, unless you want to be able to use the two different versions of the class in different circumstances. Also, if the class was originally written by someone else (including Borland), you might want to keep your changes separate.

A typical alternative is to make a copy of the original type definition, change its code to support the new features, and give a new name to the resulting class. This might work, but it also might create problems: In duplicating the code you also duplicate the bugs; and if you want to add a new feature, you'll need to add it two or more times, depending on the number of copies of the original code you've made. This approach results in two completely different data types, so the compiler cannot help you take advantage of the similarities between the two types.

To solve these kinds of problems in expressing similarities between classes, Object Pascal allows you to define a new class directly from an existing one. This technique is known as *inheritance* (or *subclassing*) and is one of the fundamental elements of object-oriented programming languages. To inherit from an existing class, you only need to indicate that class at the beginning of the declaration of the subclass. For example, Delphi does this automatically each time you create a new form:

```
type
  TForm1 = class(TForm)
end;
```

This simple definition indicates that the TForm1 class inherits all the methods, fields, properties, and events of the TForm class. You can apply any public method of the TForm class to an object of the TForm1 type. TForm, in turn, inherits some of its methods from another class, and so on, up to the TObject base class.

As an example of inheritance, we can change the ViewDate program, deriving a new class from TDate and modifying its GetText function. You can find this code in the DATES.PAS file of the NewDate example on the companion CD.

```
type
  TNewDate = class (TDate)
public
  function GetText: string;
end;
```

In this example, the TNewDate class is derived from TDate. It is common to say that TDate is an *ancestor* class or *parent* class of TNewDate and that TNewDate is a *subclass*, *descendant* class, or *child* class of TDate.

To implement the new version of the GetText function, I used the FormatDateTime function, which uses (among other features) the predefined month names available in Windows; these names depend on the user's regional and language settings. Many of these regional settings are actually copied by Delphi into constants defined in the library, such as LongMonthNames, ShortMonthNames, and many others you can find under the "Currency and date/time formatting variables" topic in the Delphi Help file. Here is the GetText method, where 'dddddd' stands for the long date format:

```
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

## TIP

Using regional information, the NewDate program automatically adapts itself to different Windows user settings. If you run this same program on a computer with regional settings referring to a language other than English, it will automatically show month names in that language. To test this behavior, you just need to change the regional settings; you don't need a new version of Windows. Notice that regional-setting changes immediately affect the running programs.

Once we have defined the new class, we need to use this new data type in the code of the form of the NewDate example. Simply define the TheDay object of type TNewDate, and call its constructor in the FormCreate method:

```
type
  TDateForm = class(TForm)
  ...
private
  TheDay: TNewDate; // updated declaration
end;
```

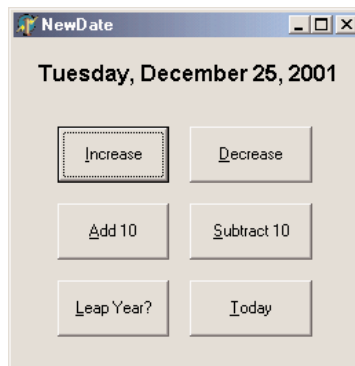


```
procedure TDateForm.FormCreate(Sender: TObject);  
begin  
    TheDay := TNewDate.Create (2001, 12, 25); // updated  
    DateLabel.Caption := TheDay.GetText;  
end;
```

Without any other changes, the new NewDate example will work properly. The TNewDate class inherits the methods to increase the date, add a number of days, and so on. In addition, the older code calling these methods still works. Actually, to call the new version of the GetText method, we don't need to change the source code! The Delphi compiler will automatically bind that call to a new method. The source code of all the other event handlers remains exactly the same, although its meaning changes considerably, as the new output demonstrates (see Figure 3.1).

**FIGURE 3.1:**

The output of the NewDate program, with the name of the month and of the day depending on Windows regional settings



## Protected Fields and Encapsulation

The code of the GetText method of the TNewDate class compiles only if it is written in the same unit as the TDate class. In fact, it accesses the fDate private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the fDate field as protected or add a protected access method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write subclasses. However, this violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If ten derived classes are accessing this data, changing its definition means potentially modifying the code in each of the ten classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate solution can be obtained by using a virtual method, a topic I'll discuss in detail later in the section "Late Binding and Polymorphism." If you choose not to use encapsulation in order to obtain faster coding of the subclasses, then your design might not follow the object-oriented principles.

## Accessing Protected Data of Other Classes

We've seen that in Delphi, the `private` and `protected` data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this class (part of the Protection example on the companion CD):

```
type
  TTest = class
    protected
      ProtectedData: Integer;
    public
      PublicData: Integer;
      function GetValue: string;
    end;
```

The `GetValue` method simply returns a string with the two integer values:

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

Once you place this class in its own unit, you won't be able to access its protected portion from other units directly. Accordingly, if you write the following code,

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20; // won't compile
  ShowMessage (Obj.GetValue);
  Obj.Free;
end;
```

the compiler will issue an error message, "Undeclared identifier: 'ProtectedData.'"

*Continued on next page*

At this point, you might think there is no way to access the protected data of a class defined in a different unit. (This is what Delphi manuals and most Delphi books say.) However, there is a way around it. Consider what happens if you create an apparently useless derived class, such as

```
type  
  TFake = class (TTest);
```

Now, if you make a direct cast of the object to the new class and access the protected data through it, this is how the code will look:

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
  Obj: TTest;  
begin  
  Obj := TTest.Create;  
  Obj.PublicData := 10;  
  TFake (Obj).ProtectedData := 20; // compiles!  
  ShowMessage (Obj.GetValue);  
  Obj.Free;  
end;
```

This code compiles and works properly, as you can see by running the Protection program. How is it possible for this approach to work? Well, if you think about it, the TFake class automatically inherits the protected fields of the TTest base class, and because the TFake class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible. As you would expect, if you move the declaration of the TFake class to a secondary unit, the program won't compile any more.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are times when using this technique is the best solution, as you'll see by looking at the VCL source code and the code of many Delphi components. Two examples that come to mind are accessing the Text property of the TControl class and the Row and Col positions of the DBGrid control. These two ideas are demonstrated by the TextProp and DBGridCol examples, respectively. (These examples are quite advanced, so I suggest that only programmers with a good background of Delphi programming read them at this point in the text—other readers might come back later.) Although the first example shows a reasonable example of using the typecast *cracker*, the DBGrid example of Row and Col is actually a counterexample, one that illustrates the risks of accessing bits that the class writer chose not to expose. The row and column of a DBGrid do not mean the same thing as they do in a DrawGrid or StringGrid (the base classes). First, DBGrid does not count the fixed cells as actual cells (it distinguishes data cells

*Continued on next page*

from decoration), so your row and column indexes will have to be adjusted by whatever decorations are currently in effect on the grid (and those can change on the fly). Second, the `DBGrid` is a virtual view of the data. When you scroll up in a `DBGrid`, the data may move underneath it, but the currently selected row might not change.

This technique—declaring a local type only so that you can access protected data members of a class—is often described as a *hack*, and it should be avoided whenever possible. The problem is not accessing protected data of a class in the same unit but declaring a class for the sole purpose of accessing protected data of an existing object of a different class! The danger of this technique is in the hard-coded typecast of an object from a class to a different one.

## Inheritance and Type Compatibility

Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a `Boolean` variable, unless you use an explicit typecast. The rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type refers to a single type definition.

**WARNING** If you redefine the same data type in two different units, they won't be compatible, even if their name is identical. A program using two equally named types of two different units will be a nightmare to compile and debug.

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. That is because a dog is an animal! So, although this might surprise you, the following constructor calls are both legal:

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

As a general rule, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
type
  TDog = class (TAnimal)
    ...
  end;
```

```

var
  MyAnimal: TAnimal;
  MyDog: TDog;

begin
  MyAnimal := MyDog; // This is OK
  MyDog := MyAnimal; // This is an error!!!

```

Before we look at the implications of this important feature of the language, you can try out the `Animals1` example from the companion CD, which defines the two `TAnimal` and `TDog` classes:

```

type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;

```

The two `Create` methods set the value of `Kind`, which is returned by the `GetKind` function. The form displayed by this example, shown in Figure 3.2, has a private field `MyAnimal` of type `TAnimal`. An instance of this class is created and initialized when the form is created and each time one of the radio buttons is selected:

```

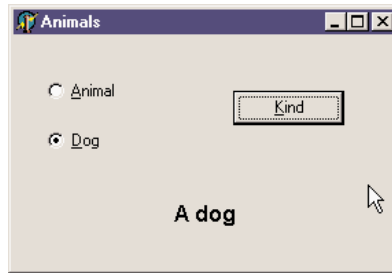
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioDogClick(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;

```

**FIGURE 3.2:**

The form of the Animals1 example



Finally, the Kind button calls the `GetKind` method for the current animal and displays the result in the label:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);  
begin  
    KindLabel.Caption := MyAnimal.GetKind;  
end;
```

## Late Binding and Polymorphism

Pascal functions and procedures are usually based on *static* or *early binding*. This means that a method call is resolved by the compiler and linker, which replace the request with a call to the specific memory location where the function or procedure resides. (This is known as the *address* of the function.) OOP languages allow the use of another form of binding, known as *dynamic* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

The advantage of this technique is known as *polymorphism*. Polymorphism means you can write a call to a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the actual class of the object the variable refers to, because of the type-compatibility rule discussed in the previous section.

### NOTE

The term *polymorphism* is quite a mouthful. A glance at the dictionary tells us that in a general sense, it refers to something having more than one form. In the OOP sense, then, it refers to the facts that there may be several versions of a given method across several related classes and that a single method call on an object instance of a particular class type can refer to one of these versions. Which version of the method gets called depends on the type of the object instance used to make the call at run time.

For example, suppose that a class and its subclass (let's say `TAnimal` and `TDog`) both define a method, and this method has late binding. Now you can apply this method to a generic variable, such as `MyAnimal`, which at run time can refer either to an object of class `TAnimal` or to an object of class `TDog`. The actual method to call is determined at run time, depending on the class of the current object.

The `Animals2` example extends the `Animals1` program to demonstrate this technique. In the new version, the `TAnimal` and the `TDog` classes have a new method: `Voice`, which means to output the sound made by the selected animal, both as text and as sound. This method is defined as `virtual` in the `TAnimal` class and is later overridden when we define the `TDog` class, by the use of the `virtual` and `override` keywords:

```

type
  TAnimal = class
    public
      function Voice: string; virtual;

  TDog = class (TAnimal)
    public
      function Voice: string; override;

```

Of course, the two methods also need to be implemented. Here is a simple approach:

```

uses
  MMSystem;

function TAnimal.Voice: string;
begin
  Voice := 'Voice of the animal';
  PlaySound ('Anim.wav', 0, snd_Async);
end;

function TDog.Voice: string;
begin
  Voice := 'Arf Arf';
  PlaySound ('dog.wav', 0, snd_Async);
end;

```

### TIP

This example uses a call to the `PlaySound` API function, defined in the `MMSystem` unit. The first parameter of this function is the name of the WAV sound file or the system sound you want to execute. The second parameter indicates an optional resource file containing the sound. The third parameter indicates (among other options) whether the call should be synchronous or asynchronous; that is, whether the program should wait for the sound to finish before continuing with the following statements.

Now what is the effect of the call `MyAnimal.Voice`? It depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice` instead. This happens only because the function is `virtual` (as you can experiment by removing this keyword and recompiling).

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined in other units—or that haven't been written yet! The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` subclasses.

**NOTE**

This is the key technical reason why object-oriented programming languages favor reusability. You can write code that uses classes within a hierarchy without any knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition: the ancestor classes of the hierarchy need to be designed very carefully.

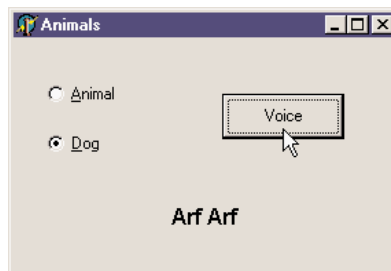
The `Animals2` program demonstrates the use of these new classes and has a form similar to that of the previous example. This code is executed by clicking the button:

```
procedure TFormAnimals.BtnVerseClick(Sender: TObject);  
begin  
    LabelVoice.Caption := MyAnimal.Voice;  
end;
```

In Figure 3.3, you can see an example of the output of this program. By running it, you'll also hear the corresponding sounds produced by the `PlaySound` API call.

**FIGURE 3.3:**

The output of the `Animals2` example





## Overriding and Redefining Methods

As we have just seen, to override a late-bound method in a descendant class, you need to use the `override` keyword. Note that this can take place only if the method was defined as `virtual` in the ancestor class. Otherwise, if it was a static method, there is no way to activate late binding, other than by changing the code of the ancestor class.

The rules are simple: A method defined as static remains static in every subclass, unless you hide it with a new virtual method having the same name. A method defined as `virtual` remains late-bound in every subclass. There is no way to change this, because of the way the compiler generates different code for late-bound methods.

### NOTE

The new C# programming language proposed by Microsoft (which is in essence a clone of Java) has the same notion as the Object Pascal language of marking the overridden version of a method with a specific keyword.

To redefine a static method, you add a method to a subclass having the same parameters or different parameters than the original one, without any further specifications. To override a `virtual` method, you must specify the same parameters and use the `override` keyword:

```
type
  MyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;

  MySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

There are typically two ways to override a method. One is to replace the method of the ancestor class with a new version. The other is to add some more code to the existing method. This can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write

```
procedure MySubClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

You might wonder why you need to use the `override` keyword. In other languages, when you redefine a method in a subclass, you automatically override the original one. However,

having a specific keyword allows the compiler to check the correspondence between the names of the methods of the ancestor class and the subclass (misspelling a redefined function is a common error in other OOP languages), check that the method was virtual in the ancestor class, and so on.

When you override an existing virtual method of a base class, you must use the same parameters. When you introduce a new version of a method in a descendent class, you can declare it with the parameters you want. In fact, this will be a new method unrelated to the ancestor method of the same name. They only happen to use the same name. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string);
  end;
```

#### NOTE

Using the class definitions above, when you create an object of the `TMySubClass` class, you can apply to it the `One` method with the string parameter, but not the parameter-less version defined in the base class. If this is what you need, it can be accomplished by marking the re-declared method (the one in the derived class) with the `overload` keyword. If the method has different parameters than the version in the base class, it becomes effectively an overloaded method; otherwise it replaces the base class method. Notice that the method doesn't need to be marked as `overload` in the base class. However, if the method in the base class is virtual, the compiler issues the warning "Method 'One' hides virtual method of base type 'TMyClass.'" To avoid this message and to instruct the compiler more precisely on your intentions, you can use the `reintroduce` directive. If you are interested in this advanced topic, you can find this code in the `Reintr` example on the companion CD and experiment with it further.

## Virtual versus Dynamic Methods

In Delphi, there are two different ways to activate late binding. You can declare the method as `virtual`, as we have seen before, or declare it as `dynamic`. The syntax of these two keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

`virtual` methods are based on a *virtual method table* (VMT, also known as a *vtable*), which is an array of method addresses. For a call to a `virtual` method, the compiler generates code to jump to an address stored in the *n*th slot in the object's virtual method table.

Virtual method tables allow fast execution of the method calls. Their main drawback is that they require an entry for each `virtual` method for each descendant class, even if the method is not overridden in the subclass. At times, this has the effect of propagating VMT entries throughout a class hierarchy (even for methods that aren't redefined). This might require a lot of memory just to store the same method address multiple times.

Dynamic method calls, on the other hand, are dispatched using a unique number indicating the method. The search for the corresponding function is generally slower than the one-step table lookup for `virtual` methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method. For large or deep object hierarchies, using dynamic methods instead of `virtual` methods can result in significant memory savings with only a minimal speed penalty.

From a programmer's perspective, the difference between these two approaches lies only in a different internal representation and slightly different speed or memory usage. Apart from this, `virtual` and dynamic methods are the same.

## Message Handlers

A late-bound method can be used to handle a Windows message, too, although the technique is somewhat different. For this purpose Delphi provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter. The `message` directive is followed by the number of the Windows message the method wants to handle.

### WARNING

The `message` directive is also available in Delphi for Linux and is fully supported by the language and the RTL. However, the visual portion of the CLX application framework does not use message methods to dispatch notifications to controls. For this reason, whenever possible, you should use a virtual method provided by the library rather than handle a Windows message directly. Of course, this matters only if you want your code to be more portable.

For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `wm_User` Windows constant:

```
type
  TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage);
      message wm_User;
    end;
```

The name of the procedure and the actual type of the parameters are up to you, although there are several predefined record types for the various Windows messages. You could later send this message, invoking the corresponding method, by writing:

```
PostMessage (Form1.Handle, wm_User, 0, 0);
```

This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions. You can also dispatch a message to an object by calling the `TObject.Dispatch` method on the object. This will be a synchronous message call, not asynchronous like `PostMessage`. `TObject.Dispatch` is fully platform independent.

The ability to handle Windows messages and call API functions as you do when you are programming Windows with the C language may horrify some programmers and delight others. But in Delphi, when writing Windows applications, you will seldom need to use message methods or call Windows APIs directly. Obviously, these techniques will also affect the portability of your code to other platforms.

## Abstract Methods

The `abstract` keyword is used to declare methods that will be defined only in subclasses of the current class. The `abstract` directive fully defines the method; it is not a forward declaration. If you try to provide a definition for the method, the compiler will complain. In Object Pascal, you can create instances of classes that have abstract methods. However, when you try to do so, Delphi's 32-bit compiler issues the warning message "Constructing instance of <class name> containing abstract methods." If you happen to call an abstract method at run time, Delphi will raise an exception, as demonstrated by the following `Animals3` example.

### NOTE

C++ and Java use a more strict approach: in these languages, you cannot create instances of classes containing abstract methods.

You might wonder why you would want to use abstract methods. The reason lies in the use of polymorphism. If class `TAnimal` has the abstract method `Voice`, every subclass can redefine it. The advantage is that you can now use the generic `MyAnimal` object to refer to each animal defined by a subclass and invoke this method. If this method was not present in the interface of the `TAnimal` class, the call would not have been allowed by the compiler, which performs static type checking. Using a generic `MyAnimal` object, you can call only the method defined by its own class, `TAnimal`.

You cannot call methods provided by subclasses, unless the parent class has at least the declaration of this method—in the form of an abstract method. The next example, `Animals3`, demonstrates the use of abstract methods and the abstract call error. In Listing 3.1, you can see the interfaces of the classes of this new example. (Here `TAnimal` is an abstract class.)

**Listing 3.1: Declaration of the three classes of the Animals3 example**

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Voice: string; virtual; abstract;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;

  TCat = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;
```

The most interesting portion of Listing 3.1 is the definition of the class `TAnimal`, which includes a virtual abstract method: `Voice`. It is also important to notice that each derived class overrides this definition and adds a new virtual method, `Eat`. What are the implications of these two different approaches? To call the `Voice` function, we can write the same code as in the previous version of the program:

```
LabelVoice.Caption := MyAnimal.Voice;
```

How can we call the `Eat` method? We cannot apply it to an object of the `TAnimal` class. The statement

```
LabelVoice.Caption := MyAnimal.Eat;
```

generates the compiler error “Field identifier expected.”

To solve this problem, you can use run-time type information (RTTI) to cast the `TAnimal` object to a `TCat` or `TDog` object; but without the proper cast, the program will raise an exception. You will see an example of this approach in the next section. Adding the method definition to the `TAnimal` class is a typical solution to the problem, and the presence of the `abstract` keyword favors this choice.

**NOTE**

What happens if a method overriding an abstract method calls `inherited`? In past versions of Delphi, this resulted in an abstract method call. In Delphi 6, the compiler has been enhanced to notice the presence of the abstract method and simply skip the `inherited` call. This means you can safely always use `inherited` in every overridden method, unless you specifically want to disable executing some code of the base class.

## Type-Safe Down-Casting

The Object Pascal type-compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned earlier, the reverse is not possible.

Now suppose that the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, it should be possible to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, we could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods we are calling actually exist.

To solve the problem, we can use techniques based on *run-time type information* (RTTI, for short). Essentially, because each object “knows” its type and its parent class, and we can ask for this information with the `is` operator or using the `InheritsFrom` method of the `TObject` class. The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then ...
```

The `is` expression evaluates as `True` only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from `TDog`. This means that if you test whether a `TDog` object is of type `TAnimal`, the test will succeed. In other words, this expression evaluates as `True` if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

Now that you know for sure that the animal is a dog, you can make a safe typecast (or type conversion). You can accomplish this direct cast by writing the following code:

```
var
  MyDog: TDog;
begin
  if MyAnimal is TDog then
  begin
    MyDog := TDog (MyAnimal);
    Text := MyDog.Eat;
  end;
```

This same operation can be accomplished directly by the second RTTI operator, `as`, which converts the object only if the requested class is compatible with the actual one. The parameters of the `as` operator are an object and a class type, and the result is an object converted to the new class type. We can write the following snippet:

```
MyDog := MyAnimal as TDog;  
Text := MyDog.Eat;
```

If we only want to call the `Eat` function, we might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second raises an exception if the type of the object is incompatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described at the end of this chapter).

To avoid this exception, use the `is` operator and, if it succeeds, make a plain typecast (in fact, there is no reason to use `is` and `as` in sequence, doing the type check twice):

```
if MyAnimal is TDog then  
    TDog(MyAnimal).Eat;
```

Both RTTI operators are very useful in Delphi because you often want to write generic code that can be used with several components of the same type or even of different types. When a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`), so you often need to cast it back to the original component type:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if Sender is TButton then  
        ...  
end;
```

This is a common technique in Delphi, and I'll use it in examples throughout the book. The two RTTI operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism first. Only in special cases, where polymorphism alone cannot be applied, should you try using the RTTI operators to complement it. *Do not use RTTI instead of polymorphism.* This is bad programming practice, and it results in slower programs. RTTI, in fact, has a negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As we have seen, virtual method calls require just a memory lookup, which is much faster.

**NOTE**

There is actually more to run-time type information (RTTI) than the `is` and `as` operators. You can access to detailed class and type information at run time, particularly for published properties, events, and methods. More on this topic in Chapter 5.

## Using Interfaces

When you define an abstract class to represent the base class of a hierarchy, you can come to a point in which the abstract class is so abstract that it only lists a series of virtual functions without providing any actual implementation. This kind of *purely abstract class* can also be defined using a specific technique, an `interface`. For this reason, we refer to these classes as *interfaces*.

Technically, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element with distinctive features:

- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to how Delphi manages long strings and makes memory management almost automatic.
- A class can inherit from a single base class, but it can implement multiple interfaces.
- As all classes descend from `TObject`, all interfaces descend from `IInterface`, forming a totally separate hierarchy.

The base interface class used to be `IUnknown` until Delphi 5, but Delphi 6 introduces a new name for it, `IInterface`, to mark even more clearly the fact that this language feature is separate from Microsoft's COM. In fact, Delphi interfaces are available also in the Linux version of the product.

You can use this rule: Interface types describing things that relate to COM and the related operating-system services should inherit from `IUnknown`. Interface types that describe things that do not necessarily require COM (for example, interfaces used for the internal application structure) should inherit from `IInterface`. Doing this consistently in your applications will make it easier to identify which portions of your application probably assume or require the Windows operating system and which portions are probably OS-independent.

**NOTE**

Borland introduced interfaces in Delphi 3 along with the support COM programming. Though the interface language syntax may have been created to support COM, interfaces do not require COM. You can use interfaces to implement abstraction layers within your applications, without building COM server objects. For example, the Delphi IDE uses interfaces extensively in its internal architecture. COM is discussed in Chapter 19.



From a more general point of view, interfaces support a slightly different object-oriented programming model than classes. Objects implementing interfaces are subject to polymorphism for each of the interfaces they support. Indeed, the interface-based model is powerful. But having said that, I'm not interested in trying to assess which approach is better in each case. Certainly, interfaces favor encapsulation and provide a looser connection between classes than inheritance. Notice that the most recent OOP languages, from Java to C#, have the notion of interfaces.

Here is the syntax of the declaration of an interface (which, by convention, starts with the letter *I*):

```
type
  ICanFly = interface
    ['{EAD9C4B4-E1C5-4CF4-9FA0-3B812C880A21}']
    function Fly: string;
  end;
```

The above interface has a GUID, a numeric ID following its declaration and based on Windows conventions. You can generate these identifiers (called GUIDs in jargon) by pressing Ctrl+Shift+G in the Delphi editor.

Although you can compile and use interfaces even without specifying a GUID (as in the code above) for them, you'll generally want to do it, as this is required to perform `QueryInterface` or dynamic as typecasts using that interface type. Since the whole point of interfaces is (usually) to take advantage of greatly extended type flexibility at run time, if compared with class types, interfaces without GUIDs are not very useful.

Once you've declared an interface, you can define a class to implement it, as in:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

The RTL already provides a few base classes to implement the basic behavior required by the `IInterface` interface. The simplest one is the `TInterfacedObject` class I've used in this code.

You can implement interface methods with static methods (as in the code above) or with virtual methods. You can override virtual methods in subclasses by using the `override` directive. If you don't use virtual methods, you can still provide a new implementation in a subclass by redeclaring the interface type in the subclass, rebinding the interface methods to new versions of the static methods. At first sight, using virtual methods to implement interfaces seems to allow for smoother coding in subclasses, but both approaches are equally powerful and flexible. However, the use of virtual methods affects code size and memory.

**NOTE**

The compiler has to generate stub routines to fix up the interface call entry points to the matching method of the implementing class, and adjust the `self` pointer. The interface method stubs for static methods are very simple: adjust `self` and jump to the real method in the class. The interface method stubs for virtual methods are much more complicated, requiring about four times more code (20 to 30 bytes) in each stub than the static case. Also, adding more virtual methods to the implementing class just bloats the virtual method table (VMT) that much more in the implementing class and all its descendents. Interfaces already have their own VMT, and redeclaring interfaces in descendents to rebind the interface to new methods in the descendent is just as polymorphic as using virtual methods, but much smaller in code size.

Now that we have defined an implementation of the interface, we can write some code to use an object of this class, as usual:

```
var
    Airplane1: TAirplane;
begin
    Airplane1 := TAirplane.Create;
    Airplane1.Fly;
    Airplane1.Free;
end;
```

But we can also use an interface-type variable:

```
var
    Flyer1: ICanFly;
begin
    Flyer1 := TAirplane.Create;
    Flyer1.Fly;
end;
```

As soon as you assign an object to an interface-type variable, Delphi automatically checks to see whether the object implements that interface, using the `as` operator. You can explicitly express this operation as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

**NOTE**

The compiler generates different code for the `as` operator when used with interfaces or with classes. With classes, the compiler introduces run-time checks to verify that the object is effectively “type-compatible” with the given. With interfaces, the compiler sees at compile time that it can extract the necessary interface from the available class type, so it does. This operation is like a “compile-time `as`,” not something that exists at run time.

Whether we use the direct assignment or the `as` statement, Delphi does one extra thing: it calls the `_AddRef` method of the object (defined by `IInterface` and implemented by `TInterfacedObject`), increasing its reference count. At the same time, as soon as the

Flyer1 variable goes out of scope, Delphi calls the `_Release` method (again part of `IInterface`), which decreases the reference count, checks whether the reference count is zero, and if necessary, destroys the object. For this reason in the listing above, there is no code to free the object we've created.

In other words, in Delphi, objects referenced by interface variables are reference-counted, and they are automatically de-allocated when no interface variable refers to them any more.

**WARNING** When using interface-based objects, you should generally access them only with object variables or only with interface variables. Mixing the two approaches breaks the reference counting scheme provided by Delphi and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables.

## Interface Properties, Delegation, Redefinitions, Aggregation, and Reference Counting Blues

To demonstrate a few technical elements related to interfaces, I've written the `IntfDemo` example. This example is based on two different interfaces, `IWalker` and `IJumper`, defined as follows:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
  property Position: Integer read GetPos write SetPos;
end;
```

```
IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
  property Position: Integer read GetPos write SetPos;
end;
```

Notice that the first interface also defines a property. An interface property is just a name mapped to a read and a write method. You cannot map an interface property to a field, simply because an interface cannot have a data field.

Here comes a sample implementation of the `IWalker` interface. Notice that you don't have to define the property, only its access methods:

```
TRunner = class (TInterfacedObject, IWalker)
private
    Pos: Integer;
public
    function Walk: string;
    function Run: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
end;
```

The code is trivial, so I'm going to skip it (you can find it in the `IntfDemo` example, where there is also a destructor showing a message, used to verify that reference counting works properly). I've implemented the same interface also in another class, `TAthlete`, that I'll discuss in a second.

As I want to implement also the `IJumper` interface in two different classes, I've followed a different approach. Delphi allows you to delegate the implementation of an interface inside a class to an object exposed with a property. In other words, I want to share the actual implementation code for an interface implemented by several unrelated classes.

To support this technique, Delphi has a special keyword, `implements`. For example, you can write:

```
TMyJumper = class (TInterfacedObject, IJumper)
private
    fJumpImpl: IJumper;
public
    constructor Create;
    property Jumper: IJumper read fJumpImpl implements IJumper;
end;
```

In this case the property refers to an interface variable, but you can also use a plain object variable (my preferred approach). The constructor is required for initializing the internal *implementation* object:

```
    constructor TMyJumper.Create;
begin
    fJumpImpl := TJumperImpl.Create;
end;
```

As a first attempt (and in the last edition of the book), I defined the implementation class as follows:

```
TJumperImpl = class (TInterfacedObject, IJumper)
private
    Pos: Integer;
public
    function Jump: string;
```

```

function Walk: string;
procedure SetPos (Value: Integer);
function GetPos: Integer;
end;

```

If you try this code, the program will compile and everything will run smoothly, until you try to check out what happens with reference counting. It won't work, period. The problem lies in the fact that when the program extracts the `IJumper` interface from the `TMyJumper` object, it actually increases and decreases the reference counting of the inner object, instead of the external one. In other words, you have a single compound object and two separate reference counts going on. This can lead to objects being both kept in memory and released too soon.

The solution to this problem is to have a single reference count, by redirecting the `_AddRef` and `_Release` calls of the internal object to the external one (actually we need to do the same also for `QueryInterface`). In the example, I've used the `TAggregatedObject` provided in Delphi 6 by the system unit; refer to the sidebar "Implementing Aggregates" for more details.

As a result of this approach, the implementation class is now defined as follows:

```

TJumperImpl = class (TAggregatedObject, IJumper)
private
  Pos: Integer;
public
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer read GetPos write SetPos;
end;

```

An object using this class for implementing the `IJumper` interface must have a `Create` constructor, to create the internal object, and a destructor, to destroy it. The constructor of the aggregate object requires the container object as parameter, so that it can redirect back the `IInterface` calls. The key element, of course, is the property mapped to the interface with the `implements` keyword:

```

TMyJumper = class (TInterfacedObject, IJumper)
private
  fJumpImpl: TJumperImpl;
public
  constructor Create;
  property Jumper: TJumperImpl read fJumpImpl implements IJumper;
  destructor Destroy; override;
end;

constructor TMyJumper.Create;
begin
  fJumpImpl := TJumperImpl.Create (self);
end;

```

This example is simple, but in general, things get more complex as you start to modify some of the methods or add other methods that still operate on the data of the internal `fJumpImpl` object. This final step is demonstrated, along with other features, by the `TAthlete` class, which implements both the `IWalker` and `IJumper` interfaces:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)
private
    fJumpImpl: TJumperImpl;
public
    constructor Create;
    destructor Destroy; override;
    function Run: string; virtual;
    function Walk1: string; virtual;
    function IWalker.Walk = Walk1;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;

    property Jumper: TJumperImpl read fJumpImpl implements IJumper;
end;
```

One of the interfaces is implemented directly, whereas the other is delegated to the internal `fJumpImpl` object. Notice also that by implementing two interfaces that have a method in common, we end up with a name clash. The solution is to rename one of the methods, with the statement

```
function IWalker.Walk = Walk1;
```

This declaration indicates that the class implements the `Walk` method of the `IWalker` interface with a method called `Walk1` (instead of with a method having the same name). Finally, in the implementation of all of the methods of this class, we need to refer to the `Position` property of the `fJumpImpl` internal object. By declaring a new implementation for the `Position` property, we'll end up with two positions for a single athlete, a rather odd situation. Here are a couple of examples:

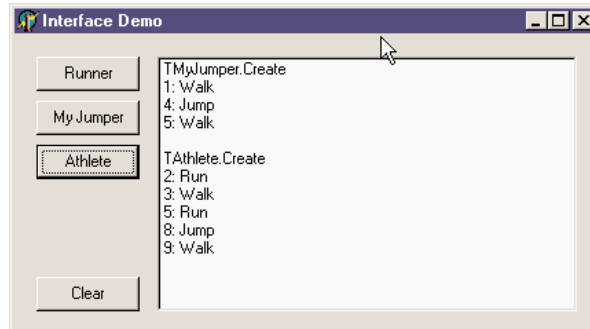
```
function TAthlete.GetPos: Integer;
begin
    Result := fJumpImpl.Position;
end;

function TAthlete.Run: string;
begin
    fJumpImpl.Position := fJumpImpl.Position + 2;
    Result := IntToStr (fJumpImpl.Position) + ': Run';
end;
```

You can further experiment with the `IntfDemo` example, which has a simple form with buttons to create and call methods of the various objects. Nothing fancy, though, as you can see in Figure 3.4. Simply keep in mind that each call returns the position after the requested

movement and a description of the movement itself. Also, each object notifies with a message when it is destroyed.

**FIGURE 3.4:**  
The IntfDemo example



## Implementing Aggregates

As mentioned, when you want to use an internal object to implement an interface, you are faced with reference counting problems. Of course, you can provide your own version of the `_AddRef` and `_Release` methods of `IInterface`, but having a ready-to-use solution might help. In fact, `QueryInterface` on the internal object must also be reflected to the outer object. The user of the interface (whether it works on the outer object or the internal one) should never be able to discern any difference in behavior between `_AddRef`, `_Release`, and `QueryInterface` calls on the aggregated interface and any other interface obtained from the implementing class.

Borland provides a solution to this problem with the `TAggregatedObject` class. In past version of Delphi, this was defined in the `ComObj` unit, but now it has been moved into the `System` unit, to make this feature also available to Linux and to separate it completely from COM support.

The `TAggregatedObject` class keeps a reference to the controller, the external object, passed as parameter in the constructor. This *weak reference* is kept using a pointer type variable to avoid artificially increasing the reference count of the controller from the aggregated object, something that will prevent the object's reference count from reaching zero. You create an object of this type (used as internal object) passing the reference to the controller (the external object), and all of the `IInterface` methods are passed back to the controller. A similar class, `TContainedObject`, lets the controller resolve reference counting, but handles the `QueryInterface` call internally, limiting the type resolution only to interfaces supported by the internal object.

## Working with Exceptions

Another key feature of Object Pascal I'll cover in this chapter is the support for *exceptions*. The idea of exceptions is to make programs more robust by adding the capability of handling software or hardware errors in a uniform way. A program can survive such errors or terminate gracefully, allowing the user to save data before exiting. Exceptions allow you to separate the error-handling code from your normal code, instead of intertwining the two. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

Another benefit is that exceptions define a uniform and universal error-reporting mechanism, which is also used by Delphi components. At run time, Delphi raises exceptions when something goes wrong (in the run-time code, in a component, in the operating system). From the point of the code in which it is raised, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, Delphi handles it, by displaying a standard error message and trying to continue the program, by handing the next system message or user request.

The whole mechanism is based on four keywords:

**try** delimits the beginning of a protected block of code.

**except** delimits the end of a protected block of code and introduces the exception-handling statements, with this syntax form:

**on** *exception-type* **do** *statement*

**finally** is used to specify blocks of code that must always be executed, even when exceptions occur. This block is generally used to perform cleanup operations that should always be executed, such as closing files or database tables, freeing objects, and releasing memory and other resources acquired in the same program block.

**raise** is the statement used to generate an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also raise exceptions in your own code when it discovers invalid or inconsistent data at run time. The **raise** keyword can also be used inside a handler to *re-raise* an exception; that is, to propagate it to the next handler.

The most important element to notice up front is that exception handling is no substitute for **if** statements or for tests on input parameters of functions. So *in theory* we could write this code:

```
function DivideTwicePlusOne (A, B: Integer): Integer;  
begin  
  try  
    // error if B equals 0  
    Result := A div B;
```



```

    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
except
    on EDivByZero do
        Result := 0;
    end;
end;

```

*In practice*, however, this is certainly not a good way of writing your programs. The `except` block above, like most of the `except` blocks of the simple examples presented here, has almost no sense at all. In the code above, you should probably not handle the exception but let the program display the error message to the user. An algorithm calling this `DivideTwicePlusOne` function should not continue (with a meaningless zero value) when this internal error is encountered.

## Program Flow and the *finally* Block

But how do we stop the algorithm? The power of exceptions in Delphi relates to the fact that they are “passed” from a routine or method to the calling one, up to a global handler (if the program provides one, as Delphi applications generally do). So the real problem you might have is not how to stop an exception but how to execute some code when an exception is raised.

Consider this method (part of the `TryFinally` example from the CD), which performs some time-consuming operations and uses the hourglass cursor to show the user that it’s doing something:

```

procedure TForm1.BtnWrongClick(Sender: TObject);
var
    I, J: Integer;
begin
    Screen.Cursor := crHourglass;
    J := 0;
    // long (and wrong) computation...
    for I := 1000 downto 0 do
        J := J + J div I;
    MessageDlg ('Total: ' + IntToStr (J), mtInformation, [mbOK], 0);
    Screen.Cursor := crDefault;
end;

```

Because there is an error in the algorithm (as the variable `I` can reach a value of 0 and is also used in a division), the program will break, but it won’t reset the default cursor. This is what a `try/finally` block is for:

```

procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
    I, J: Integer;
begin

```

```
Screen.Cursor := crHourglass;
J := 0;
try
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;
  MessageDlg ('Total: ' + IntToStr (J), mtInformation, [mbOK], 0);
finally
  Screen.Cursor := crDefault;
end;
end;
```

When the program executes this function, it always resets the cursor, whether an exception (of any sort) occurs or not.

This code doesn't handle the exception; it merely makes the program robust in case an exception is raised. As a `try` block can be followed by either an `except` or a `finally` statement, but not both of them at the same time, the typical solution if you want to also handle the exception is to use two nested `try` blocks. In this case, you associate the internal one with a `finally` statement and the external one with an `except` statement, or vice versa as the situation requires. Here is the code of this third button of the `TryFinally` example:

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  try try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    MessageDlg ('Total: ' + IntToStr (J), mtInformation, [mbOK], 0);
  finally
    Screen.Cursor := crDefault;
  end;
  except
    on E: EDivByZero do
      begin
        // re-raise the exception with a new message
        raise Exception.Create ('Error in Algorithm');
      end;
    end;
  end;
end;
```

Every time you have some finalization code at the end of a method, you should place this code in a `finally` block. You should always, invariably, and continuously (how can I stress this more?) protect your code with `finally` statements, to avoid resource or memory leaks in case an exception is raised.

**TIP**

Handling the exception is generally much less important than using `finally` blocks, since Delphi can survive most of them. And too many exception-handling blocks in your code probably indicate errors in the program flow and possibly a misunderstanding of the role of exceptions in the language. In the examples in the rest of the book you'll see many `try/finally` blocks, a few `raise` statements, and almost no `try/except` blocks.

## Exception Classes

In exception-handling statements shown earlier, we caught the `EDivByZero` exception, which is defined by Delphi's RTL. Other such exceptions refer to run-time problems (such as a wrong dynamic cast), Windows resource problems (such as out-of-memory errors), or component errors (such as a wrong index). Programmers can also define their own exceptions; you can create a new subclass of the default exception class or one of its subclasses:

```
type
  EArrayFull = class (Exception);
```

When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

This `Create` method (inherited from the `Exception` class) has a string parameter to describe the exception to the user. You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

The code presented in the previous excerpts is part of a sample program, called `Exception1`. Some of the routines have actually been slightly modified, as in the following `DivideTwicePlusOne` function:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
      begin
        Result := 0;
        MessageDlg ('Divide by zero corrected.', mtError, [mbOK], 0);
      end;
    on E: Exception do
      begin
        Result := 0;
```

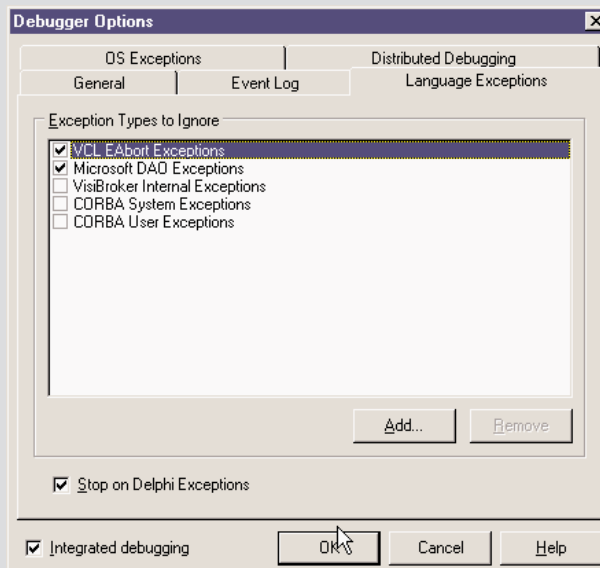
```
    MessageDlg (E.Message, mtError, [mbOK], 0);  
end;  
end; // end except  
end;
```

## Debugging and Exceptions

When you start a program from the Delphi environment (for example, by pressing the F9 key), you'll generally run it within the debugger. When an exception is encountered, the debugger will stop the program by default. This is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step-by-step. You can also use the Stack Trace feature of Delphi to see the sequence of function and method calls, which caused the program to raise an exception.

In the case of the Exception1 test program, however, this behavior will confuse the program's execution. In fact, even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step-by-step through the code, you can see how it is handled.

If you just want to let the program run when the exception is properly handled, run the program from Windows Explorer, or temporarily disable the Stop on Delphi Exceptions options in the Language Exceptions page of the Debugger Options dialog box (activated by the Tools > Debugger Options command), shown in the Language Exceptions page of the Debugger Options dialog box shown here.



In the `Exception1` code there are two different exception handlers after the same `try` block. You can have any number of these handlers, which are evaluated in sequence. For this reason, you need to place the broader handlers (the handlers of the ancestor `Exception` classes) at the end.

In fact, using a hierarchy of exceptions, a handler is also called for the subclasses of the type it refers to, as any procedure will do. This is polymorphism in action again. But keep in mind that using a handler for every exception, such as the one above, is not usually a good choice. It is better to leave unknown exceptions to Delphi. The default exception handler in the VCL displays the error message of the exception class in a message box, and then resumes normal operation of the program. You can actually modify the normal exception handler with the `Application.OnException` event, as demonstrated in the `ErrorLog` example later in this chapter.

Another important element of the code above is the use of the exception object in the handler (see on `E: Exception` do). The object `E` of class `Exception` receives the value of the exception object passed by the `raise` statement. When you work with exceptions, remember this rule: You raise an exception by creating an object and handle it by indicating its type. This has an important benefit, because as we have seen, when you handle a type of exception, you are really handling exceptions of the type you specify as well as any descendant type.

Delphi defines a hierarchy of exceptions, and you can choose to handle each specific type of exception in a different way or handle groups of them together.

## Logging Errors

Most of the time, you don't know which operation is going to raise an exception, and you cannot (and should not) wrap each and every piece of code in a `try/except` block. The general approach is to let Delphi handle all the exceptions and eventually pass them all to you, by handling the `OnException` event of the global `Application` object. This can be done rather easily with the `ApplicationEvents` component.

In the `ErrorLog` example, I've added to the main form a copy of the `ApplicationEvents` component and added a handler for its `OnException` event:

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);  
var  
    Filename: string;  
    LogFile: TextFile;  
begin  
    // prepares log file  
    Filename := ChangeFileExt (Application.Exename, '.log');  
    AssignFile (LogFile, Filename);  
    if FileExists (FileName) then  
        Append (LogFile) // open existing file  
    else  
        Rewrite (LogFile); // create a new one  
    // write to the file and show error
```

```
WriteLn (LogFile, DateTimeToStr (Now) + ':' + E.Message);  
if not CheckBoxSilent.Checked then  
    Application.ShowException (E);  
    // close the file  
    CloseFile (LogFile);  
end;
```

**NOTE**

The ErrorLog example uses the text file support provided by the traditional Turbo Pascal TextFile data type. You can assign a text file variable to an actual file and then read or write it. You can find more on TextFile operations in Chapter 12 of *Essential Pascal*, available on the companion CD.

In the global exceptions handler, you can write to the log, for example, the date and time of the event, and also decide whether to show the exception as Delphi usually does (executing the ShowException method of the TApplication class). In fact, Delphi by default executes ShowException only if there is no OnException handler installed.

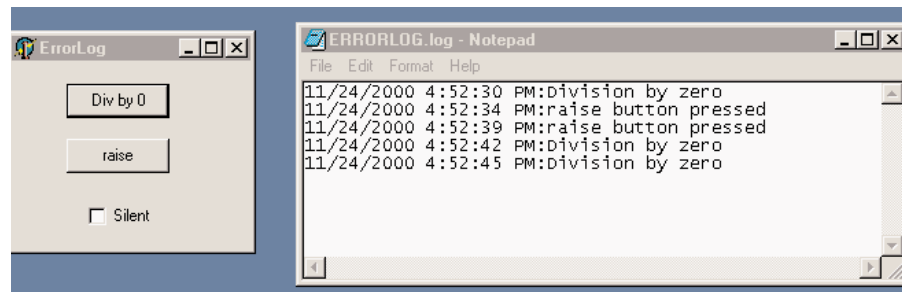
Finally, remember to close the file, flushing the buffers, every time the exception is handled or when the program terminates. I've chosen the first approach to avoid keeping the log file open for the lifetime of the application, potentially making it difficult to work on it. You can accomplish this in the OnDestroy event handler of the form:

```
procedure TFormLog.FormDestroy(Sender: TObject);  
begin  
    CloseFile (LogFile);  
end;
```

The form of the program includes a check box to determine its behavior and two buttons generating exceptions. In Figure 3.5, you can see the ErrorLog program running and a sample exceptions log open in Notepad.

**FIGURE 3.5:**

The ErrorLog example and the log it produces



## Class References

The final language feature I want to discuss in this chapter is *class references*, which implies the idea of manipulating classes themselves (not just class instances) within your code. The first point to keep in mind is that a class reference isn't a class, it isn't an object, and it isn't a reference to an object; it is simply a reference to a class type.

A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code might make this a little clearer. Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. Actually, there are not many such expressions, but the few cases are interesting. The simplest case is the creation of an object. We can rewrite the two lines above as follows:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

This time I've applied the `Create` constructor to the class reference instead of to an actual class; I've used a class reference to create an object of that class.

### NOTE

Class references remind us of the concept of *metaclass* available in other OOP languages. In Object Pascal, however, a class reference is not itself a class but only a type pointer. Therefore, the analogy with metaclasses (classes describing other classes) is a little misleading. Actually, `TMetaclass` is also the term used in Borland C++Builder.

Class reference types wouldn't be as useful if they didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef` above, you can then assign to it that specific class and any subclass. So if `MyNewClass` is a subclass of my class, you can also write

```
AClassRef := MyNewClass;
```

Delphi declares a lot of class references in the run-time library and the VCL, including the following:

```
TClass = class of TObject;  
ExceptClass = class of Exception;  
TComponentClass = class of TComponent;  
TControlClass = class of TControl;  
TFormClass = class of TForm;
```

In particular, the TClass class reference type can be used to store a reference to any class you write in Delphi, because every class is ultimately derived from TObject. The TFormClass reference, instead, is used in the source code of most Delphi projects. The CreateForm method of the Application object, in fact, requires as parameter the class of the form to create:

```
Application.CreateForm(TForm1, Form1);
```

The first parameter is a class reference; the second is a variable that stores a reference to the created object instance.

Finally, when you have a class reference you can apply to it the class methods of the related class. Considering that each class inherits from TObject, you can apply to each class reference some of the methods of TObject, as we'll see in the next chapter.

## Creating Components Using Class References

What is the *practical* use of class references in Delphi? Being able to manipulate a data type at run time is a fundamental element of the Delphi environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what Delphi does for you behind the scenes.) In other words, class references give you polymorphism for object construction.

To give you a better idea of how class references work, I've built an example named ClassRef. The form displayed by this example is quite simple. It has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, push buttons, and edit boxes.

To make this program run properly, you need to change the names of the three components. The form must also have a class reference field:

```
private  
  ClassRef: TControlClass;  
  Counter: Integer;
```

The first field stores a new data type every time the user clicks one of the three radio buttons. Here is one of the three methods:

```
procedure TForm1.RadioButtonRadioClick(Sender: TObject);  
begin  
  ClassRef := TRadioButton;  
end;
```



The other two radio buttons have `OnClick` event handlers similar to this one, assigning the value `TEdit` or `TButton` to the `ClassRef` field. A similar assignment is also present in the handler of the `OnCreate` event of the form, used as an initialization method.

The interesting part of the code is executed when the user clicks the form. Again, I've chosen the `OnMouseDown` event of the form to hold the position of the mouse click:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
var
    NewCtrl: TControl;
    MyName: String;
begin
    // create the control
    NewCtrl := ClassRef.Create (Self);
    // hide it temporarily, to avoid flickering
    NewCtrl.Visible := False;
    // set parent and position
    NewCtrl.Parent := Self;
    NewCtrl.Left := X;
    NewCtrl.Top := Y;
    // compute the unique name (and caption)
    Inc (Counter);
    MyName := ClassRef.ClassName + IntToStr (Counter);
    Delete (MyName, 1, 1);
    NewCtrl.Name := MyName;
    // now show it
    NewCtrl.Visible := True;
end;
```

The first line of the code for this method is the key. It creates a new object of the class data type stored in the `ClassRef` field. We accomplish this simply by applying the `Create` constructor to the class reference. Now you can set the value of the `Parent` property, set the position of the new component, give it a name (which is automatically used also as `Caption` or `Text`), and make it visible.

Notice in particular the code used to build the name; to mimic Delphi's default naming convention, I've taken the name of the class with the expression `ClassRef.ClassName`, using a class method of the `TObject` class. Then I've added a number at the end of the name and removed the initial letter of the string. For the first radio button, the basic string is `TRadioButton`, plus the *1* at the end, and minus the *T* at the beginning of the class name—`RadioButton1`. Sound familiar?

You can see an example of the output of this program in Figure 3.6. Notice that the naming is not exactly the same as used by Delphi. Delphi uses a separate counter for each type of

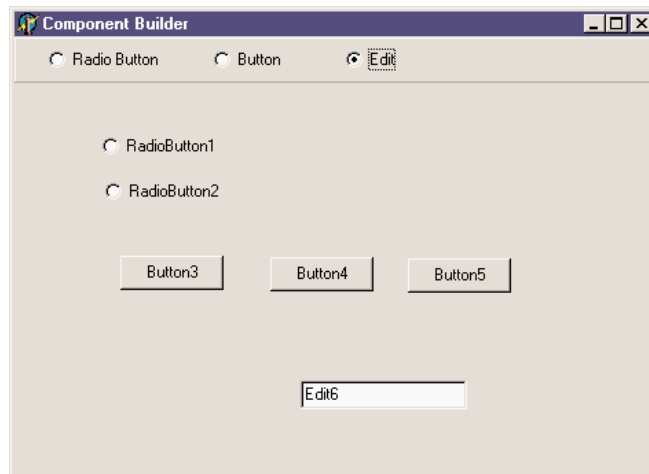
control; I've used a single counter for all of the components. If you place a radio button, a push button, and an edit box in a form of the ClassRef example, their names will be RadioButton1, Button2, and Edit3.

**NOTE**

For polymorphic construction to work, the base class type of the class reference must have a virtual constructor. If you use a virtual constructor (as in the example), the constructor call applied to the class reference will call the constructor of the type that the class reference variable *currently refers to*. But without a virtual constructor, your code will call the constructor of *fixed class type* indicated in the class reference declaration. Virtual constructors are required for polymorphic construction in the same way that virtual methods are required for polymorphism.

**FIGURE 3.6:**

An example of the output of the ClassRef example



## What's Next?

In this chapter, we have discussed the more advanced elements of object-oriented programming in Object Pascal. We have considered inheritance, virtual and abstract methods, polymorphism, safe typecasting, interfaces, exceptions, and class references.

Understanding the secrets of Object Pascal and the structure of the Delphi library is vital for becoming an expert Delphi programmer. These topics form the foundation of working with the VCL and CLX class libraries; after exploring them in the next two chapters, we'll *finally* go on in Part II of the book to explore the development of real applications using all the various components provided by Delphi.

In the meantime, the next chapter will give you an over view of the Delphi run-time library, mainly a collection of functions with little OOP involved. The RTL is an assorted collection of routines and tasks for performing basic tasks with Delphi, and it has been largely extended in Delphi 6.

Chapter 5 will give you more information about the Object Pascal language, discussing features related to the structure of the Delphi class library, such as the effect of the `published` keyword and the role of events. The chapter, as a whole, will discuss the overall architecture of the component library.

# The Run-Time Library

- Overview of the RTL
- New Delphi 6 RTL functions
- The conversion engine
- Dates, strings, and other new RTL units
- The *TObject* class
- Showing class information at run time

**D**elphi uses Object Pascal as its programming language and favors an object-oriented approach, tied with a visual development style. This is where Delphi shines, and we will cover component-based and visual development in this book; however, I want to underline the fact that a lot of ready-to-use features of Delphi come from its run-time library, or RTL for short. This is a large collection of functions you can use to perform simple tasks, as well as some complex ones, within your Pascal code. (I use “Pascal” here, because the run-time library mainly contains procedures and functions and not classes and objects.)

There is actually a second reason to devote this chapter of the book to the run-time library: Delphi 6 sees a large number of enhancements to this area. There are new groups of functions, functions have been moved to new units, and other elements have changed, creating a few incompatibilities with existing code. So even if you’ve used past versions of Delphi and feel confident with the RTL, you should still read at least portions of this chapter.

## The Units of the RTL

As I mentioned above, in Delphi 6 the RTL (run-time library) has a new structure and several new units. The reason for adding new units is that many new functions were added. In most cases, you’ll find the existing functions in the units where they used to be, but the new functions will appear in specific units. For example, new functions related to dates are now in the DateUtils unit, but existing date functions have not been moved away from SysUtils in order to avoid incompatibilities with existing code.

The exception to this rule relates to some of the variant support functions, which were moved out of the System unit to avoid unwanted linkage of specific Windows libraries, even in programs that didn’t use those features. These variant functions are now part of the new Variants unit, described later in the chapter.

**WARNING** Some of your existing Delphi code might need to use this new Variants unit to recompile. Delphi 6 is smart enough to acknowledge this and auto-include the Variants unit in projects that use the Variant type, issuing only a warning.

A little bit of fine-tuning has also been applied to reduce the minimum size of an executable file, at times enlarged by the unwanted inclusion of global variables or initialization code.

### Executable Size under the Microscope

While touching up the RTL, Borland engineers have been able to trim a little “fat” out of each and every Delphi application. Reducing the minimum program size of a few KB seems quite odd, with all the bloated applications you find around these days, but it is a good service to developers. There are cases in which even few KB (multiplied by many applications) can reduce size and eventually download time.

*Continued on next page*

As a simple test, I've built the MiniSize program, which is not an attempt to build the smallest possible program, but rather an attempt to build a very small program that does something interesting: It reports the size of its own executable file. All of the code of this example is in the source code on the companion CD:

```
program MiniSize;  
  
uses  
  Windows;  
  
{ $R *.RES }  
  
var  
  nSize: Integer;  
  hFile: THandle;  
  strSize: String;  
  
begin  
  // open the current file and read the size  
  hFile := CreateFile (PChar (ParamStr (0)),  
    0, FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);  
  nSize := GetFileSize (hFile, nil);  
  CloseHandle (hFile);  
  
  // copy the size to a string and show it  
  SetLength (strSize, 20);  
  Str (nSize, strSize);  
  MessageBox (0, PChar (strSize),  
    'Mini Program', MB_OK);  
end.
```

The program opens its own executable file, after retrieving its name from the first command-line parameter (`ParamStr (0)`), extracts the size, converts it into a string using the simple `Str` function, and shows the result in a message. The program does not have top-level windows. Moreover, I use the `Str` function for the integer-to-string conversion to avoid including `SysUtils`, which defines all the more complex formatting routines and would impose a little extra overhead.

If you compile this program with Delphi 5, you obtain an executable size of 18,432 bytes. Delphi 6 reduces this size to only 15,360 bytes, trimming about 3 KB. Replacing the long string with a short string, and modifying the code a little, you can trim down the program further, up to 9,216 bytes. This is because you'll end up removing the string support routines and also the memory allocator, something possible only in programs using exclusively low-level calls. You can find both versions in the source code of the example.

*Continued on next page*

Notice, anyway, that decisions of this type always imply a few trade-offs. In eliminating the overhead of variants from Delphi applications that don't use them, for example, Borland added a little extra burden to applications that do. The real advantage of this operation, though, is in the reduced memory footprint of Delphi applications that do not use variants, as a result of not having to bring in several megabytes of the Ole2 system libraries.

What is really important, in my opinion, is the size of full-blown Delphi applications based on run-time packages. A simple test with a do-nothing program, the `MiniPack` example, shows an executable size of 15,972 bytes.

In the following sections is a list of the RTL units in Delphi 6, including all the units available (with the complete source code) in the `Source\Rt1\Sys` subfolder of the `Delphi` directory and some of those available in the new subfolder `Source\Rt1\Common`. This new directory hosts the source code of units that make up the new RTL package, which comprises both the function-based library and the core classes, discussed in the next chapter.

#### NOTE

The `VCL50` package has now been split into the `VCL` and `RTL` packages, so that nonvisual applications using run-time packages don't have the overhead of also deploying visual portions of the `VCL`. Also, this change helps with Linux compatibility, as the new package is shared between the `VCL` and `CLX` libraries. Notice also that the package names in Delphi 6 don't have the version number in their name anymore. When they are compiled, though, the `BPL` does have the version in its file name, as discussed in more detail in Chapter 12.

I'll give a short overview of the role of each unit and an overview of the groups of functions included. I'll also devote more space to the new Delphi 6 units. I won't provide a detailed list of the functions included, because the online help includes similar reference material. However, I've tried to pick a few interesting or little-known functions, and I will discuss them shortly.

## The System and SysInit Units

`System` is the core unit of the RTL and is automatically included in any compilation (considering an automatic and implicit uses statement referring to it). Actually, if you try adding the unit to the uses statement of a program, you'll get the compile-time error:

```
[Error] Identifier redeclared: System
```

The `System` unit includes, among other things:

- The `TObject` class, the base class of any class defined in the Object Pascal language, including all the classes of the `VCL`. (This class is discussed later in this chapter.)

- The `IUnknown` and `IDispatch` interfaces as well as the simple implementation class `TInterfacedObject`. There are also the new `IInterface` and `IInvokable` interfaces. `IInterface` was added to underscore the point that the interface type in Delphi's Object Pascal language definition is in no way dependent on the Windows operating system (and never has been). `IInvokable` was added to support SOAP-based invocation. (Interfaces and related classes were introduced in the last chapter and will be discussed further in multiple sections of the book.)
- Some variant support code, including the variant type constants, the `TVarData` record type and the new `TVariantManager` type, a large number of variant conversion routines, and also variant records and dynamic arrays support. This area sees a lot of changes compared to Delphi 5. The basic information on variants is provided in Chapter 10 of *Essential Pascal* (available on the companion CD), while an introduction to custom variants is available later in this chapter.
- Many base data types, including pointer and array types and the `TDateTime` type I've already described in the last chapter.
- Memory allocation routines, such as `GetMem` and `FreeMem`, and the actual memory manager, defined by the `TMemoryManager` record and accessed by the `GetMemoryManager` and `SetMemoryManager` functions. For information, the `GetHeapStatus` function returns a `THeapStatus` data structure. Two new global variables (`AllocMemCount` and `AllocMemSize`) hold the number and total size of allocated memory blocks. There is more on memory and the use of these functions in Chapter 10.
- Package and module support code, including the `PackageInfo` pointer type, the `GetPackageInfoTable` global function, and the `EnumModules` procedure (packages internals are discussed in Chapter 12).
- A rather long list of global variables, including the Windows application instance `MainInstance`; `IsLibrary`, indicating whether the executable file is a library or a stand-alone program; `IsConsole`, indicating console applications; `IsMultiThread`, indicating whether there are secondary threads; and the command-line string `CmdLine`. (The unit includes also the `ParamCount` and `ParamStr` for an easy access to command-line parameters.) Some of these variables are specific to the Windows platform.
- Thread-support code, with the `BeginThread` and `EndThread` functions; file support records and file-related routines; wide string and OLE string conversion routines; and many other low-level and system routines (including a number of automatic conversion functions).

The companion unit of `System`, called `SysInit`, includes the system initialization code, with functions you'll seldom use directly. This is another unit that is always implicitly included, as it is used by the `System` unit.



## New in System Unit

I've already described some interesting new features of the System unit in the list above, and most of the changes relate to making the core Delphi RTL more cross-platform portable, replacing Windows-specific features with generic implementations. Along this line, there are new names for interface types, totally revised support for variants, new pointer types, dynamic array support, and functions to customize the management of exception objects.

Another addition for compatibility with Linux relates to line breaks in text files. There is a new `DefaultTextLineBreakStyle` variable, which can be set to either `tlbsLF` or `tlbsCRLF`, and a new `sLineBreak` string constant, which has the value `#13#10` in the Windows version of Delphi and the value `#10` in the Linux version. The line break style can also be set on a file-by-file basis with `SetTextLineBreakStyle` function.

Finally, the System unit now includes the `TFileRec` and `TTextRec` structures, which were defined in the `SysUtils` unit in earlier versions of Delphi.

## The SysUtils and SysConst Units

The `SysConst` unit defines a few constant strings used by the other RTL units for displaying messages. These strings are declared with the `resourcestring` keyword and saved in the program resources. As other resources, they can be translated by means of the Integrated Translation Manager or the External Translation Manager.

The `SysUtils` unit is a collection of system utility functions of various types. Different from other RTL units, it is in large part an operating system-dependent unit. The `SysUtils` unit has no specific focus, but it includes a bit of everything, from string management to locale and multibyte-characters support, from the `Exception` class and several other derived exception classes to a plethora of string-formatting constants and routines.

Some of the features of `SysUtils` are used every day by every programmer as the `IntToStr` or `Format` string-formatting functions; other features are lesser known, as they are the Windows version information global variables. These indicate the Windows platform (Window 9x or NT/2000), the operating system version and build number, and the eventual service pack installed on NT. They can be used as in the following code, extracted from the `WinVersion` example on the companion CD:

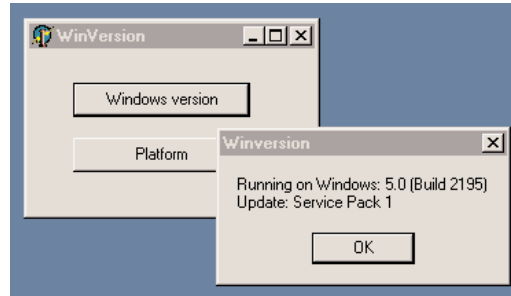
```
case Win32Platform of
  VER_PLATFORM_WIN32_WINDOWS: ShowMessage ('Windows 9x');
  VER_PLATFORM_WIN32_NT:      ShowMessage ('Windows NT');
end;

ShowMessage ('Running on Windows: ' + IntToStr (Win32MajorVersion) + '.' +
  IntToStr (Win32MinorVersion) + ' (Build ' + IntToStr (Win32BuildNumber) +
  ') ' + #10#13 + 'Update: ' + Win32CSDVersion);
```

The second code fragment produces a message like the one in Figure 4.1, depending, of course, on the operating-system version you have installed.

**FIGURE 4.1:**

The version information displayed by the WinVersion example



Another little-known feature, but one with a rather long name, is a class that supports multithreading: `TMultiReadExclusiveWriteSynchronizer`. This class allows you to work with resources that can be used by multiple threads at the same time for reading (multiread) but must be used by one single thread when writing (exclusive-write). This means that the writing cannot start until all the reading threads have terminated.

#### NOTE

The multi-read synchronizer is unique in that it supports recursive locks and promotion of read locks to write locks. The main purpose of the class is to allow multiple threads easy, fast access to read from a shared resource, but still allow one thread to gain exclusive control of the resource for relatively infrequent updates. There are other synchronization classes in Delphi, declared in the `SyncObjs` unit and closely mapped to operating-system synchronization objects (such as events and critical sections in Windows).

## New SysUtils Functions

Delphi 6 has some new functions within the `SysUtils` unit. One of the new areas relates to Boolean to string conversion. The `BoolToStr` function generally returns '-1' and '0' for true and false values. If the second optional parameter is specified, the function returns the first string in the `TrueBools` and `FalseBools` arrays (by default 'TRUE' and 'FALSE'):

```
BoolToStr (True) // returns '-1'  
BoolToStr (False, True) // returns 'FALSE' by default
```

The reverse function is `StrToBool`, which can convert a string containing either one of the values of two Boolean arrays mentioned above or a numeric value. In the latter case, the result will be true unless the numeric value is zero. You can see a simple demo of the use of the Boolean conversion functions in the `StrDemo` example, later in this chapter.

Other new functions of `SysUtils` relate to floating-point conversions to currency and date time types: `FloatToCurr` and `FloatToDateTime` can be used to avoid an explicit type cast. The `TryStrToFloat` and `TryStrToCurr` functions try to convert a string into a floating point or currency value and will return `False` in case of error instead of generating an exception (as the classic `StrToFloat` and `StrToCurr` functions do).

There is an `AnsiDequotedStr` function, which removes quotes from a string, matching the `AnsiQuoteStr` function added in Delphi 5. Speaking of strings, Delphi 6 has much-improved support for wide strings, with a series of new routines, including `WideUpperCase`, `WideLowerCase`, `WideCompareStr`, `WideSameStr`, `WideCompareText`, `WideSameText`, and `WideFormat`. All of these functions work like their `AnsiString` counterparts.

Three functions (`TryStrToDate`, `TryEncodeDate`, and `TryEncodeTime`) try to convert a string to a date or to encode a date or time, without raising an exception, similarly to the `Try` functions previously mentioned. In addition, the `DecodeDateFully` function returns more detailed information, such as the day of the week, and the `CurrentYear` function returns the year of today's date.

There is a portable, friendly, overloaded version of the `GetEnvironmentVariable` function. This new version uses string parameters instead of `PChar` parameters and is definitely easier to use:

```
function GetEnvironmentVariable(Name: string): string;
```

Other new functions relate to interface support. Two new overloaded versions of the little-known `Support` function allow you to check whether an object or a class supports a given interface. The function corresponds to the behavior of the `is` operator for classes and is mapped to the `QueryInterface` method. Here's an example in the code of the `IntfDemo` program from Chapter 3:

```
var  
  W1: IWalker;  
  J1: IJumper;  
begin  
  W1 := TAthlete.Create;  
  // more code...  
  if Supports (w1, IJumper) then  
  begin  
    J1 := W1 as IJumper;  
    Log (J1.Walk);  
  end;
```

There are also an `IsEqualGUID` function and two functions for converting strings to GUIDs and vice versa. The function `CreateGUID` has been moved to `SysUtils`, as well, to make it also available on Linux (with a custom implementation, of course).

Finally, Delphi 6 has some more Linux-compatibility functions. The `AdjustLineBreaks` function can now do different types of *adjustments* to carriage-return and line-feed sequences, along with the introduction of new global variables for text files in the `System` unit, as described earlier. The `FileCreate` function has an overloaded version in which you can specify file-access rights *the Unix way*. The `ExpandFileName` function can locate files (on case-sensitive file systems) even when their cases don't exactly correspond. The functions related to path delimiters (backslash or slash) have been made more generic and renamed accordingly. (For example, the `IncludeTrailingBackslash` function is now better known as `IncludingTrailingPathDelimiter`.)

## The Math Unit

The `Math` unit hosts a collection of mathematical functions: about forty trigonometric functions, logarithmic and exponential functions, rounding functions, polynomial evaluations, almost thirty statistical functions, and a dozen financial functions.

Describing all of the functions of this unit would be rather tedious, although some readers are probably very interested in the mathematical capabilities of Delphi. Here are some of the newer math functions.

### New Math Functions

Delphi 6 adds to the `Math` unit quite a number of new features. There is support for infinite constants (`Infinity` and `NegInfinity`) and related comparison functions (`IsInfinite` and `IsNan`). There are new trigonometric functions for cosecants and cotangents and new angle-conversion functions.

A handy feature is the availability of an overloaded `IfThen` function, which returns one of two possible values depending on a Boolean expression. (A similar function is now available also for strings.) You can use it, for example, to compute the minimum of two values:

```
nMin := IfThen (nA < nB, na, nB);
```

#### NOTE

The `IfThen` function is similar to the `?:` operator of the C/C++ language, which I find very handy because you can replace a complete `if/then/else` statement with a much shorter expression, writing less code and often declaring fewer temporary variables.

The `RandomRange` and `RandomFrom` can be used instead of the traditional `Random` function to have more control on the random values produced by the RTL. The first function returns a number within two extremes you specify, while the second selects a random value from an array of possible numbers you pass to it as a parameter.

The `InRange` Boolean function can be used to check whether a number is within two other values. The `EnsureRange` function, instead, forces the value to be within the specified range.

The return value is the number itself or the lower limit or upper limit, in the event the number is out of range. Here is an example:

```
// do something only if value is within min and max
if InRange (value, min, max) then
    ...

// make sure the value is between min and max
value := EnsureRange (value, min, max);
...
```

Another set of very useful functions relates to comparisons. Floating-point numbers are fundamentally inexact; a floating-point number is an approximation of a theoretical real value. When you do mathematical operations on floating-point numbers, the inexactness of the original values accumulates in the results. Multiplying and dividing by the same number might not return exactly the original number but one that is very close to it. The `SameValue` function allows you to check whether two values are close enough in value to be considered equal. You can specify how close the two numbers should be or let Delphi compute a reasonable error range for the representation you are using. (This is why the function is overloaded.) Similarly, the `IsZero` function compares a number to zero, with the same “fuzzy logic.”

The `CompareValue` function uses the same rule for floating-point numbers but is available also for integers; it returns one of the three constants `LessThanValue`, `EqualsValue`, and `GreaterThanValue` (corresponding to  $-1$ ,  $0$ , and  $1$ ). Similarly, the new `Sign` function returns  $-1$ ,  $0$ , and  $1$  to indicate a negative value, zero, or a positive value.

The `DivMod` function is equivalent to both `div` and `mod` operations, returning the result of the integer division and the remainder (or modulus) at once. The `RoundTo` function allows you to specify the rounding digit—allowing, for example, rounding to the nearest thousand or to two decimals:

```
RoundTo (123827, 3); // result is 124,000
RoundTo (12.3827, -2); // result is 12.38
```

---

**WARNING**

Notice that the `RoundTo` function uses a positive number to indicate the power of ten to round to (for example,  $2$  for hundreds) or a negative number for the number of decimal places. This is exactly the opposite of the `Round` function used by spreadsheets such as Excel.

There are also some changes to the standard rounding operations provided by the `Round` function: You can now control how the FPU (the floating-point unit of the CPU) does the rounding by calling the `SetRoundMode` function. There are also functions to control the FPU precision mode and its exceptions.

## The New ConvUtils and StdConvs Units

The new ConvUtils unit contains the core of the conversion engine. It uses the conversion constants defined by a second unit, StdConvs. I'll cover these two units later in this chapter, showing you also how to extend them with new measurement units.

## The New DateUtils Unit

The DateUtils unit is a new collection of date and time-related functions. It includes new functions for picking values from a TDateTime variable or counting values from a given interval, such as

```
// pick value
function DayOf(const AValue: TDateTime): Word;
function HourOf(const AValue: TDateTime): Word;
// value in range
function WeekOfYear(const AValue: TDateTime): Integer;
function HourOfWeek(const AValue: TDateTime): Integer;
function SecondOfHour(const AValue: TDateTime): Integer;
```

Some of these functions are actually quite odd, such as MilliSecondOfMonth or SecondOfWeek, but Borland developers have decided to provide a complete set of functions, no matter how impractical they sound. I actually used some of these functions in Chapter 2, to build the TDate class.

There are functions for computing the initial or final value of a given time interval (day, week, month, year) including the current date, and for range checking and querying; for example:

```
function DaysBetween(const ANow, AThen: TDateTime): Integer;
function WithinPastDays(const ANow, AThen: TDateTime;
    const ADays: Integer): Boolean;
```

Other functions cover incrementing and decrementing by each of the possible time intervals, encoding and “recoding” (replacing one element of the TDateTime value, such as the day, with a new one), and doing “fuzzy” comparisons (approximate comparisons where a difference of a millisecond will still make two dates equal). Overall, DateUtils is quite interesting and not terribly difficult to use.

## The New StrUtils Unit

The StrUtils unit is a new unit with some new string-related functions. One of the key features of this unit is the availability of many new string comparison functions. There are functions based on a “soundex” algorithm (AnsiResembleText), some providing lookup in arrays of strings (AnsiMatchText and AnsiIndexText), sub-string location, and replacement (including AnsiContainsText and AnsiReplaceText).

**NOTE**

*Soundex* is an algorithm to compare names based on how they sound rather than how they are spelled. The algorithm computes a number for each word sound, so that comparing two such numbers you can determine whether two names sound similar. The system was first applied 1880 by the U.S. Bureau of the Census, patented in 1918, and is now in the public domain. The soundex code is an indexing system that translates names into a four-character code consisting of one letter and three numbers. More information is at [www.nara.gov/genealogy/coding.html](http://www.nara.gov/genealogy/coding.html).

Beside comparisons, other functions provide a two-way test (the nice `IfThen` function, similar to the one we've already seen for numbers), duplicate and reverse strings, and replace sub-strings. Most of these string functions were added as a convenience to Visual Basic programmers migrating to Delphi.

I've used some of these functions in the `StrDemo` example on the companion CD, which uses also some of the new Boolean-to-string conversions defined within the `SysUtils` unit. The program is actually a little more than a test for a few of these functions. For example, it uses the "soundex" comparison between the strings entered in two edit boxes, converting the resulting Boolean into a string and showing it:

```
ShowMessage (BoolToStr (AnsiResemblesText  
  (EditResemble1.Text, EditResemble2.Text), True));
```

The program also showcases the `AnsiMatchText` and `AnsiIndexText` functions, after filling a dynamic array of strings (called `strArray`) with the values of the strings inside a list box. I could have used the simpler `IndexOf` method of the `TStrings` class, but this would have defeated the purpose of the example. The two list comparisons are done as follows:

```
procedure TForm1.ButtonMatchesClick(Sender: TObject);  
begin  
  ShowMessage (BoolToStr (AnsiMatchText(EditMatch.Text, strArray), True));  
end;  
  
procedure TForm1.ButtonIndexClick(Sender: TObject);  
var  
  nMatch: Integer;  
begin  
  nMatch := AnsiIndexText(EditMatch.Text, strArray);  
  ShowMessage (IfThen (nMatch >= 0, 'Matches the string number ' +  
    IntToStr (nMatch), 'No match'));  
end;
```

Notice the use of the `IfThen` function in the last few lines of code, with two alternative output strings, depending on the result of the initial test (`nMatch <= 0`).

Three more buttons do simple calls to three other new functions, with the following lines of code (one for each):

```
// duplicate (3 times) a string
ShowMessage (DupeString (EditSample.Text, 3));
// reverse the string
ShowMessage (ReverseString (EditSample.Text));
// choose a random string
ShowMessage (RandomFrom (strArray));
```

## The New Types Unit

The Types unit is a new Pascal file holding data types common to multiple operating systems. In past versions of Delphi, the same types were defined by the Windows unit; now they've been moved to this common unit, shared by Delphi and Kylix. The types defined here are simple ones and include, among others, the TPoint, TRect, and TSmallPoint record structures plus their related pointer types.

## The New Variants and VarUtils Units

Variants and VarUtils are two new variant-related units. The Variants unit contains generic code for variants. As mentioned earlier, some of the routines in this unit have been moved here from the System unit. Functions include generic variant support, variant arrays, variant copying, and dynamic array to variant array conversions. There is also the TCustomVariantType class, which defines customizable variant data types.

The Variants unit is totally platform independent and uses the VarUtils unit, which contains OS-dependent code. In Delphi, this unit uses the system APIs to manipulate variant data, while in Kylix it uses some custom code provided by the RTL library.

## Custom Variants and Complex Numbers

The possibility to extend the type system with custom variants is brand new in Delphi 6. It allows you to define a new data type that, contrary to a class, overloads standard arithmetic operators.

In fact, a variant is a type holding both type specification and the actual value. A variant can contain a string, another can contain a number. The system defines automatic conversions among variant types, allowing you to mix them inside operations (including custom variants). This flexibility comes at a high cost: operations on variants are much slower than on native types, and variants use extra memory.

As an example of a custom variant type, Delphi 6 ships with an interesting definition for complex numbers, found in the VarCmplx unit (available in source-code format in the



Rt1\Common folder). You can create complex variants by using one of the overloaded `VarComplexCreate` functions and use them in any expression, as the following code fragment demonstrates:

```
var
  v1, v2: Variant;
begin
  v1 := VarComplexCreate (10, 12);
  v2 := VarComplexCreate (10, 1);
  ShowMessage (v1 + v2 + 5);
```

The complex numbers are actually defined using classes, but they are surfaced as variants by inheriting a new class from the `TCustomVariantType` class (defined in the `Variants` unit), overriding a few virtual abstract functions, and creating a global object that takes care of the registration within the system.

Beside these internal definitions, the unit includes a long list of routines for operating on variant, including mathematical and trigonometric operations. I'll leave them to your study, as not all readers may be interested in complex numbers for their programs.

---

**WARNING** Building a custom variant is certainly not an easy task, and I can hardly find reasons for using them instead of objects and classes. In fact, with a custom variant you gain the advantage of using operator overloading on your own data structures, but you lose compile-time checking, make the code much slower, miss several OOP features, and have to write a lot of rather complex code.

## The DelphiMM and ShareMem Units

The `DelphiMM` and `ShareMem` units relate to memory management. The actual Delphi memory manager is declared in the `System` unit. The `DelphiMM` unit defines an alternative memory manager library to be used when passing strings from an executable to a DLL (a Windows dynamic linking library), both built with Delphi.

The interface to this memory manager is defined in the `ShareMem` unit. This is the unit you must include (compulsory as first unit) in the projects of both your executable and library (or libraries). Then, you'll also need to distribute and install the `Bor1ndmm.dll` library file along with your program.

## COM-Related Units

`ComConts`, `ComObj`, and `ComServ` provide low-level COM support. As these units are not really part of the RTL, from my point of view, I won't discuss them here in any detail. You can refer to Chapter 20 for all the related information. In any case, these units have not changed a lot since the last version of Delphi.

## Converting Data

Delphi 6 includes a new conversion engine, defined in the `ConvUtils` unit. The engine by itself doesn't include any definition of actual measurement units; instead, it has a series of core functions for end users.

The key function is the actual conversion call, the `Convert` function. You simply provide the amount, the units it is expressed in, and the units you want it converted into. The following would convert a temperature of 31 degrees Celsius to Fahrenheit:

```
Convert (31, tuCelsius, tuFahrenheit)
```

An overloaded version of the `Convert` function allows converting values that have two units, such as speed (which has both a length and a time unit). For example, you can convert miles per hours to meters per second with this call:

```
Convert (20, duMiles, tuHours, duMeters, tuSeconds)
```

Other functions in the unit allow you to convert the result of an addition or a difference, check if conversions are applicable, and even list the available conversion families and units.

A predefined set of measurement units is provided in the `StdConvs` unit. This unit has conversion families and an impressive number of actual values, as in the following reduced excerpt:

```
// Distance Conversion Units
// basic unit of measurement is meters
cbDistance: TConvFamily;

duAngstroms: TConvType;
duMicrons: TConvType;
duMillimeters: TConvType;
duMeters: TConvType;
duKilometers: TConvType;
duInches: TConvType;
duMiles: TConvType;
duLightYears: TConvType;
duFurlongs: TConvType;
duHands: TConvType;
duPicas: TConvType;
```

This family and the various units are registered in the conversion engine in the initialization section of the unit, providing the conversion ratios (saved in a series of constants, as `MetersPerInch` in the code below):

```
cbDistance := RegisterConversionFamily('Distance');
duAngstroms := RegisterConversionType(cbDistance, 'Angstroms', 1E-10);
duMillimeters := RegisterConversionType(cbDistance, 'Millimeters', 0.001);
duInches := RegisterConversionType(cbDistance, 'Inches', MetersPerInch);
```

To test the conversion engine, I built a generic example (ConvDemo on the companion CD) that allows you to work with the entire set of available conversions. The program fills a combo box with the available conversion families and a list box with the available units of the active family. This is the code:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    i: Integer;  
begin  
    GetConvFamilies (aFamilies);  
    for i := Low(aFamilies) to High(aFamilies) do  
        ComboFamilies.Items.Add (ConvFamilyToDescription (aFamilies[i]));  
    // get the first and fire event  
    ComboFamilies.ItemIndex := 0;  
    ChangeFamily (self);  
end;  
  
procedure TForm1.ChangeFamily(Sender: TObject);  
var  
    aTypes: TConvTypeArray;  
    i: Integer;  
begin  
    ListTypes.Clear;  
    CurrFamily := aFamilies [ComboFamilies.ItemIndex];  
    GetConvTypes (CurrFamily, aTypes);  
    for i := Low(aTypes) to High(aTypes) do  
        ListTypes.Items.Add (ConvTypeToDescription (aTypes[i]));  
end;
```

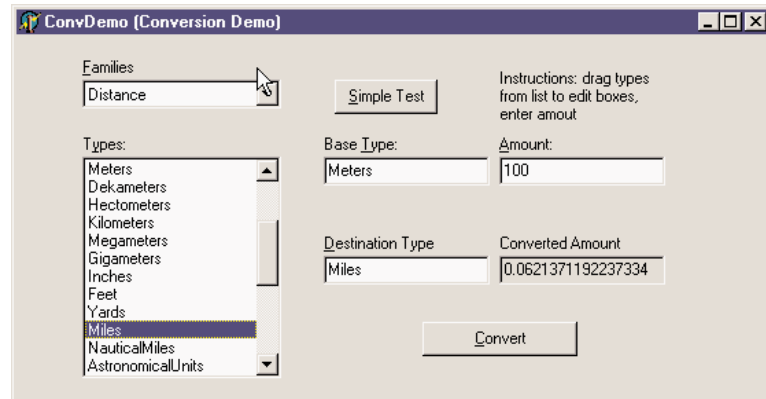
The aFamilies and CurrFamily variables are declared in the private section of the form as follows:

```
aFamilies: TConvFamilyArray;  
CurrFamily: TConvFamily;
```

At this point, a user can enter two measurement units and an amount in the corresponding edit boxes of the form, as you can see in Figure 4.2. To make the operation faster, it is actually possible to select a value in the list and drag it to one of the two Type edit boxes. The dragging support is described in the sidebar “Simple Dragging in Delphi.”

**FIGURE 4.2:**

The ConvDemo example  
at run time



## Simple Dragging in Delphi

The ConvDemo example I've built to show how to use the new conversion engine of Delphi 6 uses an interesting technique: dragging. In fact, you can move the mouse over the list box, select an item, and then keep the left mouse button pressed and drag the item over one of the edit boxes in the center of the form.

To accomplish this, I had to set the `DragMode` property of the list box (the source component) to `dmAutomatic` and implement the `OnDragOver` and `OnDragDrop` events of the target edit boxes (the two edit boxes are connected to the same event handlers, sharing the same code). In the first method, the program indicates that the edit boxes always accept the dragging operation, regardless of the source. In the second method, the program copies the text selected in the list box (the `Source` control of the dragging operation) to the edit box that fired the event (the `Sender` object). Here is the code for the two methods:

```
procedure TForm1.EditTypeDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;
end;
```

```
procedure TForm1.EditTypeDragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  (Sender as TEdit).Text := (Source as TListBox).Items
    [(Source as TListBox).ItemIndex];
end;
```

The units must match those available in the current family. In case of error, the text of the Type edit boxes is shown in red. This is the effect of the first part of the DoConvert method of the form, which is activated as soon as the value of one of the edit boxes for the units or the amount changes. After checking the types in the edit boxes, the DoConvert method does the actual conversion, displaying the result in the fourth, grayed edit box. In case of errors, you'll get a proper message in the same box. Here is the code:

```
procedure TForm1.DoConvert(Sender: TObject);  
var  
    BaseType, DestType: TConvType;  
begin  
    // get and check base type  
    if not DescriptionToConvType(CurrFamily, EditType.Text, BaseType) then  
        EditType.Font.Color := clRed  
    else  
        EditType.Font.Color := clBlack;  
  
    // get and check destination type  
    if not DescriptionToConvType(CurrFamily, EditDestination.Text,  
        DestType) then  
        EditDestination.Font.Color := clRed  
    else  
        EditDestination.Font.Color := clBlack;  
  
    if (DestType = 0) or (BaseType = 0) then  
        EditConverted.Text := 'Invalid type'  
    else  
        EditConverted.Text := FloatToStr (Convert (  
            StrToFloat (EditAmount.Text), BaseType, DestType));  
end;
```

If all this is not interesting enough for you, consider that the conversion types provided serve only as a demo: You can fully customize the engine, by providing the measurement units you are interested in, as described in the next section.

## What About Currency Conversions?

Converting currencies is not exactly the same as converting measurement units, as currency rates change at very high speed. In theory, you can register a conversion rate with Delphi's conversion engine. From time to time, you check the new rate exchange, unregister the existing conversion, and register a new one. However, keeping up with the actual rate means changing the conversion so often that the operation might not make a lot of sense. Also, you'll have to triangulate conversions: you have to define a base unit (probably the U.S. dollar if you live in America) and convert to and from this currency even for converting between two different ones.

What's more interesting is to use the engine for converting member currencies of the euro, for two reasons. First, conversion rates are fixed (until the single euro currency actually takes over). Second, the conversion among euro currencies is legally done by converting a currency to euros first and then from the euro amount to the other currency, the exact behavior of Delphi's conversion engine. There is only a small problem, as you should apply a rounding algorithm at every step of the conversion. I'll consider this problem after I've provided the base code for integrating euro currencies with Delphi 6 conversion engine.

**NOTE**

The ConvertIt demo of Delphi 6 provides support for euro conversions, using a slightly different rounding approach (which might be more correct or not, I'm not really sure). I've decided to keep this example anyway, as it is instructive in showing how to create a new measurement system (and I lacked another example as good).

The example, called EuroConv, is actually meant to teach how to register any new measurement unit with the engine. Following the template provided by the StdConvs unit, I've created a new unit (called EuroConvConst) and in the interface portion I've declared variables for the family and the specific units, as follows:

**interface****var**

```
// Euro Currency Conversion Units
cbEuroCurrency: TConvFamily;

cuEUR: TConvType;
cuDEM: TConvType; // Germany
cuESP: TConvType; // Spain
cuFRF: TConvType; // France
cuIEP: TConvType; // Ireland
cuITL: TConvType; // Italy
// and so on...
```

In the implementation portion of the unit, I've defined constants for the various official conversion rates:

**implementation****const**

```
DEMPerEuros = 1.95583;
ESPPerEuros = 166.386;
FRFPerEuros = 6.55957;
IEPPerEuros = 0.787564;
ITLPerEuros = 1936.27;
// and so on...
```

Finally, in the unit initialization code I've registered the family and the various currencies, each with its own conversion rate and a readable name:

#### initialization

```
// Euro Currency's family type
cbEuroCurrency := RegisterConversionFamily('EuroCurrency');

cuEUR := RegisterConversionType(
  cbEuroCurrency, 'EUR', 1);
cuDEM := RegisterConversionType(
  cbEuroCurrency, 'DEM', 1 / DEMPerEuros);
cuESP := RegisterConversionType(
  cbEuroCurrency, 'ESP', 1 / ESPPerEuros);
cuFRF := RegisterConversionType(
  cbEuroCurrency, 'FRF', 1 / FRFPerEuros);
cuIEP := RegisterConversionType(
  cbEuroCurrency, 'IEP', 1 / IEPPerEuros);
cuITL := RegisterConversionType(
  cbEuroCurrency, 'ITL', 1 / ITLPerEuros);
```

#### NOTE

The engine uses as a conversion factor the amount of the base unit to obtain the secondary ones, with a constant like `MetersPerInch`, for example. The standard rate of euro currencies is defined in the opposite way. For this reason, I've decided to keep the conversion constants with the official values (as `DEMPerEuros` above) and pass them to the engine as fractions (`1/DEMPerEuros`).

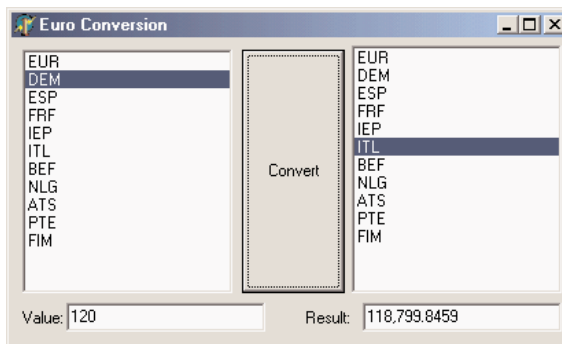
Having registered this unit, we can now convert 120 German marks to Italian liras by writing:

```
Convert (120, cuDEM, cuITL)
```

The demo program actually does a little more, providing two list boxes with the available currencies, extracted as in the previous example, and edit boxes for the input value and final result. You can see the form at run time in Figure 4.3.

**FIGURE 4.3:**

The output of the EuroConv unit, showing the use of Delphi's conversion engine with a custom measurement unit



The program works nicely but is not perfect, as the proper rounding is not applied. In fact, you should round not only the final result of the conversion but also the intermediate value. Using the conversion engine to accomplish this directly is not easy. The engine allows you to provide either a custom conversion function or a conversion rate. But writing identical conversion functions for all the various currencies seems a bad idea, so I've decided to go a different path. (You can see examples of custom conversion functions in the StdConvs unit, in the portion related to temperatures.)

In the EuroConv example, I've added to the unit with the conversion rates a custom function, called EuroConv, that does the proper conversion. Simply calling this function instead of the standard Convert function does the trick (and I really see no drawback to this approach, because in programs like this, you'll hardly mix currencies with meters or temperatures). As an alternative, I could inherit a new class from TConvTypeFactor, providing a new version of the FromCommon and ToCommon methods, or I could have called the overloaded versions of the RegisterConversionType that accepts these two functions as parameters. None of these techniques, however, would have allowed me to handle special cases, such as the conversion of a currency to itself.

This is the code of the EuroConv function, which uses the internal EuroRound function for rounding to the number of digits specified in the Decimals parameter (which must be between 3 and 6, according with the official rules):

```
type
    TEuroDecimals = 3..6;

function EuroConvert (const AValue: Double;
    const AFrom, ATo: TConvType;
    const Decimals: TEuroDecimals = 3): Double;

    function EuroRound (const AValue: Double): Double;
    begin
        Result := AValue * Power (10, Decimals);
        Result := Round (Result);
        Result := Result / Power (10, Decimals);
    end;

begin
    // check special case: no conversion
    if AFrom = ATo then
        Result := AValue
    else
        begin
            // convert to Euro, then round
            Result := ConvertFrom (AFrom, AValue);
            Result := EuroRound (Result);
```



```
    // convert to currency then round again
    Result := ConvertTo (Result, ATo);
    Result := EuroRound (Result);
end;
end;
```

Of course, you might want to extend the example by providing conversion to other non-euro currencies, eventually picking the values automatically from a Web site. I'll leave this as a rather complex exercise.

## The *TObject* Class

As mentioned earlier, a key element of the System unit is the definition of the *TObject* class, the *mother of all Delphi classes*. Every class in the system is a subclass of the *TObject* class, either directly (for example, if you indicate no base class) or indirectly. The whole hierarchy of the classes of an Object Pascal program has a single root. This allows you to use the *TObject* data type as a replacement for the data type of any class type in the system.

For example, event handlers of components usually have a *Sender* parameter of type *TObject*. This simply means that the *Sender* object can be of any class, since every class is ultimately derived from *TObject*. The typical drawback of such an approach is that to work on the object, you need to know its data type. In fact, when you have a variable or a parameter of the *TObject* type, you can apply to it only the methods and properties defined by the *TObject* class itself. If this variable or parameter happens to refer to an object of the *TButton* type, for example, you cannot directly access its *Caption* property. The solution to this problem lies in the use of the safe down-casting or run-time type information (RTTI) operators (*is* and *as*) discussed in Chapter 3.

There is another approach. For any object, you can call the methods defined in the *TObject* class itself. For example, the *ClassName* method returns a string with the name of the class. Because it is a class method (see Chapter 2 for details), you can actually apply it both to an object and to a class. Suppose you have defined a *TButton* class and a *Button1* object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

There are occasions when you need to use the name of a class, but it can also be useful to retrieve a class reference to the class itself or to its base class. The class reference, in fact, allows you to operate on the class at run time (as we've seen in the preceding chapter), while the class name is just a string. We can get these class references with the *ClassType* and *ClassParent* methods. The first returns a class reference to the class of the object, the second

to its base class. Once you have a class reference, you can apply to it any class methods of TObject—for example, to call the ClassName method.

Another method that might be useful is InstanceSize, which returns the run-time size of an object. Although you might think that the SizeOf global function provides this information, that function actually returns the size of an object reference—a pointer, which is invariably four bytes—instead of the size of the object itself.

In Listing 4.1, you can find the complete definition of the TObject class, extracted from the System unit. Beside the methods I've already mentioned, notice InheritsFrom, which provides a test very similar to the is operator but that can be applied also to classes and class references (while the first argument of is must be an object).



#### **Listing 4.1:** The definition of the *TObject* class (in the System RTL unit)

```
type
  Tobject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): Tobject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(
      const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID;out Obj): Boolean;
    class function GetInterfaceEntry(
      const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: Tobject;
      ExceptAddr: Pointer): HRESULT; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: Tobject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

**NOTE**

The `ClassInfo` method returns a pointer to the internal run-time type information (RTTI) of the class, introduced in the next chapter.

These methods of `TObject` are available for objects of every class, since `TObject` is the common ancestor class of every class. Here is how we can use these methods to access class information:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
    Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);

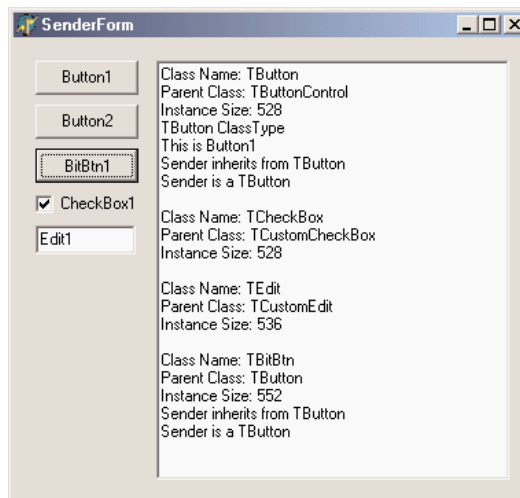
    if Sender.ClassParent <> nil then
        Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);

    Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
end;
```

The code checks to see whether the `ClassParent` is `nil` in case you are actually using an instance of the `TObject` type, which has no base type. This `ShowSender` method is part of the `IfSender` example on the companion CD. The method is connected with the `OnClick` event of several controls: three buttons, a check box, and an edit box. When you click each control, the `ShowSender` method is invoked with the corresponding control as sender (more on events in the next chapter). One of the buttons is actually a Bitmap button, an object of a `TButton` subclass. You can see an example of the output of this program at run time in Figure 4.4.

**FIGURE 4.4:**

The output of the `IfSender` example



You can use other methods to perform tests. For example, you can check whether the `Sender` object is of a specific type with the following code:

```
if Sender.ClassType = TButton then ...
```

You can also check whether the `Sender` parameter corresponds to a given object, with this test:

```
if Sender = Button1 then...
```

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class *or* one of its subclasses. This lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the `InheritsFrom` method, which is also called when you use the `is` operator. The following two tests are equivalent:

```
if Sender.InheritsFrom (TButton) then ...  
if Sender is TButton then ...
```

## Showing Class Information

I've extended the `IfSender` example to show a complete list of base classes of a given object or class. Once you have a class reference, in fact, you can add all of its base classes to the `ListParent` list box with the following code:

```
with ListParent.Items do  
begin  
  Clear;  
  while MyClass.ClassParent <> nil do  
  begin  
    MyClass := MyClass.ClassParent;  
    Add (MyClass.ClassName);  
  end;  
end;
```

You'll notice that we use a class reference at the heart of the `while` loop, which tests for the absence of a parent class (so that the current class is `TObject`). Alternatively, we could have written the `while` statement in either of the following ways:

```
while not MyClass.ClassNameIs ('TObject') do...  
while MyClass <> TObject do...
```

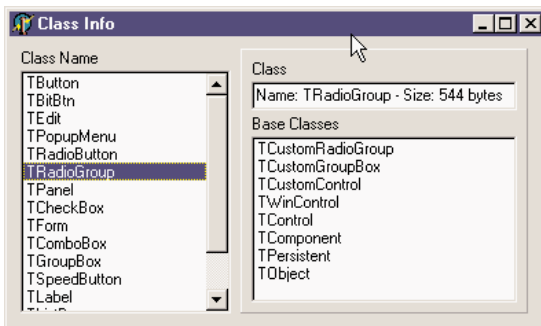
The code in the `with` statement referring to the `ListParent` list box is part of the `ClassInfo` example (see the companion CD), which displays the list of parent classes and some other information about a few components of the VCL, basically those on the Standard page of the Component Palette. These components are manually added to a dynamic array holding classes and declared as

```
private  
  ClassArray: array of TClass;
```

When the program starts, the array is used to show all the class names in a list box. Selecting an item from the list box triggers the visual presentation of its details and its base classes, as you can see in the output of the program, in Figure 4.5.

**FIGURE 4.5:**

The output of the ClassInfo example



**NOTE**

As a further extension to this example, it is possible to create a tree with all of the base classes of the various components in a hierarchy. To do that, I've created the VclHierarchy program, which you can find on my Web site, [www.marcocantu.com](http://www.marcocantu.com), in the CanTools section.

## What's Next?

In this chapter I've focused my attention on new features of the Delphi 6 function-based run-time library. I have provided only a summary of the entire RTL, not a complete overview, as this would have taken too much space. You can find more examples of the basic RTL functions of Delphi in my free electronic book *Essential Pascal*, which is featured on the companion CD.

In the next chapter, we'll start moving from the function-based RTL to the class-based RTL, which is the core of Delphi's class library. I won't debate whether the core classes common to the VCL and CLX, such as TObject, actually belong to the RTL or the class library. I've covered everything defined in System, SysUtils, and other units hosting functions and procedures in this chapter, while the next chapter focuses on the Classes unit and other core units defining classes.

Along with the preceding two chapters on the Object Pascal language, this will provide a foundation for discussing visual- and database-oriented classes, or components, if you prefer. Looking to the various library units, we'll find many more global functions, which don't belong to the core RTL but are still quite useful!

# Core Library Classes

- The RTL package, CLX, and VCL
- *TPersistent* and *published*
- The *TComponent* base class and its properties
- Components and ownership
- Events
- Lists, container classes, and collections
- Streaming
- Summarizing the units of the RTL package

**W**e saw in the preceding chapter that Delphi includes a large number of functions and procedures, but the real power of Delphi's visual programming lies in the huge class library it comes with. Delphi's standard class library contains hundreds of classes, with thousands of methods, and it is so large that I certainly cannot provide a detailed reference in this book. What I'll do, instead, is explore various areas of this library starting with this chapter and continuing through the following ones.

This first chapter is devoted to the core classes of the library as well as to some standard programming techniques, such as the definition of events. We'll explore some commonly used classes, such as lists, string lists, collections, and streams. We'll devote most of our time to exploring the content of the Classes unit, but we'll devote time also to other core units of the library.

Delphi classes can be used either entirely in code or within the visual form designer. Some of them are component classes, which show up in the Component Palette, while others are more general-purpose. The terms *class* and *component* can be used almost as synonyms in Delphi. Components are the central elements of Delphi applications. When you write a program, you basically choose a number of components and define their interactions. That's all there is to Delphi visual programming.

Before reading this chapter, you need to have a good understanding of the Object Pascal programming language, including inheritance, properties, virtual methods, class references, and so on, as discussed in Chapters 2 and 3 of this book.

## The RTL Package, VCL, and CLX

Until version 5, Delphi's class library was known as VCL, which stands for Visual Components Library. Kylix, the Delphi version for Linux, introduced a new component library, called CLX (pronounced "clicks" and standing for Component Library for X-Platform or Cross Platform). Delphi 6 includes both the VCL and CLX libraries. For visual components, the two class libraries are alternative one to the other. However, the core classes and the database and Internet portions of the two libraries are basically shared.

VCL was considered as a single large library, although programmers used to refer to different parts of it (components, controls, nonvisual components, data sets, data-aware controls, Internet components, and so on). CLX introduces a distinction in four parts: BaseCLX, VisualCLX, DataCLX, and NetCLX. Only in VisualCLX does the library use a totally different approach between the two platforms, with the rest of the code being inherently portable to Linux. In the following section, I discuss the sections of these two libraries, while the rest of the chapter focuses on the common core classes.

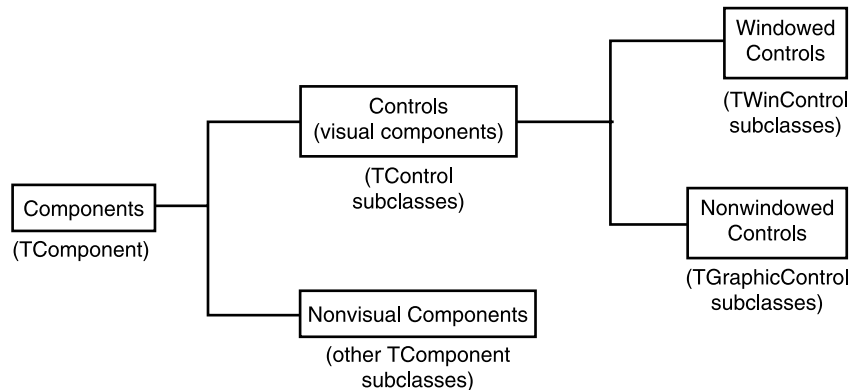
In Delphi 6, this distinction is underlined by the fact that the core non-visual components and classes of the library are part of the new RTL package, which is used by both VCL and CLX. Moreover, using this package in non-visual applications (for example, Web server programs) allows you to reduce the size of the files to deploy and load in memory considerably.

## Traditional Sections of VCL

Delphi programmers use to refer to different sections of VCL with names Borland originally suggested in its documentation, and names that became common afterwards for different groups of components. Technically, components are subclasses of the `TComponent` class, which is one of the root classes of the hierarchy, as you can see in Figure 5.1. Actually the `TComponent` class inherits from the `TPersistent` class; the role of these two classes will be explained in the next section.

**FIGURE 5.1:**

A graphical representation of the main groups of components of VCL



Besides components, the library includes classes that inherit directly from `TObject` and from `TPersistent`. These classes are collectively known as *Objects* in portions of the documentation, a rather confusing name for me. These noncomponent classes are often used for values of properties or as utility classes used in code; not inheriting from `TComponent`, these classes cannot be used directly in visual programming.

### NOTE

To be more precise, noncomponent classes cannot be made available in the Component Palette and cannot be dropped directly into a form, but they can be visually managed with the Object Inspector, as subproperties of other properties or items of collections of various types. So even noncomponent classes are often easily used when interacting with the Form Designer.



The component classes can be further divided into two main groups: controls and nonvisual components. Controls groups all the classes that descend from `TControl`.

**Controls** have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different subspecifications, window-based or graphical, but I'll discuss them in more detail in the next chapter.

**Nonvisual components** are all the components that are not controls—all the classes that descend from `TComponent` but not from `TControl`. At design time, a nonvisual component appears on the form as an icon (optionally with a caption below it). At run time, some of these components may be visible (for example, the standard dialog boxes), and others are always invisible (for example, the database table component).

**TIP**

You can simply move the mouse cursor over a control or component in the Form Designer to see a Tooltip with its name and class type (and, in Delphi 6, some extended information). You can also use an environment option, Show Component Captions, to see the name of a nonvisual component right under its icon.

## The Structure of CLX

This is the traditional subdivision of VCL, which is very common for Delphi programmers. Even with the introduction of CLX and some new naming schemes, the traditional names will probably survive and merge into Delphi programmers' jargon.

Borland now refers to different portions of the CLX library using one terminology under Linux and a slightly different (and less clear) naming structure in Delphi. This is the subdivision of the Linux-compatible library:

**BaseCLX** forms the core of the class library, the topmost classes (such as `TComponent`), and several general utility classes (including lists, containers, collections, and streams). Compared to the corresponding classes of VCL, BaseCLX is largely unchanged and is highly portable between the Windows and Linux platforms. This chapter is largely devoted to exploring BaseCLX and the common VCL core classes.

**VisualCLX** is the collection of visual components, generally indicated as controls. This is the portion of the library that is more tightly related to the operating system: VisualCLX is implemented on top of the Qt library, available both on Windows and on Linux. Using VisualCLX allows for full portability of the visual portion of your application between Delphi on Windows and Kylix on Linux. However, most of the VisualCLX components have corresponding VCL controls, so that you can also easily move your code from one library to the other. I'll discuss VisualCLX and the controls of VCL in the next chapter.

**DataCLX** comprises all the database-related components of the library. Actually, DataCLX is the front end of the new dbExpress database engine included in Delphi 6 and Kylix. Delphi includes also the traditional BDE front end, dbGo, and InterBase Express (IBX). If we can consider all these components as part of DataCLX, only the dbExpress front end and IBX are portable between Windows and Linux. DataCLX includes also the ClientDataSet component, now indicated as MyBase, and other related classes. Delphi's data access components are discussed in Part III of the book.

**NetCLX** includes the Internet-related components, from the WebBroker framework, to the HTML producer components, from Indy (Internet Direct) to Internet Express, from the new Site Express to XML support. This part of the library is, again, highly portable between Windows and Linux. Internet support is discussed in the last part of the book.

## VCL-Specific Sections of the Library

The preceding areas of the library are available, with the differences I've mentioned, on both Delphi and Kylix. In Delphi 6, however, there are other sections of VCL, which for one reason or another are specific to Windows only:

- The Delphi ActiveX (DAX) framework provides support for COM, OLE Automation, ActiveX, and other COM-related technologies. See Chapter 16 for more information on this area of Delphi.
- The Decision Cube components provide OLAP support but have ties with the BDE and haven't been updated recently. Decision Cube is not discussed in the book.

Finally, the default Delphi 6 installation includes some third-party components, such as TeeChart for business graphics and QuickReport for reporting. These components will be mentioned in the book but are not discussed in detail.

## The *TPersistent* Class

The first core class of the Delphi library we'll look at is the *TPersistent* class, which is quite a strange one: it has very little code and almost no direct use, but it provides a foundation for the entire idea of visual programming. You can see the definition of the class in Listing 5.1.



**Listing 5.1:** The definition of the *TPersistent* class, from the *Classes* unit

```
{ $M+ }  
TPersistent = class(TObject)  
  private  
    procedure AssignError(Source: TPersistent);
```

```
protected
  procedure AssignTo(Dest: TPersistent); virtual;
  procedure DefineProperties(Filer: TFile); virtual;
  function GetOwner: TPersistent; dynamic;
public
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); virtual;
  function GetNamePath: string; dynamic;
end;
```

---

As the name implies, this class handles persistency—that is, saving the value of an object to a file to be used later to re-create the object in the same state and with the same data. Persistency is a key element of visual programming. In fact (as we saw in Chapter 1) at design time in Delphi you manipulate actual objects, which are saved to DFM files and re-created at run time when the specific component container—form or data module—is created.

The streaming support, though, is not embedded in the `TPersistent` class but is provided by other classes, which target `TPersistent` and its descendants. In other words, you can “persist” with Delphi default streaming-only objects of classes inheriting from `TPersistent`. One of the reasons for this behavior lies in the fact that the class is compiled with a special option turned on, `{M+}`. This flag activates the generation of extended RTTI information for the published portion of the class.

Delphi’s streaming system, in fact, doesn’t try to save the in-memory data of an object, which would be complex because of the many pointers to other memory locations, totally meaningless when the object would be reloaded. Instead, Delphi saves objects by listing the value of all of properties marked with a special keyword, `published`. When a property refers to another object, Delphi saves the name of the object or the entire object (with the same mechanism) depending on its type and relationship with the main object.

Of the methods of the `TPersistent` class, the only one you’ll generally use is the `Assign` procedure, which can be used for copying the actual value of an object. In the library, this method is implemented by many noncomponent classes but by very few components. Actually, most subclasses reimplement the virtual protected `AssignTo` method, called by the default implementation of `Assign`.

---

**NOTE**

Other methods include `DefineProperties`, used for customizing the streaming system and adding extra information (pseudo-properties), and the `GetOwner` and `GetNamePath` methods used by collections and other special classes to identify themselves to the Object Inspector.

## The *published* Keyword

Along with the `public`, `protected`, and `private` access directives, you can use a fourth one, called `published`. For any `published` field, property, or method, the compiler generates extended RTTI information, so that Delphi's run time environment or a program can query a class for its published interface. For example, every Delphi component has a published interface that is used by the IDE, in particular the Object Inspector. A regular use of `published` fields is important when you write components. Usually, the `published` part of a component contains no fields or methods but properties and events.

When Delphi generates a form or data module, it places the definitions of its components and methods (the event handlers) in the first portion of its definition, before the `public` and `private` keywords. These fields and methods of the initial portion of the class are `published`. The default is `published` when no special keyword is added before an element of a component class.

To be more precise, `published` is the default keyword only if the class was compiled with the `$M+` compiler directive or is descended from a class compiled with `$M+`. As this directive is used in the `TPersistent` class, most classes of `VCL` and all of the component classes default to `published`. However, noncomponent classes in Delphi (such as `TStream` and `TList`) are compiled with `$M-` and default to `public` visibility.

The methods assigned to any event should be `published` methods, and the fields corresponding to your components in the form should be `published` to be automatically connected with the objects described in the DFM file and created along with the form. (We'll see later in this chapter the details of this situation and the problems it generates.)

### Accessing Published Fields and Methods

As I've mentioned, there are three different declarations that make sense in the `published` section of a class: fields, methods, and properties. I'll discuss properties in the section "Accessing Properties by Name," while here I'll introduce possible ways of interacting with fields and methods first. The `TObject` class, in fact, has three interesting methods for this area: `MethodAddress`, `MethodName`, and `FieldAddress`.

The first function, `MethodAddress`, returns the memory address of the compiled code (a sort of function pointer) of the method passed as parameter in a string. By assigning this method address to the `Code` field of a `TMethod` structure and assigning an object to the `Data` field, you can obtain a complete method pointer. At this point, to call the method you'll need to cast it to

*Continued on next page*

the proper method pointer type. This is a code fragment highlighting the key points of this technique:

```
var
    Method: TMethod;
    Evt: TNotifyEvent;
begin
    Method.Code := MethodAddress ('Button1Click');
    Method.Data := Self;
    Evt := TNotifyEvent(Method);
    Evt (Sender); // call the method
end;
```

Delphi uses similar code to assign an event handler when it loads a DFM file, as these files store the name of the methods used to handle the events, while the components actually store the method pointer. The second method, `MethodName`, does the opposite transformation, returning the name of the method at a given memory address. This can be used to obtain the name of an event handler, given its value, something Delphi does when streaming a component into a DFM file.

Finally, the `FieldAddress` method of `TObject` returns the memory location of a published field, given its name. This is used by Delphi to connect components created from the DFM files with the fields of their owner (for example, a form) having the same name.

Notice that these three methods are seldom used in “normal” programs but play a key role to make Delphi work as it actually does and are strictly related to the streaming system. You’ll need to use these methods only when writing extremely dynamic programs or special-purpose wizards or other Delphi extensions.

## Accessing Properties by Name

The Object Inspector displays a list of an object’s published properties, even for components you’ve written. To do this, it relies on the RTTI information generated for published properties. Using some advanced techniques, an application can retrieve a list of the published properties of an object and use them.

Although this capability is not very well known, in Delphi it is possible to access properties by name simply by using the string with the name of the property and then retrieving its value. Access to the RTTI information of properties is provided through a group of undocumented subroutines, part of the `TypInfo` unit.

**WARNING** These subroutines have always been undocumented in past versions of Delphi, so that Borland remained free to change them. However, from Delphi 1 to Delphi 5, changes were actually very limited and related only to the data structures declared in `TypInfo`, not the functions provided by the unit. In Delphi 5 Borland actually added many more goodies, and a few “helper” routines, that are officially promoted (even if still not fully documented in the help file but only with comments provided in the unit).

Rather than explore the entire `TypInfo` unit here, we will look at only the minimal code required to access properties by name. Prior to Delphi 5 it was necessary to use the `GetPropInfo` function to retrieve a pointer to some internal property information and then apply one of the access functions, such as `GetStrProp`, to this pointer. You also had to check for the existence and the type of the property.

Delphi 5 introduced a new set of `TypInfo` routines, including the handy `GetPropValue`, which returns a variant with the value of the property or `varNULL` if the property doesn't exist. You simply pass to this function the object and a string with the property name. A further optional parameter allows you to choose the format for returning values of properties of the set type.

For example, we can call

```
ShowMessage (GetPropValue (Button1, 'Caption'));
```

This call has the same effect as calling `ShowMessage`, passing as parameter `Button1.Caption`. The only real difference is that this version of the code is much slower, since the compiler generally resolves normal access to properties in a more efficient way. The advantage of the run-time access is that you can make it very flexible, as in the following `RunProp` example (also available on the companion CD).

This program displays in a list box the value of a property of any type for each component of a form. The name of the property we are looking for is provided in an edit box. This makes the program very flexible. Besides the edit box and the list box, the form has a button to generate the output and some other components added only to test their properties. When you click the button, the following code is executed:

```
uses
  TypInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Value: Variant;
begin
  ListBox1.Clear;
```

```

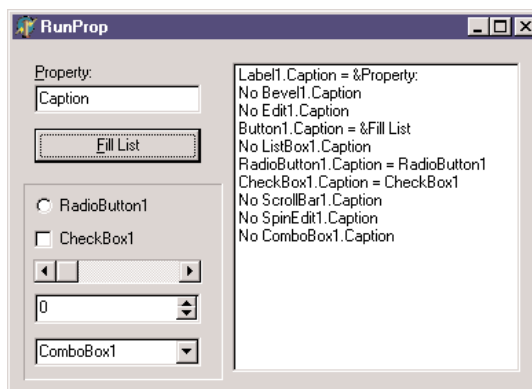
for I := 0 to ComponentCount -1 do
begin
  if IsPublishedProp (Components[I], Edit1.Text) then
  begin
    Value := GetPropValue (Components[I], Edit1.Text);
    ListBox1.Items.Add (Components[I].Name + '.' + Edit1.Text + '=' + string (Value));
  end
  else
    ListBox1.Items.Add ('No ' + Components[I].Name + '.' + Edit1.Text);
end

```

You can see the effect of pressing the Fill List button while using the default *Caption* value in the edit box in Figure 5.2. You can try with any other property name. Numbers will be converted to strings by the variant conversion. Objects (such as the value of the Font property) will be displayed as memory addresses.

**FIGURE 5.2:**

The output of the RunProp example, which accesses properties by name at run time



### WARNING

Do not use regularly the `TypeInfo` unit instead of polymorphism and other property-access techniques. Use base-class property access first, or use the `safe as` typecast when required, and reserve RTTI access to properties as a very last resort. Using `TypeInfo` techniques makes your code slower, more complex, and more prone to human error; in fact, it skips the compile-time type-checking.

## The *TComponent* Class

If the `TPersistent` class is really more important than it seems at first sight, the key class at the heart of Delphi's component-based class library is `TComponent`, which inherits from

TPersistent (and from TObject). The TComponent class defines many core elements of components, but it is not as complex as you might think, as the base classes and the language already provide most of what's actually needed.

I won't explore all of the details of the TComponent class, some of which are more important for component designers than they are for component users. I'll just discuss ownership (which accounts for some public properties of the class) and the two published properties of the class, Name and Tag.

## Ownership

One of the core features of the TComponent class is the definition of ownership. When a component is created, it can be assigned an owner component, which will be responsible for destroying it. So every component can have an owner and can also be the owner of other components. Several public methods and properties of the class are actually devoted to handling the *two sides* of ownership. Here is a list, extracted from the class declaration (in the Classes unit of VCL):

```
type
TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
public
  constructor Create(AOwner: TComponent); virtual;
  procedure DestroyComponents;
  function FindComponent(const AName: string): TComponent;
  procedure InsertComponent(AComponent: TComponent);
  procedure RemoveComponent(AComponent: TComponent);

  property Components[Index: Integer]: TComponent read GetComponent;
  property ComponentCount: Integer read GetComponentCount;
  property ComponentIndex: Integer
    read GetComponentIndex write SetComponentIndex;
  property Owner: TComponent read FOwner;
```

If you create a component giving it an owner, this will be added to the list of components (InsertComponent), which is accessible using the Components array property. The specific component has an Owner and knows its position in the owner components list, with the ComponentIndex property. Finally, the destructor of the owner will take care of the destruction of the object it owns, calling DestroyComponents. There are a few more protected methods involved, but this should give you the overall picture.

What is important to emphasize is that component ownership can solve a large part of the memory management problems of your applications, if used properly. If you always create components with an owner—the default operation if you use the visual designers of the IDE—you only need to remember to destroy these component containers when they are not needed anymore, and you can forget about the components they contain. For example, you



delete a form to destroy all of the components it contains at once, which is a large simplification compared to having to remember to free each and every object individually.

## The Components Array

The `Components` property can also be used to access one component owned by another—let's say, a form. This can be very handy (compared to using directly a specific component) for writing generic code, acting on all or many components at a time. For example, you can use the following code to add to a list box the names of all the components of a form (this code is actually part of the `ChangeOwner` example, presented in the next section):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do
    ListBox1.Items.Add (Components [I].Name);
end;
```

This code uses the `ComponentCount` property, which holds the total number of components owned by the current form, and the `Components` property, which is actually the list of the owned components. When you access a value from this list you get a value of the `TComponent` type. For this reason you can directly use only the properties common to all components, such as the `Name` property. To use properties specific to particular components, you have to use the proper type-downcast (`as`).

### NOTE

In Delphi, some components are also component containers: the `GroupBox`, `Panel`, `PageControl`, and, of course, `Form` components. When you use these controls, you can add other components inside them. In this case, the container is the parent of the components (as indicated by the `Parent` property), while the form is their owner (as indicated by the `Owner` property). You can use the `Controls` property of a form or group box to navigate the child controls, and you can use the `Components` property of the form to navigate all the owned components, regardless of their parent.

Using the `Components` property, we can always access each component of a form. If you need access to a specific component, however, instead of comparing each name with the name of the component you are looking for, you can let Delphi do this work, by using the `FindComponent` method of the form. This method simply scans the `Components` array looking for a name match. More information about the role of the `Name` property for a component is in a later section.

## Changing the Owner

We have seen that almost every component has an owner. When a component is created at design time (or from the resulting DFM file), its owner will invariably be its form. When you create a component at run time, the owner is passed as a parameter to the Create constructor.

Owner is a read-only property, so you cannot change it. The owner is set at creation time and should generally not change during the lifetime of a component. To understand why you should not change the owner of a component at design time nor freely change its name, read the following discussion. Be warned, that the topic covered is not simple, so if you're only starting with Delphi, you might want to come back to this section at a later time.

To change the owner of a component, you can call the InsertComponent and RemoveComponent methods of the owner itself, passing the current component as parameter. Using these methods you can change a component's owner. However, you cannot apply them directly in an event handler of a form, as we attempt to do here:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    RemoveComponent (Button1);  
    Form2.InsertComponent (Button1);  
end;
```

This code produces a memory access violation, because when you call RemoveComponent, Delphi disconnects the component from the form field (Button1), setting it to nil. The solution is to write a procedure like this:

```
procedure ChangeOwner (Component, NewOwner: TComponent);  
begin  
    Component.Owner.RemoveComponent (Component);  
    NewOwner.InsertComponent (Component);  
end;
```

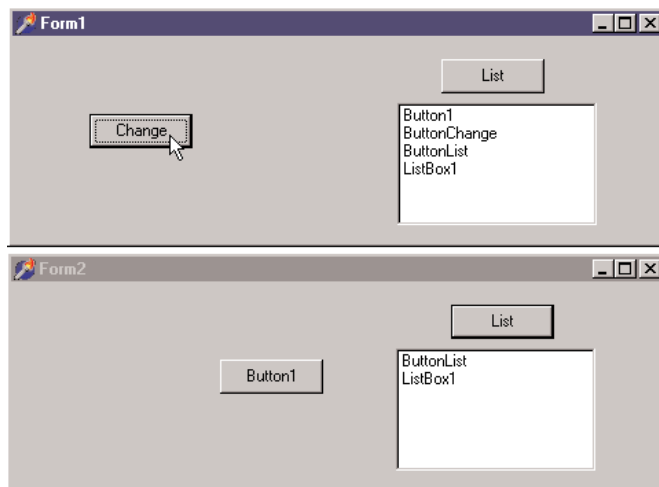
This method (extracted from the ChangeOwner example) changes the owner of the component. It is called along with the simpler code used to change the parent component; the two commands combined move the button *completely* to another form, changing its owner:

```
procedure TForm1.ButtonChangeClick(Sender: TObject);  
begin  
    if Assigned (Button1) then  
        begin  
            // change parent  
            Button1.Parent := Form2;  
            // change owner  
            ChangeOwner (Button1, Form2);  
        end;  
end;
```

The method checks whether the `Button1` field still refers to the control, because while moving the component, Delphi will set `Button1` to `nil`. You can see the effect of this code in Figure 5.3.

**FIGURE 5.3:**

In the `ChangeOwner` example, clicking the `Change` button moves the `Button1` component to the second form.



To demonstrate that the `Owner` of the `Button1` component actually changes, I've added another feature to both forms. The `List` button fills the list box with the names of the components each form owns, using the procedure shown in the previous section. Click the two `List` buttons before and after moving the component, and you'll see what happens behind the scenes. As a final feature, the `Button1` component has a simple handler for its `OnClick` event, to display the caption of the owner form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage ('My owner is ' + ((Sender as TButton).Owner as TForm).Caption);
end;
```

## The *Name* Property

Every component in Delphi should have a name. The name must be unique within the owner component, which is generally the form into which you place the component. This means that an application can have two different forms, each with a component with the same name, although you might want to avoid this practice to prevent confusion. It is generally better to keep component names unique throughout an application.

Setting a proper value for the `Name` property is very important: If it's too long, you'll need to type a lot of code to use the object; if it's too short, you may confuse different objects.

Usually the name of a component has a prefix with the component type; this makes the code more readable and allows Delphi to group components in the combo box of the Object Inspector, where they are sorted by name. There are three important elements related to the Name property of the components:

- First, the value of the Name property is used to define the name of the object in the declaration of the form class. This is the name you're generally going to use in the code to refer to the object. For this reason, the value of the name property must be a legal Pascal identifier (it has to be without spaces and must start with a letter, not a number).
- Second, if you set the Name property of a control before changing its Caption or Text property, the new name is often copied to the caption. That is, if the name and the caption are identical, then changing the name will also change the caption.
- Third, Delphi uses the name of the component to create the default name of the methods related to its events. If you have a Button1 component, its default OnClick event handler will be called Button1Click, unless you specify a different name. If you later change the name of the component, Delphi will modify the names of the related methods accordingly. For example, if you change the name of the button to MyButton, the Button1Click method automatically becomes MyButtonClick.

As mentioned earlier, if you have a string with the name of a component, you can get its instance by calling the FindComponent of its owner, which returns nil in case the component is not found. For example, you can write

```
var
  Comp: TComponent;
begin
  Comp := FindComponent ('Button1');
  if Assigned (Comp) then
    with Comp as TButton do
      // some code...
```

## NOTE

Delphi includes also a FindGlobalComponent function, which finds a top-level component, basically a form or data module, that has a given name. To be precise, the FindGlobalComponent function calls one or more installed functions, so in theory you can modify the way the function works. However, as FindGlobalComponent is used by the streaming system, I strongly recommend against installing your own replacement functions. If you want to have a customized way to search for components on other containers, simply write a new function with a custom name.

## Removing Form Fields

Every time you add a component to a form, Delphi adds an entry for it, along with some of its properties, to the DFM file. To the Pascal file, Delphi adds the corresponding field in the form class declaration. When the form is created, Delphi loads the DFM file and uses it to re-create all the components and set their properties back. Then it hooks the new object with the form field corresponding to its Name property.

For this reason, it is certainly possible to have a component without a name. If your application will not manipulate the component or modify it at run time, you can remove the component name from the Object Inspector. Examples are a static label with fixed text, or a menu item, or even more obviously, menu item separators. By blanking out the name, you'll remove the corresponding element from the form class declaration. This reduces the size of the form object (by only four bytes, the size of the object reference) and it reduces the DFM file by not including a useless string (the component name). Reducing the DFM also implies reducing the final EXE file size, even if only slightly.

### WARNING

If you blank out component names, just make sure to leave at least one named component of each class used on the form so that the smart linker will link in the required code for the class. If, as an example, you remove from a form all the fields referring to `TLabel` components, the Delphi linker will remove the implementation of the `TLabel` class from the executable file. The effect is that when the system loads the form at run time, it is unable to create an object of an unknown class and issues an error indicating that the class is not available.

You can also keep the component name and manually remove the corresponding field of the form class. Even if the component has no corresponding form field, it is created anyway, although using it (through the `FindComponent` method, for example) will be a little more difficult.

## Hiding Form Fields

Many OOP purists complain that Delphi doesn't really follow the encapsulation rules, because all of the components of a form are mapped to public fields and can be accessed from other forms and units. Fields for components, in fact, are listed in the first unnamed section of a class declaration, which has a default visibility of published. However, Delphi does that only as a default to help beginners learn to use the Delphi visual development environment quickly. A programmer can follow a different approach and use properties and methods to operate on forms. The risk, however, is that another programmer of the same team might inadvertently bypass this approach, directly accessing the components if they are left in the published section. The solution, which many programmers don't know about, is to move the components to the private portion of the class declaration.

As an example, I've taken a very simple form with an edit box, a button, and a list box. When the edit box contains text and the user presses the button, the text is added to the list box. When the edit box is empty, the button is disabled. This is the simple code of the HideComp example:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ListBox1.Items.Add (Edit1.Text);  
end;  
  
procedure TForm1.Edit1Change(Sender: TObject);  
begin  
    Button1.Enabled := Length (Edit1.Text) <> 0;  
end;
```

I've listed these methods only to show you that in the code of a form we usually refer to the available components, defining their interactions. For this reason it seems impossible to get rid of the fields corresponding to the component. However, what we can do is hide them, moving them from the default published section to the private section of the form class declaration:

```
TForm1 = class(TForm)  
    procedure Button1Click(Sender: TObject);  
    procedure Edit1Change(Sender: TObject);  
    procedure FormCreate(Sender: TObject);  
private  
    Button1: TButton;  
    Edit1: TEdit;  
    ListBox1: TListBox;  
end;
```

Now if you run the program you'll get in trouble: The form will load fine, but because the private fields are not initialized, the events above will use `nil` object references. Delphi usually initializes the published fields of the form using the components created from the DFM file. What if we do it ourselves, with the following code?

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Button1 := FindComponent ('Button1') as TButton;  
    Edit1 := FindComponent ('Edit1') as TEdit;  
    ListBox1 := FindComponent ('ListBox1') as TListBox;  
end;
```

It will *almost* work, but it generates a system error, similar to the one we discussed in the previous section. This time, the private declarations will cause the linker to link in the implementations of those classes, but the problem is that the streaming system needs to know the names of the classes in order to locate the class reference needed to construct the components while loading the DFM file.

The final touch we need is some registration code to tell Delphi at run time about the existence of the component classes we want to use. We should do this before the form is created, so I generally place this code in the initialization section of the unit:

**initialization**

```
RegisterClasses ([TButton, TEdit, TListBox]);
```

Now the question is, is this really worth the effort? What we obtain is a higher degree of encapsulation, protecting the components of a form from other forms (and other programmers writing them). I have to say that replicating these steps for each and every form can be tedious, so I ended up writing a wizard to generate this code for me on the fly. The wizard is far from perfect, as it doesn't handle changes automatically, but it is usable. You can find it on my Web site, [www.marcocantu.com](http://www.marcocantu.com), under the CanTools section. My simple wizard apart, for a large project built according to the principles of object-oriented programming, I recommend you consider this or a similar technique.

## The Customizable Tag Property

The Tag property is a strange one, because it has no effect at all. It is merely an extra memory location, present in each component class, where you can store custom values. The kind of information stored and the way it is used are completely up to you.

It is often useful to have an extra memory location to attach information to a component without needing to define your component class. Technically, the Tag property stores a long integer so that, for example, you can store the entry number of an array or list that corresponds to an object. Using typecasting, you can store in the Tag property a pointer, an object, or anything else that is four bytes wide. This allows a programmer to associate virtually anything with a component using its tag. We'll see how to use this property in several examples in future chapters, including the ODMenu examples in Chapter 5.

## Events

Now that I've introduced the TComponent class, there is one more element of Delphi we have to introduce. Delphi components, in fact, are programmed using "PME," properties, methods, and events. If methods and properties should be clear by now, events have not been fully introduced yet. The reason is that events don't imply a new language feature but are simply a standard coding technique. An event, in fact, is technically a property, with the only difference being that it refers to a method (a method pointer type, to be precise) instead of other types of data.

## Events in Delphi

When a user does something with a component, such as clicking it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding operating system message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than operating system messages, and Delphi provides a number of extra inter-component messages.

From a theoretical point of view, an event is the result of a request sent to a component or control, which can respond to the message. Following this approach, to handle the click event of a button, we would need to subclass the `TButton` class and add the new event handler code inside the new class.

In practice, creating a new class for every component you want to use is too complex to be a reasonable solution. In Delphi, the event handler of a component usually is a method of the form that holds the component, not of the component itself. In other words, the component relies on its owner, the form, to handle its events. This technique is called *delegation*, and it is fundamental to the Delphi component-based model. This way, you don't have to modify the `TButton` class, unless you want to define a new type of component, but simply customize its owner to modify the behavior of the button.

## Method Pointers

Events rely on a specific feature of the Object Pascal language: *method pointers*. A method pointer type is like a procedural type, but one that refers to a method. Technically, a method pointer type is a procedural type that has an implicit `Self` parameter. In other words, a variable of a procedural type stores the address of a function to call, provided it has a given set of parameters. A method pointer variable stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as `Self` inside the method body when the method code is called using this method pointer.

---

**NOTE**

This explains the definition of Delphi's generic `TMethod` type, a record with a `Code` field and a `Data` field.



The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords of object at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

When you have declared a method pointer, such as the one above, you can declare a variable of this type and assign to it a compatible method—a method that has the same parameters—of another object.

When you add an `OnClick` event handler for a button, Delphi does exactly that. The button has a method pointer type property, named `OnClick`, and you can directly or indirectly assign to it a method of another object, such as a form. When a user clicks the button, this method is executed, even if you have defined it inside another class.

What follows is a sketch of the code actually used by Delphi to define the event handler of a button component and the related method of a form:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;
```

Now inside a procedure, you can write

```
MyButton.OnClick := Form1.Button1Click;
```

The only real difference between this code fragment and the code of VCL is that `OnClick` is a property name, and the actual data it refers to is called `FOnClick`. An event that shows up in the Events page of the Object Inspector, in fact, is nothing more than a property of a method pointer type. This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it.

## Events Are Properties

Another important concept I've already mentioned is that events are properties. This means that to handle an event of a component, you assign a method to the corresponding event property. When you double-click an event in the Object Inspector, a new method is added to the owner form and assigned to the proper event property of the component.

This is why it is possible for several events to share the same event handler or change an event handler at run time. To use this feature, you don't need much knowledge of the language. In fact, when you select an event in the Object Inspector, you can press the arrow button on the right of the event name to see a drop-down list of "compatible" methods—a list of methods having the same method pointer type. Using the Object Inspector, it is easy to select the same method for the same event of different components or for different, compatible events of the same component.

As we've added some properties to the `TDate` class in Chapter 3, we can add one event. The event is going to be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the value of the date has changed. To define an event, we simply define a property corresponding to it, and we add some data to store the actual method pointer the event refers to. These are the new definitions added to the class, available in the `DateEvt` example:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FOnChange write FOnChange;
    ...
end;
```

The property definition is actually very simple. A user of this class can assign a new value to the property and, hence, to the `FOnChange` private field. The class doesn't assign a value to this `FOnChange` field; it is the user of the component who does the assignment. The `TDate` class simply calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned. The `DoChange`

method (declared as a dynamic method as it is traditional with event firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
    if Assigned (FOnChange) then
        FOnChange (Self);
end;
```

The DoChange method in turn is called every time one of the values changes, as in the following method:

```
procedure TDate.SetValue (y, m, d: Integer);
begin
    fDate := EncodeDate (y, m, d);
    // fire the event
    DoChange;
```

Now if we look at the program that uses this class, we can simplify its code considerably. First, we add a new custom method to the form class:

```
type
    TDateForm = class(TForm)
        ...
        procedure DateChange(Sender: TObject);
```

The code of this method simply updates the label with the current value of the Text property of the TDate object:

```
procedure TDateForm.DateChange;
begin
    LabelDate.Caption := TheDay.Text;
end;
```

This event handler is then installed in the FormCreate method:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
    TheDay := TDate.Init (2001, 7, 4);
    LabelDate.Caption := TheDay.Text;
    // assign the event handler for future changes
    TheDay.OnChange := DateChange;
end;
```

Well, this seems like a lot of work. Was I lying when I told you that the event handler would save us some coding? No. Now, after we've added some code, we can completely forget about updating the label when we change some of the data of the object. Here, as an example, is the handler of the OnClick event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
    TheDay.Increase;
end;
```

The same simplified code is present in many other event handlers. Once we have installed the event handler, we don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that we had to write some code at the beginning because this is not a component installed in Delphi but simply a class. With a component, you simply select the event handler in the Object Inspector and write a single line of code to update the label. That's all.

**NOTE**

This is meant to be just a short introduction to defining events. A basic understanding of these features is important for every Delphi programmer. If your aim is to write new components, with complex events, you'll find a lot more information on all these topics in Chapter 11.

## Lists and Container Classes

It is often important to handle groups of components or objects. Besides using standard arrays and dynamic arrays, there are a few classes of VCL that represent lists of other objects. These classes can be divided into three groups: simple lists, collections, and containers. The last group was introduced in Delphi 5 and has been further expanded in Delphi 6.

### Lists and String Lists

Lists are represented by the generic list of objects, `TList`, and by the two lists of strings, `TStrings` and `TStringList`:

- `TList` defines a list of pointers, which can be used to store objects of any class. A `TList` is more flexible than a dynamic array, because it is expanded automatically, simply by adding new items to it. The advantage of dynamic arrays over a `TList`, instead, is that dynamic arrays allow you to indicate a specific type for contained objects and perform the proper compile-time type checking.
- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves, such as a list box.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with their own storage. You can use this class to define a list of strings in a program.

`TStringList` and `TStrings` objects have both a list of strings and a list of objects associated with the strings. This opens up a number of different uses for these classes. For example, you can use them for dictionaries of associated objects or to store bitmaps or other elements to be used in a list box.

The two classes of lists of strings also have ready-to-use methods to store or load their contents to or from a text file, `SaveToFile` and `LoadFromFile`. To loop through a list, you can use a simple `for` statement based on its index, as if the list were an array. All these lists have a number of methods and properties. You can operate on lists using the array notation (“[” and “]”) both to read and to change elements. There is a `Count` property, as well as typical access methods, such as `Add`, `Insert`, `Delete`, `Remove`, and search methods (for example, `IndexOf`). In Delphi 6, the `TList` class has an `Assign` method that, besides copying the source data, can perform set operations on the two lists, including *and*, *or*, and *xor*.

To fill a string list with items and later check whether one is present, you can write code like this:

```
var
  sl: TStringList;
  idx: Integer;
begin
  sl := TStringList.Create;
  try
    sl.Add ('one');
    sl.Add ('two');
    sl.Add ('three');
    // later
    idx := sl.IndexOf ('two');
    if idx >= 0 then
      ShowMessage ('String found');
  finally
    sl.Free;
  end;
end;
```

## Using Lists of Objects

We can write an example focusing on the use of the generic `TList` class. When you need a list of any kind of data, you can generally declare a `TList` object, fill it with the data, and then access the data while casting it to the proper type. The `ListDemo` example demonstrates just this. It also shows the pitfalls of this approach. Its form has a private variable, holding a list of dates:

```
private
  ListDate: TList;
```

This list object is created when the form itself is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ListDate := TList.Create;
end;
```

A button of the form adds a random date to the list (of course, I've included in the project the unit containing the date component built in the previous chapter):

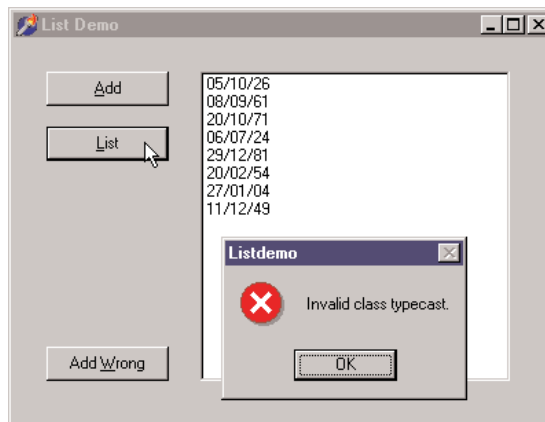
```
procedure TForm1.ButtonAddClick(Sender: TObject);  
begin  
    ListDate.Add (TDate.Create (1900 + Random (200), 1 + Random (12),  
        1 + Random (30)));  
end;
```

When you extract the items from the list, you have to cast them back to the proper type, as in the following method, which is connected to the List button (you can see its effect in Figure 5.4):

```
procedure TForm1.ButtonListDateClick(Sender: TObject);  
var  
    I: Integer;  
begin  
    ListBox1.Clear;  
    for I := 0 to ListDate.Count - 1 do  
        ListBox1.Items.Add ((TObject(ListDate [I]) as TDate).Text);  
end;
```

**FIGURE 5.4:**

The list of dates shown by the ListDemo example



At the end of the code above, before we can do an `as` downcast, we first need to hard-cast the pointer returned by the `TList` into a `TObject` reference. This kind of expression can result in an invalid typecast exception, or it can generate a memory error when the pointer is not a reference to an object.

To demonstrate that things can indeed go wrong, I've added one more button, which adds a `TButton` object to the list:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);  
begin  
    // add a button to the list  
    ListDate.Add (Sender);  
end;
```

If you click this button and then update one of the lists, you'll get an error. Finally, remember that when you destroy a list of objects, you should remember to destroy all of the objects of the list first. The `ListDemo` program does this in the `FormDestroy` method of the form:

```
procedure TForm1.FormDestroy(Sender: TObject);  
var  
    I: Integer;  
begin  
    for I := 0 to ListDate.Count - 1 do  
        TObject(ListDate [I]).Free;  
    ListDate.Free;  
end;
```

## Collections

The second group, collections, contains only two classes, `TCollection` and `TCollectionItem`. `TCollection` defines a homogeneous list of objects, which are owned by the collection class. The objects in the collection must be descendants of the `TCollectionItem` class. If you need a collection storing specific objects, you have to create both a subclass of `TCollection` and a matching subclass of `TCollectionItem`.

Collections are used to specify values of properties of components. It is very unusual to work with collections for storing your own objects, so I won't discuss them here.

## Container Classes

Delphi 5 introduced a new series of container classes, defined in the `Contnrs` unit. Delphi 6 extends these classes by adding hashed associative lists, as discussed in the following section. The container classes extend the `TList` classes by adding the idea of ownership and by defining specific extraction rules (mimicking stacks and queues) or sorting capabilities.

The basic difference between `TList` and the new `TObjectList` class, for example, is that the latter is defined as a list of `TObject` objects, not a list of pointers. Even more important, however, is the fact that if the object list has the `OwnsObjects` property set to `True`, it automati-

cally deletes an object when it is replaced by another one and deletes each object when the list itself is destroyed. Here's a list of all the new container classes:

- The `TObjectList` class I've already described represents a list of objects, eventually owned by the list itself.
- The inherited class `TComponentList` represents a list of components, with full support for destruction notification (an important safety feature when two components are connected using their properties; that is, when a component is the value of a property of another component).
- The `TClassList` class is a list of class references. It inherits from `TList` and requires no destruction.
- The classes `TStack` and `TObjectStack` represent lists of pointers and objects, from which you can only extract elements starting from the last one you've inserted. A stack follows the LIFO order (Last In, First Out). The typical methods of a stack are `Push` for insertion, `Pop` for extraction, and `Peek` to preview the first item without removing it. You can still use all the methods of the base class, `TList`.
- The classes `TQueue` and `TObjectQueue` represent lists of pointers and objects, from which you always remove the *first* item you've inserted (FIFO: first in, first out). The methods of these classes are the same as those of the stack classes but behave differently.

**WARNING**

Unlike the `TObjectList`, the `TObjectStack` and the `TObjectQueue` do not own the inserted objects and will not destroy those objects left in the data structure when it is destroyed. You can simply `Pop` all the items, destroy them once you're finished using them, and then destroy the container.

To demonstrate the use of these classes, I've modified the earlier `ListDate` example into the new `Contain` example on the CD. First, I changed the type of the `ListDate` variable to `TObjectList`. In the `FormCreate` method, I've modified the list creation to the following code, which activates the list ownership:

```
ListDate := TObjectList.Create (True);
```

At this point, we can simplify the destruction code, as applying `Free` to the list will automatically free the dates it holds.

I've also added to the program a stack and a queue object, filling each of them with numbers. One of the form's two buttons displays a list of the numbers in each container, and the other removes the last item (displayed in a message box):

```
procedure TForm1.btnQueueClick(Sender: TObject);  
var  
    I: Integer;
```



```
begin
  ListBox1.Clear;
  for I := 0 to Stack.Count - 1 do begin
    ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));
    Queue.Push(Queue.Pop);
  end;
  ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));
end;
```

By pressing the two buttons, you can see that calling `Pop` for each container returns the last item. The difference is that the `TQueue` class inserts elements at the beginning, and the `TStack` class inserts them at the end.

## Hashed Associative Lists

After whetting our appetite in Delphi 5, Borland has pushed the idea of container classes a little further in Delphi 6, introducing a new set of lists, particularly `TBucketList` and `TObjectBucketList`. These two lists are associative, which means they have a key and an actual entry. The key is used to identify the items and search for them. To add an item, you call the `Add` method, with two parameters, the key and the actual data. When you use the `Find` method, you pass the key and retrieve the data. The same effect is achieved by using the `Data` array property, passing the key as parameter.

These lists are also based on a hash system. The lists create an internal array of items, called buckets, each having a sub-list of actual elements of the list. As you add an item, its key value is used to compute the *hash* value, which determines the bucket to add the item to. When searching the item, the hash is computed again, and the list immediately grabs the sublist containing the item, searching for it there. This makes for very fast insertion and searches, but only if the hash algorithm distributes the items evenly among the various buckets and if there are enough different entries in the array. In fact, when many elements can be in the same bucket, searching gets slower.

For this reason, as you create the `TObjectBucketList` you can specify the number of entries for the list, using the parameter of the constructor, choosing a value between 2 and 256. The value of the bucket is determined by taking the first byte of the pointer (or number) passed as key and doing an `and` operation with a number corresponding to the entries.

### NOTE

I don't find this algorithm very convincing for a hash system, but replacing it with your own implies only overriding the `BucketFor` virtual function and eventually changing the number of entries in the array, by setting a different value for the `BucketCount` property.

Another interesting feature, not available for lists, is the `ForEach` method, which allows you to execute a given function on each item contained in the list. You pass to the `ForEach` method a pointer to data of your own and a procedure, which receives four parameters,

including your custom pointer, each key and object of the list, and a Boolean parameter you can set to False to stop the execution. In other words, these are the two signatures:

```
type
  TBucketProc = procedure(AInfo, AItem, AData: Pointer;
    out AContinue: Boolean);

function TCustomBucketList.ForEach(AProc: TBucketProc;
  AInfo: Pointer): Boolean;
```

**NOTE**

Besides these containers, Delphi includes also a `THashedStringList` class, which inherits from `TStringList`. This class has no direct relationship with the hashed lists and is even defined in a different unit, `IniFiles`. The hashed string list has two associated hash tables (of type `TStringHash`), which are completely refreshed every time the content of the string list changes. So this class is useful only for reading a large set of fixed strings, not for handling a list of strings changing often over time. On the other hand, the `TStringHash` support class seems to be quite useful in general cases, and has a good algorithm for computing the hash value of a string.

## Type-Safe Containers and Lists

Containers and lists have a problem: They are not type-safe, as I've shown in both examples by adding a button object to a list of dates. To ensure that the data in a list is homogenous, you can check the type of the data you extract before you insert it, but as an extra safety measure you might also want to check the type of the data while extracting it. However, adding run-time type checking slows down a program and is risky—a programmer might fail to check the type in some cases.

To solve both problems, you can create specific list classes for given data types and fashion the code from the existing `TList` or `TObjectList` classes (or another container class). There are two approaches to accomplish this:

- Derive a new class from the list class and customize the `Add` method and the access methods, which relate to the `Items` property. This is also the approach used by Borland for the container classes, which all derive from `TList`.

**NOTE**

Delphi container classes use static overrides to perform simple type conveniences (parameters and function results of the desired type). Static overrides are not the same as polymorphism; someone using a container class via a `TList` variable will not be calling the container's specialized functions. Static override is a simple and effective technique, but it has one very important restriction: The methods in the descendent should not do anything beyond simple type-casting, because you aren't guaranteed that the descendent methods will be called. The list might be accessed and manipulated using the ancestor methods as much as by the descendent methods, so their actual operations must be identical. The only difference is the type used in the descendent methods, which allows you to avoid extra typecasting.

- Create a brand-new class that contains a `TList` object, and map the methods of the new class to the internal list using proper type checking. This approach defines a wrapper class, a class that “wraps” around an existing one to provide a different or limited access to its methods (in our case, to perform a type conversion).

I’ve implemented both solutions in the `DateList` example, which defines lists of `TDate` objects. In the code that follows, you’ll find the declaration of the two classes, the inheritance-based `TDateListI` class and the wrapper class `TDateListW`.

```

type
// inheritance-based
TDateListI = class (TObjectList)
protected
    procedure SetObject (Index: Integer; Item: TDate);
    function GetObject (Index: Integer): TDate;
public
    function Add (Obj: TDate): Integer;
    procedure Insert (Index: Integer; Obj: TDate);
    property Objects [Index: Integer]: TDate
        read GetObject write SetObject; default;
end;

// wrapper based
TDateListW = class(TObject)
private
    FList: TObjectList;
    function GetObject (Index: Integer): TDate;
    procedure SetObject (Index: Integer; Obj: TDate);
    function GetCount: Integer;
public
    constructor Create;
    destructor Destroy; override;
    function Add (Obj: TDate): Integer;
    function Remove (Obj: TDate): Integer;
    function IndexOf (Obj: TDate): Integer;
    property Count: Integer read GetCount;
    property Objects [Index: Integer]: TDate
        read GetObject write SetObject; default;
end;

```

Obviously, the first class is simpler to write—it has fewer methods, and they simply call the inherited ones. The good thing is that a `TDateListI` object can be passed to parameters expecting a `TList`. The problem is that the code that manipulates an instance of this list via a generic `TList` variable will not be calling the specialized methods, because they are not virtual and might end up adding to the list objects of other data types.

Instead, if you decide not to use inheritance, you end up writing a lot of code, because you need to reproduce each and every one of the original `TList` methods, simply calling the methods of the internal `FList` object. The drawback is that the `TDateListW` class is not type compatible with `TList`, which limits its usefulness. It can't be passed as parameter to methods expecting a `TList`.

Both of these approaches provide good type checking. After you've created an instance of one of these list classes, you can add only objects of the appropriate type, and the objects you extract will naturally be of the correct type. This is demonstrated by the `DateList` example. This program has a few buttons, a combo box to let a user choose which of the lists to show, and a list box to show the actual values of the list. The program stretches the lists by trying to add a button to the list of `TDate` objects. To add an object of a different type to the `TDateListI` list, we can simply convert the list to its base class, `TList`. This might accidentally happen if you pass the list as a parameter to a method that expects a base class object. In contrast, for the `TDateListW` list to fail we must explicitly cast the object to `TDate` before inserting it, something a programmer should never do:

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);  
begin  
    ListW.Add (TDate(TButton.Create (nil)));  
    TList(ListI).Add (TButton.Create (nil));  
    UpdateList;  
end;
```

The `UpdateList` call triggers an exception, displayed directly in the list box, because I've used an `as` typecast in the custom list classes. A wise programmer should never write the above code. To summarize, writing a custom list for a specific type makes a program much more robust. Writing a wrapper list instead of one that's based on inheritance tends to be a little safer, although it requires more coding.

**NOTE**

Instead of rewriting wrapper-style list classes for different types, you can use my List Template Wizard, available on my Web site, [www.marcocantu.com](http://www.marcocantu.com).

## Streaming

Another core area of the Delphi class library is its support for streaming, which includes file management, memory, sockets, and other sources of information arranged in a sequence. The idea of streaming is that you move along the data while reading it, much like the traditional `read` and `write` functions used by the Pascal language (and discussed in Chapter 12 of *Essential Pascal*, available on the companion CD).

## The *TStream* Class

The VCL defines the abstract *TStream* class and several subclasses. The parent class, *TStream*, has just a few properties, and you'll never create an instance of it, but it has an interesting list of methods you'll generally use when working with derived stream classes.

The *TStream* class defines two properties, *Size* and *Position*. All stream objects have a specific size (which generally grows if you write something after the end of the stream), and you must specify a position within the stream where you want to either read or write information.

Reading and writing bytes depends on the actual stream class you are using, but in both cases you don't need to know much more than the size of the stream and your relative position in the stream to read or write data. In fact, that's one of the advantages of using streams. The basic interface remains the same whether you're manipulating a disk file, a binary large object (BLOB) field, or a long sequence of bytes in memory.

In addition to the *Size* and *Position* properties, the *TStream* class also defines several important methods, most of which are virtual and abstract. (In other words, the *TStream* class doesn't define what these methods do; therefore, derived classes are responsible for implementing them.) Some of these methods are important only in the context of reading or writing components within a stream (for instance, *ReadComponent* and *WriteComponent*), but some are useful in other contexts, too. In Listing 5.2, you can find the declaration of the *TStream* class, extracted from the *Classes* unit.



### Listing 5.2: The public portion of the definition of the *TStream* class

```
TStream = class(TObject)
public
    // read and write a buffer
    function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
    function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
    procedure ReadBuffer(var Buffer; Count: Longint);
    procedure WriteBuffer(const Buffer; Count: Longint);

    // move to a specific position
    function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;
    function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
        overload; virtual;

    // copy the stream
    function CopyFrom(Source: TStream; Count: Int64): Int64;

    // read or write a component
    function ReadComponent(Instance: TComponent): TComponent;
    function ReadComponentRes(Instance: TComponent): TComponent;
    procedure WriteComponent(Instance: TComponent);
    procedure WriteComponentRes(const ResName: string; Instance: TComponent);
```

```
procedure WriteDescendent(Instance, Ancestor: TComponent);
procedure WriteDescendentRes(
  const ResName: string; Instance, Ancestor: TComponent);
procedure WriteResourceHeader(const ResName: string; out FixupInfo: Integer);
procedure FixupResourceHeader(FixupInfo: Integer);
procedure ReadResHeader;

// properties
property Position: Int64 read GetPosition write SetPosition;
property Size: Int64 read GetSize write SetSize64;
end;
```

---

The basic use of a string involves calling the `ReadBuffer` and `WriteBuffer` methods, which are very powerful but not terribly easy to use. The first parameter, in fact, is an untyped buffer in which you can pass the variable to save from or load to. For example, you can save into a file a number (in binary format) and a string, with this code:

```
var
  stream: TStream;
  n: integer;
  str: string;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create('c:\tmp\test', fmCreate);
  stream.WriteBuffer(n, sizeof(integer));
  stream.WriteBuffer(str[1], Length(str));
  stream.Free;
```

A totally alternative approach is to let specific components save or load data to and from streams. Many VCL classes define a `LoadFromStream` or a `SaveToStream` method, including `TStrings`, `TStringList`, `TBlobField`, `TMemoField`, `TIcon`, and `TBitmap`.

## Specific Stream Classes

Creating a `TStream` instance makes no sense, because this class is abstract and provides no direct support for saving data. Instead, you can use one of the derived classes to load data from or store it to an actual file, a BLOB field, a socket, or a memory block. Use `TFileStream` when you want to work with a file, passing the filename and some file access options to the `Create` method. Use `TMemoryStream` to manipulate a stream in memory and not an actual file.

Several units define `TStream`-derived classes. In the `Classes` unit are the following classes:

- `THandleStream` defines a stream that manipulates a disk file represented by a Windows file handle.

- `TFileStream` defines a stream that manipulates a disk file (a file that exists on a local or network disk) represented by a filename. It inherits from `THandleStream`.
- `TCustomMemoryStream` is the base class for streams stored in memory but is not used directly.
- `TMemoryStream` defines a stream that manipulates a sequence of bytes in memory. It inherits from `TCustomMemoryStream`.
- `TStringStream` provides a simple way for associating a stream to a string in memory, so that you can access the string with the `TStream` interface and also copy the string to and from another stream.
- `TResourceStream` defines a stream that manipulates a sequence of bytes in memory, and provides read-only access to resource data linked into the executable file of an application (an example of these resource data are the DFM files). It inherits from `TCustomMemoryStream`.

Stream classes defined in other units include

- `TBlobStream` defines a stream that provides simple access to database BLOB fields. There are similar BLOB streams for other database access technologies rather than the BDE.
- `TOLEStream` defines a stream for reading and writing information over the interface for streaming provided by an OLE object.
- `TWinSocketStream` provides streaming support for a socket connection.

## Using File Streams

Creating and using a file stream can be as simple as creating a variable of a type that descends from `TStream` and calling components methods to load content from the file:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
    begin
      S := TFileStream.Create (OpenDialog1.FileName, fmOpenRead);
      try
        Memo1.Lines.LoadFromStream (S);
      finally
        S.Free;
      end;
    end;
end;
```

As you can see in this code, the `Create` method for file streams has two parameters: the name of the file and a flag indicating the requested access mode. In this case, we want to read the file, so we used the `fmOpenRead` flag (other available flags are documented in the Delphi help).

**NOTE**

Of the different modes, the most important are `fmShareDenyWrite`, which you'll use when you're simply reading data from a shared file, and `fmShareExclusive`, which you'll use when you're writing data to a shared file.

A big advantage of streams over other file access techniques is that they're very interchangeable, so you can work with memory streams and then save them to a file, or you can perform the opposite operations. This might be a way to improve the speed of a file-intensive program. Here is a snippet of code, a file-copying function, to give you another idea of how you can use streams:

```
procedure CopyFile (SourceName, TargetName: String);  
var  
    Stream1, Stream2: TFileStream;  
begin  
    Stream1 := TFileStream.Create (SourceName, fmOpenRead);  
    try  
        Stream2 := TFileStream.Create (TargetName, fmOpenWrite or fmCreate);  
        try  
            Stream2.CopyFrom (Stream1, Stream1.Size);  
        finally  
            Stream2.Free;  
        end  
    finally  
        Stream1.Free;  
    end  
end;
```

Another important use of streams is to handle database BLOB fields or other large fields directly. In fact, you can export such data to a stream or read it from one by simply calling the `SaveToStream` and `LoadFromStream` methods of the `TBlobField` class.

## The *TReader* and *TWriter* Classes

By themselves, the stream classes of VCL don't provide much support for reading or writing data. In fact, stream classes don't implement much beyond simply reading and writing blocks of data. If you want to load or save specific data types in a stream (and don't want to perform a great deal of typecasting), you can use the `TReader` and `TWriter` classes, which derive from the generic `TFile` class.



Basically, the TReader and TWriter classes exist to simplify loading and saving stream data according to its type, and not just as a sequence of bytes. To do this, TWriter embeds special signatures into the stream that specify the type for each object's data. Conversely, the TReader class reads these signatures from the stream, creates the appropriate objects, and then initializes those objects using the subsequent data from the stream.

For example, I could have written out a number and a string to a stream by writing:

```
var
  stream: TStream;
  n: integer;
  str: string;
  w: TWriter;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test.txt', fmCreate);
  w := TWriter.Create (stream, 1024);
  w.WriteInteger (n);
  w.WriteString (str);
  w.Free;
  stream.Free;
```

This time the actual file will include also the extra signature characters, so that I can read back this file only by using a TReader object. For this reason, using the TReader and TWriter is generally confined to components streaming and is seldom applied in general file management.

## Streams and Persistency

In Delphi, streams play a considerable role for persistency. For this reason, many methods of TStream relate to saving and loading a component and its subcomponents. For example, you can store a form in a stream by writing

```
stream.WriteComponent(Form1);
```

If you examine the structure of a Delphi DFM file, you'll discover that it's really just a resource file that contains a custom format resource. Inside this resource, you'll find the component information for the form or data module and for each of the components it contains. As you would expect, the stream classes provide two methods to read and write this custom resource data for components: `WriteComponentRes` to store the data, and `ReadComponentRes` to load it.

For your experiment in memory (not involving actual DFM files), though, using `WriteComponent` is generally better suited. After you create a memory stream and save the current form to it, the problem is how to display it. This can be accomplished by transforming the binary representation of forms to a textual representation. Even though the Delphi

IDE, since version 5, can save DFM files in text format, the representation used internally for the compiled code is invariably a binary format.

The form conversion can be accomplished by the IDE, generally with the View as Text command of the form designer, and in other ways. There is also a command-line utility, CONVERT.EXE, found in the Delphi Bin directory. Within your own code, the standard way to obtain a conversion is to call the specific methods of VCL. There are four functions for converting to and from the internal object format obtained by the WriteComponent method:

```

procedure ObjectBinaryToText(Input, Output: TStream); overload;
procedure ObjectBinaryToText(Input, Output: TStream;
    var OriginalFormat: TStreamOriginalFormat); overload;
procedure ObjectTextToBinary(Input, Output: TStream); overload;
procedure ObjectTextToBinary(Input, Output: TStream;
    var OriginalFormat: TStreamOriginalFormat); overload;

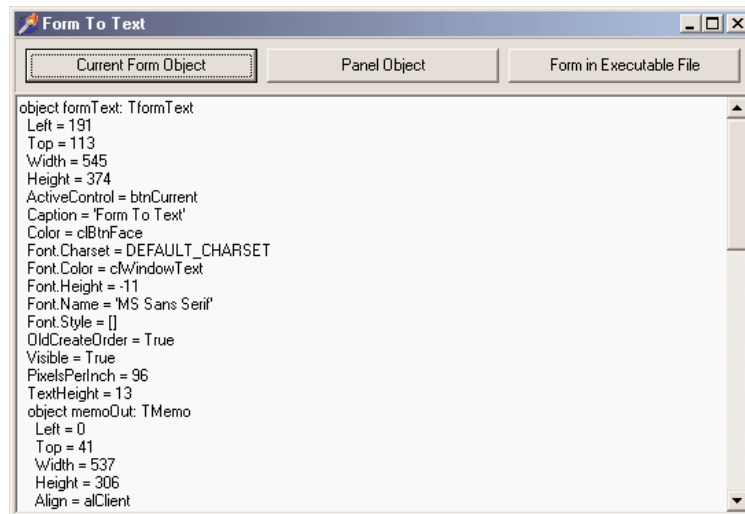
```

Four different functions, with the same parameters and names containing the name *Resource* instead of *Binary* (as in ObjectResourceToText), convert the resource format obtained by WriteComponentRes. A final method, TestStreamFormat, indicates whether a DFM is storing a binary or textual representation.

In the FormToText program, I've used the ObjectBinaryToText method to copy the binary definition of a form into another stream, and then I've displayed the resulting stream in a memo, as you can see in Figure 5.5. This is the code of the two methods involved:

**FIGURE 5.5:**

The textual description of a form component, displayed inside itself by the FormToText example



```
procedure TFormText.btnCurrentClick(Sender: TObject);  
var  
    MemStr: TStream;  
begin  
    MemStr := TMemoryStream.Create;  
    try  
        MemStr.WriteComponent (Self);  
        ConvertAndShow (MemStr);  
    finally  
        MemStr.Free  
    end;  
end;  
  
procedure TFormText.ConvertAndShow (aStream: TStream);  
var  
    ConvStream: TStream;  
begin  
    aStream.Position := 0;  
    ConvStream := TMemoryStream.Create;  
    try  
        ObjectBinaryToText (aStream, ConvStream);  
        ConvStream.Position := 0;  
        MemoOut.Lines.LoadFromStream (ConvStream);  
    finally  
        ConvStream.Free  
    end;  
end;
```

Notice that by repeatedly clicking the Current Form Object button you'll get more and more text, and the text of the memo is included in the stream. After a few times, the entire operation will get extremely slow, so that the program seems to be hung up. In this code, we start to see some of the flexibility of using streams—we can write a generic procedure we can use to convert any stream.

---

**NOTE**

It's important to stress that after you've written data to a stream, you must explicitly seek back to the beginning (or set the `Position` property to 0) before you can use the stream further, unless you want to append data to the stream, of course.

Another button, labeled Panel Object, shows the textual representation of a specific component, the panel, passing the component to the `WriteComponent` method. The third button, Form in Executable File, does a different operation. Instead of streaming an existing object in

memory, it loads in a `TResourceStream` object the design-time representation of the form—that is, its DFM file—from the corresponding resource embedded in the executable file:

```
procedure TFormText.btnResourceClick(Sender: TObject);  
var  
    ResStr: TResourceStream;  
begin  
    ResStr := TResourceStream.Create(hInstance, 'TFORMTEXT', RT_RCDATA);  
    try  
        ConvertAndShow (ResStr);  
    finally  
        ResStr.Free  
    end;  
end;
```

By clicking the buttons in sequence (or modifying the form of the program) you can compare the form saved in the DFM file to the current run-time object.

## Writing a Custom Stream Class

Besides using the existing stream classes, Delphi programmers can write their own stream classes, and use them in place of the existing ones. To accomplish this, you need only specify how a generic block of raw data is saved and loaded, and VCL will be able to use your new class wherever you call for it. You may not need to create a brand-new stream class for working with a new type of media, but only need to customize an existing stream. In that case, all you have to do is write the proper read and write methods.

As an example, I created a class to encode and decode a generic file stream. Although this example is limited by its use of a totally dumb encoding mechanism, it fully integrates with VCL and works properly. The new stream class simply declares the two core reading and writing methods and has a property that stores a key.

```
type  
    TEncodedStream = class (TFileStream)  
private  
    FKey: Char;  
public  
    constructor Create(const FileName: string; Mode: Word);  
    function Read(var Buffer; Count: Longint): Longint; override;  
    function Write(const Buffer; Count: Longint): Longint; override;  
    property Key: Char read FKey write FKey;  
end;
```

*Continued on next page*

The value of the key is simply added to each of the bytes saved to a file, and subtracted when the data is read. Here is the complete code of the `Write` and `Read` methods, which uses pointers quite heavily:

```

constructor TEncodedStream.Create( const FileName: string; Mode: Word);
begin
    inherited Create (FileName, Mode);
    FKey := 'A'; // default
end;

function TEncodedStream.Write(const Buffer; Count: Longint): Longint;
var
    pBuf, pEnc: PChar;
    I, EncVal: Integer;
begin
    // allocate memory for the encoded buffer
    GetMem (pEnc, Count);
    try
        // use the buffer as an array of characters
        pBuf := PChar (@Buffer);
        // for every character of the buffer
        for I := 0 to Count - 1 do
            begin
                // encode the value and store it
                EncVal := ( Ord (pBuf[I]) + Ord(Key) ) mod 256;
                pEnc [I] := Chr (EncVal);
            end;
        // write the encoded buffer to the file
        Result := inherited Write (pEnc^, Count);
    finally
        FreeMem (pEnc, Count);
    end;
end;

function TEncodedStream.Read(var Buffer; Count: Longint): Longint;
var
    pBuf, pEnc: PChar;
    I, CountRead, EncVal: Integer;
begin
    // allocate memory for the encoded buffer
    GetMem (pEnc, Count);
    try

```

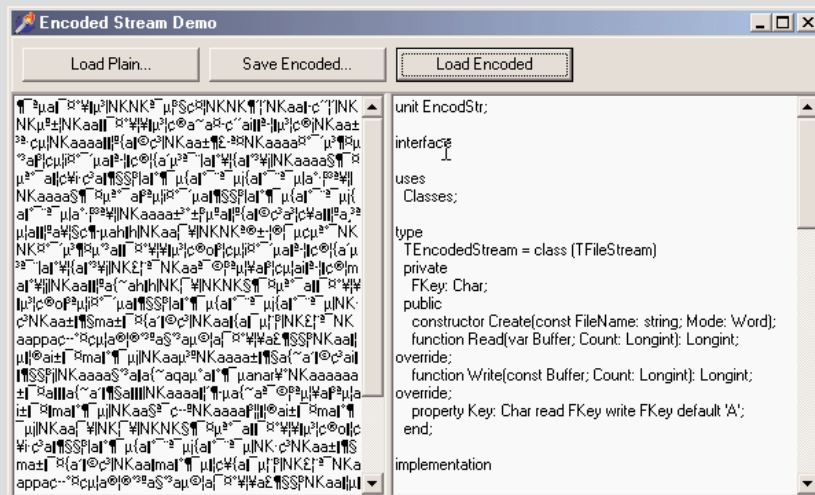
*Continued on next page*

```

// read the encoded buffer from the file
CountRead := inherited Read (pEnc^, Count);
// use the output buffer as a string
pBuf := PChar (@Buffer);
// for every character actually read
for I := 0 to CountRead - 1 do
begin
  // decode the value and store it
  EncVal := ( Ord (pEnc[I]) - Ord(Key) ) mod 256;
  pBuf [I] := Chr (EncVal);
end;
finally
  FreeMem (pEnc, Count);
end;
// return the number of characters read
Result := CountRead;
end;

```

The comments in this rather complex code should help you understand the details. Now that we have an encoded stream, we can try to use it in a demo program, which is called EncDemo. The form of this program has two memo components and three buttons, as you can see in the graphic below. The first button loads a plain text file in the first memo; the second button saves the text of this first memo in an encoded file; and the last button reloads the encoded file into the second memo, decoding it. In this example, after encoding the file, I've reloaded it in the first memo as a plain text file on the left, which of course is unreadable.



Continued on next page

Since we have the encoded stream class available, the code of this program is very similar to that of any other program using streams. For example, here is the method used to save the encoded file (you can compare its code to that of earlier examples based on streams):

```
procedure TFormEncode.BtnSaveEncodedClick(Sender: TObject);  
var  
    EncStr: TEncodedStream;  
begin  
    if SaveDialog1.Execute then  
        begin  
            EncStr := TEncodedStream.Create(SaveDialog1.FileName, fmCreate);  
            try  
                Memo1.Lines.SaveToStream (EncStr);  
            finally  
                EncStr.Free;  
            end;  
        end;  
end;  
end;
```

## Summarizing the Core VCL and BaseCLX Units

We've spent most of the space of this chapter discussing the classes of a single unit of the library, `Classes`. This unit is certainly important, but it is not the only core unit of the library (although there aren't many others). In this section, I'm providing an overview of these units and their content.

### The Classes Unit

The `Classes` unit is at the heart of both VCL and CLX libraries, and though it sees many internal changes from the last version of Delphi, there is little new for the average users. (Most changes are related to modified IDE integration and are meant for expert component writers.)

Here is a list of what you can find in the `Classes` unit, a unit that every Delphi programmer should spend some time with:

- Many enumerated types, the standard method pointer types (including `TNotifyEvent`), and many exception classes.

- Core library classes, including `TPersistent` and `TComponent` but also `TBasicAction` and `TBasicActionLink`.
- List classes, including `TList`, `TThreadList` (a thread-safe version of the list), `TInterfaceList` (a list of interfaces, used internally), `TCollection`, `TCollectionItem`, `TOwnedCollection` (which is simply a collection with an owner), `TStrings`, and `TStringList`.
- All the stream classes I discussed in the previous section but won't list here again. There are also the `TFilter`, `TReader`, and `TWriter` classes and a `TParser` class used internally for DFM parsing.
- Utility classes, such as `TBits` for binary manipulation and a few utility routines (for example, point and rectangle constructors, and string list manipulation routines such as `LineStart` and `ExtractStrings`). There are also many registration classes, to notify the system of the existence of components, classes, special utility functions you can replace, and much more.
- The `TDataModule` class, a simple object container alternative to a form. Data modules can contain only nonvisual components and are generally used in database and Web applications.

**NOTE**

In past versions of Delphi, the `TDataModule` class was defined in the Forms unit; now it has been moved to the Classes unit. This was done to eliminate the code overhead of the GUI classes from non-visual applications (for example, Web server modules) and to better separate non-portable Windows code from OS-independent classes, such as `TDataModule`. Other changes relate to the data modules, for example, to allow the creation of Web applications with multiple data modules, something not possible in Delphi 5.

- New interface-related classes, such as `TInterfacedPersistent`, aimed at providing further support for interfaces. This particular class allows Delphi code to hold onto a reference to a `TPersistent` object or any descendent implementing interfaces, and is a core element of the new support for interfaced objects in the Object Inspector (see Chapter 11 for an example).
- The new `TRecall` class, used to maintain a temporary copy of an object, particularly useful for graphical-based resources.
- The new `TClassFinder` class used for finding a registered class instead of the `FindClass` method.
- The `TThread` class, which provides the core to operating system-independent support for multithreaded applications.



## Other Core Units

Other units that are part of the RTL package are not directly used by typical Delphi programmers as often as Classes. Here is a list:

- The TypInfo unit includes support for Accessing RTTI information for published properties, as we've seen in the section "Accessing Properties by Name."
- The SyncObjs unit contains a few generic classes for thread synchronization.

Of course, the RTL package also includes the units with functions and procedures discussed in the preceding chapter, such as Math, SysUtils, Variants, VarUtils, StrUtils, DateUtils, and so on.

## What's Next?

As we have seen in this chapter, the Delphi class library has a few root classes that play a considerable role and that you should learn to leverage to the maximum possible extent. Some programmers tend to become expert on the components they use every day, and this is important, but without understanding the core classes (and ideas such as ownership and streaming), you'll have a tough time grasping the full power of Delphi.

Of course, in this book, we also need to discuss visual and database classes, which I will do in the next chapter. Now that we've seen all the base elements of Delphi (language, RTL, core classes), we are ready to discuss the development of real applications with this tool.

Part II of the book, which starts with the next chapter, is fully devoted to examples of the use of the various components, particularly visual components with the development of the user interface. We'll start with the advanced use of traditional controls and menus, discuss the actions architecture, cover the TForm class, and then examine toolbars, status bars, dialog boxes, and MDI applications in later chapters. Then we'll move to the development of database applications in Part III of the book.



**PART II**

---

# Visual Programming

- Chapter 6: Controls: VCL Versus VisualCLX
- Chapter 7: Advanced VCL Controls
- Chapter 8: Building the User Interface
- Chapter 9: Working with Forms
- Chapter 10: The Architecture of Delphi Applications
- Chapter 11: Creating Components
- Chapter 12: Libraries and Packages

# Controls: VCL Versus VisualCLX

- VCL versus VisualCLX
- TControl, TWinControl, and TWidgetControl
- An overview of the standard components
- Basic and advanced menu construction
- Modifying the system menu
- Graphics in menus and list boxes
- OwnerDraw and styles

**N**ow that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the base elements of component library, we are ready to delve into the second part of the book: the use of components and the development of the user interface of applications. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

Delphi comes with a large number of ready-to-use components. I won't describe every component in detail, examining each of its properties and methods; if you need this information, you can find it in the Help system. The aim of Part II of this book is to show you how to use some of the advanced features offered by the Delphi predefined components to build applications and to discuss specific programming techniques.

I'll start with a comparison of the VCL and VisualCLX libraries available in Delphi 6 and a coverage of the core classes (particularly `TControl`). Then I'll try to list all the various visual components you have, because choosing the right basic controls is often a way to get into a project faster.

## VCL versus VisualCLX

As we've seen in the last chapter, Delphi 6 introduces the new CLX library alongside the traditional VCL library. There are certainly many differences, even in the use of the RTL and code library classes, between developing programs specifically for Windows or with a cross-platform attitude, but the user interface portion is where differences are most striking.

The visual portion of VCL is a wrapper of the Window API. It includes wrappers of the native Windows controls (like buttons and edit boxes), of the common controls (like tree views and list views), plus a bunch of native Delphi controls bound to the Windows concept of a window. There is also a `TCanvas` class that wraps the basic graphic calls, so you can easily paint on the surface of a window.

VisualCLX, the visual portion of CLX, is a wrapper of the Qt (pronounced "cute") library. It includes wrappers of the native Qt widgets, which range from basic to advanced controls, very similar to Windows' own standard and common controls. It includes also painting support using another, similar, `TCanvas` class. Qt is a C++ class library, developed by Trolltech ([www.trolltech.com](http://www.trolltech.com)), a Norwegian company with a strong relationship with Borland.

On Linux, Qt is one of the de facto standard user-interface libraries and is the basis of the KDE desktop environment. On Windows, Qt provides an alternative to the use of the native APIs. In fact, unlike VCL, which provides a wrapper to the native controls, Qt provides an alternate implementation to those controls. This allows programs to be truly portable, as

there are no hidden differences created by the operating system (and that the operating system vendor can introduce behind the scenes). It also allows us to avoid an extra layer; CLX on top of Qt on top of Windows native controls suggests three layers, but in fact there are two layers in each solution (CLX controls on top of Qt, VCL controls on top of Windows).

**NOTE**

Distributing Qt applications on Windows implies the distribution of the Qt library itself (something you can generally take for granted on the Linux platform). Distributing the Qt libraries with a professional application (as opposed to an open source project) generally implies paying a license to Trolltech. If you use Delphi or Kylix to build Qt applications, however, Borland has already paid the license to Trolltech for you. However, you must use the CLX classes wrapping Qt: If you use the Qt classes directly, you apparently still owe the license to Qt, even when using Delphi or Kylix.

Technically, there are huge differences behind the scenes between a native Windows application built with VCL and a portable Qt program developed with VisualCLX. Suffice to say that at the low level, Windows uses API function calls and messages to communicate with controls, while Qt uses class methods and direct method callbacks and has no internal messages. Technically, the Qt classes offer a high-level object-oriented architecture, while the Windows API is still bound to its C legacy and a message-based system dated 1985 (when Windows was released). VCL offers an object-oriented abstraction on top of a low-level API, while VisualCLX remaps an already high-level interface into a more *familiar* class library.

**NOTE**

To be honest, Microsoft has apparently reached the point of starting to abandon the traditional low-level Windows API for a native high-level class library, part of the dotNet architecture. Of course, this change won't happen overnight, but new high-level user-interface technologies might be introduced only in dotNet. Actually, dotNet consists of multiple technologies, including a virtual machine or runtime interpreter, a low-level nonvisual RTL, and a class framework for visual stuff (partially overlapping with VCL). If having a new visual class library on top of the Windows API might be of little use to programmers already using a modern class library (like VCL) other areas of dotNet would be of interest to Delphi programmers. So far, Borland has released no official statement regarding possible support for the dotNet byte code and virtual machine, or other areas of the future Microsoft operating system offering.

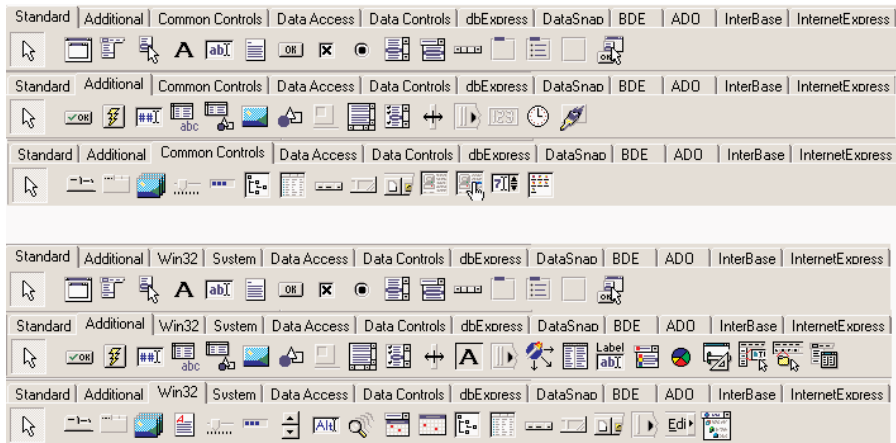
Having a familiar class library on top of a totally new platform is the advantage for Delphi programmers of using VisualCLX on Linux. This implies that the two class libraries, CLX and VCL, are very similar for their users, even if they are very different internally, as I mentioned. From the outside, a button is an object of the TButton class for both libraries, and it has more or less the same set of methods, properties, and events. In many occasions, you can recompile your existing programs for the new class library in a matter of minutes, if they don't map directly to low-level APIs.

## Delphi 6 Dual Libraries Support

Delphi 6 has full support for both libraries at design time and at run time. As you start developing a new application, you can use the File > New Application command to create a new VCL-based program and File > New CLX Application for a new CLX-based program. After giving one of these two commands, Delphi's IDE will create a VCL or CLX design-time form and update the Component Palette so that it displays only the visual components compatible with the type of application you've selected (see Figure 6.1 for a comparison). In fact, you cannot place a VCL button into a CLX form, and you cannot even mix forms of the libraries within a single executable file. In other words, the user interface of every application must be built using exclusively one of the two libraries, which (aside from the technical implications) actually makes a lot of sense to me.

**FIGURE 6.1:**

A comparison of the first three pages of the Component Palette for a CLX-based application (above) and a VCL-based application (below)



If you haven't already done so, I suggest you to try experimenting with the creation of a CLX application, looking at the available controls and trying to use them. You'll find very few differences in the use of the components, and if you have been using Delphi for some time, you'll probably be immediately adept with CLX.

### Same Classes, Different Units

One of the cornerstones of the source-code compatibility between CLX and VCL code is that fact that similar classes in the two libraries have exactly the same class name. Each library has a class called `TButton` representing a push button; the methods and properties are so similar, this code will work with both libraries:

```
with TButton.Create (Self) do
begin
  SetBounds (20, 20, 80, 35);
```

```
    Caption := 'New';  
    Parent := Self;  
end;
```

The two `TButton` classes have the same name, and this is possible because they are saved in two different units, called `StdCtrls` and `QStdCtrls`. Of course, you cannot have the two components available at design time in the palette, as the Delphi IDE can register only components with unique names. The entire VisualCLX library is defined by units corresponding to the VCL units, but with the letter *Q* as a prefix—so there is a `QForms` unit, a `QDialogs` unit, a `QGraphics` unit, and so on. There are also a few peculiar ones, such as the `QStyle` unit, that have no correspondence in VCL.

Notice that there are no compile settings or other hidden techniques to distinguish between the two libraries; what matters is the set of units referenced in the code. Remember that these references must be consistent, as you cannot mix visual controls of the two libraries in a single form and not even in a single program.

## DFM and XFM

As you create a form at design time, this is saved to a form definition file. Traditional VCL applications use the DFM extension, which stands for Delphi form module. CLX applications use the XFM extension, which stands for cross-platform (i.e., *X*) form modules. The actual format of DFM or XFM files, which can be based on a textual or binary representation, is identical. A form module is the result of streaming the form and its components, and the two libraries share the streaming code, so they produce a fairly similar effect.

So the reason for having two different extensions doesn't lie in internal compiler tricks or incompatible formats. It is merely an indication to programmers and to the IDE of the type of components you should expect to find within that definition (as this indication is *not* included in the file itself).

If you want to convert a DFM file into an XFM file, you can simply rename the file. However, expect to find some differences in the properties, events, and available components, so that reopening the form definition for a different library will probably cause quite a few warnings.

### TIP

Apparently Delphi's IDE chooses the active library only by looking at the extension of the form module, ignoring the references in the `uses` statements. For this reason, do change the extension if you plan using CLX. On Kylix, a different extension is pretty useless, because any form is opened in the IDE as a CLX form, regardless of the extension. On Linux, there is only the Qt-based CLX library, which is both the cross-platform *and* the native library.



As an example, I've built two simple identical applications, LibComp and QLibComp (available on this book's CD-ROM), with only a few components and a single event handler. Listing 6.1 presents the textual form definitions for two applications, built using the same steps in the Delphi 6 IDE, after choosing a CLX or VCL application. I've marked out differences in bold; as you can see, there are very few, most relating to the form and its font. The `OldCreateOrder` is a legacy property, used for compatibility with Delphi 3 and older code; standard colors have different names; and CLX saves the scrollbars' ranges.



### Listing 6.1: An XFM file (left) and an equivalent DFM file (right)

<pre> <b>object</b> Form1: TForm1   Left = 192   Top = 107   Width = 350   Height = 210   Caption = 'QLibComp'   <b>Color = clBackground</b>   <b>VertScrollBar.Range = 161</b>   <b>HorzScrollBar.Range = 297</b>    TextHeight = 13   <b>TextWidth = 6</b>   PixelsPerInch = 96   <b>object</b> Button1: TButton     Left = 56     Top = 64     Width = 75     Height = 25     Caption = 'Add'     TabOrder = 0     OnClick = Button1Click   <b>end</b>   <b>object</b> Edit1: TEdit     Left = 40     Top = 32     Width = 105     Height = 21     TabOrder = 1     Text = 'my name'   <b>end</b>   <b>object</b> ListBox1: TListBox     Left = 176     Top = 32     Width = 121     Height = 129     <b>Rows = 3</b>     Items.Strings = ( </pre>	<pre> <b>object</b> Form1: TForm1   Left = 192   Top = 107   Width = 350   Height = 210   Caption = 'LibComp'   <b>Color = clBtnFace</b>   <b>Font.Charset = DEFAULT_CHARSET</b>   <b>Font.Color = clWindowText</b>   <b>Font.Height = -11</b>   <b>Font.Name = 'MS Sans Serif'</b>   <b>Font.Style = []</b>   TextHeight = 13   <b>OldCreateOrder = False</b>   PixelsPerInch = 96   <b>object</b> Button1: TButton     Left = 56     Top = 64     Width = 75     Height = 25     Caption = 'Add'     TabOrder = 0     OnClick = Button1Click   <b>end</b>   <b>object</b> Edit1: TEdit     Left = 40     Top = 32     Width = 105     Height = 21     TabOrder = 1     Text = 'my name'   <b>end</b>   <b>object</b> ListBox1: TListBox     Left = 176     Top = 32     Width = 121     Height = 129     <b>ItemHeight = 13</b>     Items.Strings = ( </pre>
---	--



```
'marco'
'john'
'helen')
TabOrder = 2
end
end
```

---

```
'marco'
'john'
'helen')
TabOrder = 2
end
end
```

## uses Statements

By looking at the source code of the two examples, the differences are even less relevant, as they simply relate to the uses statements. The form of the CLX application has the following initial code:

```
unit QLibCompForm;
interface
uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;
```

The form of the VCL program has the traditional uses statement:

```
unit LibCompForm;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
```

The code of the class and of the only event handler is absolutely identical. Of course, the classic compiler directive {\$R \*.dfm} is replaced by {\$R \*.xpm} in the CLX version of the program.

## Disabling the Dual Library Help Support

In Delphi 6, when you press the F1 key in the editor asking for help on a routine, class, or method of the Delphi library, you'll usually get a choice between the VCL and CLX declarations of the same feature. You'll need to make a choice to proceed to the related help page, which can be quite annoying after a while (especially as the two pages are often identical).

If you don't care about CLX and are planning to use only VCL (or vice versa), you can disable this alternative by choosing the Help > Customize command, removing everything with *CLX* in the name from Contents, Index, and Link, and saving the project. Then restart the Delphi IDE, and the Help engine won't bother asking you about CLX any more. Of course, don't forget to add those help files again in case you decide to start using CLX.

## Choosing a Visual Library

Because you have two different user interface libraries available in Delphi 6, you'll have to choose one for each *visual* application. You must evaluate multiple criteria to come to the proper decision, which isn't always easy.

The first criterion is portability. If running your program on Windows and on Linux, with the same user interface, is a major concern to you, using CLX will probably make your life simpler and let you keep a single source code file with very limited IFDEFs. The same applies if you consider Linux to be (or possibly become) your key platform. Instead, if most of your users are on Windows and you just want to extend your offering with a Linux version, you might want to keep a dual VCL/CLX system. This probably implies two different sets of source code files, or too many for IFDEFs.

In fact, another criterion is the native look-and-feel. By using CLX on Windows, some of the controls will behave slightly differently than users will expect—at least expert users. For a simple user interface (edits, buttons, grids), this probably won't matter much, but if you have many tree view and list view controls, the differences will be quite clear. On the other hand, with CLX you'll be able to let your users select a look-and-feel of their choice, different from the basic Windows look, and use it consistently across platforms.

Using native controls implies also that as soon as you get a new version of the Windows operating system, your application will (probably) adapt to it. This is good for the user, but might cause you a lot of headaches in case of incompatibilities. Differences in the Microsoft common controls library over the last few years have been a major source of frustration for Windows programmers in general, including Delphi programmers.

Another criterion is the deployment: If you use CLX, you'll have to ship your Windows program with the Qt libraries, which are not commonly available on Windows systems.

Finally, I've done a little testing, and it seems that the speed of VCL and CLX applications is similar. I've tried creating a thousand components, showing them on screen, and the speed differences are few, with a slight advantage for the VCL-based solution. You can try them out with the LibSpeed and QLibSpeed applications on the companion CD.

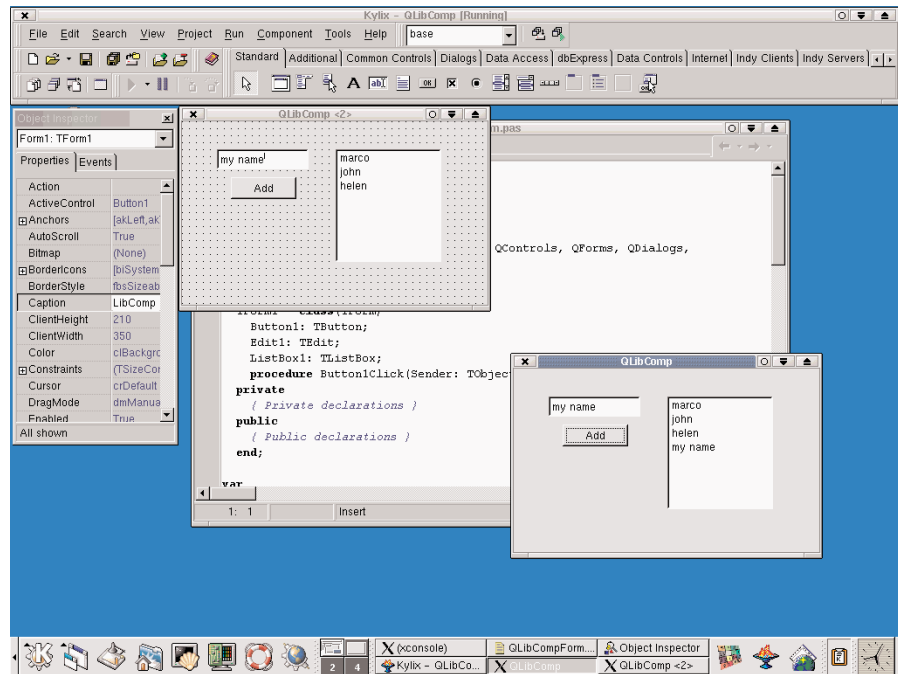
## Running It on Linux

So the real issue of choosing the library resolves to the importance of Linux for you and your users. What is very important to notice is that, if you create a CLX application, you'll be able to recompile it unchanged (with the exact source code) with Kylix producing a native Linux application.

As an example, I've recompiled the QLibComp example introduced earlier, and you can see it running in Figure 6.2, where you can also see the Kylix IDE in action on a KDE 2 SuSE system.

**FIGURE 6.2:**

An application written with CLX can be directly recompiled under Linux with Kylix (displayed in the background).



## Conditional Compilation for Libraries

If you want to keep a single source code file but compile with VCL on Windows and CLX on Linux, you can use platform-specific symbols (such as `$IFDEF LINUX`) to distinguish the two situations in case of conditional compilation. But what if you want to be able to compile a portion of code for both libraries on Windows?

You can either define a symbol of your own, and use conditional compilation, or (at times) test for the presence of identifiers that exist only in VCL or CLX only, as in:

```
{ $IF Declared(QForms) }
  ..CLX-specific code
{ $IFEND }
```

## Converting Existing Applications

Besides starting with new CLX applications, you might want to convert some of your existing VCL applications to the new class library. There are a series of operations you have to do, without any specific help from the Delphi IDE:

- You'll have to rename the DFM file as XFM and update all of the `{ $R *.DFM }` statements as `{ $R *.XFM }`.
- You'll have to update all of the uses statements of your program (in the units and project files) to refer to the CLX units instead of the VCL units. Notice that by missing even a few, you'll bump into trouble when running your application.

### TIP

To prevent a CLX application from compiling if it contains references to VCL units, you can move the VCL units to a different directory under `lib` and avoid including this folder in your search path. This way, eventual leftover references to VCL units will cause a "Unit not found" error.

Table 6.1 is a comparison of the names of the visual VCL and CLX units, excluding the database portion and some rarely referenced units:

**TABLE 6.1:** Names of Equivalent VCL and CLX Units

VCL	CLX
ActnList	QActnList
Buttons	QButtons
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	QConsts
Controls	QControls
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdCtrls	QStdCtrls

You might also convert references to Windows and Messages into references to the Qt unit. Some Windows data structures are now also available in the Types unit (see Chapter 4, “The Run-Time Library,” for details), so you might have to add it to your CLX programs. Notice, however, that the QTypes unit is not the CLX version of VCL’s Types unit; these two units are totally unrelated.

**WARNING**

Watch out for your `uses` statements! If you happen to compile a project that includes a CLX form, but fail to update the project unit, leaving a reference to the VCL Forms unit there, your program will run but stop immediately. The reason is that no VCL form was created, so the program terminated right away. In other cases, trying to create a CLX form within a VCL application will cause run-time errors. Finally, the Delphi IDE might inappropriately add references to `uses` statements of the wrong library, so you end up with a single `uses` statement referring to the same unit for both, but only the second of the two will be effective. This rarely prevents the program from compiling, but you won’t be able to run it.

## The VclToClx Helper Tool

As a helper in converting some of my own programs, I’ve written a simple unit-replacement tool, called `VclToClx` and available with its complete source code in the `Tools` folder of the book CD and on my Web site.

The program converts unit names, based on a configuration file, and fixes the DFM issue, by renaming the DFM files to XFM and fixing the references in the source code. The program is quite naive, as it doesn’t really parse the source code, but simply looks for the occurrences of the unit names followed by a comma or semicolon, as happens in a `uses` statement. It also requires that the unit name is preceded by a space, but of course you can modify the program to look for a comma. Don’t skip this extra test; otherwise the Forms unit will be turned to `QForms`, but the `QForms` unit will be converted again to `QQForms`!

## TControl and Derived Classes

In the preceding chapter, I discussed the base classes of the Delphi library, focusing particularly on the `TComponent` class. One of the most important subclasses of `TComponent` is `TControl`, which corresponds to visual components. This base class is available both in CLX and VCL and defines general concepts, such as the position and the size of the control, the parent control hosting it, and more. For an actual implementation, though, you have to refer to its two subclasses. In VCL these are `TWinControl` and `TGraphicControl`; in CLX they are `TWidgetControl` and `TGraphicControl`. Here are their key features:

- *Window-based controls* (also called *windowed controls*) are visual components based on an operating-system window. A `TWinControl` in VCL has a window handle, a number referring to an internal Windows structure. A `TWidgetControl` in CLX has a Qt handle,

a reference to the internal Qt object. From a user perspective, windowed controls can receive the input focus, and some of them can contain other controls. This is the biggest group of components in the Delphi library. We can further divide windowed controls in two groups: wrappers of native controls of Windows or Qt, and custom controls, which generally inherit from `TControl`.

- *Graphical controls* (also called *nonwindowed controls*) are visual components that are not based on an operating-system window. Therefore, they have no handle, cannot receive the focus, and cannot contain other controls. These controls inherit from `TGraphicControl` and are painted by their parent form, which sends them mouse-related and other events. Examples of nonwindowed controls are the `Label` and `SpeedButton` components. There are just a few controls in this group, which were critical to minimizing the use of system resources in the early days of Delphi (on 16-bit Windows). Using graphical controls to save Windows resources is still quite useful on Win9x/Me, which has pushed the system limits higher but hasn't fully gotten rid of them (unlike Windows NT/2000).

## A Short History of Windows Controls

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window that has a specific behavior and some styles and is capable of responding to specific messages. These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components. (Actually, Microsoft's third step was its ActiveX technology, which is now followed by the dotNet framework, which is more or less at the level of the VCL controls.)

Windows 3.1 had six kinds of predefined controls, which were generally used in dialog boxes. Still used in Win32, they are buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Windows 95 added new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. Win32 developers can use the standard common controls provided by the system, and Delphi developers have the further advantage of having corresponding easy-to-use components.

As we have seen, Qt offers to CLX comparable basic and common controls, and even if there are internal differences, the Delphi libraries exposing those controls provide wrappers that can minimize those differences. VCL, in fact, literally wraps Windows predefined controls in some of its basic components. A Delphi wrapper class—for example, `TEdit`—simply surfaces the capabilities of the underlying Windows control, making it easier to use. However, Delphi adds nothing to the capabilities of this control. In Windows 95/98, an edit or memo control has a physical limit of 32 KB of text, and this limit is retained by the Delphi component.

*Continued on next page*

Why hasn't Borland overcome this limit? Why can't we change the color of a button? Simply because by replacing a Windows control with a custom version, we would lose the close connection with the operating system. Suppose Microsoft improves some of the controls in the next version of Windows. If we use our own version of the component, the application we build won't have the new features. By using controls that are based on the operating-system capabilities, instead, our programs have the opportunity to migrate through different versions of the OS and retain all the features provided by the specific version. This doesn't apply to the use of Qt, of course, but you have the advantage of being able to have an identical application based on the same source code running on Linux.

Note that wrapping an existing Windows or Qt control is an effective way of reusing code and also helps reduce the size of your compiled program. Implementing yet another button control from scratch requires custom code in your application, while a wrapper around the OS-supplied button control requires less code and makes use of system code shared by many applications.

## Parent and Controls

The Parent property of a control indicates which other control is responsible for displaying it. When you drop a component into a form in the Form Designer, the form will become both parent and owner of the new control. But if you drop the component inside a Panel, ScrollBox, or any other *container* component, this will become its parent, while the form will still be the owner of the control.

When you create the control at run time, you'll need to set the owner (using the Create constructor parameter); but you must also set the Parent property, or the control won't be visible.

Like the Owner property, the Parent property has an inverse. The Controls array, in fact, lists all of the controls parented by the current one, numbered from 0 to ControlsCount - 1. You can scan this property to operate on all of the controls hosted by another one, eventually using a recursive method that operates on the controls parented by each subcontrol.

## Properties Related to Control Size and Position

Some of the properties introduced by TControl and common to all controls are those related to size and position. The position of a control is determined by its Left and Top properties, its size by the Height and Width properties. Technically, all components have a position, because when you reopen an existing form at design time, you want to be able to see the icons for the nonvisual components in exactly the position where you've placed them. This position is visible in the form file.

**TIP**

As you change any of the positional or size properties, you end up calling the single `SetBounds` method. So any time you need to change two or more of these properties at once, calling `SetBounds` directly will speed up the program. Another method, `BoundsRect`, returns the rectangle bounding of the control and corresponds to accessing those four properties.

An important feature of the position of a component is that, like any other coordinate, it always relates to the client area of its parent component (indicated by its `Parent` property). For a form, the client area is the surface included within its borders (excluding the borders themselves). It would have been messy to work in screen coordinates, although there are some ready-to-use methods that convert the coordinates between the form and the screen and vice versa.

Note, however, that the coordinates of a control are always relative to the parent control, such as a form or another *container* component. If you place a panel in a form, and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In fact, in this case, the parent component of the button is the panel.

## Activation and Visibility Properties

There are two basic properties you can use to let the user activate or hide a component. The simpler is the `Enabled` property. When a component is disabled (when `Enabled` is set to `False`), usually some visual hint indicates this state to the user. At design time, the “disabled” property does not always have an effect, but at run time, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding `Hide` method or by setting its `Visible` property to `False`. Be aware, however, that reading the status of the `Visible` property does not tell you whether the control is actually visible. In fact, if the container of a control is hidden, even if the control is set to `Visible`, you cannot see it. For this reason, there is another property, `Showing`, which is a run-time and read-only property. You can read the value of `Showing` to know whether the control is really visible to the user; that is, if it is visible, its parent control is also visible, the parent control of the parent control is also visible, and so on.

## Fonts

Two properties often used to customize the user interface of a component are `Color` and `Font`. Several properties are related to the color. The `Color` property itself usually refers to the background color of the component. Also, there is a `Color` property for fonts and many other graphic elements. Many components also have a `ParentColor` and a `ParentFont` property, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the `Font` property of the form itself.



When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Delphi allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users' computers.

If your program uses a font that your user doesn't have, Windows will select some other font to use in its place. A program's carefully formatted output can be ruined by the font substitution. For this reason, you should probably rely only on standard Windows fonts (such as MS Sans Serif, System, Arial, Times New Roman, and so on).

## Colors

There are various ways to set the value of a color. The type of this property is `TColor`. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include `clBlue`, `clSilver`, `clWhite`, `clGreen`, `clRed`, and many others.

### Tip

Delphi 6 adds four new standard colors: `clMoneyGreen`, `clSkyBlue`, `clCream`, and `clMedGray`.

As a better alternative, you can use one of the colors used by the system to denote the status of given elements. These sets of colors are different in VCL and CLX. VCL includes predefined Windows colors such as the background of a window (`clWindow`), the color of the text of a highlighted menu (`clHighlightText`), the active caption (`clActiveCaption`), and the ubiquitous button face color (`clBtnFace`).

CLX includes a different and incompatible set of system colors, including `clBackground`, which is the standard color of a form; `clBase`, used by edit boxes and other visual controls; `clActiveForeground`, the foreground color for active controls; and `clDisabledBase`, the background color for disabled text controls. All the color constants mentioned here are listed in VCL and CLX Help files under the "TColor type" topic.

Another option is to specify a `TColor` as a number (a 4-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low three bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value `$00FF0000` corresponds to a pure blue color, the value `$0000FF00` to green, the value `$000000FF` to red, the value `$00000000` to black, and the value `$00FFFFFF` to white. By specifying intermediate values, you can obtain any of 16 million possible colors.

Instead of specifying these hexadecimal values directly, you should use the Windows RGB function, which has three parameters, all ranging from 0 to 255. The first indicates the amount of red, the second the amount of green, and the last the amount of blue. Using the RGB function makes programs generally more readable than using a single hexadecimal

constant. Actually, RGB is *almost* a Windows API function. It is defined by the Windows-related units and not by Delphi units, but a similar function does not exist in the Windows API. In C, there is a macro that has the same name and effect, so this is a welcome addition to the Pascal interface to Windows. RGB is not available on CLX, so I've written my own version as:

```
function RGB (red, green, blue: Byte): Cardinal;  
begin  
    Result := blue + green * 256 + red * 256 * 256;  
end;
```

The highest-order byte of the TColor type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.) Regarding palettes and color matching, note that Windows sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, Windows uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors. In 16-color (VGA) adapters and at higher resolutions, you often end up seeing strange patterns of pixels of different colors and not the color you had in mind.

## The *TWinControl* Class (VCL)

In Windows, most elements of the user interface are windows. From a user standpoint, a window is a portion of the screen surrounded by a border, having a caption and usually a system menu. But technically speaking, a window is an entry in an internal system table, often corresponding to an element visible on the screen that has some associated code. Most of these windows have the role of controls; others are temporarily created by the system (for example, to show a pull-down menu). Still other windows are created by the application but remain hidden from the user and are used only as a way to receive a message (for example, nonblocking sockets use windows to communicate with the system).

The common denominator of all windows is that they are known by the Windows system and refer to a function for their behavior; each time something happens in the system, a notification message is sent to the proper window, which responds by executing some code. Each window of the system, in fact, has an associated function (generally called its *window procedure*), which handles the various messages the window is interested in.

In Delphi, any TWinControl class can override the WndProc method or define a new value for the WindowProc property. Interesting Windows messages, however, can be better tracked by providing specific message handlers. Even better, VCL converts these lower-level messages into events. In short, Delphi allows us to work at a high level, making application development easier, but still allows us to go low-level when this is required.

Notice also that creating a `WinControl` doesn't automatically create its corresponding `Window` handle. Delphi, in fact, uses a lazy initialization technique, so that the low control is only created when this is required, generally as soon as a method accesses the `Handle` property. The get method for this property the first time calls `HandleNeeded`, which eventually calls `CreateHandle...` and so on reaching `CreateWnd`, `CreateParams`, and `CreateWindowHandle` (the sequence is rather complex, and I don't think it is necessary to know it in detail). At the opposite end, you can keep an existing (perhaps invisible) control in memory but destroy its window handle, to save system resources.

## The *TWidgetControl* Class (CLX)

In CLX, every `TWidgetControl` has an internal Qt object, referenced using the `Handle` property. This property has the same name as the corresponding Windows property, but it is totally different behind the scenes.

The Qt object is generally owned by the `TWidgetControl`, which automatically frees the object when it is destroyed. The class also uses delayed construction, as you can see in the `InitWidget` method, similar to `CreateWindow`. However it is also possible to create a widget around an existing Qt object: in this case, the widget won't own the Qt object and won't destroy it. The behavior is indicated by the `OwnHandle` property.

Actually each `VisualCLX` component has two associated C++ objects, the Qt `Handle` and the Qt `Hook`, which is the object receiving the system events. With the current Qt design, this has to be a C++ object, which acts as an intermediary to the event handlers of the Object Pascal control. The `HookEvents` method associates the hook object to the CLX control.

Differently from Windows, Qt defines two different types of events:

- *Events* are the translation of input or system events (such as key press, mouse move, and paint).
- *Signals* are internal component events (corresponding to VCL internal or abstract operations, such as `OnClick` and `OnChange`)

### NOTE

In CLX there is a seldom-used `EventHandlerer` method, which corresponds more or less to the `WndProc` method of VCL.

## Opening the Component Tool Box

So you want to write a Delphi application. You open a new Delphi project and find yourself faced with a large number of components. The problem is that for every operation, there are multiple alternatives. For example, you can show a list of values using a list box, a combo box,

a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which should you use? That's difficult to say. There are many considerations, depending on what you want your application to do. For this reason, I've provided a highly condensed summary of alternative options for a few common tasks.

**NOTE**

For some of the controls described in the following sections, Delphi also includes a data-aware version, usually indicated by the *DB* prefix. As you'll see in Chapter 13, "Delphi's Database Architecture," the DB version of a control typically serves a role similar to that of its "standard" equivalent; but the properties and the ways you use it are often quite different. For example, in an Edit control you use the `Text` property, while in a DBEdit component you access the `Value` of the related field object.

## The Text Input Components

Although a form or component can handle keyboard input directly, using the `OnKeyPress` event, this isn't a common operation. Windows provides ready-to-use controls you can use to get string input and even build a simple text editor. Delphi has several slightly different components in this area.

### The Edit Component

The Edit component allows the user to enter a single line of text. You can also display a single line of text with a `Label` or a `StaticText` control, but these components are generally used only for fixed text or program-generated output, not for input. In CLX, there is also a native LCD digit control you can use to display numbers.

The Edit component uses the `Text` property, whereas many other controls use the `Caption` property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the `OnKeyPress` event of the edit box. For example, we can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's not, we change the value of the key to the null character (#0), so that it won't be processed by the edit control and will produce a warning beep:

```
procedure TForm1.Edit1KeyPress(  
    Sender: TObject; var Key: Char);  
begin  
    // check if the key is a number or backspace  
    if not (Key in ['0'..'9', #8]) then  
        begin  
            Key := #0;  
            Beep;  
        end;  
end;  
end;
```

**NOTE**

A minor difference of CLX is that the Edit control has no Undo mechanism built in. Another is that the `PasswordChar` property is *replaced* by the `EchoMode` property. You don't determine the character to display, but whether to echo the entered text or display an asterisk instead.

## The New LabeledEdit Control

Delphi 6 adds a very nice control, called `LabeledEdit`, which is an Edit control with a label attached to it. The Label appears as a property of the compound control, which inherits from `TCustomEdit`.

I have to say this component is very handy, because it allows you to reduce the number of components on your forms, move them around more easily, and have a more standard layout for labels, particularly when they are placed above the edit box. The `EditLabel` property is connected with the subcomponent, which has the usual properties and events. Two more properties, `LabelPosition` and `LabelSpacing`, allow you to configure the relative positions of the two controls.

**NOTE**

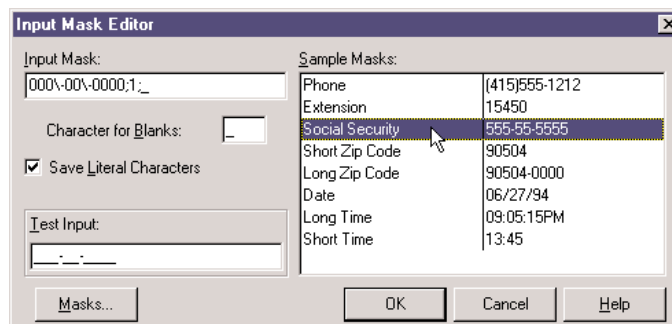
This component has been added to the `ExtCtrls` unit to demonstrate the use of subcomponents in the Object Inspector, which is a new feature of Delphi 6. I'll discuss the development of these components in Chapter 11, "Creating Components." Notice also that this component, along with all of the other new Delphi 6 components, is not (yet) available on CLX and on the first release of Kylix. However, we can expect all non-Windows-specific additions to VCL, including subcomponents in general and the `LabeledEdit` control in particular, to be available in the next release of Kylix.

## The MaskEdit Component

To customize the input of an edit box further, you can use the `MaskEdit` component, which has an `EditMask` property. This is a string indicating for each character whether it should be uppercase, lowercase, or a number, and other similar conditions. You can see the editor of the `EditMask` property in Figure 6.3.

**FIGURE 6.3:**

The `MaskEdit` component's `EditMask` property editor



**TIP**

You can display any property's editor by selecting the property in the Object Inspector and clicking the ellipsis (...) button.

The Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to display the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated by semicolons).

**TIP**

Clicking the Masks button of the Mask Editor lets you choose predefined input masks for different countries.

## The Memo and RichEdit Components

Both of the controls discussed so far allow a single line of input. The Memo component, by contrast, can host several lines of text but (on the Win95/98 platforms) still retains the 16-bit Windows text limit (32 KB) and allows only a single font for the entire text. You can work on the text of the memo line by line (using the Lines string list) or access the entire text at once (using the Text property).

If you want to host a large amount of text or change fonts and paragraph alignments, in VCL you should use the RichEdit control, a Win32 common control based on the RTF document format. You can find an example of a complete editor based on the RichEdit component among the sample programs that ship with Delphi. (The example is named RichEdit, too.)

The RichEdit component has a DefAttributes property indicating the default styles and a SelAttributes property indicating the style of the current selection. These two properties are not of the TFont type, but they are compatible with fonts, so we can use the Assign method to copy the value, as in the following code fragment:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign (RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  end;
end;
```

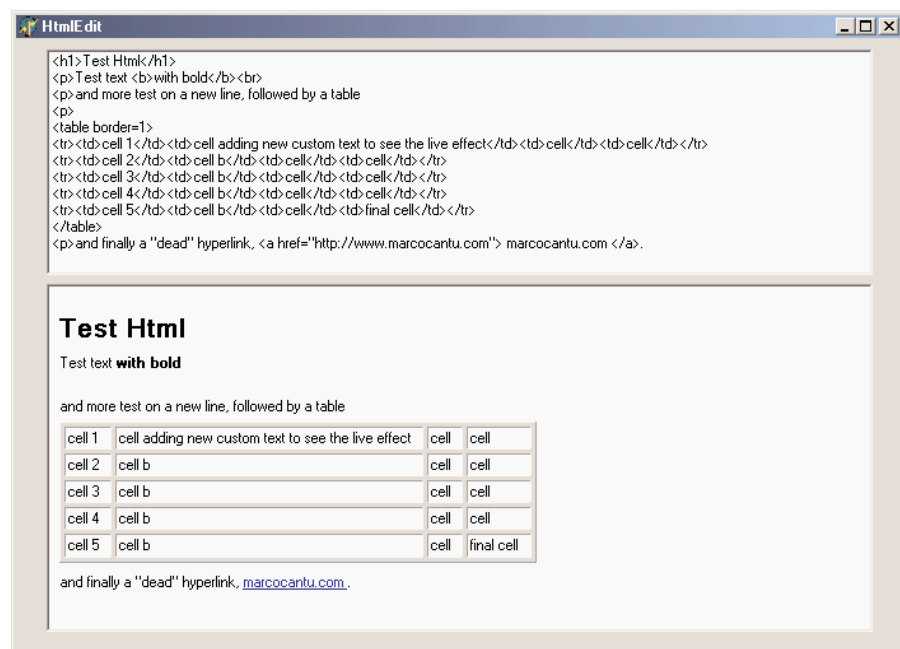
## The TextViewer CLX Control

Among all of the common controls, CLX and Qt lack a RichEdit control. However, they provide a full-blown HTML viewer, which is very powerful for displaying formatted text but not for typing it. This HTML viewer is embedded in two different controls, the single-page TextViewer control or the TextBrowser control with active links.

As a simple demo, I've added a memo and a text viewer to a CLX form and connected them so that everything you type on the memo is immediately displayed in the viewer. I've called the example HtmlEdit not because this is a real HTML editor, but because this is the simplest way I know of to build an HTML preview inside a program. The form of the program is visible at run time in Figure 6.4, while typing some text inside a cell of the table.

**FIGURE 6.4:**

The HtmlEdit example at run time: when you add new HTML text to the memo, you get an immediate preview.



### TIP

I originally built this example with Kylix on Linux. To port it to Windows and Delphi 6, all I had to do was to copy the files and recompile.

## Selecting Options

There are two standard Windows controls that allow the user to choose different options, as well as controls for grouping sets of options.

## The CheckBox and RadioButton Components

The first standard option-selecting control is the *check box*, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the `AllowGrayed` property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks the check box.

The second type of control is the *radio button*, which corresponds to an exclusive selection. Two radio buttons on the same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected (as programmer, you are responsible for selecting one of the radio buttons at design time).

## The GroupBox Components

To host several groups of radio buttons, you can use a `GroupBox` control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the `GroupBox` component on a form and then add the radio buttons to the group box.

You can handle the radio buttons individually, but it's easier to navigate through the array of controls owned by the group box, as discussed in the previous chapter. Here is a small code excerpt used to get the text of the selected radio button of a group:

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := (GroupBox1.Controls[I] as TRadioButton).Caption;
```

## The RadioGroup Component

Delphi has a similar component that can be used specifically for radio buttons: the `RadioGroup` component. A `RadioGroup` is a group box with some radio button *clones* painted inside it. The term *clone* in this context refers to the fact that the `RadioGroup` component is a single control, a single window, with elements similar to radio buttons painted on its surface.

Using the radio group is generally easier than using the group box, since the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Technically, a `RadioGroup` uses fewer resources and less memory, and it should be faster to create and paint. Also, the `RadioGroup` component can automatically align its radio buttons in one or more columns (as indicated by the `Columns` property), and you can easily add new choices at run time, by adding strings to the `Items` string list. By contrast, adding new radio buttons to a group box would be quite complex.



## Lists

When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six, to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that display lists of items and allow the selection of one of them.

### The ListBox Component

The selection of an item in a list box uses the `Items` and `ItemIndex` properties as in the code shown above for the `RadioGroup` control. If you need access to the text of selected list box items often, you can write a small wrapper function like this:

```
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := '';
end;
```

Another important feature is that by using the `ListBox` component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the `MultiSelect` property. There are two kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case, a user selects multiple items simply by clicking them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items, respectively. This second choice is determined by the `ExtendedSelect` property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the `SelCount` property, and it can determine which items are selected by examining the `Selected` array. This array of Boolean values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the `Selected` array as follows:

```
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```

In CLX the `ListBox` can be configured to use a fixed number of columns and rows, using the `Columns`, `Row`, `ColumnLayout` and `RowLayout` properties. Of these, the VCL `ListBox` has only the `Columns` property.

## The ComboBox Component

List boxes take up a lot of screen space, and they offer a fixed selection—that is, a user can choose only among the items in the list box and cannot enter any choice that the programmer did not specifically foresee.

You can solve both problems by using a `ComboBox` control, which combines an edit box and a drop-down list. The behavior of a `ComboBox` component changes a lot depending on the value of its `Style` property:

- The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request.
- The `csDropDownList` style defines a combo box that does not allow editing (but uses the keystrokes to select an item).
- The `csSimple` style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a `ComboBox` is easier than doing the same operation for a list box, since you can simply use the `Text` property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the `Enter` key. The following method first tests whether the user has pressed that key, by looking for the character with the numeric (ASCII) value of 13. It then tests to make sure the text of the combo box is not empty and is not already in the list—if its position in the list is less than zero. Here is the code:

```
procedure TForm1.ComboBox1KeyPress(  
    Sender: TObject; var Key: Char);  
begin  
    // if the user presses the Enter key  
    if Key = Chr (13) then  
        with ComboBox3 do  
            if (Text <> '') and (Items.IndexOf (Text) < 0) then  
                Items.Add (Text);  
end;
```

### NOTE

In CLX, the combo box can automatically add the text typed into the edit to the drop-down list, when the user presses the `Enter` key. Also, some events fire at different times than in VCL.

Delphi 6 includes two new events for the combo box. The `OnCloseUp` event corresponds to the closing of the drop-down list and complements the preexisting `OnDropDown` event. The `OnSelect` event fires only when the user selects something in the drop-down list, as opposed to typing in the edit portion.

Another very nice addition is the `AutoComplete` property. When it is set, the `ComboBox` component (and the `ListBox`, as well) automatically locates the string nearest to the one the user is entering, suggesting the final part of the text. The core of this feature, available also in CLX, is implemented in the `TCustomListBox.KeyPress` method.

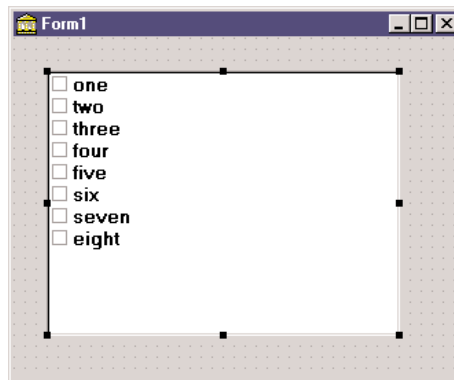
## The CheckListBox Component

Another extension of the list box control is represented by the `CheckListBox` component, a list box with each item preceded by a check box (as you can see in Figure 6.5). A user can select a single item of the list, but can also click the check boxes to toggle their status. This makes the `CheckListBox` a very good component for multiple selections or for highlighting the status of a series of independent items (as in a series of check boxes).

To check the current status of each item, you can use the `Checked` and the `State` array properties (use the latter if the check boxes can be grayed). Delphi 5 introduced the `ItemEnabled` array property, which you can use to enable or disable each item of the list. We'll use the `CheckListBox` in the `DragList` example, later in this chapter.

**FIGURE 6.5:**

The user interface of the `CheckListBox` control, basically a list of check boxes



**TIP**

Most of the list-based controls share a common and important feature. Each item of the list has an associated 32-bit value, usually indicated by the `TObject` type. This value can be used as a tag for each list item, and it's very useful for storing additional information along with each item. This approach is connected to a specific feature of the native Windows list box control, which offers four bytes of extra storage for each list box item. We'll use this feature in the `ODList` example later on in this chapter.

## New Combo Boxes: `ComboBoxEx` and `ColorBox`

The `ComboBoxEx` (where *ex* stands for extended) is the wrapper of a new Win32 common controls, which extends the traditional combo box by allowing images to appear next to the items in the list. You attach an image list to the combo, and then select an image index for each item to display. The effect of this change is that the simple `Items` string list is replaced by a more complex collection, the `ItemsEx` property.

The `ColorBox` control is a new version of the combo box specifically aimed at selecting colors. You can use its `Style` property for choosing which groups of colors you want to see in the list (standard color, extended colors, system colors, and so on).

## The `ListView` and `TreeView` Components

If you want an even more sophisticated list, you can use the `ListView` common control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described at the beginning of the next chapter, “Advanced VCL Controls.” Other alternatives for listing values are the `TreeView` common control, which shows items in a hierarchical output, and the `StringGrid` control, which shows multiple elements for each line. The string grid control is described in the “Graphics in Delphi” bonus chapter, available on the companion CD.

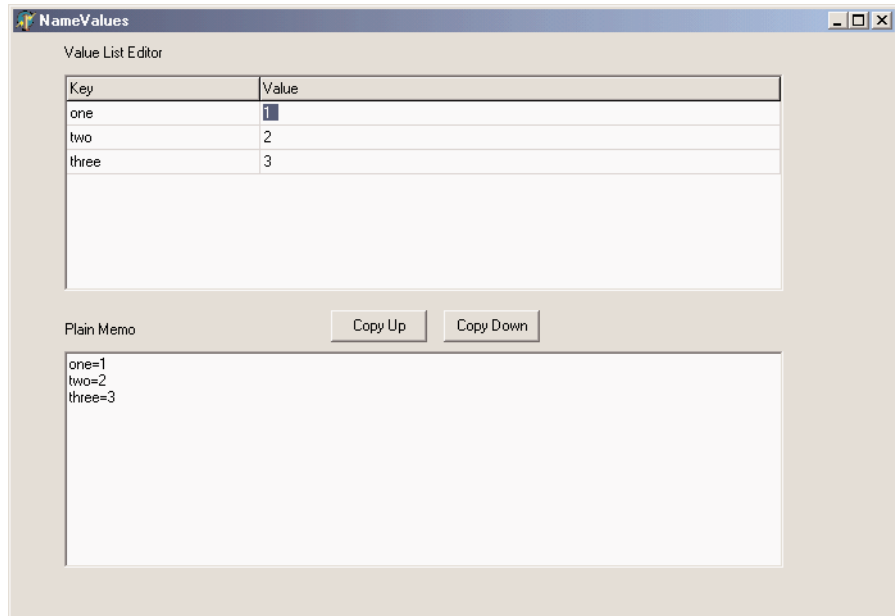
If you use the common controls in your application, users will already know how to interact with them, and they will regard the user interface of your program as up to date. `TreeView` and `ListView` are the two key components of Windows Explorer, and you can assume that many users will be familiar with them, even more than with the traditional Windows controls. `CLX` adds also an `IconView` control, which parallels part of the features of the VCL `ListView`.

## The New `ValueListEditor` Component

Delphi applications often use the name/value structure natively offered by string lists, which I discussed in the last chapter. Delphi 6 introduces a version of the `StringGrid` component specifically geared towards this type of string lists. The `ValueListEditor` has two columns where you can display and let the user edit the contents of a string list with name/value pairs, as you can see in Figure 6.6. This string list is indicated in the `Strings` property of the control.

**FIGURE 6.6:**

The NameValues example has the new ValueListEditor component, which shows the name/value or key/value pairs of a string list, visible also in a plain memo.



The power of this control lies in the fact you can customize the editing options for each position of the grid or for each key value, using the run-time-only `ItemProps` array property. For each item, you can indicate:

- Whether it is read-only
- The maximum number of characters of the string
- An edit mask (eventually requested in the `OnGetEditMask` event)
- The items of a drop-down pick list (eventually requested in the `OnGetPickList` event)
- The display of a button for showing an editing dialog (in the `OnEditButtonClick` event)

Needless to say, this behavior resembles what is available generally for string grids and the `DBGrid` control, but also the behavior of the Object Inspector.

The `ItemProps` property has to be set up at run time, by creating an object of the `TItemProp` class and assigning it to an index or a key of the string list. To have a default editor for each line, you can assign the same item property object multiple times. In the example, this shared editor sets an edit mask for up to three numbers:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
```

```
begin
  SharedItemProp := TItemProp.Create (ValueListEditor1);
  SharedItemProp.EditMask := '999;0; ';

  Memo1.Lines := ValueListEditor1.Strings;
  for I := 0 to ValueListEditor1.Strings.Count - 1 do
    ValueListEditor1.ItemProps [I] := SharedItemProp;
  end;
```

Similar code has to be repeated in case the number of lines changes—for example, by adding new elements in the memo and copying them up to the value list:

```
procedure TForm1.ValueListEditor1StringsChange(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ValueListEditor1.Strings.Count - 1 do
    if not Assigned (ValueListEditor1.ItemProps [I]) then
      ValueListEditor1.ItemProps [I] := SharedItemProp;
  end;
```

**NOTE**

Apparently reassigning the same editor twice causes some trouble, so I've assigned the editor only to the lines not having already one.

Another property, `KeyOptions`, allows you to let the user also edit the keys (the names), add new entries, delete existing ones, and allow for duplicated names in the first portion of the string. Oddly enough, you cannot add new keys unless you also activate the edit options, which makes it hard to let the user add extra entries while preserving the names of the basic ones.

## Ranges

Finally, there are a few components you can use to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

### The ScrollBar Component

The stand-alone `ScrollBar` control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border, a feature governed by a specific Windows style of the form window. By *resembling*, I mean that it is not technically a separate window of the `ScrollBar` component type. These “fake” scroll bars are usually controlled in Delphi using specific properties of the form and the other components hosting them.

## The TrackBar and ProgressBar Components

Direct use of the ScrollBar component is quite rare, especially with the TrackBar component introduced with Windows 95, which is used to let a user select a value in a range. Among Win32 common controls is the companion ProgressBar control, which allows the program to output a value in a range, showing the progress of a lengthy operation.

## The UpDown Component

Another related control is the UpDown component, which is usually connected to an edit box so that the user can either type a number in it or increase and decrease the number using the two small arrow buttons. To connect the two controls, you set the Associate property of the UpDown component. Nothing prevents you from using the UpDown component as a stand-alone control, displaying the current value in a label or in some other way.

### NOTE

In CLX there is no UpDown control, but a SpinEdit that bundles an Edit with the UpDown in a single control.

## The PageScroller Component

The Win32 PageScroller control is a container allowing you to scroll the internal control. For example, if you place a toolbar in the page scroller and the toolbar is larger than the available area, the PageScroller will display two small arrows on the side. Clicking these arrows will scroll the internal area. This component can be used as a scrollbar, but it also partially replaces the ScrollBox control.

## The ScrollBox Component

The ScrollBox control represents a region of a form that can scroll independently from the rest of the surface. For this reason, the ScrollBox has two scrollbars used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many Windows applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scrollbars of the form, in fact, you might allow the user to move the toolbar or status bar out of view, a very odd situation.

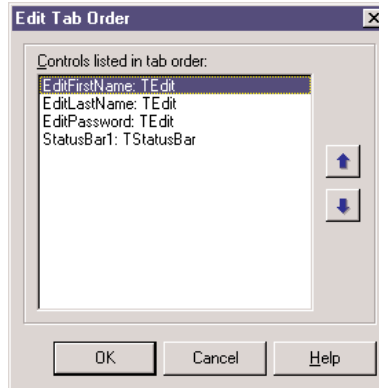
## Handling the Input Focus

Using the TabStop and TabOrder properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key.

Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, as shown in Figure 6.7.

**FIGURE 6.7:**

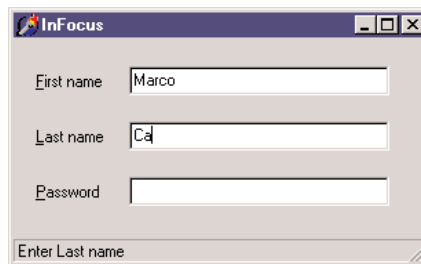
The Edit Tab Order dialog box



Besides these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the `InFocus` example, which creates a fairly typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in Figure 6.8. At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.

**FIGURE 6.8:**

The `InFocus` example at run time



For the output of the status information, I've used the `StatusBar` component, with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the `&` character in the labels, indicating a shortcut key,



and the connection of these labels with corresponding edit boxes (using the `FocusControl` property):

```
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
object Label1: TLabel
  Caption = '&First name'
  FocusControl = EditFirstName
end
object EditFirstName: TEdit
  OnEnter = GlobalEnter
  OnExit = EditFirstNameExit
end
object Label2: TLabel
  Caption = '&Last name'
  FocusControl = EditLastName
end
object EditLastName: TEdit
  OnEnter = GlobalEnter
end
object Label3: TLabel
  Caption = '&Password'
  FocusControl = EditPassword
end
object EditPassword: TEdit
  PasswordChar = '*'
  OnEnter = GlobalEnter
end
object StatusBar1: TStatusBar
  SimplePanel = True
end
end
```

The program is very simple and does only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' `OnEnter` event, possibly using a single generic event handler to avoid repetitive code. In the example, instead of storing some extra information for each edit box, I've checked each control of the form to determine which label is connected to the current edit box (indicated by the `Sender` parameter):

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    // if the control is a label
```

```

    if (Controls [I] is TLabel) and
        // and the label is connected to the current edit box
        (TLabel(Controls[I]).FocusControl = Sender) then
        // copy the text, leaving off the initial & character
        StatusBar1.SimpleText := 'Enter ' +
            Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;

```

The second event handler of the form relates to the `OnExit` event of the first edit box. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```

procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
    if EditFirstName.Text = '' then
        begin
            // don't let the user get out
            EditFirstName.SetFocus;
            MessageDlg ('First name is required', mtError, [mbOK], 0);
        end
    else if EditFirstName.Text = 'Admin' then
        begin
            // fill the second edit and jump to the third
            EditLastName.Text := 'Admin';
            EditPassword.SetFocus;
        end;
end;

```

### TIP

The CLX version of this example has exactly the same code and is available as the `QInFocus` program. The same happens for most of the other examples of this chapter. Notice that some of the examples are quite complex, but I rarely had to touch the code at all.

## Working with Menus

Working with menus and menu items is generally quite simple. This section offers only some very brief notes and a few more advanced examples. The first thing to keep in mind about menu items is that they can serve different purposes:

**Commands** are menu items used to execute an action.

**State-setters** are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they

are active. In Delphi 6 you can automatically obtain this behavior using the handy `AutoCheck` property.

**Radio items** have a round check mark and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, simply set the `RadioItem` property to `True` and set the `GroupIndex` property for the alternative menu items to the same value.

**Dialog menu items** cause a dialog box to appear and are usually indicated by an ellipsis (three dots) after the text.

As you enter new elements in the Menu Designer, Delphi creates a new component for each menu item and lists it in the Object Inspector (although nothing is added to the form). To name each component, Delphi uses the caption you enter and appends a number (so that *Open* becomes *Open1*). Because Delphi removes spaces and other special characters in the caption when it creates the name, and the menu item separators are set up using a hyphen as caption, these items would have an empty name. For this reason Delphi adds the letter *N* to the name, appending the number and generating items called *N1*, *N2*, and so on.

---

**WARNING**

Do not use the `Break` property, which is used to lay out a pull-down menu on multiple columns. The `mbMenuBarBreak` value indicates that this item will be displayed in a second or subsequent line; the `mbMenuBreak` value that this item will be added to a second or subsequent column of the pull-down.

## Accelerator Keys

Since Delphi 5, you don't need to enter the `&` character in the `Caption` of a menu item; it provides an automatic accelerator key if you omit one. Delphi's automatic accelerator-key system can also figure out if you have entered conflicting accelerator keys and fix them on-the-fly. This doesn't mean you should stop adding custom accelerator keys with the `&` character, because the automatic system simply uses the first available letter, and it doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This feature is controlled by the `AutoHotkeys` property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to `maAutomatic`, while in the pull-downs and menu items it defaults to `maParent`, so that the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of `maAutomatic` or `maManual`.

The engine behind this system is the `RethinkHotkeys` method of the `TMenuItem` class, and the companion `InternalRethinkHotkeys`. There is also a `RethinkLines` method, which

checks whether a pull-down has two consecutive separators or begins or ends with a separator. In all these cases, the separator is automatically removed.

One of the reasons Delphi includes this feature is the Integrated Translation Environment (ITE). When you need to translate the menu of an application, it is convenient if you don't have to deal with the accelerator keys, or at least if you don't have to worry about whether two items on the same menu conflict. Having a system that can automatically resolve similar problems is definitely an advantage. Another motivation was Delphi's IDE itself. With all the dynamically loaded packages that install menu items in the IDE main menu or in pop-up menus, and with different packages loaded in different versions of the product, it's next to impossible to get nonconflicting accelerator-key selections in each menu. That is why this mechanism isn't a wizard that does static analysis of your menus at design time; it was created to deal with the real problem of managing menus created dynamically at run time.

**WARNING**

This feature is certainly very handy, but because it is active by default, it can break existing code. I had to modify two of this chapter's program examples, between the Delphi 4 and Delphi 5 edition of the book, just to avoid run-time errors caused by this change. The problem is that I use the caption in the code, and the extra & broke my code. The change was quite simple, though: All I had to do was to set the `AutoHotkeys` property of the main menu component to `maManual`.

## Pop-Up Menus and the *OnContextPopup* Event

Besides the `MainMenu` component, you can use the similar `PopupMenu` component. This is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property.

However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, in this code fragment a label:

```
procedure TForm1.Label13MouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
    ScreenPoint: TPoint;
begin
    // if some condition applies...
    if Button = mbRight then
        begin
            ScreenPoint := Label13.ClientToScreen(Point(X, Y));
            PopupMenu1.Popup(ScreenPoint.X, ScreenPoint.Y)
        end;
end;
```

An alternative approach is the use of the `OnContextMenu` event. This event, introduced in Delphi 5, fires when a user right-clicks a component—exactly what we’ve traced above with the test `if Button = mbRight`. The advantage is that the same event is also fired in response to a `Shift+F10` key combination, as well as by any other user-input methods defined by Windows Accessibility options or hardware (including the shortcut-menu key of some Windows-compatible keyboards). We can use this event to fire a pop-up menu with little code:

```
procedure TFormPopup.Label1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now), 0, False, True, nil, 0, ''));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This is not particularly useful, but I’ve done it to highlight that if you need to display a plain pop-up menu, you can easily use the `PopupMenu` property of the control in question or one of its parent controls. Handling the `OnContextMenu` event makes sense only when you want to do some extra processing.

The `Handled` parameter is preinitialized to `False`, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set `Handled` to `True` and the system will stop processing the message. Setting `Handled` to `True` should be fairly rare, as you’ll generally handle the `OnContextPopup` to dynamically create or customize the pop-up menu, but then you can let the default handler actually show the menu.

The handler of an `OnContextPopup` event isn’t limited to displaying a pop-up menu. It can do any other operation, such as directly display a dialog box. Here is an example of a right-click operation used to change the color of the control:

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
```

```

ColorDialog1.Color := Label2.Color;
if ColorDialog1.Execute then
  Label2.Color := ColorDialog1.Color;
Handled := True;
end;

```

All the code snippets of this section are available in the simple CustPop example for VCL and QCustPop for CLX, on the book's companion CD.

## Creating Menu Items Dynamically

Besides defining the structure of a menu with the Menu Designer and modifying the status of the items using the `Checked`, `Visible`, and `Caption` properties, you can create an entire menu or portions of one at run time. This makes sense, for example, when you have many repetitive items, or when the menu items depend on some system configuration or user permissions.

The basic idea is that each object of the `TMenuItem` class—which Delphi uses for both menu items and pull-down menus—contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each submenu has a list of submenus, each with its own list of submenus, and so on. The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`, which contains the number of subitems. Adding new menu items or entire pull-down menus to an existing menu is fairly easy, particularly if you can write a single event handler for all of them.

This is demonstrated by the `DynaMenu` example (and its `QDynaMenu` counterpart), which also illustrates the use of menu check marks, radio items, and many other features of menus that aren't described in detail in the text. As soon as you start this program, it creates a new pull-down with menu items used to change the font size of a big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you.

The new pull-down menu should be inserted in the `Items` property of the `MainMenu1` component. You can calculate the position by asking the `MainMenu` component for the previous pull-down menu:

```

procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pull-down menu
  PullDown := TMenuItem.Create (Self);
  PullDown.AutoHotkeys := maManual;
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it

```

```

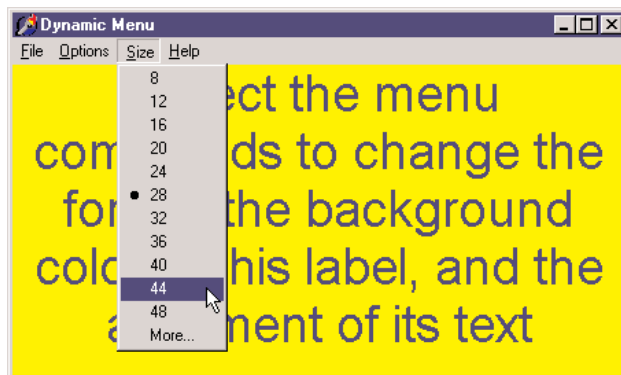
Position := MainMenu1.Items.IndexOf (Options1);
MainMenu1.Items.Insert (Position + 1, PullDown);
// create menu items for various sizes
I := 8;
while I <= 48 do
begin
    // create the new item
    Item := TMenuItem.Create (Self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
end;
// add extra item at the end
Item := TMenuItem.Create (Self);
Item.Caption := 'More...';
// make it a radio item
Item.GroupIndex := 1;
Item.RadioItem := True;
// handle it by showing the font selection dialog
Item.OnClick := Font1Click;
PullDown.Insert (PullDown.Count, Item);
end;

```

As you can see in the preceding code, the menu items are created in a `while` loop, setting the radio item style and calling the `Insert` method with the number of items as a parameter to add each item at the end of the pull-down. At the end, the program adds one extra item, which is used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method (also connected to a specific menu item), which displays the font selection dialog box. You can see the dynamic menu in Figure 6.9.

**FIGURE 6.9:**

The Size pull-down menu of the `DynaMenu` example is created at run time, along with all of its menu items.



**WARNING**

Because the program uses the `Caption` of the new items dynamically, we should either disable the `AutoHotkeys` property of the main menu component, or disable this feature for the pull-down menu we are going to add (and thus automatically disable it for the menu items). This is what I've done in the code above by setting the `AutoHotkeys` property of the dynamically created pull-down component to `maManual`. An alternative approach is to let the menu display the automatic captions and then call the new `StripHotkeys` function before converting the caption to a number. There is also a new `GetHotkey` function, which returns the *active* character of the caption.

The handler for the `OnClick` event of these dynamically created menu items uses the caption of the `Sender` menu item to set the size of the font:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
    with Sender as TMenuItem do
        Label1.Font.Size := StrToInt (Caption);
end;
```

This code doesn't set the proper radio-item mark next to the selected item, because the user can select a new size also by changing the font. The proper radio item is checked in the `OnClick` event handler of the entire pull-down menu, which is connected just after the pull-down is created and activated just before showing the pull-down. The code scans the items of the pull-down menu (the `Sender` object) and checks whether the caption matches the current `Size` of the font. If no match is found, the program checks the last menu item, to indicate that a different size is active:

```
procedure TFormColorText.SizeClick (Sender: TObject);
var
    I: Integer;
    Found: Boolean;
begin
    Found := False;
    with Sender as TMenuItem do
        begin
            // look for a match, skipping the last item
            for I := 0 to Count - 2 do
                if StrToInt (Items [I].Caption) = Label1.Font.Size then
                    begin
                        Items [I].Checked := True;
                        Found := True;
                        System.Break; // skip the rest of the loop
                    end;
                if not Found then
                    Items [Count - 1].Checked := True;
            end;
        end;
end;
```



When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the DynaMenu and QDynaMenu examples. As an alternative, you can also use some global functions available in the Menus unit: `NewMenu`, `NewPopupMenu`, `NewSubMenu`, `NewItem`, and `NewLine`.

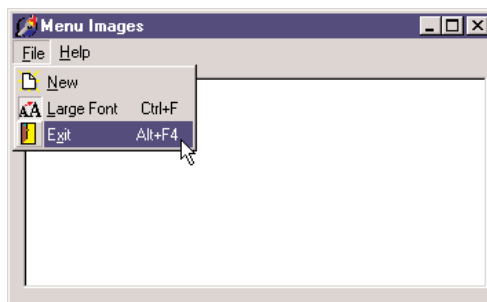
## Using Menu Images

In Delphi it is very easy to improve a program's user interface by adding images to menu items. This is becoming common in Windows applications, and it's very nice that Borland has added all the required support, making the development of graphical menu items trivial.

All you have to do is add an image list control to the form, add a series of bitmaps to the image list, connect the image list to the menu using its `Images` property, and set the proper `ImageIndex` property for the menu items. You can see the effect of these simple operations in Figure 6.10. (You can also associate a bitmap with the menu item directly, using the `Bitmap` property.)

**FIGURE 6.10:**

The simple graphical menu of the `Menuimg` example



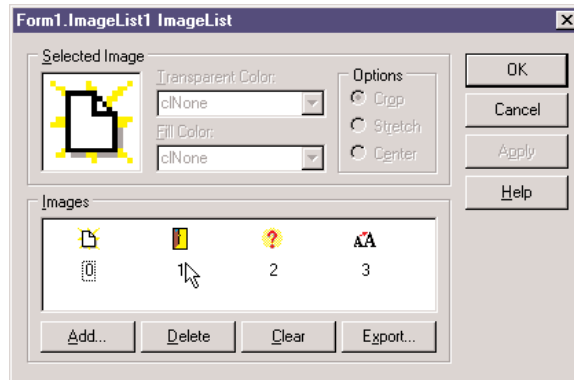
### TIP

The definition of images for menus is quite flexible, as it allows you to associate an image list with any specific pull-down menu (and even a specific menu item) using the `SubMenuImages` property. Having a specific and smaller image list for each pull-down menu, instead of one single huge image list for the entire menu, allows for more run-time customization of an application.

To create the image list, double-click the component, activating the corresponding editor (shown in Figure 6.11), then import existing bitmap or icon files. You can actually prepare a single large bitmap and let the image editor divide it according to the `Height` and `Width` properties of the `ImageList` component, which refer to the size of the individual bitmaps in the list.

**FIGURE 6.11:**

The Image List editor, with the bitmaps of the MenuImg example

**TIP**

As an alternative, you can use the series of images that ship with Delphi and are stored by default in the `\Program Files\Common Files\Borland Shared\Images\Buttons` directory. Each bitmap contains both an “enabled” and a “disabled” image. As you import them, the Image List editor will ask you whether to split them in two, a suggestion you should accept. This operation adds to the image list a normal image and a disabled one, which is not generally used (as it can be built automatically when needed). For this reason I generally delete the disabled part of the bitmap from the image list.

The program’s code is very simple. The only element I want to emphasize is that if you set the `Checked` property of a menu item with an image instead of displaying a check mark, the item paints its image as “sunken” or “recessed.” You can see this in the Large Font menu of the MenuImg example in Figure 6.10. Here is the code for that menu item selection:

```
procedure TForm1.LargeFont1Click(Sender: TObject);
begin
    if Memo1.Font.Size = 8 then
        Memo1.Font.Size := 12
    else
        Memo1.Font.Size := 8;
        // changes the image style near the item
        LargeFont1.Checked := not LargeFont1.Checked;
end;
```

**WARNING**

To make the CLX version of the program, QMenuImg, display the bitmaps properly, I had to reimport them. Simply converting the Image List component data didn’t work.

## Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and “quick-and-dirty” applications.

Adding a single menu item to the system menu is straightforward:

```
AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');  
AppendMenu (GetSystemMenu (Handle, FALSE), MF_STRING, idSysAbout, '&About...');
```

This code fragment (extracted from the `OnCreate` event handler of the `SysMenu` example) adds a separator and a new item to the system menu item. The `GetSystemMenu` API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The `AppendMenu` API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I’ve defined this identifier as:

```
const idSysAbout = 100;
```

Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the `wm_Command` Windows message. This is handled internally by Delphi, which activates the `OnClick` event of the corresponding menu item component. The selection of system menu commands, instead, generates a `wm_SysCommand` message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.

We can intercept this command and check to see whether the command identifier (passed in the `CmdType` field of the `TWMSysCommand` parameter) of the menu item is `idSysAbout`. Since there isn’t a corresponding event in Delphi, we have to define a new message-response method for the form class:

```
public  
  procedure WMSysCommand (var Msg: TMessage);  
  message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);  
begin  
  if Msg.CmdType = idSysAbout then  
    ShowMessage ('Mastering Delphi: SysMenu example');  
  inherited;  
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a `MainMenu` component to

the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the `Menu` property of the form, removing the menu bar.

Now we can add some code to the `SysMenu` example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is clicked. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
begin
    // add a separator
    AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
    // add the main menu to the system menu
    with MainMenu1 do
        for I := 0 to Items.Count - 1 do
            AppendMenu (GetSystemMenu (Self.Handle, FALSE),
                mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
    // disable the button
    Button1.Enabled := False;
end;

```

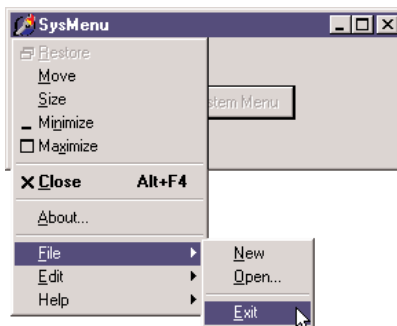
### TIP

This code uses the expression `Self.Handle` to access the handle of the form. This is required because we are currently working on the `MainMenu1` component, as specified by the `with` statement.

The menu flag used in this case, `mf_Popup`, indicates that we are adding a pull-down menu. In this function call, the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example, we passed the identifier of the menu, instead). Since we are adding to the system menu items with submenus, the final structure of the system menu will have two levels, as you can see in Figure 6.12.

**FIGURE 6.12:**

The second-level system menu items of the `SysMenu` example are the result of copying a complete main menu to the system menu.



---

**WARNING** The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms to mean different things. Pop-up menus are shortcut menus, and pull-down menus are the secondary menus of the menu bar. Apparently, Microsoft uses the terms in this way because the two elements are implemented with the same kind of internal windows; the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course, you can check for each menu item in the `WMSysCommand` method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the `OnClick` event of each item, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a `FindItem` method.

When (and if) we have found a main menu item that corresponds to the item selected in the system menu, we can call its `Click` method (which invokes the `OnClick` handler). Here is the code I've added to the `WMSysCommand` method:

```
var
    Item: TMenuItem;
begin
    ...
    Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
    if Item <> nil then
        Item.Click;
```

In this code, the `CmdType` field of the message structure that is passed to the `WMSysCommand` procedure holds the command of the menu item being called.

You can also use a simple `if` or `case` statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as `sc_Close`, `sc_Minimize`, `sc_Maximize`, and so on. For more information, you can see the description of the `wm_SysCommand` message in the Windows API Help file.

This application works but has one glitch. If you click the right mouse button over the Taskbar icon representing the application, you get a plain system menu (actually different from the default one). The reason is that this system menu belongs to a different window, the window of the `Application` global object. I'll discuss the `Application` object, and update this example to make it work with the Taskbar button, in Chapter 9, "Working with Forms."

---

**NOTE** Because this program uses low-level Windows features (API calls and messages), it is not possible to compile it with CLX, so there is no Qt version of this example.

## Owner-Draw Controls and Styles

Let's return briefly to menu graphics. Besides using an `ImageList` to add glyphs to the menu items, you can turn a menu into a completely graphical element, using the owner-draw technique. The same technique also works for other controls, such as list boxes. In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically, these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In VCL, the situation is slightly more complex. The components can take care of painting themselves in this case (as in the `TBitBtn` class for bitmap buttons) and possibly activate corresponding events. The system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers.

In CLX, some of the controls, such as `ListBoxes` and `ComboBoxes`, surface events very similar to Windows owner-draw, but menus lack them. The native approach of Qt is to use styles to determine the graphical behavior of all of the controls in the system, of a specific application, or of a given control. I'll introduce styles shortly, later in this section.

### NOTE

Most of the Win32 common controls have support for the owner-draw technique, generally called *custom drawing*. You can fully customize the appearance of a `ListView`, `TreeView`, `TabControl`, `PageControl`, `HeaderControl`, `StatusBar`, and `ToolBar`. The `ToolBar`, `ListView`, and `TreeView` controls also support *advanced* custom drawing, a more fine-tuned drawing capability introduced by Microsoft in the latest versions of the Win32 common controls library. The downside to owner-draw is that when the Windows user interface style changes in the future (and it always does), your owner-draw controls that fit in perfectly with the current user interface styles will look outdated and out of place. Since you are creating a custom user interface, you'll need to keep it updated yourself. By contrast, if you use the standard output of the controls, your applications will automatically adapt to a new version of such controls.

## Owner-Draw Menu Items

VCL makes the development of graphical menu items quite simple compared to the traditional approach of the Windows API: You set the `OwnerDraw` property of a menu item component to `True` and handle its `OnMeasureItem` and `OnDrawItem` events. This same feature is not available on CLX.

In the `OnMeasureItem` event, you can determine the size of the menu items. This event handler is activated once for each menu item when the pull-down menu is displayed and has two reference parameters you can set:

```
procedure ColorMeasureItem (Sender: TObject; ACanvas: TCanvas;  
    var Width, Height: Integer);
```

The other parameter, `ACanvas`, is typically used to determine the height of the current font.

In the `OnDrawItem` event, you paint the actual image. This event handler is activated every time the item has to be repainted. This happens when Windows first displays the items and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on. Luckily enough, the Delphi event passes to the handler the `Canvas` where it should paint, the output rectangle, and the status of the item (selected or not):

```
procedure ColorDrawItem(Sender: TObject; ACanvas: TCanvas; ARect: TRect;
    Selected: Boolean);
```

In the `ODMenu` example, I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've set the `OwnerDraw` property of the menu and written handlers for some of the menu items. To write a single handler for each event of the three color-related menu items, I've set their `Tag` property to the value of the actual color in the `OnCreate` event handler of the form. This makes the handler of the actual `OnClick` event of the items quite straightforward:

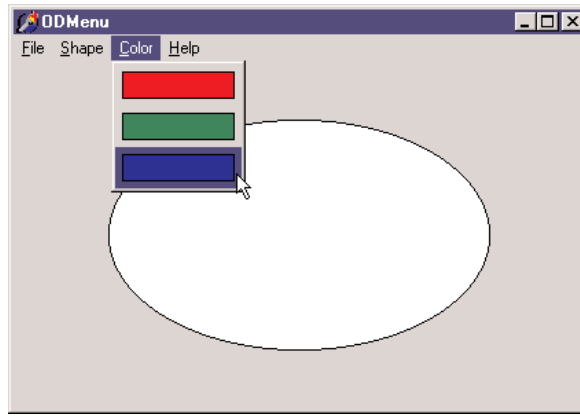
```
procedure TForm1.ColorClick(Sender: TObject);
begin
    ShapeDemo.Brush.Color := (Sender as TComponent).Tag
end;
```

The handler of the `OnMeasureItem` event doesn't depend on the actual items, but uses fixed values (different from the handler of the other pull-down). The most important portion of the code is in the handlers of the `OnDrawItem` events. For the color, we use the value of the tag to paint a rectangle of the given color, as you can see in Figure 6.13. Before doing this, however, we have to fill the background of the menu items (the rectangular area passed as a parameter) with the standard color for the menu (`clMenu`) or the selected menu items (`clHighlight`):

```
procedure TForm1.ColorDrawItem(Sender: TObject; ACanvas: TCanvas;
    ARect: TRect; Selected: Boolean);
begin
    // set the background color and draw it
    if Selected then
        ACanvas.Brush.Color := clHighlight
    else
        ACanvas.Brush.Color := clMenu;
    ACanvas.FillRect (ARect);
    // show the color
    ACanvas.Brush.Color := (Sender as TComponent).Tag;
    InflateRect (ARect, -5, -5);
    ACanvas.Rectangle (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

**FIGURE 6.13:**

The owner-draw menu of the ODMenu example



The three handlers for this event of the Shape pull-down menu items are all different, although they use similar code:

```
procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
    ACanvas.FillRect (ARect);
  // draw the ellipse
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

**NOTE**

To accommodate the increasing number of states in the Windows 2000 user interface style, since version 5, Delphi has included the `OnAdvancedDrawItem` event for menus.

## A ListBox of Colors

As we have just seen for menus, list boxes have an owner-draw capability, which means a program can paint the items of a list box. The same support is provided for combo boxes and is also available on CLX. To create an owner-draw list box, we set its `Style` property to `lbOwnerDrawFixed` or `lbOwnerDrawVariable`. The first value indicates that we are going to set the height of the items of the list box by specifying the `ItemHeight` property and that this will be



the height of each and every item. The second owner-draw style indicates a list box with items of different heights; in this case, the component will trigger the `OnMeasureItem` event for each item, to ask the program for their heights.

In the `ODList` example (and its `QODList` version), I'll stick with the first, simpler, approach. The example stores color information along with the items of the list box and then draws the items in colors (instead of using a single color for the whole list).

The `DFM` or `XFM` file of every form, including this one, has a `TextHeight` attribute, which indicates the number of pixels required to display text. This is the value we should use for the `ItemHeight` property of the list box. An alternative solution is to compute this value at run time, so that if we later change the font at design time, we don't have to remember to set the height of the items accordingly.

**NOTE**

I've just described `TextHeight` as an *attribute* of the form, not a property. And in fact it isn't a property but a local value of the form. If it is not a property, you might ask, how does Delphi save it in the `DFM` file? Well, the answer is that Delphi's streaming mechanism is based on properties plus special *property clones* created by the `DefineProperties` method.

Since `TextHeight` is *not* a property, although it is listed in the form description, we cannot access it directly. Studying the `VCL` source code, I found that this value is computed by calling a private method of the form: `GetTextHeight`. Since it is private, we cannot call this function. What we can do is duplicate its code (which is actually quite simple) in the `FormCreate` method of the form, after selecting the font of the list box:

```
Canvas.Font := ListBox1.Font;  
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

The next thing we have to do is add some items to the list box. Since this is a list box of colors, we want to add color names to the `Items` of the list box and the corresponding color values to the `Objects` data storage related to each item of the list. Instead of adding the two values separately, I've written a procedure to add new items to the list:

```
procedure TODListForm.AddColors (Colors: array of TColor);  
var  
  I: Integer;  
begin  
  for I := Low (Colors) to High (Colors) do  
    ListBox1.Items.AddObject (ColorToString (Colors[I]), TObject(Colors[I]));  
end;
```

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. For each item passed as a parameter, we add the name of the color to the list, and we add its value to the related data, by calling the `AddObject` method. To obtain the string corresponding to the color, we call the Delphi `ColorToString` function. This returns a string

containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the `TObject` data type (a four-byte reference), as required by the `AddObject` method.

**TIP**

Besides `ColorToString`, which converts a color value into the corresponding string with the identifier or the hexadecimal value, there is also a Delphi function to convert a properly formatted string into a color, `StringToColor`.

In the `ODList` example, this method is called in the `OnCreate` event handler of the form (after previously setting the height of the items):

```
AddColors ([c1Red, c1Blue, c1Yellow, c1Green, c1Fuchsia, c1Lime, c1Purple,
           c1Gray, RGB (213, 23, 123), RGB (0, 0, 0), c1Aqua, c1Navy, c1Olive, c1Teal]);
```

To compile the CLX version of this code, I've added to it the `RGB` function described earlier in the section "Colors." The code used to draw the items is not particularly complex. We simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl; Index: Integer;
    Rect: TRect; State: TOwnerDrawState);
begin
    with Control as TListBox do
        begin
            // erase
            Canvas.FillRect(Rect);
            // draw item
            Canvas.Font.Color := TColor (Items.Objects [Index]);
            Canvas.TextOut(Rect.Left, Rect.Top, ListBox1.Items[Index]);
        end;
    end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on our part. You can see an example of the output of this program at startup in Figure 6.14.

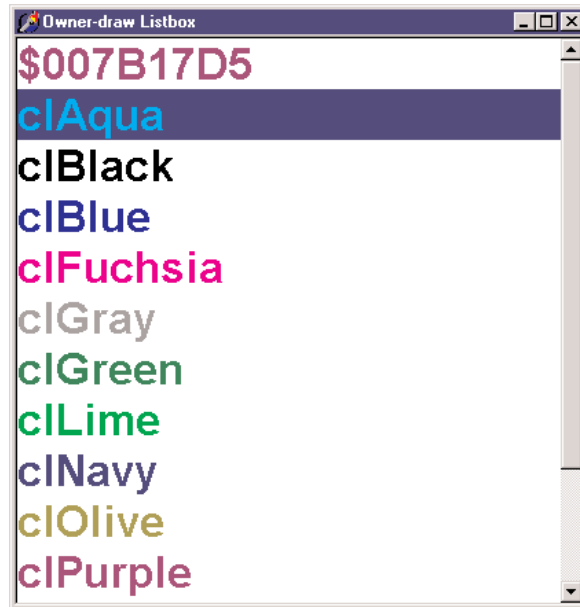
The example also allows you to add new items, by double-clicking the list box:

```
procedure TODListForm.ListBox1Db1Click(Sender: TObject);
begin
    if ColorDialog1.Execute then
        AddColors ([ColorDialog1.Color]);
    end;
```

If you try using this capability, you'll notice that some colors you add are turned into color names (one of the Delphi color constants) while others are converted into hexadecimal numbers.

**FIGURE 6.14:**

The output of the `ODList` example, with a colored owner-draw list box



## CLX Styles

In Windows, the system has full control of the user interface of the controls, unless the program takes over using owner-draw or other advanced techniques. In Qt (and in Linux in general), the user chooses the user interface style of the controls. A system will generally offer a few basic styles, such as the Windows look-and-feel, the Motif one, and others. A user can add also install new styles in the system and make them available to applications.

### NOTE

The styles I'm discussing here refer to the user interface of the controls, not of the forms and their borders. This is generally configurable on Linux systems but is technically a separate element of the user interface.

Because this technique is embedded in Qt, it is also available on the Windows version of the library, and CLX makes it available to Delphi developers. The `Application` global object of CLX has a `Style` property, which can be used to set a custom style or a default one, indicated by the `DefaultStyle` subproperty. For example, you can select a Motif look-and-feel with this code:

```
Application.Style.DefaultStyle := dsMotif;
```

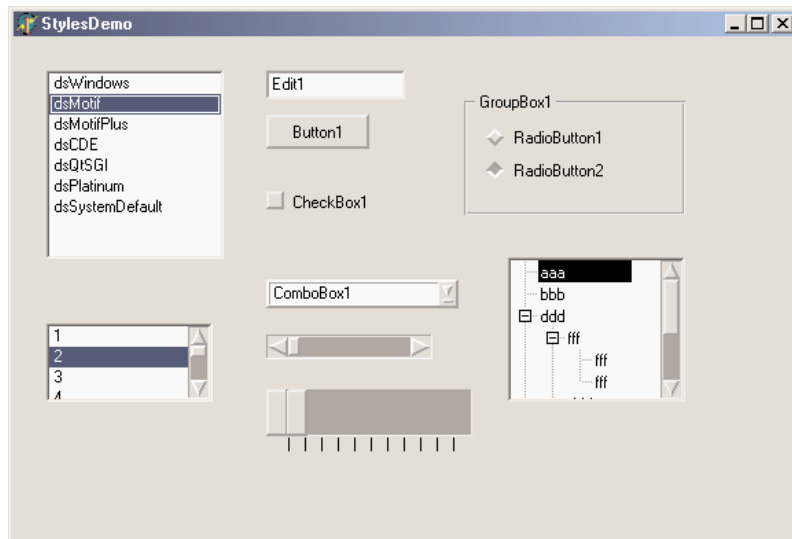
In the StylesDemo program, I've added, among various sample controls, a list box with the names of the default styles, as indicated in the `TDefaultStyle` enumeration, and this code for its `OnDb1Click` event:

```
procedure TForm1.ListBox1Db1Click(Sender: TObject);  
begin  
    Application.Style.DefaultStyle := TDefaultStyle (ListBox1.ItemIndex);  
end;
```

The effect is that, by double-clicking the list box, you can change the current application style and immediately see its effect on screen, as demonstrated in Figure 6.15.

**FIGURE 6.15:**

The StylesDemo program, a Windows application currently with an unusual Motif layout



## What's Next?

In this chapter, we have explored the foundations of the libraries available in Delphi for building user interfaces, the native-Windows VCL and the Qt-based CLX. We've discussed the `TControl` class, its properties, and its most important derived classes.

Then we've started to explore some of the basic components available in Delphi, looking at both libraries. These components correspond to the standard Windows controls and some of the common controls, and they are extremely common in applications. You've also seen how to create main menus and pop-up menus and how to add extra graphics to some of these controls.

The next step, however, is to explore in depth the elements of a complete user interface, discussing other common controls, multipage forms, action lists, and the new Delphi 6 Action Manager, to end up discussing technical details of forms. All of these topics will be covered in the next three chapters.

# Advanced VCL Controls

- ListView and TreeView controls
- Multipage forms
- Pages and tabs
- Form-splitting techniques
- Control anchors
- A ToolBar and a StatusBar for the RichEdit control
- Customizing hints

In the preceding chapter, I discussed the core concepts of the `TControl` class and its derived classes in the VCL and VisualCLX libraries. After that, I provided a sort of rapid tour of the key controls you can use to build a user interface, including editing components, lists, range selectors, and more.

This chapter provides more details on some of these components (such as the `ListView` and `TreeView`) and then discusses other controls used to define the overall design of a form, such as the `PageControl`, `TabControl`, and `Splitter`. The chapter also presents examples of splitting forms and resizing controls dynamically. These topics are not particularly complex, but it is worth examining their key concepts briefly.

After these components, I'll introduce toolbars and status bars, including the customization of hints and other slightly more advanced features. This will give us all the foundation material for the following chapter, which covers actions and the new action manager architecture of Delphi 6.

## Listview and Treeview Controls

In Chapter 6, I introduced all the various visual controls you can use to display lists of values. The standard list box and combo box components are still very common, but they are often replaced by the more powerful `ListView` and `TreeView` controls. Again, these two controls are part of the Win32 common controls, stored in the `ComCtl32.DLL` library. Similar controls are available in Qt and VisualCLX.

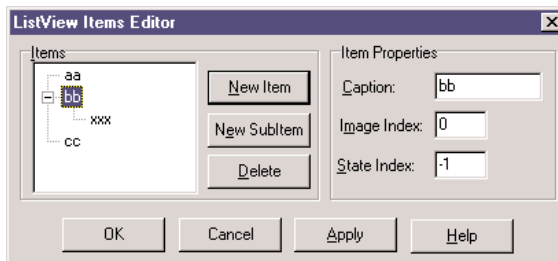
### A Graphical Reference List

When you use a `ListView` component, you can provide bitmaps both indicating the status of the element (for example, the selected item) and describing the contents of the item in a graphical way.

How do we connect the images to a list or tree? We need to refer to the `ImageList` component we've already used for the images of the menu. A `ListView` can actually have three image lists: one for the large icons (the `LargeImages` property), one for the small icons (the `SmallImages` property), and one used for the state of the items (the `StateImages` property). In the `RefList` example on the companion CD, I've set the first two properties using two different `ImageList` components.

Each of the items of the `ListView` has an `ImageIndex`, which refers to its image in the list. For this to work properly, the elements in the two image lists should follow the same order. When you have a fixed image list, you can add items to it using Delphi's `ListView Item Editor`, which is connected to the `Items` property. In this editor, you can define items and so-called subitems. The subitems are displayed only in the detailed view (when you set the

vsReport value of the ViewStyle property) and are connected with the titles set in the Columns property.



In my RefList example (a simple list of references to books, magazines, CD-ROMs, and Web sites), the items are stored to a file, since users of the program can edit the contents of the list, which are automatically saved as the program exits. This way, edits made by the user become persistent. Saving and loading the contents of a ListView is not trivial, since the TListItem type doesn't have an automatic mechanism to save the data. As an alternative simple approach, I've copied the data to and from a string list, using a custom format. The string list can then be saved to a file and reloaded with a single command.

The file format is simple, as you can see in the following saving code. For each item of the list, the program saves the caption on one line, the image index on another line (prefixed by the @ character), and the subitems on the following lines, indented with a tab character:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
    I, J: Integer;
    List: TStringList;
begin
    // store the items
    List := TStringList.Create;
    try
        for I := 0 to ListView1.Items.Count - 1 do
            begin
                // save the caption
                List.Add (ListView1.Items[I].Caption);
                // save the index
                List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
                // save the subitems (indented)
                for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
                    List.Add (#9 + ListView1.Items[I].SubItems [J]);
            end;
        List.SaveToFile (ExtractFilePath (Application.ExeName) + 'Items.txt');
    finally
```

```
List.Free;
end;
```

```
end;
```

The items are then reloaded in the FormCreate method:

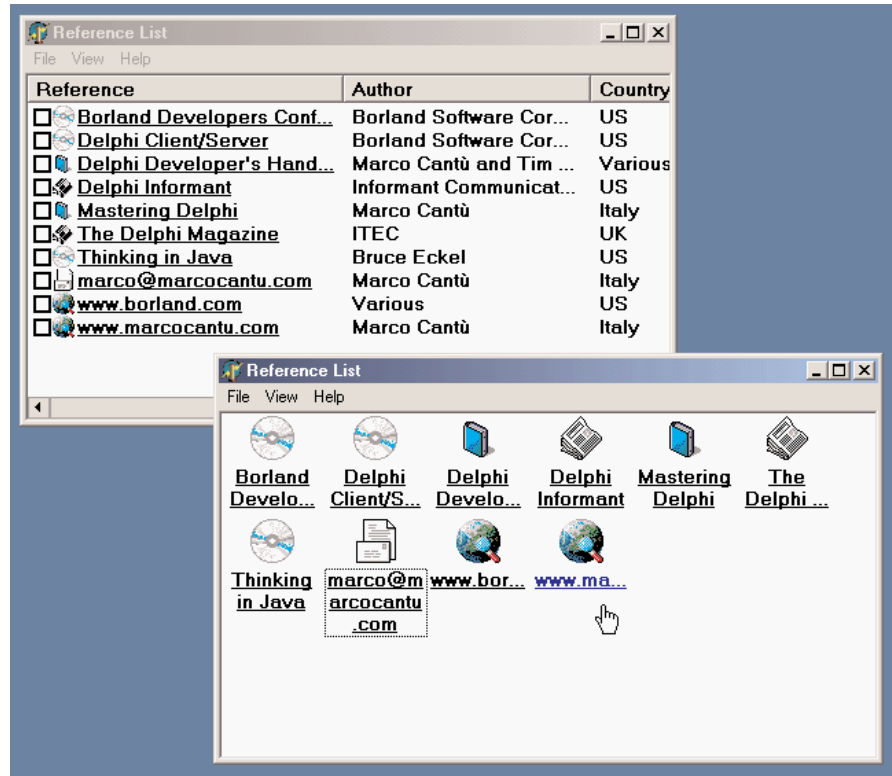
```
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStringList;
 NewItem: TListItem;
  I: Integer;
begin
  // stops warning message
  NewItem := nil;
  // load the items
  ListView1.Items.Clear;
  List := TStringList.Create;
  try
    List.LoadFromFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
    for I := 0 to List.Count - 1 do
      if List [I][1] = #9 then
        NewItem.SubItems.Add (Trim (List [I]))
      else if List [I][1] = '@' then
        NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
      else
        begin
          // a new item
          NewItem := ListView1.Items.Add;
          NewItem.Caption := List [I];
        end;
    finally
      List.Free;
    end;
  end;
end;
```

The program has a menu you can use to choose one of the different views supported by the ListView control, and to add check boxes to the items, as in a CheckListBox. You can see some of the various combinations of these styles in Figure 7.1.



**FIGURE 7.1:**

Different examples of the output of a ListView component of the RefList program, obtained by changing the ViewStyle property and adding the check boxes



Another important feature, which is common in the detailed or report view of the control, is to let a user sort the items on one of the columns. To accomplish this requires three operations. The first is to set the SortType property of the ListView to stBoth or stData. In this way, the ListView will operate the sorting not based on the captions, but by calling the OnCompare event for each two items it has to sort. Since we want to do the sorting on each of the columns of the detailed view, we also handle the OnColumnClick event (which takes place when the user clicks the column titles in the detailed view, but only if the ShowColumnHeaders property is set to True). Each time a column is clicked, the program saves the number of that column in the nSortCol private field of the form class:

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
    Column: TListColumn);
begin
    nSortCol := Column.Index;
    ListView1.AlphaSort;
end;
```

Then, in the third step, the sorting code uses either the caption or one of the subitems according to the current sort column:

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem;
  Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
      Item2.SubItems [nSortCol - 1]);
end;
```

The final features I've added to the program relate to mouse operations. When the user left-clicks an item, the RefList program shows a description of the selected item. Right-clicking the selected item sets it in edit mode, and a user can change it (keep in mind that the changes will automatically be saved when the program terminates). Here is the code for both operations, in the OnMouseDown event handler of the ListView control:

```
procedure TForm1.ListView1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
      begin
        // create and show a description
        strDescr := ListView1.Columns [0].Caption + #9 +
          ListView1.Selected.Caption + #13;
        for I := 1 to ListView1.Selected.SubItems.Count do
          strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
            ListView1.Selected.SubItems [I-1] + #13;
        ShowMessage (strDescr);
      end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
end;
```

Although it is not feature-complete, this example shows some of the potential of the ListView control. I've also activated the "hot-tracking" feature, which lets the list view highlight and

underline the item under the mouse. The relevant properties of the ListView can be seen in its textual description:

```
object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
    item
      Caption = 'Country'
      Width = 80
    end>
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  FullDrag = True
  HideSelection = False
  HotTrack = True
  HotTrackStyles = [htHandPoint, htUnderlineHot]
  SortType = stBoth
  ViewStyle = vsList
  OnColumnClick = ListView1ColumnClick
  OnCompare = ListView1Compare
  OnMouseDown = ListView1MouseDown
end
```

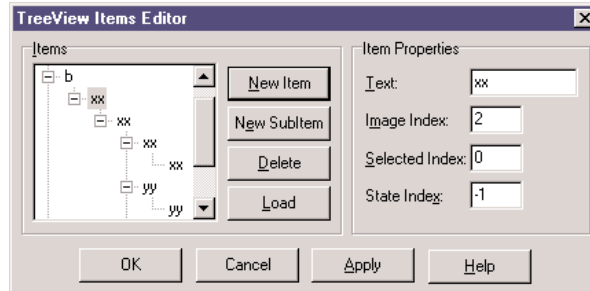
This program is actually quite interesting, and I'll further extend it in Chapter 9, adding a dialog box to it.

To build its CLX version, QRefList, I had to use only one of the image lists, and disable the small images and large images menus, as a ListView is limited to the list and report view styles. Large and small icons are available in a different control, called IconView.

## A Tree of Data

Now that we've seen an example based on the ListView, we can examine the TreeView control. The TreeView has a user interface that is flexible and powerful (with support for editing and dragging elements). It is also standard, because it is the user interface of the Windows Explorer. There are properties and various ways to customize the bitmap of each line or of each type of line.

To define the structure of the nodes of the TreeView at design time, you can use the TreeView Items property editor. In this case, however, I've decided to load it in the TreeView data at startup, in a way similar to the last example.



The Items property of the TreeView component has many member functions you can use to alter the hierarchy of strings. For example, we can build a two-level tree with the following lines:

```
var
    Node: TTreeNode;
begin
    Node := TreeView1.Items.Add (nil, 'First level');
    TreeView1.Items.AddChild (Node, 'Second level');
```

Using these two methods (Add and AddChild), we can build a complex structure at run time. But how do we load the information? Again, you can use a StringList at run time, load a text file with the information, and parse it.

However, since the TreeView control has a LoadFromFile method, the DragTree and QDragTree examples use the following simpler code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    TreeView1.LoadFromFile (ExtractFilePath (Application.ExeName) +
        'TreeText.txt');
end;
```

The LoadFromFile method loads the data in a string list and checks the level of each item by looking at the number of tab characters. (If you are curious, see the TTreeStrings.GetBufStart method, which you can find in the ComCtrls unit in the VCL source code included in Delphi.) By the way, the data I've prepared for the TreeView is the organizational chart of a multinational company.

Besides loading the data, the program saves it when it terminates, making the changes persistent. It also has a few menu items to customize the font of the TreeView control and change some other simple settings. The specific feature I've implemented in this example is support for dragging items and entire subtrees. I've set the `DragMode` property of the component to `dmAutomatic` and written the event handlers for the `OnDragOver` and `OnDragDrop` events.

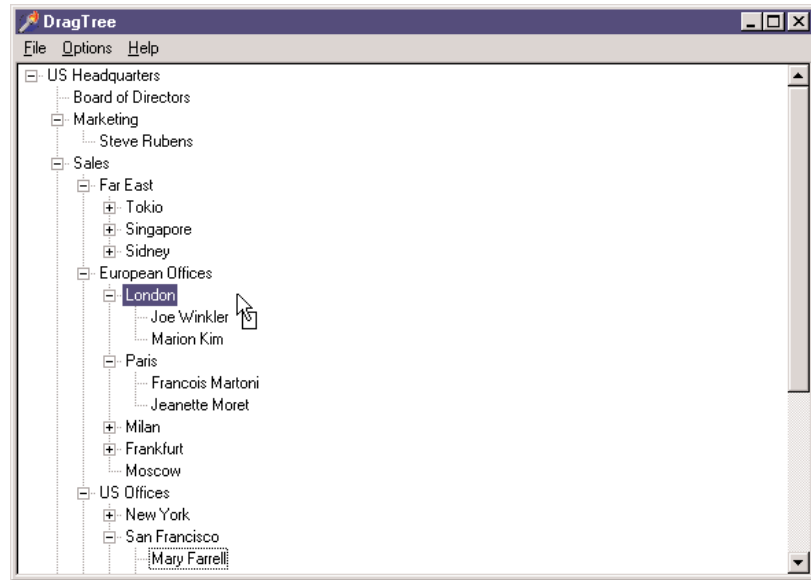
In the first of the two handlers, the program makes sure the user is not trying to drag an item over a child item (which would be moved along with the item, leading to an infinite recursion):

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
    begin
      Accept := True;
      // determines source and target
      SourceNode := TreeView1.Selected;
      // look up the target parent chain
      while (TargetNode.Parent <> nil) and (TargetNode <> SourceNode) do
        TargetNode := TargetNode.Parent;
      // if source is found
      if TargetNode = SourceNode then
        // do not allow dragging over a child
        Accept := False;
    end
  else
    Accept := False;
end;
```

The effect of this code is that (except for the particular case we need to disallow) a user can drag an item of the TreeView over another one, as shown in Figure 7.2. Writing the actual code for moving the items is simple, because the TreeView control provides the support for this operation, through the `MoveTo` method of the `TTreeNode` class.

**FIGURE 7.2:**

The DragTree example during a dragging operation



```

procedure TForm1.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
    TargetNode, SourceNode: TTreeNode;
begin
    TargetNode := TreeView1.GetNodeAt (X, Y);
    if TargetNode <> nil then
        begin
            SourceNode := TreeView1.Selected;
            SourceNode.MoveTo (TargetNode, naAddChildFirst);
            TargetNode.Expand (False);
            TreeView1.Selected := TargetNode;
        end;
    end;

```

**NOTE**

Among the demos shipping with Delphi is an interesting one showing a custom-draw TreeView control. The example is in the CustomDraw subdirectory.

## Custom Tree Nodes

Delphi 6 adds a few new features to the TreeView controls, including multiple selection (see the MultiSelect and MultiSelectStyle properties and the Selections array), improved sorting, and several new events.

Another improved area relates to the creation of custom tree node items, which is useful to add extra custom information to each node and possibly create nodes of different classes. To support this technique, there is a new `AddNode` method for the `TTreeItems` class and a new specific event, `OnCreateNodesClass`. In the handler of this event, you return the class of the object to be created, which must inherit from `TTreeNode`.

As this is a very common technique, I've built an example to discuss it in detail. The `CustomNodes` example on the CD doesn't focus on a real-world case, but it shows a rather complex situation, in which there are two different custom tree node classes, derived one from the other. The base class adds an `ExtraCode` property, mapped to virtual methods, and the subclass overrides one of these methods. For the base class the `GetExtraCode` function simply returns the value, while for the derived class the value is multiplied to the parent node value. Here are the classes and this second method:

```
type
  TMyNode = class (TTreeNode)
  private
    FExtraCode: Integer;
  protected
    procedure SetExtraCode(const Value: Integer); virtual;
    function GetExtraCode: Integer; virtual;
  public
    property ExtraCode: Integer read GetExtraCode write SetExtraCode;
  end;

  TMySubNode = class (TMyNode)
  protected
    function GetExtraCode: Integer; override;
  end;

function TMySubNode.GetExtraCode: Integer;
begin
  Result := fExtraCode * (Parent as TMyNode).ExtraCode;
end;
```

With these custom tree node classes available, the program creates a tree of items, using the first type for the first-level nodes and the second class for the other nodes. As we have only one `OnCreateNodeClass` event handler, it uses the class reference stored in a private field of the form `(CurrentNodeClass of type TTreeNodeClass)`:

```
procedure TForm1.TreeView1CreateNodeClass(Sender: TCustomTreeView;
  var NodeClass: TTreeNodeClass);
begin
  NodeClass := CurrentNodeClass;
end;
```

The program sets this class reference before creating nodes of each type—for example, with code like

```
var
  MyNode: TMyNode;
begin
  CurrentNodeClass := TMyNode;
  MyNode := TreeView1.Items.AddChild (nil, 'item' + IntToStr (nValue))
    as TMyNode;
  MyNode.ExtraCode := nValue;
```

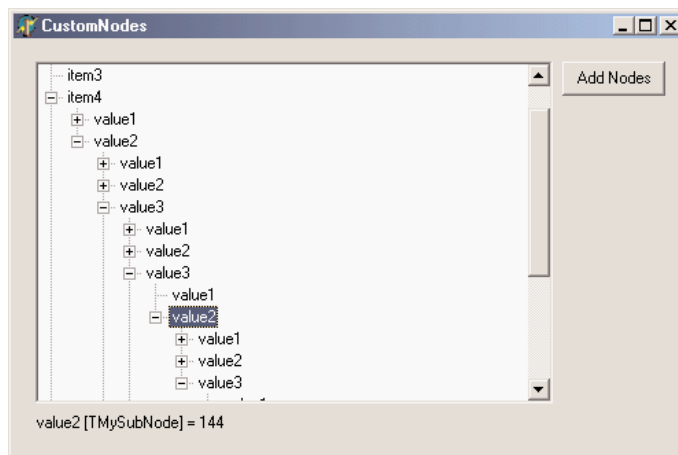
Once the entire tree has been created, as the user selects an item, you can cast its type to TMyNode and access to the extra properties (but also methods and data):

```
procedure TForm1.TreeView1Click(Sender: TObject);
var
  MyNode: TMyNode;
begin
  MyNode := TreeView1.Selected as TMyNode;
  Label1.Caption := MyNode.Text + ' [' + MyNode.ClassName + '] = ' +
    IntToStr (MyNode.ExtraCode);
end;
```

This is the code used by the CustomNodes example to display the description of the selected node into a label, as you can see in Figure 7.3. Note that when you select an item down into the tree, its value is multiplied for that of each of the parent nodes. Though there are certainly easier ways to obtain this effect, having a tree view with item objects created from different classes of a hierarchy provides an object-oriented structure upon which you can base some very complex code.

**FIGURE 7.3:**

The CustomNodes example has a tree view with node objects based on different custom classes, thanks to the new OnCreateNodesClass event.





## Multiple-Page Forms

When you have a lot of information and controls to display in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages.

There are two controls you can use to build a multiple-page application in Delphi:

- You can use the PageControl component, which has tabs on one of the sides and multiple pages (similar to panels) covering the rest of its surface. As there is one page per tab, you can simply place components on each page to obtain the proper effect both at design time and at run time.
- You can use the TabControl, which has only the tab portion but offers no pages to host the information. In this case, you'll want to use one or more components to mimic the *page change* operation.

A third related class, the TabSheet, represents a single page of the PageControl. This is not a stand-alone component and is not available on the Component Palette. You create a TabSheet at design time by using the local menu of the PageControl or at run time by using methods of the same control.

### NOTE

Delphi still includes the Notebook, TabSet, and TabbedNotebook components introduced in early versions. Use these components only if you need to create a 16-bit version of an application. For any other purpose, the PageControl and TabControl components, which encapsulate Win32 common controls, provide a more modern user interface. Actually, in 32-bit versions of Delphi, the TabbedNotebook component was reimplemented using the Win32 PageControl internally, to reduce the code size and update the look.

## PageControls and TabSheets

As usual, instead of duplicating the Help system's list of properties and methods of the PageControl component, I've built an example that stretches its capabilities and allows you to change its behavior at run time. The example, called Pages, has a PageControl with three pages. The structure of the PageControl and of the other key components is listed here:

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
object PageControl1: TPageControl
  ActivePage = TabSheet1
  Align = alClient
```

```

HotTrack = True
Images = ImageList1
MultiLine = True
object TabSheet1: TTabSheet
  Caption = 'Pages'
  object Label3: TLabel
  object ListBox1: TListBox
end
object TabSheet2: TTabSheet
  Caption = 'Tabs Size'
  ImageIndex = 1
  object Label1: TLabel
  // other controls
end
object TabSheet3: TTabSheet
  Caption = 'Tabs Text'
  ImageIndex = 2
  object Memo1: TMemo
    Anchors = [akLeft, akTop, akRight, akBottom]
    OnChange = Memo1Change
  end
  object BitBtnChange: TBitBtn
    Anchors = [akTop, akRight]
    Caption = '&Change'
  end
end
end
object BitBtnPrevious: TBitBtn
  Anchors = [akRight, akBottom]
  Caption = '&Previous'
  OnClick = BitBtnPreviousClick
end
object BitBtnNext: TBitBtn
  Anchors = [akRight, akBottom]
  Caption = '&Next'
  OnClick = BitBtnNextClick
end
object ImageList1: TImageList
  Bitmap = {...}
end
end

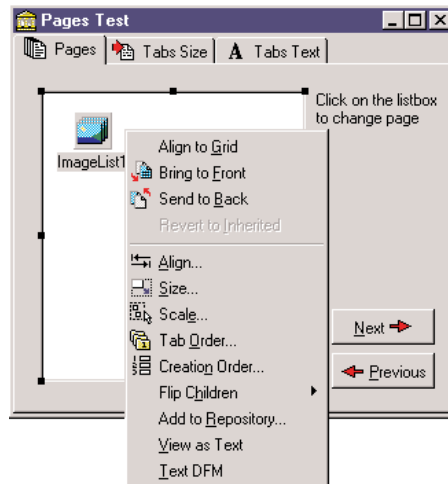
```

Notice that the tabs are connected to the bitmaps provided by an ImageList control and that some controls use the Anchors property to remain at a fixed distance from the right or bottom borders of the form. Even if the form doesn't support resizing (this would have been far too complex to set up with so many controls), the positions can change when the tabs are displayed on multiple lines (simply increase the length of the captions) or on the left side of the form.

Each TabSheet object has its own Caption, which is displayed as the sheet's tab. At design time, you can use the local menu to create new pages and to move between pages. You can see the local menu of the PageControl component in Figure 7.4, together with the first page. This page holds a list box and a small caption, and it shares two buttons with the other pages.

**FIGURE 7.4:**

The first sheet of the PageControl of the Pages example, with its local menu



If you place a component on a page, it is available only in that page. How can you have the same component (in this case, two bitmap buttons) in each of the pages, without duplicating it? Simply place the component on the form, outside of the PageControl (or before aligning it to the client area) and then move it in front of the pages, calling the Bring To Front command of the form's local menu. The two buttons I've placed in each page can be used to move back and forth between the pages and are an alternative to using the tabs. Here is the code associated with one of them:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
    PageControl1.SelectNextPage (True);
end;
```

The other button calls the same procedure, passing False as its parameter to select the previous page. Notice that there is no need to check whether we are on the first or last page, because the SelectNextPage method considers the last page to be the one before the first and will move you directly between those two pages.

Now we can focus on the first page again. It has a list box, which at run time will hold the names of the tabs. If a user clicks an item of this list box, the current page changes. This is the third method available to change pages (after the tabs and the Next and Previous buttons). The list box is filled in the FormCreate method, which is associated with the OnCreate

event of the form and copies the caption of each page (the Page property stores a list of TabSheet objects):

```
for I := 0 to PageControl1.PageCount - 1 do
    ListBox1.Items.Add (PageControl1.Pages.Caption);
```

When you click a list item, you can select the corresponding page:

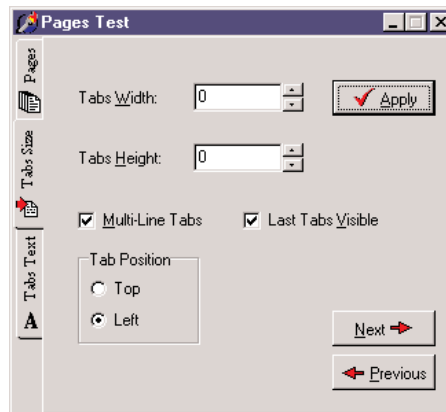
```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
    PageControl1.ActivePage := PageControl1.Pages [ListBox1.ItemIndex];
end;
```

The second page hosts two edit boxes (connected with two UpDown components), two check boxes, and two radio buttons, as you can see in Figure 7.5. The user can input a number (or choose it by clicking the up or down buttons with the mouse or pressing the Up or Down arrow key while the corresponding edit box has the focus), check the boxes and the radio buttons, and then click the Apply button to make the changes:

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
    // set tab width, height, and lines
    PageControl1.TabWidth := StrToInt (EditWidth.Text);
    PageControl1.TabHeight := StrToInt (EditHeight.Text);
    PageControl1.MultiLine := CheckBoxMultiLine.Checked;
    // show or hide the last tab
    TabSheet3.TabVisible := CheckBoxVisible.Checked;
    // set the tab position
    if RadioButton1.Checked then
        PageControl1.TabPosition := tpTop
    else
        PageControl1.TabPosition := tpLeft;
end;
```

**FIGURE 7.5:**

The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control.



With this code, we can change the width and height of each tab (remember that 0 means the size is computed automatically from the space taken by each string). We can choose to have either multiple lines of tabs or two small arrows to scroll the tab area, and move them to the left side. The control also allows tabs to be placed on the bottom or on the right, but our program doesn't allow that, because it would make the placement of the other controls quite complex.

You can also hide the last tab on the PageControl, which corresponds to the TabSheet3 component. If you hide one of the tabs by setting its TabVisible property to False, you cannot reach that tab by clicking on the Next and Previous buttons, which are based on the SelectNextPage method. Instead, you should use the FindNextPage function, as shown below in this new version of the Next button's OnClick event handler:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);  
begin  
    PageControl1.ActivePage := PageControl1.FindNextPage (  
        PageControl1.ActivePage, True, False);  
end;
```

The last page has a memo component, again with the names of the pages (added in the FormCreate method). You can edit the names of the pages and click the Change button to change the text of the tabs, but only if the number of strings matches the number of tabs:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);  
var  
    I: Integer;  
begin  
    if Memo1.Lines.Count <> PageControl1.PageCount then  
        MessageDlg ('One line per tab, please', mtError, [mbOK], 0)  
    else  
        for I := 0 to PageControl1.PageCount -1 do  
            PageControl1.Pages [I].Caption := Memo1.Lines [I];  
        BitBtnChange.Enabled := False;  
end;
```

Finally the last button, Add Page, allows you to add a new tab sheet to the page control, although the program doesn't add any components to it. The (empty) tab sheet object is created using the page control as its owner, but it won't work unless you also set the PageControl property. Before doing this, however, you should make the new tab sheet visible. Here is the code:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);  
var  
    strCaption: string;  
    NewTabSheet: TTabSheet;  
begin  
    strCaption := 'New Tab';  
    if InputQuery ('New Tab', 'Tab Caption', strCaption) then
```

```

begin
    // add a new empty page to the control
    NewTabSheet := TTabSheet.Create (PageControl1);
    NewTabSheet.Visible := True;
    NewTabSheet.Caption := strCaption;
    NewTabSheet.PageControl := PageControl1;
    PageControl1.ActivePage := NewTabSheet;
    // add it to both lists
    Memo1.Lines.Add (strCaption);
    ListBox1.Items.Add (strCaption);
end;
end;

```

**Tip**

Whenever you write a form based on a PageControl, remember that the first page displayed at run time is the page you were in before the code was compiled. This means that if you are working on the third page and then compile and run the program, it will start with that page. A common way to solve this problem is to add a line of code in the `FormCreate` method to set the PageControl or notebook to the first page. This way, the current page at design time doesn't determine the initial page at run time.

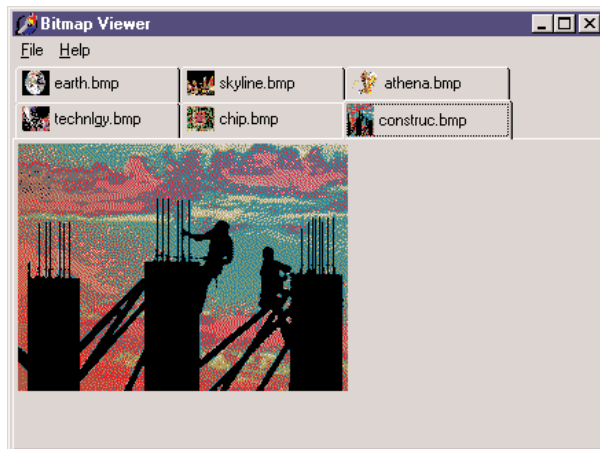
## An Image Viewer with Owner-Draw Tabs

The use of the TabControl and of a dynamic approach, as described in the last example, can also be applied in more general (and simpler) cases. Every time you need multiple pages that all have the same type of content, instead of replicating the controls in each page, you can use a TabControl and change its contents when a new tab is selected.

This is what I'll do in the multiple-page bitmap viewer example, called `BmpViewer`. The image that appears in the TabControl of this form, aligned to the whole client area, depends on the selection in the tab above it (as you can see in Figure 7.6).

**FIGURE 7.6:**

The interface of the bitmap viewer in the `BmpViewer` example. Notice the owner-draw tabs.



At the beginning, the `TabControl` is empty. After selecting `File > Open`, the user can choose various files in the File Open dialog box, and the array of strings with the names of the files (the `Files` property of the `OpenDialog1` component) is added to the tabs (the `Tabs` property of `TabControl1`):

```
procedure TFormBmpViewer.Open1Click(Sender: TObject);  
begin  
  if OpenDialog1.Execute then  
    begin  
      TabControl1.Tabs.AddStrings (OpenDialog1.Files);  
      TabControl1.TabIndex := 0;  
      TabControl1Change (TabControl1);  
    end;  
end;
```

After we display the new tabs, we have to update the image so that it matches the first tab. To accomplish this, the program calls the method connected with the `OnChange` event of the `TabControl`, which loads the file corresponding to the current tab in the image component:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);  
begin  
  Image1.Picture.LoadFromFile (TabControl1.Tabs [TabControl1.TabIndex]);  
end;
```

This example works, unless you select a file that doesn't contain a bitmap. The program will warn the user with a standard exception, ignore the file, and continue its execution.

The program also allows pasting the bitmap on the clipboard (without actually copying it, though) and copying the current bitmap to it. Clipboard support is available in Delphi via the global `Clipboard` object defined in the `ClipBrd` unit. For copying or pasting bitmaps, you can use the `Assign` method of the `TClipboard` and `TBitmap` classes. When you select the `Edit > Paste` command of the example, a new tab named `Clipboard` is added to the tab set (unless it is already present). Then the number of the new tab is used to change the active tab:

```
procedure TFormBmpViewer.Paste1Click(Sender: TObject);  
var  
  TabNum: Integer;  
begin  
  // try to locate the page  
  TabNum := TabControl1.Tabs.IndexOf ('Clipboard');  
  if TabNum < 0 then  
    // create a new page for the Clipboard  
    TabNum := TabControl1.Tabs.Add ('Clipboard');  
    // go to the Clipboard page and force repaint  
    TabControl1.TabIndex := TabNum;  
    TabControl1Change (Self);  
end;
```

The Edit ➤ Copy operation, instead, is as simple as copying the bitmap currently in the image control:

```
Clipboard.Assign (Image1.Picture.Graphic);
```

To account for the possible presence of the Clipboard tab, the code of the `TabControl1Change` method becomes:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
var
    TabText: string;
begin
    Image1.Visible := True;
    TabText := TabControl1.Tabs [TabControl1.TabIndex];
    if TabText <> 'Clipboard' then
        // load the file indicated in the tab
        Image1.Picture.LoadFromFile (TabText)
    else
        {if the tab is 'Clipboard' and a bitmap
         is available in the clipboard}
        if Clipboard.HasFormat (cf_Bitmap) then
            Image1.Picture.Assign (Clipboard)
        else
            begin
                // else remove the clipboard tab
                TabControl1.Tabs.Delete (TabControl1.TabIndex);
                if TabControl1.Tabs.Count = 0 then
                    Image1.Visible := False;
            end;
    end;
```

This program pastes the bitmap from the Clipboard each time you change the tab. The program stores only one image at a time, and it has no way to store the Clipboard bitmap. However, if the Clipboard content changes and the bitmap format is no longer available, the Clipboard tab is automatically deleted (as you can see in the listing above). If no more tabs are left, the Image component is hidden.

An image can also be removed using either of two menu commands: Cut or Delete. Cut removes the tab after making a copy of the bitmap to the Clipboard. In practice, the `Cut1Click` method does nothing besides calling the `Copy1Click` and `Delete1Click` methods. The `Copy1Click` method is responsible for copying the current image to the Clipboard, `Delete1Click` simply removes the current tab. Here is their code:

```
procedure TFormBmpViewer.Copy1Click(Sender: TObject);
begin
    Clipboard.Assign (Image1.Picture.Graphic);
end;
```



```

procedure TFormBmpViewer.Delete1Click(Sender: TObject);
begin
  with TabControl1 do
    begin
      if TabIndex >= 0 then
        Tabs.Delete (TabIndex);
      if Tabs.Count = 0 then
        Image1.Visible := False;
    end;
end;

```

One of the special features of the example is that the TabControl has the OwnerDraw property set to True. This means that the control won't paint the tabs (which will be empty at design time) but will have the application do this, by calling the OnDrawTab event. In its code, the program displays the text vertically centered, using the DrawText API function. The text displayed is not the entire file path but only the filename. Then, if the text is not *None*, the program reads the bitmap the tab refers to and paints a small version of it in the tab itself. To accomplish this, the program uses the TabBmp object, which is of type TBitmap and is created and destroyed along with the form. The program also uses the BmpSide constant to position the bitmap and the text properly:

```

procedure TFormBmpViewer.TabControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
  TabText: string;
  OutRect: TRect;
begin
  TabText := TabControl1.Tabs [TabIndex];
  OutRect := Rect;
  InflateRect (OutRect, -3, -3);
  OutRect.Left := OutRect.Left + BmpSide + 3;
  DrawText (Control.Canvas.Handle, PChar (ExtractFileName (TabText)),
    Length (ExtractFileName (TabText)), OutRect,
    dt_Left or dt_SingleLine or dt_VCenter);
  if TabText = 'Clipboard' then
    if Clipboard.HasFormat (cf_Bitmap) then
      TabBmp.Assign (Clipboard)
    else
      TabBmp.FreeImage
    else
      TabBmp.LoadFromFile (TabText);
  OutRect.Left := OutRect.Left - BmpSide - 3;
  OutRect.Right := OutRect.Left + BmpSide;
  Control.Canvas.StretchDraw (OutRect, TabBmp);
end;

```

The program has also support for printing the current bitmap, after showing a page preview form in which the user can select the proper scaling. This extra portion of the program I built for earlier editions of the book is not discussed in detail, but I've left the code in the program so that you can have a look at its code anyway.

## The User Interface of a Wizard

Just as you can use a `TabControl` without pages, you can also take the opposite approach and use a `PageControl` without tabs. What I want to focus on now is the development of the user interface of a wizard. In a wizard, you are directing the user through a sequence of steps, one screen at a time, and at each step you typically want to offer the choice of proceeding to the next step or going back to correct input entered in a previous step. So instead of tabs that can be selected in any order, wizards typically offer `Next` and `Back` buttons to navigate. This won't be a complex example; its purpose is just to give you a few guidelines. The example is called `WizardUI`.

The starting point is to create a series of pages in a `PageControl` and set the `TabVisible` property of each `TabSheet` to `False` (while keeping the `Visible` property set to `True`). Unlike past versions, since Delphi 5 you can also hide the tabs at design time. In this case, you'll need to use the shortcut menu of the page control or the combo box of the Object Inspector to move to another page, instead of the tabs. But why don't you want to see the tabs at design time? You can place controls on the pages and then place extra controls in front of the pages (as I've done in the example), without seeing their relative positions change at run time. You might also want to remove the useless captions of the tabs, which take up space in memory and in the resources of the application.

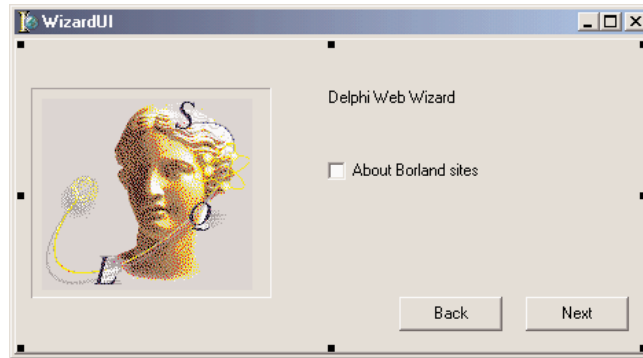
In the first page, I've placed on one side an image and a bevel control and on the other side some text, a check box, and two buttons. Actually, the `Next` button is inside the page, while the `Back` button is over it (and is shared by all the pages). You can see this first page at design time in Figure 7.7. The following pages look similar, with a label, check boxes, and buttons on the right side and nothing on the left.

When you click the `Next` button on the first page, the program looks at the status of the check box and decides which page is the following one. I could have written the code like this:

```
procedure TForm1.btnNext1Click(Sender: TObject);  
begin  
  BtnBack.Enabled := True;  
  if CheckInprise.Checked then  
    PageControl1.ActivePage := TabSheet2  
  else  
    PageControl1.ActivePage := TabSheet3;  
    // move image and bevel  
    Bevel1.Parent := PageControl1.ActivePage;  
    Image1.Parent := PageControl1.ActivePage;  
end;
```

**FIGURE 7.7:**

The first page of the WizardUI example at design time



After enabling the common Back button, the program changes the active page and finally moves the graphical portion to the new page. Because this code has to be repeated for each button, I've placed it in a method after adding a couple of extra features. This is the actual code:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
    if CheckInprise.Checked then
        MoveTo (TabSheet2)
    else
        MoveTo (TabSheet3);
end;

procedure TForm1.MoveTo(TabSheet: TTabSheet);
begin
    // add the last page to the list
    BackPages.Add (PageControl1.ActivePage);
    BtnBack.Enabled := True;
    // change page
    PageControl1.ActivePage := TabSheet;
    // move image and bevel
    Bevel1.Parent := PageControl1.ActivePage;
    Image1.Parent := PageControl1.ActivePage;
end;
```

Besides the code I've already explained, the MoveTo method adds the last page (the one before the page change) to a list of visited pages, which behaves like a stack. In fact, the BackPages object of the TList class is created as the program starts and the last page is always added to the end. When the user clicks the Back button, which is not dependent on the page, the program extracts the last page from the list, deletes its entry, and moves to that page:

```
procedure TForm1.btnBackClick(Sender: TObject);
var
```

```

    LastPage: TTabSheet;
begin
    // get the last page and jump to it
    LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
    PageControl1.ActivePage := LastPage;
    // delete the last page from the list
    BackPages.Delete (BackPages.Count - 1);
    // eventually disable the back button
    BtnBack.Enabled := not (BackPages.Count = 0);
    // move image and bevel
    Bevel1.Parent := PageControl1.ActivePage;
    Image1.Parent := PageControl1.ActivePage;
end;

```

With this code, the user can move back several pages until the list is empty, at which point we disable the Back button. The complication we need to deal with is that while moving from a particular page, we know which pages are its “next” and “previous,” but we don’t know which page we came from, because there can be multiple paths to reach a page. Only by keeping track of the movements with a list can we reliably go back.

The rest of the code of the program, which simply shows some Web site addresses, is very simple. The good news is that you can reuse the navigational structure of this example in your own programs and modify only the graphical portion and the content of the pages. Actually, as most of the labels of the programs show HTTP addresses, a user can click those labels to open the default browser showing that page. This is accomplished by extracting the HTTP address from the label and calling the `ShellExecute` function.

```

procedure TForm1.LabelLinkClick(Sender: TObject);
var
    Caption, StrUrl: string;
begin
    Caption := (Sender as TLabel).Caption;
    StrUrl := Copy (Caption, Pos ('http://', Caption), 1000);
    ShellExecute (Handle, 'open', PChar (StrUrl), '', '', sw_Show);
end;

```

The method above is hooked to the `OnClick` event of many labels of the form, which have been turned into *links* by setting its `Cursor` to a hand. This is one of the labels:

```

object Label12: TLabel
    Cursor = crHandPoint
    Caption = 'Main site: http://www.borland.com'
    OnClick = LabelLinkClick
end

```

## Form-Splitting Techniques

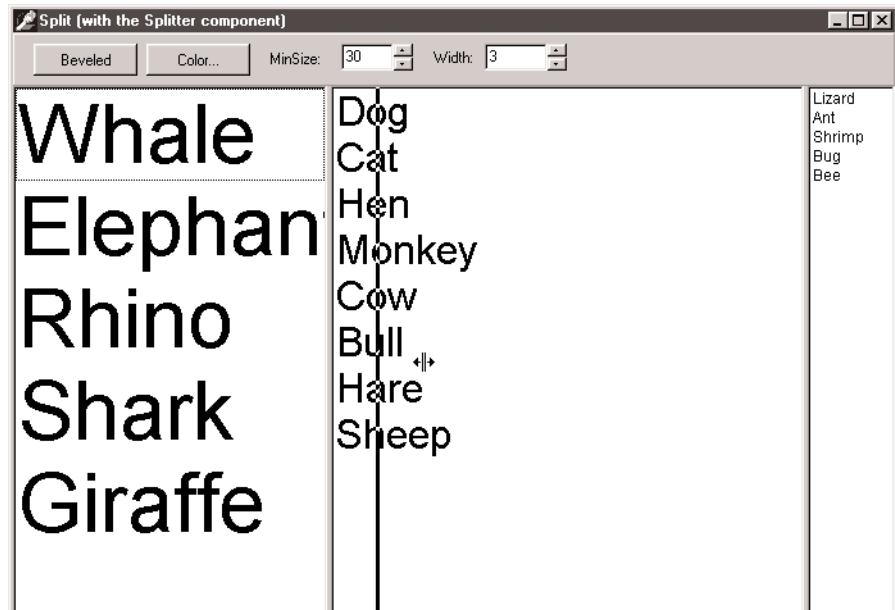
There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the Constraints property of the controls it relates to. As we'll see in the Split1 example, this allows us to define maximum and minimum positions of the splitter and of the form.

To build this example, simply place a ListBox component in a form; then add a Splitter component, a second ListBox, another Splitter, and finally a third ListBox component. The form also has a simple toolbar based on a panel.

By simply placing these two splitter components, you give your form the complete functionality of moving and sizing the controls it hosts at run time. The Width, Beveled, and Color properties of the splitter components determine their appearance, and in the Split1 example you can use the toolbar controls to change them. Another relevant property is MinSize, which determines the minimum size of the components of the form. During the splitting operation (see Figure 7.8), a line marks the final position of the splitter, but you cannot drag this line beyond a certain limit. The behavior of the Split1 program is not to let controls become too small. An alternative technique is to set the new AutoSnap property of the splitter to True. This property will make the splitter hide the control when its size goes below the MinSize limit.

**FIGURE 7.8:**

The splitter component of the Split1 example determines the minimum size for each control on the form, even those not adjacent to the splitter itself.



I suggest you try using the `Split1` program, so that you'll fully understand how the splitter affects its adjacent controls and the other controls of the form. Even if we set the `MinSize` property, a user of this program can reduce the size of its entire form to a minimum, hiding some of the list boxes. If you test the `Split2` version of the example, instead, you'll get better behavior. In `Split2`, I've set some `Constraints` for the `ListBox` controls—for example,

```
object ListBox1: TListBox
    Constraints.MaxHeight = 400
    Constraints.MinHeight = 200
    Constraints.MinWidth = 150
```

The size constraints are applied only as you actually resize the controls, so to make this program work in a satisfactory way, you have to set the `ResizeStyle` property of the two splitters to `rsUpdate`. This value indicates that the position of the controls is updated for every movement of the splitter, not only at the end of the operation. If you select the `rsLine` or the new `rsPattern` values, instead, the splitter simply draws a line in the required position, checking the `MinSize` property but not the constraints of the controls.

**TIP**

When you set the `Splitter` component's `AutoSnap` property to `True`, the splitter will completely hide the neighboring control when the size of that control is below the minimum set for it in the `Splitter` component.

## Horizontal Splitting

The `Splitter` component can also be used for horizontal splitting, instead of the default vertical splitting. However, this approach is a little more complicated. Basically you can place a component on a form, align it to the top, and then place the splitter on the form. By default, it will be left aligned. Choose the `alTop` value for the `Align` property, and then resize the component manually, by changing the `Height` property in the `Object Inspector` (or by resizing the component).

You can see a form with a horizontal splitter in the `SplitH` example. This program has two memo components you can open a file into, and it has a splitter dividing them, defined as:

```
object Splitter1: TSplitter
    Cursor = crVSplit
    Align = alTop
    OnMoved = Splitter1Moved
end
```

When you double-click a memo, the program loads a text file into it (notice the structure of the `with` statement):

```
procedure TForm1.MemoDb1Click(Sender: TObject);
begin
```

```
with Sender as TMemo, OpenFileDialog do
  if Execute then
    Lines.LoadFromFile (FileName);
end;
```

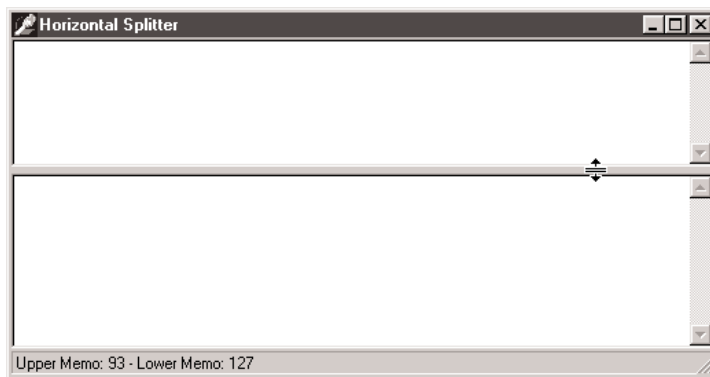
The program features a status bar, which keeps track of the current height of the two memo components. It handles the `OnMoved` event of the splitter (the only event of this component) to update the text of the status bar. The same code is executed whenever the form is resized:

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format ('Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

You can see the effect of this code by looking at Figure 7.9, or by running the `SplitH` example.

**FIGURE 7.9:**

The status bar of the `SplitH` example indicates the position of the horizontal splitter component.



## Splitting with a Header

An alternative to using splitters is to use the standard `HeaderControl` component. If you place this control on a form, it will be automatically aligned with the top of the form. Then you can add the three list boxes to the rest of the client area of the form. The first list box can be aligned on the left, but this time you cannot align the second and third list box as well. The problem is that the sections of the header can be dragged outside the visible surface of the form. If the list boxes use automatic alignment, they cannot move outside the visible surface of the form, as the program requires.

The solution is to define the sections of the header, using the specific editor of the `Sections` property. This property editor allows you to access the various subobjects of the collection, changing various settings. You can set the caption and alignment of the text; the current, minimum, and maximum size of the header; and so on. Setting the limit values is a powerful tool,

and it replaces the `MinSize` property of the splitter or the constraints of the list boxes we've used in past examples. You can see the output of this program, named `HdrSplit`, in Figure 7.10.

**FIGURE 7.10:**

The output of the `HdrSplit` example



We need to handle two events: `OnSectionResize` and `OnSectionClick`. The first handler simply resizes the list box connected with the modified section (determined by associating numbers with the `ImageIndex` property of each section and using it to determine the name of the list box control):

```
procedure TForm1.HeaderControl1SectionResize(
    HeaderControl: THeaderControl; Section: THeaderSection);
var
    List: TListBox;
begin
    List := FindComponent ('ListBox' + IntToStr (Section.ImageIndex))
        as TListBox;
    List.Width := Section.Width;
end;
```

Along with this event, we need to handle the resizing of the form, using it to synchronize the list boxes with the sections, which are all resized by default:

```
procedure TForm1.FormResize(Sender: TObject);
var
    I: Integer;
    List: TListBox;
begin
    for I := 0 to 2 do
        begin
            List := FindComponent ('ListBox' + IntToStr (
                HeaderControl1.Sections[I].ImageIndex)) as TListBox;
```



```

    List.Left := HeaderControl1.Sections[I].Left;
    List.Width := HeaderControl1.Sections[I].Width;
  end;
end;

```

After setting the height of the list boxes, this method simply calls the previous one, passing parameters that we won't use in this example. The second method of the HeaderControl, called in response to a click on one of the sections, is used to sort the contents of the corresponding list box:

```

procedure TForm1.HeaderControl1SectionClick(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (Section.ImageIndex))
    as TListBox;
  List.Sorted := not List.Sorted;
end;

```

Of course, this code doesn't provide the common behavior of sorting the elements when you click the header and then sorting them in the reverse order if you click again. To implement this, you should write your own sorting algorithm.

Finally, the HdrSplit example uses a new feature for the header control. It sets the `DragReorder` property to enable dragging operations to reorder the header sections. When this operation is performed, the control fires the `OnSectionDrag` event, where you can exchange the positions of the list boxes. This event fires before the sections are actually moved, so I have to use the coordinates of the other section:

```

procedure TForm1.HeaderControl1SectionDrag(Sender: TObject; FromSection,
  ToSection: THeaderSection; var AllowDrag: Boolean);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (FromSection.ImageIndex))
    as TListBox;
  List.Left := ToSection.Left;
  List.Width := ToSection.Width;

  List := FindComponent ('ListBox' + IntToStr (ToSection.ImageIndex))
    as TListBox;
  List.Left := FromSection.Left;
  List.Width := FromSection.Width
end;

```

## Control Anchors

In this chapter, I've described how you can use alignment and splitters to create nice, flexible user interfaces, that adapt to the current size of the form, giving users maximum freedom. Delphi also supports right and bottom anchors. Before this feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom, unless it was aligned to the bottom or right sides. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to have a button anchored to the bottom-right corner of the form, place the button in the required position and set its `Anchors` property to `[akRight, akBottom]`. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

On the other hand, if you place a large component such as a `Memo` or a `Listbox` in the middle of a form, you can set its `Anchors` property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

---

**TIP**

Anchors, like constraints, work both at design time and at run time, so you should set them up as early as possible, to benefit from this feature while you're designing the form as well as at run time.

As an example of both approaches, you can try out the `Anchors` application, which has two buttons on the bottom-right corner and a list box in the middle. As shown in Figure 7.11, the controls automatically move and stretch as the form size changes. To make this form work properly, you must also set its `Constraints` property; otherwise, as the form becomes too small the controls can overlap or disappear.

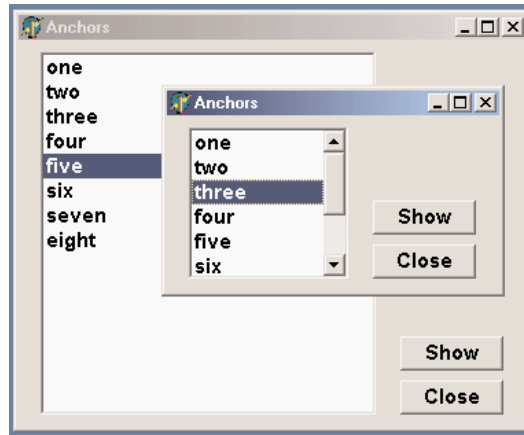
---

**TIP**

If you remove all of the anchors, or two opposite ones (for example, left and right), the resize operations will cause the control to float. The control keeps its current size, and the system adds or removes the same number of pixels on each side of it. This can be defined as a centered anchor, because if the component is initially in the middle of the form it will keep that position. In any case, if you want a centered control, you should generally use both opposite anchors, so that if the user makes the form larger, the control size will grow as well. In the case just presented, in fact, making the form larger leaves a small control in its center.

**FIGURE 7.11:**

The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only a proper use of the Anchors property.



## The ToolBar Control

In early versions of Delphi, toolbars had to be created using panels and speed buttons. Starting with version 3, Delphi introduced a specific ToolBar component, which encapsulates the corresponding Win32 common control or the corresponding Qt widget in VisualCLX. This component provides a toolbar, with its own buttons, and it has many advanced capabilities. To use this component, you place it on a form and then use the component editor (the shortcut menu activated by a right mouse button click) to create a few buttons and separators.

### Building a Toolbar with a Panel

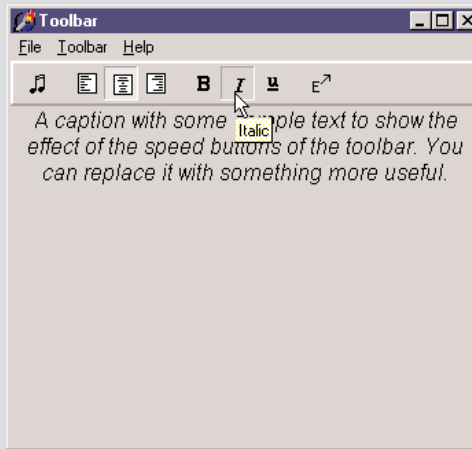
Before the toolbar control was available in Delphi, the standard approach for building a toolbar was to use a panel aligned to the top of the form and place SpeedButton components inside it. A *speed button* is a lightweight graphical control (consuming no Windows resources); it cannot receive the input focus, it has no tab order, and it is faster to create and paint than a bitmap button.

Speed buttons can behave like push buttons, check boxes, or radio buttons, and they can have different bitmaps depending on their status. To make a group of speed buttons work like radio buttons, just place some speed buttons on the panel, select all of them, and give the same value to each one's `GroupIndex` property. All the buttons having the same `GroupIndex` become mutually exclusive selections. One of these buttons should always be selected, so remember to set the `Down` property to `True` for one of them at design time or as soon as the program starts.

*Continued on next page*

By setting the `AllowAllUp` property, you can create a group of mutually exclusive buttons, each of which can be *up*—that is, a group from which the user can select one option or leave them all unselected. As a special case, you can make a speed button work as a check box, simply by defining a group (the `GroupIndex` property) that has only one button and that allows it to be deselected (the `AllowAllUp` property).

Finally, you can set the `Flat` property of all the `SpeedButton` components to `True`, obtaining a more modern user interface. If you are interested in this approach, you can look at the `PanelBar` example, illustrated here:



The use of `SpeedButton` controls is becoming less common. Besides the fact that the `ToolBar` control is very handy and definitely more standard, speed buttons have two big problems. First, each of them requires a specific bitmap and cannot use one from an image list (unless you write some complex code). Second, speed buttons don't work very well with actions, because some properties, such as the `Down` state, do not map directly.

The toolbar is populated with objects of the `TToolBarButton` class. These objects have a fundamental property, `Style`, which determines their behavior:

- The `tbsButton` style indicates a standard push button.
- The `tbsCheck` style indicates a button with the behavior of a check box, or that of a radio button if the button is `Grouped` with the others in its block (determined by the presence of separators).

- The `tbsDropDown` style indicates a drop-down button, a sort of combo box. The drop-down portion can be easily implemented in Delphi by connecting a `PopupMenu` control to the `DropDownMenu` property of the control.
- The `tbsSeparator` and `tbsDivider` styles indicate separators with no or different vertical lines (depending on the `Flat` property of the toolbar).

To create a graphic toolbar, you can add an `ImageList` component to the form, load some bitmaps into it, and then connect the `ImageList` with the `Images` property of the toolbar. By default, the images will be assigned to the buttons in the order they appear, but you can change this quite easily by setting the `ImageIndex` property of each toolbar button. You can prepare further `ImageLists` for special conditions of the buttons and assign them to the `DisabledImages` and `HotImages` properties of the toolbar. The first group is used for the disabled buttons; the second for the button currently under the mouse.

**NOTE**

In a nontrivial application, you would generally create toolbars using an `ActionList` or the new Action Manager architecture, both discussed in the next chapter. In this case, you'll attach very little behavior to the toolbar buttons, as their properties and events will be managed by the action components.

## The RichBar Example

As an example of the use of a toolbar, I've built the `RichBar` application, which has a `RichEdit` component you can operate by using the toolbar. The program has buttons for loading and saving files, for copy and paste operations, and to change some of the attributes of the current font.

I don't want to cover the details of the features of the `RichEdit` control, which are many, nor discuss the details of this application, which has quite a lot of code. All I want to do is to focus on features specific to the `ToolBar` used by the example and visible in Figure 7.12. This toolbar has buttons, separators, and even a drop-down menu and two combo boxes discussed in the next section.

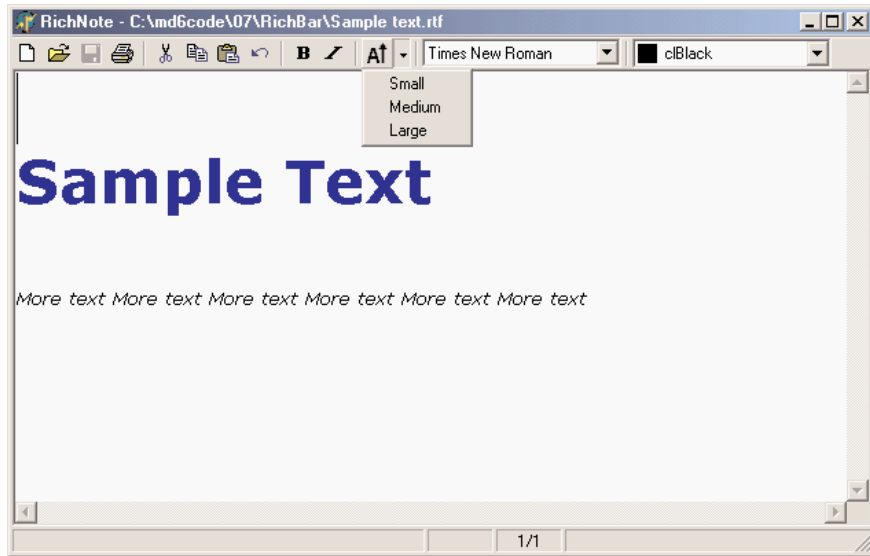
The various buttons implement features, one of them being a complete scheme for opening and saving the text files, including the ability to ask the user to save any modified file before opening a new one, to avoid losing any changes. The file-handling portion of the program is quite complex, but it is worth exploring, as many file-based applications will use similar code. I've made more details available in the bonus chapter "The RichBar Example" on the companion CD.

Besides file operations, the program supports copy and paste operations and font management. The copy and paste operations don't require an actual interaction with the clipboard, as the component can handle them with simple commands, such as:

```
RichEdit.CutToClipboard;
RichEdit.CopyToClipboard;
RichEdit.PasteFromClipboard;
RichEdit.Undo;
```

**FIGURE 7.12:**

The toolbar of the RichBar example. Notice the drop-down menu.



It is a little more advanced to know when these operations (and the corresponding buttons) should be enabled. We can enable Copy and Cut buttons when some text is selected, in the `OnSelectionChange` event of the `RichEdit` control:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
    btnCut.Enabled := RichEdit.SelLength > 0;
    btnCopy.Enabled := btnCut.Enabled;
end;
```

The Copy operation, instead, cannot be determined by an action of the user, as it depends on the content of the Clipboard, influenced also by other applications. One approach is to use a timer and check the clipboard content from time to time. A better approach is to use the `OnIdle` event of the `Application` object (or the `ApplicationEvents` component). As the

RichEdit supports multiple clipboard formats, the code cannot simply look at those, but should ask the component itself, using a low-level feature not surfaced by the Delphi control:

```
procedure TFormRichNote.ApplicationEvents1Idle(Sender: TObject);  
  var Done: Boolean);  
begin  
  // update toolbar buttons  
  tbtnPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;  
end;
```

Basic font management is given by the Bold and Italic buttons, which have similar code. The Bold button toggles the relative attribute from the selected text (or changes the style at the current edit position):

```
procedure TFormRichNote.BoldExecute(Sender: TObject);  
begin  
  with RichEdit.SelAttributes do  
    if fsBold in Style then  
      Style := Style - [fsBold]  
    else  
      Style := Style + [fsBold];  
end;
```

Again, the current status of the button is determined by the current selection, so we'll need to add the following line to the RichEditSelectionChange method:

```
tbtnBold.Down := fsBold in RichEdit.SelAttributes.Style;
```

## A Menu and a Combo Box in a Toolbar

Besides a series of buttons, the RichBar example has a drop-down menu and a couple of combo boxes, a feature shared by many common applications. The drop-down button allows selection of the font size, while the combo boxes allow rapid selection of the font family and the font color. This second combo is actually built using a ColorBox control.

The Size button is connected to a PopupMenu component (called SizeMenu) using the DropDownMenu property. A user can press the button, firing its OnClick event as usual, or select the drop-down arrow, open the pop-up menu (see again Figure 7.12), and choose one of its options. This case has three possible font sizes, per the menu definition:

```
object SizeMenu: TPopupMenu  
  object Small1: TMenuItem  
    Tag = 10  
    Caption = 'Small'  
    OnClick = SetFontSize  
  end  
  object Medium1: TMenuItem  
    Tag = 16
```

```

    Caption = 'Medium'
    OnClick = SetFontSize
end
object Large1: TMenuItem
    Tag = 32
    Caption = 'Large'
    OnClick = SetFontSize
end
end

```

Each menu item has a tag indicating the actual size of the font, activated by a shared event handler:

```

procedure TFormRichNote.SetFontSize(Sender: TObject);
begin
    RichEdit.SelAttributes.Size := (Sender as TMenuItem).Tag;
end;

```

As the `ToolBar` control is a full-featured control container, you can directly take an edit box, a combo box, and other controls and place them inside the toolbar. The combo box in the toolbar is initialized in the `FormCreate` method, which extracts the screen fonts available in the system:

```

ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (RichEdit.Font.Name)

```

The combo box initially displays the name of the default font used in the `RichEdit` control, which is set at design time. This value is recomputed each time the current selection changes, using the font of the selected text, along with the current color for the `ColorBox`:

```

procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
    ComboFont.ItemIndex :=
        ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name);
    ColorBox1.Selected := RichEdit.SelAttributes.Color;
end;

```

When a new font is selected from the combo box, the reverse action takes place. The text of the current combo box item is assigned as the name of the font for any text selected in the `RichEdit` control:

```

RichEdit.SelAttributes.Name := ComboFont.Text;

```

The selection of a color in the `ColorBox` activates similar code.

## Toolbar Hints

Another common element in toolbars is the *fly-by hint*, also called *balloon help*—some text that briefly describes the button currently under the cursor. This text is usually displayed in a yel-



low box after the mouse cursor has remained steady over a button for a set amount of time. To add hints to an application's toolbar, simply set its `ShowHints` property to `True` and enter some text for the `Hint` property of each button (more on hints text in the next section, "A Simple Status Bar").

If you want to have more control on how hints are displayed, you can use some of the properties and events of the `Application` object. This global object has, among others, the following properties:

<b>Property</b>	<b>Defines</b>
<code>HintColor</code>	The background color of the hint window
<code>HintPause</code>	How long the cursor should remain on a component before hints are displayed
<code>HintHidePause</code>	How long the hint will be displayed
<code>HintShortPause</code>	How long the system should wait to display a hint if another hint has just been displayed

A program, for example, might allow a user to customize the hint background color by selecting a specific with the following code:

```
ColorDialog.Color := Application.HintColor;  
if ColorDialog.Execute then  
    Application.HintColor := ColorDialog.Color;
```

**NOTE**

As an alternative, you can change the hint color by handling the `OnShowHint` property of the `Application` object. This handler can change the color of the hint just for specific controls. The `OnShowHint` event is used in the `CustHint` example described later in this chapter.

## A Simple Status Bar

Building a status bar is even simpler than building a toolbar. Delphi includes a specific `StatusBar` component, based on the corresponding Windows common control (a similar control is available also in `VisualCLX`). This component can be used almost as a panel when its `SimplePanel` property is set to `True`. In this case, you can use the `SimpleText` property to output some text. The real advantage of this component, however, is that it allows you to define a number of subpanels just by activating the editor of its `Panels` property. (You can also display this property editor by double-clicking the status bar control.) Each subpanel has its own graphical attributes, which you can customize using the editor. Another feature of the status bar component is the "size grip" area added to the lower-right corner of the bar, which is useful for resizing the form itself. This is a typical element of the Windows user interface, and you can control it with the `SizeGrip` property.

There are various uses for a status bar. The most common is to display information about the menu item currently selected by the user. Besides this, a status bar often displays other information about the status of a program: the position of the cursor in a graphical application, the current line of text in a word processor, the status of the lock keys, the time and date, and so on. To show information on a panel, you simply use its `Text` property, generally using an expression like this:

```
StatusBar1.Panels[1].Text := 'message';
```

In the `RichBar` example, I've built a status bar with three panels, for command hints, the status of the Caps Lock key, and the current editing position. The `StatusBar` component of the example actually has four panels; we need to define the fourth in order to delimit the area of the third panel. The last panel, in fact, is always large enough to cover the remaining surface of the status bar.

**TIP**

Again, for more detail on the `RichBar` program, see the bonus chapter "The `RichBar` Example" on the companion CD.

The panels are not independent components, so you cannot access them by name, only by position as in the preceding code snippet. A good solution to improve the readability of a program is to define a constant for each panel you want to use, and then use these constants when referring to the panels. This is my sample code:

```
const
  sbpMessage = 0;
  sbpCaps = 1;
  sbpPosition = 2;
```

In the first panel of the status bar, I want to display the hint message of the toolbar button. The program obtains this effect by handling the application's `OnHint` event, again using the `ApplicationEvents` component, and copying the current value of the application's `Hint` property to the status bar:

```
procedure TFormRichNote.ApplicationEvents1Hint (Sender: TObject);
begin
  StatusBar1.Panels[sbpMessage].Text := Application.Hint;
end;
```

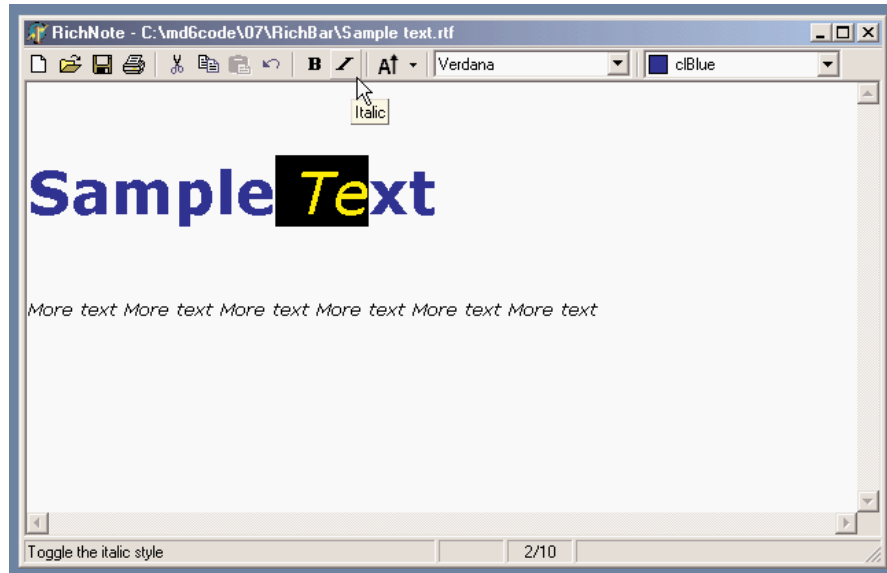
By default, this code displays in the status bar the same text of the fly-by hints. Actually, we can use the `Hint` property to specify different strings for the two cases, by writing a string divided into two portions by a separator, the pipe (`|`) character. For example, you might enter the following as the value of the `Hint` property:

```
'New|Create a new document'
```

The first portion of the string, *New*, is used by fly-by hints, and the second portion, *Create a new document*, by the status bar. You can see an example in Figure 7.13.

**FIGURE 7.13:**

The StatusBar of the RichBar example displays a more detailed description than the fly-by hint.

**TIP**

When the hint for a control is made up of two strings, you can use the `GetShortHint` and `GetLongHint` methods to extract the first (short) and second (long) substrings from the string you pass as a parameter, which is usually the value of the `Hint` property.

The second panel displays the status of the Caps Lock key, obtained by calling the `GetKeyState` API function, which returns a state number. If the low-order bit of this number is set (that is, if the number is odd), then the key is pressed. When do we check this state? I've decided to do this when the application is idle, so that this test is executed every time a key is pressed, but also as soon as a message reaches the window (in case the user changes this setting while working with another program). I've added to the `ApplicationEvents1Idle` handler a call to the custom `CheckCapsLock` method, implemented as follows:

```

procedure TFormRichNote.CheckCapsLock;
begin
    if Odd (GetKeyState (VK_CAPITAL)) then
        StatusBar1.Panels[spbCaps].Text := 'CAPS'
    else
        StatusBar1.Panels[spbCaps].Text := '';
end;

```

Finally, the program uses the third panel to display the current cursor position (measured in lines and characters per line) every time the selection changes. Because the `CaretPos`

values are zero-based (that is, the upper-left corner is line 0, character 0), I've decided to add one to each value, to make them more reasonable for a casual user:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
    ...
    // update the position in the status bar
    StatusBar.Panels[sbpPosition].Text := Format ('%d/%d',
        [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```

## Customizing the Hints

Just as we have added hints to an application's toolbar, we can add hints to forms or to the components of a form. For a large control, the hint will show up near the mouse cursor. In some cases, it is important to know that a program can freely customize how hints are displayed.

The simplest thing you can do is, change the value of the properties of the `Application` object as I mentioned at the end of the last section. To obtain more control over hints, you can customize them even further by assigning a method to the application's `OnShowHint` event. You need to either hook them up manually or—better—add an `ApplicationEvents` component to the form and handle its `OnShowHint` event.

The method you have to define has some interesting parameters, such as a string with the text of the hint, a Boolean flag for its activation, and a `THintInfo` structure with further information, including the control, the hint position, and its color. Each of the parameters is passed by reference, so you have a chance to change them and also modify the values of the `THintInfo` structure; for example, you can change the position of the hint window before it is displayed.

This is what I've done in the `CustHint` example, which shows the hint of the label at the center of its area. Here is what you can write to show the hint for the big label in the center of its surface:

```
procedure TForm1.ShowHint (var HintStr: string; var CanShow: Boolean;
    var HintInfo: THintInfo);
begin
    with HintInfo do
        if HintControl = Label1 then
            HintPos := HintControl.ClientToScreen (Point (
                HintControl.Width div 2, HintControl.Height div 2));
end;
```

The code has to retrieve the center of the generic control (the `HintInfo.HintControl`) and then convert its coordinates to screen coordinates, applying the `ClientToScreen` method to the control itself. We can further update the `CustHint` example in a different way. The `RadioGroup`

control in the form has three radio buttons. However, these are not stand-alone components, but simply radio button clones painted on the surface of the radio group. What if we want to add a hint for each of them?

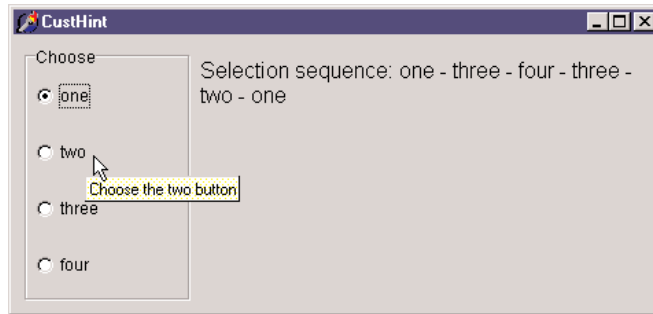
The `CursorRect` field of the `THintInfo` record can be used for this purpose. It indicates the area of the component that the cursor can move over without disabling the hint. When the cursor moves outside this area, Delphi hides the hint window. If we specify a different text for the hint and a different area for each of the radio buttons, we can in practice provide three different hints. Because computing the actual position of each radio button isn't easy, I've simply divided the surface of the radio group into as many equal parts as there are radio buttons. The text of the radio button (not the selected item, but the item under the cursor) is then added to the text of the hint:

```
procedure TForm1.ShowHint (var HintStr: string;
var CanShow: Boolean; var HintInfo: THintInfo);
var
  RadioItem, RadioHeight: Integer;
  RadioRect: TRect;
begin
  with HintInfo do
    if HintControl = Label1 ... // as before
    else
      if HintControl = RadioGroup1 then
        begin
          RadioHeight := (RadioGroup1.Height) div RadioGroup1.Items.Count;
          RadioItem := CursorPos.Y div RadioHeight;
          HintStr := 'Choose the ' + RadioGroup1.Items [RadioItem] + ' button';
          RadioRect := RadioGroup1.ClientRect;
          RadioRect.Top := RadioRect.Top + RadioHeight * RadioItem;
          RadioRect.Bottom := RadioRect.Top + RadioHeight;
          // assign the hints rect and pos
          CursorRect := RadioRect;
        end;
    end;
```

The final part of the code builds the rectangle for the hint, starting with the rectangle corresponding to the client area of the component and moving its `Top` and `Bottom` values to the proper section of the `RadioGroup1` component. The resulting effect is that each radio button of the radio group appears to have a specific hint, as shown in Figure 7.14.

**FIGURE 7.14:**

The RadioGroup control of the CustHint example shows a different hint, depending on which radio button the mouse is over.



## What's Next?

In this chapter I've discussed the use of some Delphi common controls, including the ListView, TreeView, PageControl, TabControl, ToolBar, StatusBar, and RichEdit. For each of these controls, I've built one example, trying to discuss it in the context of an actual application, even if most of the programs have been quite simple. I've also covered the Splitter component and various form-splitting techniques, the anchors for control positioning, and the customization of hints.

What is still missing is the development of an application with a complete user interface, including a menu and one or more toolbars. The reason I haven't covered this topic in the current chapter is that Delphi 6 adds quite a lot to VCL in this respect, including a complete architecture for letting the end users configure menus and toolbars based on a number of predefined actions. As this topic and related ones, such as docking toolbars, are complex, I've devoted the entire next chapter to them.

After this step, we'll move to the development of applications with multiple forms, including advanced dialog boxes, MDI, visual form inheritance, and the use of frames. All these topics are covered in Chapters 9 and 10.

# Building the User Interface

- Actions and ActionList
- Predefined actions in Delphi 6
- The ControlBar and CoolBar components
- Docking toolbars and other controls
- The Action Manager architecture

**M**odern Windows applications usually have multiple ways of giving a command, including menu items, toolbar buttons, shortcut menus, and so on. To separate the actual commands a user can give from their multiple representations in the user interface, Delphi has the idea of actions. In Delphi 6 this architecture has been largely extended to make the construction of the user interface on top of actions totally visual. You can now also easily let the user of your programs customize this interface, as happens in many professional programs.

This chapter focuses on actions, action lists and action managers, and the related components. It also covers a few related topics, such as toolbar container controls and toolbar docking, and docking in general.

## The ActionList Component

Delphi's event architecture is very open: You can write a single event handler and connect it to the `OnClick` events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, as the event handler can use the `Sender` parameter to refer to the object that fired the event. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

To overcome this problem, Delphi 4 introduced an event-handling architecture based on actions. An *action* (or command) both indicates the operation to do when a menu item or button is clicked and determines the status of all the elements connected to the action. The connection of the action with the user interface of the linked controls is very important and should not be underestimated, because it is where you can get the real advantages of this architecture.

### NOTE

If you have ever written code using the MFC class library of Visual C++, you'll recognize that a Delphi action maps to both a command and a `CCommandUpdateUI` object. The Delphi architecture is more flexible, though, because it can be extended by subclassing the action classes.

There are many players in this event-handling architecture. The central role is certainly played by the action objects. An action object has a name, like any other component, and other properties that will be applied to the linked controls (called action clients). These properties include the `Caption`, the graphical representation (`ImageIndex`), the status (`Checked`, `Enabled`, and `Visible`), and the user feedback (`Hint` and `HelpContext`). There is also the `Shortcut` and a list of `SecondaryShortCuts`, the `AutoCheck` property for two-state actions, the help support, and a `Category` property used to arrange actions in logical groups.



The base class for an all action object is `TBasicAction`, which introduces the abstract core behavior of an action, without any specific binding or correction (not even to menu items or controls). The derived `TContainedAction` class introduces properties and methods that enable actions to appear in an action list or action manager. The further-derived `TCustomAction` class introduces support for the properties and methods of menu items and controls that are linked to action objects. Finally, there is the derived ready-to-use `TAction` class.

Each action object is connected to one or more client objects through an `ActionLink` object. Multiple controls, possibly of different types, can share the same action object, as indicated by their `Action` property. Technically, the `ActionLink` objects maintain a bidirectional connection between the client object and the action. The `ActionLink` object is required because the connection works in both directions. An operation on the object (such as a click) is forwarded to the action object and results in a call to its `OnExecute` event; an update to the status of the action object is reflected in the connected client controls. In other words, one or more client controls can create an `ActionLink`, which registers itself with the action object.

You should not set the properties of the client controls you connect with an action, because the action will override the property values of the client controls. For this reason, you should generally write the actions first and then create the menu items and buttons you want to connect with them. Note also that when an action has no `OnExecute` handler, the client control is automatically disabled (or grayed), unless the `DisableIfNoHandler` property is set to `False`.

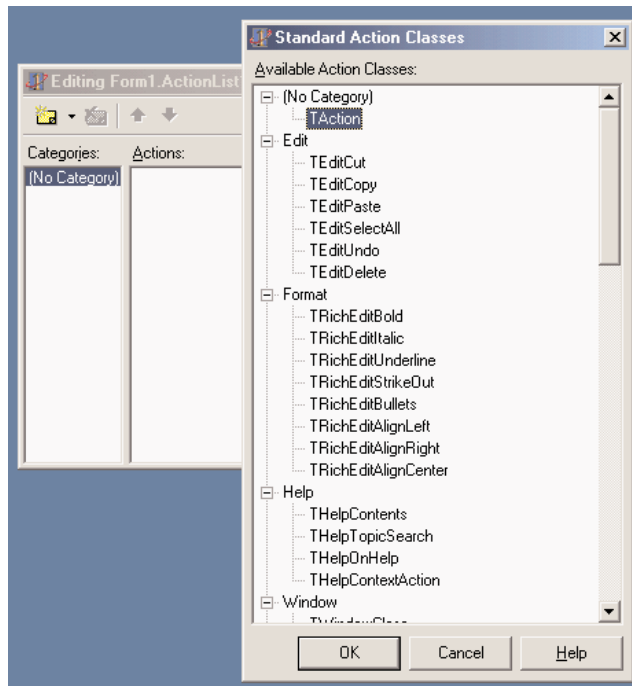
The client controls connected to actions are usually menu items and various types of buttons (push buttons, check boxes, radio buttons, speed buttons, toolbar buttons, and the like), but nothing prevents you from creating new components that hook into this architecture. Component writers can even define new actions, as we'll do in Chapter 11, and new link action objects.

Besides a client control, some actions can also have a target component. Some predefined actions hook to a specific target component (for examples, see the coverage of the `DataSet` components in the Chapter 13 section “Looking for Records in a Table”). Other actions automatically look for a target component in the form that supports the given action, starting with the active control.

Finally, the action objects are held by an `ActionList` component, the only class of the basic architecture that shows up on the Component Palette. The action list receives the execute actions that aren't handled by the specific action objects, firing the `OnExecuteAction`. If even the action list doesn't handle the action, Delphi calls the `OnExecuteAction` event of the `Application` object. The `ActionList` component has a special editor you can use to create several actions, as you can see in Figure 8.1.

**FIGURE 8.1:**

The ActionList component editor, with a list of pre-defined actions you can use



In the editor, actions are displayed in groups, as indicated by their Category property. By simply setting this property to a brand-new value, you instruct the editor to introduce a new category. These categories are basically logical groups, although in some cases a group of actions can work only on a specific type of target component. You might want to define a category for every pull-down menu or group them in some other logical way.

## Predefined Actions in Delphi 6

With the action list editor, you can create a brand new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box, as shown in Figure 8.1. There are many predefined actions, which can be divided into logical groups:

**File actions** include open, save as, open with, run, print setup, and exit.

**Edit actions** are illustrated in the next example. They include cut, copy, paste, select all, undo, and delete.

**RichEdit actions** complement the edit actions for RichEdit controls and include bold, italic, underline, strikeout, bullets, and various alignment actions.

**MDI window actions** will be demonstrated in Chapter 10, as we examine the Multiple Document Interface approach. They include all the most common MDI operations: arrange, cascade, close, tile (horizontally or vertically), and minimize all.

**Dataset actions** relate to database tables and queries and will be discussed in Chapter 13. There are many dataset actions, representing all the main operations you can perform on a dataset.

**Help actions** allow you to activate the contents page or index of the Help file attached to the application.

**Search actions** include find, find first, find next, and replace.

**Tab and Page control actions** include previous page and next page navigation.

**Dialog actions** activate color, font, open, save, and print dialogs.

**List actions** include clear, copy, move, delete, and select all. These actions let you interact with a list control. Another group of actions, including static list, virtual list, and some support classes, allow the definition of lists that can be connected to a user interface. More on this topic is in the section “Using List Actions” toward the end of this chapter.

**Web actions** include browse URL, download URL, and send mail actions.

**Tools actions** include only the dialog to customize the action bars.

## NOTE

You can also define new custom actions and register them in Delphi’s IDE, as we’ll see in Chapter 11.

Besides handling the `OnExecute` event of the action and changing the status of the action to affect the user interface of the client controls, an action can also handle the `OnUpdate` event, which is activated when the application is idle. This gives you the opportunity to check the status of the application or the system and change the user interface of the controls accordingly. For example, the standard `PasteEdit` action enables the client controls only when there is some text in the Clipboard.

## Actions in Practice

Now that you understand the main ideas behind this very important Delphi feature, let’s try out an example from the companion CD. The program is called `Actions` and demonstrates a number of features of the action architecture. I began building it by placing a new `ActionList` component in its form and adding the three standard edit actions and a few custom ones. The form also has a panel with some speed buttons, a main menu, and a `Memo` control (the automatic target of the edit actions). Listing 8.1 is the list of the actions, extracted from the DFM file.

**Listing 8.1: The actions of the Actions example**

```
object ActionList1: TActionList
  Images = ImageList1
  object ActionCopy: TEditCopy
    Category = 'Edit'
    Caption = '&Copy'
    ShortCut = <Ctrl+C>
  end
  object ActionCut: TEditCut
    Category = 'Edit'
    Caption = 'Cu&t'
    ShortCut = <Ctrl+X>
  end
  object ActionPaste: TEditPaste
    Category = 'Edit'
    Caption = '&Paste'
    ShortCut = <Ctrl+V>
  end
  object ActionNew: TAction
    Category = 'File'
    Caption = '&New'
    ShortCut = <Ctrl+N>
    OnExecute = ActionNewExecute
  end
  object ActionExit: TAction
    Category = 'File'
    Caption = 'E&xit'
    ShortCut = <Alt+F4>
    OnExecute = ActionExitExecute
  end
  object NoAction: TAction
    Category = 'Test'
    Caption = '&No Action'
  end
  object ActionCount: TAction
    Category = 'Test'
    Caption = '&Count Chars'
    OnExecute = ActionCountExecute
    OnUpdate = ActionCountUpdate
  end
  object ActionBold: TAction
    Category = 'Edit'
    Caption = '&Bold'
    ShortCut = <Ctrl+B>
    OnExecute = ActionBoldExecute
  end
  object ActionEnable: TAction
    Category = 'Test'
    Caption = '&Enable NoAction'
```

```
    OnExecute = ActionEnableExecute
end
object ActionSender: TAction
    Category = 'Test'
    Caption = 'Test &Sender'
    OnExecute = ActionSenderExecute
end
end
```

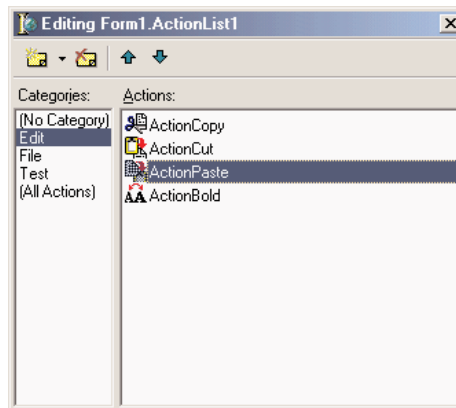
**NOTE**

The shortcut keys are stored in the DFM files using virtual key numbers, which also include values for the Ctrl and Alt keys. In this and other listings throughout the book, I've replaced the numbers with the literal values, enclosing them in angle brackets.

All of these actions are connected to the items of a *MainMenu* component and some of them also to the buttons of a *Toolbar* control. Notice that the images selected in the *ActionList* control affect the actions in the editor only, as you can see in Figure 8.2. For the images of the *ImageList* to show up also in the menu items and in the toolbar buttons, you must also select the image list in the *MainMenu* and in the *Toolbar* components.

**FIGURE 8.2:**

The ActionList editor of the Actions example



The three predefined actions for the *Edit* menu don't have associated handlers, but these special objects have internal code to perform the related action on the active edit or memo control. These actions also enable and disable themselves, depending on the content of the Clipboard and on the existence of selected text in the active edit control. Most other actions have custom code, except for the *NoAction* object. Having no code, the menu item and the button connected with this command are disabled, even if the *Enabled* property of the action is set to *True*.

I've added to the example, and to the Test menu, another action that enables the menu item connected to the NoAction object:

```
procedure TForm1.ActionEnableExecute(Sender: TObject);
begin
    NoAction.DisableIfNoHandler := False;
    NoAction.Enabled := True;
    ActionEnable.Enabled := False;
end;
```

Simply setting Enabled to True will produce the effect for only a very short time, unless you set the DisableIfNoHandler property, as discussed in the previous section. Once this operation is done, I disable the current action, since there is no need to issue the same command again.

This is different from an action you can toggle, such as the Edit > Bold menu item and the corresponding speed button. Here is the code of the Bold action:

```
procedure TForm1.ActionBoldExecute(Sender: TObject);
begin
    with Memo1.Font do
        if fsBold in Style then
            Style := Style - [fsBold]
        else
            Style := Style + [fsBold];
    // toggle status
    ActionBold.Checked := not ActionBold.Checked;
end;
```

The ActionCount object has very simple code, but it demonstrates an OnUpdate handler; when the memo control is empty, it is automatically disabled. We could have obtained the same effect by handling the OnChange event of the memo control itself, but in general it might not always be possible or easy to determine the status of a control simply by handling one of its events. Here is the code of the two handlers of this action:

```
procedure TForm1.ActionCountExecute(Sender: TObject);
begin
    ShowMessage ('Characters: ' + IntToStr (Length (Memo1.Text)));
end;

procedure TForm1.ActionCountUpdate(Sender: TObject);
begin
    ActionCount.Enabled := Memo1.Text <> '';
end;
```

Finally, I've added a special action to test the sender object of the action event handler and get some other system information. Besides showing the object class and name, I've added

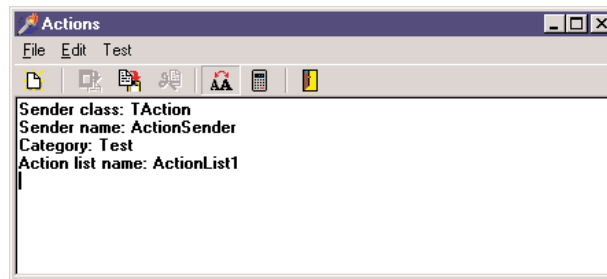
code that accesses the action list object. I've done this mainly to show that you can access this information and how to do it:

```
procedure TForm1.ActionSenderExecute(Sender: TObject);  
begin  
    Memo1.Lines.Add ('Sender class: ' + Sender.ClassName);  
    Memo1.Lines.Add ('Sender name: ' + (Sender as TComponent).Name);  
    Memo1.Lines.Add ('Category: ' + (Sender as TAction).Category);  
    Memo1.Lines.Add (  
        'Action list name: ' + (Sender as TAction).ActionList.Name);  
end;
```

You can see the output of this code in Figure 8.3, along with the user interface of the example. Notice that the Sender is not the menu item you've selected, even if the event handler is connected to it. The Sender object, which fires the event, is the action, which intercepts the user operation.

**FIGURE 8.3:**

The Actions example, with a detailed description of the Sender of an Action object's OnExecute event



Finally, keep in mind that you can also write handlers for the events of the ActionList object itself, which play the role of global handlers for all the actions of the list, and for the Application global object, which fires for all the actions of the application. Before calling the action's OnExecute event, in fact, Delphi activates the OnExecute event of the ActionList and the OnActionExecute event of the Application global object. These events can have a look at the action, eventually execute some shared code, and then stop the execution (using the Handled parameter) or let it reach the next level.

If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

## NOTE

When an action is executed, it searches for a control to play the role of the action target, by looking at the active control, the active form, and other controls on the form. For example, edit actions refer to the currently active control (if they inherit from TCustomEdit), while dataset controls look for the dataset connected with the data source of the data-aware control having the input focus. Other actions follow different approaches to find a target component, but the overall idea is shared by most standard actions.

## The Toolbar and the ActionList of an Editor

In the previous chapter, I built the RichBar example to demonstrate the development of an editor with a toolbar and a status bar. Of course, I should have also added a menu bar to the form, but this would have created quite a few troubles in synchronizing the status of the toolbar buttons with those of the menu items. A very good solution to this problem is to use actions, which is what I've done in the MdEdit example, discussed in this section and available on the CD.

The application is based on an ActionList component, which includes actions for file handling and Clipboard support, with code similar to the RichBar version. The Font type and color selection is still based on combo boxes, so this doesn't involve action—same for the drop-down menu of the Size button. The menu, however, has a few extra commands, including one for character counting and one for changing the background color. These are based on actions, and the same happens for the three new paragraph justification buttons (and menu commands).

One of the key differences in this new version is that the code never refers to the status of the toolbar buttons, but eventually modifies the status of the actions. In other cases I've used the actions OnUpdate events. For example, the RichEditSelectionChange method doesn't update the status of the bold button, which is connected to an action with the following OnUpdate handler:

```
procedure TFormRichNote.acBoldUpdate(Sender: TObject);  
begin  
    acBold.Checked := fsBold in RichEdit.SelAttributes.Style;  
end;
```

Similar OnUpdate event handlers are available for most actions, including the counting operations (available only if there is some text in the RichEdit control), the Save operation (available if the text has been modified), and the Cut and Copy operations (available only if some text is selected):

```
procedure TFormRichNote.acCountcharsUpdate(Sender: TObject);  
begin  
    acCountChars.Enabled := RichEdit.GetTextLen > 0;  
end;  
  
procedure TFormRichNote.acSaveUpdate(Sender: TObject);  
begin  
    acSave.Enabled := Modified;  
end;  
  
procedure TFormRichNote.acCutUpdate(Sender: TObject);  
begin  
    acCut.Enabled := RichEdit.SelLength > 0;  
    acCopy.Enabled := acCut.Enabled;  
end;
```



In the older example, the status of the Paste button was updated in the `OnIdle` event of the `Application` object. Now that we use actions we can convert it into yet another `OnUpdate` handler (see the preceding chapter for details on this code):

```
procedure TFormRichNote.acPasteUpdate(Sender: TObject);  
begin  
    acPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;  
end;
```

Finally, the program has an addition compared to the last version: the three paragraph-alignment buttons. These toolbar buttons and the related menu items should work like radio buttons, being mutually exclusive with one of the three options always selected. For this reason the actions have the `GroupIndex` set to 1, the corresponding menu items have the `RadioItem` property set to `True`, and the three toolbar buttons have their `Grouped` property set to `True` and the `AllowAllUp` property set to `False`. (They are also visually enclosed between two separators.)

This is required so that the program can set the `Checked` property for the action corresponding to the current style, which avoids unchecking the other two actions directly. This code is part of the `OnUpdate` event of the action list, as it applies to multiple actions:

```
procedure TFormRichNote.ActionListUpdate(Action: TBasicAction;  
    var Handled: Boolean);  
begin  
    // check the proper paragraph alignment  
    case RichEdit.Paragraph.Alignment of  
        taLeftJustify: acLeftAligned.Checked := True;  
        taRightJustify: acRightAligned.Checked := True;  
        taCenter: acCentered.Checked := True;  
    end;  
    // checks the caps lock status  
    CheckCapslock;  
end;
```

Finally, when one of these buttons is selected, the shared event handler uses the value of the `Tag`, set to the corresponding value of the `TAlignment` enumeration, to determine the proper alignment:

```
procedure TFormRichNote.ChangeAlignment(Sender: TObject);  
begin  
    RichEdit.Paragraph.Alignment := TAlignment ((Sender as TAction).Tag);  
end;
```

## Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement this differently. Delphi has two ready-to-use toolbar containers, the `CoolBar` and the `ControlBar` components. They have differences in their user interface, but the biggest one is that the `CoolBar` is a Win32 common control, part of the operating system, while the `ControlBar` is a VCL-based component.

Both components can host toolbar controls as well as some extra elements such as combo boxes and other controls. Actually, a toolbar can also replace the menu of an application, as we'll see later on.

We'll investigate the two components in the next two sections, but I want to emphasize here (without getting too far ahead of myself) that I generally favor the use of the `ControlBar`. It is based on VCL (and not subject to upgrade along with each minor release of Microsoft Internet Explorer), and its user interface is nicer and more similar to that of common office applications.

### A Really Cool Toolbar

The `CoolBar` component is basically a collection of `TCoolBand` objects that you can activate by selecting the Band Editor item of the `CoolBar` shortcut menu, the `Bands` property, or the Object TreeView. You can customize the `CoolBar` component in many ways: You can set a bitmap for its background, add some bands to the `Bands` collection, and then assign to each band an existing component or component container. You can use any window-based control (not graphic controls), but only some of them will show up properly. If you want to have a bitmap on the background of the `CoolBar`, for example, you need to use partially transparent controls.

The typical component used in a `CoolBar` is the `Toolbar` (which can be made completely transparent), but combo boxes, edit boxes, and animation controls are also quite common. This is often inspired by the user interface of Internet Explorer, the first Microsoft application featuring the `CoolBar` component.

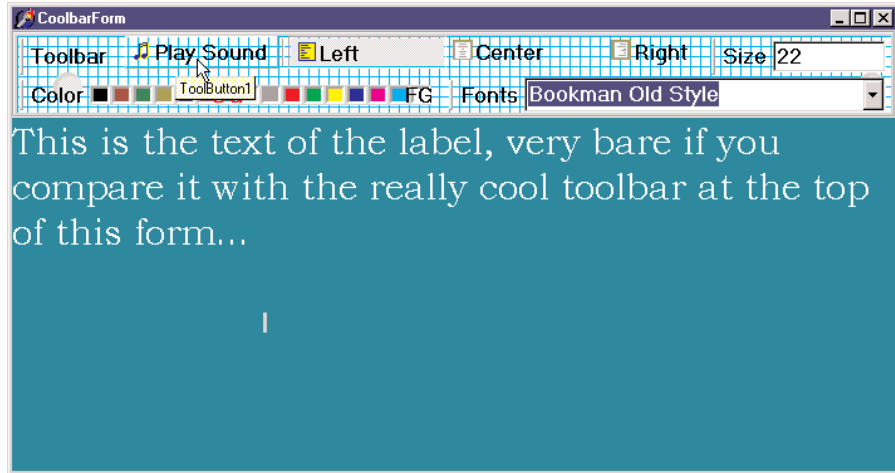
You can place one band on each line or all of them on the same line. Each would use a part of the available surface, and it would be automatically enlarged when the user clicks on its title. It is easier to use this new component than to explain it. Try it yourself or follow the description below, in which we build a new version of our continuing toolbar example based on a `CoolBar` control. You can see the form displayed by this application at run time in Figure 8.4.

The `CoolBar` example has a `TCoolBar` component with four bands, two for each of the two lines. The first band includes a subset of the toolbar of the previous example, this time adding an `ImageList` for the highlighted images. The second has an edit box used to set the

font of the text; the third has a ColorGrid component, used to choose the font color and that of the background. The last band has a ComboBox control with the available fonts.

**FIGURE 8.4:**

The form of the CoolBar example at run time



The user interface of the CoolBar component is really very nice, and Microsoft is increasingly using it in its applications. However, the Windows CoolBar control has had many different and incompatible versions, as Microsoft has released different versions of the common control library with different versions of the Internet Explorer. Some of these versions “broke” existing programs built with Delphi.

#### NOTE

It is interesting to note that Microsoft applications generally don't use the common control libraries. Word and Excel use their own internal versions of the common controls, and VB uses an OCX, not the common controls directly. Part of the reason that Borland had so much trouble with the common controls is that it uses them more (and in more ways) than even Microsoft does.

For this reason, Borland introduced (in Delphi 4) a toolbar container called the ControlBar. A control bar hosts several controls, as a CoolBar does, and offers a similar user interface that lets a user drag items and reorganize the toolbar at run time. A good example of the use of the ControlBar control is Delphi's own toolbar, but Microsoft applications use a very similar user interface.

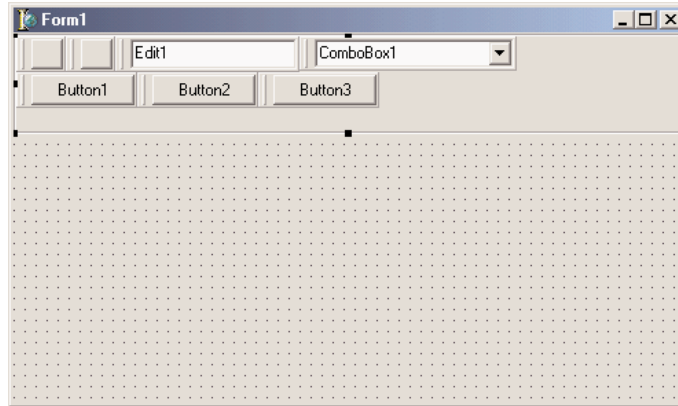
## The ControlBar

The ControlBar is a control container, and you build it just by placing other controls inside it, as you do with a panel (there is no list of Bands in it). Every control placed in the bar gets its own dragging area (a small panel with two vertical lines, on the left of the control), as you can

see in Figure 8.5. For this reason, you should generally avoid placing specific buttons inside the ControlBar, but rather add containers with buttons inside them. Rather than using a panel, you should generally use one ToolBar control for every section of the ControlBar.

**FIGURE 8.5:**

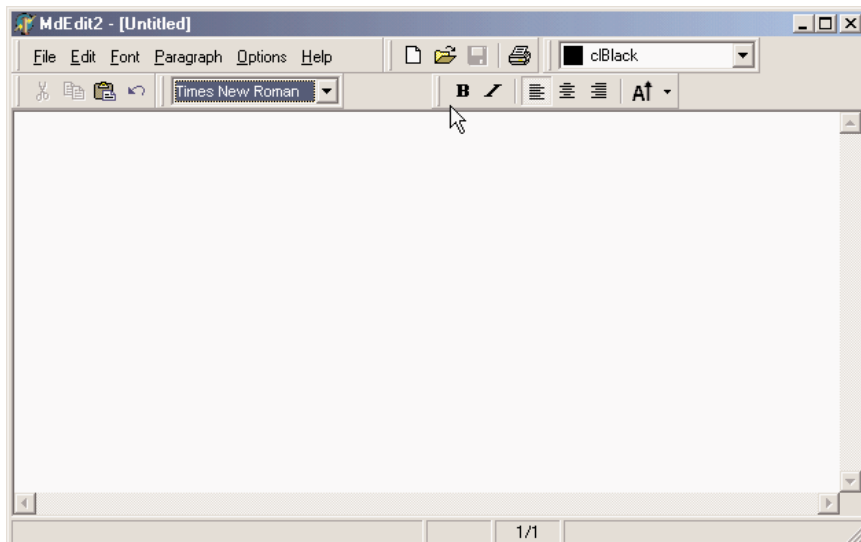
The ControlBar is a container that allows a user to drag all the elements, using the special drag bar on the side. Notice that each button gets a separate drag bar, something you'll generally try to avoid.



The MdEdit2 example is another version of the demo we've developed throughout the last and this chapter. I've basically grouped the buttons into three toolbars (instead of a single one) and left the two combo boxes as stand-alone controls. All these components are inside a ControlBar, so that a user can arrange them at runtime, as you can see in Figure 8.6.

**FIGURE 8.6:**

The MdEdit2 example at run time, while a user is rearranging the toolbars in the control bar



The following snippet of the DFM listing of the MdEdit2 example shows how the various toolbars and controls are embedded in the ControlBar component:

```
object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  PopupMenu = BarMenu
object ToolBarFile: TToolBar
  Flat = True
  Images = Images
  Wrapable = False
object ToolButton1: TToolButton
  Action = acNew
end
  // more buttons...
end
object ToolBarEdit: TToolBar...
object ToolBarFont: TToolBar...
object ToolBarMenu: TToolBar
  AutoSize = True
  Flat = True
  Menu = MainMenu
end
object ComboFont: TComboBox
  Hint = 'Font Family'
  Style = csDropDownList
  OnClick = ComboFontClick
end
object ColorBox1: TColorBox...
end
```

To obtain the standard effect, you have to disable the edges of the toolbar controls and set their style to flat. Sizing all the controls alike, so that you obtain one or two rows of elements of the same height, is not as easy as it might seem at first. Some controls have automatic sizing or various constraints. In particular, to make the combo box the same height as the toolbars, you have to tweak the type and size of its font. Resizing the control itself has no effect.

The ControlBar also has a shortcut menu that allows you to show or hide each of the controls currently inside it. Instead of writing code specific to this example, I've implemented a more generic (and reusable) solution. The shortcut menu, called BarMenu, is empty at design time and is populated when the program starts:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
  mItem: TMenuItem;
```

```

begin
  ...
  // populate the control bar menu
  for I := 0 to ControlBar.ControlCount - 1 do
  begin
    mItem := TMenuItem.Create (Self);
    mItem.Caption := ControlBar.Controls [I].Name;
    mItem.Tag := Integer (ControlBar.Controls [I]);
    mItem.OnClick := BarMenuClick;
    BarMenu.Items.Add (mItem);
  end;

```

The `BarMenuClick` procedure is a single event handler that is used by all of the items of the menu and uses the `Tag` property of the `Sender` menu item to refer to the element of the `ControlBar` associated with the item in the `FormCreate` method:

```

procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
  aCtrl: TControl;
begin
  aCtrl := TControl ((Sender as TComponent).Tag);
  aCtrl.Visible := not aCtrl.Visible;
end;

```

Finally, the `OnPopup` event of the menu is used to refresh the check mark of the menu items:

```

procedure TFormRichNote.BarMenuPopup(Sender: TObject);
var
  I: Integer;
begin
  // update the menu checkmarks
  for I := 0 to BarMenu.Items.Count - 1 do
    BarMenu.Items [I].Checked := TControl (BarMenu.Items [I].Tag).Visible;
end;

```

## A Menu in a Control Bar

If you look at the user interface of the `MdEdit2` application, in Figure 8.6, you'll notice that the menu of the form actually shows up inside a toolbar, hosted by the control bar, and below the application caption. In prior versions of Delphi, this required writing some custom code. In Delphi 6, instead, all you have to do is to set the `Menu` property of the toolbar. You must also remove the main menu from the `Menu` property of the form, to avoid having two menus.

## Delphi's Docking Support

Another feature added in Delphi 4 was support for *dockable* toolbars and controls. In other words, you can create a toolbar and move it to any of the sides of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

First of all, Delphi's docking support is connected with container controls, not with forms. A panel, a ControlBar, and other containers (technically, any control derived from `TWinControl`) can be set up as dock targets by enabling their `DockSite` property. You can also set the `AutoSize` property of these containers, so that they'll show up only if they actually hold a control.

To be able to drag a control (an object of any `TControl`-derived class) into the dock site, simply set its `DragKind` property to `dkDock` and its `DragMode` property to `dmAutomatic`. This way, the control can be dragged away from its current position into a new docking container. To undock a component and move it to a special form, you can set its `FloatingDockSiteClass` property to `TCustomDockForm` (to use a predefined stand-alone form with a small caption).

All the docking and undocking operations can be tracked by using special events of the component being dragged (`OnStartDock` and `OnEndDock`) and the component that will receive the docked control (`OnDragOver` and `OnDragDrop`). These docking events are very similar to the dragging events available in earlier versions of Delphi.

There are also commands you can use to accomplish docking operations in code and to explore the status of a docking container. Every control can be moved to a different location using the `Dock`, `ManualDock`, and `ManualFloat` methods. A container has a `DockClientCount` property, indicating the number of docked controls, and a `DockClients` property, with the array of these controls.

Moreover, if the dock container has the `UseDockManager` property set to `True`, you'll be able to use the `DockManager` property, which implements the `IDockManager` interface. This interface has many features you can use to customize the behavior of a dock container, even including support for streaming its status.

As you can see from this brief description, docking support in Delphi is based on a large number of properties, events, methods and objects (such as dock zones and dock trees)—more features than we have room to explore in detail. The next example introduces the main features you'll generally need.

---

**NOTE**

Docking support is not currently available in VisualCLX on either platform.

## Docking Toolbars in ControlBars

In the MdEdit2 example, already discussed, I've included docking support. The program has a second ControlBar at the bottom of the form, which accepts dragging one of the toolbars in the ControlBar at the top. Since both toolbar containers have the AutoSize property set to True, they are automatically removed when the host contains no controls. I've also set to True the AutoDrag and AutoDock properties of both ControlBars.

Actually, I had to place the bottom ControlBar inside a panel, together with the RichEdit control. Without this trick, the ControlBar, when activated and automatically resized, kept moving below the status bar, which I don't think is the correct behavior. Because, in the example, the ControlBar is the only control of the panel aligned to the bottom, there is no possible confusion.

To let users drag the toolbars out of the original container, all you have to do is, once again (as stated previously), set their DragKind property to dkDock and their DragMode property to dmAutomatic. The only two exceptions are the menu toolbar, which I decided to keep close to the typical position of a menu bar, and the ColorBox control, as unlike the combo box this component doesn't expose the DragMode and DragKind properties. (Actually, in the FormCreate method of the example, you'll find code you can use to activate docking for the component, based on the "protected hack" discussed in Chapter 3.) The Fonts combo box can be dragged, but I don't want to let a user dock it in the lower control bar. To implement this constraint, I've used the control bar's OnDockOver event handler, by accepting the docking operation only for toolbars:

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;  
    Source: TDragDockObject; X, Y: Integer; State: TDragState;  
    var Accept: Boolean);  
begin  
    Accept := Source.Control is TToolbar;  
end;
```

When you move one of the toolbars outside of any container, Delphi automatically creates a floating form; you might be tempted to set it back by closing the floating form. This doesn't work, as the floating form is removed along with the toolbar it contains. However, you can use the shortcut menu of the topmost ControlBar, attached also to the other ControlBar, to show this hidden toolbar.

The floating form created by Delphi to host undocked controls has a thin caption, the so-called *toolbar caption*, which by default has no text. For this reason, I've added some code to the OnEndDock event of each dockable control, to set the caption of the newly created form into which the control is docked. To avoid a custom data structure for this information, I've



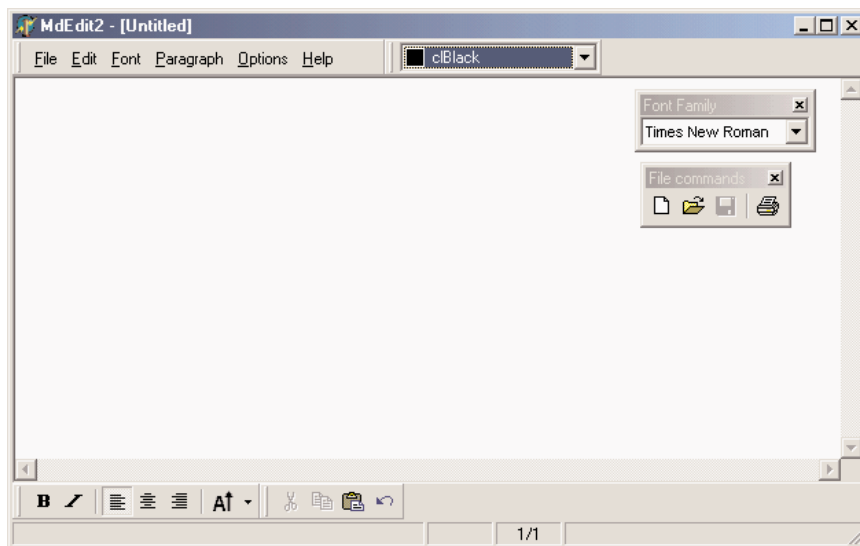
used the text of the Hint property of these controls, which is basically not used, to provide a suitable caption:

```
procedure TFormRichNote.EndDock(Sender, Target: TObject; X, Y: Integer);  
begin  
    if Target is TCustomForm then  
        TCustomForm(Target).Caption := GetShortHint((Sender as TControl).Hint);  
end;
```

You can see an example of this effect in the MdEdit2 program in Figure 8.7. Another extension of the example, one which I haven't done, could be the addition of dock areas on the two sides of the form. The only extra effort this requires would be a routine to turn the toolbars vertically, instead of horizontally. This basically implies switching the Left and Top properties of each button, after disabling the automatic sizing.

**FIGURE 8.7:**

The MdEdit2 example allows you to dock the toolbars (but not the menu) at the top or bottom of the form or to leave them floating.



## Controlling Docking Operations

Delphi provides many events and methods that give you a lot of control over docking operations, including a dock manager. To explore some of these features, try out the DockTest example, a test bed for docking operations. The program assigns the FloatingDockSiteClass property of a Memo component to TForm2, so that you can design specific features and add them to the floating frame that will host the control when it is floating, instead of using an instance of the default TCustomDockForm class.

Another feature of the program is that it handles the `OnDockOver` and `OnDockDrop` events of a dock host panel to display messages to the user, such as the number of controls currently docked:

```
procedure TForm1.Panel1DockDrop(Sender: TObject; Source: TDragDockObject;
  X, Y: Integer);
begin
  Caption := 'Docked: ' + IntToStr (Panel1.DockClientCount);
end;
```

In the same way, the program also handles the main form's docking events. Another control, a list box, has a shortcut menu you can invoke to perform docking and undocking operations in code, without the usual mouse dragging:

```
procedure TForm1.DocktoPanel1Click(Sender: TObject);
begin
  // dock to the panel
  ListBox1.ManualDock (Panel1, Panel1, alBottom);
end;
```

```
procedure TForm1.DocktoForm1Click(Sender: TObject);
begin
  // dock to the current form
  ListBox1.Dock (Self, Rect (200, 100, 100, 100));
end;
```

```
procedure TForm1.Floating1Click(Sender: TObject);
begin
  // toggle the floating status
  if ListBox1.Floating then
    ListBox1.ManualDock (Panel1, Panel1, alBottom)
  else
    ListBox1.ManualFloat (Rect (100, 100, 200, 300));
    Floating1.Checked := ListBox1.Floating;
end;
```

The final feature of the example is probably the most interesting one: Every time the program closes, it saves the current docking status of the panel, using the dock manager support. When the program is reopened, it reapplies the docking information, restoring the previous configuration of the windows. The program does this only with the panel, so the other floating windows will be displayed in their original positions. Here is the code for saving and loading:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  FileStr: TFileStream;
begin
```

```
if Panel1.DockClientCount > 0 then
begin
  FileStr := TFileStream.Create (DockFileName, fmCreate or fmOpenWrite);
  try
    Panel1.DockManager.SaveToStream (FileStr);
  finally
    FileStr.Free;
  end;
end
else
  // remove the file
  DeleteFile (DockFileName);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  FileStr: TFileStream;
begin
  // reload the settings
  DockFileName := ExtractFilePath (Application.Exename) + 'dock.dck';
  if FileExists (DockFileName) then
  begin
    FileStr := TFileStream.Create (DockFileName, fmOpenRead);
    try
      Panel1.DockManager.LoadFromStream (FileStr);
    finally
      FileStr.Free;
    end;
  end;
  Panel1.DockManager.ResetBounds (True);
end;
```

There are more features one might theoretically add to a docking program, but to add those you should remove other features, as some of them might conflict. For example, automatic alignments don't work terribly well with the docking manager's code for restoring. I suggest you take this program and explore its behavior, extending it to support the type of user interface you prefer.

**NOTE**

Remember that although docking panels make an application look nice, some users get confused by the fact that their toolbars might disappear or be in a different position than they are used to. Don't overuse the docking features, or some of your inexperienced users may get lost.

## Docking to a PageControl

Another interesting feature of page controls is the specific support for docking. As you dock a new control over a PageControl, a new page is automatically added to host it, as you can easily see in the Delphi environment. To accomplish this, you simply set the PageControl as a dock host and activate docking for the client controls. This works best when you have secondary forms you want to host. Moreover, if you want to be able to move the entire PageControl into a floating window and then dock it back, you'll need a docking panel in the main form.

This is exactly what I've done in the DockPage example, which has a main form with the following settings:

```
object Form1: TForm1
  Caption = 'Docking Pages'
  object Panel1: TPanel
    Align = alLeft
    DockSite = True
    OnMouseDown = Panel1MouseDown
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    DockSite = True
    DragKind = dkDock
  object TabSheet1: TTabSheet
    Caption = 'List'
    object ListBox1: TListBox
      Align = alClient
    end
  end
end
end
object Splitter1: TSplitter
  Cursor = crHSplit
end
object Memo1: TMemo
  Align = alClient
end
end
```

Notice that the Panel has the UseDockManager property set to True and that the PageControl invariably hosts a page with a list box, as when you remove all of the pages, the code used for automatic sizing of dock containers might cause you some trouble. Now the program has two other forms, with similar settings (although they host different controls):

```
object Form2: TForm2
  Caption = 'Small Editor'
```

```
DragKind = dkDock
DragMode = dmAutomatic
object Memo1: TMemo
    Align = alClient
end
end
```

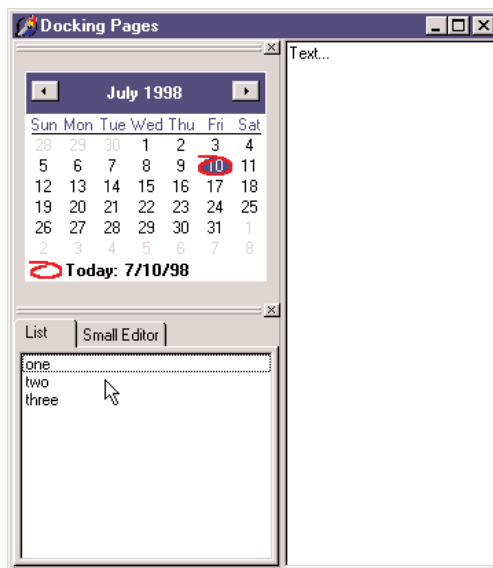
You can drag these forms onto the page control to add new pages to it, with captions corresponding with the form titles. You can also undock each of these controls and even the entire PageControl. To do this, the program doesn't enable automatic dragging, which would make it impossible to switch pages anymore. Instead, the feature is activated when the user clicks on the area of the PageControl that has no tabs—that is, on the underlying panel:

```
procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    PageControl1.BeginDrag (False, 10);
end;
```

You can test this behavior by running the DockPage example, although Figure 8.8 tries to depict it. Notice that when you remove the PageControl from the main form, you can directly dock the other forms to the panel and then split the area with other controls. This is the situation captured by the figure.

**FIGURE 8.8:**

The main form of the Dock-Page example after a form has been docked to the page control on the left. Notice that another form uses part of the area of a hosting panel.



## The ActionManager Architecture

We have seen that actions and the ActionManager component can play a central role in the development of Delphi applications, since they allow a much better separation of the user interface from the actual code of the application. The user interface, in fact, can now easily change without impacting the code too much. The drawback of this approach is that a programmer has more work to do. To have a new menu item, you need to add the corresponding action first, then move to the menu, add the menu item, and connect it to the action.

To solve this issue, and to provide developers and end users with some advanced features, Delphi 6 introduces a brand new architecture, based on the ActionManager component, which largely extends the role of actions. The ActionManager, in fact, has a collection of actions but also a collection of toolbars and menus tied to them. The development of these toolbars and menus is completely visual: you drag actions from a special component editor of the ActionManager to the toolbars to have the buttons you need. Moreover, you can let the end user of your programs do the same operation, and rearrange their own toolbars and menus starting with the actions you provide them.

In other words, using this architecture allows you to build applications with a modern user interface, customizable by the user. The menu can show only the recently used items (as many Microsoft programs do, nowadays), allows for animation, and more.

This architecture is centered on the ActionManager component, but includes also a few other components found at the end of the Additional page of the palette:

- The ActionManager component is a replacement of the ActionList (but can also use one or more existing ActionLists) adding to the architecture visual containers of actions.
- The ActionMainMenuBar control is a toolbar used to display the menu of an application based on the actions of an ActionManager component.
- The ActionToolBar control is a toolbar used to host buttons based on the actions of an ActionManager component.
- The CustomizeDlg component includes the dialog box you can use to let users customize the user interface of an application based on the ActionManager component.

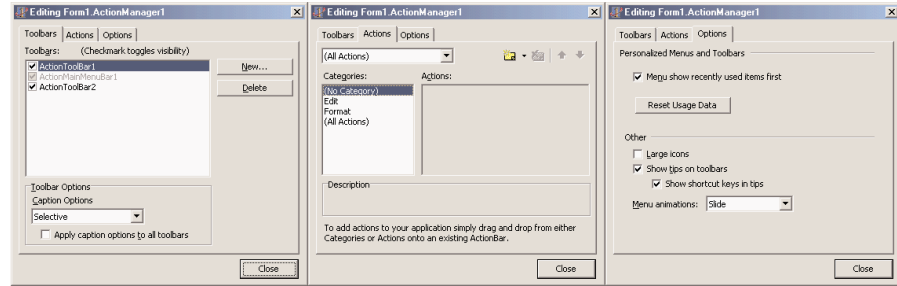
### Building a Simple Demo

As this architecture is mostly a visual architecture, a demo is probably worth more than a general discussion (although a printed book is not the best way to discuss a highly visual series of operations). To create a sample program based on this architecture, first drop an ActionManager component on a form, then double click it to open its component editor,

shown in Figure 8.9. Notice that this editor is not modal, so you can keep it open while doing other operations in Delphi. Consider also that this same dialog box is displayed by the `CustomizeDlg` component, although with some limited features (for example, adding new actions is disabled).

**FIGURE 8.9:**

The three pages of the ActionManager editor dialog box

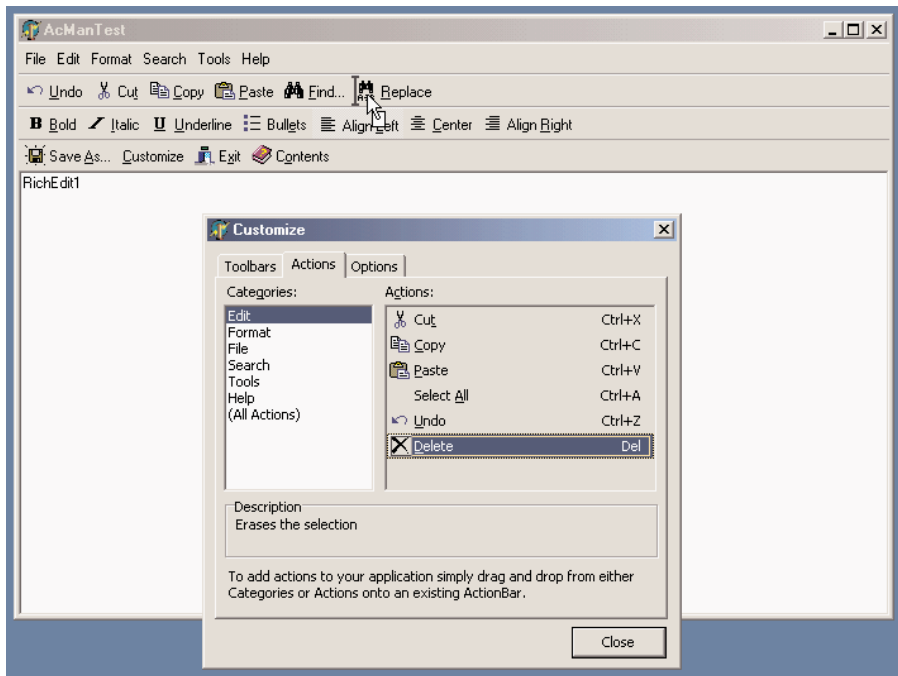


- The first page of this editor provides a list of visual containers of actions (toolbars or menus). You add new toolbars by clicking the `New` button. To add new menus, you have to add the corresponding component to the form, then open the `ActionBars` collection of the `ActionManager`, select an action bar or add a new one, and hook the menu to it using the `ActionBar` property. These are the same steps you could follow to connect a new toolbar to this architecture at run time.
- The second page of the `ActionManager` editor is very similar to the `ActionList` editor, providing a way to add new standard or custom action, arrange them in categories, and change their order. The new feature of this page, though, is that fact you can drag a category or a single action from it and drop it onto an action bar control. If you drag a category to a menu, you obtain a pull-down menu with all of the items of the category; if you drag it to a toolbar, each of the actions of the category gets a button on the toolbar. If you drag a single action to a toolbar, you get the corresponding button; if you drag it to the menu, you get a direct menu command, which is something you should generally avoid.
- The last page of the `ActionManager` editor allows you (and optionally an end user) to activate the display of recently used menu items and to modify some of the visual properties of the toolbars.

The `AcManTest` program is an example that uses some of the standard actions and a `RichEdit` control to showcase the use of this architecture (I haven't actually written any custom code to make the actions work better, as I wanted to focus only on the action manager for this example). You can experiment with it at design time or run it, click the `Customize` button, and see what an end user can do to customize the application (see Figure 8.10).

**FIGURE 8.10:**

Using the `CustomizeDlg` component, you can let a user customize the toolbars and the menu of an application, simply by dragging items from the dialog box or moving them around in the actions bars themselves.



Actually, in the program you can prevent the user from doing some operations on actions. Any specific element of the user interface (a `TActionClient` object) has a `ChangedAllowed` property that you can use to disable modify, move, and delete operations. Any action client container (the visual bars) has a property to disable hiding itself (`AllowHiding` by default is set to `True`). Each `ActionBar` `Items` collection has a `Customizable` option you can turn off to disable all user changes to the entire bar.

**TIP**

When I say “`ActionBar`” I don’t mean the visual toolbars containing action items, but the items of the `ActionBars` collection of the `ActionManager` component, which in turn has an `Items` collection. The best way to understand this structure is to look at the sub-tree displayed by the `ObjectTreeView` for an `ActionManager` component. Each `TActionBar` collection item has an actual `TCustomActionBar` visual component connected, but not the reverse (so, for example, you cannot reach this `Customizable` property if you start by selecting the visual toolbar). Due to the similarity of the two names, it can take a while to understand what the Delphi help actually means.

To make user settings persistent, I’ve connected a file (called `settings`) to the `FileName` property of the `ActionManager` component. When you assign this property, you should enter



a name of the file you want to use; when you start the program, the file will be created for you by the ActionManager.

The persistency is accomplished by streaming each `ActionClientItem` connected with the action manager. As these action client items are based on the user settings and maintain state information, a single file collects both user changes to the interface and usage data.

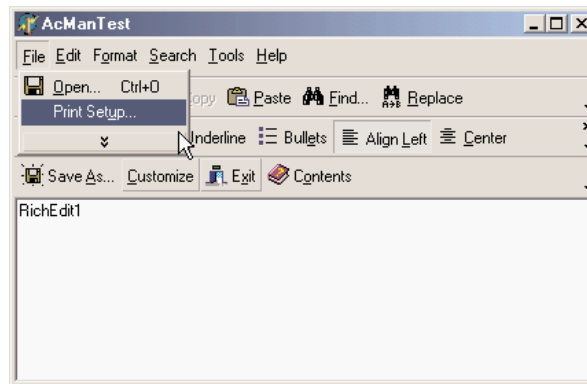
Since Delphi stores user setting and status information in a file you provide, you can make your application support multiple users on a single computer. Simply use a file of settings for each of them and connect it to the action manager as the program starts (using the current user of the computer or after some custom login). Another possibility is to store these settings over the network, so that even when a user moves to a different computer, the current personal settings will move along.

## Least-Recently Used Menu Items

Once a file for the user settings is available, the ActionManager will save into it the user preferences and also use it to track the user activity. This is essential to let the system remove menu items which haven't been used for some time, making them available in an extended menu, using the same user interface adopted by Microsoft (see Figure 8.11 for an actual example).

**FIGURE 8.11:**

The ActionManager disables least recently used menu items that you can still see by selecting the menu extension command.



The ActionManager doesn't simply show the least recently used items: it allows you to customize this behavior in a very precise way. Each action bar has a `SessionCount` property that keeps track of the number of times the application has been executed. Each `ActionClientItem` has a `LastSession` property and a `UsageCount` property used to track user operations. Notice, by the way, that a user can reset all this dynamic information by using the Reset Usage Data button of the customization dialog.

The system calculates the number of sessions the action has gone unused, by computing the difference between the number of times the application has been executed (`SessionCount`) and the last session in which the action has been used (`LastSession`). The value of `UsageCount` is used to look up in the `PrioritySchedule` how many sessions the items can go unused before it is removed. In other words, the `PrioritySchedule` maps each the usage count with a number of *unused* sessions. By modifying the `PrioritySchedule`, you can determine how fast the items are removed in case they are not used.

You can also prevent this system to be activated for specific actions or groups of actions. The `Items` property of the `ActionBars` of the `ActionManager` has a `HideUnused` property you can toggle to disable this feature for an entire menu. To make a specific item always visible, regardless of the actual usage, you can also set its `UsageCount` property to `-1`. However, the user settings might override this value.

To understand a little better how this system works, I've added a custom action (`ActionShowStatus`) to the `AcManTest` example. The action has the following code that saves the current action manager settings to a memory stream, converts it to text, and shows it inside the memo (refer to Chapter 5 for more information about streaming):

```

procedure TForm1.ActionShowStatusExecute(Sender: TObject);
var
    memStr, memStr2: TMemoryStream;
begin
    memStr := TMemoryStream.Create;
    try
        memStr2 := TMemoryStream.Create;
        try
            ActionManager1.SaveToStream(memStr);
            memStr.Position := 0;
            ObjectBinaryToText(memStr, memStr2);
            memStr2.Position := 0;
            RichEdit1.Lines.LoadFromStream(memStr2);
        finally
            memStr2.Free;
        end;
    finally
        memStr.Free;
    end;
end;

```

The output you obtain is the textual version of the `settings` file automatically updated at each execution of the program. Here a small portion of this file, with the details of one of pull-down menus and plenty of extra comments:

```

item // File pulldown of the main menu action bar
    Items = <

```

```
    item
      Action = Form1.FileOpen1
      LastSession = 19 // was used in the last session
      UsageCount = 4 // was used four times
    end
  item
    Action = Form1.FileSaveAs1 // never used
  end
  item
    Action = Form1.FilePrintSetup1
    LastSession = 7 // used some time ago
    UsageCount = 1 // only once
  end
  item
    Action = Form1.FileRun1 // never used
  end
  item
    Action = Form1.FileExit1 // never used
  end>
Caption = '&File'
LastSession = 19
UsageCount = 5 // the sum of the usage count of the items
end
```

## Porting an Existing Program

If this architecture is nice, you'll probably need to redo most of your applications to take advantage of it. However, if you're already using actions (with the `ActionList` component), this conversion will be much simpler. In fact, the `ActionManager` has its own set of actions but can also use actions from another `ActionManager` or `ActionList`. The `LinkedActionLists` property of the `ActionManager` is a collection of other containers of actions (`ActionLists` or `ActionManagers`), which can be associated with the current one. Associating all the various groups of action is useful to let a user customize the entire user interface with a single dialog box.

If you hook external actions and open the `ActionManager` editor, you'll see in the `Actions` page a combo box listing the current `ActionManager` plus the other action containers linked to it. You can choose one of these containers to see its set of actions and change their properties. The `All Action` option of this combo box allows you to work on all of the actions from the various containers at once, but I've noticed that at startup it is selected but not always *effective*. Reselect it to actually see all of the actions.

As an example of porting an existing application, I've extended the program built throughout this chapter, into the `MdEdit3` example. This example uses the same action list of the previous version hooked to an `ActionManager` that has the extra `customize` property, to let

users rearrange the user interface. Differently from the earlier AcManDemo program, the MdEdit3 example uses a ControlBar as a container for the action bars (a menu, three toolbars, and the usual combo boxes) and has full support for dragging them outside of the container as floating bars and dropping them into the lower ControlBar.

To accomplish this, I only had to modify the source code slightly to refer to the new classes for the containers (that is, TCustomActionToolBar instead of TToolBar) in the ControlBar-LowerDockOver method. I also found out that the OnEndDock event of the ActionToolBar component passes as parameter an empty target when the system creates a floating form to host the control, so that I couldn't easily give to this forms a new custom caption (see the EndDock method of the form).

## Using List Actions

We'll see more examples of the use of this architecture in the chapters devoted to MDI and database programming. For the moment, I just want to add an extra example showing how to use a rather complex group of standard actions introduced in Delphi 6, the list actions. List actions, in fact, comprise two different groups. Some of them (such as the Move, Copy, Delete, Clear, and Select All) actions are normal actions working on list boxes or other lists. The VirtualListAction and StaticListAction elements, instead, define actions based multiple choices, which are going to be displayed in a toolbar as a combo box.

The ListActions demo highlights both groups of list actions, as its ActionManager has five of them, displayed on two separate toolbars. This is a summary of the actions of the actions manager (I've omitted the action bars portion of the component's DFM file):

```
object ActionManager1: TActionManager
  ActionBars.SessionCount = 1
  ActionBars = <...>
  object StaticListAction1: TStaticListAction
    Caption = 'Numbers'
    Items.CaseSensitive = False
    Items.SortType = stNone
    Items = <
      item
        Caption = 'one'
      end
      item
        Caption = 'two'
      end
      ...>
    OnItemSelected = ListActionItemSelected
  end
  object VirtualListAction1: TVirtualListAction
    Caption = 'Items'
```

```

    OnGetItem = VirtualListAction1GetItem
    OnGetItemCount = VirtualListAction1GetItemCount
    OnItemSelected = ListActionItemSelected
end
object ListControlCopySelection1: TListControlCopySelection
    Caption = 'Copy'
    Destination = ListBox2
    ListControl = ListBox1
end
object ListControlDeleteSelection1: TListControlDeleteSelection
    Caption = 'Delete'
end
object ListControlMoveSelection2: TListControlMoveSelection
    Caption = 'Move'
    Destination = ListBox2
    ListControl = ListBox1
end
end

```

The program has also two list boxes in its form, used as action targets. The Copy and Move actions are tied to these two list boxes by their `ListControl` and `Destination` properties. The Delete action, instead, automatically works with the list box having the input focus.

The `StaticListAction` defines a series of alternative items, in its `Items` collection. This is not a plain string list, as any item has also an `ImageIndex`, which allows turning the combo box in graphical selection. You can, of course, add more items to this list programmatically. However, in case of a highly dynamic list, you can also use the `VirtualListAction`. This component doesn't define a list of items but has two events you can use to provide strings and images for the list. The `OnGetItemCount` event allows you to indicate the number of items to display; the `OnGetItem` event is then called for each specific item.

In the `ListActions` demo, the `VirtualListAction` has the following event handlers for its definition, producing the list you can see in the active combo box of Figure 8.12:

```

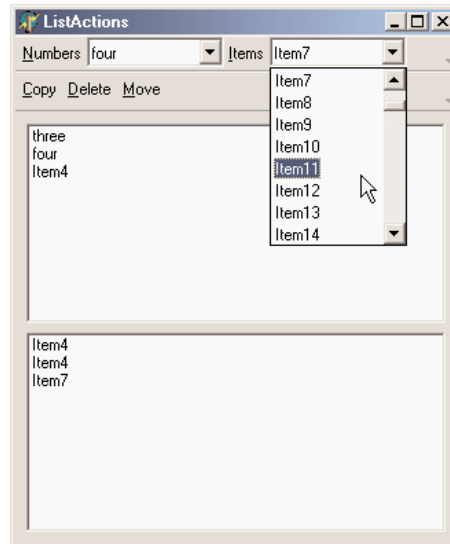
procedure TForm1.VirtualListAction1GetItemCount(Sender: TCustomListAction;
    var Count: Integer);
begin
    Count := 100;
end;

procedure TForm1.VirtualListAction1GetItem(Sender: TCustomListAction;
    const Index: Integer; var Value: String;
    var ImageIndex: Integer; var Data: Pointer);
begin
    Value := 'Item' + IntToStr (Index);
end;

```

**FIGURE 8.12:**

The ListActions application has a toolbar hosting a static list and a virtual one.

**NOTE**

I thought that the virtual action items were actually requested only when needed to display them, making this actually a virtual list. Instead, all the items are created right away, as you can prove by enabling the commented code of the `VirtualListItem.GetItem` method (not in the listing above), which adds to each item the time its string is requested.

Both the static and the virtual list have an `OnItemSelected` event. In the shared event handler, I've written the following code, to add the current item to the first list box of the form:

```
procedure TForm1.ListActionItemSelected(Sender: TCustomListAction;
    Control: TControl);
begin
    ListBox1.Items.Add ((Control as TCustomActionCombo).SelText);
end;
```

In this case, the sender is the custom action list, but the `ItemIndex` property of this list is not updated with the selected item. However, accessing the visual control that displays the list, we can obtain the value of the selected item.

## What's Next?

In this chapter, I've introduced the use of actions, the actions list, and action manager architectures. As you've seen, this is an extremely powerful architecture to separate the user interface from the actual code of your applications, which uses and refers to the actions and not the menu items or toolbar button related to them. The Delphi 6 extension of this architecture allows users of your programs to have a lot of control, and makes your applications resemble high-end programs without much effort on your part. The same architecture is also very handy to let you design the user interface of your program, regardless of whether you give this ability to users.

I've also covered other user-interface techniques, such as docking toolbars and other controls. You can consider this chapter the first step toward building professional applications. We will take other steps in the following chapters; but you already know enough to make your programs similar to some best-selling Windows applications, which may be very important for your clients.

Now that the elements of the main form of our programs are properly set up, we can consider adding secondary forms and dialog boxes. This is the topic of the next chapter, along with a general introduction to forms. The following chapter will then cover the overall structure of a Delphi application.

# Working with Forms

- Form styles, border styles, and border icons
- Mouse and keyboard input
- Painting and special effects
- Positioning, scaling, and scrolling forms
- Creating and closing forms
- Modal and modeless dialog boxes and forms
- Creating secondary forms dynamically
- Predefined dialog boxes
- Building a splash screen



If you've read the previous chapters, you should now be able to use Delphi's visual components to create the user interface of your applications. So let's turn our attention to another central element of development in Delphi: forms. We have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the `TForm` class are particularly interesting.

This chapter looks at some of the properties and styles of forms and at sizing and positioning them. I'll also introduce applications with multiple forms, the use of dialog boxes (custom and predefined ones), frames, and visual form inheritance. I'll also devote some time to input on a form, both from the keyboard and the mouse.

## The *TForm* Class

Forms in Delphi are defined by the `TForm` class, included in the Forms unit of VCL. Of course, there is now a second definition of forms inside VisualCLX. Although I'll mainly refer to the VCL class in this chapter, I'll also try to highlight differences with the cross-platform version provided in CLX.

The `TForm` class is part of the windowed-controls hierarchy, which starts with the `TWinControl` (or `TWidgetControl`) class. Actually, `TForm` inherits from the *almost complete* `TCustomForm`, which in turn inherits from `TScrollingWinControl` (or `TScrollingWidget`). Having all of the features of their many base classes, forms have a long series of methods, properties, and events. For this reason, I won't try to list them here, but I'd rather present some interesting techniques related to forms throughout this chapter. I'll start by presenting a technique for *not* defining the form of a program at design time, using the `TForm` class directly, and then explore a few interesting properties of the form class.

Throughout the chapter, I'll point out a few differences between VCL forms and CLX forms. I've actually built a CLX version for most of the examples of this chapter, so you can immediately start experimenting with forms and dialog boxes in CLX, as well as VCL. As in past chapters, the CLX version of each example is prefixed by the letter *Q*.

## Using Plain Forms

Generally, Delphi developers tend to create forms at design time, which implies deriving a new class from the base one, and build the content of the form visually. This is certainly a reasonable standard practice, but it is not compulsory to create a descendant of the `TForm` class to show a form, particularly if it is a simple one.

Consider this case: you have to show a rather long message (based on a string) to a user, and you don't want to use the simple predefined message box, as it will show up too large and

not provide scroll bars. You can create a form with a memo component in it, and display the string inside it. Nothing prevents you from creating this form in the standard visual way, but you might consider doing this in code, particularly if you need a large degree of flexibility.

The `DynaForm` and `QDynaForm` examples (both on the companion CD), which are somewhat extreme, have no form defined at design time but include a unit with this function:

```
procedure ShowStringForm (str: string);  
var  
    form: TForm;  
begin  
    Application.CreateForm (TForm, form);  
    form.caption := 'DynaForm';  
    form.Position := poScreenCenter;  
    with TMemo.Create (form) do  
        begin  
            Parent := form;  
            Align := alClient;  
            Scrollbars := ssVertical;  
            ReadOnly := True;  
            Color := form.Color;  
            BorderStyle := bsNone;  
            WordWrap := True;  
            Text := str;  
        end;  
    form.Show;  
end;
```

Besides the fact I had to create the form using the `Application` global object, a feature required by Delphi applications and discussed in the next chapter, this code simply does dynamically what you generally do with the form designer. Writing this code is undoubtedly more tedious, but it allows also a greater deal of flexibility, because any parameter can depend on external settings.

The `ShowStringForm` function above is not executed by an event of another form, as there are no traditional forms in this program. Instead, I've modified the project's source code to the following:

```
program DynaForm;  
  
uses  
    Forms,  
    DynaMemo in 'DynaMemo.pas';  
  
{ $R *.RES }  
  
var
```

```

str: string;

begin
  str := '';
  Randomize;
  while Length (str) < 2000 do
    str := str + Char (32 + Random (94));
  ShowStringForm (str);

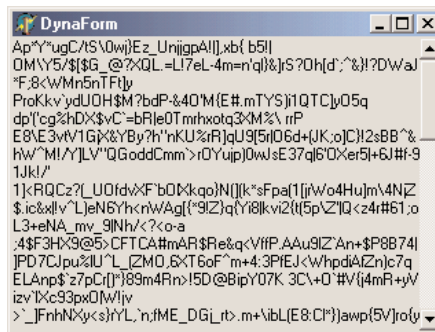
  Application.Run;
end.

```

The effect of running the DynaForm program is a strange-looking form filled with random characters (as you can see in Figure 9.1), not terribly useful in itself but for the idea it underscores.

**FIGURE 9.1:**

The dynamic form generated by the DynaForm example is completely created at run time, with no design-time support.



## TIP

An indirect advantage of this approach, compared to the use of DFM files for design-time forms, is that it would be much more difficult for an external programmer to grab information about the structure of the application. In Chapter 5 we saw that you can extract the DFM from the current Delphi executable file, but the same can be easily accomplished for any executable file compiled with Delphi for which you don't have the source code. If it is really important for you to keep to yourself a specific set of components you are using (maybe those in a specific form), and the default values of their properties, writing the extra code might be worth the effort.

## The Form Style

The `FormStyle` property allows you to choose between a normal form (`fsNormal`) and the windows that make up a Multiple Document Interface (MDI) application. In this case, you'll use the `fsMDIForm` style for the MDI parent window—that is, the frame window of the MDI application—and the `fsMDIChild` style for the MDI child window. To know more about the development of an MDI application, look at Chapter 10.

A fourth option is the `fsStayOnTop` style, which determines whether the form has to always remain on top of all other windows, except for any that also happen to be “stay-on-top” windows.

To create a top-most form (a form whose window is always on top), you need only set the `FormStyle` property, as indicated above. This property has two different effects, depending on the kind of form you apply it to:

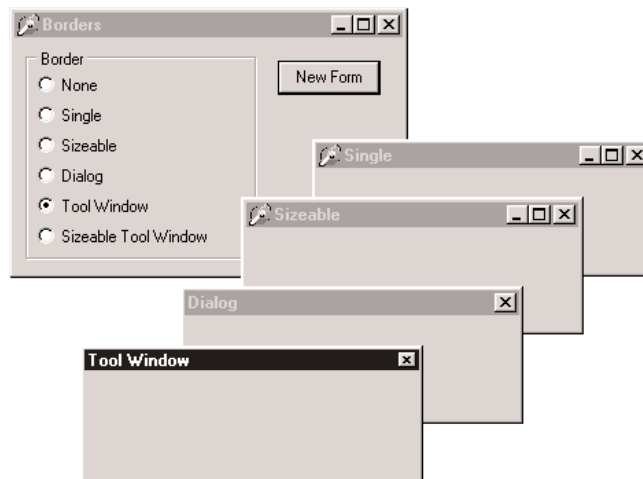
- The main form of an application will remain in front of every other application (unless other applications have the same top-most style, too). At times, this generates a rather ugly visual effect, so this makes sense only for special-purpose alert programs.
- A secondary form will remain in front of any other form of the application it belongs to. The windows of other applications are not affected, though. This is often used for floating toolbars and other forms that should stay in front of the main window.

## The Border Style

Another important property of a form is its `BorderStyle`. This property refers to a visual element of the form, but it has a much more profound influence on the *behavior* of the window, as you can see in Figure 9.2.

**FIGURE 9.2:**

Sample forms with the various border styles, created by the Borders example



At design time, the form is always shown using the default value of the `BorderStyle` property, `bsSizeable`. This corresponds to a Windows style known as *thick frame*. When a main window has a thick frame around it, a user can resize it by dragging its border. This is made clear by the special *resize* cursors (with the shape of a double-pointer arrow) displayed when the user moves the mouse onto this thick window border.

A second important choice for this property is `bsDialog`. If you select it, the form uses as its border the typical dialog-box frame—a thick frame that doesn't allow resizing. In addition to this graphical element, note that if you select the `bsDialog` value, the form becomes a dialog box. This involves several changes. For example, the items on its system menu are different, and the form will ignore some of the elements of the `BorderIcons` set property.

---

**WARNING**

Setting the `BorderStyle` property at design time produces no visible effect. In fact, several component properties do not take effect at design time, because they would prevent you from working on the component while developing the program. For example, how could you resize the form with the mouse if it were turned into a dialog box? When you run the application, though, the form will have the border you requested.

There are four more values we can assign to the `BorderStyle` property. The style `bsSingle` can be used to create a main window that's not resizable. Many games and applications based on windows with controls (such as data-entry forms) use this value, simply because resizing these forms makes no sense. Enlarging a form to see an empty area or reducing its size to make some components less visible often doesn't help a program's user (although Delphi's automatic scroll bars partially solve the last problem). The value `bsNone` is used only in very special situations and inside other forms. You'll never see an application with a main window that has no border or caption (except maybe as an example in a programming book to show you that it makes no sense).

The last two values, `bsToolWindow` and `bsSizeToolWin`, are related to the specific Win32 extended style `ws_ex_ToolWindow`. This style turns the window into a floating toolbox, with a small title font and close button. This style should not be used for the main window of an application.

To test the effect and behavior of the different values of the `BorderStyle` property, I've written a simple program called `Borders`, available also as `QBorders` in the CLX version. You've already seen its output, in Figure 9.2. However, I suggest you run this example and experiment with it for a while to understand all the differences in the forms.

---

**WARNING**

In CLX, the enumeration for the `BorderStyle` property uses slightly different values, prefixed by the letters *fbs* (form border style). So we have `fbsSingle`, `fbsDialog`, and so on.

The main form of this program contains only a radio group and a button. There is also a secondary form, with no components and the `Position` property set to `poDefaultPosOnly`. This affects the initial position of the secondary form we'll create by clicking the button. (I'll discuss the `Position` property later in this chapter.)

The code of the program is very simple. When you click the button, a new form is dynamically created, depending on the selected item of the radio group:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);  
var  
    NewForm: TForm2;  
begin  
    NewForm := TForm2.Create (Application);  
    NewForm.BorderStyle := TFormBorderStyle (BorderRadioGroup.ItemIndex);  
    NewForm.Caption := BorderRadioGroup.Items[BorderRadioGroup.ItemIndex];  
    NewForm.Show;  
end;
```

This code actually uses a trick: it casts the number of the selected item into the TFormBorderStyle enumeration. This works because I've given the radio buttons the same order as the values of this enumeration:

```
type  
    TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsTo1Window,  
                        bsSizeToolWin);
```

The BtnNewFormClick method then copies the text of the radio button to the caption of the secondary form. This program refers to TForm2, the secondary form defined in a secondary unit of the program, saved as SECOND.PAS. For this reason, to compile the example, you must add the following lines to the implementation section of the unit of the main form:

```
uses  
    Second;
```

### TIP

Whenever you need to refer to another unit of a program, place the corresponding **uses** statement in the **implementation** portion instead of the **interface** portion if possible. This speeds up the compilation process, results in cleaner code (because the units you include are separate from those included by Delphi), and prevents circular unit compilation errors. To accomplish this, you can also use the File > Use Unit menu command.

## The Border Icons

Another important element of a form is the presence of icons on its border. By default, a window has a small icon connected to the system menu, a Minimize button, a Maximize button, and a Close button on the far right. You can set different options using the BorderIcons property, a set with four possible values: biSystemMenu, biMinimize, biMaximize, and biHelp.

**NOTE**

The biHelp border icon enables the “What’s this?” Help. When this style is included and the biMinimize and biMaximize styles are excluded, a question mark appears in the form’s title bar. If you click this question mark and then click a component inside the form (but not the form itself!), Delphi activates the Help about that object inside a pop-up window. This is demonstrated by the BIIcons example, which has a simple Help file with a page connected to the HelpContext property of the button in the middle of the form.

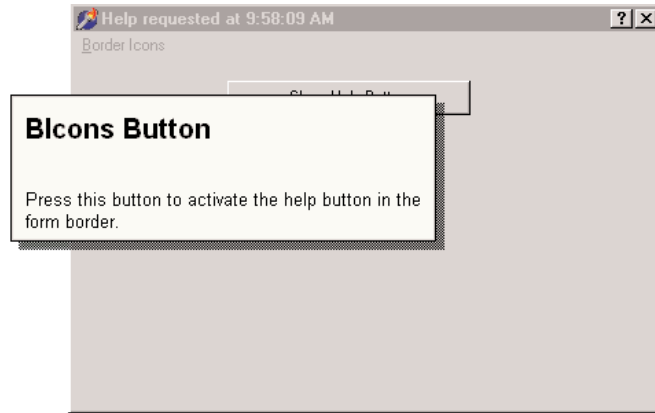
The BIIcons example demonstrates the behavior of a form with different border icons and shows how to change this property at run time. The form of this example is very simple: It has only a menu, with a pull-down containing four menu items, one for each of the possible elements of the set of border icons. I’ve written a single method, connected with the four commands, that reads the check marks on the menu items to determine the value of the BorderIcons property. This code is therefore also a good exercise in working with sets:

```
procedure TForm1.SetIcons(Sender: TObject);  
var  
    BorIco: TBorderIcons;  
begin  
    (Sender as TMenuItem).Checked := not (Sender as TMenuItem).Checked;  
    if SystemMenu1.Checked then  
        BorIco := [biSystemMenu]  
    else  
        BorIco := [];  
    if MaximizeBox1.Checked then  
        Include (BorIco, biMaximize);  
    if MinimizeBox1.Checked then  
        Include (BorIco, biMinimize);  
    if Help1.Checked then  
        Include (BorIco, biHelp);  
    BorderIcons := BorIco;  
end;
```

While running the BIIcons example, you can easily set and remove the various visual elements of the form’s border. You’ll immediately see that some of these elements are closely related: if you remove the system menu, all of the border icons will disappear; if you remove either the Minimize or Maximize button, it will be grayed; if you remove both these buttons, they will disappear. Notice also that in these last two cases, the corresponding items of the system menu are automatically disabled. This is the standard behavior for any Windows application. When the Maximize and Minimize buttons have been disabled, you can activate the Help button. As a shortcut to obtain this effect, you can click the button inside the form. Also, you can click the button after clicking the Help Menu icon to see a Help message, as you can see in Figure 9.3.

**FIGURE 9.3:**

The BIcons example. By selecting the help border icon and clicking over the button, you get the help displayed in the figure.



As an extra feature, the program also displays the time that the Help was invoked in the caption, by handling the `OnHelp` event of the form. This effect is visible in the figure.

**WARNING**

By looking at the `QBIcons` version, built with `CLX`, you can clearly notice that a bug in the library prevents you from changing the border icons at run time, while the different design-time settings fully work.

## Setting More Window Styles

The border style and border icons are indicated by two different Delphi properties, which can be used to set the initial value of the corresponding user interface elements. We have seen that besides changing the user interface, these properties affect the behavior of a window. It is important to know that in `VCL` (and obviously not in `CLX`), these border-related properties and the `FormStyle` property mainly correspond to different settings in the *style* and *extended style* of a window. These two terms reflect two parameters of the `CreateWindowEx` API function Delphi uses to create forms.

It is important to acknowledge this, because Delphi allows you to modify these two parameters freely by overriding the `CreateParams` virtual method:

```
public
  procedure CreateParams (var Params: TCreateParams); override;
```

This is the only way to use some of the peculiar window styles that are not directly available through form properties. For a list of window styles and extended styles, see the API Help under the topics “`CreateWindow`” and “`CreateWindowEx`.” You’ll notice that the `Win32` API has styles for these functions, including those related to tool windows.



To show how to use this approach, I've written the NoTitle example on the companion CD, which lets you create a program with a custom caption. First we have to remove the standard caption but keep the resizing frame by setting the corresponding styles:

```
procedure TForm1.CreateParams (var Params: TCreateParams);  
begin  
  inherited CreateParams (Params);  
  Params.Style := (Params.Style or ws_Popup) and not ws_Caption;  
end;
```

**NOTE**

Besides changing the style and other features of a window when it is created, you can change them at run time, although some of the settings do not take effect. To change most of the creation parameters at run time, you can use the `SetWindowLong` API function, which allows you to change the internal information of a window. The companion `GetWindowLong` function can be used to read the current status. Two more functions, `GetClassLong` and `SetClassLong`, can be used to read and modify class styles (the information of the `WindowClass` structure of `TCreateParams`). You'll seldom need to use these low-level Windows API functions in Delphi, unless you write advanced components.

To remove the caption, we need to change the overlapped style to a pop-up style; otherwise, the caption will simply stick. Now how do we add a custom caption? I've placed a label aligned to the upper border of the form and a small button on the far end. You can see this effect at run time in Figure 9.4.

**FIGURE 9.4:**

The NoTitle example has no real caption but a fake one made with a label.



To make the fake caption work, we have to tell the system that a mouse operation on this area corresponds to a mouse operation on the caption. This can be done by intercepting the `wm_NCHitTest` Windows message, which is frequently sent to Windows to determine where

the mouse currently is. When the hit is in the client area and on the label, we can pretend the mouse is on the caption by setting the proper result:

```
procedure TForm1.HitTest (var Msg: TWmNCHitTest);  
    // message wm_NCHitTest  
begin  
    inherited;  
    if (Msg.Result = htClient) and  
        (Msg.YPos < Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then  
        Msg.Result := htCaption;  
end;
```

The `GetSystemMetrics` API function used in the listing above is used to query the operating system about the size of the various visual elements. It is important to make this request every time (and not cache the result) because users can customize most of these elements by using the Appearance page of the Desktop options (in Control Panel) and other Windows settings. The small button, instead, has a call to the `Close` method in its `OnClick` event handler. The button is kept in its position even when the window is resized by using the `[akTop,akRight]` value for the `Anchors` property. The form also has size constraints, so that a user cannot make it too small, as described in the “Form Constraints” section later in this chapter.

## Direct Form Input

Having discussed some special capabilities of forms, I’ll now move to a very important topic: user input in a form. If you decide to make limited use of components, you might write complex programs as well, receiving input from the mouse and the keyboard. In this chapter, I’ll only introduce this topic.

### Supervising Keyboard Input

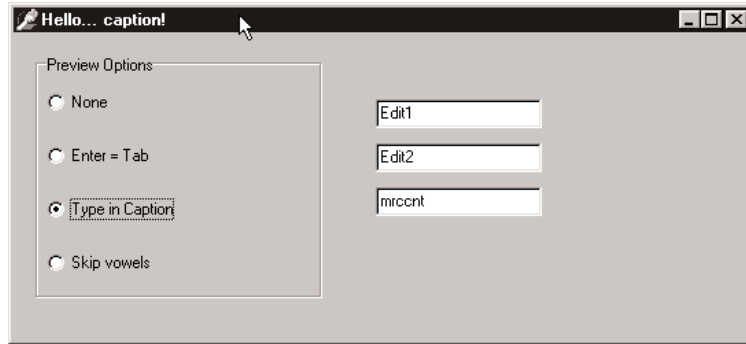
Generally, forms don’t handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. What you can do in these cases is turn on the `KeyPreview` property of the form. Then, even if you have some input controls, the form’s `OnKeyPress` event will always be activated for any keyboard-input operation. The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character 0, but the value 0 of the character set, indicated as #0).

The example I've built to demonstrate this, KPreview, has a form with no special properties (not even KeyPreview), a radio group with four options, and some edit boxes, as you can see in Figure 9.5.

**FIGURE 9.5:**

The KPreview program allows you to type into the caption of the form (among other things).



By default the program does nothing special, except when the various radio buttons are used to enable the key preview:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
    KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```

Now we'll start receiving the OnKeyPress events, and we can do one of the three actions requested by the three special buttons of the radio group. The action depends on the value of the ItemIndex property of the radio group component. This is the reason the event handler is based on a case statement:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
    case RadioPreview.ItemIndex of
        ...
```

In the first case, if the value of the Key parameter is #13, which corresponds to the Enter key, we disable the operation (setting Key to zero) and then mimic the activation of the Tab key. There are many ways to accomplish this, but the one I've chosen is quite particular. I send the CM\_DialogKey message to the form, passing the code for the Tab key (VK\_TAB):

```
1: // Enter = Tab
    if Key = #13 then
        begin
            Key := #0;
            Perform (CM_DialogKey, VK_TAB, 0);
        end;
```

**NOTE**

The `CM_DialogKey` message is an internal, undocumented Delphi message. There are a few of them, actually quite interesting to build advanced components for and for some special coding, but Borland never described those. For more information on this topic, refer to the sidebar “Component Messages and Notifications” in Chapter 11. Notice also that this exact message-based coding style is not available under CLX.

To type in the caption of the form, the program simply adds the character to the current `Caption`. There are two special cases. When the Backspace key is pressed, the last character of the string is removed (by copying to the `Caption` all the characters of the current `Caption` but the last one). When the Enter key is pressed, the program stops the operation, by resetting the `ItemIndex` property of the radio group control. Here is the code:

```
2: // type in caption
begin
  if Key = #8 then // backspace: remove last char
    Caption := Copy (Caption, 1, Length (Caption) - 1)
  else if Key = #13 then // enter: stop operation
    RadioPreview.ItemIndex := 0
  else // anything else: add character
    Caption := Caption + Key;
    Key := #0;
end;
```

Finally, if the last radio item is selected, the code checks whether the character is a vowel (by testing for its inclusion in a constant “vowel set”). In this case, the character is skipped altogether:

```
3: // skip vowels
  if Key in ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'] then
    Key := #0;
```

## Getting Mouse Input

When a user clicks one of the mouse buttons over a form (or over a component, by the way), Windows sends the application some messages. Delphi defines some events you can use to write code that responds to these messages. The two basic events are `OnMouseDown`, received when a mouse button is clicked, and `OnMouseUp`, received when the button is released. Another fundamental system message is related to mouse movement; the event is `OnMouseMove`. Although it should be easy to understand the meaning of the three messages—down, up, and move—the question that might arise is, how do they relate to the `OnClick` event we have often used up to now?

We have used the `OnClick` event for components, but it is also available for the form. Its general meaning is that the left mouse button has been clicked and released on the same window or

component. However, between these two actions, the cursor might have been moved outside the area of the window or component, while the left mouse button was held down.

Another difference between the `OnMouseXX` and `OnClick` events is that the latter relates only to the *left* mouse button. Most of the mouse types connected to a Windows PC have two mouse buttons, and some even have three. Usually we refer to these buttons as the left mouse button, generally used for selection; the right mouse button, for local menus; and the middle mouse button, seldom used. Nowadays most new mouse devices have a “button wheel” instead of the middle button. Users typically use the wheel for scrolling (causing an `OnMouseWheel` event), but they can also press it (generating the `OnMouseWheelDown` and `OnMouseWheelUp` events). Mouse wheel events are automatically converted into scrolling events.

## Using Windows without a Mouse

A user should always be able to use any Windows application without the mouse. This is not an option; it is a Windows programming rule. Of course, an application might be easier to use with a mouse, but that should never be mandatory. In fact, there are users who for various reasons might not have a mouse connected, such as travelers with a small laptop and no space, workers in industrial environments, and bank clerks with other peripherals around.

There is another reason to support the keyboard: Using the mouse is nice, but it tends to be slower. If you are a skilled touch typist, you won't use the mouse to drag a word of text; you'll use shortcut keys to copy and paste it, without moving your hands from the keyboard.

For all these reasons, you should always set up a proper tab order for a form's components, remember to add keys for buttons and menu items for keyboard selection, use shortcut keys on menu commands, and so on.

## The Parameters of the Mouse Events

All of the lower-level mouse events have the same parameters: the usual `Sender` parameter; a `Button` parameter indicating which of the three mouse buttons has been clicked (`mbRight`, `mbLeft`, or `mbCenter`); the `Shift` parameter indicating which of the *mouse-related keys* (`Alt`, `Ctrl`, and `Shift`, plus the three mouse buttons themselves) were pressed when the event occurred; and the `x` and `y` coordinates of the position of the mouse, in *client area* coordinates of the current window.

Using this information, it is very simple to draw a small circle in the position of a left mouse button-down event:

```
procedure TForm1.FormMouseDown(  
    Sender: TObject; Button: TMouseButton;
```

```
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Button = mbLeft then  
        Canvas.Ellipse (X-10, Y-10, X+10, Y+10);  
end;
```

**NOTE**

To draw on the form, we use a very special property: `Canvas`. A `TCanvas` object has two distinctive features: it holds a collection of drawing tools (such as a pen, a brush, and a font) and it has some drawing methods, which use the current tools. The kind of direct drawing code in this example is not correct, because the on-screen image is not persistent: moving another window over the current one will clear its output. The next example demonstrates the Windows “store-and-draw” approach.

## Dragging and Drawing with the Mouse

To demonstrate a few of the mouse techniques discussed so far, I’ve built a simple example based on a form without any component and called `MouseOne` in the VCL version and `QMouseOne` in the CLX version. The first feature of this program is that it displays in the Caption of the form the current position of the mouse:

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;  
    X, Y: Integer);  
begin  
    // display the position of the mouse in the caption  
    Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);  
end;
```

You can use this simple feature of the program to better understand how the mouse works. Make this test: run the program (this simple version or the complete one) and resize the windows on the desktop so that the form of the `MouseOne` or `QMouseOne` program is behind another window and inactive but with the title visible. Now move the mouse over the form, and you’ll see that the coordinates change. This means that the `OnMouseMove` event is sent to the application even if its window is not active, and it proves what I have already mentioned: Mouse messages are always directed to the window under the mouse. The only exception is the mouse capture operation I’ll discuss in this same example.

Besides showing the position in the title of the window, the `MouseOne/QMouseOne` example can track mouse movements by painting small pixels on the form if the user keeps the Shift key pressed. (Again this direct painting code produces non-persistent output.)

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;  
    X, Y: Integer);  
begin  
    // display the position of the mouse in the caption  
    Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
```

```
    if ssShift in Shift then
        // mark points in yellow
        Canvas.Pixels [X, Y] := c1Yellow;
    end;
```

**TIP**

The TCanvas class of the CLX library doesn't include a Pixels array. Instead, you can call the DrawPoint method after setting a proper color for the pen, as I've done in the QMouseOne example.

The real feature of this example, however, is the direct mouse-dragging support. Contrary to what you might think, Windows has no system support for dragging, which is implemented in VCL by means of lower-level mouse events and operations. (An example of dragging from one control to another was discussed in the last chapter.) In VCL, forms cannot originate dragging operations, so in this case we are obliged to use the low-level approach. The aim of this example is to draw a rectangle from the initial position of the dragging operation to the final one, giving the users some visual clue of the operation they are doing.

The idea behind dragging is quite simple. The program receives a sequence of button-down, mouse-move, and button-up messages. When the button is clicked, dragging begins, although the real actions take place only when the user moves the mouse (without releasing the mouse button) and when dragging terminates (when the button-up message arrives). The problem with this basic approach is that it is not reliable. A window usually receives mouse events only when the mouse is over its client area; so if the user clicks the mouse button, moves the mouse onto another window, and then releases the button, the second window will receive the button-up message.

There are two solutions to this problem. One (seldom used) is mouse clipping. Using a Windows API function (namely ClipCursor), you can force the mouse not to leave a certain area of the screen. When you try to move it outside the specified area, it stumbles against an invisible barrier. The second and more common solution is to capture the mouse. When a window captures the mouse, all the subsequent mouse input is sent to that window. This is the approach we will use for the MouseOne/QMouseOne example.

The code of the example is built around three methods: FormMouseDown, FormMouseMove, and FormMouseUp. Clicking the left mouse button over the form starts the process, setting the fDragging Boolean field of the form (which indicates that dragging is in action in the other two methods). The method also uses a TRect variable used to keep track of the initial and current position of the dragging. Here is the code:

```
procedure TMouseForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if Button = mbLeft then
```

```

begin
  fDragging := True;
  Mouse.Capture := Handle;
  fRect.Left := X;
  fRect.Top := Y;
  fRect.BottomRight := fRect.TopLeft;
  Canvas.DrawFocusRect (fRect);
end;
end;

```

An important action of this method is the call to the `SetCapture` API function, obtained by setting the `Capture` property of the global object `Mouse`. Now even if a user moves the mouse outside of the client area, the form still receives all mouse-related messages. You can see that for yourself by moving the mouse toward the upper-left corner of the screen; the program shows negative coordinates in the caption.

### TIP

The global `Mouse` object allows you to get global information about the mouse, such as its presence, its type, and the current position, as well as set some of its global features. This global object hides a few API functions, making your code simpler and more portable.

When dragging is active and the user moves the mouse, the program draws a dotted rectangle corresponding to the actual position. Actually, the program calls the `DrawFocusRect` method twice. The first time this method is called, it deletes the current image, thanks to the fact that two consecutive calls to `DrawFocusRect` simply reset the original situation. After updating the position of the rectangle, the program calls the method a second time:

```

procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
    begin
      // remove and redraw the dragging rectangle
      Canvas.DrawFocusRect (fRect);
      fRect.Right := X;
      fRect.Bottom := Y;
      Canvas.DrawFocusRect (fRect);
    end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
  end;

```



When the mouse button is released, the program terminates the dragging operation by resetting the `Capture` property of the `Mouse` object, which internally calls the `ReleaseCapture` API function, and by setting the value of the `fDragging` field to `False`:

```
procedure TMouseForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
    begin
      Mouse.Capture := 0; // calls ReleaseCapture
      fDragging := False;
      Invalidate;
    end;
end;
```

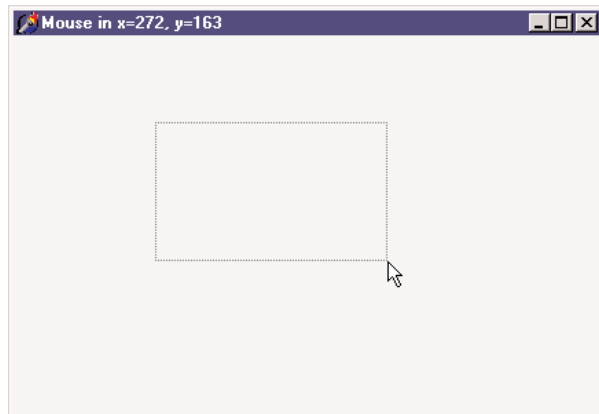
The final call, `Invalidate`, triggers a painting operation and executes the following `OnPaint` event handler:

```
procedure TMouseForm.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle (fRect.Left, fRect.Top, fRect.Right, fRect.Bottom);
end;
```

This makes the output of the form persistent, even if you hide it behind another form. Figure 9.6 shows a previous version of the rectangle and a dragging operation in action.

**FIGURE 9.6:**

The `MouseOne` example uses a dotted line to indicate, during a dragging operation, the final area of a rectangle.



**TIP**

Under Qt, there are no Windows handles, but the `Capture` property of the mouse is still available. You assign to it, however, the object of the component that has to capture the mouse (for example, `Self` to indicate the form), or set the property to `nil` to release it. You can see this code in the `QMouseOne` example.

## Painting in Windows

Why do we need to handle the `OnPaint` event to produce a proper output, and why can't we paint directly over the form canvas? It depends on Windows' default behavior. As you draw on a window, Windows does *not* store the resulting image. When the window is covered, its contents are usually lost.

The reason for this behavior is simple: to save memory. Windows assumes it's "cheaper" in the long run to redraw the screen using code than to dedicate system memory to preserving the display state of a window. It's a classic memory-versus-CPU-cycles trade-off. A color bitmap for a 300×400 image at 256 colors requires about 120 KB. By increasing the color count or the number of pixels, you can easily have full-screen bitmaps of about 1 MB and reach 4 MB of memory for a 1280×1024 resolution at 16 million colors. If storing the bitmap was the default choice, running half a dozen simple applications would require at least 8 MB of memory, if not 16 MB, just for remembering their current output.

In the event that you want to have a consistent output for your applications, there are two techniques you can use. The general solution is to store enough data about the output to be able to reproduce it when the system sends a *painting* requested. An alternative approach is to save the output of the form in a bitmap while you produce it, by placing an `Image` component over the form and drawing on the canvas of this image component.

The first technique, painting, is the common approach to handling output in Windows, aside from particular graphics-oriented programs that store the form's whole image in a bitmap. The approach used to implement painting has a very descriptive name: *store and paint*. In fact, when the user clicks a mouse button or performs any other operation, we need to store the position and other elements; then, in the painting method, we use this information to actually paint the corresponding image.

The idea of this approach is to let the application repaint its whole surface under any of the possible conditions. If we provide a method to redraw the contents of the form, and if this method is automatically called when a portion of the form has been hidden and needs repainting, we will be able to re-create the output properly.

Since this approach takes two steps, we must be able to execute these two operations in a row, asking the system to repaint the window—without waiting for the system to ask for this. You can use several methods to invoke repainting: `Invalidate`, `Update`, `Repaint`, and `Refresh`. The first two correspond to the Windows API functions, while the latter two have been introduced by Delphi.

- The `Invalidate` method informs Windows that the entire surface of the form should be repainted. The most important thing is that `Invalidate` does *not* enforce a painting operation immediately. Windows simply stores the request and will respond to it only

after the current procedure has been completely executed and as soon as there are no other events pending in the system. Windows deliberately delays the painting operation because it is one of the most time-consuming operations. At times, with this delay, it is possible to paint the form only after multiple changes have taken place, avoiding multiple consecutive calls to the (slow) `Paint` method.

- The `Update` method asks Windows to update the contents of the form, repainting it immediately. However, remember that this operation will take place only if there is an *invalid area*. This happens if the `Invalidate` method has just been called or as the result of an operation by the user. If there is no invalid area, a call to `Update` has no effect at all. For this reason, it is common to see a call to `Update` just after a call to `Invalidate`. This is exactly what is done by the two Delphi methods, `Repaint` and `Refresh`.
- The `Repaint` method calls `Invalidate` and `Update` in sequence. As a result, it activates the `OnPaint` event immediately. There is a slightly different version of this method called `Refresh`. For a form the effect is the same; for components it might be slightly different.

When you need to ask the form for a repaint operation, you should generally call `Invalidate`, following the standard Windows approach. This is particularly important when you need to request this operation frequently, because if Windows takes too much time to update the screen, the requests for repainting can be accumulated into a simple repaint action. The `WM_PAINT` message in Windows is a sort of low-priority message. To be more precise, if a request for repainting is pending but other messages are waiting, the other messages are handled before the system actually performs the paint action.

On the other hand, if you call `Repaint` several times, the screen must be repainted each time before Windows can process other messages, and because paint operations are computationally intensive, this can actually make your application less responsive. There are times, however, when you want the application to repaint a surface as quickly as possible. In these less-frequent cases, calling `Repaint` is the way to go.

#### NOTE

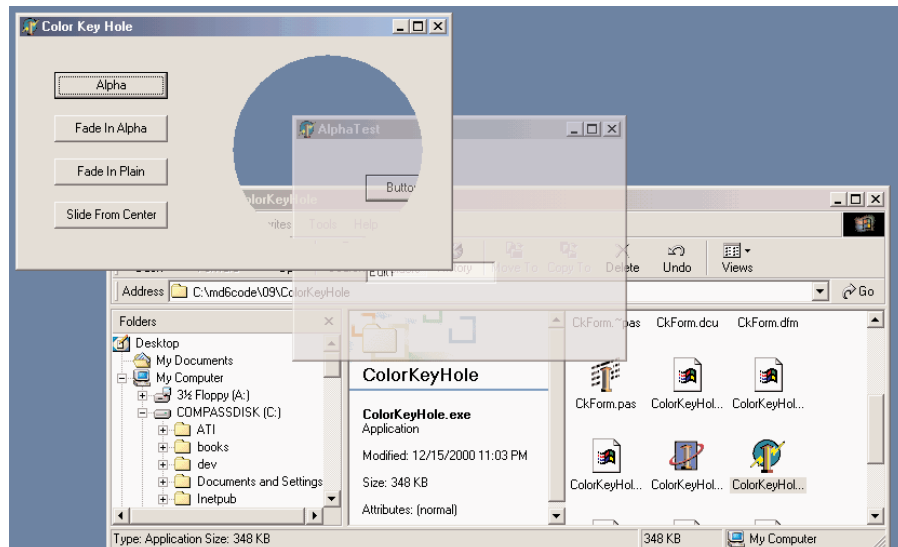
Another important consideration is that during a paint operation Windows redraws only the so-called *update region*, to speed up the operation. For this reason if you invalidate only a portion of a window, only that area will be repainted. To accomplish this you can use the `InvalidateRect` and `InvalidateRegion` functions. Actually, this feature is a double-edged sword. It is a very powerful technique, which can improve speed and reduce the flickering caused by frequent repaint operations. On the other hand, it can also produce incorrect output. A typical problem is when only some of the areas affected by the user operations are actually modified, while others remain as they were even if the system executes the source code that is supposed to update them. In fact, if a painting operation falls outside the update region, the system ignores it, as if it were outside the visible area of a window.

## Unusual Techniques: Alpha Blending, Color Key, and the Animate API

One of the few new features of Delphi 6 related to forms is support for some new Windows APIs regarding the way forms are displayed (not available under Qt/CLX). For a form, *alpha blending* allows you to merge the content of a form with what's behind it on the screen, something you'll rarely need, at least in a business application. The technique is certainly more interesting when applied to bitmap (with the new `AlphaBlend` and `AlphaDIBBlend` API functions) than to a form itself. In any case, by setting the `AlphaBlend` property of a form to `True` and giving to the `AlphaBlendValue` property a value lower than 255, you'll be able to see, in transparency, what's behind the form. The lower the `AlphaBlendValue`, the more the form will *fade*. You can see an example of alpha blending in Figure 9.7, taken from the `CkKeyHole` example

**FIGURE 9.7:**

The output of the `CkKeyHole`, showing the effect of the new `TransparentColor` and `AlphaBlend` properties, and also the `AnimateWindow` API.



This is not the only new Delphi feature in the area of what I can only call *unusual*. The second is the new `TransparentColor` property, which allows you to indicate a transparent color, which will be replaced by the background, creating a sort of hole in a form. The transparent color is indicated by the `TransparentColorValue` property. Again, you can see an example of this effect in Figure 9.7.

Finally, you can use a native Windows technique, animated display, which is not directly supported by Delphi (beyond the display of hints). For example, instead of calling the `Show` method of a form, you can write:

```
Form3.Hide;  
AnimateWindow (Form3.Handle, 2000, AW_BLEND);  
Form3.Show;
```

Notice you have to call the `Show` method at the end for the form to behave properly. A similar animation effect can also be obtained by changing the `AlphaBlendValue` in a loop. The `AnimateWindow` API can also be used to obtain the display of the form starting from the center (with the `AW_CENTER` flag) or from one of its sides (`AW_HOR_POSITIVE`, `AW_HOR_NEGATIVE`, `AW_VER_POSITIVE`, or `AW_VER_NEGATIVE`), as is common for slide shows.

This same function can also be applied to windowed controls, obtaining a fade-in effect instead of the usual direct appearance. I keep having serious doubts about the waste of CPU cycles these animations cause, but I have to say that if they are applied properly and in the right program, they can improve the user interface.

## Position, Size, Scrolling, and Scaling

Once you have designed a form in Delphi, you run the program, and you expect the form to show up exactly as you prepared it. However, a user of your application might have a different screen resolution or might want to resize the form (if this is possible, depending on the border style), eventually affecting the user interface. We've already discussed (mainly in Chapter 7) some techniques related to controls, such as alignment and anchors. Here I want to specifically address elements related to the form as a whole.

Besides differences in the user system, there are many reasons to change Delphi defaults in this area. For example, you might want to run two copies of the program and avoid having all the forms show up in exactly the same place. I've collected many other related elements, including form scrolling, in this portion of the chapter.

### The Form Position

There are a few properties you can use to set the position of a form. The `Position` property indicates how Delphi determines the initial position of the form. The default `poDesigned` value indicates that the form will appear where you designed it and where you use the positional (`Left` and `Top`) and size (`Width` and `Height`) properties of the form.

Some of the other choices (`poDefault`, `poDefaultPosOnly`, and `poDefaultSizeOnly`) depend on a feature of the operating system: using a specific flag, Windows can position and/or size new windows using a cascade layout. In this way, the positional and size properties you set at

design time will be ignored, but running the application twice you won't get overlapping windows. The default positions are ignored when the form has a dialog border style.

Finally, with the `poScreenCenter` value, the form is displayed in the center of the screen, with the size you set at design time. This is a very common setting for dialog boxes and other secondary forms.

Another property that affects the initial size and position of a window is its *state*. You can use the `WindowState` property at design time to display a maximized or minimized window at startup. This property, in fact, can have only three values: `wsNormal`, `wsMinimized`, and `wsMaximized`. The meaning of this property is intuitive. If you set a minimized window state, at startup the form will be displayed in the Windows Taskbar. For the main form of an application, this property can be automatically set by specifying the corresponding attributes in a shortcut referring to the application.

Of course, you can maximize or minimize a window at run time, too. Simply changing the value of the `WindowState` property to `wsMaximized` or to `wsNormal` produces the expected effect. Setting the property to `wsMinimized`, however, creates a minimized window that is placed over the Taskbar, not within it. This is not the expected action for a main form, but for a secondary form! The simple solution to this problem is to call the `Minimize` method of the `Application` object. There is also a `Restore` method in the `TApplication` class that you can use when you need to restore a form, although most often the user will do this operation using the `Restore` command of the system menu.

## The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the `Width` and `Height` properties or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the `OnResize` event, where you can perform custom actions to adapt the user interface to the new size of the form).

However, if you look at a form's properties in source code or in the online Help, you can see that there are two properties referring to its width and two referring to its height. `Height` and `Width` refer to the size of the form, including the borders; `ClientHeight` and `ClientWidth` refer to the size of the internal area of the form, excluding the borders, caption, scroll bars (if any), and menu bar. The client area of the form is the surface you can use to place components on the form, to create output, and to receive user input.

Since you might be interested in having a certain available area for your components, it often makes more sense to set the client size of a form instead of its global size. This is straightforward, because as you set one of the two client properties, the corresponding form property changes accordingly.

**TIP**

In Windows, it is also possible to create output and receive input from the nonclient area of the form—that is, its border. Painting on the border and getting input when you click it are complex issues. If you are interested, look in the Help file at the description of such Windows messages as `wm_NCPaint`, `wm_NCCalcSize`, and `wm_NCHitTest` and the series of nonclient messages related to the mouse input, such as `wm_NCLButtonDown`. The difficulty of this approach is in combining your code with the default Windows behavior.

## Form Constraints

When you choose a resizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the `wm_Size` message, which generates the `OnResize` event. `OnResize` takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

Delphi provides a specific property for forms and also for all controls: the `Constraints` property. Simply setting the subproperties of the `Constraints` property to the proper maximum and minimum values creates a form that cannot be resized beyond those limits. Here is an example:

```
object Form1: TForm1
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

Notice that as you set up the `Constraints` property, it has an immediate effect even at design time, changing the size of the form if it is outside the permitted area.

Delphi also uses the maximum constraints for maximized windows, producing an awkward effect. For this reason, you should generally disable the `Maximize` button of a window that has a maximum size. There are cases in which maximized windows with a limited size make sense—this is the behavior of Delphi's main window. In case you need to change constraints at run time, you can also consider using two specific events, `OnCanResize` and `OnConstrainedResize`. The first of the two can also be used to disable resizing a form or control in given circumstances.

## Scrolling a Form

When you build a simple application, a single form might hold all of the components you need. As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms. If you reduce the space occupied by the components, you

might add some capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen (or at least bigger than its visible portion on the screen).

Adding a scroll bar to a form is simple. In fact, you don't need to do anything. If you place several components in a big form and then reduce its size, a scroll bar will be added to the form automatically, as long as you haven't changed the value of the `AutoScroll` property from its default of `True`.

Along with `AutoScroll`, forms have two properties, `HorzScrollBar` and `VertScrollBar`, which can be used to set several properties of the two `TFormScrollBar` objects associated with the form. The `Visible` property indicates whether the scroll bar is present, the `Position` property determines the initial status of the scroll thumb, and the `Increment` property determines the effect of clicking one of the arrows at the ends of the scroll bar. The most important property, however, is `Range`.

The `Range` property of a scroll bar determines the virtual size of the form, not the actual range of values of the scroll bar. Suppose you need a form that will host several components and will therefore need to be 1000 pixels wide. We can use this value to set the "virtual range" of the form, changing the `Range` of the horizontal scroll bar.

The `Position` property of the scroll bar will range from 0 to 1000 minus the current size of the client area. For example, if the client area of the form is 300 pixels wide, you can scroll 700 pixels to see the far end of the form (the thousandth pixel).

## A Scroll Testing Example

To demonstrate the specific case I've just discussed, I've built the `Scroll1` example, which has a virtual form 1000 pixels wide. To accomplish this, I've set the range of the horizontal scroll bar to 1000:

```
object Form1: TForm1
  Width = 458
  Height = 368
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  Caption = 'Scrolling Form'
  OnResize = FormResize
  ...
```

The form of this example has been filled with meaningless list boxes, and I could have obtained the same scroll-bar range by placing the right-most list box so that its position (`Left`) plus its size (`Width`) would equal 1000.



The interesting part of the example is the presence of a toolbox window displaying the status of the form and of its horizontal scroll bar. This second form has four labels; two with fixed text and two with the actual output. Besides this, the secondary form (called Status) has a `bsToolWindow` border style and is a top-most window. You should also set its `Visible` property to `True`, to have its window automatically displayed at startup:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  Caption = 'Status'
  FormStyle = fsStayOnTop
  Visible = True
object Label1: TLabel1...
...

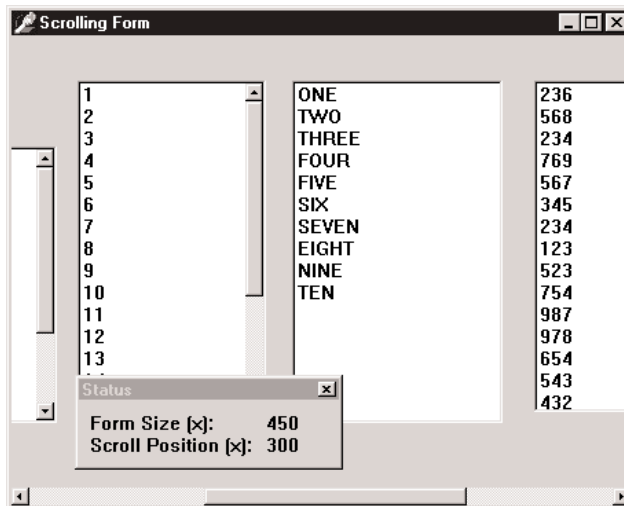
```

There isn't much code in this program. Its aim is to update the values in the toolbox each time the form is resized or scrolled (as you can see in Figure 9.8). The first part is extremely simple. You can handle the `OnResize` event of the form and simply copy a couple of values to the two labels. The labels are part of another form, so you need to prefix them with the name of the form instance, `Status`:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

**FIGURE 9.8:**

The output of the Scroll1 example



If we wanted to change the output each time the user scrolls the contents of the form, we could not use a Delphi event handler, because there isn't an `OnScroll` event for forms (although there is one for stand-alone `ScrollBar` components). Omitting this event makes sense, because Delphi forms handle scroll bars automatically in a powerful way. In Windows, by contrast, scroll bars are extremely low-level elements, requiring a lot of coding. Handling the scroll event makes sense only in special cases, such as when you want to keep track precisely of the scrolling operations made by a user.

Here is the code we need to write. First, add a method declaration to the class and associate it with the Windows horizontal scroll message (`wm_HScroll`):

```
public
  procedure FormScroll (var ScrollData: TWMScroll);
  message wm_HScroll;
```

Then write the code of this procedure, which is almost the same as the code of the `FormResize` method we've seen before:

```
procedure TForm1.FormScroll (var ScrollData: TWMScroll);
begin
  inherited;
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

It's important to add the call to `inherited`, which activates the method related to the same message in the base class form. The `inherited` keyword in Windows message handlers calls the method of the base class we are overriding, which is the one associated with the corresponding Windows message (even if the procedure name is different). Without this call, the form won't have its default scrolling behavior; that is, it won't scroll at all.

## NOTE

Because in CLX you cannot handle the low-level scroll messages, there seems to be no easy way to create a program similar to `Scroll1`. This isn't terribly important in real-world applications, as the scrolling system is automatic, and can probably be accomplished by hooking in the CLX library at a lower level.

## Automatic Scrolling

The scroll bar's `Range` property can seem strange until you start to use it consistently. When you think about it a little, you'll start to understand the advantages of the "virtual range" approach. First of all, the scroll bar is automatically removed from the form when the client area of the form is big enough to accommodate the virtual size; and when you reduce the size of the form, the scroll bar is added again.

This feature becomes particularly interesting when the `AutoScroll` property of the form is set to `True`. In this case, the extreme positions of the right-most and lower controls are automatically copied into the `Range` properties of the form's two scroll bars. Automatic scrolling works well in Delphi. In the last example, the virtual size of the form would be set to the right border of the last list box. This was defined with the following attributes:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Therefore, the horizontal virtual size of the form would be 977 (the sum of the two preceding values). This number is automatically copied into the `Range` field of the `HorzScrollBar` property of the form, unless you change it manually to have a bigger form (as I've done for the `Scroll1` example, setting it to 1000 to leave some space between the last list box and the border of the form). You can see this value in the Object Inspector, or make the following test: run the program, size the form as you like, and move the scroll thumb to the right-most position. When you add the size of the form and the position of the thumb, you'll always get 1000, the virtual coordinate of the right-most pixel of the form, whatever the size.

## Scrolling and Form Coordinates

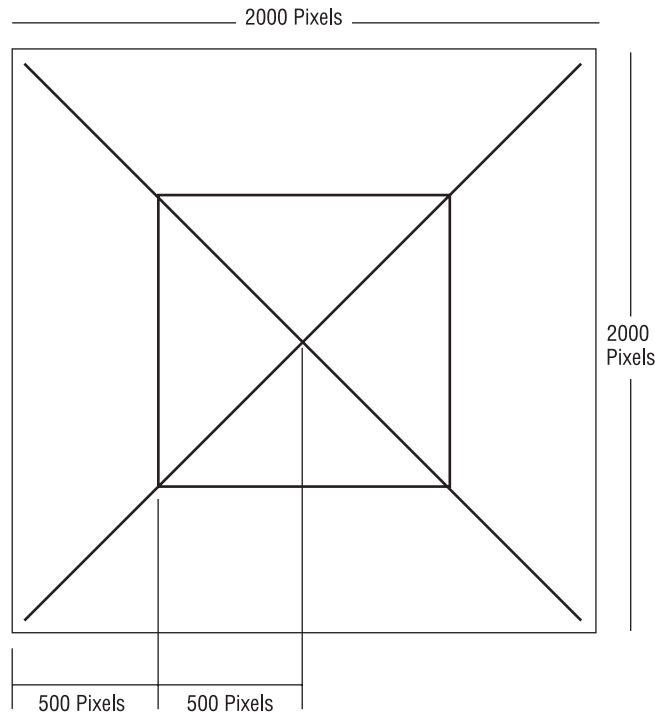
We have just seen that forms can automatically scroll their components. But what happens if you paint directly on the surface of the form? Some problems arise, but their solution is at hand. Suppose that we want to draw some lines on the virtual surface of a form, as shown in Figure 9.9.

Since you probably do not own a monitor capable of displaying 2000 pixels on each axis, you can create a smaller form, add two scroll bars, and set their `Range` property, as I've done in the `Scroll2` example. Here is the textual description of the form:

```
object Form1: TForm1
  HorzScrollBar.Range = 2000
  VertScrollBar.Range = 2000
  ClientHeight = 336
  ClientWidth = 472
  OnPaint = FormPaint
end
```

**FIGURE 9.9:**

The lines to draw on the virtual surface of the form



If we simply draw the lines using the virtual coordinates of the form, the image won't display properly. In fact, in the `OnPaint` response method, we need to compute the virtual coordinates ourselves. Fortunately, this is easy, since we know that the virtual `X1` and `Y1` coordinates of the upper-left corner of the client area correspond to the current positions of the two scroll bars:

```
procedure TForm1.FormPaint(Sender: TObject);
var
    X1, Y1: Integer;
begin
    X1 := HorzScrollBar.Position;
    Y1 := VertScrollBar.Position;

    // draw a yellow line
    Canvas.Pen.Width := 30;
    Canvas.Pen.Color := clYellow;
    Canvas.MoveTo (30-X1, 30-Y1);
    Canvas.LineTo (1970-X1, 1970-Y1);
    // and so on ...
```

As a better alternative, instead of computing the proper coordinate for each output operation, we can call the `SetWindowOrgEx` API to move the origin of the coordinates of the Canvas itself. This way, our drawing code will directly refer to virtual coordinates but will be displayed properly:

```
procedure TForm2.FormPaint(Sender: TObject);  
begin  
    SetWindowOrgEx (Canvas.Handle, HorzScrollbar.Position,  
        VertScrollbar.Position, nil);  
  
    // draw a yellow line  
    Canvas.Pen.Width := 30;  
    Canvas.Pen.Color := clYellow;  
    Canvas.MoveTo (30, 30);  
    Canvas.LineTo (1970, 1970);  
  
    // and so on ...
```

This is the version of the program you'll find in the source code on the CD. Try using the program and commenting out the `SetWindowOrgEx` call to see what happens if you don't use virtual coordinates: You'll find that the output of the program is not correct—it won't scroll, and the same image will always remain in the same position, regardless of scrolling operations. Notice also that the Qt/CLX version of the program, called `QScroll2`, doesn't use virtual coordinates but simply subtracts the scroll positions from each of the hard-coded coordinates.

## Scaling Forms

When you create a form with multiple components, you can select a fixed size border or let the user resize the form and automatically add scroll bars to reach the components falling outside the visible portion of the form, as we've just seen. This might also happen because a user of your application has a display driver with a much smaller number of pixels than yours.

Instead of simply reducing the form size and scrolling the content, you might want to reduce the size of each of the components at the same time. This automatically happens also if the user has a system font with a different pixel-per-inch ratio than the one you used for development. To address these problems, Delphi has some nice scaling features, but they aren't fully intuitive.

The form's `ScaleBy` method allows you to scale the form and each of its components. The `PixelsPerInch` and `Scaled` properties allow Delphi to resize an application automatically when the application is run with a different system font size, often because of a different screen resolution. In both cases, to make the form scale its window, be sure to also set the

`AutoScroll` property to `False`. Otherwise, the contents of the form will be scaled, but the form border itself will not. These two approaches are discussed in the next two sections.

**NOTE**

Form scaling is calculated based on the difference between the font height at run time and the font height at design time. Scaling ensures that edit and other controls are large enough to display their text using the user's font preferences without clipping the text. The form scales as well, as we will see later on, but the main point is to make edit and other controls readable.

## Manual Form Scaling

Any time you want to scale a form, including its components, you can use the `ScaleBy` method, which has two integer parameters, a multiplier and a divisor—it's a fraction. For example, with this statement the size of the current form is reduced to three-quarters of its original size:

```
ScaleBy (3, 4);
```

Generally, it is easier to use percentage values. The same effect can be obtained by using:

```
ScaleBy (75, 100);
```

When you scale a form, all the proportions are maintained, but if you go below or above certain limits, the text strings can alter their proportions slightly. The problem is that in Windows, components can be placed and sized only in whole pixels, while scaling almost always involves multiplying by fractional numbers. So any fractional portion of a component's origin or size will be truncated.

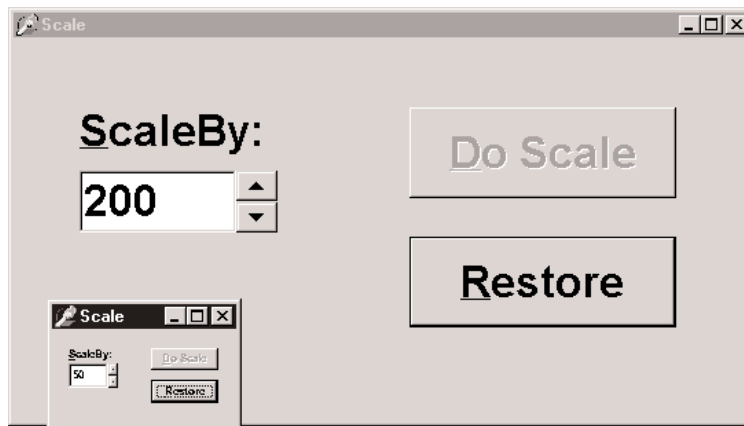
I've built a simple example, `Scale` or `QScale`, to show how you can scale a form manually, responding to a request by the user. The form of this application (see Figure 9.10) has two buttons, a label, an edit box, and an `UpDown` control connected to it (via its `Associate` property). With this setting, a user can type numbers in the edit box or click the two small arrows to increase or decrease the value (by the amount indicated by the `Increment` property). To extract the input value, you can use the `Text` property of the edit box or the `Position` of the `UpDown` control.

When you click the `Do Scale` button, the current input value is used to determine the scaling percentage of the form:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);  
begin  
    AmountScaled := UpDown1.Position;  
    ScaleBy (AmountScaled, 100);  
    UpDown1.Height := Edit1.Height;  
    ScaleButton.Enabled := False;  
    RestoreButton.Enabled := True;  
end;
```

**FIGURE 9.10:**

The form of the Scale example after a scaling with 50 and 200



This method stores the current input value in the form's `AmountScaled` private field and enables the Restore button, disabling the one that was clicked. Later, when the user clicks the Restore button, the opposite scaling takes place. By having to restore the form before another scaling operation takes place, I avoid an accumulation of round-off errors. I've added also a line to set the `Height` of the `UpDown` component to the same `Height` as the edit box it is attached to. This prevents small differences between the two, due to scaling problems of the `UpDown` control.

**NOTE**

If you want to scale the text of the form properly, including the captions of components, the items in list boxes, and so on, you should use TrueType fonts exclusively. The system font (MS Sans Serif) doesn't scale well. The font issue is important because the size of many components depends on the text height of their captions, and if the caption does not scale well, the component might not work properly. For this reason, in the Scale example I've used an Arial font.

Exactly the same scaling technique also works in CLX, as you can see by running the `QScale` example. The only real difference is that I have to replace the `UpDown` component (and the related Edit box) with a `SpinEdit` control, as the former is not available in Qt.

## Automatic Form Scaling

Instead of playing with the `ScaleBy` method, you can ask Delphi to do the work for you. When Delphi starts, it asks the system for the display configuration and stores the value in the `PixelsPerInch` property of the `Screen` object, a special global object of VCL, available in any application.

`PixelsPerInch` sounds like it has something to do with the pixel resolution of the screen, but unfortunately, it doesn't. If you change your screen resolution from 640×480 to 800×600 to 1024×768 or even 1600×1280, you will find that Windows reports the same `PixelsPerInch` value in all cases, unless you change the system font. What `PixelsPerInch` really refers to is the screen pixel resolution that the currently installed system font was designed for. When a user changes the system font scale, usually to make menus and other text easier to read, the user will expect all applications to honor those settings. An application that does not reflect user desktop preferences will look out of place and, in extreme cases, may be unusable to visually impaired users who rely on very large fonts and high-contrast color schemes.

The most common `PixelsPerInch` values are 96 (small fonts) and 120 (large fonts), but other values are possible. Newer versions of Windows even allow the user to set the system font size to an arbitrary scale. At design time, the `PixelsPerInch` value of the screen, which is a read-only property, is copied to every form of the application. Delphi then uses the value of `PixelsPerInch`, if the `Scaled` property is set to `True`, to resize the form when the application starts.

As I've already mentioned, both automatic scaling and the scaling performed by the `ScaleBy` method operate on components by changing the size of the font. The size of each control, in fact, depends on the font it uses. With automatic scaling, the value of the form's `PixelsPerInch` property (the design-time value) is compared to the current system value (indicated by the corresponding property of the `Screen` object), and the result is used to change the font of the components on the form. Actually, to improve the accuracy of this code, the final height of the text is compared to the design-time height of the text, and its size is adjusted if they do not match.

Thanks to Delphi automatic support, the same application running on a system with a different system font size automatically scales itself, without any specific code. The application's edit controls will be the correct size to display their text in the user's preferred font size, and the form will be the correct size to contain those controls. Although automatic scaling has problems in some special cases, if you comply with the following rules, you should get good results:

- Set the `Scaled` property of forms to `True`. (This is the default.)
- Use only `TrueType` fonts.
- Use Windows small fonts (96 dpi) on the computer you use to develop the forms.
- Set the `AutoScroll` property to `False`, if you want to scale the form and not just the controls inside it. (`AutoScroll` defaults to `True`, so don't forget to do this step.)
- Set the form position either near the upper-left corner or in the center of the screen (with the `poScreenCenter` value) to avoid having an out-of-screen form. Form position is discussed in the next section.



## Creating and Closing Forms

Up to now we have ignored the issue of form creation. We know that when the form is created, we receive the `OnCreate` event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in this project's source file:

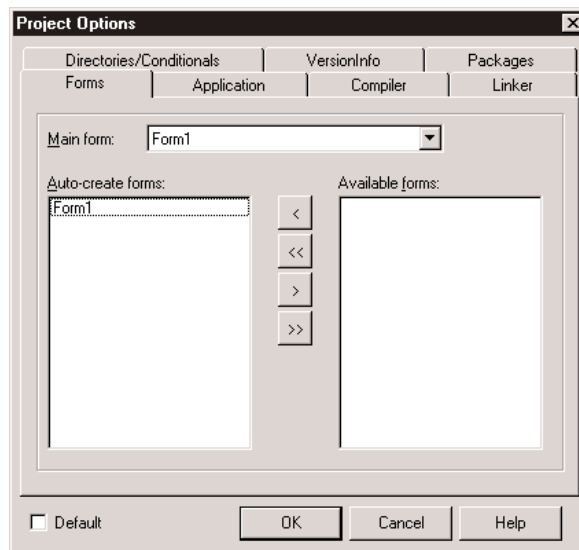
```
begin  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    Application.Run;  
end.
```

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box (see Figure 9.11). In this dialog box, you can decide whether the form should be automatically created. If you disable the automatic creation, the project's initialization code becomes the following:

```
begin  
    Applications.Initialize;  
    Application.Run;  
end.
```

**FIGURE 9.11:**

The Forms page of the Delphi Project Options dialog box



If you now run this program, nothing happens. It terminates immediately because no main window is created. So what is the effect of the call to the application's `CreateForm` method? It creates a new instance of the form class passed as the first parameter and assigns it to the variable passed as the second parameter.

Something else happens behind the scenes. When `CreateForm` is called, if there is currently no main form, the current form is assigned to the application's `MainForm` property. For this reason, the form indicated as `Main Form` in the dialog box shown in Figure 9.11 corresponds to the first call to the application's `CreateForm` method (that is, when several forms are created at start-up).

The same holds for closing the application. Closing the main form terminates the application, regardless of the other forms. If you want to perform this operation from the program's code, simply call the `Close` method of the main form, as we've done several times in past examples.

**TIP**

You can control the automatic creation of secondary forms by using the `Auto Create Forms` check box on the `Preferences` page of the `Environment Options` dialog box.

## Form Creation Events

Regardless of the manual or automatic creation of forms, when a form is created, there are many events you can intercept. Form-creation events are fired in the following order:

1. `OnCreate` indicates that the form is being created.
2. `OnShow` indicates that the form is being displayed. Besides main forms, this event happens after you set the `Visible` property of the form to `True` or call the `Show` or `ShowModal` methods. This event is fired again if the form is hidden and then displayed again.
3. `OnActivate` indicates that the form becomes the active form within the application. This event is fired every time you move from another form of the application to the current one.
4. Other events, including `OnResize` and `OnPaint`, indicate operations always done at start-up but then repeated many times.

As you can see in the list above, every event has a specific role apart from form initialization, except for the `OnCreate` event, which is guaranteed to be called only once as the form is created.

However, there is an alternative approach to adding initialization code to a form: overriding the constructor. This is usually done as follows:

```
constructor TForm1.Create(AOwner: TComponent);  
begin  
    inherited Create (AOwner);  
    // extra initialization code  
end;
```

Before the call to the `Create` method of the base class, the properties of the form are still not loaded and the internal components are not available. For this reason the standard approach is to call the base class constructor first and then do the custom operations.

## Old and New Creation Orders

Now the question is whether these custom operations are executed before or after the `OnCreate` event is fired. The answer depends on the value of the `OldCreateOrder` property of the form, introduced in Delphi 4 for backward compatibility with earlier versions of Delphi. By default, for a new project, all of the code in the constructor is executed before the `OnCreate` event handler. In fact, this event handler is not activated by the base class constructor but by its `AfterConstruction` method, a sort of constructor introduced for compatibility with C++Builder.

To study the creation order and the potential problems, you can examine the `CreatOrd` program. This program has an `OnCreate` event handler, which creates a list box control dynamically. The constructor of the form can access this list box or not, depending on the value of the `OldCreateOrder` property.

## Closing a Form

When you close the form using the `Close` method or by the usual means (Alt+F4, the system menu, or the Close button), the `OnCloseQuery` event is called. In this event, you can ask the user to confirm the action, particularly if there is unsaved data in the form. Here is a simple scheme of the code you can write:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);  
begin  
    if MessageDlg ('Are you sure you want to exit?', mtConfirmation,  
        [mbYes, mbNo], 0) = idNo then  
        CanClose := False;  
end;
```

If `OnCloseQuery` indicates that the form should still be closed, the `OnClose` event is called. The third step is to call the `OnDestroy` event, which is the opposite of the `OnCreate` event and is generally used to de-allocate objects related to the form and free the corresponding memory.

### NOTE

To be more precise, the `BeforeDestruction` method generates an `OnDestroy` event before the `Destroy` destructor is called. That is, unless you have set the `OldCreateOrder` property to `True`, in which case Delphi uses a different closing sequence.

So what is the use of the intermediate `OnClose` event? In this method, you have another chance to avoid closing the application, or you can specify alternative “close actions.” The method, in fact, has an `Action` parameter passed by reference. You can assign the following values to this parameter:

**caNone** The form is not allowed to close. This corresponds to setting the `CanClose` parameter of the `OnCloseQuery` method to `False`.

**caHide** The form is not closed, just hidden. This makes sense if there are other forms in the application; otherwise, the program terminates. This is the default for secondary forms, and it’s the reason I had to handle the `OnClose` event in the previous example to actually close the secondary forms.

**caFree** The form is closed, freeing its memory, and the application eventually terminates if this was the main form. This is the default action for the main form and the action you should use when you create multiple forms dynamically (if you want to remove the Windows and destroy the corresponding Delphi object as the form closes).

**caMinimize** The form is not closed but only minimized. This is the default action for MDI child forms.

#### NOTE

When a user shuts down Windows, the `OnCloseQuery` event is activated, and a program can use it to stop the shut-down process. In this case, the `OnClose` event is not called even if `OnCloseQuery` sets the `CanClose` parameter to `True`.

## Dialog Boxes and Other Secondary Forms

When you write a program, there is really no big difference between a dialog box and another secondary form, aside from the border, the border icons, and similar user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have non-modal—or *modeless*—dialog boxes. So if you think that dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. We have to consider two different elements:

- The form’s border and its user interface determine whether it looks like a dialog box.
- The use of two different methods (`Show` or `ShowModal`) to display the secondary form determines its behavior (modeless or modal).

## Adding a Second Form to a Program

To add a second form to an application, you simply click on the New Form button on the Delphi toolbar or use the File > New Form menu command. As an alternative you can select File > New, move to the Forms or Dialogs page, and choose one of the available form templates or form wizards.

If you have two forms in a project, you can use the Select Form or Select Unit button of the Delphi toolbar to navigate through them at design time. You can also choose which form is the main one and which forms should be automatically created at start-up using the Forms page of the Project Options dialog box. This information is reflected in the source code of the project file.

### TIP

Secondary forms are automatically created in the project source-code file depending on a new Delphi 5 setting, which is the Auto Create Forms check box of the Preferences page of the Environment Options dialog box. Although automatic creation is the simplest and most reliable approach for novice developers and quick-and-dirty projects, I suggest that you disable this check box for any serious development. When your application contains hundreds of forms, you really shouldn't have them all created at application start-up. Create instances of secondary forms when and where you need them, and free them when you're done.

Once you have prepared the secondary form, you can simply set its `Visible` property to `True`, and both forms will show up as the program starts. In general, the secondary forms of an application are left “invisible” and are then displayed by calling the `Show` method (or setting the `Visible` property at run time). If you use the `Show` function, the second form will be displayed as modeless, so you can move back to the first one while the second is still visible. To close the second form, you might use its system menu or click a button or menu item that calls the `Close` method. As we've just seen, the default close action (see the `OnClose` event) for a secondary form is simply to hide it, so the secondary form is not destroyed when it is closed. It is kept in memory (again, not always the best approach) and is available if you want to show it again.

## Creating Secondary Forms at Run Time

Unless you create all the forms when the program starts, you'll need to check whether a form exists and create it if necessary. The simplest case is when you want to create multiple copies of the same form at run time. In the `MultiWin/QMultiWin` example, I've done this by writing the following code:

```
procedure TForm1.btnMultipleClick(Sender: TObject);
begin
    with TForm3.Create (Application) do
        Show;
end;
```

Every time you click the button, a new copy of the form is created. Notice that I don't use the `Form3` global variable, because it doesn't make much sense to assign this variable a new value every time you create a new form object. The important thing, however, is not to refer to the global `Form3` object in the code of the form itself or in other portions of the application. The `Form3` variable, in fact, will invariably be a pointer to `nil`. My suggestion, in such a case, is to actually remove it from the unit to avoid any confusion.

**TIP**

In the code of a form, you should never explicitly refer to the form by using the global variable that Delphi sets up for it. For example, suppose that in the code of `TForm3` you refer to `Form3.Caption`. If you create a second object of the same type (the class `TForm3`), the expression `Form3.Caption` will invariably refer to the caption of the form object referenced by the `Form3` variable, which might not be the current object executing the code. To avoid this problem, refer to the `Caption` property in the form's method to indicate the caption of the current form object, and use the `Self` keyword when you need a specific reference to the object of the current form. To avoid any problem when creating multiple copies of a form, I suggest removing the global form object from the interface portion of the unit declaring the form. This global variable is required only for the automatic form creation.

When you create multiple copies of a form dynamically, remember to destroy each form object as is it closed, by handling the corresponding event:

```
procedure TForm3.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := caFree;  
end;
```

Failing to do so will result in a lot of memory consumption, because all the forms you create (both the windows and the Delphi objects) will be kept in memory and simply hidden from view.

## Creating Single-Instance Secondary Forms

Now let us focus on the dynamic creation of a form, in a program that accounts for only one copy of the form at a time. Creating a modal form is quite simple, because the dialog box can be destroyed when it is closed, with code like this:

```
procedure TForm1.btnModalClick(Sender: TObject);  
var  
    Modal: TForm4;  
begin  
    Modal := TForm4.Create (Application);  
    try  
        Modal.ShowModal;  
    finally  
        Modal.Free;  
    end;  
end;
```

Because the `ShowModal` call can raise an exception, you should write it in a `finally` block to make sure the object will be de-allocated. Usually this block also includes code that initializes the dialog box before displaying it and code that extracts the values set by the user before destroying the form. The final values are read-only if the result of the `ShowModal` function is `mrOK`, as we'll see in the next example.

The situation is a little more complex when you want to display only one copy of a modeless form. In fact, you have to create the form, if it is not already available, and then show it:

```
procedure TForm1.btnSingleClick(Sender: TObject);  
begin  
    if not Assigned (Form2) then  
        Form2 := TForm2.Create (Application);  
        Form2.Show;  
end;
```

With this code, the form is created the first time it is required and then is kept in memory, visible on the screen or hidden from view. To avoid using up memory and system resources unnecessarily, you'll want to destroy the secondary form when it is closed. You can do that by writing a handler for the `OnClose` event:

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := caFree;  
    // important: set pointer to nil!  
    Form2 := nil;  
end;
```

Notice that after we destroy the form, the global `Form2` variable is set to `nil`. Without this code, closing the form would destroy its object, but the `Form2` variable would still refer to the original memory location. At this point, if you try to show the form once more with the `btnSingleClick` method shown earlier, the `if not Assigned` test will succeed, as it simply checks whether the `Form2` variable is `nil`. The code fails to create a new object, and the `Show` method, invoked on a nonexistent object, will result in a system memory error.

As an experiment, you can generate this error by removing the last line of the listing above. As we have seen, the solution is to set the `Form2` object to `nil` when the object is destroyed, so that properly written code will “see” that a new form has to be created before using it. Again, experimenting with the `MultiWin/QMultiWin` example can prove useful to test various conditions. I haven't illustrated any screens from this example because the forms it displays are quite bare (totally empty except for the main form, which has three buttons).

## NOTE

Setting the form variable to `nil` makes sense—and works—if there is to be only one instance of the form present at any given instant. If you want to create multiple copies of a form, you'll have to use other techniques to keep track of them. Also keep in mind that in this case we cannot use the `FreeAndNil` procedure, because we cannot call `Free` on `Form2`. The reason is that we cannot destroy the form before its event handlers have finished executing.

## Creating a Dialog Box

I stated earlier in this chapter that a dialog box is not very different from other forms. To build a dialog box instead of a form, you just select the `bsDialog` value for the `BorderStyle` property. With this simple change, the interface of the form becomes like that of a dialog box, with no system icon, and no Minimize or Maximize boxes. Of course, such a form has the typical thick dialog box border, which is non-resizable.

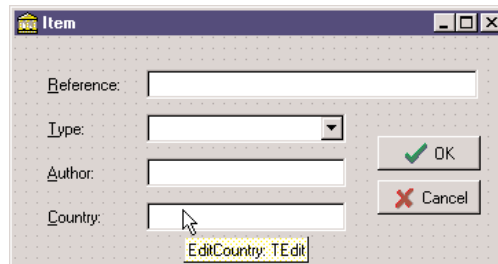
Once you have built a dialog box form, you can display it as a modal or modeless window using the two usual show methods (`Show` and `ShowModal`). Modal dialog boxes, however, are more common than modeless ones. This is exactly the reverse of forms; modal forms should generally be avoided, because a user won't expect them.

### The Dialog Box of the RefList Example

In Chapter 6 we explored the `RefList/QRefList` program, which used a `ListView` control to display references to books, magazines, Web sites, and more. In the `RefList2` version on the CD (and its `QRefList2 CLX` counterpart), I'll simply add to the basic version of that program a dialog box, used in two different circumstances: adding new items to the list and editing existing items. You can see the form of the dialog box in Figure 9.12 and its textual description in the following listing (detailed because it has many interesting features, so I suggest you read this code with care).

**FIGURE 9.12:**

The form of the dialog box of the `RefList2` example at design time



```

object FormItem: TFormItem
  Caption = 'Item'
  Color = clBtnFace
  Position = poScreenCenter
  object Label1: TLabel
    Caption = '&Reference:'
    FocusControl = EditReference
  end
  object EditReference: TEdit...
  object Label2: TLabel

```



```

    Caption = '&Type:'
    FocusControl = ComboType
end
object ComboType: TComboBox
    Style = csDropDownList
    Items.Strings = (
        'Book'
        'CD'
        'Magazine'
        'Mail Address'
        'Web Site')
end
object Label3: TLabel
    Caption = '&Author:'
    FocusControl = EditAuthor
end
object EditAuthor: TEdit...
object Label4: TLabel
    Caption = '&Country:'
    FocusControl = EditCountry
end
object EditCountry: TEdit...
object BitBtn1: TBitBtn
    Kind = bkOK
end
object BitBtn2: TBitBtn
    Kind = bkCancel
end
end
end

```

**TIP**

The items of the combo box in this dialog describe the available images of the image list so that a user can select the type of the item and the system will show the corresponding glyph. An even better option would have been to show those glyphs in a graphical combo box, along with their descriptions.

As I mentioned, this dialog box is used in two different cases. The first takes place as the user selects File ➤ Add Items from the menu:

```

procedure TForm1.AddItem1Click(Sender: TObject);
var
   NewItem: TListItem;
begin
    FormItem.Caption := 'New Item';
    FormItem.Clear;
    if FormItem.ShowModal = mrOK then
    begin

```

```

   NewItem := ListView1.Items.Add;
NewItem.Caption := FormItem.EditReference.Text;
NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
NewItem.SubItems.Add (FormItem.EditAuthor.Text);
NewItem.SubItems.Add (FormItem.EditCountry.Text);
end;
end;

```

Besides setting the proper caption of the form, this procedure needs to initialize the dialog box, as we are entering a brand-new value. If the user clicks OK, however, the program adds a new item to the list view and sets all its values. To empty the edit boxes of the dialog, the program calls the custom `Clear` method, which resets the text of each edit box control:

```

procedure TFormItem.Clear;
var
    I: Integer;
begin
    // clear each edit box
    for I := 0 to ControlCount - 1 do
        if Controls [I] is TEdit then
            TEdit (Controls[I]).Text := '';
end;

```

Editing an existing item requires a slightly different approach. First, the current values are moved to the dialog box before it is displayed. Second, if the user clicks OK, the program modifies the current list item instead of creating a new one. Here is the code:

```

procedure TForm1.ListView1Db1Click(Sender: TObject);
begin
    if ListView1.Selected <> nil then
        begin
            // dialog initialization
            FormItem.Caption := 'Edit Item';
            FormItem.EditReference.Text := ListView1.Selected.Caption;
            FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
            FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
            FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

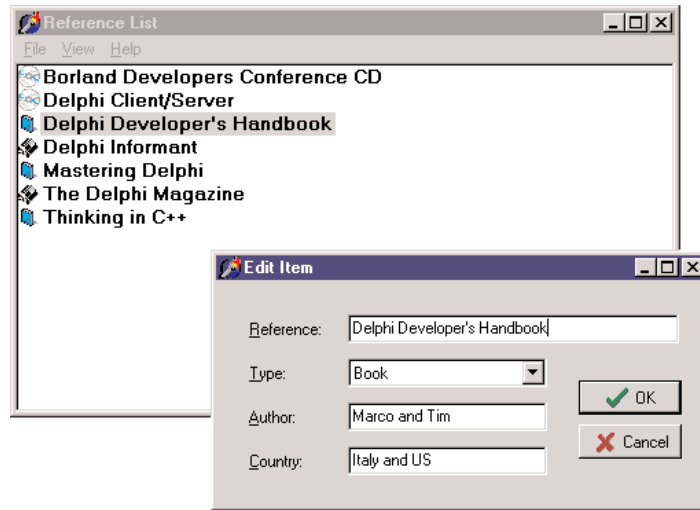
            // show it
            if FormItem.ShowModal = mrOK then
                begin
                    // read the new values
                    ListView1.Selected.Caption := FormItem.EditReference.Text;
                    ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
                    ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
                    ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
                end;
            end;
        end;
end;

```

You can see the effect of this code in Figure 9.13. Notice that the code used to read the value of a new item or modified one is similar. In general, you should try to avoid this type of duplicated code and possibly place the shared code statements in a method added to the dialog box. In this case, the method could receive as parameter a `TListItem` object and copy the proper values into it.

**FIGURE 9.13:**

The dialog box of the `RefList2` example used in edit mode



**NOTE**

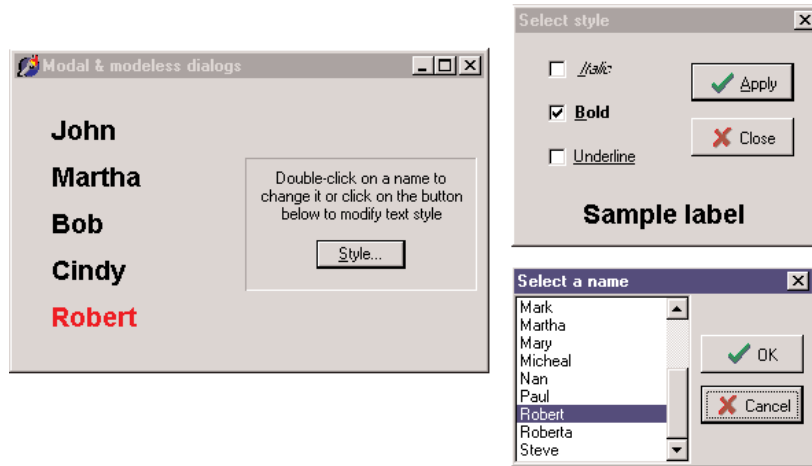
What happens internally when the user clicks the OK or Cancel button of the dialog box? A modal dialog box is closed by setting its `ModalResult` property, and it returns the value of this property. You can indicate the return value by setting the `ModalResult` property of the button. When the user clicks on the button, its `ModalResult` value is copied to the form, which closes the form and returns the value as the result of the `ShowModal` function.

## A Modeless Dialog Box

The second example of dialog boxes shows a more complex modal dialog box that uses the standard approach as well as a modeless dialog box. The main form of the `DlgApply` example (and of the identical CLX-based `QDlgApply` demo) has five labels with names, as you can see in Figure 9.14 and by viewing the source code on the companion CD.

**FIGURE 9.14:**

The three forms (a main form and two dialog boxes) of the DlgApply example at run time



If the user clicks a name, its color turns to red; if the user double-clicks it, the program displays a modal dialog box with a list of names to choose from. If the user clicks the Style button, a modeless dialog box appears, allowing the user to change the font style of the main form's labels. The five labels of the main form are connected to two methods, one for the `OnClick` event and the second for the `OnDoubleClick` event. The first method turns the last label a user has clicked to red, resetting to black all the others (which have the `Tag` property set to 1, as a sort of group index). Notice that the same method is associated with all of the labels:

```

procedure TForm1.LabelClick(Sender: TObject);
var
    I: Integer;
begin
    for I := 0 to ComponentCount - 1 do
        if (Components[I] is TLabel) and (Components[I].Tag = 1) then
            TLabel (Components[I]).Font.Color := clBlack;
        // set the color of the clicked label to red
        (Sender as TLabel).Font.Color := clRed;
end;

```

The second method common to all of the labels is the handler of the `OnDoubleClick` event. The `LabelDoubleClick` method selects the `Caption` of the current label (indicated by the `Sender` parameter) in the list box of the dialog and then shows the modal dialog box. If the user closes the dialog box by clicking OK and an item of the list is selected, the selection is copied back to the label's caption:

```

procedure TForm1.LabelDoubleClick(Sender: TObject);
begin
    with ListDial.ListBox1 do

```

```

begin
    // select the current name in the list box
    ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
    // show the modal dialog box, checking the return value
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
        // copy the selected item to the label
        (Sender as TLabel).Caption := Items [ItemIndex];
    end;
end;

```

**TIP**

Notice that all the code used to customize the modal dialog box is in the `LabelDoubleClick` method of the main form. The form of this dialog box has no added code.

The modeless dialog box, by contrast, has a lot of coding behind it. The main form simply displays the dialog box when the `Style` button is clicked (notice that the button caption ends with three dots to indicate that it leads to a dialog box), by calling its `Show` method. You can see the dialog box running in Figure 9.14 above.

Two buttons, `Apply` and `Close`, replace the `OK` and `Cancel` buttons in a modeless dialog box. (The fastest way to obtain these buttons is to select the `bkOK` or `bkCancel` value for the `Kind` property and then edit the `Caption`.) At times, you may see a `Cancel` button that works as a `Close` button, but the `OK` button in a modeless dialog box usually has no meaning. Instead, there might be one or more buttons that perform specific actions on the main window, such as `Apply`, `Change Style`, `Replace`, `Delete`, and so on.

If the user clicks one of the check boxes of this modeless dialog box, the style of the sample label's text at the bottom changes accordingly. You accomplish this by adding or removing the specific flag that indicates the style, as in the following `OnClick` event handler:

```

procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);
begin
    if ItalicCheckBox.Checked then
        LabelSample.Font.Style := LabelSample.Font.Style + [fsItalic]
    else
        LabelSample.Font.Style := LabelSample.Font.Style - [fsItalic];
    end;
end;

```

When the user selects the `Apply` button, the program copies the style of the sample label to each of the form's labels, rather than considering the values of the check boxes:

```

procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
begin
    Form1.Label1.Font.Style := LabelSample.Font.Style;
    Form1.Label2.Font.Style := LabelSample.Font.Style;
    ...
end;

```

As an alternative, instead of referring to each label directly, you can look for it by calling the `FindComponent` method of the form, passing the label name as a parameter, and then casting the result to the `TLabel` type. The advantage of this approach is that we can create the names of the various labels with a `for` loop:

```
procedure TStyleDialog.ApplyBitBtnClick(Sender: TObject);  
var  
    I: Integer;  
begin  
    for I := 1 to 5 do  
        (Form1.FindComponent ('Label' + IntToStr (I)) as TLabel).Font.Style :=  
            LabelSample.Font.Style;  
end;
```

**TIP**

The `ApplyBitBtnClick` method could also be written by scanning the `Controls` array in a loop, as I've already done in other examples. I decided to use the `FindComponent` method, instead, to demonstrate a different technique.

This second version of the code is certainly slower, because it has more operations to do, but you won't notice the difference, because it is very fast anyway. Of course, this second approach is also more flexible; if you add a new label, you only need to fix the higher limit of the `for` loop, provided all the labels have consecutive numbers. Notice that when the user clicks the `Apply` button, the dialog box does not close. Only the `Close` button has this effect. Consider also that this dialog box needs no initialization code because the form is not destroyed, and its components maintain their status each time the dialog box is displayed.

## Predefined Dialog Boxes

Besides building your own dialog boxes, Delphi allows you to use some default dialog boxes of various kinds. Some are predefined by Windows; others are simple dialog boxes (such as message boxes) displayed by a Delphi routine. The Delphi Component Palette contains a page of dialog box components. Each of these dialog boxes—known as *Windows common dialogs*—is defined in the system library `ComDlg32.DLL`.

### Windows Common Dialogs

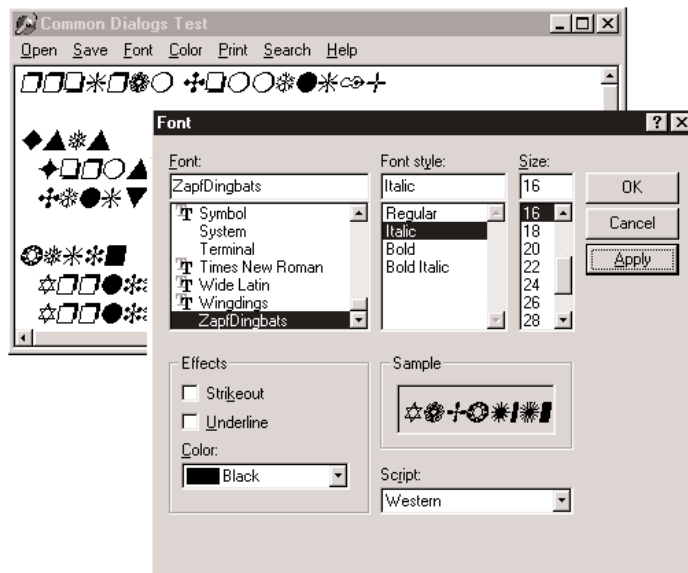
I have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the `Execute` method, returning a Boolean value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, I've built the `CommDlg` test program.

What I want to do is simply highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The Open Dialog Component can be customized by setting different file extensions filters, using the `Filter` property, which has a handy editor and can be assigned directly with a string like `Text File (*.txt)|*.txt`. Another handy feature is to let the dialog check whether the extension of the selected file matches the default extension, by checking the `ofExtensionDifferent` flag of the `Options` property after executing the dialog. Finally, this dialog allows multiple selections by setting its `ofAllowMultiSelect` option. In this case you can get the list of the selected files by looking at the `Files` string list property.
- The `SaveDialog` component is used in similar ways and has similar properties, although you cannot select multiple files, of course.
- The `OpenPictureDialog` and `SavePictureDialog` components provide similar features but have a customized form, which shows a preview of an image. Of course, it makes sense to use them only for opening or saving graphical files.
- The `FontDialog` component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (WYSIWYG), or only TrueType fonts. You can show or hide the portion related to the special effects, and obtain other different versions by setting its `Options` property. You can also activate an `Apply` button simply by providing an event handler for its `OnApply` event and using the `fdApplyButton` option. A Font dialog box with an `Apply` button (see Figure 9.15) behaves almost like a modeless dialog box (but isn't one).

**FIGURE 9.15:**

The Font selection dialog box with an `Apply` button



- The `ColorDialog` component is used with different options, to show the dialog fully open at first or to prevent it from opening fully. These settings are the `cdFullOpen` or `cdPreventFullOpen` values of the `Options` property.
- The `Find` and `Replace` dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the `CommDlg` example. The custom code is connected to the buttons of the two dialog boxes by providing the `OnFind` and `OnReplace` events.

**NOTE**

Qt offers a similar set of predefined dialog boxes, only the set of options is often more limited. I've created the `QCommDlg` version of the example you can use to experiment with these settings. The `CLX` program has fewer menu items, as some of the options are not available and there are other minimal changes in the source code.

## A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. There are many Delphi procedures and functions you can use to display simple dialog boxes:

- The `MessageDlg` function shows a customizable message box, with one or more buttons and usually a bitmap. The `MessageDlgPos` function is similar to the `MessageDlg` function, but the message box is displayed in a given position, not in the center of the screen.
- The `ShowMessage` procedure displays a simpler message box, with the application name as the caption and just an OK button. The `ShowMessagePos` procedure does the same, but you also indicate the position of the message box. The `ShowMessageFmt` procedure is a variation of `ShowMessage`, which has the same parameters as the `Format` function. It corresponds to calling `Format` inside a call to `ShowMessage`.
- The `MessageBox` method of the `Application` object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the `MessageBox` function of the Windows API, which passes as a main window parameter the handle of the `Application` object. This handle is required to make the message box behave like a modal window.
- The `InputBox` function asks the user to input a string. You provide the caption, the query, and a default string. The `InputQuery` function asks the user to input a string, too. The only difference between this and the `InputBox` function is in the syntax. The `InputQuery` function has a Boolean return value that indicates whether the user has clicked OK or Cancel.



To demonstrate some of the message boxes available in Delphi, I've written another sample program, with a similar approach to the preceding `CommDlg` example. In the `MBParade` example, you have a high number of choices (radio buttons, check boxes, edit boxes, and spin edit controls) to set before you click one of the buttons that displays a message box. The similar `QMbParade` example misses only the possibility of the help button, not available in the CLX message boxes.

## About Boxes and Splash Screens

Applications usually have an About box, where you can display information, such as the version of the product, a copyright notice, and so on. The simplest way to build an About box is to use the `MessageDlg` function. With this method, you can show only a limited amount of text and no special graphics.

Therefore, the usual method for creating an About box is to use a dialog box, such as the one generated with one of the Delphi default templates. In this about box you might want to add some code to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

## Building a Splash Screen

Another typical technique used in applications is to display an initial screen before the main form is shown. This makes the application seem more responsive, because you show something to the user while the program is loading, but it also makes a nice visual effect. Sometimes, this same window is displayed as the application's About box.

For an example in which a splash screen is particularly useful, I've built a program displaying a list box filled with prime numbers. The prime numbers are computed on program startup, so that they are displayed as soon as the form becomes visible:

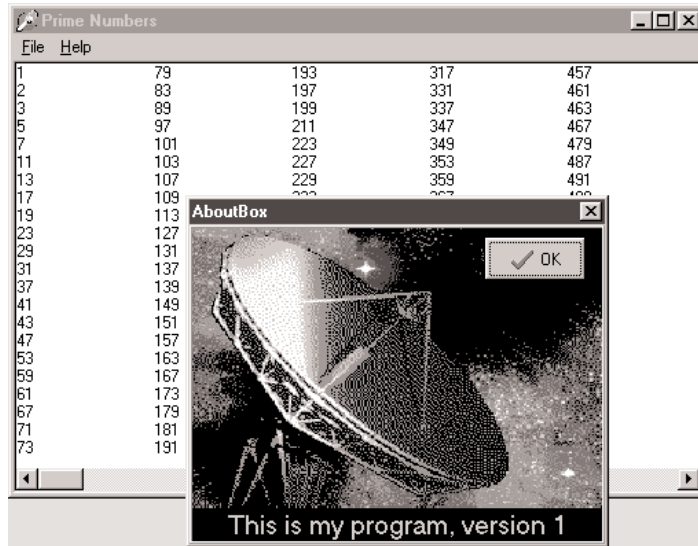
```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    I: Integer;  
begin  
    for I := 1 to 30000 do  
        if IsPrime (I) then  
            ListBox1.Items.Add (IntToStr (I));  
end;
```

This method calls an `IsPrime` function I've added to the program. This function, which you can find in the source code, computes prime numbers in a terribly slow way; but I needed a slow form creation to demonstrate my point. The numbers are added to a list box

that covers the full client area of the form and allows multiple columns to be displayed, as you can see in Figure 9.16.

**FIGURE 9.16:**

The main form of the Splash example, with the About box activated from the menu



There are three versions of the Splash program (plus the three corresponding CLX versions). As you can see by running the Splash0 example, the problem with this program is that the initial operation, which takes place in the `FormCreate` method, takes a lot of time. When you start the program, it takes several seconds to display the main form. If your computer is very fast or very slow, you can change the upper limit of the for loop of the `FormCreate` method to make the program faster or slower.

This program has a simple dialog box with an image component, a simple caption, and a bitmap button, all placed inside a panel taking up the whole surface of the About box. This form is displayed when you select the `Help > About` menu item. But what we really want is to display this About box while the program starts. You can see this effect by running the Splash1 and Splash2 examples, which show a splash screen using two different techniques.

First of all, I've added a method to the `TAboutBox` class. This method, called `MakeSplash`, changes some properties of the form to make it suitable for a splash form. Basically it removes the border and caption, hides the OK button, makes the border of the panel thick (to replace the border of the form), and then shows the form, repainting it immediately:

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
  Panel1.BorderWidth := 3;
end
```

```
Show;
Update;
end;
```

This method is called after creating the form in the project file of the Splash1 example. This code is executed before creating the other forms (in this case only the main form), and the splash screen is then removed before running the application. These operations take place within a `try/finally` block. Here is the source code of the main block of the project file for the Splash2 example:

```
var
  SplashAbout: TAboutBox;

begin
  Application.Initialize;

  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  try
    SplashAbout.MakeSplash;
    // standard code...
    Application.CreateForm(TForm1, Form1);
    // get rid of the splash form
    SplashAbout.Close;
  finally
    SplashAbout.Free;
  end;

  Application.Run;
end.
```

This approach makes sense only if your application's main form takes a while to create, to execute its startup code (as in this case), or to open database tables. Notice that the splash screen is the first form created, but because the program doesn't use the `CreateForm` method of the `Application` object, this doesn't become the main form of the application. In this case, in fact, closing the splash screen would terminate the program!

An alternative approach is to keep the splash form on the screen a little longer and use a timer to get rid of it after a while. I've implemented this second technique in the Splash2 example. This example also uses a different approach for creating the splash form: instead of creating it in the project source code, it creates the form at the very beginning of the `FormCreate` method of the main form.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  SplashAbout: TAboutBox;
```

```
begin  
  // create and show the splash form  
  SplashAbout := TAboutBox.Create (Application);  
  SplashAbout.MakeSplash;  
  // standard code...  
  for I := 1 to 30000 do  
    if IsPrime (I) then  
      ListBox1.Items.Add (IntToStr (I));  
  // get rid of the splash form, after a while  
  SplashAbout.Timer1.Enabled := True;  
end;
```

The timer is enabled just before terminating the method. After its interval has elapsed (in the example, 3 seconds) the `OnTimer` event is activated, and the splash form handles it by closing and destroying itself:

```
procedure TAboutBox.Timer1Timer(Sender: TObject);  
begin  
  Close;  
  Release;  
end;
```

## NOTE

The `Release` method of a form is similar to the `Free` method of objects, only the destruction of the form is delayed until all event handlers have completed execution. Using `Free` inside a form might cause an access violation, as the internal code, which fired the event handler, might refer again to the form object.

There is one more thing to fix. The main form will be displayed later and in front of the splash form, unless you make this a top-most form. For this reason I've added one line to the `MakeSplash` method of the About box in the `Splash2` example:

```
FormStyle := fsStayOnTop;
```

## What's Next?

In this chapter we've explored some important form properties. Now you know how to handle the size and position of a form, how to resize it, and how to get mouse input and paint over it. You know more about dialog boxes, modal forms, predefined dialogs, splash screens, and many other techniques, including the funny effect of alpha blending. Understanding the details of working with forms is critical to a proper use of Delphi, particularly for building complex applications (unless, of course, you're building services or Web applications with no user interface).

In the next chapter we'll continue by exploring the overall structure of a Delphi application, with coverage of the role of two global objects, `Application` and `Screen`. I'll also discuss MDI development as you learn some more advanced features of forms, such as visual form inheritance. I'll also discuss frames, visual component containers similar to forms.

In this chapter, I've also provided a short introduction to direct painting and to the use of the `TCanvas` class. More about graphics in Delphi forms can also be found in the bonus chapter "Graphics in Delphi" on the companion CD.

# The Architecture of Delphi Applications

- The *Application* and *Screen* global objects
- Messages and multitasking in Windows
- Finding the previous instance of an application
- MDI applications
- Visual form inheritance
- Frames
- Base forms and interfaces

**A**lthough together we've built Delphi applications since the beginning of the book, we've never really focused on the structure and the architecture of an application built with Delphi's class library. For example, there hasn't been much coverage about the global `Application` object, about techniques for keeping tracks of the forms we've created, about the flow of messages in the system, and other such elements.

In the last chapter you saw how to create applications with multiple forms and dialog boxes, but we haven't discussed how these forms can be related one to the other, how can you share similar features of forms, and how you can operate on multiple similar forms in a coherent way. All of this is the ambitious goal of this chapter, which covers both basic and advanced techniques, including visual form inheritance, the use of frames, and MDI development, but also the use of interfaces for building complex hierarchies of form classes.

## The *Application* Object

I've already mentioned the `Application` global object on multiple occasions, but as in this chapter we are focusing on the structure of Delphi applications, it is time to delve into some more details of this global object and its corresponding class. `Application` is a global object of the `TApplication` class, defined in the `Forms` unit and created in the `Controls` unit.

The `TApplication` class is a component, but you cannot use it at design time. Some of its properties can be directly set in the `Application` page of the `Project Options` dialog box; others must be assigned in code.

To handle its events, instead, Delphi includes a handy `ApplicationEvents` component. Besides allowing you to assign handlers at design time, the advantage of this component is that it allows for multiple handlers. If you simply place two instances of the `ApplicationEvents` component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple `ApplicationEvents` components can chain the handlers.

Some of these application-wide events, including `OnActivate`, `OnDeactivate`, `OnMinimize`, and `OnRestore`, allow you to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as in `OnActionExecute`, `OnActionUpdate`, `OnHelp`, `OnHint`, `OnShortCut`, and `OnShowHint`. Finally, there is the `OnException` global exception handler we used in Chapter 3, the `OnIdle` event used for background computing, and the `OnMessage` event, which fires whenever a message is posted to any of the windows or windowed controls of the application.

Although its class inherits directly from `TComponent`, the `Application` object has a window associated with it. The application window is hidden from sight but appears on the `Taskbar`. This is why Delphi names the window *Form1* and the corresponding `Taskbar` icon *Project1*.

The window related to the `Application` object—the application window—serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this invisible owner window, for example, is fundamental when the application is activated. In fact, when the windows of your program are behind those of other programs, clicking one window in your application will bring all of that application’s windows to the front. In other words, the unseen application window is used to connect the various forms of the application. Actually the application window is not *hidden*, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.

**TIP**

In Windows, the Minimize and Maximize operations are associated by default with system sounds and a visual animated effect. Applications built with Delphi (starting with version 5) produce the sound and display the visual effect by default.

When you create a new, blank application, Delphi generates a code for the project file, which includes the following:

```
begin  
  Application.Initialize;  
  Application.CreateForm(TForm1, Form1);  
  Application.Run;  
end.
```

As you can see in this standard code, the `Application` object can create forms, setting the first one as the `MainForm` (one of the `Application` properties) and closing the entire application when this main form is destroyed. Moreover, it contains the Windows message loop (started by the `Run` method) that delivers the system messages to the proper windows of the application. A message loop is required by any Windows application, but you don’t need to write one in Delphi because the `Application` object provides a default one.

If this is the main role of the `Application` object, it manages few other interesting areas as well:

- Hints (discussed at the end of Chapter 7)
- The help system, which in Delphi 6 includes the ability to define the type of help viewer (something not covered in detail in this book)
- Application activation, minimize, and restore
- A global exceptions handler, as discussed in Chapter 3 in the `ErrorLog` example
- General application information, including the `MainForm`, executable file name and path (`ExeName`), the `Icon`, and the `Title` displayed in the Windows taskbar and when you scan the running applications with the `Alt+Tab` keys



**TIP**

To avoid a discrepancy between the two titles, you can change the application's title at design time. As an alternative, at run time, you can copy the form's caption to the title of the application with this code: `Application.Title := Form1.Caption`.

In most applications, you don't care about the application window, apart from setting its `Title` and icon and handling some of its events. There are some simple operations you can do anyway. Setting the `ShowMainForm` property to `False` in the project source code indicates that the main form should not be displayed at startup. Inside a program, instead, you can use the `MainForm` property of the `Application` object to access the main form, which is the first form created in the program.

## Displaying the Application Window

There is no better proof that a window indeed exists for the `Application` object than to display it. Actually, we don't need to show it—we just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. We can perform these operations by using Windows API functions on the window indicated by the `Handle` property of the `Application` object:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    OldStyle: Integer;  
begin  
    // add border and caption to the app window  
    OldStyle := GetWindowLong (Application.Handle, gw1_Style);  
    SetWindowLong (Application.Handle, gw1_Style,  
        OldStyle or ws_ThickFrame or ws_Caption);  
    // set the size of the app window  
    SetWindowPos (Application.Handle, 0, 0, 0, 200, 100,  
        swp_NoMove or swp_NoZOrder);  
end;
```

The two `GetWindowLong` and `SetWindowLong` API functions are used to access the system information related to the window. In this case, we are using the `gw1_Style` parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. The code above gets the current styles and adds (using an `or` statement) a standard border and a caption to the form. As we'll see later in this chapter, you seldom need to use these low-level API functions in Delphi, because there are properties of the `TForm` class that have the same effect. We need this code here because the application window is not a form.

Executing this code displays the project window, as you can see in Figure 10.1. Although there's no need to implement something like this in your own programs, running this program will reveal the relation between the application window and the main window of a Delphi program. This is a very important starting point if you want to understand the internal structure of Delphi applications.

**FIGURE 10.1:**

The hidden application window revealed by the ShowApp program



## The Application System Menu

Unless you write a very odd program like the example we've just looked at, users will only see the application window in the Taskbar. There, they can activate the window's system menu by right-clicking it. As I mentioned in the SysMenu example in Chapter 6, when discussing the system menu, an application's menu is not the same as that of the main form. In that example, I added custom items to the system menu of the main form. Now in the SysMenu2 example, I want to customize the system menu of the application window in the Taskbar.

First we have to add the new items to the system menu of the application window when the program starts. Here is the updated code of the FormCreate method:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    // add a separator and a menu item to the system menu  
    AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');  
    AppendMenu (GetSystemMenu (Handle, FALSE), MF_STRING, idSysAbout,  
                '&About...');  
    // add the same items to the application system menu  
    AppendMenu (GetSystemMenu (Application.Handle, FALSE), MF_SEPARATOR, 0, '');  
    AppendMenu (GetSystemMenu (Application.Handle, FALSE), MF_STRING, idSysAbout,  
                '&About...');  
end;
```

The first part of the code adds the new separator and item to the system menu of the main form. The other two calls add the same two items to the application's system menu, simply by referring to `Application.Handle`. This is enough to display the updated system menu, as you can see by running this program. The next step is to handle the selection of the new menu item.

To handle form messages, we can simply write new event handlers or message-handling methods. We cannot do the same with the application window, simply because inheriting from the `TApplication` class is quite a complex issue. Most of the time we can just handle

the `OnMessage` event of this class, which is activated for every message the application retrieves from the message queue.

To handle the `OnMessage` event of the global `Application` object, simply add an `ApplicationEvents` component to the main form, and define a handler for the `OnMessage` event of this component. In this case, we only need to handle the `wm_SysCommand` message, and we only need to do that if the `wParam` parameter indicates that the user has selected the menu item we've just added, `idSysAbout`:

```
procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
var Handled: Boolean);
begin
  if (Msg.Message = wm_SysCommand) and (Msg.wParam = idSysAbout) then
    begin
      ShowMessage ('Mastering Delphi: SysMenu2 example');
      Handled := True;
    end;
end;
```

This method is very similar to the one used to handle the corresponding system menu item of the main form:

```
procedure WMSysCommand (var Msg: TWMSysCommand);
  message wm_SysCommand;
  ...
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  // handle a specific command
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu2 example');
  inherited;
end;
```

## Activating Applications and Forms

To show how the activation of forms and applications works, I've written a simple, self-explanatory example, available on the companion CD, called `ActivApp`. This example has two forms. Each form has a `Label` component (`LabelForm`) used to display the status of the form. The program uses text and color for this, as the handlers of the `OnActivate` and `OnDeactivate` events of the first form demonstrate:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Active';
  LabelForm.Color := clRed;
end;

procedure TForm1.FormDeactivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Not Active';
```

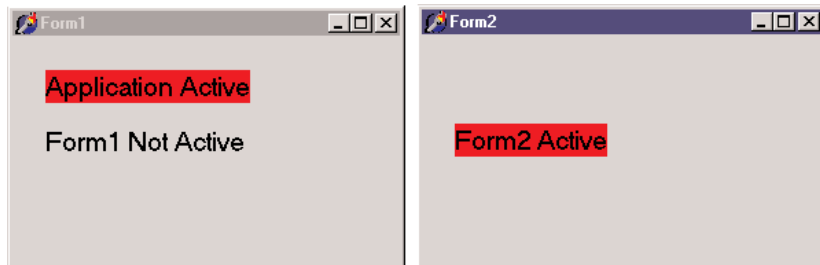
```
LabelForm.Color := clBtnFace;  
end;
```

The second form has a similar label and similar code. The main form also displays the status of the entire application. It uses an `ApplicationEvents` component to handle the `OnActivate` and `OnDeactivate` events of the `Application` object. These two event handlers are similar to the two listed previously, with the only difference being that they modify the text and color of a second label of the form.

If you try running this program, you'll see whether this application is the active one and, if so, which of its forms is the active one. By looking at the output (see Figure 10.2) and listening for the beep, you can understand how each of the activation events is triggered by Delphi. Run this program and play with it for a while to understand how it works. We'll get back to other events related to the activation of forms in a while.

**FIGURE 10.2:**

The `ActivApp` example shows whether the application is active and which of the application's forms is active.



## Tracking Forms with the *Screen* Object

We have already explored some of the properties and events of the `Application` object. Other interesting global information about an application is available through the `Screen` object, whose base class is `TScreen`. This object holds information about the system display (the screen size and the screen fonts) and also about the current set of forms in a running application. For example, you can display the screen size and the list of fonts by writing:

```
Label1.Caption := IntToStr (Screen.Width) + 'x' + IntToStr (Screen.Height);  
ListBox1.Items := Screen.Fonts;
```

`TScreen` also reports the number and resolution of monitors in a multimonitor system. What I want to focus on now, however, is the list of forms held by the `Forms` property of the `Screen` object, the top-most form indicated by the `ActiveForm` property, and the related `OnActiveFormChange` event. Note that the forms the `Screen` object references are the forms of the application and not those of the system.

These features are demonstrated by the `Screen` example on the CD, which maintains a list of the current forms in a list box. This list must be updated each time a new form is created,

an existing form is destroyed, or the active form of the program changes. To see how this works, you can create secondary forms by clicking the button labeled New:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
var
    NewForm: TSecondForm;
begin
    // create a new form, set its caption, and run it
    NewForm := TSecondForm.Create (Self);
    Inc (nForms);
    NewForm.Caption := 'Second ' + IntToStr (nForms);
    NewForm.Show;
end;
```

One of the key portions of the program is the OnCreate event handler of the form, which fills the list a first time and then connects a handler to the OnActiveFormChange event:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    FillFormsList (Self);
    // set the secondary forms counter to 0
    nForms := 0;
    // set an event handler on the screen object
    Screen.OnActiveFormChange := FillFormsList;
end;
```

The code used to fill the Forms list box is inside a second procedure, FillFormsList, which is also installed as an event handler for the OnActiveFormChange event of the Screen object:

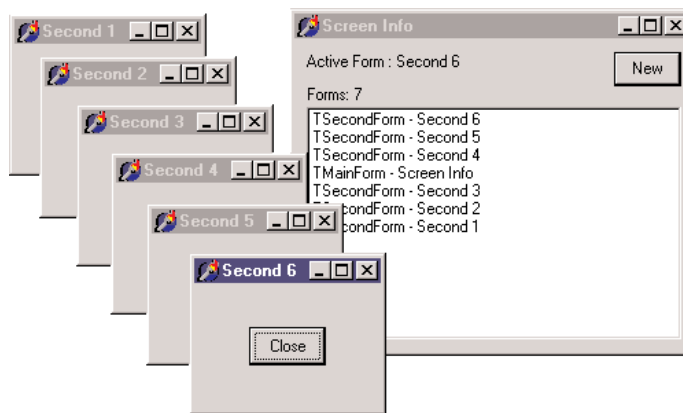
```
procedure TMainForm.FillFormsList (Sender: TObject);
var
    I: Integer;
begin
    // skip code in destruction phase
    if Assigned (FormsListBox) then
        begin
            FormsLabel.Caption := 'Forms: ' + IntToStr (Screen.FormCount);
            FormsListBox.Clear;
            // write class name and form title to the list box
            for I := 0 to Screen.FormCount - 1 do
                FormsListBox.Items.Add (Screen.Forms[I].ClassName + ' - ' +
                    Screen.Forms[I].Caption);
            ActiveLabel.Caption := 'Active Form : ' + Screen.ActiveForm.Caption;
        end;
    end;
```

**WARNING** It is very important not to execute this code while the main form is being destroyed. As an alternative to testing for the listbox not to be set to `nil`, you could as well test the form's `ComponentState` for the `csDestroying` flag. Another approach would be to remove the `OnActiveFormChange` event handler before exiting the application; that is, handle the `OnClose` event of the main form and assign `nil` to `Screen.OnActiveFormChange`.

The `FillFormsList` method fills the list box and sets a value for the two labels above it to show the number of forms and the name of the active one. When you click the `New` button, the program creates an instance of the secondary form, gives it a new title, and displays it. The `Forms` list box is updated automatically because of the handler we have installed for the `OnActiveFormChange` event. Figure 10.3 shows the output of this program when some secondary windows have been created.

**FIGURE 10.3:**

The output of the `Screen` example with some secondary forms



**TIP** The program always updates the text of the `ActiveLabel` above the list box to show the currently active form, which is always the same as the first one in the list box.

The secondary forms each have a `Close` button you can click to remove them. The program handles the `OnClose` event, setting the `Action` parameter to `caFree`, so that the form is actually destroyed when it is closed. This code closes the form, but it doesn't update the list of the windows properly. The system moves the focus to another window first, firing the event that updates the list, and destroys the old form only after this operation.

The first idea I had to update the windows list properly is to introduce a delay, posting a user-defined `Windows` message. Because the posted message is queued and not handled

immediately, if we send it at the last possible moment of life of the secondary form, the main form will receive it when the other form is destroyed.

The trick is to post the message in the `OnDestroy` event handler of the secondary form. To accomplish this, we need to refer to the `MainForm` object, by adding a `uses` statement in the implementation portion of this unit. I've posted a `wm_User` message, which is handled by a specific message method of the main form, as shown here:

```
public  
  procedure ChildClosed (var Message: TMessage);  
    message wm_User;
```

Here is the code for this method:

```
procedure TMainForm.ChildClosed (var Message: TMessage);  
begin  
  FillFormsList (Self);  
end;
```

The problem here is that if you close the main window before closing the secondary forms, the main form exists, but its code cannot be executed anymore. To avoid another system error (an Access Violation Fault), you need to post the message only if the main form is not closing. But how do you know that? One way is to add a flag to the `TMainForm` class and change its value when the main form is closing, so that you can test the flag from the code of the secondary window.

This is a good solution—so good that the VCL already provides something similar. There is a barely documented `ComponentState` property. It is a Pascal set that includes (among other flags) a `csDestroying` flag, which is set when the form is closing. Therefore, we can write the following code:

```
procedure TSecondForm.FormDestroy(Sender: TObject);  
begin  
  if not (csDestroying in MainForm.ComponentState) then  
    PostMessage (MainForm.Handle, wm_User, 0, 0);  
end;
```

With this code, the list box always lists all of the forms in the application. Note that you need to disable the automatic creation of the secondary form by using the `Forms` page of the `Project Options` dialog box.

After giving it some thought, however, I found an alternative and much more Delphi-oriented solution. Every time a component is destroyed, it tells its owner about the event by calling the `Notification` method defined in the `TComponent` class. Because the secondary forms are owned by the main one, as specified in the code of the `NewButtonClick` method, we can override this method and simplify the code. In the form class, simply write

```
protected  
  procedure Notification(AComponent: TComponent;  
    Operation: TOperation); override;
```

Here is the code of the method:

```
procedure TMainForm.Notification(AComponent: TComponent;
    Operation: TOperation);
begin
    inherited Notification(AComponent, Operation);
    if (Operation = opRemove) and Showing and (AComponent is TForm) then
        FillFormsList;
end;
```

You'll find the complete code of this version in the Screen2 directory on the CD.

## NOTE

In case the secondary forms were not owned by the main one, we could have used the `FreeNotification` method to get the secondary form to notify the main form when they are destroyed. `FreeNotification` receives as parameter the component to notify when the current component is destroyed. The effect is a call to the `Notification` method coming from a component other than the owned ones. `FreeNotification` is generally used by component writers to safely connect components on different forms or data modules.

The last feature I've added to both versions of the program is a simple one. When you click an item in the list box, the corresponding form is activated, using the `BringToFront` method:

```
procedure TMainForm.FormsListBoxClick(Sender: TObject);
begin
    Screen.Forms [FormsListBox.ItemIndex].BringToFront;
end;
```

Nice—well, almost nice. If you click the list box of an inactive form, the main form is activated first, and the list box is rearranged, so you might end up selecting a different form than you were expecting. If you experiment with the program, you'll soon realize what I mean. This minor glitch in the program is an example of the risks you face when you dynamically update some information and let the user work on it at the same time.

## Events, Messages, and Multitasking in Windows

To understand how Windows applications work internally, we need to spend a minute discussing how multitasking is supported in this environment. We also need to understand the role of timers (and the `Timer` component) and of background (or *idle*) computing.

In short, we need to delve deeper into the event-driven structure of Windows and its multitasking support. Because this is a book about *Delphi* programming, I won't discuss this topic in detail, but I will provide an overview for readers who have limited experience with Windows API programming.



## Event-Driven Programming

The basic idea behind event-driven programming is that specific events determine the control flow of the application. A program spends most of its time waiting for these events and provides code to respond to them. For example, when a user clicks one of the mouse buttons, an event occurs. A message describing this event is sent to the window currently under the mouse cursor. The program code that responds to events for that window will receive the event, process it, and respond accordingly. When the program has finished responding to the event, it returns to a waiting or “idle” state.

As this explanation shows, events are serialized; each event is handled only after the previous one is completed. When an application is executing event-handling code (that is, when it is not waiting for an event), other events for that application have to wait in a message queue reserved for that application (unless the application uses multiple threads). When an application has responded to a message and returned to a waiting state, it becomes the last in the list of programs waiting to handle additional messages. In every version of Win32 (9x, NT, Me, and 2000), after a fixed amount of time has elapsed, the system interrupts the current application and immediately gives control to the next one in the list. The first program is resumed only after each application has had a turn. This is called preemptive multitasking.

So, an application performing a time-consuming operation in an event handler doesn't prevent the system from working properly, but is generally unable even to repaint its own windows properly, with a very nasty effect. If you've never experienced this problem, try for yourself: Write a time-consuming loop executed when a button is pressed, and try to move the form or move another window on top of it. The effect is really annoying. Now try adding the call `Application.ProcessMessages` within the loop, and you'll see that the operation becomes much slower, but the form will be immediately refreshed.

If an application has responded to its events and is waiting for its turn to process messages, it has no chance to regain control until it receives another message (unless it uses multi-threading). This is a reason to use timers, a system component that will send a message to your application every time a time interval elapses.

One final note—when you think about events, remember that input events (using the mouse or the keyboard) account for only a small percentage of the total message flow in a Windows application. Most of the messages are the system's internal messages or messages exchanged between different controls and windows. Even a familiar input operation such as clicking a mouse button can result in a huge number of messages, most of which are internal Windows messages. You can test this yourself by using the WinSight utility included in Delphi. In WinSight, choose to view the Message Trace, and select the messages for all of the windows. Select Start, and then perform some normal operations with the mouse. You'll see hundreds of messages in a few seconds.

## Windows Message Delivery

Before looking at some real examples, we need to consider another key element of message handling. Windows has two different ways to send a message to a window:

- The `PostMessage` API function is used to place a message in the application's message queue. The message will be handled only when the application has a chance to access its message queue (that is, when it receives control from the system), and only after earlier messages have been processed. This is an asynchronous call, since you do not know when the message will actually be received.
- The `SendMessage` API function is used to execute message-handler code immediately. `SendMessage` bypasses the application's message queue and sends the message directly to a target window or control. This is a synchronous call. This function even has a return value, which is passed back by the message-handling code. Calling `SendMessage` is no different than directly calling another method or function of the program.

The difference between these two ways of sending messages is similar to that between mailing a letter, which will reach its destination sooner or later, and sending a fax, which goes immediately to the recipient. Although you will rarely need to use these low-level functions in Delphi, this description should help you determine which one to use if you do need to write this type of code.

## Background Processing and Multitasking

Suppose that you need to implement a time-consuming algorithm. If you write the algorithm as a response to an event, your application will be stopped completely during all the time it takes to process that algorithm. To let the user know that something is being processed, you can display the hourglass cursor, but this is not a user-friendly solution. Win32 allows other programs to continue their execution, but the program in question will freeze; it won't even update its own user interface if a repaint is requested. In fact, while the algorithm is executing, the application won't be able to receive and process any other messages, including the paint messages.

The simplest solution to this problem is to call the `ProcessMessages` method of the `Application` object many times within the algorithm, usually inside an internal loop. This call stops the execution, allows the program to receive and handle a message, and then resumes execution. The problem with this approach, however, is that while the program is paused to accept messages, the user is free to do any operation and might again click the button or press the keystrokes that started the algorithm. To fix this, you can disable the buttons and commands you don't want the user to select, and you can display the hourglass cursor (which technically doesn't prevent a mouse click event, but it does suggest that the user

should wait before doing any other operation). An alternative solution is to split the algorithm into smaller pieces and execute each of them in turn, letting the application respond to pending messages in between processing the pieces. We can use a timer to let the system notify us once a time interval has elapsed. Although you can use timers to implement some form of background computing, this is far from a good solution. A slightly better technique would be to execute each step of the program when the `Application` object receives the `OnIdle` event.

The difference between calling `ProcessMessages` and using the `OnIdle` events is that by calling `ProcessMessages`, you will give your code more processing time than with the `OnIdle` approach. Calling `ProcessMessages` is a way to let the system perform other operations while your program is computing; using the `OnIdle` event is a way to let your application perform background tasks when it doesn't have pending requests from the user.

**NOTE**

All these techniques for background computing were necessary in 16-bit Windows days. In Win32, you should generally use secondary threads to perform lengthy or background operations.

## Checking for a Previous Instance of an Application

One form of multitasking is the execution of two or more instances of the same application. Any application can generally be executed by a user in more than one instance, and it needs to be able to check for a previous instance already running, in order to disable this default behavior and allow for one instance at most. This section demonstrates several ways of implementing such a check, allowing me to discuss some interesting Windows programming techniques.

### Looking for a Copy of the Main Window

To find a copy of the main window of a previous instance, use the `FindWindow` API function and pass it the name of the window class (the name used to register the form's window type, or `WNDCLASS`, in the system) and the caption of the window for which you are looking. In a Delphi application, the name of the `WNDCLASS` window class is the same as the Object Pascal name for the form's class (for example, `TForm1`). The result of the `FindWindow` function is either a handle to the window or zero (if no matching window was found).

The main code of your Delphi application should be written so that it will execute only if the `FindWindow` result is zero:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', nil);
  if Hwnd = 0 then
```

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end
else
  SetForegroundWindow (Hwnd)
end.
```

To activate the window of the previous instance of the application, you can use the `SetForegroundWindow` function, which works for windows owned by other processes. This call produces its effect only if the window passed as parameter hasn't been minimized. When the main form of a Delphi application is minimized, in fact, it is hidden, and for this reason the activation code has no effect.

Unfortunately, if you run a program that uses the `FindWindow` call just shown from within the Delphi IDE, a window with that caption and class may already exist: the design-time form. Thus, the program won't start even once. However, it will run if you close the form and its corresponding source code file (closing only the form, in fact, simply hides the window), or if you close the project and run the program from the Windows Explorer.

## Using a Mutex

A completely different approach is to use a *mutex*, or mutual exclusion object. This is a typical Win32 approach, commonly used for synchronizing threads, as we'll see later in this chapter. Here we are going to use a mutex for synchronizing two different applications, or (to be more precise) two instances of the same application.

Once an application has created a mutex with a given name, it can test whether this object is already owned by another application, calling the `WaitForSingleObject` Windows API function. If the mutex has no owner, the application calling this function becomes the owner. If the mutex is already owned, the application waits until the time-out (the second parameter of the function) elapses. It then returns an error code.

To implement this technique, you can use the following project source code, which you'll find in the `OneCopy` example:

```
var
  hMutex: THandle;
begin
  HMutex := CreateMutex (nil, False, 'OneCopyMutex');
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
```

```

    end;
end.

```

If you run this example twice, you'll see that it creates a new, temporary copy of the application (the icon appears in the Taskbar) and then destroys it when the time-out elapses. This approach is certainly more robust than the previous one, but it lacks a feature: how do we enable the existing instance of the application? We still need to find its form, but we can use a better approach.

## Searching the Window List

When you want to search for a specific main window in the system, you can use the EnumWindows API functions. Enumeration functions are quite peculiar in Windows, because they usually require another function as a parameter. These enumeration functions require a pointer to a function (often described as a *callback* function) as parameter. The idea is that this function is applied to each element of the list (in this case, the list of main windows), until the list ends or the function returns False. Here is the enumeration function from the OneCopy example:

```

function EnumWndProc (hwnd: THandle;
    Param: Cardinal): Bool; stdcall;
var
    ClassName, WinModuleName: string;
    WinInstance: THandle;
begin
    Result := True;
    SetLength (ClassName, 100);
    GetClassName (hwnd, PChar (ClassName), Length (ClassName));
    ClassName := PChar (ClassName);
    if ClassName = TForm1.ClassName then
    begin
        // get the module name of the target window
        SetLength (WinModuleName, 200);
        WinInstance := GetWindowLong (hwnd, GWL_HINSTANCE);
        GetModuleFileName (WinInstance,
            PChar (WinModuleName), Length (WinModuleName));
        WinModuleName := PChar(WinModuleName); // adjust length
        // compare module names
        if WinModuleName = ModuleName then
        begin
            FoundWnd := Hwnd;
            Result := False; // stop enumeration
        end;
    end;
end;

```

This function, called for each nonchild window of the system, checks the name of each window's class, looking for the name of the TForm1 class. When it finds a window with this string in its class name, it uses `GetModuleFileName` to extract the name of the executable file of the application that owns the matching form. If the module name matches that of the current program (which was extracted previously with similar code), you can be quite sure that you have found a previous instance of the same program. Here is how you can call the enumerated function:

```

var
  FoundWnd: THandle;
  ModuleName: string;
begin
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
    ...
  else
    begin
      // get the current module name
      SetLength (ModuleName, 200);
      GetModuleFileName (HInstance, PChar (ModuleName), Length (ModuleName));
      ModuleName := PChar (ModuleName); // adjust length
      // find window of previous instance
      EnumWindows (@EnumWndProc, 0);
    end;
  end;
end;

```

## Handling User-Defined Window Messages

I've mentioned earlier that the `SetForegroundWindow` call doesn't work if the main form of the program has been minimized. Now we can solve this problem. You can ask the form of another application—the previous instance of the same program in this case—to restore its main form by sending it a user-defined window message. You can then test whether the form is minimized and post a new user-defined message to the old window. Here is the code; in the `OneCopy` program, it follows the last fragment shown in the preceding section:

```

if FoundWnd <> 0 then
begin
  // show the window, eventually
  if not IsWindowVisible (FoundWnd) then
    PostMessage (FoundWnd, wm_User, 0, 0);
    SetForegroundWindow (FoundWnd);
end;
end;

```

Again, the `PostMessage` API function sends a message to the message queue of the application that owns the destination window. In the code of the form, you can add a special function to handle this message:

```

public
  procedure WMUser (var msg: TMessage);
  message wm_User;

```

Now you can write the code of this method, which is simple:

```
procedure TForm1.WMUser (var msg: TMessage);  
begin  
    Application.Restore;  
end;
```

## Creating MDI Applications

A common approach for the structure of an application is MDI (Multiple Document Interface). An MDI application is made up of several forms that appear inside a single main form. If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application. But with your favorite word processor, you can probably open several different documents, each in its own child window, because they are MDI applications. All these document windows are usually held by a *frame*, or *application*, window.

### NOTE

Microsoft is departing more and more from the MDI model stressed in Windows 3 days. Starting with Resource Explorer in Windows 95 and even more with Office 2000, Microsoft tends to use a specific main window for every document, the classic SDI (Single Document Interface) approach. In any case, MDI isn't dead and can sometimes be a useful structure.

## MDI in Windows: A Technical Overview

The MDI structure gives programmers several benefits automatically. For example, Windows handles a list of the child windows in one of the pull-down menus of an MDI application, and there are specific Delphi methods that activate the corresponding MDI functionality, to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows:

- The main window of the application acts as a frame or a container.
- A special window, known as the *MDI client*, covers the whole client area of the frame window. This MDI client is one of the Windows predefined controls, just like an edit box or a list box. The MDI client window lacks any specific user-interface element, but it is visible. In fact, you can change the standard system color of the MDI work area (called the Application Background) in the Appearance page of the Display Properties dialog box in Windows.
- There are multiple child windows, of the same kind or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the frame window.

## Frame and Child Windows in Delphi

Delphi makes the development of MDI applications easy, even without using the MDI Application template available in Delphi (see the Applications page of the File > New dialog box). You only need to build at least two forms, one with the `FormStyle` property set to `fsMDIForm` and the other with the same property set to `fsMDIChild`. Set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at startup, and you need to provide a way to create one or more child windows. This can be done by adding a menu with a New menu item and writing the following code:

```
var
    ChildForm: TChildForm;
begin
    ChildForm := TChildForm.Create (Application);
    ChildForm.Show;
```

Another important feature is to add a “Window” pull-down menu and use it as the value of the `WindowMenu` property of the form. This pull-down menu will automatically list all the available child windows. Of course, you can choose any other name for the pull-down menu, but `Window` is the standard.

To make this program work properly, we can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
    ChildForm: TChildForm;
begin
    WindowMenu := Window1;
    Inc (Counter);
    ChildForm := TChildForm.Create (Self);
    ChildForm.Caption := ChildForm.Caption + ' ' + IntToStr (Counter);
    ChildForm.Show;
end;
```

You can also open child windows, minimize or maximize each of them, close them, and use the `Window` pull-down menu to navigate among them. Now suppose that we want to close some of these child windows, to unclutter the client area of our program. Click the Close boxes of some of the child windows and they are minimized! What is happening here? Remember that when you close a window, you generally hide it from view. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, hiding them won't work, because the MDI `Window` menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi minimizes the MDI child windows when you



try to close them. To solve this problem, we need to delete the child windows when they are closed, setting the `Action` reference parameter of the `OnClose` event to `caFree`.

## Building a Complete Window Menu

Our first task is to define a better menu structure for the example. Typically the `Window` pull-down menu has at least three items, titled `Cascade`, `Tile`, and `Arrange Icons`. To handle the menu commands, we can use some of the predefined methods of `TForm` that can be used only for `MDI` frames:

- The `Cascade` method cascades the open `MDI` child windows. The windows overlap each other. Iconized child windows are also arranged (see `ArrangeIcons` below).
- The `Tile` method tiles the open `MDI` child windows; the child forms are arranged so that they do not overlap. The default behavior is horizontal tiling, although if you have several child windows, they will be arranged in several columns. This default can be changed by using the `TileMode` property (either `tbHorizontal` or `tbVertical`).
- The `ArrangeIcons` procedure arranges all the iconized child windows. Open forms are not moved.

As a better alternative to calling these methods, you can place an `ActionList` in the form and add to it a series of predefined `MDI` actions. The related classes are `TWindowArrange`, `TWindowCascade`, `TWindowClose`, `TWindowTileHorizontal`, `TWindowTileVertical`, and `TWindowMinimizeAll`. The connected menu items will perform the corresponding actions and will be disabled if no child window is available. The `MdiDemo` example, which we'll look at next, demonstrates the use of the `MDI` actions, among other things.

There are also some other interesting methods and properties related strictly to `MDI` in Delphi:

- `ActiveMDIChild` is a run-time and read-only property of the `MDI` frame form, and it holds the active child window. The user can change this value by selecting a new child window, or the program can change it using the `Next` and `Previous` procedures, which activate the child window following or preceding the currently active one.
- The `ClientHandle` property holds the `Windows` handle of the `MDI` client window, which covers the client area of the main form.
- The `MDIChildren` property is an array of child windows. You can use this and the `MDIChildCount` property to cycle among all of the child windows. This can be useful for finding a particular child window or to operate on each of them.

**NOTE**

Note that the internal order of the child windows is the reverse order of activation. This means that the last child window selected is the active window (the first in the internal list), the second-to-last child window selected is the second, and the first child window selected is the last. This order determines how the windows are arranged on the screen. The first window in the list is the one above all others, while the last window is below all others, and probably hidden away. You can imagine an axis (the z axis) coming out of the screen toward you. The active window has a higher value for the z coordinate and, thus, covers other windows. For this reason, the Windows ordering schema is known as the *z-order*.

## The MdiDemo Example

I've built a first example to demonstrate most of the features of a simple MDI application. `MdiDemo` is actually a full-blown MDI text editor, because each child window hosts a `Memo` component and can open and save text files. The child form has a `Modified` property used to indicate whether the text of the memo has changed (it is set to `True` in the handler of the memo's `OnChange` event). `Modified` is set to `False` in the `Save` and `Load` custom methods and checked when the form is closed (prompting to save the file).

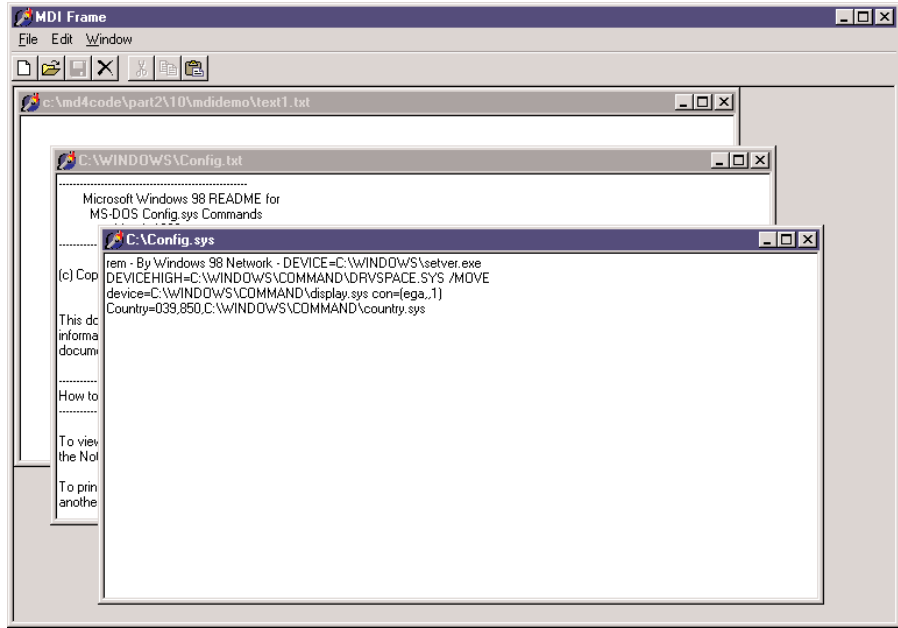
As I've already mentioned, the main form of this example is based on an `ActionList` component. The actions are available through some menu items and a toolbar, as you can see in Figure 10.4. You can see the details of the `ActionList` in the source code of the example. Next, I want to focus on the code of the custom actions. Once more, this example demonstrates that using actions makes it very simple to modify the user interface of the program, without writing any extra code. In fact, there is no code directly tied to the user interface.

One of the simplest actions is the `ActionFont` object, which has both an `OnExecute` handler, which uses a `FontDialog` component, and an `OnUpdate` handler, which disables the action (and hence the associated menu item and toolbar button) when there are no child forms:

```
procedure TMainForm.ActionFontExecute(Sender: TObject);  
begin  
    if FontDialog1.Execute then  
        (ActiveMDIChild as TChildForm).Memo1.Font := FontDialog1.Font;  
end;  
  
procedure TMainForm.ActionFontUpdate(Sender: TObject);  
begin  
    ActionFont.Enabled := MDIChildCount > 0;  
end;
```

**FIGURE 10.4:**

The MdiDemo program uses a series of predefined Delphi actions connected to a menu and a toolbar.



The action named New creates the child form and sets a default filename. The Open action calls the ActionNewExecute method prior to loading the file:

```

procedure TMainForm.ActionNewExecute(Sender: TObject);
var
    ChildForm: TChildForm;
begin
    Inc (Counter);
    ChildForm := TChildForm.Create (Self);
    ChildForm.Caption :=
        LowerCase (ExtractFilePath (Application.Exename)) + 'text' +
        IntToStr (Counter) + '.txt';
    ChildForm.Show;
end;

procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
    if OpenDialog1.Execute then
        begin
            ActionNewExecute (Self);
            (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
        end;
end;

```

The actual file loading is performed by the Load method of the form. Likewise, the Save method of the child form is used by the Save and Save As actions. Notice the OnUpdate handler of the Save action, which enables the action only if the user has changed the text of the memo:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);  
begin  
    // suggest the current file name  
    SaveDialog1.FileName := ActiveMDIChild.Caption;  
    if SaveDialog1.Execute then  
        begin  
            // modify the file name and save  
            ActiveMDIChild.Caption := SaveDialog1.FileName;  
            (ActiveMDIChild as TChildForm).Save;  
        end;  
end;  
  
procedure TMainForm.ActionSaveUpdate(Sender: TObject);  
begin  
    ActionSave.Enabled := (MDIChildCount > 0) and  
        (ActiveMDIChild as TChildForm).Modified;  
end;  
  
procedure TMainForm.ActionSaveExecute(Sender: TObject);  
begin  
    (ActiveMDIChild as TChildForm).Save;  
end;
```

## MDI Applications with Different Child Windows

A common approach in complex MDI applications is to include child windows of different kinds (that is, based on different child forms). I will build a new example, called MdiMulti, to highlight some problems you may encounter with this approach. This example has two different types of child forms. The first type will host a circle drawn in the position of the last mouse click, while the second will contain a bouncing square. Another feature I'll add to the main form is a custom background obtained by painting a tiled image in it.

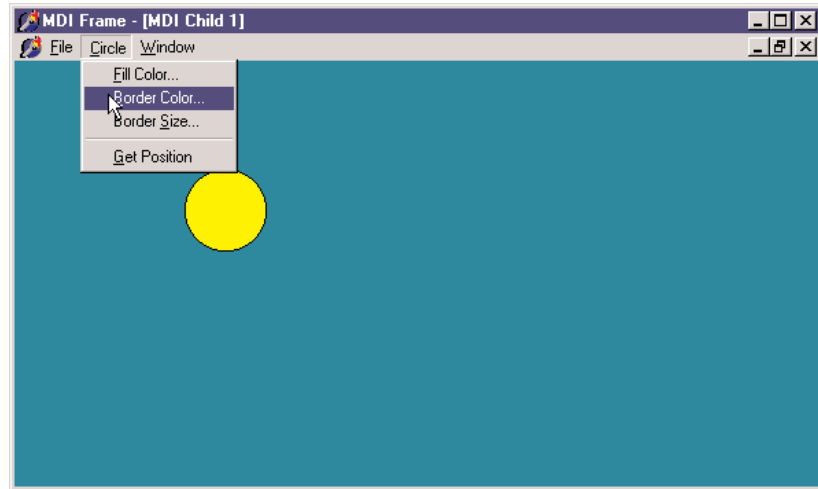
### Child Forms and Merging Menus

The first type of child form can display a circle in the position where the user clicked one of the mouse buttons. Figure 10.5 shows an example of the output of the MdiMulti program. The program includes a Circle menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. What is interesting here is that to program the child form, we do not need to consider the existence of other forms or of the

frame window. We simply write the code of the form, and that's all. The only special care required is for the menus of the two forms.

**FIGURE 10.5:**

The output of the MdiMulti example, with a child window that displays circles



If we prepare a main menu for the child form, it will replace the main menu of the frame window when the child form is activated. An MDI child window, in fact, cannot have a menu of its own. But the fact that a child window can't have any menus should not bother you, because this is the standard behavior of MDI applications. You can use the menu bar of the frame window to display the menus of the child window. Even better, you can merge the menu bar of the frame window and that of the child form. For example, in this program, the menu of the child form can be placed between the frame window's File and Window pull-down menus. You can accomplish this using the following GroupIndex values:

- File pull-down menu, main form: 1
- Window pull-down menu, main form: 3
- Circle pull-down menu, child form: 2

Using these settings for the menu group indexes, the menu bar of the frame window will have either two or three pull-down menus. At startup, the menu bar has two menus. As soon as you create a child window, there are three menus, and when the last child window is closed (destroyed), the Circle pull-down menu disappears. You should also spend some time testing this behavior by running the program.

The second type of child form shows a moving image. The square, a Shape component, moves around the client area of the form at fixed time intervals, using a Timer component,

and bounces on the edges of the form, changing its direction. This turning process is determined by a fairly complex algorithm, which we don't have space to examine. The main point of the example, instead, is to show you how menu merging behaves when you have an MDI frame with child forms of different types. (You can study the source code on the companion CD to see how it works.)

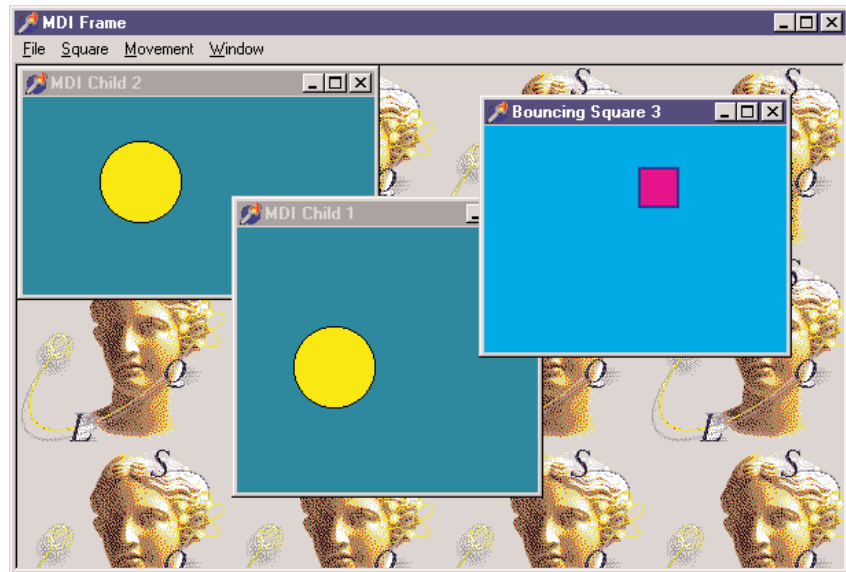
## The Main Form

Now we need to integrate the two child forms into an MDI application. The File pull-down menu here has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. The Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the original menu bar of the main form is reset. By using the proper menu group indexes, we let Delphi accomplish everything automatically, as you can see in Figure 10.6.

**FIGURE 10.6:**

The menu bar of the Mdi-Multi Demo4 application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of Figure 10.5.



I've added a few other menu items in the main form, to close every child window and show some statistics about them. The method related to the Count command scans the `MDIChildren` array property to count the number of child windows of each kind (using the RTTI operator `is`):

```
for I := 0 to MDIChildCount - 1 do
  if MDIChildren is TBounceChildForm then
    Inc (NBounce)
  else
    Inc (NCircle);
```

## Subclassing the MdiClient Window

Finally, the program includes support for a background-tiled image. The bitmap is taken from an `Image` component and should be painted on the form in the `wm_EraseBkgnd` Windows message's handler. The problem is that we cannot simply connect the code to the main form, as a separate window, the `MdiClient`, covers its surface.

We have no corresponding Delphi form for this window, so how can we handle its messages? We have to resort to a low-level Windows programming technique known as *subclassing*. (In spite of the name, this has little to do with OOP inheritance.) The basic idea is that we can replace the window procedure, which receives all the messages of the window, with a new one we provide. This can be done by calling the `SetWindowLong` API function and providing the memory address of the procedure, the function pointer.

### NOTE

A window procedure is a function receiving all the messages for a window. Every window must have a window procedure and can have only one. Even Delphi forms have a window procedure; although this is hidden in the system, it calls the `WndProc` virtual function, which you can use. But the VCL has a predefined handling of the messages, which are then forwarded to the message-handling methods of a form after some preprocessing. With all this support, you need to handle window procedures explicitly only when working with non-Delphi windows, as in this case.

Unless we have some reason to change the default behavior of this system window, we can simply store the original procedure and call it to obtain a default processing. The two function pointers referring to the two procedures (the old and the new one) are stored in two local fields of the form:

```
private
  OldWinProc, NewWinProc: Pointer;
  procedure NewWinProcedure (var Msg: TMessage);
```

The form also has a method we'll use as a new window procedure, with the actual code used to paint on the background of the window. Because this is a method and not a plain window procedure, the program has to call the `MakeObjectInstance` method to add a prefix to

the method and let the system use it as if it were a function. All this description is summarized by just two complex statements:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    NewWinProc := MakeObjectInstance (NewWinProcedure);
    OldWinProc := Pointer (SetWindowLong (ClientHandle, gw1_WndProc, Cardinal
        (NewWinProc)));
    OutCanvas := TCanvas.Create;
end;
```

The window procedure we install calls the default one. Then, if the message is `wm_EraseBkgnd` and the image is not empty, we draw it on the screen many times using the `Draw` method of a temporary canvas. This canvas object is created when the program starts (see the code above) and connected to the handle passed as `wParam` parameter by the message. With this approach, we don't have to create a new `TCanvas` object for every background painting operation requested, thus saving a little time in the frequent operation. Here is the code, which produces the output already seen in Figure 10.6:

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
    BmpWidth, BmpHeight: Integer;
    I, J: Integer;
begin
    // default processing first
    Msg.Result := CallWindowProc (OldWinProc, ClientHandle, Msg.Msg, Msg.wParam,
        Msg.lParam);

    // handle background repaint
    if Msg.Msg = wm_EraseBkgnd then
    begin
        BmpWidth := MainForm.Image1.Width;
        BmpHeight := MainForm.Image1.Height;
        if (BmpWidth <> 0) and (BmpHeight <> 0) then
        begin
            OutCanvas.Handle := Msg.wParam;
            for I := 0 to MainForm.ClientWidth div BmpWidth do
                for J := 0 to MainForm.ClientHeight div BmpHeight do
                    OutCanvas.Draw (I * BmpWidth, J * BmpHeight,
                        MainForm.Image1.Picture.Graphic);
        end;
    end;
end;
```



## Visual Form Inheritance

When you need to build two or more similar forms, possibly with different event handlers, you can use dynamic techniques, hide or create new components at run time, change event handlers, and use `if` or case statements. Or you can apply the object-oriented techniques, thanks to visual form inheritance. In short, instead of creating a form based on `TForm`, you can inherit a form from an existing one, adding new components or altering the properties of the existing ones. But what is the real advantage of visual form inheritance?

Well, this mostly depends on the kind of application you are building. If it has multiple forms, some of which are very similar to each other or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the forms of an application.

You can also use visual form inheritance to customize an application for different clients, without duplicating any source code or form definition code; just inherit the specific versions for a client from the standard forms. Remember that the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method in a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

### NOTE

Delphi includes another feature, frames, which resembles visual form inheritance. In both cases, you can work at design time on two versions of a form/frame. However, in visual form inheritance, you are defining two different classes (parent and derived), whereas with frames, you work on a class and an instance. Frames will be discussed in detail later in this chapter.

## Inheriting from a Base Form

The rules governing visual form inheritance are quite simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two

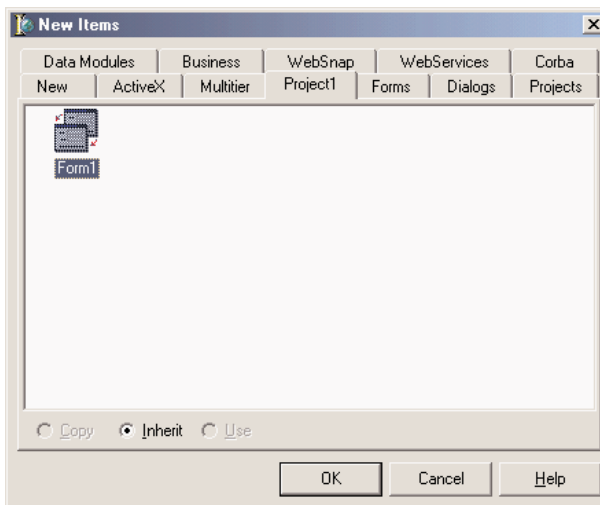
property values by using the Revert to Inherited local menu command of the Object Inspector. The same thing is accomplished by setting the two properties to the same value and recompiling the code. After modifying multiple properties, you can resynchronize them all to the base version by applying the Revert to Inherited command of the component's local menu.

Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To describe how visual form inheritance works, I've built a very simple example, called VFI. I'll describe step-by-step how to build it. First, start a new project, and add four buttons to its main form. Then select File > New > Other, and choose the page with the name of the project in the New Items dialog box (see Figure 10.7).

**FIGURE 10.7:**

The New Items dialog box allows you to create an inherited form.



In the New Items dialog, you can choose the form from which you want to inherit. The new form has the same four buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

And here is its initial class declaration, where you can see that the base class is not the usual TForm but the actual base class form:

```
type
  TForm2 = class(TForm1)
private
  { Private declarations }
```

```

public
  { Public declarations }
end;

```

Notice the presence of the `inherited` keyword in the textual description; also notice that the form indeed has some components, although they are defined in the base class form. If you move the form and add the caption of one of the buttons, the textual description will change accordingly:

```

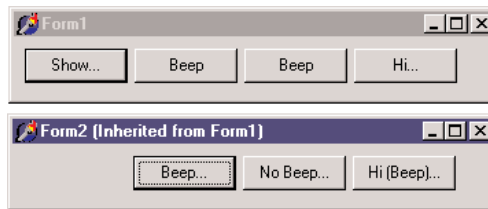
inherited Form2: TForm2
  Left = 313
  Top = 202
  Caption = 'Form2'
  inherited Button2: TButton
    Caption = 'Beep...'
end
end

```

Only the properties with a different value are listed (and by removing these properties from the textual description of the inherited form, you can reset them to the value of the base form, as I mentioned before). I've actually changed the captions of most buttons, as you can see in Figure 10.8.

**FIGURE 10.8:**

The two forms of the VFI example at run time



Each of the buttons of the first form has an `OnClick` handler, with simple code. The first button shows the inherited form calling its `Show` method; the second and the third buttons call the `Beep` procedure; and the last button displays a simple message.

What happens in the inherited form? First we should remove the `Show` button, because the secondary form is already visible. However, we cannot delete a component from an inherited form. An alternative solution is to leave the component there but set its `Visible` property to `False`. The button will still be there but not visible (as you can guess from Figure 10.8). The other three buttons will be visible but with different handlers. This is simple to accomplish. If you select the `OnClick` event of a button in the inherited form (by double-clicking it), you'll get an empty method slightly different from the default one:

```

procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
end;

```

The `inherited` keyword stands for a call to the corresponding event handler of the base form. This keyword is always added by Delphi, even if the handler is not defined in the parent class (and this is reasonable, because it might be defined later) or if the component is not present in the parent class (which doesn't seem like a great idea to me). It is very simple to execute the code of the base form and perform some other operations:

```
procedure TForm2.Button2Click(Sender: TObject);  
begin  
  inherited;  
  ShowMessage ('Hi');  
end;
```

This is not the only choice. An alternative approach is to write a brand-new event handler and not execute the code of the base class, as I've done for the third button of the VFI example: To accomplish this, simply remove the `inherited` keyword. Still another choice includes calling a base-class method after some custom code has been executed, calling it when a condition is met, or calling the handler of a different event of the base class, as I've done for the fourth button:

```
procedure TForm2.Button4Click(Sender: TObject);  
begin  
  inherited Button3Click (Sender);  
  inherited;  
end;
```

You probably won't do this very often, but you must be aware that you can. Of course, you can consider each method of the base form as a method of your form, and call it freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at real-world examples more complex than this book has room to explore. There is something else I want to show you here: *visual form polymorphism*.

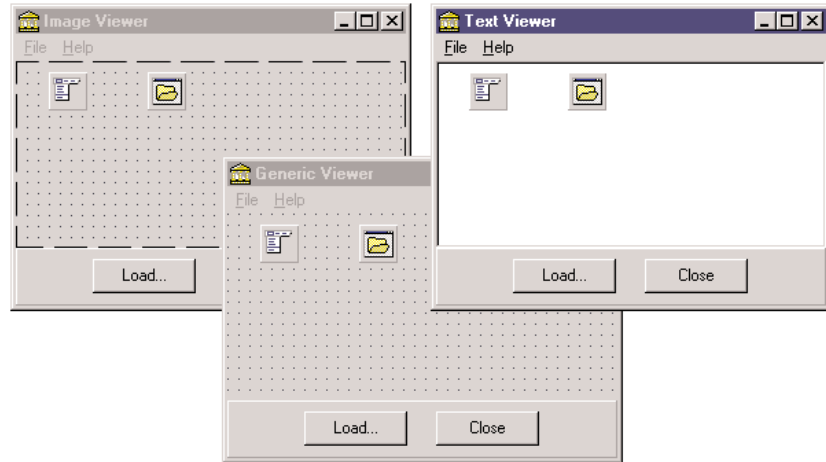
## Polymorphic Forms

The problem is simple. If you add an event handler to a form and then change it in an inherited form, there is no way to refer to the two methods using a common variable of the base class, because the event handlers use static binding by default.

Confusing? Here is an example, which is intended for experienced Delphi programmers. Suppose you want to build a bitmap viewer form and a text viewer form in the same program. The two forms have similar elements, a similar toolbar, a similar menu, an `OpenDialog` component, and different components for viewing the data. So you decide to build a base-class form containing the common elements and inherit the two forms from it. You can see the three forms at design time in Figure 10.9.

**FIGURE 10.9:**

The base-class form and the two inherited forms of the Poliform example at design time



The main form contains a toolbar panel with a few buttons (real toolbars apparently have a few problems with visual form inheritance), a menu, and an open dialog component. The two inherited forms have only minor differences, but they feature a new component, either an image viewer (TImage) or a text viewer (TMemo). They also modify the settings of the OpenFileDialog component, to refer to different types of files.

The main form includes some common code. The Close button and the File > Close command call the Close method of the form. The Help > About command shows a simple message box. The Load button of the base form has the following code:

```
procedure TViewerForm.ButtonLoadClick(Sender: TObject);
begin
    ShowMessage ('Error: File-loading code missing');
end;
```

The File > Load command, instead, calls another method:

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
    LoadFile;
end;
```

This method is defined in the TViewerForm class as a virtual abstract method (so that the class of the base form is actually an abstract class). Because this is an abstract method, we will need to redefine it (and override it) in the inherited forms. The code of this LoadFile method simply uses the OpenFileDialog component to ask the user to select an input file and loads it into the image component:

```
procedure TImageViewerForm.LoadFile;
begin
```

```

if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (OpenDialog1.FileName);
end;

```

The other inherited class has similar code, loading the text into the memo component. The project has one more form, a main form with two buttons, used to reload the files in each of the viewer forms. The main form is the only form created by the project when it starts. The generic viewer form is never created: it is only a generic base class, containing common code and components of the two subclasses. The forms of the two subclasses are created in the OnCreate event handler of the main form:

```

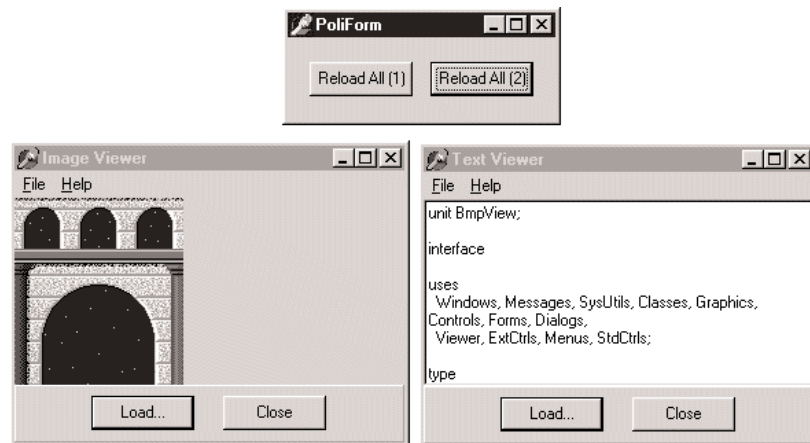
procedure TMainForm.FormCreate(Sender: TObject);
var
    I: Integer;
begin
    FormList [1] := TTextViewerForm.Create (Application);
    FormList [2] := TImageViewerForm.Create (Application);
    for I := 1 to 2 do
        FormList[I].Show;
end;

```

See Figure 10.10 for the resulting forms (with text and image already loaded in the viewers). FormList is a *polymorphic* array of generic TViewerForm objects, declared in the TMainForm class.

**FIGURE 10.10:**

The PoliForm example at run time



Note that to make this declaration in the class, you need to add the Viewer unit (but not the specific forms) in the uses clause of the interface portion of the main form. The array of

forms is used to load a new file in each viewer form when one of the two buttons is pressed. The handlers of the two buttons' `OnClick` events use different approaches:

```

procedure TMainForm.ReloadButton1Click(Sender: TObject);
var
    I: Integer;
begin
    for I := 1 to 2 do
        FormList [I].ButtonLoadClick (Self);
end;

procedure TMainForm.ReloadButton2Click(Sender: TObject);
var
    I: Integer;
begin
    for I := 1 to 2 do
        FormList [I].LoadFile;
end;

```

The second button simply calls a virtual method, and it will work without any problem. The first button calls an event handler and will always reach the generic `TFormView` class (displaying the error message of its `ButtonLoadClick` method). This happens because the method is static, not virtual.

Is there a way to make this approach work? Sure. Declare the `ButtonLoadClick` method of the `TFormView` class as virtual, and declare it as overridden in each of the inherited form classes, as we do for any other virtual method:

```

type
    TViewerForm = class(TForm)
        // components and plain methods...
        procedure ButtonLoadClick(Sender: TObject); virtual;
    public
        procedure LoadFile; virtual; abstract;
    end;
...
type
    TImageViewerForm = class(TViewerForm)
        Image1: TImage;
        procedure ButtonLoadClick(Sender: TObject); override;
    public
        procedure LoadFile; override;
    end;

```

This trick really works, although it is never mentioned in the Delphi documentation. This ability to use virtual event handlers is what I actually mean by visual form polymorphism. In other (more technical) words, you can assign a virtual method to an event property, which will take the address of the method according to the instance available at run time.

## Understanding Frames

Chapter 1 briefly discussed frames, which were introduced in Delphi 5. We've seen that you can create a new frame, place some components in it, write some event handlers for the components, and then add the frame to a form. In other words, a frame is similar to a form, but it defines only a portion of a window, not a complete window. This is certainly not a feature worth a new construct. The totally new element of frames is that you can create multiple instances of a frame at design time, and you can modify the class and the instance at the same time. This makes frames an effective tool for creating customizable composite controls at design time, something close to a visual component-building tool.

In visual form inheritance you can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one, unless this overrides some property or event. With frames, you work on a class (as usual in Delphi), but the difference is that you can also customize one or more instances of the class at design time. When you work on a form, you cannot change a property of the `TForm1` class for the `Form1` object at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, in fact, you can customize each of the instances at design time. Wasn't this already possible with component templates? It was, but component templates were based on the concept of copying and pasting some components and their code. There was no way to change the original definition of the template and see the effect in every place it was used. That is what happens with frames (and in a different way with visual form inheritance); changes to the original version (the class) are reflected in the copies (the instances).

Let's discuss a few more elements of frames with an example from the CD, called `Frames2`. This program has a frame with a list box, an edit box, and three buttons with simple code operating on the components. The frame also has a bevel aligned to its client area, because frames have no border. Of course, the frame has also a corresponding class, which looks like a form class:

```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
```



```

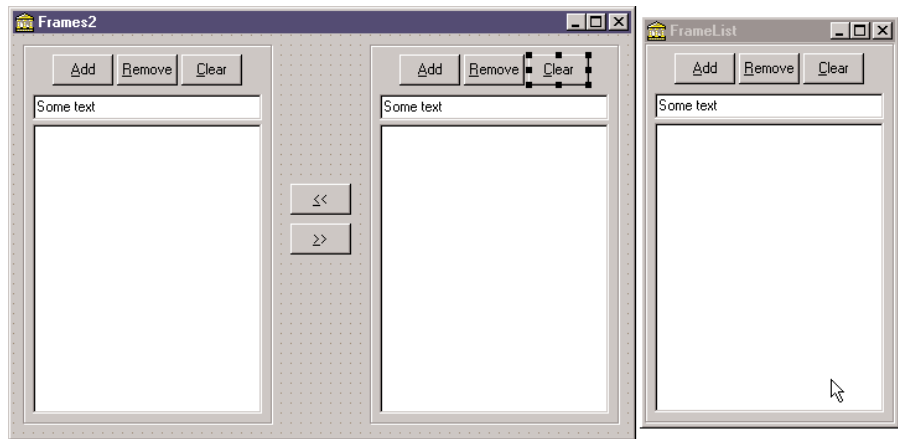
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

```

What is different is that you can add the frame to a form. I've used two instances of the frame in the example (as you can see in Figure 10.11) and modified the behavior slightly. The first instance of the frame has the list box items sorted. When you change a property of a component of a frame, the DFM file of the hosting form will list the differences, as it does with visual form inheritance:

**FIGURE 10.11:**

A frame and two instances of it at design time, in the Frames2 example



```

object FormFrames: TFormFrames
    Caption = 'Frames2'
    inline FrameList1: TFrameList
        Left = 8
        Top = 8
        inherited ListBox: TListBox
            Sorted = True
        end
    end
    inline FrameList2: TFrameList
        Left = 232
        Top = 8
        inherited btnClear: TButton
            OnClick = FrameList2btnClearClick
        end
    end
end

```

As you can see from the listing, the DFM file for a form that has frames uses a new DFM keyword, `inline`. The references to the modified components of the frame, instead, use the `inherited` keyword, although this term is used with an extended meaning. `inherited` here doesn't refer to a base class we are inheriting from, but to the class we are instantiating (or inheriting) an object from. It was probably a good idea, though, to use an existing feature of visual form inheritance and apply it to the new context. The effect of this approach, in fact, is that you can use the Revert to Inherited command of the Object Inspector or of the form to cancel the changes and get back to the default value of properties.

Notice also that unmodified components of the frame class are not listed in the DFM file of the form using the frame, and that the form has two frames with different names, but the components on the two frames have the same name. In fact, these components are not owned by the form, but are owned by the frame. This implies that the form has to reference those components through the frame, as you can see in the code for the buttons that copy items from one list box to the other:

```
procedure TFormFrames.btnLeftClick(Sender: TObject);  
begin  
    FrameList1.ListBox.Items.AddStrings (FrameList2.ListBox.Items);  
end;
```

Finally, besides modifying properties of any instance of a frame, you can change the code of any of its event handlers. If you double-click one of the buttons of a frame while working on the form (not on the stand-alone frame), Delphi will generate this code for you:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);  
begin  
    FrameList2.btnClearClick(Sender);  
end;
```

The line of code automatically added by Delphi corresponds to a call to the inherited event handler of the base class in visual form inheritance. This time, however, to get the default behavior of the frame we need to call an event handler and apply it to a specific instance—the frame object itself. The current form, in fact, doesn't include this event handler and knows nothing about it.

Whether you leave this call in place or remove it depends on the effect you are looking for. In the example I've decided to conditionally execute the default code, depending on the user confirmation:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);  
begin  
    if MessageDlg ('OK to empty the list box?', mtConfirmation,  
        [mbYes, mbNo], 0) = idYes then  
        // execute standard frame code  
        FrameList2.btnClearClick(Sender);  
end;
```

**TIP**

By the way, note that because the event handler has some code, leaving it empty and saving the form won't remove it as usual: in fact, it isn't empty! Instead, if you simply want to omit the default code for an event, you need to add at least a comment to it, to avoid it being automatically removed by the system!

## Frames and Pages

When you have a dialog box with many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared in a single form. Also, creating all these components (and initializing them) might result in a delay in the display of the dialog box. Frames actually don't reduce the construction and initialization time of equivalently loaded forms; quite the contrary, as loading frames is more complicated for the streaming system than loading simple components. However, using frames you can load only the visible pages of a multipage dialog box, reducing the *initial* load time, which is what the user perceives.

Frames can solve both of these issues. First, you can easily divide the code of a single complex form into one frame per page. The form will simply host all of the frames in a PageControl. This certainly helps you to have simpler and more focused units and makes it simpler to reuse a specific page in a different dialog box or application. Reusing a single page of a PageControl without using a frame or an embedded form, in fact, is far from simple.

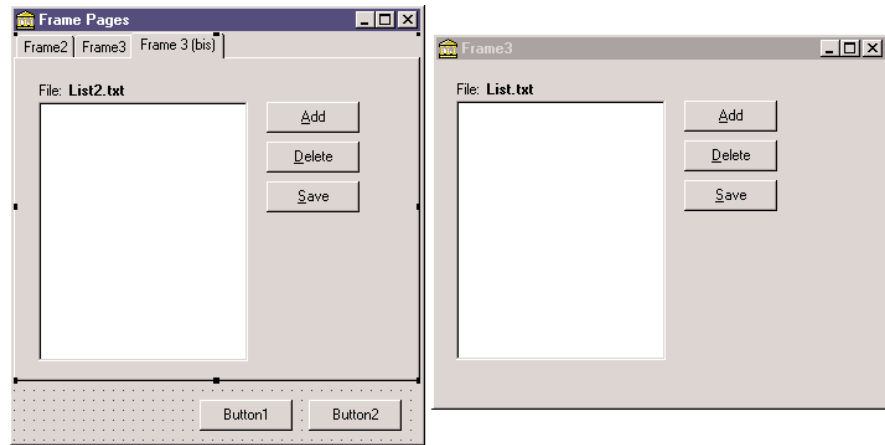
As an example of this approach I've built the FramePag example, which has some frames placed inside the three pages of a PageControl, as you can see in Figure 10.12. All of the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them. Actually two of the pages have the same frame, but the two instances of the frame have some differences at design time. The frame, called Frame3 in the example, has a list box that is populated with a text file at startup, and has buttons to modify the items in the list and saves them to a file. The filename is placed inside a label, so that you can easily select a file for the frame at design time by changing the Caption of the label.

**TIP**

Being able to use multiple instances of a frame is one of the reasons this technique was introduced, and customizing the frame at design time is even more important. Because adding properties to a frame and making them available at design time requires some customized and complex code, it is nice to use a component to host these custom values. You have the option of hiding these components (such as the label in our example) if they don't pertain to the user interface.

**FIGURE 10.12:**

Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks.



In the example, we need to load the file when the frame instance is created. Because frames have no `OnCreate` event, our best choice is probably to override the `CreateWnd` method. Writing a custom constructor, in fact, doesn't work as it is executed too early—before the specific label text is available. Here is the frame class code:

```
type
  TFrame3 = class(TFrame)
    ...
  public
    procedure CreateWnd; override;
```

Within the `CreateWnd` method, we simply load the list box content from a file.

## Multiple Frames with No Pages

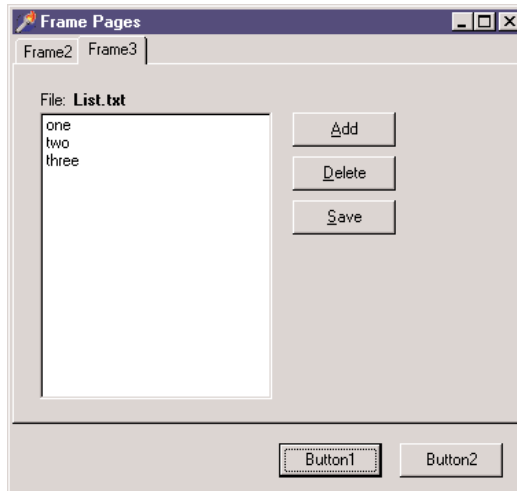
Another approach is to avoid creating all of the pages along with the form hosting them. This can be accomplished by leaving the `PageControl` empty and creating the frames only when a page is displayed. Actually, when you have frames on multiple pages of a `PageControl`, the windows for the frames are created only when they are first displayed, as you can find out by placing a breakpoint in the creation code of the last example.

As an even more radical approach, you can get rid of the page controls and use a `TabControl`. Used this way, the tab has no connected tab sheets (or pages) but can display only one set of information at a time. For this reason, we'll need to create the current frame and destroy the previous one or simply hide it by setting its `Visible` property to `False` or by calling the `BringToFront` of the new frame. Although this sounds like a lot of work, in a large application this technique can be worth it for the reduced resource and memory usage you can obtain by applying it.

To demonstrate this approach, I've built an example similar to the previous one, this time based on a `TabControl` and dynamically created frames. The main form, visible at run time in Figure 10.13, has only a `TabControl` with one page for each frame:

**FIGURE 10.13:**

The first page of the Frame-Tab example at run time. The frame inside the tab is created at run time.



```

object Form1: TForm1
  Caption = 'Frame Pages'
  OnCreate = FormCreate
  object Button1: TButton...
  object Button2: TButton...
  object Tab: TTabControl
    Anchors = [akLeft, akTop, akRight, akBottom]
    Tabs.Strings = ( 'Frame2' 'Frame3' )
    OnChange = TabChange
  end
end

```

I've given each tab a caption corresponding to the name of the frame, because I'm going to use this information to create the new pages. When the form is created, and whenever the user changes the active tab, the program gets the current caption of the tab and passes it to the custom `ShowFrame` method. The code of this method, listed below, checks whether the requested frame already exists (frame names in this example follow the Delphi standard of having a number appended to the class name), and then brings it to the front. If the frame doesn't exist, it uses the frame name to find the related frame class, creates an object of that

class, and assigns a few properties to it. The code makes extensive use of class references and dynamic creation techniques:

```
type
  TFrameClass = class of TFrame;

procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + '1') as TFrame;
  if not Assigned (Frame) then
    begin
      FrameClass := TFrameClass (FindClass ('T' + FrameName));
      Frame := FrameClass.Create (Self);
      Frame.Parent := Tab;
      Frame.Visible := True;
      Frame.Name := FrameName + '1';
    end;
  Frame.BringToFront;
end;
```

To make this code work, you have to remember to add a call to `RegisterClass` in the initialization section of each unit defining a frame.

## Base Forms and Interfaces

We have seen that when you need two similar forms inside an application, you can use visual form inheritance to inherit one from the other or both of them from a common ancestor. The advantage of visual form inheritance is that you can use it to inherit the visual definition, the DFM. However, this is not always requested.

At times, you might want several forms to exhibit a common behavior, or respond to the same commands, without having any shared component or user interface elements. Using visual form inheritance with a base form that has no extra components makes little sense to me. I rather prefer defining my own custom form class, inherited from `TForm`, and then manually editing the form class declarations to inherit from this custom base form class instead of the standard one. If all you need is to define some shared methods, or override `TForm` virtual methods in a consistent way, defining custom form classes can be a very good idea.

## Using a Base Form Class

A simple demonstration of this technique is available in the FormIntf demo, showcasing also the use of interfaces for forms. In a new unit, called SaveStatusForm, I've defined the following form class (with no related DFM file—don't use the New Form command, but create a new unit and type the code in it):

```
type
  TSaveStatusForm = class (TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

The two overridden methods are called at the same time of the event handler, so that I can attach extra code (allowing the event handler to be defined as usual). Inside the two methods I simply load or save the form position inside an INI file of the application, in a section marked with the form caption. Here is the code of the two methods:

```
procedure TSaveStatusForm.DoCreate;
var
  Ini: TIniFile;
begin
  inherited;
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Left := Ini.ReadInteger(Caption, 'Left', Left);
  Top := Ini.ReadInteger(Caption, 'Top', Top);
  Width := Ini.ReadInteger(Caption, 'Width', Width);
  Height := Ini.ReadInteger(Caption, 'Height', Height);
  Ini.Free;
end;

procedure TSaveStatusForm.DoDestroy;
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Ini.WriteInteger(Caption, 'Left', Left);
  Ini.WriteInteger(Caption, 'Top', Top);
  Ini.WriteInteger(Caption, 'Width', Width);
  Ini.WriteInteger(Caption, 'Height', Height);
  Ini.Free;
  inherited;
end;
```

Again, this is a simple common behavior for your forms, but you can define a very complex class here. To use this as a base class of the forms you create, simply let Delphi create the forms as usual (with no inheritance) and then update the form declaration to something like:

```
type
  TFormBitmap = class(TSaveStatusForm)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    ...
```

Simple as it seems, this is a very powerful technique, as all you need to do is change the definition of the forms of your application to refer to this base class. If even this is too tedious, as you might want to change this base class in the life of your program, you can use an extra trick, “interposer” classes.

## INI Files and the Registry in Delphi

If you want to save information about the status of an application in order to restore it the next time the program is executed, you can use the explicit support that Windows provides for storing this kind of information. INI files, the old Windows standard, are once again the preferred way to save application data. The alternative is the Registry, which is still quite popular. Delphi provides ready-to-use classes to manipulate both.

### The *TIniFile* Class

For INI files, Delphi has a `TIniFile` class. Once you have created an object of this class and connected it to a file, you can read and write information to it. To create the object, you need to call the constructor, passing a filename to it, as in the following code:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create ('myprogram.ini');
```

There are two choices for the location of the INI file. The code just listed will store the file in the Windows directory or a user folder for settings in Windows 2000. To store data locally to the application (as opposed to local to the current user), you should provide a full path to the constructor.

INI files are divided into sections, each indicated by a name enclosed in square brackets. Each section can contain multiple items of three possible kinds: strings, integers, or Booleans. The `TIniFile` class has three Read methods, one for each kind of data: `ReadBool`, `ReadInteger`, and `ReadString`. There are also three corresponding methods to write the data: `WriteBool`, `WriteInteger`, and `WriteString`. Other methods allow you to read or erase a whole section. In the Read methods, you can also specify a default value to be used if the corresponding entry doesn't exist in the INI file.

*Continued on next page*



By the way, notice that Delphi uses INI files quite often, but they are disguised with different names. For example, the desktop (.dsk) and options (.dof) files are structured as INI files.

### The *TRegistry* and *TRegIniFile* classes

The Registry is a hierarchical database of information about the computer, the software configuration, and the user preferences. Windows has a set of API functions to interact with the Registry; you basically open a key (or folder) and then work with subkeys (or subfolders) and with values (or items), but you must be aware of the structure and the details of the Registry.

Delphi provides basically two approaches to the use of the Registry. The *TRegistry* class provides a generic encapsulation of the Registry API, while the *TRegIniFile* class provides the interface of the *TIniFile* class but saves the data in the Registry. This class is the natural choice for portability between INI-based and Registry-based versions of the same program. When you create a *TRegIniFile* object, your data ends up in the current user information, so you'll generally use a constructor like this:

```
IniFile := TRegIniFile.Create ('Software\MyCompany\MyProgram');
```

By using the *TIniFile* and the *TRegistryIniFile* classes offered by the VCL, you can move from one model of local and per-user storage to the other. Not that I think you should use the Registry a lot, as the idea of having a centralized repository for the settings of each application was a architectural error. Even Microsoft acknowledges this (without really admitting the error) by suggesting, in the Windows 2000 Compatibility Requirements, that you not use the Registry anymore for applications settings, but go back to the use of INI files.

## An Extra Trick: Interposer Classes

In contrast with Delphi VCL components, which must have unique names, Delphi classes in general must be unique only within their unit. This means you can have two different units defining a class with the same name. This looks really weird, at first sight, but can be useful. For example, Borland is using this technique to provide compatibility between VCL and VisualCLX classes. Both have a *TForm* class, one defined in the Forms unit and the other in the QForms unit. How can this be interesting for the topic discussed here?

### NOTE

This technique is actually much older than CLX/VCL. For example, the service and control panel applet units define their own *TApplication* object, which is not related to the *TApplication* used by VCL visual GUI applications and defined in the Forms unit.

There is a technique that I've seen mentioned with the name "interposer classes" in an old issue of *The Delphi Magazine*, which suggested replacing standard Delphi class names with your own versions, having the same class name. This way you can use Delphi designer referring to Delphi standard components at design time, but using your own classes at run time.

The idea is simple. In the `SaveStatusForm` unit, I could have defined the new form class as follows:

```
type
  TForm = class (Forms.TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

This class is called `TForm`, and inherits from `TForm` of the `Forms` unit (this last reference is compulsory to avoid a kind of recursive definition). In the rest of the program, at this point, you don't need to change the class definition for your form, but simply add the unit defining the interposer class (the `SaveStatusForm` unit in this case) in the `uses` statement *after* the unit defining the Delphi class. The order of the unit in the `uses` statement is important here, and the reason some people criticize this technique, as it is hard to know what is going on. I have to agree: I find interposer classes handy at times (more for components than for forms, I have to say), but their use makes programs less readable and at times even harder to debug.

## Using Interfaces

Another technique, which is slightly more complex but even more powerful than the definition of a common base form class, is to have forms that implement specific interfaces. This way you can have forms implementing one or more of these interfaces, query each form for the interfaces it implements, and call the supported methods.

As an example (available in the same `FormIntf` program I began discussing in the last section), I've defined a simple interface for loading and storing:

```
type
  IFormOperations = interface
    ['{DACFDB76-0703-4A40-A951-10D140B4A2A0}']
    procedure Load;
    procedure Save;
  end;
```

Each form can optionally implement this interface, as the following `TFormBitmap` class:

```
type
  TFormBitmap = class(TForm, IFormOperations)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    SavePictureDialog1: TSavePictureDialog;
  public
    procedure Load;
    procedure Save;
  end;
```

You can see the actual code of the example for the code of the Load and Save methods, which use the standard dialog boxes to load or save the image. (In the example's code, the form also inherits from the TSaveStatusForm class.)

When an application has one or more forms implementing interfaces, you can apply a given interface method to all the forms supporting it, with code like this (extracted from the main form of the IntfForm example):

```
procedure TFormMain.btnLoadClick(Sender: TObject);  
var  
    i: Integer;  
begin  
    for i := 0 to Screen.FormCount - 1 do  
        if Supports (Screen.Forms [i], IFormOperations) then  
            (Screen.Forms [i] as IFormOperations).Load;  
end;
```

Consider a business application when you can synchronize all of the forms to the data of a specific company, or a specific business event. And consider also that, unlike inheritance, you can have several forms each implementing multiple interfaces, with unlimited combinations. This is why using an architecture like this can improve a complex Delphi application a great deal, making it much more flexible and easier to adapt to implementation changes.

## What's Next?

After the detailed description of forms and secondary forms in the previous chapters, I have focused on the architecture of applications, discussing both how Delphi's Application object works and how we can structure applications with multiple forms.

In particular, I've discussed MDI, visual form inheritance, and frames. Toward the end I also discussed custom architectures, with form inheritance and interfaces. Now we can move forward to another key element of non-trivial Delphi applications: building custom components to use within your programs. It is possible to write a specific book about this, so the description won't be exhaustive, but you should be able to get a comprehensive overview.

Another element related to the architecture of Delphi applications is the use of packages, which I'll introduce as a technology related to components but which really goes beyond this. In fact, you can structure the code of a large application in multiple packages, containing forms and other units. The development of programs based on multiple executable files, libraries, and packages, is discussed in Chapter 12.

After this further step, I will start delving into Delphi database programming, certainly another key element of the Borland development environment.

# Creating Components

- Extending the Delphi library
- Writing packages
- Customizing existing components
- Building graphical components
- Defining custom events
- Using array properties
- Placing a dialog box in a component
- Writing property and component editors

**W**hile most Delphi programmers are probably familiar with using existing components, at times it can also be useful to write our own components or to customize existing ones. One of the most interesting aspects of Delphi is that creating components is simple. For this reason, even though this book is intended for Delphi application programmers and not Delphi tool writers, this chapter will cover the topic of creating components and introduce Delphi add-ins, such as component and property editors.

This chapter gives an overview of writing Delphi components and presents some simple examples. There is not enough space to present very complex components, but the ideas in this chapter will cover all the basics to get you started.

**NOTE**

You'll find a more information about writing components in Chapter 18, "Writing Database Components," including how to build data-aware components.

## Extending the Delphi Library

Delphi components are classes, and the Visual Components Library (VCL) is the collection of all the classes defining Delphi components. Each time you add a new package with some components to Delphi, you actually extend VCL with a new class. This new class will be derived from one of the existing component-related classes or the generic `TComponent` class, adding new capabilities to those it inherits.

You can derive a new component from an existing component or from an *abstract component class*—one that does not correspond to a usable component. The VCL hierarchy includes many of these intermediate classes (often indicated with the `TCustom` prefix in their name) to let you choose a default behavior for your new component and to change its properties.

## Component Packages

Components are added to component packages. Each component package is basically a DLL (a dynamic link library) with a BPL extension (which stands for Borland Package Library).

Packages come in two flavors: design-time packages used by the Delphi IDE and run-time packages optionally used by applications. The design-only or run-only package option determines the package's type. When you attempt to install a package, the IDE checks whether it has the design-only or run-only flags, and decides whether to let the user install the package and whether it should be added to the list of run-time packages. Since there are two

nonexclusive options, each with two possible states, there are four different kinds of component packages—two main variations and two special cases:

- Design-only component packages can be installed in the Delphi environment. These packages usually contain the design-time parts of a component, such as its property editors and the registration code. Often they can also contain the components themselves, although this is not the most professional approach. The code of the components of a design-only package is usually statically linked into the executable file, using the code of the corresponding Delphi Compiled Unit (DCU) files. Keep in mind, however, that it is also technically possible to use a design-only package as a run-time package.
- Run-only component packages are used by Delphi applications at run time. They cannot be installed in the Delphi environment, but they are automatically added to the list of run-time packages when they are required by a design-only package you install. Run-only packages usually contain the code of the component classes, but no design-time support (this is done to minimize the size of the component libraries you ship along with your executable file). Run-only packages are important because they can be freely distributed along with applications, but other Delphi programmers won't be able to install them in the environment to build new programs.
- Plain component packages (having neither the design-only nor the run-only option set) cannot be installed and will not be added to the list of run-time packages automatically. This might make sense for utility packages used by other packages, but they are certainly rare.
- Packages with both flags set can be installed and are automatically added to the list of run-time packages. Usually these packages contain components requiring little or no design-time support (apart from the limited component registration code). Keep in mind, however, that users of applications built with these packages can use them for their own development.

**TIP**

The filenames of Delphi's own design-only packages start with the letters *DCL* (for example, *DCLSTD60.BPL*); filenames of run-only packages start with the letters *VCL* (for example, *VCL60.BPL*). You can follow the same approach for your own packages, if you want.

In Chapter 1, “The Delphi 6 IDE,” we discussed the effect of packages on the size of a program's executable file. Now we'll focus on building packages, since this is a required step in creating or installing components in Delphi.

When you compile a run-time package, you produce both a dynamic link library with the compiled code (the BPL file) and a file with only symbol information (a DCP file), including no compiled machine code. The latter file is used by the Delphi compiler to gather symbol

information about the units that are part of the package without having access to the unit (DCU) files, which contain both the symbol information and the compiled machine code. This reduces compilation time and allows you to distribute just the packages without the precompiled unit files. The precompiled units are still required to statically link the components into an application. Distribution of precompiled DCU files (or source code) may make sense depending on the kind of components you develop. We'll see how to create a package after we've discussed some general guidelines and built our very first component.

**NOTE**

DLLs are executable files containing collections of functions and classes, which can be used by an application or another DLL at run time. The typical advantage is that if many applications use the same DLL, only one copy needs to be on the disk or loaded in memory, and the size of each executable file will be much smaller. This is what happens with Delphi packages, as well. Chapter 12, "Libraries and Packages," looks at DLLs and packages in more detail.

## Rules for Writing Components

Some general rules govern the writing of components. You can find a detailed description of most of them in the *Delphi Component Writer's Guide* Help file, which is required reading for Delphi component writers.

Here is my own summary of the rules for component writers:

- Study the Object Pascal language with care. Particularly important concepts are inheritance, method overriding and overloading, the difference between public and published sections of a class, and the definition of properties and events. If you don't feel confident with the Object Pascal language or the basic ideas about VCL, you can refer to the overall description of the language and library presented in Part I of the book, particularly Chapters 3 ("The Object Pascal Language: Inheritance and Polymorphism") and 5 ("Core Library Classes").
- Study the structure of the VCL class hierarchy and keep a graph of the classes at hand (such as the one included with Delphi).
- Follow the standard Delphi naming conventions. There are several of them for components, as we will see, and following these rules makes it easier for other programmers to interact with your components and further extend them.
- Keep components simple, mimic other components, and avoid dependencies. These three rules basically mean that a programmer using your components should be able to use them as easily as preinstalled Delphi components. Use similar property, method, and event names whenever possible. If users don't need to learn complex rules about the use of your component (that is, if the dependencies between methods or properties are limited) and can simply access properties with meaningful names, they'll be happy.

- Use exceptions. When something goes wrong, the component should raise an exception. When you are allocating resources of any kind, you must protect them with `try/finally` blocks and destructor calls.
- To complete a component, add a bitmap to it, to be used by Delphi's Component Palette. If you intend your component to be used by more than a few people, consider adding a Help file as well.
- Be ready to write *real* code and forget about the visual aspects of Delphi. Writing components generally means writing code without visual support (although Class Completion can speed up the coding of plain classes quite a lot). The exception to this rule is that you can use frames to write components visually.

**NOTE**

You can also use a third-party component writing tool to build your component or to speed up its development. The most powerful third-party tool for creating Delphi components I know of is the Component Development Kit (CDK) from Eagle Software ([www.eagle-software.com](http://www.eagle-software.com)), but many others are available.

## The Base Component Classes

To build a new component you generally start from an existing one, or from one of the base classes of VCL. In both cases your component is in one of three broad categories of components (introduced in Chapter 5), set by the three basic classes of the component hierarchy:

- `TWinControl` is the parent class of any component based on a window. Components that descend from this class can receive the input focus and get Windows messages from the system. You can also use their window handle when calling API functions. When creating a brand-new window control, you'll generally inherit from the derived class `TCustomControl`, which has a few extra useful features (particularly some support for painting the control).
- `TGraphicControl` is the parent class of visible components that have no Windows handle (which saves some Windows resources). These components cannot receive the input focus or respond to Windows messages directly. When creating a brand-new graphical control, you'll inherit directly from this class (which has a set of features very similar to `TCustomControl`).
- `TComponent` is the parent class of all components (including the controls) and can be used as a direct parent class for nonvisual components.

In the rest of the chapter, we will build some components using various parent classes, and we'll look at the differences among them. We'll start with components inheriting from existing components or classes at a low level of the hierarchy, and then we'll see examples of classes inheriting directly from the ancestor classes mentioned above.



## Building Your First Component

Building components is an important activity for Delphi programmers. The basic idea is that any time you need the same behavior in two different places in an application, or in two different applications, you can place the shared code inside a class—or, even better, a component.

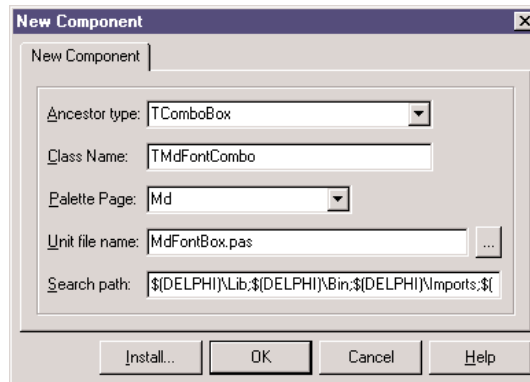
In this section I'll just introduce a couple of simple components, to give you an idea of the steps required to build one and to show you different things you can do to customize an existing component with a limited amount of code.

### The Fonts Combo Box

Many applications have a toolbar with a combo box you can use to select a font. If you often use a customized combo box like this, why not turn it into a component? It would probably take less than a minute. To begin, close any active projects in the Delphi environment and start the Component Wizard, either by choosing Component > New Component or by selecting File > New to open the Object Repository and then choosing the Component in the New page. As you can see in Figure 11.1, the Component Wizard requires the following information:

**FIGURE 11.1:**

Defining the new TMDFont-Combo component with the Component Wizard



- The name of the ancestor type: the component class you want to inherit from. In this case we can use `TComboBox`.
- The name of the class of the new component you are building; we can use `TMdFontCombo`.
- The page of the Component Palette where you want to display the new component, which can be a new or an existing page. We can create a new page, called *Md*.

- The filename of the Pascal unit where Delphi will place the source code of the new component; we can type MdFontBox.
- The current search path (which should be set up automatically).

Click the OK button, and the Component Wizard will generate the following simple Pascal source file with the structure of your component. The Install button can be used to install the component in a package immediately. Let's look at the code first, Listing 11.1, and then discuss the installation.



### Listing 11.1: Code of the TMdFontCombo, generated by the Component Wizard

```
unit MdFontBox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TMdFontCombo = class (TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md', [TMdFontCombo]);
end;

end.
```

One of the key elements of this listing is the class definition, which begins by indicating the parent class. The only other relevant portion is the Register procedure. In fact, you can see that the Component Wizard does very little work.

**WARNING** Starting with Delphi 4, the `Register` procedure *must* be written with an uppercase *R*. This requirement is apparently imposed for C++Builder compatibility (identifiers in C++ are case-sensitive).

**TIP** Use a naming convention when building components. All the components installed in Delphi should have different class names. For this reason most Delphi component developers have chosen to add a two- or three-letter signature prefix to the names of their components. I've done the same, using *Md* (for *Mastering Delphi*) to identify components built in this book. The advantage of this approach is that you can install my `TMdFontCombo` component even if you've already installed a component named `TFontCombo`. Notice that the unit names must also be unique for all the components installed in the system, so I've applied the same prefix to the unit names.

That's all it takes to build a component. Of course, in this example there isn't a lot of code. We need only copy all the system fonts to the `Items` property of the combo box at startup. To accomplish this, we might try to override the `Create` method in the class declaration, adding the statement `Items := Screen.Fonts`. However, this is not the correct approach. The problem is that we cannot access the combo box's `Items` property before the window handle of the component is available; the component cannot have a window handle until its `Parent` property is set; and that property isn't set in the constructor, but later on.

For this reason, instead of assigning the new strings in the `Create` constructor, we must perform this operation in the `CreateWnd` procedure, which is called to create the window control after the component is constructed, its `Parent` property is set, and its window handle is available. Again, we execute the default behavior, and then we can write our custom code. I could have skipped the `Create` constructor and written all the code in `CreateWnd`, but I decided to use both startup methods to demonstrate the difference between them. Here is the declaration of the component class:

```
type
  TMdFontCombo = class (TComboBox)
  private
    FChangeFormFont: Boolean;
    procedure SetChangeFormFont(const Value: Boolean);
  public
    constructor Create (AOwner: TComponent); override;
    procedure CreateWnd; override;
    procedure Change; override;
  published
    property Style default csDropDownList;
    property Items stored False;
```

```
    property ChangeFormFont: Boolean
      read FChangeFormFont write SetChangeFormFont default True;
  end;
```

And here is the source code of its two methods executed at startup:

```
constructor TMdFontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
  FChangeFormFont := True;
end;

procedure TMdFontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items.Assign (Screen.Fonts);

  // grab the default font of the owner form
  if FChangeFormFont and Assigned (Owner) and (Owner is TForm) then
    ItemIndex := Items.IndexOf ((Owner as TForm).Font.Name);
end;
```

Notice that besides giving a new value to the component's `Style` property, in the `Create` method, I've redefined this property by setting a value with the `default` keyword. We have to do both operations because adding the `default` keyword to a property declaration has no direct effect on the property's initial value. Why specify a property's default value then? Because properties that have a value equal to the default are not streamed with the form definition (and they don't appear in the textual description of the form, the DFM file). The `default` keyword tells the streaming code that the component initialization code will set the value of that property.

#### TIP

Why is it important to specify a default value for a published property? To reduce the size of the DFM files and, ultimately, the size of the executable files (which include the DFM files).

The other redefined property, `Items`, is set as a property that should not be saved to the DFM file at all, regardless of the actual value. This is obtained with the `stored` directive followed by the value `False`. The component and its window are going to be created again when the program starts, so it doesn't make any sense to save in the DFM file information that will be discarded later on (to be replaced with the new list of fonts).

#### NOTE

We could have even written the code of the `CreateWnd` method to copy the fonts to the combo box items only at run time. This can be done by using statements such as `if not (csDesigning in ComponentState) then...` But for this first component we are building, the less efficient but more straightforward method described above offers a clearer illustration of the basic procedure.

The third property, `ChangeFormFont`, is not inherited but introduced by the component. It is used to determine whether the current font selection of the combo box should determine the font of the form hosting the component. Again this property is declared with a default value, set in the constructor. The `ChangeFormFont` property is used in the code of the `CreateWnd` method, shown before, to set up the initial selection of the combo depending on the font of the form hosting the component. This is generally the `Owner` of the component, although I could have also walked the `Parent` tree looking for a form component. This code isn't perfect, but the `Assigned` and `is` tests provide some extra safety.

The `ChangeFormFont` property and the same `if` test play a key role in the `Changed` method, which in the base class triggers the `OnChange` event. By overriding this method we provide a default behavior, which can be disabled by toggling the value of the property, but also allow the execution of the `OnChange` event, so that users of this class can fully customize its behavior. The final method, `SetChangeFormFont`, has been modified to refresh the form's font in case the property is being turned on. This is the complete code:

```

procedure TMdFontCombo.Change;
begin
    // assign the font to the owner form
    if FChangeFormFont and Assigned (Owner) and (Owner is TForm) then
        TForm (Owner).Font.Name := Text;
    inherited;
end;

procedure TMdFontCombo.SetChangeFormFont(const Value: Boolean);
begin
    FChangeFormFont := Value;
    // refresh font
    if FChangeFormFont then
        Change;
end;

```

## Creating a Package

Now we have to install the component in the environment, using a package. For this example, we can either create a new package or use an existing one, like the default user's package.

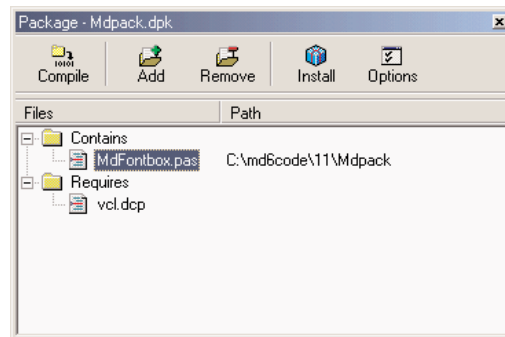
In each case, choose the `Component` > `Install Component` menu command. The resulting dialog box has a page to install the component into an existing package, and a page to create a new package. In this last case, simply type in a filename and a description for the package. Clicking `OK` opens the `Package Editor` (see Figure 11.2), which has two parts:

- The `Contains` list indicates the components included in the package (or, to be more precise, the units defining those components).

- The `Requires` list indicates the packages required by this package. Your package will generally require the `rtl` and `vcl` packages (the main run-time library package and core VCL package), but it might also need the `vcldb` package (which includes most of the database-related classes) if the components of the new package do any database-related operations.

**FIGURE 11.2:**

The Package Editor

**NOTE**

Package names in Delphi 6 aren't version specific any more, even if the compiled packages still have a version number in the filename. See the section "Project and Library Names in Delphi 6" in Chapter 12, "Libraries and Packages," for more details on how this is technically achieved.

If you add the component to the new package we've just defined, and then simply compile the package and install it (using the two corresponding toolbar buttons of the package editor), you'll immediately see the new component show up in the `Md` page of the Component Palette. The `Register` procedure of the component unit file told Delphi where to install the new component. By default, the bitmap used will be the same as the parent class, because we haven't provided a custom bitmap (we will do this in later examples). Notice also that if you move the mouse over the new component, Delphi will display as a hint the name of the class without the initial letter *T*.

## What's Behind a Package?

What is behind the package we've just built? The Package Editor basically generates the source code for the package project: a special kind of DLL built in Delphi. The package project is saved in a file with the `DPK` (for `Delphi PacKage`) extension. A typical package project looks like this:

```
package MdPack;  
  
{ $R *.RES }
```

```

{$ALIGN ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
...
{$DESCRIPTION 'Mastering Delphi Package'}
{$IMPLICITBUILD ON}

```

**requires**

```
vc1;
```

**contains**

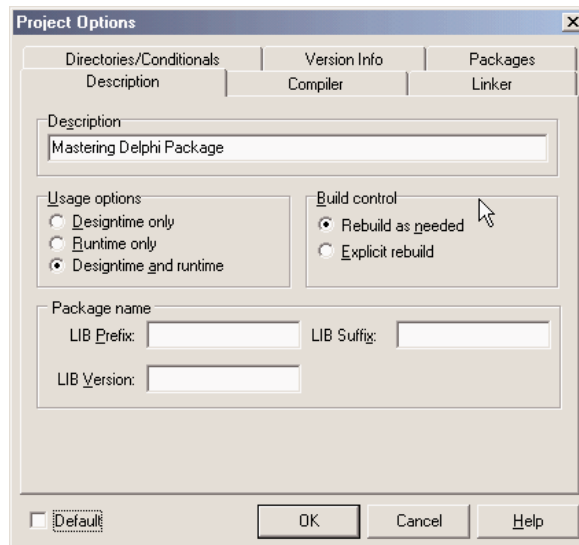
```
MdFontBox in 'MdFontBox.pas';
```

```
end.
```

As you can see, Delphi uses specific language keywords for packages: the first is the package keyword (which is similar to the `library` keyword I'll discuss in the next chapter). This keyword introduces a new package project. Then comes a list with all the compiler options, some of which I've omitted from the listing. Usually the options for a Delphi project are stored in a separate file; packages, by contrast, include all the compiler options directly in their source code. Among the compiler options there is a `DESCRIPTION` compiler directive, used to make the package description available to the Delphi environment. In fact, after you've installed a new package, its description will be shown in the Packages page of the Project Options dialog box, a page you can also activate by selecting the `Component > Install Packages` menu item. This dialog box is shown in Figure 11.3.

**FIGURE 11.3:**

The Project Options for packages. You can see the new package we've just created.



Besides common directives like the DESCRIPTION one, there are other compiler directives specific to packages. The most common of these options are easily accessible through the Options button of the Package Editor. After this list of options come the `requires` and `contains` keywords, which list the items displayed visually in the two pages of the Package Editor. Again, the first is the list of packages required by the current one, and the second is a list of the units installed by this package.

What is the technical effect of building a package? Besides the DPK file with the source code, Delphi generates a BPL file with the dynamic link version of the package and a DCP file with the symbol information. In practice, this DCP file is the sum of the symbol information of the DCU files of the units contained in the package.

At design time, Delphi requires both the BPL and DCP files, because the first has the actual code of the components created on the design form and the symbol information required by the code insight technology. If you link the package dynamically (using it as a run-time package), the DCP file will also be used by the linker, and the BPL file should be shipped along with the main executable file of the application. If you instead link the package statically, the linker refers to the DCU files, and you'll need to distribute only the final executable file.

For this reason, as a component designer, you should generally distribute at least the BPL file, the DCP file, and the DCU files of the units contained in the package and any corresponding DFM files, plus a Help file. As an option, of course, you might also make available the source code files of the package units (the PAS files) and of the package itself (the DPK file).

---

**WARNING**

Delphi, by default, will place all the compiled package files (BPL and DCP) not in the folder of the package source code but under the `\Projects\BPL` folder. This is done so that the IDE can easily locate them, and creates no particular problem. When you have to compile a project using components declared on those packages, though, Delphi might complain that it cannot find the corresponding DCU files, which are stored in the package source code folder. This problem can be solved by indicating the package source code folder in the Library Path (in the Environment Options, which affect all projects) or by indicating it in the Search Path of the current project (in the Project Options). If you choose the first approach, placing different components and packages in a single folder might result in a real time-saver.

## Installing the Components of This Chapter

Having built our first package, we can now start using the component we've added to it. Before we do so, however, I should mention that I've extended the MdPack package to include all of the components we are going to build in this chapter, including different versions of the same component. I suggest you install this package. The best approach is to copy it into a directory of your path, so that it will be available both to the Delphi environment and to the programs you build with it. I've collected all the component source code files and the package definition



in a single subdirectory, called MdPack. This allows the Delphi environment, or a specific project, to refer only to one directory when looking for the DCU files of this package. As suggested in the warning above, I could have collected all of the components presented in the book in a single folder on the companion CD, but I decided that keeping the chapter-based organization was actually more understandable for readers.

Remember, anyway, that if you compile an application using the packages as run-time DLLs, you'll need to install these new libraries on your clients' computers. If you instead compile the programs by statically linking the package, the DLL will be required only by the development environment and not by the users of your applications.

## Using the Font Combo Box

Now you can create a new Delphi program to test the Font combo box. Move to the Component Palette, select the new component, and add it to a new form. A traditional-looking combo box will appear. However, if you open the Items property editor, you'll see a list of the fonts installed on your computer. To build a simple example, I've added a Memo component to the form with some text inside it. By leaving the `ChangeFormFont` property on, you don't need to write any other code to the program, as you'll see in the example. As an alternative I could have turned off the property and handled the `OnChange` event of the component, with code like this:

```
Memo1.Font.Name := MdFontCombo1.Text;
```

The aim of this simple program is only to test the behavior of the new component we have built. The component is still not very useful—we could have added a couple of lines of code to a form to obtain the same effect—but looking at a couple of simple components should help you get an idea of what is involved in component building.

## Creating Compound Components

The next component I want to focus on is a digital clock. This example has some interesting features. First, it embeds a component (a `Timer`) in another component; second, it shows the live-data approach.

### NOTE

The first feature has become even more relevant in Delphi 6, as the Object Inspector of the latest version of Delphi allows you to expose properties of subcomponents directly. As an effect, the example presented in this section has been modified (and simplified) compared to the previous edition of the book. I'll actually mention the differences, when relevant.

Since the digital clock will provide some text output, I considered inheriting from the `TLabel` class. However, this would allow a user to change the label's caption—that is, the text of the clock. To avoid this problem, I simply used the `TCustomLabel` component as the parent class.

A `TCustomLabel` object has the same capabilities as a `TLabel` object, but few published properties. In other words, a `TCustomLabel` subclass can decide which properties should be available and which should remain hidden.

**NOTE**

Most of the Delphi components, particularly the Windows-based ones, have a `TCustomXxx` base class, which implements the entire functionality but exposes only a limited set of properties. Inheriting from these base classes is the standard way to expose only some of the properties of a component in a customized version. In fact, you cannot hide public or published properties of a base class.

With past versions of Delphi, the component had to define a new property, `Active`, wrapping the `Enabled` property of the `Timer`. A *wrapper* property means that the get and set methods of this property read and write the value of the *wrapped* property, which belongs to an internal component (a wrapper property generally has no local data). In this specific case, the code looked like this:

```
function TMdClock.GetActive: Boolean;
begin
    Result := FTimer.Enabled;
end;

procedure TMdClock.SetActive (Value: Boolean);
begin
    FTimer.Enabled := Value;
end;
```

## Publishing Subcomponents in Delphi 6

With Delphi 6 we can simply expose the entire subcomponent, the timer, in a property of its own, that will be regularly expanded by the Object Inspector, allowing a user to set each and every of its subproperties, and even to handle its events.

Here is the full type declaration for the `TMdClock` component, with the subcomponent declared in the private data and exposed as a published property (in the last line):

```
type
    TMdClock = class (TCustomLabel)
    private
        FTimer: TTimer;
    protected
        procedure UpdateClock (Sender: TObject);
    public
        constructor Create (AOwner: TComponent); override;
    published
        property Align;
```

```
    property Alignment;  
    property Color;  
    property Font;  
    property ParentColor;  
    property ParentFont;  
    property ParentShowHint;  
    property PopupMenu;  
    property ShowHint;  
    property Transparent;  
    property Visible;  
    property Timer: TTimer read FTimer;  
end;
```

The Timer property is read-only, as I don't want users to select another value for this component in the Object Inspector (or detach the component by clearing the value of this property). Developing sets of subcomponents that can be used alternately is certainly possible, but adding write support for this property in a safe way is far from trivial (considering that the users of your component might not be very expert Delphi programmers). So I suggest you to stick with read-only properties for subcomponents.

To create the Timer, we must override the constructor of the clock component. The Create method calls the corresponding method of the base class and creates the Timer object, installing a handler for its OnTimer event:

```
    constructor TMDClock.Create (AOwner: TComponent);  
    begin  
        inherited Create (AOwner);  
        // create the internal timer object  
        FTimer := TTimer.Create (Self);  
  
        FTimer.Name := 'ClockTimer';  
        FTimer.OnTimer := UpdateClock;  
        FTimer.Enabled := True;  
        FTimer.SetSubComponent (True);  
    end;
```

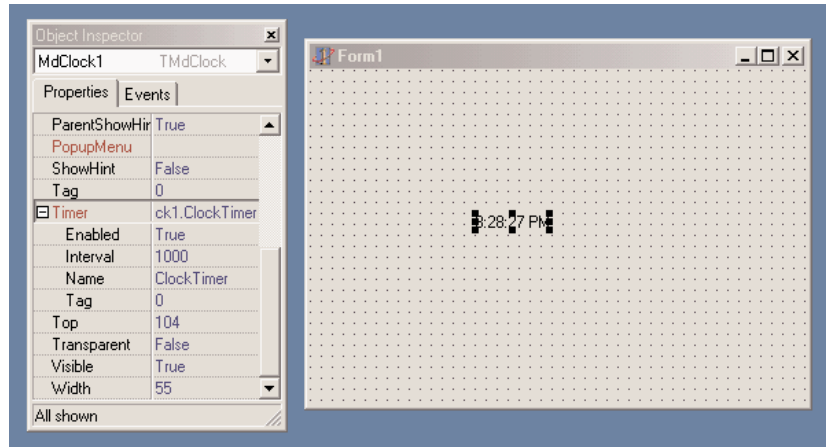
The code gives the component a name, for display in the Object Inspector (see Figure 11.4), and calls the specific SetSubComponent method. We don't need a destructor, simply because the FTimer object has our TMDClock component as owner (as indicated by the parameter of its Create constructor), so it will be destroyed automatically when the clock component is destroyed.

## NOTE

What is the actual effect of the call to the SetSubComponent method in the code above? This call sets an internal flag, saved in the ComponentStyle property set. The flag (csSubComponent) affects the streaming system, allowing the subcomponent and its properties to be saved in the DFM file. In fact, the streaming system by default ignores components that are not owned by the form.

**FIGURE 11.4:**

In Delphi 6, the Object Inspector can automatically expand subcomponents, showing their properties, as in the case of the Timer property of the MdLabel-Clock component.



The key piece of the component's code is the `UpdateClock` procedure, which is just one statement:

```
procedure TMdLabelClock.UpdateClock (Sender: TObject);
begin
    // set the current time as caption
    Caption := TimeToStr (Time);
end;
```

This method uses `Caption`, which is an unpublished property, so that a user of the component cannot modify it in the Object Inspector. The result of this statement is to display the current time. This happens continuously, because the method is connected to the Timer's `OnTimer` event.

## The Component Palette Bitmaps

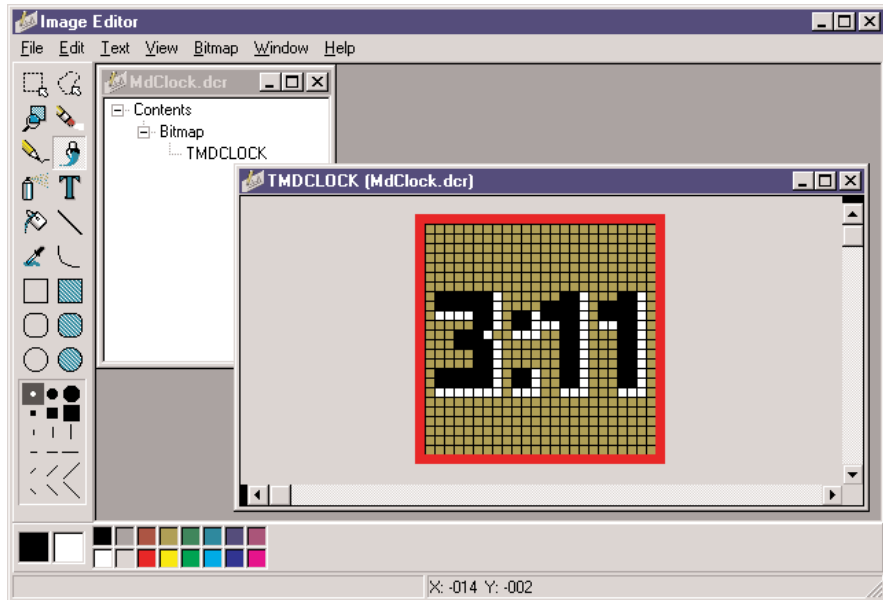
Before installing this second component, we can take one further step: define a bitmap for the Component Palette. If we fail to do so, the Palette uses the bitmap of the parent class, or a default object's bitmap if the parent class is not an installed component (as is the case of the `TCustomLabel`). Defining a new bitmap for the component is easy, once you know the rules. You can create one with the Image Editor (as shown in Figure 11.5), starting a new project and selecting the Delphi Component Resource (DCR) project type.

### TIP

DCR files are simply standard RES files with a different extension. If you prefer, you can create them with any resource editor, including the Borland Resource Workshop, which is certainly a more powerful tool than the Delphi Image editor. When you finish creating the resource file, simply rename the RES file to use a DCR extension.

**FIGURE 11.5:**

The definition of a Component Palette bitmap in Delphi's Image Editor



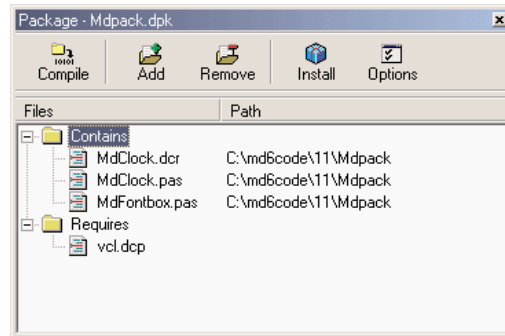
Now we can add a new bitmap to the resource, choosing a size of 24×24 pixels, and we are ready to draw the bitmap. The other important rules refer to naming. In this case, the naming rule is not just a convention; it is a requirement so that the IDE can find the image for a given component class:

- The name of the bitmap resource must match the name of the component, including the initial *T*. In this case, the name of the bitmap resource should be TMDLOCK. The name of the bitmap resource must be uppercase—this is mandatory.
- If you want the Package Editor to recognize and include the resource file, the name of the DCR file must match the name of the compiled unit that defines the component. In this case, the filename should be MdClock.DCR. If you manually include the resource file, via a \$R directive, you can give it the name you like, and also use a RES or DCR file with multiple palette icons.

When the bitmap for the component is ready, you can install the component in Delphi, by using the Package Editor's Install Package toolbar button. After this operation, the Contains section of the editor should list both the PAS file of the component and the corresponding DCR file. In Figure 11.6 you can see all the files (including the DCR files) of the final version of the MdPack package. If the DCR installation doesn't work properly, you can manually add the `{ $R uni tname.dcr }` statement in the package source code.

**FIGURE 11.6:**

The Contains section of the Package Editor shows both the units that are included in the package and the component resource files.



## Building Compound Components with Frames

Instead of building the compound component in code and hooking up the timer event manually, we could have obtained a similar effect by using a frame. Frames make the development of compound components with custom event handlers a visual operation, and thus simpler. You can share this frame by adding it to the Repository or by creating a template using the Add to Palette command of the frame's shortcut menu.

As an alternative, you might want to share the frame by placing it in a package and registering it as a component. Technically, this is not difficult. You add a **Register** procedure to the frame's unit, add the unit to a package, and build it. The new component/frame will be in the Component Palette, like any other component. In Delphi 6, when you place this component/frame on a form, you'll see its subcomponents. You cannot select these subcomponents with a mouse click in the Form Designer, but can do it in the Object TreeView. However, any change you make to these components at design time will inevitably get lost when you run the program or save and reload the form, because the changes to those subcomponents won't be streamed, contrary to what happened with standard frames you place inside a form.

If this is not what you might expect, I've found a *reasonable* way to use frames in packages, demonstrated by the `MdFramedClock` component, part of the examples on the CD for this chapter. The idea is to turn the components owned by the form into actual subcomponents, by calling the new `SetSubComponent` method. As I was up to it, I've also exposed the internal components with properties, even if this isn't compulsory as they can be selected in the Object TreeView anyway. This is the declaration of the component and the code of its methods:

**type**

```
TMdFramedClock = class(TFrame)
  Label1: TLabel;
  Timer1: TTimer;
  Bevel1: TBevel;
```

*Continued on next page*

```

    procedure Timer1Timer(Sender: TObject);
public
    constructor Create(AOwner: TComponent); override;
published
    property SubLabel: TLabel read Label1;
    property SubTimer: TTimer read Timer1;
end;

constructor TMDFramedClock.Create(AOwner: TComponent);
begin
    inherited;
    Timer1.SetSubComponent (true);
    Label1.SetSubComponent (true);
end;

procedure TMDFramedClock.Timer1Timer(Sender: TObject);
begin
    Label1.Caption := TimeToStr (Time);
end;

```

In contrast to the clock component built earlier, there is no need to set up the properties of the timer, or to connect the timer event to its handler function manually, as this is done visually and saved in the DFM file of the frame. Notice also that I haven't exposed the Bevel component—I haven't called `SetSubComponent` on it—so that you can try editing it at design time and see that all the changes get lost, as I mentioned above.

After you install this frame/component, you can use it inside any application. In this particular case, as soon as you drop the frame on the form, the timer will start to update the label with the current time. However, you can still handle its `OnTimer` event, and the Delphi IDE (recognizing that the component is inside a frame) will define a method with this predefined code:

```

procedure TForm1.MdFramedClock1Timer1Timer(Sender: TObject);
begin
    MdFramedClock1.Timer1Timer(Sender);
end;

```

As soon as this timer is connected, even at design time, the live clock will stop, as its original event handler is disconnected. After compiling and running the program, however, the original behavior will be restored (at least if you don't delete the line above) and your extra custom code will be executed as well. This is exactly what you'll expect from frames. You can find a complete demo of the use of this frame/component in the `FrameClock` example.

*Continued on next page*

As a short conclusion of this digression on frames compiled inside packages, I can certainly say that this approach is still far from linear. It is certainly much better than in Delphi 5, where frames inside packages were really unusable. The question is, is it worth the effort? In short, I'd say no. If you work alone or with a small team, it's better to use plain frames stored in the Repository. In larger organizations and for distributing your frames to a larger audience, I bet most people will rather build their components in the traditional way, without trying to use frames. In other words, I'm still hoping that Borland will address more complete support to the visual development of packaged components based on frames.

## A Complex Graphical Component

The graphical component I want to build is an arrow component. You can use such a component to indicate a flow of information, or an action, for example. This component is quite complex, so I'll show you the various steps instead of looking directly at the complete source code. The component I've added to the MdPack package on the CD is only the final version of this process, which will demonstrate several important concepts:

- The definition of new enumerated properties, based on custom enumerated data types.
- The use of properties of `TPersistent`-derived classes, such as `TPen` and `TBrush`, and the issues related to their creation and destruction, and to handling their `OnChange` events internally in our component.
- The implementation of the `Paint` method of the component, which provides its user interface and should be generic enough to accommodate all the possible values of the various properties, including its `Width` and `Height`. The `Paint` method plays a substantial role in this graphical component.
- The definition of a custom event handler for the component, responding to user input (in this case, a double-click on the point of the arrow). This will require direct handling of Windows messages and the use of the Windows API for graphic regions.
- The registration of properties in Object Inspector categories and the definition of a custom category.

### Defining an Enumerated Property

After generating the new component with the Component Wizard and choosing `TGraphicControl` as the parent class, we can start to customize the component. The arrow can point in any of four directions: up, down, left, or right. An enumerated type expresses these choices:

```
type
    TMdArrowDir = (adUp, adRight, adDown, adLeft);
```



This enumerated type defines a private data member of the component, a parameter of the procedure used to change it, and the type of the corresponding property. Two more simple properties are `ArrowHeight` and `Filled`, the first determining the size of the arrowhead and the second whether to fill the arrowhead with color:

```

type
  TMDArrow = class (TGraphicControl)
  private
    fDirection: TMDArrowDir;
    fArrowHeight: Integer;
    fFilled: Boolean;
    procedure SetDirection (Value: TMD4ArrowDir);
    procedure SetArrowHeight (Value: Integer);
    procedure SetFilled (Value: Boolean);
  published
    property Width default 50;
    property Height default 20;
    property Direction: TMD4ArrowDir
      read fDirection write SetDirection default adRight;
    property ArrowHeight: Integer
      read fArrowHeight write SetArrowHeight default 10;
    property Filled: Boolean read fFilled write SetFilled default False;

```

## NOTE

A graphic control has no default size, so when you place it in a form, its size will be a single pixel. For this reason it is important to add a default value for the `Width` and `Height` properties and set the class fields to the default property values in the constructor of the class.

The three custom properties are read directly from the corresponding field and are written using three `Set` methods, all having the same standard structure:

```

procedure TMDArrow.SetDirection (Value: TMDArrowDir);
begin
  if fDirection <> Value then
    begin
      fDirection := Value;
      ComputePoints;
      Invalidate;
    end;
end;

```

Notice that we ask the system to repaint the component (by calling `Invalidate`) only if the property is really changing its value and after calling the `ComputePoints` method, which computes the triangle delimiting the arrowhead. Otherwise, the code is skipped and the method ends immediately. This code structure is very common, and we will use it for most of the `Set` procedures of properties.

We must also remember to set the default values of the properties in the component's constructor:

```
constructor TmdArrow.Create (AOwner: TComponent);  
begin  
    // call the parent constructor  
    inherited Create (AOwner);  
    // set the default values  
    fDirection := adRight;  
    Width := 50;  
    Height := 20;  
    fArrowHeight := 10;  
    fFilled := False;
```

In fact, as mentioned before, the default value specified in the property declaration is used only to determine whether to save the property's value to disk. The `Create` constructor is defined in the public section of the type definition of the new component, and it is indicated by the `override` keyword. It is fundamental to remember this keyword; otherwise, when Delphi creates a new component of this class, it will call the constructor of the base class, rather than the one you've written for your derived class.

## Property-Naming Conventions

In the definition of the `Arrow` component, notice the use of several naming conventions for properties, access methods, and fields. Here is a summary:

- A property should have a meaningful and readable name.
- When a private data field is used to hold the value of a property, the field should be named with an *f* (field) at the beginning, followed by the name of the corresponding property.
- When a function is used to change the value of the property, the function should have the word *Set* at the beginning, followed by the name of the corresponding property.
- A corresponding function used to read the property should have the word *Get* at the beginning, again followed by the property name.

These are just guidelines to make programs more readable. The compiler doesn't enforce them. These conventions are described in the *Delphi Component Writers' Guide* and are followed by the Delphi's class completion mechanism.

## Writing the *Paint* Method

Drawing the arrow in the various directions and with the various styles requires a fair amount of code. To perform custom painting, you override the `Paint` method and use the protected `Canvas` property.

Instead of computing the position of the arrowhead points in drawing code that will be executed often, I've written a separate function to compute the arrowhead area and store it in an array of points defined among the private fields of the component as:

```
fArrowPoints: array [0..3] of TPoint;
```

These points are determined by the `ComputePoints` private method, which is called every time some of the component properties change. Here is an excerpt of its code:

```
procedure TMDArrow.ComputePoints;
var
  XCenter, YCenter: Integer;
begin
  // compute the points of the arrowhead
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  case FDirection of
    adUp: begin
      fArrowPoints [0] := Point (0, FArrowHeight);
      fArrowPoints [1] := Point (XCenter, 0);
      fArrowPoints [2] := Point (Width-1, FArrowHeight);
    end;
  // and so on for the other directions
```

The code computes the center of the component area (simply dividing the `Height` and `Width` properties by two) and then uses it to determine the position of the arrowhead. Besides changing the direction or other properties, we need to refresh the position of the arrowhead when the size of the component changes. What we can do is to override the `SetBounds` method of the component, which is called by VCL every time the `Left`, `Top`, `Width`, and `Height` properties of a component change:

```
procedure TMDArrow.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited SetBounds (ALeft, ATop, AWidth, AHeight);
  ComputePoints;
end;
```

Once the component knows the position of the arrowhead, its painting code becomes simpler. Here is an excerpt of the `Paint` method:

```
procedure TMDArrow.Paint;
var
  XCenter, YCenter: Integer;
```

```
begin
  // compute the center
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;

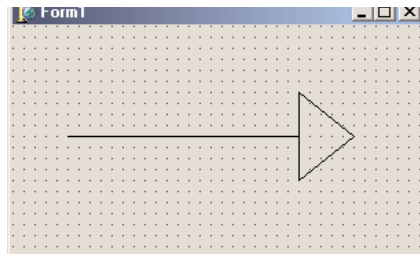
  // draw the arrow line
  case FDirection of
    adUp: begin
      Canvas.MoveTo (XCenter, Height-1);
      Canvas.LineTo (XCenter, FArrowHeight);
    end;
    // and so on for the other directions
  end;

  // draw the arrow point, eventually filling it
  if FFilled then
    Canvas.Polygon (fArrowPoints)
  else
    Canvas.PolyLine (fArrowPoints);
  end;
```

You can see an example of the output of this component in Figure 11.7.

**FIGURE 11.7:**

The output of the Arrow component



## Adding *TPersistent* Properties

To make the output of the component more flexible, I've added to it two new properties, defined with a class type (specifically, a *TPersistent* data type, which defines objects that can be automatically streamed by Delphi). These properties are a little more complex to handle, because the component now has to create and destroy these internal objects (as we did with the internal *Timer* of the clock component). This time, however, we also export the internal objects using some properties, so that users can directly change them from the Object Inspector. To update the component when these subobjects change, we'll also need to handle their

internal `OnChange` property. Here is the definition of the two new `TPersistent`-type properties and the other changes to the definition of the component class:

```

type
  TmdArrow = class (TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    ...
    procedure SetPen (Value: TPen);
    procedure SetBrush (Value: TBrush);
    procedure RepaintRequest (Sender: TObject);
  published
    property Pen: TPen read FPen write SetPen;
    property Brush: TBrush read FBrush write SetBrush;
  end;

```

The first thing to do is to create the objects in the constructor and set their `OnChange` event handler:

```

constructor TmdArrow.Create (AOwner: TComponent);
begin
  ...
  // create the pen and the brush
  FPen := TPen.Create;
  FBrush := TBrush.Create;
  // set a handler for the OnChange event
  FPen.OnChange := RepaintRequest;
  FBrush.OnChange := RepaintRequest;
end;

```

These `OnChange` events are fired when one of the properties of these subobjects changes; all we have to do is to ask the system to repaint our component:

```

procedure TmdArrow.RepaintRequest (Sender: TObject);
begin
  Invalidate;
end;

```

You must also add to the component a destructor, to remove the two graphical objects from memory (and free their system resources):

```

destructor TmdArrow.Destroy;
begin
  FPen.Free;
  FBrush.Free;
  inherited Destroy;
end;

```

The properties related to these two components require some special handling: instead of copying the pointer to the objects, we should copy the internal data of the object passed as parameter. The standard := operation copies the pointers, so in this case we have to use the Assign method instead. Here is one of the two Set procedures:

```
procedure TmdArrow.SetPen (Value: TPen);
begin
  FPen.Assign(Value);
  Invalidate;
end;
```

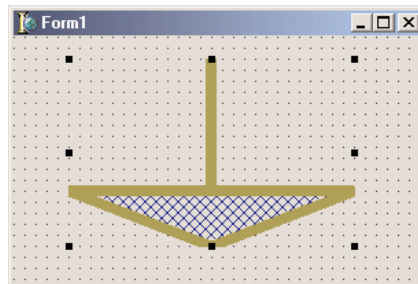
Many TPersistent classes have an Assign method that should be used when we need to update the data of these objects. Now, to actually use the pen and brush for the drawing, you have to modify the Paint method, setting the Pen and the Brush properties of the component Canvas to the value of the internal objects before drawing any line:

```
procedure TmdArrow.Paint;
begin
  // use the current pen and brush
  Canvas.Pen := FPen;
  Canvas.Brush := FBrush;
```

You can see an example of the new output of the component in Figure 11.8.

**FIGURE 11.8:**

The output of the Arrow component with a thick pen and a special hatch brush



## Defining a New Custom Event

To complete the development of the Arrow component, let's add a custom event. Most of the time, new components use the events of their parent classes. For example, in this component, I've made some standard events available simply by redeclaring them in the published section of the class:

```
type
  TmdArrow = class (TGraphicControl)
  published
    property OnClick;
```

```
property OnDragDrop;  
property OnDragOver;  
property OnEndDrag;  
property OnMouseDown;  
property OnMouseMove;  
property OnMouseUp;
```

Thanks to this declaration, the above events (originally declared in a parent class) will now be available in the Object Inspector when the component is installed.

Sometimes, however, a component requires a custom event. To define a brand-new event, you first need to add to the class a field of the type of the event. This type is actually a method pointer type (see Chapter 5 for details). Here is the definition I've added in the private section of the `TMdArrow` class:

```
fArrowDb1Click: TNotifyEvent;
```

In this case I've used the `TNotifyEvent` type, which has only a `Sender` parameter and is used by Delphi for many events, including `OnClick` and `OnDb1Click` events. Using this field I've defined a very simple published property, with direct access to the field:

```
property OnArrowDb1Click: TNotifyEvent  
  read fArrowDb1Click write fArrowDb1Click;
```

Notice again the standard naming convention, with event names starting with *On*. The `fArrowDb1Click` method pointer is activated (executing the corresponding function) inside the specific `ArrowDb1Click` dynamic method. This happens only if an event handler has been specified in the program that uses the component:

```
procedure TMdArrow.ArrowDb1Click;  
begin  
  if Assigned (FArrowDb1Click) then  
    FArrowDb1Click (Self);  
end;
```

This method is defined in the protected section of the type definition to allow future subclasses to both call and change it. Basically, the `ArrowDb1Click` method is called by the handler of the `wm_LButtonDb1Click` Windows message, but only if the double-click took place inside the arrow's point. To test this condition, we can use some of the Windows API's region functions.

## NOTE

A *region* is an area of the screen enclosed by any shape. For example, we can build a polygonal region using the three vertices of the arrow-point triangle. The only problem is that to fill the surface properly, we must define an array of `TPoints` in a clockwise direction (see the description of the `CreatePolygonalRgn` in the Windows API Help for the details of this strange approach). That's what I did in the `ComputePoints` method.

Once we have defined a region, we can test whether the point where the double-click occurred is inside the region by using the `PtInRegion` API call. You can see the complete source code of this procedure in the following listing:

```
procedure TMDArrow.WMLButtonDb1Click (  
    var Msg: TWMLButtonDb1Click); // message wm_LButtonDb1Click;  
var  
    HRegion: HRgn;  
begin  
    // perform default handling  
    inherited;  
  
    // compute the arrowhead region  
    HRegion := CreatePolygonRgn (fArrowPoints, 3, WINDING);  
    try // check whether the click took place in the region  
        if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then  
            ArrowDb1Click;  
    finally  
        DeleteObject (HRegion);  
    end;  
end;
```

## Registering Property Categories

We've added to this component some custom properties and a new event. If you arrange the properties in the Object Inspector by category (a feature available since Delphi 5), all the new elements will show up in the generic Miscellaneous category. Of course, this is far from ideal, but we can easily register the new properties in one of the available categories.

We can register a property (or an event) in a category by calling one of the four overloaded versions of the `RegisterPropertyInCategory` function, defined in the new `DesignIntf` unit. When calling this function, you indicate the name of the category, and you can specify the property name, its type, or the property name and the component it belongs to. For example, we can add the following lines to the `Register` procedure of the unit to register the `OnArrowDb1Click` event in the `Input` category and the `Filled` property in the `Visual` category:

```
uses  
    DesignIntf;  
  
procedure Register;  
begin  
    RegisterComponents('Md', [TMDArrow]);  
    RegisterPropertyInCategory ('Input', TMDArrow, 'OnArrowDb1Click');  
    RegisterPropertyInCategory ('Visual', TMDArrow, 'Filled');  
end;
```



In Delphi 5, the first parameter was a class indicating the category type; now the parameter is simply a string, a much simpler solution. This change also makes it straightforward to define new categories: you simply pass its name as the first parameter of the `RegisterPropertyInCategory` function, as in:

```
RegisterPropertyInCategory ('Arrow', TMDArrow, 'Direction');  
RegisterPropertyInCategory ('Arrow', TMDArrow, 'ArrowHeight');
```

Creating a brand new category for the specific properties of our component can make it much simpler for a user to locate its specific features. Notice, though, that since we rely on the `DesignIntf` unit, you should compile the unit containing these registrations in a design-time package, not a run-time one (in fact, the required `DesignIde` unit cannot be distributed). For this reason, I've written this code in a separate unit than the one defining the component and added the new unit (`MdArrReg`) to the package `MdDesPk`, including all of the design-time-only units; this is discussed later, in the section “Installing the Property Editor.”

---

**WARNING**

It's debatable whether using a category for the specific properties of a component is a good idea. On one side, a user of the component can easily spot specific properties. At the same time, some of the new properties might not pertain to any of the existing categories. On the other side, however, categories can be overused. If every component introduces new categories, users may get confused. You also face the risk of having as many categories as there are properties.

Notice that my code registers the `Filled` property in two different categories. This is not a problem, because the same property can show up multiple times in the Object Inspector under different groups, as you can see in Figure 11.9.

To test the arrow component I've written a very simple example program, `ArrowDemo`, which allows you to modify most of its properties at run time. This type of test, after you have written a component or while you are writing it, is very important.

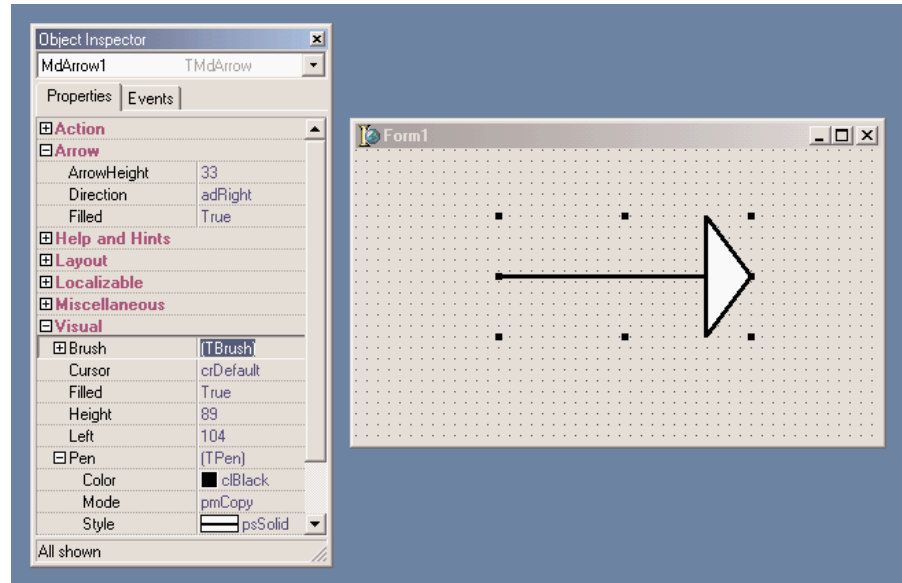
---

**NOTE**

The *Localizable* property category has a special role, related to the use of the ITE (Integrated Translation Environment). When a property is part of this category, its value will be listed in the ITE as a property that can be translated into another language. (A complete discussion of the ITE is beyond the scope of this book.)

**FIGURE 11.9:**

The Arrow component defines a custom property category, *Arrow*, as you can see in the Object Inspector.



## Customizing Windows Controls

One of the most common ways of customizing existing components is to add some predefined behavior to their event handlers. Every time you need to attach the same event handler to components of different forms, you should consider adding the code of the event right into a subclass of the component. An obvious example is that of edit boxes accepting only numeric input. Instead of attaching to each of them a common `OnChar` event handler, we can define a simple new component. This component, however, won't handle the event; events are for component users only. Instead, the component can either handle the Windows message directly or override a method, as described in the next two sections.

### Overriding Message Handlers: The Numeric Edit Box

To customize an edit box component to restrict the input it will accept, all you need to do is handle the `wm_Char` Windows messages that occur when the user presses any but a few specific keys (namely, the numeric characters).

One way to respond to a message for a given window (whether it's a form or a component) is to create a new *message-response* method that you declare using the `message` keyword. Delphi's message-handling system makes sure that your message-response method has a chance to

respond to a given message before the form or component's default message handler does. You'll see in the next section that, instead of creating a new method (as we do here), you can override an existing virtual method that responds to a given message. Below is the code of the `TMdNumEdit` class:

```

type
  TMdNumEdit = class (TCustomEdit)
  private
    fInputError: TNotifyEvent;
  protected
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
  public
    procedure WmChar (var Msg: TWmChar); message wm_Char;
    constructor Create (Owner: TComponent); override;
  published
    property OnInputError: TNotifyEvent read fInputError write fInputError;
    property Value: Integer read GetValue write SetValue default 0;
    property AutoSelect;
    property AutoSize;
    property BorderStyle;
    // and so on...

```

This component inherits from `TCustomEdit` instead of `TEdit` so that it can hide the `Text` property and surface the `Integer Value` property instead. Notice that I don't create a new field to store this value, because we can use the existing (but now unpublished) `Text` property. To do this, we'll simply convert the numeric value to and from a text string. The `TCustomEdit` class (or actually the Windows control it wraps) automatically paints the information from the `Text` property on the surface of the component:

```

function TMdNumEdit.GetValue: Integer;
begin
  // set to 0 in case of error
  Result := StrToIntDef (Text, 0);
end;

procedure TMdNumEdit.SetValue (Value: Integer);
begin
  Text := IntToStr (Value);
end;

```

The most important method is the response for the `wm_Char` message. In the body of this method, the component filters out all the nonnumeric characters and raises a specific event in case of an error:

```

procedure TMdNumEdit.WmChar (var Msg: TWmChar);
begin

```

```
if not (Char (Msg.CharCode) in ['0'..'9']) and not (Msg.CharCode = 8) then
begin
  if Assigned (fInputError) then
    fInputError (Self);
  end
  else
    inherited;
end;
```

This method checks each character as the user enters it, testing for numerals and the Backspace key (which has an ASCII value of 8). The user should be able to use Backspace in addition to the system keys (the arrow keys and Del), so we need to check for that value. We don't have to check for the system keys, because they are surfaced by a different Windows message, `wm_SysChar`.

That's it. Now if you place this component on a form, you can type something in the edit box and see how it behaves. You might also want to attach a method to the `OnInputError` event to provide feedback to the user when a wrong key is typed.

## A Numeric Edit with Thousands Separators

As a further extension to the example, when typing large numbers it would be nice for the thousands separators to automatically appear and update themselves as required by the user input. You can do this by overriding the internal `Change` method and formatting the number properly. There are only a couple of small problems to consider. The first is that to format the number you need to have one, but the text of the edit with the thousands separators (possibly misplaced) cannot be converted to a number directly. I've written a modified version of the `StringToFloat` function, called `StringToFloatSkipping`, to accomplish this.

The second small problem is that if you modify the text of the edit box the current position of the cursor will get lost. So you need to save the original cursor position, reformat the number, and then reapply the cursor position considering that if a separator has been added or removed, it should change accordingly. All these considerations are summarized by the following complete code of the `TmDThousandEdit` class:

```
type
  TmDThousandEdit = class (TmDNumEdit)
  public
    procedure Change; override;
  end;

function StringToFloatSkipping (s: string): Extended;
var
  s1: string;
  I: Integer;
```

```

begin
    // remove non-numbers, but keep the decimal separator
    s1 := '';
    for i := 1 to length (s) do
        if s[i] in ['0'..'9'] then
            s1 := s1 + s[i];
        Result := StrToFloat (s1);
    end;

procedure TMdThousandEdit.Change;
var
    CursorPos, // original position of the cursor
    LengthDiff: Integer; // number of new separators (+ or -)
begin
    if Assigned (Parent) then
        begin
            CursorPos := SelStart;
            LengthDiff := Length (Text);
            Text := FormatFloat ('#,###',
                StringToFloatSkipping (Text));
            LengthDiff := Length (Text) - LengthDiff;
            // move the cursor to the proper position
            SelStart := CursorPos + LengthDiff;
        end;
    inherited;
end;
end;

```

## Overriding Dynamic Methods: The Sound Button

Our next component, `TMdSoundButton`, plays one sound when you press the button and another sound when you release it. The user specifies each sound by modifying two `String` properties that name the appropriate WAV files for the respective sounds. Once again, we need to intercept and modify some system messages (`wm_LButtonDown` and `wm_LButtonUp`), but instead of handling the messages by writing a new message-response method, we'll override the appropriate *second-level* handlers.

### NOTE

When most VCL components handle a Windows message, they call a *second-level* message handler (usually a dynamic method), instead of executing code directly in the message-response method. This makes it simpler for you to customize the component in a derived class. Typically, a second-level handler will do its own work and then call any event handler that the component user has assigned.

Here is the code of the `TMdSoundButton` class, with the two protected methods that override the second-level handlers, and the two string properties that identify the sound files.

You'll notice that in the property declarations, we read and write the corresponding private fields without calling a get or set method, simply because we don't need to do anything special when the user makes changes to those properties.

```
type
  TMdSoundButton = class(TButton)
private
  FSoundUp, FSoundDown: string;
protected
  procedure MouseDown(Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer); override;
  procedure MouseUp(Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer); override;
published
  property SoundUp: string read FSoundUp write FSoundUp;
  property SoundDown: string read FSoundDown write FSoundDown;
end;
```

There are several reasons why overriding existing second-level handlers is generally a better approach than handling straight Windows messages. First, this technique is more sound from an object-oriented perspective. Instead of duplicating the message-response code from the base class and then customizing it, you're overriding a virtual method call that the VCL designers planned for you to override. Second, if someone needs to derive another class from one of your component classes, you'll want to make it as easy for them to customize as possible, and overriding second-level handlers is less likely to induce strange errors (if only because you're writing less code). Finally, this will make your component classes more consistent with VCL—and therefore easier for someone else to figure out. Here is the code of the two second-level handlers:

```
uses
  MMSystem;

procedure TMdSoundButton.MouseDown(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
  PlaySound (PChar (FSoundDown), 0, snd_Async);
end;

procedure TMdSoundButton.MouseUp(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseUp (Button, Shift, X, Y);
  PlaySound (PChar (FSoundUp), 0, snd_Async);
end;
```

In both cases, you'll notice that we call the inherited version of the methods *before* we do anything else. For most second-level handlers, this is a good practice, since it ensures that we execute the standard behavior before we execute any custom behavior.

Next, you'll notice that we call the `PlaySound Win32` API function to play the sound. You can use this function (which is defined in the `MmSystem` unit to play either WAV files or system sounds, as the `SoundB` example demonstrates. Here is a textual description of the form of this sample program (from the DFM file):

```
object MdSoundButton1: TMdSoundButton
  Caption = 'Press'
  SoundUp = 'RestoreUp'
  SoundDown = 'RestoreDown'
end
```

**NOTE**

Selecting a proper value for these sound properties is far from simple. Later in this chapter, I'll show you how to add a property editor to the component to simplify the operation.

## Handling Internal Messages: The Active Button

The Windows interface is evolving toward a new standard, including components that become highlighted as the mouse cursor moves over them. Delphi provides similar support in many of its built-in components, but what does it take to mimic this behavior for a simple button? This might seem a complex task to accomplish, but it is not.

The development of a component can become much simpler once you know which virtual function to override or which message to hook onto. The next component, the `TMdActiveButton` class, demonstrates this by handling some internal Delphi messages to accomplish its task in a very simple way. (For information about where these internal Delphi messages come from, see the sidebar “Component Messages and Notifications.”)

The `ActiveButton` component handles the `cm_MouseEnter` and `cm_MouseExit` internal Delphi messages, which are received when the mouse cursor enters or leaves the area corresponding to the component:

```
type
  TMdActiveButton = class (TButton)
  protected
    procedure MouseEnter (var Msg: TMessage);
      message cm_mouseEnter;
    procedure MouseLeave (var Msg: TMessage);
      message cm_mouseLeave;
  end;
```

The code you write for these two methods can do whatever you want. For this example, I've decided to simply toggle the bold style of the font of the button itself. You can see the effect of moving the mouse over one of these components in Figure 11.10.

```
procedure TMdActiveButton.MouseEnter (var Msg: TMessage);  
begin  
    Font.Style := Font.Style + [fsBold];  
end;  
  
procedure TMdActiveButton.MouseLeave (var Msg: TMessage);  
begin  
    Font.Style := Font.Style - [fsBold];  
end;
```

**FIGURE 11.10:**

An example of the use of the ActiveButton component



You can add other effects at will, including enlarging the font itself, making the button the default, or increasing its size a little. The best effects usually involve colors, but you should inherit from the `TBitBtn` class to have this support (`TButton` controls have a fixed color).

## Component Messages and Notifications

To build the `ActiveButton` component, I've used two internal Delphi component messages, as indicated by their *cm* prefix. These messages can be quite interesting, as the example highlights, but they are almost completely undocumented by Borland. There is also a second group of internal Delphi messages, indicated as component notifications and distinguished by their *cn* prefix. I don't have enough space here to discuss each of them or provide a detailed analysis; browse the VCL source code if you want to learn more.



**WARNING** As this is a rather advanced topic, feel free to skip this section if you are new to writing Delphi components. But component messages are not documented in the Delphi help file, so I felt it was important to at least *list* them here.

## Component Messages

A Delphi component passes *component messages* to other components to indicate any change in its state that might affect those components. Most of these messages start as Windows messages, but some of them are more complex, higher-level translations and not simple remappings. Also, components send their own messages as well as forwarding those received from Windows. For example, changing a property value or some other characteristic of the component may necessitate telling one or more other components about the change.

We can group these messages into categories:

- Activation and input focus messages are sent to the component being activated or deactivated, receiving or losing the input focus:

<code>cm_Activate</code>	Corresponds to the <code>OnActivate</code> event of forms and of the application
<code>cm_Deactivate</code>	Corresponds to <code>OnDeactivate</code>
<code>cm_Enter</code>	Corresponds to <code>OnEnter</code>
<code>cm_Exit</code>	Corresponds to <code>OnExit</code>
<code>cm_FocusChanged</code>	Sent whenever the focus changes between components of the same form (later, we'll see an example using this message)
<code>cm_GotFocus</code>	Declared but not used
<code>cm_LostFocus</code>	Declared but not used

- Messages sent to child components when a property changes:

<code>cm_BiDiModeChanged</code>	<code>cm_IconChanged</code>
<code>cm_BorderChanged</code>	<code>cm_ShowHintChanged</code>
<code>cm_ColorChanged</code>	<code>cm_ShowingChanged</code>
<code>cm_Ct13DChanged</code>	<code>cm_SysFontChanged</code>
<code>cm_CursorChanged</code>	<code>cm_TabStopChanged</code>
<code>cm_EnabledChanged</code>	<code>cm_TextChanged</code>
<code>cm_FontChanged</code>	<code>cm_VisibleChanged</code>

Monitoring these messages can help track changes in a property. You might need to respond to these messages in a new component, but it's not likely.

- Messages related to *ParentXxx* properties: `cm_ParentFontChanged`, `cm_ParentColorChanged`, `cm_ParentCtl3DChanged`, `cm_ParentBiDiModeChanged`, and `cm_ParentShowHintChanged`. These are very similar to the messages of the previous group.
- Notifications of changes in the Windows system: `cm_SysColorChange`, `cm_WinIniChange`, `cm_TimeChange`, and `cm_FontChange`. Handling these messages is useful only in special components that need to keep track of system colors or fonts.
- Mouse messages: `cm_Drag` is sent many times during dragging operations. `cm_MouseEnter` and `cm_MouseLeave` are sent to the control when the cursor enters or leaves its surface, but these are sent by the `Application` object as low-priority messages. `cm_MouseWheel` corresponds to wheel-based operations.

`cm_Drag` has a `DragMessage` parameter that indicates a sort of submessage, and the address of the `TDragRec` record that indicates the mouse position and the components involved in the dragging operation. The `cm_Drag` message isn't that important, because Delphi defines many drag events and drag methods you can override. However, you can respond to `cm_Drag` for a few things that don't generate an event or method call. This message is sent to find the target component (when the `DragMessage` field is `dmFindTarget`); to indicate that the cursor has reached a component (the `dmDragEnter` submessage), is being moved over it (`dmDragMove`), or has left it (`dmDragLeave`); when the drop operation is accepted (`dmDragDrop`); and when it is aborted (`dmDragCancel`).

- Application messages:

<code>cm_AppKeyDown</code>	Sent to the <code>Application</code> object to let it determine whether a key corresponds to a menu shortcut
<code>cm_AppSysCommand</code>	Corresponds to the <code>wm_SysCommand</code> message
<code>cm_DialogHandle</code>	Sent in a DLL to retrieve the value of the <code>DialogHandle</code> property (used by some dialog boxes not built with Delphi)
<code>cm_InvokeHelp</code>	Sent by code in a DLL to call the <code>InvokeHelp</code> method
<code>cm_WindowHook</code>	Sent in a DLL to call the <code>HookMainWindow</code> and <code>UnhookMainWindow</code> methods

You'll rarely need to use these messages yourself. There is also a `cm_HintShowPause` message, which is apparently never handled in VCL.

- Delphi internal messages:

<code>cm_CancelMode</code>	Terminates special operations, such as showing the pull-down list of a combo box
<code>cm_ControlChange</code>	Sent to each control before adding or removing a child control (handled by some common controls)
<code>cm_ControlListChange</code>	Sent to each control before adding or removing a child control (handled by the <code>DBCtrGrid</code> component)
<code>cm_DesignHitTest</code>	Determines whether a mouse operation should go to the component or to the form designer
<code>cm_HintShow</code>	Sent to a control just before displaying its hint (only if the <code>ShowHint</code> property is <code>True</code> )
<code>cm_HitTest</code>	Sent to a control when a parent control is trying to locate a child control at a given mouse position (if any)
<code>cm_MenuChanged</code>	Sent after MDI or OLE menu-merging operations
- Messages related to special keys:

<code>cm_ChildKey</code>	Sent to the parent control to handle some special keys (in Delphi, this message is handled only by <code>DBCtrGrid</code> components)
<code>cm_DialogChar</code>	Sent to a control to determine whether a given input key is its accelerator character
<code>cm_DialogKey</code>	Handled by modal forms and controls that need to perform special actions
<code>cm_IsShortCut</code>	I haven't yet figured out the exact role of this new message.
<code>cm_WantSpecialKey</code>	Handled by controls that interpret special keys in an unusual way (for example, using the <code>Tab</code> key for navigation, as some <code>Grid</code> components do)

- Messages for specific components:

<code>cm_GetDataLink</code>	Used by <code>DBCtrGrid</code> controls (and discussed in Chapter 18)
<code>cm_TabFontChanged</code>	Used by the <code>TabbedNotebook</code> components
<code>cm_ButtonPressed</code>	Used by <code>SpeedButtons</code> to notify other sibling <code>SpeedButton</code> components (to enforce radio-button behavior)
<code>cm_DeferLayout</code>	Used by <code>DBGrid</code> components
- OLE container messages: `cm_DocWindowActivate`, `cm_IsToolControl`, `cm_Release`, `cm_UIActivate`, and `cm_UIDeactivate`.
- Dock-related messages, including `cm_DockClient`, `cm_DockNotification`, `cmFloat`, and `cm_UndockClient`.
- Method-implementation messages, such as `cm_RecreateWnd`, called inside the `RecreateWnd` method of `TControl`; `cm_Invalid`, called inside `TControl.Invalid`; `cm_Changed`, called inside `TControl.Changed`; and `cm_AllChildrenFlipped`, called in the `DoFlipChildren` methods of `TWinControl` and `TScrollingWinControl`. In the similar group fall two action list-related messages, `cm_ActionUpdate` and `cm_ActionExecute`.

Finally, there are messages defined and handled by specific components and declared in the respective units, such as `cm_DeferLayout` for `DBGrid` controls and a group of almost 10 messages for action bar components.

## Component Notifications

Component notification messages are those sent from a parent form or component to its children. These notifications correspond to messages sent by Windows to the parent control's window, but logically intended for the control. For example, interaction with controls such as buttons, edit, or list boxes, causes Windows to send a `wm_Command` message to the parent of the control. When a Delphi program receives these messages, it forwards the message to the control itself, as a notification. The Delphi control can handle the message and eventually fire an event. Similar dispatching operations take place for many other commands.

The connection between Windows messages and component notification ones is so tight that you'll often recognize the name of the Windows message from the name of the notification message, simply replacing the initial `cn` with `wm`. There are several distinct groups of component notification messages:

- General keyboard messages: `cn_Char`, `cn_KeyUp`, `cn_KeyDown`, `cn_SysChar`, and `cn_SysKeyDown`.

- Special keyboard messages used only by list boxes with the `lbs_WantKeyboardInput` style: `cn_CharToItem` and `cn_VKeyToItem`.
- Messages related to the owner-draw technique: `cn_CompareItem`, `cn_DeleteItem`, `cn_DrawItem`, and `cn_MeasureItem`.
- Messages for scrolling, used only by scroll bar and track bar controls: `cn_HScroll` and `cn_VScroll`.
- General notification messages, used by most controls: `cn_Command`, `cn_Notify`, and `cn_ParentNotify`.
- Control color messages: `cn_CtlColorBtn`, `cn_CtlColorDlg`, `cn_CtlColorEdit`, `cn_CtlColorListBox`, `cn_CtlColorMsgbox`, `cn_CtlColorScrollbar`, and `cn_CtlColorStatic`.

Some more control notifications are defined for common controls support (in the `ComCtrls` unit).

## An Example of Component Messages

As a very simple example of the use of some component messages, I've written the `CMNTest` program. This program has a form with three edit boxes, and associated labels. The first message it handles, `cm_DialogKey`, allows it to treat the Enter key as if it were a Tab key. The code of this method checks for the Enter key's code and sends the same message, but passes the `vk_Tab` key code. To halt further processing of the Enter key, we set the result of the message to 1:

```
procedure TForm1.CMDialogKey(var Message: TCMDialogKey);
begin
  if (Message.CharCode = VK_RETURN) then
    begin
      Perform (CM_DialogKey, VK_TAB, 0);
      Message.Result := 1;
    end
  else
    inherited;
  end;
```

The second message, `cm_DialogChar`, monitors accelerator keys. This can be useful to provide custom shortcuts without defining an extra menu for them. In this case, I'm simply logging the special keys in a label:

```
procedure TForm1.CMDialogChar(var Msg: TCMDialogChar);
begin
  Label1.Caption := Label1.Caption + Char (Msg.CharCode);
  inherited;
end;
```

Finally, the form handles the `cm_FocusChanged` message, to respond to focus changes without having to handle the `OnEnter` event of each of its components. Again, the simple action is to display a description of the focused component:

```
procedure TForm1.CmFocusChanged(var Msg: TCMFocusChanged);  
begin  
    Label15.Caption := 'Focus on ' + Msg.Sender.Name;  
end;
```

The advantage of this approach is that it works independently of the type and number of components you add to the form, and it does so without any special action on your part. Again, this is a trivial example for such an advanced topic, but if you add to this the code of the `ActiveButton` component, you have at least a few reasons to look into these special, undocumented messages. At times, writing the same code without their support can become extremely complex.

## A Nonvisual Dialog Component

The next component we'll examine is completely different from the ones we have seen up to now. After building window-based controls and simple graphic components, I'm now going to build a nonvisual component.

The basic idea is that forms are components. When you have built a form that might be particularly useful in multiple projects, you can add it to the Object Repository or make a component out of it. The second approach is more complex than the first, but it makes using the new form easier and allows you to distribute the form without its source code. As an example, I'll build a component based on a custom dialog box, trying to mimic as much as possible the behavior of standard Delphi dialog box components.

The first step in building a dialog box in a component is to write the code of the dialog box itself, using the standard Delphi approach. Just define a new form and work on it as usual. When a component is based on a form, you can almost visually design the component. Of course, once the dialog box has been built, you have to define a component around it in a nonvisual way.

The standard dialog box I want to build is based on a list box, because it is common to let a user choose a value from a list of strings. I've customized this common behavior in a dialog box and then used it to build a component. The simple `ListBoxForm` form I've built has a list box and the typical OK and Cancel buttons, as shown in its textual description:

```
object MdListBoxForm: TMDListBoxForm  
    BorderStyle = bsDialog  
    Caption = 'ListBoxForm'  
object ListBox1: TListBox
```

```

        OnDb1Click = ListBox1Db1Click
    end
    object BitBtn1: TBitBtn
        Kind = bkOK
    end
    object BitBtn2: TBitBtn
        Kind = bkCancel
    end
end

```

The only method of this dialog box form relates to the double-click event of the list box, which closes the dialog box as though the user clicked the OK button, by setting the `ModalResult` property of the form to `mrOk`. Once the form works, we can start changing its source code, adding the definition of a component and removing the declaration of the global variable for the form.

## NOTE

For components based on a form, you can use two Pascal source code files: one for the form and the other for the component encapsulating it. It is also possible to place both the component and the form in a single unit, as I've done for this example. In theory it would be even nicer to declare the form class in the implementation portion of this unit, hiding it from the users of the component. In practice this is not a good idea. To manipulate the form visually in the Form Designer, the form class declaration must appear in the interface section of the unit. The rationale behind this behavior of the Delphi IDE is that, among other things, this constraint minimizes the amount of code the module manager has to scan to find the form declaration—an operation that must be performed often to maintain the synchronization of the visual form with the form class definition.

The most important of these operations is the definition of the `TMdListBoxDialog` component. This component is defined as “nonvisual” because its immediate ancestor class is `TComponent`. The component has one public property and these three published properties:

- `Lines` is a `TStrings` object, which is accessed via two methods, `GetLines` and `SetLines`. This second method uses the `Assign` procedure to copy the new values to the private field corresponding to this property. This internal object is initialized in the `Create` constructor and destroyed in the `Destroy` method.
- `Selected` is an integer that directly accesses the corresponding private field. It stores the selected element of the list of strings.
- `Title` is a string used to change the title of the dialog box.

The public property is `SelectedItem`, a read-only property that automatically retrieves the selected element of the list of strings. Notice that this property has no storage and no data: it simply accesses other properties, providing a virtual representation of data:

```

type
    TMdListBoxDialog = class (TComponent)

```

```

private
  FLines: TStrings;
  FSelected: Integer;
  FTitle: string;
  function GetSelItem: string;
  procedure SetLines (Value: TStrings);
  function GetLines: TStrings;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  function Execute: Boolean;
  property SelItem: string read GetSelItem;
published
  property Lines: TStrings read GetLines write SetLines;
  property Selected: Integer read FSelected write FSelected;
  property Title: string read FTitle write FTitle;
end;

```

Most of the code of this example is in the Execute method, a function that returns True or False depending on the modal result of the dialog box. This is consistent with the Execute method of most standard Delphi dialog box components. The Execute function creates the form dynamically, sets some of its values using the component's properties, shows the dialog box, and if the result is correct, updates the current selection:

```

function TMDListBoxDialog.Execute: Boolean;
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create ('No items in the list');
  ListBoxForm := TListBoxForm.Create (Self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
      begin
        Result := True;
        Selected := ListBoxForm.ListBox1.ItemIndex;
      end
    else
      Result := False;
    finally
      ListBoxForm.Free;
    end;
end;

```



Notice that the code is contained within a `try/finally` block, so if a run-time error occurs when the dialog box is displayed, the form will be destroyed anyway. I've also used exceptions to raise an error if the list is empty when a user runs it. This error is by design, and using an exception is a good technique to enforce it. The other methods of the component are quite straightforward. The constructor creates the `FLines` string list, which is deleted by the destructor; the `GetLines` and `SetLines` methods operate on the string list as a whole; and the `GetSelectedItem` function (listed below) returns the text of the selected item:

```
function TMdListBoxDialog.GetSelectedItem: string;
begin
    if (Selected >= 0) and (Selected < FLines.Count) then
        Result := FLines [Selected]
    else
        Result := '';
end;
```

Of course, since we are manually writing the code of the component and adding it to the source code of the original form, we have to remember to write the `Register` procedure.

## Using the Nonvisual Component

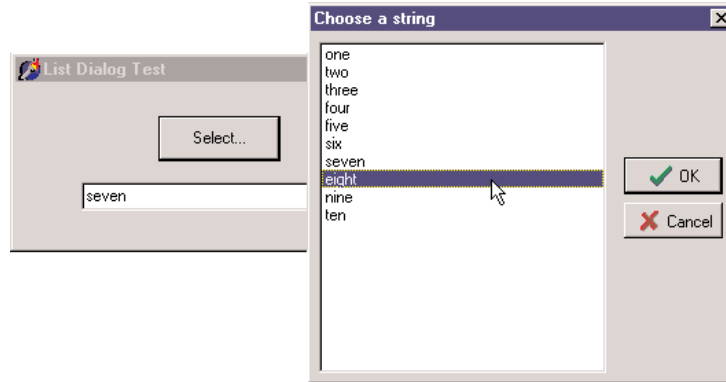
Once you've done that and the component is ready, you must provide a bitmap. For nonvisual components, bitmaps are very important because they are used not only for the Component Palette, but also when you place the component on a form. After preparing the bitmap and installing the component, I've written a simple project to test it. The form of this test program has a button, an edit box, and the `MdListDialog` component. In the program, I've added only a few lines of code, corresponding to the `OnClick` event of the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // select the text of the edit, if corresponding to one of the strings
    MdListDialog1.Selected := MdListDialog1.Lines.IndexOf (Edit1.Text);
    // run the dialog and get the result
    if MdListDialog1.Execute then
        Edit1.Text := MdListDialog1.SelectedItem;
end;
```

That's all you need to run the dialog box we have placed in the component, as you can see in Figure 11.11. As you've seen, this is an interesting approach to the development of some common dialog boxes.

**FIGURE 11.11:**

The ListDialogDemo example shows the dialog box I've encapsulated in the ListDialog component.



## Defining Custom Actions

Besides defining custom components, you can define and register new standard actions, which will be made available in the Action Editor of the Action List component. Creating new actions is not complex. You have to inherit from the `TAction` class and override some of the methods of the base class.

There are basically three methods to override. The `HandlesTarget` function returns whether the action object wants to handle the operation for the current target, which is by default the control with the focus. The `UpdateTarget` procedure can set the user interface of the controls connected with the action, eventually disabling the action if the operation is currently not available. Finally, you can implement the `ExecuteTarget` method to determine the actual code to execute, so that the user can simply select the action and doesn't have to implement it.

To show you this approach in practice, I've implemented the three cut, copy, and paste actions for a list box, in a way similar to what VCL does for an edit box (although I've actually simplified the code a little). I've written a base class, which inherits from the generic `TListControlAction` class of the new `ExtActns` unit. This base class, `TMdCustomListAction`, adds some common code, shared by all the specific actions, and publishes a few action properties. The three derived classes have their own `ExecuteTarget` code, plus little more. Here are the four classes:

```

type
  TMdCustomListAction = class (TListControlAction)
  protected
    function TargetList (Target: TObject): TCustomListBox;
    function GetControl (Target: TObject): TCustomListControl;
  public

```

```

    procedure UpdateTarget (Target: TObject); override;
published
    property Caption;
    property Enabled;
    property HelpContext;
    property Hint;
    property ImageIndex;
    property ListControl;
    property ShortCut;
    property SecondaryShortCuts;
    property Visible;
    property OnHint;
end;

TMdListCutAction = class (TMdCustomListAction)
public
    procedure ExecuteTarget(Target: TObject); override;
end;

TMdListCopyAction = class (TMdCustomListAction)
public
    procedure ExecuteTarget(Target: TObject); override;
end;

TMdListPasteAction = class (TMdCustomListAction)
public
    procedure UpdateTarget (Target: TObject); override;
    procedure ExecuteTarget (Target: TObject); override;
end;

```

The `HandlesTarget` method, one of the three key methods of actions, is provided by the `TListControlAction` class, with this code:

```

function TListControlAction.HandlesTarget(Target: TObject): Boolean;
begin
    Result := ((ListControl <> nil) or
        (ListControl = nil) and (Target is TCustomListControl)) and
        TCustomListControl(Target).Focused;
end;

```

The `UpdateTarget` method, instead, has two different implementations. The default one is provided by the base class and used by the copy and cut actions. These actions are enabled only if the target list box has at least one item and an item is currently selected. The status of the paste action depends instead on the Clipboard status:

```

procedure TMdCustomListAction.UpdateTarget (Target: TObject);
begin
    Enabled := (TargetList (Target).Items.Count > 0)

```

```
        and (TargetList (Target).ItemIndex >= 0);
    end;

    function TMdCustomListAction.TargetList (Target: TObject): TCustomListBox;
    begin
        Result := GetControl (Target) as TCustomListBox;
    end;

    function TMdCustomListAction.GetControl(Target: TObject): TCustomListControl;
    begin
        Result := Target as TCustomListControl;
    end;

    procedure TMdListPasteAction.UpdateTarget (Target: TObject);
    begin
        Enabled := Clipboard.HasFormat (CF_TEXT);
    end;
```

The `TargetList` function uses the `GetControl` function of the `TListControlAction` class, which returns either the list box connected to the action at design time or the target control, the list box control with the input focus.

Finally, the three `ExecuteTarget` methods simply perform the corresponding actions on the target list box:

```
    procedure TMdListCopyAction.ExecuteTarget (Target: TObject);
    begin
        with TargetList (Target) do
            Clipboard.AsText := Items [ItemIndex];
        end;

    procedure TMdListCutAction.ExecuteTarget(Target: TObject);
    begin
        with TargetList (Target) do
            begin
                Clipboard.AsText := Items [ItemIndex];
                Items.Delete (ItemIndex);
            end;
        end;

    procedure TMdListPasteAction.ExecuteTarget(Target: TObject);
    begin
        (TargetList (Target)).Items.Add (Clipboard.AsText);
    end;
```

Once you've written this code in a unit and added it to a package (in this case, the `MdPack` package), the final step is to register the new custom actions in a given category. This is indicated as the first parameter of the `RegisterActions` procedure, while the second is the list of action classes to register:

```
procedure Register;  
begin  
  RegisterActions ('List',  
    [TMDListCutAction, TMDListCopyAction, TMDListPasteAction], nil);  
end;
```

To test the use of these three custom actions, I've written the `ListTest` example on the companion CD. This program has two list boxes plus a toolbar that contains three buttons connected with the three custom actions and an edit box for entering new values. The program allows a user to cut, copy, and paste list box items. Nothing special, you might think, but the strange fact is that the program has absolutely no code!

## Writing Property Editors

Writing components is certainly an effective way to customize Delphi, helping developers to build applications faster without requiring a detailed knowledge of low-level techniques. The Delphi environment is also quite open to extensions. In particular, you can extend the Object Inspector by writing custom property editors and to extend the Form Designer by adding component editors.

### NOTE

Along with these techniques, Delphi offers some internal interfaces to add-on tool developers. Using these interfaces, known as OpenTools API, requires an advanced understanding of how the Delphi environment works and a fairly good knowledge of many advanced techniques that are not discussed in this book. You can find technical information and some examples of these techniques on my Web site, [www.marcocantu.com](http://www.marcocantu.com), along with links to other sites where these techniques are presented.

Every property editor must be a subclass of the abstract `TPropertyEditor` class, which is defined in the `DesignEditors` unit of the `ToolsApi` and provides a standard implementation for the `IProperty` interface.

### NOTE

The Tools API in Delphi 6 has changed considerably, also for consistency with Kylix. For example, the `DsgnIntf` unit of Delphi 5 has been split into the units `DesignIntf`, `DesignEditors`, and other specific units. Borland has also introduced interfaces to define the sets of methods of each kind of editor. However, most of the simpler examples, such as those presented in this book, compile almost unchanged. As this is not an in-depth analysis of the Tools API, I'm not providing a list of changes from Delphi 5 to Delphi 6, although they are substantial. For more information, you can study the extensive source code in the `\Source\ToolsApi` directory of Delphi 6.

Delphi already defines some specific property editors for strings (the `TStringProperty` class), integers (the `TIntegerProperty` class), characters (the `TCharProperty` class), enumerations (the `TEnumProperty` class), sets (the `TSetProperty` class), so you can actually inherit your property editor from the one of the type of property you are working with.

In any custom property editor, you have to redefine the `GetAttributes` function so it returns a set of values indicating the capabilities of the editor. The most important attributes are `paValueList` and `paDialog`. The `paValueList` attribute indicates that the Object Inspector will show a combo box with a list of values (eventually sorted if the `paSortList` attribute is set) provided by overriding the `GetValues` method. The `paDialog` attribute style activates an ellipsis button in the Object Inspector, which executes the `Edit` method of the editor.

## An Editor for the Sound Properties

The sound button we built earlier had two sound-related properties: `SoundUp` and `SoundDown`. These were actually strings, so we were able to display them in the Object Inspector using a default property editor. However, requiring the user to type the name of a system sound or an external file is not very friendly, and it's a bit error-prone.

When you need to select a file for a string property, you can reuse an existing property editor, the `TMPFileNameProperty` class. All you have to do is register this editor for the property using the special `RegisterPropertyEditor` procedure, as in:

```
RegisterPropertyEditor (TypeInfo (string), TDdhSoundButton, 'SoundUp',
    TMPFileNameProperty);
```

This editor allows you to select a file for the sound, but we want to be able to choose the name of a system sound as well. (As described earlier, system sounds are predefined names of sounds connected with user operations, associated with actual sound files in the Sounds applet of the Windows Control Panel.) For this reason, instead of using this simple approach I'll build a more complex property editor. My editor for sound strings allows a user to either choose a value from a drop-down list or display a dialog box from which to load and test a sound (from a sound file or a system sound). For this reason, the property editor provides both `Edit` and `GetValues` methods:

```
type
  TSoundProperty = class (TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure Edit; override;
  end;
```

### TIP

The default Delphi convention is to name a property editor class with a name ending with *Property* and all component editors with a name ending with *Editor*.

The `GetAttributes` function combines both the `paValueList` (for the drop-down list) and the `paDialog` attributes (for the custom edit box), and also sorts the lists and allows the selection of the property for multiple components:

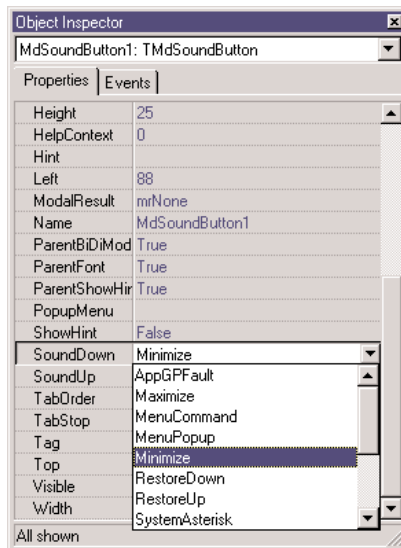
```
function TSoundProperty.GetAttributes: TPropertyAttributes;
begin
    // editor, sorted list, multiple selection
    Result := [paDialog, paMultiSelect, paValueList, paSortList];
end;
```

The `GetValues` method simply calls the procedure it receives as parameter many times, once for each string it wants to add to the drop-down list (as you can see in Figure 11.12):

```
procedure TSoundProperty.GetValues(Proc: TGetStrProc);
begin
    // provide a list of system sounds
    Proc ('Maximize');
    Proc ('Minimize');
    Proc ('MenuCommand');
    Proc ('MenuPopup');
    Proc ('RestoreDown');
    Proc ('RestoreUp');
    Proc ('SystemAsterisk');
    Proc ('SystemDefault');
    Proc ('SystemExclamation');
    Proc ('SystemExit');
    Proc ('SystemHand');
    Proc ('SystemQuestion');
    Proc ('SystemStart');
    Proc ('AppGPFault');
end;
```

**FIGURE 11.12:**

The list of sounds provides a hint for the user, who can also type in the property value or double-click to activate the editor (shown later, in Figure 11.13).



A better approach would be to extract these values from the Windows Registry, where all these names are listed. The `Edit` method is very straightforward, as it simply creates and displays a dialog box. You'll notice that we could have just displayed the Open dialog box directly, but we decided to add an intermediate step to allow the user to test the sound. This is similar to what Delphi does with graphic properties. You open the preview first, and load the file only after you've confirmed that it's correct. The most important step is to load the file and test it before you apply it to the property. Here is the code of the `Edit` method:

```
procedure TSoundProperty.Edit;  
begin  
    SoundForm := TSoundForm.Create (Application);  
    try  
        SoundForm.ComboBox1.Text := GetValue;  
        // show the dialog box  
        if SoundForm.ShowModal = mrOK then  
            SetValue (SoundForm.ComboBox1.Text);  
    finally  
        SoundForm.Free;  
    end;  
end;
```

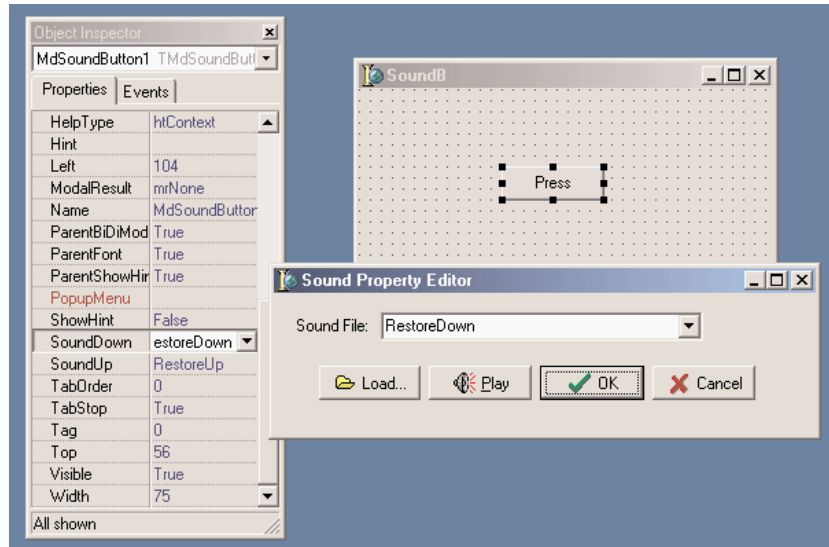
The `GetValue` and `SetValue` methods called above are defined by the base class, the string property editor. They simply read and write the value of the current component's property that we are editing. As an alternative, you can access the component you're editing by using the `GetComponent` method (which requires a parameter indicating which of the selected components you are working on—0 indicates the first component). When you access the component directly, you also need to call the `Modified` method of the `Designer` object (a property of the base class property editor). We don't need this `Modified` call in the example, as the base class `SetValue` method does this automatically for us.

The `Edit` method above displays a dialog box, a standard Delphi form that is built visually, as always, and added to the package hosting the design-time components. The form is quite simple; a `ComboBox` displays the values returned by the `GetValue`s method, and four buttons allow you to open a file, test the sound, and terminate the dialog box by accepting the values or canceling. You can see an example of the dialog box in Figure 11.13. Providing a drop-down list of values *and* a dialog box for editing a property causes the Object Inspector to display only the arrow button that indicates a drop-down list and to omit the ellipsis button to indicate that a dialog box editor is available.



**FIGURE 11.13:**

The Sound Property Editor's form displays a list of available sounds and lets you load a file and hear the selected sound.



The first two buttons of the form each have a simple method assigned to their `OnClick` event:

```

procedure TSoundForm.btnLoadClick(Sender: TObject);
begin
    if OpenDialog1.Execute then
        ComboBox1.Text := OpenDialog1.FileName;
    end;

procedure TSoundForm.btnPlayClick(Sender: TObject);
begin
    PlaySound (PChar (ComboBox1.Text), 0, snd_Async);
end;

```

Unfortunately, I haven't found a simple way to determine whether a sound is properly defined and is available. (Checking the file is possible, but the system sounds create a few issues.) The `PlaySound` function returns an error code when played synchronously, but only if it can't find the default system sound it attempts to play if it can't find the sound you asked for. If the requested sound is not available, it plays the default system sound and doesn't return the error code. `PlaySound` looks for the sound in the Registry first and, if it doesn't find the sound there, checks to see whether the specified sound file exists.

**TIP**

If you want to further extend this example, you might add graphics to the drop-down list displayed in the Object Inspector—if you can decide which graphics to attach to particular sounds.

## Installing the Property Editor

After you've written this code, you can install the component and its property editor in Delphi. To accomplish this, you have to add the following statement to the `Register` procedure of the unit:

```
procedure Register;
begin
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundUp',
    TSoundProperty);
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundDown',
    TSoundProperty);
end;
```

This call registers the editor specified in the last parameter for use with properties of type `string` (the first parameter), but only for a specific component and for a property with a specific name. These last two values can be omitted to provide more general editors. Registering this editor allows the Object Inspector to show a list of values and the dialog box called by the `Edit` method.

To install this component we can simply add its source code file into an existing or new package. Instead of adding this unit and the others of this chapter to the `MdPack` package, I built a second package, containing all the add-ins built in this chapter. The package is named `MdDesPk` (which stands for “*Mastering Delphi* design package”). What’s new about this package is that I’ve compiled it using the `{$DESIGNONLY}` compiler directive. This directive is used to mark packages that interact with the Delphi environment, installing components and editors, but are not required at run time by applications you’ve built.

### NOTE

The source code of all of the add-on tools is in the `MdDesPk` subdirectory, along with the code of the package used to install them. There are no examples demonstrating how to use these design-time tools, because all you have to do is select the corresponding components in the Delphi environment and see how they behave.

The property editor’s unit uses the `SoundB` unit, which defines the `TMdSoundButton` component. For this reason the new package should refer to the existing package. Here is its initial code (I’ll add other units to it later in this chapter):

```
package MdDesPk;

{$R *.RES}
{$ALIGN ON}
...
{$DESCRIPTION 'Mastering Delphi DesignTime Package'}
{$DESIGNONLY}
```

**requires**

```
vc1,  
Mdpack;
```

**contains**

```
PeSound in 'PeSound.pas',  
PeFSound in 'PeFSound.pas' {SoundForm};
```

## Writing a Component Editor

Using property editors allows the developer to make a component more user-friendly. In fact, the Object Inspector represents one of the key pieces of the user interface of the Delphi environment, and Delphi developers use it quite often. However, there is a second approach you can adopt to customize how a component interacts with Delphi: write a custom component editor.

Just as property editors extend the Object Inspector, component editors extend the Form Designer. In fact, when you right-click within a form at design time, you see some default menu items, plus the items added by the component editor of the selected component. Examples of these menu items are those used to activate the Menu Designer, the Fields Editor, the Visual Query Builder, and other editors of the environment. At times, displaying these special editors becomes the default action of a component when it is double-clicked.

Common uses of component editors include adding an About box with information about the developer of the component, adding the component name, and providing specific wizards to set up its properties.

### Subclassing the *TComponentEditor* Class

A component editor should generally inherit from the *TComponentEditor* class, which provides the base implementation of the *IComponentEditor* interface. The most important methods of this interface are:

- `GetVerbCount` returns the number of menu items to add to the local menu of the Form Designer when the component is selected.
- `GetVerb` is called once for each new menu item and should return the text that will go in the local menu for each.
- `ExecuteVerb` is called when one of the new menu items is selected. The number of the item is passed as the method's parameter.
- `Edit` is called when the user double-clicks the component in the Form Designer to activate the default action.

Once you get used to the idea that a “verb” is nothing but a new menu item with a corresponding action to execute, the names of the methods of this interface become quite intuitive. This interface is actually much simpler than those of property editors we’ve seen before.

**NOTE**

Like property editors, component editors were modified extensively from Delphi 5 to Delphi 6, and are now defined in the `DesignEditors` and `DesignIntf` units. But, again, the simpler examples like this one keep compiling almost unchanged, so I won’t delve into the differences.

## A Component Editor for the ListDialog

Now that I’ve introduced the key ideas about writing component editors, we can look at an example, an editor for the `ListDialog` component built earlier. In my component editor, I simply want to be able to show an About box, add a copyright notice to the menu (an improper but very common use of component editors), and allow users to perform a special action—pre-viewing the dialog box connected with the dialog component. I also want to change the default action to simply show the About box after a beep (which is not particularly useful but demonstrates the technique).

To implement this property editor, the program must override the four methods listed above:

```
uses
    DesignIntf;

type
    TMDListCompEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
    end;
```

The first method simply returns the number of menu items I want to add to the local menu:

```
function TMDListCompEditor.GetVerbCount: Integer;
begin
    Result := 3;
end;
```

This method is called only once, before displaying the menu. The second method, instead, is called once for each menu item, so in this case it is called three times:

```
function TMDListCompEditor.GetVerb (Index: Integer): string;
begin
```

```

case Index of
  0: Result := ' MdListDialog (©Cantù)';
  1: Result := '&About this component...';
  2: Result := '&Preview...';
end;
end;

```

The effect of this code is to add the menu items to the local menu of the form, as you can see in Figure 11.14. Selecting any of these menu items just activates the `ExecuteVerb` method of the component editor:

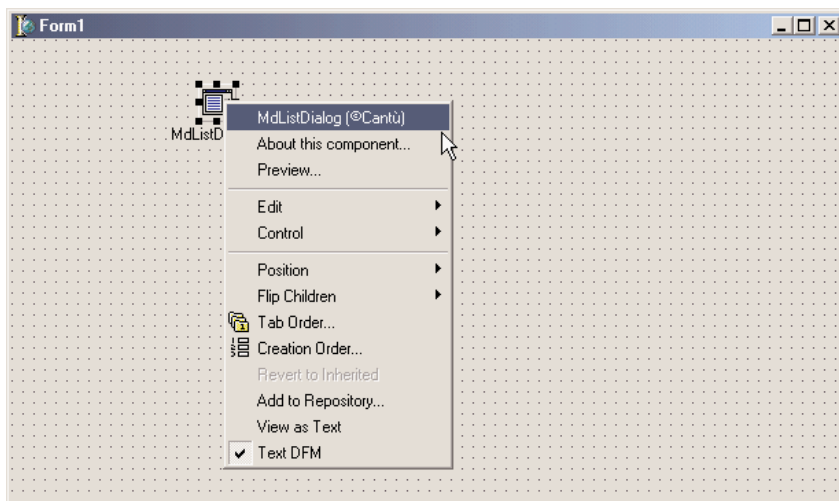
```

procedure TMdListCompEditor.ExecuteVerb (Index: Integer);
begin
  case Index of
    0..1: MessageDlg ('This is a simple component editor'#13 +
      'built by Marco Cantù'#13 +
      'for the book "Mastering Delphi"', mtInformation, [mbOK], 0);
    2: with Component as TMdListDialog do
      Execute;
  end;
end;

```

**FIGURE 11.14:**

The custom menu items added by the property editor of the ListDialog component



I decided to handle the first two items in a single branch of the case statement, although I could have skipped the code for the copyright notice item. The other command changes calls the `Execute` method of the component we are editing, determined using the `Component` property of the `TComponentEditor` class. Knowing the type of the component, we can easily access its methods after a dynamic type cast.

The last method refers to the default action of the component and is activated by double-clicking it in the Form Designer:

```
procedure Edit;  
begin  
  x  
    Beep;  
    ExecuteVerb (0);  
end;
```

## Registering the Component Editor

To make this editor available to the Delphi environment, we need to register it. Once more we can add to its unit a Register procedure and call a specific registration procedure for component editors:

```
procedure TMDListCompEditor.Edit;  
begin  
  // produce a beep and show the about box  
  Beep;  
  ExecuteVerb (0);  
end;
```

I've added this unit to the MdDesPk package, which includes all of the design-time extensions of this chapter. After installing and activating this package you can create a new project, place a tabbed list component in it, and experiment with it.

## What's Next?

In this chapter we have seen how to define various types of properties, how to add events, and how to define and override component methods. We have seen various examples of components, including simple changes to existing ones, new graphical components, and, in the final section, a dialog box inside a component. While building these components, we have faced some new Windows programming challenges. In general, programmers often need to use the Windows API directly when writing new Delphi components.

Writing components is a very handy technique for reusing software, but to make your components easier to use, you should try to integrate them as much as possible within the Delphi environment, writing property editors and component editors.

There are many more extensions of the Delphi IDE you can write, including custom wizards. I've personally built many Delphi extensions, some of which are available (with source code) on my Web site, [www.marcocantu.com](http://www.marcocantu.com).

After discussing components and delving a little into the Delphi environment, the next chapter focuses on Delphi DLLs. We have already met DLLs in many previous chapters, and it is time for a detailed discussion of their role and how to build them. In the same chapter, I'll also further discuss the use of Delphi packages, which are a special type of DLL.

# Libraries and Packages

- DLLs in Windows 95, 98, and NT
- Building and using DLLs in Delphi
- Calling DLL functions at run time
- Sharing data in DLLs
- The structure of Delphi packages
- Placing forms in packages and DLLs



**W**indows executable files come in two flavors: *programs* and *dynamic link libraries* (DLLs). When you write a Delphi application, you typically generate a program file, an EXE. However, Delphi applications often use calls to functions stored in DLLs. Each time you call a Windows API function directly, you actually access a DLL. Delphi also allows programmers to use run-time DLLs for the component library. When you create a package, you basically create a DLL. Delphi can also generate plain dynamic link libraries. The New page of the Object Repository includes a DLL skeleton generator, which generates very few lines of source code.

It is very simple to generate a DLL in the Delphi environment. However, some problems arise from the nature of DLLs. Writing a DLL in Windows is not always as simple as it seems, because the DLL and the calling program need to agree on calling conventions, parameter types, and other details. This chapter covers the basics of DLL programming from the Delphi point of view and provides some simple examples of what you can place in a Delphi DLL. While discussing the examples, I'll also refer to other programming languages and environments, simply because one of the key reasons for writing a procedure in a DLL is to be able to call it from a program written in a different language.

The second part of the chapter will focus on a specific type of dynamic link library, the Delphi *package*. These packages are not as easy to use as they first seem, and it took Delphi programmers some time to figure out how to take advantage of them effectively. Here I'm going to share with you some of these interesting tips and techniques.

## The Role of DLLs in Windows

Before delving into the development of DLLs in Delphi and other programming languages, I'll give you a short technical overview of DLLs in Windows, highlighting the key elements. We will start by looking at dynamic linking, then see how Windows uses DLLs, explore the differences between DLLs and executable files, and end with some general rules to follow when writing DLLs.

### What Is Dynamic Linking?

First of all, you need to understand the difference between static and dynamic linking of functions or procedures. When a subroutine is not directly available in a source file, the compiler adds the subroutine to an internal table, which includes all external symbols. Of course, the compiler must have seen the declaration of the subroutine and know about its parameters and type, or it will issue an error.

After compilation of a normal—*static*—subroutine, the linker fetches the subroutine’s compiled code from a Delphi compiled unit (or static library) and adds it to the executable. The resulting EXE file includes all the code of the program and of the units involved. The Delphi linker is smart enough to include only the minimum amount of code of the units used by the program and to link only the functions and methods that are actually used.

**NOTE**

A notable exception to this rule is the inclusion of virtual methods. The compiler cannot determine in advance which virtual methods the program is going to call, so it has to include them all. For this reason, programs and libraries with too many virtual functions tend to generate larger executable files. While developing the VCL, the Borland developers had to balance the flexibility obtained with virtual functions against the reduced size of the executable files achieved by limiting the virtual functions.

In the case of dynamic linking, which occurs when your code calls a DLL-based function, the linker simply uses the information in the external declaration of the subroutine to set up some tables in the executable file. When Windows loads the executable file in memory, first it loads all the required DLLs, and then the program starts. During this loading process, Windows fills the program’s internal tables with the addresses of the functions of the DLLs in memory. If for some reason the DLL is not found, the program won’t even start, often complaining with nonsense error messages (such as the notorious “a device attached to your system is not functioning”).

Each time the program calls an external function, it uses this internal table to forward the call to the DLL code (which is now located in the program’s address space). Note that this scheme does not involve two different applications. The DLL becomes part of the running program and is loaded in the same address space. All the parameter passing takes place on the application’s stack (because the DLL doesn’t have a separate stack).

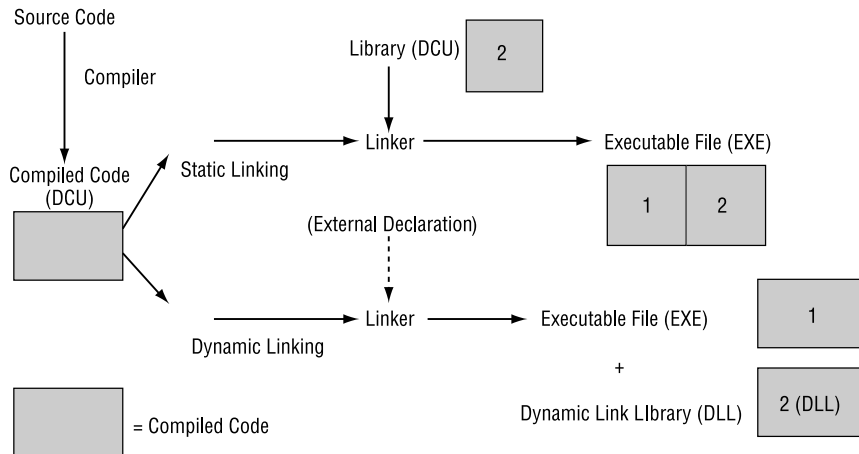
You can see a sketch of how the program calls statically or dynamically linked functions in Figure 12.1. Notice that I haven’t yet discussed compilation of the DLL—because I wanted to focus on the two different linking mechanisms first.

**NOTE**

The term *dynamic linking*, when referring to DLLs, has nothing to do with the late-binding feature of object-oriented programming languages. Virtual and dynamic methods in Object Pascal have nothing to do with DLLs. Unfortunately, the same term is used for both kinds of procedures and functions, which causes a lot of confusion. When I speak of dynamic linking in this chapter, I am referring not to polymorphism but to DLL functions.

**FIGURE 12.1:**

Static and dynamic linking in Windows



There is another approach to using DLLs, which is even more dynamic than the one we have just discussed. In fact, at run time, you can load a DLL in memory, search for a function (provided you know its name), and call the function by name. This approach requires more complex code and takes some extra time to locate the function. The execution of the function, however, has the same speed of the call of an implicitly loaded DLL. On the positive side, you don't need to have the DLL available to start the program. We will use this approach in the DynaCall example later in the chapter.

For the most part, the internal structure of a normal executable file (an EXE file) and a dynamic link library (a DLL, whatever its extension) is the same. They are both executable files. An important difference between programs and DLLs is that a DLL, even when loaded in memory, is not a running program. It is only a collection of procedures and functions that other programs can call. These procedures and functions use the stack of the calling program (the *calling thread*, to be precise). So another difference between a program and a library is that a library doesn't create its own stack—it uses the stack of the program calling it. In Win32, because a DLL is loaded into the application's address space, any memory allocations of the DLL or any global data it creates reside in the address space of the main process.

## What Are DLLs For?

Now that you have a general idea of how DLLs work, we can focus on the reasons for using them in Windows:

- If different programs use the same DLL, the DLL is loaded in memory only once, thus saving system memory. DLLs are mapped into the private address space of each process (each running application), but their code is loaded in memory only once.

**NOTE**

The operating system will try to load the DLL at the same address in each application's address space (using the preferred base address specified by the DLL). If that address is not available in a particular application's virtual address space, the DLL code image for that process will have to be relocated, an operation that is expensive in both performance and memory use. The reason is that the relocation happens on a per-process basis, not system-wide.

- You can provide a different version of a DLL, replacing the current one. If the subroutines in the DLL have the same parameters, you can run the program with the new version of the DLL without having to recompile it. If the DLL has new subroutines, it doesn't matter at all. Problems might arise only if a routine in the older version of the DLL is missing in the new one. Problems also arise if the new DLL does not implement the functions in a manner that is compatible with the operation of the old DLL.

These generic advantages apply in several cases. If you have a complex algorithm, or some complex forms required by several applications, you can store them in a DLL. This will let you reduce the executable's size and save some memory when you run several programs using those DLLs at the same time.

The second advantage is particularly applicable to complex applications. If you have a very big program that requires frequent updates and bug fixes, dividing it into several executables and DLLs allows you to distribute only the changed portions instead of one single large executable. This makes sense for Windows system libraries in particular: You generally don't need to recompile your code if Microsoft provides you an updated version of Windows system libraries—for example, in a new version of the operating system.

Another common technique is to use DLLs to store nothing except resources. You can build different versions of a DLL containing strings for different languages and then change the language at run time, or you can prepare a library of icons and bitmaps and then use them in different applications. The development of language-specific versions of a program is particularly important, and Delphi includes support for it through the Integrated Translation Environment (ITE) and the external environment, which are more advanced topics than I have room to go into.

Another key advantage is that DLLs are independent of the programming language. Most Windows programming environments, including most macro languages in end-user applications, allow a programmer to call a subroutine stored in a DLL. This means you can build a DLL in Delphi and call it from Visual Basic, Excel, and many other Windows applications.

## Understanding System DLLs

The Windows system DLLs take advantage of all the key benefits of DLLs I've just highlighted. For this reason, it is worth examining them. First of all, Windows has many system DLLs. The three central portions of Windows—Kernel, User, and GDI—are implemented using DLLs (with 32-bit or 16-bit code depending on the OS version). Other system DLLs are operating-system extensions, such as the DLLs for common dialog boxes and controls, OLE, device drivers, fonts, ActiveX controls, and hundreds of others.

Dynamic system libraries are one of the technical foundations of the Windows operating systems. Since each application uses the system DLLs for anything from creating a window to producing output, every program is linked to those DLLs. When you change your printer, you do not need to rebuild your application or get a new version of the Windows GDI library, which manages the printer output. You only need to provide a specific driver, which is a DLL called by GDI to access your printer. Each printer type has its own driver DLL, which makes the system extremely flexible.

From a different point of view, version handling is important for the system itself. If you have an application compiled for Windows 95, you should be able to run it on Windows Me, Windows 2000, and (possibly) future versions of Windows, but the application might behave differently, as each version of Windows has different system code.

The system DLLs are also used as system-information archives. For example, the User DLL maintains a list of all the active windows in the system, and the GDI DLL holds the list of active pens, brushes, icons, bitmaps, and the like. The free memory area of these two system DLLs is usually called “free system resources,” and the fact that it is limited plays a very important role in Windows versions still relying on 16-bit code, such as the Windows 9x family. On NT platforms, GDI and User resources are limited only by available system memory.

## Rules for Delphi DLL Writers

In short, there are some rules for Delphi DLL programmers. A DLL function or procedure to be called by external programs must follow these guidelines:

- It has to be listed in the DLL's `exports` clause. This makes the routine visible to the outside world.
- Exported functions should also be declared as `stdcall`, to use the standard Win32 parameter-passing technique instead of the optimized `register` parameter-passing technique (which is the default in Delphi). The exception to this rule is if you want to use these libraries only from other Delphi applications.

- The types of the parameters of a DLL should be the default Windows types, at least if you want to be able to use the DLL within other development environments. There are further rules for exporting strings, as we'll see in the FirstDll example.
- A DLL can use global data that won't be shared by calling applications. Each time an application loads a DLL, it stores the DLL's global data in its own address space, as we will see in the DllMem example.

## Using Existing DLLs

We have already used existing DLLs in examples in this book, when calling Windows API functions. As you might remember, all the API functions are declared in the system Windows unit. Functions are declared in the interface portion of the unit, as shown here:

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL; stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints; p4: Integer):
    BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL; stdcall;
```

Then, in the implementation portion, instead of providing each function's code, the unit refers to the external definition in a DLL:

```
const
    gdi32 = 'gdi32.dll';

function PlayMetaFile; external gdi32 name 'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name 'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

### NOTE

In Windows.PAS there is a heavy use of the {\$EXTERNALSYM identifier} directive. This has little to do with Delphi itself; it applies to C++Builder. This symbol prevents the corresponding Pascal symbol from appearing in the C++ translated header file. This helps keep the Delphi and C++ identifiers in synch, so that code can be shared between the two languages.

The external definition of these functions refers to the name of the DLL they use. The name of the DLL must include the .DLL extension, or the program will not work under Windows 2000 (even though it will work under Windows 9x). The other element is the name of the DLL function itself. The name directive is not necessary if the Pascal function (or procedure) name matches the DLL function name (which is case-sensitive).

To call a function that resides in a DLL, you can provide its declaration and external definition, as shown above, or you can merge the two in a single declaration. Once the function is properly defined, you can call it in the code of your Delphi application just like any other function.

## Using a C++ DLL

As an example, I've written a very simple DLL in C++, with some trivial functions, just to show you how to call DLLs from a Delphi application. I won't explain the C++ code in detail (it's basically C code, anyway) but will focus instead on the calls between the Delphi application and the C++ DLL. In Delphi programming it is common to use DLLs written in C or C++.

Suppose you are given a DLL built in C or C++. You'll generally have in your hands a .DLL file (the compiled library itself), an .H file (the declaration of the functions inside the library), and a .LIB file (another version of the list of the exported functions for the C/C++ linker). This LIB file is totally useless in Delphi, while the DLL file is used as-is, and the H file has to be translated into a Pascal unit with the corresponding declarations.

In the following listing, you can see the declaration of the C++ functions I've used to build the CppDll library example. The complete source code and the compiled version of the C++ DLL and of the source code of the Delphi application using it are in the CppDll directory on the CD. You should be able to compile this code with any C++ compiler; I've tested it only with recent Borland C++ compilers. Here are the C++ declarations of the functions:

```
extern "C" __declspec(dllexport)
int WINAPI Double (int n);
extern "C" __declspec(dllexport)
int WINAPI Triple (int n);
__declspec(dllexport)
int WINAPI Add (int a, int b);
```

The three functions perform some basic calculations on the parameters and return the result. Notice that all the functions are defined with the `WINAPI` modifier, which sets the proper parameter-calling convention; and they are preceded by the `__declspec(dllexport)` declaration, which makes the functions available to the outside world.

Two of these C++ functions also use the C naming convention (indicated by the `extern "C"` statement), but the third one, `Add`, doesn't. This affects the way we call these functions in Delphi. In fact, the internal names of the three functions correspond to their names in the C++ source code file, except for the `Add` function. Since we didn't use the `extern "C"` clause for this function, the C++ compiler used *name mangling*. This is a technique used to include information about the number and type of parameters in the function name, which the C++ language requires in order to implement function overloading. The result when using the Borland C++ compiler is a funny function name: `@Add$qqsi`. This is actually the name we have to use in our Delphi example to call the `Add` DLL function (which explains why you'll generally avoid C++ name mangling in exported functions, and why you'll generally declare them all as `extern "C"`). The following are the declarations of the three functions in the Delphi `CallCpp` example:

```
function Add (A, B: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name '@Add$qqsi';
```

```
function Double (N: Integer): Integer;  
  stdcall; external 'CPPDLL.DLL' name 'Double';  
function Triple (N: Integer): Integer;  
  stdcall; external 'CPPDLL.DLL';
```

As you can see, you can either provide or omit an alias for an external function. I've provided one for the first function (there was no alternative, because the exported DLL function name `@Add$qqsi` is not a valid Pascal identifier) and for the second, although in the second case it was unnecessary. If the two names match, in fact, you can omit the `name` directive, as I did for the third function above. If you are not sure of the actual names of the functions exported by the DLL, you can use the optional Windows viewer for executable files, with the `QuickView` command of Windows Explorer or Borland's `TDump32` command-line program, available in the Delphi BIN folder.

Remember to add the `stdcall` directive to each definition, so that the caller module (the application) and the module being called (the DLL) use the same parameter-passing convention. If you fail to do so, you will get random values passed as parameters, a bug that is very hard to trace.

**NOTE**

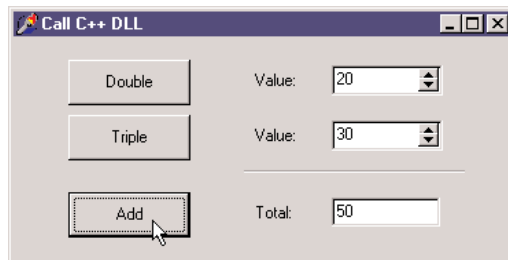
When you have to convert a large C/C++ header file to the corresponding Pascal declarations, instead of doing a manual conversion you can use a tool to partially automate the process. One of these tools is `HeadConv`, written by Bob Swart. You'll find a copy on his Web site, [www.drbob42.com](http://www.drbob42.com). Notice, though, that automatic header translation from C/C++ to Pascal is not possible, because Pascal is more strongly typed than C/C++, so you have to use types more precisely.

To use this C++ DLL, I've built a Delphi example, named `CallCpp`. Its simple form has buttons to call the functions of the DLL and some visual components for input and output parameters (see Figure 12.2). The code of the button event handlers looks like:

```
procedure TForm1.BtnDoubleClick(Sender: TObject);  
begin  
  SpinEdit1.Value := Double (SpinEdit1.Value);  
end;
```

**FIGURE 12.2:**

The output of the `CallCpp` example when you have pressed each of the buttons





Notice that to run this application, you should have the DLL in the same directory as the project, in one of the directories on the path, or in the `Windows` or `System` directories. If you move the executable file to a new directory and try to run it, you'll get a run-time error indicating that the DLL is missing.

## Creating a DLL in Delphi

Besides using DLLs written in other environments, you can use Delphi to build DLLs that can be used by Delphi programs or with any other development tool that supports DLLs. Building DLLs in Delphi is so easy that you might overuse this feature. In general, I suggest you try to build components and packages instead of plain DLLs. As I'll discuss later in this chapter, packages often contain components, but they can also include plain noncomponent classes, allowing you to write object-oriented code and to reuse it effectively.

Placing a collection of functions in a DLL is a more traditional approach to programming: DLLs cannot export classes and objects, at least unless you use Microsoft's COM technology or some other advanced techniques.

As I've already mentioned, building a DLL is useful when a portion of the code of a program is subject to frequent changes. In this case you can frequently replace the DLL, keeping the rest of the program unchanged. Similarly, when you need to write a program that provides different features to different groups of users, you can distribute different versions of a DLL to those users.

### A Simple Delphi DLL

As a starting point in exploring the development of DLLs in Delphi, I'll show you a very simple library built in Delphi. The primary focus of this example will be to show the syntax you use to define a DLL in Delphi, but it will also illustrate a few considerations involved in passing string parameters. To start, select the `File > New > Other` command and choose the `DLL` option in the `New` page of the `Object Repository`. This creates a very simple source file that starts with the following definition:

```
library Project1;
```

The `library` statement indicates that we want to build a DLL instead of an executable file. Now we can add routines to the library and list them in an `exports` statement:

```
function Triple (N: Integer): Integer; stdcall;  
begin  
    Result := N * 3;  
end;
```

```
function Double (N: Integer): Integer; stdcall;  
begin  
    Result := N * 2;  
end;  
  
exports  
    Triple, Double;
```

In this basic version of the DLL, we don't need a `uses` statement; but in general, the main project file includes only the `exports` statement, while the function declarations are placed in a separate unit. In the final source code of the `FirstDll` example on the CD, I've actually changed the code slightly from the version listed above, to show a message each time a function is called. There are two ways to accomplish this. The simplest is to use the `Dialogs` unit and call the `ShowMessage` function.

The code requires Delphi to link a lot of VCL code into the application. If you statically link the VCL into this DLL, the resulting size will be about 375 KB. The reason is that the `ShowMessage` function displays a VCL form that contains VCL controls and uses VCL graphics classes; and those indirectly refer to things like the VCL streaming system and the VCL application and screen objects. For this simple case, a better alternative is to show the messages using direct API calls, using the `Windows` unit and calling the `MessageBox` function, so that the VCL code is not required. This change in code brings the size of the application down to only about 40 KB.

**NOTE**

This huge difference in size underlines the fact that you should not overuse DLLs in Delphi, to avoid compiling the code of the VCL in multiple executable files. Of course, you can reduce the size of a Delphi DLL by using run-time packages, as detailed later in this chapter.

If you run a test program like the `CallFrst` example (described later) using the API-based version of the DLL, its behavior won't be correct. In fact, you can click the buttons that call the DLL functions several times without first closing the message boxes displayed by the DLL. This happens because the first parameter of the `MessageBox` API call above is zero. Its value should instead be the handle of the program's main form or the application form, information I don't have at hand in the DLL itself.

## Overloaded Functions in Delphi DLLs

When you create a DLL in C++, overloaded functions use name mangling to generate a different name for each function, including the type of the parameters right in the name, as we've seen in the `CppDll` example.

When you create a DLL in Delphi and use overloaded functions (that is, multiple functions using the same name and marked with the `overload` directive), Delphi allows you to export only one of the overloaded functions with the original name, indicating its parameters list in the `exports` clause. If you want to export multiple overloaded functions, you should specify different names in the `exports` clause to distinguish the overloads. This is demonstrated by this portion of the `FirstDLL` code:

```
function Triple (C: Char): Integer; stdcall; overload;  
function Triple (N: Integer): Integer; stdcall; overload;  
  
exports  
  Triple (N: Integer),  
  Triple (C: Char) name 'TripleChar';
```

**NOTE**

The reverse is possible as well: You can import a series of similar functions from a DLL and define them all as overloaded functions in the Pascal declaration. Delphi's `OpenGL.PAS` unit contains a series of examples of this technique.

## Exporting Strings from a DLL

In general, functions in a DLL can use any type of parameter and return any type of value. There are two exceptions to this rule:

- If you plan to call the DLL from other programming languages, you should probably try using Windows native data types instead of Delphi-specific types. For example, to express color values, you should use integers or the Windows `ColorRef` type instead of the Delphi native `TColor` type, doing the appropriate conversions (as in the `FormDLL` example, described in the next section). Other Delphi types that, for compatibility, you should avoid using include objects, which cannot be used by other languages at all, and Pascal strings, which can be replaced by `PChar` strings. In other words, every Windows development environment must support the basic types of the API, and if you stick to them, your DLL will be usable with other development environments. Also, Pascal file variables (text files and binary file of record) should not be passed out of DLLs, but you can use Win32 file handles.
- Even if you plan to use the DLL only from a Delphi application, you cannot pass Delphi strings (and dynamic arrays) across the DLL boundary without taking some precautions. This is because of the way Delphi manages strings in memory—allocating, real-locating, and freeing them automatically. The solution to the problem is to include the `ShareMem` system unit both in the DLL and in the program using it. This unit must be included as the first unit of each of the projects.

**NOTE**

Objects actually can be passed out of a DLL, if the objects are designed to be used like interfaces or pure abstract classes. All methods of these objects must be virtual, and objects must be created by the DLL. This is more or less what happens with COM objects, the approach you should use for multilanguage applications. For Delphi-only projects with libraries exporting objects, you should rather use packages, as we'll see later in this chapter.

In the FirstDLL example, I've actually included both approaches: One function receives and returns a Pascal string, and another one receives as parameter a PChar pointer, which is then filled by the function itself. The first function is very simple:

```
function DoubleString (S: string; Separator: Char): string; stdcall;  
begin  
    Result := S + Separator + S;  
end;
```

The second one is quite complex because PChar strings don't have a simple + operator, and they are not directly compatible with characters; the separator must be turned into a string before adding it. Here is the complete code; it uses input and output PChar buffers, which are compatible with any Windows development environment:

```
function DoublePChar (BufferIn, BufferOut: PChar;  
    BufferOutLen: Cardinal; Separator: Char): LongBool; stdcall;  
var  
    SepStr: array [0..1] of Char;  
begin  
    // if the buffer is large enough  
    if BufferOutLen > StrLen (BufferIn) * 2 + 2 then  
        begin  
            // copy the input buffer in the output buffer  
            StrCopy (BufferOut, BufferIn);  
            // build the separator string (value plus null terminator)  
            SepStr [0] := Separator;  
            SepStr [1] := #0;  
            // append the separator  
            StrCat (BufferOut, SepStr);  
            // append the input buffer once more  
            StrCat (BufferOut, BufferIn);  
            Result := True;  
        end  
    else  
        // not enough space  
        Result := False;  
end;
```

This second version of the code is certainly more complex, but the first can be used only from Delphi. Moreover, the first version requires us to include the ShareMem unit in the DLL (and in the programs using it) and to deploy the file Bor1ndMM.DLL (the name stands for Borland Memory Manager) along with the program and the specific library.

## Calling the Delphi DLL

How can we use the library we've just built? We can call it from within another Delphi project or from other environments. As an example, I've built the CallFirst project (stored in the FirstDLL directory).

To access the DLL functions, we must declare them as `external`, as we've done with the C++ DLL. This time, however, we can simply copy and paste the definition of the functions from the source code of the Delphi DLL, adding the `external` clause, as in:

```
function Double (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';
```

This declaration is similar to those used to call the C++ DLL. This time, however, we have no problems with function names. The source code of the example is actually quite simple. Once they are redeclared as `external`, the functions of the DLL can simply be used as if they were local functions. Here are two examples, with calls to the string-related functions:

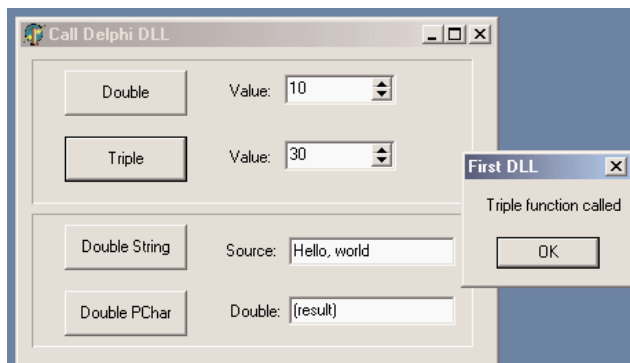
```
procedure TForm1.BtnDoubleStringClick(Sender: TObject);
begin
  // call the DLL function directly
  EditDouble.Text := DoubleString (EditSource.Text, ',');
end;

procedure TForm1.BtnDoublePCharClick(Sender: TObject);
var
  Buffer: string;
begin
  // make the buffer large enough
  SetLength (Buffer, 1000);
  // call the DLL function
  if DoublePChar (PChar (EditSource.Text), PChar (Buffer), 1000, '/') then
    EditDouble.Text := Buffer;
end;
```

Figure 12.3 shows the effect of this program's calls to the DLL.

**FIGURE 12.3:**

The output of the CallFirst example, which calls the DLL we've built in Delphi



## Project and Library Names in Delphi 6

For a library, as for a standard application, you end up with a library name matching a Delphi project filename. Following a similar technique introduced in Kylix for compatibility with standard Linux naming conventions for shared object libraries (the Linux equivalent of Windows DLLs), Delphi 6 introduced special compiler directives you can use in libraries to determine their executable filename. Some of these directives make more sense in the Linux world than on Windows, but they've all been added anyway.

- `$LIBPREFIX` is used to add something in front of the library name. Paralleling the Linux technique of adding *lib* in front of library names, this directive is used by Kylix to add *bpl* at the beginning of package names. This is due to the fact that Linux uses a single extension (`.so`) for libraries, while in Windows you can have different library extensions, something Borland uses for packages (`.bpl`).
- `$LIBSUFFIX` is used to add text after the library name and before the extension. This can be used to specify versioning information or other variations on the library name and can be quite useful also on Windows.
- `$LIBVERSION` is used to add a version number after the extension—something very common in Linux, but you should generally avoid this on Windows.

As an example, consider the following directives, which generate a library called `MarcoNameTest60.d11`:

```
library NameTest;  
{ $LIBPREFIX 'Marco' }  
{ $LIBSUFFIX '60' }
```

### NOTE

Unlike past versions, Delphi 6 packages use the `$LIBSUFFIX` directive extensively. For this reason, the VCL package generates the `VCL.DCP` file and the `VCL60.BPL` file. The advantage of this approach is that you won't need to change the `requires` portions of your packages for every new version of Delphi. Of course, this will become handy to maintain Delphi 7- and Delphi 6-compatible packages, but isn't helpful now for Delphi 5 compatibility.

## A Delphi Form in a DLL

Besides writing simple DLLs with functions and procedures, you can place a complete form built with Delphi into a DLL. This can be a dialog box or any other kind of form, and it can be used not only by other Delphi programs, but also by other development environments or macro languages.

To build the FormDLL example, I've built a simple form with three scroll bars you can use to select a color and two preview areas for the resulting pen and brush colors. The form also contains two bitmap buttons and has its `BorderStyle` property set to `bsDialog`. Aside from developing a form as usual, I've only added two new subroutines to the unit that defines the form. In the interface portion of the unit, I've added the following declarations:

```
function GetColor (Col: LongInt): LongInt; stdcall;  
procedure ShowColor (Col: LongInt;  
    FormHandle: THandle; MsgBack: Integer); stdcall;
```

In both subroutines the `Col` parameter is the initial color. Notice that I've passed it as a long integer, which corresponds to the Windows `ColorRef` data type. As mentioned before, using the `TColor` Delphi type might have caused problems with non-Delphi applications: Even though a `TColor` is very similar to a `ColorRef`, these types don't always correspond. When you write a DLL, I suggest you use only the Windows native data types (unless you are sure only Delphi programs will use the DLL).

The `GetColor` function returns the final color (which is the same as the initial color if the user clicks the Cancel button). The value is returned immediately because the function shows the form as a modal form. The `ShowColor` procedure, instead, simply displays the form (as a modeless form) and returns immediately. For this reason the form needs a way to communicate back to the calling form. In this case I've decided to pass as parameters the handle for the window of the calling form and the ID of the message to use to communicate back with it.

In the next sections, you'll see how to write the code of the two subroutines; and you'll also see what problems arise, particularly when you place a modeless form in a DLL. Of course, I'll also provide a few alternative fixes.

## Using the DLL Form as Modal

When you want to place a Delphi component (such as a form) in a DLL, you can only provide functions that create, initialize, or run the component or access its properties and data. The simplest approach is to have a single function that sets the data, runs the component, and returns the result, as in the modal version. Here is the code of the function, added to the implementation portion of the unit that defines the form:

```
function GetColor (Col: LongInt): LongInt; stdcall;  
var  
    FormScroll: TFormScroll;  
begin  
    // default value  
    Result := Col;  
    try  
        FormScroll := TFormScroll.Create (Application);  
    try
```

```
    // initialize the data
    FormScroll.SelectedColor := Col;
    // show the form
    if FormScroll.ShowModal = mrOK then
        Result := FormScroll.SelectedColor;
    finally
        FormScroll.Free;
    end;
except
    on E: Exception do
        MessageDlg ('Error in FormDLL: ' + E.Message, mtError, [mbOK], 0);
    end;
end;
```

An important element is the structure of the `GetColor` function. The code creates the form at the beginning, sets some initial values, and then runs the form, eventually extracting the final data. What makes this different from the code we generally write in a program is the use of exception handling:

- A try/except block protects the whole function, so that any exception generated by the function will be trapped, displaying a proper message. The reason for handling every possible exception is that the calling application might be written in any language, in particular one that doesn't know how to handle exceptions. Even when the caller is a Delphi program, it is sometimes useful to use the same protective approach.
- A try/finally block protects the operations on the form, ensuring that the form object will be properly destroyed, even when an exception is raised.

By checking the return value of the `ShowModal` method, the program determines the result of the function. I've set the default value before entering the try block to ensure that it will always be executed (and also to avoid the compiler warning indicating that the result of the function might be undefined).

Now that we have updated the form and written the code of the unit, we can move to the project source code, which (temporarily) becomes the following:

```
library FormDLL;

uses
  ScrollF in 'SCROLLF.PAS' {FormScroll};

exports
  GetColor;
end.
```

We can now use a Delphi program to test the form we have placed in the DLL. The `UseCol` example is in the same directory as the previous DLL, `FormDLL` (and both projects



are part of the FormDLL project group, the file FormD11.BPG). The form of the UseCol example contains a button to call the GetColor function of the DLL. Here is the definition of this function and the code of the Button1Click method:

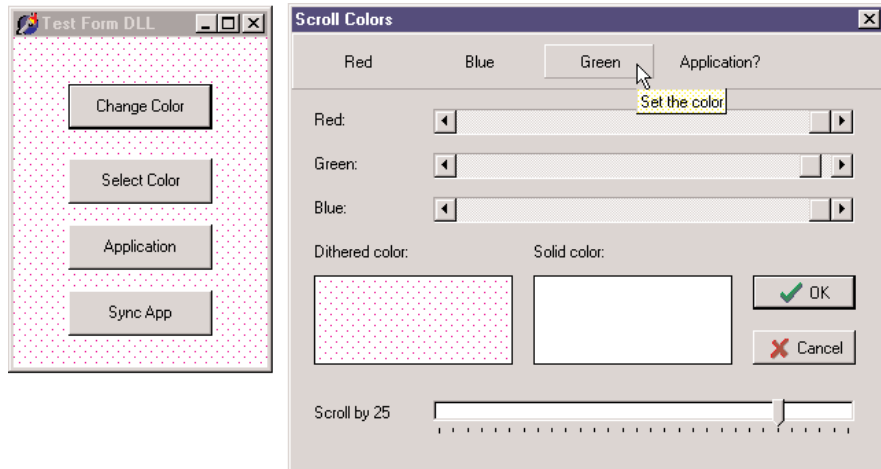
```
function GetColor (Col: LongInt): LongInt; stdcall; external 'FormDLL.DLL';

procedure TForm1.Button1Click(Sender: TObject);
var
    Col: LongInt;
begin
    Col := ColorToRGB (Color);
    Color := GetColor (Col)
end;
```

Running this program (see Figure 12.4) displays the dialog box, using the current background color of the main form. If you change the color and click OK, the program uses the new color as the background color for the main form.

**FIGURE 12.4:**

The execution of the UseCol test program when it calls the dialog box we have placed in the FormDLL



If you execute this as a modal dialog box, almost all the features of the form work fine. You can see the tooltips, the flat speed buttons in the toolbar behave properly, and you get no extra entry in the task bar. This might be obvious, but is not what will happen when we use the form inside the DLL as a modeless form. Even with modal forms, however, I recommend synchronizing the application objects of the DLL and the executable file, as described in the next section.

## A Modeless Form in a DLL

The second subroutine of the FormDLL example uses a different approach. As mentioned, it receives three parameters: the color, the handle of the main form, and the message number for notification when the color changes. These values are stored in the private data of the form:

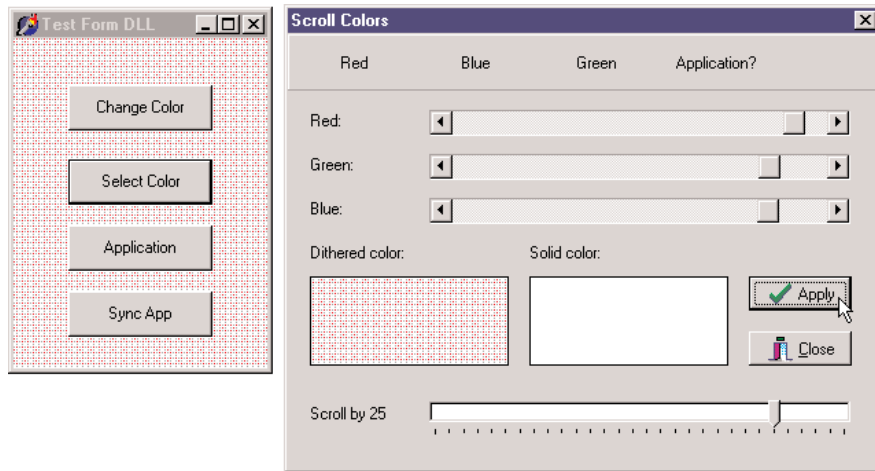
```
procedure ShowColor (Col: LongInt;
  FormHandle: THandle; MsgBack: Integer); stdcall;
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  try
    // initialize the data
    FormScroll.FormHandle := FormHandle;
    FormScroll.MsgBack := MsgBack;
    FormScroll.SelectedColor := Col;
    // show the form
    FormScroll.Show;
  except
    on E: Exception do
      begin
        MessageDlg ('Error in FormDLL: ' + E.Message, mtError, [mbOK], 0);
        FormScroll.Free;
      end;
    end;
end;
```

When the form is activated, it checks to see if it was created as a modal form (simply testing the FormHandle field). In this case, the form changes the caption and the behavior of the OK button, as well as the overall style of the Cancel button (you can see the modified buttons in Figure 12.5):

```
procedure TFormScroll.FormActivate(Sender: TObject);
begin
  // change buttons for modeless form
  if FormHandle <> 0 then
    begin
      BitBtn1.Caption := 'Apply';
      BitBtn1.OnClick := ApplyClick;
      BitBtn2.Kind := bkClose;
    end;
end;
```

**FIGURE 12.5:**

When the DLL-based form is used as a modeless form, its buttons are slightly modified (as you can see comparing this image with that of Figure 12.4).



The `ApplyClick` method I've manually added to the form simply sends the notification message to the main form, using one of the parameters to send back the selected color:

```
SendMessage (FormHandle, MsgBack, SelectedColor, 0);
```

Finally, the form's `OnClose` event destroys the form object, by setting the `Action` parameter to `caFree`. Now let's move back to the demo program. The second button of the `UseForm` example's form has the following code:

```
procedure TForm1.BtnSelectClick(Sender: TObject);
var
    Col: LongInt;
begin
    Col := ColorToRGB (Color);
    ShowColor (Col, Handle, wm_user);
end;
```

The form also has a message-handling method, connected with the `wm_user` message. This method reads the value of the parameter corresponding to the color and sets it:

```
procedure TForm1.UserMessage(var Msg: TMessage);
begin
    Color := Msg.WParam;
end;
```

Running this program produces some strange effects. Basically, the modeless form and the main form are not synchronized, so they both show up in the Windows Taskbar; and when you minimize the main form, the other one remains on the screen. The two forms behave as if they were part of separate applications, because two Delphi programs (the DLL and the EXE) have two separate global `Application` objects, and only the `Application` object of the executable file has an associated window.

To test this situation, I've added a button to both the main form and the DLL form, showing the numeric value of the `Application` object's handle. Here is the code for one of them:

```
ShowMessage ('Application Handle: ' + IntToStr (Application.Handle));
```

For the form in the DLL, you'll invariably get the value 0, while for the form in the executable you get a numeric value determined each time by Windows.

To fix the problem we can add to the DLL an initialization function that passes the handle of the application window to the library. In practice, we copy the `Handle` of the executable's `Application` object to the same property of the DLL's `Application` object. This is enough to synchronize the two `Application` objects and make the two forms behave as in a simple Delphi program. Here is the code of the function in the DLL:

```
procedure SyncApp (AppHandle: THandle); stdcall;  
begin  
    Application.Handle := AppHandle;  
end;
```

And here is the call to it in the executable file:

```
procedure TForm1.BtnSyncClick(Sender: TObject);  
begin  
    SyncApp (Application.Handle);  
    BtnSync.Enabled := False;  
end;
```

## NOTE

Assigning the handle of the application object of the DLL is not a work-around for a bug but a documented operation required by the VCL. The VCL `Application` object supports assignment to its `Handle` property (unlike most other `Handle` properties of the VCL) specifically to allow programmers to tie DLL-based forms into the environment of a host application.

I've connected this code to a button, instead of executing it automatically at startup, to let you test the behavior in the two different cases. Before you click the Sync App button, the secondary modeless form behaves oddly. If you close it, synchronize the applications, and then create another instance of the modeless form, it will behave almost correctly. The only visible problem is that the flat speed buttons of the modeless form won't be highlighted when the mouse moves over them. We'll see how to fix this problem using run-time packages at the end of the chapter.

## NOTE

Technically this behavior of the speed buttons depends on the fact that the controls in the DLL form don't receive the `cm_MouseEnter` and `cm_MouseLeave` messages, because the DLL's `Application.Idle` method is never called. The DLL's `Application` object, in fact, is not running the application's message loop. You can activate it by exporting from the DLL a function that calls the internal `Application.Idle` routine, and call that function from the host application when its message loop goes idle. As I've mentioned, however, all these problems (and a few others) can be better solved by using run-time packages.

## Calling a Delphi DLL from Visual Basic for Applications

We can also display this color dialog box from other programming languages. Calling this DLL from C or C++ is easy. To link the application, you need to generate an import library (using the `IMPLIB` command-line utility) and add the resulting LIB file to the project. Since I've already used a C++ compiler in this chapter, this time I will write a similar example using Microsoft Word for Windows and Visual Basic for Applications instead.

To start, open Microsoft Word. Then open its Macro dialog box (with the `Tools > Macro` menu item or a similar command, depending on your version of Word), type a new macro name, such as **DelphiColor**, and click the Create button. You can now write the BASIC code, which declares the function of our DLL and calls it. The BASIC macro uses the result of the DLL function in two ways. By calling `Insert`, it adds to the current document a description of the color with the amount of red, green, and blue; and by calling `Print` it displays the numeric value in the status bar:

```
Declare Function GetColor Lib "FormDLL" (Col As Long) As Long
Sub MAIN
    NewColor = GetColor(0)
    Print "The code of the color is " + Str$(NewColor)
    Insert "Red:" + Str$(NewColor Mod 256) + Chr$(13)
    Insert "Green:" + Str$(Int(NewColor / 256) Mod 256) + Chr$(13)
    Insert "Blue:" + Str$(Int(NewColor / (256 * 256))) + Chr$(13)
End Sub
```

Unfortunately, there is no easy way to use RGB colors in Word, since Word's color schemes are based on fixed color codes. Here is an example of the output of this macro:

```
Red: 141
Green: 109
Blue: 179
```

You can find the text of this macro in the file `WORDCALL.TXT`, in the directory containing this DLL. If you want to test it, remember to first copy the DLL file into one of the directories of the path or into the Windows system directory.

### NOTE

A better way to integrate Delphi code with Office applications is to use OLE Automation, instead of writing custom DLLs and calling them from the macro language. We'll see examples of OLE Automation in Chapter 20.

## Calling a DLL Function at Run Time

Up to now, we've always referenced in our code the functions exported by the libraries, so that the DLLs will be loaded along with the program. I mentioned earlier that we can also delay the loading of a DLL until the moment it is actually needed, so we'd be able to use the rest of the program in case the DLL is not available.

Dynamic loading of a DLL in Windows is accomplished by calling the `LoadLibrary` API function, which searches the DLL in the program folder, in the folders on the path, and in some system folders. If the DLL is not found, Windows will show an error message, something you can skip by calling Delphi's `SafeLoadLibrary` function. This function has the same effect as the API it encapsulates, but it suppresses the standard Windows error message and should be the preferred way to load libraries dynamically in Delphi.

If the library is found and loaded (something you know by checking the return value of `LoadLibrary` or `SafeLoadLibrary`), a program can call the `GetProcAddress` API function, which searches the DLL's exports table, looking for the name of the function passed as a parameter. If `GetProcAddress` finds a match, it returns a pointer to the requested procedure. Now we can simply cast this function pointer to the proper data type and call it.

Whichever loading functions you've used, don't forget to call `FreeLibrary` at the end, so that the DLL can be properly released from memory. In fact, the system uses a reference-counting technique for libraries, releasing them when each loading request has been followed by a freeing request.

The example I've built to show dynamic DLL loading is named `DynaCall` and uses the `FirstDLL` library we built earlier in this chapter (to make the program work, I've copied the DLL into the same folder as the `DynaCall` example). Instead of declaring the `Double` and `Triple` functions and using them directly, this example obtains the same effect with somewhat more complex code. The advantage, however, is that the program will run even without the DLL. Also, if new *compatible* functions are added to the DLL, we won't have to revise the program's source code and recompile it to access those new functions. Here is the core code of the program:

```
type
  TIntFunction = function (I: Integer): Integer; stdcall;

const
  DLLName = 'Firstdll.dll';

procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFunct: TIntFunction;
begin
  HInst := SafeLoadLibrary (DLLName);
  if HInst > 0 then
  try
    FPointer := GetProcAddress (HInst,
      PChar (Edit1.Text));
```

```
    if FPointer <> nil then
    begin
        MyFunc := TIntFunction (FPointer);
        SpinEdit1.Value := MyFunc (SpinEdit1.Value);
    end
    else
        ShowMessage (Edit1.Text + ' DLL function not found');
finally
    FreeLibrary (HInst);
end
else
    ShowMessage (DllName + ' library not found');
end;
```

How do you call a procedure in Delphi, once you have a pointer to it? One solution is to convert the pointer to a procedural type and then call the procedure using the procedural-type variable, as in the listing above. Notice that the procedural type you define must be compatible with the definition of the procedure in the DLL. This is the Achilles' heel of this method—there is no check of the parameter types.

What is the advantage of this approach? In theory, you can use it to access any function of any DLL at any time. In practice, it is useful when you have different DLLs with compatible functions or a single DLL with several compatible functions, as in our case. What we can do is to call the `Double` and `Triple` methods simply by entering their names in the edit box. Now, if someone gives us a DLL with a new function receiving an integer as a parameter and returning an integer, we can call it simply by entering its name in the edit box. We don't even need to recompile the application.

With this code, the compiler and the linker ignore the existence of the DLL. When the program is loaded, the DLL is not loaded immediately. We might make the program even more flexible and let the user enter the name of the DLL to use. In some cases, this is a great advantage. A program may switch DLLs at run time, something the direct approach does not allow. Note that this approach to loading DLL functions is common in macro languages and is used by many visual programming environments. Also, the code of the `Word` macro we saw earlier in this chapter uses this approach to load the DLL and to call the external function. Well, you don't want to recompile `Word`, do you?

Only a system based on a compiler and a linker, such as Delphi, can use the direct approach, which is generally more reliable and also a little bit faster. I think the indirect loading approach of the `DynaCall` example is useful only in special cases, but it can be extremely powerful.

## A DLL in Memory: Code and Data

We can use this technique, based on the `GetProcAddress` API function, to test which memory address of the current process a function has been mapped to, with the following code:

```
procedure TForm1.Button3Click(Sender: TObject);  
var  
    HDLLInst: THandle;  
begin  
    HDLLInst := SafeLoadLibrary ('dllmem');  
    Label1.Caption := Format ('Address: %p', [  
        GetProcAddress (HDLLInst, 'SetData')]);  
    FreeLibrary (HDLLInst);  
end;
```

This code displays, in a label, the memory address of the function, within the address space of the calling application. If you run two programs using this code, they'll generally both show the same address. This demonstrates that the code is loaded only once at a common memory address.

### NOTE

The memory address will be different if the DLL had to be relocated in one of the processes, or if each process has relocated the DLL to a different base address. In these cases, the code is generally not shared in memory but actually loaded multiple times, because absolute address references in the code must be rewritten to refer to the proper new addresses. As the code is modified by the loader, it cannot be shared.

If the code of the DLL is loaded only once, what about the global data? Basically, each copy of the DLL has its own copy of the data, in the address space of the calling application. However, it is indeed possible to share global data between applications using a DLL. The most common technique for sharing data is to use memory-mapped files. I'll use this technique for a DLL, but it can also be used to share data directly among applications.

This example is called `DllMem` and uses a project group with the same name, as in the previous examples of this chapter. The `DllMem` project group includes the `DllMem` project (the DLL itself) and the `UseMem` project (the demo application). The DLL code has a simple project file, which exports four subroutines:

```
library dllmem;  
  
uses  
    SysUtils,  
    DllMemU in 'DllMemU.pas';  
  
exports  
    SetData, GetData,  
    GetShareData, SetShareData;  
end.
```



The actual code is in the secondary unit (`D11MemU.PAS`), which has the code of the four routines that read or write two global memory locations. These hold an integer and a pointer to an integer. Here are the variable declarations and the two `Set` routines:

```

var
  PlainData: Integer = 0; // not shared
  ShareData: ^Integer; // shared

procedure SetData (I: Integer); stdcall;
begin
  PlainData := I;
end;

procedure SetShareData (I: Integer); stdcall;
begin
  ShareData^ := I;
end;

```

## Sharing Data with Memory-Mapped Files

For the data that isn't shared, there isn't anything else to do. To access the shared data, however, the DLL has to create a memory-mapped file and then get a pointer to this memory area. These two operations require two Windows API calls:

- `CreateFileMapping` requires as parameters the filename (or `$FFFFFFFF` to use a virtual file in memory), some security and protection attributes, the size of the data, and an internal name (which must be the same to share the mapped file from multiple calling applications).
- `MapViewOfFile` requires as parameters the handle of the memory mapped file, some attributes and offsets, and the size of the data (again).

Here is the source code of the `initialization` section, executed every time the DLL is loaded into a new process space (that is, once for each application that uses the DLL):

```

var
  hMapFile: THandle;

const
  VirtualFileName = 'ShareD11Data';
  DataSize = sizeof (Integer);

initialization
  // create memory mapped file
  hMapFile := CreateFileMapping ($FFFFFFFF, nil,
    Page_ReadWrite, 0, DataSize, VirtualFileName);
  if hMapFile = 0 then
    raise Exception.Create ('Error creating memory-mapped file');

```

```
// get the pointer to the actual data
ShareData := MapViewOfFile (
    hMapFile, File_Map_Write, 0, 0, DataSize);
```

When the application terminates and the DLL is released, it has to free the pointer to the mapped file and the file mapping itself:

**finalization**

```
UnmapViewOfFile (ShareData);
CloseHandle (hMapFile);
```

The code of the program using this DLL, UseMem, is very simple. The form of this application has four edit boxes (two with an UpDown control connected), five buttons, and a label. The first button saves the value of the first edit box in the DLL data, getting the value from the connected UpDown control:

```
SetData (UpDown1.Position);
```

If you click the second button, the program copies the DLL data to the second edit box:

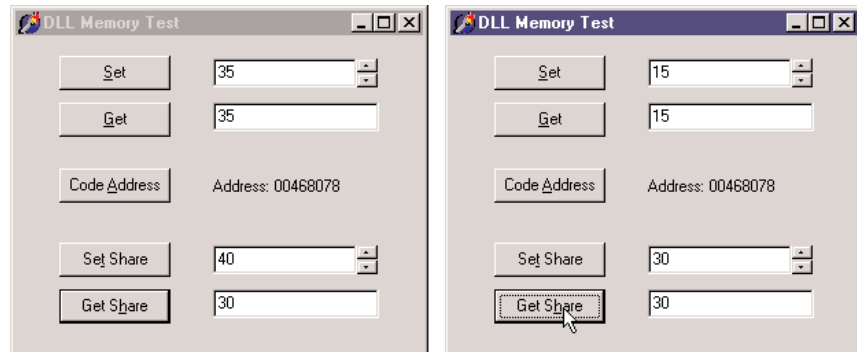
```
Edit2.Text := IntToStr(GetData);
```

The third button is used to display the memory address of a function, with the source code shown at the beginning of this section, and the last two buttons have basically the same code as the first two, but they call the SetShareData procedure and GetShareData function.

If you run two copies of this program, you can see that each copy has its own value for the plain global data of the DLL, while the value of the shared data is common. Set different values in the two programs and then get them in both, and you'll see what I mean. This situation is illustrated in Figure 12.6.

**FIGURE 12.6:**

If you run two copies of the UseMem program, you'll see that the global data in its DLL is not shared.



**WARNING** Memory-mapped files reserve a minimum of a 64 KB range of virtual addresses and consume physical memory in 4 KB pages. The example's use of 4-byte Integer data in shared memory is rather expensive, especially if you use the same approach for sharing multiple values. If you need to share several variables, you should place them all in a single shared memory area (accessing the different variables using pointers or building a record structure for all of them).

## Using Delphi Packages

In Delphi, component packages are an important type of DLL. Packages allow you to bundle a group of components and then link the components either statically (adding their compiled code to the executable file of your application) or dynamically (keeping the component code in a DLL, the run-time package that you'll distribute along with your program). In the last chapter, you saw how to build a package. Now I want to underline some advantages and disadvantages of the two forms of linking for a package. There are many elements to keep in mind:

- Using a package as a DLL makes the executable files much smaller.
- Linking the package units into the program allows you to distribute only part of the package code. Generally, the size of the executable file of an application plus the size of the required package DLLs that it requires is much bigger than the size of the statically linked program. The linker includes only the code actually used by the program, whereas a package must link in all the functions and classes declared in the interface sections of all the units contained in the package.
- If you distribute several Delphi applications based on the same packages, you might end up distributing less code, because the run-time packages are shared. In other words, once the users of your application have the standard Delphi run-time packages, you can ship them very small programs.
- If you run several Delphi applications based on the same packages, you can save some memory space at run time; the code of the run-time packages is loaded in memory only once between the multiple Delphi applications.
- Don't worry too much about distributing a large executable file. Keep in mind that when you make minor changes to a program, you can use any of various tools to create a *patch file*, so that you distribute only a file containing the differences, not a complete copy of the files.
- If you place a few of your program's forms in a run-time package, you can share them among programs. When you modify these forms, however, you'll generally need to recompile the main program as well, and distribute both of them again to your users. The next section discusses this complex topic in detail.

## Package Versioning

A very important and often misunderstood element is the distribution of updated packages. When you update a DLL, you can ship the new version, and the executable programs requiring this DLL will generally still work (unless you've removed existing exported functions or changed some of their parameters).

When you distribute a Delphi package, however, if you update the package and modify the interface portion of any unit of the package, you might need to recompile all the applications that use the package. This is required if you add methods or properties to a class, but not if you add new global symbols (or modify anything not used by client applications). There is no problem at all for changes affecting only the implementation section of the package's units.

A DCU file in Delphi has a version tag based on its timestamp and a checksum computed from the interface portion of the unit. When you change the interface portion of a unit, every other unit based on it should be recompiled. The compiler compares the timestamp and checksum of the unit of previous compilations with the new timestamp and checksum, and decides whether the dependent unit must be recompiled. This is why you have to recompile each unit when you get a new version of Delphi, which has modified system units.

A package is a collection of units. In Delphi 3, a checksum of the package, obtained from the checksum of the units it contains and the checksum of the packages it requires, was added as an extra entry function to the package library, so that any executable based on an older version of the package would fail at startup.

Delphi 4 and following versions have relaxed the run-time constraints of the package. The design-time constraints on DCU files remain identical, though. The checksum of the packages is not checked anymore, so you can directly modify the units that are part of a package and deploy a new version of the package to be used with the existing executable file. Since methods are referenced by name, you cannot remove any existing method. You cannot even change its parameters, because of name-mangling techniques specifically added to the packages to protect against changes in parameters.

Removing a method referenced from the calling program will stop the program during the loading process. If you make other changes, however, the program might fail unexpectedly during its execution. For example, if you replace a component placed on a form compiled in a package with a similar component, the calling program might still be able to access the one in that memory location, although it is now a different component!

If you decide to follow this treacherous road of changing the interface of units in a package without recompiling all the programs that use it, you should at least limit your changes. When you add new properties or nonvirtual methods to the form, you should be able to maintain full compatibility with existing programs already using the package. Also, adding fields and virtual

methods might affect the internal structure of the class, leading to problems with existing programs that expect a different class data and virtual method table (VMT) layout. Of course, this applies to the binary compatibility between the EXE and the BPL (Borland Package Library).

**WARNING**

Here I'm referring to the distribution on compiled programs divided between EXE and packages, not to the distribution of components to other Delphi developers. In this latter case the versioning rules are more stringent, and you must take extra care in package versioning.

Having said this, I recommend never changing the interface of any unit exported by your packages. To accomplish this, you can add to your package a unit with form-creation functions (as in the DLL with forms presented earlier) and use it to access another unit, which defines the form. Although there is no way to *hide* a unit that is linked into a package, if you never directly use the class defined in a unit, but use it only through other routines, you'll have more flexibility in modifying it. You can also use form inheritance to modify a form within a package without really affecting the original version.

The most stringent rule for packages is the following one used by component writers: For long-term deployment and maintenance of code in packages, plan on having a major release with minor maintenance releases. A major release of your package will require all client programs to be recompiled from source; the package file itself should be renamed with a new version number, and the interface sections of units can be modified. Maintenance releases of that package should be restricted to implementation changes to preserve full compatibility with existing executables and units.

## Forms Inside Packages

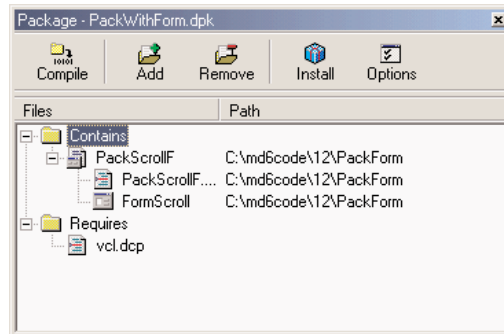
We've already discussed (in Chapter 11, "Creating Components") the use of component packages in Delphi applications. As I'm discussing the use of packages and DLLs for partitioning an application, here I'll start discussing the development of packages holding forms. We've seen earlier in this chapter that you can use forms inside DLLs, but this sometimes causes a few problems. If you are building both the library and the executable file in Delphi, using packages results in a much better and cleaner solution.

At first sight, you might believe that Delphi packages are a way to distribute components to be installed in the environment. Instead, you can use packages as a way to structure your code but, unlike DLLs, retain the full power of Delphi's OOP. Consider this: A package is a collection of compiled units and your program uses several units. The units the program refers to will be compiled inside the executable file, unless you ask Delphi to place them inside a package.

So how do you set up an application so that its code is split among one or more packages and a main executable file? You only need to compile some of the units in a package and then set up the options of the main program to dynamically link this package. For example, I've made a copy of the “usual” color selection form and renamed its unit as `PackScrollF`, then I've created a new package and added the unit to it, as you can see in Figure 12.7.

**FIGURE 12.7:**

The structure of the package hosting a form in Delphi's Package Editor



Before compiling this package, you should change its default output directories to refer to the current folder, not the standard `/Projects/Bp1` subfolder of Delphi. To do this, go to the `Directories/Conditional` page of the package Project Options, and set the current directory (a single dot, for short) for the Output directory (for the BPL) and DCP output directory. Then compile the package and do not install it in Delphi—there's no need to.

At this point, you can create a normal application and write the standard code you'll use in a program to show a secondary form, as in the following listing:

```

uses
    PackScrollF;

procedure TForm1.BtnChangeClick(Sender: TObject);
var
    FormScroll: TFormScroll;
begin
    FormScroll := TFormScroll.Create (Application);
    try
        // initialize the data
        FormScroll.SelectedColor := Color;
        // show the form
        if FormScroll.ShowModal = mrOK then
            Color := FormScroll.SelectedColor;
    finally
        FormScroll.Free;
    end;
end;

```

```
procedure TForm1.BtnSelectClick(Sender: TObject);  
var  
    FormScroll: TFormScroll;  
begin  
    FormScroll := TFormScroll.Create (Application);  
    // initialize the data and UI  
    FormScroll.SelectedColor := Color;  
    FormScroll.BitBtn1.Caption := 'Apply';  
    FormScroll.BitBtn1.OnClick := FormScroll.ApplyClick;  
    FormScroll.BitBtn2.Kind := bkClose;  
    // show the form  
    FormScroll.Show;  
end;
```

One of the advantages of this approach is that you can refer to a form compiled into a package with the exact same code you'll use for a form compiled in the program. In fact, if you simply compile this program, the unit of the form will actually be bound to it. To keep it in the package, you'll have to use run-time packages for the application and manually add the `PackWithForm` package to the list of run-time packages (this is not suggested by the Delphi IDE as we have not installed the package in the development environment).

Once you've done this step, compile the program and it will behave exactly as usual. But now the form is in a DLL package, and you can even modify the form in the package, recompile it, and simply run the application to see the effects. Notice, though, that for most changes affecting the interface portion of the units of the package (for example, adding a component or a method to the form), you should also recompile the executable program calling the package.

**NOTE**

You can find the package and the program testing it in the `PackForm` folder of the source code related to the current chapter. The code of the next example is in the same folder.

## Loading Packages at Run Time

In the example above, I indicated that the `PackWithForm` package is a run-time package to be used by the application. This means that the package is required to run the application and is loaded when the program starts. Both aspects can be avoided by loading the package dynamically, as we've done with DLLs. The resulting program will be more flexible, start more quickly, and use less memory.

An important element to keep in mind is that you'll need to call the `LoadPackage` and `UnloadPackage` Delphi functions rather than the `LoadLibrary` and `FreeLibrary` Windows API functions. The difference is that the functions provided by Delphi load the packages, but also call their proper initialization and finalization code.

Besides this important element—easy to accomplish once you know about it—the program will require some extra code, as we cannot refer from the main program to the unit hosting the form. We cannot use the form class directly, nor access to its properties or components. At least not with the standard Delphi code. Both issues, however, can be solved using class references, class registration, and RTTI (run-time type information). Let me start with the first one. In the form unit, in the package, I've added this initialization code:

```
initialization  
  RegisterClass (TFormScroll);
```

As the package is loaded, the main program can use Delphi's `GetClass` function to get the class reference of the registered class and then call the `Create` constructor for this class reference.

To solve the second problem, I've made the `SelectedColor` property of the form in the package a published property, so that it is accessible via RTTI. Then I've replaced the code accessing this property (`FormScroll.Color`) with the following:

```
SetPropValue (FormScroll, 'SelectedColor', Color);
```

Summing up all of these changes, here is the code used by the main program (the `DynaPackForm` application) to show the modal form from the dynamically loaded package:

```
procedure TForm1.BtnChangeClick(Sender: TObject);  
var  
  FormScroll: TForm;  
  FormClass: TFormClass;  
  HandlePack: HModule;  
begin  
  // try to load the package  
  HandlePack := LoadPackage ('PackWithForm.bpl');  
  if HandlePack > 0 then  
  begin  
    FormClass := TFormClass(GetClass ('TFormScroll'));  
    if Assigned (FormClass) then  
    begin  
      FormScroll := FormClass.Create (Application);  
      try  
        // initialize the data  
        SetPropValue (FormScroll, 'SelectedColor', Color);  
        // show the form  
        if FormScroll.ShowModal = mrOK then  
          Color := GetPropValue (FormScroll, 'SelectedColor');  
      finally  
        FormScroll.Free;  
      end;  
    end  
  else  
    ShowMessage ('Form class not found');
```



```
    UnloadPackage (HandlePack);  
end  
else  
    ShowMessage ('Package not found');  
end;
```

Notice that the program unloads the package as soon as it is done with it. This is not compulsory. I could have moved the `UnloadPackage` call in the `OnDestroy` handler of the form, and avoided reloading the package after the first time.

Now you can try running this program without the package available, and you'll see that it will start properly, only to complain it cannot find the package as you click the `Change` button. In this program, you don't need to use run-time packages to keep the unit outside of your executable file, as you are not referring to the unit in your code. Also, the `PackWithForm` package doesn't need to be listed in the run-time packages. However, you must use run-time packages, or else your program will include the VCL global variables (as the `Application` object) and the dynamically loaded package will include another version, because it will refer to the VCL packages anyway.

## Using Interfaces in Packages

Accessing the classes of the forms by means of methods and properties is much simpler than using RTTI all over the place. To build a larger application, I definitely try to use interfaces and to have multiple forms, each implementing a few standard interfaces defined by the program. An example cannot really do justice to this type of architecture, which becomes relevant for a large program, but I've tried nonetheless to build a program to show how this idea can be applied in practice.

### NOTE

If you don't know much about interfaces, I suggest you to refer to the related portion of Chapter 3 before reading this section.

To architect the `IntfPack` project, I've used three packages plus a demo application. Two of the three packages (called `IntfFormPack` and `IntfFormPack2`) define alternative forms used to select a color. The third package (called `IntfPack`) hosts a shared unit, used by both other packages. This unit basically includes the definition of the interface. I couldn't add it to both other packages because you cannot load two packages with a unit having the same name (even by run-time loading).

The only file of the `IntfPack` package is the `IntfColSel` unit, displayed in Listing 12.1. This unit defines the common interface (and you'll probably have a number of them in real-world application) plus a list of registered classes, which mimics Delphi's `RegisterClass` approach, but makes available the complete list so that you can easily scan it.

## Listing 12.1: The IntfColSel unit of the IntfPack package

```

unit IntfColSel;

interface

uses
    Graphics, Contnrs;

type
    IColorSelect = interface
        ['{3F961395-71F6-4822-BD02-3B475FF516D4}']
        function Display (Modal: Boolean = True): Boolean;
        procedure SetSelColor (Col: TColor);
        function GetSelColor: TColor;
        property SelColor: TColor
            read GetSelColor write SetSelColor;
    end;

procedure RegisterColorSelect (AClass: TClass);

var
    ClassesColorSelect: TClassList;

implementation

procedure RegisterColorSelect (AClass: TClass);
begin
    if ClassesColorSelect.IndexOf (AClass) < 0 then
        ClassesColorSelect.Add (AClass);
end;

initialization
    ClassesColorSelect := TClassList.Create;

finalization
    ClassesColorSelect.Free;

end.

```

Once we have this interface available, we can define forms that implement it, as in the following example, taken from the IntfFormPack:

```

type
    TFormSimpleColor = class(TForm, IColorSelect)
        ...
    private
        procedure SetSelColor (Col: TColor);
        function GetSelColor: TColor;
    public
        function Display (Modal: Boolean = True): Boolean;

```

The two access methods simply read and write the value of the color from some components of the form (a simple `ColorGrid` in this specific case), while the `Display` method internally calls either `Show` or `ShowModal`, depending on the parameter:

```
function TFormSimpleColor.Display(Modal: Boolean): Boolean;
begin
    Result := True; // default
    if Modal then
        Result := (ShowModal = mrOK)
    else
        begin
            BitBtn1.Caption := 'Apply';
            BitBtn1.OnClick := ApplyClick;
            BitBtn2.Kind := bkClose;
            Show;
        end;
    end;
```

The form is structured like that of the last example, still available in the second package, and has an OK button that is turned into an Apply button. Finally, the unit has the registration code in the initialization section, so that it is executed when the package is dynamically loaded:

```
RegisterColorSelect (TFormSimpleColor);
```

With this architecture in place, we can build a rather elegant and flexible main program, which is based on a single form. When the form is created, it defines a list of packages (called `HandlesPackages`) and loads them all. I've hard-coded the package in the code of the example, but of course you can as well search for the packages of the current folder or use a configuration file to make the application structure more flexible. Finally, after loading the packages, the program shows the registered classes in a list box. This is the code of the `LoadDynaPackage` and `FormCreate` methods:

```
procedure TFormUseIntf.FormCreate(Sender: TObject);
var
    I: Integer;
begin
    // loads all runtime packages
    HandlesPackages := TList.Create;
    LoadDynaPackage ('IntfFormPack.bp1');
    LoadDynaPackage ('IntfFormPack2.bp1');

    // add class names and select the first
    for I := 0 to ClassesColorSelect.Count - 1 do
        lbClasses.Items.Add (ClassesColorSelect [I].ClassName);
        lbClasses.ItemIndex := 0;
    end;
```

```
procedure TFormUseIntf.LoadDynaPackage(PackageName: string);  
var  
    Handle: HModule;  
begin  
    // try to load the package  
    Handle := LoadPackage (PackageName);  
    if Handle > 0 then  
        // add to the list for later removal  
        HandlesPackages.Add (Pointer(Handle))  
    else  
        ShowMessage ('Package ' + PackageName + ' not found');  
end;
```

The main reason for keeping the list of package handles is to be able to unload them all when the program ends. In fact, we don't need these handles to access the forms defined in those packages. The run-time code used to create and show a form simply uses the corresponding component classes. This is a snippet of code used to display a modeless form (an option controlled by a check box):

```
var  
    AComponent: TComponent;  
    ColorSelect: IColorSelect;  
begin  
    AComponent := TComponentClass  
        (ClassesColorSelect[LbClasses.ItemIndex]).Create (Application);  
    ColorSelect := AComponent as IColorSelect;  
    ColorSelect.SelectColor := Color;  
    ColorSelect.Display (False);
```

The program actually uses the `Supports` function to check that the form really does support the interface before using it, and also accounts for the modal version of the form, but its essence is properly depicted in the four statements above. By the way, notice that the code doesn't actually require a form. A nice exercise would be to add to the architecture a package with a component encapsulating the color selection dialog box or inheriting from it.

---

**WARNING** The main program refers to the unit hosting the interface definition but should not link this file in. Rather, it should use the run-time package containing this unit, as the dynamically loaded packages do. Otherwise the main program will use a different copy of the same code, including a different list of global classes. It is this list of global classes, not the use of the same interface, that should not be duplicated in memory.

## Packages Versus DLLs

In the preceding section, we've seen that using packages is a fine alternative to using DLLs for sharing compiled code among multiple Delphi applications or splitting a large executable into multiple (and partially independent) modules. As a summary, here are a few of the differences between the two approaches:

- DLLs are collections of functions; packages can easily “export” classes and objects.
- Dynamically loading a DLL implies losing any safety in the function call, in case you pass the wrong parameters. Dynamically loading packages requires some extra coding as well, but is definitely simpler and safer, particularly if you use interfaces.
- Packages force you to use the VCL run-time package for the application, although even when using DLLs, run-time packages help solve quite a few difficulties (as we'll discuss shortly).
- DLLs can be used across programming languages and development environments, but packages are limited to Delphi and C++Builder. If you need libraries in a Delphi-only environment, packages are the native solution and should generally be preferred.

Using DLLs also accounts for a few extra troubles that you can partially solve by letting the DLLs share run-time packages. The following sections discuss the problem briefly, as this is not the recommended approach anyway.

## Executables and DLLs Sharing the VCL Packages

In the FormDLL example, we faced a problem: When you place forms inside a DLL, you don't get the proper behavior for the flat buttons even if you synchronize the two application objects. Moreover, both the executable file and the DLL contain the compiled code of the VCL library, leading to useless duplication. As discussed earlier, the simplest solution to this issue is to use a package instead of a DLL.

Another solution is to keep the DLL in its format, but let it use run-time packages, so that no global objects will be duplicated between the executable and the library. In this case there will be only one `Application` object, shared by the program and the DLL, instead of two separate objects, so we don't need the synchronization code any more.

Another simplification to the program comes from the fact that the modeless form inside the DLL can communicate back to the main form by accessing the list of the forms (available to the shared global `Screen` object) or simply using the `Application.MainForm` property. This is what I've done in the FormDllP example on the CD.

With this approach, you face the risk of having the main form and the form in the DLL not synchronized at all, with two entries in the Taskbar; also, this code still has all the other

problems of the first version of the FormDLL example. The problem lies in the fact that when you run the program, the DLL is initialized before the application, so it is the DLL that initializes the Forms unit of the VCL. Within a DLL, the VCL creates the Application object but doesn't create the corresponding window.

There are two radically different approaches to this initialization issue: One is to change the initialization order by loading the DLL dynamically after the application has started; the second is to add some extra initialization code in the program. None of these techniques provides a better solution than using packages altogether!

## Dynamically Loading the DLL with Packages

The first solution is demonstrated by the FirstDLLD library and the UseDyna example, which dynamically loads the DLL built with run-time packages. The main program loads the DLL at startup, in the OnCreate event handler of the form:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    hInstDll := SafeLoadLibrary ('FormDllD.dll');  
    if hInstDll <= 0 then  
        raise Exception.Create ('FormDllD library not found');  
end;
```

In the program I haven't declared the functions exported by the DLL, to avoid the implicit link of the library. Instead I've declared two procedure types:

```
type  
    TGetColorProc = function (Col: LongInt): LongInt; stdcall;  
    TShowColorProc = procedure (Col: LongInt); stdcall;
```

These types are used for converting the generic pointer returned by the GetProcAddress function, as we've already seen in the DynaCall example:

```
procedure TForm1.BtnChangeClick(Sender: TObject);  
var  
    Col: LongInt;  
    GetColorProc: TGetColorProc;  
    FPointer: TFarProc;  
begin  
    FPointer := GetProcAddress (hInstDll, 'GetColor');  
    if FPointer = nil then  
        raise Exception.Create ('GetColor DLL function not found');  
    GetColorProc := TGetColorProc (FPointer);  
    // original code  
    Col := ColorToRGB (Color);  
    Color := GetColorProc (Col);  
end;
```

Using dynamic loading is the correct approach, officially supported by Delphi. Still, you have to call the functions dynamically, which requires a little extra coding.

## Fixing the Initialization Code

An alternate solution is to keep the external functions defined in the main program, let the DLL start first and initialize the VCL, and let the VCL create the `Application` object without the connected window. In fact, we can add one line of code to the library to ask for the creation of the window of the `Application` object during the library initialization process (before the executable creates its own main objects). We accomplish this by writing the code in the initialization section of one of the units of the DLL:

```
initialization  
  Application.CreateHandle;
```

Because this code is in the DLL, the application fails to load its icon. The solution is actually very simple. In the `OnCreate` event handler of the main form (in the main program), simply reload the current icon:

```
Application.Icon.Handle := LoadIcon (HInstance, 'MAINICON');
```

## Exploring the Structure of a Package

You might wonder: is it possible to know whether a unit has been linked in the executable file or if it's part of a run-time package? Not only is this possible in Delphi, but you can also explore the overall structure of an application. A component can use the undocumented `ModuleIsPackage` global variable, declared in the `SysInit` unit. You should never need this, but it is technically possible for a component to have different code depending on whether it is packaged or not. The following code extracts the name of the run-time package hosting the component, if any:

```
var  
  fPackageName: string;  
begin  
  // get package name  
  SetLength (fPackageName, 100);  
  if ModuleIsPackage then  
    begin  
      GetModuleFileName (HInstance, PChar (fPackageName), Length (fPackageName));  
      fPackageName := PChar (fPackageName) // string length fixup  
    end  
  else  
    fPackageName := 'Not packaged';
```

Besides accessing package information from within a component (as in the code above), you can also do so from a special entry point of the package libraries, the `GetPackageInfoTable`

function. This function returns some specific package information that Delphi stores as resources and includes in the package DLL. Fortunately, we don't need to use low-level techniques to access this information, since Delphi provides some high-level functions to manipulate it.

You can use two functions to access package information:

- `GetPackageDescription` returns a string that contains a description of the package. To call this function, you must supply the name of the module (the package library) as the only parameter.
- `GetPackageInfo` doesn't directly return information about the package. Instead, you pass it a function that it calls for every entry in the package's internal data structure. In practice, `GetPackageInfo` will call your function for every one of the package's contained units and required packages. In addition, `GetPackageInfo` sets several flags in an Integer variable.

These two function calls allow us to access internal information about a package, but how do we know which packages our application is using? You could determine this by looking at an executable file using low-level functions, but Delphi helps you again by supplying a simpler approach. The `EnumModules` function doesn't directly return information about an application's modules but allows you to pass it a function, which it calls for each module of the application, the main executable file, and for each of the packages the application relies on.

To demonstrate this approach, I've built a simple example program that displays the module and package information in a `TreeView` component. Each first-level node corresponds to a module, and within each module I've built a subtree that displays the contained and required packages for that module, as well as the package description and compiler flags (`RunOnly` and `DesignOnly`). You can see the output of this example in Figure 12.8.

In addition to the `TreeView` component, I've added several other components to the main form, but hidden them from view: a `DBEdit`, a `Chart`, and a `FilterComboBox`. I added these components simply to include more run-time packages in the application, beyond the ubiquitous `VCL60.BPL`. The only method of the form class is `FormCreate`, which calls the module enumeration function:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    EnumModules(ForEachModule, nil);  
end;
```



**FIGURE 12.8:**

The output of the PackInfo example, with the details of the packages it uses



The `EnumModules` function accepts two parameters. The first is the callback function (in our case, `ForEachModule`), and the second is a pointer to a data structure that the callback function will use (in our case, `nil`, since we didn't need this). The callback function must accept two parameters—an `HInstance` value and an untyped pointer—and must return a `Boolean` value. The `EnumModules` function will, in turn, call our callback function for each module, passing the instance handle of each module as the first parameter and the data structure pointer (`nil` in our example) as the second.

```
function ForEachModule (HInstance: Longint;
    Data: Pointer): Boolean;
var
    Flags: Integer;
```

```

ModuleName, ModuleDesc: string;
ModuleNode: TTreeNode;
begin
  with Form1.TreeView1.Items do
    begin
      SetLength (ModuleName, 200);
      GetModuleFileName (HInstance,
        PChar (ModuleName), Length (ModuleName));
      ModuleName := PChar (ModuleName); // fixup
      ModuleNode := Add (nil, ModuleName);

      // get description and add fixed nodes
      ModuleDesc := GetPackageDescription (PChar (ModuleName));
      ContNode := AddChild (ModuleNode, 'Contains');
      ReqNode := AddChild (ModuleNode, 'Requires');

      // add information if the module is a package
      GetPackageInfo (HInstance, nil, Flags, ShowInfoProc);
      if ModuleDesc <> '' then
        begin
          AddChild (ModuleNode, 'Description: ' + ModuleDesc);
          if Flags and pfDesignOnly = pfDesignOnly then
            AddChild (ModuleNode, 'Design Only');
          if Flags and pfRunOnly = pfRunOnly then
            AddChild (ModuleNode, 'Run Only');
        end;
      end;
      Result := True;
    end;
end;

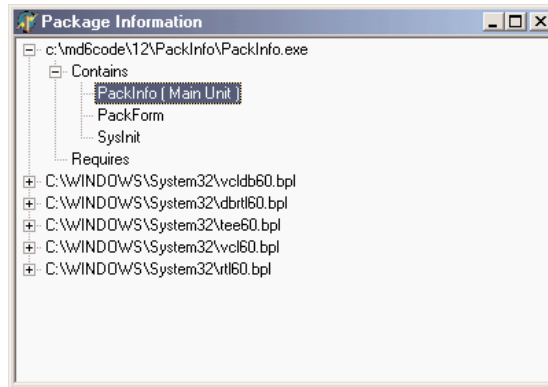
```

As you can see in the preceding code, the `ForEachModule` function begins by adding the module name as the main node of the tree (by calling the `Add` method of the `TreeView1.Items` object and passing `nil` as the first parameter). It then adds two fixed child nodes, which are stored in the `ContNode` and `ReqNode` variables declared in the implementation section of this unit.

Next, the program calls the `GetPackageInfo` function and passes it another callback function, `ShowInfoProc`, which I'll discuss shortly. The program adds the details for the main module (see Figure 12.9), simply because this will provide a list of the application's units. At the end of this function, we add more information if the module is a package, such as its description and compiler flags (we know it's a package if its description isn't an empty string).

**FIGURE 12.9:**

The PackInfo example also lists the units that are part of the current application.



Earlier, I mentioned passing another callback function, the ShowInfoProc procedure, to the GetPackageInfo function, which in turn calls our callback function for each contained or required package of a module. This procedure creates a string that describes the package and its main flags (added within parentheses), and then inserts that string under one of the two nodes (ContNode and ReqNode), depending on the type of the module. We can determine the module type by examining the NameType parameter. Here is the complete code of our second callback function:

```

procedure ShowInfoProc (const Name: string; NameType: TNameType; Flags: Byte;
    Param: Pointer);
var
    FlagStr: string;
begin
    FlagStr := ' ';
    if Flags and ufMainUnit <> 0 then
        FlagStr := FlagStr + 'Main Unit ';
    if Flags and ufPackageUnit <> 0 then
        FlagStr := FlagStr + 'Package Unit ';
    if Flags and ufWeakUnit <> 0 then
        FlagStr := FlagStr + 'Weak Unit ';
    if FlagStr <> ' ' then
        FlagStr := ' (' + FlagStr + ')';
    with Form1.TreeView1.Items do
        case NameType of
            ntContainsUnit: AddChild (ContNode, Name + FlagStr);
            ntRequiresPackage: AddChild (ReqNode, Name);
        end;
    end;
end;

```

Here, you'll notice that the Flags parameter doesn't contain flag style information, as the online help seems to imply. If you want to investigate this topic further, examine the SysUtils unit.

## What's Next?

In this chapter we have seen how you can call functions that reside in DLLs, how to create DLLs using Delphi, and how to use strings and place Delphi forms inside a library. Another technique for placing Delphi forms and other classes in libraries is to use packages, special DLLs that the IDE uses for installing components, but that you can use for dividing an application into multiple executable files.

I will get back to the topics of libraries that expose objects and classes when I discuss COM and OLE in Chapters 19 and 20. For the moment, instead, we'll move to a totally different topic, the development of database-oriented and client/server applications with Delphi.



**PART III**

---

# **Database Programming**

- **Chapter 13: Delphi's Database Architecture**
- **Chapter 14: Client/Server Programming**
- **Chapter 15: InterBase and IBX**
- **Chapter 16: ActiveX Data Objects**
- **Chapter 17: Multitier Database Applications with DataSnap**
- **Chapter 18: Writing Database Components**

# Delphi's Database Architecture

- Delphi's database components
- Database access alternatives
- Using data-aware controls
- The DBGrid and multirecord objects
- Manipulating table fields
- Database applications with standard controls

**D**elphi's support for database applications is one of the key features of the programming environment. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming.

What you won't find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports database access.

I'll begin with an explanation of the alternatives Delphi offers in terms of data access, and then I'll provide an overview of the database components that are available in Delphi. This chapter includes an overview of the `TDataSet` class, an in-depth analysis of the `TField` components, and the use of data-aware controls. The following chapters will provide information on more advanced database programming topics, such as client/server programming, the use of `dbGo`, `dbExpress`, and `InterBase Express`.

## Accessing a Database: BDE, dbExpress, and Other Alternatives

In the first few versions of Delphi, the only available technology to access database data was to use the Borland Database Engine (BDE). Starting with Delphi 3, the portion of VCL related to Database access has been restructured to open it up to multiple database access solutions. Delphi 5 saw the introduction of specific sets of components supporting Microsoft's ActiveX Data Objects (ADO) and `InterBase Express` (IBX). Delphi 6 adds to the picture `dbExpress`, which is a brand-new cross-platform and database-independent data-access technology provided by Borland with `Kylix` on Linux and Delphi 6 on Windows.

With all these alternatives, it is easy to get confused on which approach to use. In the following sections I've provided a short description of the key elements of these data-access technologies available in Delphi, trying to suggest in which case you'll want to use each of them.

### Borland Database Engine (BDE)

The BDE originated with `Paradox`, well before Delphi existed, and was extended by Borland to support other local databases and many SQL servers. The BDE has direct access to `dBASE`, `Paradox`, `ASCII`, `FoxPro`, and `Access` tables. A series of drivers (called SQL Links and available only in Delphi Enterprise) allows access to some SQL servers, including `Oracle`, `Sybase`, `Microsoft`, `Informix`, `InterBase`, and `DB2` servers. If you need access to a different database, the BDE can also interface with `ODBC` drivers.

The advantage of using a common database engine is that your application will be portable among different servers of the same category (porting from a local database to an SQL server is generally much more complex). The specific advantages of using the BDE are that this technology is very well integrated in Delphi; its elements are very well documented; and it is the only viable solution for accessing local files such as Paradox and dBase tables.

The disadvantages of this solution: Borland has stopped developing it (there will be no further updates); you'll have to install and configure it on the client computers; it is quite a "heavyweight" engine, with large installation files and memory requirements; and it is available only on Windows. If you have existing Delphi BDE applications accessing local files, there is no hurry to convert them and get rid of the BDE, unless you want to move your applications to Linux. If you are using an SQL server, migrating to another data-access technology will probably be easier.

BDE is still a good solution, if you balance advantages and disadvantages, but its long-term viability is certainly in doubt. I'll keep using the BDE for the simpler examples of this chapter, but only for the sake of simplicity. In any case, I'll try to stress the elements common to all the dataset components rather than focus on specific features of BDE or Paradox.

The Delphi components related to the BDE are all hosted in the Data Access page of the Components palette. There are three dataset components, Table, Query, and StoredProc, plus the UpdateSQL used in connection with the Query component. The Database and Session components are used to set up the database connection. The BatchMove component is for copying data; the rarely used NestedTable component allows you to nest master-detail data in a sub-table; and the BDEClientDataSet component, introduced in Delphi 6, merges a ClientDataSet with a BDE-related data-access component.

## ActiveX Data Objects (ADO)

ADO, which stands for ActiveX Data Objects, is Microsoft's high-level interface for database access. ADO is implemented on Microsoft's data-access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. ADO is an engine with features comparable to the BDE: database server independence supporting local and SQL servers alike, a really heavyweight engine, and a simplified configuration (because it is not centralized). Installation should in theory not be an issue, as the engine is part of recent versions of Windows. However, the limited compatibility among versions of ADO will force you to upgrade your users' computers to the same version you've used for developing the program—and the sheer size of the MDAC (Microsoft Data Access Components) installation, which updates large portions of the operating system, makes this operation far from simple.



ADO offers some definite advantages if you plan on using Access or SQL Server, as Microsoft's drivers for their own databases are of better quality than the average OLE DB providers. For Access databases, specifically, using Delphi's ADO components is a good solution. But if you plan using other SQL servers, first check the availability of a good quality driver, as you might have some surprises. ADO is very powerful, but you have to learn living with it, as it really stands in the way between your program and the database, providing services but occasionally also issuing different commands than you are expecting. On the negative side, do not even think of using ADO if you plan future cross-platform development: this Microsoft-specific technology is not available on Linux or other operating systems.

In short, use ADO if you plan working only on Windows, want to use Access or other Microsoft databases, or you find a good OLE DB provider for each of the database servers you plan working with (at the moment, for example, this excludes InterBase and many other SQL servers).

ADO components (part of a package Borland called ADO Express in Delphi 5 and now calls dbGo in Delphi 6) are all grouped in the ADO page of the Components palette. The three core components are ADOConnection (for database connection), ADOCommand (for executing SQL commands), and ADODataset (for executing requests that return a result set). There are also three compatibility components—ADOTable, ADOQuery, and ADOSToredProc—which you can use for porting BDE-based applications to ADO. Finally, there is the RDSConnection component, for accessing data in remote multitier applications.

**NOTE**

Chapter 16, "ActiveX Data Objects," covers ADO and related technologies in great detail.

## The dbExpress Library

One of the relevant new features of Delphi 6 is the introduction of the dbExpress database library for the Windows platform. I say "library" because, unlike BDE and ADO, dbExpress uses a lightweight approach; and I underline "Windows" because the same library is available also for Linux in Borland Kylix.

Being light and portable are actually the two key characteristics of dbExpress and the reasons it has been introduced by Borland, along with the development of the Kylix project. There are certainly other database libraries you could use in the past and can still use with Delphi, but this new offering is worth a thought. Consider also it requires basically no configuration on the user machines.

Compared to other powerhouses, dbExpress is really limited in its capabilities. It can access only SQL servers (no local files); it has no caching capabilities and provides only uni-directional access to the data; it can natively work only with SQL queries and is unable of generating the corresponding SQL update statements.

At first sight, you might think that these limitations make the library pretty useless. On the contrary, these are *features* that make it interesting. Unidirectional datasets with no direct update are the norm if you need to produce reporting, including generating HTML pages showing the content of a database. If you want to build a user interface to edit the data, instead, consider that Delphi includes specific components (the ClientDataSet and Provider, in particular) that provide caching and query resolution. These components allow your dbExpress-based application to have much more control than you can have with a separate (and monolithic) database engine, which does extra things for you but often does it the way it wants, not the way you would like.

Considering that Borland is pushing this library, and it is the only viable database-independent solution on Linux, I really urge you to consider it for new applications, and even to think about updating existing Delphi applications to this new architecture.

## InterBase Express (IBX)

Delphi includes components for native access to Borland's own open-source (and free) InterBase server. Unlike BDE, ADO, and dbExpress, this is not a server-independent database engine, but a technology for accessing a specific database server. If you plan using only InterBase as your back-end RDBMS, using a specific set of components can give you more control of the server, provide the best performance, and allow you also to configure and maintain the server from within a custom client application.

### NOTE

The use of InterBase Express highlights the case of database-specific custom datasets, which are available from third-party vendors for many servers (there are other dataset components for InterBase, as there are for Oracle, Access, dBase files, and many others).

In short, you can consider using InterBase Express (or other comparable sets of components) if you are sure you won't change your database and want to achieve best performance and control at the expense of flexibility and portability. The down side is that the extra performance and control you gain might be limited, and you'll have to learn how to use another set of components with a specific behavior, compared to learning how to use a generic engine and applying your knowledge to different situations.

## The ClientDataSet Component

Finally, there is a component derived from TDataSet that has a peculiar behavior and can be combined with other data-access components. The ClientDataSet component, in fact, is a dataset accessing data kept in memory. The in-memory data can be totally temporary (lost as you exit the program), saved to a local file as a snapshot, and imported by another dataset using a Provider component. This last situation is certainly the most common: You can hook a ClientDataSet to any other local dataset, or use Borland's multitier support (discussed in

Chapter 17, “Multitier Database Applications with DataSnap”) to retrieve data from a dataset hosted by a different application, possibly running on a separate computer.

The ClientDataSet component becomes particularly useful if the data-access components you are using provide limited or no caching. This is particularly true of the new dbExpress engine, but can equally help you when using the BDE or other native components. On the other hand, ADO already provides most of the services of the ClientDataSet component and using these two at the same time can be useful only in limited situations.

## Classic BDE Components

Each of the database-access solutions discussed above has its own set of data-access, database connection, and extra utility components on a specific page of the Component palette. In Delphi 6, the classic BDE components have been moved to the new BDE page and include the Table, Query, and StoredProc components. The ADO, dbExpress, and InterBase Express components are each in specific pages, and all include specific dataset components and others that tend to mimic the BDE components, simplifying the porting of existing applications. The Data Access page of the Component palette in Delphi 6 includes only the Data Source component and others not specifically related with any single data access technology.

Besides the data-access component of your choice, a Delphi visual application generally uses some data-aware controls (in the Data Controls page) and the DataSource component. Data-aware controls are visual components used to view and edit the data in a form and are extensions of standard components such as edit and list boxes, radio buttons, images, and the grid. The DataSource component has the role of connector between the data-aware controls and a dataset component.

## Tables and Queries

The simplest traditional way to specify data access in Delphi was to use the BDE Table component. A Table object simply refers to a database table. When you use a Table component, you need to indicate the name of the database you want to use in its DatabaseName property. You can enter an alias or the path of the directory with the table files. The Object Inspector lists the available names, which depend on the aliases installed in the BDE.

You also need to indicate a proper value in the TableName property. The Object Inspector lists the available tables of the current database (or directory), so you should generally select the DatabaseName property first.

Another classic dataset is the BDE Query component. A query requires a SQL language command. You can customize a query using SQL more easily than you can customize a table (as long as you know at least the basic elements of SQL, of course). The Query component

has a `DatabaseName` property like the `Table` component, but it does not have a `TableName` property. The table is indicated in the SQL statement, stored in the `SQL` property.

For example, you can write a simple SQL statement like this:

```
select * from Country
```

where `Country` is the name of a table and the asterisk (\*) indicates that you want to use all of the fields in the table.

The efficiency of a table or a query varies depending on the database you are using. In general, we can say that the `Table` component tends to be faster on local tables, while the `Query` component tends to be faster on SQL servers, although this is just a very general rule, and in many cases you might have the opposite effect. We'll see some efficiency issues while discussing client/server development in Chapter 14, "Client/Server Programming."

The third BDE dataset component is `StoredProc`, which refers to stored procedures of a SQL server database. You can run these procedures and get the results in the form of a database table. Stored procedures can only be used with SQL servers.

## Specific Table Features

The BDE `Table` component has specific features not shared by all datasets. For example, it has filters, ranges, and specific techniques for locating records. A filter, set in the `Filter` property and activated by toggling the `Filtered` property, is available in each dataset, although its role changes depending on the underlying implementation. A range, instead, is specific to a `Table` and allows you to specify the two extreme values and consider only the record falling within that interval.

When using a `Table`, and particularly a local one, there are specific methods you can use to find a record, such as `GotoKey`, `FindKey`, `GotoNearest`, `FindNearest`, and `Locate`. The `Locate` method is shared by all datasets, and I'll discuss it later along with other general features of the `TDataSet` class. The other methods are specific of the `TTable` class and work in conjunction with the index set in the `IndexFieldNames` property of the component.

The simplest approach is to use the `FindNearest` method for the approximate search and the `FindKey` method to look for an exact match:

```
// goto
Table1.FindNearest ([EditName.Text]);

// go near
if not Table1.FindKey ([EditName.Text]) then
    MessageDlg ('Name not found', mtError, [mbOk], 0);
```

Both find methods use as parameters an array of constants. Each array element corresponds to one of the fields of the current index. You can also pass only the value for the initial field or fields of the index, so the following fields will not be considered.

**NOTE**

I won't discuss these features in details, showing complete examples, because some of them are limited to the BDE and makes sense only for local tables, not for SQL server-based tables. Actually if you set a filter or a range over a Table connected with a SQL server, the BDE will try to generate a proper select statement, avoiding fetching all the data and filtering it locally. The problem is that this isn't always possible and you lose most of your control, two good reasons to use the Query component when working with SQL servers.

## A Query with Parameters

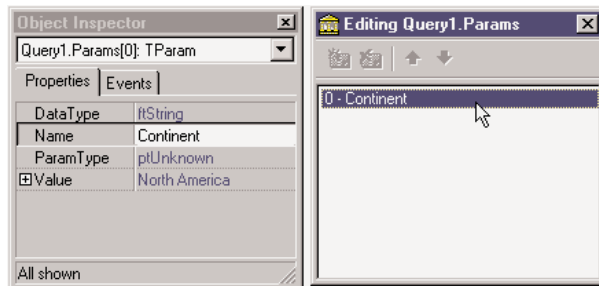
When you need slightly different versions of the same SQL query, instead of modifying the text of the Query (stored in the SQL property) each time, you can write a query with a parameter and simply change the value of the parameter. For example, if you decide to have a user choose the countries of a continent (using the Country table of the DBDEMOS database), you can write the following parametric query:

```
select *
from Country
where Continent = :Continent
```

In this SQL clause, `:Continent` is a parameter. We can set its data type and startup value, using the editor of the Params property collection of the Query component. When the Parameters collection editor is open, as shown in Figure 13.1, you see a list of the parameters defined in the SQL statement and set the data type and the initial value of these parameters.

**FIGURE 13.1:**

Editing the collection of parameters of a Query component



The form displayed by this program, called ParQuery and available on the companion CD, uses a list box to provide all the available values for the parameters. Instead of preparing the items of the list box at design time, we can extract the available continents from the same

database table as the program starts. This is accomplished using a second query component, with this SQL statement:

```
select distinct Continent  
from Country
```

After activating this query, the program scans its result set, extracting all the values and adding them to the list box:

```
procedure TQueryForm.FormCreate(Sender: TObject);  
begin  
  // get the list of continents  
  Query2.Open;  
  while not Query2.EOF do  
  begin  
    ListBox1.Items.Add (Query2.Fields [0].AsString);  
    Query2.Next;  
  end;  
  ListBox1.ItemIndex := 0;  
  
  // open the first query  
  Query1.Params[0].Value := ListBox1.Items [0];  
  Query1.Open;  
end;
```

Before opening the query, the program selects as its parameter the first item of the list box, which is also activated by setting the `ItemIndex` property to 0. When the list box is selected, the program closes the query and changes the parameter:

```
procedure TQueryForm.ListBox1Click(Sender: TObject);  
begin  
  Query1.Close;  
  Query1.Params[0].Value := ListBox1.Items [ListBox1.ItemIndex];  
  Query1.Open;  
end;
```

This displays the countries of the selected continent in the list box, as you can see in Figure 13.2. The final refinement is that when the user enters a record with a new continent, it is added automatically to the list box. Instead of refreshing the entire list, with the same code executed in the `FormCreate` method, we can do this by handling the `BeforePost` event and adding the continent to the list if it is not already there:

```
procedure TQueryForm.Query1BeforePost(DataSet: TDataSet);  
var  
  StrNewCont: string;  
begin  
  // add the continent, if not already in the list  
  StrNewCont := Query1.FieldName ('Continent').AsString;  
  if ListBox1.Items.IndexOf (StrNewCont) < 0 then  
    ListBox1.Items.Add (StrNewCont);  
end;
```

**FIGURE 13.2:**

The ParQuery example at run time



We can add a little extra code to this program to take advantage of a specific feature of parameterized queries. To react faster to a change in the parameters, these queries can be optimized, or *prepared*. Simply call the Prepare method before the program first opens the query (after setting the Active property of the Query component to False at design time) and call Unprepare once the query won't be used anymore:

```

procedure TQueryForm.FormCreate(Sender: TObject);
begin
    ...
    // prepare and open the first query
    Query1.Prepare;
    Query1.Params[0].Value := ListBox1.Items [0];
    Query1.Open;
end;

procedure TQueryForm.FormDestroy(Sender: TObject);
begin
    Query1.Close;
    Query1.Unprepare;
end;

```

Prepared parameterized queries are very important when you work on a complex query. In fact, the BDE or the SQL server must read the text of the query and determine how to process it. If you use the same query (even if a parametric one) over and over, the engine doesn't need to reprocess the query but already knows how to handle it.

## Master/Detail Structures

Often you need to relate tables, which have a one-to-many relationship. This means that for a single record of the master table, there are many detailed records in a secondary table. A classic example is that of an invoice and the items of the invoice; another is a list of customers

and the orders each customer has made. This is very common situation in database programming, and Delphi provides explicit support for it with the master/detail structure. We'll see this structure for BDE Table and Query components, but the same technique applies to almost all of the datasets available in Delphi.

**NOTE**

The `TDataSet` class has a generic `DataSource` property for setting up a master data source, but the Table component, for example, uses a different property (`MasterSource`) to express the same concept.

## Master/Detail with Tables

The simplest way to create a master/detail structure in Delphi is to use the Database Form Wizard, selecting a master/detail form in the first page. To accomplish the same effect manually, place two table components in a form or data module, connect them with the same database, and connect each with a table. In the `MastDet` example, I've used the customer and orders tables of the `DBDEMOS` database, and I've used a data module. Now add a `DataSource` component for each table, and for the secondary table set a master source to the data source connected to the first table. Finally relate the secondary table to a field (called `MasterField`) of the main table, using the special property editor provided.

## A Data Module for Data-Access Components

To build a Delphi database application, you can place data-access components and the data-aware controls in a form. This is handy for a simple program, but having the user interface and the data access and data model in a single, often large, unit is far from a good idea. For this reason, Delphi implements the idea of *data module*, a container of nonvisual components I already introduced in Chapter 1, "The Delphi 6 IDE."

At design time, a data module is similar to a form, but at run time it exists only in memory. The `TDataModule` class derives directly from `TComponent`, so it is completely unrelated to the Windows concept of a window (and is fully portable among different operating systems). Unlike a form, a data module has just a few properties and events. For this reason, it's useful to think of data modules as components and method containers.

Like a form or a frame, a data module has a designer. This means Delphi creates for a data module a specific Object Pascal unit for the definition of its class and a form definition file that lists its components and their properties.

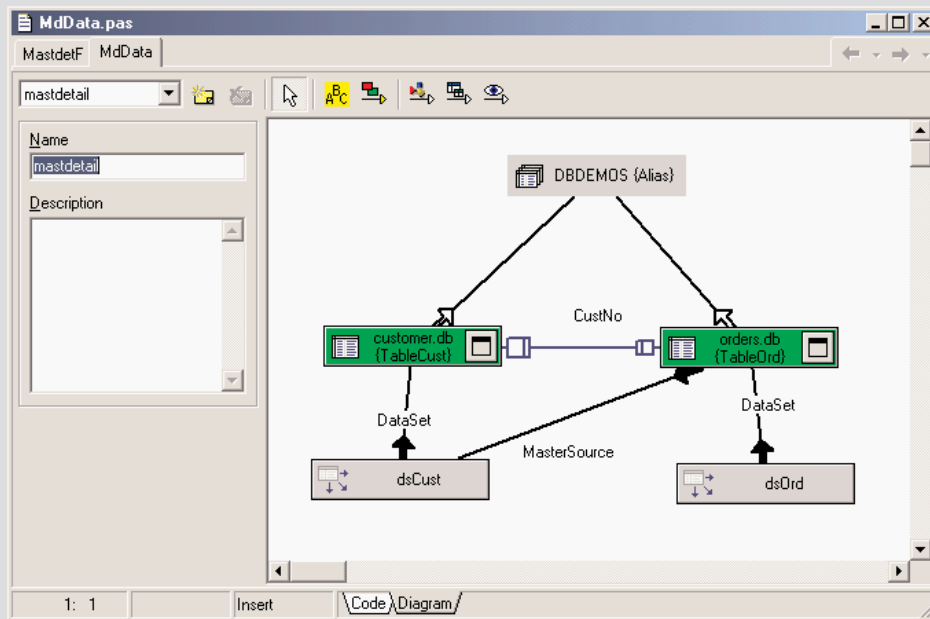
There are several reasons to use data modules. The simplest one is to share data-access components among multiple forms, as I'll demonstrate at the beginning of the next chapter. This technique works in conjunction with visual form linking, the ability to access components of another form or data module at design time (with the `File > Use Unit` command). The second

*Continued on next page*



reason is to separate the data from the user interface, improving the structure of an application. Data modules in Delphi even exist in versions specific for multitier applications (remote data modules) and server-side HTTP applications (Web data modules).

Finally, remember that you can use the Diagram page of the editor, introduced in Chapter 1, to see a graphical representation of the connections among the components of a data module, as you can see in this example for the MastDet application:



The following is the complete listing (only without the irrelevant positional properties) of the Data Module used by the MastDet program on the CD:

```

object DataModule1: TDataModule1
  OnCreate = DataModule1Create
object TableCust: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object TableOrd: TTable
  DatabaseName = 'DBDEMOS'
  IndexName = 'CustNo'
  MasterFields = 'CustNo'

```

```

    MasterSource = dsCust
    TableName = 'orders.db'
end
object dsCust: TDataSource
    DataSet = TableCust
end
object dsOrd: TDataSource
    DataSet = TableOrd
end
end
end

```

**TIP**

Starting with Delphi 5, you can also create a master/detail structure using the Data Diagram view of a data module.

In Figure 13.3 you can see an example of the main form of the MastDet program at run time. I've placed data-aware controls related to the master table in the upper portion, and I've placed a grid connected with the detail table in the lower portion of the form. This way, for every master record, you immediately see the list of the connected detail record, in this case all the orders by the current client. Each time you select a new customer, the grid below displays only the orders pertaining to that customer.

**FIGURE 13.3:**

The MastDet example at run time

The screenshot shows a window titled "Master Detail" with a standard Windows-style title bar. Below the title bar is a toolbar with navigation icons (back, forward, home, end, search, refresh). The main area is divided into two sections. The upper section contains a form with several text boxes for customer data: Company (Tom Sawyer Diving Centre), Addr1 (632-1 Third Frydenhoj), Addr2 (empty), City (Christiansted), State (St. Croix), Zip (00820), Country (US Virgin Islands), Phone (504-798-3022), and FAX (504-798-7772). The lower section is a data grid with the following columns: OrderNo, CustNo, SaleDate, ShipDate, EmpNo, ShipToContact, and ShipToAddr1. The grid contains six rows of data, with the third row (OrderNo 1072) selected.

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact	ShipToAddr1
1005	1356	20/04/88	21/01/88 12	110		
1059	1356	24/02/89	25/02/89	109		4-976 Sugarloaf H
1072	1356	11/04/89	12/04/89	29		
1080	1356	05/05/89	06/05/89	45		
1105	1356	21/07/92	21/07/92	28		
1180	1356	06/08/94	06/08/94	144		

## A Master/Detail Structure with Queries

The previous example used two tables to build a master/detail form. As an alternative, you can define this type of join using a SQL statement. After setting the master DataSource for

the detailed query, you simply set up its SQL statement with a parameter having the same name of the field of the master dataset this data source refers to.

For this example (called Orders), I've joined the ORDERS.DB table with ITEMS.DB, which describes the items of each order. The two tables can be joined using the OrderNo field. When you generate the code, the program behaves exactly like the previous one, Mast-Det. This time, however, the trick is in the SQL statements of the second query object:

```
select OrderNo, ItemNo, PartNo, Qty
from items
where OrderNo = :OrderNo
```

As you can see, this SQL statement uses a parameter, OrderNo. This parameter is connected directly to the first query, because the DataSource property of QueryItems is set to dsOrders, which is connected to QueryOrders. In other words, the second query is considered to be a data control connected to the first data source. Each time the current record in the first data source changes, the QueryItems component is updated, just like any other component connected to dsOrders. The field used for the connection, in this case, is the field having the same name as the query parameter.

## Other BDE Related Components

Along with Table, Query, StoredProc, and DataSource, other components are on the Data Access page of the Component palette, the BDE page. I'll cover these components in the next chapter, but here is a short summary:

- The Database component is used for transaction control, security, and connection control. It is generally used only to connect to remote databases in client/server applications or to avoid the overhead of connecting to the same database in several forms. The Database component is also used to set a local alias, one used only inside a program. Once this local alias is set to a given path, the Table and Query components of the application can refer to the local database alias. This is much better than replicating the hard-coded path in each DataSet component of the program.

### TIP

The Borland Database Engine (BDE) uses an alias to refer to a database file or directory. You can define new aliases for databases by using the Database Explorer or the Database Engine Configuration utility. It is also possible to define them by writing code in Delphi that calls the `AddStandardAlias` and `AddAlias` methods of the `Session` global object, followed by a call to `SaveConfigFile` to make the alias persistent. The alternative is the low-level `DbiAddAlias` BDE function. In some of the program of this chapter I'll use the `DBDEMOS` database alias, which refers to Delphi's demo database, installed by default in the `C:\Program Files\Common Files\Borland Shared\Data` directory.

- The Session component provides global control over database connections for an application, including a list of existing databases and aliases and an event to customize database login.
- The BatchMove component is used to perform batch operations, such as copying, appending, updating, or deleting values, on one or more databases.
- The UpdateSQL component allows you to write SQL statements to perform various update operations on the dataset, when using a read-only query (that is, when working with a complex query). This component is used as the value of the UpdateObject property of tables or queries.

## Using Data-Aware Controls

Once you've set up the proper data-access components, you can build a user interface to let a user view the data and eventually edit it. Delphi provides many components that resemble the usual Windows controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all of these components in the Data Controls page of the Delphi Component palette.

All of these components are connected to a data source using the corresponding property, `DataSource`. Some of them relate to the entire dataset, such as the DBGrid and DBNavigator components, while the others refer to a specific field of the data source, as indicated by the `DataField` property. Once you select the `DataSource` property, the `DataField` property will have a list of values available in the drop-down combo box of the Object Inspector.

### NOTE

In Chapter 18, "Writing Database Components," we'll discuss the technical details of these controls, as we'll see how to write custom data-aware components.

Notice that all the data-aware components are totally unrelated to the data-access technology, provided the data-access component inherits from `TDataSet`. This means that your investment on the user interface is totally preserved when you change the data-access technology. What is true, however, is that some of the lookup components and an extended use of the DBGrid, displaying a lot of data, only make more sense when working with local data, and should generally be avoided in a client/server situation, as we'll see in the next chapter.

## Data in a Grid

The DBGrid is a grid capable of displaying a whole table at once. It allows scrolling and navigation, and you can edit the grid's contents. It is an extension of the other Delphi grid controls.

You can customize the DBGrid by setting the various flags of its `Options` property and modifying its `Columns` collection. The grid allows a user to navigate the data, using the scrollbars, and perform all the major actions. A user can edit the data directly, insert a new record in a given position by pressing the `Insert` key, append a new record at the end by going to the last record and pressing the `Down` arrow key, and delete the current record by pressing `Ctrl+Del`.

The `Columns` property is a collection where you can choose the fields of the table you want to see in the grid and set column and title properties (color, font, width, alignment, caption, and so on) for each field. Some of the more advanced properties, such as `ButtonStyle` and `DropDownRows`, can be used to provide custom editors for the cells of a grid or a drop-down list of values (indicated in the `PickList` property of the column).

An alternative to the DBGrid is the DBCtrlGrid component, a multirecord grid that can host panels with other data-aware controls. These controls are duplicated in each panel for each record of the dataset. I'll discuss the DBCtrlGrid control at the end of this chapter.

## DBNavigator and Dataset Actions

DBNavigator is a collection of buttons used to navigate and perform actions on the database. You can disable some of the buttons of the DBNavigator control, by removing some of the elements of the `VisibleButtons` set.

The buttons perform basic actions on the connected dataset, so you can easily replace them with your own toolbar, particularly if you use an `ActionList` component with the predefined database actions provided by Delphi. In this case, in fact, you get all the standard behaviors, but you'll also see the various buttons enabled only when their action is legitimate.

### TIP

If you use the standard actions, you can avoid connecting them to a specific `DataSource` component, and the actions will be applied to the dataset connected to the visual control that currently has the input focus. This way a single toolbar can be used for multiple datasets displayed by a form.

## Text-Based Data-Aware Controls

There are multiple text-oriented components:

- `DBText` displays the contents of a field that cannot be modified by the user. It is a data-aware `Label` graphical control. It can be very useful, but users might confuse this control with the plain labels that indicate the content of each field-based control.

- DBEdit lets the user edit a field (change the current value) using an Edit control. At times, you might want to disable editing and use a DBEdit as if it were a DBText, but highlighting the fact that this is data coming from the database.
- DBMemo lets the user see and modify a large text field, eventually stored in a memo or BLOB (binary large object) field. It resembles the Memo component and has full editing capabilities, but all the text is rendered in a single font.
- DBRichEdit is a component that lets the user edit a formatted text file; it is based on a RichEdit Windows common control and, in contrast to DBMemo, it allows text with multiple fonts and paragraph styles.

## List-Based Data-Aware Controls

For letting a user choose a value in a predefined list (which reduces input errors), you can use many different components. DBListBox, DBComboBox, and DBRadioGroup are similar, providing a list of strings in the `Items` property, but they do have some differences:

- The DBListBox component allows selection of predefined items (“closed selection”), but not text input, and can be used to list many elements. Generally it’s best to show only about six or seven items, to avoid using up too much space on the screen.
- The DBComboBox component can be used both for closed selection and for user input. The `csDropDown` style of the DBComboBox, in fact, allows a user to enter a new value, besides selecting one of the available ones. The component also uses a smaller area of the form because the drop-down list is usually displayed only on request.
- The DBRadioGroup component presents radio buttons (which permit only one selection), allows only closed selection, and should be used only for a limited number of alternatives. A nice features of this component is that the values displayed can be exactly those you want to insert in the database, but you can also choose to provide some sort of mapping. The values of the user interface (some descriptive strings stored in the `Items` property) will map to corresponding values stored in the database (some numeric or character-based codes listed in the `Values` property). For example, you can map some numeric codes indicating departments to a few descriptive strings:

```
object DBRadioGroup1: TDBRadioGroup
    Caption = 'Department'
    DataField = 'Department'
    DataSource = DataSource1
    Items.Strings = (
        'Sales'
        'Accounting'
        'Production'
        'Management')
```

```
Values.Strings = (  
    '1'  
    '2'  
    '3'  
    '4')  
end
```

A slightly different component is the `DBCkCbOx`, used to show and toggle an option, corresponding to a Boolean field. It is a limited list, because it has only two possible values, plus the undetermined state for fields with null values. You can determine which are the values to send back to the database by setting the `ValueChecked` and `ValueUnchecked` properties of this component.

The usage of a `DBRadioGroup` control, with the settings discussed above, and a `DBCkCbOx` control is highlighted by the `DbAware` example.

## Creating Local Tables with `FieldDefs`

The `DbAware` example would be a rather simple program if it didn't have an extra feature: It can create a new table for the `DBDEMOS` database. Delphi allows you to set the definition of the fields of a table—its internal structure—at design time, using the collection editor of the `FieldDefs` property. Once you've defined the fields, you can then right-click the table component at design time and select the `Create Table` command.

This list of field definitions is generally extracted from the database, but if you set the `StoreDefs` property of the table to `True`, it will be saved in the DFM file along with the other table properties. The effect of the `StoreDefs` property is more complex than it seems at first. If you right-click the form, you'll notice that its local menu offers an `Update Table Definition` option, along with the expected `Delete Table` and `Rename Table`. That is, you can store the field definitions locally, but if the structure of the physical table changes, you should update this definition as well. Until Delphi 4, the field definitions were invariably loaded from the database table at run time; now you can preload them, speeding up the table opening. However, if the local and the actual table definitions do not match, you can get in trouble.

In the `DbAware` example, I've used this technique to create a new database table, called `Workers`, which stores data about the employees of a company. This is the definition of the fields of the table, along with the other key properties:

```
object Table1: TTable  
    DatabaseName = 'DBDEMOS'  
    FieldDefs = <  
        item  
            Name = 'LastName'  
            DataType = ftString
```

*Continued on next page*

```
        Size = 20
    end
    item
        Name = 'FirstName'
        DataType = ftString
        Size = 20
    end
    item
        Name = 'Department'
        DataType = ftSmallint
    end
    item
        Name = 'Branch'
        DataType = ftString
        Size = 20
    end
    item
        Name = 'Senior'
        DataType = ftBoolean
    end
    item
        Name = 'HireDate'
        DataType = ftDate
    end>
StoreDefs = True
TableName = 'Workers'
end
```

Regardless of the fact you might have created the table at design time, the program must do so the first time it is executed on a different computer. In practice, when the program starts, it checks whether the table already exists and creates one if it doesn't:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    if not Table1.Exists then
        Table1.CreateTable;
        Table1.Open;
    end;
```

Finally, the program has some code to fill in the table with random values. As this is kind of tedious but not too complex, I won't discuss the details here, but let you look at the source code of the DbAware example if you are interested. I'll use the table produced by this program in other examples in this book, so you might want to run it once and create the table anyway.



## Using Lookup Controls

If the list of values is extracted from another dataset, then instead of the `DBListBox` and `DBComboBox` controls you should use the specific `DBLookupListBox` or `DBLookupComboBox` components. These components are used every time you want to select for a field a record of another dataset.

For example, if you build a standard form for taking orders, the orders dataset will generally have a field hosting a number indicating the customer who made the order. Working directly with the customer number is not the most natural way; most users will prefer to work with customer names. However, in the database, the names of the customers are stored in a different table, to avoid duplicating the customer data for each order by the same customer. To get around such a situation, with local databases or small lookup tables, you can use a `DBLookupComboBox` control. (This technique doesn't port very well to client/server architecture with large lookup tables, as discussed in the next chapter.)

The `DBLookupComboBox` component can be connected to two data sources at the same time, one source containing the actual data and a second containing the display data. Basically, I've built a standard form using the `ORDERS.DB` tables of the `DBDEMOS` database, with several `DBEdit` controls (you can as well use the Database Form Wizard to build this plain form). The example actually uses a Query component selecting most fields of the orders table.

At this point we want to remove the standard `DBEdit` component connected to the customer number and replace it with a `DBLookupComboBox` component (and a `DBText` component for understanding what exactly is going on). The lookup component (and the `DBText`) is connected with the `DataSource` for the order and with the `CustNo` field. To let the lookup component show the information extracted from another table, `CUSTOMER.DB`, we need to add another table component referring to it, and new data source connected to the table.

For the program to work, you need to set several properties of the `DBLookupComboBox1` component. Here is a list of the relevant values:

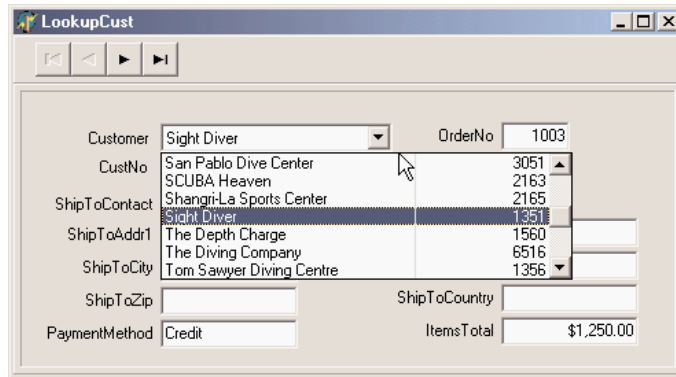
```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSourceOrders
  KeyField = 'CustNo'
  ListField = 'Company;CustNo'
  ListSource = DataSourceCustomer
  DropDownWidth = 300
end
```

The first two properties determine the main connection, as usual. The other three properties determine the secondary source (`ListSource`), the field used for the join (`KeyField`), and the information to display (`ListField`). Besides entering the name of a single field, you can

provide multiple fields, as I've done in the example. Only the first field is displayed as combo box text, but if you set a large value for the `DropDownWidth` property, the pull-down list of the combo box will include multiple columns of data. You can see this output in Figure 13.4.

**FIGURE 13.4:**

The output of the Cust-Lookup example, with the `DBLookupComboBox` showing multiple fields in its drop-down list



**TIP**

If you set the index of the table connected with the `DBLookupComboBox` to the `Company` field, the drop-down list will show the companies in alphabetical order instead of customer-number order. This is what I've done in the example.

What about the code of this program? Well, there is none. Everything works just by setting the correct properties. The three joined data sources do not need custom code. This demonstrates that using master/detail and lookup connections can be very fast to set up and very efficient. The only real drawback is that these techniques, particularly the lookup, cannot be used when the number of records becomes too large, particularly in a networked or client/server environment. Moving hundreds of thousands of records just to make a nice-looking lookup combo box probably won't be very effective.

**NOTE**

In Delphi 6, both the `TDBLookupComboBox` and `TDBLookupListBox` controls have a `NullValueKey` property, which indicates the shortcut that can be used to set the value to null, by calling the `Clear` method of the corresponding field.

## Graphical Data-Aware Controls

Finally, Delphi includes two graphical data-aware controls:

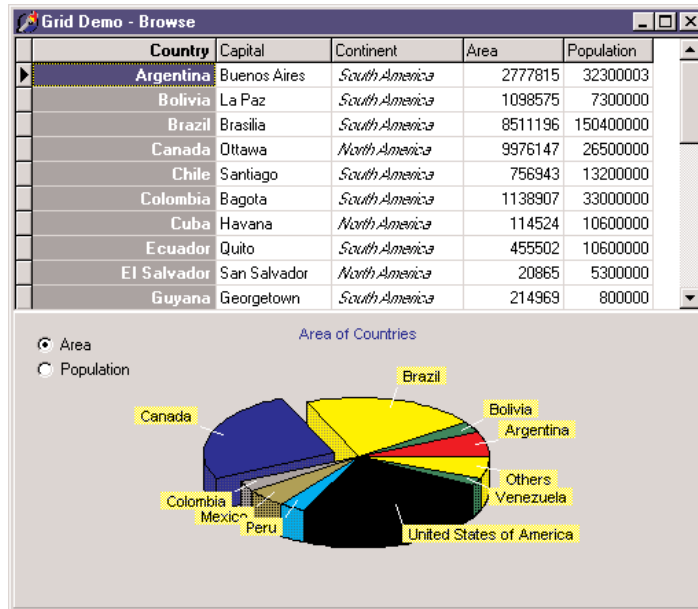
- `DBImage` is an extension of an `Image` component that shows a picture stored in a `BLOB` field (provided the database uses a graphic format that the `Image` component supports, such as `BMP` and `JPEG`).

- DBChart is a data-aware business graphic component or the data-aware version of the TeeChart control built by David Berneda.

To demonstrate the use of the DBChart control, I've added this component to a simple example showing a data grid. The application, called ChartDB, shows a pie chart with the surface of each country of the COUNTRY.DB table, as you can see in Figure 13.5.

**FIGURE 13.5:**

The output of the ChartDB example, which is based on the TDbChart control



The program has almost no code, as all the settings can be done using the specific component editor, which has several options but is quite easy to use. Here are some of the key properties of the component, taken from the form description:

```

object DBChart1: TDBChart
  Legend.Visible = False
  Align = alClient
object Series1: TPieSeries
  Marks.ArrowLength = 8
  Marks.Visible = True
  DataSource = Table1
  XLabelsSource = 'Name'
  ExplodeBiggest = 3
  OtherSlice.Style = poBelowPercent
  OtherSlice.Text = 'Others'
  OtherSlice.Value = 2
  PieValues.ValueSource = 'Area'
end
end

```

What I've done is show the area field as the data source for the pie chart (the `PieValues.ValueSource` property of the series), use the name field for the labels (the `XLabelsSource` property of the series), and condense all the countries with a value below 2 percent in a single section indicated as Others (the `OtherSlide` subproperties).

As a minor addition to the code, I've added two radio buttons you can use to toggle between the area and the population. The code of the two radio buttons simply sets the source of the series, after casting it to the proper series type, as in:

```
procedure TForm1.RadioPopulationClick(Sender: TObject);  
begin  
    DBChart1.Title.Text [0] := 'Population of Countries';  
    (DBChart1.Series [0] as TPieSeries).PieValues.ValueSource := 'Population';  
end;
```

## The DataSet Component

Instead of focusing right away on the use of a specific dataset, I prefer starting with a generic introduction of the features of the `TDataSet` class, which are shared by all inherited data-access classes. The `DataSet` component is a very complex one, so I won't list all of its capabilities but only discuss its core elements.

The idea behind this component is to provide access to a series of records that are read from some source of data, kept in internal buffers (for performance reasons), and eventually modified by a user, with the possibility of writing back changes to the persistent storage. This approach is generic enough to be applied to different types of data (even non-database data) but has a few rules. First, there can be only one active record at a time, so if you need to access data in multiple records, you must move to each of them, read the data, then move again, and so on. You'll find an example of this and related techniques in the section about navigation.

Second, you can edit only the active record: you cannot modify a set of records at the same time, as you can in a relational database. Moreover, you can modify data in the active buffer only after you explicitly declare you want to do so, by giving the `Edit` command to the dataset. You can also use the `Insert` command to create a new blank record, and close both operations (insert or edit) by giving a `Post` command.

Other interesting elements of a dataset I will explore in the following sections are its status (and the status change events), navigation and record positions, and the role of the field objects. As a summary of the capabilities of the `DataSet` component, I've included the public methods of its class in Listing 13.1 (the code has been edited and commented for clarity). Not all of these methods are directly used everyday, but I decided to keep them all in the listing. In

Chapter 18, I'll also discuss the virtual methods of the protected portion of the class, which we'll need to override to build custom dataset components.



### Listing 13.1: The public interface of the *TDataSet* class (excerpted)

```
TDataSet = class(TComponent, IProviderSupport)
...
public
  // create and destroy, open and close
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure Open;
  procedure Close;
  property BeforeOpen: TDataSetNotifyEvent read FBeforeOpen write FBeforeOpen;
  property AfterOpen: TDataSetNotifyEvent read FAfterOpen write FAfterOpen;
  property BeforeClose: TDataSetNotifyEvent
    read FBeforeClose write FBeforeClose;
  property AfterClose: TDataSetNotifyEvent read FAfterClose write FAfterClose;

  // status information
  function IsEmpty: Boolean;
  property Active: Boolean read GetActive write SetActive default False;
  property State: TDataSetState read FState;
  function ActiveBuffer: PChar;
  property IsUniDirectional: Boolean
    read FIsUniDirectional write FIsUniDirectional default False;
  function UpdateStatus: TUpdateStatus; virtual;
  property RecordSize: Word read GetRecordSize;
  property ObjectView: Boolean read FObjectView write SetObjectView;
  property RecordCount: Integer read GetRecordCount;
  function IsSequenced: Boolean; virtual;
  function IsLinkedTo(DataSource: TDataSource): Boolean;

  // datasource
  property DataSource: TDataSource read GetDataSource;
  procedure DisableControls;
  procedure EnableControls;
  function ControlsDisabled: Boolean;

  // fields, including blobs, details, calculated, and more
  function FieldByName(const FieldName: string): TField;
  function FindField(const FieldName: string): TField;
  procedure GetFieldList(List: TList; const FieldNames: string);
  procedure GetFieldNames(List: TStrings);
  property FieldCount: Integer read GetFieldCount;
  property FieldDefs: TFieldDefs read FFieldDefs write SetFieldDefs;
  property FieldDefList: TFieldDefList read FFieldDefList;
  property Fields: TFields read FFields;
  property FieldList: TFieldList read FFieldList;
  property FieldValues[const FieldName: string]: Variant
```

```
    read GetFieldValue write SetFieldValue; default;
property AggFields: TFields read FAggFields;
property DataSetField: TDataSetField
    read FDataSetField write SetDataSetField;
property DefaultFields: Boolean read FDefaultFields;
procedure ClearFields;
function GetBlobFieldData(FieldNo: Integer;
    var Buffer: TBlobByteData): Integer; virtual;
function CreateBlobStream(Field: TField;
    Mode: TBlobStreamMode): TStream; virtual;
function GetFieldData(Field: TField;
    Buffer: Pointer): Boolean; overload; virtual;
procedure GetDetailDataSets(List: TList); virtual;
procedure GetDetailLinkFields(MasterFields, DetailFields: TList); virtual;
function GetFieldData(FieldNo: Integer;
    Buffer: Pointer): Boolean; overload; virtual;
function GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean):
    Boolean; overload; virtual;
property AutoCalcFields: Boolean
    read FAutoCalcFields write FAutoCalcFields default True;
property OnCalcFields: TDataSetNotifyEvent
    read FOnCalcFields write FOnCalcFields;

// position, movement
procedure CheckBrowseMode;
procedure First;
procedure Last;
procedure Next;
procedure Prior;
function MoveBy(Distance: Integer): Integer;
property RecNo: Integer read GetRecNo write SetRecNo;
property Bof: Boolean read FBOF;
property Eof: Boolean read FEOF;
procedure CursorPosChanged;
property BeforeScroll: TDataSetNotifyEvent
    read FBeforeScroll write FBeforeScroll;
property AfterScroll: TDataSetNotifyEvent
    read FAfterScroll write FAfterScroll;

// bookmarks
procedure FreeBookmark(Bookmark: TBookmark); virtual;
function GetBookmark: TBookmark; virtual;
function BookmarkValid(Bookmark: TBookmark): Boolean; virtual;
procedure GotoBookmark(Bookmark: TBookmark);
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; virtual;
property Bookmark: TBookmarkStr read GetBookmarkStr write SetBookmarkStr;

// find, locate
function FindFirst: Boolean;
function FindLast: Boolean;
function FindNext: Boolean;
```

```

function FindPrior: Boolean;
property Found: Boolean read GetFound;
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean; virtual;
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant; virtual;

// filtering
property Filter: string read FFilterText write SetFilterText;
property Filtered: Boolean read FFiltered write SetFiltered default False;
property FilterOptions: TFilterOptions
  read FFilterOptions write SetFilterOptions default [];
property OnFilterRecord: TFilterRecordEvent
  read FOnFilterRecord write SetOnFilterRecord;

// refreshing, updating
procedure Refresh;
property BeforeRefresh: TDataSetNotifyEvent
  read FBeforeRefresh write FBeforeRefresh;
property AfterRefresh: TDataSetNotifyEvent
  read FAfterRefresh write FAfterRefresh;
procedure UpdateCursorPos;
procedure UpdateRecord;
function GetCurrentRecord(Buffer: PChar): Boolean; virtual;
procedure Resync(Mode: TResyncMode); virtual;

// editing, inserting, posting, and deleting
property CanModify: Boolean read GetCanModify;
property Modified: Boolean read FModified;
procedure Append;
procedure Edit;
procedure Insert;
procedure Cancel; virtual;
procedure Delete;
procedure Post; virtual;
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
procedure SetFields(const Values: array of const);

// events related to editing, inserting, posting, and deleting
property BeforeInsert: TDataSetNotifyEvent
  read FBeforeInsert write FBeforeInsert;
property AfterInsert: TDataSetNotifyEvent
  read FAfterInsert write FAfterInsert;
property BeforeEdit: TDataSetNotifyEvent read FBeforeEdit write FBeforeEdit;
property AfterEdit: TDataSetNotifyEvent read FAfterEdit write FAfterEdit;
property BeforePost: TDataSetNotifyEvent read FBeforePost write FBeforePost;
property AfterPost: TDataSetNotifyEvent read FAfterPost write FAfterPost;
property BeforeCancel: TDataSetNotifyEvent
  read FBeforeCancel write FBeforeCancel;
property AfterCancel: TDataSetNotifyEvent

```

```
    read FAfterCancel write FAfterCancel;
property BeforeDelete: TDataSetNotifyEvent
    read FBeforeDelete write FBeforeDelete;
property AfterDelete: TDataSetNotifyEvent
    read FAfterDelete write FAfterDelete;
property OnDeleteError: TDataSetErrorEvent
    read FOnDeleteError write FOnDeleteError;
property OnEditError: TDataSetErrorEvent
    read FOnEditError write FOnEditError;
property OnNewRecord: TDataSetNotifyEvent
    read FOnNewRecord write FOnNewRecord;
property OnPostError: TDataSetErrorEvent
    read FOnPostError write FOnPostError;

// support, utilities
function Translate(Src, Dest: PChar;
    ToOem: Boolean): Integer; virtual;
property Designer: TDataSetDesigner read FDesigner;
property BlockReadSize: Integer read FBlockReadSize write SetBlockReadSize;
property SparseArrays: Boolean read FSparseArrays write SetSparseArrays;
end;
```

---

## The Status of a Dataset

When you operate on a dataset in Delphi, you can work in different states, indicated by a specific `State` property, which can assume several different values:

**dsBrowse** indicates that the dataset is in normal browse mode, used to look at the data and scan the records.

**dsEdit** indicates that the dataset is in edit mode. A dataset enters this state when the program calls the `Edit` method or the `DataSource` has the `AutoEdit` property set to `True`, and the user starts editing a data-aware control, such as a `DBGrid` or `DBEdit`. When the changed record is posted, the dataset exits the `dsEdit` state.

**dsInsert** indicates that a new record is being added to the dataset. Again, this might happen when calling the `Insert` method, moving to the last line of a `DBGrid`, or using the corresponding command of the `DBNavigator` component.

**dsInactive** is the state of a closed dataset.

**dsSetKey** indicates that we are preparing a search on the dataset. This is the state between a call to the `SetKey` method and a call to the `GotoKey` or `GotoNearest` methods (see the `Search` example later in this chapter).

**dsCalcFields** is the state of a dataset while a field calculation is taking place; that is, during a call to an `OnCalcFields` event handler. Again, I'll show this in an example.



**dsNewValue**, **dsOldValue**, and **dsCurValue** are the states of a dataset when an update of the cache is in progress.

**dsFilter** is the state of a dataset while setting a filter; that is, during a call to an `OnFilterRecord` event handler.

In simple examples, the transitions between these states are handled automatically, but it is important to understand them because there are many events referring to the state transitions. For example, every dataset fires events before and after any state change. When a program requests an `Edit` operation, the component fires the `BeforeEdit` event just before entering in edit mode (an operation you can stop by raising an exception). Immediately after entering edit mode, the dataset receives the `AfterEdit` event. After the user has finished editing and requests to store the data, executing the `Post` command, the dataset fires a `BeforePost` event, which can be used to check the input before sending the data to the database, and an `AfterPost` event after the operation has been successfully completed.

Another more general state-change tracking technique is to handle the `OnStateChange` event of the `DataSource` component. As a very simple example you can show the current status with code like this:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  strStatus: string;
begin
  case Table1.State of
    dsBrowse: strStatus := 'Browse';
    dsEdit: strStatus := 'Edit';
    dsInsert: strStatus := 'Insert';
  else
    strStatus := 'Other state';
  end;
  StatusBar.Panels[0].Text := strStatus;
end;
```

The code considers only the three most common states a dataset component, ignoring the inactive state and other special cases.

## The Fields of a Dataset

I mentioned earlier that a dataset has only one record that is the current, or active, one. The record is stored in a buffer, and you can operate on it with some generic methods, but to access the data of the record you need to use the field objects of the dataset. This explains why field components (technically instances of class derived from the `TField` class) play a

fundamental role in every Delphi database application. Data-aware controls are directly connected to these field objects, which correspond to database fields.

By default, Delphi automatically creates the `TField` components at run time, each time the program opens a dataset component. This is done after reading the metadata associated with the table or the query the dataset refers to. These field components are stored in the `Fields` array property of a dataset. You can access these values by number (accessing the array directly) or by name (using the `FieldByName` method). Each field can be used for reading or modifying the data of the current record, using its `Value` property or type-specific properties, like `AsDate`, `AsString`, `AsInteger`, and so on:

```
var
  strName: string;
begin
  strName := Table1.Fields[0].AsString
  strName := Table1.FieldByName('LastName').AsString
```

`Value` is a variant type property, so using the type-specific access properties is a little more efficient. The dataset component has also a shortcut property for accessing the variant-type value of a field, the default `FieldValues` property. Being a default property means you can omit it from the code by applying the square brackets directly to the dataset:

```
strName := Table1.FieldValues ['LastName'];
strName := Table1 ['LastName'];
```

Creating the field components each time a dataset is opened is only a default behavior. As an alternative, you can create the field components at design time, using the Fields editor (double-click a dataset to see the Fields editor in action, or activate its local menu and choose the Fields Editor command). After creating a field for the `LastName` column of a table, for example, you can refer to its value by applying one of the `AsXxx` methods to the proper field object:

```
strName := Table1.LastName.AsString;
```

Besides being used to access the value of a field, each field object also has properties for controlling visualization and editing of its value, including range of values, edit masks, display format, constraints, and many others. These properties, of course, depend on the type of the field—that it is, on the specific class of the field object. If you create persistent fields you can set some properties at design time, instead of writing code at run time, maybe in the `AfterOpen` event of the dataset.

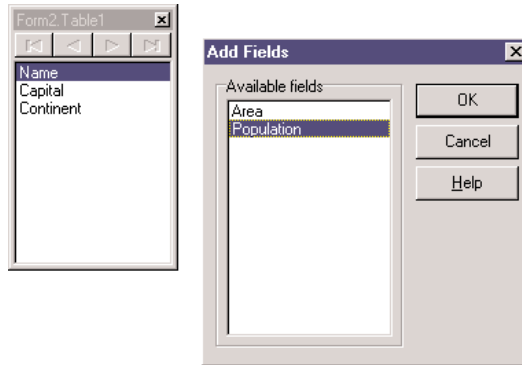
## NOTE

Although the Fields editor is similar to the editors of the collections used by Delphi, fields are not part of a collection. They are components created at design time, listed in its published section of the form class, and available in the drop-down combo box at the top of the Object Inspector.

As you open the Fields editor for a dataset, it appears empty. You have to activate the local menu of this editor to access its capabilities. The simplest operation you can do is to select the Add command, which allows you to add any other fields in the dataset to the list of fields. Figure 13.6 shows the Add Fields dialog box, which lists all the fields that are available in a table. These are the database table fields that are not already present in the list of fields in the editor.

**FIGURE 13.6:**

The Fields editor with the Add Fields dialog box



The Define command of the Fields editor, instead, lets you define a new calculated field, lookup field, or field with a modified type. In this dialog box, you can enter a descriptive field name, which might include blank spaces. Delphi generates an internal name—the name of the field component—which you can further customize. Next, select a data type for the field. If this is a calculated field or a lookup field, and not just a copy of a field redefined to use a new data type, simply check the proper radio button. We'll see how to define a calculated field in the section “Adding a Calculated Field” and a lookup field in two following sections.

**NOTE**

A `TField` component has both a `Name` property and a `FieldName` property. The `Name` property is the usual component name. The `FieldName` property is either the name of the column in the database table or the name you define for the calculated field. It can be more descriptive than the `Name`, and it allows blank spaces. The `FieldName` property of the `TField` component is copied to the `DisplayLabel` property by default, but this field name can be changed to any suitable text. It is used, among other things, to search a field in the `FieldByName` method of the `TDataSet` class and when using the array notation.

All of the fields that you add or define are included in the Fields editor and can be used by data-aware controls or displayed in a database grid. If a field of the original database table is not in this list, it won't be accessible. When you use the Fields editor, Delphi adds the declaration of the available fields to the class of the form, as new components (much as the Menu Designer adds `TMenuItem` components to the form). The components of the `TField` class, or more specifically its subclasses, are fields of the form, and you can refer to these components

directly in the code of your program to change their properties at run time or to get or set their value.

In the Fields editor, you can also drag the fields to change their order. Proper field ordering is particularly important when you define a grid, which arranges its columns using this order.

**TIP**

An even better feature of the Fields editor is that you can drag fields from this editor to the surface of a form and have Delphi automatically create a corresponding data-aware control (such as a DBEdit, a DBMemo, or a DBImage). The type of control created depends on the data type of the field. This is a very fast way to generate custom forms, and I suggest you try it out if you've never used it before. This is my preferred way to build database-related forms, much better than using the Database Form Wizard.

## Using Field Objects

Before we look at an example, let's go over the use of the `TField` class. The importance of this component should not be underestimated. Although it is often used behind the scenes, its role in database applications is fundamental. As I already mentioned, even if you do not define specific objects of this kind, you can always access the fields of a table or a query using their `Fields` array property, the `FieldValues` indexed property, or the `FieldByName` method. Both the `Fields` property and the `FieldByName` function return an object of type `TField`, so you sometimes have to use the `as` operator to downcast their result to its actual type (like `TFloatField` or `TDateField`) before accessing specific properties of these subclasses.

The `FieldAcc` example is a simple extension of a form generated by the Database Form Wizard. I've added to it three speed buttons in the toolbar panel, accessing various field properties at run time. The first button changes the formatting of the population column of the grid. To do this, we have to access the `DisplayFormat` property, a specific property of the `TFloatField` class. For this reason we have to write:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);  
begin  
    (Table1.FieldByName ('Population') as  
    TFloatField).DisplayFormat := '###,###,###';  
end;
```

When you set field properties related to data input or output, the change applies to every record in the table. When you set properties related to the value of the field, instead, you always refer to the current record only. For example, we can output the population of the current country in a message box by writing:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);  
begin  
    ShowMessage (string (Table1 ['Name']) + ': ' + string (Table1 ['Population']));  
end;
```

When you access the value of a field, you can use a series of *As* properties to handle the current field value using a specific data type (if this is available; otherwise, an exception is raised):

```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt;
AsString: string;
AsVariant: Variant;
```

These properties can be used to read or change the value of the field. Changing the value of a field is possible only if the dataset is in edit mode. As an alternative to the *As* properties indicated above, you can access the value of a field by using its *Value* property, which is defined as a variant.

Most of the other properties of the *TField* component, such as *Alignment*, *DisplayLabel*, *DisplayWidth*, and *Visible*, reflect elements of the field's user interface and are used by the various data-aware controls, particularly *DBGrid*. In the *FieldAcc* example, clicking the third speed button changes the *Alignment* of every field:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields[I].Alignment := taCenter;
end;
```

This affects the output of the *DBGrid*, and of the *DBEdit* control I've added to the toolbar, which shows the name of the country. You can see this effect, along with the new display format, in Figure 13.7.

**FIGURE 13.7:**

The output of the *FieldAcc* example after the *Center* and *Format* buttons have been pressed

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32,300,003
Bolivia	La Paz	South America	1098575	7,300,000
Brazil	Brasilia	South America	8511196	150,400,000
Canada	Ottawa	North America	9976147	26,500,000
Chile	Santiago	South America	756943	13,200,000
Colombia	Bagota	South America	1136907	33,000,000
Cuba	Havana	North America	114524	10,600,000
Ecuador	Quito	South America	455502	10,600,000
El Salvador	San Salvador	North America	20865	5,300,000
Guyana	Georgetown	South America	214969	800,000

## A Hierarchy of Field Classes

There are several field class types in VCL. Delphi automatically uses one of them depending on the data definition in the database, when you open a table at run time or when you use the Fields editor at design time. Table 13.1 shows the complete list of subclasses of the `TField` class.

**TABLE 13.1:** The Subclasses of `TField`

Subclass	Base Class	Definition
<code>TADTField</code>	<code>TObjectField</code>	An ADT (Abstract Data Type) field, corresponding to an object field in an object relational database.
<code>TAggregateField</code>	<code>TField</code>	An aggregate field represents a maintained aggregate. It is used in the <code>ClientDataSet</code> component and discussed in Chapter 14, "Client/Server Programming."
<code>TArrayField</code>	<code>TObjectField</code>	An array of objects in an object relational database.
<code>TAutoIncField</code>	<code>TIntegerField</code>	Whole positive number connected with a Paradox auto-increment field of a table, a special field automatically assigned a different value for each record. Note that Paradox AutoInc fields do not always work perfectly, as discussed in the next chapter.
<code>TBCDField</code>	<code>TNumericField</code>	Real numbers, with a fixed number of digits after the decimal point.
<code>TBinaryField</code>	<code>TField</code>	Generally not used directly. This is the base class of the next two classes.
<code>TBlobField</code>	<code>TField</code>	Binary data and no size limit (BLOB stands for binary large object). The theoretical maximum limit is 2 GB.
<code>TBooleanField</code>	<code>TField</code>	Boolean value.
<code>TBytesField</code>	<code>TBinaryField</code>	Arbitrary data with a large (up to 64 KB characters) but fixed size.
<code>TCurrencyField</code>	<code>TFloatField</code>	Currency values, with the same range as the new <code>Real</code> data type.
<code>TDataSetField</code>	<code>TObjectField</code>	An object corresponding to a separate table in an object relational database.
<code>TDateField</code>	<code>TDateTimeField</code>	Date value.
<code>TDateTimeField</code>	<code>TField</code>	Date and time value.
<code>TFloatField</code>	<code>TNumericField</code>	Floating-point numbers (8 byte).
<code>TFMTBCDField</code>	<code>TNumericField</code>	(New field type in Delphi 6) A true binary-coded decimal (BCD), as opposed to the existing <code>TBCDField</code> type, which converted BCD values to the Currency type. This field type is used automatically only by dbExpress datasets.
<code>TGraphicField</code>	<code>TBlobField</code>	Graphic of arbitrary length.
<code>TGuidField</code>	<code>TStringField</code>	A field representing a COM Globally Unique Identifier, part of the ADO support.

*Continued on next page*

**TABLE 13.1 continued:** The Subclasses of TField

Subclass	Base Class	Definition
TIDispatchField	TInterfaceField	A field representing pointers to IDispatch COM interfaces, part of the ADO support.
TIntegerField	TNumericField	Whole numbers in the range of long integers (32 bits).
TInterfacedField	TField	Generally not used directly. This is the base class of fields that contain pointers to interfaces (TUnknown) as data.
TLargeIntField	TIntegerField	Very large integers (64 bit).
TMemoField	TBlobField	Text of arbitrary length.
TNumericField	TField	Generally not used directly. This is the base class of all the numeric field classes.
TObjectField	TField	Generally not used directly. The base class for the fields providing support for object relational databases.
TReferenceField	TObjectField	A pointer to an object in an object relational database.
TSmallIntField	TIntegerField	Whole numbers in the range of integers (16 bits).
TSQLErrorField	TField	(New field type in Delphi 6) Supports the date/time representation used in dbExpress drivers
TStringField	TField	Text data of a fixed length (up to 8192 bytes).
TTimeField	TDateTimeField	Time value.
TVarBytesField	TBytesField	Arbitrary data, up to 64 KB characters. Very similar to the TBytesField base class.
TVariantField	TField	A field representing a variant data type, part of the ADO support.
TWideStringField	TStringField	A field representing a Unicode (16 bits per character) string.
TWordField	TIntegerField	Whole positive numbers in the range of words or unsigned integers (16 bits).

The availability of any particular field type, and the correspondence with the data definition, depends on the database in use. This is particularly true for the new field types that provide support for object relational databases.

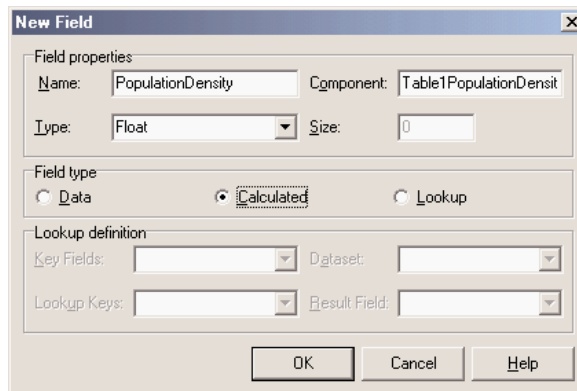
## Adding a Calculated Field

Now that you've been introduced to TField objects and seen an example of their run-time use, it is time to build a simple example based on the declaration of field objects at design time using the Fields editor. We can start again from the first example we built, GridDemo, and add a calculated field. The COUNTRY.DB database table we are accessing has both the population and the area of each country, so we can use this data to compute the population density.

To build the new example, named Calc, select the Table component in the form, and open the Fields editor. In this editor, choose the **Add Fields** command, and select some of the fields. (I've decided to include them all.) Now select the **New Field** command, and enter a proper name and data type (TFloatField) for the new calculated field, as you can see in Figure 13.8.

**FIGURE 13.8:**

The definition of a calculated field in the Calc example



**WARNING**

It is obvious that as you create some field components at design time using the Fields editor, the fields you skip won't get a corresponding object. What might not be obvious is that the fields you skip will not be available even at run time, with Fields or `FieldByName`. When a program opens a table at run time, if there are no design-time field components, Delphi creates field objects corresponding to the table definition. If there are some design-time fields, however, Delphi uses those fields without adding any extra ones.

Of course, we also need to provide a way to calculate the new field. This is accomplished in the `OnCalcFields` event of the Table component, which has the following code (at least in a first version):

```
procedure TForm2.Table1CalcFields(DataSet: TDataSet);  
begin  
    Table1PopulationDensity.Value := Table1Population.Value / Table1Area.Value;  
end;
```

**NOTE**

Calculated fields are computed for each record and recalculated each time the record is loaded in an internal buffer, invoking the `OnCalcFields` event over and over again. For this reason, a handler of this event should be extremely fast to execute, and cannot alter the status of the dataset, by accessing different records. A more time-efficient (but less memory-efficient) version of a calculated field is provided by the ClientDataSet component with "internally calculated" fields, which are evaluated only once, when they are loaded, with the result stored in memory for future requests.



Everything fine? Not at all! If you enter a new record and do not set the value of the population and area, or if you accidentally set the area to zero, the division will raise an exception, making it quite problematic to continue using the program. As an alternative, we could have handled every exception of the division expression and simply set the resulting value to zero:

```
try
  Table1PopulationDensity.Value := Table1Population.Value / Table1Area.Value;
except
  on Exception do
    Table1PopulationDensity.Value := 0;
end;
```

However, we can do even better. We can check if the value of the area is defined—if it is not null—and if it is not zero. It is better to avoid using exceptions when you can anticipate the possible error conditions:

```
if not Table1Area.IsNull and
  (Table1Area.Value <> 0) then
  Table1PopulationDensity.Value := Table1Population.Value / Table1Area.Value
else
  Table1PopulationDensity.Value := 0;
```

The code of the `Table1CalcFields` method above (in each of the three versions) accesses some fields directly. This is possible because I used the Fields editor, and it automatically created the corresponding field declarations, as you can see in this excerpt of the interface declaration of the form:

```
type
  TCalcForm = class(TForm)
    Table1: TTable;
    Table1PopulationDensity: TFloatField;
    Table1Area: TFloatField;
    Table1Population: TFloatField;
    Table1Name: TStringField;
    Table1Capital: TStringField;
    Table1Continent: TStringField;
    procedure Table1CalcFields(DataSet: TDataSet);
    ...
```

Each time you add or remove fields in the Fields editor, you can see the effect of your action immediately in the grid present in the form. Of course, you won't see the values of a calculated field at design time; they are available only at run time, because they result from the execution of compiled Pascal code.

Since we have defined some components for the fields, we can use them to customize some of the visual elements of the grid. For example, to set a display format that adds a comma to separate thousands, we can use the Object Inspector to change the `DisplayFormat` property of some field components to “###,###,###”. This change has an immediate effect on the grid at design time.

**NOTE**

The display format I've just mentioned (and used in the previous example) uses the Windows International Settings to format the output. When Delphi translates the numeric value of this field to text, the comma in the format string is replaced by the proper `ThousandSeparator` character. For this reason, the output of the program will automatically adapt itself to different International Settings. On computers that have the Italian configuration, for example, the comma is replaced by a period.

After working on the table components and the fields, I've customized the `DBGrid` using its `Columns` property editor. I've set the `Population Density` column to read-only and set its `ButtonStyle` property to `cbsEllipsis`, to provide a custom editor. When you set this value, a small button with an ellipsis is displayed when the user tries to edit the grid cell. Pressing the button invokes the `OnEditButtonClick` event of the `DBGrid`:

```

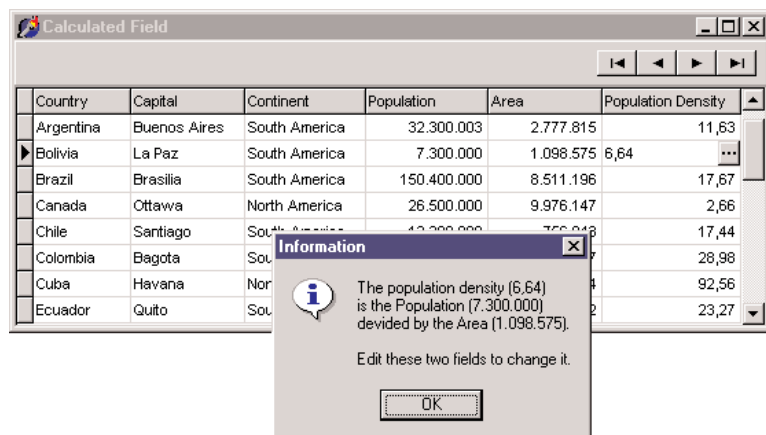
procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
    MessageDlg (Format (
        'The population density (%.2n)'#13 +
        'is the Population (%.0n)'#13 +
        'divided by the Area (%.0n).'#13#13 +
        'Edit these two fields to change it.',
        [Table1PopulationDensity.AsFloat,
        Table1Population.AsFloat,
        Table1Area.AsFloat]),
        mtInformation, [mbOK], 0);
end;

```

Actually, I haven't provided a real editor, but rather a message describing the situation, as you can see in Figure 13.9, which shows the values of the calculated fields. To create an editor, you might build a secondary form to handle special data entries.

**FIGURE 13.9:**

The output of the Calc example. Notice the `Population Density` calculated column, the ellipsis button, and the message displayed when you select it.



## Lookup Fields

As an alternative to placing a `DBLookupComboBox` component in a form (discussed earlier in this chapter under “Using Lookup Controls”), we can also define a lookup field, which can be displayed with a drop-down lookup list inside a `DBGrid` component. We’ve seen that to add a fixed selection to a `DBGrid`, we can simply edit the `PickList` subproperty of the `Columns` property. To customize the grid with a live lookup, instead, we have to define a lookup field using the Fields editor.

As an example, I’ve built the `FieldLookup` program, which has a grid displaying orders with a lookup field to display the name of the employee who took the order, instead of the code number of this employee. To accomplish this, I added to the data module a `Table` component referring to the `EMPLOYEE.DB` table. Then I opened the Fields editor for the `ORDERS` table and added all the fields. I selected the `EmpNo` field and set its `Visible` property to `False`, to remove it from the grid (we cannot remove it altogether, because it is used to build the cross-reference with the corresponding field of the `EMPLOYEE` table).

Now it is time to define the lookup field. If you’ve followed the preceding steps, you can use the Fields editor of the `ORDERS` table and select the `New Field` command, obtaining the `New Field` dialog box. (As an alternative in Delphi 5, it’s possible to use the `Diagram` page of the editor, drop the two tables there, and drag a lookup relation from the `ORDERS` table to the `EMPLOYEE` table, connecting the two in the resulting `New Field` dialog box. In Delphi 6, though, the lookup relation button is still part of the `Diagram` page but doesn’t seem to be working at all.)

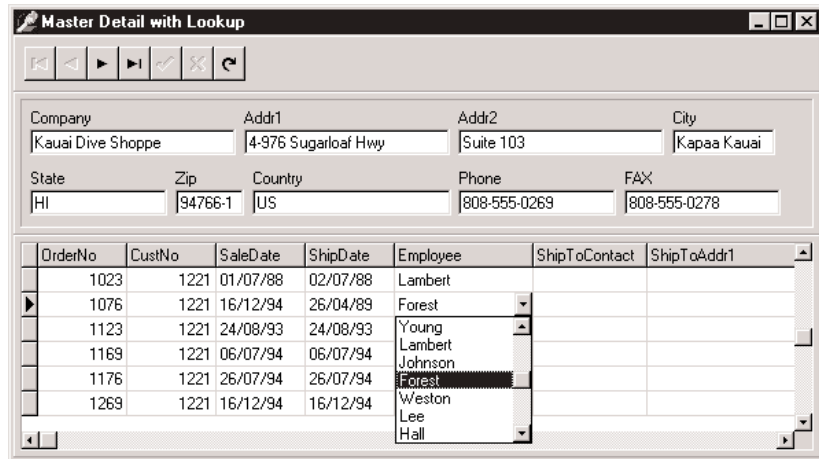
The values you specify in the `New Lookup Field` dialog box will affect the properties of a new `TField` added to the table, as demonstrated by the DFM description of the field:

```
object Table2Employee: TStringField
  FieldKind = fkLookup
  FieldName = 'Employee'
  LookupDataSet = Table2
  LookupKeyFields = 'EmpNo'
  LookupResultField = 'LastName'
  KeyFields = 'EmpNo'
  Size = 30
  Lookup = True
end
```

This is all that is needed to make the drop-down list work (see Figure 13.10) and to view the value of the cross-references field at design time, too. Notice that there is no need to customize the `Columns` property of the grid, because the drop-down button and the value of seven rows are taken by default. This doesn’t mean you cannot use this property to further customize these and other visual elements of the grid.

**FIGURE 13.10:**

The output of the Field-Lookup example, with the drop-down list inside the grid displaying values taken from another database table



## Handling Null Values with Field Events

Beyond a few interesting properties, the field objects have a few key events. The `OnValidate` event can be used to provide extended validation of the value of a field, and should be used whenever you need a complex rule that the ranges and constraints provided by the field cannot express. This event is triggered before the data is written to the record buffer, whereas the `OnChange` event is fired soon after the data has been written.

Two more events, `OnGetText` and `OnSetText`, can be used to customize the output of a field. These two events are extremely powerful: they allow you to use data-aware controls even when the representation of a field you want to display is different from the one Delphi will provide by default.

An example of the use of these events is the handling of null value. On SQL servers, storing an empty value or a null value for a field are two separate operations. The latter tends to be more correct, but Delphi by default uses empty values and displays the same output for an empty or a null field. Although this can be useful in general for strings and numbers, it becomes extremely important for dates, where it is hard to set a reasonable default value and where if the user blanks out the field you might have an invalid input.

The `NullDates` program displays a specific text for dates that have a null value and clears the field (setting it to the null value) when the user uses an empty string in input. Here is the relevant code of the two event handlers for the field:

```
procedure TForm1.Table1ShipDateGetText(Sender: TField;
var Text: String; DisplayText: Boolean);
begin
```

```

if Sender.IsNull then
  Text := '<undefined>'
else
  Text := Sender.AsString;
end;

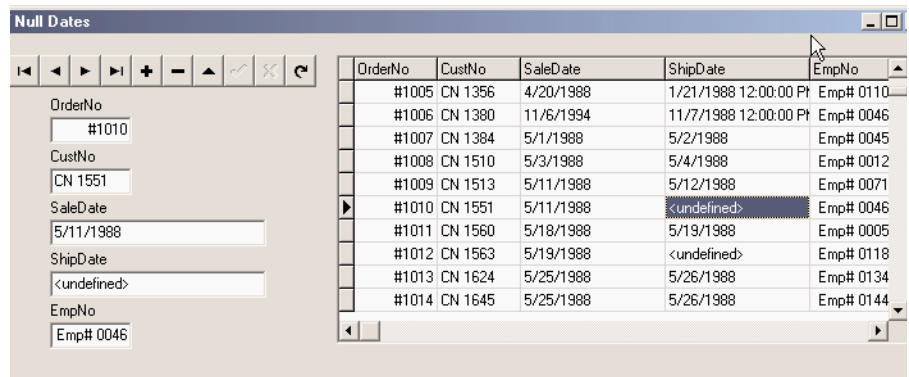
procedure TForm1.Table1ShipDateSetText(Sender: TField; const Text: String);
begin
  if Text = '' then
    Sender.Clear
  else
    Sender.AsString := Text;
end;

```

In Figure 13.11 you can see an example of the output of this program, with undefined (or null) values for some shipping dates.

**FIGURE 13.11:**

By handling the `OnGetText` and `OnSetText` events of a date field, the `NullDates` example displays a specific output for null values



## Navigating a Dataset

We've seen that a dataset has only one active record, and you can imagine that the active record changes often, in response of user actions or because of internal commands given to the dataset. To move around the dataset and change the active record, there are methods of the `TDataSet` class, as you can see in Listing 13.1, particularly in the section commented as "position, movement." You can move to the next or previous record, jump back and forth by a given number of records (with `MoveBy`), or go directly to the first or last record of the dataset. These operations of the dataset are generally available in the `DBNavigator` component or in the standard dataset actions, and they are not particularly complex to understand.

What is not obvious, though, is how a dataset handles the extreme positions. If you open any dataset with a navigator attached, you can see that as you move on record by record, the Next button remains enabled even when you've reached the last record. It's only when you try to move forward after the last record that the current record apparently doesn't change and the button is disabled. This is because the Eof test (end of file) succeeds only when the cursor has been moved to a special position after the last record. If you jump to the end with the Last button, instead, you'll immediately be at the very end. You'll see exactly the same behavior for the first record (and the Bof test). As we'll see in a while, this approach is very handy, as we can scan a dataset testing for Eof to be True and, at this point, we know we've also already processed the last record of the dataset.

**NOTE**

Handling this special record positions before the beginning and after the end of the dataset, which are called *cracks*, is very important (and quite confusing) when you write a custom dataset, as we'll see in Chapter 18.

Besides moving around record by record or by a given number of records, programs might need to jump to specific records or positions. Some datasets support the RecordCount property and allow movement to a record at a given position in the dataset using the RecNo property. These properties can be used only for datasets that support positions natively, which basically excludes all client/server architectures, unless you grab all of the records in a local cache (something you'll generally want to avoid) and then navigate on the cache. As we'll see in the next chapter, when you open a query on a SQL server you fetch only the records you are using, so Delphi doesn't know the record count, at least not in advance.

There are two alternatives you can use to refer to a record in a dataset, regardless of its type:

- You can save a reference to the current record and then jump back to it after moving around. This is accomplished by using bookmarks, either in the TBookmark or the more modern TBookmarkStr form. This approach is discussed in the upcoming section "Using Bookmarks."
- You can locate a record of the dataset matching given criteria, using the Locate method. This even works after you close and reopen the dataset, because you're working at a logical (and not physical) level. This approach is presented in the next section.

## Locating Records in a Table

To show you an example of the use of the Locate method, I've built the Search example, which has a table connected to EMPLOYEE.DB. The form I've prepared has the data-aware edit boxes inside a scroll box aligned to the client area, so that a user can freely resize the form without any problems. When the form becomes too small, scroll bars will appear automatically in the area holding the edit boxes. Another feature is a toolbar with buttons connected to

some of the predefined dataset actions available in the `ActionList` component plus two custom actions to host the search code.

The searching capabilities are activated by the two buttons connected to custom actions. The first button is connected to `ActionGoto`, used for an exact match, and the second to `ActionGoNear`, for a partial match. In both cases, we want to compare the text in the edit box with the `LastName` fields of the `EMPLOYEE` table. If the local table has an index on the field (as in the specific case) `Locate` will use it, but the method will work with or without indexes (only at a different speed).

If you've never used `Locate`, at first sight the help file won't be terribly clear. The idea is that you must provide a list of fields you want to search, and a list of values, one for each field. If you pass only one field, the value is passed directly, as in the case of the example:

```
procedure TSearchForm.ActionGotoExecute(Sender: TObject);
begin
  if not Table1.Locate ('LastName', EditName.Text, []) then
    MessageDlg ('"' + EditName.Text + '"' not found', mtError, [mbOk], 0);
end;
```

If you search for multiple fields, you have to pass a variant array with the list of the values you want to match. The variant array can be created from a constant array with the `VarArrayOf` function or from scratch using the `VarArrayCreate` call. This is a code snippet from the example:

```
Table1.Locate ('LastName;FirstName', VarArrayOf (['Cook', 'Kevin']), [])
```

Finally, we can use the same method to look for a record even if we know only the initial portion of the field we are looking for. Simply add the `loPartialKey` flag to the `Options` parameter (the third) of the `Locate` call.

## NOTE

Using `Locate` makes sense specifically for local tables, but doesn't port very well to client/server applications. In fact, on local tables this technique is rather optimized by letting the dataset read in only the records it is looking for, using local indexes. On a SQL server, instead, similar client-side techniques imply moving all the data to the client application first, which is generally a bad idea. Locating the data should be performed with restricted SQL statements. You can still call `Locate` after you've retrieved a limited dataset. For example, you can search a customer by name after you've selected all the customer of a given town or area, obtaining a result set of a limited size. There's more on this topic in the next chapter, which is devoted to client/server development.

## The Total of a Table Column

So far in our examples, the user can view the current contents of a database table and manually edit the data or insert new records. Now we will see how we can change some data in the table through the program code. The idea behind this example is quite simple. The EMPLOYEE table we have been using has a Salary field. A manager of the company could indeed browse through the table and change the salary of a single employee. But what will be the total salary expense for the company? And what if the manager wants to give a 10 percent salary increase (or decrease) to everyone?

These are the two aims of the Total example, which is an extension of the previous program. The toolbar of this new example has some more buttons and actions. There are a few other minor changes from the previous example. I opened the Fields editor of the table and removed the Table1Salary field, which was defined as a TFloatField. Then I selected the New Field command and added the same field, with the same name, but using the TCurrencyField data type. This is not a calculated field; it's simply a field converted into a new (but equivalent) data type. Using this new field type the program will default to a new output format, suitable for currency values.

Now we can turn our attention to the code of this new program. First, let's look at the code of the total action. This action lets you calculate the sum of the salaries of all the employees, then edit some of the values, and compute a new total. Basically, we need to scan the table, reading the value of the Table1Salary field for each record:

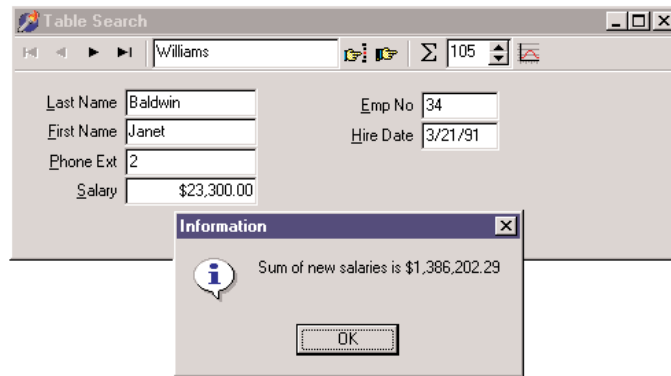
```
var
    Total: Real;
begin
    Total := 0;
    Table1.First;
    while not Table1.EOF do
        begin
            Total := Total + Table1Salary.Value;
            Table1.Next;
        end;
    MessageDlg ('Sum of new salaries is ' +
        Format ('%m', [Total]), mtInformation, [mbOk], 0);
end
```

This code works, as you can see from the output in Figure 13.12, but it has some problems. One problem is that the record pointer is moved to the last record, so the previous position in the table is lost. Another is that the user interface is refreshed many times during the operation.



**FIGURE 13.12:**

The output of the Total program, showing the total salaries of the employees



## Using Bookmarks

To avoid these two problems, we need to disable updates and to store the current position of the record pointer in the table and restore it at the end. This can be accomplished using a *table bookmark*, a special variable storing the position of a record in a database table. Delphi's traditional approach is to declare a variable of the TBookmark data type, and initialize it while getting the current position from the table:

```
var
    Bookmark: TBookmark;
begin
    Bookmark := Table1.GetBookmark;
```

At the end of the ActionTotalExecute method, we can restore the position and delete the bookmark with the following two statements:

```
Table1.GotoBookmark (Bookmark);
Table1.FreeBookmark (Bookmark);
```

As a better (and more up-to-date) alternative, we can use the Bookmark property of the TDataset class, which refers to a bookmark that is disposed of automatically. (This is technically implemented as an *opaque string*, a structure subject to string lifetime management, but it is not a string, so you're not supposed to look at what's inside it.) This is how you can modify the code above:

```
var
    Bookmark: TBookmarkStr;
begin
    Bookmark := Table1.Bookmark;
    ...
    Table1.Bookmark := Bookmark;
```

To avoid the other side effect of the program (we see the records scrolling while the routine browses through the data), we can temporarily disable the visual controls connected with the table. The table has a `DisableControls` method we can call before the `while` loop starts and an `EnableControls` method we can call at the end, after the record pointer is restored.

**Tip**

Disabling the data-aware controls connected with a table during long operations not only improves the user interface (since the output is not changing constantly), it also speeds up the program considerably. In fact, the time spent to update the user interface is much greater than the time spent performing the calculations. To test this, try commenting out the `DisableControls` and `EnableControls` methods of the `Total` example, and see the speed difference.

Finally, we face some dangers from errors in reading the table data, particularly if the program is reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. So we should use a `try/finally` block. Actually, if you want to make the program 100 percent error-proof, you should use two nested `try/finally` blocks. Including this change and the two discussed above, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);  
var  
    Bookmark: TBookmarkStr;  
    Total: Real;  
begin  
    Bookmark := Table1.Bookmark;  
    try  
        Table1.DisableControls;  
        Total := 0;  
        try  
            Table1.First;  
            while not Table1.EOF do  
                begin  
                    Total := Total + Table1Salary.Value;  
                    Table1.Next;  
                end;  
            finally  
                Table1.EnableControls;  
            end  
        finally  
            Table1.Bookmark := Bookmark;  
        end;  
    MessageDlg ('Sum of new salaries is ' +  
        Format ('%m', [Total]), mtInformation, [mbOK], 0);  
end;
```

**NOTE**

I've written this code to show you an example of a loop to browse the contents of a table, but keep in mind that there is an alternative approach based on the use of a SQL query returning the sum of the values of a field. When you use a SQL server, the speed advantage of a SQL call to compute the total can be very large, since you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result.

## Editing a Table Column

The code of the increase action is similar to the one we have just seen. The `ActionIncrease-Execute` method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference. When you increase the salary, you actually change the data in the table. The two key statements are within the `while` loop:

```
while not Table1.EOF do
begin
    Table1.Edit;
    Table1Salary.Value := Round (Table1Salary.Value * SpinEdit1.Value) / 100;
    Total := Total + Table1Salary.Value;
    Table1.Next;
end;
```

The first statement brings the table into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary, multiplying the old one by the value of the `SpinEdit` component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount—even double the salary of each employee—with the click of a button.

**WARNING**

Notice that the table enters the edit mode every time the `while` loop is executed. This is because in a dataset, edit operations can take place only one record at a time. You must finish the edit operation, calling `Post` or moving to a different record as in the code above. At that time, if you want to change another record, you have to enter edit mode once more.

## Customizing a Database Grid

Unlike most other data-aware controls, which are quite simple to use, the `DBGrid` control has many options and is more powerful than you might think. The following sections explore some of the advanced operations you can do using a `DBGrid` control. A first example shows how to draw in a grid, a second one shows how to clone the behavior of a check box for a

Boolean selection inside a grid, and the final example shows how to use the multiple-selection feature of the grid.

## Painting a DBGrid

There are many reasons you might want to customize the output of a grid. A good example is to highlight specific fields or records. Another is to provide some form of output for fields that usually don't show up in the grid, such as BLOB, graphic, and memo fields.

To thoroughly customize the drawing of a DBGrid control, you have to set its `DefaultDrawing` property to `False` and handle its `OnDrawColumnCell` event. In fact, if you leave the value of `DefaultDrawing` set to `True`, the grid will display the default output before the method is called. This way, all you can do is add something to the default output of the grid, unless you decide to draw over it, which will take extra time and cause flickering.

The alternative approach is to call the `DefaultDrawColumnCell` method of the grid, perhaps after changing the current font or restricting the output rectangle. In this last case you can provide an extra drawing in a cell and let the grid fill the remaining area with the standard output. This is what I've done in the `DrawData` program.

The DBGrid control in this example, which is connected to the commonly used `BIOLIFE` table of the `DBDEMOS` database, has the following properties:

```
object DBGrid1: TDBGrid
  Align = alClient
  DataSource = DataSource1
  DefaultDrawing = False
  Font.Height = -16
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  TitleFont.Height = -11
  TitleFont.Name = 'MS Sans Serif'
  TitleFont.Style = []
  OnDrawColumnCell = DBGrid1DrawColumnCell
end
```

The `OnDrawColumnCell` event handler is called once for every cell of the grid and has several parameters, including the rectangle corresponding to the cell, the index of the column we have to draw, the column itself (with the field, its alignment, and other subproperties), and the status of the cell. How can we set the color of specific cells to red? We can simply change it in the special cases:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
```

```

// red font color if length > 100
if (Column.Field = Table1Lengthcm) and (Table1Lengthcm.AsInteger > 100) then
    DBGrid1.Canvas.Font.Color := clRed;

// default drawing
DBGrid1.DefaultDrawDataCell (Rect, Column.Field, State);
end;

```

The next step is to draw the memo and the graphic fields. For the memo we can simply implement the memo field's `OnGetText` and `OnSetText` events. In fact, the grid will even allow editing on a memo field if its `OnSetText` event is not `nil`. Here is the code of the two event handlers. I've used `Trim` to remove trailing nonprinting characters, which make the text appear to be empty when editing:

```

procedure TForm1.Table1NotesGetText(Sender: TField;
    var Text: String; DisplayText: Boolean);
begin
    Text := Trim (Sender.AsString);
end;

procedure TForm1.Table1NotesSetText(Sender: TField; const Text: String);
begin
    Sender.AsString := Text;
end;

```

For the image, the simplest approach is to create a temporary `TBitmap` object, assign the graphics field to it, and paint the bitmap to the `Canvas` of the grid. As an alternative, I've removed the graphic field from the grid, by setting its `Visible` property to `False`, and added the image to the fish name, with the following extra code in the `OnDrawColumnCell` event handler:

```

var
    Bmp: TBitmap;
    OutRect: TRect;
    BmpWidth: Integer;
begin
    // default output rectangle
    OutRect := Rect;

    if Column.Field = Table1Common_Name then
    begin
        // draw the image
        Bmp := TBitmap.Create;
        try
            Bmp.Assign (Table1Graphic);
            BmpWidth := (Rect.Bottom - Rect.Top) * 2;
            OutRect.Right := Rect.Left + BmpWidth;
        finally
            Bmp.Free;
        end;
    end;

```

```

        DBGrid1.Canvas.StretchDraw (OutRect, Bmp);
    finally
        Bmp.Free;
    end;
    // reset output rectangle, leaving space for the graphic
    OutRect := Rect;
    OutRect.Left := OutRect.Left + BmpWidth;
end;

// red font color if length > 100 (omitted - see above)

// default drawing
DBGrid1.DefaultDrawDataCell (OutRect, Column.Field, State);


```

As you can see in the code above, the program shows the image in a small rectangle on the left of the grid cell and then changes the output rectangle to the remaining area before activating the default drawing. You can see the effect in Figure 13.13.

**FIGURE 13.13:**

The DrawData program displays a grid that includes the text of a memo field and the ubiquitous Borland fishes.



Category	Common_Name	Length (cm)	Notes
Triggerfish	 Clown Triggerfish	50	Also known as the big spotted triggerfish.
Snapper	 Red Emperor	60	Called seaperch in Australia. Inhabits the
Wrasse	 Giant Maori Wrasse	229	This is the largest of all the wrasse. It is f
Angelfish	 Blue Angelfish	30	Habitat is around boulders, caves, coral le
Cod	 Lunartail Rockcod	80	Also known as the coronation trout. It is f
Scorpionfish	 Firefish	38	
Butterflyfish	 Ornate r Butterflyfish	19	Normally seen in pairs around dense coral
Shark	 Swell Shark	102	Inhabits shallow reef caves and crevices e
Ray	 Bat Ray	56	Also know as the grinder ray because of it
Eel	 California Moray	150	This fish hides in a shallow-water lair with
Cod	 Lingcod	150	Widely found from near the shore to very c
Sculpin	 Cabezon	99	Often called the great marbled sculpin. Fi
Spadefish	 Atlantic Spadefish	90	Found in mid-water areas around reefs. wi

## A Check Box Cell

Another common extension of the DBGrid control, found in many third-party components, is the use of check boxes to select the status of Boolean field values. A simple way to do this is to place a DBCheckBox control in front of the grid when the user selects the corresponding item. I've done this in the CheckDbg example, which uses the Workers table created in the DbAware example discussed earlier in this chapter.

The form displayed by the program contains only the grid and the check box. This is a summary of the textual description of the form:

```

object DbForm: TDbForm
    OnCreate = FormCreate
object DBGrid1: TDBGrid

```

```

    Align = alClient
    DataSource = DataSource1
    OnColEnter = DBGrid1ColEnter
    OnDrawColumnCell = DBGrid1DrawColumnCell
    OnKeyPress = DBGrid1KeyPress
end
object DBCheckBox1: TDBCheckBox
    Caption = 'Senior'
    DataField = 'Senior'
    DataSource = DataSource1
    ValueChecked = 'True'
    ValueUnchecked = 'False'
    Visible = False
end
object Table1: TTable
    DatabaseName = 'DBDEMOS'
    TableName = 'Workers'
end
end

```

Notice that the check box is initially hidden and that the program handles several events of the DBGrid control. The first is the `OnDrawColumnCell` event, which is not used to customize the drawing (the `DefaultDrawing` property is set to `True`), but only to compute the position of the check box when a cell of the corresponding field is selected:

```

procedure TDbaForm.DBGrid1DrawColumnCell(Sender: TObject;
    const Rect: TRect; DataCol: Integer; Column: TColumn;
    State: TGridDrawState);
begin
    if (gdFocused in State) and (Column.Field = Table1Senior) then
        begin
            DBCheckBox1.SetBounds (
                Rect.Left + DBGrid1.Left + 1,
                Rect.Top + DBGrid1.Top + 1,
                Rect.Right - Rect.Left + 1,
                Rect.Bottom - Rect.Top + 1);
        end;
    end;
end;

```

The check box itself is displayed or hidden as the user enters or exits the corresponding column, by the handler of the `OnColEnter` event. Note that we cannot refer to the column by position, since a user can move the columns:

```

procedure TDbaForm.DBGrid1ColEnter(Sender: TObject);
begin
    if DBGrid1.Columns [DBGrid1.SelectedIndex].Field = Table1Senior then
        DBCheckBox1.Visible := True
    else
        DBCheckBox1.Visible := False;
    end;
end;

```

Finally, as an extra extension, when the check box is visible (that is, when the user has activated the corresponding field), the program intercepts the keyboard input in the grid, toggling the selection of the check box instead of accepting the input:

```
procedure TDbaForm.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
  if DBCheckBox1.Visible and (Ord (Key) > 31) then
  begin
    Key := #0;
    Table1.Edit;
    DBCheckBox1.Checked := not DBCheckBox1.Checked;
    DBCheckBox1.Field.AsBoolean := DBCheckBox1.Checked;
  end;
end;
```

To make this work we must not only toggle the status of the check box, but also go into edit mode and update the data of the field. You can see an example of the output of this program in Figure 13.14.

**FIGURE 13.14:**

The grid of the CheckDbg example uses a check box for selecting the value of a Boolean field.

LastName	FirstName	Department	Branch	Senior	HireDate
Parker	John	Production	Salt Lake City	True	2/3/97
Parker	werwr	Sales	San Diego	True	1/16/98
Young	Tim	Management	Minneapolis	False	1/19/97
Lee	Tim	Sales	Tokio	True	6/22/96
Reed	Ralph	Sales	New York	True	1/28/96
Young	Gary	Accounting	Las Vegas	True	12/14/96
Lee	Joseph	Accounting	Cape Town	<input checked="" type="checkbox"/> Senior	12/6/96
Young	Ralph	Accounting	Chicago	False	9/18/95
Osborne	Joseph	Production	Salt Lake City	False	8/15/96
Young	Bill	Management	Brasilia	True	12/12/96
MacDonald	Joseph	Management	New York	False	5/28/97
Parker	Bob	Management	San Jose	False	4/1/98
MacDonald	Paul	Sales	Denver	False	2/16/98

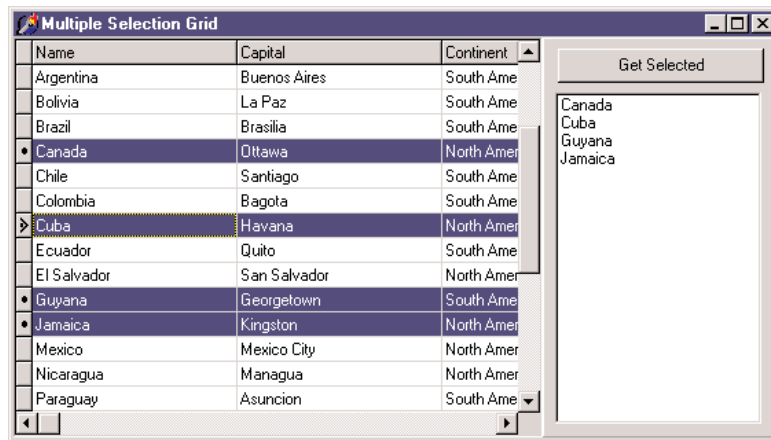
## A Grid Allowing Multiple Selection

The third and last example of customizing the DBGrid control relates to multiple selection. You can set up the DBGrid so that a user can select multiple rows (that is, multiple records). This is very easy, since all you have to do is toggle the `dgMultiSelect` element of the `Options` property of the grid. Once you've selected this option, a user can keep the `Ctrl` key pressed and click with the mouse to select multiple rows of the grid, with the effect you can see in Figure 13.15.



**FIGURE 13.15:**

The MltGrid example has a DBGrid control that allows the selection of multiple rows.



Since the database table can have only one active record, what information is stored in the grid for the selected items? The grid simply keeps a list of bookmarks to the selected records. This list is available in the `SelectedRows` property, which is of type `TBookmarkList`. Besides accessing the number of objects in the list with the `Count` property, you can get to each bookmark with the `Items` property, which is the default array property. Each item of the list is on a `TBookmarkStr` type, which represents a bookmark pointer you can assign to the `Bookmark` property of the table.

**NOTE**

The `TBookmarkStr` is a string type for convenience, but its data should be considered “opaque” and volatile. You shouldn’t rely on any particular structure to the data you may find if you peek at a bookmark’s value, and you shouldn’t hold on to the data too long or store it in a separate file. Bookmark data will vary with database driver and index configuration, and it may be rendered unusable when rows are added to or deleted from the dataset (by you or by other users of the database).

To summarize the steps, here is the code of the MltGrid example, activated by pressing the button to move the Name field of the selected records to the list box:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
    BookmarkList: TBookmarkList;
    Bookmark: TBookmarkStr;
begin
    // store the current position
    Bookmark := Table1.Bookmark;
    try
        // empty the list box
        ListBox1.Items.Clear;

```

```

// get the selected rows of the grid
BookmarkList := DbGrid1.SelectedRows;
for I := 0 to BookmarkList.Count - 1 do
begin
  // for each, move the table to that record
  Table1.Bookmark := BookmarkList[I];
  // add the name field to the listbox
  ListBox1.Items.Add (Table1.FieldByName ('Name').AsString);
end;
finally
  // go back to the initial record
  Table1.Bookmark := Bookmark;
end;
end;

```

## Dragging to a Grid

Another interesting technique is to use dragging with grids. Dragging *from* a grid is not particularly difficult, as you know which are the current record and the column the user has selected. Dragging *to* a grid, instead, is apparently hard to program. You might remember that in Chapter 3, “The Object Pascal Language: Inheritance and Polymorphism,” I mentioned the “protected hack;” this is the technique I’m going to use to implement dragging to a grid.

The example, called DragToGrid, has a grid connected to the COUNTRY.DB demo table, an edit where you can type the new value for a field, and a label you can drag over a cell of the grid to modify the related field. The real problem is how to determine this field. The code is only a few lines, as you can see below, but it is certainly cryptic, and requires some explanation:

```

type
  TDBGHack = class (TDbGrid)
  end;

procedure TFormDrag.DBGrid1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  gc: TGridCoord;
begin
  gc := TDBGHack (DbGrid1).MouseCoord (x, y);
  if (gc.y > 0) and (gc.x > 0) then
  begin
    DbGrid1.DataSource.DataSet.MoveBy (gc.y - TDBGHack(DbGrid1).Row);
    DbGrid1.DataSource.DataSet.Edit;
    DBGrid1.Columns.Items [gc.X - 1].Field.AsString := EditDrag.Text;
  end;
  DBGrid1.SetFocus;
end;

```

The first operation is to determine the cell over which the mouse was released. Starting with the *X* and *Y* mouse coordinates, we can call the protected `MouseCoord` method to access the row and column of the cell. Unless the drag target is the very first row (usually hosting the titles) or the first column (usually hosting the indicator), the program moves the current record by the difference between the requested row (`gc.y`) and the current active row (the protected `Row` property of the grid). The next step is to put the dataset into edit mode, grab the field of the target column (`Columns.Items [gc.X - 1].Field`), and change its text.

## Database Applications with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this is certainly not required. When you need to have very precise control over the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. My personal view is that this is necessary only in very specific cases, as you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you understand the default behavior of Delphi, and it will help me introduce some more database-related events (discussed in the sections “Database Events” and “Field Events”).

The development of an application not based on data-aware controls can follow two different approaches. You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a much more customized approach. I'll demonstrate the first technique in the `NonAware` example and the latter in the `SendToDb` example.

### Mimicking Delphi Data-Aware Controls

If you want to build an application that doesn't use data-aware controls but behaves like a standard Delphi application, you can simply write event handlers for the operations that would be performed automatically by data-aware controls. Basically you need to place the dataset in edit mode as the user changes the content of the visual controls, and update the field objects of the dataset as the user exits from the controls, moving the focus to another element.

---

**Tip**

This approach can be handy for integrating a control that's not data-aware into a standard application. A good example is the use of the `DateTimePicker` control for selecting a date, as demonstrated later in the section “Editing Dates with a Calendar.”

The other element of the `NonAware` example is a list of buttons corresponding to some of those in the `DBNavigator` control and connected to five custom actions. I cannot use the standard dataset actions for this example simply because they automatically hook to the data

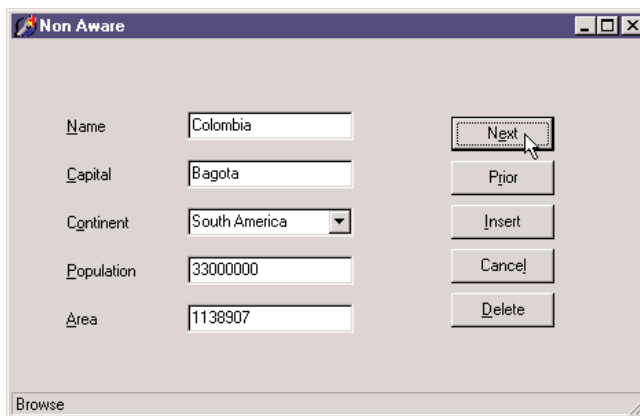
source associated with the control having the focus, a mechanism that fails with the non-data-aware edit boxes of this example.

The program has several event handlers we've not used for past applications using data-aware controls. First of all, we have to show the data of the current record in the visual controls (as in Figure 13.16), by handling the `OnDataChange` event of the `DataSource1` component:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
    EditName.Text := Table1Name.AsString;
    EditCapital.Text := Table1Capital.AsString;
    ComboContinent.Text := Table1Continent.AsString;
    EditArea.Text := Table1Area.AsString;
    EditPopulation.Text := Table1Population.AsString;
end;
```

**FIGURE 13.16:**

The output of the Non-Aware example in Browse mode. The program manually fetches the data every time the current record changes.



The handler of the `OnStateChange` event of the control displays the status of the table in a status bar control. As the user starts typing in one of the edit boxes or drops down the combo box list, the program sets the table in edit mode:

```
procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
    if not (Table1.State in [dsEdit, dsInsert]) then
        Table1.Edit;
end;
```

This method is connected with the `OnKeyPress` event of the five components and is similar to the `OnDropDown` event handler of the combo box. As the user leaves one of the visual

controls, the handler of the `OnExit` event copies the data to the corresponding field, as in this case:

```
procedure TForm1.EditCapitalExit(Sender: TObject);  
begin  
    if (Table1.State in [dsEdit, dsInsert]) then  
        Table1.Capital.AsString := EditCapital.Text;  
end;
```

The operation takes place only if the table is in edit mode; that is, only if the user has typed in this or another control. This is not really ideal, because extra operations are done even if the text of the edit box didn't change, but the extra steps happen fast enough not to be a concern. For the first edit box, we check the text before copying it, raising an exception if the edit box is empty:

```
procedure TForm1.EditNameExit(Sender: TObject);  
begin  
    if (Table1.State in [dsEdit, dsInsert])then  
        if EditName.Text <> '' then  
            Table1Name.AsString := EditName.Text  
        else  
            begin  
                EditName.SetFocus;  
                raise Exception.Create ('Undefined Country');  
            end;  
end;
```

An alternative approach for testing the value of a field is to handle the `BeforePost` event of the dataset. Keep in mind that in this example, the posting operation is not handled by a specific button but takes place as soon as a user moves to a new record or inserts a new one:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);  
begin  
    if Table1Area.Value < 100 then  
        raise Exception.Create ('Area too small');  
end;
```

In each of these cases, an alternative to raising an exception is to set a default value. However, if a field has a default value it is better to set it up front, so that a user can see which value will be sent to the database. To accomplish this, you can handle the `AfterInsert` event of a dataset, which is fired immediately after a new record has been created (we could have used the `OnNewRecord` event, as well):

```
procedure TForm1.Table1AfterInsert(DataSet: TDataSet);  
begin  
    Table1Continent.Value := 'Asia';  
end;
```

## Sending Requests to the Database

You can further customize the user interface of your application if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This allows you complete freedom, although there might be some side effects (such as limited ability to handle concurrency, something I'll discuss in the next chapter).

For this new example, I've replaced the first edit box with another combo box, and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom ones, used to get the data from the database and send an update to it. To underline the difference of this example, I've even removed the DataSource component.

The GetData method, connected with the corresponding button, simply gets the fields corresponding to the record indicated in the first combo box:

```
procedure TForm1.GetData;  
begin  
    Table1.FindNearest ([ComboName.Text]);  
    ComboName.Text := Table1Name.AsString;  
    EditCapital.Text := Table1Capital.AsString;  
    ComboContinent.Text := Table1Continent.AsString;  
    EditArea.Text := Table1Area.AsString;  
    EditPopulation.Text := Table1Population.AsString;  
end;
```

This method is called whenever the user presses the button, selects an item of the combo box, or presses the Enter key while in the combo box:

```
procedure TForm1.ComboNameClick(Sender: TObject);  
begin  
    GetData;  
end;  
  
procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);  
begin  
    if Key = #13 then  
        GetData;  
end;
```

To make this example work smoothly, at start-up the combo box is filled with all the names of the countries of the table:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    // fill the list of names  
    Table1.Open;  
    while not Table1.Eof do  
        begin
```

```

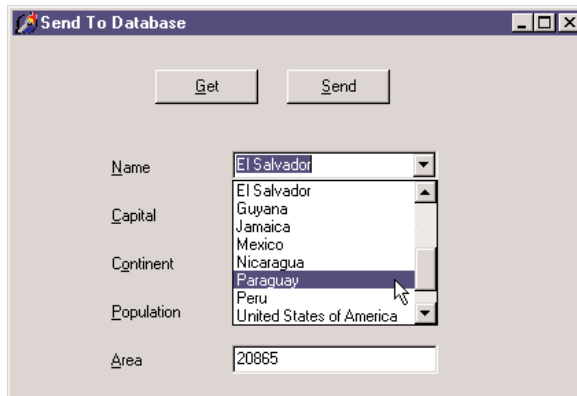
        ComboName.Items.Add (Table1Name.AsString);
        Table1.Next;
    end;
end;

```

With this approach, the combo box becomes a sort of selector of the record, as you can see in Figure 13.17. Notice that thanks to this selection, the program doesn't need navigational buttons.

**FIGURE 13.17:**

In the SendToDb example, you can select the record you want to see in a combo box.



Finally, the user can change the values of the controls and click the Send button. The code to be executed depends on whether the operation is an update or an insert. We can determine this by looking at the name (although with this code, a wrong name cannot be modified any more):

```

procedure TForm1.SendData;
begin
    // raise an exception if there is no name
    if ComboName.Text = '' then
        raise Exception.Create ('Insert the name');

    // check if the record is already in the table
    if Table1.FindKey ([ComboName.Text]) then
        begin
            // modify found record
            Table1.Edit;
            Table1Capital.AsString := EditCapital.Text;
            Table1Continent.AsString := ComboContinent.Text;
            Table1Area.AsString := EditArea.Text;
            Table1Population.AsString := EditPopulation.Text;
            Table1.Post;
        end
    else

```

```

begin
  // insert new record
  Table1.InsertRecord ([ComboName.Text, EditCapital.Text,
    ComboContinent.Text, EditArea.Text, EditPopulation.Text]);
  // add to list
  ComboName.Items.Add (ComboName.Text)
end;

```

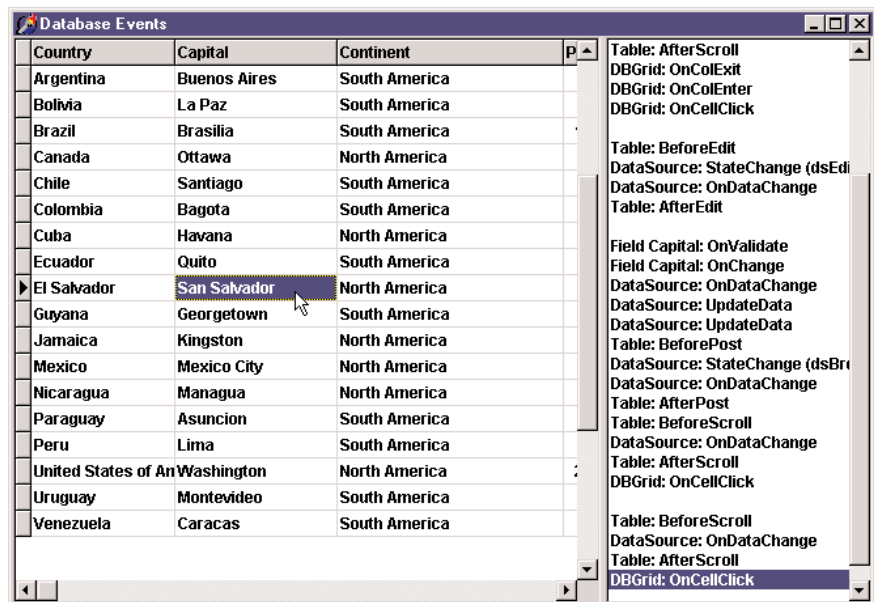
Before sending the data to the table, you can do any sort of validation test on the values. In this case, it doesn't make much sense to handle the events of the database components, because we have full control on when the update or insert operation is done.

## Database Events

To further illustrate how you can use the events of a database application, I've written a simple program that logs all the events being fired. This program handles all of the events of a table and a data source component (although some of these events won't actually be executed, unless you add some extra code, as described later). For each event, I simply send its description to a list box, with the effect you can see in Figure 13.18.

**FIGURE 13.18:**

The output of the DbEvts program, which logs all the events related to database components





Most of the event handlers simply display the name of the component and that of the event, as in

```
procedure TForm1.Table1AfterEdit(DataSet: TDataSet);  
begin  
    AddToList ('Table: AfterEdit');  
end;
```

The field events are slightly more complex, but they use a single handler for the various field components:

```
procedure TForm1.FieldChange(Sender: TField);  
begin  
    AddToList ('Field ' + Sender.FieldName + ': OnChange');  
end;
```

The form's AddToList method adds a new item to the list box and selects it, automatically scrolling the list if required:

```
procedure TForm1.AddToList(Str: string);  
begin  
    // add item and select it  
    Listbox1.ItemIndex := Listbox1.Items.Add (Str);  
end;
```

Finally, the program has a pop-up menu connected to the list box to clear the list or save the items to a file. The menu also has a command you can use to add a blank line, thus separating blocks of events. This operation is also done automatically by a timer, which adds a blank line to the list box unless the last item is already an empty string. This makes the output more readable, as you can see in Figure 13.18.

It is very important to study the output of this program as well as its code. You can try doing all the various operations on the table using the DBGrid, such as inserting, editing, and deleting records, and see the corresponding effect in terms of events fired by the VCL components. To see even more events, you can set the `Filtered` property of the table to `True`, define a calculated field, try to cause errors (for example, by duplicating the value of the name field), add a check box to open or close the table, and so forth.

## Field Events

The `DbEvts` program shows the calls to the `OnChange` and `OnValidate` events of the field objects. Two other events, `OnSetText` and `OnGetText`, are not shown, because the handlers of these events are not simply called to indicate that an operation occurred. On the contrary, their event handlers must perform the operation of getting data from or setting it to the corresponding field objects.

These two events are quite special, and their use is not as simple as it might seem at first sight. For this reason, they require a separate example, named `FldText`. This is only a slight revision of the `DbAware` example described earlier in this chapter, replacing the `DBRadioGroup` control with a `DBListBox` control. The problem is that a `DBListBox` control directly connects with a string field, while I want to connect it with an integer field, with each value indicating an option. Of course, I don't want a user to see or select a number, so I have to map the numbers stored in the database to the strings visible on the screen. In the earlier example, the `DBRadioGroup` control provided that mapping. Now I have to use an alternative approach.

In the `FldText` example, the `Department` field has two handlers for the `OnGetText` and `OnSetText` events. In the `OnGetText` event handler, you can extract the numeric value of the `Sender` field and set the value of the `Text` reference parameter:

```
procedure TDbaForm.Table1DepartmentGetText(Sender: TField;  
  var Text: String; DisplayText: Boolean);  
begin  
  case Sender.AsInteger of  
    1: Text := 'Sales';  
    2: Text := 'Accounting';  
    3: Text := 'Production';  
    4: Text := 'Management';  
  else  
    Text := '[Error]';  
  end;  
end;
```

**WARNING**

In the code of the `OnGetText` event handler you cannot refer to the text of the field, for example, using the `DisplayText` property or the `GetData` method, since they would call the `OnGetText` event, in an infinite recursion.

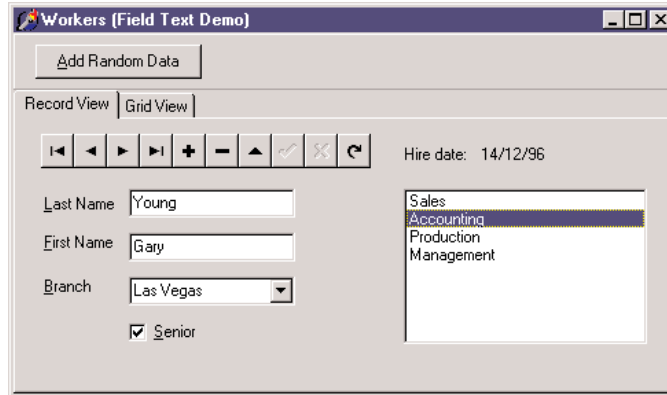
In the `OnSetText` event handler, you can examine the string and decide the value of the field according to the conversion rule, in this case a simple mapping of values done with an `if/then/else` statement:

```
procedure TDbaForm.Table1DepartmentSetText(Sender: TField; const Text: String);  
begin  
  if Text = 'Sales' then  
    Sender.Value := 1  
  else if Text = 'Accounting' then  
    Sender.Value := 2  
  else if Text = 'Production' then  
    Sender.Value := 3  
  else if Text = 'Management' then  
    Sender.Value := 4  
  else  
    raise Exception.Create ('Error in Department field conversion');  
end;
```

The effect is that not only is the value visible in the DBListBox (as you can see in Figure 13.19), it also shows up in the DBGrid. By contrast, in the DbAware example, the grid displayed the numeric value.

**FIGURE 13.19:**

The output of the FldText example, which demonstrates the use of the OnGetText and OnSetText events of the field objects



## Editing Dates with a Calendar

As a final example of the use of non-data-aware controls, the DbDates application shows how to use a MonthCalendar component to handle dates with a nice graphical component instead of a plain edit box. This example is based on the Events table from the DBDEMOS database, which lists Olympic events. The example uses (for the first time) a DBImage control, with the following settings (whose effect is illustrated in Figure 13.20):

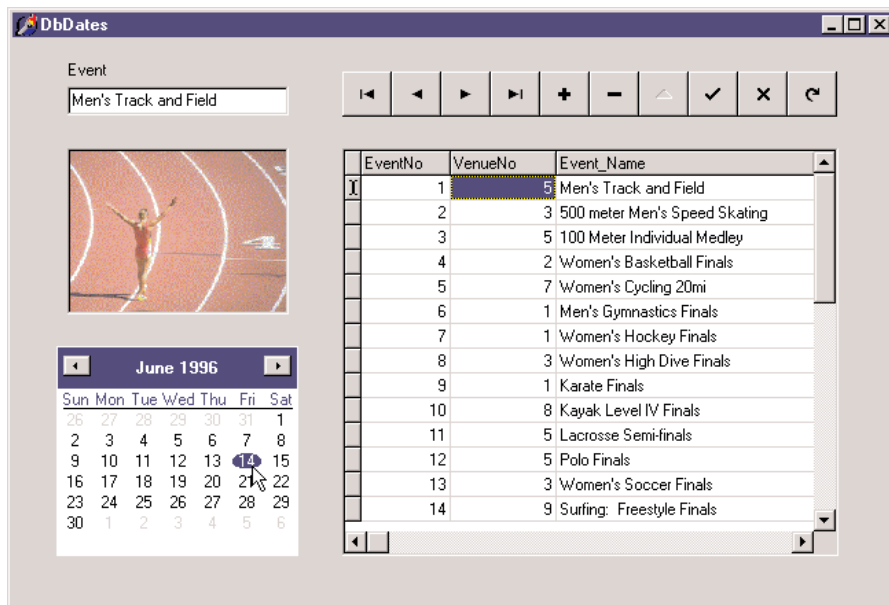
```
object DBImage1: TDBImage
  DataField = 'Event_Photo'
  DataSource = DataSource1
  Stretch = True
end
```

### NOTE

Graphic, memo, and BLOB fields in Delphi are handled exactly like other fields. Just connect the proper editor or viewer, and most of the work is done behind the scenes by the system.

**FIGURE 13.20:**

The selection of a date with the monthly calendar



Although the DBImage control works with no extra effort on our part, we must connect the MonthCalendar control with the corresponding field by handling two events of the DataSource control:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
    MonthCalendar1.Date := Table1Event_Date.Value;
end;
```

```
procedure TForm1.DataSource1UpdateData(Sender: TObject);
begin
    Table1Event_Date.Value := MonthCalendar1.Date;
end;
```

Besides copying the data back and forth, with the code listed above, the program must also put the table into edit mode as the user clicks the calendar control. The most obvious approach is to write a handler for the OnClick event of the control:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
    Table1.Edit;
end;
```

However, this code doesn't work properly. As you set the table in edit mode, the `OnDataChange` event is executed once more, resetting the selection in the calendar. The overall effect is that the user's first click doesn't change the selection. To avoid this problem we can set a flag in the `OnClick` event handler and test it in the `OnDataChange` event handler, or we can temporarily disconnect the second event handler. In the following code, I've taken the second approach:

```

procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
    // disconnect handler
    DataSource1.OnDataChange := nil;
    // set table in edit mode
    Table1.Edit;
    // reconnect handler
    DataSource1.OnDataChange := DataSource1DataChange;
end;

```

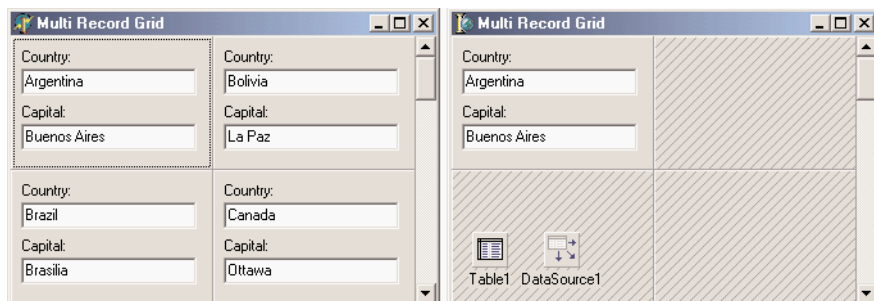
## A Multirecord Grid

So far we have seen that you can either use a grid to display records of a database table or build a form with specific data-aware controls for the various fields, accessing the records one by one. There is a third alternative: use a multirecord object (a `DBCtrGrid`), which allows you to place many data-aware controls in a small area of a form and automatically duplicate these controls for multiple records.

Here is what we can do to build the `Multi1` example. Create a new blank form, place a `Table` component and a `DataSource` component in it, and connect them to the `COUNTRY.DB` table. Now place a `DBCtrGrid` on the form, set its size and the number of rows and columns, and place two edit components connected with the `Name` and `Capital` fields of the table. To place these `DBEdit` components, you can also open the `Fields` editor and drag the two fields to the control grid. At design time, you simply work on the active portion of the grid (see Figure 13.21, on the right), and at run time, you can see these controls replicated multiple times (see Figure 13.21, on the left).

**FIGURE 13.21:**

The `DBCtrGrid` of the `Multi1` example at design time (on the right) and at run time (on the left)



You can simply set the number of columns and rows. Then each time you resize the control, the width and height of each panel are set accordingly. What is not available is a way to align the grid automatically to the client area of the form.

## Moving Control Grid Panels

To improve the last example, we might resize the grid using the `FormResize` method. We could simply write the following code (in the `Multi2` example):

```
procedure TForm1.FormResize(Sender: TObject);  
begin  
    DBCtrlGrid1.Height := ClientHeight - Panel1.Height;  
    DBCtrlGrid1.Width := ClientWidth;  
end;
```

This works, but it is not what I want. I'd like to increase the number of panels, not enlarge them. To accomplish this, we can define a minimum height for the panels and compute how many panels can fit in the available area each time the form is resized. For example, in `Multi2`, I've added one more statement to the `FormResize` method above, which now becomes

```
procedure TForm1.FormResize(Sender: TObject);  
begin  
    DBCtrlGrid1.RowCount := (ClientHeight - Panel1.Height) div 100;  
    DBCtrlGrid1.Height := ClientHeight - Panel1.Height;  
    DBCtrlGrid1.Width := ClientWidth;  
end;
```

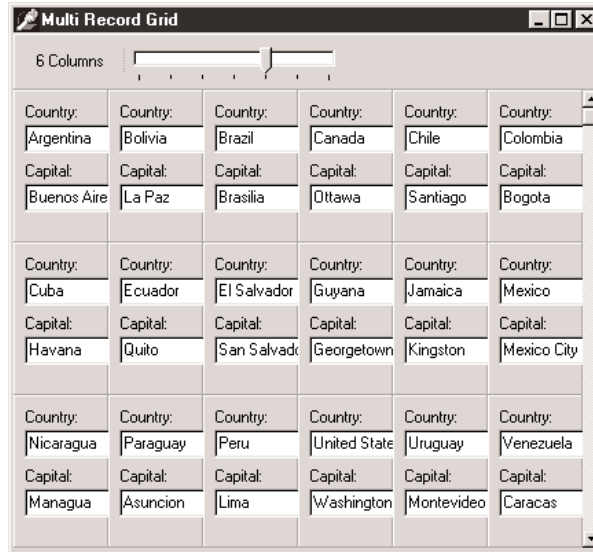
Instead of doing the same for the columns of the control grid component, I've added a `TrackBar` component to a panel. When the position of the trackbar changes (the range is from 2 to 10), the program sets the number of columns of the control grid and resizes it. In fact, if you simply set the number of columns, they'll have the same width as before. Here is the code of the trackbar's `OnChange` event handler:

```
procedure TForm1.TrackBar1Change(Sender: TObject);  
begin  
    LabelCols.Caption := Format ('%d Columns', [TrackBar1.Position]);  
    DBCtrlGrid1.ColCount := TrackBar1.Position;  
    DBCtrlGrid1.Width := ClientWidth;  
end;
```

This code and the `FormResize` method above allow you to change the configuration of the control grid at run time in various ways. You can see an example of a crammed version of the form in Figure 13.22.

**FIGURE 13.22:**

The output of the Multi2 example, with an excessive number of columns



## Handling Database Errors

Another important element of database programming is handling database errors in custom ways. Of course, you can let Delphi show an exception message each time a database error occurs, but you might want to try to correct the errors or simply show more details. There are basically three approaches you can use to handle database-related errors:

- You can wrap a try/except block around risky database operations, such as a call to the `Open` method of a `Query` or to the `Post` method of a dataset. This is not possible when the operation is generated by the interaction with a data-aware control.
- You can install a handler for the `OnException` event of the global `Application` object or use the `ApplicationEvents` component, as described in the next example.
- You can handle specific events of the datasets related to errors, as `OnPostError`, `OnEditError`, `OnDeleteError`, and `OnUpdateError`. These events will be discussed later in the example.

While most of the exception classes in Delphi simply deliver an error message, with database exceptions you see a list of errors, showing local BDE error codes and also the native error codes of the SQL server you are connected to. Besides the `Message` property, the `EDBEngineError` class has two more properties, `ErrorCount` and `Errors`. This last property is a list of errors:

```
property Errors[Index: Integer]: TDBError;
```

Each item within this list is an object of the class `TDBError`, which has the following properties:

```
type  
  TDBError = class  
    ...  
  public  
    property Category: Byte read GetCategory;  
    property ErrorCode: DBIResult read FErrorCode;  
    property SubCode: Byte read GetSubCode;  
    property Message: string read FMessage;  
    property NativeError: Longint read FNativeError;  
  end;
```

I've used this information to build a simple database program showing the details of the errors in a memo component. To handle all of the errors, the `DBError` example installs a handler for the `OnException` event of an `ApplicationEvents` component. The event handler simply calls a specific method used to show the details of the database error, in case it is an `EDBEngineError`:

```
procedure TForm1.ApplicationEvents1Exception (Sender: TObject; E: Exception);  
begin  
  Beep;  
  if E is EDBEngineError then  
    ShowError (EDBEngineError (E))  
  else  
    ShowMessage (E.Message);  
end;
```

I decided to separate the code used to show the error to make it easier for you to copy this code and use it in different contexts. Here is the code of the `ShowError` method, which outputs all of the available information to the `Memo1` component that I've added to the form:

```
procedure TForm1.ShowError(E: EDBEngineError);  
var  
  I: Integer;  
begin
```



```
Memo1.Lines.Add('');
Memo1.Lines.Add('Error: ' + (E.Message));
Memo1.Lines.Add('Number of errors: ' + IntToStr(E.ErrorCount));
// iterate through the Errors records
for I := 0 to E.ErrorCount - 1 do
begin
    Memo1.Lines.Add('Message: ' + E.Errors[I].Message);
    Memo1.Lines.Add('    Category: ' + IntToStr(E.Errors[I].Category));
    Memo1.Lines.Add('    Error Code: ' + IntToStr(E.Errors[I].ErrorCode));
    Memo1.Lines.Add('    SubCode: ' + IntToStr(E.Errors[I].SubCode));
    Memo1.Lines.Add('    Native Error: ' + IntToStr(E.Errors[I].NativeError));
    Memo1.Lines.Add('');
end;
end;
```

Besides this error-handling code, the program has a table and a query, along with the error-related event handlers. As already mentioned, you can install an event handler related to specific errors of a dataset. The three events `OnPostError`, `OnDeleteError`, and `OnEditError` have the same structure. Their handlers receive as parameters the dataset, the error itself, and an action you can request from the system; this can be set to `daFail`, `daAbort`, or `daRetry`:

```
procedure TForm1.Table1PostError(DataSet: TDataSet; E: EDatabaseError;
    var Action: TDataAction);
begin
    Memo1.Lines.Add (' -> Post Error: ' + E.Message);
end;
```

If you don't specify an action, as in the code above, the default `daFail` is used, and the exception reaches the global handler. Using `daAbort` stops the exception and can be used if your event handler already displays a message. Finally, if you have a way to determine the cause of the error and fix it, you can use the `daRetry` action.

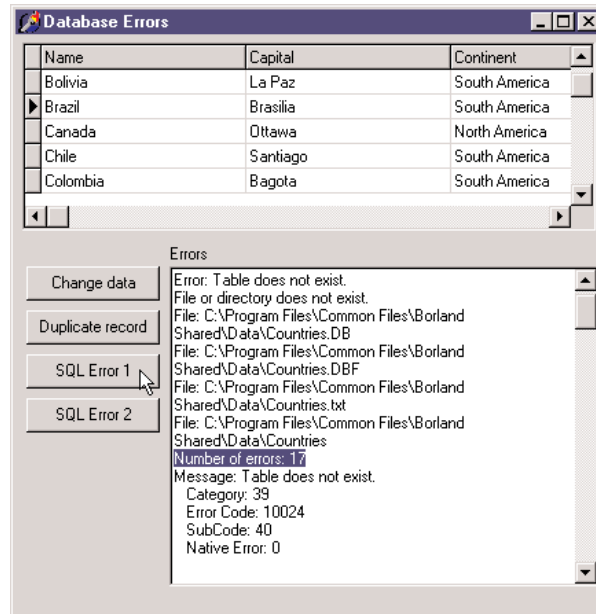
## NOTE

The fourth error event, `OnUpdateError`, has a different structure and is used along with cached updates as the information is sent back from the local cache to the database. This handler is important for handling update conflicts among different users as described in the next example.

The example has also a `DBGrid` connected with the table. You can use the `DBGrid` to perform some illegal operations, such as adding a new record with the same key as an existing one or trying to execute illegal SQL queries. Pressing the four buttons on the left of the memo generate errors, as you can see in Figure 13.23.

**FIGURE 13.23:**

The third button of the DBError form generates an exception with 17 database errors!



## What's Next?

In this chapter, we have seen examples of database access from Delphi programs. I have covered the basic data-aware components as well as the development of database applications based on standard controls. We've explored the internal architecture of the TDataSet class and of field objects, and discussed many of the events and properties shared by all datasets and used by all database applications. Even though most of the demonstrations used BDE tables and queries, in the entire text there was little or no code specific to those components. Most of what I've described equally applies to client/server applications, built with BDE, dbExpress, IBX, ADO or other dataset components.

We've discussed calculated fields, lookup fields, customizations of the DBGrid control, and many rather advanced techniques. What we really haven't delved into is the database and data-access side of the picture, which depends on the actual type of database engine and server you plan using. The next chapter will start focusing in this topic, with an in-depth overview of client/server development. Following chapters will put the accent on specific data-access technologies and provide even more advanced information.

# Client/Server Programming

- Overview of client/server
- Porting local applications
- Elements of database design
- Client/server with BDE
- The dbExpress library
- Using the ClientDataSet component
- Local databases with MyBase

In the last chapter, we examined Delphi's support for database programming, using local files (particularly Paradox) in most of examples but not focusing on any specific database technology. This chapter moves on to the use of SQL server databases, focusing on client/server development with the BDE and the new dbExpress technology. A single chapter cannot cover this complex topic in detail, so I'll simply introduce it from the perspective of the Delphi developer and add some tips and hints. The next chapter extends our discussion of client/server programming, providing some real-world examples. I'll use InterBase in both chapters, because this is the RDBMS (relational database management system), or SQL server, that is included in the Enterprise edition of Delphi and because it is a free and open-source server.

In a rapid application development tool such as Delphi, you can indeed take the same components and code developed for a local database application and use them in a client/server environment. However, this handy feature may prove to be dangerous to beginners, as a standard technique that works well for local access might become extremely inefficient in a client/server application.

## An Overview of Client/Server Programming

The database applications in previous chapters used the BDE to access data stored in files either on the local machine or on a networked computer. In both cases we used a file server, whose only role was to store the file on a hard disk, because the database engine (the BDE) was running exclusively on the computer that also hosted the application. In this configuration, when we query one of the tables, its data is first copied into a local cache of the BDE and then processed.

As an example, consider taking a table like EMPLOYEE (part of the InterBase sample database, which ships with Delphi), adding thousands of records to it, and placing it on a networked computer working as a file server. If we want to know the highest salary paid by the company, we can open a Table component (EmpTable) connected with the database table and run this code:

```
EmpTable.Open;
EmpTable.First;
MaxSalary := 0;
while not EmpTable.Eof do
begin
  if EmpTable.FieldName ('Salary').AsCurrency > MaxSalary then
    MaxSalary := EmpTable.FieldName ('Salary').AsCurrency;
  EmpTable.Next;
end;
```

The effect of this approach is to move all the data of the (large) table from the networked computer to the local machine, an operation that might take minutes. Because Delphi includes a Query component, you might think of using the following SQL code to compute this maximum value:

```
select Max(Salary) from Employee
```

In case of a local table, this query would be processed by the local SQL engine of the BDE, and the entire dataset of the table would still have to be moved from the networked computer to the local one, with similarly poor performance. But if you use InterBase and let the server execute the SQL code, only the result set—a single number—will need to be transferred to the local computer.

## NOTE

The two code excerpts above are part of the GetMax example, which includes some code to time the two approaches. Using the Table component on the small EMPLOYEE table takes about ten times longer than using the query, even if the InterBase server is installed on the client computer.

If you want to store a large amount of data on a central computer and avoid moving the data to client computers for processing, the only solution is to let the central computer manipulate the data and send back to the client only a limited amount of information. This is the foundation of client/server programming.

In general, you'll use an existing program on the server (an RDBMS) and write a custom client application that connects to it. Sometimes, however, you might even want to write both a custom client and a custom server, as in three-tier applications. Delphi support for this type of program—what has been called the MIDAS architecture—is covered in Chapter 17, “Multitier Database Applications with DataSnap.”

The *upsizing* of an application—that is, the transfer of data from local files to a SQL server database engine—is generally done for performance reasons and to allow for larger amounts of data. Going back to the previous example, in a client/server environment, the query used to select the maximum salary would be computed by the RDBMS, which would send back to the client computer only the final result, a single number! With a powerful server computer (such as a multiprocessor Sun SparcStation), the total time required to compute the result might be minimal.

However, there are also other reasons to choose a client/server architecture:

**The amount of data** A Paradox table cannot exceed 2 GB, but even around 300 MB you might start having serious speed problems, and errors in the indexes become more frequent.

**The need for concurrent access to the data** Paradox uses the Paradox.NET file to keep track of which user is accessing the various tables and records. The Paradox approach to

handling multiple users is based on *pessimistic locking*. When a user starts an editing operation on a record, none of the other users can do the same (to avoid any update conflict), as we saw in the last chapter. In a system with tens of users, this might lead to serious problems, because a single user might block the work of many others. SQL server databases, by contrast, generally use *optimistic locking*, an approach that allows multiple users to work on the same data and delays the concurrency control until the time the users send back some updates.

**Protection and security** An RDBMS usually has many more protection mechanisms than the simple password you can add to a Paradox table. When your application is based on files, a malicious or careless user might simply delete those vital files. When SQL servers are based on robust operating systems, instead, they provide multiple levels of protection, make backup easier, and often allow only the database administrator to modify the structure of the tables.

**Programmability** An RDBMS database can host business rules, in the form of stored procedures, triggers, table views, and other techniques we'll discuss in this and the next chapter. Choosing how to divide the application code between the client and the server is one of the main issues of client/server programming.

**Transaction control** Local files offer some support for transactions, but the transaction support provided by an RDBMS database is generally much greater. This is another important aspect of the overall robustness of the system.

## From Local to Client/Server

Now we can start focusing on particular techniques useful for client/server programming. Keep in mind that the general goal is to distribute the workload properly between the client and the server and reduce the network bandwidth required to move information back and forth.

The foundation of this approach is good database design, which involves both table structure and appropriate data validation and constraints, or business rules. Enforcing the validation of the data on the server is important, as the integrity of the database is one of the key aims of any program. However, the client side should include data validation as well, to improve the user interface and make the input and the processing of the data more user-friendly. It makes little sense to let the user enter invalid data and then receive an error message from the server, when we can prevent the wrong input in the first place.

### NOTE

If you use a CASE tool for the definition of the database, or import the definition in such a tool afterward, you can use Delphi's Case Wizard to generate a corresponding data dictionary and have the field objects created at design time automatically import the constraints specified on the server.

## Unidirectional Cursors

In local databases, tables are sequential files whose order is either the physical order or is defined by an index. By contrast, SQL servers work on logical sets of data, not related to a physical order. A *relational* database server handles data according to the relational model, a mathematical model based on set theory.

What is important for the present discussion is that in a relational database, the records (sometimes called tuples) of a table are identified not by position but exclusively through a primary key, based on one or more fields. Once you've obtained a set of records, the server adds to each of them a reference to the following one, which makes it fast to move from a record to the following one but terribly slow to move back to the previous record. For this reason, it is common to say that an RDBMS uses a *unidirectional* cursor. Connecting such a table or query to a DBGrid control is practically impossible, as this would make it terribly slow when browsing the grid backward.

The BDE helps a lot to handle unidirectional cursors, as it keeps in a local cache the records already loaded in the table. Thus, when we move to following records, they are requested from the SQL server; but when we go back, the BDE jumps in and provides the data. In other words, the BDE makes these cursors fully bidirectional, although this might use quite a lot of memory. When using dbExpress, which doesn't provide a similar caching system, a program needs to keep in memory the records it has already accessed. This can be easily accomplished by means of the ClientDataSet component.

### NOTE

The simple case of a DBGrid used to browse an entire table is common in local programs but should generally be avoided in a client/server environment. It's better to filter out only part of the records and only the fields you are interested in. Do you need to see a list of names? Return all those starting with the letter *A*, then those with *B*, and so on, or ask the user for the initial letter of the name.

If proceeding backward might result in problems, keep in mind that jumping to the last record of a table is even worse; usually this operation implies fetching all the records! A similar situation applies to the RecordCount property of datasets. Computing the number of records often implies moving them all to the client computer. This is the reason why the thumb of the vertical scrollbar of the DBGrid works for a local table but not for a remote one. If you need to know the number of records, run a separate query to let the server (and not the client) compute it. For example, you can see how many records will be selected from the EMPLOYEE table if you are interested in those records having a salary field higher than 50,000:

```
select count(*)  
from Employee  
where Salary > 50000
```

**TIP**

Using the SQL instruction `count(*)` is a handy way to compute the number of records returned by a query. Instead of the `*` wildcard, we could have used the name of a specific field, as in `count(First_Name)`, possibly combined with either `distinct` or `all`, to count only records with different values for the field or all the records having a non-`null` value.

## Parametric Queries and Null Values

Parametric queries are a very useful technique. Essentially, they allow you to run multiple queries with different result sets while the server only needs to work out the access strategy for solving the query once.

You can force this initial preparation of the query access strategy by calling the `Prepare` method of a `Query` component. With this operation, the server receives the query, checks its syntax, and while compiling it determines how to use indexes and other access techniques. Multiple executions of the query will be faster because these initial operations have already been executed once for all. Of course, you should call `Prepare` again if you change the SQL text of the query. Also, remember to call `Unprepare` at the end, to free some BDE resources.

Note that some powerful SQL servers can do the same operation by caching the requests and automatically determining that you are sending the same request twice. If the server is smart enough, preparing the query might result in little or no performance gain.

**NOTE**

When you write parametric queries against a SQL server, you should consider `null` values with care. In fact, to test for a `null` value, you should not write a `field = null` test, but use the specific expression `field is null` instead.

## Elements of Database Design

Although this is a book on Delphi programming and not on databases, I feel it's quite important to discuss a few elements of good (and modern) database design. The reason is simple: if your database design is incorrect or convoluted, you'll either have to write terribly complex SQL statements and server-side code, or write a lot of Delphi code to access your data, possibly even fighting against the design of the `TDataSet` class.

### Entities and Relations

The *classic* relational database design approach, based on the entity–relation (E-R) model, involves having one table for every entity you need to represent in your database, with one field for each data element you need plus one field for every one-to-one or one-to-many



relation to another entity (or table). For many-to-many relations, instead, you'll need a separate table.

As an example of a one-to-one relation, consider a table representing a university course. This would have a field for each relevant data element (name and description, room where it is held, and so on) plus a single field indicating the teacher. The data of the teacher, in fact, should not be stored within the course data, but in a separate table, as it might be referenced from elsewhere.

The schedule of each course can include an undefined number of hours in different days, so they cannot be added inside the same table describing the course. Instead, this information must be placed on a separate table, including all of the schedules, with a field referring to the class each schedule is for. In a one-to-many relation, such as this, "many" records of the schedule table point back to the same "one" record of the course table.

A more complex situation is required to store which student is taking which class. Students cannot be listed directly in the course table, as their number is not fixed, and the classes cannot be stored within the student's data for the same reason. In a similar many-to-many relation, the only approach is to have an extra table representing the relation, which lists references to students and courses.

## Normalization Rules

The *classic* design principles include a series of so-called *normalization* rules. The goal of these rules is to avoid duplicating data in your database (not only for saving space, but mainly to avoid ending up with incongruous data). For example, you don't repeat all of the customer details in each order, but refer to a separate customer entity. This way you save memory, but when the customer details change (for example, because of a change of address) all of the orders of this customer will reflect the new data. Other tables that relate to the same customer will probably be automatically updated as well.

Normalization rules imply using codes for commonly repeated values. For example, if you have a few different shipment options, you won't use a string-based description for these options within the orders table, but would rather use a short numeric code, mapped to a description in a separate lookup table.

This last rule, which should not be taken to the extreme, is to avoid having to join a large number of tables for every query. You can either account for some de-normalization (leaving a short shipment description within the orders table) or use the client program to provide the description, again ending up with a formally incorrect database design. This last option is practical only when you use a single development environment (let's say, Delphi) to access this database.

## From Primary Keys to OIDs

In a relational database, records are not identified by a physical position (as in Paradox and other local databases) but only by the data within the record itself. Typically, you don't need the data of all of the fields to identify a record, but only a subset of the data, forming the so-called *primary key*. If the fields that are part of the primary key must identify an individual record, their value must be different for each possible record of the table.

### NOTE

Technically, many database servers add internal record identifiers to the tables, but this happens only for internal optimizations and has little to do with the logical design of a relational database. Also, these internal identifiers work differently in different SQL servers and might even change among versions, a good reason not to rely on them.

The early incarnations of the relational theory dictated the use of *logical keys*, which means selecting one or more records that indicate an entity without risk of any confusion. This is often easier to say than to accomplish. For example, company names are not generally unique, and even the company name and its location don't provide a complete guarantee. Moreover, if a company changes its name (not an unlikely event, as Borland can teach us) or its location, and you have references to the company within other tables, you must change all those references as well, with the risk of ending up with dangling references.

For this reason, and also for efficiency (using strings for references implies using a lot of space in secondary tables, where references often occurs), logical keys have been invariably phased out for physical or surrogate keys. *Physical keys* refer to a single field of the table identifying an element in a unique way. For example, each person in the U.S. has a Social Security number (SSN), but almost every country has a tax ID or other government-assigned number that identifies each person. The same typically exists for companies. Although these ID numbers are guaranteed to be unique, they might change depending on the country (creating troubles for the database of a company also selling its goods abroad) or even within a single country (to account for new tax laws). They are also often inefficient, as they might be quite large (Italy, for example, uses a 16-character code, letters and numbers, to identify people).

Another common approach is to use *surrogate keys*, which are basically numbers identifying each record, in the form of client codes, order numbers, and so on. These surrogate keys are commonly used in database design. However, in many cases, these end up being some sort of *logical identifiers*, with client codes showing up all over the places, not a great idea overall.

---

**WARNING** The situation becomes particularly troublesome when these surrogate keys also have a meaning and must follow specific rules. For example, companies must number invoices with unique and consecutive numbers, without leaving holes in the numbering sequence. This situation is extremely complex to handle programmatically, if you consider that only the database can determine these unique consecutive numbers when we send new data to it. At the same time, we need to identify the record before we send it to the database, otherwise we won't be able to fetch it again. Practical examples of how to solve this situation are discussed in the next chapter.

## OIDs to the Extreme

An extension to the use of surrogate keys is the use of a unique identifier, also called object identifier (OID). An OID is a number, or a string with sequence of numbers and digits, added to each record of each table representing an entity (and at times even to records of tables representing relations). Differently from client codes, invoice numbers, SSN, or purchase order numbers, OIDs are totally random, without any sequencing rule, and never visible to the end user. This means you can still use surrogate keys (if your company is used to them) along with OIDs, but all the external references to the table will be based on OIDs.

Another common rule suggested by the promoters of this approach (which is part of the theories supporting object-relational mapping) is the use of system-wide unique identifiers. If you have a table of client companies and a table of employees, you might wonder why you should use a unique ID for such diverse data. The reason is that, if you do so, you'll be able to sell goods to an employee without having to duplicate the employee information into the customers table, but simply referring to the employee in your order and invoice. An order is placed by someone identified by an OID, and this OID can refer to many different tables (but of course not all of them).

---

**NOTE** Using OIDs and the object-relational mapping is an advanced element of the design of Delphi database applications. My personal suggestion is to investigate this topic before embracing medium or large-size Delphi projects, as the benefit can be relevant (after some investment in studying this approach and building some basic support code).

## External Keys and Referential Integrity

Getting back to the standard database design, the keys identifying a record (whichever their type) can be used as external keys in other tables, for example to represent the various types of relations discussed earlier. All SQL servers are capable of verifying these external references, so that you cannot refer to a nonexistent record of another table. These referential integrity constraints are expressed when you create a table.

Besides not being allowed to add references to nonexistent records, you're generally prevented from deleting a record if there are external references to it. Some SQL servers go one step further: As you delete a record, instead of simply denying the operation, they can automatically delete all records that refer to it from other tables.

## More Constraints

Besides the uniqueness of primary keys and the referential constraints, you can generally use the database to impose more validity rules on the data. You can ask for specific columns (such as those referring to a tax ID or a purchase order number) to include only unique values. You can impose uniqueness of the values of multiple columns—for example, to indicate you cannot run two classes in the same room at the same time.

In general, simple rules can be expressed imposing constraints on a table, while more complex rules generally imply the execution of stored procedures activated by triggers (every time the data changes, for instance, or there is new data).

Again, there is much more to proper database design, but the simple elements discussed in this section can provide a starting point, or a good refresher.

---

**NOTE**

For more on the Data Definition Language and Data Manipulation Language of SQL, see the bonus chapter "Essential SQL" on the *Mastering Delphi* CD.

## Client/Server with the BDE

Now let's consider how Delphi fits into the client/server picture. How does it help us build client/server applications? As I've mentioned, you can still use all the components and techniques discussed in the Chapter 13, "Delphi's Database Architecture," although in some cases alternate approaches will help you leverage the power of the RDBMS your application is dealing with.

As a starting point, let's cover a few considerations on Delphi client/server development using the BDE and its components. After this I'll move to dbExpress, which in Delphi 6 is the recommended general solution for client/server development.

---

**NOTE**

For a list of the alternative approaches Delphi 6 offers for database access, see the initial part of the preceding chapter.

## SQL Links

The BDE doesn't know how to handle the RDBMS; it uses some further drivers, called SQL Links, to perform this operation. As an alternative, the BDE can also interact with ODBC drivers. Borland provides native BDE drivers for InterBase, Oracle, Informix, Microsoft SQL Server, Sybase, and DB2.

If the BDE is still required on the local machines, it can actually be very efficient. For example, when you use the pass-through mode for queries, the BDE doesn't try to interpret the SQL code but passes it directly to the RDBMS server. This allows you to use a server's specific SQL commands and also to speed up the execution. The pass-through mode is activated using the BDE Administrator utility.

Having the BDE between the client and the server can also help in building applications designed to work with multiple servers. In practice, however, it's not easy to do this and still obtain the best performance, because of differences in the SQL dialects understood by each SQL server. In particular, data types are handled differently by the various servers. If the same table were placed on two servers that have data type differences, Delphi would need to use two different `TField` objects (which creates a few headaches if you want to define the fields at design time).

## The Database Component

In local BDE applications, programmers usually refer to the database by indicating the alias of the file path in the `DatabaseName` property of the `Table` and `Query` components. A better approach is to use the `Database` component to define a local alias and then let all the `DataSet` components refer to this local alias.

As an example, consider the components of the `GetMax` application mentioned at the beginning of this chapter:

```
object Database1: TDatabase
  AliasName = 'IBLOCAL'
  Connected = True
  DatabaseName = 'IB'
  LoginPrompt = False
  Params.Strings = (
    'USER NAME=SYSDBA'
    'PASSWORD=masterkey')
  SessionName = 'Default'
end
object EmpTable: TTable
  DatabaseName = 'IB'
  TableName = 'EMPLOYEE'
end
```

```
object EmpQuery: TQuery
  DatabaseName = 'IB'
  SQL.Strings = (
    'select Max(Salary) from Employee ')
end
```

In a client/server application, using the Database component is almost mandatory, as it is required to define connectivity and login parameters (the user name and password, as you can see in the Params property above) and to handle transactions.

Keep in mind that the Database component establishes a connection with the RDBMS, representing one of the clients of the system. As such, on most servers it requires a license, and your organization is typically paying for a fixed number of licenses. If the same application or the same computer uses multiple connections to the server, it can count as multiple clients! Fortunately, by setting the `KeepConnection` property of the Database component, you can specify whether to keep the database connection active even when there is no active `DataSet` component using the connection. If your program can fetch some data and then operate on it locally, disconnecting from the server might help you conserve licenses.

## BDE Table and Query Components in Client/Server

In Delphi there are two BDE components you use to access an existing database table: Table and Query. When building client/server applications, programmers tend to use the Query component exclusively, but that is certainly not mandatory, and there are cases in which using the simpler Table component has no drawback. Here's a quick look at the pros and cons of both components:

- While the Table component should not be used to access a large table, it can work perfectly well with a small lookup table. By opening a Table component, you don't transfer the entire content of the table to the local machine; the data is moved only when you access specific records.
- Consider also that with the Table component, the BDE asks the server first for the table structure and then for the table data. These two steps are necessary for setting up the proper internal structures of the BDE, and they are not executed by the Query component. If you activate the BDE's Schema Caching feature, the logical structure of the table will be kept locally, saving this extra step. Of course, this might create problems if the logical structure of the table changes on the server.
- One problem with the Table component is that the BDE mimics a bidirectional cursor by caching the data locally. With a Query component, instead, you can specify whether you want this caching or not with the `Unidirectional` property.

- Another point to consider is that you can generally edit the result of a simple query, sending the data back to the SQL server. This is accomplished by setting the `RequestLive` property to `True`. For more complex queries, however, you'll need to use an `UpdateSQL` component, something we'll discuss later in this chapter.
- When trying to minimize the data moved between the server and the client, you need to consider the size of each record as well as the total number of them. When you select only a few fields with a query, only part of the data is considered. A `Table` component, instead, always entails transferring the entire record to the local machine, even if you've filtered out some fields using the `Fields` editor. The same problem takes place when you ask for a live query (by setting the `RequestLive` property). In this case, the BDE needs to see the entire record in order to send back the proper update commands. This means that selecting all the records of a table with a live query is equivalent to using the `Table` component.
- The `Query` component is not limited to `select` SQL statements; you can also use it to insert or delete records. When the `Query` component returns a dataset, you generally activate it with the `Open` method (or with the equivalent operation, setting the `Active` property to `True`). When the `Query` component is used to perform an operation on the server, you activate it by calling the `ExecSQL` method.

## Using Table and Query Filters

One way to limit the amount of data returned by a table is to filter it. Using the `Filter` property of the `Table` component, you can specify a condition similar to the `where` clause of a query. When you work with local databases, the filter is applied by the BDE, but with a SQL server, the BDE passes the condition to the server in the query generated for the table. This makes filtered tables very portable between local and client/server applications.

---

**WARNING** The situation is different if you filter the records in the Pascal code, using the `OnFilterRecord` event. In this case, all the records are sent to the client application, which does its own custom filtering.

If you use a filter with a `Query` component, the filtering operation will always be performed locally by the BDE, even when you are working with a SQL server. In this case, the BDE asks the server for the entire result set of the query. This would be reasonable only when the user of the application changes the filtering condition often. For a query, only the local filter will be modified, and the data in the local cache will be used. For a table, the BDE will generate an updated query to be executed.

## Live Queries and Cached Updates

When working with local data, it is very common to use grids and other visual controls, edit the data, and send it back to the database. We've already seen that using a DBGrid might cause problems when working with an RDBMS, as moving on the grid might send numerous data requests to the server, creating a huge amount of network traffic.

When you use the Query component to connect to some data, you cannot edit the data unless its `RequestLive` property is set to `True`. If you are working with local tables, the query is always elaborated by the BDE with the Local SQL engine. The BDE will allow for a live query only if it is quite simple: All joins should be outer joins; there cannot be a `distinct` key; there can be no aggregation, no `group by` or `having` clause, no subqueries, and no `order by` unless supported by an index; and there are other rules you can find in Delphi's Help.

If you are working with a SQL server, setting a live query will put the BDE in control of the query, instead of the server. When connected to a SQL server, a live query behaves like a Table component. (So it makes sense to use the table anyway, in these cases.)

### TIP

Most SQL servers, including InterBase, allow you to define updateable views based on the result of a `select` statement that the Local SQL engine of the BDE won't consider updateable. Then you can simply hook a Table component to the view, letting the SQL server do the work and bypassing the Local SQL engine of the BDE.

If the BDE determines that the dataset cannot be updated, it sets the `CanModify` property to `False`. The `DataSource` component checks this value before allowing an editing operation. A solution to this problem is to avoid the use of data-aware controls, as discussed in the last chapter, and use specific SQL queries to update, insert, and delete records.

A better approach is to automate this process (retaining the capabilities of the data-aware controls) by using the `UpdateSQL` component together with the Query component. The `UpdateSQL` can be used only in conjunction with cached updates, a topic discussed in the last chapter. The basic idea is that the update operations are kept in a local cache until the program calls the `ApplyUpdates` method of the Query component. This operation corresponds to the execution of a series of `update`, `insert`, and `delete` SQL operations on the server, using the data in the cache. The required SQL commands are held by the `UpdateSQL` component, which has a design-time editor you can use to generate these SQL commands almost automatically.

Cached updates solve the live queries issue, reduce network traffic, define a standard way to solve updates conflicts, and reduce the server load, but they require more memory on the client computer.



**TIP**

A much better approach than using cached updates is to rely on the ClientDataSet component, which is extremely powerful and allows you to do similar things yet leaving you a lot more programmatic control.

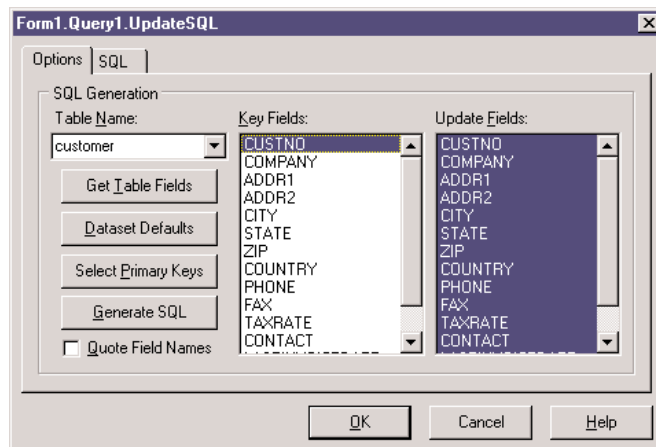
## The UpdateSQL Component

The role of the UpdateSQL component is to provide a query with the update statements required to make its result set editable. Its key properties are DeleteSQL, InsertSQL, and ModifySQL, but the most important element is the UpdateObject property of the related Query component. The update SQL statements are executed when you apply the cached updates, sending the changes to the server. Because cached updates maintain the information on the original records, the updates usually indicate which record to update by passing the original data. This is the only way we have to identify a record on a SQL server, and this technique also helps the server to track any updates on the same record done by other users.

All this setup might seem to imply a lot of work, but it is actually very simple. After you've written a query, you can connect the UpdateSQL component to it and activate the component editor, as shown in Figure 14.1.

**FIGURE 14.1:**

The UpdateSQL component editor in action



This component editor has two tabs. The first indicates the criteria used to generate the SQL statements for adding, deleting, or modifying records. With a join, you can select the table to update and the fields involved. When you've completed this step, click the Generate SQL button and the editor will move to the second tab, where you can inspect the generated SQL code for the three operations.

## The UpdateSQL Example

To demonstrate the real power of the UpdateSQL component, I've built a complex example called UpdateSQL, based on the Employee, Department, and Job tables of the IBLocal database we've used in the past. Here is the textual description of the UpdateSQL component of the example:

```

object EmpUpdate: TUpdateSQL
  ModifySQL.Strings = (
    'update EMPLOYEE'
    'set'
    '  FIRST_NAME = :FIRST_NAME,'
    '  LAST_NAME = :LAST_NAME,'
    '  SALARY = :SALARY,'
    '  DEPT_NO = :DEPT_NO,'
    '  JOB_CODE = :JOB_CODE,'
    '  JOB_GRADE = :JOB_GRADE,'
    '  JOB_COUNTRY = :JOB_COUNTRY'
    'where'
    '  EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '  (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE,'
    '  JOB_GRADE, JOB_COUNTRY)'
    'values'
    '  (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, '
    '  :JOB_GRADE, :JOB_COUNTRY)')
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where'
    '  EMP_NO = :OLD_EMP_NO')
end

```

To delete the employee records, the program uses a stored procedure, which is already available in the sample database and is connected to the following component:

```

object spDelEmployee: TStoredProc
  DatabaseName = 'AppDB'
  StoredProcName = 'DELETE_EMPLOYEE'
  ParamData = <
    item
      DataType = ftInteger
      Name = 'EMP_NUM'
      ParamType = ptInput
    end>
end

```

The OnUpdateRecord event of the Query component uses the stored procedure instead of the default UpdateSQL component for deleting records. Here is the code of the event handler:

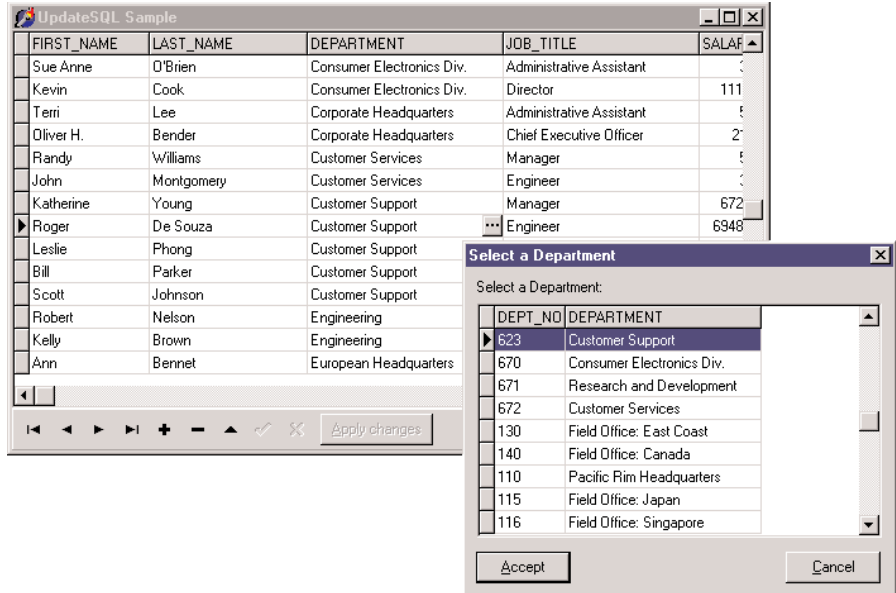
```
procedure TdmData.qryEmployeeUpdateRecord(DataSet: TDataSet;
UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  // when deleting the record, use the stored procedure
  if UpdateKind = ukDelete then
    begin
      // assign emp_no value
      with dmData do
        spDelEmployee.Params[0].Value := qryEmployeeEMP_NO.01dValue;
      try
        // invoke stored procedure that tries to delete employee
        dmData.spDelEmployee.ExecProc;
        UpdateAction := uaApplied; // success
      except
        UpdateAction := uaFail;
      end;
    end
  else
    try
      // apply updates
      dmData.EmpUpdate.Apply(UpdateKind);
      UpdateAction := uaApplied;
    except
      UpdateAction := uaFail;
    end;
  end;
```

Notice that because we perform the update operation directly, we must indicate in the UpdateAction parameter whether it succeeds or not. This code is part of the data module. The main form, visible at run time in Figure 14.2, has a couple of extra features. If the user closes the form with any updates pending, the OnCloseQuery event of the form displays a warning message, allowing the user to apply the updates or skip them:

```
procedure TMainForm.FormCloseQuery(Sender: TObject;
var CanClose: Boolean);
var
  Res: Integer;
begin
  with dmData do
    if qryEmployee.UpdatesPending then
      begin
        Res := MessageDlg (CloseMsg, mtInformation, mbYesNoCancel, 0);
        if Res = mrYes then
          AppDB.ApplyUpdates ([qryEmployee]);
        CanClose := Res <> mrCancel;
      end;
    end;
  end;
```

**FIGURE 14.2:**

The main form of the UpdateSql example along with a secondary form



The second feature is the use of a secondary form to update the fields that are related to other tables—the fields involved in the joins. The program uses two secondary dialog boxes, which get the data from other two Query components. The dialog boxes are displayed when the user clicks the ellipsis button of the DBGrid control, in the OnEditButtonClick event. Here is the first part of this event handler, related to the selection of the department:

```

procedure TMainForm.DBGrid1EditButtonClick(Sender: TObject);
begin
    // check whether this is the department field
    if DBGrid1.SelectedField = dmData.qryEmployeeDEPARTMENT then
        with TfrmDepartments.Create(self) do
            try
                dmData.qryDepartment.Locate('DEPT_NO',
                    dmData.qryEmployeeDEPT_NO.Value, []);
                if ShowModal = mrOk then
                    with dmData do
                        begin
                            if not (qryEmployee.State in [dsEdit, dsInsert]) then
                                qryEmployee.Edit;
                            qryEmployeeDEPT_NO.Value := qryDepartment.Fields[0].Value;
                            qryEmployeeDEPARTMENT.Value := qryDepartment.Fields[1].Value;
                        end;
                    finally
                        Free;
                    end
                else // similar code for the job fields...

```

Finally, the Apply button simply calls the ApplyUpdates method if there are pending updates and then refreshes the data of the query:

```
procedure TMainForm.btnApplyClick(Sender: TObject);  
begin  
  with dmData do  
    if qryEmployee.UpdatesPending then  
      begin  
        AppDB.ApplyUpdates([qryEmployee]);  
        // refresh the data  
        qryEmployee.Close;  
        qryEmployee.Open;  
        btnApply.Enabled := False;  
      end;  
end;
```

If you run this program, you'll notice that even if the underlying query is read-only, you can change data directly in the DBGrid, as you would do with a regular Table component. The visual operations you do are temporarily stored in the cache; then, when you issue the update operation, the UpdateSQL and the StoredProc components provide the actual code. Also keep in mind that the Salary field has some constraints (defined in the sample database), so you have to change it carefully to avoid errors on the server when the changes are applied.

## Update Conflicts

When you are working with local tables, using cached updates might cause concurrency problems. A plain edit operation usually places a lock on the table, so that the other users cannot modify the same record until the first user has posted the updates. The previous chapter covered locking and concurrency issues in detail.

When working with SQL servers, however, the default locking behavior is optimistic. Multiple users can update the same records, and only when the data is sent back does the server verify the original data of the record before updating it, potentially raising an error. More precisely, the update statement uses one or more original fields to locate the record you want to update. If you use all fields and another user has changed the record, then the server will not find the original record and will cause an update error.

You can manually control this behavior either in the code of the UpdateSQL component (indicating to include all the fields read in the query) or by using the UpdateMode property of the Table and Query components. The default value, upWhereAll, indicates that the update query will have a where clause with all the original fields of the record. In many cases, the fact that another user has modified a field different from those we have modified is not an error. We can set the upWhereChanged mode to let Delphi generate an exception and show an error message only if the current and the other user have both modified the same fields. The

third alternative is to use the key field only to identify the record, which means that update conflicts will be ignored and that the last user posting the data will simply override any previous change. As you can imagine, this is generally an option to avoid in a client/server, multi-user environment.

## Using Transactions

Whether you are working with a SQL server, you should use *transactions* to make your applications more robust. The idea of a transaction can be described as a series of operations to be considered as a single, “atomic” whole that cannot be split.

An example may help to clarify the concept. Suppose you have to raise the salary of each employee of a company by a fixed rate, as we did in the Total example of the preceding chapter. Now if during the operation an error occurs, you might want to undo the previous changes. If you consider the operation “raise the salary of each employee” as a single transaction, it should either be completely done or completely ignored. Or consider the analogy with financial transactions—if only part of the operation is performed, because of an error, you might end up with a missed credit or with some extra money!

Working with database operations as transactions serves a useful purpose. You can start a transaction and do several operations that should all be considered parts of a single larger operation; then, at the end, you can either commit the changes or *roll back* the transaction, discarding all the operations done up to now. Typically, you might want to roll back a transaction if an error occurred during its operations.

Handling transactions in Delphi is quite simple. By default, each edit/post operation is considered a single *implicit* transaction, but you can alter this behavior by handling them explicitly. Simply use the following three methods of the BDE Database component (other database connection components have similar methods):

**StartTransaction** marks the beginning of a transaction.

**Commit** confirms all the updates to the database done during the transaction.

**Rollback** returns the database to its state prior to starting the transaction.

The Database component determines the transaction isolation level using the `TransactionIsolation` property. When one user starts a transaction and modifies data, should such changes be visible to other users? And what happens if the user rolls back the transaction? To such questions there isn't a universal answer; every programmer should try to answer them according to the requirements or business rules of the application. There are three alternative values for transaction isolation in the BDE:

**tiDirtyRead** makes the updates of a transaction immediately visible to other transactions and users even before they are committed. This is the only possibility for local databases, which have very limited transaction support.

**tiReadCommitted** makes available to other transactions only the updates already committed.

**tiRepeatableRead** hides every other transaction started by other users after the current one. Following repeat calls within a transaction will always produce the same result, as if the database took a snapshot of the data when the current transaction started.

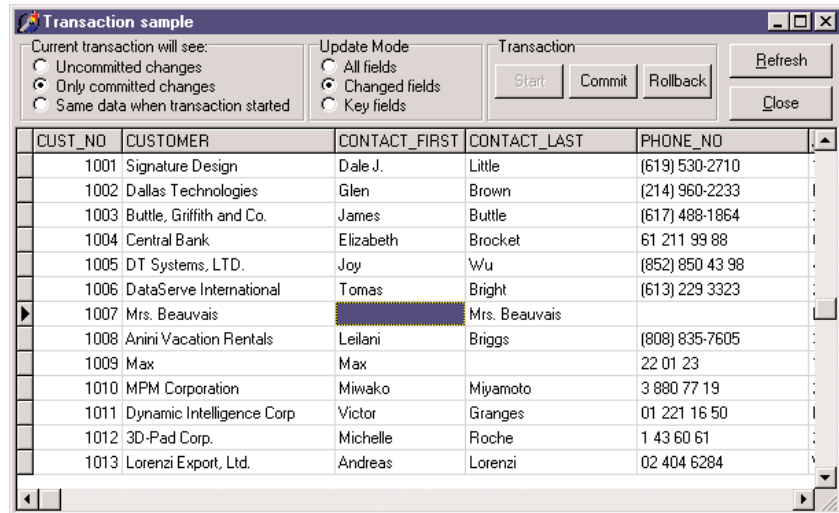
Most but not all SQL servers support only the most advanced levels. The default choice should be **tiReadCommitted**, which is quite powerful but not too heavy on the SQL server (as it adds very few internal locks).

As a general suggestion, transactions should involve only a minimal number of updates (only those strictly related and part of a single atomic operation) and should be kept short in time. You should avoid transactions that wait for user input to complete them, as the user might be temporarily gone and the transaction might remain active for a long time. Using update statements on multiple records and using cached updates can help us make the transactions small and fast.

To further inspect transactions and experiment with the update mode of the Table component, you can use the TranSample application. As you can see in Figure 14.3, you can simply use the radio buttons to choose the alternatives, and click the push buttons on the right of the toolbar to manually start, commit, and roll back a transaction. To get a real idea of the different effects, you should run multiple copies of the program (provided you have enough licenses on your InterBase server).

**FIGURE 14.3:**

The TranSample application allows you to test the transaction isolation of a database and the update modes of a table.



## Using SQL Monitor

Just as you need a debugger to test a Delphi application, you need some tools to test how a client/server application behaves and to speed it up if possible. In particular, it is very important to look at the information moving from the client to the server (the explicit SQL requests our program does and those added by the BDE) and from the server to the client (the actual data). This is what the SQL Monitor tool included in Delphi Enterprise is for.

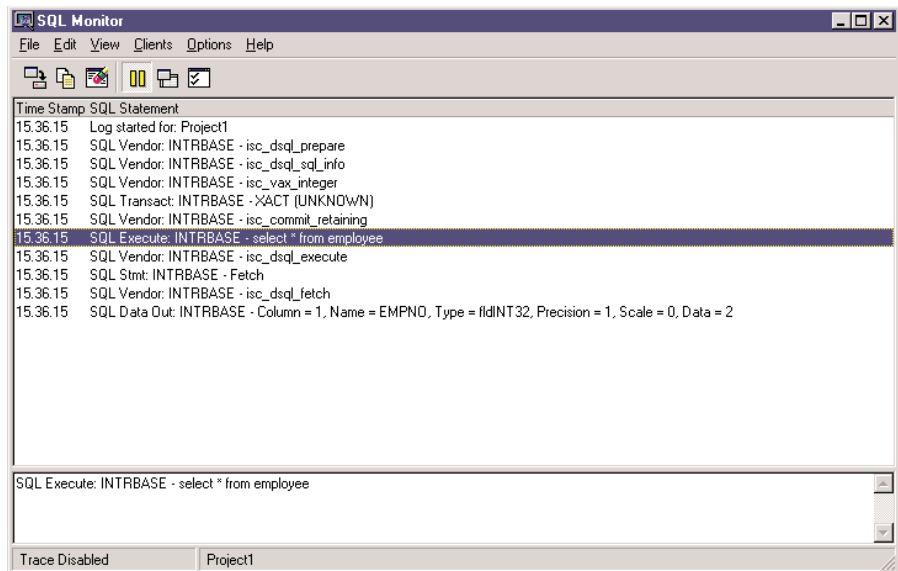
### NOTE

Notice that the SQL Monitor tool is specific to the BDE. The dbExpress and InterBase Express component sets have similar monitoring capabilities, directly embedded inside specific components.

As you can see in Figure 14.4, the central window of SQL Monitor shows a list of the low-level commands sent to the server. The bottom portion of the window shows the selected line of the above list on multiple rows, which helps when the line is too long.

**FIGURE 14.4:**

The SQL Monitor running



To use SQL Monitor, simply select the client program you want to inspect. Then set the proper trace options (by using the corresponding speed button or the Options > Trace Options command). The available options are listed in Table 14.1.



**TABLE 14.1:** The Trace Options of the SQL Monitor

Trace Option	Meaning
Prepared Query Statement	Enables tracing of the SQL statements every time they are prepared.
Executed Query Statement	Traces all the SQL statements sent to the server.
Input Parameters	Shows input parameters as they become available. This is important for testing whether the parameters are correct.
Fetches Data	Shows the data sent by the server (a very slow operation).
Statement Operations	Shows the requests preceding the execution of a SQL statement, such as the allocation, preparation, and parsing of the input.
Connect/Disconnect	Shows the connection and disconnection events. This is an important test when the <b>KeepConnection</b> of the <b>Database</b> component is set to <b>False</b> , as the client won't maintain the connection with the server but will establish it only as needed (with the side effect of reducing the number of licenses required, in some cases). Looking at the frequency of these events might help you understand whether it is better to keep the connection active or not.
Transactions	Traces the transactions, including those activated automatically by the BDE if you don't use transactions directly.
Blob I/O	Shows the data about BLOB fields.
Miscellaneous	Traces other operations that don't fit any of the above categories.
Vendor Errors	Shows server error messages.
Vendor Calls	Shows client API calls.

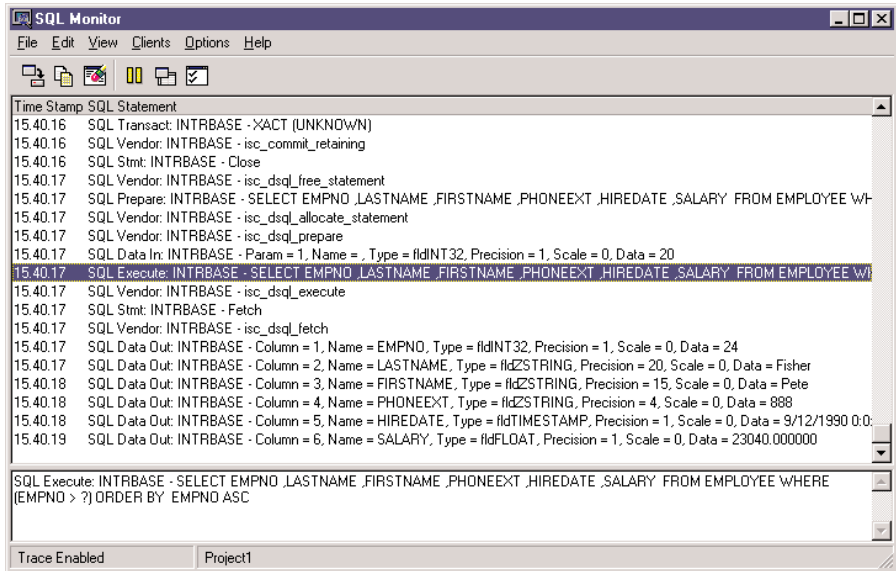
SQL Monitor is useful for seeing if the SQL statements sent by the BDE to the server are correct, but it also helps you see how many operations are done behind the scenes. Along with the time-stamp information for each operation, the number of operations can give some clue about your application's speed (although you should remember that the presence of SQL Monitor slows down the connection quite a lot).

In other words, SQL Monitor should be your guide in determining how to speed up your client/server application, using some of the tricks described in this chapter. At the same time, however, it takes a lot of experience and a good understanding of SQL to interpret its output properly.

As an example of the use of SQL Monitor, we can test what happens when we use the **Filter** property of a **Table** component. In a new project, simply place a **Table**, a **DataSource**, and a **DBGrid**. Select a database and a table (for example, the **EMPLOYEE** table of **IBLocal**) and set the **Filtered** property to **True** and the **Filter** property to **EmpNo>20**. If you now run the program, SQL Monitor will show you that the **select** statement generated by the BDE has a **where** clause corresponding to the filter. You can see this situation in Figure 14.5.

**FIGURE 14.5:**

SQL Monitor showing SQL statements generated by a Table component



## Performance Tuning

Besides using the SQL Monitor (or another monitoring technique) to determine the potential bottlenecks in your applications, you can do several things to speed up your client/server programs. The key element to keep in mind—as I've stressed many times in this chapter—is to reduce the network traffic, by reducing the result sets returned by the server both in the number of records and in the size of each.

Besides a good overall database design and a good Delphi implementation of it, there are many settings you can check. The following tips might come in handy, but they won't help as much as a better design!

- In InterBase, you can set an automatic sweep (or "garbage collection") interval. The operation is also automatically performed when you do a backup. Because a sweep slows down the database, it should not be done too frequently. However, if you never do it, the database will keep track of many leftover deleted records, reducing the overall performance and using extra memory.
- Use indexes on the fields used more often, particularly if you sort the result set on them. Keep in mind, though, that a good RDBMS will add at least temporary indexes for you. Using indexes can speed up queries quite a lot, particularly if the indexed fields are used to join two tables.

*Continued on next page*

- If you sort a field in descending order, a corresponding descending index might help.
- If you're an expert user, you might examine the *query plan*, the approach used by the server to perform a query, which is displayed (for example) when you use WISQL. The query plan will show you whether the SQL server is using indexes. In some cases, you might need to modify some complex queries to help the query optimizer built into the RDBMS.
- Check the server settings, including its cache, to obtain the best overall performance. The operating system cache on the server computer might help as well. In InterBase, if you want to perform all the updates physically, you can set the Forced Writes option in the Maintenance ➤ Database Properties menu of the InterBase Server Manager.
- Whenever possible, avoid an excessive use of transactions and try to keep them short and focused. Use a local cache instead of transactions (or together with them) to let the client computer do some more work for you, and skip some costly server operations.
- Handle transactions directly, disabling the auto-commit feature of the BDE; to do this, set `SQLPASSTHRU MODE` to `SHARED NOAUTOCOMMIT`. (You can set this and other BDE features described in this list with the BDE Administrator program.)
- If you have no licensing problems, set the `KeepConnecti on` property of the BDE Database component to `True`.
- With the BDE, set `TRACE MODE` to `0` when you are not debugging, to avoid having the drivers send trace strings to the debugger and slowing down the operations. Also the other monitoring services should be disabled when you're not debugging the application.
- In the BDE, enable schema caching (set `ENABLE SCHEMA CACHE` to `TRUE`). This setting reduces the time required to open a table, as the client doesn't need to ask for the metadata. You can also use the Delphi `FieldDefs` and `StoreDefs` properties of the Table component to store the metadata directly in the client program.
- With Microsoft and Sybase SQL Servers, try to set the `PACKETSIZE` parameter to a minimum of 4 KB, also modifying the corresponding value on the server. With these servers, also check that the `DRIVER FLAGS` parameter is set to `0`. If it is `2048`, queries will be executed in asynchronous mode and will be much slower.
- With ORACLE, DB/2, and ODBC drivers, try to fine-tune the `ROWSET SIZE` parameter until you obtain the best performance.
- With the InterBase driver, if you don't use explicit transactions, set the `DRIVER FLAGS` parameter to `4096`. This value enables *soft commits*, meaning that after each commit or rollback operation the open cursors won't have to be refreshed.

## The dbExpress Library

As I mentioned in Chapter 13, one of the most notable new features of Delphi 6 is the adoption of the dbExpress database access library, a new SQL server access layer introduced by Borland in its Kylix product for Linux, and now with Delphi 6 also for Windows. As I've already provided a general overview of dbExpress in the preceding chapter, let's focus right away on the technical details.

### Working with Unidirectional Cursors

The motto of dbExpress could be “fetch but don't cache.” The key difference between this library and BDE or ADO is that dbExpress can only execute SQL queries and fetch the results in a *unidirectional cursor*. In “unidirectional” database access, you can move from one record to the next, but you cannot get back to a previous record of the dataset. This is because the library doesn't store the data it has retrieved in a local cache, but only passes it from the database server to the calling application.

Using a unidirectional cursor might sound like a limitation, and it really is! Besides having problems with the navigation, you cannot connect a database grid to a dataset like this. So what is a unidirectional dataset good for?

- You can use a unidirectional dataset for reporting purposes. In a printed report, but also an HTML page or an XML transformation, you move from record to record, produce the output, and that's it. No need to get back to past records and, in general, no interaction of the user with the data. Unidirectional datasets are probably the best option for Web and multitier architectures.
- You can use a unidirectional dataset to feed a local cache, such as the one provided by a ClientDataSet component. At this point, you can connect visual components to the in-memory dataset and operate on it with all the standard techniques, including the use of visual grids. You can freely navigate and edit the data in the in-memory cache, but also control it far better than with the BDE or ADO.

The important thing to notice is that, in these circumstances, avoiding the caching of the database engine actually saves time and memory. The library doesn't have to use extra memory for the cache and doesn't need to waste time storing data, duplicating information. Over the last couple of years, many programmers moved from BDE-based cached updates to the ClientDataSet component, which provides more flexibility in managing the content of the data and update information they keep in memory. However, using a ClientDataSet on top of the BDE (or ADO) exposes you to the risk of having two separate caches, actually wasting a lot of memory.

Another advantage of using the ClientDataSet component is that its cache supports editing operations, and the updates stored in this cache can be applied to the original database server by the DatasetProvider component. This component can generate the proper SQL update statements, and can do so in a more flexible way than the BDE (although ADO is quite powerful in this respect). In general, the provider can also use a dataset for the updates, but this isn't directly possible with the dbExpress dataset components.

## Platforms and Databases

A key element of the dbExpress library is its availability for both Windows and Linux, in contrast to all the other database engines available for Delphi (BDE and ADO), which are available only for Windows. Notice, though, that some of the database-specific components, such as InterBase Express, are also available on multiple platforms.

When you use dbExpress, you are provided with a common framework, which is independent from the actual SQL database server you are planning to use. dbExpress comes with drivers for MySQL, InterBase, Oracle, and IBM DB2. These drivers are available as separate DLLs you have to deploy along with your program or as compiled units you can link into the executable file.

### NOTE

It is actually possible to write custom drivers for the dbExpress architecture. This is documented in details in the paper *dbExpress Draft Specification*, published on the Borland Community Web site. At the time of this writing, this document is at <http://community.borland.com/article/0,1410,22495,00.html>. You'll probably be able to find third-party drivers. For example, there is a free one (available also in Kylix), which bridges dbExpress and ODBC.

## The dbExpress Components

The VCL components used to interface the dbExpress library encompass a group of dataset components plus a few ancillary ones. To differentiate these components from other database-access families, the components are prefixed with the letters *SQL*, underlining the fact that they are used for accessing RDBMS servers.

These components include a database connection component, a few dataset components (a generic one, three specific versions for tables, queries, and stored procedures, and one encapsulating a ClientDataSet component), and a monitor utility.

### The SQLConnection Component

The TSQLConnection class inherits from the TCustomConnection component, and handles database connections, the same as its sibling classes (the Database, ADOConnection, and IBConnection components).

**TIP**

Unlike other component families, in dbExpress the connection is compulsory. In each of the dataset components, you cannot specify directly which database to use, but can only refer to a `SQLConnection`.

The connection component uses the information available in the `drivers.ini` and `connections.ini` files, which are the only two configuration files of dbExpress (these files are saved by default under `\Program Files\Common Files\Borland Shared\DBExpress`). The first, `drivers.ini`, lists the available dbExpress drivers, one for each supported database. For each driver, there is a set of default connection parameters. For example, the `InterBase` section reads as follows:

```
[Interbase]
GetDriverFunc=getSQLDriverINTERBASE
LibraryName=dbexpint.dll
VendorLib=GDS32.DLL
BlobSize=32
CommitRetain=True
Database=database.gdb
Password=masterkey
RoleName=RoleName
TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

The parameters indicate the dbExpress driver DLL (the `LibraryName` value), the entry function to use (`GetDriverFunc`), the vendor client library, and some more specific parameters that depend on the database. If you read the entire `drivers.ini` file, you'll see that the parameters are really database-specific. I have to say that some of these parameters don't make a lot of sense at the driver level, such as the database to connect to, but the list includes all the available parameters, regardless of their actual usage.

The `connections.ini` file provides the database specific description. This list resembles the aliases of the BDE, and you can enter multiple connection details for every database driver. The connection describes the physical database you want to connect to. As an example, this is the portion for the default `IBLocal` definition:

```
[IBLocal]
BlobSize=32
CommitRetain=True
Database=database.gdb
DriverName=Interbase
Password=masterkey
RoleName=RoleName
TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

As you can see by comparing the two listings, this is a subset of the parameters of the driver. When you create a new connection, the system will copy the default parameters from the driver; you can then edit them for the specific connection—for example, providing a proper database name. Each connection relates to the driver for its key attributes, as indicated by the `DriverName` property.

The important thing to notice is that these initialization files are used only at design time. In fact, when you select a driver or a connection at design time, the values of these files are copied to corresponding properties of the `SQLConnection` component, as in this example:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  DriverName = 'Interbase'
  GetDriverFunc = 'getSQLDriverINTERBASE'
  LibraryName = 'dbexpint.dll'
  LoginPrompt = False
  Params.Strings = (
    'BlobSize=-1'
    'CommitRetain=False'
    'Database=c:\program files\interbase corp\interbase6\examples\' +
    'database\employee.gdb'
    'DriverName=Interbase'
    'LocaleCode=0x0000'
    'Password=masterkey'
    'RoleName=RoleName'
    'ServerCharSet=ASCII'
    'SQLDialect=1'
    'Interbase TransIsolation=ReadCommitted'
    'User_Name=sysdba'
    'WaitOnLocks=True')
  VendorLib = 'GDS32.DLL'
end
```

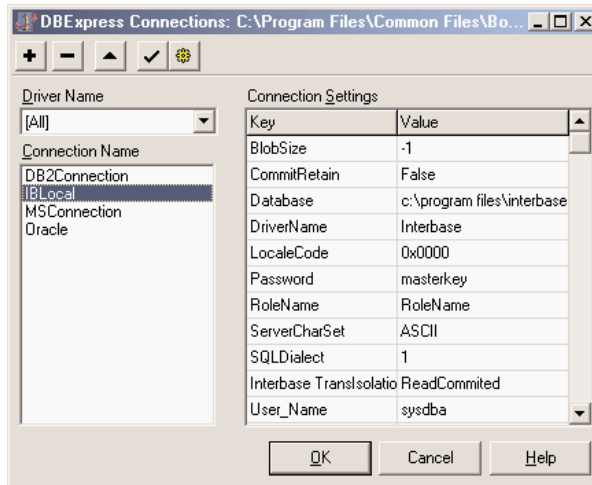
At run time, your program will rely on the properties to have all the required information, so you don't need to deploy the two INI files along with your programs. In theory, the files will be required if you want to change the `DriverName` or `ConnectionName` properties at run time. However, in case you want to connect your program to a new database, you can set directly the relevant properties.

When you add a new `SQLConnection` component to an application, you can proceed in different ways. You can set up a driver, using the list of values available for the `DriverName` property, and then select a predefined connection, by selecting one of the values available in the `ConnectionName` property. This second list is filtered according with the driver you've already selected. As an alternative, you can start by selecting directly the `ConnectionName` property, which in this case includes the entire list.

Instead of hooking up an existing connection, you can define a new one (or see the details of the existing connections) by double-clicking the SQLConnection component and launching the dbExpress Connection Editor (Figure 14.6). This editor lists, on the left side, all of the predefined connections, for a specific driver or all of them, and allows you to edit the connection properties using the grid on the right. You can use the toolbar buttons to add, delete, rename, and test connections, and to open the read-only dbExpress Drivers Settings window, shown in Figure 14.7.

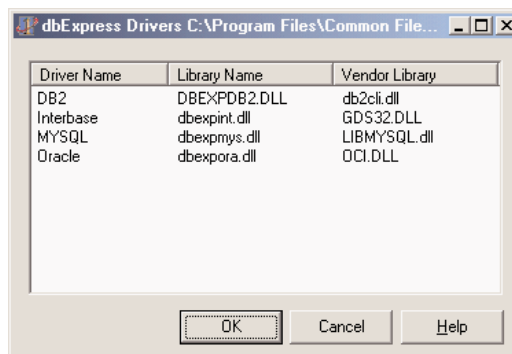
**FIGURE 14.6:**

The dbExpress Connection Editor



**FIGURE 14.7:**

The dbExpress Drivers Settings window of the dbExpress Connection Editor





Besides editing the predefined connection settings, the dbExpress Connection Editor allows you also to select a connection for the `SQLConnection` component. This is what the OK button is for. Notice, in fact, that if you change some of the settings, the data is immediately written to the INI files: clicking the Cancel button doesn't revert your editing!

If you want to define access to a database, editing the connection properties is certainly the suggested approach. This way when you need to access the same database from another application, or another connection within the same application, all you need to do is to select the connection. However, since this operation copies the connection data, notice that updating the connection doesn't automatically refresh the values within other `SQLConnection` components referring to the same named connection: you have to reselect the connection these other components refer to. In this respect, the predefined connections are very different from the BDE aliases.

What really matters for the `SQLConnection` component is the value of its properties. Driver and vendor libraries are listed in properties you can freely change at design time (although you'll rarely want to do this), while the database and other database-specific connection settings are specified in the `Params` properties. This is a string list including information such as the database name, the user name and password, and so on. In practice, you could set up a `SQLConnection` component by setting up the driver and then assigning the database name directly in the `Params` property, forgetting about the predefined connection. I'm not suggesting this as the best option, but it is certainly a possibility; the predefined connections are handy, but when the data changes, you still have to manually refresh every `SQLConnection` component.

Actually, to be complete, I have to mention that there is an alternative. You can set the `LoadParamsOnConnect` property to indicate that you want to refresh the component parameters from the initialization files every time you open the connection. In this case, a change in the predefined connections will be reloaded when you open the connection, at either design time or run time. At design time, this provides a handy technique (which has the same effect as reselecting the connection), but using it at run time means you'll also have to deploy the `connections.ini` file, which can be a good idea or an inconvenient one, depending on your deployment environment.

The only property of the `SQLConnection` component that is not related to the driver and database settings is `LoginPrompt`. Setting it to `False` allows you to provide a password skipping the login request, both at design time and run time. If this is very handy for development, it can reduce the security of your system. Of course this is also the option you'll want to use for unattended connections, for example on a Web server.

## The dbExpress Dataset Components

The dbExpress component's family provides four different dataset components: a generic dataset, a table, a query, and a stored procedure. The latter three components are provided for compatibility with the equivalent BDE components and have similarly named properties. If you don't have to port existing code, you should tend to use the general `SQLDataSet` component, which can be used to execute a query but also to access a table or a stored procedure.

The first important thing to notice is that all of these datasets inherit from a new special base class, `TCustomSQLDataSet`. This class and its derived classes represent unidirectional datasets, with the key features I've already described. In practice, this means that the browse operations are limited to calling `First` and `Next`, while `Prior`, `Last`, `Locate`, the use of bookmarks, and all other navigational features are disabled.

### NOTE

Technically, some of the moving operations call the `CheckBiDirectional` internal function and eventually raise an exception. `CheckBiDirectional` refers to the public `IsUnidirectional` property of the `TDataSet` class, which you can eventually use in your own code to disable operations that are illegal on unidirectional datasets.

Besides having limited navigational capabilities, these datasets have no editing support, so a lot of methods and events common to other datasets are not available. For example, there is no `AfterEdit` or `BeforePost` event.

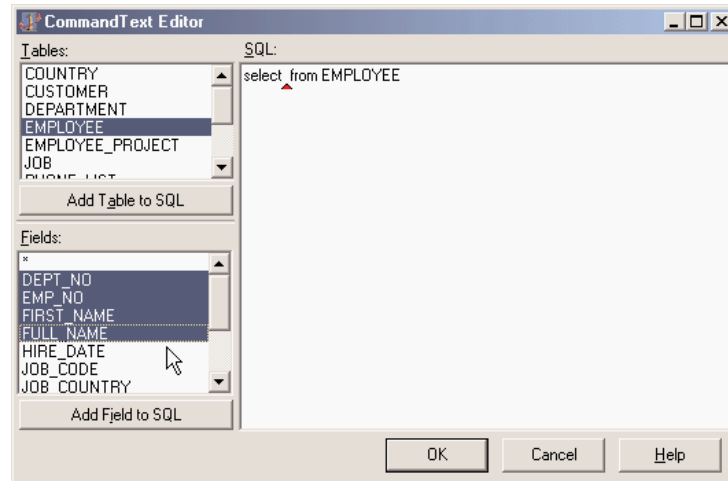
As I mentioned earlier, of the four dataset components for dbExpress, the fundamental one is `TSQLDataSet`, which can be used both to retrieve a dataset and to execute a command. The two alternatives are activated by calling the `Open` method (or setting the `Active` property to `True`) and by calling the `ExecSQL` method.

The `SQLDataSet` component can retrieve an entire table, or use a SQL query or a stored procedure for reading a dataset or issuing a command. The `CommandType` property determines one of the three access modes. The possible values are `ctQuery`, `ctStoredProc`, and `ctTable`, which determine the value of the `CommandText` property (and also the behavior of the related property editor in the Object Inspector). For a table or stored procedure, the `CommandText` property indicates the name of the related element of the database, and the editor provides a drop-down list with the possible values. For a query, the `CommandText` property stores the text of the SQL command, and the editor provides a little help in building the SQL query (in case it is a select statement). You can see the editor in Figure 14.8.

When you use a table, the component will generate a SQL query for you, as dbExpress targets only SQL databases. The generated query will include all the fields of the table, and if you specify the `SortFieldNames` property, it will include a `sort by` directive.

**FIGURE 14.8:**

The CommandText Editor used by the SQLDataSet component for queries



The three *specific* dataset components offer a similar behavior, but you specify the SQL query in the SQL string list property, the stored procedure in the StoredProcName property, and the table name in the TableName property, as in the three corresponding BDE components.

## The SQLClientDataSet Component

The SQLClientDataSet is a combination of three components: the SQLDataSet component, a (hidden) provider, and the ClientDataSet. The idea is to be a helper, as you need only one component instead of three, which must also be connected. However, it doesn't surface all of the properties and events of the underlying components, so in complex situations, it's better to use the various components it stands for. I'll cover this and other variations of the ClientDataSet component later in this chapter, after I discuss the ClientDataSet itself in detail.

## The SQLMonitor Component

The final component of the dbExpress group is the SQLMonitor, used to log the requests sent from dbExpress to the database server. This monitor provides capabilities similar to the stand-alone SQL Monitor application, which is bound to the BDE and cannot be used with the dbExpress library (and the analogous IBXMonitor component of the InterBase Express family, as we'll see in the next chapter).

## The TimeStamp Field Type

Along with dbExpress, Delphi 6 introduces the `TSQLTimeStampField` field type, mapped to the timestamp data type that many SQL servers have (InterBase included). This data type is now available also in Delphi and is called `TSQLTimeStamp`. A time stamp is a simple record-based representation of a date or time, quite different from the floating-point representation used by the `TDateTime` data type. A time stamp is defined as:

```
TSQLTimeStamp = packed record
  Year : SmallInt;
  Month : Word;
  Day : Word;
  Hour : Word;
  Minute : Word;
  Second : Word;
  Fractions : LongWord;
end;
```

A time stamp field can automatically covert standard date and time values using the `AsDateTime` property (as opposed to the native `AsSQLTimeStamp` property). You can also do custom conversions and further manipulation of time stamps by using the routines provided by the `SqlTimSt` unit, including functions like `DateTimeToSQLTimeStamp`, `SQLTimeStampToStr`, and `VarSQLTimeStampCreate`.

## A Simple dbExpress Demo

After that introduction, let's have a look at an actual demonstration, highlighting the key features of these components and showing how to use the `ClientDataSet` to provide caching and editing support for the unidirectional datasets. In another example, later on, I'll show you an example of native use of the unidirectional query, with no caching and editing support required.

The standard visual application based on dbExpress uses this series of components:

- The `SQLConnection` component provides the connection with the database and the proper dbExpress driver.
- The `SQLDataSet` component, which is hooked to the connection (via the `SQLConnection` property), indicates which SQL query to execute or table to open (using the `CommandType` and `CommandText` properties discussed earlier).
- The `DataSetProvider` component, connected with the dataset, extracts the data from the `SQLDataSet` and can generate the proper SQL update statements.

- The ClientDataSet component reads from the data provider and stores all the data (if its PacketRecords property is set to -1) in memory. This component has a lot of extra features and provides the actual data to the application, with full navigation and editing capabilities. You'll need at least to call its ApplyUpdates method to send the actual updates back to the database server (through the provider).
- The DataSource component allows you to surface the data from the ClientDataSet to the visual data-aware controls.

As I mentioned earlier, the picture can be simplified by using the SQLClientDataSet, which replaces the two datasets and the provider. The SQLClientDataSet combines most of the properties of the components it replaces. For a simple example, you'll have to set the DBConnection property for connecting to the proper database, the CommandType and CommandText properties to specify which data to fetch, and the PacketRecords property to indicate how many records to retrieve in each block. You'll also need to call to the ApplyUpdates method to send the actual updates back to the database.

These are the key properties of the core components of the DbxSingle example:

```
object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
end
object SQLClientDataSet1: TSQLClientDataSet
    CommandText = 'EMPLOYEE'
    CommandType = ctTable
    DBConnection = SQLConnection1
end
```

As an alternative, the DbxMulti example uses the entire sequence of components:

```
object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
end
object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
end
object ClientDataSet1: TClientDataSet
    ProviderName = 'DataSetProvider1'
end
object DataSource1: TDataSource
    DataSet = ClientDataSet1
end
```

Both examples also have some visual controls: a grid and a toolbar based on the action manager architecture.

## Applying Updates

What is important to do in every example based on a local cache, like the one provided by the `ClientDataSet` and `SQLClientDataSet` components, is to write the local changes back to the database server. This is typically accomplished by calling the `ApplyUpdates` method. But when should you call it? You can either keep the changes in the local cache for a while and then apply a bunch of updates at once, or post each change right away. In these two simple examples, I've gone for the latter approach, attaching the following event handler to the `AfterPost` (fired after an edit or an insert operation) and `AfterDelete` events of the `ClientDataSet` components:

```
procedure TForm1.DoUpdate(DataSet: TDataSet);  
begin  
    // immediately apply local changes to the database  
    SQLClientDataSet1.ApplyUpdates(0);  
end;
```

If you want to apply all the updates in a single batch, you can either do this when the form is closed or the program ends, or let a user do the update operation by selecting a specific command. We'll explore some of these alternatives when discussing the `ClientDataSet` component in more detail.

## Monitoring the Connection

Another feature, which I've added only to the `DbxSingle` example, is the monitoring capability offered by the `SQLMonitor` component. In the example, the component is activated as the program starts.

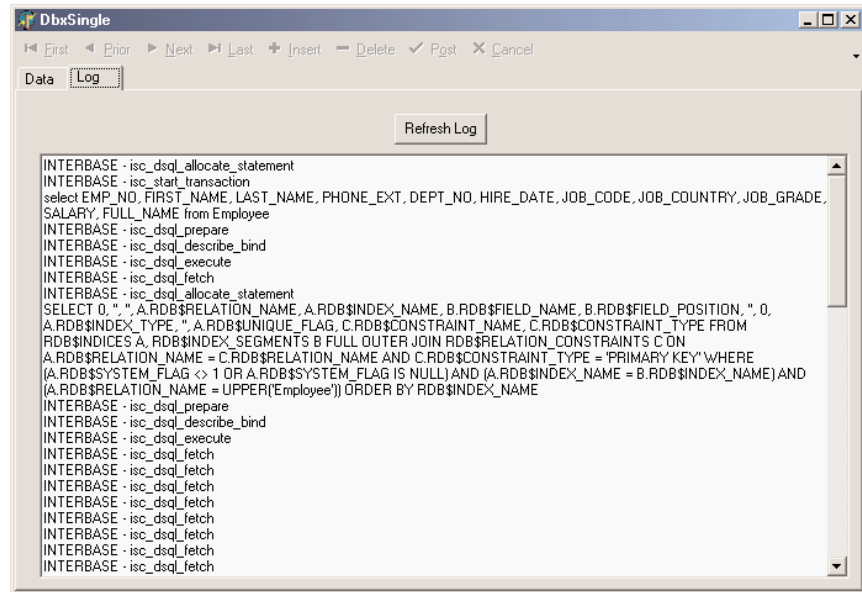
Every time there is a tracing string available, the component fires the `OnTrace` event to let you choose whether to include the string in the log. If the `LogTrace` parameter of this event is `True` (the default value), the component logs the message in the `TraceList` string list and fires the `OnLogTrace` event to indicate that a new string has been added to the log.

The component can also automatically store the log into the file indicated by its `FileName` property, but I haven't used this feature in the example. All I've done is to handle the `OnLogTrace` event, adding the last message to a memo component with the following code (and the output of Figure 14.9):

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject;  
    CBIInfo: pSQLTRACEDesc);  
begin  
    MemoLog.Lines.Add (CBIInfo.pszTrace);  
end;
```

**FIGURE 14.9:**

A sample log obtained by the SQLMonitor in the DbxSingle example.



Also, a button allows a user to refresh the memo with the entire content of the trace list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    MemoLog.Lines := SQLMonitor1.TraceList;
end;
```

## Controlling the SQL Update Code

If you run the DbxSingle program and change, for example, the telephone number of an employee, the monitor will log this update operation:

```
update EMPLOYEE set
    PHONE_EXT = ?
where
    EMP_NO = ? and
    FIRST_NAME = ? and
    LAST_NAME = ? and
    PHONE_EXT = ? and
    HIRE_DATE = ? and
    DEPT_NO = ? and
    JOB_CODE = ? and
    JOB_GRADE = ? and
    JOB_COUNTRY = ? and
    SALARY = ? and
    FULL_NAME = ?
```

The structure of the update statement depends on the `UpdateMode` property of the `SQL-ClientDataSet` component. Trying to use `upWhereChanged` or `upWhereKeyOnly`, however, causes an error, as the component is unable to determine which are the key records, and the provider doesn't generate a correct update statement. In fact, it simply tries to update the record based on the specific field changed, without including the key field in the where statement:

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  PHONE_EXT = ?
```

The update statement should rather be:

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  PHONE_EXT = ?
```

How can we obtain the proper update call? Considering we cannot directly attach a component like the `UpdateSQL` (which is a BDE-only component), a simple solution would be to force the inclusion of the key field. This can be accomplished in the `ClientDataSet` architecture by turning on the `pfInKey` flag of the `ProviderOptions` property of the source field.

This can easily be accomplished in the `DbxMulti` example, after adding persistent fields for the `SQLDataSet` component, but the problem is that the database library should locate the key fields automatically. In the `DbxSingle` example, we have no control on the source dataset and its fields, so the only solution is to write the update statements in a totally custom way, with quite some effort.

## NOTE

We'll be able to discuss this type of problem again as we get into the details of the `ClientDataSet` component, the `Provider`, the `Resolver`, and other technical details later in this chapter and in Chapter 17.

## Accessing Database Metadata with *SetSchemaInfo*

All RDBMS systems use special-purpose tables (generally called *system tables*) for storing metadata, such as the list of the tables, their fields, indexes, and constraints, and any other system information. As `dbExpress` provides a unified API for working with different SQL servers, it provides also a common way for accessing metadata. The `TSQLDataSet` component has a method, `SetSchemaInfo`, which fills the dataset with system information. This `SetSchemaInfo` method has three parameters:

**SchemaType** indicates the type of information requested and includes `stTables`, `stSysTables`, `stProcedures`, `stColumns`, and `stProcedureParams`.



**SchemaObject** indicates the object you are referring to, such as the name of the table for which you are requesting the columns.

**SchemaPattern** is a filter, so that you can limit your request to tables, columns, or procedures starting with the given letters. This is very handy if you use prefixes to identify groups of elements.

For example, in the SchemaTest program, a button reads into the dataset all of the tables of the connected database:

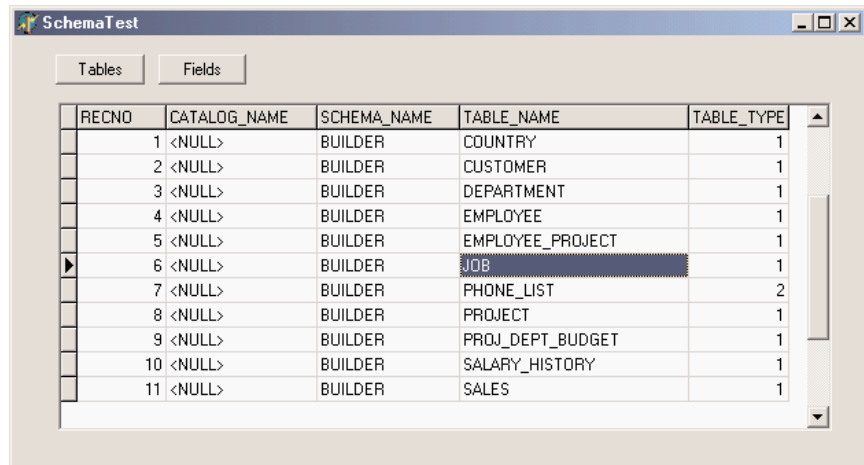
```
ClientDataSet1.Close;  
SQLDataSet1.SetSchemaInfo (stTables, '', '');  
ClientDataSet1.Open;
```

The program uses the usual group of dataset provider, client dataset and data source component to display the resulting data in a grid, as you can see in Figure 14.10. After you've retrieved the tables, you can select a row of the grid and press the second button to see a list of the fields of this table:

```
SQLDataSet1.SetSchemaInfo (stColumns, ClientDataSet1['Table_Name'], '');  
ClientDataSet1.Close;  
ClientDataSet1.Open;
```

**FIGURE 14.10:**

The SchemaTest example allows you to see the tables of a database and the columns of a given table.



The screenshot shows a window titled "SchemaTest" with two buttons: "Tables" and "Fields". Below the buttons is a table with the following data:

RECNO	CATALOG_NAME	SCHEMA_NAME	TABLE_NAME	TABLE_TYPE
1	<NULL>	BUILDER	COUNTRY	1
2	<NULL>	BUILDER	CUSTOMER	1
3	<NULL>	BUILDER	DEPARTMENT	1
4	<NULL>	BUILDER	EMPLOYEE	1
5	<NULL>	BUILDER	EMPLOYEE_PROJECT	1
6	<NULL>	BUILDER	JOB	1
7	<NULL>	BUILDER	PHONE_LIST	2
8	<NULL>	BUILDER	PROJECT	1
9	<NULL>	BUILDER	PROJ_DEPT_BUDGET	1
10	<NULL>	BUILDER	SALARY_HISTORY	1
11	<NULL>	BUILDER	SALES	1

Besides accessing database metadata, dbExpress provides a way to access to its own configuration information, including the installed drivers and the configured connections. The unit `DbConnAdmin` defines a `TConnectionAdmin` class for this purpose, but the aim of this support is probably limited to dbExpress add-on utilities for developers, as letting end users access multiple databases in a totally dynamic way is not very common.

**TIP**

The DbxExplorer demo included in Delphi 6 shows how to access both dbExpress administration files and schema information. Also check the help file under “The structure of metadata datasets” within the section “Developing database applications.”

## A Round-Up on dbExpress

After we’ve delved a little more into the dbExpress architecture and SQL components, I can try to add a few comments about this solution and its alternatives. On the whole, dbExpress provides a much neater architecture, compared to the BDE. In particular, I like it a lot when a database engine does things for me but lets me control what’s going on and fine-tune every element. Ready-to-use defaults, such as those provided by the `SQLClientDataSet` component, are nice, but it is important for me in real-world applications to be able to take full control and write the exact SQL code I want my system to execute.

**NOTE**

Another complaint I have is that I don’t really like the architecture used for the specific `ClientDataSet` components bound to the various technologies (BDE, ADO, dbExpress and so on). In Chapter 19, “COM Programming,” I’ll discuss an alternate approach. In particular, I dislike the fact that you cannot code for a generic base component, but have to tune your code to the specific version of the `ClientDataSet`. Moreover, this architecture isn’t extensible: you’ll have to write a new specific component for each data access class you want to use (or to write). This seems really contrary to the spirit of OOP and to the overall architecture of VCL. My suggestion is simply to avoid using these all-in-one components and get used to dropping the `DataSet–Provider–ClientDataSet` triad every time you need to (or build a custom compound component for them).

## When One-Way Is Enough: Printing Data

We have seen that one of the key elements of the dbExpress library is that it returns unidirectional datasets and that we can use the `ClientDataSet` component (in one of its incarnations) to store the records in a local cache. Now it is interesting to discuss at least a simple example where a unidirectional dataset is all we need.

This is common in *reporting*, that is, to produce information for each record in sequence without needing any further access to the data. This broad category includes producing printed reports (via a set of reporting components or using the printer directly), sending data to other applications like Microsoft Excel or Word, saving data to files (including HTML and XML formats), and more.

As I don’t want to delve into HTML and XML right now, and we still haven’t discussed COM-based automation, I’ll go ahead with an example of printing—nothing fancy and nothing based on reporting components, but a simple way to produce a draft report on your video

and printer. For this reason, I'm going to use Delphi's simplest technique to produce a print-out: assigning a file to the printer with the `AssignPrn` RTL procedure.

The example, called `UniPrint`, has a unidirectional `SQLDataSet` component, hooked to an `InterBase` connection and based on the following SQL statement, which joins the employee table with the department table to display the name of the department where each employee works:

```
select d.DEPARTMENT, e.FULL_NAME, e.JOB_COUNTRY, e.HIRE_DATE
from EMPLOYEE e
inner join DEPARTMENT d on d.DEPT_NO = e.DEPT_NO
```

To handle printing, I've written a somewhat generic routine, requiring as parameters the data to print, a progress bar for status information, the output font, and the maximum format size of each field. The entire routine, listed below, uses file-print support and the graphic objects recall technique, and formats each field in a fixed-size, left-aligned string, to produce a columnar type of report. The call to the `Format` function has a parametric format string, built dynamically using the size of the field.

Here is the code, which uses three nested `try/finally` blocks to release all the resources properly:

```
procedure PrintOutDataSet (data: TDataSet;
progress: TProgressBar; Font: TFont; maxSize: Integer = 30);
var
  PrintFile: TextFile;
  I: Integer;
  sizeStr: string;
  oldFont: TFontRecall;
begin
  // assign the printer to a file
  AssignPrn (PrintFile);
  Rewrite (PrintFile);

  // set the font and keep the original one
  oldFont := TFontRecall.Create (Printer.Canvas.Font);
  try
    Printer.Canvas.Font := Font;
    try
      data.Open;
      try
        // print header (field names) in bold
        Printer.Canvas.Font.Style := [fsBold];
        for I := 0 to data.FieldCount - 1 do
          begin
            sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
            Write (PrintFile, Format ('%-' + sizeStr + 's',
```

```

        [data.Fields[i].FieldName]));
    end;
    Writeln (PrintFile);

    // for each record of the dataset
    Printer.Canvas.Font.Style := [];
    while not data.EOF do
    begin
        // print out each field of the record
        for I := 0 to data.FieldCount - 1 do
        begin
            sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
            Write (PrintFile, Format ('%-' + sizeStr + 's',
                [data.Fields[i].AsString]));
            end;
            Writeln (PrintFile);
            // advance ProgressBar
            progress.Position := progress.Position + 1;
            data.Next;
        end;
    finally
        // close the dataset
        data.Close;
    end;
finally
    // reassign the original printer font
    oldFont.Free;
end;
finally
    // close the printer/file
    System.CloseFile (PrintFile);
end;
end;

```

The program invokes this routine when the Print All button is clicked. The program executes a separate query, which returns the number of records of the employee table to set up the progress bar (the unidirectional dataset, in fact, has no way to know how many records it is going to retrieve until it has reached the last one). Then it sets the output font, possibly using a fixed-width font, and calls the PrintOutDataSet routine.

```

procedure TNavigator.PrintAllButtonClick(Sender: TObject);
var
    Font: TFont;
begin
    // set ProgressBar range
    Emp1CountData.Open;
    try

```

```
        ProgressBar1.Max := Emp1CountData.Fields[0].AsInteger;
finally
    Emp1CountData.Close;
end;

Font := TFont.Create;
try
    Font.Name := 'Courier New';
    Font.Size := 9;
    PrintOutDataSet (Emp1Data, ProgressBar1, Font);
finally
    Font.Free;
end;
end;
```

## ClientDataSet and MyBase

The general idea of a client/server application implies that the computation workload is shared between two separate programs, the RDBMS and a client application. Although it is very hard to strike a precise line between the two sides, it is certainly useful to do operations on the client. Most database engines (BDE, as we've seen in this chapter, and ADO, as we'll see in Chapter 16, "ActiveX Data Objects") can manipulate client-side data stored in a cache. Using the ClientDataSet component, you can do the same regardless of the database engine you are using, which makes your program more flexible, particularly if you want to use dbExpress, which doesn't provide a similar feature natively.

A practical example will underline what I mean: Suppose you've written a SQL query to retrieve a rather large dataset, and a user wants to see the same data in a different order. You can certainly run a new query, with the proper order by clause, but this implies sending the same (possibly large) dataset once more from the server to the client. Since the client already has the data in memory, it would be more practical and generally faster to re-sort the data in memory and present the same data to the user with a different ordering.

The ClientDataSet component allows you to do this: Attaching the code to sort the data by assigning a proper field name to the IndexFieldNames property. This is often accomplished when the user clicks the field title in a DBGrid component (firing the OnTitleClick event):

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
    ClientDataSet1.IndexFieldNames := Column.Field.FieldName;
end;
```

**TIP**

Unlike local databases, a `ClientDataSet` can have dynamic indexes, as they are computed in memory anyway. The component also supports indexes based on a calculated field, specifically an *internally calculated* field, a type of field available only for this dataset. Unlike ordinary calculated fields, which are computed every time the record is used, values of internally calculated fields are kept in memory. This is why indexes consider them as plain fields.

Indexing is not all the `ClientDataSet` has to offer. When you have an index, you can define groups based on it, possibly with multiple levels of grouping. There is even specific support for determining the position of a record within a group (first, last, or middle position). Over groups or entire tables, you can define aggregates; that is, you can compute the sum or average value of a column for the entire table or the current group on-the-fly. The data doesn't need to be posted to a physical server, because these aggregate operations take place in memory. You can even define new aggregate fields, to which you can directly connect data-aware controls. I'll explore these capabilities in the next section.

Another very interesting area of the `ClientDataSet` component is its ability to handle the updates log, undoing changes, looking at their list before committing them, and so on. I'll explore this next.

The `ClientDataSet` component supports many features, only some of which are related to the three-tier architecture (covered in Chapter 17). This component represents a database completely mapped in memory and can also be made persistent to a local file. Borland marketing has introduced the name `MyBase` to describe this feature of the `ClientDataSet` component, which was formerly called the briefcase model.

The important thing to keep in mind is that all of these features are available to any client/server and even local applications. The `ClientDataSet` component, in fact, can get its data from a remote connection, from a local dataset (as you must do with `dbExpress`), or from a local `MyBase` file. This is another huge area to explore, so I'll simply show you a couple of examples highlighting key features.

**WARNING**

The use of the `ClientDataSet` component, in each of its incarnations, requires either the deployment of the `Midas.dll` library or the inclusion in the project of the `MidasLib` unit (available in compiled format only). The core code of this component, in fact, is not directly part of the library and is not available in source code format. This is unfortunate, as many Delphi developers are accustomed to debugging into the source code and using it as the ultimate reference. It is noteworthy, though, the inclusion in Delphi 6 of the DCU version of the library, obtained from a C-language source code. This allows you to avoid deploying the actual library along with your program.

## The Packets and the Cache

The ClientDataSet component reads data in packets made of the number of records indicated by the PacketRecords property. The default value of this property is -1, which means that the provider will pull all the records at once (this is reasonable only for a small dataset). Alternatively, you can set this value to zero to ask the server for only the field descriptors and no actual data or use any positive value to specify an actual number.

If you retrieve only a partial dataset, as you browse past the end of local cache, if FetchOnDemand property is set to True (the default value), the ClientDataSet component will get more records from its source. This same property also controls whether BLOB fields and nested datasets of the current records are fetched automatically (these values might not be already part of the data packet, depending on the value of the Options of the dataset provider).

If you turn off this property, you'll need to manually fetch more records, by calling the GetNextPacket method, until the method returns zero. (You'll call FetchBlobs and FetchDetails for these other elements.)

### WARNING

Notice, by the way, that before you set a index for the data, you should retrieve the entire dataset (either by going to its last record or by setting the PacketRecords property to -1). Otherwise you'll have an odd index based on partial data.

## Filtering

As with any other dataset, you can use the Filter property to specify the inclusion in the dataset of portions of the data the component is bound to. When manipulating a large table, of course, you should use a proper query so that you don't retrieve a large dataset from a SQL server. Filtering up-front in the server should generally be your first choice.

However, local filtering in the ClientDataSet can be quite useful, particularly because the filter expressions you can use are much more extensive than those you can use with other datasets. In particular, you can use the standard comparison and logical operators (`Population > 1000` and `Area < 1000`) and arithmetic operators (`Population / Area < 10`), but also string functions (`Substring(Last_Name), 1, 2 = 'Ca'`), date and time functions (`Year (Invoice_Date) = 2002`), and others, including a Like function, wildcards, and an In operator.

These filtering capabilities are fully documented in the VCL Help file. Notice that the documentation was already there for Delphi 5, but most of these features didn't actually work. Now they do.

## Grouping and Aggregates

We've already seen that a `ClientDataSet` can have an index different than the order in which it received the data. Once you've defined an index, you can group the data by that index. In practice, a group is defined as a list of consecutive records (according to the index) for which the value of the indexed field doesn't change. For example, if you have an index by state, all the addresses within that state will fall in the group.

### Grouping

The `CdsCalcs` example has a `ClientDataSet` component that extracts its data from the `Country` table of the familiar `DBDEMOS` database. This operation is performed using a `DataSetProvider` component to the form, connecting the three components as follows:

```
object Table1: TTable
  Active = True
  DatabaseName = 'DBDEMOS'
  TableName = 'COUNTRY.DB'
end
object DataSetProvider1: TDataSetProvider
  DataSet = Table1
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'DataSetProvider1'
end
```

I could have also used the specific `BDEClientDataSet` component, as I'll discuss later. Now we can focus on the definition of the group. This is obtained, along with the definition of an index, by specifying a grouping level for the index itself:

```
object ClientDataSet1: TClientDataSet
  IndexDefs = <
    item
      Name = 'ClientDataSet1Index1'
      Fields = 'Continent'
      GroupingLevel = 1
    end>
  IndexName = 'ClientDataSet1Index1'
```

When you have a group active, you can make this obvious to the user by displaying the grouping structure in the `DBGrid`, as shown in Figure 14.11. Simply handle the `OnGetText` event for the grouped field (the `Continent` field in the example), and show the text only if the record is the first of the group:

```
procedure TForm1.ClientDataSet1ContinentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
```



```

if gbFirst in ClientDataSet1.GetGroupState (1) then
    Text := Sender.AsString
else
    Text := '';
end;

```

**FIGURE 14.11:**

The CdsCalcs example demonstrates that by writing a little code, you can have the DBGrid control visually show the grouping defined in the ClientDataSet.

Continent	Name	Capital	Area	Population
Europe	Italy	Rome		
	France	Paris		
North America	Mexico	Mexico City	1967180	88600000
	Nicaragua	Managua	139000	3900000
	El Salvador	San Salvador	20865	5300000
	Cuba	Havana	114524	10600000
	Jamaica	Kingston	11424	2500000
	United States of America	Washington	9363130	249200000
	Canada	Ottawa	9976147	26500000
South America	Paraguay	Asuncion	406576	4660000
	Uruguay	Montevideo	176140	3002000
	Venezuela	Caracas	912047	19700000
	Peru	Lima	1285215	21600000
	Argentina	Buenos Aires	2777815	32300003
	Guyana	Georgetown	214969	800000
	Ecuador	Quito	455502	10600000
	Colombia	Bogotá	1138907	33000000

TotalArea aggregate  Get Aggregates Label1

## Defining Aggregates

Another feature of the ClientDataSet component is support for *aggregates*. An aggregate is a calculated value based on multiple records, such as the sum or the average value of a field for the entire table or a group of records (defined with the grouping logic I've just discussed). Aggregates are *maintained*; that is, they are recalculated immediately if one of the records changes. For example, the total of an invoice can be maintained automatically while the user types in the invoice items.

### NOTE

Aggregates are maintained incrementally, not by recalculating all the values every time one value changes. Aggregate updates take advantage of the deltas tracked by the ClientDataSet. For example, to update a sum when a field is changed, the ClientDataSet subtracts the old value from the aggregate and adds the new value. Only two calculations are needed, even if there are thousands of rows in that aggregate group. For this reason, aggregate updates are instantaneous.

There are two ways to define aggregates. You can use the `Aggregates` property of the `ClientDataSet`, which is a collection, or you can define aggregate fields using the Fields editor. In both cases, you define the aggregate expression, give it a name, and connect it to an index and a grouping level (unless you want to apply it to the entire table). Here is the `Aggregates` collection of the `CdsCalcs` example:

```
object ClientDataSet1: TClientDataSet
  Aggregates = <
    item
      Active = True
      AggregateName = 'Count'
      Expression = 'COUNT (NAME)'
      GroupingLevel = 1
      IndexName = 'ClientDataSet1Index1'
      Visible = False
    end
    item
      Active = True
      AggregateName = 'TotalPopulation'
      Expression = 'SUM (POPULATION)'
      Visible = False
    end>
  AggregatesActive = True
```

Notice in the last line above that you must activate the support for aggregates, in addition to activating each specific aggregate you want to use. Disabling aggregates is important, because having too many of them can slow down a program. The alternative approach, as I mentioned, is to use the Fields editor, select the `New Field` command of its shortcut menu, and choose the `Aggregate` option (available, along with the `InternalCalc` option, only in a `ClientDataSet`). This is the definition of an aggregate field:

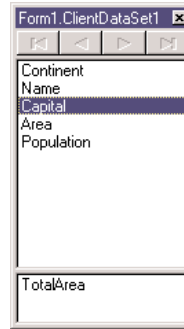
```
object ClientDataSet1: TClientDataSet
  object ClientDataSet1TotalArea: TAggregateField
    FieldName = 'TotalArea'
    ReadOnly = True
    Visible = True
    Active = True
    DisplayFormat = '###,###,###'
    Expression = 'SUM(AREA)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
  end
```

The aggregate fields are displayed in the Fields editor in a separate group, as you can see in Figure 14.12. The advantage of using an aggregate field, compared to a plain aggregate, is that you can define the display format and hook the field directly to a data-aware control,

such as a DBEdit in the CdsCalcs example. Because the aggregate is connected to a group, as soon as you select a record of a different group, the output will be automatically updated. Also, if you change the data, the total will immediately show the new value.

**FIGURE 14.12:**

The bottom portion of the Fields editor of a ClientDataSet displays aggregate fields.



To use plain aggregates, instead, you have to write a little code, as in the following example (notice that the Value of the aggregate is a variant):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption :=
    'Area: ' + ClientDataSet1.TotalArea.DisplayText + #13'Population : '
    + FormatFloat ('###,###,###', ClientDataSet1.Aggregates [1].Value) +
    #13'Number : ' + IntToStr (ClientDataSet1.Aggregates [0].Value);
end;
```

## Manipulating Updates

One of the core ideas behind the ClientDataSet component is that it is used as a local cache to collect some input from a user and then send a batch of update requests to the database. The component has both a list of the changes to apply to the database server, stored in the same format used by the ClientDataSet (accessible through the Delta property), and a complete updates log that you can manipulate with a few methods (including an Undo capability).

## The Status of the Records

The component lets us monitor what's going on within the data packets. The UpdateStatus method returns one of the following indicators for the current record:

```
type TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

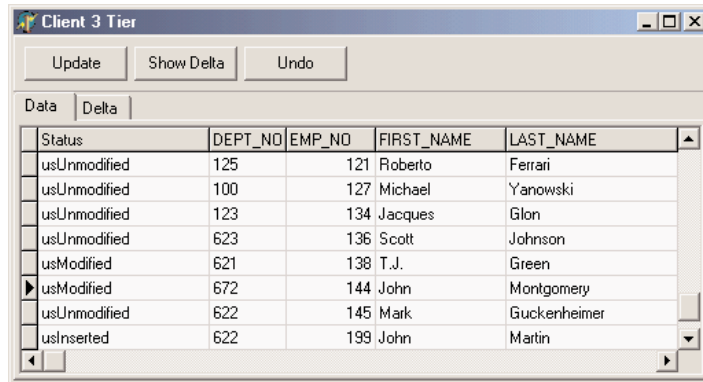
To check the status of every record in the client dataset easily, you can add a string-type calculated field to the dataset (I've called it `ClientDataSet1Status`) and compute its value with the following `OnCalcFields` event handler:

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
    ClientDataSet1Status.AsString := GetEnumName (TypeInfo(TUpdateStatus),
        Integer (ClientDataSet1.UpdateStatus));
end;
```

This method (based on the RTTI `GetEnumName` function) converts the current value of the `TUpdateStatus` enumeration to a string, with the effect you can see in Figure 14.13.

**FIGURE 14.13:**

The `CdsDelta` program displays the status of each record of a `ClientDataSet`.



## Accessing the Delta

Beyond examining the status of each record, the best way to understand which changes have occurred in a given `ClientDataSet` (but haven't been uploaded to the server) is to look at the delta, the list of changes waiting to be applied to the server. This property is defined as follows:

```
property Delta: OleVariant;
```

The format used by the `Delta` property is the same as that used to transmit the data from the client to the server. What we can do, then, is add another `ClientDataSet` component to an application and connect it to the data in the `Delta` property of the first client dataset:

```
if ClientDataSet1.ChangeCount > 0 then
begin
    ClientDataSet2.Data := ClientDataSet1.Delta;
    ClientDataSet2.Open;
```

In the `CdsDelta` example, I've added a data module with the two `ClientDataSet` components and an actual source of data, a `SQLDataSet` mapped to InterBase's `EMPLOYEE` demo

table. Both client datasets have the extra status calculated field, with a slightly more generic version than the code discussed earlier, because the event handler is shared between them.

**TIP**

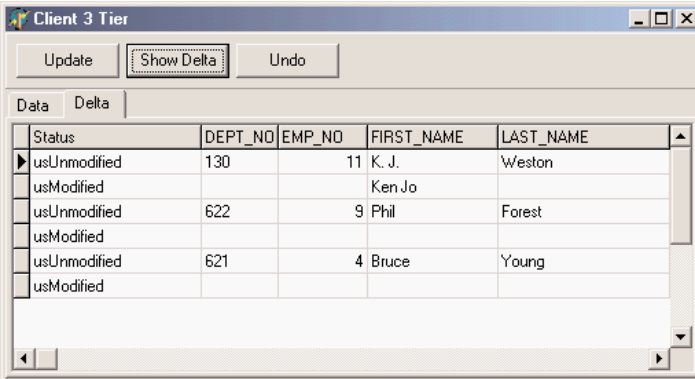
To create persistent fields for the ClientDataSet hooked to the delta (at run time), I've temporarily connected it, at design time, to the same provider of the main ClientDataSet. The structure of the delta, in fact, is the same of the dataset it refers to. After creating the persistent fields, I've removed the connection.

The form of this application has a page control with two pages, each with a DBGrid, one for the actual data and one for the delta. Some code hides or shows the second tab depending on the existence of data in the change log, as returned by the `ChangeCount` method, and updates the delta when the corresponding tab is selected. The core of the code used to handle the delta is very similar to the last code snippet above, and you can study the example source code on the CD to see more details.

You can see the change log of the `CdsDelta` application in Figure 14.14. Notice that the delta dataset has two entries for each modified record: the original values and the modified fields, unless this is a new or deleted record, as indicated by its status.

**FIGURE 14.14:**

The `CdsDelta` example allows you to see the temporary update requests stored in the `Delta` property of the `ClientDataSet`.



Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME
usUnmodified	130	11	K. J.	Weston
usModified			Ken Jo	
usUnmodified	622	9	Phil	Forest
usModified				
usUnmodified	621	4	Bruce	Young
usModified				

**TIP**

You can also filter the delta dataset (or any other `ClientDataSet`) depending on its update status, using the `StatusFilter` property. This allows you to show new, updated, and deleted records in separate grids or in a grid filtered by selecting an option in a `TabControl`.

## Undo and *SavePoint*

Because the update data is stored in the local memory (in the delta), besides applying the updates and sending them to the application server, we can reject them, removing entries

from the delta. The `ClientDataSet` component has a specific `UndoLastChange` method to accomplish this. The parameter of this method allows you to *follow* the undo operation (the name of this parameter is `FollowChange`). This means the client dataset will move to the record that has been restored by the undo operation.

Here is the code connected to the Undo button of the `CdsDelta` example:

```
procedure TForm1.ButtonUndoClick(Sender: TObject);  
begin  
    DmCds.cdsEmployee.UndoLastChange (True);  
end;
```

An extension of the undo support is the possibility to save a sort of bookmark of the change log position (the current status) and to restore it later by undoing all successive changes. The `SavePoint` property can be used either to save the number of changes in the log or to reset the log to a past situation. Notice, anyway, that you can only remove records from the change log, not reinsert changes. In other words, the `ChangeLog` refers to a position in a log, so it can only go back to a position in which there were fewer records! This position is just a number of changes, so if you undo some changes and then do more edits, that number of changes will become meaningless.

## Enabling and Disabling Logging

Keeping track of changes makes sense if you need to send the updated data back to a server database. In local applications with data stored to a `MyBase` file, keeping this log around can become useless and consumes memory. For this reason, you can disable logging altogether with the `LogChanges` property.

You can also call the `MergeChangesLog` method to remove all current editing from the change log. This makes sense if the dataset doesn't directly originate by a provider but was built with custom code, or in case you want to add or edit the data programmatically, without having to send it to the back-end database server.

### TIP

The `ClientDataSet` in Delphi 6 has a new property, `DisableStringTrim`, which allows you to keep trailing spaces in field values. In past versions, in fact, string fields were invariably trimmed, which creates trouble with some databases.

## Updating the Data

Now that we have a better understanding of what goes on during local updates, we can try to make this program work by sending the local update (stored in the delta) back to the database server. To apply all the updates from a dataset at once, pass `-1` to the `ApplyUpdates` method.

If the provider (or actually the Resolver component inside it) has trouble applying an update, it triggers the `OnReconcileError` event. This can take place because of a concurrent update by two different people. As we tend to use optimistic locking in client/server applications, this should be regarded as a normal situation.

The `OnReconcileError` event allows you to modify the `Action` parameter (passed by reference), which determines how the server should behave:

```
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;  
    E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
```

This method has three parameters: the client dataset component (in case more than one client application is interacting with the application server), the exception that caused the error (with the error message), and the kind of operation that failed (`ukModify`, `ukInsert`, or `ukDelete`). The return value, which you'll store in the `Action` parameter, can be any one of the following:

```
type TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,  
    raRefresh);
```

- The `raSkip` value specifies that the server should skip the conflicting record, leaving it in the delta (this is the default value).
- The `raAbort` value tells the server to abort the entire update operation and not even try to apply the remaining changes listed in the delta.
- The `raMerge` value tells the server to merge the data of the client with the data on the server, applying only the modified fields of this client (and keeping the other fields modified by other clients).
- The `raCorrect` value tells the server to replace its data with the current client data, overriding all field changes already done by other clients.
- The `raCancel` value cancels the update request, removing the entry from the delta and restoring the values originally fetched from the database (thus ignoring changes done by other clients).
- The `raRefresh` value tells the server to dump the updates in the client delta and to replace them with the values currently on the server (thus keeping the changes done by other clients).

If you want to test a collision, you can simply launch two copies of the client application, change the same record in both clients, and then post the updates from both. We'll do this later to generate an error, but let's first see how to handle the `OnReconcileError` event.

This is actually a simple thing to accomplish, but only because we'll receive a little help. Since building a specific form to handle an `OnReconcileError` event is very common, Delphi

already provides such a form in the Object Repository. Simply go to the Dialogs page and select the Reconcile Error Dialog item. This unit exports a function you can directly use to initialize and display the dialog box, as I've done in the CdsDelta example:

```
procedure TdmCds.cdsEmployeeReconcileError (DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

### WARNING

As the source code of the Reconcile Error Dialog unit suggests, you should use the Project Options dialog to remove this form from the list of automatically created forms (if you don't, an error will occur when you compile the project). Of course, you need to do this only if you haven't set up Delphi to skip the automatic form creation.

The `HandleReconcileError` function simply creates the form of the dialog box and shows it, as you can see in the code provided by Borland:

```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
  ReconcileError: EReconcileError): TReconcileAction;
var
  UpdateForm: TReconcileErrorForm;
begin
  UpdateForm := TReconcileErrorForm.CreateForm(DataSet, UpdateKind,
    ReconcileError);
  with UpdateForm do
  try
    if ShowModal = mrOK then
    begin
      Result := TReconcileAction(ActionGroup.Items.Objects[
        ActionGroup.ItemIndex]);
      if Result = raCorrect then
        SetFieldValues(DataSet);
    end
  else
    Result := raAbort;
  finally
    Free;
  end;
end;
```

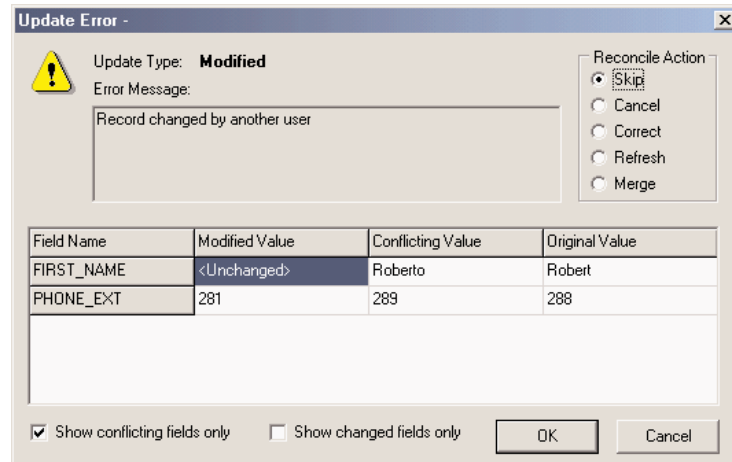
The `Reconc` unit, which hosts the Reconcile Error dialog, contains over 350 lines of code, so we can't describe it in detail. However, you should be able to understand the source code by studying it carefully. Alternatively, you can simply use it without caring about how everything works.



The dialog box will appear in case of an error, reporting the requested change that caused the conflict and allowing the user to choose one of the possible `TReconcileAction` values. You can see an example in Figure 14.15.

**FIGURE 14.15:**

The Reconcile Error dialog provided by Delphi in the Object Repository and used by the `CdsDelta` example



### TIP

When you call `ApplyUpdates`, you start a rather complex update sequence, discussed in more detail in Chapter 17 for multitier architectures. In short, the delta is sent to the provider, which fires the `OnUpdateData` event and then receives a `BeforeUpdateRecord` event for every record to update. These are two chances you have to take a look at the changes and force specific operations on the database server.

## MyBase (or the Briefcase Model)

The last capability of the `ClientDataSet` component I want to discuss in this chapter is its support for mapping memory data to local files, building stand-alone applications. The same technique can be applied in multitier applications to use the client program even when you're not physically connected to the application server. In this case, you can save all the data you expect to need in a local file for travel with a laptop (perhaps visiting client sites). You'll use the client program to access the local version of the data, edit the data normally, and when you reconnect, apply all the updates you've performed while disconnected.

To map a `ClientDataSet` to a local file you only need to set its `FileName` property, which requires an absolute pathname. To build a minimal `MyBase` program (called `MyBase1`), all

you need is a ClientDataSet component hooked to a file and with a few fields defined (in the FieldDefs property):

```

object ClientDataSet1: TClientDataSet
  FileName = 'C:\md6code\14\MyBase1\test'
  FieldDefs = <
    item
      Name = 'one'
      DataType = ftString
      Size = 20
    end
    item
      Name = 'two'
      DataType = ftSmallint
    end>
  StoreDefs = True
end

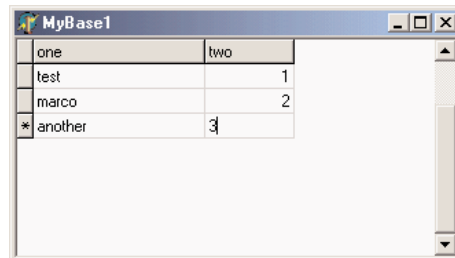
```

At this point you can use the Create DataSet command of the local menu of the ClientDataSet at design time, or call its CreateDataSet method at run time, to physically create the file for the table. As you make changes and close the application, the data will be automatically saved to the file. (You might want to disable the change log, though, to reduce the size of this data.) The dataset, in any case, also has a SaveToFile method and a LoadFromFile method you can use in your code.

MyBase1, my example program, shown in Figure 14.16, doesn't require any database server or database connection to work. It needs only your own program and the Midas.dll file, but you can even get rid of it by including the MidasLib unit in the project. And the program doesn't require any actual Pascal code, either.

**FIGURE 14.16:**

The MyBase1 example, which saves data directly to a MyBase file



one	two
test	1
marco	2
*another	3

### TIP

MyBase generally saves the datasets in XML format, although the internal CDS format is still available. I'll explore this format in detail when I discuss XML in Chapter 23, "XML and SOAP." For the moment, suffice to say this is a text-based format (so it is less space-efficient than the internal format), which can be manipulated programmatically but immediately makes some sense even if you try reading it.

The MyBase support in Delphi 6 also includes the possibility of extracting the XML representation of a memory dataset by using the `XMLData` property. In Delphi 5, you could obtain the same by saving the `ClientDataSet` in XML format in a memory stream.

## Abstract Data Types in MyBase

The `ClientDataSet` component supports most data types provided by Delphi, including nested data types and abstract data types, the case I want to investigate with this second MyBase example. In the `FieldDefs` property editor of a `ClientDataSet` component you can and select the `ftADT` value for the `DataType` property of one of fields. Now move to the `ChildDefs` property and define the child fields. This is the field definition of the `AdtDemo` example:

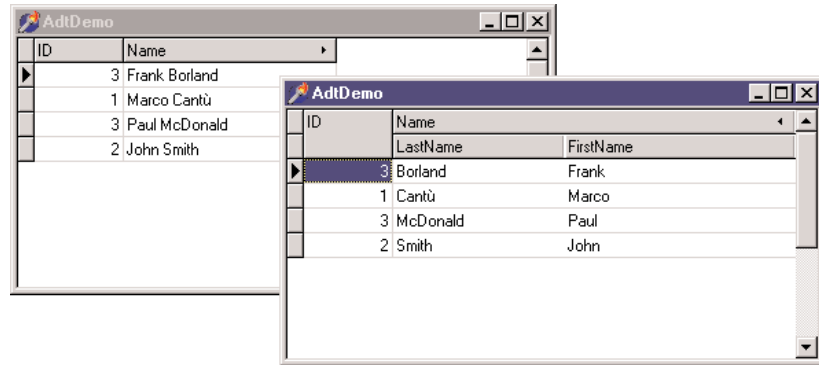
```
FieldDefs = <
  item
    Name = 'ID'
    DataType = ftInteger
  end
  item
    Name = 'Name'
    ChildDefs = <
      item
        Name = 'LastName'
        DataType = ftString
        Size = 20
      end
      item
        Name = 'FirstName'
        DataType = ftString
        Size = 20
      end
    end>
    DataType = ftADT
    Size = 2
  end>
```

At this point, provide the `FileName`, create the dataset, and you are ready to compile and run the application. If you use a `DBGrid` to view the resulting dataset, it will allow you to expand or collapse the subfields of the ADT field, as you can see in Figure 14.17. The *condensed* value of the field is defined in the `AdtDemo` program by handling the `OnGetText` event of the ADT field:

```
procedure TForm1.ClientDataSet1NameGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := ClientDataSet1NameFirstName.AsString + ' ' +
    ClientDataSet1NameLastName.AsString;
end;
```

**FIGURE 14.17:**

The AdtDemo example shows the support for expanding or collapsing the definition of an ADT field.



## Indexing for ADT Fields

We've seen how easily you can set up an index as the user selects the title of a DBGrid. In ADT fields, the situation becomes a little more complex. The AdtDemo program, in fact, uses the `FullName` property of the field (not the `FieldName` property) because of the ADT definition. For the `LastName` child field, in fact, the index should be based on `Name.LastName`, not simply on `LastName`. Also, the ADT field cannot itself be indexed, so if it is selected, the program uses as index the `LastName` subfield. Here is the code:

```

procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
    if Column.Field.FullName = 'Name' then
        ClientDataSet1.IndexFieldNames := 'Name.LastName'
    else
        ClientDataSet1.IndexFieldNames := Column.Field.FullName;
end;

```

## What's Next?

This chapter has presented a somewhat detailed introduction to client/server programming with Delphi. We saw what the key issues are and delved a little into some interesting areas of client/server programming. After a general introduction, I discussed the use of the dbExpress database library Borland is introducing in Delphi 6 and of the `ClientDataSet` component and MyBase technology.

There is certainly more we can say about client/server programming in Delphi, and in the next chapter I'll discuss some real-world examples, after introducing InterBase and the IBX components. Chapter 16 will then focus on Microsoft's ADO database engine.

# InterBase and IBX

- Getting started with InterBase 6
- Server-side programming: views, stored procedures, and triggers
- Using InterBase Express
- Pieces for a real-world example

**C**lient/server programming requires two sides: a client application that you probably want to build with Delphi, and a relational database management system (RDBMS), usually a “SQL server.” In this chapter, I focus on one specific SQL server, InterBase. There are many reasons for this choice. InterBase is the SQL server developed by Borland; it is an open source project and can be obtained for free; and it has traditionally been bound with Delphi, which has specific dataset components for it.

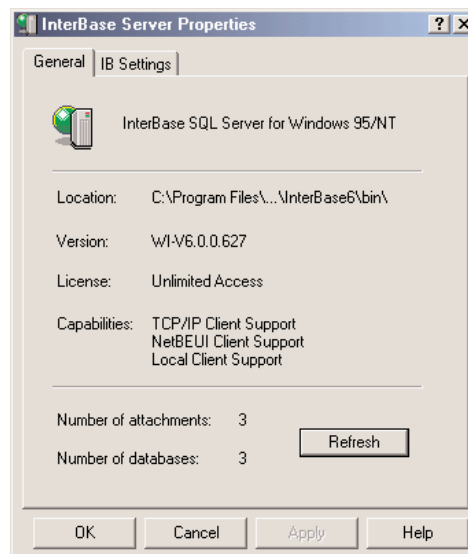
For all of these reasons, InterBase should be a good choice for your Delphi client/server development, although there are many other equally powerful alternatives. I’ll discuss InterBase from the Delphi perspective, without delving in to its internal architecture. A lot of the information presented also applies to other SQL servers, so even if you’ve decided not to use InterBase, you might still find it valuable.

## Getting Started with InterBase 6

After installing InterBase 6, you’ll be able to activate the server from the Windows Start menu, but if you plan on using it frequently, you should install it as a Windows service (of course, only if you have Windows NT/2000, as Windows 9x/Me doesn’t have support for services). When the server is active, you’ll see a corresponding icon in the Tray Icon area of the Windows Taskbar (unless you start it as a service). The menu connected with this icon allows you to see status information (see Figure 15.1) and do some very limited configuration.

**FIGURE 15.1:**

The status information displayed by InterBase when you double-click its tray icon



## Inside InterBase

Even though it has a limited market share, InterBase is a very powerful RDBMS. In this section I'll introduce the key technical features of InterBase, without getting into too much detail. This is a book on Delphi programming, in fact. Unfortunately, there is currently very little published about InterBase, although there are some ongoing efforts for an InterBase book and there is a wealth of information in the documentation accompanying the product and on a few Web sites devoted to the product.

InterBase was built from the beginning with a very modern and robust architecture. Its original author, Jim Starkey, invented an architecture for handling concurrency and transactions without imposing physical locks on portions of the tables, something other well-known database servers can hardly do even today. InterBase architecture is called Multi-Generational Architecture (MGA), and it handles concurrent access to the same data by multiple users, who can modify records without affecting what other concurrent users see in the database.

This approach naturally maps to the *Repeatable Read* transaction isolation mode, in which a user within a transaction keeps seeing the same data, regardless of changes done and committed by other users. Technically, the server handles this by maintaining a different version of each accessed record for each open transaction. Even if this approach (also called *versioning*) can lead to larger memory consumption, it avoids almost any physical lock on the tables and makes the system much more robust in case of a crash. Also, MGA pushes toward a very clear programming model—Repeatable Read—which other well-known SQL servers don't even support without losing most of their performance.

If Multi-Generational Architecture is at the heart of InterBase, the server has many other technical advantages:

- A limited footprint, which makes InterBase the ideal candidate for running directly on client computers, including portables. The disk space required by InterBase for a minimal installation is well below 10 MB, and its memory requirements are also incredibly limited.
- Good performance on large amounts of data.
- Availability on many different platforms (including 32-bit Windows, Solaris, and Linux), with totally compatible versions, which makes the server scalable from very small to huge systems without notable differences.
- A very good track record, as InterBase has been in use for 15 years with very few problems.
- A language very close to the SQL standard.

- Advanced programming capabilities, with positional triggers, selectable stored procedures, updateable views, exceptions, events, generators, and more.
- Simple installation and management, with limited administration headaches.

## A Short History of InterBase

Jim Starkey wrote InterBase for his own Groton Database Systems company (hence the `.gds` extension still in use for InterBase files). The company was later bought by Ashton-Tate, which was then acquired by Borland. Borland handled InterBase directly for a while, then created an InterBase subsidiary, which was later re-absorbed into the parent company.

Starting with Delphi 1, an evaluation copy of InterBase has been distributed along with the development tool, spreading the database server among developers. Although it doesn't have a large piece of the RDBMS market, which is dominated by a handful of players, InterBase has been chosen by a few very relevant organizations, from Ericsson to the U.S. Department of Defense, from stock exchanges to home banking systems.

More recent events include the announcement of InterBase 6 as an open source database (December 1999), the effective release of source code to the community (July 2000), and the release of the officially certified version of InterBase 6 by Borland (March 2001).

In between these events, there were announcements of the spin-off of a separate company to run the consulting and support business on top of the open source database. Contacts with a group of former InterBase developers and managers (who had left Borland) didn't lead to an agreement, but the group decided to go ahead even without Borland's help and formed IBPhoenix ([www.ibphoenix.com](http://www.ibphoenix.com)) with the plan of supporting InterBase users.

At the same time, independent groups of InterBase experts formed the InterBase Developer Initiative (IBDI; [www.interbase2000.org](http://www.interbase2000.org)) and started the Firebird open source project to further extend InterBase. For this reason, SourceForge currently hosts two different versions of the project, InterBase itself run by Borland and the Firebird project run by this independent group. You see that the picture is rather complex, but this certainly isn't a problem for InterBase, as there are currently many organizations pushing it, along with Borland.

## IBConsole

In past versions of InterBase, there were two main tools you could use to interact directly with the program: the Server Manager application, which could be used to administer both a local and a remote server; and Windows Interactive SQL (WISQL). Version 6 includes a much more powerful front-end application, called IBConsole. This is a full-fledged Windows program (built with Delphi) that allows you to administer, configure, test, and query an InterBase server, whether local or remote.

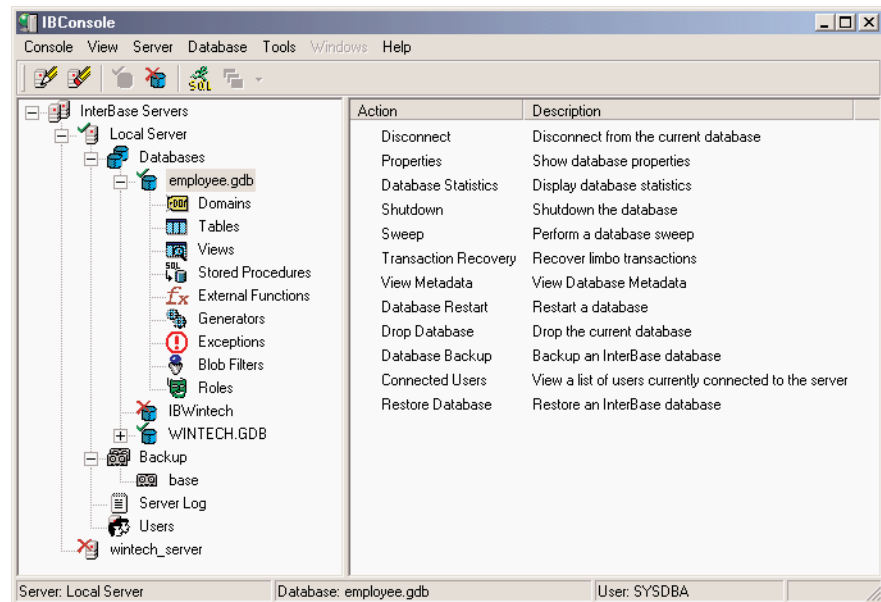


IBConsole is a simple and complete system for managing InterBase servers and their databases. You can use it to look into the details of the database structure, modify it, query the data (which can be useful to develop the queries you want to embed in your program), back up and restore the database, and perform all the other administrative tasks.

As you can see in Figure 15.2, IBConsole allows you to manage multiple servers and databases, all listed in a single, handy configuration tree. You can ask for general information about the database and list its entities (tables, domains, stored procedures, triggers, and everything else), accessing the details of each. You can also create new databases and configure them, back up the files, update the definitions, check what's going on and who is currently connected, and so on.

**FIGURE 15.2:**

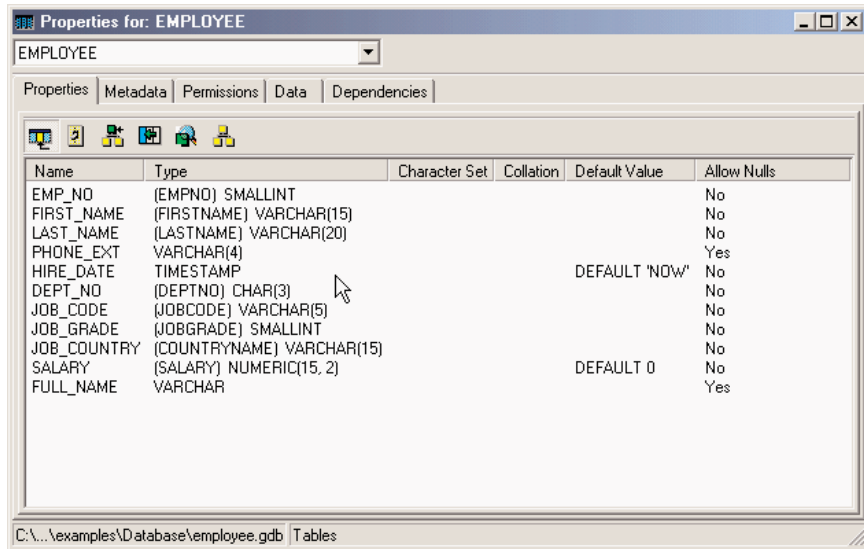
IBConsole allows you to manage, from a single computer, InterBase databases hosted by multiple servers.



The IBConsole application allows you to open multiple windows to look at detailed information, such as the tables window depicted in Figure 15.3. In this window, you can see lists of the key properties of each table (columns, triggers, constraints, and indexes), see the raw metadata (the SQL definition of the table), access permissions, have a look at the actual data, modify it, and study the dependencies of the table. Similar windows are available for each of the other entities you can define in a database.

**FIGURE 15.3:**

IBConsole can open separate windows to show you the details of each entity—in this case, a table.

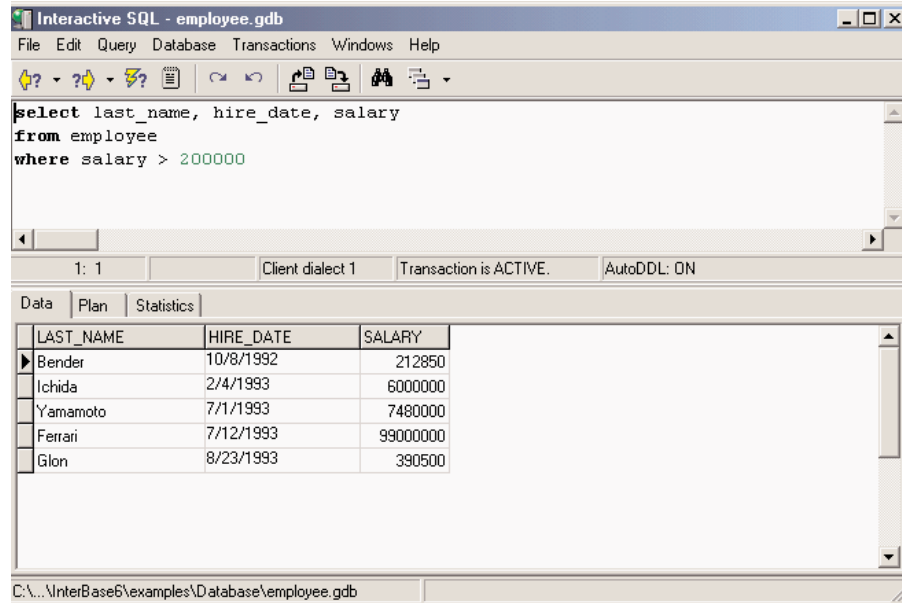


Finally, IBConsole embeds an improved version of the original Windows Interactive SQL application (see Figure 15.4). You can directly type a SQL statement in the upper portion of the window (without any actual help from the tool, unfortunately) and then execute the SQL query. As a result, you'll see the data, but also the access *plan* used by the database (which an expert can use to determine the efficiency of the query) and some statistics on the actual operation performed by the server.

This is really a minimal description of IBConsole, which is a rather powerful tool and the only one included by Borland with the server besides command-line tools. IBConsole is probably not the most complete tool in its category, though. Quite a few third-party InterBase management applications are more powerful, although they are not all very stable or user-friendly. Some InterBase tools are shareware programs, while others are totally free. Two examples, out of many, are InterBase Workbench ([www.interbaseworkbench.com](http://www.interbaseworkbench.com)) and IB\_WISQL (done with and part of InterBase Objects, [www.ibobjects.com](http://www.ibobjects.com)).

**FIGURE 15.4:**

The Interactive SQL window of IBConsole allows you to try out in advance the queries you plan to include in your Delphi programs.

**Tip**

To find the latest third-party InterBase tools, have a look at [www.interbase2000.org/tools](http://www.interbase2000.org/tools), which hosts an up-to-date list.

## Server-Side Programming

At the beginning of the previous chapter, I underlined the fact that one of the objectives of client/server programming—and one of its problems—is the division of the workload between the computers involved. When you activate SQL statements from the client, the burden falls on the server to do most of the work. However, you should try to use `select` statements that return a large result set, to avoid jamming the network.

Besides accepting DDL and DML requests, most RDBMS servers allow you to create routines directly on the server using the standard SQL commands plus their own server-specific extensions (which are generally not portable). These routines typically come in two forms, stored procedures and triggers.

## Stored Procedures

Stored procedures are like the global functions of a Delphi unit and must be explicitly called by the client side. Stored procedures are generally used to define routines for data maintenance, to group sequences of operations you need in different circumstances, or to hold complex `select` statements.

Like Pascal procedures, stored procedures can have one or more typed parameters. Unlike Pascal procedures, they can have more than one return value. As an alternative to returning a value, a stored procedure can also return a result set, the result of an internal `select` statement or a custom fabricated one.

The following is a stored procedure written for InterBase; it receives a date in input and computes the highest salary among the employees hired on that date:

```
create procedure maxsaloftheday(ofday date)
returns (maxsal decimal(8,2)) as
begin
  select max(salary)
  from employee
  where hiredate = :ofday
  into :maxsal;
end
```

Notice the use of the `into` clause, which tells the server to store the result of the `select` statement in the `maxsal` return value. To modify or delete a stored procedure, you can later use the `alter procedure` and `drop procedure` commands.

Looking at this stored procedure, you might wonder what its advantage is compared to the execution of a similar query activated from the client. The difference between the two approaches is not in the result you obtain but in its speed. A stored procedure is compiled on the server in an intermediate and faster notation when it is created, and the server determines at that time the strategy it will use to access the data. By contrast, a query is compiled every time the request is sent to the server. For this reason, a stored procedure can replace a very complex query, provided it doesn't change too often!

From Delphi you can activate a stored procedure returning a result set by using either a Query or a StoredProc component. With a Query, you can use the following SQL code:

```
select *
from MaxSalOfTheDay ('01/01/1990')
```

## Triggers (and Generators)

Triggers behave more or less like Delphi events and are automatically activated when a given *event* occurs. Triggers can have specific code or call stored procedures; in both cases, the execution is done completely on the server. Triggers are used to keep data consistent, checking new data in more complex ways than a check constraint allows, and to automate the side effects of some input operations (such as creating a log of previous salary changes when the current salary is modified).

Triggers can be fired by the three basic data update operations: `insert`, `update`, and `delete`. When you create a trigger, you indicate whether it should fire before or after one of these three actions.

As an example of a trigger, we can use a generator to create a unique index in a table. Many tables use a unique index as primary key. InterBase doesn't have an `AutoInc` field, unlike Paradox and other local databases. Because multiple clients cannot generate unique identifiers, we can rely on the server to do this. Almost all SQL servers offer a counter you can call to ask for a new ID, which you should later use for the table. InterBase calls these automatic counters *generators*, while Oracle calls them *sequences*. Here is the sample InterBase code:

```
create generator cust_no_gen;  
...  
gen_id (cust_no_gen, 1);
```

The `gen_id` function then extracts the new unique value of the generator passed as first parameter, with the second parameter indicating how much to increase (in this case, by one).

At this point you can add a trigger to a table, an automatic handler for one of the table's events. A trigger is similar to the event handler of the `Table` component, but you write it in SQL and execute it on the server, not on the client. Here is an example:

```
create trigger set_cust_no for customers  
before insert position 0 as  
begin  
    new.cust_no = gen_id (cust_no_gen, 1);  
end
```

This trigger is defined for the `Customer` table and is activated each time a new record is inserted. The `new` symbol indicates the new record we are inserting. The `position` option indicates the order of execution of multiple triggers connected to the same event. Triggers with the lowest values will be executed first.

Inside a trigger, you can write DML statements that also update other tables, but watch out for updates that end up reactivating the trigger, creating an endless recursion. You can later modify or disable a trigger by calling the `alter trigger` statement or `drop trigger`.

**TIP**

Triggers fire automatically for specified events. If you have to make many changes in the database using batch operations, the presence of a trigger might slow down the process. If the input data has already been checked for consistency, you can temporarily deactivate the trigger. These batch operations are often coded in stored procedures, but stored procedures generally cannot issue DDL statements, like those required for deactivating and reactivating the trigger. In this situation, you can define a view based on a simple `select * from table` command, thus creating an alias for the table. Then you can let the stored procedure do the batch processing on the table and apply the trigger to the view (which should also be used by the client program).

## Using InterBase Express

The examples built in the last chapter either still used the BDE or were done with the new dbExpress database engine. Using this server-independent engine could allow you to switch the database server used by your application, although in practice this is often far from simple. You might decide that an application you are building will invariably use a given database server, possibly the internal server of the company you are working for. In this case, you can decide to skip any database engine or library as well and write programs that are tied directly to the API of the specific database server, which will make your program intrinsically non-portable to other SQL servers.

Of course, you won't generally use similar APIs directly, but rather base your development on some native or third-party dataset components, which wrap these APIs and naturally fit into Delphi and the architecture of its class library. An example of such a family of components is InterBase Express (IBX). Applications built using these components should work better and faster (even if only marginally), giving you more control over the specific features of the server. For example, IBX provides you a set of administrative components specifically built for InterBase 6.

**NOTE**

I'll examine the IBX components because they are tied to InterBase (the database server discussed in this chapter) and because that set is the only one available in the standard Delphi installation. Other similar sets of components (for InterBase, Oracle, and other database servers) are equally powerful and well-regarded in the Delphi programmers' community. A good example (and an alternative to IBX) is InterBase Objects, [www.ibobjects.com](http://www.ibobjects.com).

## IBX Dataset Components

The IBX components include custom dataset components and a few others. The dataset components inherit from the base `TDataSet` class, can use all the common Delphi data-aware

controls, provide a field editor and all the usual design-time features, and can be used in the Data Module Designer, but they don't require the BDE.

You can actually choose among multiple dataset components. Three datasets of IBX have a role and a set of properties similar to their BDE counterparts:

- `IBTable` resembles the `Table` component and allows you to access a single table or view.
- `IBQuery` resembles the `Query` component and allows you to execute a SQL query, returning a result set. The `IBQuery` component can be used together with the `IBUpdateSQL` component to obtain a live (or editable) dataset.
- `IBStoredProc` resembles the `StoredProc` component and allows you to execute a stored procedure.

For new applications, you should generally use the `IBDataSet` component, which allows you to work with a live result set obtained by executing a `select` query. It basically merges `IBQuery` with `IBUpdateSQL` in a single component. The three components above, in fact, are provided mainly for compatibility with Delphi BDE applications.

Many other components in InterBase Express don't belong to the dataset category, but are still used in applications that need to access to a database:

- `IBDatabase` mimics the BDE `Database` component and is used to set up the database connection. The BDE also uses the specific `Session` component to perform some global tasks done by the `IBDatabase` component.
- `IBTransaction` allows complete control over transactions. It is important in InterBase to use transactions explicitly and isolate each transaction properly, using the `Snapshot` isolation level for reports and the `Read Committed` level for interactive forms. Each dataset explicitly refers to a given transaction, so you can have multiple concurrent transactions against the same database, choosing which datasets take part in which transaction.
- `IBSQL` lets you execute SQL statements that don't return a dataset (for example, DDL requests, or update and delete statements) without the overhead of a dataset component.
- `IBDatabaseInfo` is used for querying the database structure and status.
- `IBSQLMonitor` is used for debugging the system, since the `SQL Monitor` debugger provided by Delphi is a BDE-specific tool.
- `IBEvents` receives events posted by the server.

This group of components provides greater control over the database server than you can have with the BDE. For example, having a specific transaction component allows you to manage multiple concurrent transactions over one or multiple databases, as well as a single

transaction spanning multiple databases. The `IBDatabase` component allows you to create databases, test the connection, and generally access system data, something the `Database` and `Session BDE` components don't fully provide.

**TIP**

A feature of the IBX datasets that is new in Delphi 6 is the ability to set up the automatic behavior of a generator as a sort of auto-incremental field. This is accomplished by setting the `GeneratorField` property using its specific property editor. An example of this is discussed later in this chapter in the section "Generators and IDs."

## IBX Administrative Components

A new page of Delphi 6 Component palette, `InterBase Admin`, hosts InterBase 6 administrative components. Although your aim is probably not to build a full InterBase console application, including some administrative features (such as backup handling or user monitoring) can make sense in applications meant for power users.

Most of these components have self-explanatory names. They are `IBConfigService`, `IBBackupService`, `IBRestoreService`, `IBValidationService`, `IBStatisticalService`, `IBLogService`, `IBSecurityService`, `IBServerProperties`, `IBInstall`, and `IBUninstall`. I won't build any advanced examples of the use of these components, as they are more focused towards the development of server management applications than that of client programs. I'll only embed a couple of them in a simple example, later in this chapter.

## From BDE to IBX

To demonstrate how simple it can be to move from the use of the BDE to the use of IBX, I've built a trivial application, using the `Database Form Wizard` (which is strictly bound to the BDE). The application, on the companion CD, is called `IbEmp` and shows only a few fields of the *usual* `EMPLOYEE` table of the corresponding InterBase demo database.

All of the features of the `IbEmp` example are summarized by the properties of its `Query` component:

```
object Query1: TQuery
  DatabaseName = 'IBLocal'
  RequestLive = True
  SQL.Strings = (
    'SELECT * '
    'FROM EMPLOYEE')
end
```

I could have extended the structure of this example generated by Delphi, adding a `DataBase` component to handle the connection, but I decided this was useless, as my intention was only to port the example to the use of the IBX components. The interesting example, in fact,



is IbEmp2, which I started by copying all of the source code files of the version generated by the wizard. (The previous example is available on the companion CD just so you can try this type of porting, as the example by itself is not particularly interesting.)

After replacing the Query component with an IBQuery, I had to add two more components: IBTransaction and IBDatabase. Any IBX application requires at least an instance of each of these two components. You cannot set database connections in a dataset (as you can do with a plain Query), and at least a transaction object is required even to read the result of a query.

Here are the key properties of these components in the IbEmp2 example:

```
object IBTransaction1: TIBTransaction
  Active = False
  DefaultDatabase = IBDatabase1
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  SQL.Strings = (
    'SELECT * FROM EMPLOYEE')
end
object IBDatabase1: TIBDatabase
  DatabaseName = 'C:\Program Files\InterBase ' +
    'Corp\InterBase6\examples\Database\employee.gdb'
  Params.Strings = (
    'user_name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  IdleTimer = 0
  SQLDialect = 1
  TraceFlags = []
end
```

The changes don't take too much time to perform, and if you are accessing the same database table as in the BDE-based program, you won't need to change the data-aware components at all, but only hook the DataSource component to IBQuery1. Because I'm not using the BDE, I had to type in the pathname of the InterBase database. However, not everyone in the world has the Program Files folder, which depends on the local version of Windows, and of course the InterBase sample data files could have been installed in any other location of the disk. We'll try to solve these problems in the next example.

**WARNING** Notice that I've embedded the password in the code, a very naïve approach to security. Not only can anyone run the program, but someone could even extract the password by looking at the hexadecimal code of the executable file. I used this approach so I wouldn't need to keep typing in my password while testing a program, but in a real application you should require your users to do so if they care about the security of their data.

## Building a Live Query

The `IbEmp2` example has a query that doesn't allow editing. To activate editing, you need to use an `IBTable` component or add to the query an `IBUpdateSQL` component, even if the query is very simple. Usually the BDE does the behind-the-scenes work that lets you edit the result set of a simple query, but we are not using the BDE now.

The relationship between the `IBQuery` and `IBUpdateSQL` components is the same as between the `Query` and `UpdateSQL` components. To highlight this, I've taken the main form of the `UpdateSql` example discussed in the last chapter and ported it to the InterBase Express components, building the `UpdSql2` example. I've simply copied the two components from the original example, pasted them into an editor, changed the type of the object, and copied the resulting text into a new form. The properties are so similar that I had only to ignore a couple of missing ones (the `DatabaseName` and the `UpdateMode` properties).

At this point, I simply added an `IBDatabase` and an `IBTransaction` component, a data source and a grid, and my program was up and running. The key element of these components, in fact, is their SQL code, which is attached to the `SQL` property of the query and the `ModifySQL`, `DeleteSQL`, and `InsertSQL` properties of the update component.

However, this time I've made the reference to the database a little more flexible. Instead of typing in the database name at design time, I've extracted the InterBase folder from the Windows Registry (where Borland saves it while installing the programs). This is the code executed when the program starts:

```
uses
    Registry;

procedure TForm1.FormCreate(Sender: TObject);
var
    Reg: TRegistry;
begin
    Reg := TRegistry.Create;
    try
        Reg.RootKey := HKEY_LOCAL_MACHINE;
        Reg.OpenKey('\Software\Borland\InterBase\CurrentVersion', False);
        IBDatabase1.DatabaseName := Reg.ReadString('RootDirectory') +
```

```
        'examples\database\employee.gdb';  
    finally  
        Reg.CloseKey;  
        Reg.Free;  
    end;  
    EmpDS.DataSet.Open;  
end;
```

The source code actually contains alternate code for using the database installed in the Data subfolder of the Borland Shared folder, used for Delphi sample databases. Notice also that InterBase 6 places the sample databases in a different subfolder than InterBase 5 did.

**NOTE**

For more information about the Windows Registry and INI files, see the related sidebar in Chapter 10, “The Architecture of Delphi Applications.”

The new feature of this example, compared to the last version, is the presence of a transaction component. As I’ve already said, the InterBase Express components make the use of a transaction component compulsory, explicitly following a requirement of InterBase. Simply adding a couple of buttons to the form to commit or roll back the transaction would be enough, because a transaction starts automatically as you edit any dataset attached to it.

I’ve also improved the program a little by adding an ActionList component to it. This includes all the standard database actions and adds two custom actions for transaction support, Commit and Rollback. Both actions are enabled when the transaction is active:

```
procedure TForm1.ActionUpdateTransactions(Sender: TObject);  
begin  
    acCommit.Enabled := IBTransaction1.InTransaction;  
    acRollback.Enabled := acCommit.Enabled;  
end;
```

When executed, they perform the main operation but also need to reopen the dataset in a new transaction (which can also be done by “retaining” the transaction context). Actually, CommitRetaining doesn’t reopen a new transaction, but it allows the current transaction to remain open. This way, you can keep using your datasets, which won’t be refreshed (so you won’t see edits already committed by other users) but will keep showing the data you’ve modified. This is the code:

```
procedure TForm1.acCommitExecute(Sender: TObject);  
begin  
    IBTransaction1.CommitRetaining;  
end;  
  
procedure TForm1.acRollbackExecute(Sender: TObject);  
begin  
    IBTransaction1.Rollback;
```

```

// reopen the dataset in a new transaction
IBTransaction1.StartTransaction;
EmpDS.DataSet.Open;
end;

```

**WARNING**

Be aware that InterBase closes any opened cursors when a transaction ends, which means you have to reopen them and refetch the data even if you haven't made any changes. When committing data, instead, you can ask InterBase to retain the "transaction context"—not to close open datasets—by issuing a `CommitRetaining` command, as mentioned before. The reason for this behavior of InterBase depends on the fact that a transaction corresponds to a snapshot of the data. Once a transaction is finished, you are supposed to read the data again to refetch records that may have been modified by other users. Version 6.0 of InterBase includes also a `RollbackRetaining` command, which I've decided not to use, because in a rollback operation, the program should refresh the dataset data to show the original values on screen, not the updates you've discarded.

The last operation refers to a generic dataset and not a specific one because I'm going to add a second alternate dataset to the program. The actions are connected to a text-only toolbar, as you can see in Figure 15.5. The program opens the dataset at startup and automatically closes the current transaction on exit, after asking the user what to do, with the following `OnClose` event handler:

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
    nCode: Word;
begin
    if IBTransaction1.InTransaction then
        begin
            nCode := MessageDlg ('Commit Transaction? (No to rollback)',
                mtConfirmation, mbYesNoCancel, 0);
            case nCode of
                mrYes: IBTransaction1.Commit;
                mrNo: IBTransaction1.Rollback;
                mrCancel: Action := caNone; // don't close
            end;
        end;
    end;
end;

```

An alternative to using the `IBQuery` and `IBUpdateSQL` components is to use the `IBDataSet` component, which combines the two. An InterBase dataset, in fact, is a live query with a complete set of SQL statements for all the main operations. The differences between using the

two components and the single one are minimal. Using IBQuery and IBUpdateSQL is probably better when porting an existing application based on the two equivalent BDE components, even if porting the program directly to the IBDataSet component doesn't really require a lot of extra work.

**FIGURE 15.5:**

The output of the UpdSql2 example

EMP_NO	FIRST_NAME	LAST_NAME	DEPARTMENT	JOB_TITLE	SALARY
65	Sue Anne	O'Brien	Consumer Electronics Div.	Administrative Assistant	31275
107	Kevin	Cook	Consumer Electronics Div.	Director	111262.5
12	Terri	Lee	Corporate Headquarters	Administrative Assistant	53793
105	Oliver H.	Bender	Corporate Headquarters	Chief Executive Officer	212850
94	Randy	Williams	Customer Services	Manager	56295
144	John	Montgomery	Customer Services	Engineer	35000
15	Katherine	Young	Customer Support	Manager	67241.25
29	Roger	De Souza	Customer Support	Engineer	69482.63
44	Leslie	Phong	Customer Support	Engineer	56034.38
114	Bill	Parker	Customer Support	Engineer	35000
136	Scott	Johnson	Customer Support	Technical Writer	60000
2	Robert	Nelson	Engineering	Vice President	105900
109	Kelly	Brown	Engineering	Administrative Assistant	27000
28	Ann	Bennet	European Headquarters	Administrative Assistant	22935
36	Roger	Reeves	European Headquarters	Sales Co-ordinator	33620.63
37	Willie	Stansbury	European Headquarters	Engineer	39224.06
72	Claudia	Sutherland	Field Office: Canada	Sales Representative	100914
5	Kim	Lambert	Field Office: East Coast	Engineer	102750
11	K. J.	Weston	Field Office: East Coast	Sales Representative	86292.94
134	Jacques	Gilon	Field Office: France	Sales Representative	390500
121	Roberto	Ferrari	Field Office: Italy	Sales Representative	99000000

In the UpdSql2 example, I've provided both alternatives, so that you can test the differences yourself. Here is part of the DFM description of the dataset component:

```

object IBDataSet1: TIBDataSet
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  BufferChunks = 32
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    ' (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, ' +
    ' JOB_COUNTRY)'
    'values'
  
```

```

        ' (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, ' +
        ' :JOB_GRADE, :JOB_COUNTRY)')
SelectSQL.Strings = (...)
UpdateRecordTypes = [cusUnmodified, cusModified, cusInserted]
ModifySQL.Strings = (...)
end

```

If you connect the `IBQuery1` or the `IBDataSet1` components to the data source and run the program, you'll see that the behavior is identical. Not only do the components have a similar effect; the available properties and events are also very similar.

## Monitoring InterBase Express

SQL Monitor works by using a hook into the BDE architecture. For this reason, you cannot use it with applications based on the InterBase Express components. Instead, you can simply embed in your application a copy of the `IBSQLMonitor` component and produce a custom log.

You can even write a more generic monitoring application, as I've done in the `IbxMon` example. I've placed in its form a monitoring component and a `RichEdit` control, and written the following handler for the `OnSQL` event:

```

procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
    if Assigned (RichEdit1) then
        RichEdit1.Lines.Add (TimeToStr (Now) + ': ' + EventText);
end;

```

The `if Assigned` test can be useful when receiving a message during shutdown, and it is required when you add this code directly inside the application you are monitoring.

To receive the messages from other applications (or from the current one), you have to turn on the tracing options of the `IBDatabase` component. In the `UpdSql2` example (discussed earlier, in the section “Building a Live Query”), I turned them all on:

```

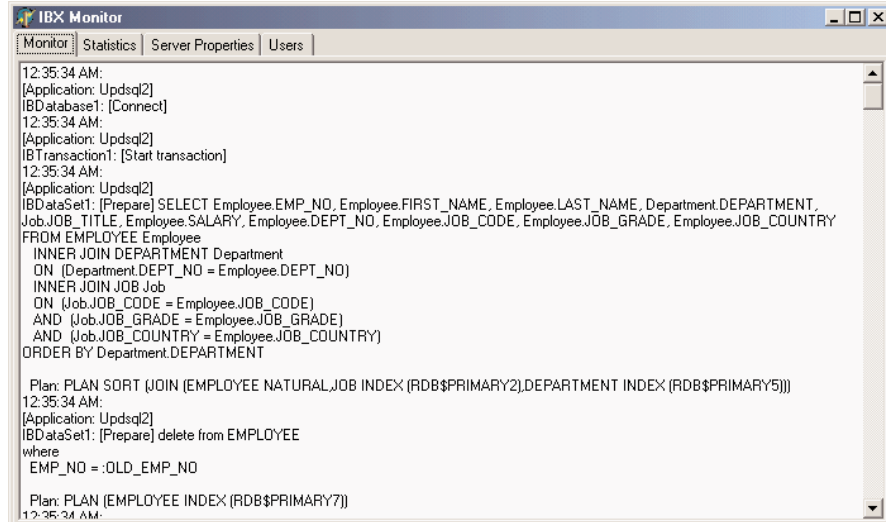
object IBDatabase1: TIBDatabase
    ...
    TraceFlags = [tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
        tfConnect, tfTransact, tfBlob, tfService, tfMisc]

```

If you run the two examples at the same time, the output of the `IbxMon` program will list the details about the `UpdSql2` program's interaction with InterBase, as you can see in Figure 15.6.

**FIGURE 15.6:**

The output of the IbxMon example, based on the IbxMonitor component



```

IBX Monitor
Monitor | Statistics | Server Properties | Users
12:35:34 AM:
[Application: Updsql2]
IBDatabase1: [Connect]
12:35:34 AM:
[Application: Updsql2]
IBTransaction1: [Start transaction]
12:35:34 AM:
[Application: Updsql2]
IBDataSet1: [Prepare] SELECT Employee.EMP_NO, Employee.FIRST_NAME, Employee.LAST_NAME, Department.DEPARTMENT,
Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY
FROM EMPLOYEE Employee
INNER JOIN DEPARTMENT Department
ON (Department.DEPT_NO = Employee.DEPT_NO)
INNER JOIN JOB Job
ON (Job.JOB_CODE = Employee.JOB_CODE)
AND (Job.JOB_GRADE = Employee.JOB_GRADE)
AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)
ORDER BY Department.DEPARTMENT

Plan: PLAN SORT (JOIN (EMPLOYEE NATURAL JOIN DEPARTMENT INDEX (RDB$PRIMARY2), DEPARTMENT INDEX (RDB$PRIMARY5)))
12:35:34 AM:
[Application: Updsql2]
IBDataSet1: [Prepare] delete from EMPLOYEE
where
EMP_NO = :OLD_EMP_NO

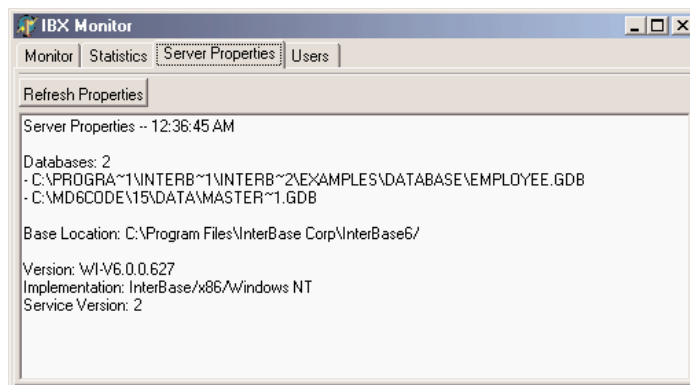
Plan: PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
12:35:34 AM:
  
```

## Getting More System Data

The IbxMon example doesn't only monitor the InterBase connection, but it allows you also to query some settings to the server using the various tabs of its page control. The example embeds a few IBX administrative components, showing server statistics, a few server properties, and all connected users. You can see an example of server properties in Figure 15.7 and the code for extracting the users in the following code fragment.

**FIGURE 15.7:**

The assorted server information displayed by the IbxMon application



```

IBX Monitor
Monitor | Statistics | Server Properties | Users
Refresh Properties
Server Properties -- 12:36:45 AM
Databases: 2
- C:\PROGRAM FILES\INTERBASE\INTERBASE6\EXAMPLES\DATABASE\EMPLOYEE.GDB
- C:\MD6CODE\15\DATA\MASTER~1.GDB
Base Location: C:\Program Files\InterBase Corp\InterBase6\
Version: WI-V6.0.0.627
Implementation: InterBase/x86/Windows NT
Service Version: 2
  
```

```
// grab the users data
IBSecurityService1.DisplayUsers;
// display the name of each user
for i := 0 to IBSecurityService1.UserInfoCount - 1 do
  with IBSecurityService1.UserInfo[i] do
    RichEdit4.Lines.Add (Format ('User: %s, Full Name: %s, Id: %d',
      [UserName, FirstName + ' ' + LastName, UserId]));
```

## Real-World Blocks

Up to now, we've discussed specific techniques related to InterBase programming, but we haven't delved into the development of an actual application, with the problems this presents in practice. In the following subsections, I'll discuss a few practical techniques, with no specific order.

Nando Dessena (who knows InterBase much better than I do) and I have used all of these techniques in a seminar discussing the porting of an internal Paradox application to InterBase. The application discussed in that circumstance was much larger and more complex, and I've trimmed it down to only a few tables to make it fit into the space I have for this chapter.

---

### TIP

The database discussed in this section is called `mastering.gdb` and is hosted on the companion CD inside the `data` subfolder of the folder for this chapter. You can examine it using InterBase Console, possibly after making a copy to a writable drive so that you can fully interact with it.

## Generators and IDs

I've mentioned in the last chapter that I'm quite a fan of an extensive use of IDs to identify the records in each table of a database.

---

### NOTE

I even tend to use a single sequence of IDs for an entire system, something often indicated as an Object ID (OID). The advantage is that I can place a series of related objects in different tables, depending on their internal structure, one of the possible approaches for implementing inheritance using relational tables. In such a circumstance, however, the IDs of the two tables must be unique. As you might not know in advance which objects could be used in place of others, adopting a global OID allows more freedom later. The drawback is that, if you have lots of data, using an integer as the ID (that is, having only 4 billion objects) might not be enough. For this reason, InterBase 6 supports 64-bit generators.

How do you generate the unique values for these IDs when multiple clients are running? Keeping a table with a *latest* value is going to create troubles, as multiple concurrent transactions (from different users) will see the same values. If you don't use tables, you can use a



database-independent mechanism, including the rather large Windows GUIDs or the so-called high-low technique (the assignment of a base number to each client at startup—the high number—that is combined with a consecutive number—the low number—determined by the client).

Another approach, bound to the database, is the use of internal mechanisms for sequences, indicated with different names in each SQL server. In InterBase they are called *generators*. The characteristic of these sequences is that they operate and are incremented outside of transactions, so that they provide unique numbers even to concurrent users (remember that InterBase forces you to open a transaction even to read data).

We've already seen how to create a generator. Here is the definition for the one in my demo database, followed by the definition of the view you can use to query for a new value:

```
create generator g_master;  
  
create view v_next_id (  
    next_id  
) as  
select gen_id(g_master, 1) from rdb$database  
;
```

Inside the RWBlocks application, I've added to a data module an IBQuery component (as I don't need it to be an editable dataset) with the following SQL:

```
select next_id from v_next_id;
```

The advantage, compared to using the direct statement, is that this is easier to write and maintain, even if the underlying generator changes (or in case you switch to a different approach behind the scenes). Moreover, in the same data module I've added a function, which returns a new value for the generator:

```
function TDmMain.GetNewId: Integer;  
begin  
    // return the next value of the generator  
    QueryId.Open;  
    try  
        Result := QueryId.Fields[0].AsInteger;  
    finally  
        QueryId.Close;  
    end;  
end;
```

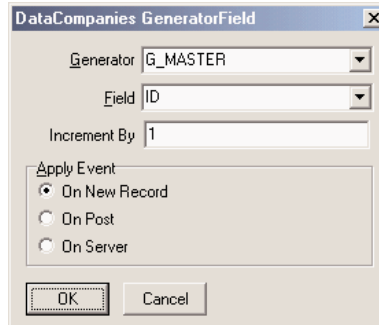
This method can be called in the AfterInsert event of any dataset, to fill in the value for the ID:

```
mydataset.FieldName ('ID').AsInteger := data.GetNewId;
```

As I've mentioned, the IBX datasets in Delphi 6 can be tied directly to a generator, simplifying the overall picture quite a lot. Thanks to the specific property editor (shown in Figure 15.8), in fact, connecting a field of the dataset to the generator becomes trivial.

**FIGURE 15.8:**

The editor of the `GeneratorField` property of the IBX datasets in Delphi 6



Notice that both these approaches are much better than the one, based on a server-side trigger, discussed earlier in this chapter. In that case, in fact, the Delphi application didn't know the ID of the record sent to the database and so was unable to refresh it. Not having the record ID (which is also the only key field) on the Delphi side implies it is almost impossible to insert such a value directly inside a `DBGrid`. If you try, you'll see that the value you insert apparently gets lost right away, only to reappear in case of a full refresh.

Using client-side techniques instead, based on the manual code or the `GeneratorField` property, causes no trouble, as the Delphi application knows the ID, the record key, before posting it, so it can easily place it in a grid and refresh it properly.

## Case-Insensitive Searches

An interesting issue with SQL servers in general, not specifically InterBase, has to do with case-insensitive searches. Suppose you don't want to show a large amount of data inside a grid (which is rather a bad idea for a client/server application). You instead choose to let the user type the initial portion of a name and then filter a query on this input, displaying only the smaller resulting record set in a grid. I've done this for a table of companies.

This search by company name is going to be executed quite frequently and will probably take place on a large table. However, if we simply search using the `starting with` or `like` operators, the search will be case sensitive, as in the following SQL statement:

```
select * from companies
where name starting with 'win';
```

To make a case-insensitive search, you can use the `upper` function on both sides of the comparison to test the uppercase values of each string, but a similar query would be very slow, as it

won't be based on an index. On the other hand, saving the company names (or any other name) in uppercase letters would be rather silly, because when you have to print out those names, the result will be quite unnatural (even if very common in old information systems).

If we can trade off some disk space and memory for the extra speed, we can use a trick: add an extra field to the table, to store the uppercase value of the company name, using a server-side trigger to generate it and update it. We can then ask the database to maintain an index on the uppercase version of the name, to speed our search operation even further.

In practice, the table definition will look like this:

```
create domain d_uid as integer;
create table companies
(
  id          d_uid not null,
  name       varchar(50),
  tax_code   varchar(16),
  name_upper varchar(50),
  constraint companies_pk primary key (id)
);
```

To copy the uppercase name of each company into the related field, we cannot rely on client-side code, as an inconsistency would cause problems. In a case like this, it is better to use a trigger on the server, so that each time the company name changes, its uppercase version is updated accordingly. Another trigger will be used for the insertion of a new company:

```
create trigger companies_bi for companies
active before insert position 0
as
begin
  new.name_upper = upper(new.name);
end;

create trigger companies_bu for companies
active before update position 0
as
begin
  if (new.name <> old.name) then
    new.name_upper = upper(new.name);
  end;
```

Finally, I've added an index to the table with this DDL statement:

```
create index i_companies_name_upper on companies(name_upper);
```

With this structure behind the scenes, we can now select all the companies starting with the text of an edit box (edSearch) by writing the following code in a Delphi application:

```
dm.DataCompanies.Close;
dm.DataCompanies.SelectSQL.Text :=
  'select c.id, c.name, c.tax_code,' +
```

```

    ' from companies c ' +
    ' where name_upper starting with ''' +
    UpperCase (edSearch.Text) + '''';
dm.DataCompanies.Open;

```

**TIP**

Using a prepared parametric query, we might be able to make this code even faster.

As an alternative, we could have created a server-side calculated field in the table definition, but this would have prevented us from having an index on the field, which speeds up our queries considerably:

```
name_upper varchar(50) computed by (upper(name))
```

## Handling Locations and People

You might notice that the table describing companies is quite bare. In fact, it has no company address, nor any contact information. The reason is simple: I want to be able to handle companies that have multiple offices (or locations) and list contact information about multiple employees of those companies.

Every location is bound to a company. Notice, though, that I've decided not to use a location identifier related to the company (such as a progressive location number for each company) but a global ID for all of the locations. This way I can refer to a location ID (let's say, for shipping goods) without having to refer also to the company ID. This is the definition of the table storing company locations:

```

create table locations
(
    id          d_uid not null,
    id_company  d_uid not null,
    address     varchar(40),
    town        varchar(30),
    zip         varchar(10),
    state       varchar(4),
    phone       varchar(15),
    fax         varchar(15),
constraint locations_pk primary key (id),
constraint locations_uc unique (id_company, id)
);

alter table locations add constraint locations_fk_companies
foreign key (id_company) references companies (id)
on update no action on delete no action;

```

The final definition of a foreign key relates the `id_company` field of the `locations` table with the `ID` field of the `companies` table. The other table lists names and contact information for

people at specific company locations. To follow the database normalization rules, I should have added to this table only a reference to the location, as each location relates to a company. However, to make it simpler to change the location of a person within a company and to make my queries much more efficient (avoiding an extra step), I've added to the people table both a reference to the location and to the company.

The table also has another unusual feature: One of the people working for a company can be set as the key contact. This is obtained with a Boolean field (defined with a domain, as the Boolean type is not supported by InterBase) and by adding triggers to the table so that only one employee of each company can have this flag active:

```
create domain d_boolean as char(1)
  default 'F'
  check (value in ('T', 'F')) not null

create table people
(
  id          d_uid not null,
  id_company  d_uid not null,
  id_location d_uid not null,
  name        varchar(50) not null,
  phone       varchar(15),
  fax         varchar(15),
  email       varchar(50),
  key_contact d_boolean,
  constraint people_pk primary key (id),
  constraint people_uc unique (id_company, name)
);

alter table people add constraint people_fk_companies
  foreign key (id_company) references companies (id)
  on update no action on delete cascade;
alter table people add constraint people_fk_locations
  foreign key (id_company, id_location)
  references locations (id_company, id);

create trigger people_ai for people
active after insert position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T') then
    update people
      set key_contact = 'F'
      where id_company = new.id_company
      and id <> new.id;
end;
```

```

create trigger people_au for people
active after update position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T' and old.key_contact = 'F') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;

```

## Building a User Interface

The three tables we have discussed so far have a clear master/detail relation. For this reason, the RWBlocks example uses three IBDataSet components for accessing the data, hooking up the two secondary tables to the main one. The code for the master/detail support is that of a standard database example based on queries, so I won't discuss it further (but I suggest you study the source code of the example).

Each of the datasets has a full set of SQL statements, to make the data editable. Whenever you enter a new detail element, the program hooks it to its master tables, as in the two following methods:

```

procedure TDmCompanies.DataLocationsAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataLocationsID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
end;

procedure TDmCompanies.DataPeopleAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataPeopleID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
  // the suggested location is the active one, if available
  if not DataLocations.IsEmpty then
    DataPeopleID_LOCATION.AsInteger := DataLocationsID.AsInteger;
  // the first person added becomes the key contact
  // (checks whether the filtered dataset of people is empty)
  DataPeopleKEY_CONTACT.AsBoolean := DataPeople.IsEmpty;
end;

```

As this code suggests, a data module hosts the dataset components. Actually, the program has a data module for every form (hooked up dynamically, as you can create multiple instances of each form). Each of these data modules has a separate transaction, so that the various operations done in different pages are totally independent. The database connection, instead, is centralized. A main data module hosts the corresponding component, which is referenced by all the datasets. Each of the data modules is created dynamically by the form referring to it, and its value is stored in the `dm` private field of the form:

```
procedure TFormCompanies.FormCreate(Sender: TObject);
begin
    dm := TDmCompanies.Create (Self);
    dsCompanies.Dataset := dm.DataCompanies;
    dsLocations.Dataset := dm.DataLocations;
    dsPeople.Dataset := dm.DataPeople;
end;
```

This way we can easily create multiple instances of a form, with an instance of the data module connected to each of them. The form connected to the data module has three DBGrid controls, each tied to a data module and one of the corresponding datasets. You can see this form at run time, with some actual data, in Figure 15.9.

**FIGURE 15.9:**

A form showing companies, office locations, and people (part of the RWBlocks example)

The screenshot shows the RWBlocks application window. The form has three data grids. The top grid is titled 'Companies' and shows a list of companies with columns 'ID' and 'NAME'. The selected row is '8 Wintech Italia Srl'. Below this is a search box with the text 'win' and a 'Search' button. The middle grid is titled 'Locations' and shows a list of office locations with columns 'ID\_COMPANY', 'ADDRESS', 'FAX', 'PHONE', 'STATE', and 'TOWN'. The selected row is '10 8 Via Castello 65 Piacenza'. The bottom grid is titled 'People' and shows a list of people with columns 'ID', 'ID\_COMPANY', 'ID\_LOCATION', 'KEY\_CONTACT', 'NAME', and 'EMAIL'. The selected row is '25 8 10 T Marco Cantù'.

ID	ID_COMPANY	ADDRESS	FAX	PHONE	STATE	TOWN
8		Wintech Italia Srl				
10	8	Via Castello 65				Piacenza

ID	ID_COMPANY	ID_LOCATION	KEY_CONTACT	NAME	EMAIL
25	8	10	T	Marco Cantù	

The form is actually hosted by a main form, which in turn is based on a page control, with the other forms embedded. Only the form connected with the first page is created when the program starts. The ShowForm method I've written takes care of parenting the form to the tab sheet of the page control, after removing the form border:

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
    ShortDateFormat := 'dd/mm/yyyy';
    ShowForm (TFormCompanies.Create (self), TabCompanies);
end;

procedure TFormMain.ShowForm (Form: TForm; Tab: TTabSheet);
begin
    Form.BorderStyle := bsNone;
    Form.Align := alClient;
    Form.Parent := Tab;
    Form.Show;
end;
```

The other two pages, instead, are populated at runtime:

```
procedure TFormMain.PageControl1Change(Sender: TObject);
begin
    if PageControl1.ActivePage.ControlCount = 0 then
        if PageControl1.ActivePage = TabFreeQ then
            ShowForm (TFormFreeQuery.Create (self), TabFreeQ)
        else if PageControl1.ActivePage = TabClasses then
            ShowForm (TFormClasses.Create (self), TabClasses);
end;
```

The companies form hosts the search by company name we've already discussed in the last section, plus a search by location. You enter the name of a town and get back a list of companies having an office in that town:

```
procedure TFormCompanies.btnTownClick(Sender: TObject);
begin
    with dm.DataCompanies do
        begin
            Close;
            SelectSQL.Text :=
                'select c.id, c.name, c.tax_code' +
                ' from companies c ' +
                ' where exists (select loc.id from locations loc ' +
                ' where loc.id_company = c.id and upper(loc.town) = ''' +
                UpperCase(edTown.Text) + ''' )';
            Open;
            dm.DataLocations.Open;
            dm.DataPeople.Open;
        end;
    end;
```



If you look at the source code of the form, you'll find a lot more code. Some of it is related to closing permission (as a user cannot close the form while there are pending edits not posted to the database), while a good amount relates to the use of the form as a lookup dialog, as described later.

## Booking Classes

Another portion of the program and of the database involves booking training classes and courses. (Needless to say, although I built this program as a showcase, it also helps me run my own business.) In the database is a classes table listing all the training courses, each with a title and the planned date. Another table hosts registration by company, including the classes registered for, the ID of the company, and some notes. Finally, a third table has a list of people who've signed up, each hooked to a registration for his or her company, with the amount paid.

The rationale behind this company-based registration is that invoices are sent out to companies, which book the classes for their programmers and can receive specific discounts. This is a case in which the database is a little more normalized, as the people registration doesn't refer directly to a class, but only to the company registration for that class. Here is the definition of the tables involved (I've omitted foreign key constraints and other elements):

```
create table classes
(
  id          d_uid not null,
  description varchar(50),
  starts_on   timestamp not null,
  constraint classes_pk primary key (id)
);
create table classes_reg
(
  id          d_uid not null,
  id_company  d_uid not null,
  id_class    d_uid not null,
  notes       varchar(255),
  constraint classes_reg_pk primary key (id),
  constraint classes_reg_uc unique (id_company, id_class)
);
create domain d_amount as numeric(15, 2);
create table people_reg
(
  id          d_uid not null,
  id_classes_reg d_uid not null,
  id_person   d_uid not null,
  amount      d_amount,
  constraint people_reg_pk primary key (id)
);
```

The data module for this group of tables uses a master/detail/detail relationship, and has code to set the connection with the active master record when a new detail record is created. Each dataset has a generator field for its ID, and each has the proper update and insert SQL statements. These statements have been generated by the corresponding component editor using only the ID field to identify existing records and updating only the fields of the original table. In fact, each of the two secondary datasets retrieves data from a lookup table, either the list of companies or the list of people. Finally, I had to edit manually the RefreshSQL statements to repeat the proper inner join. Here is an example:

```

object IBClassReg: TIBDataSet
  Database = DmMain.IBDatabase1
  Transaction = IBTransaction1
  AfterInsert = IBClassRegAfterInsert
  DeleteSQL.Strings = (
    'delete from classes_reg'
    'where id = :old_id')
  InsertSQL.Strings = (
    'insert into classes_reg (id, id_class, id_company, notes)'
    'values (:id, :id_class, :id_company, :notes)')
  RefreshSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id = :id')
  SelectSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id_class = :id')
  ModifySQL.Strings = (
    'update classes_reg'
    'set'
    ' id = :id,'
    ' id_class = :id_class,'
    ' id_company = :id_company,'
    ' notes = :notes'
    'where id = :old_id')
  GeneratorField.Field = 'id'
  GeneratorField.Generator = 'g_master'
  DataSource = dsClasses
end

```

To complete the discussion of IBClassReg, here is its only event handler:

```

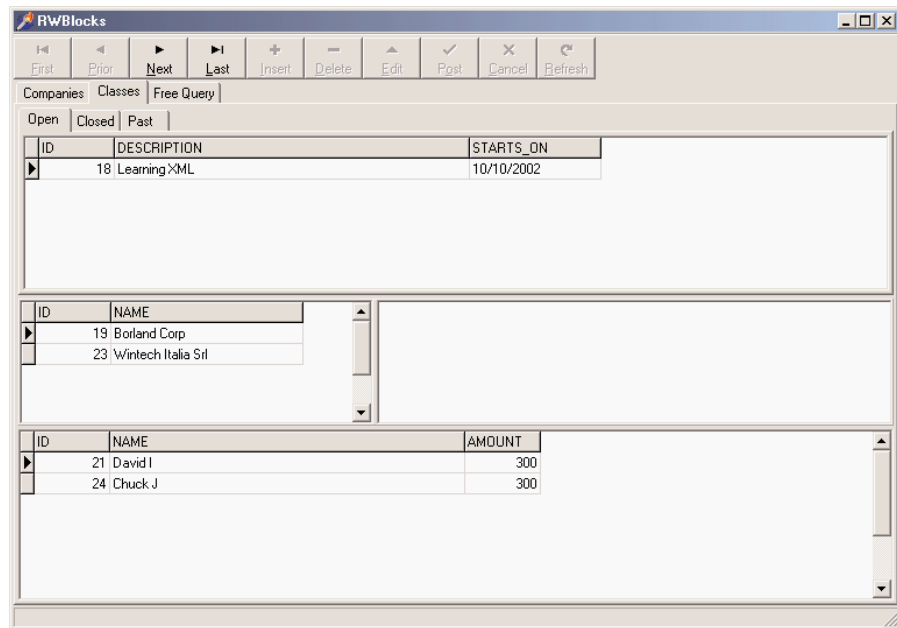
procedure TDmClasses.IBClassRegAfterInsert(DataSet: TDataSet);
begin
  IBClassReg.FieldName ('id_class').AsString :=
    IBClasses.FieldName ('id').AsString;
end;

```

The IBPeopleReg dataset has similar settings, but the IBClasses dataset is simpler, at design time. At run time, the SQL code of this dataset is dynamically modified, using three alternatives to display scheduled classes (whenever the date is after today's date), classes already started or finished in the current year, and classes of past years. A user selects one of the three groups of records for the table with a tab control, which hosts the DBGrid for the main table (see Figure 15.10).

**FIGURE 15.10:**

The form for class registrations of the RWBlocks example



The three alternative SQL statements are created when the program starts, or actually when the class registrations form is created and displayed. The program stores the final portion of the three alternative instructions (the where clause) in a string list, and selects one of the strings when the tab changes:

```
procedure TFormClasses.FormCreate(Sender: TObject);
begin
    dm := TdmClasses.Create (Self);
    // connect the datasets to the data sources
    dsClasses.Dataset := dm.IBClasses;
    dsClassReg.Dataset := dm.IBClassReg;
    dsPeopleReg.Dataset := dm.IBPeopleReg;
    // open the datasets
    dm.IBClasses.Active := True;
    dm.IBClassReg.Active := True;
    dm.IBPeopleReg.Active := True;
```

```

// prepare the SQL for the three tabs
SqlCommands := TStringList.Create;
SqlCommands.Add (' where Starts_On > ''now''');
SqlCommands.Add (' where Starts_On <= ''now'' and ' +
  ' extract (year from Starts_On ) >= extract(year from current_timestamp)');
SqlCommands.Add (' where extract (year from Starts_On) < ' +
  ' extract(year from current_timestamp)');
end;
procedure TFormClasses.TabChange(Sender: TObject);
begin
  dm.IBClasses.Active := False;
  dm.IBClasses.SelectSQL [1] := SqlCommands [Tab.TabIndex];
  dm.IBClasses.Active := True;
end;

```

## Building a Lookup Dialog

The two detail datasets of this class registration form display some lookup fields. Instead of showing the ID of the company that booked the class, for example, it shows the company name. This is obtained with an inner join in the SQL statement and by configuring the DBGrid columns not to display the company ID. In a local application, or one with a limited amount of data, we could have used a lookup field. However, copying the entire lookup dataset locally or opening it for browsing should be limited to tables with about a hundred records at most, embedding some search capabilities.

If you have a large table, like a table of companies, an alternative solution can be to use a secondary dialog box to do the lookup selection. For example, we can choose a company using the form we've already built and taking advantage of its search capabilities. To display this form as a dialog box, the program creates a new instance of it, shows some hidden buttons already there at design time, and lets the user select a company to refer to from the other table.

To simplify the use of this lookup, which can happen multiple times in a large program, I've added to the companies form a class function, having as output parameters the name and ID of the selected company. An initial ID can be passed to the function to determine its initial selection. Here is the complete code of this class function, which creates an object of its class, selects the initial record if requested, shows the dialog box, and finally extracts the return values:

```

class function TFormCompanies.SelectCompany (
  var CompanyName: string; var CompanyId: Integer): Boolean;
var
  FormComp: TFormCompanies;
begin

```

```

Result := False;
FormComp := TFormCompanies.Create (Application);
FormComp.Caption := 'Select Company';
try
  // activate dialog buttons
  FormComp.btnCancel.Visible := True;
  FormComp.btnOK.Visible := True;
  // select company
  if CompanyId > 0 then
    FormComp.dm.DataCompanies.SelectSQL.Text :=
      'select c.id, c.name, c.tax_code' +
      ' from companies c ' +
      ' where c.id = ' + IntToStr (CompanyId)
  else
    FormComp.dm.DataCompanies.SelectSQL.Text :=
      'select c.id, c.name, c.tax_code' +
      ' from companies c ' +
      ' where name_upper starting with ''a''';
  FormComp.dm.DataCompanies.Open;
  FormComp.dm.DataLocations.Open;
  FormComp.dm.DataPeople.Open;

  if FormComp.ShowModal = mrOK then
    begin
      Result := True;
      CompanyId := FormComp.dm.DataCompanies.FieldByName ('id').AsInteger;
      CompanyName := FormComp.dm.DataCompanies.FieldByName ('name').AsString;
    end;
  finally
    FormComp.Free;
  end;
end;

```

Another slightly more complex class function (available within the example's source code, but not listed here) allows the selection of a person of a given company to register people for classes. In this case, the form is displayed after disallowing searching another company or changing it.

In both cases, the lookup is triggered by adding an ellipsis button to the column of the DBGrid—for example, the grid column listing the names of companies registered for classes. When this button is pressed, the program calls the class function to display the dialog box and uses its result for updating the hidden ID field and the visible name field:

```

procedure TFormClasses.DBGridClassRegEditButtonClick(Sender: TObject);
var
  CompanyName: string;

```

```
    CompanyId: Integer;  
begin  
    CompanyId := dm.IBClassReg.FieldName ('id_Company').AsInteger;  
    if TFormCompanies.SelectCompany (CompanyName, CompanyId) then  
        begin  
            dm.IBClassReg.Edit;  
            dm.IBClassReg.FieldName ('Name').AsString := CompanyName;  
            dm.IBClassReg.FieldName ('id_Company').AsInteger := CompanyId;  
        end;  
end;
```

## Adding a Free Query Form

The final feature of the program is a form where a user can directly type in a SQL statement and run it. As a helper, the form lists in a combo box the available tables of the database, obtained when the form is created by calling:

```
DmMain.IBDatabase1.GetTableNames (ComboTables.Items);
```

Selecting an item of the combo box generates a simple SQL query:

```
MemoSql.Lines.Text := 'select * from ' + ComboTables.Text;
```

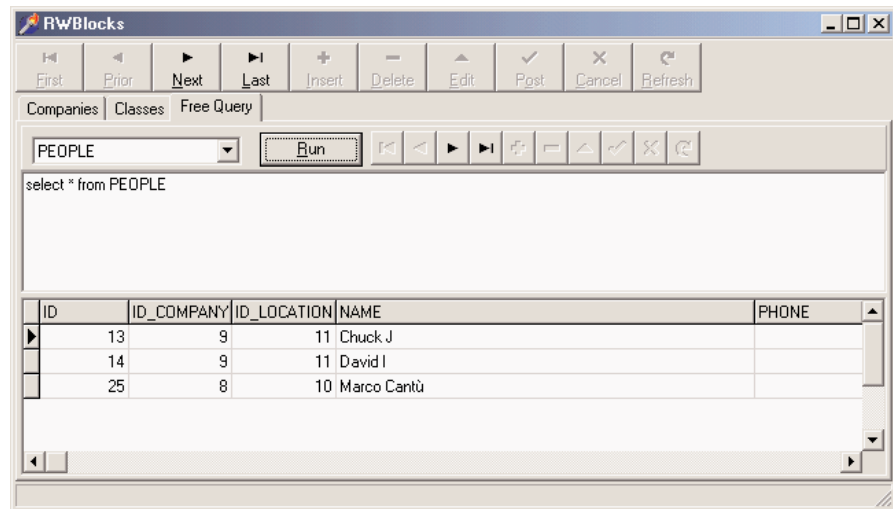
The user, if an expert, can then edit the SQL, possibly introducing restrictive clauses, and then run the query:

```
procedure TFormFreeQuery.ButtonRunClick(Sender: TObject);  
begin  
    QueryFree.Close;  
    QueryFree.SQL := MemoSql.Lines;  
    QueryFree.Open;  
end;
```

You can see this third form of the RWBlocks program in Figure 15.11. Of course, I'm not suggesting that you add SQL editing to programs intended for all of your users. This feature is intended for power users, maybe programmers. I basically wrote it for myself!

**FIGURE 15.11:**

The free query form of the RWBlocks example is intended for power users.



## What's Next?

After looking at other, more general, database access technologies Delphi provides (such as the BDE and dbExpress), in this chapter I've introduced the InterBase database and the IBX family of components. The last part of the chapter presented a complete real-world application, discussing a series of general-purpose techniques you can probably apply even to completely different InterBase applications.

Now we are ready to focus on another data access alternative, Microsoft's own ADO technology, which Delphi fully supports with a specific set of dataset components since version 5. In subsequent chapters, we'll continue to explore database development, with multitier architectures and the development of database-oriented Delphi components.

# ActiveX Data Objects

- Microsoft Data Access Components (MDAC)
- dbGo
- Data link files
- Getting schema information
- Using the Jet engine
- Transaction processing
- Disconnected and persistent recordsets
- The briefcase model and deploying MDAC



---

I wish to acknowledge and thank Guy Smith-Ferrier for writing this chapter. Guy is a programmer, author, and speaker. He is the author of several commercial software products and countless internal systems for independent and blue-chip companies alike. He has written many articles for *The Delphi Magazine* and for others on topics beyond Delphi, and he has spoken at numerous conferences in North America and Europe. Guy lives in England with his wife, his son, and his cat.

---

**S**ince the mid-1980s, database programmers have been on a quest for the “holy grail” of *database independence*. The idea is to use a single API that applications can use to interact with many different sources of data. The use of such an API would release developers from a dependence upon a single database engine and allow them to adapt to the world’s changing demands. Vendors have produced many solutions to this goal, the two most notable early solutions being Microsoft’s Open Database Connectivity (ODBC) and Borland’s Independent Database Application Programming Interface (IDAPI), more commonly known as the Borland Database Engine (BDE).

Microsoft started to replace ODBC with OLE DB in the mid-1990s with the success of COM. However, OLE DB is what Microsoft would class a *system-level* interface and is intended to be used by system-level programmers. It is very large and complex and unforgiving. It makes greater demands on the programmer and requires a higher level of knowledge in return for lower productivity. ActiveX Data Objects (ADO) is a layer on top of OLE DB and is referred to as an *application-level* interface. It is considerably simpler than OLE DB and more forgiving. In short, it is designed for use by application programmers.

ADO has great similarities with the BDE. They are, after all, designed to solve very similar problems. Both support navigation of datasets, manipulation of datasets, transaction processing, and cached updates (called *batch updates* in ADO), so the concepts and issues involved in using ADO are similar to those of the BDE. However, there are also differences. ADO is a more recent technology. This gives it an advantage over the BDE because it is better suited to today’s needs and doesn’t need to carry so much deadwood around with it. Perhaps more importantly, ADO has a wider interpretation of “data.” The BDE is used for accessing “rectangular” data—that is, data in rows and columns. This is ideal for accessing data from databases. ADO can be used for accessing this data but can also be used for accessing non-rectangular data, including directory structures, documents, Web sites, and e-mail.

In this chapter we will look at ADO and *dbGo*. (This set of Delphi components was called ADOExpress but has been renamed dbGo in Delphi 6, because Microsoft has objected to the use of the term *ADO* in product names.) It is possible to use ADO in Delphi without using dbGo. By importing the ADO type library, you can gain direct access to the ADO interfaces; this is, indeed, how Delphi programmers used ADO before the release of Delphi 5. However,

this path bypasses Delphi's database infrastructure and ensures that you are unable to make use of other Delphi technologies such as DataSnap. Alternatively, you can turn to Delphi's active third-party market for other ADO component suites such as Adonis, AdoSolutio, Diamond ADO, and Kamiak.

This chapter uses dbGo for all of its examples, not only because it is readily available and supported but also because it is a very viable solution. Regardless of your final choice, you will find the information here useful.

## Microsoft Data Access Components (MDAC)

ADO is part of a bigger picture called Microsoft Data Access Components (MDAC). MDAC is an umbrella for Microsoft's database technologies and includes ADO, OLE DB, ODBC, and RDS (Remote Data Services). Often you will hear people use the terms MDAC and ADO interchangeably (but incorrectly) because their version numbers and releases are now aligned. As ADO is only distributed as part of MDAC, we talk in terms of MDAC releases. The major releases of MDAC have been versions 1.5, 2.0, 2.1, 2.5, and 2.6. Microsoft releases MDAC independently and makes it available for free download and virtually free distribution (there are distribution requirements, but almost all Delphi developers will not have trouble meeting these requirements). MDAC is also distributed with most Microsoft products that have some kind of database content. This includes Windows 98, Windows 2000, Windows Millennium Edition, Microsoft Office, Internet Explorer, and SQL Server. In addition, Delphi 6 Enterprise ships with MDAC 2.5, and Delphi 5 Enterprise ships with MDAC 2.1.

There are two consequences of this level of availability. First, it is highly likely that your users will already have MDAC installed on their machines. Second, whatever version your users have, or you upgrade them to, it is also virtually certain that someone, either you, your users, or other application software, will upgrade their existing MDAC to whatever the current release of MDAC is. There is no way you will be able to prevent this, as MDAC is installed with such commonly used software as Internet Explorer. Add to this the fact that Microsoft supports only the current release of MDAC and the release before it, and you are forced to arrive at a conclusion:

Your applications must be designed to work with the current release of MDAC or the release before it.

If you chart the releases of MDAC, you can expect to see a new version of MDAC every 10 months on average (Delphi itself has a new release every 14 months on average).

As an ADO developer, you should regularly check the MDAC pages on Microsoft's site at [www.microsoft.com/data](http://www.microsoft.com/data). From there you can download the latest version of MDAC for free. At the time of writing, this is MDAC 2.6, which you can download from [www.microsoft.com/data/download\\_260rtm.htm](http://www.microsoft.com/data/download_260rtm.htm) (5.2 MB), but you should check for a more recent version first.

While you are on this Web site, you should take the opportunity to download the MDAC SDK (13 MB) if you do not already have it or the Platform SDK (the MDAC SDK is part of the Platform SDK). The MDAC SDK is your bible. Download it and consult it regularly and use it to answer your ADO questions. You should treat this as your first port of call when you need MDAC information. Beyond this, you should read the README files that come with MDAC. Look in \Program Files\Common Files\System\ADO for all of the files ending README.TXT.

Finally, in getting ready for using ADO in your Delphi applications, you should check for dbGo/ADOExpress updates on Borland's excellent community site (<http://community.borland.com>). Informal patches are released here that are often a must-have. For example, there is a show-stopping problem when using MDAC 2.6 and Delphi 5's ADOExpress, which requires a patch.

## OLE DB Providers

OLE DB providers enable access to a source of data. They are ADO's equivalent to the BDE's drivers. However, although the BDE's Driver SDK has been available for many years, there are no third-party BDE drivers. This is not the case for OLE DB providers. MDAC includes many providers that I'll discuss, but many more are available from Microsoft and, more prolifically, the third-party market. It is no longer possible to reliably list all available OLE DB providers, because the list is so large and ever-changing, but here are some of the main vendors of these drivers:

Company	Web Site	OLE DB Provider
B2 Systems	<a href="http://www.b2systems.com">www.b2systems.com</a>	SQL Server, Oracle, Sybase, Informix, RDB, DB2, flat files
ISG	<a href="http://www.isgsoft.com">www.isgsoft.com</a>	ISG Navigator (ISAM, DB2, IMS, Informix, Jasmine, Open Ingres, Oracle, SQL Server, Sybase, Adabas, RDB, RMS, VSAM)
Merant	<a href="http://www.merant.com">www.merant.com</a>	DB2, Informix, Lotus Notes, SQL Server, Ingres, Oracle, Sybase

You should add to this list almost all database vendors, as the majority now supply their own OLE DB providers. For example, Oracle supplies the ORAOLEDB provider. Notable omissions include InterBase; at the time of writing, Borland doesn't plan to write an OLE DB provider for InterBase. Your only solutions are to access InterBase either using the ODBC Driver, or through Jason Wharton's InterBase provider ([www.ibobjects.com](http://www.ibobjects.com), although this is

still in development) or Dmitry Kovalenko's IBProvider ([www.1cpi.1ipetsk.ru/prog/eng/index.html](http://www.1cpi.1ipetsk.ru/prog/eng/index.html)).

**TIP**

Jason Wharton's OLE DB Provider is additionally interesting because it is written using Binh Ly's OLE DB Provider Development Toolkit ([www.techvanguards.com/products/optk/install.htm](http://www.techvanguards.com/products/optk/install.htm)). If you want to write your own OLE DB provider, this is an easier way than most.

## MDAC OLE DB Providers

When you install MDAC, you automatically install the OLE DB providers shown in Table 16.1:

**TABLE 16.1:** OLE DB Providers Included with MDAC

Driver	Provider	Description
MSDASQL	ODBC Drivers	ODBC drivers (default)
Microsoft.Jet.OLEDB.3.5	Jet 3.5	MS Access 97 databases only
Microsoft.Jet.OLEDB.4.0	Jet 4.0	MS Access databases et al.
SQLOLEDB	SQL Server	MS SQL Server databases
MSDAORA	Oracle	Oracle databases
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	Example of an OLE DB provider for CSV files
MSDAOSP	Simple provider	For creating your own providers for simple text data

If you do not specify which OLE DB provider you are using, OLE DB defaults to the ODBC OLE DB Provider, which is used for backward compatibility with ODBC. As you learn more about ADO, you will discover the limitations of this provider. It's probable that you will eventually tire of these limitations, and I recommend that you look from the beginning for an OLE DB provider that is specific to your database rather than struggle with the ODBC OLE DB Provider.

The Jet OLE DB Providers support MS Access and other “desktop” databases. We will return to these providers later.

The SQL Server Provider supports SQL Server 7, SQL Server 2000, and Microsoft Database Engine (MSDE). MSDE is worth taking a moment's thought over. MSDE is SQL Server with most of the tools removed and some code added to deliberately degrade performance when there are more than 5 active connections. MSDE is important for two reasons. First, it is free. You can download it from Microsoft's Web site and, with very few restrictions, distribute it with your application. Second, MSDE is SQL Server. Of course you don't get the

SQL Server tools, and it does deliberately degrade performance, but it is SQL Server. This means that it is perfect for use with low numbers of users. When the number of users increases and performance starts to suffer, your upgrade path to SQL Server is just a question of paying for SQL Server. Compatibility is virtually assured. You use the same OLE DB provider, and you use it in exactly the same way with exactly the same names and parameters. This is because MSDE is SQL Server. Because of these reasons and because Microsoft is moving their emphasis away from Access, MSDE is worth considering for future developments.

The OLE DB Provider For OLAP can be used directly but is more often used by ADO Multi-Dimensional (ADOMD). ADOMD is an additional ADO technology designed to provide Online Analytical Processing (OLAP). If you have used Delphi's Decision Cube, or Excel's Pivot Tables, or Access's Cross Tabs, then you have used some form of OLAP.

In addition to these MDAC OLE DB providers, Microsoft supplies other OLE DB providers with other products or with downloadable SDKs. The Active Directory Services OLE DB Provider is included with the ADSI SDK; the AS/400 And VSAM OLE DB Provider is included with SNA Server; and the Exchange OLE DB Provider is included with Microsoft Exchange 2000.

The OLE DB Provider For Indexing Service provides access to (and is part of) Microsoft Indexing Service, a Windows NT and 2000 mechanism that speeds up file searches by building catalogs of file information. Indexing Service is integrated into IIS and, consequently, is often used for indexing Web sites. Microsoft Indexing Service is also available for Windows NT 4 as part of the NT 4 Option Pack.

The OLE DB Provider For Internet Publishing is included with Internet Explorer 5, Windows 2000, and Office 2000 and allows developers to manipulate directories and files using HTTP. This is useful for maintaining Web sites that support either FrontPage Web Extender (WEC) or Web Distributed Authoring and Versioning (WebDAV).

Still more OLE DB providers come in the form of *service providers*. As their name implies, OLE DB service providers provide a service to other OLE DB providers. Often these service providers will go unnoticed because they are invoked automatically as needed without programmer intervention. The Cursor Service, for example, is invoked when you create a client-side cursor, and the Persisted Recordset provider is invoked to save data locally.

## dbGo

The set of components that make up dbGo (Table 16.2) should be easily recognizable by programmers familiar with the BDE, dbExpress, or IBExpress.

**TABLE 16.2:** dbGo Components

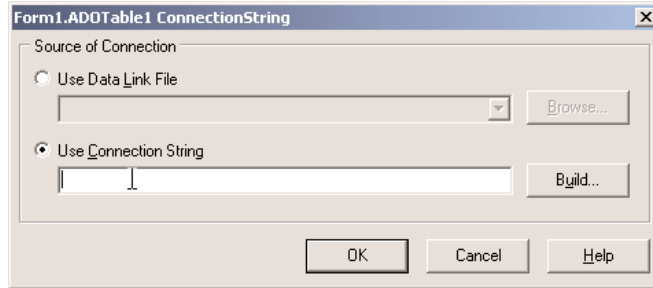
dbGo Component	Description	BDE Equivalent Component
TADOConnection	Connection to a database	TDatabase
TADOCommand	Executes an action SQL command	No equivalent
TADODataset	All-purpose TDataSet	No equivalent
TADOTable	Encapsulation of a table	TTable
TADOQuery	Encapsulation of SQL SELECT	TQuery
TADOStoredProc	Encapsulation of a stored procedure	TStoredProc
TRDSCConnection	Remote Data Services connection	No equivalent

The four dataset components (TADODataset, TADOTable, TADOQuery, TADOStoredProc) are implemented almost entirely by their immediate ancestor TCustomADODataset. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component. As such, the components have a lot in common. In general, however, TADOTable, TADOQuery, and TADOStoredProc are viewed as “compatibility” components and are used to aid the transition of knowledge and code from their BDE counterparts. Be warned, though: These compatibility components are similar to their counterparts but not exactly the same. You will find differences in any application except the most trivial. TADODataset is the component of choice partly because of its versatility but also because it is closer in appearance to the ADO Recordset interface upon which it is based. Throughout this chapter, we will use all of the TDataSet components to give you the experience of using each.

Enough theory. Let’s see some action. Drop a TADOTable onto a form. Look in the Object Inspector and you will not see any DatabaseName or AliasName properties. ADO doesn’t use aliases, so there are no alias-related properties. Instead ADO runs on *connection strings*. Connection strings are the lifeblood of ADO, and you should take time out to master this subject. You can type in a connection string by hand if you know what you are doing, but only programmers who think that VI is a great editor of our time enjoy doing this. For the rest of us, there is the connection string editor. In the Object Inspector, click the ellipses in the ConnectionString property. This invokes Delphi’s connection string editor (Figure 16.1).

**FIGURE 16.1:**

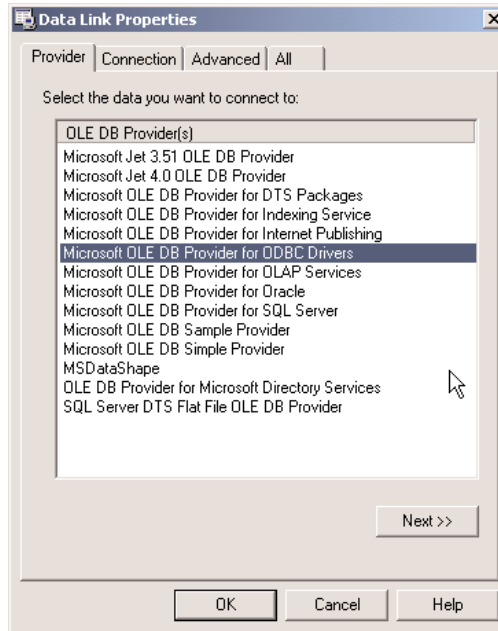
Delphi's connection string editor



This editor adds little value to the process of entering a connection string, so you can click Build to go straight to Microsoft's connection string editor (Figure 16.2).

**FIGURE 16.2:**

Microsoft's connection string editor



This is a tool you will come to know and love—or maybe just to know. The first tab shows the OLE DB providers and service providers installed on your computer. The list will vary according to the version of MDAC and other software installed on your computer. You can see that the OLE DB Provider for ODBC Drivers is selected by default. In our first example, we will open the infamous MS Access Northwind database using the Jet 4.0 OLE DB Provider. Northwind is the Microsoft equivalent of DBDEMOS; it is the test data used in

many examples on ADO because it is so widely available. The exact location of Northwind and its name are not fixed, but you should search your hard disk for `Northwind.mdb` or `NWind.mdb`. Microsoft SQL Server comes with a very similar version of the same database, also called Northwind.

Double-click the Jet 4.0 OLE DB Provider and you will be presented with the Connection tab. This page varies according to the provider you select, but for Jet it simply asks you for the name of the database and your login details. If you have Microsoft Office installed on your computer, then the database name will probably be

```
c:\program files\microsoft office\office\samples\northwind.mdb
```

Click the Test Connection button to test the validity of your selections.

The Advanced tab handles access control to the database, and this is where you would specify exclusive or read-only access to the database.

The All tab lists all the parameters in the connection string. The list is specific to the OLE DB provider you selected on the first page. You should make a mental note of this page, because it contains many parameters that are the answers to many problems. Click OK to close the Microsoft connection string editor, click OK again to close the Borland connection string editor, and the value will be returned to the `ConnectionString` property, which will now be set to

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\program files\microsoft office\office\samples\northwind.mdb;Persist Security Info=False
```

So connection strings are just a string with many parameters delimited by semicolons. If you want to add, edit, or delete any of these parameter values programmatically, you must write your own routines to find the parameter in the list and amend it appropriately.

Now that we have set the connection string, we can select a table. Drop down the list of tables using the `TableName` property in the Object Inspector. Select the Customers table and set `Active` to `True`. Add a `TDataSource` and a `TDBGrid` and connect them all together, and you are now using ADO.

Incidentally, if you are going to use dbGo on a permanent basis, you might benefit from a simple tip. If you followed along with the last example, you will have noticed that you had to flip back and forth between the Data Access page and the ADO page to drop a `TDataSource` component onto the form. If you use dbGo exclusively, then the `TDataSource` component is the only component you will ever need on the Data Access page. Delphi's IDE prevents you from adding the same component to multiple pages, but you can move components. To move the `TDataSource` component from the Data Access page to the ADO page, right-click the palette, select Properties, and drag `TDataSource` onto the ADO page. If, however, you use both ADO and another database technology such as the BDE, then you can simulate installing



TDataSource on multiple pages by creating a Component Template for a TDataSource and installing it on the ADO page. This is a more elegant solution than creating a TADODataSource descendant, because the component that is dropped onto the form is still a genuine TDataSource and not some other component.

## TADOConnection

When a TADOTable component is used in this way, it creates its own connection component behind the scenes in the same way that the BDE components create their temporary TDatabase component. You do not have to accept the default connection it creates, and you should not accept it. Instead, you should create your own connection in the form of a TADOConnection component.

The TADOConnection component is used for many of the same purposes as the BDE's TDatabase component. It allows you to customize the login procedure, control transactions, execute action commands directly, and reduce the number of connections in an application. Using a TADOConnection is easy. Place one on a form and set its `ConnectionString` property in the same way as for the TADOTable. Alternatively, you can double-click a TADOConnection to invoke the connection string editor directly. With the `ConnectionString` set to `Northwind.mdb`, you can disable the login dialog box by setting `LoginPrompt` to `False`. To make use of the new connection, set `ADOTable1`'s `Connection` property to `ADOConnection1`. You will see `ADOTable1`'s `ConnectionString` property reset because `Connection` and `ConnectionString` are mutually exclusive. One of the benefits of using a TADOConnection is that the connection string is centralized instead of scattered throughout many components. Another, more important, benefit is that all of the components that share the TADOConnection share a single connection. Without your own TADOConnection, each ADO dataset uses its own connection.

## Data Link Files

So a TADOConnection allows us to centralize the definition of a connection string within a form or data module. However, even though this is a worthwhile step forward from scattering the same connection string throughout all ADO datasets, it still suffers from a fundamental flaw: If you use a database engine that defines the database in terms of a filename, then the path to the database file(s) is hard-coded in the EXE. This makes for a very fragile application. The BDE uses aliases to overcome this problem; ADO uses Data Link files. A Data Link file is a connection string in an INI file. The following is an example of a Data Link file:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\program files\microsoft office\
office\samples\northwind.mdb;Persist Security Info=False
```

Although you can give a Data Link file any extension, the recommended extension is .UDL. You can create a Data Link using any text editor, or you can right-click Windows Explorer, select New, then Text Document, rename the file with a UDL extension, and then double-click the file to invoke the Microsoft connection string editor.

To use the Data Link file, set the TADOConnection's ConnectionString to:

```
File Name=TEST.UDL
```

assuming that the file is called TEST.UDL and it is in the same directory as the EXE. You can place your Data Link files anywhere on the hard disk, but if you are looking for a common, shared location, then you can use the DataLinkDir function in ADO DB.PAS:

```
ShowMessage('The Data Link directory is ' + DataLinkDir);
```

If you haven't altered MDAC's defaults, DataLinkDir will return

```
C:\Program Files\Common Files\System\OLE DB\Data Links
```

Delphi 5's ADOExpress suffers from a flaw when using data link files, which is fixed in Delphi 6's dbGo. In 5, set the Connected property of the TADOConnection to True and watch the ConnectionString property. The ConnectionString becomes the actual connection string that is in use (i.e., the one from the data link file). The problem is that when the property is streamed to the form when the application is saved, it is the active connection string that is saved. The reference to the data link file is permanently forgotten. If you change the data link file, you will not see any change to your application.

If you want to use data link files and you don't want to suffer from this problem, use the following TADOConnectionX component instead of TADOConnection:

```
TADOConnectionX = class(TADOConnection)
private
    FUDLFile: string;
protected
    procedure DoConnect; override;
published
    property UDLFile: string read FUDLFile write FUDLFile;
end;
```

The component has a UDLFile property where the UDL filename is permanently stored (you must manually set this property yourself). The class has a single method, DoConnect, which ensures that the UDL file is always read each time the connection is opened:

```
procedure TADOConnectionX.DoConnect;
begin
    if FUDLFile <> '' then
        ConnectionString:= 'File Name=' + FUDLFile;
    inherited;
end;
```

## Dynamic Properties

Imagine that you are responsible for designing a new database middleware architecture. You have to reconcile two opposing goals of a single API for all databases and access to database-specific features. You could take the approach of designing an interface that is the sum of all of the features of every database ever created. Each class would have every property and method imaginable, but it would only use the properties and methods it had support for. It doesn't take much discussion to realize that this isn't a good solution. ADO has to solve these apparently mutually exclusive goals, and it does so using *dynamic properties*. Almost all ADO interfaces, and their corresponding dbGo components, have a property called Properties that is a collection of database-specific properties. These properties can be accessed by their ordinal position, like this:

```
ShowMessage(ADOTable1.Properties[1].Value);
```

But they are more usually accessed by name like this:

```
ShowMessage(ADOConnection1.Properties['DBMS Name'].Value);
```

It would be tedious to list all of the different dynamic properties for all of the different classes for all of the different OLE DB providers for all of the situations in which they are used, but to give you an idea of their importance, a typical ADO Connection or Recordset has approximately 100 dynamic properties. As we will see throughout this chapter, the answers to many ADO questions lie in dynamic properties, so keep your eyes and ears open for the ones that solve your problems. An important event, `OnRecordsetCreate`, was planned to be added to `TCustomADODataset` in Delphi 6, which you may need to be aware of when using dynamic properties. (This event was not yet included as this book went to press.) `OnRecordsetCreate` is called immediately after the recordset has been created but has not been opened. This is useful when setting some dynamic properties as certain properties can only be set when the recordset is closed.

## Getting Schema Information

One of the BDE components for which there is no apparent ADO alternative is `TSession`. `TSession` is used for several purposes, but a common use is to retrieve schema information (information about the structure of the database and its contents). In ADO this information can be retrieved using `TADOConnection`'s `OpenSchema` method. This method accepts four parameters. The first, and most interesting, is the kind of data that `OpenSchema` should return. It is a `TSchemaInfo` value, which is a set of 40 values including those for retrieving a list of tables, indexes, columns, views, and stored procedures. The second parameter is a filter to place on the data before it is returned. We will see an example of this parameter in a moment.

The third parameter is a GUID for a provider-specific query and is only used if the first parameter is `siProviderSpecific`. The fourth and final parameter is a `TADODataSet` into which the data is returned. This last parameter illustrates a common theme in ADO: any method that needs to return more than a small amount of data will return its data as a `Recordset`, or in `dbGo` terms, a `TADODataSet`.

To use `TADOConnection.OpenSchema`, you need an open `TADOConnection`. The following example retrieves a list of primary keys for every table into a `TADODataSet`:

```
ADOConnection1.OpenSchema(siPrimaryKeys, EmptyParam, EmptyParam, ADODataSet1);
```

Each field in a primary key has a single row in the result set. So a table with a composite key of two fields has two rows. The two `EmptyParam` values indicate that these parameters are given empty values and are ignored.

When `EmptyParam` is passed as the second parameter, the result set includes all information of the requested type for the entire database. For many kinds of information, you will want to filter the result set. You can, of course, apply a traditional Delphi filter to the result set using the `Filter` and `Filtered` properties or the `OnFilterRecord` event. However, this applies the filter on the client side in this example. Using the second parameter, we can apply a more efficient filter at the source of the schema information. The filter is specified as an array of values. Each element of the array has a specific meaning relevant to the kind of data being returned. For example, the filter array for primary keys has three elements: the first is the catalog (*catalog* is ANSI-speak for the database), the second is the schema, and the third is the table name. This example returns a list of primary keys for the `Customers` table:

```
var
  Filter: OLEVariant;
begin
  Filter := VarArrayCreate([0, 2], varVariant);
  Filter[2] := 'CUSTOMERS';
  ADOConnection1.OpenSchema(
    siPrimaryKeys, Filter, EmptyParam, ADODataSet1);
end;
```

You can retrieve the same information using `ADOX`, and this warrants a brief comparison between `OpenSchema` and `ADOX`. `ADOX` is an additional ADO technology that allows you to retrieve and update schema information. It is ADO's equivalent to SQL's Data Definition Language (DDL, i.e., `CREATE`, `ALTER`, `DROP`) and Data Control Language (DCL, i.e., `GRANT`, `REVOKE`). `ADOX` is not directly supported in `dbGo`, but you can easily import the `ADOX` type library and use it successfully in Delphi applications. Unfortunately, `ADOX` is not as universally implemented as `OpenSchema`, so there are greater gaps. If you just want to retrieve information and not to update it, then `OpenSchema` is usually a better choice.

## Using the Jet Engine

Now that you have some of the MDAC and ADO basics under your belt, we can take a moment out to look at the Jet engine. This engine is of great interest to some and of no interest to others. If you're interested in Access, Paradox, dBase, text, Excel, Lotus 1-2-3, or HTML, then this section is for you. If you have no interest in any of these formats, you can safely skip this section.

The Jet database engine is usually associated with Microsoft Access databases, and this is, indeed, its forte. However, the Jet engine is also an all-purpose desktop database engine, and this lesser-known attribute is where much of its strength lies. Since using the Jet engine with Access is its default mode and is straightforward, this section mostly covers use of non-Access formats, which are not so obvious.

Before we look at these formats, we should discuss the availability of the Jet engine. It was included with MDAC from v1.5c until, and including, v2.5. From MDAC v2.6, the Jet engine was dropped from MDAC (which accounts for the reduction in download size from 7.5 MB to 5.2 MB). You can download a distributable version of the Jet engine from [www.microsoft.com/data/download.htm](http://www.microsoft.com/data/download.htm). Of course, if you have installed MDAC prior to v2.6, or Microsoft Access, Office, Excel, Visual Basic, or Visual C++ on your user's machine, then your user will already have the Jet engine.

There are two Jet OLE DB providers: the Jet 3.51 OLE DB Provider and the Jet 4.0 OLE DB Provider. The Jet 3.51 OLE DB Provider uses the Jet 3.51 engine and supports Access 97 databases only. If you intend to use Access 97 and not Access 2000, then you will get better performance using this OLE DB provider in most situations than using the Jet 4.0 OLE DB Provider. The Jet 4.0 OLE DB Provider supports Access 97, Access 2000, and Installable Indexed Sequential Access Method (IISAM) drivers. Installable ISAM drivers are those written specifically for the Jet engine to support access to ISAM formats such as Paradox, dBase, and text, and it is this facility that makes the Jet engine so useful and versatile. The complete list of ISAM drivers installed on your machine depends on what software you have installed on your machine. You can find this list by looking in the registry at

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM Formats
```

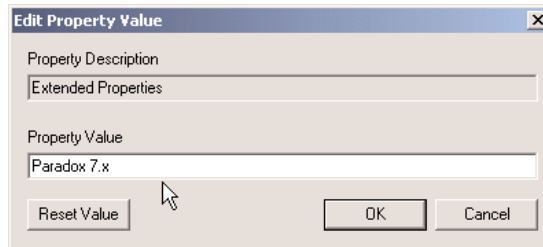
However, the Jet engine includes drivers for Paradox, dBase, Excel, text, and HTML.

### Paradox

The Jet engine, naturally, expects to be used with Access databases. To use it with any database other than Access, you need to tell it which IISAM driver to use. This is a painless process that involves setting the Extended Properties connection string argument in the connection string editor. We'll do a quick example. Add a TADOTable to a form and invoke the

connection string editor. Select the Jet 4.0 OLE DB Provider. Select the All page, locate the Extended Properties property, and double-click it to show the dialog box illustrated in Figure 16.3.

**FIGURE 16.3:**  
Setting Extended Properties



Enter **Paradox 7.x** in the Property Value as shown and click OK. Now go back to the Connection tab and enter the name of the directory containing the Paradox tables. For example you can enter

```
c:\program files\common files\borland shared\data
```

which contains Delphi's DBDEMOS Paradox tables. Unfortunately, the Browse button showing the ellipses does not react to the Extended Properties value and always expects to select a file and not a directory, and so it has little value when used with Paradox databases. Click OK on both dialog boxes and select a table in the TADOTable's TableName. Set Active to True and you are now using Paradox through ADO.

Sadly, I have some bad news for Paradox users. Under certain circumstances, you will need to install the BDE in addition to the Jet engine. It is bizarre that in a chapter dedicated to ADO we are talking about the need to install the BDE in addition to MDAC, but depending on your application this may be true. Jet 4.0 requires the BDE in order to be able to update Paradox tables, but it doesn't require the BDE just to read them. The same is true for most releases of the Paradox ODBC Driver. As disastrous as this sounds, all is not lost. Microsoft has received justified criticism on this point and has made a new Paradox IISAM available that does not require the BDE. You can get these updated drivers from Microsoft Technical Support.

#### NOTE

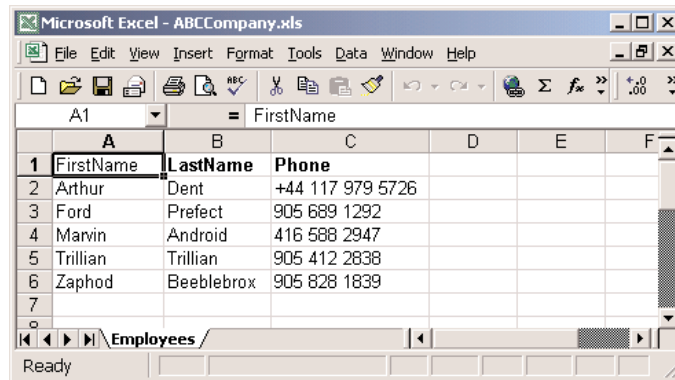
As you learn more and more about ADO, you will discover how much of ADO depends on the OLE DB provider and the DBMS (database management system) in question. You will see how the desktop databases such as Paradox and dBase have more restrictions and fewer functional features. If you are using Paradox simply because it is free, then you would be well advised to use another free database such as Access or MSDE. Alternatively, if leaving Paradox is not an option, then you should closely compare the BDE's support for Paradox with ADO's support for Paradox. In some cases, you will find the BDE's support better.

## Excel

Excel is easily accessed using the Jet OLE DB Provider. Once again, we use the Extended Properties property and set it to Excel 8.0. Assume that we have an Excel spreadsheet called ABCCompany.xls that, in Excel, looks like Figure 16.4.

**FIGURE 16.4:**

ABCCompany.xls in Excel



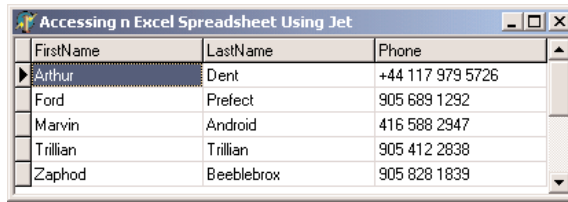
	A	B	C	D	E	F
1	FirstName	LastName	Phone			
2	Arthur	Dent	+44 117 979 5726			
3	Ford	Prefect	905 689 1292			
4	Marvin	Android	416 588 2947			
5	Trillian	Trillian	905 412 2838			
6	Zaphod	Beeblebrox	905 828 1839			
7						

Notice that the sheet is called Employees. Our mission is to open and read this file using Delphi. You can, of course, solve this problem by automating Excel with only a small knowledge of COM. However, the ADO solution is considerably easier to implement.

Ensure that your spreadsheet is not open in Excel. Add a TADODataset to a form. Set ConnectionString to use the Jet 4.0 OLE DB Provider and set Extended Properties to Excel 8.0. In the Connection tab, set the database name to the full file and path specification of the Excel spreadsheet. Close the connection string editor. The TADODataset component works by opening or executing a value in its CommandText property. This value might be the name of a table or an SQL statement or a stored procedure or the name of a file. You specify how this value is interpreted by setting the CommandType property. Set CommandType to cmdTableDirect to indicate that the value in CommandText is the name of a table and that all columns should be returned from this table. Select CommandText in the Object Inspector and you will see a drop-down arrow. Drop down the arrow and a single “table” will be displayed: Employees\$. (Excel workbooks are suffixed with a \$.) Set Active to True, add a TDataSource and a TDBGrid and connect them altogether, and you will see the Excel spreadsheet. It will be a little difficult to view in the grid because each column has a width of 255 characters. You can change this either by adding columns to the grid and changing their Width properties, or by adding persistent fields and changing their Size or DisplayWidth properties. After a little rearranging, you should see something like Figure 16.5.

**FIGURE 16.5:**

ABCCompany.xls in Delphi



FirstName	LastName	Phone
Arthur	Dent	+44 117 979 5726
Ford	Prefect	905 689 1292
Marvin	Android	416 588 2947
Trillian	Trillian	905 412 2838
Zaphod	Beeblebrox	905 828 1839

Now save your application. If you run it from the IDE, you will discover the first of the limitations of the Excel IISAM: the XLS file is opened exclusively. To run the application, you will first need to close the application that is open in the IDE and then run it from Windows Explorer. When you run the program, you will notice another limitation of this IISAM driver: you can add new rows and edit existing rows, but you cannot delete rows.

Incidentally, you could have used either `TADOTable` or `TADOQuery`, instead of `TADODataSet`, but you need to be aware of how ADO treats symbols in things like table names and field names. If you were to use a `TADOTable` and drop down the list of tables, you would see the `Employees$` table as you would expect. Unfortunately if you attempt to open the table, you will receive an error. The same is true for `SELECT * FROM Employees$` in a `TADOQuery`. The problem lies with the dollar sign in the table name. If you use characters such as dollars, dots, or, more importantly, spaces in table names or field names, then you must enclose the name in square brackets (e.g., `[Employees$]`).

## Text Files

One of the very useful IISAM drivers that comes with the Jet engine is the Text IISAM. This driver allows you to read and update text files of almost any structured format. We will start with a simple text file to get up and running and then cover the variations later. Assume we have a simple text file called `NightShift.TXT` that contains the following text:

```
CrewPerson ,HomeTown
Neo      ,Cincinnati
Trinity  ,London
Morpheus ,Milan
```

Add a `TADOTable` to a form, set its `ConnectionString` to use the Jet 4.0 OLE DB Provider, and set `Extended Properties` to `Text`. The Text IISAM considers a directory a database, so you need to enter the directory that contains the `NightShift.TXT` file as the database name. Back in the Object Inspector, drop down the list of tables in the `TableName` property. The “database” consists of all of the text files in this directory. You will notice that the dot in the filename has been converted to a hash, as in `NightShift#TXT`. Set `Active` to `True`, add a `TDataSource` and a `TDBGrid` and connect them altogether, and you will see the contents of the text file in a grid.



If your computer's settings are such that the decimal separator is a comma instead of a period (so that 1,000.00 is displayed as 1.000,00), then you will need to either change your Regional Settings (Start > Settings > Control Panel > Regional Settings > Numbers) or take advantage of SCHEMA.INI, described shortly.

Of course, the grid indicates that the widths of the columns are 255 characters. You can change these just as we did in Excel by adding persistent fields or columns to the grid and then setting the relevant width property. Alternatively you can define the structure of the text file more specifically using SCHEMA.INI.

Text files come in all shapes and sizes. Often you do not need to worry about the format of a text file because the Text IISAM takes a peek at the first 25 rows to see whether it can determine the format for itself. It uses this information and some additional information in the Registry to decide how to interpret the file and how to behave. If you have a file that doesn't match a regular format the Text IISAM can determine, then you can provide this information in the shape of SCHEMA.INI. SCHEMA.INI is an INI file located in the same directory as the text files to which it refers. It contains *schema* information, also called metadata, about any or all of the text files in the same directory. Each text file is given its own section, identified by the name of the text file, such as [NightShift.TXT].

Thereafter you can specify the format of the file, the names, types, and sizes of columns, any special character sets to use, and any special column formats (e.g., date/time, currency). Let's assume that we change our NightShift.TXT file to the following format:

```
Neo      |Cincinnati  
Trinity  |London  
Morpheus |Milan
```

In this example, the column names are not included in the text file and the delimiter is a vertical bar. An associated SCHEMA.INI file might look something like the following:

```
[NightShift.TXT]  
Format=Delimited(|)  
ColumnNameHeader=False  
Col1=CrewPerson Char Width 10  
Col2=HomeTown Char Width 30
```

Regardless of whether or not you use a SCHEMA.INI file, you will encounter two limitations of the Text IISAM: rows cannot be deleted, and rows cannot be edited.

## Importing and Exporting

The Jet engine is particularly adept at importing and exporting data. The process of exporting data is the same for each export format and consists of executing a SELECT statement with a special syntax. Let's start with an example of exporting data from the Northwind

Access database to a Paradox table. You will need an active TADOCConnection, called ADO-Connection1 in our example, that uses the Jet 4.0 OLE DB Provider to open the Northwind.mdb Access database. The following code exports the Customers table to a Paradox Customers.db file:

```
ADOConnection1.Execute('SELECT * INTO Customers ' +  
    'IN "C:\Temp" "Paradox 7.x;" FROM CUSTOMERS');
```

Let's look at the pieces of this SELECT statement. The INTO clause specifies the new table that will be created by the SELECT statement; this table must not already exist. The IN clause specifies the database to which the new table is added; in Paradox, this is a directory that already exists. The clause immediately following the database is the name of the IISAM driver to be used to perform the export. *You must include the trailing semicolon at the end of the driver name.* The FROM clause is a regular part of any SELECT statement.

All export statements follow these same basic clauses, but you will find that some IISAM drivers have differing interpretations of what a database is. I'll do another couple of examples to demonstrate the differences. Here, we export the same data to Excel:

```
ADOConnection1.Execute('SELECT * INTO Customers ' +  
    'IN "Northwind.xls" "Excel 8.0;" FROM CUSTOMERS');
```

A new Excel file called Northwind.xls is created in the application's current directory. A workbook called Customers is added, containing all of the data of the Customers table in Northwind.mdb. You can also export data to Excel by automating Excel, but if you have ever done this you will know that this ADO solution is simpler by far.

This next example exports the same data to HTML:

```
ADOConnection1.Execute('SELECT * INTO [Customers.htm] ' +  
    'IN "C:\Temp" "HTML Export;" FROM CUSTOMERS');
```

In this example, the database is the directory, as it was for Paradox but not for Excel. The table name must include the .htm extension and, therefore, it must be enclosed in square brackets. Notice that the name of the IISAM driver is "HTML Export", not just "HTML", because this driver can only be used for exporting to HTML.

The last IISAM driver we'll look at in this investigation of the Jet engine is the sister to HTML Export: HTML Import. Add a TADOTable to a form, set its ConnectionString to use the Jet 4.0 OLE DB Provider and Extended Properties to HTML Import. Set the database name to the name of the HTML file created by the export a few moments ago—that is, C:\Temp\Customers.htm. Close the connection string editors and set the TableName to Customers. Open the table and you have just imported the HTML file! Bear in mind, though, that the name of this IISAM driver is "HTML Import", not just "HTML". If you attempt to update the data in any way, you'll receive an error because this driver is intended for import only. Finally, if you create your own HTML files containing tables and want to

open these tables using this driver, then remember that the name of the table is the value of the `CAPTION` tag of the `HTML TABLE`.

## Cursor Locations and Cursor Types

There are two properties of ADO datasets that have a fundamental impact on your application and are inextricably linked with each other: `CursorLocation` and `CursorType`. The key to a successful application and to understanding your dataset's behavior and capabilities and the performance of your application lies in understanding these two properties.

The `CursorLocation`, of type `TCursorLocation`, allows you to specify what is in control of the retrieval and update of your data. You have two choices: client (`clUseClient`) or server (`clUseServer`). Your choice affects your dataset's functionality, performance, and scalability.

A client cursor is managed by the ADO Cursor Engine. This engine is an excellent example of an OLE DB service provider: it provides a service to other OLE DB providers. The ADO Cursor Engine manages the data from the client side of the application. All data in the result set is retrieved from the server to the client when the dataset is opened. Thereafter, the data is held in memory and updates and manipulation are managed by the ADO Cursor Engine. One benefit is that manipulation of the data, after the initial retrieval, is considerably faster. Furthermore, as the manipulation is performed in memory, the ADO Cursor Engine is more versatile than most server-side cursors and offers facilities that cannot be reproduced by server-side cursors. I'll examine these benefits later, as well as other technologies that depend on client-side cursors such as disconnected and persistent recordsets.

A server-side cursor is managed by the DBMS. In a client/server database such as SQL Server, Oracle, or InterBase, this means that the cursor is managed physically on the server. In a desktop database such as Access or Paradox, the "server" location is simply a logical location, as the database is running on the desktop. Server-side cursors are often faster to load than client-side cursors because not all of the data is transferred to the client when the dataset is opened. This also makes them more suitable for very large result sets where the client has insufficient memory to hold the entire result set in memory. Often you can determine what kinds of features will be available to you with each cursor location by thinking through how the cursor works. A good example of how features determine cursor type is locking, which I will cover in more detail later. To place a lock on a record requires a server-side cursor, because there must be a conversation between the application and the DBMS.

Another issue that will affect your choice of cursor location is scalability. Server-side cursors are managed by the DBMS; in a client/server database, this will be located on the server. As more and more users use your application, the load on the server increases with each server-side cursor. A greater workload on the server means that the DBMS becomes a bottleneck

faster, so the application is less scalable. You can achieve better scalability by using client-side cursors. The initial hit on opening the cursor is often heavier, because all the data is transferred to the client, but the maintenance of the open cursor can be lower. As you can see, many conflicting issues are involved in choosing the correct cursor location for your datasets.

Your choice of cursor location directly affects your choice of cursor type. To all intents and purposes there are four cursor types, but I will digress for a moment to explain why there is one unused value. There is a cursor type that means “unspecified.” Many values in ADO signify an unspecified value, and I will cover them all here and explain why you won’t have much to do with them. They exist in Delphi because they exist in ADO. ADO was mostly designed for Visual Basic and C programmers. In these languages, you use the objects directly without any of the assistance that dbGo provides. As such, you can create and open *recordsets*, as they are called in ADO-speak, without having to specify every value for every property. The properties for which a value has not been specified have an unspecified value. However, in dbGo we use components. These components have constructors, and these constructors initialize the properties of the components. So from the moment you create a dbGo component, it will usually have a value for each and every property. The consequence is that we have little need for the unspecified values in many enumerated types.

Back to the cursor types. Cursor types, of type `TCursorType`, affect how your data is read and updated. There are four choices: forward-only, static, keyset, and dynamic. Before we get too involved in all of the permutations of cursor locations and cursor types, you should be aware that there is only one cursor type available for client-side cursors: the static cursor. All other cursor types are only available to server-side cursors. I’ll return to the subject of cursor type availability after we have looked at the various cursor types, in increasing order of expensiveness.

The least expensive cursor type, and therefore the type with the best possible performance, is the forward-only cursor, which, as the name implies, will let you navigate forward. The cursor reads the number of records specified by `CacheSize` (default of 1) and each time it runs out of records, it reads another `CacheSize` set. Any attempt to navigate backward through the result set beyond the number of records in the cache will result in an error.

Knowing how a forward-only cursor works should help you to understand why they do not support bookmarks. A bookmark normally allows your dataset to navigate by jumping to a selected row. Because you can only travel forward through a forward-only cursor and any bookmark placed will be behind the current cursor position, bookmarks are not supported. As such a forward-only cursor is not suitable for use in the user interface where the user can control the direction through the result set. However, it is eminently suitable for batch operations and reports, because these situations start at the top of the result set and work progressively toward the end, then the result set is closed.

A static cursor works by reading the complete result set and providing a window of `CacheSize` records into the result set. As the complete result set has been retrieved by the server, you can navigate both forward and backward through the result set. However, in exchange for this facility, the data is static—that is, updates, insertions, and deletions made by other users cannot be seen because the cursor’s data has already been read.

A keyset cursor is best understood by breaking *keyset* down into its two words *key* and *set*. *Key*, in this context, refers to an identifier for each row. Often this will be a primary key. A keyset cursor, therefore, is a set of keys. When the result set is opened, the complete list of keys for the result set is read. If, for example, the dataset was a query like `SELECT * FROM CUSTOMERS`, then the list of keys would be built from `SELECT CUSTID FROM CUSTOMERS`. This set of keys is held until the cursor is closed. When the application requests data, the OLE DB provider reads the rows using the keys in the set of keys. Consequently, the data is always up to date. If another user changes a row in the result set, then the changes will be seen when the data is reread. However, the set of keys, itself, is static; it is read only when the result set is first opened. So if another user adds new records, these additions will not be seen. Deleted records become inaccessible, and changes to primary keys (you don’t let your users change primary keys, do you?) are also inaccessible.

The last, and most expensive, cursor type is dynamic. A dynamic cursor is almost identical to a keyset cursor. The sole difference is that the set of keys is reread when the application requests data that is not in the cache. As the default for `TADODataset.CacheSize` is 1, this is very frequent. You can imagine the additional load this places on the DBMS and the network and why this is the most expensive cursor. However, the result set can see and respond to the additions and deletions made by other users.

## Ask and Ye Shall Not Receive

Now that we know all about cursor locations and cursor types, a word of warning: not all combinations of cursor location and cursor type are possible. Usually, this is a limitation imposed by the DBMS and/or the OLE DB provider as a result of the functionality and architecture of the DBMS. For example, client cursors always force the cursor type to static. You can see this for yourself. Add a `TADODataset` to a form, set its `ConnectionString` to any database, set `ClientLocation` to `clUseCursor` and `CursorType` to `ctDynamic`. Now set `Active` to `True` and keep your eye on the `CursorType`; it changes to `ctStatic`. We learn an important lesson from this example:

What you ask for is not necessarily what you get.

Always check your properties after opening a dataset for what you think you “know” is true.

Each OLE DB provider will make different changes according to different requests and circumstances, but to give you a rough idea of what you can expect here are a few examples. The Jet 4.0 OLE DB Provider changes most cursor types to keyset. The SQL Server OLE DB Provider often changes keyset and static to dynamic. The Oracle OLE DB Provider changes all cursor types to forward-only. The ODBC OLE DB Provider makes various changes according to the ODBC driver in use.

## **RecordCount = -1**

Armed with all of this knowledge about cursors, we can explain why ADO datasets sometimes return -1 for their RecordCount. A forward-only cursor cannot know how many records are in the result set until it reaches the end, so it returns -1 for the RecordCount. A static cursor always knows how many records are in the result set, because it reads the entire set when it is opened, so it returns the number of records in its result set. A keyset cursor also knows how many records are in the result set, because it has to retrieve a fixed set of keys when the result set is opened, so it also returns a useful value for RecordCount. A dynamic cursor does not reliably know how many records are in the result set, because it is regularly rereading the set of keys, so it returns -1. You could, of course, avoid using RecordCount altogether and execute `SELECT COUNT(*) FROM tablename`, but this will be an accurate reflection of the number of records in the database, which is not necessarily the same as the number of records in the dataset.

## **Client Indexes**

One of the many benefits of client-side cursors is the ability to create local, or *client*, indexes. You can try this out for yourself. Assuming that you have an ADO client-side dataset for the Northwind's Customer table, which has a grid attached to it, set the dataset's `IndexFieldNames` property to `CompanyName`. Immediately the grid will show that the data is in `CompanyName` order. There is an important point to make here: In order to index the data, ADO did not have to reread the data from its source. The index was created from the data in memory. This means not only is the creation of the index just about as fast as it could possibly be, but the network and the DBMS are not overloaded with transferring the same data over and over in different orders. The `IndexFieldNames` property has more potential. Set it to `Country;CompanyName` and you will see the data ordered first by country and then, within country, in company name order. Now set `IndexFieldNames` to `CompanyName DESC`. Be sure to write `DESC` in capitals and not "desc" or "Desc". I'm sure you won't be surprised to see that the data is now sorted in descending order.

This simple but powerful feature allows us to solve one of the great bugbears of database developers. From time to time users seem to ask the inevitable, and quite reasonable, question, “Can I click the columns of the grid to sort my data?” There are three traditional answers to this question:

- “Yes. I can replace all of my grids with non–data-aware controls such as `TListView` that have the sorting built into the control. Of course, I lose all of the benefits of data-aware controls with this solution.”
- “Yes. I can trap the `TDBGrid`’s `OnTitleClick` event and rebuild the SQL `SELECT` statement to include an appropriate `ORDER BY` clause and then reissue the `SELECT` statement. Of course, this will mean requerying exactly the same data as I already have just to get it in a different order, but the user is always right.”
- “No. I admit that this would be a cool feature, but the extra load on the DBMS and the network is antisocial to other users.”

Sadly, none of the above are acceptable answers. Client indexes to the rescue! Add the following `OnTitleClick` event to the grid:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);  
begin  
    if ADODataset1.IndexFieldNames = Column.Field.FieldName then  
        ADODataset1.IndexFieldNames := Column.Field.FieldName + ' DESC'  
    else  
        ADODataset1.IndexFieldNames := Column.Field.FieldName  
end;
```

This simple event checks to see whether the current index is built on the same field as the column. If it is, then a new index is built on the column but in descending order. If not, then a new index is built on the column. When the user clicks the column for the first time, it is sorted in ascending order, and when it is clicked for the second time, it is sorted in descending order. You could extend this to allow the user to `Ctrl-click` several column titles to build up more complicated indexes. Of course, all of this can be achieved using `TClientDataSet`, but that solution is not as elegant for two reasons: descending indexes must be built from scratch (because `TClientDataSet` does not support the `DESC` keyword) and existing ascending indexes cannot be changed to descending indexes (they must be deleted and rebuilt).

## Cloning

ADO is crammed full of features. You can argue that “feature-rich” can translate into “footprint-rich,” but it also translates into more powerful and reliable applications. One such powerful feature is cloning. A cloned recordset is a new recordset that has all of the same properties

as the original from which it is cloned. First, I'll explain how you can create and use a clone, and then I'll explain why they are so useful.

You can clone a recordset, or, in dbGo-speak, a dataset, using the `Clone` method. You can clone any ADO dataset, but we will use `TADOTable` in this example. Add a `TADOTable` to a form and set its `ConnectionString` to use any OLE DB provider that returns rectangular data (e.g., the Jet or SQL Server OLE DB providers). Set its `TableName` to any table and open the table. Add a `TDataSource` and a `TDBGrid` to allow you to view the table. Now add a second `TADOTable` and a button with the following code:

```
ADOTable2.Clone(ADOTable1);
```

This line clones `ADOTable1` and assigns the clone to `ADOTable2`. If you add another `TDataSource` and `TDBGrid` to show `ADOTable2`, you will see a second view of the data. The two datasets have their own record pointers and other status information, so the clone does not interfere with its original copy.

This behavior makes them ideal for *black box programming*. This term comes from an old story, in which you could ask the black box any question at all and it would guarantee to provide an answer. You couldn't see inside the black box and you didn't know how it worked, just that it did. The black box did what it was supposed to and no more. It is this last part that is so essential to programming: functions and procedures do *no more than they are supposed to*. Sometimes this is also referred to as having “zero side effects.” So in this example, the `CountSelected` function attempts to count all of the rows where the `Selected` field is `True`:

```
function CountSelected(
ADODataset: TCustomADODataset): integer;
var
    Bookmark: TBookmark;
begin
    Result := 0;
    Bookmark := ADODataset.GetBookmark;
    try
        ADODataset.First;
        while not ADODataset.EOF do
            begin
                if ADODataset.FieldByName('Selected').AsBoolean then
                    Inc(Result);
                ADODataset.Next;
            end;
        ADODataset.GoToBookmark(Bookmark);
    finally
        ADODataset.FreeBookmark(Bookmark);
    end;
end;
```



At first sight, this function appears to be a black box function. It dutifully saves the current row position of the dataset using a bookmark and restores the position before returning, so that the net effect of moving the row position is zero. This is a standard approach to black box programming: if you change something in your routine that you are not supposed to change, then you must restore it again afterward.

Sadly, this routine is not black box. The movement of the row pointer will be observed by any data-aware controls, and the user interface will be updated by every row movement. A better solution to this problem is to use cloning:

```
function CountSelected(ADODataset: TCustomADODataset): integer;  
var  
    Clone: TADODataset;  
begin  
    Result := 0;  
    Clone := TADODataset.Create(nil);  
    try  
        Clone.Clone(ADODataset);  
        Clone.First;  
        while not Clone.EOF do  
            begin  
                if Clone.FieldName('Selected').AsBoolean then  
                    Inc(Result);  
                Clone.Next;  
            end;  
        Clone.Close;  
    finally  
        Clone.Free;  
    end;  
end;
```

The new clone will not interfere with its original in any way, making it ideal for black box programming. In particular, note that the closing of the clone does not close the original or other clones. In fact, not even the closing of the original closes its clones. I will use cloning later in this chapter, but before we close this subject, there are two points worth mentioning.

---

**WARNING**

A recordset must support bookmarks in order to be cloned, so forward-only and dynamic cursors cannot be cloned. You can determine whether a recordset supports bookmarks using the Supports method (e.g., `ADOTable1.Supports([coBookMark])`).

---

**TIP**

One of the useful side effects of clones is that the bookmarks created by one clone are usable by all other clones.

# Transaction Processing

*Transaction processing* allows developers to group together individual updates to a database into a single logical unit of work. The benefit is that a database engine can be told to accept or reject the complete unit of work as a single entity. The facility is present in nearly all of today's DBMSs, because it's essential for maintaining the integrity of the database. The classic example used to illustrate the need for transaction processing is the transfer of money to and from a bank account. The movement of money from one account into another consists of two steps: the removal of money from one account and the addition of the same money into another account. The entire transaction must either wholly succeed or wholly fail in order for integrity to be maintained. If only one half of the transaction succeeds, the money will either have been created (by adding money to one account while failing to remove it from the other) or destroyed (by removing money from one account while failing to add it to the other). Database programming is full of examples that require transaction processing, and in this section we will see how ADO handles this subject.

ADO's transaction processing support is controlled using a `TADOConnection`. The following list summarizes the relevant methods:

<b>TADOConnection Method</b>	<b>Description</b>	<b>TDatabase Equivalent</b>
<code>BeginTrans</code>	Begins a transaction	<code>StartTransaction</code>
<code>CommitTrans</code>	Commits a transaction	<code>Commit</code>
<code>RollbackTrans</code>	Rolls back a transaction	<code>Rollback</code>

To investigate ADO's transaction processing support, we will build a simple test program. We will use this test program to investigate the different levels of transaction processing support offered by different OLE DB providers and different databases.

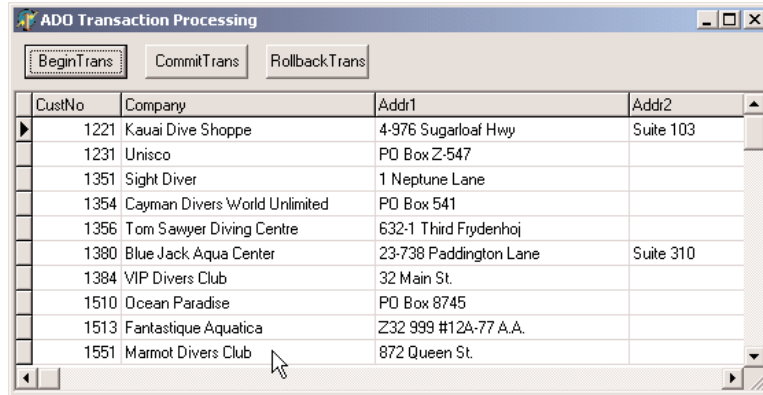
Create a new application and add a `TADOConnection`. Set the `ConnectionString` to use the ODBC OLE DB Provider and the `DBDEMOS Paradox` data supplied with Delphi. You might find it easier to first create an ODBC Data Source, say `DBDEMOS`, and refer to that. Set the `LoginPrompt` to `False` and `Connected` to `True`. Add a `TADOTable`, set `Connection` to `ADOConnection1`, `TableName` to `Customers`, and `Active` to `True`. Add a `TDataSource` and a `TDBGrid` and connect them so that the `Customers` table is shown in the grid. Add three buttons to the top of the form to execute each of the following commands:

```
ADOConnection1.BeginTrans;  
ADOConnection1.CommitTrans;  
ADOConnection1.RollbackTrans;
```

The running application should look something like Figure 16.6.

**FIGURE 16.6:**

ADO transaction processing



Click the `BeginTrans` button and you will receive an error indicating that the DBMS or the OLE DB provider is not capable of beginning a transaction. The problem lies with the Paradox ODBC Driver, which simply doesn't support transaction processing. You can find out what level of transaction processing support you have using the connection's `Transaction DDL` dynamic property—for example,

```
if ADOConnection1.Properties['Transaction DDL'].Value > DBPROPVAL_TC_NONE then
    ADOConnection1.BeginTrans;
```

The `DBPROPVAL_TC_NONE` constant comes from `OLEDB.PAS` along with several others like it. The related constants indicate that transaction support is available for Data Manipulation Language (DML) statements, meaning that SQL `UPDATE`, `INSERT`, and `DELETE` statements will all be included in transactions. ADO updates databases using SQL, so all of the `dbGo` components' DML methods (e.g., `Append`, `Post`, `Delete`) are also included in this list. The difference between the `DBPROPVAL_TC` constants is the level of support for Data Definition Language (DDL) commands included in a transaction. DDL is the set of SQL commands that alter the structure of the database and allow developers to perform actions such as adding and deleting columns and adding new tables. Given the nature of these commands, it is unlikely that you would want to mix DDL commands in a transaction with DML commands, so your interest in the `Transaction DDL` dynamic property might not extend beyond the `DBPROPVAL_TC_NONE` constant. If you do intend to include DDL commands in a transaction, you should check the other constants to see how the DDL will be treated by the DBMS. Some DBMSs will ignore the DDL, some will generate an exception if it is included in a transaction, some will cause certain DDL commands to lock a table, and others will automatically commit a transaction when the DDL is executed.

Let's get back to our test application. Change the `TADOConnection`'s `ConnectionString` to use the Jet 4.0 OLE DB Provider and set its `Extended Properties` (in the `All` tab) to

Paradox 7.x. Set the database name to Delphi's DBDEMOS directory (c:\program files\common files\borland shared\data). Save the connection string and reopen the TADOTable. You should see no difference from the previous example. Now run the application and click the BeginTrans button. Success? Unfortunately, no. This is a cruel, cruel trick on the part of the Jet 4.0 OLE DB Provider. Make a few changes and click the RollbackTrans button; close the application down and restart it. You will see that the changes that you made were permanent and your act of rolling them back made no difference at all. As I said, it is a cruel trick, because transactions on a Paradox database cannot be rolled back, and this makes them rather useless. If you read through the section in this chapter on the Jet engine and, in particular, the section on Paradox, you will recall that I mentioned that the support for Paradox is lower than most developers would like. Transaction processing is one such example.

Now let's look at how Access handles transaction processing. Create an ODBC System Data Source Name using the ODBC Manager. Call it Northwind DSN, use the Microsoft Access ODBC driver, and set the database to Northwind.mdb. In Delphi, change the TADO-Connection to use the ODBC OLE DB Provider and the new Northwind DSN. Now save the connection string, change the TADOTable's TableName from Customer to Customers, and open it. Run the application and click the BeginTrans button. At last, transaction processing that works. You can make changes to the Northwind database and roll back the changes, and they will be rolled back! However, something you cannot do is start a transaction within a transaction. Try clicking the BeginTrans button twice and you get a "Cannot start more transactions on this session" error. This is a limitation of ODBC and not the Jet engine and is a useful example of why you should always try to locate an OLE DB provider for your DBMS instead of an ODBC driver.

## Nested Transactions

If you try the same test again but use the Jet 4.0 OLE DB Provider, you will see that you can click the BeginTrans button five times before receiving an error on the sixth attempt. Jet supports *nested transactions*, which are transactions that exist within the context of another transaction. The nested or *inner* transaction can be committed or rolled back without affecting the outcome of the *outer* transaction. Let's work through a sequence of steps to be sure of how this works:

1. Begin a transaction.
2. Change the ContactName of the Around The Horn record from Thomas Hardy to Dick Solomon.
3. Begin a nested transaction.

4. Change the ContactName of the Bottom-Dollar Markets record from Elizabeth Lincoln to Sally Solomon.
5. Roll back the inner transaction.
6. Commit the outermost transaction.

The net effect is that only the change to the Around The Horn record is permanent. If, however, the inner transaction had been committed and the outer transaction rolled back, then the net effect would be that *none* of the changes were permanent (even the changes in the inner transaction). Although we are using Access to illustrate this behavior, the behavior is the same for all OLE DB providers that support nested transactions. This, of course, leads us to an ongoing theme in our ADO exploration: The ADO documentation simply states how any given feature is supposed to work when it is fully implemented by the OLE DB provider and the DBMS, but it does not necessarily follow that it *is* implemented for all OLE DB providers and all DBMSs. In the example of nested transactions, we have seen that ODBC does not support them and that the Jet OLE DB Provider supports up to five levels of nested transactions. The SQL Server OLE DB Provider **does not** support nesting.

In SQL Server 7.0 and 2000, full nested transaction support is provided. In SQL Server 6.5, only “fake nesting” is supported, where all inner-transaction instructions are ignored. You can commit and roll back inner transactions without any effect. It is only the outermost transaction that decides whether the complete sum of all of the work is committed or rolled back.

There is another issue that you should consider if you intend to use nested transactions. TADOCConnection has a property called `Attributes`, which determines how the connection should behave when a transaction is committed or rolled back. It is a set of `TXActAttributes` that, by default, is empty. There are only two values in `TXActAttributes`: `xaCommitRetaining` and `xaAbortRetaining` (this value is often mistakenly written as `xaRollbackRetaining` because this would have been a more logical name for it). When `xaCommitRetaining` is included in `Attributes` and a transaction is committed, a new transaction is automatically started. When `xaAbortRetaining` is included in `Attributes` and a transaction is rolled back, a new transaction is automatically started. This means that if you include these values in `Attributes`, a transaction will always be in progress, because when you end one transaction another will always be started. Most programmers prefer to be in greater control of their transactions than allowing them to be automatically started, so these values are not commonly used. However, they have a special relevance to nested transactions. If you nest a transaction and

set `Attributes` to `[xaCommitRetaining, xaAbortRetaining]`, then the outermost transaction can never be ended. Consider the sequence of events:

1. An outer transaction is started.
2. An inner transaction is started.
3. The inner transaction is committed or rolled back.
4. A new inner transaction is automatically started as a consequence of the `Attributes` property.

The outermost transaction can never be ended because a new inner transaction will also be started when one ends. The conclusion is that the use of the `Attributes` property and the use of nested transactions should be considered mutually exclusive.

## Lock Types

ADO supports four different approaches to locking your data for update. In this section I will provide an overview of the four approaches, and in subsequent sections we will take a closer look. The four approaches are made available to you through the dataset's `LockType` property, of type `TLockType`, and can be `ltReadOnly`, `ltPessimistic`, `ltOptimistic`, or `ltBatchOptimistic` (there is, of course, an `ltUnspecified` but, for the reasons mentioned earlier, we are ignoring “unspecified” values).

The `ltReadOnly` value specifies that the data is read-only and cannot be updated. As such, there is effectively no locking control required because the data cannot be updated.

The `ltPessimistic` and `ltOptimistic` values offer the same “pessimistic” and “optimistic” locking control that the BDE offers. One important benefit that ADO offers over the BDE in this respect is that the choice of locking control remains yours. If you use the BDE, the decision to use pessimistic or optimistic locking is made for you by the BDE driver you use. If you use a desktop database such as dBase or Paradox, then the BDE driver uses pessimistic locking; if you use a client/server database such as InterBase, SQL Server, or Oracle, the BDE driver uses optimistic locking.

## Pessimistic Locking

The words *pessimistic* and *optimistic* in this context refer to the developer's expectations of conflict between user updates. Pessimistic locking assumes that there is a high probability that users will attempt to update the same records at the same time and that a conflict is likely. In order to prevent such a conflict, the record is locked when the edit begins. The record lock is maintained until the update is completed or cancelled. A second user who

attempts to edit the same record at the same time will fail in their attempt to place their record lock and will receive a “Could not update; currently locked” exception.

This approach to locking will be familiar to developers who have worked with desktop databases such as dBase and Paradox. The benefit is that the user knows that if they can begin editing a record, then they will succeed in saving their update. The disadvantage of pessimistic locking is that the user is in control of when the lock is placed and when it is removed. If the user is skilled with the application, then this could be as short as a couple of seconds. However, in database terms a couple of seconds is an eternity. At the worst end of the scale, the user can begin an edit and go to lunch, and the record could be locked until the user returns. As a consequence of this, most proponents of pessimistic locking guard against this eventuality by using a `TTimer` or other such device to time out any locks after a certain amount of keyboard and mouse inactivity.

Another problem with pessimistic locking is that it requires a server-side cursor. Earlier we looked at cursor locations and saw that they have an impact on the availability of the different cursor types. Now we can see that cursor locations also have an impact on locking types. Later in this chapter we will see more benefits of client-side cursors, and if you choose to take advantage of these benefits, then you’ll be unable to use pessimistic locking.

Pessimistic locking is one of the areas of `ADOExpress/dbGo` that changed in Delphi 6. This section describes the way pessimistic locking works in version 6 (which is now the same as for `ADO`). Create a new application and add a `TADODataset`. Set its `ConnectionString` to use either the Jet or SQL Server Northwind database. Set its `CommandType` to `cmdTable` and `CommandText` to `Customers`. As we will be using pessimistic locking, we must set the `CursorLocation` to `clUseServer` and `LockType` to `ltPessimistic`. Finally, set `Active` to `True`. Add a `TDataSource` and a `TDBGrid`, connect them altogether, and ensure that the grid is aligned to client.

Now for the test. Run the application and begin editing a record. Using Windows Explorer, run a second copy of the same application and attempt to edit the same record; you will fail because the record is locked by another user.

If you were using `ADOExpress` in Delphi 5, the attempt to edit the same record at the same time would have succeeded, because `ADOExpress` in version 5 did not lock the record at the beginning of the edit. The work-around for this problem in Delphi 5 involved creating a clone of the original recordset and forcing a record lock in the clone before an edit, and releasing the record lock before the actual post or when the edit was cancelled.

## Jet Page and Row Locking

This section demystifies some of the issues surrounding locking in Microsoft's Jet engine. If you have no interest in Access as a database, you can safely skip over this section.

The Jet 4.0 OLE DB Provider, which you will recall is used primarily for Access 2000 databases although it can also be used with Access 97 databases, supports both page-level and row-level locking. The Jet 3.5 OLE DB Provider, which is used solely for Access 97 databases, supports page-level locking only. A *page* is a length of data. The page in question completely contains the record for which the lock is required. In Jet 3.5, a page is 2 KB in length, and in Jet 4.0 it is 4 KB. It is unlikely that a record is exactly 2 KB or 4 KB, so the locking of a single record usually includes the locking of one or more subsequent records. Clearly this locks additional records unnecessarily, which is a disadvantage of page locking and is the main reason why Jet 4.0 offers a choice of page or row locking.

Row locking, sometimes referred to as Alcatraz in the Jet engine, allows a single row to be locked individually with no additional space locked. As such, this solution provides the least lock contention. The path to understanding locking in Jet lies in Jet's dynamic properties. Most of your control over Jet locking is provided by the ADO Connection object's dynamic properties. The first and most important dynamic property is Jet OLEDB:Database Locking Mode, which can be revealed in a TADODConnection as follows:

```
ShowMessage(ADOConnection1.Properties[
    'Jet OLEDB:Database Locking Mode'].Value);
```

By default this value is 1, which means that the connection will allow recordsets to choose between row locking and page locking. The only alternative value is 0, which forces page locking. However, in order to ensure that all users use the same locking mechanism, the first user who opens the database dictates the locking mode used by all users. This mode remains in force until all users have disconnected from the database.

The second part of the locking jigsaw is the recordset. The recordset itself can specify its locking mode, using the Jet OLEDB:Locking Granularity dynamic property. By default this value is 2, which indicates that it should use row-level locking. Setting this value to 1 indicates that the recordset should use page locking. The Jet OLEDB:Locking Granularity dynamic property is ignored if the connection's Jet OLEDB:Database Locking Mode is not 1. You will have to use the `OnRecordsetCreate` event added in Delphi 6 to set the dynamic property, because it can only be set when the recordset is created but closed. (`OnRecordsetCreate` was not yet functional as this book went to press. Also, if you are using ADOExpress in Delphi 5, you will have to modify the source code of `TCustomADODataset.OpenCursor` to add in your own `OnRecordsetCreate` event immediately after the recordset is created.) Thus, so far, Jet 4.0 uses row-level locking by default. This is true but there is an extra twist to add to this tale.



Row-level locking is only the default for recordsets. It is not the default for SQL you execute directly, or what BDE developers refer to as “non-passthrough SQL.” For SQL statements you write yourself and execute directly, the default locking mode is still page locking, even using Jet 4.0.

Having gone through these trials and tribulations you will be pleased to learn that Jet provides a significant level of control over its locking facilities. Once again, this control is offered by dynamic properties and is available through TADOConnection. The following list shows the relevant dynamic properties:

<b>Property</b>	<b>Description</b>
Jet OLEDB:Lock Delay	Milliseconds to wait before attempting to reacquire a lock (default is 0)
Jet OLEDB:Lock Retry	Number of times to retry a failed lock (default is 0)
Jet OLEDB:Max Locks Per File	Maximum number of locks that Jet can place on a database (default is 9500)
Jet OLEDB:Page Locks To Table	Number of page locks before Jet promotes to a table lock (default is 0)

## Updating JOINS

One of the reasons why people used to turn to cached updates in the BDE and, more recently, TClientDataSet, is to make an SQL JOIN updatable. Consider the following SQL equi-join:

```
SELECT * FROM Products, Suppliers
WHERE Products.SupplierID=Suppliers.SupplierID
```

This statement provides a list of products and the details of the suppliers of those products. The BDE considers any SQL JOIN to be read-only because inserting, updating, and deleting rows in a join is ambiguous. For example, should the insert of a row into the above join result in a new product and also a new supplier or just a new product?

The BDE supports cached updates, which allow the developer to resolve this ambiguity by specifying exactly what the developer wants to happen. Although the BDE's cached updates implementation is often flawed, and cached updates are now discouraged in favor of the more reliable TClientDataSet, the concept is sound.

ADO supports an equivalent to cached updates, called batch updates, which are very similar. In the next section we will take a closer look at ADO's batch updates, what they can offer you, and why they are so important. However, in this section they will not be needed to solve

the problem of updating a join for a very simple reason: in ADO, joins are naturally updatable. Place a TADOQuery on a form and set its connection string to use a Northwind database. Enter the SQL join above in its SQL property and set Active to True. Add a TDataSource and a TDBGrid and connect them altogether and run the program. Now edit one of the Product's fields and save the changes (by moving off the record). No error occurs because the update has been applied successfully. ADO has taken a more practical approach to the problem: it has made some intelligent guesses. In an ADO join, each field object knows which underlying table it belongs to. If you update a field of the Products table and post the change, then a SQL UPDATE statement is generated to update the field in the Products table. If you change a field in the Products table and a field in the Suppliers table, then two SQL UPDATE statements are generated, one for each table.

The inserting of a row into a join follows a similar behavior. If you insert a row and enter values for the Products table only, then a SQL INSERT statement is generated for the Products table. If you enter values for both tables, two SQL INSERT statements are generated, one for each table. The order in which the statements are executed is important, because the new product might relate to the new supplier, so the new supplier is inserted first.

The biggest problem with ADO's solution can be seen when a row in a join is deleted. The deletion attempt will appear to fail. The exact message you see depends on the version of ADO you are using and the DBMS, but it will be along the lines that you cannot delete the row because other records relate to it. The error message can be confusing. In our scenario, the error message implies that a product cannot be deleted because there are records that relate to the product. The error occurs whether the product has any related records or not. The explanation can be found by following through the same logic for deletions as for insertions. Two SQL DELETE statements are generated: one for the Suppliers table and then another for the Products table. Contrary to appearances, the DELETE statement for the Product table succeeds. It is the DELETE statement for the Suppliers table that fails, because the Supplier cannot be deleted while it still has dependent records.

If you are curious about the SQL statements that get generated and you use SQL Server, you can see these statements using SQL Server Profiler.

Despite understanding how this works, a better way of looking at this problem is through the user's eyes. From their point of view, when they delete a row in the grid, do they intend to delete just the product or both the product and the supplier? I would wager that 99% of users expect the former and not the latter. Fortunately you can achieve exactly this with our old friend, the dynamic property—in this case, the Unique Table dynamic property. You can specify that deletes refer to just the Products table and not to Suppliers using the following line of code:

```
ADOQuery1.Properties['Unique Table'].Value := 'Products';
```

As this value cannot be assigned at design time, the next best alternative is to place this line in the form's `OnCreate` event.

For me, updatable joins are just one of many examples of how the designers of ADO have replaced traditional problems with elegant solutions.

## Batch Updates

Batch updates are ADO's equivalent to the BDE's cached updates; they are similar in functionality, syntax, and, to some extent, implementation, with the all-important difference being that their implementation is not fundamentally flawed. The idea is the same for both database technologies: By using batch/cached updates, any changes you make to your records can be made in memory and then later the entire "batch" of changes can be submitted as one operation. There are some performance benefits to this approach, but there are more practical reasons why this technology is a necessity: the user might not be connected to the database at the time they make their updates. This would be the case in the infamous "briefcase" application, which we will return to later, but this can also be the case in Web applications that use another ADO technology, Remote Data Services (RDS).

You can enable batch updates in any ADO dataset by setting `LockType` to `ltBatchOptimistic` before the dataset is opened. In addition, you will need to set the `CursorLocation` to `clUseClient`, as batch updates are managed by ADO's cursor engine. Hereafter, changes are all made to a "delta" (i.e., a list of changes). The dataset looks to all intents and purposes as if the data has changed, but the changes have only been made in memory; they have not been applied to the database. To make the changes permanent, use `UpdateBatch` (equivalent to cached updates' `ApplyUpdates`):

```
ADODataset1.UpdateBatch;
```

(Fortunately, there is no equivalent to the cached update's `CommitUpdates` method, because the successful changes are automatically removed from the batch.) To reject the entire batch of updates, use either `CancelBatch` or `CancelUpdates`. There are many similarities in method and property names between ADO's batch updates and BDE's cached updates and `TClientDataSet.UpdateStatus`, for example, can be used in exactly the same way as for cached updates to identify records according to whether they have been inserted, updated, deleted, or unmodified. This is particularly useful for highlighting records in different colors in a grid or showing their status on a status bar. Some differences between the syntaxes are slight, such as changing `RevertRecord` to `CancelBatch(arCurrent)`. Others require more effort.

One useful cached update feature that is not present in ADO batch updates is the dataset's `UpdatesPending` property. This property is true if changes have been made but not yet applied. This is particularly useful in a form's `OnCloseQuery` event:

```
procedure TForm1.FormCloseQuery(
  Sender: TObject; var CanClose: Boolean);
begin
  CanClose := True;
  if ADODataset1.UpdatesPending then
    CanClose := (MessageDlg('Updates are still pending' #13 +
      'Close anyway?', mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

However, with a little knowledge and a little ingenuity we can implement a suitable `ADODataset1.UpdatesPending` function. The little knowledge is that ADO datasets have a property called `FilterGroup`, which is a kind of filter. Unlike a dataset's `Filter` property, which filters the data based on a comparison of the data against a condition, `FilterGroup` filters based on the status of the record. One such status is `fgPendingRecords`, which includes all records that have been modified but not yet applied. So to allow the user to look through all of the changes they have made so far, you need only execute two lines:

```
ADODataset1.FilterGroup := fgPendingRecords;
ADODataset1.Filtered := True;
```

Naturally, the result set will now include the records that have been deleted. If you try this yourself, the effect that you will see will depend on the version of dbGo you have and the patches you have applied to it. In early versions of ADOExpress, the deleted record showed the fields of the previous record. This either was confusing (if there was a previous record) or resulted in a fatal error (if there wasn't). In later versions, the fields are just left blank, which also is not very helpful because you don't know what record has been deleted.

Back to the `UpdatesPending` problem. The "little ingenuity" is the knowledge of clones, discussed earlier. The idea of the `ADODataset1.UpdatesPending` function is that it will set the `FilterGroup` to restrict the dataset to only those changes that have not yet been applied. All we need to do is to see whether there are any records in the dataset once the `FilterGroup` has been applied. If there are, then some updates are pending. However, if we do this with the actual dataset, then the setting of the `FilterGroup` will move the record pointer and the user interface will be updated. The best solution is to use a clone.

```
function ADODataset1.UpdatesPending(ADODataset: TCustomADODataset): boolean;
var
  Clone: TADODataset;
begin
  Clone := TADODataset.Create(nil);
  try
    Clone.Clone(ADODataset);
```

```
Clone.FilterGroup := fgPendingRecords;
Clone.Filtered := True;
Result := not (Clone.BOF and Clone.EOF);
Clone.Close;
finally
  Clone.Free;
end;
end;
```

In this function we clone the original dataset, set the `FilterGroup`, and check to see whether the dataset is at both beginning of file and also end of file. If it is, then no records are pending.

## Optimistic Locking

Earlier we looked at the `LockType` property and saw how pessimistic locking worked. In this section, we'll look at optimistic locking, not only because it is the preferred locking type for medium- to high-throughput transactions but also because it is the locking scheme employed by batch updates.

Optimistic locking assumes that there is a low probability that users will attempt to update the same records at the same time and that a conflict is unlikely. As such, the attitude is that all users can edit any record at any time, and we deal with the consequences of conflicts between different users' updates to the same records when the changes are saved. Thus, conflicts are considered an exception to the rule. This means that there are no controls to prevent two users from editing the same record at the same time. The first user to save their changes will succeed. The second user's attempt to update the same record might fail. This behavior is essential for briefcase applications and Web applications, where there is no permanent connection to the database and, therefore, no way to implement pessimistic locking. In contrast with pessimistic locking, optimistic locking has the additional considerable benefit that resources are consumed only momentarily and, therefore, the average resource usage is much lower, making the database more scalable.

Let's consider an example. Assume that we have a `TADODataset` connected to the `Customers` table of `Northwind`, that `LockType` is set to `ltBatchOptimistic`, and the contents are displayed in a grid. Assume that we also have a button to call `UpdateBatch`. Run the program twice and begin editing a record in the first copy of the program. Although for the sake of simplicity we will be demonstrating a conflict using just a single machine, the scenario and subsequent events are unchanged when using multiple machines. In this example I will choose the `Bottom-Dollar Markets` company in Canada and change the name to `Bottom-Franc Markets`. Save the change, move off the record to post it, and click the button to update the batch. Now, in the second copy of the program, locate the same record and change the company name to `Bottom-Pound Markets`. Move off the record and click the

button to update the batch. It will fail. As with many ADO error messages, the exact message you receive will depend not only on the version of ADO you are using but also on how closely you followed the example. In ADO 2.6, the error message is “Row cannot be located for updating. Some values may have been changed since it was last read.” This is the nature of optimistic locking. The update to the record is performed by executing the following SQL statement:

```
UPDATE CUSTOMERS SET CompanyName="Bottom-Pound Markets"
WHERE CustomerID="BOTTM" AND CompanyName="Bottom-Dollar Markets"
```

The number of records affected by this update statement is expected to be 1, because it locates the original record using the primary key and the contents of the CompanyName field as it was when the record was first read. In our example, however, the number of records affected by the UPDATE statement is 0. This can only occur if the record has been deleted, or the record’s primary key has changed, or the field that we are changing was changed by someone else. Hence, the update fails.

If our “second user” had changed the ContactName field and not the CompanyName field, then the UPDATE statement would have looked like this:

```
UPDATE CUSTOMERS SET ContactName="Liz Lincoln"
WHERE CustomerID="BOTTM" AND ContactName="Elizabeth Lincoln"
```

In our scenario, this statement would have succeeded because the other user didn’t change the primary key or the contact name.

This behavior differs from the BDE’s behavior in the same scenario. In this example, using the BDE the attempt to update the contact name would have failed because, by default, the BDE includes every field in the WHERE clause. The consequence of this is that any change to the record will fail if any other user has already changed any field, regardless of whether the changed fields are the same fields or different fields. Fortunately, both the BDE and ADO allow you to specify how you want to locate the original record: in the BDE you use the UpdateMode property, and in ADO the Update Criteria dynamic property of a dataset. The following list shows the possible values that can be assigned to this dynamic property:

<b>Constant</b>	<b>Locate Records By</b>
adCriteriaKey	Primary key columns only
adCriteriaAllCols	All columns
adCriteriaUpdCols	Primary key columns and changed columns only
adCriteriaTimeStamp	Primary key columns and a timestamp column only

The reason why the BDE and ADO differ in their behavior is that their defaults differ. The BDE’s default behavior is equivalent to ADO’s adCriteriaAllCols, whereas ADO’s default is adCriteriaUpdCols. Don’t fall into the trap of thinking that one of these settings is

better than another for your whole application. In practice, your choice of setting will be influenced by the contents of each table. Say that the Customers table has just CustomerID, Name, and City fields. In this case, the update of any one of these fields is logically not mutually exclusive with the update of any of the other fields, so a good choice for this table would be adCriteriaUpdCols (i.e., the default). If, however, the Customers table included a PostalCode field, then the update of a PostalCode field would be mutually exclusive with the update of the City field by another user (because if the city changes, then surely so should the postal code, and possibly vice versa). In this case, you could argue that adCriteriaAllCols would be a safer solution.

Another issue to be aware of is how ADO deals with errors during the update of multiple records. Using the BDE's cached updates and TClientDataSet, you can use the OnUpdateError event to handle each update error as the error occurs and resolve the problem before moving on to the next record. In ADO, you cannot establish such a dialog. You can monitor the progress and success or failure of the updating of the batch using the dataset's OnWillChangeRecord and OnRecordChangeComplete, but you cannot revise the record and resubmit it during this process as you can with the BDE and TClientDataSet. There's more: if an error occurs during the update process, the updating does not stop. It continues to the end until all updates have been applied or have failed. This can produce a rather unhelpful and blatantly incorrect error message. If more than one record cannot be updated, or the single record that failed is not the last record to be applied, then the error message in ADO 2.6 is "Multiple-step OLE DB operation generated errors. Check each OLE DB status value, if available. No work was done." The last sentence is the problem; it states that "No work was done," but this is incorrect. It is true that no work was done on the record that failed, but other records were successfully applied and their updates stand.

## Resolving Update Conflicts

As a consequence of the nature of applying updates, the approach that you need to take to update the batch is to update the batch, let the individual records fail, and then deal with the failed records once the process is over. You can determine which records have failed by setting the dataset's FilterGroup to fgConflictingRecords:

```
ADODataset1.FilterGroup := fgConflictingRecords;
ADODataset1.Filtered := True;
ADODataset1.Recordset.Resync(adAffectGroup, adResyncUnderlyingValues);
```

For each failed record, you can inform the user of three critical pieces of information about each field using the following TField properties:

Property	Description
NewValue	The value this user changed it to
CurrentValue	The new value from the database
OldValue	The value when first read from the database

Users of `TClientDataSet` will be aware of the very handy `TReconcileErrorForm` dialog, which wraps up the process of showing the user the old and new records and allows them to specify what action to take. Unfortunately, there is no ADO equivalent to this form, and `TReconcileErrorForm` has been written with `TClientDataSet` so much in mind that it is difficult to convert it for use with ADO datasets.

One last gotcha to point out when using these `TField` properties: They are taken straight from the underlying ADO `Field` objects to which they refer. This means, as is common in ADO, that you are at the mercy of your chosen OLE DB provider to support the features you hope to use. All is well for most providers, but the Jet OLE DB Provider returns the same value for `CurValue` as it does for `OldValue`. In other words, if you use Jet, you cannot determine what the other user changed the field to unless you resort to your own measures.

## Disconnected Recordsets

All this knowledge of batch updates allows us to take advantage of our next ADO feature: disconnected recordsets. A disconnected recordset is a recordset that has been disconnected from its connection. What is impressive about this feature is that the user cannot tell the difference between a regular recordset and a disconnected one; their feature sets and behavior are almost identical. To disconnect a recordset from its connection, the `CursorType` must be set to `clUseClient` and the `LockType` must be set to `ltBatchOptimistic`. You then simply tell the dataset that it no longer has a connection:

```
ADODataset1.Connection := nil;
```

Hereafter, the recordset will continue to contain the same data, support the same navigational features, and allow records to be added, edited, and deleted. The only relevant difference is that you cannot update the batch because you need to be connected to the server to update the server. To reconnect the connection (and use `UpdateBatch`):

```
ADODataset1.Connection := ADOConnection1;
```

This same feature is available to the BDE and other database technologies by switching over to `TClientDataSets`, but the beauty of the ADO solution is that you can build your entire application using `dbGo` dataset components and be unaware of disconnected recordsets. At the point that you discover this feature and want to take advantage of it, you can continue to use the same components that you always used.

So why would you want to disconnect your recordsets? For two reasons:

- To keep the total number of connections lower
- To create a briefcase application



I'll cover keeping down the number of connections here and return to briefcase applications later. Most regular client/server business applications open tables and maintain a permanent connection to their database while the table is open. However, there are usually only two reasons why you want to be connected to the database: to retrieve data and to update data. If you change your regular client/server application so that, after the table is opened and the data retrieved, then the dataset is disconnected from the connection and the connection dropped, your user will be none the wiser and the application will not need to maintain an open database connection. The following code shows the two steps:

```
ADODataset1.Connection := nil;  
ADODataset1.Connected := False;
```

The only other point at which a connection is required is when the batch of updates needs to be applied, so the update code would look like this:

```
ADODataset1.Connected := True;  
ADODataset1.Connection := ADODataset1;  
try  
    ADODataset1.UpdateBatch;  
finally  
    ADODataset1.Connection := nil;  
    ADODataset1.Connected := False;  
end;
```

If this approach were followed throughout the application, the average number of open connections at any one time would be minimal because the connections would only be open for the small amount of time that they are required. The consequence of this change is scalability; The application will be able to cope with significantly more simultaneous users than one that maintains an open connection. The downside, of course, is that the reopening of the connection can be a lengthy process on some, but not all, database engines, so the application will be slower to update the batch.

## Connection Pooling

All of this talk of dropping connections and reopening them brings us to the subject of connection pooling. Connection pooling, not to be confused with session pooling, allows connections to a database to be reused once they have been finished with. This happens automatically and, if your OLE DB provider supports it and it is enabled, you need take no action to take advantage of connection pooling. There is a single reason why you would want to pool your connections: performance. The problem with database connections is that it can take time to establish a connection. In a desktop database such as Access, this is typically a small amount of time. In a client/server database such as Oracle used on a network, this time could be measured in seconds. Given such an expensive (in performance terms) resource, it makes sense to

promote its reuse. With ADO's connection pooling enabled, ADO Connection objects are placed in a pool when the application "destroys" them. Subsequent attempts to create an ADO connection will automatically search the connection pool for a connection with the same connection string. If a suitable connection is found, it is reused; otherwise, a new connection is created. The connections themselves stay in the pool until they are reused, the application closes, or they time out. By default, connections will time out after 60 seconds, but from MDAC 2.5 onward you can set this using the HKEY\_CLASSES\_ROOT\CLSID\<ProviderCLSID>\SPTimeout registry key. The connection pooling process occurs seamlessly, without the intervention or knowledge of the developer. This process is similar to the BDE's database pooling under Microsoft Transaction Server (MTS) and COM+, with the important exception that ADO performs its own connection pooling without the aid of MTS or COM+.

By default, connection pooling is enabled on all of the MDAC OLE DB providers for relational databases (including SQL Server and Oracle) with the notable exception of the Jet OLE DB Provider. If you use ODBC you should choose between ODBC's connection pooling and ADO's connection pooling, but you should not use both. From MDAC 2.1 on, ADO's connection pooling is enabled and ODBC's is disabled.

**NOTE**

Connection pooling does not occur on Windows 95 regardless of the OLE DB provider.

To be truly comfortable with connection pooling, you will need to see the connections getting pooled and timed out. Unfortunately, there are no adequate ADO connection pool spying tools available at the time of writing, so we will use SQL Server's Performance Monitor as it can accurately spy on SQL Server database connections. Figure 16.7 is a look at SQL Server's Performance Monitor with all of the "counters" deleted except User Connections. This allows us to concentrate on the subject of connection pooling.

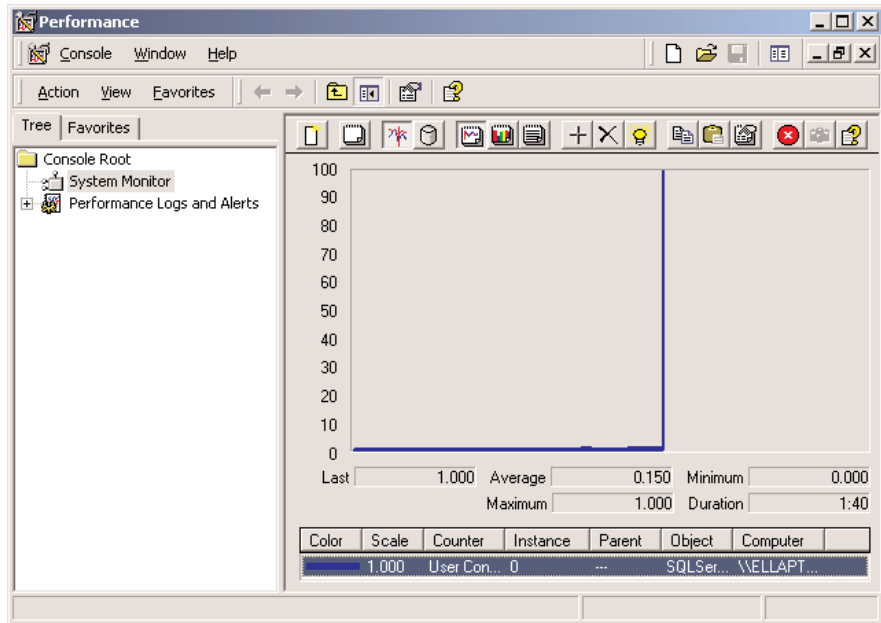
The Last field under the graph shows us the number of active connections to the database.

To see how connection pooling works, you can set up a very simple test. Create a new application and add a TADOConnection to the form. Set the ConnectionString to use the SQL Server OLE DB Provider and the Northwind database but leave Connected as False. Now add a check box with the following OnClick event:

```
procedure TForm1.CheckBox1Click(Sender: TObject);  
begin  
    ADOConnection1.Connected := CheckBox1.Checked;  
end;
```

**FIGURE 16.7:**

SQL Server's Performance Monitor



Run the program and make sure that you can see the Performance Monitor at the same time. Now click the check box to open the connection. In the Performance Monitor, you will see the connection count increase by one. Now close the application and the count immediately decreases by one, because the connection pool is destroyed with the application. Now rerun the program, check the check box, and check it a second time to close the connection. You will see that the connection count does not decrease by one. Observe Performance Monitor for a further 60 seconds, and the connection count will then decrease by one when the pooled connection times out.

You can enable or disable connection pooling either in the Registry or in the connection string. The key in the Registry is `OLEDB_SERVICES` and can be found at `HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>`. It is a bit array that allows you to disable several OLE DB services, including connection pooling, transaction enlistment, and the cursor engine. To disable connection pooling using the connection string, include  `;OLE DB Services=-2` at the end of the connection string. To enable connection pooling for the Jet OLE DB Provider, you can include  `;OLE DB Services=-1` at the end of the connection string, which enables all OLE DB services.

## Persistent Recordsets

One of the very useful features that contributes to the briefcase model is persistent recordsets. These allow you to save the contents of any recordset to a local file, which can be loaded later. Apart from aiding with the briefcase model, this feature allows developers to create true single-tier applications. It means that you can deploy a database application without having to deploy a database. This makes for a very small footprint on your client's machine.

You can “persist” your datasets using the `SaveToFile` method:

```
ADODataset1.SaveToFile('Local.ADTG');
```

This will save the data and its delta in a file on your hard disk. You can reload this file using the `LoadFromFile` method, which accepts a single parameter indicating the file to load. The format of the file is Advanced Data Table Gram (ADTG), which is a proprietary Microsoft format. It does, however, have the advantage of being very efficient. If you prefer, you can save the file as XML by passing a second parameter to `SaveToFile`:

```
ADODataset1.SaveToFile('Local.XML', pfXML);
```

However, ADO does not have its own built-in XML parser (as `TClientDataSet` does), so it must use the MSXML parser. Your user must either install Internet Explorer 5 or later or download the MSXML parser from the Microsoft Web site. If you intend to persist your files locally in XML format, be aware of a few disadvantages. First, the saving and loading of XML files is slower than the saving and loading of ADTG files. Second, ADO's XML files (and XML files in general) are significantly larger than their ADTG counterparts (XML files are typically twice as large as their ADTG counterparts). Third, ADO's XML format is specific to Microsoft, as most companies' XML implementations are. This means that the XML generated in ADO is not readable by Borland's `TClientDataSet` and vice versa. In fact, it's worse than that, because the XML generated in ADO 2.1 is incompatible with the XML in ADO 2.5. Fortunately this last problem can be overcome using Delphi 6's new TXML-Transform component, which can be used to translate between different XML structures.

If you intend to use these features solely for single-tier applications and not as part of the briefcase model, then you can save yourself a little effort by using a `TADODataset` and setting its `CommandType` to `cmdFile` and its `CommandText` to the name of the file. This will save you the effort of having to call `LoadFromFile` manually. You will, however, still have to call `SaveToFile`. In a briefcase application, however, this approach is too limiting, as the dataset can be used in two different modes.

## The Briefcase Model

Our new-found knowledge of batch updates, disconnected recordsets, and persistent recordsets allows us to take advantage of the “briefcase model.” The idea behind the briefcase model is that your users want to be able to use your application while they are out on the road. They want to take the same application that they use on the desktops in the office and use it on their laptops while on their clients’ sites. The problem with this scenario traditionally is that when your users are at their clients’ sites, they are not connected to their database server, because their database server is running on their network back at their office. Consequently, there is no data on their laptop, and the data cannot be updated anyway.

This is where that new-found knowledge comes in. Assume that the application has already been written; the user has requested this new briefcase enhancement, and you have to retrofit it into your existing application. You need to add a new option for your users to allow them to “prepare” the briefcase application. This simply consists of executing `SaveToFile` for each and every table in the database. The result is a collection of ADTG or XML files that mirror the contents of the database. These files are then copied to the laptop where a copy of the application has previously been installed.

The application needs to be sensitive to whether it is running locally or connected to the network. You can decide this either by attempting to connect to the database and seeing whether it fails, by detecting the presence of a local “briefcase” file, or by creating some flag of your own design. If the application decides it is running in briefcase mode, then it needs to use `LoadFromFile` for each table instead of setting `Connected` to `True` for the `TADOCConnections` and `Active` to `True` for the ADO datasets. Thereafter, the briefcase application needs to use `SaveToFile` instead of `UpdateBatch` whenever data is saved. Upon return to the office, there needs to be an update process where each table is loaded from its local file, the dataset is connected to the database, and the changes are applied using `UpdateBatch`. Voilà, the briefcase model.

## Deploying MDAC

MDAC, and therefore ADO, can be almost freely distributed. There are some conditions on its distribution, but these are to protect Microsoft from unreasonable behavior and it is unlikely that regular application developers will fall afoul of them. To distribute MDAC, you distribute and execute `MDAC_TYP.EXE`. You may not distribute and install MDAC components individually. If you use `InstallShield Express For Delphi`, then you will have to run `MDAC_TYP.EXE` separately from your regular `InstallShield Setup` program, because `InstallShield Express For Delphi` cannot shell out to an external program—i.e., `MDAC_TYP.EXE`. If you use an installation program that can shell out to another program, you might want to be aware of some of the parameters

you can pass to `MDAC_TYP.EXE`. The various parameters affect whether the user has to specifically accept the end user license, whether the file copy dialog is shown, whether there is an automatic reboot on completion (or the user decides to reboot or there is no reboot), and, finally, whether `MDACSET.LOG` is a summary log file or a full log file. The `/Q` parameter is a quiet, but not completely silent, setup. A truly silent setup can be achieved with `/Q:A /C: "setup /QNT"` parameters. See the Platform SDK for a complete list of setup parameters. Although I have not tested installing every version of MDAC on top of every other version of MDAC, my experience is that, despite the progress bar indicating the successful progress of the installation, an earlier version of MDAC does not overwrite a later version of MDAC.

In addition to installing MDAC, you will also need to install DCOM if the target is Windows 95.

One invaluable tool that you should add to your toolbox is Component Checker. This is available for free download from [www.microsoft.com/data/download.htm](http://www.microsoft.com/data/download.htm). Component Checker is the most accurate method of determining which version of MDAC is installed on a machine. It scans for all ADO, OLE DB, and ODBC files and gets their version numbers. It can compare all of these version numbers against its own internal database of correct version numbers for each release of MDAC. At the end of its analysis, it reveals the version of MDAC that most closely matches the files installed on a machine. It is also useful as a kind of “REGEDIT For MDAC,” as it reports on all of the MDAC registry information using a considerably more relevant user interface than REGEDIT. Finally, it is the only safe way of removing MDAC from a machine.

## ADO.NET

ADO.NET is part of Microsoft’s new dotNet (or “.NET”) architecture—their redesign of application development tools to better suit the needs of Web development. At the time of writing, Visual Studio.NET was in beta, so this section has been included solely as a means to give you an idea of where ADO is heading.

ADO.NET is a revolution of ADO. It looks at the problems of Web development and addresses shortcomings of ADO’s solution. The problem with ADO’s solution is that it is based on COM. For one- and two-tier applications, COM imposes few problems, but in the world of Web development it is unacceptable as a transport mechanism. COM suffers from three main problems for use in Web development: it (mostly) runs only on Windows; the transmission of recordsets from one process requires COM marshalling; and COM calls cannot penetrate corporate firewalls. ADO.NET’s solution to all of these problems is to use XML.

*Continued on next page*

Some other redesign issues focus around breaking up the ADO recordset into separate classes. The resulting classes are adept at solving a single problem instead of multiple problems. For example, the ADO.NET class currently called `DataSetReader` is very similar to a read-only, forward-only server-side recordset and, as such, is best suited to reading a result set very quickly. A `DataTable` is most like a disconnected, client-side recordset. A `DataRelation` has similarities with the `MSDataShape` OLE DB Provider. So you can see that your knowledge of how ADO works is of great benefit in understanding the basic principles of ADO.NET.

If you wish to experiment with ADO.NET in Delphi before direct support is added to Delphi, then you will need to give Delphi access to the ADO.NET classes, which are called “managed” classes and are based on a new run-time environment called the Common Language Runtime (CLR). These classes are not COM classes and, as such, Delphi cannot normally access them. However, Visual Studio.NET includes a utility called `REGASM.EXE`, which takes any “assembly” (library of managed classes) and creates a COM type library interface to the managed classes. As Delphi can easily access COM classes, simply import the resulting type library into Delphi and use the classes as if they were COM classes.

## What’s Next?

This chapter described ActiveX Data Objects (ADO) and `dbGo`, the set of Delphi components for accessing the ADO interfaces. You’ve seen how to take advantage of Microsoft Data Access Components (MDAC) and various server engines, and I’ve described some of the benefits and hurdles you’ll encounter in using ADO.

The next chapter will take you into the world of Delphi’s `DataSnap` architecture, for developing custom client and server applications in a three-tier environment.

# Multitier Database Applications with DataSnap

- Logical three-tier architecture
- The technical foundation of DataSnap
- The connection protocols and the data packets
- Delphi's support components (client-side and server-side)
- The connections broker and other new Delphi 6 features



**L**arge companies often have broader needs than applications using local database and SQL servers can meet. In the past few years, Borland Software Corporation has been addressing the needs of large corporations, and it even temporarily changed its own name to Inprise to underline this new enterprise focus. The name was changed back to Borland, but the focus on enterprise development remains.

Delphi is targeting many different technologies: three-tier architectures based on Windows NT and DCOM, CORBA architectures based on NT and Unix servers, TCP/IP and socket applications, and—most of all—SOAP- and XML-based Web services. This chapter focuses on database-oriented multitier architectures, while XML-oriented solutions will be discussed in Chapter 23, “XML and SOAP.”

Even though I haven't yet discussed COM and sockets (covered in Chapters 19 to 21), in this chapter we'll build multitier architectures based on those technologies. As we'll use high-level Delphi support, not knowing the details of some of the foundations should not create any problem. I'll concentrate more on the programming aspects of these architectures than on installation and configuration (the latter aspects are subject to change across different operating systems and are too complex to cover thoroughly).

Before proceeding, I should emphasize two important elements. First, the tools to support this kind of development are available only in the Enterprise version of Delphi; and second, you'll have to pay a license fee to Borland in order to deploy the necessary server-side software for DataSnap. This second requirement makes this architecture cost-effective mainly for large systems (that is, servers connected to dozens or even hundreds of clients). The license fee is only required for deployment of the server application and is a flat fee for each server you deploy to (regardless of the number of clients that will connect). The license fee is not required for development or evaluation purposes.

---

**NOTE**

You spend money on the DataSnap license, but you might save on the SQL server client licenses. When SQL server licenses were based on the number of connections, companies have saved tens of thousands of dollars in those licenses by connecting the hundreds of clients to a few instances of the DataSnap server, using few connections with the SQL server. Nowadays, the licenses for most SQL servers are based on the number of users who connect to the database, not the number of connections active at each time, so this kind of savings doesn't always apply.

## One, Two, Three Levels

Initially, database PC applications were client-only solutions: the program and the database files were on the same computer. From there, adventuresome programmers moved the database files onto a network file server. The client computers still hosted the application software and the entire database engine, but the database files were now accessible to several users at the same time. You can still use this type of configuration with a Delphi application and Paradox files (or, of course, Paradox itself), but the approach was much more widespread just few years ago.

The next big transition was to client/server development, embraced by Delphi since its first version. In the client/server world, the client computer requests the data from a server computer, which hosts both the database files and a database engine to access them. This architecture downplays the role of the client, but it also reduces its requirements for processing power on the client machine. Depending on how the programmers implement client/server, the server can do most (if not all) of the data processing. In this way, a powerful server can provide data services to several less powerful clients.

Naturally, there are many other reasons for using centralized database servers, such as the concern for data security and integrity, simpler backup strategies, central management of data constraints, and so on. The database server is often called a SQL server, because SQL is the language most commonly used for making queries into the data, but it may also be called a DBMS (database management system), reflecting the fact that the server provides tools for managing the data, such as support for backup and replication.

Of course, some applications you build may not need the benefits of a full DBMS, so a simple client-only solution might be sufficient. On the other hand, you might need some of the robustness of a DBMS system, but on a single, isolated computer. In this case, you can use a local version of a SQL server, such as InterBase. Traditional client/server development is done with a two-tier architecture. However, if the DBMS is primarily performing data storage instead of data- and number-crunching, the client might contain both user interface code (formatting the output and input with customized reports, data-entry forms, query screens, and so on) and code related to managing the data (also known as *business rules*). In this case, it's generally a good idea to try to separate these two sections of the program and build a logical three-tier architecture. The term *logical* here means that there are still just two computers (that is, two physical tiers), but we've now partitioned the application into three distinct elements.

Delphi 2 introduced support for a logical three-tier architecture with data modules. As you'll recall, a *data module* is a nonvisual container for the data access components of an application, but it often includes several handlers for database-related events. You can share a single data

module among several different forms and provide different user interfaces for the same data; there might be one or more data-input forms, reports, master/detail forms, and various charting or dynamic output forms.

The logical three-tier approach solves many problems, but it also has a few drawbacks. First, you must replicate the data-management portion of the program on different client computers, which might hamper performance, but a bigger issue is the complexity this adds to code maintenance. Second, when multiple clients modify the same data, there's no simple way to handle the resulting update conflicts. Finally, for logical three-tier Delphi applications, you must install and configure the database engine (if any) and SQL server client library on every client computer.

The next logical step up from client/server is to move the data-module portion of the application to a separate server computer and design all the client programs to interact with it. This is exactly the purpose of remote data modules, which were introduced in Delphi 3. Remote data modules run on a server computer—generally called the application server. The application server in turn communicates with the DBMS (which can run on the application server or on another dedicated computer). Therefore, the client machines don't connect to the SQL server directly, but indirectly via the application server.

At this point there is a fundamental question: Do we still need to install the database access software? The traditional Delphi client/server architecture (even with three logical tiers) requires you to install the database access on each client, something quite troublesome when you must configure and maintain hundreds of machines. In the physical three-tier architecture, you need to install and configure the database access only on the application server, not on the client computers. Since the client programs have only user interface code and are extremely simple to install, they now fall into the category of so-called *thin clients*. To use marketing-speak, we might even call this a *zero-configuration thin-client architecture*. But let us focus on technical issues instead of marketing terminology.

## The Technical Foundation of DataSnap

When Borland introduced this physical multitier architecture in Delphi, it was called MIDAS (Middle-tier Distributed Application Services). For example, Delphi 5 included the third version of this technology, MIDAS 3. Now Delphi 6 renames this technology as *DataSnap* and extends its capabilities.

DataSnap requires the installation of specific libraries on the server (actually the middle-tier computer), which provides your client computers with the data extracted from the SQL server database or other data sources. DataSnap does not require a SQL server for data storage. DataSnap can serve up data from a wide variety of sources, including SQL, CORBA, other DataSnap servers, or just data computed on the fly.

As you would expect, the client side of DataSnap is extremely thin and easy to deploy. The only file you need is `Midas.dll`, a small (260 KB) DLL that implements the `ClientDataSet` and `RemoteServer` components and provides the connection to the application server. As we've seen in Chapter 14, "Client/Server Programming," this DLL is basically a small, stand-alone database engine. It caches data from a remote data module and enforces the rules requested by the Constraint Broker.

The application server uses the same DLL to handle the datasets (called *deltas*) returned from the clients when they post updated or new records. However, the server also requires several other libraries, all of which are installed by DataSnap.

## The *IAppServer* Interface

Starting with Delphi 5, the two sides of a DataSnap application communicate using the `IAppServer` interface.

### NOTE

In Delphi 5 (and 6), the `IAppServer` interface supersedes Delphi 4's `IProvider` interface. The main reason for this change was support for stateless objects. With `IProvider`, the server stored status information about the client program—for example, which records had already been passed to the client. This made it difficult to adapt the server-side objects to stateless connection layers, like CORBA message queues and MTS, and also to move toward HTTP and Web-based support. Other reasons for moving to this new architecture were to make the system more dynamic (providers are now exported by setting a property, not by changing the type library) and to reduce the number of calls, or *round-trips*, which can affect performance. DataSnap makes fewer calls but delivers more data each time.

The `IAppServer` interface has the following methods:

- `AS_ApplyUpdates`
- `AS_GetRecords`
- `AS_DataRequest`
- `AS_GetProviderNames`
- `AS_GetParams`
- `AS_RowRequest`
- `AS_Execute`

You'll seldom need to call them directly, anyway, because there are Delphi components to be used on the client and server sides of the application that embed these calls, making them easier (and at times even hiding them completely). In practice, the server will make available to the client objects implementing this interface, possibly along with other custom interfaces.

### NOTE

A DataSnap server exposes an interface using a COM type library, a technology I'll discuss in Chapter 20, "Automation, ActiveX, and Other COM Technologies."

## The Connection Protocol

DataSnap defines only the higher-level architecture and can use different technologies for moving the data from the middle tier to the client side. DataSnap supports most of the leading standards, including the following:

**Distributed COM (DCOM) and Stateless COM (MTS or COM+)** DCOM is directly available in Windows NT/2000 and 98/Me, and it requires no additional run-time applications on the server. You still have to install it on Windows 95 machines. DCOM is basically an extension of COM technology (discussed in Chapter 19, “COM Programming,” and Chapter 20) that allows a client application to use server objects that exist and execute on a separate computer. The DCOM infrastructure allows you to use stateless COM objects, available in the COM+ and in the older MTS (Microsoft Transaction Server) architectures. Both COM+ and MTS provide features such as security, component management, and database transactions, and are available in Windows NT/2000 and in Windows 98/Me.

Due to the complexity of DCOM configuration and of its problems in passing through firewalls, even Microsoft is abandoning DCOM in favor of SOAP-based solutions.

**TCP/IP sockets** These are available on most systems. Using TCP/IP you might distribute clients over the Web, where DCOM cannot be taken for granted, and have many fewer configuration headaches. To use sockets, the middle-tier computer must run the `ScktSrvr.exe` application provided by Borland, a single program that can run either as an application or as a service. This program receives the client requests and forwards them to the remote data module (executing on the same server) using COM. Sockets provide no protection against failure on the client side, as the server is not informed and might not release resources when a client unexpectedly shuts down.

**HTTP and SOAP** The use of HTTP as a transport protocol over the Internet simplifies connections through firewalls or proxy servers (which generally don't like custom TCP/IP sockets). You need a specific Web server application, `httpsrvr.dll`, which accepts client requests and creates the proper remote data modules using COM. These Web connections can use SSL security but must register themselves by adding a call to `EnableWebTransport` in the `UpdateRegistry` method. Finally, Web connections based on HTTP transport can use DataSnap object-pooling support.

---

### NOTE

The DataSnap HTTP transport can use XML as the data packet format, enabling any platform or tool that can read XML to participate in a DataSnap architecture. This is an extension of the original DataSnap data packet format, which is also platform-independent. The use of XML over HTTP is also the foundation of SOAP. There's more on XML in Chapter 23.

**CORBA** Common Object Request Broker Architecture is an official standard for object management available on most operating systems. Compared to DCOM, the advantage is that your client and server applications can be also written with Java and other products. The Borland implementation of CORBA, VisiBroker, is available with Delphi Enterprise. CORBA provides many benefits, including location transparency, load balancing, and fail-over from the ORB run-time software. (An in-depth discussion of CORBA is certainly beyond the scope of this book, and in practice only a limited number of Delphi programmers use CORBA.)

**Internet Express** As an extension to this architecture, you can transform the data packets into XML and deliver them to a Web browser. In this case, you basically have one extra tier: the Web server gets the data from the middle tier and delivers it to the client. I'll discuss this new architecture, called Internet Express, in Chapter 23. The DLL can also be folded into the executable file by using the MidasLib unit.

## Providing Data Packets

The entire Delphi multitier data-access architecture centers around the idea of *data packets*. In this context, a data packet is a block of data that moves from the application server to the client or from the client back to the server. Technically, a data packet is a sort of subset of a dataset. It describes the data it contains (usually a few records of data), and it lists the names and types of the data fields. Even more important, a data packet includes the constraints—that is, the rules to be applied to the dataset. You'll typically set these constraints in the application server, and the server sends them to the client applications along with the data.

All communication between the client and the server occurs by exchanging data packets. The provider component on the server manages the transmission of several data packets within a big dataset, with the goal of responding faster to the user. As the client receives a data packet, in a ClientDataSet component, the user can edit the records it contains. As mentioned earlier, during this process the client also receives and checks the constraints, which are applied during the editing operations.

When the client has updated the records and sends a data packet back, that packet is known as a *delta*. The delta packet tracks the difference between the original records and the updated ones, recording all the changes the client requested from the server. When the client asks to apply the updates to the server, it sends the delta to the server, and the server tries to apply each of the changes. I say *tries* because if a server is connected to several clients, the data might have changed already, and the update request might fail.

Since the delta packet includes the original data, the server can quickly determine if another client has already changed it. If so, the server fires an OnReconcileError event, which is one of the vital elements for thin-client applications. In other words, the three-tier

architecture uses an update mechanism similar to the one Delphi uses for cached updates. As we have seen in Chapter 14, “Client/Server Programming Techniques,” the `ClientDataSet` manages data in a memory cache, and it typically reads only a subset of the records available on the server side, loading more elements only as they’re needed. When the client updates records or inserts new ones, it stores these pending changes in another local cache on the client, the *delta cache*.

The client can also save the data packets to disk and work off-line, thanks to the MyBase support discussed in Chapter 13, “Delphi’s Database Architecture.” Even error information and other data moves using the data packet protocol, so it is truly one of the foundation elements of this architecture.

**NOTE**

It’s important to remember that data packets are protocol-independent. A data packet is merely a sequence of bytes, so anywhere you can move a series of bytes, you can move a data packet. This was done to make the architecture suitable for multiple transport protocols (including DCOM, CORBA, HTTP, and TCP/IP) and for multiple platforms.

## Delphi Support Components (Client-Side)

Now that we’ve examined the general foundations of Delphi’s three-tier architecture, we can focus on the components that support it. For developing client applications, Delphi provides the `ClientDataSet` component, which provides all the standard dataset capabilities and embeds the client side of the `IAppServer` interface. In this case, the data is delivered through the remote connection.

The connection to the server application is made via another component you’ll also need in the client application. You should use one of the four specific connection components (available in the DataSnap page):

- The `DCOMConnection` component can be used on the client side to connect to a DCOM and MTS server, located either on the current computer or in another one indicated by the `ComputerName` property. The connection is with a registered object having a given `ServerGUID` or `ServerName`.
- The `CorbaConnection` component can be used to hook with a CORBA server. You indicate the `HostName` (the name or IP address) to indicate the server computer, the `RepositoryID` to request a specific data module on the server, and optionally the `ObjectName` property if the data module exports multiple objects.
- The `SocketConnection` component can be used to connect to the server via a TCP/IP socket. You should indicate the IP address or the host name, and the `GUID` of the server object (in the `InterceptGUID` property). In Delphi 5, this connection component

has an extra property, `SupportCallbacks`, which you can disable if you are not using callbacks and want to deploy your program on Windows 95 computers that don't have Winsock 2 installed.

**NOTE**

In the `WebServices` page, you can also find the `SoapConnection` component, which requires a specific type of server and will be discussed in Chapter 23.

- The `WebConnection` component is used to handle an HTTP connection that can easily get through a firewall. You should indicate the URL where your copy of `httpsrvr.dll` is located and the name or GUID of the remote object on the server.

Delphi 6 adds new client-side components to the `DataSnap` architecture, mainly for managing connections:

- The `ConnectionBroker` component can be used as an alias of an actual connection component, something useful when you have a single application with multiple client datasets. In fact, to change the physical connection of each of the datasets, you only need to change the `Connection` property of the `ConnectionBroker`. You can also use the events of this virtual connection component in place of those of the actual connections, so you don't have to change any code if you change the data transport technology. For the same reason, you can refer to the `AppServer` object of the `ConnectionBroker` instead of the corresponding property of a physical connection.
- The `SharedConnection` component can be used to connect to a secondary (or child) data module of a remote application, piggy-backing on an existing physical connection to the main data module. In other words, an application can connect to multiple data modules of the server with a single, shared connection.
- The `LocalConnection` component can be used to target a local dataset provider as the source of the data packet. The same effect can be obtained by hooking the `ClientDataSet` directory to the provider. However, using the `LocalConnection`, you can write a local application with the same code as a complete multitier application, using the `IAppServer` interface of the "fake" connection. This will make the program easier to scale up, compared to a program with a direct connection.

A few other components of the `DataSnap` page relate to the transformation of the `DataSnap` data packet into custom XML formats. These components (`XMLTransform`, `XMLTransformProvider`, and `XMLTransformClient`) will be discussed in Chapter 23.



## Delphi Support Components (Server-Side)

On the server side (or actually the middle tier), you'll need to create an application or a library that embeds a remote data module, a special version of the `TDataModule` class. There are actually specialized remote data modules for transactional COM and CORBA support. In the Multitier page of the New Items dialog box (obtained with the File > New > Others menu) are specific wizards to create remote data modules of each of these types.

The only specific component you need on the server side is the `DataSetProvider`. You need one of these components for every table or query you want to make available to the client applications, which will then use a separate `ClientDataSet` component for every exported dataset. The `DataSetProvider` was already introduced in Chapter 14.

### NOTE

The `DataSetProvider` component of Delphi 5 and 6 supersedes the stand-alone `Provider` component of Delphi 4 and the internal `Provider` object, which was embedded in the `TBDEDataSet` subclasses.

## Building a Sample Application

Now we're ready to build a sample program. This will allow us to observe some of the components I've just described in action, and it will also allow us to focus on some other problems, shedding light on other pieces of the Delphi multitier puzzle. I'll build the client and application server portions of a three-tier application in two steps. The first step will simply test the technology using a bare minimum of elements. These programs will be very simple.

From that point, we'll add more power to the client and the application server. In each of the examples, we'll display data from local Paradox tables, and we'll set up everything to allow you to test the programs on a stand-alone computer. I won't cover the steps you have to follow to install the examples on multiple computers with various technologies—that would be the subject of at least one other book.

## The First Application Server

The server side of our basic example is very easy to build. Simply create a new application and add a remote data module to it using the corresponding icon in the Multitier page of the Object Repository. The simple Remote Data Module Wizard (see Figure 17.1) will ask you for a class name and the instancing style. As you enter a class name, such as `AppServerOne`, and click the OK button, Delphi will add a data module to the program. This data module will have the usual properties and events, but its class will have the following Pascal declaration:

```
type
  TAppServerOne = class(TRemoteDataModule, IAppServerOne)
  private
```

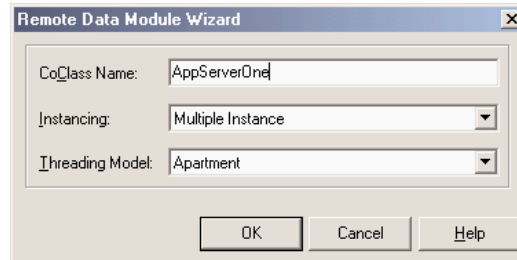
```

    { Private declarations }
protected
    class procedure UpdateRegistry(Register: Boolean;
        const ClassID, ProgID: string); override;
public
    { Public declarations }
end;

```

**FIGURE 17.1:**

The simple Remote Data Module Wizard



In addition to inheriting from the `TRemoteDataModule` base class, this class implements the custom `IAppServerOne` interface, which derives from the standard `DataSnap` interface (`IAppServer`). The class also overrides the `UpdateRegistry` method to add the support for enabling the socket and Web transports, as you can see in the code generated by the wizard. At the end of the unit, you'll find the class factory declaration, which will become clear after reading Chapter 19:

#### initialization

```

TComponentFactory.Create(ComServer, TAppServerOne,
    Class_AppServerOne, ciMultiInstance, tmApartment);
end.

```

Now you can add a dataset component to the data module (I've used the `dbExpress SQLDataSet`), connect it to a database and a table or query, activate it, and finally add a `DataSetProvider` and hook it to the dataset component. You'll obtain a DFM file like this:

```

object AppServerOne: TAppServerOne
    object SQLConnection1: TSQLConnection
        ConnectionName = 'IBLocal'
        LoginPrompt = False
    end
    object SQLDataSet1: TSQLDataSet
        SQLConnection = SQLConnection1
        CommandText = 'select * from EMPLOYEE'
    end
    object DataSetProvider1: TDataSetProvider
        DataSet = SQLDataSet1
        Constraints = True

```

```
    end  
end
```

What about the main form of this program? Well, it's almost useless, so we can simply add a label to it indicating that it's the form of the server application. When you've built the server, you should compile it and run it once. This operation will automatically register it as an Automation server on your system, making it available to client applications. Of course, you should register the server on the computer where you want it to run, either the client or the middle tier.

## The First Thin Client

Now that we have a working server, we can build a client that will connect to it. We'll again start with a standard Delphi application and add a DCOMConnection component to it (or the proper component for the specific type of connection you want to test). This component defines a ComputerName property that you'll use to specify the computer that hosts the application server. If you want to test the client and application server from the same computer, you can leave this blank.

Once you've selected an application server computer, you can simply display the ServerName property's combo-box list to view the available DataSnap servers. This combo box shows the servers' registered names, by default the name of the executable file of the server followed by the name of the remote data module class, as in AppServ1.AppServerOne. Alternatively, you can type the GUID of the server object in the ServerGUID property. Delphi will automatically fill this property as you set the ServerName property, determining the GUID by looking it up in the Registry.

At this point, if you set the DCOMConnection component's Connected property to True, the server form will appear, indicating that the client has activated the server. You don't usually need to perform this operation, because the ClientDataSet component typically activates the RemoteServer component for you. I've suggested this simply to emphasize what's happening behind the scenes.

---

**TIP**

You should generally leave the DCOMConnection component's Connected property set to False at design time, to be able to open the project in Delphi even on a computer where the DataSnap server is not already registered.

As you might expect, the next step is to add a ClientDataSet component to the form. You must connect the ClientDataSet to the DCOMConnection1 component via the RemoteServer property, and thereby to one of the providers it exports. You can see the list of available providers in the ProviderName property, via the usual combo box. In this example, you'll be able to select only DataSetProvider1, as this is the only provider available in the server we've

just built. This operation connects the dataset in the client's memory with the dbExpress dataset on the server. If you activate the client dataset and add a few data-aware controls (or a DBGrid), you'll immediately see the server data appear in them, as illustrated in Figure 17.2.

**FIGURE 17.2:**

When you activate a ClientDataSet component connected to a remote data module at design time, the data from the server becomes visible as usual.

DEPT_NO	EMP_NO	FIRST_NAME	FULL_NAME	HIRE
600	2	Robert	Nelson, Robert	12/28
621	4	Bruce	Young, Bruce	12/28
130	5	Kim	Lambert, Kim	2/6/1
180	8	Leslie	Johnson, Leslie	4/5/1
622	9	Phil	Forest, Phil	4/17/
130	11	K. J.	Weston, K. J.	1/17/
000	12	Terri	Lee, Terri	5/1/1
900	14	Stewart	Hall, Stewart	6/4/1
623	15	Katherine	Young, Katherine	6/14/
671	20	Chris	Papadopoulos, Chris	1/1/1
671	24	Pete	Fisher, Pete	9/12/
120	28	Ann	Bennet, Ann	2/1/1
623	29	Roger	De Souza, Roger	2/18/
110	34	Janet	Baldwin, Janet	3/21/

Here is the DFM file for our minimal client application, ThinCli1:

```

object Form1: TForm1
  Caption = 'ThinClient1'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object DCOMConnection1: TDCOMConnection
    ServerGUID = '{09E11D63-4A55-11D3-B9F1-0000100A27B}'
    ServerName = 'AppServ1.AppServerOne'
  end
  object ClientDataSet1: TClientDataSet
    Aggregates = <>
    Params = <>
    ProviderName = 'DataSetProvider1'
    RemoteServer = DCOMConnection1
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
end

```

Obviously, the programs of our first three-tier application are quite simple, but they demonstrate how to create a dataset viewer that splits the work between two different executable files. At this point, our client is only a viewer. If you edit the data on the client, it won't be updated

on the server. To accomplish this, you'll need to add some more code to the client. However, before we do that, let's add some features to the server.

## Adding Constraints to the Server

When you write a traditional data module in Delphi, you can easily add some of the application logic, or business rules, by handling the dataset events, and by setting field object properties and handling their events. You should avoid doing this work on the client application; instead, write your business rules on the middle tier.

In the DataSnap architecture, you can send some constraints from the server to the client and let the client program impose those constraints during the user input. You can also send field properties (such as minimum and maximum values and the display and edit masks) to the client and (using some of the data access technologies) process updates through the dataset used to access the data (or a companion `UpdateSql` object).

### Field and Table Constraints

When the provider interface creates data packets to send to the client, it includes the field definitions, the table and field constraints, and one or more records (as requested by the `ClientDataSet` component). This implies that you can customize the middle tier and build distributed application logic by using SQL-based constraints.

The constraints you create using SQL expressions can be assigned to an entire dataset or to specific fields. The provider sends the constraints to the client along with the data, and the client applies them before sending updates back to the server. This reduces network traffic, compared to having the client send updates back to the application server and eventually up to the SQL server, only to find that the data is invalid. Another advantage of coding the constraints on the server side is that if the business rules change, you need to update the single server application and not the many clients on multiple computers.

But how do you write constraints? There are several properties you can use:

- BDE datasets have a `Constraints` property, which is a collection of `TCheckConstraint` objects. Every object has a few properties, including the expression and the error message.
- Each field object defines the `CustomConstraint` and `ConstraintErrorMessage` properties. There is also an `ImportedConstraint` property for constraints imported from the SQL server.
- Each field object has also a `DefaultExpression` property, which can be used locally or passed to the `ClientDataSet`. This is not an actual constraint, only a suggestion to the end user.

Our next example, AppServ2, adds a few constraints to a remote data module connected to the sample EMPLOYEE InterBase database. After connecting the table to the database and creating the field objects for it, you can set the following special properties:

```

object SQLDataSet1: TSQLDataSet
  ...
  object SQLDataSet1EMP_NO: TSmallintField
    CustomConstraint = 'x > 0 and x < 10000'
    ConstraintErrorMessage =
      'Employee number must be a positive integer below 10000'
    FieldName = 'EMP_NO'
  end
  object SQLDataSet1FIRST_NAME: TStringField
    CustomConstraint = 'x <> '#39#39
    ConstraintErrorMessage = 'The first name is required'
    FieldName = 'FIRST_NAME'
    Size = 15
  end
  object SQLDataSet1LAST_NAME: TStringField
    CustomConstraint = 'not x is null'
    ConstraintErrorMessage = 'The last name is required'
    FieldName = 'LAST_NAME'
  end
end

```

## NOTE

The expression 'x <> '#39#39 is the DFM transposition of the string x <> '', indicating that we don't want to have an empty string. The final constraint, `not x is null`, instead allows empty strings but not null values.

## Including Field Properties

You can control whether the properties of the field objects on the middle tier are sent to the ClientDataSet (and copied into the corresponding field objects of the client side), by using the `poIncFieldProps` value of the `Options` property of the `DataSetProvider`. This flag controls the download of the field properties `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, and `DisplayValues`, if they are available in the field. Here is an example of another field of the AppServ2 example with some custom properties:

```

object SQLDataSet1SALARY: TBCDField
  DefaultExpression = '10000'
  FieldName = 'SALARY'
  DisplayFormat = '#,###'
  EditFormat = '####'
  Precision = 15
  Size = 2
end

```

With this setting, you can simply write your middle tier the way you usually set the fields of a standard client/server application. This approach also makes it faster to move existing applications from a client/server to a multitier architecture. The main drawback of sending fields to the client is that transmitting all the extra information takes time. Turning off `poIncFieldProps` can dramatically improve network performance of datasets with many columns.

A server can generally filter the fields returned to the client; it does this by declaring persistent field objects with the Fields editor and omitting some of the fields. Because a field you're filtering out might be required to identify the record for future updates (if the field is part of the primary key), you can also use the field's `ProviderFlags` property on the server to send the field value to the client but make it unavailable to the `ClientDataSet` component (this provides some extra security, compared to sending the field to the client and hiding it there).

## Field and Table Events

You can write middle-tier dataset and field event handlers as usual and let the dataset process the updates received by the client in the traditional way. This means that updates are considered to be operations on the dataset, exactly as when a user is directly editing, inserting, or deleting fields locally.

This is accomplished by setting the `ResolveToDataSet` property of the `TDataSetProvider` component, again connecting either the dataset used for input or a second one used for the updates. This approach is possible with datasets supporting editing operations. These includes BDE, ADO, and InterBase Express datasets, but not those of the new dbExpress architecture.

With this technique, the updates are performed by the dataset, which implies a lot of control (the standard events are being triggered) but generally slower performance. Flexibility is much greater, as you can use standard coding practices. Also, porting existing local or client/server database applications, which use dataset and field events, is much more straightforward with this model. However, keep in mind that the user of the client program will receive your error messages only when the local cache (the delta) is sent back to the middle tier. Saying to the user that some data prepared half an hour ago is not valid might be a little awkward. If you follow this approach, you'll probably need to apply the updates in the cache at every `AfterPost` event on the client side.

Finally, if you decide to let the dataset and not the provider do the updates, Delphi helps you a lot in handling possible exceptions. Any exceptions raised by the middle-tier update events (for example, `OnBeforePost`) are automatically transformed by Delphi into update errors, which activate the `OnReconcileError` event on the client side (more on this event later in this chapter). No exception is shown on the middle tier, but the error travels back to the client.

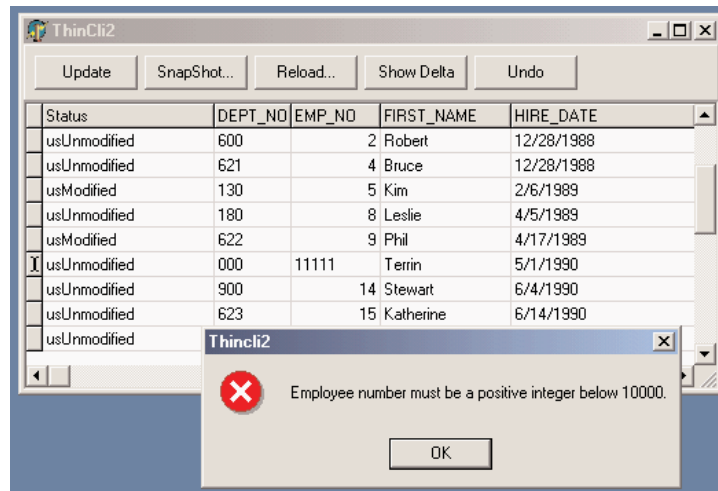
## Adding Features to the Client

After adding some constraints and field properties to the server, we can now return our attention to the client application. The first version was very simple, but now there are several features we can add to it to make it work well. In the `ThinCli2` example, I've embedded support for checking the record status and accessing the delta information (the updates to be sent back to the server), using some of the `ClientDataSet` techniques already discussed in Chapter 13. The program also handles reconcile errors and supports the briefcase model.

Keep in mind that while you're using this client to edit the data locally, you'll be reminded of any failure to match the business rules of the application, set up on the server side using constraints. The server will also provide us with a default value for the Salary field of a new record and pass along the value of its `DisplayFormat` property. In Figure 17.3 you can see one of the error messages this client application can display, which it receives from the server. This message is displayed while editing the data locally, not when you send it back to the server.

**FIGURE 17.3:**

The error message displayed by the `ThinCli2` example when the employee ID is too large



## The Update Sequence

This client program also includes a button to Apply the updates to the server and a standard reconcile dialog. Here is a summary of the complete sequence of operations related to an update request and the possible error events:

1. The client program calls the `ApplyUpdates` method of a `ClientDataSet`.
2. The delta is sent to the provider on the middle tier. The provider fires the `OnUpdateData` event, where you have a chance to look at the requested changes before they reach the



database server. At this point you can modify the delta, which is passed in a format compatible with the data of a `ClientDataSet`.

3. The provider (technically, a part of the provider called the “resolver”) applies each row of the delta to the database server. Before applying each update, the provider receives a `BeforeUpdateRecord` event. If you’ve set the `ResolveToDataSet` flag, this update will eventually fire local events of the dataset in the middle tier.
4. In case of a server error, the provider fires the `OnUpdateError` event (on the middle tier) and the program has a chance of fixing the error at that level.
5. If the middle-tier program doesn’t fix the error, the corresponding update request remains in the delta. The error is returned to the client side at this point or after a given number of errors have been collected, depending on the value of the `MaxErrors` parameter of the `ApplyUpdates` call.
6. Finally, the delta packet with the remaining updates is sent back to the client, firing the `OnReconcileError` event of the `ClientDataSet` for each remaining update. In this event handler, the client program can try to fix the problem (possibly prompting the user for help), modifying the update in the delta, and later reissuing it.

## Refreshing Data

You can obtain an updated version of the data, which other users might have modified, by calling the `Refresh` method of the `ClientDataSet`. However, this operation can be done only if there are no pending update operations in the cache, as calling `Refresh` raises an exception when the change log is not empty:

```
if cds.ChangeCount = 0 then  
    cds.Refresh;
```

If only some records have been changed, you can refresh the others by calling `RefreshRecords`. This method refreshes only the current record, but it should be used only if the user hasn’t modified the current record. In this case, in fact, `RefreshRecords` leaves the unapplied changes in the change log. As an example, you can refresh a record every time it becomes the active one, unless it has been modified and the changes have not yet been posted to the server:

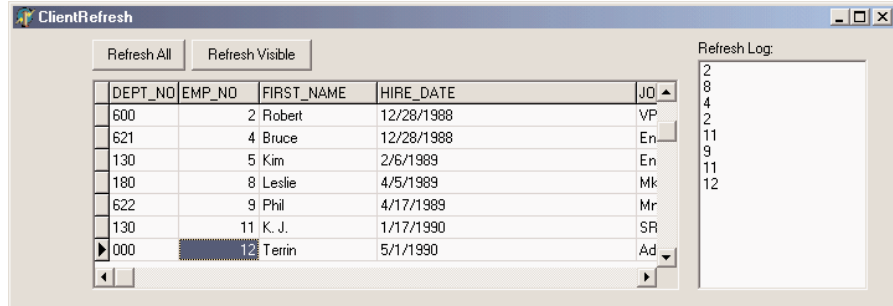
```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);  
begin  
    if cds.UpdateStatus = usUnmodified then  
        cds.RefreshRecord;  
end;
```

When the data is subject to frequent changes by many users and each user should see changes right away, you should generally apply any change immediately in the `AfterPost`

and `AfterDelete` methods, and call `RefreshRecords` for the active record (as shown above) or each of the records visible inside a grid. This code is actually part of the `ClientRefresh` example, connected to the `AppServ2` server. For debugging purposes, the program also logs the `EMP_NO` field for each record it refreshes, as you can see in Figure 17.4.

**FIGURE 17.4:**

The form of the `ClientRefresh` example, which automatically refreshes the active record and allows more extensive updates by pressing the buttons



I've done this by adding a button to the `ClientRefresh` example. The handler of this button moves from the current record to the first visible record of the grid and then to the last visible record. This is accomplished by noting that there are `RowCount - 1` rows visible, assuming that the first row is the fixed one hosting the field names. The program doesn't call `RefreshRecord` every time, as each movement will trigger an `AfterScroll` event with the code shown above. This is the code to refresh the visible rows, which might even be triggered by a timer:

```
var
  i: Integer;
  bm: TBookmarkStr;
begin
  // refresh visible rows
  cds.DisableControls;
  // start with the current row
  i := TMyGrid(DbGrid1).Row;
  bm := cds.Bookmark;
  try
    // get back to the first visible record
    while i > 1 do
      begin
        cds.Prior;
        Dec (i);
      end;
    // return to the current record
    i := TMyGrid(DbGrid1).Row;
    cds.Bookmark := bm;
    // go ahead until the grid is complete
```

```
while i < TMyGrid(DbGrid1).RowCount do
begin
  cds.Next;
  Inc (i);
end;
finally
  // set back everything and refresh
  cds.Bookmark := bm;
  cds.EnableControls;
end;
```

This approach generates a huge amount of network traffic, so you might want to trigger updates only when there are actual changes. This can be implemented by adding a callback technology to the server, so that it can inform all connected clients that a given record has changed. The client can determine whether it is interested in the change and eventually trigger the update request.

## Advanced DataSnap Features

There are many more features in DataSnap than I've covered up to now. Here is a quick tour of some of the more advanced features of the architecture, partially demonstrated by the AppSPlus and ThinPlus examples. Unfortunately, demonstrating every single idea would turn this chapter into an entire book (and not every Delphi programmer is interested in and can afford DataSnap), so I'll limit myself to an overview.

Besides the features discussed in the following sections, the AppSPlus and ThinPlus examples demonstrate the use of a socket connection, limited logging of events and updates on the server side, and direct fetching of a record on the client side. The last feature is accomplished with this call:

```
procedure TClientForm.ButtonFetchClick(Sender: TObject);
begin
  ButtonFetch.Caption := IntToStr (cds.GetNextPacket);
end;
```

This allows you to get more records than are actually required by the client user interface (the DBGrid). In other words, you can fetch records directly, without waiting for the user to scroll down in the grid. I suggest you study the details of these complex examples after reading the rest of this section.

ThinPlus example: This program requires Delphi's socket server (provided in Delphi's bin folder) to run. This is apparently not clear in the text. Without this program you'll see a socket error.

## Parametric Queries

If you want to use parameters in a query or stored procedure, then instead of building a custom solution (with a custom method call to the server), you can let Delphi help you. First define the query on the middle tier with a parameter, such as:

```
select * from customer where Country = :Country
```

Use the Params property to set the type and default value of the parameter. On the client side, you can use the Fetch Params command of the ClientDataSet's shortcut menu, after connecting it to the proper provider. At run time, you can call the equivalent FetchParams method of the ClientDataSet component.

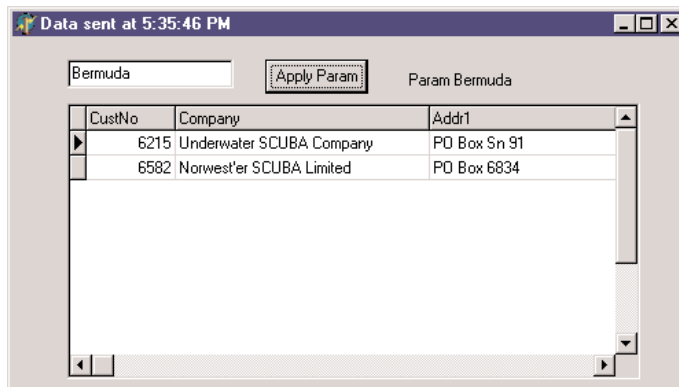
Now you can provide a local default value to the parameter by acting on the Params property. This will be sent to the middle tier when you fetch the data. The ThinPlus example refreshes the parameter with the following code:

```
procedure TFormQuery.btnParamClick(Sender: TObject);  
begin  
    cdsQuery.Close;  
    cdsQuery.Params[0].AsString := EditParam.Text;  
    cdsQuery.Open;  
end;
```

You can see the secondary form of this example, which shows the result of the parametric query in a grid, in Figure 17.5. In the figure you can also see some custom data sent by the server, as explained in the section “Customizing the Data Packets.”

**FIGURE 17.5:**

The secondary form of the ThinPlus example, showing the data of a parametric query



## Custom Method Calls

Since the server has a normal COM interface, we can add more methods or properties to it and call them from the client. Simply open the type library editor of the server and use it as with any other COM server. In the AppSPlus example, I've added a custom Login method with the following implementation:

```
procedure TAppServerPlus.Login(const Name, Password: WideString);  
begin  
    // TODO: add actual login code...  
    if Password <> Name then  
        raise Exception.Create ('Wrong name/password combination received')  
    else  
        Query.Active := True;  
        ServerForm.Add ('Login:' + Name + '/' + Password);  
end;
```

The program makes a simple test, instead of checking the name/password combination against a list of authorizations as a real application should do. Also, disabling the Query doesn't really work, as it can be activated by the provider. Disabling the DataSetProvider is actually a more robust approach. The client has a simple way to access the server, the AppServer property of the remote connection component. Here is a sample call from the ThinPlus example, which takes place in the AfterConnect event of the connection component:

```
procedure TClientForm.ConnectionAfterConnect(Sender: TObject);  
begin  
    Connection.AppServer.Login (Edit2.Text, Edit3.Text);  
end;
```

Note that you can call extra methods of the COM interface through DCOM and also using a socket-based or HTTP connection. Because the program uses the safecall calling convention, the exception raised on the server is automatically forwarded and displayed on the client side. This way, when a user selects the Connect check box, the event handler used to enable the client datasets is interrupted, and a user with the wrong password won't be able to see the data.

### NOTE

Besides direct method calls from the client to the server, you can also implement callbacks from the server to the client. This can be used, for example, to notify every client of specific events. COM events are one way to do this. As an alternative, you can add a new interface, implemented by the client, which passes the implementation object to the server. This way, the server can call the method on the client computer. Callbacks are not possible with HTTP connections, though.

## Master/Detail Relations

If your middle-tier application exports multiple datasets, you can retrieve them using multiple `ClientDataSet` components on the client side and connect them locally to form a master/detail structure. This will create quite a few problems for the detail dataset unless you retrieve all of the records locally.

This solution also makes it quite complex to apply the updates; you cannot usually cancel a master record until all related detail records have been removed, and you cannot add detail records until the new master record is properly in place. (Actually, different servers handle this differently, but in most cases where a foreign key is used, this is the standard behavior.) What you can do to solve this problem is to write complex code on the client side to update the records of the two tables according to the specific rules.

A completely different approach is to retrieve a single dataset that already includes the detail as a dataset field, a field of type `TDataSetField`. To accomplish this, you need to set up the master/detail relation on the server application:

```
object TableCustomer: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object TableOrders: TTable
  DatabaseName = 'DBDEMOS'
  MasterFields = 'CustNo'
  MasterSource = DataSourceCust
  TableName = 'ORDERS.DB'
end
object DataSourceCust: TDataSource
  DataSet = TableCustomer
end
object ProviderCustomer: TDataSetProvider
  DataSet = TableCustomer
end
```

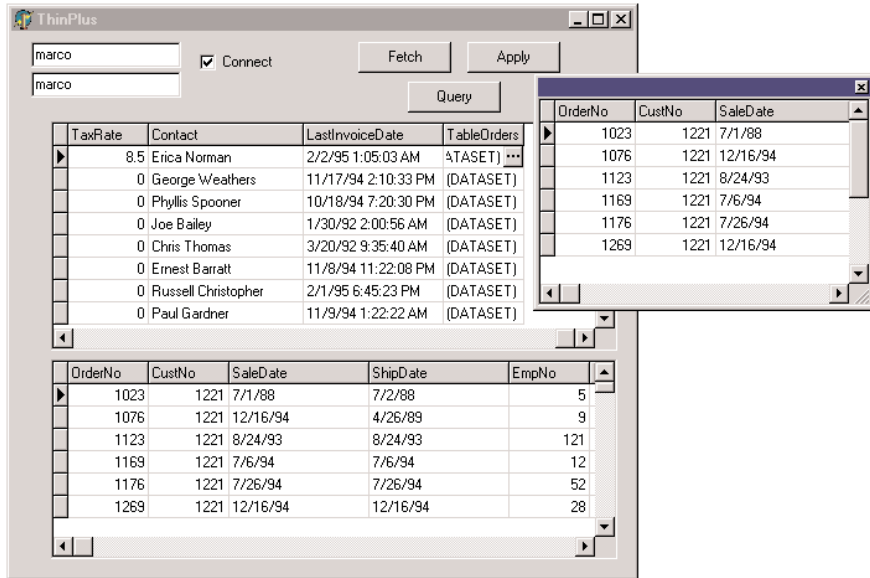
On the client side, the detail table will show up as an extra field of the `ClientDataSet`, and the `DBGrid` control will display it as an extra column with an ellipsis button. Clicking the button will display a secondary form with a grid presenting the detail table (see Figure 17.6). If you need to build a flexible user interface on the client, you can then add a secondary `ClientDataSet` connected to the dataset field of the master dataset, using the `DataSetField` property. Simply create persistent fields for the main `ClientDataSet` and then hook up the property:

```
object cdsDet: TClientDataSet
  DataSetField = cdsTableOrders
end
```

With this setting you can show the detail dataset in a separate DBGrid placed as usual in the form (the bottom grid of Figure 17.6) or in any other way you like. Note that with this structure, the updates relate only to the master table, and the server should handle the proper update sequence even in complex situations.

**FIGURE 17.6:**

The ThinPlus example shows how a dataset field can either be displayed in a grid in a floating window or extracted by a ClientDataSet and displayed in a second form. You'll generally do one of the two things, not both!



## Using the Connection Broker

I've already mentioned that the ConnectionBroker component can be helpful in case you might want to change the physical connection used by many ClientDataSet components of a single program. In fact, by hooking each ClientDataSet to the ConnectionBroker, you can change the physical connection of them all simply by updating the physical connection of the broker.

These are the settings used by the ThinPlus example:

```

object Connection: TSocketConnection
  ServerName = 'AppSPlus.AppServerPlus'
  AfterConnect = ConnectionAfterConnect
  Address = '127.0.0.1'
end
object ConnectionBroker1: TConnectionBroker
  Connection = Connection
end

```

```
object cds: TClientDataSet
  ConnectionBroker = ConnectionBroker1
end
// in the secondary form
object cdsQuery: TClientDataSet
  ConnectionBroker = ClientForm.ConnectionBroker1
end
```

That's basically all you have to do. To change the physical connection, drop a new DataSnap connection component to the main form and set the `Connection` property of the broker to it.

**WARNING**

There are some glitches with the `ConnectionBroker`, even in the shipping version of Delphi 6. If you experience unusual errors in a program that uses this component, try removing it. Of course, this note applies only until Borland provides a patch to fix this behavior.

## More Provider Options

I've already mentioned the `Options` property of the `DataSetProvider` component, noting that it can be used to add the field properties to the data packet. There are several other options you can use to customize the data packet and the behavior of the client program. Here is a short list:

- You can minimize downloading BLOB data with `poFetchBlobsOnDemand` option. In this case, the client application can download BLOBs by specifying the `FetchOnDemand` property of the `ClientDataSet` to `True` or by calling the `FetchBlobs` method for specific records. Similarly, you can disable the automatic downloading of detail records by setting the `poFetchDetailsOnDemand` option. Again, the client can use the `FetchOnDemand` property or call the `FetchDetails` method.
- When you are using a master/detail relation, you can control cascades with either of two options. The `poCascadeDeletes` flag controls whether the provider should delete detail records before deleting a master record. You can set this option if the database server performs cascaded deletes for you as part of its referential integrity support. Similarly, you can set the `poCascadeUpdates` option when the update of key values of a master/detail relation can be performed automatically by the server.
- You can limit the operations on the client side. The most restrictive option, `poReadOnly`, disables any update. If you want to give the user a limited editing capability, use `poDisableInserts`, `poDisableEdits`, or `poDisableDeletes`.
- You can resend to the client a copy of the records the client has modified with `poAutoRefresh`, which is useful in case other users have simultaneously made other, nonconflicting changes. You can also send back to the client changes done in the `BeforeUpdateRecord`



or `AfterUpdateRecord` event handlers by specifying the `poPropagateChanges` option. This option is also handy when you are using autoincrement fields, triggers, and other techniques that modify data on the server or middle tier beyond the changes requested from the client tier.

- Finally, if you want the client to drive the operations, you can enable the `poAllowCommandText` option. This lets you set the SQL query or table name of the middle tier from the client, using the `GetRecords` or `Execute` methods.

## The Simple Object Broker

The `SimpleObjectBroker` component provides an easy way to locate a server application among several server computers. You simply provide a list of available computers, and the client will try each of them in order until it finds one that is available.

Moreover, if you enable the `LoadBalanced` property, the component will randomly choose one of the servers; when many clients use the same configuration, the connections will be automatically distributed among the multiple servers. If this seems like a “poor man’s” object broker, consider that some highly expensive load-balancing systems don’t actually offer much more than this.

## Object Pooling

When multiple clients connect to your server at the same time, you have two options. The first is to create a remote data module object for each of them and let each request be processed in sequence (the default behavior for a COM server with the `ciMultiInstance` style). Alternatively, you can let the system create a different instance of the application for every client (`ciSingleInstance`). This requires more resources and more SQL server connections (and licenses), potentially overloading the BDE (as it cannot handle more than a set number of threads or processes).

An alternative approach is offered by the support in DataSnap for object pooling. All you need to do to request this feature is add a call to `RegisterPooled` in the overridden `UpdateRegistry` method. Combined with the stateless support built into this architecture, the pooling capability allows you to share some middle-tier objects among a much larger number of clients.

The users on the client computers will spend most of their time reading data and typing in updates, and they generally don’t continue asking for data and sending updates. When the client is not calling a method of the middle-tier object, this can be used for another client. Being stateless, in fact, every request reaches the middle tier as a brand-new operation, even when a server is dedicated to a specific client.

Pooling mechanisms are built into MTS and CORBA, but DataSnap makes it available also for HTTP and socket-based connections, and for the Internet Express Web client.

## Customizing the Data Packets

There are many ways to include custom information within the data packet handled by the `IAppServer` interface. The simplest is probably to handle the `OnGetDataSetProperties` event of the provider itself. This event has a `Sender` parameter, a dataset parameter indicating where the data is coming from, and an `OleVariant` array `Properties` parameter, in which you can place the extra information. You need to define one variant array for each extra property and include the name of the extra property, its value, and whether you want the data to return to the server along with the update delta (the `IncludeInDelta` parameter).

Of course, you can pass properties of the related dataset component, but you can also pass any other value (extra fake properties). In the `AppSPlus` example, I pass to the client the time the query was executed and its parameters:

```
procedure TAppServerPlus.ProviderQueryGetDataSetProperties(  
    Sender: TObject; DataSet: TDataSet; out Properties: OleVariant);  
begin  
    Properties := VarArrayCreate([0,1], varVariant);  
    Properties[0] := VarArrayOf(['Time', Now, True]);  
    Properties[1] := VarArrayOf(['Param', Query.Params[0].AsString, False]);  
end;
```

On the client side, the `ClientDataSet` component has a `GetOptionalParameter` method to retrieve the value of the extra property with the given name. The `ClientDataSet` also has the `SetOptionalParameter` method to add more properties to the dataset. These values will be saved to disk (in the briefcase model) and eventually sent back to the middle tier (by setting the `IncludeInDelta` member of the variant array to `True`). Here is a simple example of the retrieval of the dataset in the code above:

```
Caption := 'Data sent at ' + TimeToStr (TDateTime (  
    cdsQuery.GetOptionalParam('Time')));  
Label1.Caption := 'Param ' + cdsQuery.GetOptionalParam('Param');
```

The effect of this code was visible in Figure 17.5. An alternative and more powerful approach for customizing the data packet sent to the client is to handle the `OnGetData` event of the provider, which receives the outgoing data packet in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client. For example, you might encode some of the data or filter out sensitive records.

## What's Next?

Borland originally introduced its multitier technology in Delphi 3 and has kept extending it from version to version. In addition to further updates and the change of the MIDAS name to DataSnap, Delphi 6 sees the introduction of XML and SOAP support, introducing an alternate and extended architecture for multitier applications. We'll fully explore this topic in Chapter 23.

For the moment, we'll continue with database programming, discussing data-aware controls and custom datasets. In the next part of the book we'll explore COM, sockets, and Internet programming, getting to XML and SOAP at the end of the book, after we've discussed a lot of foundation material.

# Writing Database Components

- Data-aware components: the data link
- Field-oriented data-aware controls
- Data-aware TrackBar and ProgressBar
- Record-oriented data-aware controls
- A record viewer
- Building custom datasets
- Saving a dataset to a local stream

In Chapter 11, “Creating Components,” we explored the development of Delphi components in depth. Now that I’ve discussed database programming, we can get back to the earlier topic and focus on the development of database-related components.

There are basically two families of such components. There are data-aware controls you can use to present the data of a field or an entire record to the users of a program. There are dataset components you can define to provide data to existing data-aware controls, reading it from a database or any other data source. In this chapter, I’ll cover both topics.

## The Data Link

When you write a Delphi database program, you generally connect some data-aware controls to a `DataSource` component, and then connect the `DataSource` component to a dataset. The connection between the data-aware control to the `DataSource` is called a *data link* and is represented by an object of class `TDataLink`. The data-aware control creates and manages this object and represents its only connection to the data. From a more practical perspective, to make a component data-aware, you need to add a data link to it and surface some of the properties of this internal object, such as the `DataSource` and `DataField` properties.

Delphi uses the `DataSource` and `DataLink` objects for bidirectional communication. The dataset uses the connection to notify the data-aware controls that new data is available (because the dataset has been activated, or the current record has changed, and so on). Data-aware controls use the connection to ask for the current value of a field or to update it, notifying the dataset of this event.

The relations among all these components are complicated by the fact that some of the connections can be one-to-many. For example, you can connect multiple data sources to the same dataset, and you generally have multiple data links to the same data source, simply because you need one link for every data-aware component, and in most cases you connect multiple data-aware controls to each data source.

## The `TDataLink` Class

We’ll work for much of this chapter with `TDataLink` and its derived classes, which are defined in the DB unit. This class has a set of protected virtual methods, which have a role similar to events. They are “almost-do-nothing” methods you can override in a specific subclass to intercept user operations and other data-source events. Here is a list, extracted from the source code of the class:

```
type
  TDataLink = class(TPersistent)
  protected
    procedure ActiveChanged; virtual;
```

```
procedure CheckBrowseMode; virtual;  
procedure DataSetChanged; virtual;  
procedure DataSetScrolled(Distance: Integer); virtual;  
procedure FocusControl(Field: TFieldRef); virtual;  
procedure EditingChanged; virtual;  
procedure LayoutChanged; virtual;  
procedure RecordChanged(Field: TField); virtual;  
procedure UpdateData; virtual;
```

All of these virtual methods are called by the `DataEvent` private method, a sort of window procedure for a data source, a procedure triggered by several data events (see the `TDataEvent` enumeration). These events originate in the dataset, fields, or data source, and are generally applied to a dataset. The `DataEvent` method of the dataset component dispatches the events to the connected data sources. Each data source calls the `NotifyDataLinks` method to forward the event to each connected data link, and then the data source triggers either its own `OnDataChange` or `OnUpdateData` event.

## Derived DataLink Classes

The `TDataLink` class is not technically an abstract class, but you'll seldom use it directly. When you need to create data-aware controls, you'll need to use one of its derived classes or derive a new one yourself. The most important class derived from `TDataLink` is the `TFieldDataLink` class, which is used by data-aware controls that relate to a single field of the dataset. Most data-aware controls fall into this category, and the `TFieldDataLink` class solves the most common problems of this type of component.

All of the table- or record-oriented data-aware controls define specific subclasses of `TDataLink`, as we'll do later on. The `TFieldDataLink` class has a list of events corresponding to the virtual methods of the base class it overrides. This makes the class simpler to customize, as you can use event handlers instead of having to inherit a new class from it. Here's an example of an overridden method, which fires the corresponding event, if available:

```
procedure TFieldDataLink.ActiveChanged;  
begin  
    UpdateField;  
    if Assigned(FOnActiveChange) then FOnActiveChange(Self);  
end;
```

The `TFieldDataLink` class contains also the `Field` and `FieldName` properties that let you connect the data-aware control to a specific field of the dataset. The link keeps also a reference to the current visual component, using the `Control` property.

## Writing Field-Oriented Data-Aware Controls

Now that you understand the theory of how the data link classes work, I can start building some data-aware controls. The first two examples I'll build are data-aware versions of the `ProgressBar` and `TrackBar` common controls. We can use the first to display a numeric value, such as a percentage, in a visual way. We can use the second to allow a user to change the numeric value as well.

### A Read-Only ProgressBar

A data-aware version of the `ProgressBar` control is a relatively simple case of a data-aware control, because it is a read-only control. This component is derived from the version that's not data-aware and adds a few properties of the data link object it encapsulates:

```

type
  TmdbProgress = class(TProgressBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
  protected
    // data link event handler
    procedure DataChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;

```

As with every data-aware component that connects to a single field, this control makes available the `DataSource` and `DataField` properties. There is very little code to write here; simply export the properties from the internal data link object, as follows:

```

function TmdbProgress.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TmdbProgress.SetDataField (Value: string);
begin

```

```
    FDataLink.FieldName := Value;
end;

function TMDdbProgress.GetDataSource: TDataSource;
begin
    Result := FDataLink.DataSource;
end;

procedure TMDdbProgress.SetDataSource (Value: TDataSource);
begin
    FDataLink.DataSource := Value;
end;

function TMDdbProgress.GetField: TField;
begin
    Result := FDataLink.Field;
end;
```

Of course, to make this component work, you must create and destroy the data link when the component itself is created or destroyed:

```
constructor TMDdbProgress.Create (AOwner: TComponent);
begin
    inherited Create (AOwner);
    FDataLink := TFieldDataLink.Create;
    FDataLink.Control := self;
    FDataLink.OnDataChange := DataChange;
end;

destructor TMDdbProgress.Destroy;
begin
    FDataLink.Free;
    FDataLink := nil;
    inherited Destroy;
end;
```

In the preceding constructor, notice that the component installs one of its own methods as an event handler for the data link. This is where the most important code of the component resides. Every time the data changes, we modify the output of the progress bar to reflect the values of the current field:

```
procedure TMDdbProgress.DataChange (Sender: TObject);
begin
    if (FDataLink.Field <> nil) and (FDataLink.Field is TNumericField) then
        Position := FDataLink.Field.AsInteger
    else
        Position := Min;
end;
```

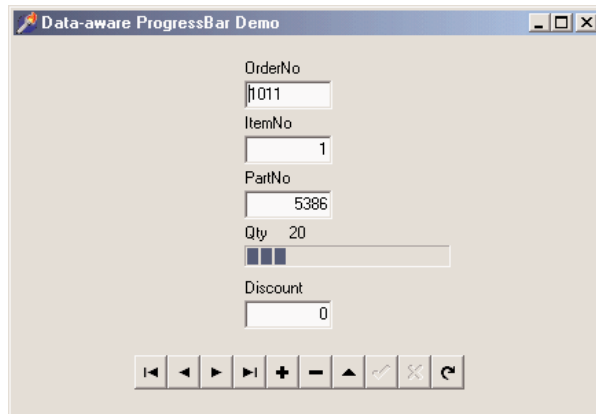


Following the convention of the VCL data-aware controls, if the field type is invalid, the component doesn't display an error message—it simply disables the output. Alternatively, you might want to check the field type when `SetDataField` method assigns it to the control.

In Figure 18.1 you can see an example of the `DbProgr` application's output, which uses both a label and a progress bar to display an order's quantity information. Thanks to this visual clue, you can step through the records and easily spot orders for many items. One obvious benefit to this component is that the application contains almost no code, since all the important code is in the component itself.

**FIGURE 18.1:**

The data-aware  
ProgressBar in action  
in the `DbProgr` example



As you've seen, a read-only data-aware component is not too difficult to write. It gets extremely complex, on the other hand, to use such a component inside a `DBCtrlGrid` container.

## NOTE

If you remember the discussion of the `Notification` method in Chapter 11, you might wonder what happens if the data source referenced by the data-aware control is destroyed. The good news is that the data source has a destructor that removes itself from its own data links. So there is no need for a `Notification` method for data-aware controls, even though you'll see books and articles suggesting it, and VCL has plenty of this extra useless code.

## Replicable Data-Aware Controls

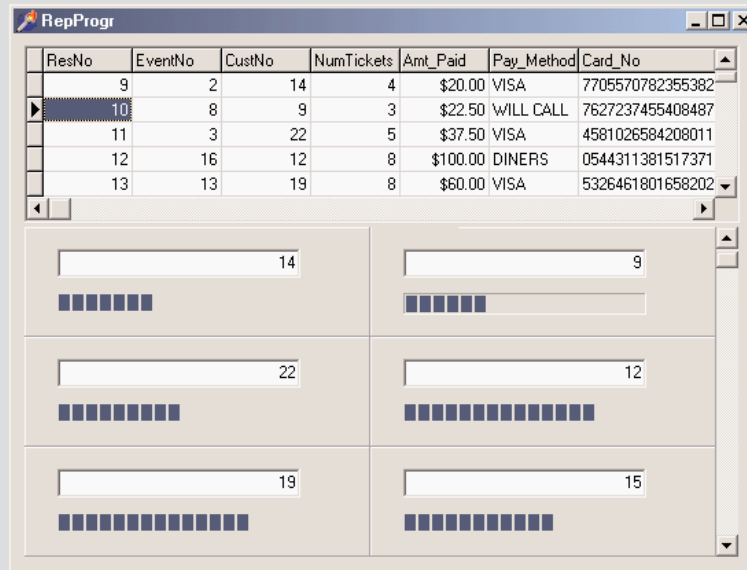
Extending a data-aware control to support its use inside a `DBCtrlGrid` component is rather complex and not well documented. You can find a complete "replicable" version of the progress bar in the `MdDataPack` package and an example of its use in the `RepProgr` folder, along with an HTML file describing its development. The `DBCtrlGrid` component has a peculiar behavior, as it displays on screen multiple versions of the same physical control, using some

*Continued on next page*

“smoke and mirrors.” The grid can attach the control to a data buffer other than the current record and redirects the control paint operations to another portion of the monitor.

In short, to appear in the DBCtrlGrid, a component must have its `csReplicable` control style set, a flag merely indicating that your component actually supports being hosted by a control grid. First, the component must respond to the `cm_GetDataLink` Delphi message and return a pointer to the data link, so that the control grid can use and change it. Second, it needs a custom `Paint` method to draw the output in the appropriate canvas object, which is provided in a parameter of the `wm_Paint` message in case the `csPaintCopy` flag of the `ControlState` property is set.

The actual code of the example is rather complex, and the `DBCtrlGrid` component is not heavily used, so I decided not to give you full details here, but you can find the full code and some more information in the source code on the companion CD. Here’s the output of a test program that uses this component:



## A Read-Write TrackBar

The next step is to write a component that allows a user to modify the data in a database, not just browse it. The overall structure of this type of component isn’t very different from the previous version, but there are a few extra elements. In particular, when the user starts interacting with the component, the code should put the dataset into edit mode and then notify

the dataset that the data has changed. The dataset will then use an event handler of the `FieldDataLink` to ask for the updated value.

To demonstrate how you can create a data-aware component that modifies the data, I've decided to extend the `TrackBar` control. This probably isn't the simplest example, but it demonstrates several important techniques.

Here's the definition of the component's class:

```

type
  TMdDbTrack = class(TTrackBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
    procedure CMExit(var Message: TCMExit); message CM_EXIT;
  protected
    // data link event handlers
    procedure DataChange (Sender: TObject);
    procedure UpdateData (Sender: TObject);
    procedure ActiveChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;

```

Compared to the read-only data-aware control built earlier, this class is a bit more complex, because it has three message handlers, including component notification handlers, and two new event handlers for the data link. The component installs these event handlers in the constructor, which also disables the component:

```

constructor TMdDbTrack.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := self;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
  FDataLink.OnActiveChange := ActiveChange;
  Enabled := False;
end;

```

All of the get and set methods and the `DataChange` event handler are very similar to those in the `TMDbProgress` component. The only difference is that whenever the data source or data field changes, the component checks the current status to see whether it should enable itself:

```
procedure TMDbTrack.SetDataSource (Value: TDataSource);  
begin  
    FDataLink.DataSource := Value;  
    Enabled := FDataLink.Active and (FDataLink.Field <> nil) and  
        not FDataLink.Field.ReadOnly;  
end;
```

This code tests three conditions: the data link should be active, the link should refer to an actual field, and the field shouldn't be read-only. When the user changes the field, the component should also consider that the field name might be invalid; to test for this condition, the component should rather use a `try/finally` block:

```
procedure TMDbTrack.SetDataField (Value: string);  
begin  
    try  
        FDataLink.FieldName := Value;  
    finally  
        Enabled := FDataLink.Active and (FDataLink.Field <> nil) and  
            not FDataLink.Field.ReadOnly;  
    end;  
end;
```

The control executes the same test when the dataset is enabled or disabled:

```
procedure TMDbTrack.ActiveChange (Sender: TObject);  
begin  
    Enabled := FDataLink.Active and (FDataLink.Field <> nil) and  
        not FDataLink.Field.ReadOnly;  
end;
```

The most interesting portion of this component's code is related to its user interface. When a user starts moving the scroll thumb, the component should do the following: put the dataset into edit mode, let the base class update the thumb position, and alert the data link (and therefore the data source) that the data has changed. Here's the code:

```
procedure TMDbTrack.CNHScroll(var Message: TWMHScroll);  
begin  
    // enter edit mode  
    FDataLink.Edit;  
    // update data  
    inherited;  
    // let the system know  
    FDataLink.Modified;  
end;
```

```

procedure TmdbTrack.CNVScroll(var Message: TMMVScroll);
begin
    // enter edit mode
    FDataLink.Edit;
    // update data
    inherited;
    // let the system know
    FDataLink.Modified;
end;

```

When the dataset needs new data—for example, to perform a Post operation—it simply requests it from the component via the TFieldDataLink class’s OnUpdateData event:

```

procedure TmdbTrack.UpdateData (Sender: TObject);
begin
    if (FDataLink.Field <> nil) and (FDataLink.Field is TNumericField) then
        FDataLink.Field.AsInteger := Position;
end;

```

If the proper conditions are met, the component simply updates the data in the proper table field. Finally, if the component loses the input focus, it should force a data update (if the data has changed) so that any other data-aware components showing the value of that field will display the correct value as soon as the user moves to a different field. If the data hasn’t changed, the component won’t bother updating the data in the table. This is the standard CmExit code for components used by VCL and borrowed for our component as well:

```

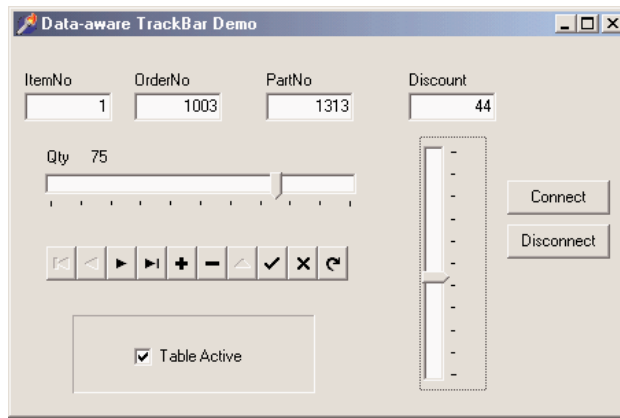
procedure TmdbTrack.CmExit(var Message: TCmExit);
begin
    try
        FDataLink.UpdateRecord;
    except
        SetFocus;
        raise;
    end;
    inherited;
end;

```

Again, there is a demo program for testing this component; you can see its output in Figure 18.2. The DbTrack program contains a check box to enable and disable the table, the visual components, and a couple of buttons you can use to detach the vertical TrackBar component from the field it relates to. Again, I placed these on the form to test enabling and disabling the track bar.

**FIGURE 18.2:**

The DbTrack example has a couple of track bars you can use to enter data in a database table. The check box and buttons are used to test the enabled status of the components.



## Creating Custom Data Links

The data-aware controls I've built up to this point all referred to specific fields of the dataset, so I was able to use a `TFieldDataLink` object to establish the connection with a data source. Now I want to build a data-aware component that works with a dataset as a whole, a simple record viewer.

Delphi's database grid shows the value of several fields and several records simultaneously. In my record viewer component, I want to list all the fields of the current record, using a customized grid. This example will show you how to build a customized grid control, and a custom data link to go with it.

### A Record Viewer Component

In Delphi there are no data-aware components that manipulate multiple fields of a single record, without displaying other records. In fact, the only component that displays multiple fields from the same table is the `DBGrid`, which displays multiple fields and multiple records.

The record viewer component I'm going to describe in this section is based on a two-column grid; the first column displays the table's field names, while the second column displays the corresponding field values. The number of rows in the grid will correspond to the number of fields, with a vertical scroll bar in case they can't fit in the visible area.

The data link we need in order to build this component is a simple class, connected only to the record viewer component, and declared directly in the implementation portion of its

unit. This is the same approach used by VCL for some specific data links. Here's the definition of the new class:

```

type
  TmDRecordLink = class (TDataLink)
  private
    RView: TmDRecordView;
  public
    constructor Create (View: TmDRecordView);
    procedure ActiveChanged; override;
    procedure RecordChanged (Field: TField); override;
  end;

```

As you can see, the class overrides the methods related to the principal event, in this case simply the activation and data (or record) change. Alternatively, I could have exported some events and then let the component handle them. That's what the `TFieldDataLink` does, but the approach I've taken makes more sense for a data link class, because you'll want to use it with different data-aware components. The constructor requires the associated component as its only parameter:

```

constructor TmDRecordLink.Create (View: TmDRecordView);
begin
  inherited Create;
  RView := View;
end;

```

After storing a reference to the associated component, the other methods can operate on it directly:

```

procedure TmDRecordLink.ActiveChanged;
var
  I: Integer;
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // repaint all...
  RView.Invalidate;
end;

procedure TmDRecordLink.RecordChanged;
begin
  inherited;
  // repaint all...
  RView.Invalidate;
end;

```

As you've seen, the record link code is very simple. Most of the difficulties in building this example depend on the use of a grid. To avoid dealing with useless properties, I've derived the record viewer grid from the `TCustomGrid` class. This class incorporates much of the code

for grids, but most of its properties, events, and methods are protected. For this reason, the class declaration is quite long, because it needs to publish many existing properties. Here is an excerpt (excluding the base class properties):

```

type
  TmRecordView = class(TCustomGrid)
  private
    // data-aware support
    FDataLink: TDataLink;
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
  protected
    // redefined TCustomGrid methods
    procedure DrawCell (ACol, ARow: Longint; ARect: TRect;
      AState: TGridDrawState); override;
    procedure ColWidthsChanged; override;
    procedure RowHeightsChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure SetBounds (ALeft, ATop, AWidth, AHeight: Integer); override;
    procedure DefineProperties (Filer: TFile); override;
    // public parent properties (omitted...)
  published
    // data-aware properties
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    // published parent properties (omitted...)
  end;

```

Besides redeclaring the properties to publish them, the component defines a data link object and the DataSource property. There's no DataField property for this component, because it refers to an entire record. The component's constructor is very important. It sets the values of many unpublished properties, including the grid options:

```

constructor TmRecordView.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TmRecordLink.Create (self);
  // set numbers of cells and fixed cells
  RowCount := 2; // default
  ColCount := 2;
  FixedCols := 1;
  FixedRows := 0;
  Options := [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goRowSizing];
  DefaultDrawing := False;
  ScrollBars := ssVertical;
  FSaveCellExtents := False;
end;

```



The grid has two columns, one of them fixed, and no fixed rows. The fixed column is used for resizing each row of the grid. Unfortunately, a user cannot drag the fixed row to resize the columns, because you can't resize fixed elements, and the grid already has a fixed column.

**NOTE**

An alternative approach could be to have an extra empty column, as the DBGrid control does. You'd be able to resize the two other columns after adding a fixed row. Overall, though, I prefer my implementation.

I've used an alternative approach to resize the columns. The first column (holding the field names) can be resized either using programming code or visually at design time, and the second column (holding the values of the fields) will be resized to use the remaining area of the component, leaving space for the borders, lines, and vertical scrollbar:

```
procedure TMdRecordView.SetBounds (ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited;
  ColWidths [1] := Width - ColWidths [0] - GridLineWidth * 3 -
    GetSystemMetrics (sm_CXVScroll) - 2; // border
end;
```

This takes place when the component size changes and when either of the columns change. With this code, the `DefaultColWidth` property of the component becomes, in practice, the fixed width of the first column.

After everything has been set up, the key method of the component is the overridden `DrawCell` method, detailed in Listing 18.1. This is where the control displays the information about the fields and their values. There are three things it needs to draw. If the data link is not connected to a data source, the grid displays an “empty element” sign (`[]`). When drawing the first column, the record viewer shows the `DisplayName` of the field, which is the same value used by the DBGrid for the heading. When drawing the second column, the component accesses the textual representation of the field value, extracted with the `DisplayText` property (or with the `AsString` property for memo fields).

**Listing 18.1: The DrawCell method of the custom RecordView component**

```
procedure TMdRecordView.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
var
  Text: string;
  CurrField: TField;
  Bmp: TBitmap;
begin
  CurrField := nil;
  Text := '[]'; // default
  // paint background
```

```

if (ACol = 0) then
    Canvas.Brush.Color := FixedColor
else
    Canvas.Brush.Color := Color;
    Canvas.FillRect (ARect);
    // leave small border
    InflateRect (ARect, -2, -2);
if (FDataLink.DataSource <> nil) and FDataLink.Active then
begin
    CurrField := FDataLink.DataSet.Fields[ARow];
    if ACol = 0 then
        Text := CurrField.DisplayName
    else if CurrField is TMemoField then
        Text := TMemoField (CurrField).AsString
    else
        Text := CurrField.DisplayText;
    end;
if (ACol = 1) and (CurrField is TGraphicField) then
begin
    Bmp := TBitmap.Create;
    try
        Bmp.Assign (CurrField);
        Canvas.StretchDraw (ARect, Bmp);
    finally
        Bmp.Free;
    end;
end
else if (ACol = 1) and (CurrField is TMemoField) then
begin
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_WordBreak or dt_NoPrefix)
end
else // draw single line vertically centered
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_vcenter or dt_SingleLine or dt_NoPrefix);
if gdFocused in AState then
    Canvas.DrawFocusRect (ARect);
end;

```

The final portion of the method is where the component considers memo and graphic fields. If the field is a `TMemoField`, the `DrawText` function call doesn't specify the `dt_SingleLine` flag, but uses `dt_WordBreak` flag to wrap the words when there's no more room. For a graphic field, of course, the component uses a completely different approach, assigning the field image to a temporary bitmap, and then stretching it to fill the surface of the cell.

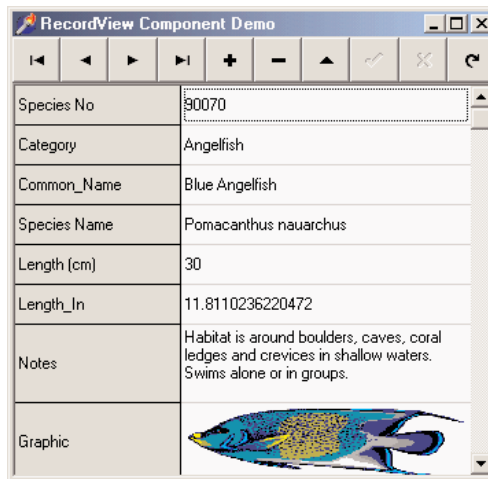
Notice also that the component sets the `DefaultDrawing` property to `False`, so that it's also responsible for drawing the background and the focus rectangle, as it does in the `DrawCell` method. The component also calls the `InflateRect` API function to leave a small area

between the cell border and the output text. The actual output is produced by calling another Windows API function, `DrawText`, which centers the text vertically in its cell.

This drawing code works both at run time, as you can see in Figure 18.3, and at design time. The output may not be perfect, but this component can certainly be very useful in many cases. If you want to display the data for a single record, instead of building a custom form with labels and data-aware controls, you can easily use this record viewer grid. Of course, it's important to remember that the record viewer is a read-only component: it's certainly possible to extend it to add editing capabilities (they're already part of the `TCustomGrid` class). However, instead of adding this support, we've decided to make the component more complete by adding support for displaying BLOB fields.

**FIGURE 18.3:**

The `ViewGrid` example demonstrates the output of the `RecordView` component, using Borland's sample `BioLife` database table.



To improve the graphical output, the control makes the lines for those fields twice as high as those for plain text fields. This operation is accomplished when the dataset connected to the data-aware control is activated. The `ActiveChanged` method of the data link is triggered also by the `RowHeightsChanged` methods, connected to the `DefaultRowHeight` property of the base class:

```
procedure TMdRecordLink.ActiveChanged;
var
  I: Integer;
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // double the height of memo and graphics
  for I := 0 to DataSet.FieldCount - 1 do
    if DataSet.Fields [I] is TBlobField then
```

```

    RView.RowHeights [I] := RView.DefaultRowHeight * 2;
    // repaint all...
    RView.Invalidate;
end;

```

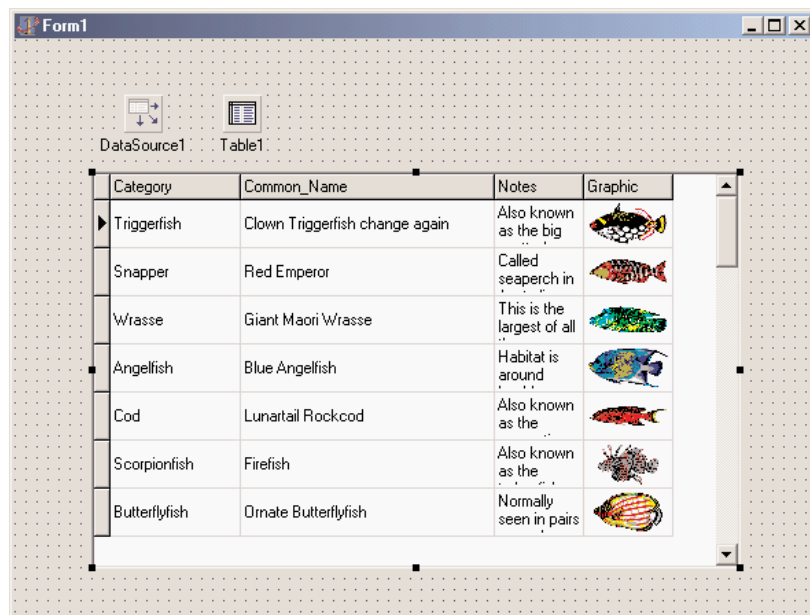
At this point, we stumble into a minor problem. In the `DefineProperties` method, the `TCustomGrid` class saves the values of the `RowHeights` and `ColHeights` properties. We could disable this streaming by overriding the method and not calling `inherited` (which is generally a bad technique to use), but it is also possible to toggle the `FSaveCellExtents` protected field to disable this feature.

## Customizing the DBGrid Component

Besides writing brand-new custom data-aware components, it's common for Delphi programmers to customize the DBGrid control. The goal for the next component is to enhance the DBGrid with the same kind of custom output I've used for the RecordView component, directly displaying graphic and memo fields. To do this, the grid needs to make the row height resizable, to allow space for a reasonable amount of text and big enough for graphics. You can see an example of this grid at design time in Figure 18.4.

**FIGURE 18.4:**

An example of the `MdDbGrid` component at design time. Notice the output of the graphics and memo fields.



While creating the output was a simple matter of adapting the code used in the record viewer component, setting the height of the grid cells ended up being a very difficult problem to solve. The lines of code you'll see for that operation may be few, but they cost me hours of work!

**NOTE**

Unlike the generic grid we've used above, a DBGrid is a virtual view on the dataset—there is no relation between the number of rows shown on the screen and the number of rows of data in the dataset. When you scroll up and down through the data records of the dataset, you are not scrolling up and down through the rows of the DBGrid; the rows are stationary while the data moves from one row to the next to give the appearance of movement. For this reason, the program doesn't try to set the height of an individual row to suit its data, but it sets the height of all the data rows to a multiline height value.

This time the control doesn't have to create a custom data link, because it is deriving from a component that already has a complex connection with the data. The new class has a new property to specify the number of lines of text for each row and overrides a few virtual methods:

```

type
  TMDdbGrid = class(TDbGrid)
  private
    FLinesPerRow: Integer;
  procedure SetLinesPerRow (Value: Integer);
  protected
  procedure DrawColumnCell(const Rect: TRect; DataCol: Integer;
    Column: TColumn; State: TGridDrawState); override;
  procedure LayoutChanged; override;
  public
  constructor Create (AOwner: TComponent); override;
  published
  property LinesPerRow: Integer
    read FLinesPerRow write SetLinesPerRow default 1;
  end;

```

The constructor simply sets the default value for the FLinesPerRow field. Here is the set method for the property:

```

procedure TMDdbGrid.SetLinesPerRow(Value: Integer);
begin
  if Value <> FLinesPerRow then
  begin
    FLinesPerRow := Value;
    LayoutChanged;
  end;
end;

```

The side effect of changing the number of lines is a call to the `LayoutChanged` virtual method. The system calls this method frequently when one of the many output parameters changes. In the code of this method, the component first calls the inherited version and then sets the height of each row. As a basis for this computation it uses the same formula of the `TCustomDBGrid` class: the text height is calculated using the sample word *Wg* in the current font (this text is used because it includes both a full-height uppercase character and a lower-case letter with a descender). Here's the code:

```

procedure TmDbGrid.LayoutChanged;
var
  PixelsPerRow, PixelsTitle, I: Integer;
begin
  inherited LayoutChanged;

  Canvas.Font := Font;
  PixelsPerRow := Canvas.TextHeight('Wg') + 3;
  if dgRowLines in Options then
    Inc (PixelsPerRow, GridLineWidth);

  Canvas.Font := TitleFont;
  PixelsTitle := Canvas.TextHeight('Wg') + 4;
  if dgRowLines in Options then
    Inc (PixelsTitle, GridLineWidth);

  // set number of rows
  RowCount := 1 + (Height - PixelsTitle) div (PixelsPerRow * FLinesPerRow);

  // set the height of each row
  DefaultRowHeight := PixelsPerRow * FLinesPerRow;
  RowHeights [0] := PixelsTitle;
  for I := 1 to RowCount - 1 do
    RowHeights [I] := PixelsPerRow * FLinesPerRow;
end;

```

**WARNING** `Font` and `TitleFont` are the grid defaults that can be overridden by properties of the individual `DBGrid` column objects. This component would currently ignore those settings.

The difficult part here was to get the last four statements correct. You can simply set the `DefaultRowHeight` property, but in that case the title row will probably be too high. At first, I tried setting the `DefaultRowHeight` and then the height of the first row, but this complicated the code used to compute the number of visible rows in the grid (the read-only `VisibleRowCount` property). If you specify the number of rows (in order to avoid having rows hidden beneath the lower edge of the grid), the base class keeps recomputing them.

Finally, here's the code used to draw the data, ported from the RecordView component and adapted slightly for the grid:

```

procedure TMDbGrid.DrawColumnCell (const Rect: TRect; DataCol: Integer;
  Column: TColumn; State: TGridDrawState);
var
  Bmp: TBitmap;
  OutRect: TRect;
begin
  if FLinesPerRow = 1 then
    inherited DrawColumnCell(Rect, DataCol, Column, State)
  else
    begin
      // clear area
      Canvas.FillRect (Rect);
      // copy the rectangle
      OutRect := Rect;
      // restrict output
      InflateRect (OutRect, -2, -2);
      // output field data
      if Column.Field is TGraphicField then
        begin
          Bmp := TBitmap.Create;
          try
            Bmp.Assign (Column.Field);
            Canvas.StretchDraw (OutRect, Bmp);
          finally
            Bmp.Free;
          end;
        end
      else if Column.Field is TMemoField then
        begin
          DrawText (Canvas.Handle, PChar (Column.Field.AsString),
            Length (Column.Field.AsString), OutRect, dt_WordBreak or dt_NoPrefix)
        end
      else // draw single line vertically centered
        DrawText (Canvas.Handle, PChar (Column.Field.DisplayText),
          Length (Column.Field.DisplayText), OutRect,
            dt_vcenter or dt_SingleLine or dt_NoPrefix);
    end;
  end;

```

In the code above you can see that if the user displays just a single line, the grid uses the standard drawing technique with no output for memo and graphic fields. However, as soon as you increase the line count, you'll see a better output.

To see this code in action, run the GridDemo example. This program has two buttons you can use to increase or decrease the row height of the grid, and two more buttons to change the font. This is an important test because the height in pixels of each cell is the height of the font multiplied by the number of lines.

## Building Custom Datasets

When discussing the `TDataSet` class and the alternative families of dataset components available in Delphi, in Chapter 13, “Delphi’s Database Architecture,” I mentioned the possibility of writing a custom dataset class. Now it’s time to have a look at an actual example. The reasons for writing a custom dataset relate to the fact that you won’t need to deploy a database engine but you’ll still be able to take full advantage of Delphi’s database architecture, including things like persistent database fields and data-aware controls.

Writing a custom dataset is one of the most complex tasks for a component developer, so this is one of the most advanced areas (as far as low-level coding practices, including tons of pointers) of the entire book. Moreover, Borland hasn’t released any official documentation on writing custom datasets. If you are early in your experience with Delphi, you might want to skip the rest of this chapter and come back here later.

The `TDataSet` class is an abstract class, which declares several virtual abstract methods—23 to be precise. Every subclass of `TDataSet` must override all of those methods.

Before discussing the development of a custom dataset, we need to explore a few technical elements of the `TDataSet` class, in particular record buffering. The class maintains a list of buffers, which store the values of different records. These buffers store the actual data, but they also usually store further information for the dataset to use when managing the records. These buffers don’t have a predefined structure, and each custom dataset must allocate the buffers, fill them, and destroy them. The custom dataset must also copy the data from the record buffers to the various fields of the dataset, and vice versa. In other words, the custom dataset is entirely responsible for handling these buffers.

In addition to managing the data buffers, the component is also responsible for navigating among the records, managing the bookmarks, defining the structure of the dataset, and creating the proper data fields. The `TDataSet` class is nothing more than a framework; you must fill it with the appropriate code. Fortunately, most of the code follows a standard structure, which the `TDataSet`-derived VCL classes use. Once you’ve grasped the key ideas, you’ll be able to build multiple custom datasets borrowing quite a lot of code.

To simplify this type of reuse, I’ve collected the common features required by any custom dataset in a `TMDCustomDataSet` class. However, I’m not going to discuss the base class first



and the specific implementation later, because that would probably be rather complex to understand. Instead, I'll detail the code required by a dataset, presenting methods of the generic and specific classes at the same time, according to a logical flow.

## The Definition of the Classes

The starting point, as usual, is the declaration of the two classes discussed in this section, the generic custom dataset I've written and the specific component storing data in a stream. These declarations of these classes is available in Listing 18.2. Besides virtual methods, the classes contain a series of protected fields used to manage the buffers, track the current position and record count, and handle many other features. You should also notice another record declaration at the beginning, a structure used to store the extra data for every data record we place in a buffer. The dataset places this information in each record buffer, following the actual data.



### Listing 18.2: The declaration of TMdCustomDataSet and TMdDataSetStream

```
// in the unit MdDsCustom
type
  EMdDataSetError = class (Exception);

  TMdRecInfo = record
    Bookmark: Longint;
    BookmarkFlag: TBookmarkFlag;
  end;
  PMdRecInfo = ^TMdRecInfo;

  TMdCustomDataSet = class(TDataSet)
  protected
    // status
    FIsTableOpen: Boolean;
    // record data
    FRecordSize,           // the size of the actual data
    FRecordBufferSize,   // data + housekeeping (TRecInfo)
    FCurrentRecord,       // current record (0 to FRecordCount - 1)
    BofCrack,             // before the first record (crack)
    EofCrack: Integer;    // after the last record (crack)
    // create, close, and so on
  procedure InternalOpen; override;
  procedure InternalClose; override;
  function IsCursorOpen: Boolean; override;
  // custom functions
  function InternalRecordCount: Integer; virtual; abstract;
  procedure InternalPreOpen; virtual;
  procedure InternalAfterOpen; virtual;
  procedure InternalLoadCurrentRecord(Buffer: PChar); virtual; abstract;
  // memory management
```

```
function AllocRecordBuffer: PChar; override;  
procedure InternalInitRecord(Buffer: PChar); override;  
procedure FreeRecordBuffer(var Buffer: PChar); override;  
function GetRecordSize: Word; override;  
// movement and optional navigation (used by grids)  
function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):  
    TGetResult; override;  
procedure InternalFirst; override;  
procedure InternalLast; override;  
function GetRecNo: Longint; override;  
function GetRecordCount: Longint; override;  
procedure SetRecNo(Value: Integer); override;  
// bookmarks  
procedure InternalGotoBookmark(Bookmark: Pointer); override;  
procedure InternalSetToRecord(Buffer: PChar); override;  
procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;  
procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;  
procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;  
function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;  
// editing (dummy versions)  
procedure InternalDelete; override;  
procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;  
procedure InternalPost; override;  
procedure InternalInsert; override;  
// other  
procedure InternalHandleException; override;  
published  
// redeclared dataset properties  
property Active;  
property BeforeOpen;  
property AfterOpen;  
property BeforeClose;  
property AfterClose;  
property BeforeInsert;  
property AfterInsert;  
property BeforeEdit;  
property AfterEdit;  
property BeforePost;  
property AfterPost;  
property BeforeCancel;  
property AfterCancel;  
property BeforeDelete;  
property AfterDelete;  
property BeforeScroll;  
property AfterScroll;  
property OnCalcFields;  
property OnDeleteError;  
property OnEditError;  
property OnFilterRecord;  
property OnNewRecord;  
property OnPostError;
```

```

    end;

// in the unit MdDsStream
type
    TMdDataFileHeader = record
        VersionNumber: Integer;
        RecordSize: Integer;
        RecordCount: Integer;
    end;

    TMdDataSetStream = class(TMdCustomDataSet)
    private
        procedure SetTableName(const Value: string);
    protected
        FDataFileHeader: TMdDataFileHeader;
        FDataFileHeaderSize, // optional file header size
        FRecordCount: Integer; // current number of records
        FStream: TStream; // the physical table
        FTableName: string; // table path and file name
        FFieldOffset: TList; // field offsets in the buffer
    protected
        // open and close
        procedure InternalPreOpen; override;
        procedure InternalAfterOpen; override;
        procedure InternalClose; override;
        procedure InternalInitFieldDefs; override;
        // edit support
        procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
        procedure InternalPost; override;
        procedure InternalInsert; override;
        // fields
        procedure SetFieldData(Field: TField; Buffer: Pointer); override;
        // custom dataset virtual methods
        function InternalRecordCount: Integer; override;
        procedure InternalLoadCurrentRecord(Buffer: PChar); override;
    public
        procedure CreateTable;
        function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
    published
        property TableName: string read FTableName write SetTableName;
    end;

```

---

In dividing the methods into sections (as you can see by looking at the source code files), I've marked each one with a roman number. You'll see those numbers in a comment describing the method, so that while browsing this long listing you'll immediately know which of the three sections you are in.

## Section I: Initialization, Opening, and Closing

The first methods I'll examine are responsible for initializing the dataset, and for opening and closing the file stream we'll use to store the data. In addition to initializing the component's internal data, these methods are responsible for initializing and connecting the proper `TFields` objects to the dataset component. To make this work, all we need to do is to initialize the `FieldsDef` property with the definitions of the fields for our dataset, then call a few standard methods to generate and bind the `TField` objects. This is the general `InternalOpen` method:

```
procedure TMDCustomDataSet.InternalOpen;
begin
    InternalPreOpen; // custom method for subclasses

    // initialize the field definitions
    InternalInitFieldDefs;

    // if there are no persistent field objects, create the fields dynamically
    if DefaultFields then
        CreateFields;
    // connect the TField objects with the actual fields
    BindFields (True);

    InternalAfterOpen; // custom method for subclasses

    // sets cracks and record position and size
    BofCrack := -1;
    EofCrack := InternalRecordCount;
    FCurrentRecord := BofCrack;
    FRecordBufferSize := FRecordSize + sizeof (TmdRecInfo);
    BookmarkSize := sizeof (Integer);

    // everything OK: table is now open
    FIsTableOpen := True;
end;
```

You'll notice that the method sets most of the local fields of the class, and also the `BookmarkSize` field of the base `TDataSet` class. Within this method, I call two custom methods I introduced in my custom dataset hierarchy: `InternalPreOpen` and `InternalAfterOpen`. The first, `InternalPreOpen`, is used for operations required at the very beginning, such as checking whether the dataset can actually be opened and reading the header information from the file. The code checks an internal version number for consistency with the value saved when the table is first created, as you'll see later on. By raising an exception in this method, we can eventually stop the open operation.

Here is the code for the two methods in the derived stream-based dataset:

```

const
    HeaderVersion = 10;

procedure TMdDataSetStream.InternalPreOpen;
begin
    // the size of the header
    FDataFileHeaderSize := sizeof (TMdDataFileHeader);

    // check if the file exists
    if not FileExists (FTableName) then
        raise EMdDataSetError.Create ('Open: Table file not found');

    // create a stream for the file
    FStream := TFileStream.Create (FTableName, fmOpenReadWrite);

    // initialize local data (loading the header)
    FStream.ReadBuffer (FDataFileHeader, FDataFileHeaderSize);
    if FDataFileHeader.VersionNumber <> HeaderVersion then
        raise EMdDataSetError.Create ('Illegal File Version');
    // let's read this, double check later
    FRecordCount := FDataFileHeader.RecordCount;
end;

procedure TMdDataSetStream.InternalAfterOpen;
begin
    // check the record size
    if FDataFileHeader.RecordSize <> FRecordSize then
        raise EMdDataSetError.Create ('File record size mismatch');
    // check the number of records against the file size
    if (FDataFileHeaderSize + FRecordCount * FRecordSize) <> FStream.Size then
        raise EMdDataSetError.Create ('InternalOpen: Invalid Record Size');
end;

```

The second method, `InternalAfterOpen`, is used for operations required after the field definitions have been set and is followed by code that compares the record size read from the file against the value computed in the `InternalInitFieldDefs` method. The code checks also that the number of records read from the header is compatible with the actual size of the file. This test might fail if the dataset wasn't closed properly: you might want to modify this code to let the dataset refresh the record size in the header anyway.

The `InternalOpen` method of the custom dataset class is specifically responsible for calling `InternalInitFieldDefs`, which determines the field definitions (at either design time or run time). For this example, I've decided to base the field definitions on an external file, a simple INI file that provides a section for every field. Each section contains the name and data type

of the field, as well as its size if it is string data. Listing 18.3 is the `Contrib.INI` file that we'll use in the component's demo application:

---

➤ **Listing 18.3: The `Contrib.INI` file for the demo application**

```
[Fields]
Number = 6

[Field1]
Type = ftString
Name = Name
Size = 30

[Field2]
Type = ftInteger
Name = Level

[Field3]
Type = ftDate
Name = BirthDate

[Field4]
Type = ftCurrency
Name = Stipend

[Field5]
Type = ftString
Name = Email
Size = 50

[Field6]
Type = ftBoolean
Name = Editor
```

---

This file, or a similar one, must use the same name as the table file and must be in the same directory. The `InternalInitFieldDefs` method (shown in Listing 18.4) will read it, using the values it finds to set up the field definitions and determine the size of each record. The method also initializes an internal `TList` object that stores the offset of every field inside the record. We'll use this `TList` to access fields' data within the record buffer, as you can see in the code listing.

---

➤ **Listing 18.4: The `InternalInitFieldDefs` method of the stream-based dataset**

```
procedure TMDatasetStream.InternalInitFieldDefs;
var
  IniFileName, FieldName: string;
  IniFile: TIniFile;
```

```

nFields, I, TmpFieldOffset, nSize: Integer;
FieldType: TFieldType;
begin
  FFieldOffset := TList.Create;
  FieldDefs.Clear;
  TmpFieldOffset := 0;
  IniFilename := ChangeFileExt(FTableName, '.ini');
  IniFile := TIniFile.Create (IniFilename);
  // protect INI file
  try
    nFields := IniFile.ReadInteger (' Fields', 'Number', 0);
    if nFields = 0 then
      raise EDataSetOneError.Create (' InitFieldsDefs: 0 fields?');
    for I := 1 to nFields do
      begin
        // create the field
        FieldType := TFieldType (GetEnumValue (TypeInfo (TFieldType),
          IniFile.ReadString ('Field' + IntToStr (I), 'Type', '')));
        FieldName := IniFile.ReadString ('Field' + IntToStr (I), 'Name', '');
        if FieldName = '' then
          raise EDataSetOneError.Create (
            'InitFieldsDefs: No name for field ' + IntToStr (I));
        nSize := IniFile.ReadInteger ('Field' + IntToStr (I), 'Size', 0);
        FieldDefs.Add (FieldName, FieldType, nSize, False);
        // save offset and compute size
        FFieldOffset.Add (Pointer (TmpFieldOffset));
        case FieldType of
          ftString:           Inc (TmpFieldOffset, nSize + 1);
          ftBoolean, ftSmallInt, ftWord:   Inc (TmpFieldOffset, 2);
          ftInteger, ftDate, ftTime:      Inc (TmpFieldOffset, 4);
          ftFloat, ftCurrency, ftDateTime: Inc (TmpFieldOffset, 8);
        else
          raise EDataSetOneError.Create (
            'InitFieldsDefs: Unsupported field type');
        end;
      end; // for
    finally
      IniFile.Free;
    end;
    FRecordSize := TmpFieldOffset;
  end;
end;

```

---

Closing the table is simply a matter of disconnecting the fields (using some standard calls). Each class must dispose the data it allocated and update the file header, the first time records are added and each time the record count has changed:

```

procedure TMDCustomDataSet.InternalClose;
begin
  // disconnect field objects
  BindFields (False);

```

```

// destroy field object (if not persistent)
if DefaultFields then
  DestroyFields;
// close the file
FIsTableOpen := False;
end;

procedure TMdDataSetStream.InternalClose;
begin
  // if required, save updated header
  if (FDataFileHeader.RecordCount <> FRecordCount) or
    (FDataFileHeader.RecordSize = 0) then
    begin
      FDataFileHeader.RecordSize := FRecordSize;
      FDataFileHeader.RecordCount := FRecordCount;
      if Assigned (FStream) then
        begin
          FStream.Seek (0, soFromBeginning);
          FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
        end;
    end;
  // free the internal list field offsets and the stream
  FFieldOffset.Free;
  FStream.Free;
  inherited InternalClose;
end;

```

Another related function is used to test whether the dataset is open, something we can solve using the corresponding local field:

```

function TMDCustomDataSet.IsCursorOpen: Boolean;
begin
  Result := FIsTableOpen;
end;

```

These are the opening and closing methods you need to implement in any custom dataset. However, most of the time, you'll also add a method to create the table. In this example, the `CreateTable` method creates an empty file and inserts information in the header: a fixed version number, a dummy record size (we don't know the size until we initialize the fields), and the record count (which is zero to start):

```

procedure TMdDataSetStream.CreateTable;
begin
  CheckInactive;
  InternalInitFieldDefs;

  // create the new file
  if FileExists (FTableName) then

```



```

    raise EmdDataSetError.Create ('File ' + FTableName + ' already exists');
    FStream := TFileStream.Create (FTableName, fmCreate or fmShareExclusive);
  try
    // save the header
    FDataFileHeader.VersionNumber := HeaderVersion;
    FDataFileHeader.RecordSize := 0;    // used later
    FDataFileHeader.RecordCount := 0;  // empty
    FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
  finally
    // close the file
    FStream.Free;
  end;
end;

```

## Section II: Movement and Bookmark Management

As mentioned earlier, one of the things every dataset must implement is *bookmark management*, which is necessary for navigating through the dataset. Logically, a bookmark is a reference to a specific record of the dataset, something that uniquely identifies the record so that a dataset can access it and compare it to other records. Technically, bookmarks are pointers. You can implement them as pointers to specific data structures that store record information, or you can implement them as simple record numbers. For simplicity, I'll use the latter approach.

Given a bookmark, you should be able to find the corresponding record, but given a record buffer, you should also be able to retrieve the corresponding bookmark. This is the reason for appending the `TMdRecInfo` structure to the record data in each record buffer. This data structure stores the bookmark for the record in the buffer, as well as some bookmark flags defined as:

```

type
  TBookmarkFlag = (bfCurrent, bfBOF, bfEOF, bfInserted);

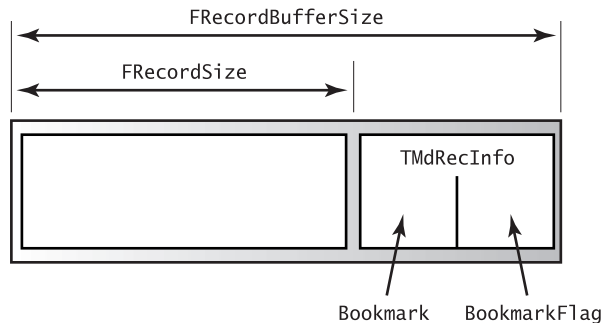
```

The system will request us to store these flags in each record buffer, and will later ask us to retrieve the flags for a given record buffer.

To summarize, the structure of a record buffer stores the data of the record, the bookmark, and the bookmark flags, as you can see in Figure 18.5.

**FIGURE 18.5:**

The structure of each buffer of the custom dataset, along with the various local fields referring to its sub-portions



To access the bookmark and flags, we can simply use as an offset the size of the actual data, casting the value to the `PMdRecInfo` pointer type, and then access the proper field of the `TmdRecInfo` structure via the pointer. The two methods used to set and get the bookmark flags demonstrate this technique:

```
procedure TMDCustomDataSet.SetBookmarkFlag (Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PMdRecInfo(Buffer + FRecordSize).BookmarkFlag := Value;
end;

function TMDCustomDataSet.GetBookmarkFlag (Buffer: PChar): TBookmarkFlag;
begin
  Result := PMdRecInfo(Buffer + FRecordSize).BookmarkFlag;
end;
```

The methods we use to set and get the current bookmark of a record are similar to the previous two, but they add some complexity because we receive a pointer to the bookmark in the `Data` parameter. Casting this pointer as an integer pointer (`PInteger`) and dereferencing it, we obtain the bookmark value:

```
procedure TMDCustomDataSet.GetBookmarkData (Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ := PMdRecInfo(Buffer + FRecordSize).Bookmark;
end;

procedure TMDCustomDataSet.SetBookmarkData (Buffer: PChar; Data: Pointer);
begin
  PMdRecInfo(Buffer + FRecordSize).Bookmark := PInteger(Data)^;
end;
```

The key bookmark management method is `InternalGotoBookmark`, which your dataset uses to make a given record the current one. You'll notice that this isn't the standard navigation technique, since it's much more common to move to the next or previous record (something

we can accomplish using the `GetRecord` method presented in the next section), or to move to the first or last record (something we'll accomplish using the `InternalFirst` and `InternalLast` methods described shortly).

Oddly enough, the `InternalGotoBookmark` method doesn't expect a bookmark parameter, but a pointer to a bookmark, so we must dereference it to determine the bookmark value. The following method, `InternalSetToRecord`, is what you use to jump to a given bookmark, but it must extract the bookmark from the record buffer passed as a parameter. Then, `InternalSetToRecord` calls `InternalGotoBookmark`. Here are the two methods:

```

procedure TMDCustomDataSet.InternalGotoBookmark (Bookmark: Pointer);
var
    ReqBookmark: Integer;
begin
    ReqBookmark := PInteger (Bookmark)^;
    if (ReqBookmark >= 0) and (ReqBookmark < InternalRecordCount) then
        FCurrentRecord := ReqBookmark
    else
        raise EMdDataSetError.Create ('Bookmark ' +
            IntToStr (ReqBookmark) + ' not found');
    end;

procedure TMDCustomDataSet.InternalSetToRecord (Buffer: PChar);
var
    ReqBookmark: Integer;
begin
    ReqBookmark := PMdRecInfo(Buffer + FRecordSize).Bookmark;
    InternalGotoBookmark (@ReqBookmark);
end;

```

In addition to the bookmark management methods just described, there are several other navigation methods we use to move to specific positions within the dataset, such as the first or last record. Actually, these two methods don't move the current record pointer to the first or last record, but move it to one of two special locations before the first record and after the last one. These are not actual records; Borland calls them *cracks*. The beginning-of-file crack, or `BofCrack`, has the value `-1` (set in the `InternalOpen` method), since the position of the first record is zero. The end-of-file crack, or `EofCrack`, has the value of the number of records, since the last record has the position `FRecordCount - 1`. We've used two local fields, called `EofCrack` and `BofCrack`, to make this code easier to read:

```

procedure TMDCustomDataSet.InternalFirst;
begin
    FCurrentRecord := BofCrack;
end;

procedure TMDCustomDataSet.InternalLast;

```

```
begin  
    EofCrack := InternalRecordCount;  
    FCurrentRecord := EofCrack;  
end;
```

The `InternalRecordCount` method is a virtual method introduced in my `TMDCustomDataSet` class, as different datasets can either have a local field for this value (as in case of the stream-based dataset, which has an `FRecordCount` field) or compute it on-the-fly.

Another group of optional methods is used to get the current record number (used by the `DBGrid` component to show a proportional vertical scroll bar), set the current record number, or determine the number of records. These methods are quite easy to understand, if you recall that the range of the internal `FCurrentRecord` field is from 0 to the number of records minus 1. In contrast, the record number reported to the system ranges from 1 to the number of records:

```
function TMDCustomDataSet.GetRecordCount: Longint;  
begin  
    CheckActive;  
    Result := InternalRecordCount;  
end;  
  
function TMDCustomDataSet.GetRecNo: Longint;  
begin  
    UpdateCursorPos;  
    if FCurrentRecord < 0 then  
        Result := 1  
    else  
        Result := FCurrentRecord + 1;  
end;  
  
procedure TMDCustomDataSet.SetRecNo(Value: Integer);  
begin  
    CheckBrowseMode;  
    if (Value > 1) and (Value <= FRecordCount) then  
        begin  
            FCurrentRecord := Value - 1;  
            Resync([]);  
        end;  
end;
```

Notice that it is the generic custom dataset class that implements all the methods of this section. The derived stream-based dataset doesn't need to modify any of them.

## Section III: Record Buffers and Field Management

Now that we've covered all the support methods, we can examine the core of a custom dataset. Besides opening and creating records and moving around between them, the component really needs to move the data from the stream (the persistent file) to the record buffers, and from the record buffers to the `TField` objects that are connected to the data-aware controls. The management of record buffers is quite complex, because each dataset also needs to allocate, empty, and free the memory it requires:

```
function TMDCustomDataSet.AllocRecordBuffer: PChar;
begin
    GetMem (Result, FRecordBufferSize);
end;

procedure TMDCustomDataSet.FreeRecordBuffer (var Buffer: PChar);
begin
    FreeMem (Buffer);
end;
```

The reason for allocating memory this way is that a dataset generally adds more information to the record buffer, so the system has no way of knowing how much memory to allocate. You'll notice that in the `AllocRecordBuffer` method, the component allocates the memory for the record buffer, including both the database data and the record information. In fact, in the `InternalOpen` method I wrote

```
FRecordBufferSize := InternalRecordSize + sizeof (TmdRecInfo);
```

The component also needs to implement a function to reset the buffer, `InternalInitRecord`, usually filling it with numeric zeros or spaces.

Oddly enough, we must also implement a method that returns the size of each record, but only the data portion—not the entire record buffer. This method is necessary for implementing the read-only `RecordSize` property, used only in a couple of peculiar cases in the entire VCL source code. In the generic custom dataset, the `GetRecordSize` method returns the value of the `FRecordSize` field.

Now we've actually reached the core of our custom dataset component. The methods of this group are `GetRecord`, which reads data from the file, `InternalPost` and `InternalAddRecord`, which update or add new data to the file, and `InternalDelete`, which removes data and is not implemented in my sample dataset.

The most complex method of this group is probably `GetRecord`, which serves multiple purposes. In fact, this method is used by the system to retrieve the data for the current record, fill a buffer passed as a parameter, and retrieve the data of the next or previous records. The `GetMode` parameter determines its action:

```
type
    TGetMode = (gmCurrent, gmNext, gmPrior);
```

Of course, a previous or next record might not exist. Even the current record might not exist; for example, when the table is empty (or in case of an internal error). In these cases we don't retrieve the data but return an error code. Therefore, this method's result can be one of the following values:

```
type
  TGetResult = (grOK, grBOF, grEOF, grError);
```

Checking to see if the requested record exists is slightly different than you might expect. We don't have to determine if the current record is in the proper range, only if the requested one is. For example, in the `gmCurrent` branch of the case statement, we use the standard expression `CurrentRecord >= InternalRecordCount`. To fully understand the various cases, you might want to read the code a couple of times.

It took me some trial and error (and system crashes caused by recursive calls) to get it straight when I wrote my first custom dataset a few years back. To test it, consider that if you use a `DBGrid`, the system will perform a series of `GetRecord` calls, until either the grid is full or `GetRecord` return `grEOF`. Here's the entire code of the `GetRecord` method:

```
// III: Retrieve data for current, previous, or next record
// (moving to it if necessary) and return the status
function TMdCustomDataSet.GetRecord(Buffer: PChar;
  GetMode: TGetMode; DoCheck: Boolean): TGetResult;
begin
  Result := grOK; // default
  case GetMode of
    gmNext: // move on
      if FCurrentRecord < InternalRecordCount - 1 then
        Inc (FCurrentRecord)
      else
        Result := grEOF; // end of file
    gmPrior: // move back
      if FCurrentRecord > 0 then
        Dec (FCurrentRecord)
      else
        Result := grBOF; // begin of file
    gmCurrent: // check if empty
      if FCurrentRecord >= InternalRecordCount then
        Result := grError;
  end;
  // load the data
  if Result = grOK then
    InternalLoadCurrentRecord (Buffer)
  else if (Result = grError) and DoCheck then
    raise EMdDataSetError.Create ('GetRecord: Invalid record');
end;
```

If there's an error and the DoCheck parameter was True, GetRecord raises an exception. If everything goes fine during record selection, the component loads the data from the stream, moving to the position of the current record (given by the record size multiplied by the record number). In addition, we need to initialize the buffer with the proper bookmark flag and bookmark (or record number) value. This is accomplished by another virtual method I introduced, so that derived classes will only need to implement this portion of the code, while the complex GetRecord method remains unchanged:

```
procedure TMdDataSetStream.InternalLoadCurrentRecord (Buffer: PChar);
begin
    FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
    FStream.ReadBuffer (Buffer^, FRecordSize);
    with PMdRecInfo(Buffer + FRecordSize)^ do
        begin
            BookmarkFlag := bfCurrent;
            Bookmark := FCurrentRecord;
        end;
    end;
```

We move data to the file in two different cases: when you modify the current record (that is, a post after an edit) or when you add a new record (a post after an insert or append). We use the InternalPost method in both cases, but we can check the dataset's State property to determine which type of post we're performing. In both cases we don't receive a record buffer as a parameter, so we must use the ActiveRecord property of TDataSet, which points to the buffer for the current record:

```
procedure TMdDataSetStream.InternalPost;
begin
    CheckActive;
    if State = dsEdit then
        begin
            // replace data with new data
            FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
            FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
        end
    else
        begin
            // always append
            InternalLast;
            FStream.Seek (0, soFromEnd);
            FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
            Inc (FRecordCount);
        end;
    end;
```

In addition, there's another related method, `InternalAddRecord`. This method is called by the `AddRecord` method, which in turn is called by `InsertRecord` and `AppendRecord`. These last two are public methods a user can call. This is an alternative to inserting or appending a new record to the dataset, editing the values of the various fields, and then posting the data, since the `InsertRecord` and `AppendRecord` calls receive the values of the fields as parameters. All we must do at that point is replicate the code used to add a new record in the `InternalPost` method:

```
procedure TMdDataSetOne.InternalAddRecord(Buffer: Pointer; Append: Boolean);  
begin  
    // always append at the end  
    InternalLast;  
    FStream.Seek (0, soFromEnd);  
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);  
    Inc (FRecordCount);  
end;
```

The last file operation I should have implemented is one that removes the current record. This operation is common, but it is actually quite complex. If we take a simple approach, such as creating an empty spot in the file, then we'll need to keep track of that spot and make the code for reading or writing a specific record work around that spot. An alternate solution is to make a copy of the entire file, without the given record, and then replace the original file with the copy. Given these choices, I felt that for this example I could forgo supporting record deletion.

## Section IV: From Buffers to Fields

In the last few methods, we've seen how datasets move data from the data file to the memory buffer. However, there's little Delphi can do with this record buffer, because it doesn't yet know how to interpret the data in the buffer. We need to provide two more methods: `GetData`, which copies the data from the record buffer to the field objects of the dataset, and `SetData`, which moves the data back from the fields to the record buffer. What Delphi will do automatically for us is move the data from the field objects to the data-aware controls, and back.

The code for these two methods isn't very complex, primarily because we saved the field offsets inside the record data in a `TList` object called `FFieldOffset`. By simply incrementing the pointer to the initial position in the record buffer of the current field's offset, we'll be able to get the specific data, which takes `Field.DataSize` bytes.

A confusing element of these two methods is that they both accept a `Field` parameter and a `Buffer` parameter. At first, one might think that the buffer passed as parameter is the record buffer. Actually, I found out that the `Buffer` is a pointer to the field object's raw data. If you use one of the field object's methods to move that data, it will call the dataset's `GetData` or



SetData methods, probably causing an infinite recursion. Instead, you should use the Active-Buffer pointer to access the record buffer, use the proper offset to get to the data for the current field in the record buffer, and then use the provided Buffer to access the field data. The only difference between the two methods is the direction we're moving the data:

```

function TMdDataSetOne.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
    FieldOffset: Integer;
    Ptr: PChar;
begin
    Result := False;
    if not IsEmpty and (Field.FieldNo > 0) then
        begin
            FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
            Ptr := ActiveBuffer;
            Inc (Ptr, FieldOffset);
            if Assigned (Buffer) then
                Move (Ptr^, Buffer^, Field.DataSize);
            Result := True;
            if (Field is TDateTimeField) and (PInteger(Ptr)^ = 0) then
                Result := False;
        end;
    end;

procedure TMdDataSetOne.SetFieldData(Field: TField; Buffer: Pointer);
var
    FieldOffset: Integer;
    Ptr: PChar;
begin
    if Field.FieldNo >= 0 then
        begin
            FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
            Ptr := ActiveBuffer;
            Inc (Ptr, FieldOffset);
            if Assigned (Buffer) then
                Move (Buffer^, Ptr^, Field.DataSize)
            else
                raise Exception.Create (
                    'Very bad error in TMdDataSetStream.SetField data');
            DataEvent (deFieldChange, Longint(Field));
        end;
    end;

```

The GetField method should return True or False to indicate whether the field contains data or is empty. However, unless you use a special marker for blank fields, it's very difficult to determine this, since we're storing values of different data types. For example, a test such

as `Ptr^<>#0` makes sense only if you are using a string representation for all of the fields. If you use this test, zero integer values and empty strings will show as null values (the data-aware controls will be empty), which may be what you want. The problem is that Boolean False values won't show up. Even worse, floating-point values with no decimals and few digits won't be displayed, because the exponent portion of their representation will be zero! However, to make this example work in Delphi 6, I had to consider as empty each date/time field with an initial zero. Without this code, Delphi tries to convert the illegal *internal* zero date (internally, date fields don't use a `TDateTime` data type but a different representation) raising an exception. The code used to work with past versions of Delphi.

**WARNING** While trying to fix this problem, I also found out that if you call `IsNull` for a field, this request is resolved by calling `GetFieldData` without passing any buffer to fill but looking only for the result of the function call. This is the reason for the `if Assigned (Buffer)` test within the code.

There's one final method, which doesn't fall into any category: `InternalHandleException`. Generally, this method silences the exception, as it is activated only at design time.

## Testing the Stream-Based DataSet

After all this work, we're finally ready to test an application example of the custom dataset component, installed in the component's package for this chapter. The form displayed by the `StreamDSDemo` program is quite simple, as you can see in Figure 18.6. It has a panel with two buttons, a check box, and a navigator component, plus a `DBGrid` filling its client area.

**FIGURE 18.6:**

The form of the `StreamDSDemo` example. The custom dataset has been activated, so we can already see the data at design time.

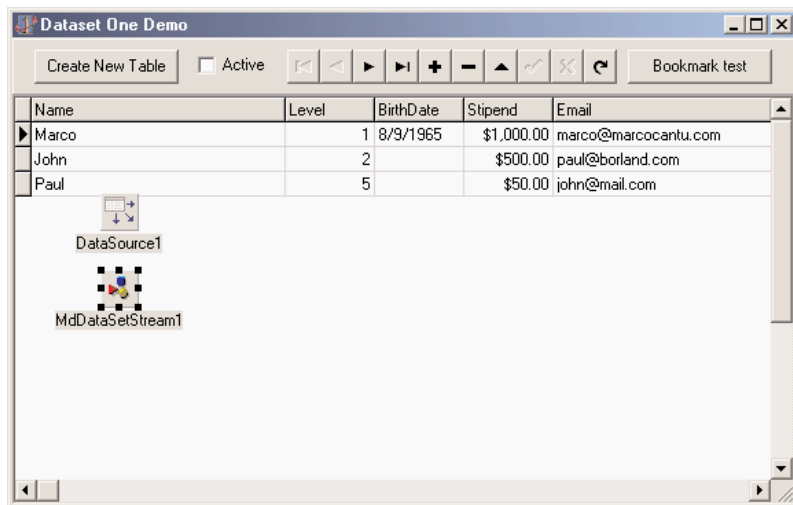


Figure 18.6 shows the form of the example at design time, but we've activated the custom dataset so that its data is already visible. Of course we'd already prepared the INI file with the table definition (it's the file we already listed when discussing the dataset initialization), and we executed the program to add some data to the file.

It's also possible to modify the form using Delphi's Fields editor and set the properties of the various field objects. Everything works as it does with one of the standard dataset controls! However, to make this work you'll need to enter the name of the custom dataset's file in the `TableName` property, using the complete path.

**WARNING**

As the demo program defines the absolute path of the table file at design time, you'll need to fix it if you copy the examples to a different drive or directory. In the example, the `TableName` property is used only at design time. At run time, in fact, the program looks for the table in the current directory.

The code of the example is rather simple, especially compared to the code of the custom dataset. If the table doesn't exist yet, you can click the Create button:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    MdDataSetStream1.CreateTable;  
    MdDataSetStream1.Open;  
    CheckBox1.Checked := MdDataSetStream1.Active;  
end;
```

You'll notice that we create the file first, open and close it, and then open the table. This is the same behavior as the `TTable` component (which accomplishes this using the `CreateTable` method). To simply open or close the table, you can click the check box:

```
procedure TForm1.CheckBox1Click(Sender: TObject);  
begin  
    MdDataSetStream1.Active := CheckBox1.Checked;  
end;
```

Finally, I've created a method that tests custom dataset's bookmark management code and seems to work.

## A Directory in a Dataset

An important idea related to datasets in Delphi is that they simply represent a set of data, regardless where this data comes from. An SQL server or a local file are examples of traditional datasets, but you can use the same technology to show a list of users of a system, a list of files of a folder, the properties of some objects, some XML-based data, and so on.

As an example, the second (and last) dataset presented in this chapter is a list of files. This is based once more on a generic approach. I've built a generic dataset based on a list of objects in memory (using a `TObjectList`), then derived a version in which the objects correspond to the files of a folder. The actual example is simplified by the fact it is a read-only dataset, so you might even find it simpler than the previous dataset I presented.

**NOTE**

Some of the ideas presented here were discussed in an article I wrote for the Borland Community Web site, <http://community.borland.com>, published in June 2000.

## A List as a Dataset

The generic list-based dataset is called `TMdListDataSet` and contains the list of objects, created when you open the dataset and freed when you close it. This dataset doesn't store the actual record data within the buffer; rather, it saves in the buffer only the position in the list of the entry corresponding to the record's data. This is the class definition:

```
type
  TMdListDataSet = class (TMdCustomDataSet)
  protected
    // the list holding the data
    FList: TObjectList;
    // dataset virtual methods
    procedure InternalPreOpen; override;
    procedure InternalClose; override;
    // custom dataset virtual methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord (Buffer: PChar); override;
  end;
```

You can see that by writing a generic custom data class, we can override few virtual methods of the `TDataSet` class and of this custom dataset class, and have a working dataset (although this is still an abstract class, which requires extra code from subclasses to work). When the dataset is opened, we have to create the list and set the record size, to indicate we're simply saving the list index in the buffer:

```
procedure TMdListDataSet.InternalPreOpen;
begin
  FList := TObjectList.Create (True); // owns the objects
  FRecordSize := 4; // an integer, the list item id
end;
```

Further subclasses at this point should also fill the list with actual objects.

**TIP**

Similarly to the ClientDataSet, my list dataset keeps all of its data in memory. However, using some smart techniques, you can also create a list of “fake” objects, and then load the actual objects only when you are accessing them.

Closing is simply a matter of freeing the list, which has a record count corresponding to the list size:

```
function TMdListDataSet.InternalRecordCount: Integer;
begin
    Result := fList.Count;
end;
```

The only other method is used to save the data of the current record in the record buffer, including the bookmark information. The core data is simply the position of the current record, which matches the list index (and also the bookmark):

```
procedure TMdListDataSet.InternalLoadCurrentRecord (Buffer: PChar);
begin
    PInteger (Buffer)^ := fCurrentRecord;
    with PMdRecInfo(Buffer + FRecordSize)^ do
        begin
            BookmarkFlag := bfCurrent;
            Bookmark := fCurrentRecord;
        end;
    end;
```

## Directory Data

The derived directory dataset class has to provide a way to load the objects in memory when the dataset is opened, to define the proper fields, and to read and write the value of those fields. Of course, it has also a property indicating the directory to work on, or to be more precise, the directory plus the file mask used for filtering the files (as in c:\docs\\*.txt):

```
type
    TMDirDataset = class(TMdListDataSet)
    private
        FDirectory: string;
        procedure SetDirectory(const NewDirectory: string);
    protected
        // TDataSet virtual methods
        procedure InternalInitFieldDefs; override;
        procedure SetFieldData(Field: TField; Buffer: Pointer); override;
        function GetCanModify: Boolean; override;
        // custom dataset virtual methods
        procedure InternalAfterOpen; override;
    public
```

```

    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
published
    property Directory: string read FDirectory write SetDirectory;
end;

```

The GetCanModify function is another virtual method of TDataSet, used to determine if the dataset is read-only. In this case, it simply returns False. Also, we won't have to write any code for the SetFieldData procedure, but we have to define it because it is an abstract virtual method.

As I am dealing with a list of objects, the unit includes also a class for those objects. In this case, I am working with file data extracted by a TSearchRec buffer by the TFileData class constructor:

```

type
  TFileData = class
    public
      ShortFileName: string;
      Time: TDateTime;
      Size: Integer;
      Attr: Integer;
      constructor Create (var FileInfo: TSearchRec);
    end;

constructor TFileData.Create (var FileInfo: TSearchRec);
begin
  ShortFileName := FileInfo.Name;
  Time := FileDateToDateTime (FileInfo.Time);
  Size := FileInfo.Size;
  Attr := FileInfo.Attr;
end;

```

This constructor is called for each folder while opening the dataset:

```

procedure TMDirDataset.InternalAfterOpen;
var
  Attr: Integer;
  FileInfo: TSearchRec;
  FileData: TFileData;
begin
  // scan all files
  Attr := faAnyFile;
  FList.Clear;
  if SysUtils.FindFirst(FDirectory, Attr, FileInfo) = 0 then
    repeat
      FileData := TFileData.Create (FileInfo);
      FList.Add (FileData);
    until SysUtils.FindNext(FileInfo) <> 0;
  SysUtils.FindClose(FileInfo);
end;

```

The next step is to define the fields of the dataset, which in this case are fixed and depend on the available directory data:

```
procedure TMDirDataset.InternalInitFieldDefs;
begin
  if fDirectory = '' then
    raise EMDataSetError.Create ('Missing directory');

  // field definitions
  FieldDefs.Clear;
  FieldDefs.Add ('FileName', ftString, 40, True);
  FieldDefs.Add ('TimeStamp', ftDateTime);
  FieldDefs.Add ('Size', ftInteger);
  FieldDefs.Add ('Attributes', ftString, 3);
  FieldDefs.Add ('Folder', ftBoolean);
end;
```

Finally, the component has to move the data from the object of the list referenced by the current record buffer (the ActiveBuffer value) to each field of the dataset, as requested by the GetFieldData method. This function uses either Move or StrCopy, depending on the data type and does some conversions for the attributes codes (*H* for hidden, *R* for read-only, and *S* for system) extracted from the related flags and used also to determine whether a file is actually a folder. Here is the code:

```
function TMDirDataset.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FileData: TFileData;
  Bool1: WordBool;
  strAttr: string;
  t: TDateTimeRec;
begin
  FileData := fList [PInteger(ActiveBuffer)^] as TFileData;
  case Field.Index of
    0: // filename
      StrCopy (Buffer, pchar(FileData.ShortFileName));
    1: // timestamp
      begin
        t := DateTimeToNative (ftdatetime, FileData.Time);
        Move (t, Buffer^, sizeof (TDateTime));
      end;
    2: // size
      Move (FileData.Size, Buffer^, sizeof (Integer));
    3: // attributes
      begin
        strAttr := '   ';
        if (FileData.Attr and SysUtils.faReadOnly) > 0 then
          strAttr [1] := 'R';
      end;
  end;
```

```
    if (FileData.Attr and SysUtils.faSysFile) > 0 then
        strAttr [2] := 'S';
    if (FileData.Attr and SysUtils.faHidden) > 0 then
        strAttr [3] := 'H';
    StrCopy (Buffer, pchar(strAttr));
end;
4: // folder
begin
    Bool1 := FileData.Attr and SysUtils.faDirectory > 0;
    Move (Bool1, Buffer^, sizeof (WordBool));
end;
end; // case
Result := True;
end;
```

The tricky part in writing this code was figuring out the internal format of dates stored within date/time fields. This is not the common `TDateTime` format used by Delphi, and not even the internal `TTimeStamp`, but what is called the internally called the “native” date time format. I’ve written a conversion function cloning one I’ve found in the VCL code for the date/time fields:

```
function DateTimeToNative(DataType: TFieldType; Data: TDateTime): TDateTimeRec;
var
    TimeStamp: TTimeStamp;
begin
    TimeStamp := DateTimeToTimeStamp(Data);
    case DataType of
        ftDate: Result.Date := TimeStamp.Date;
        ftTime: Result.Time := TimeStamp.Time;
    else
        Result.DateTime := TimeStampToMsecs(TimeStamp);
    end;
end;
```

With this dataset available, building the demo program (shown in Figure 18.7) was simply a matter of connecting a `DBGrid` component to it and adding a folder-selection component, Delphi 6’s `ShellTreeView` control. This control is set up to work only on files, by setting its `Root` property to `C:\`. When the user selects a new folder, the `OnChange` event handler of the `ShellTreeView` control refreshes the dataset.



**FIGURE 18.7:**

The output of the DirDemo example, which uses a rather unusual dataset, showing directory data.

FileName	TimeStamp	Size	Attributes	Folder
.	3/29/2001 11:17:30 AM			True
..	3/29/2001 11:17:30 AM			True
first0.gif	3/29/2001 11:17:30 AM	1,240		False
first1.gif	3/29/2001 11:17:30 AM	1,223		False
img38.htm	3/29/2001 11:17:30 AM	485		False
img38.jpg	3/29/2001 11:17:30 AM	55,957		False
img39.htm	3/29/2001 11:17:30 AM	485		False
img39.jpg	3/29/2001 11:17:30 AM	45,819		False
img40.htm	3/29/2001 11:17:30 AM	485		False
img40.jpg	3/29/2001 11:17:30 AM	51,802		False
img41.htm	3/29/2001 11:17:30 AM	485		False
img41.jpg	3/29/2001 11:17:30 AM	29,024		False
img42.htm	3/29/2001 11:17:30 AM	485		False

```

procedure TForm1.ShellTreeView1Change(Sender: TObject; Node: TTreeNode);
begin
    MDirDataset1.Close;
    MDirDataset1.Directory := ShellTreeView1.Path + '\*.*';
    MDirDataset1.Open;
end;

```

## What's Next?

In this chapter we've delved inside Delphi's database architecture, by first examining the development of data-aware controls and then studying the internals of the TDataSet class to write a couple of custom dataset components. With this information, and all the other ideas presented in this part devoted to database programming, you should probably be able to choose the architecture of your database applications, depending on your needs.

### NOTE

I've actually extended the list-based dataset to build an object-based version, hosting the business logic of an application and mapped to a relational database. Refer to my Web site ([www.marcocantu.com](http://www.marcocantu.com)) or contact me for the availability of this code.

Database programming is certainly a core element of Delphi, the reason for devoting several chapters of the book to this topic. We'll get back to this topic when focusing on presenting database data over the Web, in Chapters 21 and 22.

For the moment, though, we have to introduce another important element of Windows applications, COM and OLE Automation, covered in the next two chapters.



**PART IV**

---

# **Beyond Delphi: Connecting with the World**

- **Chapter 19: COM Programming**
- **Chapter 20: From Automation to COM+**
- **Chapter 21: Internet Programming: Sockets and Indy Components**
- **Chapter 22: Web Programming with WebBroker and WebSnap**
- **Chapter 23: XML and SOAP**

# COM Programming

- What are OLE and COM?
- COM, GUIDs, and class factories
- Delphi interfaces and COM
- The VCL COM-support classes
- Windows shell interfaces

**F**or about 10 years, starting soon after the release of Windows 3.0, Microsoft has kept promising that its operating system and their API would be based on a real object model instead of functions. According to the speculations, Windows 95 (and later Windows 2000) should have been based on this revolutionary approach. Nothing like this happened, but Microsoft kept pushing COM (Component Object Model), built the Windows 95 shell on top of it, pushed applications integration with COM and derivative technologies (such as Automation), and reached the peak by introducing COM+ with Windows 2000.

Now, soon after the release of the complete foundation required for high-level COM programming, Microsoft has decided to switch to a new core technology, part of the dotNet (or .Net, if you prefer) initiative. My impression is that COM wasn't really suited for the integration of fine-grained objects, though it succeeded in providing an architecture for integrating applications or large objects.

**NOTE**

dotNet is a mix of interesting new technologies and pure marketing hype, and this book doesn't discuss it in detail. Even if it were possible to predict how dotNet will affect programmers using Microsoft development tools, it is far from clear how it will affect Delphi programmers. dotNet, in fact, is based on a class library very similar to Delphi's VCL, and it is unclear what the advantage will be of switching to it. If Borland could bundle the VCL with an operating system, along with its core run-time packages, you'd have a situation very similar to dotNet, as far as the library is concerned. Instead, if you are looking for a virtual machine and portable code, you can certainly consider Java and the portability between Windows and Linux possible with the CLX library of Delphi 6 and Kylix. Having said this, I'm not underestimating dotNet, but this is a book on Delphi programming. In any case, we'll get back to many core elements of dotNet in the final chapters, discussing XML and SOAP. Finally, the system can expose dotNet objects as COM objects, so after you learn COM you'll also have a chance to interact with dotNet.

In this chapter, we'll build our first COM object and integrate COM objects with the Windows shell. Type libraries, Automation, and other topics will be covered in the next chapter. I will stick to the basic elements to let you understand the role of this technology without delving heavily into the details. I'll bear in mind the clouds on the future of COM, declared obsolete by Microsoft after the announcement of dotNet but still heavily used by the same company inside their applications and operating systems.

## A Short History of OLE and COM

Part of the confusion related to COM technology comes from the fact that Microsoft has used different names for it for marketing reasons. Everything started with Object Linking

and Embedding (OLE, for short), which was an extension of the DDE (Dynamic Data Exchange) model. Using the Clipboard allows you to copy some raw data, and using DDE allows you to connect parts of two documents. OLE allows you to copy data from a server application to a client application, along with information regarding the server or a reference to information stored in the Windows Registry. The raw data might be copied along with the link (object embedding) or kept in the original file (object linking). OLE Documents are now called *Active Documents*.

Microsoft updated OLE to OLE 2 and started adding new features, such as OLE Automation and OLE Controls. The next step was to build the Windows 95 shell using OLE technology and interfaces and then to rename the OLE Controls (previously known also as OCX) as ActiveX controls, changing the specification to allow for lightweight controls suitable for distribution over the Internet. For a while, Microsoft promoted ActiveX controls as suitable for the Internet, but the idea was never fully accepted by the development community, certainly not as “suitable” for Internet development.

As this technology was extended and became increasingly important to the Windows platform, Microsoft changed the name to OLE, and then to COM, and now to COM+ for Windows 2000. These changes in naming are only partially related to technological changes and are driven to a large extent by marketing purposes.

What, then, is COM? Basically, the Component Object Model, or COM, is a technology that defines a standard way for a client module and a server module to communicate through a specific interface. Here, “module” indicates an application or a library (a DLL); the two modules may execute on the same computer or on different machines connected via a network. Many interfaces are possible, depending on the role of the client and server, and you can add new interfaces for specific purposes. These interfaces are implemented by server objects. A server object usually implements more than one interface, and all the server objects have a few common capabilities, because they must all implement the IUnknown interface.

The good news is that Delphi is fully compliant with COM. When you look at the source code, Object Pascal seems to be easier to use than C++ or other languages for writing COM objects. This simplicity mainly derives from the incorporation of interface types into the Object Pascal language. By the way, interfaces are also similarly used to integrate Java with COM on the Windows platform.

The purpose of COM interfaces is to communicate between two software modules, two executable files, or one executable file and a DLL. Implementing COM objects in DLLs is generally simpler, because in Win32, a program and the DLL it uses reside in the same memory address space. This means that if the program passes a memory address to the DLL, the address remains valid. When you use two executable files, COM has a lot of work to do behind the scenes to let the two applications communicate. This mechanism is called *marshaling*. Note

that a DLL implementing COM objects is described as an *in-process* server, whereas when the server is a separate executable, it is called an *out-of-process* server. However, when DLLs are executing on another machine (DCOM) or inside a host environment (MTS), they are also out-of-process.

## Implementing *IUnknown*

Before we start looking to an example of COM development, I would like to introduce a few COM basics. The first is that every COM object must implement the *IUnknown* interface, also dubbed *IInterface* in Delphi 6 for non-COM usage of interfaces (as we saw in Chapter 3, “The Object Pascal Language: Inheritance and Polymorphism”). This is the base interface from which every Delphi interface inherits, and Delphi provides a couple of different classes with ready-to-use implementations of *IUnknown*/*IInterface*, including *TInterfacedObject* and *TComObject*. The first can be used to have an internal object unrelated with COM, while the second is used to create objects that can be exported by servers. As I’ll discuss in Chapter 20, “From Automation to COM+,” several other classes inherit from *TComObject* and provide support for more interfaces, which are required by Automation servers or ActiveX controls.

As mentioned in Chapter 3, the *IUnknown* interface has three methods: *\_AddRef*, *\_Release*, and *QueryInterface*. Here is the definition of the *IUnknown* interface (extracted from the System unit):

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

The *\_AddRef* and *\_Release* methods are used to implement reference counting. The *QueryInterface* method handles the type information and type compatibility of the objects.

### NOTE

In the code above, you can see an example of an *out* parameter, a parameter passed back from the method to the calling program but without an initial value passed by the calling program to the method. The *out* parameters have been added to Delphi’s Object Pascal language specifically to support COM. It’s also important to note that although Delphi’s language definition for the interface type is designed for compatibility with COM, Delphi interfaces do not require COM. This was already highlighted in Chapter 3, where I built a complex interface-based example with no COM support whatsoever.

You don't usually need to implement these methods, as you can inherit from one of the Delphi classes already supporting them. The most important class is `TComObject`, defined in the `ComObj` unit. When you build a COM server, you'll generally inherit from this class. Because `TComObject` is a complex class, this excerpt shows only its key elements:

```

type
  TComObject = class(TObject, IUnknown, ISupportErrorInfo)
  private
    FNonCountedObject: Boolean;
    FRefCount: Integer;
  protected
    { IUnknown }
    function IUnknown.QueryInterface = ObjQueryInterface;
    function IUnknown._AddRef = ObjAddRef;
    function IUnknown._Release = ObjRelease;
    { ISupportErrorInfo }
    function InterfaceSupportsErrorInfo(const iid: TIID): HRESULT; stdcall;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Initialize; virtual;
    function ObjAddRef: Integer; virtual; stdcall;
    function ObjQueryInterface(const IID: TGUID; out Obj): HRESULT;
      virtual; stdcall;
    function ObjRelease: Integer; virtual; stdcall;
    property RefCount: Integer read FRefCount;
  end;

```

This class implements the `IUnknown` interface (using the `ObjAddRef`, `ObjQueryInterface`, and `ObjRelease` methods, as indicated by the method-mapping statements in the protected portion of the class) and the `ISupportErrorInfo` interface (through the `InterfaceSupportsErrorInfo` method). The implementation of reference counting for the `TComObject` class has been extended to support threading. Instead of using `Inc` and `Dec`, the code uses the thread-safe `InterlockedIncrement` and `InterlockedDecrement` API functions, as you can see in the source code of the class:

```

function TComObject.ObjAddRef: Integer;
begin
  Result := InterlockedIncrement(FRefCount);
end;

function TComObject.ObjRelease: Integer;
begin
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then Destroy;
end;

```



As you can see, the implementation of `Release` destroys the object when there are no more references to it. At first sight, the need to call this method each time you operate on an object seems like a lot of work. However, you might remember from Chapter 3 that when you're using interface variables to refer to objects, Delphi automatically adds the reference-counting calls to the compiled code, which automatically destroys unreferenced objects. This labor-saving feature makes Delphi a convenient tool for COM development.

The most complex method is `QueryInterface`, which in Delphi is actually implemented through the `GetInterface` method of the `TObject` class:

```
function TComObject.ObjQueryInterface(const IID: TGUID; out Obj): HRESULT;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NOINTERFACE;
end;
```

The role of the `QueryInterface` method is twofold:

- `QueryInterface` is used for type checking. The program can ask an object the following questions: Are you of the type I'm interested in? Do you implement the interface I want to call? And the specific methods? If the answer is no, the program can look for another object, maybe asking another server.
- If the answer is yes, `QueryInterface` usually returns a pointer to the object, using its reference output parameter (`out`).

To understand the role of the `QueryInterface` method, it is important to keep in mind that a COM object can implement multiple interfaces, as the event `TComObject` does. When you call `QueryInterface`, you might ask for one of the possible interfaces of the object, using the `TGUID` parameter.

## Globally Unique Identifiers

The `QueryInterface` method has a special parameter of the `TGUID` type. This is an ID that identifies any COM server class and any interface in the system. When you want to know whether an object supports a specific interface, you ask the object whether it implements the interface that has a given ID (which for the default OLE interfaces is determined by Microsoft).

Another ID is used to indicate a specific class, a specific server. The Windows Registry stores this ID, with indications of the related DLL or executable file. The developers of an OLE server define the class identifier.



Both of these IDs are known as GUIDs, or *globally unique identifiers*. If each developer uses a number to indicate its own OLE servers, how can we be sure that these values are not duplicated? The short answer is that we cannot. The real answer is that a GUID is such a long number (with 16 bytes, or 128 bits, or a number with 38 digits!) that it is almost impossible to come up with two random numbers having the same value. Moreover, programmers should use the specific API call `CoCreateGuid` (directly or through their development environment) to come up with a valid GUID that reflects some system information.

In fact, GUIDs created on machines with network cards are guaranteed to be unique, because network cards contain unique serial numbers that form a base for the GUID creation. GUIDs created on machines with CPU IDs (such as the Pentium III) should also be guaranteed unique, even without a network card. With no unique hardware identifier, GUIDs are unlikely to ever duplicate.

**WARNING**

Besides being careful not to copy the GUID from someone else's program (which can result in two completely different COM objects using the same GUID), you should never make up your own ID by entering a casual sequence of numbers. Windows checks the IDs, and using a casual sequence won't generate a valid ID. An OLE server with an invalid ID is not recognized, and you won't get an error message! Windows also won't include an API or technique to validate a GUID. The risk with creating class or interface IDs by hand is that you could coincidentally duplicate a GUID that is already in use somewhere else in the system. However, to avoid this problem, simply press `Ctrl+Shift+G` in the Delphi editor, and you will get a new, properly defined, unique GUID.

Delphi defines a `TGUID` data type (in the `System` unit) to hold these numbers:

```
type
  TGUID = record
    D1: Integer;
    D2: Word;
    D3: Word;
    D4: array [0..7] of Byte;
  end;
```

This structure is actually quite odd but is required by Windows. You can assign a value to a GUID using the standard hexadecimal notation stored inside a string, as in this code fragment:

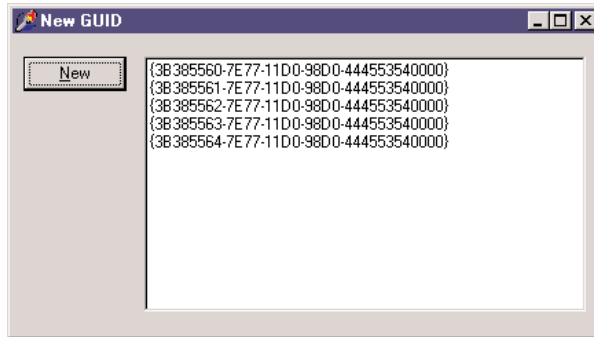
```
const
  Class_ActiveForm1: TGUID = '{1AFA6D61-7B89-11D0-98D0-444553540000}';
```

If you need to generate a GUID manually and not in the Delphi environment, you can simply call the `CoCreateGuid` Windows API function, as demonstrated by the `NewGuid` example (see Figure 19.1). This example is so simple that I've decided not to list its code.

(You can find the source code for this application, along with the chapter's other examples, in the folder for Chapter 19 on the companion CD.)

**FIGURE 19.1:**

An example of the GUIDs generated by the NewGUID example. Values depend on my computer and the time I run this program.



To handle GUIDs, Delphi provides the `GUIDToString` function and the opposite `StringToGUID` function. You can also use the corresponding Windows API functions, such as `StringFromGuid2`, but in this case, you must use the `WideString` type instead of the `string` type. Any time OLE is involved, you have to use the `WideString` type, unless you use Delphi functions that automatically do the required conversion for you. Actually, OLE API functions use the `PWChar` type (pointer to null-terminated arrays of wide characters), but simply casting a `WideString` to `PWChar` does the trick.

### TIP

Keep in mind that GUIDs come in different flavors. The two most important types are interface IDs (or IID), which refer to an interface, and class IDs (or CLSID), which refer to a specific object in a server. These two kinds of IDs both use the GUID style.

## The Role of Class Factories

When we register the GUID of a COM object in the Registry, we can use a specific API function to create the object, such as the `CreateComObject` API:

```
function CreateComObject (const ClassID: TGUID): IUnknown;
```

This API function will look into the Registry, find the server registering the object with the given GUID, load it, and, if the server is a DLL, call the `DLLGetClassObject` method of the DLL. This is a function every in-process server must provide and export:

```
function DLLGetClassObject (const CLSID, IID: TGUID;
    var Obj): HRESULT; stdcall;
```

This API function receives as parameters the requested class and interface, and it returns an object in its reference parameter. The object returned by this function is a *class factory*.

Now, what is a class factory? As the name suggests, a class factory is an object capable of creating other objects. Each server can have multiple objects. The server exposes the class factory, and the class factory can create one of these various objects. Each object, then, can have multiple interfaces. One of the many advantages of the Delphi simplified approach to COM development is that the system can provide a class factory for us. For this reason, I'm not going to add a class factory to our simple example.

The call to the `CreateComObject` API doesn't stop at the creation of the class factory, however. After retrieving the class factory, `CreateComObject` calls the `CreateInstance` method of the `IClassFactory` interface. This method creates the requested object and returns it. If no error occurs, this object becomes the return value of the `CreateComObject` API.

By setting up this mechanism (including the class factory and the `DLLGetClassObject` call), you make it very simple to create objects. `CreateComObject` is just a simple function call with a complex behavior behind the scenes. What's great in Delphi is that the complex mechanism is handled for you by the run-time system. So it's time to start looking in detail at how Delphi makes COM really easy to master.

## Class Factories and Other Delphi COM Classes

Besides the `TComObject` class, Delphi includes several other predefined COM classes. We'll use them in the following sections, but here is a list of the most important COM classes of the Delphi VCL:

- `TInterfacedObject`, defined in the `System` unit, inherits from `TObject` and implements the `IUnknown` interface. It is used only for internal objects.
- `TComObject`, defined in the `ComObj` unit, inherits from `TObject` and implements both the `IUnknown` interface and the `ISupportErrorInfo` interface. Unlike `TInterfacedObject`, this class also has a related class factory.
- `TTypedComObject`, defined in the `ComObj` unit, inherits from `TComObject` and implements the `IProvideClassInfo` interface (in addition to the `IUnknown` and `ISupportErrorInfo` interfaces already implemented by the base class, `TComObject`).
- `TAutoObject`, defined in the `ComObj` unit, inherits from `TTypedComObject` and implements also the `IDispatch` interface.
- `TActiveXObject1`, defined in the `AxCtrls` unit, inherits from `TAutoObject` and implements several interfaces (`IPersistStreamInit`, `IPersistStorage`, `IOleObject`, and `IOleControl`, to name just a few).

For each of these classes, Delphi also defines a class factory. The class factory classes form another hierarchy, with the same structure. Their names are `TComObjectFactory`, `TTypedComObjectFactory`, `TAutoObjectFactory`, and `TActiveXObject1Factory`. Class factories are important, and every COM server requires them. Usually we simply use class factories by

creating an object in the initialization section of the unit defining the corresponding server object class.

## A First COM Server

There is no better way to understand COM than to build a simple COM server hosted by a DLL. A library hosting a COM object is indicated in Delphi as an ActiveX library. For this reason we can start the development of this project by selecting File > New > Other, moving to the ActiveX page, and selecting the ActiveX Library option. This generates a project file I've saved as FirstCom on the companion CD. This is the complete source code:

```
library FirstCom;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

The four functions exported by the DLL are required for COM compliance and are used by the system as follows:

- To access the class library (DllGetClassObject)
- To check whether the server has destroyed all its objects and can be unloaded from memory (DllCanUnloadNow)
- To add or remove information about the server in the Windows Registry (DllRegisterServer and DllUnregisterServer)

You generally don't have to implement these functions, because Delphi provides a default implementation in the ComServ unit. For this reason, in the code of our server, we only need to export them.

## COM Interfaces and Objects

Now that we have the structure of our COM server in place, we can start developing it. The first step is to write the code of the interface we want to implement in the server. The interface can be very similar to the code of an abstract class, listing all the methods we want to make available from our server. (I have already discussed Object Pascal interfaces in Chapter 3.) Here is the code of a simple interface, which you should add to a separate unit (called NumIntf in the example):

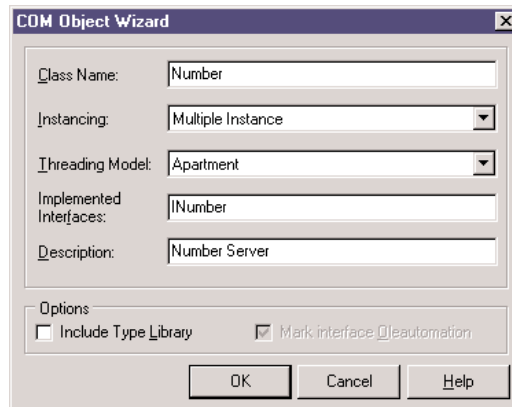
```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

The IID was added to the code by pressing the Ctrl+Shift+G key combination.

After declaring the custom interface, we can add the actual object to the server. To accomplish this, we can use the COM Object Wizard (available in the ActiveX page of the File > New > Other dialog box). You can see this wizard's dialog box in Figure 19.2. Here you should enter the name of the class of the server, the interface you want to implement, and a description. I've disabled the generation of the type library to avoid introducing too many topics at once. You should also choose an instancing and a threading model, as described in the related sidebar.

**FIGURE 19.2:**

The COM Object Wizard



The code generated by the COM Object Wizard is actually quite simple. The interface contains the definition of the class to fill with methods and data:

```
type
  TNumber = class(TComObject, INumber)
    protected
      {Declare INumber methods here}
    end;
```

The server class inherits from the TComObject class, which I discussed in the last section. In the code generated by the wizard, after the server class comes the definition of the GUID for the server:

```
const
  Class_Number: TGUID = '{5B2EF181-3AAE-11D3-B9F1-00000100A27B}';
```

Finally, there is some code in the initialization section (which uses most of the options we've set up in the wizard's dialog box):

```
initialization
  TComObjectFactory.Create(ComServer, TNumber, Class_Number, 'Number',
    'Number Server', ciMultiInstance, tmApartment);
```

This code creates an object of the TComObjectFactory class, passing as parameters the global ComServer object, a class reference to the class we've just defined, the GUID for the class, the server name, the server description, and the instancing and threading models we want to use.

The global ComServer object, defined in the ComServ unit, is a manager of the class factories available in the server library. It uses its own ForEachFactory method to look for the class supporting a given COM object request, and it keeps track of the number of allocated objects. As we've already seen, in fact, the ComServ unit implements the functions required by the DLL to be a COM library.

Having examined the source code generated by the wizard, we can now complete it by adding to the TNumber class the methods required for implementing the INumber interface. First, write the declaration of the methods:

```
type
  TNumber = class(TComObject, INumber)
    private
      fValue: Integer;
    public
      function GetValue: Integer; virtual; stdcall;
      procedure SetValue (New: Integer); virtual; stdcall;
      procedure Increase; virtual; stdcall;
    end;
```

At this point, simply activate class completion by pressing Shift+Ctrl+C and fill the methods with the proper code. This is so straightforward that I'm not going to list it here; you can find the source code on the companion CD.

## COM Instancing and Threading Models

When you create a COM server, you should choose a proper instancing and threading model, which can significantly affect the behavior of the COM server.

*Instancing* affects only out-of-process servers (any COM server in a separate executable file, rather than a DLL) and can assume three values:

**Multiple** indicates that when several client applications require the COM object, the system starts multiple instances of the server.

**Single** indicates that, even when several client applications require the COM object, there is only one instance of the server application; it creates multiple internal objects to service the requests.

**Internal** indicates that the object can only be created inside the server; client applications cannot ask for one.

The second decision relates to the thread support of the COM object, which is valid for in-process servers only (DLLs). The threading model is a joint decision of the client and the server application: if both sides agree on one model, it is used for the connection. If no agreement is found, COM can still set up a connection using marshaling, which can slow down the operations. And keep in mind that a server must not only publish its threading model in the Registry (as a result of setting the option in the wizard); it must also follow the rules for that threading model in the code. Here are the key highlights of the various threading models:

**The Single model** indicates no real support for threads. The requests reaching the COM server are serialized, so that the client can perform one operation at a time.

**The Apartment model, or "single-threaded Apartment"** indicates that only the thread that created the object can call its methods. This means that the requests for each server object are serialized, but other objects of the same server can receive requests at the same time. For this reason, the server object must take extra care only to access global data of the server (using critical sections, mutexes, or some other synchronization techniques). This is the threading model generally used for ActiveX controls inside Internet Explorer.

**The Free model, or "multi-threaded Apartment"** indicates that the client has no restrictions, which means that multiple threads can use the same object at the same time. For this reason, every method of every object must protect itself and the nonlocal data it uses against multiple simultaneous calls. This threading model is more complex for a

*Continued on next page*

server to support than the Single and Apartment models, because even access to the object's own instance data must be handled with thread-safe care.

**The fourth option, Both** indicates that the server object supports both the Apartment model and the Free model.

**The final option, Neutral** was introduced in Windows 2000 and is available only under COM+. It indicates that multiple clients can call the object on different threads at the same time, but COM guarantees you that the same method is not invoked twice at the same time. Guarding for concurrent access to the data of the object is required. Under COM, it is mapped to the Apartment model.

## Initializing the COM Object

If you look back at the definition of the `TComObject` class, you will notice it has a nonvirtual constructor. (Actually, it has multiple nonvirtual constructors, which I've omitted from the listing.) Each `TComObject` constructor calls the virtual `Initialize` method. For this reason, if you want to customize the creation of an object and then initialize it, you should not define a new constructor (which will never be called). What you should do is override its `Initialize` method, as I've done in the `TNumber` class. Here is the final version of this class:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
    procedure Initialize; override;
    destructor Destroy; override;
  end;
```

As you can see, I've also overridden the destructor of the class, because I wanted to test the automatic destruction of the COM objects provided by Delphi. Here is the code for this pseudoconstructor and the destructor:

```
procedure TNumber.Initialize;
begin
  inherited;
  fValue := 10;
end;
```



```

destructor TNumber.Destroy;
begin
  inherited;
  MessageBox (0, 'Object Destroyed', 'TDLLNumber', mb_OK); // API ca11
end;

```

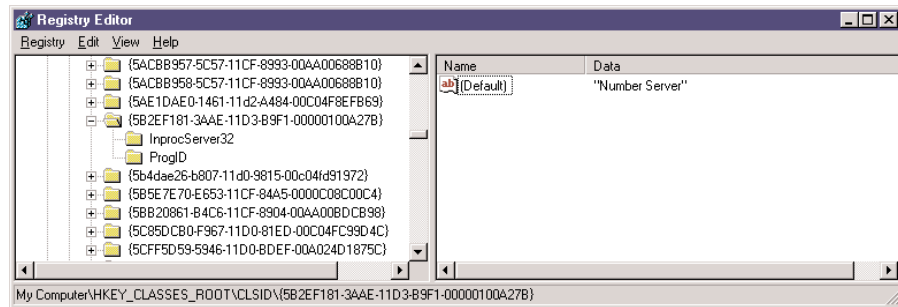
In the first method, calling the inherited version is good practice, even though the `TComObject.Initialize` method has no code in this version of Delphi. The destructor, instead, must call the base class version. This is the code required to make our COM object work properly and to let us know when an object is actually destroyed.

## Testing the COM Server

Now that we've finished writing our COM server object, we can register and use it. Simply compile its code and then use the `Run > Register ActiveX Server` menu command in Delphi. You do this to register the server on your own machine, with the results you can see in Figure 19.3.

**FIGURE 19.3:**

The new registered server in Windows RegEdit



When you distribute this server, you should install it on the client computers. To accomplish this, you can write a REG file to install the server in the Registry. However, this is not really the best approach, because the server already includes a function you can activate to register the server. This function can be activated by the Delphi environment, as we've seen, or in a few other ways:

- You can pass the COM server DLL as a command-line parameter to Microsoft's `RegSvr32.exe` program, found in the `\Windows\System` directory.
- You can use the similar `TRegSvr.exe` demo program that ships with Delphi. (The compiled version is in the `\Bin` directory, and its source code is in the `\Demos\ActiveX` directory.)
- You can let an installation builder program call the registration function of the server.

Having registered the server, we can now turn to the client side of our example. This time, the example is called `TestCom` and is stored in a separate directory. In fact, the program loads the server DLL through the OLE/COM mechanism, thanks to the server information present in the Registry, so it's not necessary for the client to know which directory the server resides in.

The form displayed by this program is very similar to the one we've used to test the object inside the DLL. In the client program, you must include the source code file with the interface and redeclare the COM server GUID. Of course, the code of the program's `FormCreate` method should be updated to create the required COM objects. The program starts with all the buttons disabled (at design time), and it enables them only after an object has been created. This way, if an exception is raised while creating one of the objects, the buttons related to the object won't be enabled:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    // create first object
    Num1 := CreateComObject (Class_Number) as INumber;
    Num1.SetValue (SpinEdit1.Value);
    Label1.Caption := 'Num1: ' + IntToStr (Num1.GetValue);
    Button1.Enabled := True;
    Button2.Enabled := True;

    // create second object
    Num2 := CreateComObject (Class_Number) as INumber;
    Label2.Caption := 'Num2: ' + IntToStr (Num2.GetValue);
    Button3.Enabled := True;
    Button4.Enabled := True;
end;
```

Notice in particular the call to `CreateComObject` and the following `as` cast. The API call starts the COM object-construction mechanism I've already described in detail. This call also dynamically loads the server DLL. The return value is an `IUnknown` object. This object must be converted to the proper interface type before assigning it to the `Num1` and `Num2` fields, which now have the interface type `INumber` as their data type:

```
type
    TForm1 = class(TForm)
        ...
    private
        Num1, Num2 : INumber;
```

**WARNING** To downcast an interface to the actual type, *always* use the `as` cast, which for interfaces performs a `QueryInterface` call behind the scenes. This provides some protection, because it raises an exception if the interface you are casting to is not supported by the given object. In the case of interfaces, the `as` cast is the only way to *extract* an interface from another interface. If you write a plain cast of the form `INumber(CreateComObject(Class_Number))`, the program will crash, even if the cast seems to make sense, as in the case above. Casting an interface pointer to another interface pointer is an error. Period. Never do it.

In Figure 19.4, you can see the output of this test program, which is very similar to the previous version. Notice that this time, `Num2` shows the initial value of the object at start-up, as set up in its `Initialize` method. Notice also that I've added one more button, which creates a third temporary COM object:

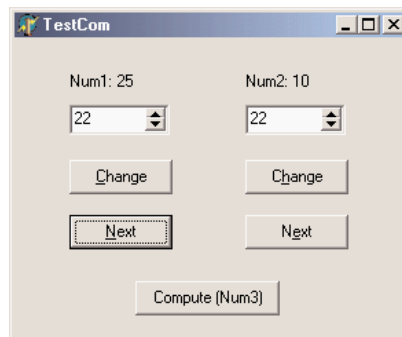
```

procedure TForm1.Button5Click(Sender: TObject);
var
    Num3: INumber;
begin
    // create a new temporary COM object
    Num3 := CreateComObject (Class_Number) as INumber;
    Num3.SetValue (100);
    Num3.Increase;
    ShowMessage ('Num3: ' + IntToStr (Num3.GetValue));
end;

```

**FIGURE 19.4:**

The output of the `TestCom` example, a COM client



Pressing this button, you simply get the value of the number following 100. To see why I added this method to the example, you need to press the button a second time, after the message showing the result. Now you get a second message, indicating that the object has been destroyed. This demonstrates that simply letting an interface object go out of scope automatically calls the object's `Release` method, decreases the object's reference count, and destroys

the object if its reference count reaches zero. Chapter 3 described this reference-counting mechanism in more detail.

The same happens to the other two objects as soon as the program terminates. Even if the program doesn't explicitly destroy the two objects in the `FormDestroy` method, they are indeed destroyed, as the message shown by their `Destroy` destructor clearly demonstrates. This happens because they were declared to be of an interface type, and Delphi is going to use reference counting for them.

## Using Interface Properties

As a further small step, we can extend the example by adding a property to the `INumber` interface. When you add a property to an interface, you indicate the data type and then the read and write directives. You can have read-only or write-only properties, but the read and write clauses must always refer to a method because interfaces don't hold anything else but methods.

Here is the updated interface, which is part of the `PropCom` example:

```
type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer read GetValue write SetValue;
    procedure Increase; stdcall;
end;
```

I've given this interface a new name and, what's even more important, a new interface ID. I could have inherited the new interface type from the previous one, but this would have provided no real advantage. COM by itself doesn't really support inheritance, and from the perspective of COM, all interfaces are different simply because they have different interface IDs. Needless to say, in Delphi we can use inheritance to improve the structure of the code of the interfaces and of the server objects implementing them.

In the `PropCom` example, I've updated the server class declaration simply by writing:

```
type
  TD11Number = class (TComObject, INumberProp)
  ...
```

This class also has a new server object ID. The client program, also on the CD, can now simply use the `Value` property instead of the `SetValue` and `GetValue` methods. Here is a small excerpt from the `FormCreate` method:

```
Num1 := CreateComObject (Class_NumPropServer) as INumberProp;
Num1.Value := SpinEdit1.Value;
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

The difference between using methods and properties for an interface is only syntactical, because interface properties cannot access private data as class properties can. By using properties, we can make the code a little more readable.

## Calling Virtual Methods

We've built a couple of examples based on COM, but you might still feel uncomfortable with the idea of a program calling methods of objects that are created within a DLL. How is this possible if those methods are not exported by the DLL? The COM server, the DLL, creates an object and returns it to the calling application. By doing this, the DLL creates an object with a *virtual method table* (VMT). (Remember that all the interface methods are virtual by default.)

Because every object embeds a pointer to its VMT, the main program receives an object, and also a way to work on it, by calling its virtual methods. The main program doesn't need to know the memory address of those methods, because the objects know it, exactly as they do with a polymorphic call. But COM is even more powerful than this: you don't even have to know which programming language was used to create the object, provided its VMT follows the standard dictated by COM.

---

**TIP**

The COM-compatible VMT implies also a strange effect. The method names are not important, provided their address is in the proper position in the VMT. This is why you can map a method of an interface to an actual function implementing it.

To sum things up, we can say that COM provides a language-independent binary standard for objects. The objects you share among modules are compiled, and their VMT has a particular structure, determined by COM and not by the development environment you've used.

## Windows Shell Programming

In the last section, we built a fully standard COM object, packaged it as an in-process server, and used it from a standard client. However, the COM interface we implemented was a custom interface we'd built. Now we can try to build clients and servers related to the Windows shell interfaces, which are all based on COM. The original Windows API was basically a collection of functions, but all the most recent APIs are generally based on COM.

The following sections use some existing servers that are part of the Windows shell; we'll write a client application and use the COM servers provided by the system. This case illustrates the difference from the traditional use of the Windows API calls. I'm also going to write some COM servers to be used by the Windows system, particularly the Explorer. This

case illustrates the difference from the traditional development of a callback function invoked by the system.

## Creating Shortcuts

One of the simplest shell interfaces we can use in a client application is the `IShellLink` interface. This interface relates to Windows shortcuts and allows programmers to access the information of an existing shortcut or to create a new one. In the `ShCut` example on the CD-ROM, I'm going to create various types of shortcuts, all referring to the program itself. Of course, once you understand how to do this, you can easily extend the example and create shortcuts for any program or file.

The example has an edit box for the name of the shortcut, a few check boxes, and two buttons. When the Create button is pressed, the text in the edit box is used as the name of a new shortcut, which is placed in the current directory, on the desktop, or in the Start menu. These options are not exclusive; a user can create multiple shortcuts at once.

The most important code is at the very beginning of this method. The `CreateComObject` call creates a system object, as indicated by the `GUID` passed as a parameter. The result of this call (which is an `IUnknown` interface) is converted both to an `IShellLink` interface and to an `IPersistFile` interface:

```
uses
    ComObj, ActiveX, ShlObj, Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
    AnObj: IUnknown;
    ShLink: IShellLink;
    PFile: IPersistFile;
    FileName: string;
    WFileName: WideString;
    Reg: TRegIniFile;
begin
    // access the two interfaces of the object
    AnObj := CreateComObject (CLSID_ShellLink);
    ShLink := AnObj as IShellLink;
    PFile := AnObj as IPersistFile;
```

Actually, we could have written the last three lines of code above using this shorter notation:

```
ShLink := CreateComObject (CLSID_ShellLink) as IShellLink;
PFile := ShLink as IPersistFile;
```

If you look at similar examples built in other languages, you'll notice that to access the `IPersistFile` interface, the programs use custom calls to the `QueryInterface` method. The two `as` expressions basically call `QueryInterface` for us.

Once we have the `IShellLink` interface, we can call some of its methods, such as `SetPath` and `SetWorkingDirectory`:

```
// get the name of the application file
FileName := ParamStr (0);
// set the link properties
ShLink.SetPath (PChar (FileName));
ShLink.SetWorkingDirectory (PChar (ExtractFilePath (FileName)));
```

Once we've set up the shell link object, we have to save it, depending on the status of the three check boxes, calling the `Save` method of the `IPersistFile` interface of the object. The simplest version is the one used to save the link in the current directory:

```
// save the file in the current dir
if cbDir.Checked then
begin
    // using a WideString
    WFileName := ExtractFilePath (FileName) + EditName.Text + '.lnk';
    PFile.Save (PWChar (WFileName), False);
end;
```

The call to the `Save` method (which creates the physical LNK file) requires a “pointer to wide char” parameter. The simplest way to obtain this is to declare a long string and then cast it to a `PWChar`. Do not try casting a plain string to `PWChar`—the compiler will emit a warning and the program won't work!

To create the shortcut on the desktop or in the Start menu, we should first determine the corresponding system folder by looking up the proper value in the Registry. By writing the program this way, we ensure it will work on different versions of Windows and on localized versions as well. Here is the source code for the last two check boxes:

```
// save on the desktop
if cbDesktop.Checked then
begin
    Reg := TRegIniFile.Create(
        'Software\Microsoft\Windows\CurrentVersion\Explorer');
    WFileName := Reg.ReadString ('Shell Folders', 'Desktop', '') +
        '\ ' + EditName.Text + '.lnk';
    Reg.Free;
    PFile.Save (PWChar (WFileName), False);
end;
// save in the Start Menu
if cbStartMenu.Checked then
begin
```

```

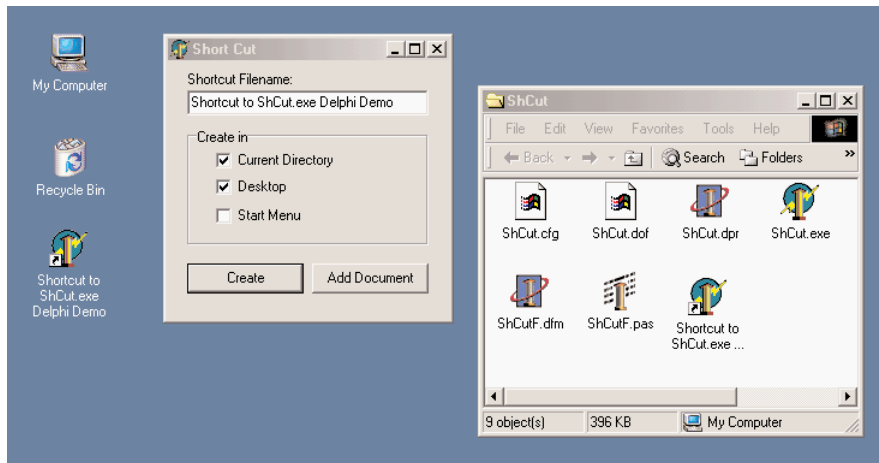
Reg := TRegIniFile.Create(
  'Software\Microsoft\Windows\CurrentVersion\Explorer');
WFileName := Reg.ReadString ('Shell Folders', 'Start Menu', '') +
  '\' + EditName.Text + '.lnk';
Reg.Free;
PFile.Save (PWChar (WFileName), False);
end;

```

To look up the information in the Registry, I've used the `TRegIniFile` class, although there are other related classes in the VCL, such as the `TRegistry` class. The effect of running this program and pressing the button is that Windows will add a new link in the directory of the project, on the desktop, or in the Start menu. You can see an example of the program in Figure 19.5.

**FIGURE 19.5:**

The simple user interface of the `ShCut` example, and two shortcuts created with it in the project folder and on the desktop



## Using Shell APIs and Objects

As an extra feature, the program can also add a new document to the list of recently used ones, calling the `SHAddToRecentDocs` method:

```

procedure TForm1.Button2Click(Sender: TObject);
var
  ProjectFile: string;
begin
  ProjectFile := ChangeFileExt (ParamStr (0), '.dpr');
  SHAddToRecentDocs (SHARD_PATH, PChar(ProjectFile));
end;

```



This has very little to do with COM, and I've added it to the example only to highlight that there is a very large number of shell-related APIs, available in the `ShlObj` unit, besides the original and more limited `ShellApi` unit.

Another example, available in the source code for this chapter and called `FindFolders`, highlights the use of another *plain* (non-COM) shell function, `SHBrowseForFolder`. In the example you can see the following code:

```
procedure TForm1.btnBrowseClick(Sender: TObject);  
var  
    bi: TBrowseInfo;  
    pidl: pItemIdList;  
    strpath, displayname: string;  
begin  
    SetLength (displayname, 100);  
  
    bi.hwndOwner := Handle;  
    bi.pidlRoot := nil;  
    bi.pszDisplayName := pChar (displayname);  
    bi.lpszTitle := 'Select a folder';  
    bi.ulFlags := bif_StatusText;  
    bi.lpfm := nil;  
    bi.lParam := 0;  
  
    pidl := SHBrowseForFolder (bi);  
  
    SetLength (strPath, 100);  
    SHGetPathFromIdList (pidl, PChar(strPath));  
    Edit1.Text := strPath;  
end;
```

The `FindFolders` example even shows some Delphi-specific APIs to interact with files and folders (available also on Linux) including `SelectDirectory`, which has the same effect of `SHBrowseForFolder` but a different user interface. The example also uses the `DirectoryExists` and `ForceDirectories` functions, available in the `FileCtrl` unit. You can see how they are used by looking in the source code of the example.

---

**NOTE**

Notice that in Delphi 6, some of the Shell API is also encapsulated in the sample `ShellListView`, `ShellTreeView`, and `ShellComboBox` controls. I've used these controls in a few examples throughout the book, including the `DirDemo` example of Chapter 18, "Writing Database Components."

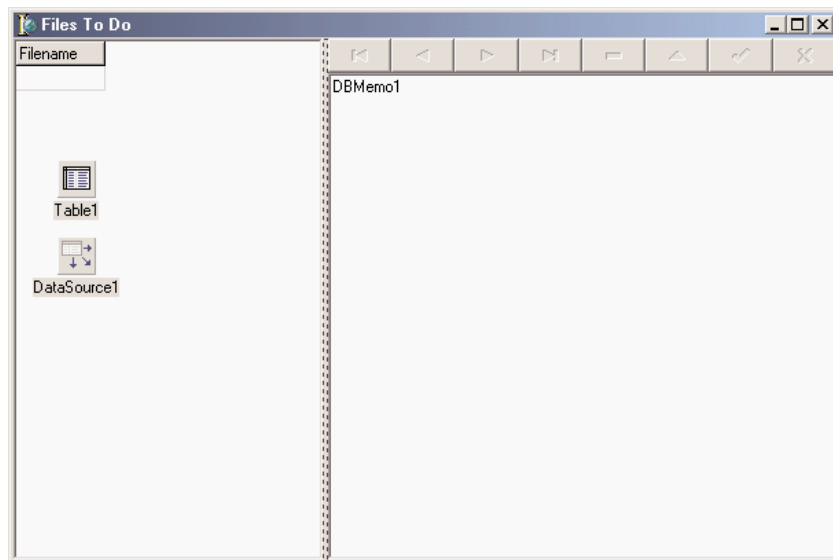
## The “To-Do File” Application

As a second example of integrating a Delphi program with the system shell, I’ve tried to write a simple real-world application that uses file dragging and a context menu handler. I’ll start with the file dragging first, because this will actually introduce some of the techniques used by the context menu handler.

As I mentioned, this application is actually useful; you can use it to create a sort of “to-do list.” It is based on a Paradox table that stores filenames and notes about the files. The form of the application has a DBGrid component showing only a single column containing the filenames and a memo control hosting the notes related to the current file. You can see this form at design time in Figure 19.6.

**FIGURE 19.6:**

The form of the ToDoFile example at design time



### TIP

Using a single-column DBGrid is the only way in Delphi to show a list of the available records in a list-box format. The alternative, of course, is to fill a list box with custom code and then manually navigate in the database table when the selection in the list box changes. This manual approach is, of course, less efficient when we have many records, because the program needs to scan them all to fill the list box, while the DBGrid loads only the record it currently displays.

Notice that the navigator component has no “New Record” button, and the DBGrid is set up as a read-only component. In fact, users should not be able to create new records except by dragging a file onto the form, and they’re not allowed to change the filename field in any way (except by deleting it). All the user can do is edit the notes field, entering a description of the operations to be done on the file.

## Creating the Database

To create the database table for this example, I've used the `FieldDefs` property to define the structure and set the `StoreDefs` property to `True` to save the table definition along with the form DFM file. The table has two fields, a string field called `Filename` and a memo field called `Notes`. Of course, you can also create the table at design time, using the table component's local menu. The program, however, calls the `CreateTable` method in the `OnCreate` event handler, unless this has already been done:

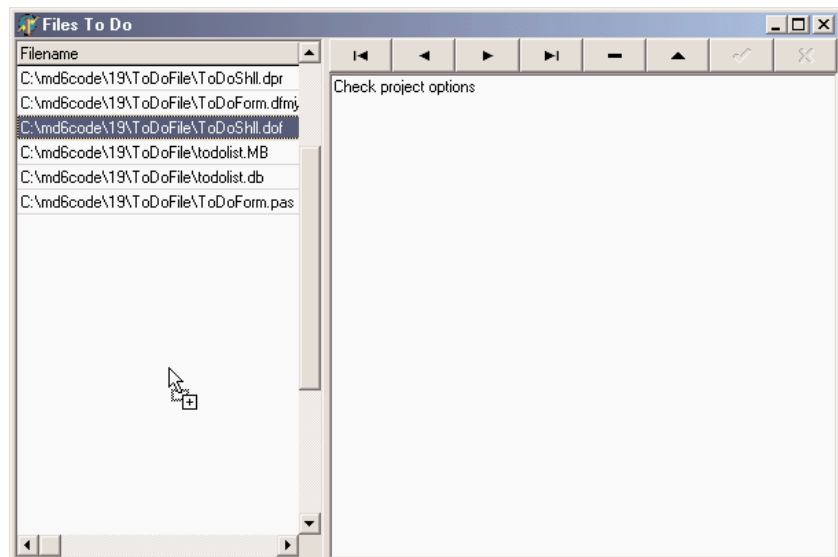
```
procedure TToDoFileForm.FormCreate(Sender: TObject);
begin
    // eventually create the table
    if not Table1.Exists then
        Table1.CreateTable;
    // activate the table
    Table1.Activate;
    // accept dragging to the form
    DragAcceptFiles(Handle, True);
end;
```

## Dragging Files to the Form

As you can see in the listing above, the form initialization code also registers the window with the system as a file-dragging target, by calling the `DragAcceptFiles` Windows API function. As a result, the application's cursor changes to the typical "drag accept" icon when a file is dragged over it. You can see an example of this cursor in Figure 19.7.

**FIGURE 19.7:**

The drag-accept cursor displayed by the `ToDoFile` application as a user drags a file over it



When a file-dragging operation is performed, the system sends the window a `wm_DropFiles` message. This message passes (among its other parameters) a handle to a file-drop structure from which you can extract information by using the `DragQueryFile` API function. When this API function is called with the `$FFFFFFFF` parameter, it returns the number of files dragged to the window; when it is called with a numeric parameter, it fills a buffer with the name of that file. For this reason, the code of a `wm_DropFiles` message handler gets the number of files first and then loops for each of the files, as the following listing demonstrates:

```
procedure TToDoFileForm.DropFiles(var Msg: TWmDropFiles);
var
    nFiles, I: Integer;
    Filename: string;
begin
    // get the number of dropped files
    nFiles := DragQueryFile (Msg.Drop, $FFFFFFFF, nil, 0);
    // for each file
    try
        for I := 0 to nFiles - 1 do
            begin
                // allocate memory
                SetLength (Filename, 80);
                // read the file name
                DragQueryFile (Msg.Drop, I, PChar (Filename), 80);
                // normalize file
                Filename := PChar (Filename);
                // add a new record
                Table1.InsertRecord ([Filename, '']);
            end;
        finally
            DragFinish (Msg.Drop);
        end;
        // open the (last) record in edit mode
        Table1.Edit;
        // move the input focus to the memo
        DBMemo1.SetFocus;
    end;
```

As you can see in the preceding code, for every new file, the program inserts a new record with the corresponding filename and an empty field for the notes. Then, for the last file being dragged, the program opens the record in edit mode and moves the focus to the memo control, so that a user can fill the notes for the file.

## Creating a Context-Menu Handler

Now that we have the base program running, we can add a shell extension to the system to let the user simply select a file and “send” it to the application without having to do the dragging operation, which is not always handy when there are many programs running. A context-menu extension is one of the available Windows shell extensions and is activated every time a user right-clicks a file in the Windows Explorer.

Technically, a context menu is a COM server exposing an internal object that is going to be created and used by the system. A context-menu COM object must implement two different interfaces, `IContextMenu` and `IShellExtInit`. The first interface defines specific actions for the context menu, such as defining the number of menu items to add and their text, while the second interface defines a way to access the file or files the user is operating on. This is the resulting definition of the COM server object class:

```

type
  TToDoMenu = class(TComObject, IUnknown, IContextMenu, IShellExtInit)
  private
    fFileName: string;
  protected
    {Declare IContextMenu methods here}
    function QueryContextMenu(Menu: HMENU; indexMenu,
      idCmdFirst, idCmdLast, uFlags: UINT): HRESULT; stdcall;
    function InvokeCommand(
      var lpici: TCMInvokeCommandInfo): HRESULT; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HRESULT; stdcall;
    {Declare IShellExtInit methods here}
    function IShellExtInit.Initialize = InitShellExt;
    function InitShellExt (pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HRESULT; stdcall;
end;

```

Notice that the class implements the `Initialize` method of the `IShellExtInit` interface with a differently named method, `InitShellExt`. The reason is that I wanted to avoid confusion with the `Initialize` method of the `TComObject` base class, which is the hook we have to initialize the object, as described earlier in this chapter. Let’s examine the `InitShellExt` method first; it is definitely the most complex one:

```

function TToDoMenu.InitShellExt(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HRESULT; stdcall;
var
  medium: TStgMedium;
  fe: TFormatEtc;
begin
  Result := E_FAIL;

```

```

// check if the lpobj pointer is nil
if Assigned (lpobj) then
begin
  with fe do
  begin
    cfFormat := CF_HDROP;
    ptd := nil;
    dwAspect := DVASPECT_CONTENT;
    lindex := -1;
    tymed := TYMED_HGLOBAL;
  end;
  // transform the lpobj data to a storage medium structure
  Result := lpobj.GetData(fe, medium);
  if not Failed (Result) then
  begin
    // check if only one file is selected
    if DragQueryFile (medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
    begin
      SetLength (fFileName, 1000);
      DragQueryFile (medium.hGlobal, 0, PChar (fFileName), 1000);
      // realign string
      fFileName := PChar (fFileName);
      Result := NOERROR;
    end
    else
      Result := E_FAIL;
    end;
  ReleaseStgMedium(medium);
end;
end;

```

The initial portion of the method transforms the pointer to the `IDataObject` interface, which we receive as a parameter, into the same data structure used in a file drop operation, so that we can read the file information by using the `DragQueryFile` function again. This complex way of coding is actually the simplest one you can use! At the end of this operation, we have the value of the filename. Any selection of multiple files is not accepted.

We can now look at the methods of the `IContextMenu` interface. The first, `QueryContextMenu`, is used to add new items to the local menu of the file. In this case, we add a new menu item (calling the `InsertMenu` API function) only if the `ToDoFile` application is running. We can determine this by searching for a window corresponding to the `TToDoFileForm` class, which should be unique in the system. The result of the function is the number of items added to the menu:

```

function TToDoMenu.QueryContextMenu(Menu: HMENU;
  indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HRESULT;
begin

```

```

// add entry only if the program is running
if FindWindow ('TToDoFileForm', nil) <> 0 then
begin
    // add a new item to context menu
    InsertMenu (Menu, indexMenu, MF_STRING or MF_BYPOSITION, idCmdFirst,
        'Send to ToDoFile');
    // return the number of menu items added
    Result := 1;
end
else
    Result := 0;
end;

```

Now that items have been added to the menu, a user can select them. While he or she moves over the items, a descriptive message is displayed in the status bar of the Windows Explorer. The menu ID (`idCmd`) we receive in the `GetCommandString` method is simply the relative number, starting with zero, of the items we have added to the menu. When the cursor is over an item, we simply copy a string with its description to the buffer provided by the system:

```

function TToDoMenu.GetCommandString(idCmd, uType: UINT;
    pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HRESULT;
begin
    if idCmd = 0 then
        begin
            // return help string for menu item
            strCopy (pszName, 'Add file to the ToDoFile database');
            Result := NOERROR;
        end
    else
        Result := E_INVALIDARG;
    end;

```

The final step is the operation to do once a menu item is actually selected. The `InvokeCommand` method receives a pointer to a structure holding the request. This method follows a standard pattern of first checking that the request is valid by looking at the two 16-bit words of the `lpici.lpVerb` value. After these preliminary (but required) steps, we check the value to see which menu item was activated; or, if the context menu has only one item, as in this case, we simply test for a value of zero. The following is the skeleton of the code, before we add the specific action:

```

function TToDoMenu.InvokeCommand (var lpici: TCMInvokeCommandInfo): HRESULT;
begin
    Result := NOERROR;
    // make sure we are not being called by an application
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
        begin
            Result := E_FAIL;
            Exit;
        end;

```

```

end;
// make sure we aren't being passed an invalid argument number
if LoWord(lpici.lpVerb) > 0 then
begin
    Result := E_INVALIDARG;
    Exit;
end;
// execute the command specified by lpici.lpVerb
if LoWord(lpici.lpVerb) = 0 then
begin
    // actual code still missing here
end
end;
end;

```

## Sending Data to Another Application with *wm\_CopyData*

Because we have the filename the user is operating on, all we have to do in the context-menu handler is send this name to the main form of the *ToDoFile* application. The problem is that the context-menu handler DLL runs in the Windows Explorer process, so it cannot send the value of a memory pointer to another process. This would simply be useless; as in Win32, different applications have separate memory address spaces.

We saw in the last chapter that one way to share data among applications is to use a memory-mapped file. Another technique, which is actually better in this case, is to use the *wm\_CopyData* message. This is a special Windows message, which can be used to send a memory buffer to another application: Windows will resolve all the memory conversion problems for us. A program basically fills the *CopyDataStruct* data structure with the data and indicates its length, and then must use the *SendMessage* API to forward it to a destination window. For this reason we need to use *FindWindow* again to get the handle of the main window of the *ToDoFile* application. Here is the rest of the code of the *InvokeCommand* method:

```

var
    hwnd: THandle;
    cds: CopyDataStruct;
begin
    ...
    if LoWord(lpici.lpVerb) = 0 then
    begin
        // get the handle of the window
        hwnd := FindWindow ('TToDoFileForm', nil);
        if hwnd <> 0 then
        begin
            // prepare the data to copy
            cds.dwData := 0;
            cds.cbData := length (fFileName);
            cds.lpData := PChar (fFileName);

```



```
        // activate the destination window
        SetForegroundWindow (hwnd);
        // send the data
        SendMessage (hwnd, WM_CopyData, lpici.hWnd, Integer (@cDs));
    end;
end;
```

**NOTE**

Before sending the data, we must activate the destination window by calling the `SetForegroundWindow` API. This is necessary because we are going to activate a window that was created by another thread, something Windows doesn't normally do. Notice also that if you write this call in the `ToDoFile` application as it receives the `WM_CopyData` message, it will produce no effect at all.

As the context-menu handler sends data to it, the application has to be extended to handle the `WM_CopyData` message. In this event handler, we receive the same structure we sent for the other side, although between the send operation done by the context-menu handler and the receive operation done by the application. Windows takes care of mapping the data properly to the other address space. As a result, extracting the filename is actually very simple, but keep in mind that this is so only because Windows does a lot of work behind the scenes. Using a plain Windows message other than `WM_CopyData` will never work!

Here is the code I've added to the form of the `ToDoFile` application. It does several things: It restores the application if it was minimized, retrieves the name of the file, inserts a new record in the database table, copies the filename, and moves the focus to the memo control once more.

```
procedure TToDoFileForm.CopyData(var Msg: TWmCopyData);
var
    Filename: string;
begin
    // restore the window if minimized
    if IsIconic (Application.Handle) then
        Application.Restore;

    // extract the filename from the data
    Filename := PChar (Msg.CopyDataStruct.lpData);
    // insert a new record
    Table1.Insert;
    // set up the file name
    Table1.FieldName ('Filename').AsString := Filename;
    // move the input focus to the memo
    DBMemo1.SetFocus;
end;
```

## Registering the Shell Extension

After writing this shell extension, we must register it. With the usual Run > Register ActiveX Server command, we can register the server in the system, but we still have to provide some extra information to register it as a shell extension, in this case for any type of file. There are several approaches: you can edit the Registry manually, you can write a REG file, or you can add registration information right into the COM server library, which is my preferred approach. In a Delphi COM server, the default registration takes place in the TComObjectFactory class, when the UpdateRegistry method is executed. We can modify the default registration by inheriting a class from the standard class factory class and overriding this method:

```
type
  TToDoMenuFactory = class (TComObjectFactory)
  public
    procedure UpdateRegistry (Register: Boolean); override;
  end;
```

In this method, we should either add the entry in the Registry or delete it, depending on the value of the Boolean parameter:

```
procedure TToDoMenuFactory.UpdateRegistry(Register: Boolean);
var
  Reg: TRegistry;
begin
  inherited UpdateRegistry (Register);

  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_CLASSES_ROOT;
  try
    if Register then
      if Reg.OpenKey('*\ShellEx\ContextMenuHandlers\ToDo', True) then
        Reg.WriteString('', GUIDToString(Class_ToDoMenuMenu))
      else
        if Reg.OpenKey('*\ShellEx\ContextMenuHandlers\ToDo', False) then
          Reg.DeleteKey ('*\ShellEx\ContextMenuHandlers\ToDo');
    finally
      Reg.CloseKey;
      Reg.Free;
    end;
  end;
```

### WARNING

I've checked this code under Windows 2000, but I'm not completely sure it works also on Windows 98/Me, as the shell portion of the registry has been subject to subtle changes among different versions of Windows.

In the initialization section of the COM object unit, we also need to create a new global object of this class instead of the base class factory class:

**initialization**

```
TToDoMenuFactory.Create (ComServer, TToDoMenu, Class_ToDoMenuMenu,  
    'ToDoMenu', 'ToDoMenu Shell Extension', ciMultiInstance, tmApartment);
```

Now you can simply register the server and set it up as a context-menu handler by using the Delphi Run > Register ActiveX Server menu command, the RegSrv32 application, or most of the tools used to create installation programs.

## What's Next?

In this chapter I have discussed the foundations of Microsoft's COM technology. We've seen how Delphi supports COM and built a few simple servers. In the second part of the chapter, we've spend some time discussing the COM-based Shell API and the development of a shell extension.

The next chapter opens up COM to its higher-level techniques, covering Automation, Documents, and ActiveX Controls. Now that we know the foundations, exploring these COM-technologies will definitely be simpler, although we won't delve into the low-level details of these technologies.

# From Automation to COM+

- OLE Automation
- Creating and using Automation servers
- Using type libraries
- Automating office programs
- The OLE Container component
- Building an ActiveX and an ActiveForm
- Introducing COM+

**A**fter the last chapter, which was devoted to the foundations of Microsoft's COM architecture, it is time to look into some of the actual high-level Windows programming techniques based on COM. We'll start by discussing Automation and the role of type libraries. Also, we'll see how to work properly with Delphi data types in Automation servers and clients.

Later, we'll focus on the use of the Automation support provided by Microsoft Office applications, made simple thanks to the ready-to-use components that embed Office server programs and documents. In the final part of the chapter, we'll explore the use of embedded objects, with the OleContainer component, and the development of OLE controls or ActiveX controls.

I'll also introduce stateless COM (MTS and COM+) technologies and a few other advanced ideas. But let's begin with more foundational material.

## OLE Automation

In the last chapter, we saw that you can use COM to let an executable file and a library share objects, and that this can be used to interact with the Windows shell. Most of the time, however, users want applications that can talk to each other. One of the approaches you can use for this goal is OLE Automation. After presenting a couple of examples that use custom interfaces based on type libraries, I'll cover the development of Word and Excel OLE controllers, showing how to transfer database information to those applications.

### NOTE

The current Microsoft documentation uses the term *Automation* instead of *OLE Automation*, and uses the terms *active document* and *compound document* instead of *OLE Document*. This book uses this new terminology along with the older "OLE" terminology incorporated into many Delphi component names and other identifiers.

In Windows, applications don't live in separate worlds; users often want them to interact. The Clipboard and DDE offer a simple way for applications to interact, as users can copy and paste data between applications. However, more and more programs offer an OLE Automation interface to let other programs drive them. Beyond the obvious advantage of programmed automation compared to manual user operations, these interfaces are completely language-neutral, so you can use Delphi, C++, Visual Basic, or a macro language to drive an OLE Automation server regardless of the programming language used to write it.

OLE Automation is quite straightforward to implement in Delphi, thanks to the extensive work by the compiler and VCL to shield developers from its intricacies. To support OLE Automation, Delphi provides a wizard and a powerful type-library editor, and it supports dual interfaces.

When you use an in-process DLL, the client application can use the server and call its methods directly, because they are in the same address space. When you use OLE Automation, the situation is more complex. The client (called the *controller*) and the server are two separate applications running in different address spaces. For this reason, the system must dispatch the method calls using a complex mechanism called *marshaling* (something I won't cover in detail). What is important to know is that there are two ways a controller can call the methods exposed by a server:

- It can ask for the execution of a method, passing its name in a string, in a way similar to the dynamic call to a DLL. This is what Delphi does when you use a variant to call the OLE Automation server. This technique is very easy to use, but it is rather slow and provides very little compiler type-checking.
- It can import the definition of a Delphi interface for the object on the server and call its methods in a more direct way (simply dispatching a number). This technique, based on interfaces, allows the compiler to check the types of the parameters and produces faster code, but it requires a little more effort from the programmer. Also, you end up binding your controller application to a specific version of the server. A variation of this technique involves the use of dispatch interfaces, based on the definition of the interfaces.

In the following examples, we'll use all these techniques and compare them a little further.

## Introducing Type Libraries

The most important difference between the two approaches is that the second generally requires a *type library*, one of the foundations of OLE and COM. A type library is basically a collection of type information. This collection generally describes all of the elements (the objects, the interfaces, and other type information) made available by a server. The key difference between a type library and other descriptions of these elements (such as some C or Pascal code) is that a type library is language-independent. The type elements are defined by OLE as a subset of the standard elements of programming languages, and any development tool can use them. Why do we need this information?

As mentioned before, an OLE Automation controller can use variants and have no type information about the server it is using. This means that, behind the scenes, every function call has to be dispatched to the server using the `Invoke` method of `IDispatch`, passing the function name as a string parameter and hoping the name corresponds to an existing function of the server.

Although this sounds difficult, a small code fragment using the old Automation interface of Microsoft Word, registered as `Word.Basic`, illustrates how simple it is for a programmer:

```
var
  VarW: Variant;
begin
  VarW := CreateOleObject ('Word.Basic');
  VarW.FileNew;
  VarW.Insert ('Mastering Delphi by Marco Cantù');
```

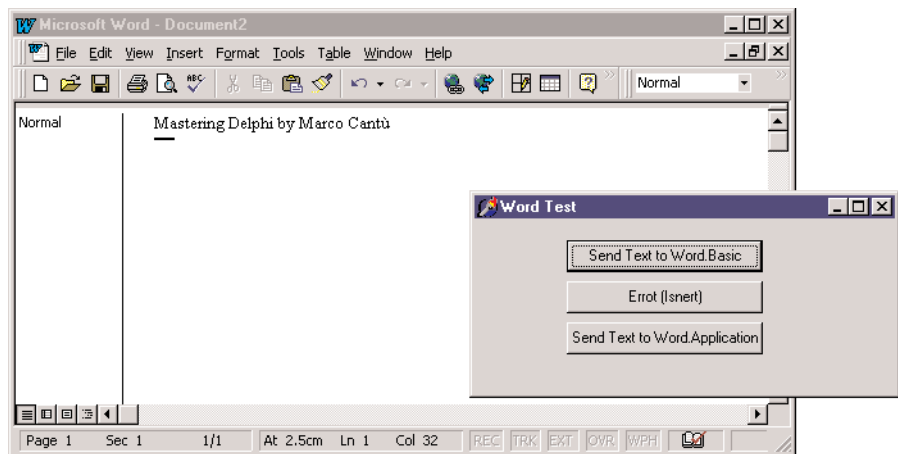
**NOTE**

As we'll see later, recent versions of Word still register the `Word.Basic` interface, which corresponds to the internal WordBasic macro language, but it also registers the new interface `Word.Application`, which corresponds to the VBA macro language. We'll also see that Delphi provides some components that simplify the connection with Microsoft Office applications.

These three lines of code start Word (unless it was already running), create a new document, and add a few words to it. You can see the effect of this code in Figure 20.1. The code uses a variant, which is a *type-variant* data type. A variant can assume as its value different data types, including a COM object supporting the `IDispatch` interface. Variants are type-checked at run time; this is why the compiler can compile the code even if it doesn't know about the methods of the OLE Automation server.

**FIGURE 20.1:**

This Word document is being created and composed by a Delphi application, WordTest.



Unfortunately, the Delphi compiler has no way to check whether the methods exist. Doing all the type checks at run time is risky, because if you make even a minor spelling error in a function name, you get no warning whatsoever of your error until you run the program and reach that line of code. For example, if you type `VarW.Insert`, the compiler will not complain about the misspelling at all, but at run time, you'll get an error. Because it doesn't recognize the name, Word assumes the method does not exist.

Although the OLE IDispatch interface supports the approach we've just seen, it is also possible—and safer—for a server to export the description of its interfaces and objects using a type library. This type library can then be converted by a specific tool (such as Delphi) into definitions written in the language you want to use to write your client or controller program (such as Object Pascal). This makes it possible for a compiler to check whether the code is correct.

Once the compiler has done its checks, it can use either of two different techniques to send the request to the server. It can use a plain VTable (that is, an entry in an interface type declaration), or it can use a `dispinterface` (dispatch interface). We used an interface type declaration in the last chapter, so it should be familiar. A `dispinterface` is basically a way to map each entry in an interface to a number. Calls to the server can then be dispatched by number. We can consider this an intermediate technique, in between dispatching by function name and using a direct call in the VTable.

**NOTE**

The term *dispinterface* is actually a keyword. A `dispinterface` is automatically generated by the type-library editor for every interface. Along with `dispinterface`, Delphi uses other related keywords: `dispid` indicates the number to associate with each element; `readonly` and `writeln` are optional specifiers for properties.

The term used to describe this ability to connect to a server in two different ways, using a more dynamic or a more static approach, is *dual interfaces*. This means that in writing an OLE controller, you can choose to access the methods of a server in two ways: you can use late binding and the mechanism provided by the `dispinterface`, or you can use early binding and the mechanism based on the VTables, the interface types.

It is important to keep in mind that (along with other considerations) different techniques result in faster or slower execution. Looking up a function by name (and doing the type checking at run time) is the slowest approach, using a `dispinterface` is much faster, and using the direct VTable call is the fastest approach. We'll do this kind of test in the `TlibCli` example, later in this chapter.

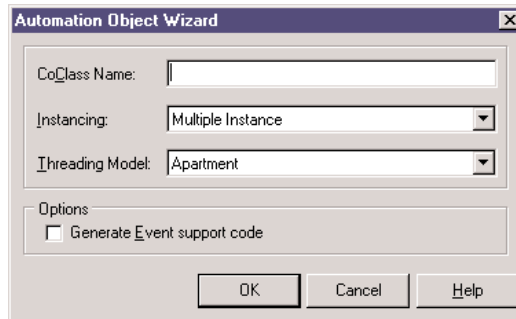
## Writing an OLE Automation Server

We'll start by writing an OLE Automation server. To create an OLE Automation object, you can use Delphi's Automation Object Wizard. Start with a new application, open the Object Repository by selecting File > New, move to the ActiveX page, and choose Automation Object. In the resulting Automation Object Wizard (shown in Figure 20.2), enter the name of the class (without the initial *T*, because this will be added automatically for you) and click OK. Delphi will now open the type-library editor.



**FIGURE 20.2:**

Delphi's Automation  
Object Wizard



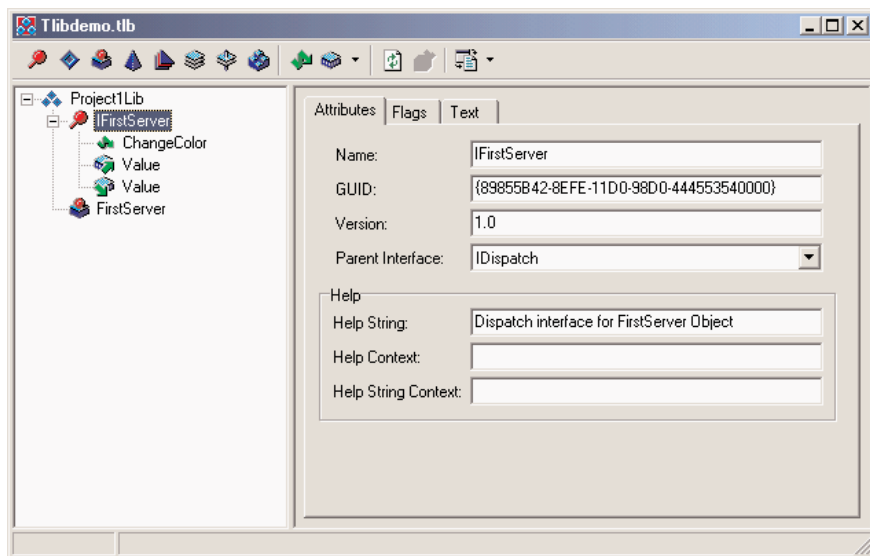
As you can see in Figure 20.2, Delphi can generate OLE Automation servers that also export events. Select the corresponding check box of the Wizard, and Delphi will add the proper entries in the type library and in the source code it generates.

## The Type-Library Editor

The type-library editor is the tool you can use to define a type library in Delphi. Figure 20.3 shows its window after I've added some elements to it. The type-library editor allows you to add methods and properties to the OLE Automation server object we've just created. Once this is done, it can generate both the type library (TLB) file and the corresponding Object Pascal source code.

**FIGURE 20.3:**

The type-library editor,  
showing the details of  
an interface



To build a first example, we can add to the server a property and a method. In the editor, we actually add these two elements to the interface, which should be called `IFirstServer`. Select it, and then click the `Method` button of the toolbar. (The names of these buttons can be displayed by using the shortcut menu of the toolbar.) Now you have to give it a name, such as `ChangeColor`. You can type the name either in the `Tree View` control on the left side of the window or in the `Name` edit box on the right side. Delphi automatically defines the new method as a function in the `Invoke Kind` box and (as you'll see on the `Parameters` page) assigns it an `HRESULT` return value and no parameters. This corresponds to the Pascal definition:

```
procedure ChangeColor; safecall;
```

There are two reasons for this difference in the type of method. The first is that in the IDL language used by COM, all methods are indicated as functions (following the C language style); the second is that Delphi handles the `HRESULT` error codes automatically in every method that uses the `safecall` calling convention.

#### NOTE

The methods contained in OLE Automation interfaces in Delphi generally use the `safecall` calling convention. This wraps a `try/except` block around each method and provides a default return value indicating error or success.

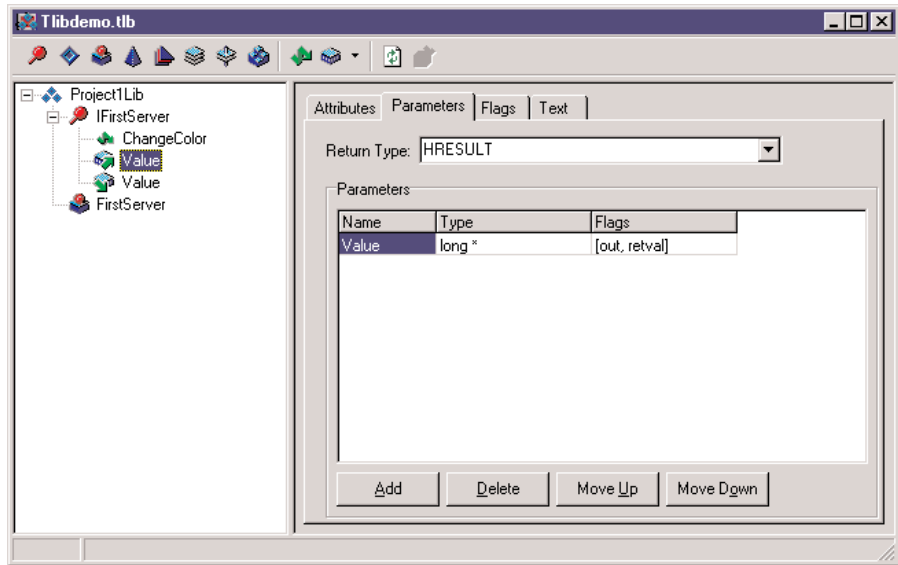
Now we can add a property to the interface by clicking the `Property` button of the type-library editor's toolbar. Again, we can type a name for it, such as `Value`, and select a data type in the `Type` combo box. Besides selecting one of the many types already listed, you can also enter other types directly, particularly interfaces of other objects. Keep in mind, however, that OLE Automation supports only a subset of Delphi types. In this example, we can select the `long` type, which corresponds to Delphi's `Integer` type.

If you look again in the `Parameters` page for this example (see Figure 20.4), you can see that both the `Set` and `Get` (actually called `Put` and `Get` in the COM jargon) methods have the `HRESULT` return value. You can also see that while the `Put` method uses the property's data type as its parameter (as with Delphi properties), the `Get` method uses a pointer to the type as its out parameter. This definition corresponds to the following elements of the Pascal interface:

```
function Get_Value: Integer; safecall;  
procedure Set_Value(Value: Integer); safecall;  
property Value: Integer read Get_Value write Set_Value;
```

**FIGURE 20.4:**

The Parameters page of the type-library editor



Clicking the Refresh button on the type-library editor toolbar generates the Pascal version of the interface. We'll examine it shortly, but first I want you to focus on the Text page of the editor, which includes the definition we've just created, written in the IDL language:

```
interface IFirstServer: IDispatch
{
    [id(0x00000001)]
    HRESULT _stdcall ChangeColor( void );
    [proppget, id(0x00000002)]
    HRESULT _stdcall Value([out, retval] long * Value );
    [propput, id(0x00000002)]
    HRESULT _stdcall Value([in] long Value );
};
```

Fortunately, Delphi's type-library editor saves you from writing similar code by hand, and the Delphi environment options (in the Type Library page) include a radio button to select Pascal or IDL in the text displayed by the type-library editor.

## The Code of the Server

Now we can close the type-library editor and save the changes. This operation adds three items to the project: the type library file, a corresponding Pascal definition, and the declaration of the server object. The type library is connected to the project using a resource-inclusion statement, added to the source code of the project file:

```
{$R *.TLB}
```

You can always reopen the type-library editor by using the View > Type Library command or by selecting the proper TLB file in the normal File Open dialog box of Delphi.

As mentioned earlier, the type library is also converted into an interface definition and added to a new Pascal unit. This unit is quite long, so I've listed in the book only its key elements. The most important part is the new interface declaration:

```
type
  IFirstServer = interface(IDispatch)
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; safecall;
    function Get_Value: Integer; safecall;
    procedure Set_Value(Value: Integer); safecall;
    property Value: Integer read Get_Value write Set_Value;
  end;
```

Then comes the dispinterface, which associates a number with each element of the IFirstServer interface:

```
type
  IFirstServerDisp = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
  end;
```

The last portion of the file includes the so-called CoClass (also shown in the type-library editor), a class used to create an object on the server (and for this reason used on the client side of the application, not on the server side):

```
type
  CoFirstServer = class
    class function Create: IFirstServer;
    class function CreateRemote(const MachineName: string): IFirstServer;
  end;
```

All the declarations of this file (I've skipped some others) can be considered an internal, hidden implementation support. You don't need to understand them fully in order to write most OLE Automation applications.

Finally, Delphi generates a file with the declaration of the actual object. This unit is added to the application and is the one we'll work on to finish the program. This unit declares the class of the server object, which must implement the interface we've just defined:

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
  end;
```

Delphi already provides us with the skeleton code of the methods, so you only need to fill the lines in between. This is the final code of the server object methods of the TLibDemo example from the companion CD:

```

function TFirstServer.Get_Value: Integer;
begin
    Result := ServerForm.Value;
end;

procedure TFirstServer.ChangeColor;
begin
    ServerForm.ChangeColor;
end;

procedure TFirstServer.Set_Value(Value: Integer);
begin
    ServerForm.Value := Value;
end;

```

In this case, the three methods refer to a property and two methods I've added to the form. In general, you should not add code related to the user interface inside the class of the server object. It is better to refer to a user interface element, such as a form class, and let it perform the actions.

I've added a property to the form because I want to change the Value property and have a side effect (displaying the value in an edit box). The server object, in this example, exposes some properties and methods of the application. Here is the part of the declaration of the TServerForm class I've edited manually:

```

type
    TServerForm = class(TForm)
    ...
    private
        CurrentValue: Integer;
    protected
        procedure SetValue (NewValue: Integer);
    public
        property Value: Integer read CurrentValue write SetValue;
        procedure ChangeColor;
    end;

```

The implementation of these methods is quite straightforward, and you can easily guess what their code looks like. What's important is the SetValue method, which might produce a side effect:

```

procedure TServerForm.SetValue (NewValue: Integer);
begin
    if NewValue <> CurrentValue then

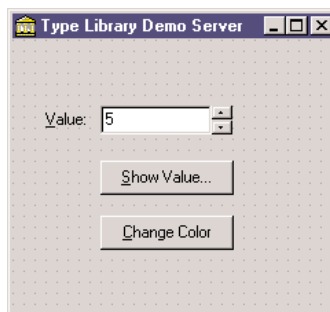
```

```
begin
    CurrentValue := NewValue;
    UpDown1.Position := CurrentValue;
end;
end;
```

The form of this example has an edit box with an associated UpDown component as well as a couple of buttons to show the current value and change the color. You can see this form at design time in Figure 20.5.

**FIGURE 20.5:**

The form of the TLibDemo example at design time



## Registering the Automation Server

The unit containing the server object has one more statement, added by Delphi to the initialization section:

```
initialization
    TAutoObjectFactory.Create(ComServer, TFirstServer, Class_FirstServer,
        ciMultiInstance);
end.
```

### NOTE

In this case, I've selected multiple instancing. For the various instancing styles possible in COM, see the sidebar "COM Instancing and Threading Models" in Chapter 19, "COM Programming."

This is not very different from the creation of class factories we saw in the examples of the last chapter. Combined with the call to the `Initialize` method of the `Application` object, which Delphi adds by default to the project source code of any program, the initialization code above makes the registration of this server straightforward.

You can add the server information to the Windows Registry by running this application on the target machine (the computer where you want to install the OLE Automation server), passing to it the `/regserver` parameter on the command line. You can do this by selecting `Start > Run`, by using the Explorer or File Manager, or by running the program within

Delphi after you've entered a command-line parameter (using the Run > Parameters command). Another command-line parameter, /unregserver, is used to remove this server from the Registry.

## Writing a Client for Our Server

Now that we have built a server, we can prepare a client program to test it. This client can connect to the server either by using variants or by using the new type library. This second approach can be implemented manually or by using the techniques introduced in Delphi 5 for wrapping components around Automation servers. We'll actually try out all of these approaches.

Create a new application—I've called it TLibCli—and then open the type library file of the server, after (optionally) copying it to the project's directory. Save the type library file, using Delphi's File > Save menu command, and a new version of the interface declarations will be generated for you. Of course, in this case you could have grabbed the Pascal declarations from the server source code, but I'm trying to follow a more general approach, which also applies when you haven't written the server yet. In fact, you can usually extract the type library directly from the executable file of the server or from a DLL shipped with the program.

**WARNING** Do not add the type library to the client application, though, because we are writing the OLE Automation controller, not a server. The Delphi project of a controller should not include the type library of the server it connects to.

You can refer to the Pascal file generated by the type-library editor in the code of the main form:

```
uses
  TlibdemoLib_TLB;
```

I've already mentioned that one of the elements of this unit generated by the type library is the *creation* class, or CoClass, a special class with two class functions you can use to create a server object locally or remotely (using DCOM). I've already shown you the interface of this class, but here is the implementation:

```
class function CoFirstServer.Create: IFirstServer;
begin
  Result := CreateComObject(Class_FirstServer) as IFirstServer;
end;

class function CoFirstServer.CreateRemote(
  const MachineName: string): IFirstServer;
begin
  Result := CreateRemoteComObject(MachineName, Class_FirstServer)
    as IFirstServer;
end;
```

You can use the first of these two functions, `Create`, to create a server object (and possibly start the server application) on the same computer. You can use the second function, `CreateRemote`, to create the server on a different computer, as long as your version of the operating system supports DCOM.

The two functions are a shortcut of the `CreateComObject` call, which allows you to create an instance of a COM object if you know its GUID. As an alternative, you can also use the `CreateOLEObject` function, which requires as a parameter the registered name of the server. There is another difference between these two creation functions: `CreateComObject` returns an object of the `IUnknown` type, whereas `CreateOLEObject` returns an object of the `IDispatch` type.

In my example, I'm going to use the `CoFirstServer.Create` shorthand. When you create the server object, you get as return value an `IFirstServer` interface. You can use it directly or store it in a variant variable. Here is an example of the first approach:

```
var
  MyServer: Variant;
begin
  MyServer := CoFirstServer.Create;
  MyServer.ChangeColor;
```

This code, based on variants, is not very different from that of the first controller we built in this chapter (the one that used Microsoft Word). Here is the alternative code, which has exactly the same effect:

```
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

## Interfaces, Variants, and Dispatch Interfaces: Testing the Speed Difference

As I mentioned in the section introducing type libraries, one of the differences between these approaches is speed. It is actually quite complex to assess the exact performance of each technique because there are many factors involved. I've added a simple test to the `TLibCli` example on the companion CD, just to give you an idea. Here is the code of the test, a loop that accesses the `Value` of the server. The total value is displayed only to fool the optimizer, which might otherwise remove some of the code. The real output of the program relates to the timing, which is determined by calling the `GetTickCount` API function before and after executing the loop. (Two alternatives are to use Delphi's own time functions, which are slightly less precise,



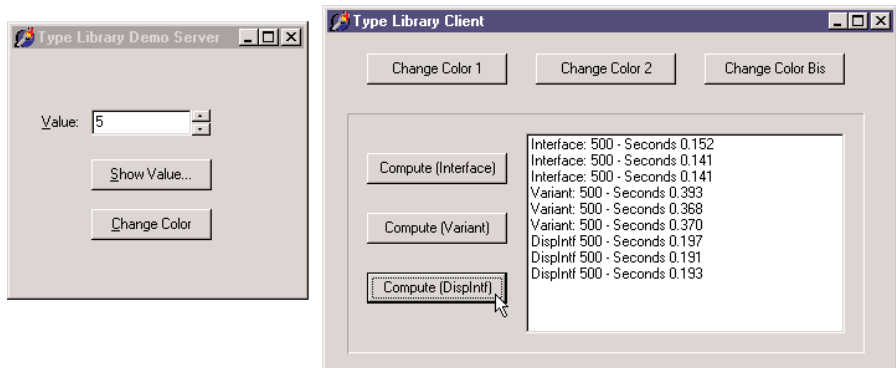
or to use the very precise timing functions of the multimedia support unit, MMSystem.) Here is the code of one of the methods; they are quite similar:

```
procedure TClientForm.BtnIntfClick(Sender: TObject);
var
  I, K: Integer;
  Ticks: Cardinal;
begin
  Screen.Cursor := crHourglass;
  try
    Ticks := GetTickCount;
    K := 0;
    for I := 1 to 100 do
      K := K + IMyServer.Value;
      Ticks := GetTickCount - Ticks;
      ListResult.Items.Add (Format (
        'Interface: %d - Seconds %.3f', [K, Ticks / 1000]));
    finally
      Screen.Cursor := crDefault;
    end;
  end;
```

With this program, you can compare the output obtained by calling this method based on an interface, the corresponding version based on a variant, and even a third version based on a dispatch interface. An example of the output (which is added to a list box so you can do several tests and compare the results) is shown in Figure 20.6. Obviously, the timing depends on the speed of your computer, and you can also alter the results by increasing or decreasing the maximum value of the loop counter.

**FIGURE 20.6:**

The TLibCll OLE Automation controller can access the server in different ways, with different performance results. Notice the server window in the background.



We've already seen how you can use the interface and the variant. What about the dispatch interface? You can declare a variable of the dispatch interface type, in this case:

```
var  
  DMyServer: IFirstServerDisp;
```

Then you can use it to call the methods as usual, after you've assigned an object to it by casting the object returned by the CoClass:

```
DMyServer := CoFirstServer.Create as IFirstServerDisp;
```

Looking at the timing and at the internal code of the example, there is apparently very little difference between the use of the interface and of the dispatch interface, because the two are actually connected. In other words, we can say that dispatch interfaces are a technique in between variants and interfaces, but they deliver almost all of the speed of interfaces.

## The Scope of Automation Objects

Another important element to keep in mind is the *scope* of the automation objects. Variants and interface objects use reference-counting techniques, so if a variable that is related to an interface object is declared locally in a method, at the end of the method the object will be destroyed and the server may terminate (if all the objects created by the server have been destroyed). For example, writing a method with this code produces very little effect:

```
procedure TClientForm.ChangeColor;  
var  
  IMyServer: IFirstServer;  
begin  
  IMyServer := CoFirstServer.Create;  
  IMyServer.ChangeColor;  
end;
```

Unless the server is already active, a copy of the program is created, and the color is changed, but then the server is immediately closed as the interface-typed object goes out of scope. The alternative approach I've used in the TLibCli example is to declare the object as a field of the form and create the COM objects at start-up, as in this procedure:

```
procedure TClientForm.FormCreate(Sender: TObject);  
begin  
  IMyServer := CoFirstServer.Create;  
end;
```

With this code, as the client program starts, the server program is immediately activated. At the program termination, the form field is destroyed and the server closes. A further alternative

is to declare the object in the form, but then create it only when it is used, as in these two code fragments:

```
// MyServerBis: Variant;
if varType (MyServerBis) = varEmpty then
  MyServerBis := CoFirstServer.Create;
MyServerBis.ChangeColor;

// IMyServerBis: IFirstServer;
if not Assigned (IMyServerBis) then
  IMyServerBis := CoFirstServer.Create;
IMyServerBis.ChangeColor;
```

## NOTE

A variant is initialized to the `varEmpty` type when it is created. If you instead assign the value null to the variant, its type becomes `varNull`. Both `varEmpty` and `varNull` represent variants with no value assigned, but they behave differently in expression evaluation. The `varNull` value always propagates through an expression (making it a null expression), while the `varEmpty` value quietly disappears.

## The Server in a Component

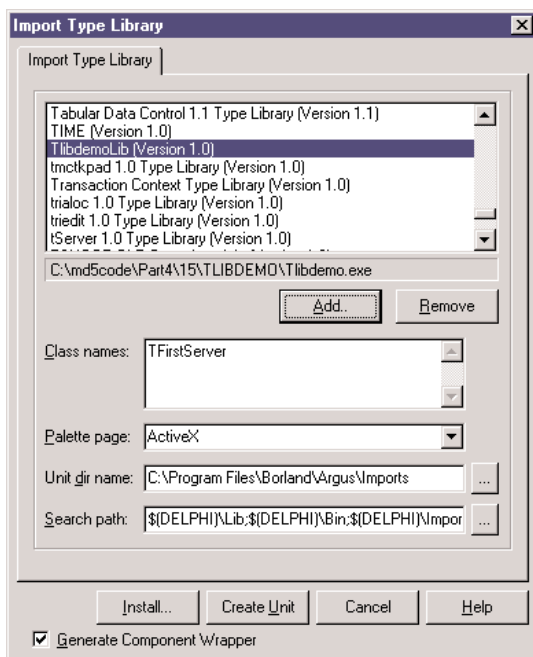
When creating a client program for our server or any other Automation server, we can use a better approach, namely, wrapping a Delphi component around the COM server. Actually, if you look at the final portion of the `TLibdemoLib_TLB` file, you can find the following declaration:

```
// OLE Server Proxy class declaration
TFirstServer = class(TOLEServer)
private
  FIntf: IFirstServer;
  FProps: TFirstServerProperties;
  function GetServerProperties: TFirstServerProperties;
  function GetDefaultInterface: IFirstServer;
protected
  procedure InitServerData; override;
  function Get_Value: Integer;
  procedure Set_Value(Value: Integer);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure Connect; override;
  procedure ConnectTo(svrIntf: IFirstServer);
  procedure Disconnect; override;
  procedure ChangeColor;
  property DefaultInterface: IFirstServer read GetDefaultInterface;
  property Value: Integer read Get_Value write Set_Value;
published
  property Server: TFirstServerProperties read GetServerProperties;
end;
```

This is a new component, derived from `TOLEServer`, that the system registers in the `Register` procedure, which is part of the unit. If you add this unit to a package, the new server component will become available on the Delphi Component Palette. You can also import the type library of the new server (with the `Project > Import Type Library` menu command), add the server to the list (by clicking the `Add` button and selecting the server's executable file), and install it in a new or existing package. The component will be placed in the `Servers` page of the Palette. The `Import Type Library` dialog box indicating these operations is visible in Figure 20.7.

**FIGURE 20.7:**

The `Import Type Library` dialog box can be used to import an Automation server object as a new Delphi component.



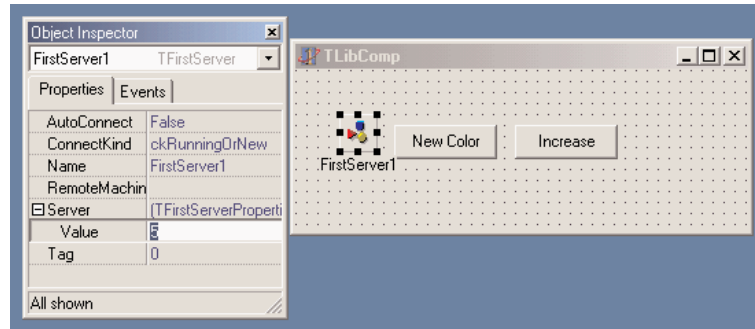
I've created a new package, `PackAuto`, available in the directory of the `TlibDemo` project. In this package, I've added the directive `LIVE_SERVER_AT_DESIGN_TIME` in the `Directories/Conditionals` page of the `Project Options` dialog box of the package. This enables an extra feature that you don't get by default: at design time, the server component will have an extra property that lists as subitems all the properties of the Automation server. You can see an example in Figure 20.8, taken from the `TLibComp` example at design time.

**WARNING**

The `LIVE_SERVER_AT_DESIGN_TIME` directive should be used with care with the most complex Automation servers (including programs such as Word, Excel, PowerPoint, and Visio). In fact, this setting requires the application to be in a particular mode before you can use some properties of their automation interfaces. For example, you'll get exceptions if you touch the Word server before a document has been opened in Word. That's why this feature is not active by default in Delphi—it's problematic at design time for many servers.

**FIGURE 20.8:**

A server component, with the live properties at design time



As you can see in the Object Inspector, the component has few properties. `AutoConnection` indicates when to start up the server component at design time and as soon as the client program starts. As an alternative, the Automation server is started the first time one of its methods is called. Another property, `ConnectKind`, indicates how to establish the connection with the server. It can always start a new instance (`ckNewInstance`), use the running instance (`ckRunningInstance`, which causes an access violation if the server is not already running), or select the current instance or start a new one if none is available (`ckRunningOrNew`). Finally, you can ask for a remote server with `ckRemote` and directly attach a server in the code after a manual connection with `ckAttachToInterface`.

## OLE Data Types

OLE and COM do not support all of the data types available in Delphi. This is particularly important for OLE Automation, because the client and the server are often executed in different address spaces, and the system must move the data from one side to the other. Also keep in mind that OLE interfaces should be accessible by programs written in any language.

COM data types include basic data types such as `Integer`, `SmallInt`, `Byte`, `Single`, `Double`, `WideString`, `Variant`, and `WordBool` (but not `Boolean`). Table 20.1 presents the mapping of some basic data types, available in the type-library editor, to the corresponding Delphi types.

**TABLE 20.1:** OLE and Delphi Data Types

OLE Type	Delphi Type
BSTR	WideString
byte	ShortInt
CURRENCY	Currency
DATE	TDateTime
DECIMAL	TDecimal

*Continued on next page*

**TABLE 20.1 continued:** OLE and Delphi Data Types

OLE Type	Delphi Type
double	Double
float	Single
GUID	GUID
int	SYSINT
long	Integer
LPSTR	PChar
LPWSTR	PWideChar
short	SmallInt
unsigned char	Byte
unsigned int	SYSUINT
unsigned long	UINT
unsigned short	Word
VARIANT	OLEVariant

Notice that `SYSINT` is currently defined as an `Integer`, so don't worry about the apparently strange type definition. Besides the basic data types, you can also use OLE types for complex elements such as fonts, string lists, and bitmaps, using the `IFontDisp`, `IStrings`, and `IPictureDisp` interfaces. The following sections describe the details of a server that provides a list of strings and a font to a client.

## Exposing Strings Lists and Fonts

The `ListServ` example is a practical demonstration of how you can expose two complex types, such as a list of strings and a font, from an OLE Automation server written in Delphi. I've chosen these two specific types simply because they are both supported by Delphi.

The `IFontDisp` interface is actually provided by Windows and is available in the `ActiveX` unit. The `AxCtrls` Delphi unit extends this support by providing conversion methods like `GetOLEFont` and `SetOLEFont`. The `IStrings` interface is provided by Delphi in the `StdVCL` unit, and the `AxCtrls` unit provides conversion functions for this type (along with a third type I'm not going to use, `TPicture`).

### WARNING

To run this and similar applications, the `StdVCL` library must be installed and registered on the client computer. On your computer, it is registered during Delphi's installation.

The server we are building has a plain form containing a list-box component. It includes an Automation object built around the following interface:

```

type
  IListServer = interface (IDispatch)
    ['{323C4A84-E400-11D1-B9F1-004845400FAA}']
    function Get_Items: IStrings; safecall;
    procedure Set_Items(const Value: IStrings); safecall;
    function Get_Font: IFontDisp; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    property Items: IStrings read Get_Items write Set_Items;
    property Font: IFontDisp read Get_Font write Set_Font;
end;

```

The server object has the same four methods listed in its interface as well as some private data storing the status, the initialization function, and the destructor:

```

type
  TListServer = class (TAutoObject, IListServer)
  private
    fItems: TStrings;
    fFont: TFont;
  protected
    function Get_Font: IFontDisp; safecall;
    function Get_Items: IStrings; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Items(const Value: IStrings); safecall;
  public
    destructor Destroy; override;
    procedure Initialize; override;
end;

```

The code of the methods is limited to few statements. The pseudoconstructor creates the internal objects, and the destructor destroys them. Here is the first of the two:

```

procedure TListServer.Initialize;
begin
  inherited Initialize;
  fItems := TStringList.Create;
  fFont := TFont.Create;
end;

```

The Set and Get methods copy information from the OLE interfaces to the local data and then from this to the form and vice versa. The two methods of the strings, for example, do this by calling the GetOLEStrings and SetOLEStrings Delphi functions.

After we've compiled and registered the server, we can turn our attention to the client application. This embeds the Pascal translation of the type library of the server, as in the previous example, and then implements an object that uses the interface. Instead of creating the

server when the object starts, the client program creates it when it is required. I've described this technique earlier, but the problem is that because there are several buttons a user can click, and we don't want to impose an order, every event should have a handler like this:

```
if not Assigned (ListSrv) then
    ListSrv := CoListServer.Create;
```

This kind of code duplication is quite dangerous, so I've decided to use an alternative approach. I've defined a property corresponding to the interface of the server and defined a read method for it. The property is mapped to some internal data I've defined with a different name to avoid the error of using it directly. Here are the definitions added to the form class:

```
private
    fInternalListSrv: IListServer;
    function GetListSrv: IListServer;
public
    property ListSrv: IListServer read GetListSrv;
```

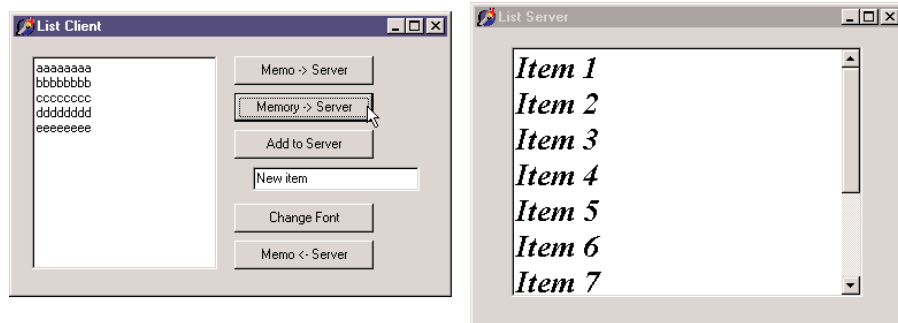
The implementation of the Get method can check whether the object already exists. This code is going to be repeated often, but that should not slow down the application noticeably:

```
function TListCliForm.GetListSrv: IListServer;
begin
    // eventually create the server
    if not Assigned (fInternalListSrv) then
        fInternalListSrv := CoListServer.Create;
    Result := fInternalListSrv;
end;
```

You can see an example of the client application running (along with the server) in Figure 20.9.

**FIGURE 20.9:**

The ListCli and ListSrv applications share complex data, namely fonts and lists of strings.



This is an example of the selection of a font, which is then sent to the server:

```
procedure TListCliForm.btnFontClick(Sender: TObject);
var
    NewFont: IFontDisp;
```



```
begin
  // select a font and apply it
  if FontDialog1.Execute then
    begin
      GetOleFont (FontDialog1.Font, NewFont);
      ListSrv.Font := NewFont;
    end;
  end;
```

There are also several methods related to the strings, which you can see by looking at the source code of the program.

## Using Office Programs

So far, we've built both the client and the server side of the OLE Automation connection. If your aim is just to let two applications you've built cooperate, this is certainly a useful technique, although it is not the only one. We've seen some alternative data-sharing approaches in the last two chapters (using memory-mapped files and the `wm_CopyData` message). The real value of OLE Automation is that it is a standard, so you can use it to integrate your Delphi programs with other applications your users own. A typical example is the integration of a program with office applications, such as Microsoft Word and Microsoft Excel, or even with stand-alone applications, such as AutoCAD.

Integration with these applications provides a two-fold advantage:

- You can let your users work in an environment they know—for example, generating reports and memos from database data in a format they can easily manipulate.
- You can avoid implementing complex functionality from scratch, such as writing your own word-processing code inside a program. Instead of just reusing components, you can reuse complex applications.

There are also some drawbacks with this approach, which are certainly worth mentioning:

- The user must own the application you plan to integrate with, and they may also need a recent version of it to support all the features you are using in your program.
- You have to learn a new programming language and programming structure, often with limited documentation at hand. It is true, of course, that you are still using Pascal, but the code you write depends on the OLE data types, the types introduced by the server, and in particular, a collection of interrelated classes that are often difficult to understand.
- You might end up with a program that works only with a specific version of the server application, particularly if you try to optimize the calls by using interfaces instead of variants. In particular, Microsoft does not attempt to maintain script compatibility between major releases of Word or other Office applications.

We've already seen a small source code excerpt from the WordTest example, but now I want to complete the coverage of this limited but interesting test program by providing a few extra features.

## Sending Data to Microsoft Word

Delphi simplifies the use of Microsoft Office applications by preinstalling some ready-to-use components that wrap the Automation interface of these servers. These components, available in the Servers page of the Palette, have been installed using the same technique I demonstrated in the last section.

### NOTE

What I want to underline here is that the real plus of Delphi lies in this technique of creating components to wrap existing Automation servers, rather than in the availability of some predefined server components.

Technically, it is possible to use variants to interact with Automation servers, as we've seen in the section "Introducing Type Libraries." Using interfaces and the type libraries is certainly better, because the compiler helps you catch errors in the source code and produces faster code. Thanks to the new server component, this process is also quite straightforward.

I've written a program, called DBOffice, which uses predefined server components to send a table to Word and to Excel. In both cases, you can use the application object, the document/worksheet object, or a combination of the two. There are other specialized components, for tasks such as handling Excel charts, but this example will suffice to introduce use of the built-in Office components.

### NOTE

The DBOffice program was tested with Office 97. I'm currently using StarOffice more often than the Microsoft suite, so I never feel compelled to give Microsoft more money by upgrading to their newer offerings.

In case of Microsoft Word, I use only a document object with default settings. The code used to send the table to Word starts by adding some text to a document:

```
procedure TFormOff.BtnWordClick(Sender: TObject);  
begin  
    WordDocument1.Activate;  
    // insert title  
    WordDocument1.Range.Text := 'American Capitals from ' + Table1.TableName;  
    WordDocument1.Range.Font.Size := 14;
```

This code follows the typical `while` loop, which scans the database table and has the following code inside:

```
while not Table1.EOF do  
begin  
    // send the two fields
```

```

WordDocument1.Range.InsertParagraphAfter;
WordDocument1.Paragraphs.Last.Range.Text :=
  Table1.FieldByName ('Name').AsString + #9 +
  Table1.FieldByName ('Capital').AsString;
Table1.Next;
end;

```

The final part of the code gets a little more complex. It works on a selection and on a row of the table, respectively stored in two variables of the Range and Row types defined by Word and available in the Word97 unit (the program will have to be updated if you choose the Office 2000 version of the server component while installing Delphi).

```

procedure TFormOff.BtnWordClick(Sender: TObject);
var
  RangeW: Word97.Range;
  v1: Variant;
  ov1: OleVariant;
  Row1: Word97.Row;
begin
  // code above...
  RangeW := WordDocument1.Content;
  v1 := RangeW;
  v1.ConvertToTable (#9, 19, 2);
  Row1 := WordDocument1.Tables.Item(1).Rows.Get_First;
  Row1.Range.Bold := 1;
  Row1.Range.Font.Size := 30;
  Row1.Range.InsertParagraphAfter;
  ov1 := ' ';
  Row1.ConvertToText (ov1);
end;

```

As you can see in the last statement above, in order to pass a parameter, you must first save it in an OleVariant variable, because many parameters are passed by reference, so you cannot pass a constant value. This implies that if there are many parameters, you must still define some, even if you are fine with the default values. An often-useful alternative is to use a temporarily variant variable and apply the method to it, because variants don't require strict type-checking on the parameters. This technique is used in the code above to call the ConvertToTable method, which has more than 10 parameters.

## Building an Excel Table

In the case of Excel, I've used a slightly different approach and worked with the application object. The code creates a new Excel spreadsheet, fills it with a database table, and formats the result. It uses an Excel internal object, Range, which is not to be confused with a similar

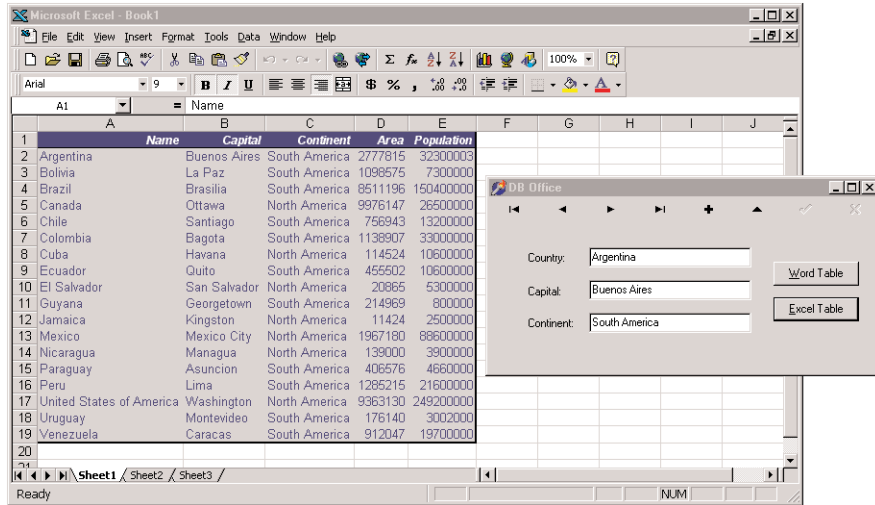
type available in Word (the reason this type is prefixed with the name of the unit defining the Excel type library). Here is the complete code:

```
procedure TFormOff.BtnExcelClick(Sender: TObject);  
var  
    RangeE: Excel97.Range;  
    I, Row: Integer;  
    Bookmark: TBookmarkStr;  
begin  
    // create and show  
    ExcelApplication1.Visible [0] := True;  
    ExcelApplication1.Workbooks.Add (NULL, 0);  
    // fill is the first row with field titles  
    RangeE := ExcelApplication1.ActiveCell;  
    for I := 0 to Table1.Fields.Count - 1 do  
    begin  
        RangeE.Value := Table1.Fields [I].DisplayLabel;  
        RangeE := RangeE.Next;  
    end;  
    // add field data in following rows  
    Table1.DisableControls;  
    try  
        Bookmark := Table1.Bookmark;  
        try  
            Table1.First;  
            Row := 2;  
            while not Table1.EOF do  
            begin  
                RangeE := ExcelApplication1.Range ['A' + IntToStr (Row),  
                'A' + IntToStr (Row)];  
                for I := 0 to Table1.Fields.Count - 1 do  
                begin  
                    RangeE.Value := Table1.Fields [I].AsString;  
                    RangeE := RangeE.Next;  
                end;  
                Table1.Next;  
                Inc (Row);  
            end;  
            finally  
                Table1.Bookmark := Bookmark;  
            end;  
        finally  
            Table1.EnableControls;  
        end;  
    // format the section  
    RangeE := ExcelApplication1.Range ['A1', 'E' + IntToStr (Row - 1)];  
    RangeE.AutoFormat (3, NULL, NULL, NULL, NULL, NULL, NULL);  
end;
```

You can see the effect of this code in Figure 20.10. Notice that in the code I don't handle any events of the Office applications, but many are available. Handling these events was quite complex in the past, but they now become as simple to handle as events of native Delphi components. The presence of these events is a reason to have specific objects for documents and other specific elements: you might want to know when the user closes a document, and that therefore this is an event of the document object, not of the application object.

**FIGURE 20.10:**

The Excel spreadsheet generated by the DBOffice application



## NOTE

When using the Office server components, one of the key problems is the lack of adequate documentation. Although Microsoft distributes some of it with the high-end version of the Office suite, this is certainly not Delphi friendly. A totally alternative approach to solve the problem is to use OfficePartner, a set of components from TurboPower Software ([www.turbopower.com](http://www.turbopower.com)). These components map the Office servers, like those available in Delphi, but they also provide extensive property editors that allow you to work visually with the internal structure of these servers. With these property editors, you can create documents, paragraphs, tables, and all the other internal objects even at design time! From my experience, this can really save a lot of time.

## Using Compound Documents

Compound documents, or active documents, are Microsoft's names for the technology that allows in-place editing of a document within another one (for example, a picture in a Word document). This is the technology that originated the term OLE, but although it is still in use, its role is definitely more limited than Microsoft envisioned when it was introduced in

the early 1990s. Compound documents actually have two different capabilities, *object linking* and *embedding* (hence the term OLE):

- Embedding an object in a compound document corresponds to a smart version of the copy and paste operations you make with the Clipboard. The key difference is that when you copy an OLE object from a server application and paste it into a container application, you copy both the data and some information about the server (its GUID). This allows you to activate the server application from within the container to edit the data.
- Linking an object to a compound document instead copies only a reference to the data and the information about the server. You generally activate object linking by using the Clipboard and making a Paste Link operation. When editing the data in the container application, you'll actually modify the original data, which is stored in a separate file.

Because the server program refers to an entire file (only part of which might be linked in the client document), the server will be activated in a stand-alone window, and it will act upon the entire original file, not just the data you've copied. When you have an embedded object, instead, the container might support visual (or *in-place*) editing, which means that you can modify the object in context, inside the container's main window. The server and container application windows, their menus, and their toolbars are merged automatically, allowing the user to work within a single window on several different object types—and therefore with several different OLE servers—without leaving the window of the container application.

Another key difference between embedding and linking is that the data of an embedded object is stored and managed by the container application. The container saves the embedded object in its own files. By contrast, a linked object physically resides in a separate file, which is handled by the server exclusively, even if the link refers only to a small portion of the file.

In both cases, the container application doesn't have to know how to handle the object and its data—not even how to display it—without the help of the server. Accordingly, the server application has a lot of work to do, even when you are not editing the data. Container applications often make a copy of the image of an OLE object and use the bitmap to represent the data, which speeds up some operations with the object itself. The drawback of this approach is that many commercial OLE applications end up with bloated files (because two copies of the same data are saved). If you consider this problem along with the relative slowness of OLE and the amount of work necessary to develop OLE servers, you can understand why the use of this powerful approach is still somewhat limited, compared with what Microsoft envisioned a few years ago.

Compound document containers can support OLE in varying degrees. You can place an object in a container by inserting a new object, by pasting or *paste-linking* one from the Clipboard, by dragging one from another application, and so on.

Once the object is placed inside the container, you can then perform operations on it, using the server's available *verbs*, or actions. Usually the *edit verb* is the default action—the action performed when you double-click on the object. For other objects, such as video or sound clips, *play* is defined as the default action. You can typically see the list of actions supported by the current contained object by right-clicking it. The same information is available in many programs via the Edit > Object menu item, which has a submenu that lists the available verbs for the current object.

**NOTE**

Delphi provides no *visual* support for building compound document servers. You can always write a server implementing the proper interfaces. Compound document container support, instead, is easily available through the OleContainer component.

## The OLE Container Component

To create an OLE container application in Delphi, place an OleContainer component in a form. Then select the component and right-click to activate its shortcut menu, which will have an Insert Object command. When you select this command, Delphi displays the standard OLE Insert Object dialog box. This dialog box allows you to choose from one of the server applications registered on the computer.

Once the OLE object is inserted in the container, the shortcut menu of the control container component will have several more custom menu items. The new menu items include commands to change the properties of the OLE object, insert another one, copy the existing object, or remove it. The list also includes the verbs, or actions, of the object (such as Edit, Open, or Play). Once you have inserted an OLE object in the container, the corresponding server will launch to let you edit the new object. As soon as you close the server application, Delphi updates the object in the container and displays it at design time in the form of the Delphi application you are developing.

If you look at the textual description of a form containing a component with an object inside, you'll notice a Data property, which contains the actual data of the OLE object. Although the client program stores the data of the object, it doesn't know how to handle and show that without the help of the proper server (which must be available on the computer where you run the program). This means that the OLE object is *embedded*.

To fully support compound documents, a program should provide a menu and a toolbar or panel. These extra components are important because in-place editing implies a merging of the user interface of the client and that of the server program. When the OLE object is activated in place, some of the pull-down menus of the server application's menu bar are added to the menu bar of the container application.

OLE menu merging is handled almost automatically by Delphi. You only need to set the proper indexes for the menu items of the container, using the `GroupIndex` property. Any menu item with an odd index number is replaced by the corresponding element of the active OLE object. More specifically, the File (0) and Window (4) pull-down menus belong to the container application. The Edit (1), View (3), and Help (5) pull-down menus (or the groups of pull-down menus with those indexes) are taken by the OLE server. A sixth group, named Object and indicated with the index 2, can be used by the container to display another pull-down menu between the Edit and View groups, even when the OLE object is active. The `OleCont` demo program I've written to demonstrate these features allows a user to create a new object by calling the `InsertObjectDialog` method of the `TOleContainer` class.

The `InsertObjectDialog` method shows a system dialog box, but it doesn't automatically activate the OLE object:

```
procedure TForm1.New1Click(Sender: TObject);  
begin  
    if OleContainer1.InsertObjectDialog then  
        OleContainer1.DoVerb (OleContainer1.PrimaryVerb);  
end;
```

Once a new object has been created, you can execute its primary verb using the `DoVerb` method. The program also displays a small toolbar with some bitmap buttons. I placed some `TWinControl` components in the form to let the user select them and thus disable the `OleContainer`. To keep this toolbar/panel visible while in-place editing is occurring, you should set its `Locked` property to `True`. This forces the panel to remain present in the application and not be replaced by a toolbar of the server.

To show what happens when you don't use this approach, I've added to the program a second panel, with some more buttons. Because I haven't set its `Locked` property, this new toolbar will be replaced with that of the active OLE server. When in-place editing launches a server application that displays a toolbar, that server's toolbar replaces the container's toolbar, as you can see in the lower part of Figure 20.11.

**TIP**

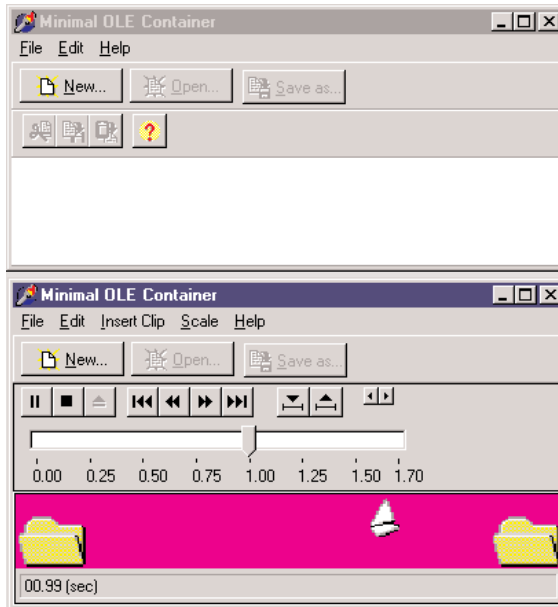
To make all the automatic resizing operations work smoothly, you should place the OLE container component in a panel component and align both of them to the client area of the form.

Another way to create an OLE object is to use the `PasteSpecialDialog` method, called in the `PasteSpecial1Click` event handler of the example. Another standard OLE dialog box, wrapped in a Delphi function, is the one showing the properties of the object, which is activated with the Object Properties item in the Edit pull-down menu by calling the `ObjectPropertiesDialog` method of the `OleContainer` component.



**FIGURE 20.11:**

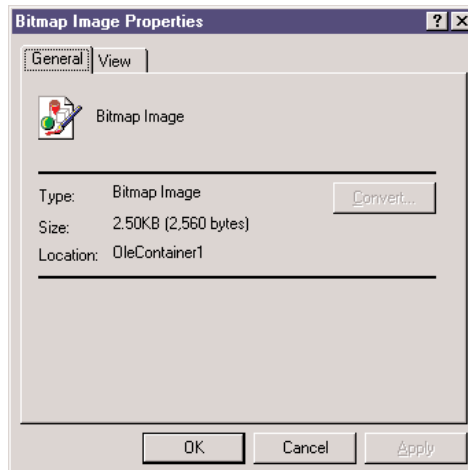
The second toolbar of the OleCont example (top) is replaced by the toolbar of the server (bottom).



You can see an example of the resulting standard OLE dialog box in Figure 20.12. Obviously, this dialog box changes depending on the nature of the active OLE object in the container. The last feature of the OleCont program is support for files; this is actually one of the simplest additions we can make, because the OLE container component already provides file support.

**FIGURE 20.12:**

The standard OLE Object Properties dialog box, available in the OleCont example



## Using the Internal Object

In the preceding program, the user determined the type of the internal object created by the program. In this case, there is little you can do to interact with the internal objects. Suppose, instead, that you want to embed a Word document in a Delphi application and then modify it by code. You can do this by using OLE Automation with the embedded object, as demonstrated by the WordCont example (the name stands for *Word container*).

### WARNING

Since the WordCont example includes an object of a specific type, a Microsoft Word document, it won't run if you don't have that server application installed. Having a different version of the server might also create problems if the Automation methods used by the client program are not available in that version of the server.

In the form of this example, I've added an OleContainer component, set its AutoActivate property to aaManual (so that the only possible interaction is with our code), and added a toolbar with a couple of buttons. The code for the two buttons is quite straightforward, once you know that the embedded object corresponds to a Word document:

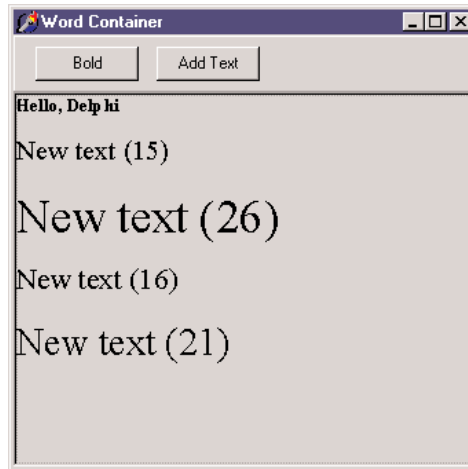
```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Document: Variant;  
begin  
    // activates if not running  
    if not (OleContainer1.State = osRunning) then  
        OleContainer1.Run;  
    // get the document  
    Document := OleContainer1.OleObject;  
    // first paragraph to bold  
    Document.Paragraphs.Item(1).Range.Bold := 1;  
end;  
  
procedure TForm1.Button3Click(Sender: TObject);  
var  
    Document, Paragraph: Variant;  
begin  
    // activate if not running  
    if not (OleContainer1.State = osRunning) then  
        OleContainer1.Run;  
    // get the document  
    Document := OleContainer1.OleObject;  
    // add paragraphs, getting the last one  
    Document.Paragraphs.Add;  
    Paragraph := Document.Paragraphs.Add;  
    // add text to the paragraph, using random font size
```

```
Paragraph.Range.Font.Size := 10 + Random (20);  
Paragraph.Range.Text := 'New text (' +  
  IntToStr (Paragraph.Range.Font.Size) + ')'#13;  
end;
```

You can see the effect of this code in Figure 20.13. The code is not terribly powerful, but it does show how you can merge the usage of OLE Containers and OLE Automation techniques.

**FIGURE 20.13:**

The WordCont example shows how to use OLE Automation with an embedded object.



## Introducing ActiveX Controls

Microsoft's Visual Basic was the first program development environment to introduce the idea of supplying software components to the mass market. Actually, the concept of reusable software components is older than Visual Basic—it's well rooted in the theories of object-oriented programming (OOP). But OOP languages never delivered the reusability they promised, probably more because of marketing and standardization problems than for any other reason. Although Visual Basic does not fully exploit OOP, it applies the component concept through its standard way of building and distributing new controls that developers can integrate into the environment.

The first technical standard promoted by Visual Basic was *VBX*, a 16-bit specification that was fully available in the 16-bit version of Delphi. In moving to the 32-bit platforms, Microsoft replaced the VBX standard with the more powerful and more open *ActiveX* controls.

**NOTE**

ActiveX controls used to be called OLE controls (or OCX). The name change reflects a new marketing strategy from Microsoft rather than a technical innovation. Technically, ActiveX can be considered a minor extension to the OCX technology. Not surprisingly, then, ActiveX controls are usually saved in files with the `.ocx` extension.

From a general perspective, an ActiveX control is not very different from a Windows, Delphi, or Visual Basic control. A control in any of these languages is always a window, with its associated code defining its behavior. The key difference between various families of controls is in the *interface* of the control—the interaction between the control and the rest of the application. Typical Windows controls use a message-based interface; VBX controls use properties and events; OLE Automation objects use properties and methods; and ActiveX controls use properties, methods, and events. These three elements of properties, methods, and events are also found in Delphi’s own components.

Using OLE jargon, an ActiveX control is a “compound document object which is implemented as an in-process server DLL and supports OLE Automation, visual editing, and inside-out activation.” Perfectly clear, right? Let’s see what this definition actually means. An ActiveX control uses the same approach as OLE server objects, which are the objects you can insert into an OLE Document, as we saw in the last chapter. The difference between a generic OLE server and an ActiveX control is that, whereas ActiveX controls can only be implemented in one way, OLE servers can be implemented in three different ways:

- As stand-alone applications (for example, Microsoft Excel)
- As out-of-process servers—that is, executables files that cannot be run by themselves and can only be invoked by a server (for example, Microsoft Graph and similar applications)
- As in-process servers, such as DLLs loaded into the same memory space as the program using them

ActiveX controls can only be implemented using the last technique, which is also the fastest: as in-process servers. Furthermore, ActiveX controls are OLE Automation servers. This means you can access properties of these objects and call their methods. You can see an ActiveX control in the application that is using it and interact with it directly in the container application window. This is the meaning of the term *visual editing*, or *in-place activation*. A single click activates the control rather than the double-click used by OLE Documents, and the control is active whenever it is visible (which is what the term *inside-out activation* means), without having to double-click it.

As I’ve mentioned before, an ActiveX control has properties, methods, and events. Properties can identify states, but they can also activate methods. (This is particularly true for ActiveX controls that are *updated* VBX controls, because in a VBX there *was* no other way to activate a

method than by setting a property.) Properties can refer to aggregate values, arrays, subobjects, and so on. Properties can also be dynamic (or read-only, to use the Delphi term).

In an ActiveX control, properties are divided into different groups: stock properties that most controls need to implement; ambient properties that offer information about the container (similar to the `ParentColor` or `ParentFont` properties in Delphi); extended properties managed by the container, such as the position of the object; and custom properties, which can be anything.

Events and methods are, well, events and methods. *Events* relate to a mouse click, a key press, the activation of a component, and other specific user actions. *Methods* are functions and procedures related to the control. There is no major difference between the ActiveX and Delphi concepts of events and methods.

## ActiveX Controls Versus Delphi Components

Before I show you how to use and write ActiveX controls in Delphi, let's go over some of the technical differences between the two kinds of controls. ActiveX controls are DLL-based. This means that when you use them, you need to distribute their code (the OCX file) along with the application using them. In Delphi, the code of the components can be statically linked to the executable file or dynamically linked to it using a run-time package, so you can always choose.

Having a separate file allows you to share code among different applications, as DLLs usually do. If two applications use the same control (or run-time package), you need only one copy of it on the hard disk and a single copy in memory. The drawback, however, is that if the two programs have to use two different versions (or builds) of the ActiveX control, some compatibility problems might arise. An advantage of having a self-contained executable file is that you will also have fewer installation problems.

Now, what is the drawback of using Delphi components? The real problem is not that there are fewer Delphi components than ActiveX controls, but that if you buy a Delphi component, you'll only be able to use it in Delphi and Borland C++Builder. If you buy an ActiveX control, on the other hand, you'll be able to use it in multiple development environments from multiple vendors. Even so, if you develop mainly in Delphi and find two similar components based on the two technologies, I suggest you buy the Delphi one—it will be more integrated with your environment, and therefore easier for you to use. Also, the native Delphi component will probably be better documented (from the Pascal perspective), and it will take advantage of Delphi and Object Pascal features not available in the general ActiveX interface, which is traditionally based on C and C++.

## Using ActiveX Controls in Delphi

Delphi comes with some preinstalled ActiveX controls, and you can buy and install more third-party ActiveX controls easily. After this description of how ActiveX controls work in general, I'll demonstrate one in an example.

The Delphi installation process is very simple. Select Component > Import ActiveX Control in the Delphi menu. This opens the Import ActiveX dialog box, where you can see the list of ActiveX control libraries registered in Windows. If you choose one, Delphi will read its type library, list its controls, and suggest a filename for its unit. If the information is correct, click the Create Unit button to view the Pascal source code file created by Delphi as a *wrapper* for the ActiveX control. Click the Install button to add this new unit to a Delphi package and to the Component Palette.

### Using the WebBrowser Control

To build my example, I've used a preinstalled ActiveX control available in Delphi. Unlike the third-party controls, this is not available in the ActiveX page of the palette, but in the Internet page. The control, called WebBrowser, is a wrapper around Microsoft's Internet Explorer engine. The example is a very limited Web browser.

The WebBrows program on the CD-ROM has a TWebBrowser ActiveX control covering its client area and a control bar at the top and a status bar at the bottom. To move to a given Web page, a user can type in the combo box of the toolbar, select one of the visited URLs (saved in the combo box), or click on the Open File button to select a local file.

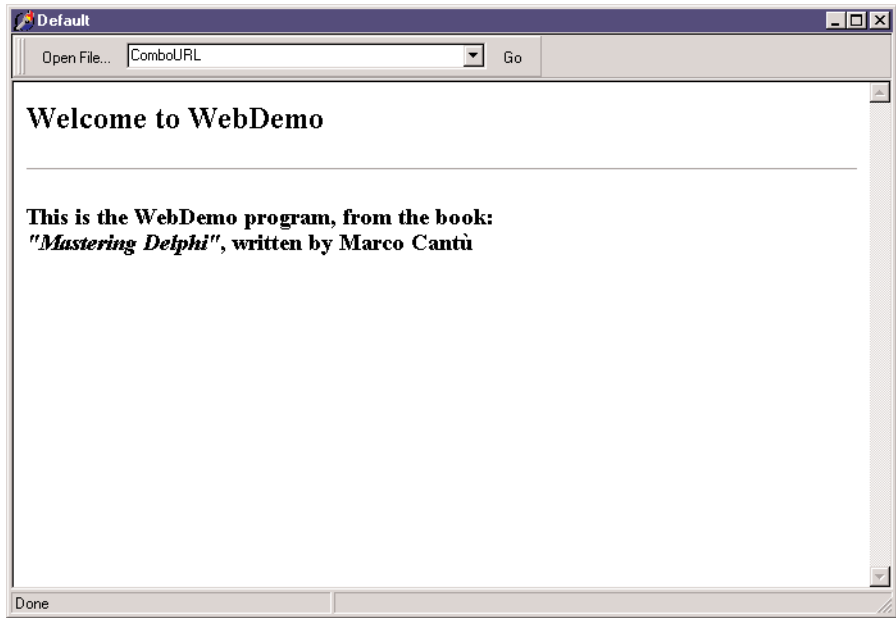
The actual implementation of the code used to select a Web or local HTML file is in the GotoPage method:

```
procedure TForm1.GotoPage(ReqUrl: string);  
begin  
    WebBrowser1.Navigate (ReqUrl, EmptyParam, EmptyParam, EmptyParam,  
        EmptyParam);  
end;
```

EmptyParam is a predefined OleVariant you can use whenever you want to pass a default value as a reference parameter. This is a handy shortcut you can use to avoid creating an empty OleVariant each time you need a similar parameter. This method is called for by a file, when the user clicks on the Enter key in the combo box, or by selecting the Go button, as you can see in the source code on the companion CD.

**FIGURE 20.14:**

The WebDemo program at startup: it fully supports graphics and all other Web extensions, as it is based on the Internet Explorer engine.



The program also handles four events of the WebBrowser control. When the download operations start and end, the program updates the text of the status bar and also the drop-down list of the combo box:

```
procedure TForm1.WebBrowser1DownloadBegin(Sender: TObject);
begin
    StatusBar1.Panels[0].Text := 'Downloading ' +
        WebBrowser1.LocationURL + '...';
end;

procedure TForm1.WebBrowser1DownloadComplete(Sender: TObject);
var
    NewUrl: string;
begin
    StatusBar1.Panels[0].Text := 'Done';
    // add URL to combobox
    NewUrl := WebBrowser1.LocationURL;
    if (NewUrl <> '') and (ComboURL.Items.IndexOf (NewUrl) < 0) then
        ComboURL.Items.Add (NewUrl);
end;
```

Two other useful events are the `OnTitleChange`, used to update the caption with the title of the HTML document, and the `OnStatusTextChange` event, used to update the second part of the status bar. This code basically duplicates the information displayed in the first part of the status bar by the previous two event handlers.

## Writing ActiveX Controls

Besides using existing ActiveX controls in Delphi, you can easily develop new ones. Although you can write the code of a new ActiveX control yourself, implementing all the required OLE interfaces (and there are many), it's much easier to use one of the techniques directly supported by Delphi:

- You can use the ActiveX Control Wizard to turn a VCL control into an ActiveX control. You start from an existing VCL component, which must be a `TWinControl` descendant, and Delphi wraps an ActiveX around it. During this step, Delphi adds a type library to the control. (Wrapping an ActiveX control around a Delphi component is exactly the opposite of what we did to use an ActiveX inside Delphi.)
- You can create an ActiveForm, place several controls inside it, and ship the entire form (without borders) as an ActiveX control. This second technique is the same one used by Visual Basic and is generally aimed at building Internet applications. However, it is also a very good alternative for the construction of an ActiveX control based on multiple Delphi controls or on Delphi components that do not descend from `TWinControl`.

An optional step you can take in both cases is to prepare a property page for the control, to use as a sort of property editor for setting the initial value of the properties of the control in any development environment—a kind of alternative to the Object Inspector in Delphi. Because most development environments allow only limited editing, it is more important to write a property page than it is to write a component or a property editor for a Delphi control.

### Building an ActiveX Arrow

As an example of the development of an ActiveX control, I've decided to take the Arrow component we developed in Chapter 11, “Creating Components,” and turn it into an ActiveX. We cannot use that component directly, because it was a graphical control, a subclass of `TGraphicControl`. However, turning a graphical control into a window-based control is usually a straightforward operation.

In this case, I've just changed the base class name to `TCustomControl` (and changed the name of the class of the control, as well, to avoid a name clash):

```
type
  TMDWArrow = class(TCustomControl)
    ...
```

The `TWinControl` class has very minimal support for graphical output. Its `TCustomControl` subclass, however, has basically the same capabilities as the `TGraphicControl` class. The key difference is that a `TCustomControl` object has a window handle.

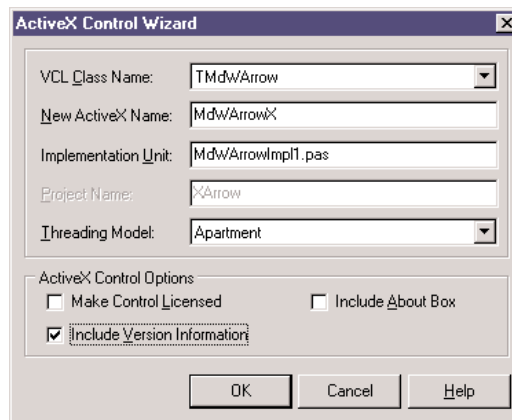


After installing this new component in Delphi, we are ready to start developing the new example. To create a new ActiveX library, select File > New, move to the ActiveX page, and choose ActiveX library. Delphi creates the bare skeleton of a DLL, as we saw at the beginning of this chapter. I've saved this library as XArrow, in a directory with the same name, as usual.

Now it is time to use the ActiveX Control Wizard, available in the ActiveX page of the Object Repository—Delphi's New dialog box. In this wizard (shown in Figure 20.15), you select the VCL class you are interested in, customize the names shown in the edit boxes, and click OK; Delphi then builds the complete source code of an ActiveX control for you.

**FIGURE 20.15:**

Delphi's ActiveX Control Wizard

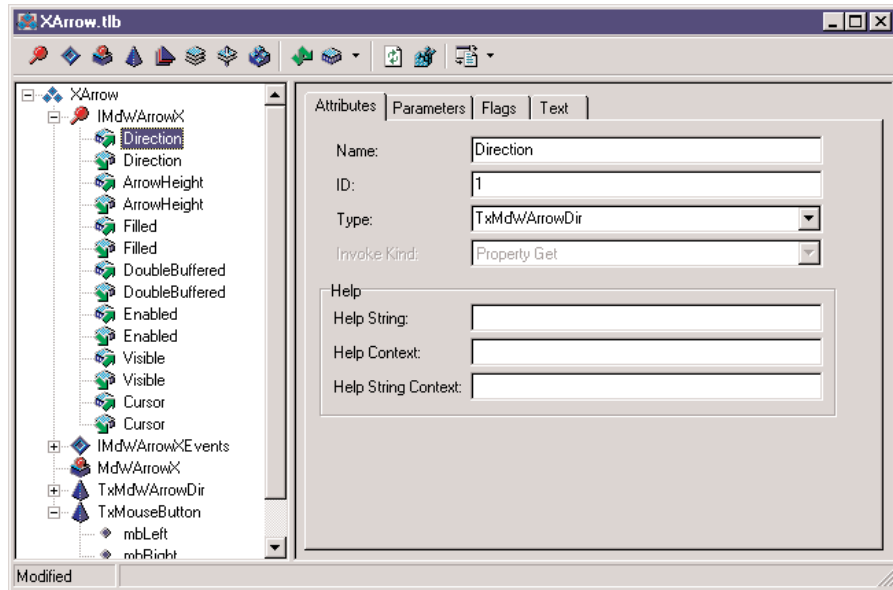


The use of the three check boxes at the bottom of the ActiveX Control Wizard window may not be obvious. If you include design-time license support, the user of the control won't be able to use it in a design environment without the proper *license key* for the control. The second check box allows you to include version information for the ActiveX, in the OCX file. If the third check box is selected, the ActiveX Control Wizard automatically adds an About box to the control.

Take a look at the code the ActiveX Control Wizard generates. The key element of this wizard is the generation of a type library. You can see the library generated for our arrow control in Delphi's type-library editor in Figure 20.16. From the type library information, the Wizard also generates an import file with the definition of an interface, the `dispinterface`, and other types and constants.

**FIGURE 20.16:**

The type-library editor with the type library of the demo ActiveX control I've created



In this example, the import file is named `XArrow_TLB.PAS`. The first part of this file includes a couple of GUIDs, one for the library as a whole and one for the control, and other constants for the definition of values corresponding to the OLE enumerated types used by properties of the Delphi control, for example:

```

type
    TxMdWArrowDir = TOLEEnum;
const
    adUp = $00000000;
    adLeft = $00000001;
    adDown = $00000002;
    adRight = $00000003;

```

The real meat is the declaration of the `IMdWArrowX` interface, which I suggest you look at in the source code. Notice that the final part of the import unit includes the declaration of the `TmdWArrowX` class. This is a `TOLEControl`-derived class you can use to install the control in Delphi, as we've seen in the first part of this chapter. You don't need this class to build the ActiveX control; you need it to install the ActiveX control in Delphi. The class used by the ActiveX server has the same class name but a different implementation.

The rest of the code, and the code you'll customize, is in the main unit, which in my example is called `MdWArrowImpl1`. This unit has the declaration of the ActiveX server object, `TmdWArrowX`, which inherits from `TActiveXControl` and implements the specific `IMdWArrowX` interface.

**NOTE**

The `TActiveXControl` class does most of the work for providing ActiveX support in Delphi. This class implements interfaces required by every ActiveX control: `IConnectionPointContainer`, `IDataObject`, `IObjectSafety`, `IObjectControl`, `IObjectInPlaceActiveObject`, `IObjectInPlaceObject`, `IObject`, `IPropertyBrowsing`, `IPersistPropertyBag`, `IPersistStorage`, `IPersistStreamInit`, `IQuickActivate`, `ISimpleFrameSite`, `ISpecifyPropertyPages`, `IViewObject`, and `IViewObject2`. Just the declaration of the `TActiveXControl` class takes more than 250 lines of code, and its implementation code is responsible for a good part of the 4,000 lines of code of the `AxCtrls` unit.

Before we customize this control in any way, let's see how it works. You should first compile the ActiveX library and then register it using Delphi's `Run > Register ActiveX Server` menu command. Now you can install the ActiveX control as we've done in the past, except you have to specify a different name for the new class to avoid a name clash. If you use this control, it doesn't look much different from the original VCL control, but the advantage is that the same component can now be installed also in other development environments.

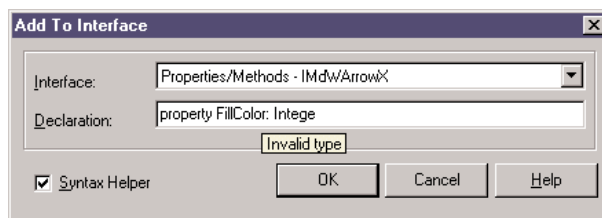
## Adding New Properties

Once you've created an ActiveX control, adding new properties, events, or methods to it is—surprisingly—simpler than doing the same operation for a VCL component. Delphi, in fact, provides specific visual support for the former, not for the latter.

You can open the Pascal unit with the implementation of the ActiveX control, and choose `Edit > Add To Interface`. As an alternative, you can use the same command from the shortcut menu of the editor. Delphi opens the `Add To Interface` dialog box (see Figure 20.17). In the combo box of this dialog box, you can choose between a new property, method, or event. In this example, the first selection will affect the `IMdWArrowX` interface and the second the `IMdWArrowXEvents` interface.

**FIGURE 20.17:**

The `Add To Interface` dialog box, with the syntax helper in action



In the edit box, you can then type the declaration of this new interface element. If the `Syntax Helper` check box is activated, you'll get hints describing what you should type next and highlighting any errors. You can see the syntax helper in action in Figure 20.17. When you define a new ActiveX interface element, keep in mind that you are restricted to OLE data

types. In the XArrow example, I've added two properties to the ActiveX control. Because the Pen and the Brush properties of the original Delphi components are not accessible, I've made their color available. These are examples of what you can write in the edit box of the Add To Interface dialog (executing it twice):

```
property FillColor: Integer;  
property PenColor: Integer;
```

**NOTE**

Since a TColor is a specific Delphi definition, it is not legal to use it. TColor is an Integer sub-range that defaults to Integer size, so I've used the standard Integer type directly.

The declarations you enter in the Add To Interface dialog box are automatically added to the control's type library (TLB) file, to its import library unit, and to its implementation unit:

```
type  
  IMdWArrowX = interface(IDispatch)  
    function Get_FillColor: Integer; safecall;  
    procedure Set_FillColor(Value: Integer); safecall;  
    function Get_PenColor: Integer; safecall;  
    procedure Set_PenColor(Value: Integer); safecall;  
    ...  
    property FillColor: Integer read Get_FillColor write Set_FillColor;  
    property PenColor: Integer read Get_PenColor write Set_PenColor;
```

All you have to do to finish the ActiveX control is fill in the Get and Set methods of the implementation. Here is the code of the first property:

```
function TMDWArrowX.Get_FillColor: Integer;  
begin  
  Result := ColorToRGB (FDelphiControl.Brush.Color);  
end;  
  
procedure TMDWArrowX.Set_FillColor(Value: Integer);  
begin  
  FDelphiControl.Brush.Color := Value;  
end;
```

If you now install this ActiveX control in Delphi once more, the two new properties will appear. The only problem with this property is that Delphi uses a plain integer editor, making it quite difficult to enter the value of a new color by hand. A program, by contrast, can easily use the RGB function to create the proper color value.

## Adding a Property Page

As it stands, other development environments can do very little with our component, because we've prepared no property page—no property editor. A property page is fundamental so

that programmers using the control can edit its attributes. However, adding a property page is not as simple as adding a form with a few controls. The property page, in fact, will integrate with the host development environment. The property page for our control will show up inside a property page dialog of the host environment, which will provide the OK, Cancel, and Apply buttons, and the tabs for showing multiple property pages (some of which might be provided by the host environment).

The nice thing is that support for property pages is built into Delphi, so adding one takes little time. You open an ActiveX project, then open the usual New Items dialog box, move to the ActiveX page, and choose Property Page. What you get is not very different from a form. In fact, the `TPropertyPage1` class (created by default) inherits from the `TPropertyPage` class of VCL, which in turn inherits from `TCustomForm`.

**TIP**

Delphi provides four built-in property pages for colors, fonts, pictures, and strings. The GUIDs of these classes are indicated by the constants `Class_DColorPropPage`, `Class_DFontPropPage`, `Class_DPicturePropPage`, and `Class_DStringPropPage` in the `AxCtrls` unit.

In the property page, you can add controls as in a normal Delphi form, and you can write code to let the controls interact. I've added to the property page a combo box with the possible values of the `Direction` property, a check box for the `Filled` property, an edit box with an `UpDown` control to set the `ArrowHeight` property, and two shapes with corresponding buttons for the colors. The only code added to the form relates to the two buttons used to change the color of the two shape components, which offer a preview of the colors of the actual ActiveX control. The `OnClick` event of the button uses a `ColorDialog` component, as usual:

```
procedure TPropertyPage1.ButtonPenClick(Sender: TObject);
begin
  with ColorDialog1 do
  begin
    Color := ShapePen.Brush.Color;
    if Execute then
    begin
      ShapePen.Brush.Color := Color;
      Modified; // enable Apply button!
    end;
  end;
end;
```

What is important to notice in this code is the call to the `Modified` method of the `TPropertyPage` class. This call is required to let the property page dialog box know we've modified one of the values and to enable the Apply button. When a user interacts with one of the other controls of this form, this call is made automatically. For the two buttons, however, we need to add this line ourselves.

**TIP**

Another tip relates to the `Caption` of the property page form. This will be used in the property dialog box of the host environment as the caption of the tab corresponding to the property page.

The next step is to associate the controls of the property page with the actual properties of the ActiveX control. The property page class automatically has two methods for this: `UpdateOleObject` and `UpdatePropertyPage`. As their names suggest, these two methods copy data from the property page to the ActiveX control and vice versa. Here is the code for my example:

```
procedure TPropertyPage1.UpdatePropertyPage;  
begin  
  { Update your controls from the OleObject }  
  ComboDir.ItemIndex := OleObject.Direction;  
  CheckFilled.Checked := OleObject.Filled;  
  EditHeight.Text := IntToStr (OleObject.ArrowHeight);  
  ShapePen.Brush.Color := OleObject.PenColor;  
  ShapePoint.Brush.Color := OleObject.FillColor;  
end;  
  
procedure TPropertyPage1.UpdateObject;  
begin  
  { Update the OleObject from your controls }  
  OleObject.Direction := ComboDir.ItemIndex;  
  OleObject.Filled := CheckFilled.Checked;  
  OleObject.ArrowHeight := UpDownHeight.Position;  
  OleObject.PenColor := ColorToRGB (ShapePen.Brush.Color);  
  OleObject.FillColor := ColorToRGB (ShapePoint.Brush.Color);  
end;
```

The final step is to connect the property page itself to the ActiveX control. When the control was created, the Delphi ActiveX Control Wizard automatically added a declaration for the `DefinePropertyPages` method to the implementation unit. In this method, we call the `DefinePropertyPage` method (this time the method name is singular) for each property page we want to add to the ActiveX. This method has as its parameter the GUID of the property page, something you can find in the corresponding unit. (Of course, you'll need to add a `uses` statement referring to that unit.) Here is the code of my example:

```
procedure TMdWArrowX.DefinePropertyPages(  
  DefinePropertyPage: TDefinePropertyPage);  
begin  
  DefinePropertyPage(Class_PropertyPage1);  
end;
```

**NOTE**

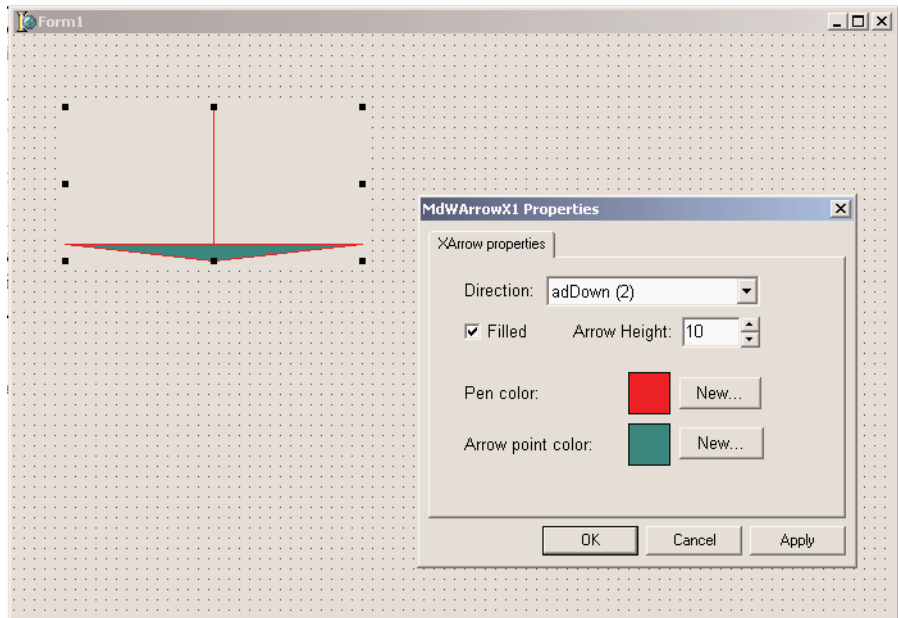
The connection between the ActiveX control and its property page takes place using a GUID. This is possible because the property page object can be created through a class factory, and its GUID is stored in the Windows Registry when you register the ActiveX control library. To see

what's going on, look at the initialization section of the property page unit, which calls `TActiveXPropertyPageFactory.Create`.

Now that we've finished developing the property page, and after recompiling and reregistering the ActiveX library, we can install the ActiveX control inside a host development environment (including Delphi itself) and see how it looks. Figure 20.18 shows an example. (If you've already installed the ActiveX control in Delphi, you should uninstall it prior to rebuilding it. This process might also require closing and reopening Delphi itself.)

**FIGURE 20.18:**

The XArrow ActiveX control and its property page, hosted by the Delphi environment



## ActiveForms

As I've mentioned, Delphi provides an alternative to the use of the ActiveX Control Wizard to generate an ActiveX control. You can use an ActiveForm, which is an ActiveX control that is based on a form and can host one or more Delphi components. This is exactly the technique used in Visual Basic to build new controls, and it makes sense when you want to create a compound component.

For example, to create an ActiveX clock, we can place on an ActiveForm a label (a graphic control that cannot be used as a starting point for an ActiveX control) and a timer, and connect

the two with a little code. The form/control becomes basically a container of other controls, which makes it very easy to build compound components (easier than for a VCL compound component).

To build such a control, close the current project, and select the ActiveForm icon in the ActiveX page of the File > New dialog box. Delphi asks you for some information in the following ActiveForm Wizard dialog box, similar to the ActiveX Control Wizard dialog box.

## ActiveForm Internals

Before we continue with the example, let's look at the code generated by the ActiveForm Wizard. The key difference from a plain Delphi form is in the declaration of the new form class, which inherits from the TActiveForm class and implements a specific ActiveForm interface:

```
type
  TAXForm1 = class(TActiveForm, IAXForm1)
```

As usual, the IAXForm interface is declared in the type library and in a corresponding Pascal file generated by Delphi. Here is a small excerpt of the IAXForm1 interface from the XF1Lib.pas file, with some comments I've added:

```
type
  IAXForm1 = interface(IDispatch)
    ['{51661AA1-9468-11D0-98D0-444553540000}']
    // Get and Set methods for TForm properties
    function Get_Caption: WideString; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    ...
    // TForm methods redeclared
    procedure Close; safecall;
    ...
    // TForm properties
    property Caption: WideString read Get_Caption write Set_Caption;
```

The code generated for the TAXForm1 class implements all the Set and Get methods, which change or return the corresponding properties of the form, and it implements the events, which again are the events of the form. Here is a small excerpt:

```
private
  procedure ActivateEvent(Sender: TObject);
protected
  procedure Initialize; override;
  function Get_Caption: WideString; safecall;
  procedure Close; safecall;
  procedure Set_Caption(const Value: WideString); safecall;
```

Let's look at the implementation of properties first:

```
function TAXForm1.Get_Caption: WideString;
```



```
begin
    Result := WideString(Caption);
end;

procedure TAXForm1.Set_Caption(const Value: WideString);
begin
    Caption := TCaption(Value);
end;
```

The TForm events are set to the internal methods when the form is created:

```
procedure TAXForm1.Initialize;
begin
    OnActivate := ActivateEvent;
    ...
end;
```

Each event then maps itself to the external ActiveX event, as in the following two methods:

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
    if FEvents <> nil then FEvents.OnActivate;
end;
```

Because of this mapping, you should not handle the events of the form directly. Instead, you can either add some code to these default handlers or override the TForm methods that end up calling the events. (This is exactly the approach you use when building a Delphi component.) Keep in mind that the interface properties of an ActiveForm are meant for developers using it as a control, not for final users of the ActiveForm on the Web. This mapping problem refers only to the events of the form itself, not to the events of the components of the form. You can continue to handle the events of the components as usual.

## The XClock ActiveX Control

Now that we've looked at the code generated by Delphi, we can return to the development of the XClock example. Place on the form a label with a large font and centered text, aligned to the client area and a timer. Then write an event handler for its OnTimer event, so that the control updates the output of the label with the current time every second:

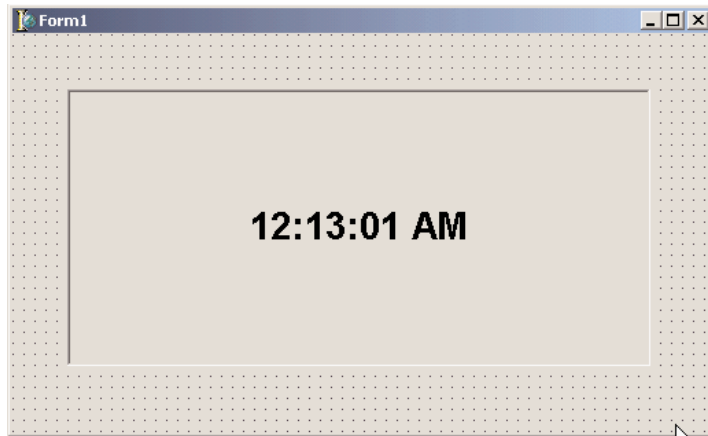
```
procedure XClock.Timer1Timer(Sender: TObject);
begin
    Label1.Caption := TimeToStr (Time);
end;
```

Now compile this library, register it, and install it in a package to test it in the Delphi environment. You can see an example of its use in Figure 20.19. Notice in this figure the effect of

the sunken border. This is controlled by the `AxBorderStyle` property of the active form, one of the few properties of active forms that is not available for a plain form.

**FIGURE 20.19:**

The ActiveX timer installed in a Delphi package



ActiveForms are usually considered as a technique to deploy a Delphi application via the Internet. However, the ActiveX and ActiveForm support provided by Delphi represent to different ways to build ActiveX controls, which can be used both on a Web page and in another development environment.

## ActiveX in Web Pages

In the last example, we used Delphi's ActiveForm technology to create a new ActiveX control. In fact, an ActiveForm is an ActiveX control based on a form. Borland documentation often implies that ActiveForms should be used in HTML pages, but you can use any ActiveX control on a Web page.

### NOTE

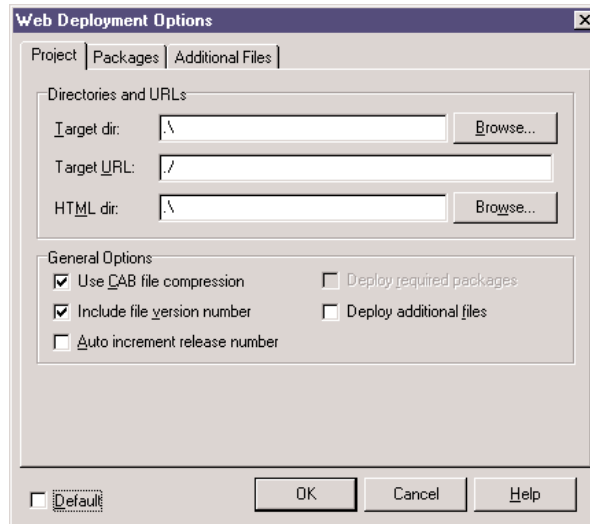
Microsoft once promoted ActiveX as an Internet technology for delivering interactive content. Due to complexities and security problems inherent in downloading executable code, the market never really bought into this. Microsoft has since dropped ActiveX from its Internet technologies list. Still, this technology might have a value in an intranet to let you deliver small applications to users of your local area network, as you can relax the security settings when accessing local Web sites.

Basically, each time you create an ActiveX library, Delphi enables the `Project > Web Deployment Options` and `Project > Web Deploy` menu items. The first allows you to specify how and where to deliver the proper files. As shown in Figure 20.20, in this dialog box you

can set the server directory for deploying the ActiveX component, the URL of this directory, and the server directory for deploying the HTML file (which will have a reference to the ActiveX library using the URL you provide).

**FIGURE 20.20:**

The Web Deployment Options dialog box



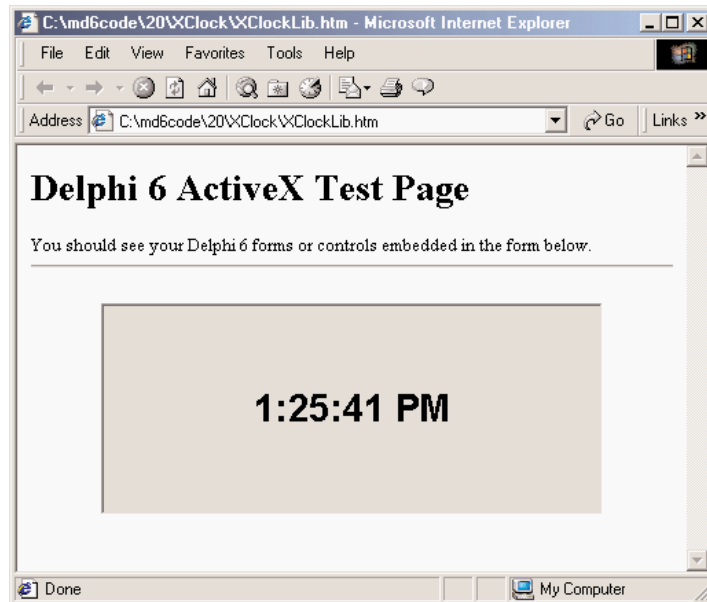
You can also specify the use of a compressed CAB file, which can store the OCX file and other auxiliary files, such as packages, making it easier and faster to deliver the application to the user. A compressed file, in fact, means a faster download. Using the options shown in Figure 20.20, Delphi generates the HTML file and CAB file for the XClock project in the same directory. Opening this HTML file in Internet Explorer produces the output shown in Figure 20.21.

**WARNING**

At times, when you load an HTML page referring to an ActiveX, all you get is a red X marker indicating a failure to download the control. There are various possible explanations for this problem. First, Internet Explorer must be set up properly, allowing the download of controls and (if the control is not signed) lowering the security level. Second, other problems might arise when the control requires a DLL or a package that is not part of the downloaded CAB file. Third, you might get the red slash marker when there is a mismatch in the version number—or you might see an older version of the control in action.

**FIGURE 20.21:**

The XClock control in the sample HTML page



Besides showing you how to deploy the XClock control on a Web page, I've created the XForm1 example to demonstrate the problems with event handlers of ActiveForms mentioned in the previous section "ActiveForm Internals." Because the form events are exported as events of the control, you should not handle the events of the form directly but add some code to the default handlers provided by the Active Form. For example, if you add a handler for the OnPaint event of the form and write the following code, it will never be executed:

```
procedure TFormX1.FormPaint(Sender: TObject);
begin
    Canvas.Brush.Color := clYellow;
    Canvas.Ellipse(0, 0, Width, Height);
end;
```

If you want to paint something on the form's background, instead, you have to modify the corresponding handler installed by the ActiveForm Wizard:

```
procedure TFormX1.PaintEvent(Sender: TObject);
begin
    Canvas.Brush.Color := clBlue;
    Canvas.Rectangle (20, 20, ClientWidth - 20, ClientHeight - 20);
    if FEvents <> nil then FEvents.OnPaint;
end;
```

As an alternative, you can place a frame, a panel, or another component on the surface of the form, and handle *its* events. In the XForm1 example, I've added a PaintBox component, with a bevel component behind it to make the area of the PaintBox visible.

## The Role of an ActiveX Form on a Web Page

Before we look at another example, it is important to stop for a second to consider the role of an ActiveX form placed inside a Web page. Basically, placing a form in a Web page corresponds to letting a user download and execute a custom Windows application. There is little else happening. You download an executable file and start it. This is one of the reasons the ActiveX technology raises so many concerns about security.

The XFUser example highlights the situation. It calls the `GetUserName` Windows API function and shows the user name on the screen. Its effect is certainly not astonishing, as the name of the user will be displayed in a label. However, this example highlights a couple of important points (which apply both to ActiveForms and ActiveX controls in general):

- In an ActiveX control or form, you can call any Windows API function (which means the user viewing the Web page must have Windows on his or her computer) or certain Windows API-compatible libraries.
- An ActiveX can access the system information of the computer, such as the user name, the directory structure, and so on. This is why, before downloading an ActiveX, Web browsers check whether the ActiveX has a proper authentication, or signature. (You should note that this signature identifies the author of the control and that the module has not been corrupted or tampered with since the author published it; it doesn't prove in any way that the control is safe.)

Well, I could continue, but I think my point is clear. ActiveX controls and ActiveForms inside Web pages have problems, and even Microsoft has slowly abandoned this technology. For this reason, I'm going to show you only one more example, which is instructive on how external environments can interact with an ActiveX control.

## Setting Properties for the XArrow

An ActiveForm has a few properties you can set when you use it inside a development environment, and a plain ActiveX control has even more. For example, if you want to set properties in the HTML file hosting the control, you can use a special `param` tag, but the control must support a special interface known as `IPersistPropertyBag`.

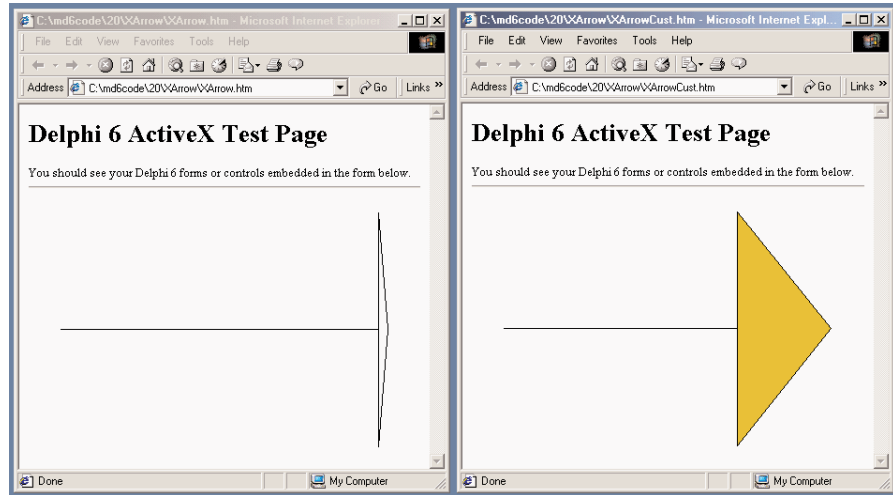
Starting with Delphi 4, the `IPersistPropertyBag` support is built in, providing support for all of the properties of the ActiveX control or ActiveForm. As an example, I've used the Web Deploy options on the XArrow control. Then, I've modified the automatically generated HTML file with three param tags:

```
<object classid="clsid:482B2145-4133-11D3-B9F1-00000100A27B"  
  codebase="./XArrow.cab" width="350" height="250" align="center"  
  hspace="0" vspace="0">  
  <param name="ArrowHeight" value="100">  
  <param name="Filled" value="-1">  
  <param name="FillColor" value="111829">  
</object>
```

You can compare the default and customized output of the control in Figure 20.22.

**FIGURE 20.22:**

By using the `param` tag, we can set values for the properties of an ActiveX control in the HTML file hosting it. The two copies of the program show the default and the customized output.



## Introducing COM+

In addition to plain COM servers, Delphi also allows you to create enhanced COM objects, including stateless objects and transaction support. This type of COM object was first introduced by Microsoft with the MTS (Microsoft Transaction Server) acronym, and later renamed as COM+ in Windows 2000.

Delphi 6 supports building both standard stateless objects and DataSnap remote data modules based on stateless objects. In both cases you'll start the development by using one of the available Delphi wizards, using the New Items dialog box and selecting the Transactional Object icon of the ActiveX page or the Transactional Data Module icon of the Multitier page. You must add these objects to an ActiveX library project, not to a plain application. Another icon, COM+ Event Object, is used to support COM+ events.

MTS is an operating-system service you can install on Windows NT and 98; it was renamed as COM+ in Windows 2000, so I'll call it COM+ but this actually refers to both. This system service provides a run-time environment supporting database transaction services, security, resource pooling, and an overall improvement in robustness for DCOM applications. The run-time environment manages objects called *COM+ components*. These are COM objects stored in an in-process server (that is, a DLL). While other COM objects run directly in the client application, COM+ objects are handled by this run-time environment, in which you install the COM+ libraries. COM+ objects must support specific COM interfaces, starting with *IObjectControl*, which is the base interface (like *IUnknown* for a COM object).

Before getting into too many technical and low-level details, let's consider COM+ from a different perspective. What are the benefits of this approach? COM+ provides a few interesting features, including:

**Role-based security** The role assigned to a client determines whether it has the right to access the interface of a data module.

**Reduced database resources** You can reduce the number of database connections, as the middle tier logs on to the server and uses the same connections for multiple clients (although you cannot have more clients connected at once than you have licenses for the server).

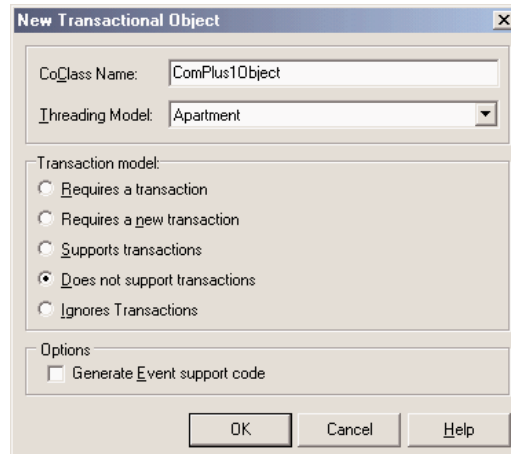
**Database transactions** COM+ transaction support includes operations on multiple databases, although few SQL servers other than Microsoft's support COM+ transactions.

## Creating a COM+ Component

The starting point for creating a COM+ component is the creation of an ActiveX library project. After this step you can select a new Transactional Object in the ActiveX page of the New Items dialog box. In the resulting dialog box (see Figure 20.23), enter the name of the new component (*ComPlus1Object* in my *ComPlus1* example).

**FIGURE 20.23:**

Delphi's New Transactional Object dialog box, used to create a COM+ object



The New Transactional Object dialog box allows you to enter a name for the class of the COM+ object, the threading model (because COM+ serializes all the requests, Single or Apartment will generally do), and a transactional model:

**Requires A Transaction** indicates that each call from the client to the server is considered to be a transaction (unless the caller supplies an existing transaction context).

**Requires A New Transaction** indicates that each call is considered a new transaction.

**Supports Transactions** indicates that the client must explicitly provide a transaction context.

**Does Not Support Transaction** (the default choice, and the one I've used) indicates that the remote data module won't be involved in any transaction.

As you close this dialog, Delphi adds a type library and an implementation unit to the project and opens the type-library editor, where you can define the interface of your new COM object. For this example I've added a `Value` integer property, an `Increase` method having as parameter an amount, and an `AsText` method returning a `WideString` with the formatted value. As you accept the edits in the type-library editor (clicking the Refresh button or closing the window), Delphi shows the Implementation File Update Wizard, if the corresponding IDE option is set. This wizard will ask for your confirmation before adding four methods to the class, including the get and set methods of the property. You can now write some code for the COM object, which for my example was quite trivial.

"Ignores Transactions" indicates that objects do not participate in transactions, regardless of whether the client has a transaction. The difference from the setting "Does not support transactions" prevents the object from being activated if the client has a transaction.



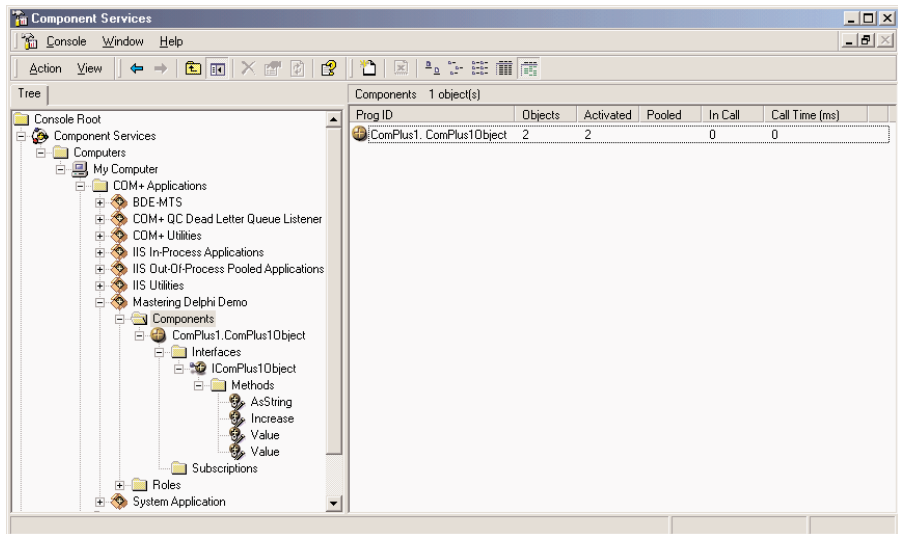
As you've compiled an ActiveX library, or COM library, which hosts a COM+ component, you can use the Component Services administrative tool (shown in the Microsoft Management Console, or MMC) to install and configure the COM+ component. Even better, you can use the Delphi IDE to install the COM+ component using the Run > Install COM+ Object menu command. In the subsequent dialog box, you'll be able to select the component to install (as a library can host multiple components), and choose the COM+ application where to install the component.

A COM+ application is nothing more than a way to group COM+ components; it is not an actual program or anything like one (why they call it application is not fully clear to me). So in the Install COM+ Object dialog, you can select an existing application/group, choose the Install Into New Application page, and enter a name and description for it.

I've called the COM+ application *Mastering Delphi Demo*, as you can see in Figure 20.24 in the Component Services administration. This is the front end you can use to fine-tune the behavior of your COM+ components, setting their activation model (just-in-time activation, object pooling, and so on), their transaction support, and the security and concurrency models you want to use. You can also use this console to monitor the objects and the actual method calls (in case these take a long time to execute). In Figure 20.24, you can see that there are currently two active objects.

**FIGURE 20.24:**

The newly installed COM+ component inside a custom COM+ application (as shown by Microsoft's Component Services tool)



**Tip**

Because you've created one or more objects, the COM library remains loaded in the COM+ environment and some of the objects might be kept in cache, even if there are no clients connected to them. For this reason, you cannot generally recompile the COM library after using it, unless you use the MMC to shut it down.

I've actually created a client program for the COM+ object, but this is exactly like any other Delphi COM client. After importing the type library, which is automatically registered while installing the component, I created an interface-type variable referring to it and called its methods as usual. You can find the example on the CD accompanying this book.

## Transactional Data Modules

The same types of features are available when creating a *transactional data module*—that is, a remote data module within a COM+ component. Once you've created a transactional data module, you can build a Delphi DataSnap application as we've done in Chapter 17, "Multi-tier Database Applications with DataSnap." You can add one or more dataset components, add one or more providers, and export the provider(s). You can also add custom methods to the data module type library by editing the type library or using the Add To Interface command.

Within a COM+ component or transactional data module, you can also use the `GetObjectContext` method, which returns the `IObjectContext` interface of the COM+ object. The `IObjectContext` interface provides support for transactions:

- You can use `SetComplete` to tell the COM+ environment that the object has finished working and can be deactivated, so that the transaction can be committed.
- You can call `EnableCommit` to indicate that the object hasn't finished but the transaction should be committed.
- You can call `DisableCommit` to stop the commit operation, even if the method is done, disabling the object deactivation between method calls.
- You can call `SetAbort` to say that the object has finished and can be activated but the transaction cannot be committed.
- You can call `IsInTransaction` to check whether the object is part of a transaction.

Other methods of the `IContextObject` interface include `CreateInstance`, which creates another COM+ object in the same context and within the current transaction, `IsCallerInRole`, which checks if the object's caller is in a particular "security" role, and `IsSecurityEnabled` (whose name is self-explanatory).

Once you've built a transactional data module within a server library, you can install it as I've shown above for a plain COM+ object. After the transactional data module has been installed, it will be directly available to other applications and visible in the management console.

An important feature of COM+ is that it becomes much easier to configure DCOM support using this environment. In fact, the COM+ environment of a client computer can grab information from the COM+ environment of a server computer, including registration information for the COM+ object you want to be able to call over a network. The same network configuration is way more complex if done with plain DCOM, without MTS or COM+.

**TIP**

Even though COM+ configuration is much better than DCOM configuration, still you are limited to computers with a recent version of the Windows operating system. Considering that even Microsoft is moving away from DCOM technology, before you build a large system based on this technology you should at least evaluate the alternative provided by SOAP (discussed in Chapter 23, "XML and SOAP").

## COM+ Events

Client applications that use traditional COM objects and Automation servers can call methods of those servers, but this is not an efficient way to check whether the server has updated data for the client. For this reason, it is possible for a client to define a COM object that implements a *callback* interface, pass this object to the server, and let the server call it. Traditional COM events (which use the `IConnecti onPoint` interface) are simplified by Delphi for Automation objects, but are still quite complex to handle.

COM+ introduces a simplified event model, in which the events are COM+ components and the COM+ environment manages the connections. In traditional COM callbacks, the server object doesn't have to keep track of the multiple clients it has to notify to, which is one of the reasons for the complexity of its code. In COM+, the server calls into a single event interface, and the COM+ environment will forward the event to all clients that have expressed interest for it. This way, the client and the server are less coupled, making it possible for a client to receive notification from different servers, without any change in its code.

**NOTE**

Some critics say that Microsoft introduced this model only because it was very complex to handle COM events in the traditional way for Visual Basic developers. Windows 2000 actually provides a few operating-system features specifically intended for VB developers.

To create a COM+ event, you should create a COM library (or ActiveX library) and use the COM+ Event Object wizard. The resulting project will contain a type library with the definition of the interface used to fire the events, plus some *fake* implementation code. The actual server that will receive the notification of the events, in fact, will provide the actual implementation of the interface. The fake code is there only to support Delphi's COM registration system.

While building the MdComEvents library, I added to the type library a single method with two parameters, resulting in the following code (in the interface definition file):

```
type
  IMdInform = interface(IDispatch)
    ['{202D2CC8-8E6C-4E96-9C14-1FAAE3920ECC}']
    procedure Informs(Code: Integer; const Message: WideString); safecall;
end;
```

The main unit includes the fake COM object and its class factory, to let the server register itself. The code looks like this (notice that the method is abstract, and it has no implementation):

```
type
  // fake abstract class
  TMdInform = class (TAutoObject, IMdInform)
  protected
    procedure Informs(Code: Integer; const Message: WideString);
    virtual; safecall; abstract;
  end;

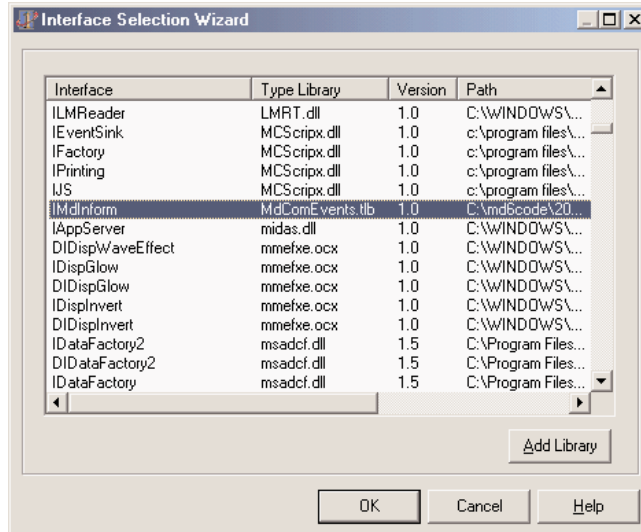
begin
  TAutoObjectFactory.Create(ComServer, TMdInform, Class_MdInform,
    ciMultiInstance, tmApartment);
end.
```

At this point, you can compile the library and install it in the COM+ environment. Again, after selecting a COM+ application (that is, a group of COM+ components), use the shortcut menu of its Components folder to add a new component to it. In the COM Component Install Wizard, click the Install New Event Class button and select the library you've just compiled. Your COM+ event definition will be automatically installed.

To test whether it works, you'll have to build an actual implementation of this event interface and a client invoking it. The implementation can be added to another ActiveX library, hosting a plain COM object. Within Delphi's COM Object Wizard you can select the interface to implement, choosing it in the list that appears when you select the List button. An example of this rather long list, dubbed Interface Selection Wizard, is shown in Figure 20.25.

**FIGURE 20.25:**

Delphi 6's new Interface Selection Wizard, used in this case to select an event interface



The resulting library, which in my example is called EvtSubscriber, exposes an Automation object, a COM object implementing the IDispatch interface (which is mandatory for COM+ Events). In my example, the object has the following definition and code:

```

type
  TInformSubscriber = class(TAutoObject, IMdInform)
  protected
    procedure Inform(Code: Integer; const Message: WideString); safecall;
  end;

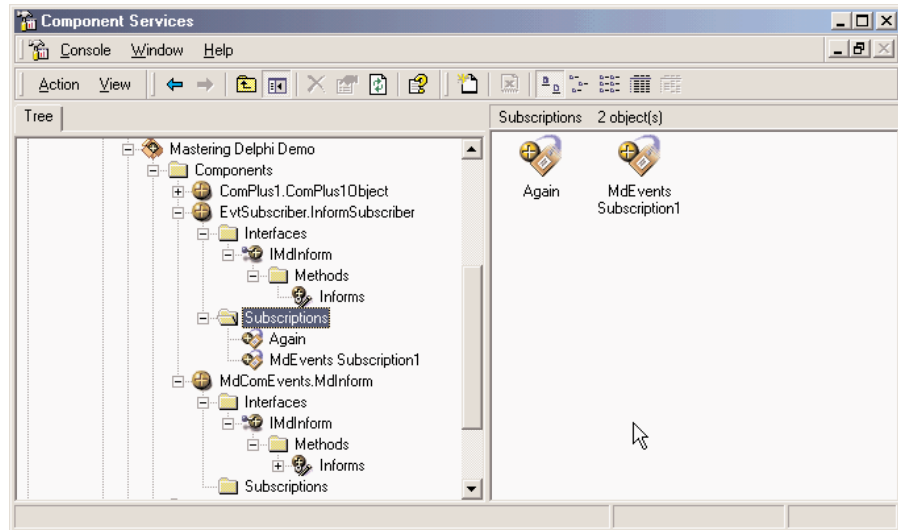
procedure TInformSubscriber.Informs(Code: Integer; const Message: WideString);
begin
  ShowMessage ('Message <' + IntToStr (Code) + '>: ' + Message);
end;

```

After compiling this library, you can first install it into the COM+ environment, then you have to bind it to the event. This second step is accomplished in the Component Services management console by selecting the Subscriptions folder under the event object registration, and using the New > Subscription shortcut menu. In the resulting wizard, you can choose the interface to implement (but there is probably only one interface in your COM+ event library), then you'll see a list of COM+ components that implement this interface. Selecting one or more of them you'll set up the subscription binding, which is listed under the Subscriptions folder. You can see an example of my configuration while building this example in Figure 20.26.

**FIGURE 20.26:**

A COM+ Event with two subscriptions in the Component Services management console.



Finally, we can focus on the application that fires the event, which I've called Publisher, as it publishes the information other COM objects are interested in. This is actually the simplest step of this process, as it is a plain COM client that uses the event server. After importing the COM+ event type library, you can add to the publisher code like this:

```
var
  Inform: IMdInform;
begin
  Inform := CoMdInform.Create;
  Inform.Informs (20, Edit1.Text);
```

My example actually creates the COM object in the `FormCreate` method to keep the reference around, but the effect is the same. Now the client program thinks it is calling the COM+ event object, but this object, provided by the COM+ environment, actually calls the method for each of the active subscribers. In this case you'll end up seeing a message box. To make things a little more interesting, you can actually subscribe twice the same server to the event interface. The net effect is, without touching your client code, you'll get two message boxes, one for each of the subscribed servers.

Obviously this effect becomes interesting when you have multiple different COM components that can handle the event, as you can easily enable and disable each of them in the management console, changing the COM+ environment without modifying the code of any program.

## What's Next?

In this chapter, I have discussed applications of Microsoft's COM technology, covering automation, documents, controls, and more. We've seen how Delphi makes the development of Automation servers and clients, and ActiveX controls, reasonably simple. Delphi even enables us to wrap components around Automation servers, such as Word and Excel.

I've also introduced elements of COM+ provided by Delphi 6 and discussed briefly the use of ActiveForms inside a browser. I've stated this is not really a very good approach to Internet Web programming—the topic discussed in the next two chapters.

As I mentioned earlier, if COM has a key role in Windows 2000, future versions of Microsoft's operating systems will downplay its role to push the dotNet infrastructure including SOAP and XML. But you'll have to *wait* until Chapter 23 to see a complete discussion of Delphi 6 XML support.

# Internet Programming: Sockets and Indy Components

- Using sockets
- The WinInet API
- Standard Internet actions
- The Internet Direct (Indy) components
- Mail and HTTP



In this chapter I'll provide an introduction to Internet programming in Delphi, using some of the components available in the IDE. With the advent of the Internet era, writing programs based on Internet protocols has become commonplace, so I've devoted three chapters to this topic. This chapter focuses on low-level socket programming and Internet protocols; the next chapter is devoted to server-side Web programming; and the final chapter of the book covers Web services, XML, and SOAP.

We'll start by looking at the use of Delphi socket components, then we'll move to the use of the Internet Direct (Indy) components supporting the most common Internet protocols. I will introduce some elements of HTTP programming, leading up to building HTML files out of database data.

Although you probably just want to use a high-level protocol, our discussion of Internet programming starts from the core concepts and low-level applications. The reason is that understanding TCP/IP and sockets will help you grasp most of the other concepts more easily.

Specifically, I'm going to focus on the use of the connectivity provided by Delphi socket components, which are based on TCP/IP and the low-level Windows sockets. Before we look into the foundations of sockets, let me list a couple of alternative approaches you can use for Internet programming, which I'll cover in more detail in later sections:

- The Delphi socket components provide a good interface for direct use of the Windows sockets API, implementing some custom protocols of your own.
- For standard protocols, you can also use the Indy components, included in Delphi 6.

## Foundations of Socket Programming

To understand the description of the socket components in the Delphi Help file, and also to read along with the description of the examples in the book, you need to be confident with several terms related to the Internet in general and with sockets in particular.

The heart of the Internet is the Transmission Control Protocol/Internet Protocol (TCP/IP for short), a combination of two separate protocols that work together to provide connection over the Internet (and can also provide connection over a private intranet). In brief, IP is responsible for defining and routing the *datagrams* (Internet transmission units) and specifying the addressing scheme. TCP is responsible for higher-level transport services.

### Configuring a Local Network: IP Addresses

If you have a local network available, you'll be able to test the following programs on it; otherwise, you can simply use the same computer as client and server. In this case, as I've

done in the examples, use the address 127.0.0.1 (or *localhost*), which is invariably the address of the current computer. If your network is complex, ask your network administrator to set up proper IP addresses for you. If you want to set up a simple network with a couple of spare computers, you can simply set up the IP address yourself, a 32-bit number usually represented with each of its four components (called *octets*) separated by dots. These numbers have a complex logic underneath them, with the first octet indicating the class of the address.

Specific IP addresses are actually reserved for unregistered internal networks. Internet routers will ignore these address ranges, so you can freely do your tests without interfering with an actual network. For example, the “free” IP address range 192.168.0.0 through 192.168.0.255 can be used for experiments on a network of fewer than 255 machines.

## Local Domain Names

How does the IP address map to a name? On the Internet, the client program looks up the values on a domain name server. But it is also possible to have a local *hosts* file, a text file that you can easily edit to provide nice local mappings. You can take a look at the `HOSTS.SAM` file (installed in a subdirectory of the Windows directory) to see a sample and then eventually rename the file as `HOSTS`, without the extension, to activate local host mapping.

Should you use an IP or a hostname in your programs? Hostnames are easier to remember and won't require a change if the IP address changes (for whatever reason). On the other hand, IP addresses don't require any resolution, while hostnames must be resolved (a time-consuming operation if the lookup takes place on the Web).

## TCP Ports

Each TCP connection takes place through a *port*, which is represented by a 16-bit number. The IP address and the TCP port together specify an Internet connection, or a *socket* (to use a more precise term). Different processes running on the same machine cannot use the same socket—the same port.

Some TCP ports have a standard usage for specific high-level protocols and services. In other words, you should use those port numbers when implementing those services and stay away from them in any other case. Here is a short list:

Protocol	Port
HTTP (Hypertext Transfer Protocol)	80
FTP (File Transfer Protocol)	21
SMTP (Simple Mail Transfer Protocol)	25
POP3 (Post Office Protocol, version 3)	110
Telnet	23

The `Services` file (another text file similar to the `Hosts` file) lists the standard ports used by services. You can add your own entry to the list, giving your service a name of your own choosing. Client sockets always specify the port number or the service name of the server socket to which they want to connect.

## High-Level Protocols

I've used the term *protocol* many times now, but what does it mean exactly? A protocol is a set of rules the client and server agree upon to determine the communication flow. The low-level Internet protocols, such as TCP/IP, are usually implemented by an operating system. But the term *protocol* is also used for high-level Internet standard protocols (such as HTTP, FTP, or SMTP). These protocols are defined in standard documents available on the Web on the site of the Internet Engineering Task Force ([www.ietf.org](http://www.ietf.org)).

If you want to implement a custom communication, you can define your own (possibly simple) protocol, a set of rules determining which request the client can send to the server and how the server can respond to the various possible requests. We'll see an example of a custom protocol later on. Transfer protocols are at a higher level than transmission protocols, because they abstract from the transport mechanism provided by TCP/IP. This makes the protocols independent not only from the operating system and the hardware but also from the physical network.

## Socket Connections

How do you start communication through a socket? The server program starts running first, but it simply waits for a request from a client. The client program requests a connection indicating the server it wishes to connect to. When the client sends the request, the server can accept the connection, starting a specific server-side socket, which connects to the client-side socket.

To support this model, there are three different types of socket connections:

- *Client connections* are initiated by the client and connect a local client socket with a remote server socket. Client sockets must describe the server they want to connect to, by providing either its hostname or IP address and its port.
- *Listening connections* are passive server sockets waiting for a client. Once a client makes a new request, the server spawns a new socket devoted to that specific connection and then gets back to listening. Listening server sockets must indicate the port that represents the service they provide. (In fact, the client is going to connect through that port.)
- *Server connections* are the connections activated by servers, as they accept a request from a client.

These different types of connections are important only for establishing the link from the client to the server. Once the link is established, both sides are free to make requests and to send data to the other side.

## Delphi Socket Components

Delphi 6 ships with three sets of socket components you can use to read and write information over a TCP/IP connection. The Internet page of the palette hosts the ClientSocket and ServerSocket components (already available in Delphi 5) plus the new TcpClient and TcpServer components (also available in Kylix). To these *native* Borland sockets, the Indy components add the IdTCPClient and the IdTCPServer components. These three sets of components have very similar features, which depend on the underlying protocol. There are technical differences, of course, and platform issues, which can determine your choice.

### Host and Port

To use a socket component, you must provide a host and a service. On the server side the host is the address of the current computer; on the client side you can indicate either a domain name or an IP address. The ClientSocket component uses two different properties for these settings (Host and Address), while the TcpClient component and the IdTCPClient component use a single string property and can determine whether it is a hostname or an address by looking at its content (the property is called RemoteHost and Host, respectively).

Similarly the service is indicated with the Port property or the Service property, in a ClientSocket, and with the single RemotePort string in a TcpClient and with the single numeric Port in an IdTCPClient. The respective servers determine their listening port using analogous properties (called Port, LocalPort, and DefaultPort in the three components).

#### NOTE

The Indy server sockets allow binding to multiple IP addresses and/or ports, using the Bindings collection.

### Blocking, Nonblocking, and Multithreaded Connections

When working with sockets in Windows, multiple approaches are possible. Reading data from a socket or writing to it can happen asynchronously, so that it does not block the execution of other code in your network application. This is called a *nonblocking connection*. Nonblocking connections read and write asynchronously: the Windows socket support basically sends a message when data is available. Using the ClientSocket and ServerSocket components, for example, the system fires the OnRead or OnWrite events of the client, and the

OnClientRead or OnClientWrite events of servers inform your socket when the other end of the connection tries to read or write some data.

As an alternative to the asynchronous approach, you can also use *blocking connections*, where your application waits for the reading or writing to be completed before executing the next line of code. In this case, you have to write the code in sequence on both sides, because otherwise the events won't be triggered. When using a blocking connection, you must use a thread on the server, and you'll generally use a thread also on the client.

The Indy components use an in-between approach. They use blocking connections exclusively, and you can either place their code in a thread or use a special helper component (IdAntiFreeze). Using blocking connections for implementing a protocol has the advantage of simplifying the program logic, because you don't have to use the state-machine approach of nonblocking connections, as exemplified later.

Finally, when writing threaded code with the ServerSocket components working on a blocking connection, you can use the TWinSocketStream class to do the actual reading and writing operations. You can use the WaitForData method of the TWinSocketStream class to wait until the socket on the other end is ready to write. You can also create the socket stream class and specify a timeout value, so that if the connection is lost, it won't hang forever.

## Using Sockets

After all that theory, let's take a look at a couple of examples. The first is the Sock1 program on the companion CD and is made of the Server1 and Client1 applications, built with the ClientSocket and ServerSocket components in nonblocking mode. The server has a form with the following component:

```
object ServerSocket1: TServerSocket
    Active = True
    Port = 50
    ServerType = stNonBlocking
    OnClientConnect = ServerSocket1ClientConnect
    OnClientDisconnect = ServerSocket1ClientDisconnect
    OnClientRead = ServerSocket1ClientRead
end
```

All the code of the application relates to the events of this component, as the program provides no specific interaction with the user. However, the server has three list boxes for outputting the status, the messages sent from the client, and a log of the events. For example, as a client connects, the server adds the client address to the log:

```
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
    Socket: TCustomWinSocket);
begin
```

```

    lbLog.Items.Add ('Connected: ' + Socket.RemoteHost + ' (' +
        Socket.RemoteAddress + ')');
    PostMessage (Handle, wm_RefreshClients, 0, 0);
end;

```

Notice that the `OnClientConnect` event indicates the first occasion for the server to know about the connected client. Using the `Socket` property, which refers to the low-level `TCustomWinSocket`, the server can track who is trying to connect. At the end of this and other events, I want to update the list of the connections, using the `ActiveConnections` property of the server. However, in the `OnClientConnect` event handler, this list is still not updated, so I post a message to the form to delay the operation:

```

const
    wm_RefreshClients = wm_User;

procedure TForm1.RefreshClients; // message wm_RefreshClients
var
    I: Integer;
begin
    lbClients.Clear;
    for I := 0 to ServerSocket1.Socket.ActiveConnections - 1 do
        with ServerSocket1.Socket.Connections [I] do
            lbClients.Items.Add (RemoteAddress + ' (' + RemoteHost + ')');
    end;
end;

```

Similar code is executed as the client disconnects from the server:

```

procedure TForm1.ServerSocket1ClientDisconnect(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    lbLog.Items.Add ('Disconnected: ' + Socket.RemoteHost + ' (' +
        Socket.RemoteAddress + ')');
    PostMessage (Handle, wm_RefreshClients, 0, 0);
end;

```

Finally, as the client sends some information to the server (writes to the socket), the server can read the message by calling the `ReceiveText` function. You should do this read operation only when there is some data available—that is, when the `OnClientRead` event is fired. Notice also that this is a *destructive* read: the information extracted from the stream is removed from it. Here is the code:

```

procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    // read from the client

```

```
    lBMsg.Items.Add (Socket.RemoteHost + ': ' + Socket.ReceiveText);  
end;
```

Now we can move to the client side of the application, which has a form hosting a client-socket component with the following properties:

```
object ClientSocket1: TClientSocket  
    Active = False  
    Address = '127.0.0.1'  
    ClientType = ctNonBlocking  
    Port = 50  
    OnConnect = ClientSocket1Connect  
    OnDisconnect = ClientSocket1Disconnect  
end
```

The client form is more interactive. It has two edit boxes and a check box. In the first edit box, you can type the address of the server you want to connect to (to replace the default value listed above), using the check box to activate or deactivate the socket connection:

```
procedure TForm1.cbActivateClick(Sender: TObject);  
begin  
    if not ClientSocket1.Active then  
        ClientSocket1.Address := EditServer.Text;  
        ClientSocket1.Active := cbActivate.Checked;  
end;
```

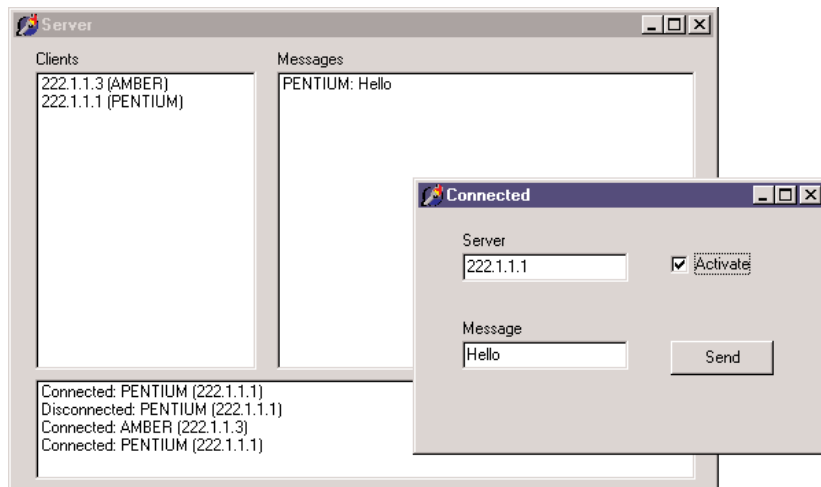
As you connect or disconnect, the program simply updates the caption of the form. In the second edit box, you can type a message to send to the server and a button you can press to send the message:

```
procedure TForm1.btnSendClick(Sender: TObject);  
begin  
    ClientSocket1.Socket.SendText (EditMsg.Text);  
end;
```

Notice that this example program doesn't check whether the connection is active before using it, which can result in errors. In Figure 21.1, you can see an example of the client and the server. As the server indicates, a second copy of the client application is running on another computer and is connected to it.

**FIGURE 21.1:**

The client and server applications of the Sock1 example, demonstrating the use of the socket components



## Using Sockets with a Custom Protocol

Unless you want to send and receive only simple text messages, you might want to define some communication rules between the client and the server. A set of communication rules is generally indicated as a protocol. Basically, the server can receive different requests and, depending on the type of request and whether it can be accomplished, reply to the client.

The server program of the Sock2 example accepts four types of requests: the listing of a directory, a bitmap file, a text file, and the execution of a program on the server. When the server sends back a file, its reply should indicate both what it is going to send back and the actual information. The only method modified from the Sock1 example is the `ServerSocket1ClientRead` procedure, which starts by extracting the five initial characters of the text received by the client that host the command:

```
strCommand := Socket.ReceiveText;
TbLog.Items.Add ('Client: ' + Socket.RemoteAddress + ': ' + strCommand);
// extract the file name (all commands have 5 characters)
strFile := Copy (strCommand, 6, Length (strCommand) - 5);
```

The actual code depends on the initial command defined by the protocol (in this case either *EXEC!* to execute a file on the server, *TEXT!* to return a text file, *BITM!* to retrieve a bitmap file, or *LIST!* to return a directory listing). Here is the code for two of these four alternatives:

```
// send back a text file
if Pos ('TEXT!', strCommand) = 1 then
begin
```



```

if FileExists (strFile) then
begin
    strFeedback := 'TEXT!';
    Socket.SendText (strFeedback);
    Socket.SendStream (TFileStream.Create (strFile,
        fmOpenRead or fmShareDenyWrite));
end
else
begin
    strFeedback := 'ERROR' + strFile + ' not found';
    Socket.SendText (strFeedback);
end;
end
// send back a directory listing
else if Pos ('LIST!', strCommand) = 1 then
begin
    if DirectoryExists (strFile) then
    begin
        strFeedback := 'LIST!';
        Socket.SendText (strFeedback);
        FileListBox1.Directory := strFile;
        Socket.SendText (FileListBox1.Items.Text);
    end
    else
    begin
        strFeedback := 'ERROR' + strFile + ' not found';
        Socket.SendText (strFeedback);
    end;
    end
else
begin
        strFeedback := 'ERROR' + 'Undefined command: ' + strCommand;
        Socket.SendText (strFeedback);
    end;
end;

```

For the directory listings, I've used an invisible FileListBox component. For sending back the text file, I've used the SendStream method, creating a new stream on the fly. The advantage is that there is no need to destroy the temporary stream, as the SendStream method becomes the owner of the stream and destroys it when it is done.

The program sends back multiple pieces of information one after the other. This will create a few problems on the client side, as all the information is received in a single stream. However, the server responds with a five-character header that we can use to determine the content of the rest of the stream. After receiving these headers, the client application sets a status field so that it knows which type of information is coming next. In other words, in the client

program, we implement a very simple finite-state machine, a typical technique for socket programming. The client application has five possible states, listed in an enumerated type:

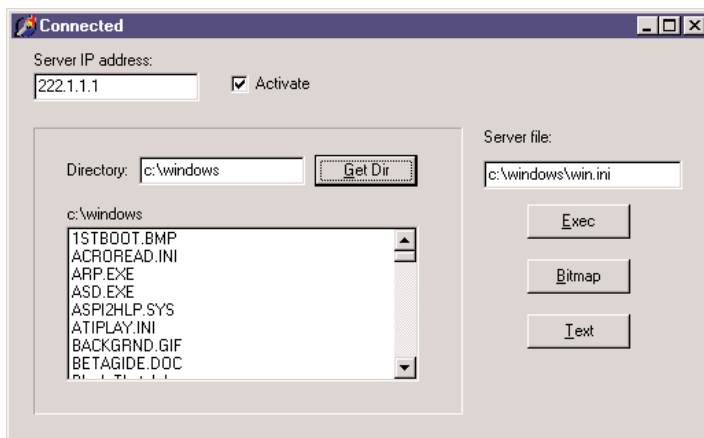
**type**

```
TCliStatus = (csIdle, csList, csBitmap, csText, csError);
```

This type is used for the `CliStatus` field of the form. The form has two edit boxes referring to a directory or a file a user can request from the server. When the user presses the `Get Dir` button, the client program passes to the server the name of the directory indicated by the first edit box. The server will return a list of files, which the client program saves in a list box. At this point, the user can select one of the files from the list box, and the client program will copy it, along with the complete path, into the second edit box. The text of this second edit box is used by the other three buttons—`Exec`, `Bitmap`, and `Text`—which send further requests to the server. In Figure 21.2, you can see an example of the main form of the client program after a directory has been retrieved.

**FIGURE 21.2:**

The form of the `Client2` program after the server has returned the list of the files of a directory



The core of the program is in the `ClientSocket1Read` method, triggered by the socket when there is data to read. The method is first used to get the header indicating which type of data is reaching the program and to set the client program to the proper status:

```
case CliStatus of
  // look for data to receive
  csIdle:
  begin
    Socket.ReceiveBuf (Buffer, 5);
    strIn := Copy (Buffer, 1, 5);
    if strIn = 'TEXT!' then
      CliStatus := csText
    else if strIn = 'BITM!' then
      CliStatus := csBitmap
    // .. and so on
```

Since we don't retrieve all the data, the event is triggered again soon afterward, and this time we are ready to get the actual data. Here are two more branches of the case statement:

```
// get a directory listing
csList:
  begin
    ListFiles.Items.Text := Socket.ReceiveText;
    cliStatus := csIdle;
  end;
// read a bitmap file
csBitmap:
  with TFormBmp.Create (Application) do
  begin
    Stream := TMemoryStream.Create;
    Screen.Cursor := crHourglass;
    try
      while True do
      begin
        nReceived := Socket.ReceiveBuf (Buffer, sizeof (Buffer));
        if nReceived <= 0 then
          Break
        else
          Stream.Write (Buffer, nReceived);
          // delay (200 milliseconds)
          Sleep (200);
        end;
        // reset and load the temporary file
        Stream.Position := 0;
        Image1.Picture.Bitmap.LoadFromStream (Stream);
      finally
        Stream.Free;
        Screen.Cursor := crDefault;
      end;
      Show;
      cliStatus := csIdle;
    end;
  end;
```

For loading the bitmap, I simply move the data to a `Buffer` (declared as array `[0..9999]` of `Char`) and then from the buffer to a memory stream, which is later loaded in the `Image` component of the secondary form. Because the data flow can slow down, the program has a hard-coded delay of 200 milliseconds every time some data is read. Unlike file-reading operations, the loop doesn't stop when the data read is less than the data requested, but only when *no* data is read. (In case of error, the value returned by the `ReceiveBuff` method is `-1`.)

## Sending Database Data over a Socket Connection

Using the techniques we've seen so far, we can write an application that moves database records over a socket. The idea will be to write a front end for data input and a back end for data storage. The client application will have a simple data-entry form and use a database table with string fields for Company, Address, State, Country, Email, and Contact, and a floating-point field for the company ID (called CompID).

### NOTE

Moving database records over a socket is exactly what you can do with DataSnap and a socket connection component, or with the SOAP support built into Delphi 6 and discussed in Chapter 23, "XML and SOAP."

The client program I've come up with works on a table with this structure saved in the current directory. (You can see the related code in the OnCreate event handler.) The core method on the client side is the handler of the OnClick event of the Send All button, which sends all the new records to the server. The new records are determined by looking to see whether the record has a valid value for the CompID field. This field, in fact, is not set up by the user but is determined by the server application when the data is sent.

For all new records, the client program packages the field information in a string list, using the structure *FieldName=Value*, obtained using the Values property of the string list. The string corresponding to the entire list is then sent to the server. At this point, the program stops in an apparently infinite loop:

```
// save database data in a string list
Data := TStringList.Create;
table1.First;
while not Table1.Eof do
begin
  // if the record is still not logged
  if Table1.CompID.IsNull or (Table1.CompID.AsInteger = 0) then
  begin
    TbLog.Items.Add ('Sending ' + Table1.Company.AsString);
    Data.Clear;
    // create strings with structure "FieldName=Value"
    for I := 0 to Table1.FieldCount - 1 do
      Data.Values [Table1.Fields[I].FieldName] := Table1.Fields [I].AsString;
    // send the record
    ClientSocket1.Socket.SendText (Data.Text);
    // wait for response
    fWaiting := True;
    while fWaiting do
      Application.ProcessMessages;
  end;
  Table1.Next;
end;
```

The program waits forever ... or until the handler of another message sets the `fWaiting` field of the form to `False`. This happens when the server sends some feedback indicating that the record was received or when the user presses the Stop button. The `btnSendAllClick` method automatically connects to the server at the beginning and disconnects at the end.

Now let us look at the server. This program has a database table, again stored in the local directory, with two new fields added to the client application's table: `LoggedBy`, a string field; and `LoggedOn`, a data field. The values of the two extra fields are determined automatically by the server as it receives data, along with the value of the `CompID` field. All these operations are done in the `ServerSocket1ClientRead` method after unpacking the data received by the client:

```
// read from the client
strCommand := Socket.ReceiveText;
// reassemble the data
Data := TStringList.Create;
try
  Data.Text := strCommand;
  // new record
  Table1.Insert;
  // set the fields using the strings
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields [I].AsString := Data.Values [Table1.Fields[I].FieldName];
  // complete with random ID, sender, and date
  Table1.CompID.AsInteger := GetTickCount;
  Table1.LoggedBy.AsString := Socket.RemoteAddress;
  Table1.LoggedOn.AsDateTime := Date;
  Table1.Post;
  // get the value to return
  strFeedback := Table1.CompID.AsString;
  // send results back
  lbLog.Items.Add (strFeedback);
  Socket.SendText (strFeedback);
finally
  Data.Free;
end;
```

Except for the fact that some data might be lost, there is no problem when fields have a different order and if they do not match, because the data is stored in the *FieldName=Field-Value* structure. After receiving all the data and posting it to the local table, the server sends back the company ID to the client. The client program, after sending the record, goes into *waiting mode*, a situation modified by receiving feedback from the server:

```
procedure TForm1.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  if fWaiting then
    begin
      Table1.Edit;
```

```

Table1CompId.AsString := Socket.ReceiveText;
Table1.Post;
TblLog.Items.Add (Table1Company.AsString + ' logged as ' +
    Table1CompId.AsString);
fWaiting := False;
end;
end;

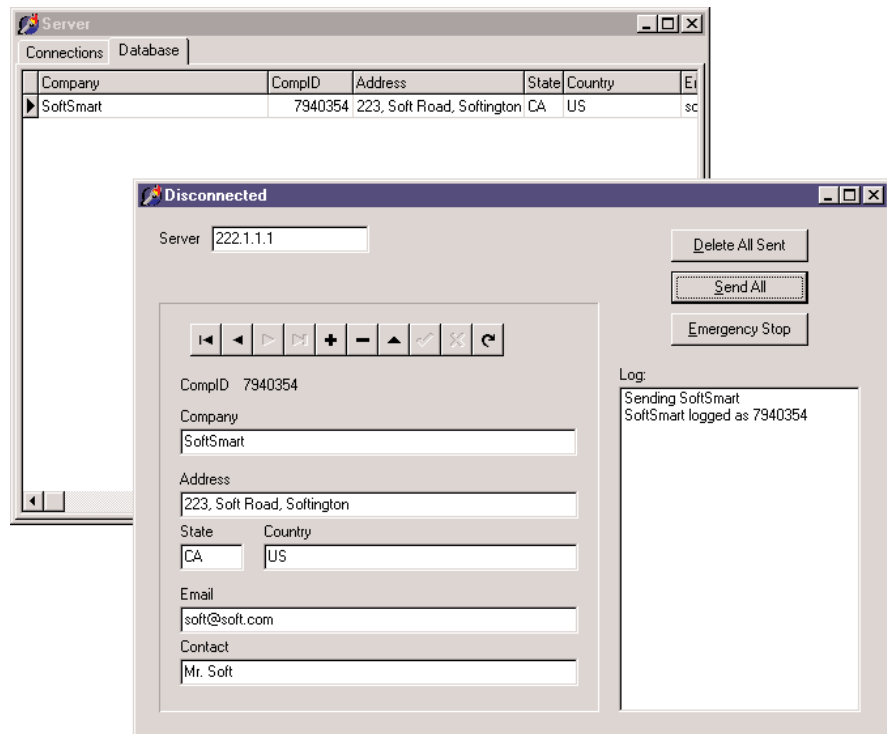
```

When receiving feedback, the client program saves the company ID, which marks the record as sent. If the user modifies the record, there is no way to send an update to the server. To accomplish this, you might add a modified field to the client database table and make the server check to see if it is receiving a new field or a modified field. With a modified field, the server should not add a new record but update the existing one.

This is one of the many additions you can make to the program, to make it usable in a real-world environment. The existing code of the program and the previous examples on sockets should provide all you need to complete a similar task. I've limited myself to this version of the application, as shown in Figure 21.3. Notice that the server program has two pages, one with the usual log and the other with a DBGrid showing the current data of the server database table.

**FIGURE 21.3:**

The client and server programs of the database socket example (DbSock)



## Working with Blocking Sockets and Threads

A program like the one I've just built is nice, but it won't really scale up on a large system, because the server uses a blocking connection and the requests are processed in sequence. Now, even if making the database-related code multithreading wouldn't be easy, I'll take the excuse of this example to show you how to build a program based on blocking sockets and threads. (If you don't know much about threads, read the sidebar "Working with Threads" before proceeding.)

### Working with Threads

Win32 has an API to allow two procedures or methods execute at the same time. Delphi provides a `TThread` class that will let us create and control threads. The first thing to know about the `TThread` class is that you never use it directly, because it is an abstract class—a class with a virtual abstract method. To use threads, you always subclass `TThread` (optionally starting with the Thread Object of the New Items dialog box (File > New > Other) and use the features of this base class.

The `TThread` class has a constructor with a single parameter (`CreateSuspended`) that lets you choose whether to start the thread immediately or suspend it until later. There are also some public synchronization methods:

```
procedure Resume;  
procedure Suspend;  
function Terminate: Integer;  
function WaitFor: Integer;
```

The published properties include `Priority`, `Suspended`, and two read-only, low-level values: `Handle` and `ThreadID`. The class also provides a protected interface, which includes two key methods for your thread subclasses:

```
procedure Execute; virtual; abstract;  
procedure Synchronize(Method: TThreadMethod);
```

The `Execute` method, declared as a virtual abstract procedure, must be redefined by each thread class. It contains the main code of the thread, the code you would typically place in a *thread function* when using the Windows API.

The `Synchronize` method is used to avoid concurrent access to VCL components. The VCL code runs inside the main thread of the program, and you need to synchronize access to VCL to avoid reentry problems (errors from reentering a function before a previous call is completed) and concurrent access to shared resources. The only parameter of `Synchronize` is a method that accepts no parameters, typically a method of the same thread class. As you cannot pass parameters to this method, it is common to save some values within the data of the thread object in the `Execute` method and use those values in the *synchronized* methods.

*Continued on next page*

Another way to avoid conflicts is to use the synchronization techniques offered by the operating system. The SyncObjs unit defines VCL classes for some of these low-level synchronization objects: events (with the `TEvent` class and the `TSingleEvent` class) and critical sections (with the `TCriticalSection` class).

In the server program (see the `SockDbThread` example on the companion CD), the socket component now has a thread-blocking type and has handlers for many events, including in particular `OnGetThread`. No methods are hooked to the read and write events, as they won't be triggered anymore (they are used exclusively by message-based nonblocking sockets):

```
object ServerSocket1: TServerSocket
  Active = True
  Port = 51
  ServerType = stThreadBlocking
  OnAccept = ServerSocket1Accept
  OnGetThread = ServerSocket1GetThread
  OnClientConnect = ServerSocket1ClientConnect
  OnClientDisconnect = ServerSocket1ClientDisconnect
end
```

The `OnAccept`, `OnClientConnect`, and `OnClientDisconnect` event handlers are used only for logging information to the screen, while the `OnGetThread` event handler has a key role of creating the server-side thread object:

```
procedure TForm1.ServerSocket1GetThread(Sender: TObject; ClientSocket:
  TServerClientWinSocket; var SocketThread: TServerClientThread);
begin
  TLog.Items.Add('GetThread: ' + ClientSocket.RemoteHost + ' (' +
    ClientSocket.RemoteAddress + ')');
  SocketThread := TDbServerThread.Create(False, ClientSocket);
end;
```

This must be an object of a class inherited by the specific `TServerClientThread` class (not the generic Delphi `TThread` class) and with its core code placed in an overridden `ClientExecute` method (not the generic `Execute` method):

```
type
  TDbServerThread = class(TServerClientThread)
private
  strCommand: string;
  strFeedback: string;
public
  procedure ClientExecute; override;
  procedure Log;
  procedure LogFeedback;
```



```
    procedure AddRecord;
end;

procedure TDbServerThread.ClientExecute;
var
    Stream: TWInSocketStream;
    Buffer, strIn: string;
    nRead: Integer;
begin
    // keep going
    Stream := TWInSocketStream.Create(ClientSocket, 5000);
    try
        while not Terminated and ClientSocket.Connected do
            begin
                // initialize (thread might be reused)
                Buffer := '';
                strIn := '';
                SetLength(Buffer, 64);
                repeat
                    nRead := Stream.Read(Buffer[1], 64);
                    if nRead = 0 then
                        begin
                            ClientSocket.Close;
                            Break;
                        end;
                    SetLength(Buffer, nRead);
                    StrIn := StrIn + Buffer;
                until (Pos(#10#13', '#10#13', Buffer) > 0);

                if strIn = '' then
                    Continue // keep going
                else
                    begin
                        // handle the request, if anything arrived
                        StrCommand := Copy(strIn, 1, Pos(#10#13', '#10#13', strIn) - 1);
                        Synchronize(Log);
                        Synchronize(AddRecord);
                        // send results back
                        Synchronize(LogFeedback);
                        Stream.Write(strFeedback[1], Length(strFeedback));
                    end;
                end;
            finally
                Stream.Free;
            end;
        end;
    end;
```

The server reads data from the client and sends the feedback using a `TWinSocketStream`, a compulsory approach for blocking servers. The thread is kept active, as it can be reused for subsequent calls, and reads from a buffer until it find a specific separator (in this case, a dot between two line separators, exactly as occurs in the SMTP protocol). When the data is received, the server does a *synchronized* call to the `AddRecord` method, which is similar to the code of the previous version of the example.

### WARNING

This approach is far from perfect, as all database accesses are serialized, but you could solve the problem by moving the database access component within the thread and adding a `Session` object in case of a BDE application. Not all databases like multithreaded access, though, so serializing the calls is not always a bad idea.

This is the multithreaded server application. The client program, instead, uses a standard thread class, derived from `TThread`. The class creates a client server object internally, so that multiple threads could spawn multiple socket connections in parallel (something the program doesn't really use) and receives a table to work on as parameter in the constructor:

```

type
  TLogEvent = procedure(Sender: TObject; LogMsg: String) of object;

  TSendThread = class(TThread)
  private
    ClientSocket: TClientSocket;
    FTable: TTable;
    FOnLog: TLogEvent;
    FLogMsg: String;
    procedure SetOnLog(const Value: TLogEvent);
  protected
    procedure Execute; override;
    procedure DoLog;
  public
    constructor Create(ATable: TTable);
    property OnLog: TLogEvent read FOnLog write SetOnLog;
  end;

```

On the client side, the finite-state machine logic (send, set the wait flag, receive another event, disable the flag, get the next record) is now replaced by a continual and more logical flow of operations, all part of the `Execute` method of the thread. You need to add some code to let the program wait until the server sends a reply. Again, the blocking socket of the client is not used directly but via a `TWinSocketStream` object:

```

procedure TSendThread.Execute;
var
  I: Integer;
  Data: TStringList;

```

```

Stream: TwinSocketStream;
Buf: String;
begin
  try
    Data := TStringList.Create;
    ClientSocket := TClientSocket.Create (nil);
    Stream := nil;
    try
      ClientSocket.Address := EditServer.Text;
      ClientSocket.ClientType := ctBlocking;
      ClientSocket.Port := 51;
      ClientSocket.Active := True;
      Stream := TwinSocketStream.Create(ClientSocket.Socket, 30000);

      FTable.First;
      while not FTable.Eof do
        begin
          // if the record is still not logged
          if FTable.FieldByName('CompID').IsNull or
            (FTable.FieldByName('CompID').AsInteger = 0) then
            begin
              FLogMsg := 'Sending ' + Table.FieldByName('Company').AsString;
              Synchronize(DoLog);
              Data.Clear;
              // create strings with structure "FieldName=Value"
              for I := 0 to FTable.FieldCount - 1 do
                Data.Values [FTable.Fields[I].FieldName] :=
                  FTable.Fields [I].AsString;
              // send the record followed by separator
              Buf := Data.Text + #10#13'. #10#13;
              ClientSocket.Socket.SendText(Buf);
              // wait for reponse
              if Stream.WaitForData(30000) then
                begin
                  FTable.Edit;
                  SetLength(Buf, 256);
                  SetLength(Buf, Stream.Read(Buf[1], Length(Buf)));
                  FTable.FieldByName('CompID').AsString := Buf;
                  FTable.Post;
                  FLogMsg := FTable.FieldByName('Company').AsString +
                    ' logged as ' + FTable.FieldByName('CompID').AsString;
                end
              else
                FLogMsg := 'No response for ' +
                  FTable.FieldByName('Company').AsString;
              Synchronize(DoLog);
            end;
          end;
        end;
      end;
    end;
  end;
end;

```

```
        FTable.Next;
    end;
finally
    ClientSocket.Active := False;
    ClientSocket.Free;
    Stream.Free;
    Data.Free;
end;
except
    // trap exceptions
end;
end;
```

The thread also has an event handler to let the forms using it define the effect of the OnLog operation, in the synchronized DoLog method. Finally, this is how the thread starts, when the user clicks a button in the form:

```
procedure TForm1.Button2Click(Sender: TObject);
var
    SendThread: TSendThread;
begin
    SendThread := TSendThread.Create(Table1);
    SendThread.OnLog := OnLog;
    SendThread.Resume;
end;
```

## Internet Protocols

After discussing the low-level socket components, we are ready to delve into the core topic of this chapter, the use of higher-level Internet protocols.

As already mentioned, Delphi 6 now ships with a collection of open-source Internet components called Internet Direct, or Indy for short. The Indy components, previously called WinShoes (a pun on the term WinSockets), are built by a group of developers led by Chad Hower and are available also in Kylix. You can find more information and possibly updated versions of the actual components at [www.nevrona.com/indy](http://www.nevrona.com/indy).

The Indy components are available within the Delphi IDE, but they are not the only set of Internet components. Delphi 6 Component Palette also has another page of Internet protocol components, the FastNet page. These are available for compatibility with Delphi 5 and earlier versions. Third-party solutions, both freely available or for sale, also provide implementations of Internet protocols.

Here and in the next chapter, I'm going to focus exclusively on the Indy components. This chapter focuses on the use of Internet protocols within Windows applications, while the next shows examples of the use of these protocols within Web server applications.

## Sending and Receiving Mail

Probably the most common operation you do on the Internet is to send and receive e-mail. There is generally very little need to write a complete application to handle e-mail, as some of the existing programs are actually rather complete. For this reason, I have no intention of writing a general-purpose mail program here. You can find some examples of those among Delphi Internet demos.

Other than creating a general-purpose mail application, what else can one do with the mail components and protocols? There are many possibilities, which I've tried to group in two areas:

**Automatic generation of mail messages** An application you've written can have an About box for sending a registration message back to your marketing department or a specific menu item for sending a request to your tech support. You might even decide to enable a tech-support connection whenever an exception occurs. Another related task would be automating the dispatching of a message to a list of people or generating an automatic message from your Web site, an example I'll show you toward the end of this chapter.

### **Use of mail protocols for communication with users who are only occasionally online**

When you must move data between users who are not always online, you can write an application on a server to synchronize among them, and you can give each user a specialized client application for interacting with the server. An alternative is to use an existing server application, such as a mail server, and write the two specialized programs based on the mail protocols. The data sent over this connection will generally be formatted in special ways, so you'll want to use a specific e-mail addresses for these messages (not your primary e-mail address). As an example, you could rewrite the earlier DbSock example to dispatch mail messages instead of using a custom socket connection. This will give you the advantage of being firewall-friendly and allowing the server to be temporarily offline, as the requests would be kept on the mail server.

## Sending Messages with Your Mail Program

The simplest technique for automating the generation of an e-mail message is to use your existing mail application, adding a message to its outbox. Using the ShellExecute API function, you can easily send a message to the default mail program registered on the computer.

To test this technique, I've prepared a simple form with two edit boxes and a memo for the input. Pressing a button creates a string with all the information about the message and then sends it, simply executing the string with the *mailto:* prefix. Here is the code of the Send button of the MailGen example from the companion CD:

```
uses
    ShellApi;

procedure TForm1.BtnSendClick(Sender: TObject);
var
    strMsg: string;
    I: Integer;
begin
    // set the basic information
    strMsg := 'mailto:' + EditAddress.Text + '?Subject=' + EditSubject.Text +
        '&Body=';
    // add first line
    if Memo1.Lines.Count > 0 then
        strMsg := strMsg + Memo1.Lines [0];
    // add subsequent lines separated by the newline symbol
    for I := 1 to Memo1.Lines.Count - 1 do
        strMsg := strMsg + '%0D%0A' + Memo1.Lines [I];
    // send the message
    ShellExecute (Handle, 'open', pChar (strMsg), '', '', SW_SHOW);
end;
```

To show the body of the message on multiple lines, you can separate each line with the carriage return and line feed characters (usually indicated in Delphi as #13 and #10). These values should be explicitly added to the string in hexadecimal format and prefixed by the % sign, as required by a URL. You can actually obtain this encoding automatically by using the NMURL component.

**NOTE**

You can also send mail with the `TSendMail` predefined action, which is based on the MAPI standard.

## Mail In and Out

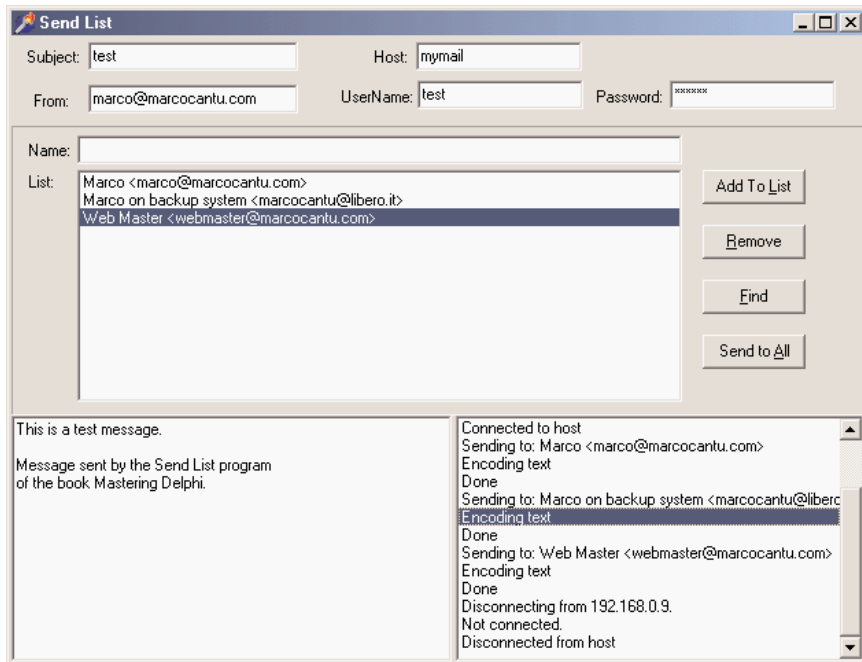
To showcase the development of simple e-mail management programs, I could build an example of how you can send and receive mail. The Indy components, though, include a rather complete set of examples, and I don't see any reason to duplicate those, as using the mail protocols means placing a message component (`IdMessage`) in your application, filling it with data, and then using the `IdSMTP` component to send the mail message. To *retrieve* a mail message from your mailbox, use the `IdPop3` component, which will return you an `IdMessage` object.

Just to give you an idea of how this works, I've written a program for sending mail to multiple people at once, using a list stored in an ASCII file. I originally used this program myself for sending mail to people who sign up on my Web site, but later I extended the program by adding database support and reading subscriber logs automatically. The original version of the program is still a good introduction to the use of the SMTP component of Indy.

The SendList program keeps a list of names and e-mail addresses in a local file, which is displayed in a list box. A few buttons allow you to add and remove items, or modify them by removing the item, editing it, and then adding the item again. When the program closes, the updated list is automatically saved. Now let's get to the interesting portion of the program. The top-most panel, shown in Figure 21.4, allows you to enter the subject, the sender address, and the information used to connect to the mail server (hostname, username, and eventually a password).

**FIGURE 21.4:**

The SendList program in action



You'll probably want to make the value of these edit boxes persistent, possibly in an INI file. I haven't done this, only because I don't really want you to see my mail connection

details! The value of these edit boxes, along with the list of addressee, allows you to send the series of mail messages, after customizing each of them, with the following code:

```

procedure TMainForm.BtnSendAllClick(Sender: TObject);
var
    nItem: Integer;
    Res: Word;
begin
    Res := MessageDlg ('Start sending from item ' +
        IntToStr (ListAddr.ItemIndex) + ' (' +
        ListAddr.Items [ListAddr.ItemIndex] + ')?'#13 +
        '(No starts from 0)', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    if Res = mrCancel then
        Exit;
    if Res = mrYes then
        nItem := ListAddr.ItemIndex
    else
        nItem := 0;
    // connect
    Mail.Host := eServer.Text;
    Mail.UserID := eUserName.Text;
    if ePassword.Text <> '' then
        begin
            Mail.Password := ePassword.Text;
            Mail.AuthenticationType := atLogin;
        end;
    Mail.Connect;
    // send the messages, one by one, prepending a custom message
    try
        // set the fixed part of the header
        MailMessage.From.Name := eFrom.Text;
        MailMessage.Subject := eSubject.Text;
        MailMessage.Body.SetText (reMessageText.Lines.GetText);
        MailMessage.Body.Insert (0, 'Hello');
        while nItem < ListAddr.Items.Count do
            begin
                // show the current selection
                Application.ProcessMessages;
                ListAddr.ItemIndex := nItem;
                MailMessage.Body [0] := 'Hello ' + ListAddr.Items [nItem];
                MailMessage.Recipients.EmailAddresses := ListAddr.Items [nItem];
                Mail.Send(MailMessage);
                Inc (nItem);
            end;
        finally // we're done
            Mail.Disconnect;
        end;
    end;
end;

```



Another interesting example of the use of the mail is to notify developers of problems within applications, something you might want to use more in an internal application than in one you'll distribute widely. You can obtain this effect by modifying the `ErrorLog` example of Chapter 4, "The Run-Time Library," and sending mail when an exception (or one of a given type only) occurs.

## Working with HTTP

Handling mail messages is certainly interesting, and mail protocols are probably still the most widespread Internet protocols. The other popular protocol is HTTP, the one used by Web servers and Web browsers. This is the protocol to which we'll devote the rest of this chapter and all of the following.

On the client side of the Web, the main activity is browsing—reading HTML files. Besides building a custom browser, you can embed the Internet Express ActiveX control within your program (as I've done in `WebDemo` example in Chapter 20, "From Automation to COM+"). You can also directly activate the browser installed on the computer of the user, for example, opening an HTML page by calling the `ShellExecute` method (defined in the `ShellApi` unit):

```
ShellExecute(Handle, 'open', FileName, '', '', sw_ShowNormal);
```

Using `ShellExecute`, we can simply execute a document, such as a file. Windows will start the program associated with the `HTM` extension, using the action passed as the parameter (in this case, `open`). You can use a similar call to view a Web site, by using a string like `'http://www.example.com'` instead of a filename. In this case, the system recognizes the `http` section of the request as requiring a Web browser and launches it.

On the server side, you generate and make available the HTML pages. At times, it may be enough to have a way to produce static pages, occasionally extracting new data from a database table to update the HTML files as needed. In other cases, you'll need to generate pages dynamically based on a request from a user.

As a starting point, I'll discuss HTTP by building a simple but complete client and server, then we'll move on to discussing HTML producer components and introducing the Web server extension technologies (CGI and ISAPI). In the next chapter, we'll move from this "core technology" level to the RAD development style for the Web supported by Delphi 6, discussing the `WebBroker` and `WebSnap` architectures.

## Grabbing HTTP Content

As an example of the use of the HTTP protocols, I've decided to write a very specific search application. The program simply hooks onto the Google Web site, searches for a keyword,

and retrieves the first hundred sites found. Instead of showing the resulting HTML file, the program parses it to extract only the URLs of the related sites to a list box. The description of these sites is kept in a separate string list and is displayed as you click a list-box item. So the program demonstrates two techniques at once: retrieving a Web page and parsing its HTML code.

To demonstrate how you should work with blocking connections, such as those used by Indy, I've implemented the program using a background thread for the actual processing. (See the sidebar "Working with Threads," earlier in this chapter, for a very short introduction to this topic.) This approach also gives the advantage of being able to start multiple searches at once. The thread class used by the WebFind application receives as input a URL to look for, `strUrl`.

The class has two output procedures, `AddToList` and `ShowStatus`, to be called inside the `Synchronize` method. The code of these two methods sends some results or some feedback to the main form, respectively adding a line to the listbox and changing the status bar `SimpleText` property. The key method of the thread is the `Execute` method. Before we look at it, however, let me show you how the thread is activated by the main form:

```
const
    strSearch = 'http://www.google.com/search?as_q=';

procedure TForm1.BtnFindClick(Sender: TObject);
var
    FindThread: TFindWebThread;
begin
    // create suspended, set initial values, and start
    FindThread := TFindWebThread.Create (True);
    FindThread.FreeOnTerminate := True;
    // grab the first 100 entries
    FindThread.strUrl := strSearch + EditSearch.Text + '&num=100';
    FindThread.Resume;
end;
```

The URL string is made of the main address of the search engine, followed by some parameters. The first, `as_q`, indicates the words you are looking for. The second, `num=100`, indicates the number of sites to retrieve; you cannot use numbers at will but are limited to few alternatives, with 100 being the largest possible value.

**WARNING**

The WebFind program works with the server on the Google Web site at the time this book was written and tested. The custom software on the site can change any day, however, which might prevent WebFind from operating correctly.

The Execute method of the thread, activated by the Resume call, simply calls the two methods actually doing the work and shown in Listing 21.1. In the first, GrabHtml, the program connects to the HTTP server using a dynamically created IdHttp component, and reads the HTML with the result of the search. The second method, HtmlToList, extracts the URLs referring to other Web sites from the result, the strRead string.



### Listing 21.1: The *TFindWebThread* class (of the WebFind program)

```

unit FindTh;

interface

uses
  Classes, IdComponent, SysUtils, IdHTTP;

type
  TFindWebThread = class(TThread)
  protected
    Addr, Text, Status: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (Sender: TObject; AWorkMode: TWorkMode;
      const AWorkCount: Integer);
  public
    strUrl: string;
    strRead: string;
  end;

implementation

{ TFindWebThread }

uses
  WebFindF;

procedure TFindWebThread.AddToList;
begin
  if Form1.ListBox1.Items.IndexOf (Addr) < 0 then
  begin
    Form1.ListBox1.Items.Add (Addr);
    Form1.DetailsList.Add (Text);
  end;
end;

procedure TFindWebThread.Execute;
begin

```

```

    GrabHtml;
    HtmlToList;
    Status := 'Done with ' + StrUrl;
    Synchronize (ShowStatus);
end;

procedure TFindWebThread.GrabHtml;
var
    Http1: TIdHTTP;
begin
    Status := 'Sending query: ' + StrUrl;
    Synchronize (ShowStatus);
    Http1 := TIdHTTP.Create (nil);
    try
        Http1.OnWork := HttpWork;
        strRead := Http1.Get (StrUrl);
    finally
        Http1.Free;
    end;
end;

procedure TFindWebThread.HtmlToList;
var
    strAddr, strText: string;
    nText: integer;
    nBegin, nEnd: Integer;
begin
    Status := 'Elaborating data for: ' + StrUrl;
    Synchronize (ShowStatus);
    strRead := LowerCase (strRead);
    repeat
        // find the initial part HTTP reference
        nBegin := Pos ('href=http', strRead);
        if nBegin <> 0 then
            begin
                // get the remaining part of the string, starting with 'http'
                strRead := Copy (strRead, nBegin + 5, 1000000);
                // find the end of the HTTP reference
                nEnd := Pos ('>', strRead);
                strAddr := Copy (strRead, 1, nEnd - 1);
                // move on
                strRead := Copy (strRead, nEnd + 1, 1000000);
                // add the URL if 'google' is not in it
                if Pos ('google', strAddr) = 0 then
                    begin
                        nText := Pos ('</a>', strRead);
                        strText := copy (strRead, 1, nText - 1);
                        // remove cached references and duplicates
                        if (Pos ('cached', strText) = 0) then
                            begin
                                Addr := strAddr;

```

```

        Text := strText;
        AddToList;
    end;
end;
end;
until nBegin = 0;
end;

procedure TFindWebThread.HttpWork(Sender: TObject; AWorkMode: TWorkMode;
    const AWorkCount: Integer);
begin
    Status := 'Received ' + IntToStr (AWorkCount) + ' for ' + strUrl;
    Synchronize (ShowStatus);
end;

procedure TFindWebThread.ShowStatus;
begin
    Form1.StatusBar1.SimpleText := Status;
end;

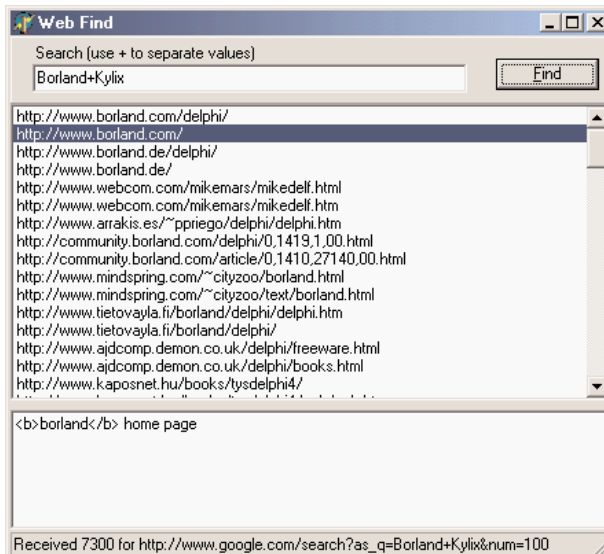
end.

```

The program looks for subsequent occurrences of the href="http substring, copying the text up to the closing > character. If the found string contains the word *google*, or its target text includes the word *cached*, it is omitted from the result. You can see the effect of this code in the output of Figure 21.5. Notice that I've already gotten the result of a request, but the program is currently retrieving another page, as indicated in the status bar. You can start multiple searches at the same time, but be aware that the results will all be added to the same memo component.

**FIGURE 21.5:**

The WebFind application can be used to search for a list of sites on the Google search engine.



## The WinInet API

When you need to use the FTP and HTTP protocols, as alternatives to using particular VCL components, you can use a specific API provided by Microsoft in the WinInet DLL. This library is part of the core operating system and implements the FTP and HTTP protocols on top of the Windows sockets API.

With just three calls—`InternetOpen`, `InternetOpenURL`, and `InternetReadFile`—you can retrieve a file corresponding to any URL and store a local copy or analyze it. Other simple methods can be used for FTP; I suggest you look for the source code of the Delphi unit, listing all the functions, and for the specific Help file for the DLL, which is not part of the SDK Help shipping with Delphi.

The `InternetOpen` function establishes a generic connection and returns a handle you can use in the `InternetOpenURL` call. This second call returns a handle to the URL that you can pass to the `InternetReadFile` function in order to read blocks of data. In the following sample code, the data is stored in a local string. When all the data has been read, the program closes the connection to the URL and the Internet session by calling the `InternetCloseHandle` function twice.

```
var
  hHttpSession, hReqUrl: HInternet;
  Buffer: array [0..1023] of Char;
  nRead: Cardinal;
  strRead: string;
  nBegin, nEnd: Integer;
begin
  strRead := '';
  hHttpSession := InternetOpen ('FindWeb', INTERNET_OPEN_TYPE_PRECONFIG,
    nil, nil, 0);
  try
    hReqUrl := InternetOpenURL (hHttpSession, PChar(StrUrl), nil, 0,0,0);
    try // read all the data
      repeat
        InternetReadFile (hReqUrl, @Buffer, sizeof (Buffer), nRead);
        strRead := strRead + string (Buffer);
      until nRead = 0;
    finally
      InternetCloseHandle (hReqUrl);
    end;
  finally
    InternetCloseHandle (hHttpSession);
  end;
end;
```

## Browsing on Your Own

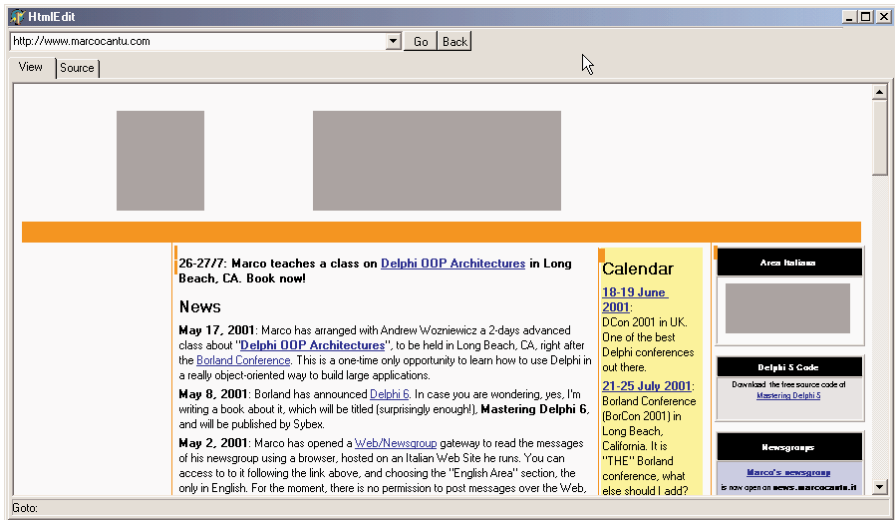
Although I doubt you are interested in writing a new Web browser, it might be interesting anyway to see how you can grab an HTML file from the Internet and display it locally, using the HTML viewer available in CLX (the TextBrowser control). Connecting this control to an Indy HTTP client, you can come up with a simplistic text-only browser with limited navigation in minutes. The core is to write

```
TextBrowser1.Text := IdHttp1.Get (NewUr1);
```

where `NewUr1` is complete location of the Web resource you want to access to. In the BrowseFast example on the CD-ROM, this URL is entered in a combo box, which keeps track of recent requests. The effect of a similar call is to return the textual portion of a Web page (see Figure 21.6), as grabbing the graphic content requires much more complex coding. The TextBrowser control, in fact, is better defined as a local file viewer than as a browser.

**FIGURE 21.6:**

The output of the BrowseFast text-only browser



In any case, I've added to the program only very limited support for hyperlinks. When a user moves the mouse over a link, its link text is copied to a local variable (`NewRequest`), which is then used in case of a click on the control to compute the new HTTP request to forward. Merging the current address (`LastUr1`) with the request, though, is far from trivial, even with the help of the `IdUr1` class provided by Indy. Here is the code I've come up with, which handles only the simplest cases:

```
procedure TForm1.TextBrowser1Click(Sender: TObject);
var
  Uri: TIdUri;
```

```

begin
  if NewRequest <> '' then
    begin
      Uri := TIdUri.Create (LastUrl);
      if Pos ('http:', NewRequest) > 0 then
        GoToUrl (NewRequest)
      else if NewRequest [1] = '/' then
        GoToUrl ('http://' + Uri.Host + NewRequest)
      else
        GoToUrl ('http://' + Uri.Host + Uri.Path + NewRequest);
    end;
  end;
end;

```

Again, this example is really trivial and is far from usable, but building a browser involves a *little* more than the ability to connect via HTTP and display HTML files.

## A Simple HTTP Server

The situation with the development of an HTTP server is quite different. Building a server to deliver static pages based on HTML files is far from simple, although one of the Indy demos provides a rather good starting point for this. A custom HTTP server, instead, might be interesting when building a totally dynamic site, something I'll focus on in more detail in the next chapter.

To show you how you can start the development of a custom HTTP server, I've built the `HttpServ` example. This program has a form with a list box used for logging requests and an `IdHTTPServer` component, with these settings:

```

object IdHTTPServer1: TIdHTTPServer
  Active = True
  DefaultPort = 8080
  OnCommandGet = IdHTTPServer1CommandGet
end

```

The server uses the port 8080 instead of the standard port 80, so that you can run it alongside another Web server. All of the custom code is in the `OnCommandGet` event handler, which simply returns a fixed page plus some information about the request itself:

```

procedure TForm1.IdHTTPServer1CommandGet(AThread: TIdPeerThread;
  RequestInfo: TIdHTTPRequestInfo; ResponseInfo: TIdHTTPResponseInfo);
var
  HtmlResult: String;
begin
  // log
  Listbox1.Items.Add (RequestInfo.Document);
  // respond
  HtmlResult := '<h1>HttpServ Demo</h1>' +

```



```

'<p>This is the only page you'll get from this example.</p><hr>' +
'<p>Request: ' + RequestInfo.Document + '</p>' +
'<p>Host: ' + RequestInfo.Host + '</p>' +
'<p>Params: ' + RequestInfo.UnparsedParams + '</p>' +
'<p>The headers of the request follow: <br>' +
RequestInfo.Headers.Text + '</p>';
ResponseInfo.ContentText := HtmlResult;
end;

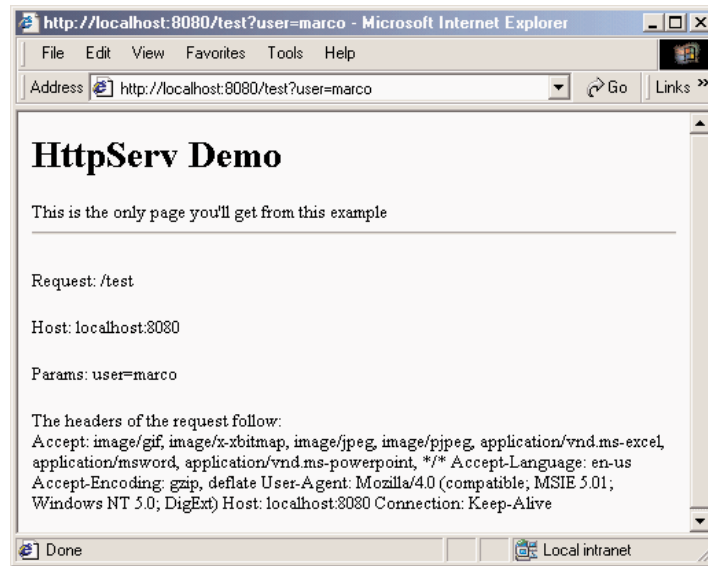
```

By passing a path and some parameters in the command line of the browser, you'll see them reinterpreted and displayed. For example, Figure 21.7 shows the effect of the command line:

```
http://localhost:8080/test?user=marco
```

**FIGURE 21.7:**

The page displayed by connecting a browser to the custom HttpServ program



If this example seems too trivial, you'll see a slightly more interesting version in the next section, as I discuss the generation of HTML with Delphi's producer components.

#### NOTE

If you plan building an advanced Web server or other Internet servers with Delphi, then as an alternative to the Indy components, have a look at the DXSock components from Brain Patchwork DX ([www.dxsock.com](http://www.dxsock.com)).

# Generating HTML

The Hypertext Markup Language, better known by its acronym HTML, is the most widespread format for content on the Web. HTML is the format Web browsers typically read; it is a standard defined by the W3C, the World Wide Web Consortium, which is one of the bodies controlling the Internet. The current standard is represented by HTML 4, although not all browsers fully support that. When building a Web site, you always need to choose a lowest-common-denominator approach to support most of the browsers in use—that is, unless you are targeting a specific group of users whom you ask to adopt a specific browser (as happens in intranet situations). If you don't know much about the tags included in HTML files, you may want to read the sidebar “The Format of HTML Files” for a fast introduction.

## The Format of HTML Files

If you have a little familiarity with HTML but don't work with it often enough to have all the basic elements “down cold,” here's a quick summary.

HTML files are basically ASCII text files. Besides plain text, an HTML file contains many tags, which might determine the style of the font, the type of paragraph, or a link to another HTML file or an image, among other things.

Most tags are paired as opening tags and closing tags (the closing tag is usually the same as the opening tag but is preceded by a slash, /) to indicate where the style or content begins and ends. For example, you write `<b>important</b>` to set the word *important* in bold, and you write `<title>Document Title</title>` to define the title of a document as *Document Title*. (A few elements, such as `<br>` for a line break and `<img>` for a graphic or “image,” stand alone and do not use a matching closing tag.)

An HTML document begins with the `<html>` tag and is divided into two parts, marked as `<head>` and `<body>`. Each of these three tags requires the corresponding terminator. In the head portion of the HTML file, you'll generally write the title (often displayed in the title bar of the browser) and a few other generic elements.

In the body, you write the contents of the file, generally starting with its visible title. You can use headings with different levels, marked with the `<hX>` tag, where you'd replace X with a number from 1 to 6. These are followed by plain paragraphs (`<p>`), preformatted paragraphs (`<pre>`, a style generally used for program listings), various types of lists, and many other elements. The text will often have links to other pages or other parts of the current page, using the `<a>` (“anchor”) tag.

Another relevant element of HTML is tables. The `<table>` and `</table>` tags indicate the beginning and the end of the table, and its optional `border` attribute displays borders with a given width. The `<tr>` and `</tr>` tags introduce and close each row, and the tags

*Continued on next page*

`<th>...</th>` and `<td>...</td>` indicate a table header cell and a table data cell, respectively. The number of columns depends on the items in each row. Different rows, in fact, can have different numbers of items.

HTML was recently refined by the W3C to be more consistent, flexible, and interoperable with advanced systems such as XML; the new version is named XHTML (Extensible HTML). HTML and XHTML are the subject of many books, and you can find dozens of tutorials on them just by browsing the Web. A good, complete source on the topic is *Mastering XHTML* by Tittel et al. (Sybex, 2001).

## Delphi's HTML Producer Components

If your version of Delphi includes the HTML producer components (available on the Internet page of the Component Palette), you can use them to generate the HTML files and particularly to turn a database table into an HTML table. Many developers believe that the use of these components makes sense only when writing a Web server extension. Although they were introduced for this purpose and are part of the WebBroker technology, you can still use three out of the four producer components in any application in which you must generate a static HTML file.

Before looking at the `HtmlProd` example, which demonstrates the use of these HTML producer components, let me summarize their role:

- The simplest of the HTML producer components is the `PageProducer`, which manipulates an HTML file in which you've embedded special tags. The advantage of this approach is that you can generate such a file using the HTML editor you prefer. At run time, the `PageProducer` converts the special tags to actual HTML code, giving you a straightforward method for modifying sections of an HTML document. The special tags have the basic format `<#tagname>`, but you can also supply named parameters within the tag. You'll process the tags in the `OnTag` event handler of the `PageProducer`.
- The `DataSetPageProducer` extends the `PageProducer` by automatically replacing tags corresponding to field names of a connected data source.
- The `DataSetTableProducer` component is generally useful for displaying the contents of a table, query, or other dataset. The idea is to produce an HTML table from a dataset, in a simple yet flexible way. The component has a very nice preview, so you can see how the HTML output will look in a browser directly at design time.
- The `QueryTableProducer` is similar to the previous one (it is actually a subclass), but it's specifically tailored for building parametric queries based on input from an HTML

search form. For this reason, I'll delay the coverage of this component to the next chapter.

## Producing HTML Pages

A very simple example of using tags is creating an HTML file that displays fields with the current date or a date computed relative to the current date, such as an expiration date. If you examine the HtmlProd example, you'll find the following component in the main form:

```
object PageProducer1: TPageProducer
    HTMLDoc.Strings = (...)
    OnHTMLTag = PageProducer1HTMLTag
end
```

The source HTML can be specified using an external file (with the advantage that you can edit it without having to recompile the application using it) or a string list, stored in the HTMLDoc property. This is a plain HTML file that might contain a few special tags introduced by the # symbol:

```
<html>
<head>
<title>Producer Demo</title>
</head>
<body>
<h1>Producer Demo</h1>
<p>This is a demo of the page produced by the <b><#appname></b> application on
<b><#date></b>.</p>
<hr>
<p>The prices in this catalog are valid until <b><#expiration
days=21></b>.</p>
</body>
</html>
```

### WARNING

If you prepare this file with an HTML editor (something I suggest you do), it might automatically place quotes around tag parameters, as in `days="21"`, because this is required by HTML 4 and XHTML 1. The PageProducer component has a `StripParamQuotes` property, which can be activated to remove those extra quotes when the component parses the code (before calling the `OnHTMLTag` event handler).

The Demo Page button simply copies the PageProducer component's output to the Text of a Memo with the statement

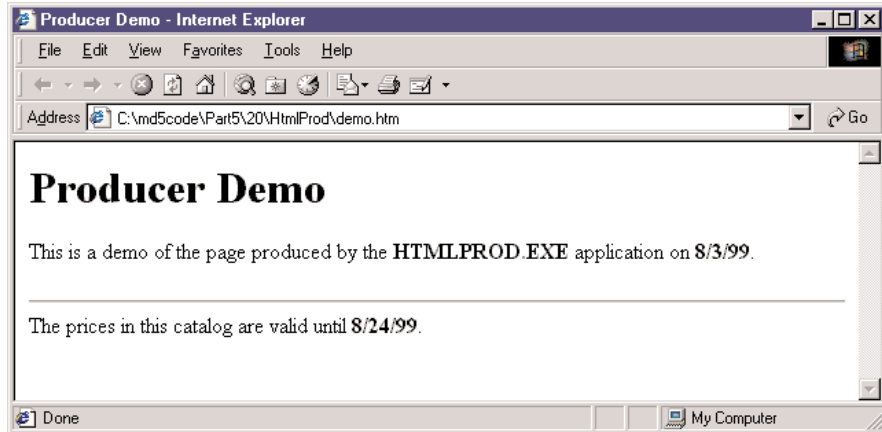
```
Memo1.Text := PageProducer1.Content;
```

As you call the `Content` function of the PageProducer component, it reads the input HTML code, parses it, and triggers the `OnTag` event handler for every special tag. In this method, we

check the value of the tag (passed in the TagString parameter) and return a different HTML text (in the ReplaceText reference parameter), producing the output of Figure 21.8.

**FIGURE 21.8:**

The output of the HtmlProd example, a simple demonstration of the Page-Producer component, when the user clicks the Demo Page button



```

procedure TFormProd.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
var
  nDays: Integer;
begin
  if TagString = 'date' then
    ReplaceText := DateToStr (Now)
  else if TagString = 'appname' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'expiration' then
    begin
      nDays := StrToIntDef (TagParams.Values['days'], 0);
      if nDays <> 0 then
        ReplaceText := DateToStr (Now + nDays)
      else
        ReplaceText := '<i>{expiration tag error}</i>';
    end;
  end;
end;

```

Notice, in particular, the code we've written to convert the last tag, #expiration, which requires a parameter. The PageProducer places the entire text of the tag parameter (in this case, *days=21*) in a string that's part of the TagParams list. To extract the value portion of this string (the portion after the equal sign), you can use the Values property of the TagParams string list and search for the proper entry at the same time. If it can't locate the parameter or if its value isn't an integer, the DLL displays an error message.

**Tip**

The PageProducer component supports user-defined tags, which can be any string you like, but you should first review the special tags defined by the TTags enumeration. The possible values include tgLink (for the `link` tag), tgImage (for the `img` tag), tgTable (for the `table` tag), and a few others. If you create a custom tag, as in the PageProd example, the value of the Tag parameter to the HTMLTag handler will be tgCustom.

## Producing Pages of Data

The HtmlProd example also has a DataSetPageProducer component, with the following settings and HTML source code:

```
object DataSetPageProducer1: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<html><head>'
    '<title>Data for <#name></title>'
    '</head><body>'
    '<h1><center>Data for <#name></center></h1>'
    '<p>Capital: <#capital></p>'
    '<p>Continent: <#continent></p>'
    '<p>Area: <#area></p>'
    '<p>Population: <#population></p>'
    '<hr>'
    '<p>Last updated on <#date><br>'
    '<HTML file produced by the program <#program>.</p>'
    '</body></html>')
  OnHTMLTag = DataSetPageProducer1HTMLTag
  DataSet = Table1
end
```

Simply by using tags with the names of the fields of the connected dataset (the usual COUNTRY.DB database table), the program automatically gets the value of the fields of the current record and replaces it automatically. This produces the output of Figure 21.9, which shows a browser connected to the HtmlProd example working as an HTTP server, as I'll discuss later. In the source code of the program related to this component, in fact, there is no reference to the database data:

```
procedure TFormProd.BtnLineClick(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Text := DataSetPageProducer1.Content;
  BtnSave.Enabled := True;
end;

procedure TFormProd.DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
```

```

begin
  if TagString = 'program' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'date' then
    ReplaceText := DateToStr (Date);
end;

```

**FIGURE 21.9:**

The output of the HtmlProd example for the Print Line button



## Producing HTML Tables

The last button of the HtmlProd example is Print Table. This button is connected to a DataSetTableProducer component, again calling its Content function and copying its result to the Text of the Memo. By simply connecting the DataSet property of the DataSetTableProducer to Table1, you can produce a standard HTML table. Actually, the component by default generates only 20 rows, as indicated by the MaxRows property. If you want to get all the records of the table, you can set this property to -1, a simple but undocumented setting.

### TIP

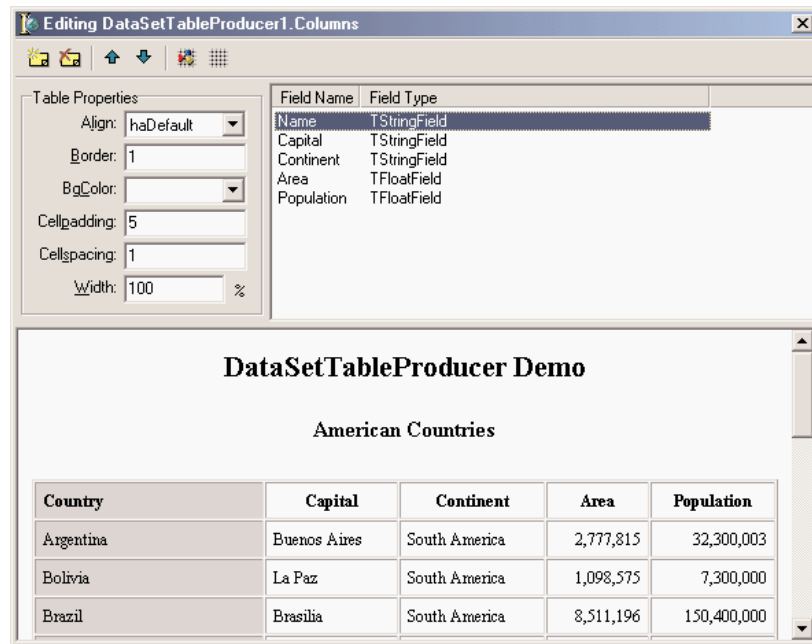
The DataSetTableProducer component starts from the current record rather than from the first one. This means that the second time you press the Print Table button, you'll see no records in the output. Adding a call to the First method of the table before calling the Content method of the producer component fixes the problem.

To make the output of this producer component more complete, you can do two different operations. The first is to provide some Header and Footer information, to generate the HTML heading and closing elements, and add a Caption to the HTML table. The second is to customize

the table itself, by using the setting specified by the `RowAttributes`, `TableAttributes`, and `Columns` properties. The property editor of the columns, which is also the default component editor, allows you to set most of these properties, providing at the same time a very nice preview of the output, as you can see in Figure 21.10. Before using this editor, you can set up properties for fields of the table, using the Fields editor. This is how, for example, you can format the output of the population and area fields to use thousands separators.

**FIGURE 21.10:**

The editor of the `Columns` property of the `DataSetTableProducer` component provides you with a preview of the final HTML table (if the database table is active).



There are three techniques you can use to customize the HTML table, and it's worth reviewing each of them:

- You can use the table producer component's `Column` property to set properties, such as the text and color of the title, or the color and the alignment for the cells in the rest of the column.
- You can use the `TField` properties, particularly those related to output. In the example, I've set the `DisplayFormat` property of the `Table1Continent` field object to `###,###,###`. This is the approach to use if you want to determine the actual output of each field. You might go even further and embed HTML tags in the output of a field.
- You can handle the `DataSetTableProducer` component's `OnFormatCell` event to customize the output further. In this event, you can set the various column attributes



uniquely for a given cell, but you can also customize the output string (stored in the `CellData` parameter) and embed HTML tags. This is something you can't do using the `Columns` property.

In the `HtmlProd` example, I've used a handler for this event to turn the text of the `Population` and `Area` columns to bold font and to a red background for large values (unless it is the header row). Here is the code:

```
procedure TFormProd.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and
    (((CellColumn = 3) and (Length (CellData) > 8)) or
    ((CellColumn = 4) and (Length (CellData) > 9))) then
    begin
      BgColor := 'red';
      CellData := '<b>' + CellData + '</b>';
    end;
end;
```

The rest of the code is summarized by the settings of the table producer component (formatted slightly to make it more readable and take less space):

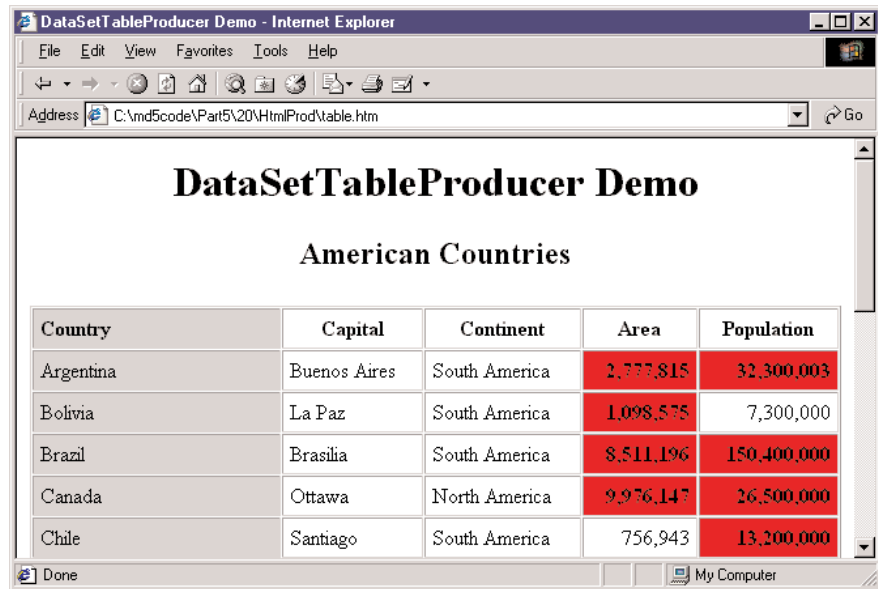
```
object DataSetTableProducer1: TDataSetTableProducer
  Caption = '<h2>American Countries</h2>'
  Columns = <
    item FieldName = 'Name'
      BgColor = 'silver'
      Title.Align = haLeft
      Title.BgColor = 'silver'
      Title.Caption = 'Country'
    end
    item FieldName = 'Capital'...
    item FieldName = 'Continent'...
    item FieldName = 'Area'
      Align = haRight
    end
    item FieldName = 'Population'
      Align = haRight
    end>
  Footer.Strings = ('<hr><i>Produced by HtmlProd</i></body></html>')
  Header.Strings = (<html><head><title>DataSetTableProducer Demo</title>'
    '</head><body><h1><center>DataSetTableProducer Demo</center></h1>')
  MaxRows = -1
  DataSet = Table1
```

```
TableAttributes.Border = 1
TableAttributes.CellPadding = 5
OnFormatCell = DataSetTableProducer1FormatCell
end
```

You can see the output of this program in Figure 21.11. I suggest you study the source code of the HTML file this program generates so that you can see the richness of its output and therefore the advantage of using this component.

**FIGURE 21.11:**

The output of the Print All button of the HtmlProd example, which is based on the DataSetTableProducer component



Country	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2,777,815	32,300,003
Bolivia	La Paz	South America	1,098,575	7,300,000
Brazil	Brasilia	South America	8,511,196	150,400,000
Canada	Ottawa	North America	9,976,147	26,500,000
Chile	Santiago	South America	756,943	13,200,000

## Using Style Sheets

The latest incarnations of HTML include a very powerful mechanism for separating content from presentation: Cascading Style Sheets (CSS). Using a style sheet, you can separate the formatting of the HTML (colors, fonts, font sizes, and so on) from the actual text displayed (the content of the page). This approach makes your code more flexible and your Web site easier to update. In addition, you can separate the task of making the site graphically appealing (the work of a Web designer) from automatic content generation (the work of a programmer). Style sheets are a rather complex technique, in which you give formatting values to the main types of HTML sections and to special “classes” (which have nothing to do with OOP). Again, see an HTML reference for the details.

How can we update table generation in the `HtmlProd` example to include style sheets? Simply enough, we can provide a link to the style sheet to use in the `Header` property of a second `DataSetTableProducer` component, with the line

```
<link rel="stylesheet" type="text/css" href="test.css">
```

We can then update the code of the `OnFormatCell` event handler with the following action (instead of the two lines changing the color and adding the bold font tag):

```
CustomAttrs := 'class="highlight"';
```

The style sheet I've provided (`test.css`, available in the source code of the example) defines a *highlight* style, which has exactly the bold font and red background that were hard-coded in the first `DataSetTableProducer` component.

The advantage of this approach is that now a graphic artist can modify the CSS file and give our table a nicer look without touching its code. When you want to provide many formatting elements, using a style sheet can also reduce the total size of the HTML file. This is an important element that can reduce download time.

## Dynamic Pages from a Custom Server

The `HtmlProd` component can be used to generate static HTML files, but doubles as a Web server, using an approach similar to what I've demonstrated in the `HttpServ` example, but in a more realistic context. The program, in fact, accesses the request of one of the possible page producers, simply passing the name of the component in a request. This is a portion of the `OnCommandGet` event handler of its `IdHTTPServer` component, which uses the `FindComponent` method to locate the proper producer component:

```
var
  Req, Html: String;
  Comp: TComponent;
begin
  Req := RequestInfo.Document;
  if Req [1] = '/' then
    Req := Copy (Req, 2, 1000); // skip '/'
    Comp := FindComponent (Req);
    if (Req <> '') and Assigned (Comp) and
      (Comp is TCustomContentProducer) then
      begin
        Table1.First;
        Html := TCustomContentProducer (Comp).Content;
      end;
  ResponseInfo.ContentText := Html;
end;
```

In case the parameter is not there (or is not valid), the server responds with an HTML-based menu of the available components:

```
Html := '<h1>Html Proc Menu</h1><p><ul>';
for I := 0 to ComponentCount - 1 do
  if Components [i] is TCustomContentProducer then
    Html := Html + '<li><a href="/" + Components [i].Name + "' +
      Components [i].Name + '</a></li>';
Html := Html + '</ul></p>';
```

Finally, if the program returns a table that uses CSS, the browser will request the CSS file from the server, so I've added some specific code to return it. With the proper generalizations, this code shows how a server can respond, returning files, and also how to indicate the MIME type of the response (ContentType):

```
if Pos ('test.css', Req) > 0 then
begin
  CssTest := TStringList.Create;
  try
    CssTest.LoadFromFile(ExtractFilePath(Application.ExeName) + 'test.css');
    ResponseInfo.ContentText := CssTest.Text;
    ResponseInfo.ContentType := 'text/css';
  finally
    CssTest.Free;
  end;
  Exit;
end;
```

## Publishing Static Databases on the Web

Once you know how to produce files, you can simply add links from one to another and produce a series of cross-linked HTML files, representing a portion of a Web site. There are circumstances in which writing a program that examines a database and produces files is the best approach for publishing database data on a Web site. You can use a similar technique if the following conditions apply:

**If the data doesn't change very often** A catalogue updated monthly or weekly is a good example. Even if you can update the site automatically every night, this is still a possible technique. (For real-time information, of course, this is certainly not a good approach!)

**If the amount of data is limited and available space not is an issue** This seems obvious, but the formatted HTML output might take much more space than the original database files. If you use a server-side program (such as those I'll be discussing in the next chapter) to generate the HTML from the database data on the fly, you might need less disk space on the Web site. Keep in mind that preparing all the HTML files beforehand usually

results in much better performance (faster server response time to Web requests, and lower memory overhead to process the requests) than generating the data on the fly.

**If the number of ways to navigate is limited** If there are three or four obvious paths of navigation (a main one and two or three cross-references), you can generate all of them statically. Otherwise, the cross-referencing HTML files will be much larger than the files with the actual data, and the time required to generate them may become excessive.

Even if only parts of these conditions apply to your specific needs, you can consider using a mixed approach. You can have a portion of the data and of the navigational files generated periodically and have a CGI and ISAPI application on the site, as well as let users do free searches and follow other less frequent paths.

---

**NOTE**

On the companion CD you can find an example, called DbCross, that generates hundreds of HTML files out of a master/detail database structure. The program also collects other data about the records, while generating the files, and produces a complete cross-reference. At the end, you can navigate by customers—orders by each customer—details of the order or by sale parts—orders where each part appears—details of each order. The code is quite complex, but I've tried to comment it in some detail, so you should be able to follow it.

## What's Next?

In this chapter, we've focused on some core Internet technologies, including the use of sockets and core Internet protocols. I've discussed the main idea and shown a few examples of the use of the mail and HTTP protocols. You can find many more examples of the use of the Indy components in the demos done by their developers.

After this introduction to the world of the Internet, we are now ready to delve into two key areas, the present and the future. The present is represented by the development of Web applications, and we'll explore the development of dynamic Web sites in the next chapter, focusing first on the *old* WebBroker technology and then moving to the new WebSnap architecture. The future is represented by the development of Web services and the use of XML and related technology, which will be discussed in Chapter 23.

# Web Programming with WebBroker and WebSnap

- Dynamic Web pages
- CGI, ISAPI, and Apache modules
- The WebBroker architecture
- The Web App debugger
- The new WebSnap architecture
- Adapters and server-side scripting

If the Internet has a growing role in the world, a good part of it depends on the success of the World Wide Web, based on the HTTP protocol. We've already discussed, in the preceding chapter, HTTP and the development of client- and server-side applications based on it. With the availability of several high-performance, scalable, and flexible Web servers, you'll rarely want to create your own. Dynamic Web server applications, in fact, are generally built by integrating scripting or compiled programs within Web servers, rather than replacing them with custom software.

This chapter is entirely focused on the development of server-side applications, which extend existing Web servers. We have already introduced the dynamic generation of HTML pages toward the end of the last chapter. Now we have to see how to integrate this dynamic generation within a server. This chapter is a logical continuation of the last one but won't complete the coverage of Internet programming, as the next chapter is further devoted to this topic, covering specifically XML and Web services.

**WARNING** To test some of the examples in this chapter, you'll need access to a Web server. The simplest solution is probably to use the version of Microsoft's IIS or Personal Web Server already installed on your computer. My personal preference, however, is to use the free and open-source Apache Web Server, available (along with extensive documentation) at [www.apache.org](http://www.apache.org). In any case, I won't spend much time giving you details on the configuration of your Web server to enable the use of applications; refer to its documentation for this.

## Dynamic Web Pages

When you browse a Web site, you generally download static pages—HTML-format text files—from the Web server to your client computer. As a Web developer, you can create these pages manually, but for most businesses, it makes more sense to build the static pages from information in a database of some type (a SQL server, a series of files, and so on). Using this approach, you're basically generating a snapshot of the data in HTML format, which is quite reasonable if the data isn't subject to frequent changes. This approach was discussed in Chapter 21, "Internet Programming: Sockets and Indy Components."

As an alternative to static HTML pages, you can build dynamic ones. To do this, you extract information directly from a database in response to the browser's request, so that the HTML sent by your application displays current data, not an old snapshot of the data. This approach makes sense if the data changes frequently.

As mentioned earlier, there are a couple of ways you can program custom behavior at the Web server, and these are ideal ways for you to generate HTML pages dynamically. The two

most common protocols for programming Web servers are CGI (the Common Gateway Interface) and the Web server APIs. Another technique, Active Server Pages (ASP), is quite popular in the Microsoft world, and I'll discuss it briefly because Delphi includes specific support for it.

**NOTE**

Keep in mind that Delphi's WebBroker technology (available in both the Enterprise and Professional editions) flattens the differences between CGI, WinCGI, and ISAPI by providing a common class framework. This way, you can easily turn a CGI application into a WinCGI one, upgrade it to use the ISAPI model, or integrate it into Apache.

## An Overview of CGI

CGI is a standard protocol for communication between the client browser and the Web server. It's not a particularly efficient protocol, but it is widely used and is not platform specific. This protocol allows the browser both to ask for and to send data, and it is based on the standard command-line input and output of an application (usually a console application). When the server detects a page request for the CGI application, it launches the application, passes command-line data from the page request to the application, and then sends the standard output of the application back to the client computer.

There are many tools and languages you can use to write CGI applications, and Delphi is only one of them. Given the obvious limitation that your Web server must be an Intel-based Windows or Linux system, you can build some fairly sophisticated CGI programs in Delphi and Kylix. Despite the fact that it's called a standard, there are actually different flavors of CGI. Traditional CGI uses the standard command-line input and output, along with environment variables. WinCGI uses an INI file passed as a command-line parameter to the application (instead of environment variables) and specific input and output files (instead of using command-line input/output). Server vendors developed WinCGI primarily for Visual Basic programmers, who cannot access environment variables. Another new variation, called FastCGI, is supposed to make the entire process of calling a CGI application much faster, but it's not widely supported yet.

To build a CGI program without using any support class, you can simply create a Delphi console application, remove the typical project source code, and replace it with the following statements:

```
program CgiDate;  
  {$APPTYPE CONSOLE}  
  
  uses SysUtils;  
  
  begin
```



```

writeln ('content-type: text/html');
writeln;
writeln ('<html><head>');
writeln ('<title>Time at this site</title>');
writeln ('</head><body>');
writeln ('<h1>Time at this site</h1>');
writeln ('<hr>');
writeln ('<h3>');
writeln (FormatDateTime("Today is " dddd, mmmm d, yyyy, ' +
    "'<br> and the time is" hh:mm:ss AM/PM', Now));
writeln ('</h3>');
writeln ('<hr>');
writeln ('<i>Page generated by CgiDate.exe</i>');
writeln ('</body></html>');

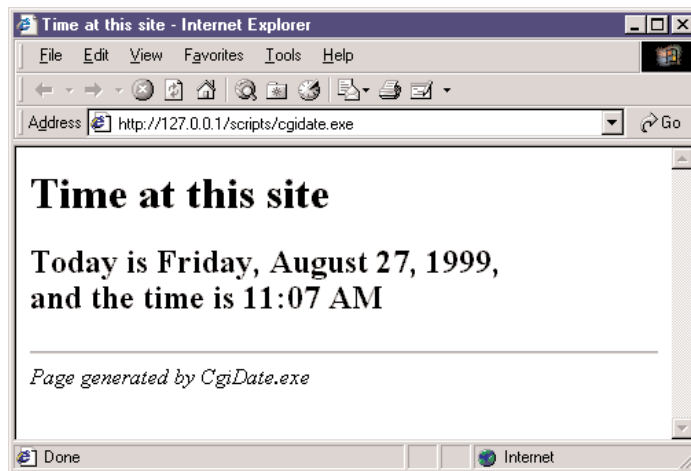
```

end.

CGI programs produce a header followed by the HTML text using the standard output. If you execute this program directly, you'll see the text in a terminal window. If you run it instead from a Web server and send the output to a browser, the formatted HTML text will appear, as shown in Figure 22.1.

**FIGURE 22.1:**

The output of the CgiDate application, as seen in a browser



Building advanced and complex applications with plain CGI requires a lot of work. For example, to extract status information on the HTTP request, you need to access to the relevant environment variables, as in:

```

// get the pathname
GetEnvironmentVariable ('PATH_INFO', PathName, sizeof (PathName));

```

## An Overview of ISAPI/NSAPI

A completely different approach is the use of the Web server APIs, the popular ISAPI (Internet Server API, introduced by Microsoft) and the less common NSAPI (Netscape Server API). These APIs allow you to write a DLL that the server loads into its own address space and usually keeps in memory for some time. Once it loads the DLL, the server can execute individual requests via threads within the main process, instead of launching a new EXE for every request (as it must in CGI applications).

When the server receives a page request, it loads the DLL (if it hasn't done so already) and executes the appropriate code, which may launch a new thread or use an existing one to process the page request (the IIS Web server offers thread pooling support to avoid creating a new thread for each request). The DLL code then sends the appropriate data back to the client that requested the page. Because this communication generally occurs in memory, this type of application is much faster than CGI, and a given system will be able to support more simultaneous page requests this way.

The main drawback to server API DLLs is that their tight integration with the server is an Achilles' heel; if the DLL crashes or produces memory leaks, the entire Web server can crash. However, the most recent releases of Microsoft's IIS Web server fix the problem by running the DLL in a *protected* space. Another problem is that when the DLL is in memory, you cannot compile an updated version; you need to unload the DLL first or momentarily stop the Web server (an operation you can do only on a test-bed computer).

Technically, ISAPI DLLs are not very different from plain Windows DLLs. They must export a couple of specific functions that the Web server will call: `GetExtensionVersion` and `HttpExtensionProc`. The server calls the first function when it loads the DLL for the first time and the second function for every following request. The parameters of these functions are complex data structures holding input data and server methods you can call to produce the result. Here is a sample of this function (taken from the `IsapiDem` example), which uses the `lpszPathInfo` field and the `WriteClient` function:

```
function HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD; stdcall;
var
    OutStr: string;
    StrLength: Cardinal;
begin
    with ECB do
        begin
            OutStr :=
                '<html><head><title>First Isapi Demo</title></head><body>' +
                '<h2><center>First Isapi Demo</center></h2>' +
                '<p>Hello Mastering Delphi Readers...</p><hr>' +
                '<p><b>Activated by ' + PChar(@lpszPathInfo[1]) + '</b></p>' +
```

```
    '<p><i>From IsapiDLL on ' + DateToStr(Now) + ' at ' + TimeToStr(Now) +  
    '</i></p></body></html>';  
    StrLength := Length(OutStr);  
    WriteClient(ConnID, PChar (OutStr), StrLength, 0);  
end;  
Result := HSE_STATUS_SUCCESS;  
end;
```

The program doesn't simply use the `lpszPathInfo` parameter but uses the substring starting with the second character, to get rid of the initial slash. To be more precise, the expression `PChar(@lpszPathInfo[1])` takes the string starting at the memory address of the second character of the path (a zero-based characters array).

## Apache Modules

Similarly to Microsoft's IIS, the Apache server of the Apache Foundation ([www.apache.org](http://www.apache.org)) allows server-side extensions by means of CGI or with specific extension libraries. In case of Apache, these libraries are called *modules*, or dynamic modules. In the Apache configuration, you can list the modules you are interested in and eventually connect them to a virtual directory.

Needless to say, you can program Apache modules in a similar low-level way to what I've just done for ISAPI, but I won't show you an example. I'll do it later using the Apache support added to the WebBroker architecture in Delphi 6.

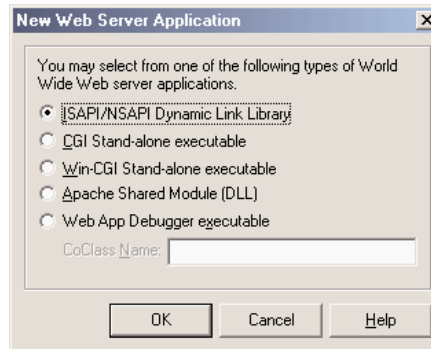
# Delphi's WebBroker Technology

The CGI and ISAPI code snippets I've shown you so far demonstrate the plain, direct approach to the protocol and API. Extending these examples at that level is certainly possible, but what is interesting in Delphi is to use the WebBroker technology. This comprises a class hierarchy within VCL and CLX, built to simplify server-side development on the Web, and a specific type of data modules, called `WebModules`. Both the Enterprise and Professional editions of Delphi 5 include this framework (differently from the more advanced and newer WebSnap framework, which is available only in the Enterprise version of Delphi 6).

Using the WebBroker technology, you can begin developing an ISAPI or CGI application or an Apache module very easily. On the first page (*New*) of the New Items dialog box, select the Web Server Application icon. The subsequent dialog box will offer you options (two more than in Delphi 5). As you can see in Figure 22.2, you can choose ISAPI, CGI, WinCGI, Apache module, and the Web App Debugger.

**FIGURE 22.2:**

The alternative options for building a Web server application in Delphi

**TIP**

As a starting point for your server-side application, you can also use the DB Web Application Wizard, available in the Business page of the New Items dialog box. This wizard generates a program with a BDE table or query connected to a DataSetTableProducer. It can be helpful, but the generated code is really very limited and there is no support for Apache and the Web App Debugger.

For example, if you select the first option, Delphi will generate the basic structure of an ISAPI application for you. The application that Delphi generates (no matter which type you choose) is based on the TWebModule class, a container very similar to a data module. The WebModule code is similar to that of a data module, as we'll see in a moment, but the code of the library is worth looking at:

```
library Project1;

uses
  WebBroker,
  ISAPIThreadPool,
  ISAPIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

{$R *.RES}

exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

**WARNING**

This code changes slightly between Delphi versions. If you have older code around, you'll need to refer to the `WebBroker` unit instead of the previous `HTTPApp` unit. Delphi 6 adds the reference to the `ISAPIThreadPool` unit, which provides support for pooling threads under ISAPI (with a couple of classes not documented in the Help file).

Although this is a library that exports the ISAPI functions, the code looks similar to that of an application. However, it uses a trick—the `Application` object used by this program is not the typical global object of class `TApplication` but an object of a new class. This new `Application` object is of class `TISAPIApplication` (or `TCGIApplication` if you've built that type of application), which derives from `TWebApplication`.

Although these application classes provide the foundations, you won't use them very often (just as you don't use the `Application` object very often in a form-based Delphi application). The most important operations take place in the `WebModule`. This component derives from `TCustomWebDispatcher`, which provides support for all the input and output of our programs.

In fact, the `TCustomWebDispatcher` class defines `Request` and `Response` properties, which store the client request and the response we're going to send back to the client. Each of these properties is defined using a base abstract class (`TWebRequest` and `TWebResponse`), but an application initializes them using a specific object (such as the `TISAPIRequest` and `TISAPIResponse` subclasses). These classes make available all the information passed to the server, so you have a single, simple approach to accessing all the information. The same is true of a response, which is very easy to manipulate. One advantage to this approach is that an ISAPI DLL written with this framework is very similar to a CGI application; in fact, they are frequently identical in the source code you write.

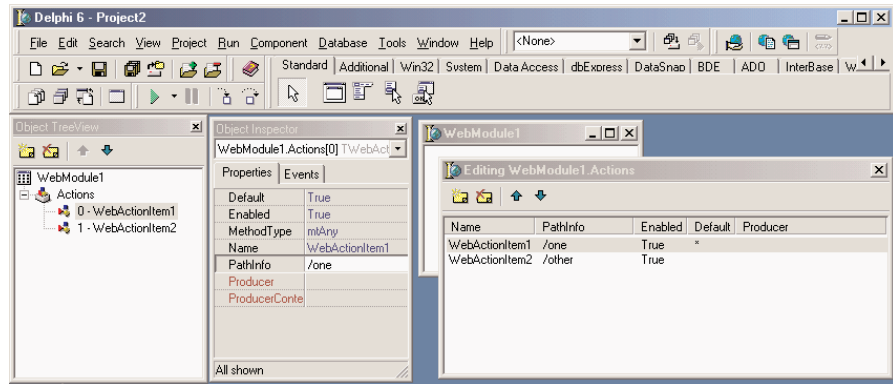
If this is the structure of Delphi's framework, how do you write the application code? Well, in the `WebModule`, you can use the Actions editor (shown in Figure 22.3) to define a series of actions (stored in the `Actions` array property) depending on the *pathname* of the request. This pathname is a portion of the CGI or ISAPI application's URL, which comes after the program name and before the parameters, such as `path1` in the following URL:

```
http://www.example.com/scripts/cgittest.exe/path1?param1=date
```

By providing different actions, your application can easily respond to requests with different pathnames, and you can assign a different producer component or call a different `OnAction` event handler for every possible pathname. Of course, you can omit the pathname to handle a generic request. Consider also that, instead of basing your application on a `WebModule`, you can use a plain data module and add a `WebDispatcher` component to it. This is a good approach if you want to turn an existing Delphi application into a Web server extension.

**FIGURE 22.3:**

The Actions property editor of the WebModule, along with the properties of one of the actions in the Object Inspector

**WARNING**

The WebModule incorporates the WebDispatcher and doesn't require it as a separate component. Unlike WebSnap applications, in fact, WebBroker programs cannot have multiple dispatchers or multiple Web modules. Notice also that the actions of the WebDispatcher have absolutely nothing to do with the actions stored in a `TActionList` component.

When you define the accompanying HTML pages that launch the application, the links will make page requests to the URLs for each of those paths. Having one single ISAPI DLL that can perform different operations depending on a parameter (in this case, the pathname) allows the server to keep a copy of this DLL in memory and respond much faster to user requests. The same is partially true for a CGI application: The server has to run several instances but can cache the file and make it available faster.

The `OnAction` event is where you put the code to specify the *response* to a given *request*, the two main parameters passed to the event handler. Here is a simple example:

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    '<html><head><title>Hello Page</title></head><body>' +
    '<h1>Hello</h1>' +
    '<hr><p><i>Page generated by Marco</i></p></body></html>';
end;

```

The `Content` property of the `Response` parameter is where you enter the HTML code that you want users to see. The only drawback of this code is that the output in a browser will be correctly displayed on multiple lines, but looking at the HTML source code, you'll see a single line corresponding with the entire string. To make the HTML source code more readable, by splitting it up onto multiple lines, you can insert the #13 newline character.

To let other actions handle this request, you'll set the last parameter, `Handled`, to `False`. Otherwise, the default value is `True`, and once you've handled the request with your action, the `WebModule` assumes you're finished. Most of an ISAPI application's code will be in the `OnAction` event handlers for the actions defined in the `WebModule` container. These actions receive a request from the client and return a response using the `Request` and `Response` parameters.

When using the producer components, your `OnAction` event often returns, as `Response.Content`, the `Content` of the producer component, with a simple assignment. You can shortcut this code by assigning a producer component to the `Producer` property of the action itself, with no need to write these simple event handlers anymore (but don't do both things, as that might get you into trouble).

**TIP**

As an alternative to the `Producer` property, you can use the `ProducerContent` property introduced in Delphi 6. This new property allows you to connect custom producer classes that don't inherit from the `TCustomContentProducer` class but implement the `IProduceContent` interface. The `ProducerContent` property is *almost* an interface property: It behaves in the same way, but thanks to its property editor and not based on the new support for interfaced properties of Delphi 6.

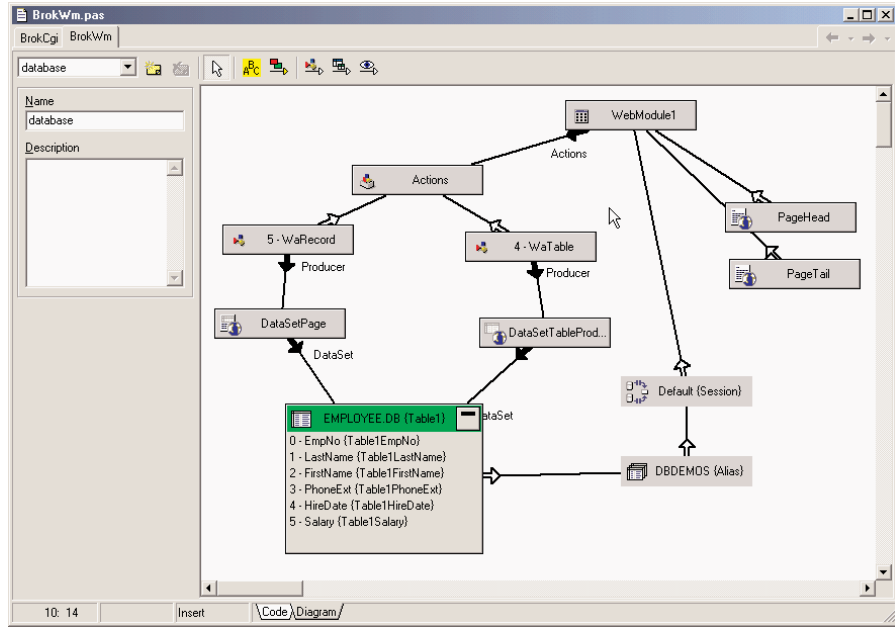
## Building a Multipurpose WebModule

To demonstrate how easily you can build a feature-rich server-side application using Delphi's support, I've created the `BrokDemo` example. This example can be compiled as a CGI or an ISAPI application, simply by choosing the proper project file. The `WebModule` is shared by the two projects, without any difference in the source code, a practical proof that, using the `WebBroker` framework, you can move from ISAPI to CGI, from Apache to the `Web App Debugger` (although this last step requires a little extra tweaking in the source code). In the past, I tended to test programs with CGI (to avoid having to stop the server to free the library and recompile it) and then deploy them with ISAPI. Now I tend to test with the `Web App Debugger` and then deploy under Apache.

A key element is the list of actions we're going to support with this application. The actions can be managed in the `Actions` editor or directly in the `Object TreeView`, as we've already seen in Figure 22.3. Actions are also visible in the `Designer` page of the editor, so you can graphically see their relationship with database objects, as shown for the `BrokDemo` in Figure 22.4. If you examine the figure or the source code, you'll notice that I've given a specific name to every action. I've also given meaningful names to the `OnAction` event handlers. For instance, `TimeAction` as a method name should be much more understandable than the `WebModule1WebActionItem1Action` name automatically generated by Delphi.

**FIGURE 22.4:**

The structure of the BrokDemo example, as shown by the Designer



Every action has a different pathname, with one of them marked as default and executed even if no pathname is specified. The first interesting idea in this program is the use of two PageProducer components, used for the initial and final portion of every page, PageHead and PageTail. Centralizing this code makes it easier to modify it, particularly if it is based on external HTML files. The HTML produced by these components is added at the beginning and the end of the resulting HTML in the OnAfterDispatch event handler of the Web module:

```
procedure TWebModule1.WebModule1AfterDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageHead.Content + Response.Content + PageTail.Content;
end;
```

I'm adding the initial and final HTML at the end of the page generation simply because this allows the components to produce the HTML as if they were making all of it. Starting with some HTML in the OnBeforeDispatch event means that you cannot directly assign the producer components to the actions, or the producer component will override the Content you've already provided in the response. The PageTail component includes a custom tag for the script name, replaced by the following code, which uses the current request object available within the Web module:

```
procedure TWebModule1.PageTailHTMLTag(Sender: TObject; Tag: TTag;
```



```

    const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    if TagString = 'script' then
        ReplaceText := Request.ScriptName;
    end;

```

This code is activated to expand the `<#script>` tag of the `PageTail` component's `HTMLDoc` property. The code of the time and date actions is straightforward. The really interesting part begins with the `Menu` path, which is the default action. In its `OnAction` event handler, the application uses a `for` loop to build a list of the available actions (using their names without the first two letters, which are always `Wa` in my example), providing a link to each of them with an anchor (an `<a>` tag):

```

procedure TWebModule1.MenuAction(Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
var
    I: Integer;
begin
    Response.Content := '<h3>Menu</h3><u1>' #13;
    for I := 0 to Actions.Count - 1 do
        Response.Content := Response.Content + '<li> <a href="' +
            Request.ScriptName + Action[I].PathInfo + '"> ' +
            Copy (Action[I].Name, 3, 1000) + '</a>' #13;
        Response.Content := Response.Content + '</u1>';
    end;

```

Another action of the `BrokDemo` example provides users with a list of the system settings related to the request, something that is quite useful for debugging. It is also instructive to learn how much information, and exactly what information, the HTTP protocol transfers from a browser to a Web server and vice versa. To produce this list, the program looks for the value of each property of the `TWebRequest` class, as this initial snippet demonstrates:

```

procedure TWebModule1.StatusAction(Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
var
    I: Integer;
begin
    Response.Content := '<h3>Status</h3>' #13 +
        'Method: ' + Request.Method + '<br>' #13 +
        'ProtocolVersion: ' + Request.ProtocolVersion + '<br>' #13 +
        'URL: ' + Request.URL + '<br>' #13 +
        'Query: ' + Request.Query + '<br>' #13 + ...

```

## Dynamic Database Reporting

The BrokDemo example defines two more actions, indicated by the `/table` and `/record` pathnames. For these two last actions, our program produces a main list of names and then displays the details of one record, using a `DataSetTableProducer` component to format the entire table and a `DataSetPageProducer` component to build the record view. Here are the properties of these two components:

```

object DataSetTableProducer1: TDataSetTableProducer
  DataSet = Table1
  OnFormatCell = DataSetTableProducer1FormatCell
end
object DataSetPage: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<h3>Employee: <#LastName></h3>'
    '<u1><li> Employee ID: <#EmpNo>'
    '<li> Name: <#FirstName> <#LastName>'
    '<li> Phone: <#PhoneExt>'
    '<li> Hired On: <#HireDate>'
    '<li> Salary: <#Salary></u1>')
  OnHTMLTag = PageTailHTMLTag
  DataSet = Table1
end

```

To produce the entire table, we simply connect the `DataSetTableProducer` to the `Producer` property of the corresponding actions, without providing any specific event handler. The table is made more powerful by adding internal links to the specific records. The following code is executed for each cell of the table but activated only for the first column or the first row (the one with the title):

```

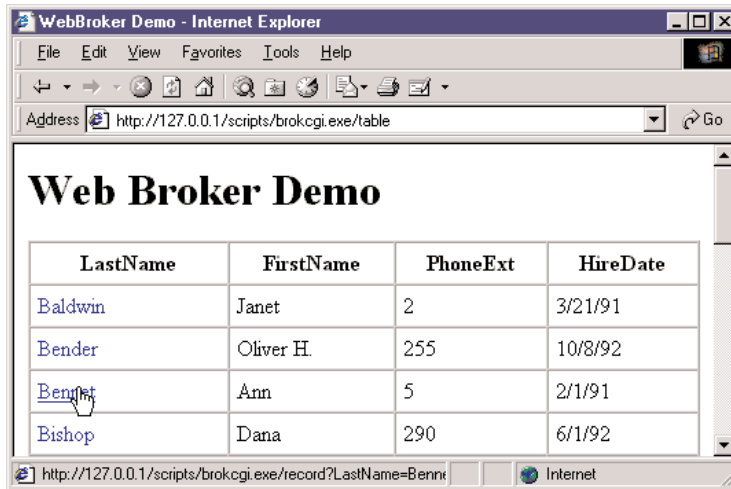
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign;
  var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + ScriptName + '/record?LastName=' +
      Table1['LastName'] + '&FirstName=' + Table1 ['FirstName'] + '"> ' +
      CellData + ' </a>';
end;

```

You can see the result of this action in Figure 22.5. When the user selects one of the links, the program is called again, and it can check the `QueryFields` string list and extract the parameters from the URL. It then uses the values corresponding to the table fields used for the record search (which is based on the `FindNearest` call).

**FIGURE 22.5:**

The output corresponding to the `table` path of the BrokDemo example, which produces an HTML table with internal hyperlinks.



```

procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
begin
    Table1.Open;
    // go to the requested record
    Table1.FindNearest ([Request.QueryFields.Values['LastName'],
        Request.QueryFields.Values['FirstName']]);
    // get the output
    Response.Content := Response.Content + DataSetPage.Content;
end;

```

**NOTE**

The example we've just built accesses a Paradox table via the BDE. The CGI version executes once for every request and will actually load and unload the BDE each time it runs. As alternatives, you can consider three different approaches: using ISAPI instead of CGI (to keep the application and the BDE loaded in memory), accessing the data from a plain file (or avoid the BDE with some other data access technology), or running another BDE application on the server (so that the BDE will remain loaded in memory). When using the BDE in an ISAPI application, though, you need to add a Session component to avoid concurrent access by multiple threads to the same BDE session.

## Of Queries and Forms

The previous example used some of the HTML producer components introduced earlier in this chapter. There is another component of this group we haven't used yet, the QueryTableProducer. As we'll see in a moment, this component makes building even complex database programs a

breeze. Suppose you want to search for some customers in a database. You might construct the following HTML form (embedded in an HTML table for better formatting):

```
<h4>Customer QueryProducer Search Form</h4>
<form action="/scripts/CustQueP.dll/search" method="POST">
<table>
<tr><td>State:</td>
  <td><input type="text" name="State"></td></tr>
<tr><td>Country:</td>
  <td><input type="text" name="Country"></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

**NOTE**

As in Delphi, an HTML form hosts a series of controls (typically, things like input fields). There are visual tools to help you design these forms, or you can manually enter the proper HTML code. The available controls include buttons, input text (or edit boxes), selections (or combo boxes), and radio buttons (or input buttons). You can define buttons as specific types, such as Submit or Reset, which imply standard behaviors. An important element of forms is the *request method*, which can be either POST (data is passed behind the scenes, and you receive it in the `ContentFields` property) or GET (data is passed as part of the URL, and you extract it from the `QueryFields` property).

There is a very important element to notice in the form: the names of the input components (*State* and *Country*), which should match the parameters of a Query component:

```
select
  Company, State, Country
from
  CUSTOMER.DB
where
  State = :State or Country = :Country
```

This code is used in the CustQueP (customer query producer) example. To build it, I've placed a Query component inside the WebModule and generated the field objects for it. In the same WebModule, I've added a QueryTableProducer component connected to the Producer property of the /search action. The program will generate the proper response. How does this work? When we activate the QueryTableProducer component by calling its Content function, it initializes the Query component by obtaining the parameters from the HTTP request. The component can automatically examine the request method and then use either the `QueryFields` property (if the request is a GET) or the `ContentFields` property (if the request is a POST).

One problem with using a static HTML form as we did before is that it doesn't tell us which states and countries we can search for. To address this, we can use a selection control instead of an edit control in the HTML form. However, if the user adds new records to the

database table, we'll need to update the element list automatically. As a final solution, we can design the ISAPI DLL to produce a form on-the-fly, and we can fill the selection controls with the available elements.

We'll generate the HTML for this page in the /form action, which we've connected to a PageProducer component. The PageProducer contains the following HTML text, which embeds two special tags:

```
<h4>Customer QueryProducer Search Form</h4>
<form action="CustQueP.dll/search" method="POST">
<table>
<tr><td>State:</td>
  <td><select name="State"><#State></select></td></tr>
<tr><td>Country:</td>
  <td><select name="Country"><option> </option><#Country></select></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

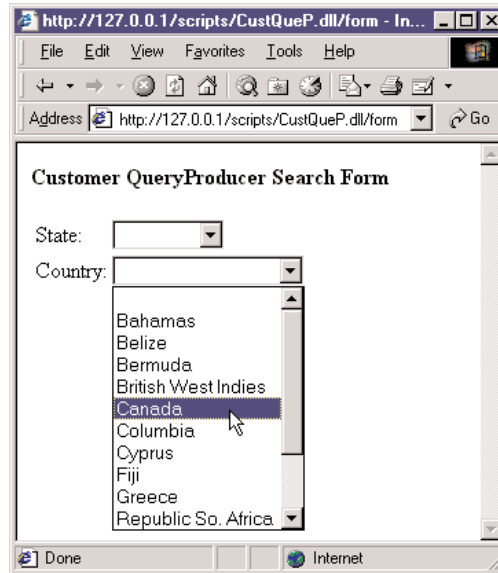
You'll notice that the tags have the same name as some of the table's fields. When the PageProducer encounters one of these tags, it adds an <option> HTML tag for every distinct value of the corresponding field. Here's the OnTag event handler's code, which is quite generic and reusable:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  ReplaceText := '';
  Query2.SQL.Clear;
  Query2.SQL.Add ('select distinct ' + TagString + ' from customer');
  try
    Query2.Open;
    try
      Query2.First;
      while not Query2.EOF do
        begin
          ReplaceText := ReplaceText +
            '<option>' + Query2.Fields[0].AsString + '</option>'#13;
          Query2.Next;
        end;
      finally
        Query2.Close;
      end;
    except
      ReplaceText := '{wrong field: ' + TagString + '}';
    end;
  end;
```

This method used a second Query component, which I manually placed on the form and connected to the DBDEMOS database, and it produces the output shown in Figure 22.6.

**FIGURE 22.6:**

The form action of the CustQueP example produces an HTML form with a selection component dynamically updated to reflect the current status of the database.



Finally, this Web server extension, like many others we've built, allows the user to view the details of a specific record. As in the last example, we can accomplish this by customizing the output of the first column (column zero), which is generated by the QueryTableProducer component:

```
procedure TWebModule1.QueryTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + Request.ScriptName + '/record?' + CellData +
      '>' + CellData + '</a>'#13;
  if CellData = '' then
    CellData := '&nbsp;';
end;
```

### TIP

When you have an empty cell in an HTML table, most browsers render it without the border. For this reason, I've added a "nonbreaking space" symbol (&nbsp;) into each empty cell. This is something you'll have to do in each HTML table generated with Delphi's table producers.

The action for this link is /record, and we'll pass a specific element after the ? parameter (without the parameter name, which is slightly nonstandard). The code we use to produce the HTML tables for the records doesn't use the producer components as we've been doing; instead, it is very similar to the code of an early ISAPI example:

```

procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  if Request.QueryFields.Count = 0 then
    Response.Content := 'Record not found'
  else
    begin
      Query2.SQL.Clear;
      Query2.SQL.Add ('select * from customer ' +
        'where Company="' + Request.QueryFields[0] + '"');
      Query2.Open;
      Response.Content :=
        '<html><head><title>Customer Record</title></head><body>'#13 +
        '<h1>Customer Record: ' + Request.QueryFields[0] + '</h1>'#13 +
        '<table border>'#13;
      for I := 1 to Query2.FieldCount - 1 do
        Response.Content := Response.Content +
          '<tr><td>' + Query2.Fields [I].FieldName + '</td>'#13'<td>' +
          Query2.Fields [I].AsString + '</td></tr>'#13;
      Response.Content := Response.Content + '</table><hr>'#13 +
        '// pointer to the query form
        '<a href="' + Request.ScriptName + '/form">' +
        ' Next Query </a>'#13 + '</body></html>'#13;
    end;
  end;

```

## Debugging with the Web App Debugger

Debugging Web applications written in Delphi is often quite difficult. In fact, you cannot simply run the program and set breakpoints in it, but should convince the Web server to run your CGI program or library within the Delphi debugger. This can be accomplished by indicating a Host application in Delphi's Run Parameters dialog box, but it implies letting Delphi run the Web server (which is often a Windows service, not a stand-alone program).

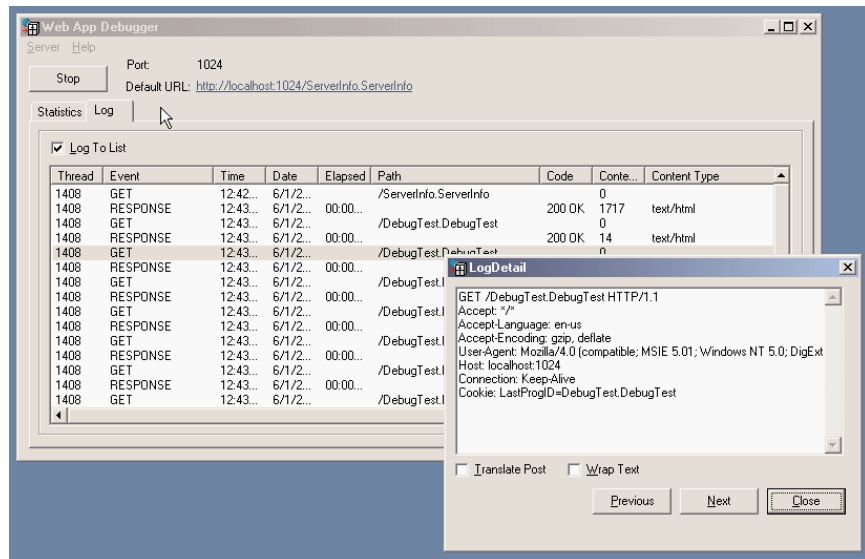
To solve all of these issues, Borland has added to Delphi 6 a specific Web App Debugger program. This tool, activated by the corresponding item of the Tools menu, is a Web server, which waits for requests on a port you can set up (1024 by default). When a request arrives, the program can forward it to a stand-alone executable, using COM-based techniques. This

means you can run the Web server application from within the Delphi IDE, set all the breakpoints you need, and then (when the program is activated through the Web App Debugger) debug the program as you'll do for a plain executable file.

The Web App Debugger does also a good job in logging all the received requests and the actual responses returned to the browser, as you can see in Figure 22.7. The program also has a Statistics page, which interestingly tracks the time required for each response, allowing you to test the efficiency of an application in different conditions.

**FIGURE 22.7:**

The log of the Web App Debugger with its LogDetail window



By using the corresponding option of the New Web Server Application dialog, you can easily create a new application compatible with the debugger. This defines a standard project, which creates both a main form and a data module. The (useless) form includes code for registering the application as an OLE automation server, as:

**const**

```
CLASS_ComWebApp: TGUID = '{33A4D4F0-E082-4723-9165-5D8F95AF1577}' ;
```

**initialization**

```
TWebAppAutoObjectFactory.Create(Class_ComWebApp, 'FirstDemo',  
    'FirstDemo Object');
```

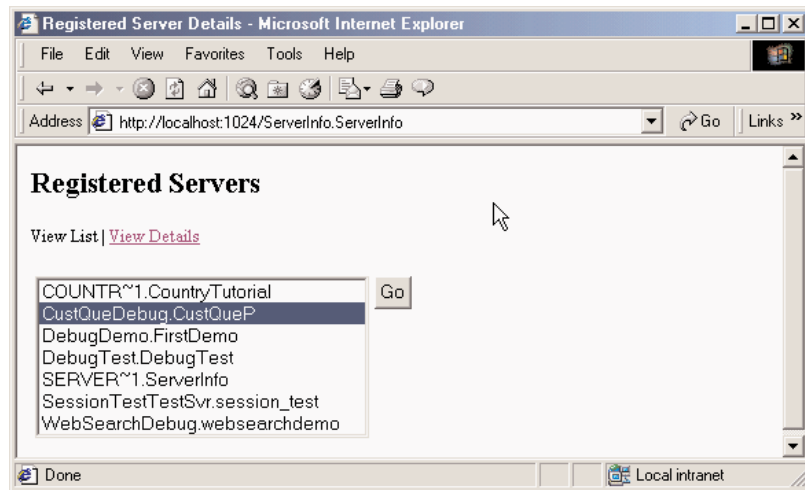
The information is used by the Web App Debugger to get a list of the available programs. This is done when you use the default URL for the debugger, indicated in the form as a link, as you can see (for example) in Figure 22.8. The list includes all of the registered servers, not



only those running. In fact, the use of COM Automation accounts for the automatic activation of a server. Not that this is a good idea, though, as running and terminating the program each time will make the process much slower. Again, the idea is to run the program within the Delphi IDE, to be able to debug it easily. Notice, though that the list can be expanded with the detailed view, which includes a list of the actual executable files and many other details.

**FIGURE 22.8:**

A list of applications registered with the Web App Debugger is displayed when you hook to its *home* page.



The data module for this type of project has some initialization code as well:

```
uses WebReq;
```

#### initialization

```
WebRequestHandler.WebModuleClass := TWebModule2;
```

This approach should be used only for debugging. To deploy the actual application you should then use one of the other options. What you can do is create the project files for another type of Web server program and add to the project the same Web module of the debug application. The presence of the extra initialization line won't create a problem.

The reverse is slightly more complex. To debug an existing application, you have to create a program of this type, remove the Web module, add the existing one, and patch it by adding a line to set the `WebModuleClass` of the `WebRequestHandler`, like the one in the preceding code snippet. To account for possible missing initialization of the `WebRequestHandler` object, you might want to change this type of code into:

```
if WebRequestHandler <> nil then
  WebRequestHandler.WebModuleClass := ... // Web module class
```

**WARNING** By doing this for the CustQueP example (it is the CustQueDebug project), I realized that some of the Web request settings are different. So instead of using the `ScriptName` property of the request (set to empty for a Web debug application), you have to use the `InternalScriptName` property.

There are other two interesting elements in the use of the Web App Debugger. The first is that you can test your programs without having a Web server installed and without having to tweak its settings. In other words, you don't have to deploy your programs to test them—you simply try them out right away. Another advantage is that, contrary to doing early development of the applications as CGI, you can start experimenting with a multithreaded architecture right away, without having to deal with the loading and unloading of libraries, which often implies shutting down the Web server and possibly even the computer.

**NOTE** If your aim is to build an ISAPI application, you can also use a specific ISAPI DLL debugging tool. One such tool, called IntraBob, has been built by Bob Swart and is available on his Web site ([www.drbob42.com](http://www.drbob42.com)) as freeware.

## Working with Apache

If you plan on using Apache instead of IIS or another Web server, you can certainly take advantage of the common CGI technology to deploy your applications on almost any Web server. However, using CGI means some reduced speed and some trouble handling state information (as you cannot keep any data in memory). This is a good reason for writing an ISAPI application or a dynamic Apache module. Using Delphi's WebBroker technology, you can also easily compile the same code for both technologies, so that moving your program to a different Web platform becomes much simpler. Finally, you can also recompile a CGI program or a dynamic Apache module with Kylix and deploy it on a Linux server.

As I've mentioned, Apache can run traditional CGI applications but has also a specific technology for keeping the server extension program loaded in memory at all times for faster response. To build such a program in Delphi 6, you can simply use the Apache Shared Module option of the New Web Server Application dialog box. You end up with a library having this type of source code for its project:

```
library Apache1;
```

```
uses
```

```
  WebBroker,  
  ApacheApp,  
  ApacheWm in 'ApacheWm.pas' {WebModule1: TWebModule};
```

```

{$R *.res}

exports
  apache_module name 'apache1_module';

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.

```

Notice in particular the exports clause, which indicates the name used by Apache configuration files to reference the dynamic module. In the project source code, you can add two more definitions, the module name and the content type, in the following way:

```

ModuleName := 'Apache1_module';
ContentType:= 'Apache1-handler';

```

If you don't set them, Delphi will assign them some default values, which are built adding the *\_module* and *-handler* strings to the project name, ending up with the two names I've used above.

An Apache module is generally not deployed within a script folder, but within the modules subfolder of the server itself (by default, c:\Program Files\Apache\modules). Then you have to edit the http.conf file, adding a line to load the module, as:

```
LoadModule apache1_module modules/apache1.dll
```

Finally, you have to indicate when the module is invoked. The handler defined by the module can be associated with a given file extension (so that your module will process all of the files having a given extension) or with a physical or virtual folder. In the latter case, the folder doesn't exist, but Apache pretends it is there. This is how you can set up a virtual folder for the simple Apache1 module:

```
<Location /Apache1>SetHandler Apache1-handler</Location>
```

As Apache is inherently case sensitive (because of its Linux heritage), you might also want to add a second, lowercase virtual folder:

```
<Location /apache1>SetHandler Apache1-handler</Location>
```

Now you can invoke the sample application with the URL <http://localhost/Apache1>. A great advantage of using virtual folder in Apache is that a user doesn't really distinguish between the physical and dynamic portions of your site, as we'll better see in the next example.

Because the development of Apache modules with WebBroker is almost identical to the development of other types of programs, instead of building an actual application (besides the over-simplistic Apache1 example) I've created a new version of the BrokDemo example, already available as a CGI or ISAPI program. To do this, I've taken the project file of an

Apache module from that example, added the local Web modules to it, and modified the project source code to reflect the proper module name and handler. I've actually defined them differently than the default, as the following code excerpt demonstrates:

```
library BrokApache;  
  
exports apache_module name 'brokdemo_module';  
  
begin  
  ContentType:= 'brokdemo-handler';
```

After compiling the module and editing the `http.conf` file as explained above, the program was ready to be used in two different ways, CGI and dynamic module. An obvious difference between the two types of invocation is their URLs:

```
http://localhost/scripts/brokcgi.exe/table  
http://localhost/brokdemo/table
```

Not only is the latter URL simpler, but it hides the fact that we are running an application with a `/table` parameter. In fact, it seems we are accessing a specific folder of the server. Actually, the Apache configuration file can be modified to also invoke CGI applications through virtual folders, which explains why CGI applications have a path-like command prefixing the request. Another related explanation is that Linux CGI applications, like any other executable file, have no extension whatsoever, so their names still seem to be part of a path.

## Practical Examples

After this general introduction to the core idea of the development of server-side applications with WebBroker, let me end this part of the chapter with two simple practical examples. The first is a classic Web counter. The second is an extension of the WebFind program presented in the preceding chapter to produce a dynamic page instead of filling a list box.

### A Web Hit Counter

The server-side applications we've built up to now were based only on text. Of course, you can easily add references to existing graphics files. What's more interesting, however, is to build server-side programs capable of generating graphics that change over time.

A typical example is a *page hit counter*. To write a Web counter, we save the current number of hits to a file and then read and increase the value every time the counter program is called. How do we return this information? If all we need is some HTML text with the number of hits, the code is straightforward:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;  
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
```

```
var
  nHit: Integer;
  LogFile: Text;
  LogFileName: string;
begin
  LogFileName := 'WebCont.log';
  System.Assign (LogFile, LogFileName);
  try
    // read if the file exists
    if FileExists (LogFileName) then
      begin
        Reset (LogFile);
        Readln (LogFile, nHit);
        Inc (nHit);
      end
    else
      nHit := 0;
      // saves the new data
      Rewrite (LogFile);
      Writeln (LogFile, nHit);
    finally
      Close (LogFile);
    end;
    Response.Content := IntToStr (nHit);
  end;
```

**WARNING**

This simple file handling does not scale. When multiple visitors hit the page at the same time, this code may return false results or fail with a file I/O error because a request in another thread has the file open for reading while this thread tries to open the file for writing. To support a similar scenario, you'll need to use a mutex (or a critical section in a multithreaded program) to let each subsequent thread wait until the thread currently using the file has completed its task.

What's a little more interesting is to create a graphical counter that can be easily embedded into any HTML page. There are basically two approaches for building a graphical counter: you can prepare a bitmap for each digit up front and then combine them in the program, or you can simply let the program draw over a memory bitmap to produce the graphic you want to return. In the WebCount program, I've chosen this second approach.

Basically, we can create an Image component that holds a memory bitmap, which we can paint on with the usual methods of the TCanvas class. Then we can attach this bitmap to a TjpegImage object. Accessing the bitmap through the JpegImage component converts the

image to the JPEG format. At this point, we can save the JPEG data to a stream and return it. As you can see, there are many steps, but the code is not really complex:

```
// create a bitmap in memory
Bitmap := TBitmap.Create;
try
    Bitmap.Width := 120;
    Bitmap.Height := 25;
    // draw the digits
    Bitmap.Canvas.Font.Name := 'Arial';
    Bitmap.Canvas.Font.Size := 14;
    Bitmap.Canvas.Font.Color := RGB (255, 127, 0);
    Bitmap.Canvas.Font.Style := [fsBold];
    Bitmap.Canvas.TextOut (1, 1, 'Hits: ' +
        FormatFloat ('###,###,###', Int (nHit)));
    // convert to JPEG and output
    Jpeg1 := TJpegImage.Create;
    try
        Jpeg1.CompressionQuality := 50;
        Jpeg1.Assign(Bitmap);
        Stream := TMemoryStream.Create;
        Jpeg1.SaveToStream (Stream);
        Stream.Position := 0;
        Response.ContentStream := Stream;
        Response.ContentType := 'image/jpeg';
        Response.SendResponse;
        // the response object will free the stream
    finally
        Jpeg1.Free;
    end;
finally
    Bitmap.Free;
end;
```

The three statements responsible for returning the JPEG image are the two that set the `ContentStream` and `ContentType` properties of the `Response` and the final call to `SendResponse`. The content type must match one of the possible MIME types accepted by the browser, and the order of these three statements is relevant. There is also a `SendStream` method in the `Response` object, but it should be called only after sending the type of the data with a separate call.

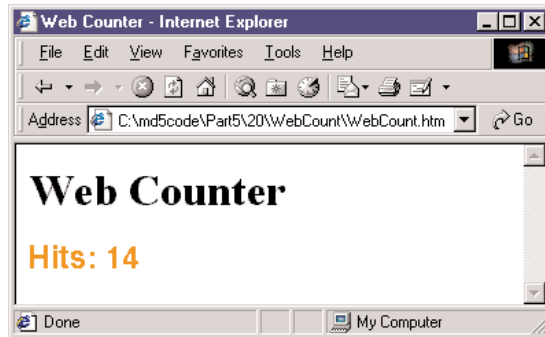
You can see the effect of this program in Figure 22.9. To obtain it, I've added the following code to an HTML page:

```

```

**FIGURE 22.9:**

The graphical Web hit counter in action



## Searching with a Web Search Engine

In Chapter 21, I discussed the use of the Indy HTTP client component to retrieve the result of a search on the Google Web site. Now I'm going to extend the example a little, turning it into a server-side application. The WebSearch program on the companion CD, available as a CGI application or a Web App Debugger executable, has an action that simply returns the HTML retrieved by the search engine and a second action that fills a client data set component, then hooked to a table page producer. This is the code of this second action:

```

const
  strSearch = 'http://www.google.com/search?as_q=borland+delphi&num=100';

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  I: integer;
begin
  if not cds.Active then
    cds.CreateDataSet
  else
    cds.EmptyDataSet;
  for i := 0 to 5 do // how many pages?
  begin
    // get the data form the search site
    GrabHtml (strSearch + '&start=' + IntToStr (i*100));
    // scan it to fill the cds
    HtmlStringToCds;
  end;
  cds.First;
  // return producer content
  Response.Content := DataSetTableProducer1.Content;
end;

```

The `GrabHtml` method is identical to the `WebFind` example, while the `HtmlStringToCds` method is similar to corresponding method (which adds the items to a list box) and adds the addresses and their textual descriptions by calling:

```
cds.InsertRecord ([0, strAddr, strText]);
```

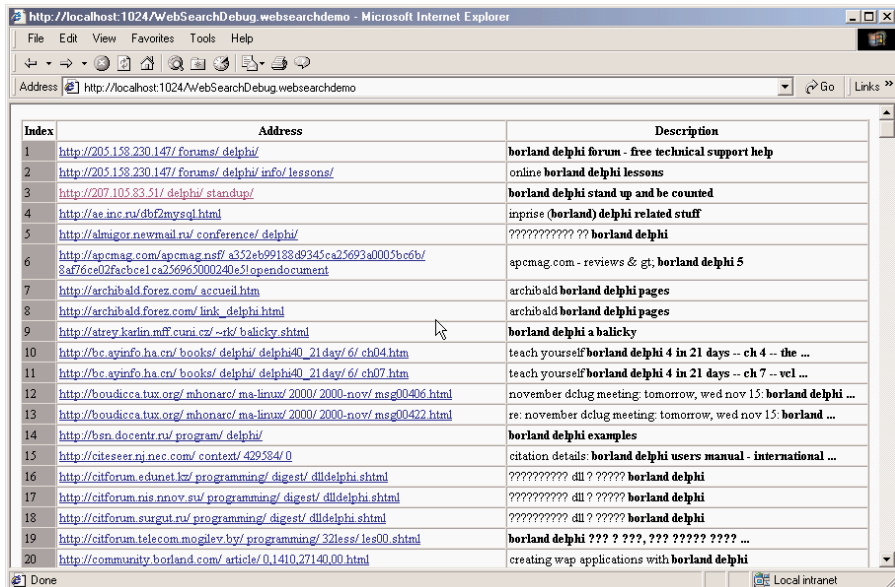
The `ClientDataSet` component, in fact, is set up with three fields: the two strings plus a line counter. This extra empty field is used to have the extra column in the table producer. The code fills the column in the cell-formatting event, which also adds the hyperlink:

```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if CellRow <> 0 then
  case CellColumn of
    0: CellData := IntToStr (CellRow);
    1: CellData := '<a href="' + CellData + '">' + SplitLong(CellData) + '</a>';
    2: CellData := SplitLong (CellData);
  end;
end;
```

The call to `SplitLong` is used to add some extra spaces within the output text, to avoid having grid columns that are too large, as the browser won't split the text on multiple lines unless it contains spaces or other special characters. The result of this program is a rather slow application (because of the multiple HTTP requests it must forward) producing output like Figure 22.10.

**FIGURE 22.10:**

The `WebSearch` program shows the result of the multiple searches done on Google.





## Active Server Pages

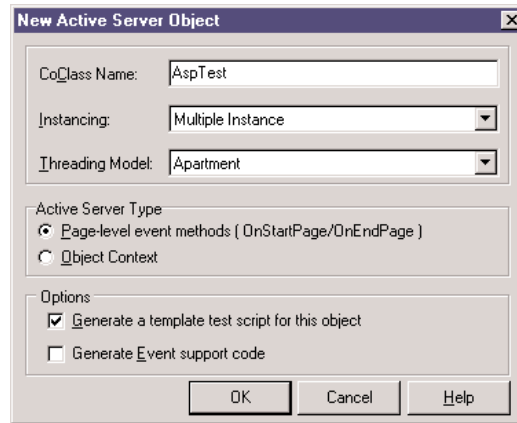
Another approach to the development of server-side applications is the use of scripting. Before looking at the scripting technology embedded in the WebSnap framework, let me shortly discuss Microsoft's Active Server Pages (ASP) technology and how you can use Delphi to support it. The idea behind ASP is to add scripts to the HTML code, so that part of the text on a Web page is directly available while other information can be added at run time on the server. The client receives a plain HTML file. The difference between this approach and ISAPI is that you don't need to recompile a program on the server to see a change; you simply update the script. ASP offers a complex model, where you can attach persistent data to a session (for example, a user moving from page to page of a section of your Web site) and to the entire application (the section of the Web site, regardless of the user).

ASP is quite a complex technology, and here I can only discuss it in relation to Delphi programming. One of the features of ASP is that it allows you to create COM objects within a script, and you can write those COM objects in Delphi. The Delphi IDE even provides specific support classes and a wizard to help you build ASP objects. Compared to ISAPI or CGI, one of the advantages is that your ASP object built in Delphi can get access to session and application information, exactly as an ASP script does. This means we automatically get extra features such as persistent user data built into our server-side object. By building a compiled ASP object, we can also increase the speed of complex server-side code. (ASP scripts are not always the best solution in term of performance.) But, again, I don't want to discuss ASP in detail, only focus on Delphi support.

To try this out, simply create a new ActiveX library, and then start the Active Server Object Wizard (from the ActiveX page of the File > New dialog box). As you can see in Figure 22.11, the wizard has a couple of options. You can build an object integrated with the ASP script by selecting the Page-Level Event Methods radio button, or an internal object (which can be installed as an MTS object) by using the Object Context option. Only in the first case does the object automatically handle the `OnStartPage` method, which receives as parameter the *scripting context*. In both cases, however, the VCL classes you inherit from (`TASPObject` and `TASPMTSObject`, respectively) have properties to access the `Request`, `Response`, `Session`, `Server`, and `Application` ASP objects.

**FIGURE 22.11:**

The new Active Server  
Object wizard



Once you've created the ASP object with the wizard (I've used the Page-Level Event Methods option for the AspTest example), Delphi will bring up the Type Library editor, where you can prepare a list of properties and methods for your ASP object. Simply add the features you need, and then write their code. For example, you can write the following simple test method:

```
procedure Tasptest.ShowData;
begin
    Response.Write ('<h3>Delphi wrote this text</h3>');
end;
```

and activate it from the following ASP script (only slightly modified from the demo script the Delphi wizard will generate for you):

```
<h4>Message</h4>
<% Set DelphiASPObj = Server.CreateObject("asptest1.asptest")
    DelphiASPObj.showData
%>
```

The interesting element is that the same script (or another ASP script of the same application) can also set global values our Delphi object can access. Similarly, multiple objects can communicate, setting global variables for the application and session variables for the specific user. For example, we can add the following text to the ASP page:

```
<h4>hello</h4>
<%
    Session.Value("UserName") = "Marco"
    DelphiASPObj.Hello
%>
```

I've written the code used to set the property and the method invocation one after the other, but they can even be in different pages. This new *dynamic* property (Microsoft's term

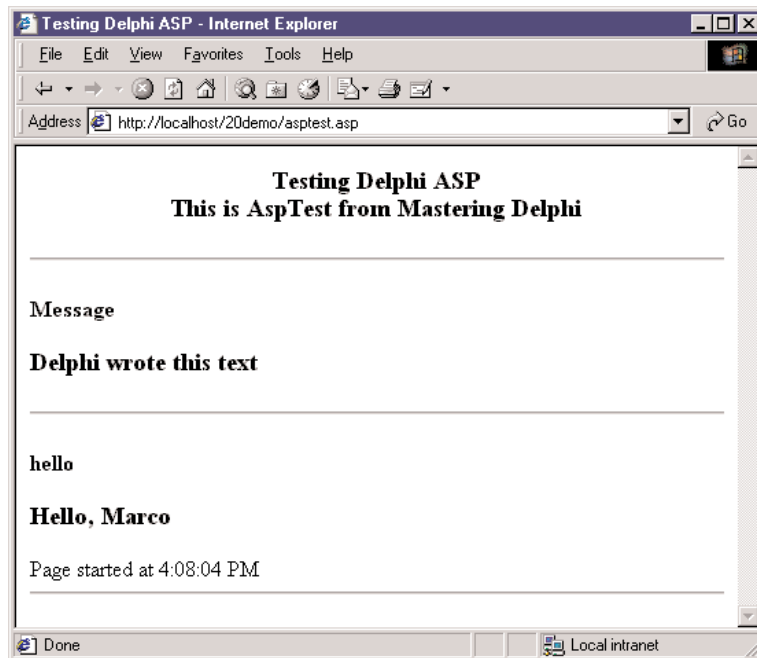
for these values added to an object) is saved in the session, so it depends on the current user. The `Hello` method can use the username to welcome them:

```
procedure Tasptest.Hello;
var
    strName: string;
begin
    strName := Session ['UserName'];
    Response.Write ('<h3>Hello, ' + strName + '</h3>');
    Response.Write ('<p>Page started at ' + TimeToStr (StartTime) + '</p>');
end;
```

You can see the result of this and the previous method combined in Figure 22.12. The last line of the method uses a variable that's set when the page is first loaded, in the `OnStartPage` method (despite the name, this is not an event handler, but a method the ASP engine will call as the page containing the object is activated):

**FIGURE 22.12:**

The Web page generated by the `AspTest` object I've built with Delphi



```
procedure Tasptest.OnStartPage(const AScriptingContext: IUnknown);
begin
    inherited OnStartPage(AScriptingContext);
    StartTime := Now;
end;
```

Technically, this method retrieves the scripting context. The `TASPObject` base class uses the method to initialize all the ASP objects (including the two, `Response` and `Session`, I use in the code), surfacing them as properties.

To generate more complex HTML from the Delphi ASP object, you can use `Producer` components, optionally connecting them to a dataset. In the `AspTest` example, I've added a `Table` component and a `DataSetTableProducer`, connected them as usual, and written the following code to activate it:

```
procedure Tasptest.ShowTable;  
begin  
  DataModule1 := TDataModule1.Create (nil);  
  try  
    Response.Write (DataModule1.DataSetTableProducer1.Content)  
  finally  
    DataModule1.Free;  
  end;  
end;
```

It will actually make more sense to create the data module when the `COM` object is created and destroyed (overriding `Initialize` and `Destroy`) or when the page is loaded and unloaded (with `OnStartPage` and `OnEndPage`).

## WebSnap

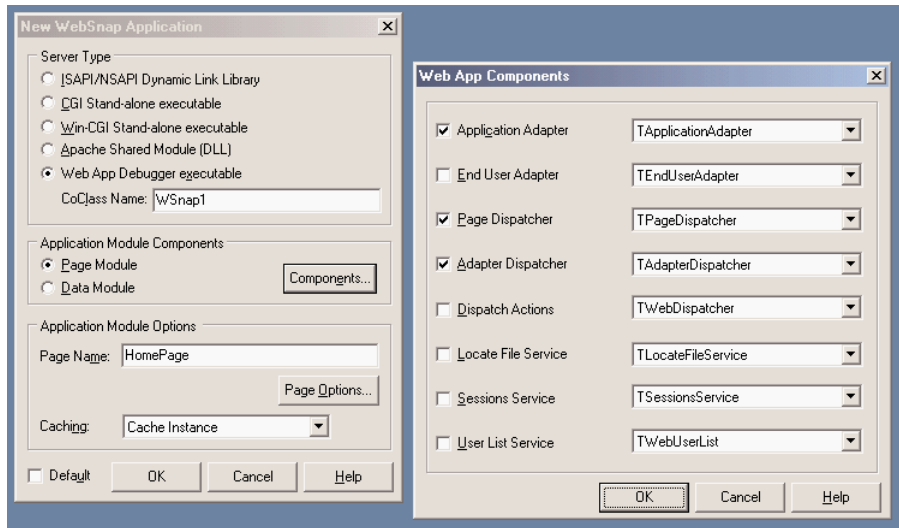
After this lengthy introduction of the core elements of the development of Web server applications with Delphi, we can finally focus on some of the new related technologies introduced in Delphi 6. There were two good reasons for not jumping right into this topic from the beginning of this chapter. The first is that `WebSnap` builds on the foundation offered by `WebBroker`, so that you cannot learn how to use the new features if you don't know the core ones. For example, a `WebSnap` application is technically a `CGI` or `WinCGI` program, or an `ISAPI` or `Apache` module. The second reason is that since `WebSnap` is included only in the Enterprise version of Delphi, not all Delphi programmers have the chance to use it (needing to limit their expense to the Professional version of Delphi 6, which includes `WebBroker`).

`WebSnap` has a few definitive advantages over the plain `WebBroker`, such as allowing for multiple pages, integrating server-side scripting, and `XSL` and Delphi 5 Internet Express technology (these last two elements will be covered in the next chapter). Moreover, there are many ready-to-use components for handling common tasks, such as users' login, session management, and so on. Instead of listing all the features of `WebSnap` right away, though, I've decided to cover them in a sequence of simple and focused applications. All of these applications have been built using the `Web App Debugger`, for testing purposes, but you'll be able to easily deploy them using one of the other available technologies.

The starting point of the development of a WebSnap application is a dialog box that you can invoke either in the WebSnap page of the New items dialog box (File > New > Other) or using the new Internet toolbar of the IDE. The resulting dialog box, shown in Figure 22.13, allows you to choose the type of application (like in a WebBroker application) and to customize the initial application components (but you'll be able to add more later on). The bottom portion of the dialog determines the behavior of the first page, usually the default or home page of the program. A similar dialog box is displayed also for subsequent pages.

**FIGURE 22.13:**

The options offered by the New WebSnap Application dialog box include the type of server and a button for the selection of the core application components.



If you go ahead, choosing the defaults and typing in a name for the home page, the dialog box will create a project and open up a `TWebAppPageModule` for you. This module contains the components you've chosen, by default:

- A `WebAppComponents` component is a container of all of the centralized services of the WebSnap application, such as the user list, core dispatcher, session services, and so on. Not all of its properties must be available, as an application might not need all of the available services.
- One of these core services is offered by the `PageDispatcher` component, which (automatically) holds a list of the available pages of the application and defines the default one.
- Another core service is given by the `AdapterDispatcher` component, which handles HTML form submissions and image requests.
- The `ApplicationAdapter` is the first component we encounter of the *adapters* family. These components offer fields and actions to the server-side scripts evaluated by the

program. Specifically, the `ApplicationAdapter` is a fields adapter that exposes the value of its own `ApplicationTitle` property. By entering a value for this property, it will be made available to the scripts.

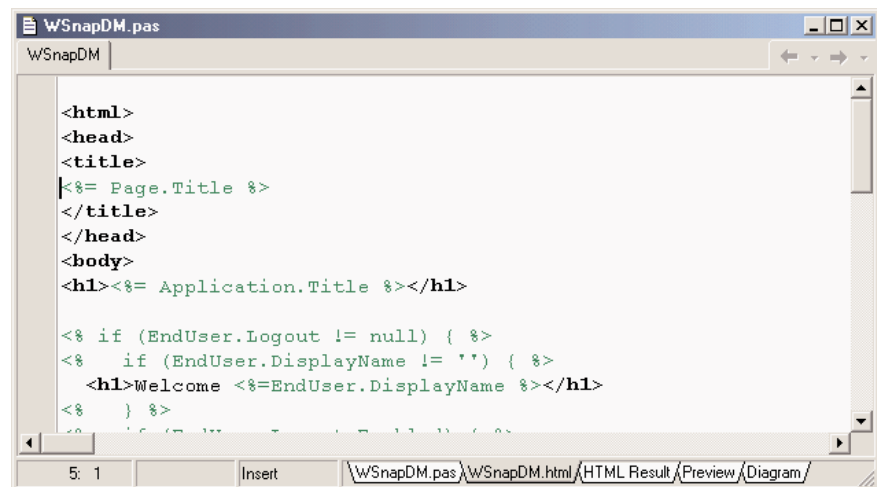
- Finally, the module hosts a `PageProducer` that includes the HTML code of the page—in this case, the default page of the program. Unlike `WebBroker` applications, the HTML for this component is not stored inside its `HTMLDoc` string list property or referenced by its `HTMLFile` property. The HTML file is an external file, stored by default in the folder hosting the source code of the project and referenced from the application using a statement similar to a resource include statement: `{*.html}`.

Because the HTML file included by the `PageProducer` is kept as a separate file (the `LocateFileService` component will eventually help you for its deployment), you can edit it to change the output of a page of your program without having to recompile the application. These possible changes relate not only to the fixed portion of the HTML file but also to some of its dynamic content, thanks to the support for server-side scripting. The default HTML file, based on a standard template, actually already has some scripting in it.

The HTML file is visible within the Delphi editor with reasonably good syntax highlighting, simply by selecting the corresponding lower tab, such as `WSnapDM.html` in my simple example, shown in Figure 22.14. The editor also has other pages for a `WebSnap` module, including by default an HTML Result page, where you can see the HTML generated after evaluating the scripts, and a Preview page hosting what a user will see inside a browser.

**FIGURE 22.14:**

The Delphi 6 editor for a `WebSnap` module includes a simple HTML editor and a preview of its output.



**TIP**

If you prefer editing the HTML of your Web application with another more sophisticated editor, you can set up your choice in the Internet page of the Environment Options dialog box. Within this page, you can see a list of file extensions. Selecting the Edit button for one of these groups of extensions, you can choose an external editor to use for these files. At this point, the External Editor button of the Internet toolbar will become active.

The standard HTML template used by WebSnap adds to any page of the program its title and the application title, using simple script lines such as:

```
<h1><%= Application.Title %></h1>
<h2><%= Page.Title %></h2>
```

We'll get back to the scripting in a while. But let me start the development of the WSnap1 example by simply creating a program with multiple pages. Before I do this, let me finish this overview by showing you the extra source code of a sample Web page module:

```
type
  Thome = class(TWebAppPageModule)
  ...
end;

function home: Thome;

implementation

{$R *.dfm} {*.html}

uses WebReq, WebCntxt, WebFact, Variants;

function home: Thome;
begin
  Result := Thome(WebContext.FindModuleClass(Thome));
end;

initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(TWebAppPageModuleFactory.Create(
      Thome, TWebPageInfo.Create([wpPublished {}, wpLoginRequired]), '.html'),
      caCache));
end.
```

The module uses a global function instead of a typical global object of forms to support caching of the pages. This Web App Debugger application also has some extra code in the initialization section, particularly some registration code, to let the application know the role of the page and its behavior.

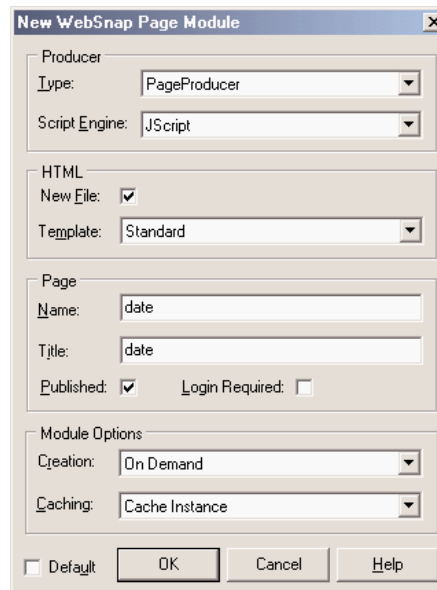
## Managing Multiple Pages

The first notable difference between WebSnap and WebBroker is that, instead of having a single data module with multiple actions eventually connected to producer components, WebSnap has multiple data modules, each corresponding to an action and having a producer component with an HTML file attached to it. Actually, you can still add multiple actions to a page/module, but the idea is that you structure applications around pages and not around actions. Like actions, the name of the page is indicated in the request path.

As an example, I've added to the WebSnap application, built with default settings, two more pages. For the first, in the New WebSnap Page Module dialog (see Figure 22.15), I've chosen a standard page producer and given to it the name *date*. For the second, I've gone with a DataSetPageProducer and given it the name *country*. After saving the files, you can start testing the application. Thanks to some of the scripting I'll discuss later, each page lists all of the available pages (unless you've unchecked the Published check box in the New WebSnap Page Module dialog).

**FIGURE 22.15:**

The New WebSnap Page Module dialog box



All of the pages will be rather empty, but at least we have the structure in place. To complete the home page, I've simply edited its linked HTML file directly. For the date page, I've



employed the same approach as a WebBroker application. I've added to the HTML text some custom tags, as in:

```
<p>The time at this site is <#time>.</p>
```

and I've added some code to the OnTag event handler of the producer component to replace this tag with the current time.

For the third page, the country page, I've modified the HTML to include tags for the various fields of the country table, as in:

```
<h3>Country: <#name></h3>
```

Then I've attached the table to the page producer (and I've also added a session component to account for concurrent requests in multiple threads):

```
object DataSetPageProducer: TDataSetPageProducer
    DataSet = Table1
end
object Table1: TTable
    DatabaseName = 'DBDEMOS'
    SessionName = 'Session1_2'
    TableName = 'country.db'
end
object Session1: TSession
    Active = True
    AutoSessionName = True
end
```

To open this table when the page is first created and reset it to the first record in further invocations, I've handled the OnBeforeDispatchPage event of the Web page module, adding this code to it:

```
Table1.Open;
Table1.First;
```

The fact that a WebSnap page can be very similar to a portion of a WebBroker application (basically an action tied to a producer) is quite important, in case you want to port existing WebBroker code to this new architecture. You can even port your existing DataSetTableProducer components to the new architecture. Technically, you can generate a new page, remove its producer component, replace it with a DataSetTableProducer, and hook this component to the PageProducer property of the Web page module. In practice, this approach would cut out the HTML file of the page and its scripts.

In the WSnap1 program, I've used a better technique. I've added a custom tag (<#htmltable>) to the HTML file and used the OnTag event of the page producer to add to the HTML the result of the data set table:

```
if TagString = 'htmltable' then
    ReplaceText := DataSetTableProducer1.Content;
```

## Server-Side Scripts

If having multiple pages in a server-side program, each associated with a different page module, changes the way you write a program, having the server-side scripts at hand offers an even more powerful approach. For example, the standard scripts of the `WSnap1` example account for the application and page titles, and for the index of the pages. This is generated by an enumerator, the technique used to scan a list from within a WebSnap script code. Let's have a look at it:

```
<table cellpadding="0" cellspacing="0"><td>
<% e = new Enumerator(Pages)
   s = ''
   c = 0
   for (; !e.atEnd(); e.moveNext())
   {
       if (e.item().Published)
       {
           if (c > 0) s += '&nbsp;|&nbsp;';
           if (Page.Name != e.item().Name)
               s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
           else
               s += e.item().Title
           c++
       }
   }
   if (c>1) Response.Write(s)
%>
</td></table>
```

### NOTE

Typically, WebSnap scripts are written in JavaScript, an object-based language very common for Internet programming because it is the only scripting language generally available in browsers (on the client side). JavaScript, technically indicated as ECMAScript, borrows the core syntax of the C language and has almost nothing to do with Java. Actually, WebSnap uses Microsoft's ActiveScripting engine, which supports both JScript (a variation of JavaScript) and VBScript.

Inside the single cell of this table (which, oddly enough, has no rows), the script outputs a string with the `Response.Write` command. This string is built with a `for` loop over an enumerator of the pages of the application, stored in the `Pages` global entity. The title of each page is added to the string, only if the page is published and using an hyperlink only for pages different than the current one. Having this code in a script, instead of hard-coded into a Delphi component, allows you to pass it over to a good Web designer to turn it into something a little more visually appealing.

**TIP**

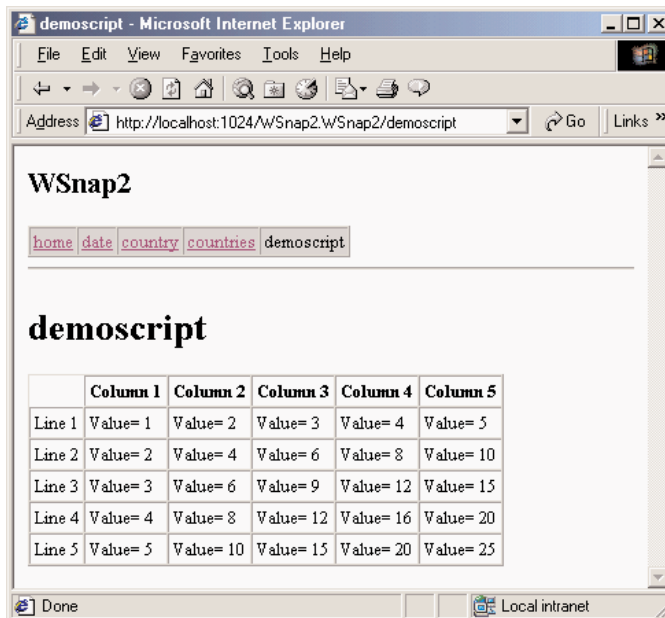
To publish or unpublish a page, don't look for a property in the Web page module. This status is controlled by a flag of the `AddWebModuleFactory` method called in the Web page module initialization code. Simply comment or uncomment this flag to obtain the desired effect.

As a sample of what you can do with scripting, I've added to the WSNap2 example (an extension of the WSNap1 example) a *demoscrypt* page. The script of this page can generate a full table of multiplied values with the following scripting code (see Figure 22.16 for its output):

```
<table border=1 cellspacing=0>
<tr>
  <th>&nbsp;</th>
  <% for (j=1;j<=5;j++) { %>
  <th>Column <%=j %></th>
  <% } %>
</tr>
<% for (i=1;i<=5;i++) { %>
<tr>
  <td>Line <%=i %></td>
  <% for (j=1;j<=5;j++) { %>
  <td>Value= <%=i*j %></td>
  <% } %>
</tr>
<% } %>
</table>
```

**FIGURE 22.16:**

The WSNap2 example has a custom menu stored in an included file reference by each page.



In this script, the `<%=` symbol replaces the longer `Response.Write` command. Another important feature of server-side scripting is the inclusion of pages within other pages. For example, if you plan on modifying the menu, you can include the related HTML and script in a single file, instead of changing it and maintaining it in multiple pages. File inclusion is generally done with a statement like:

```
<!-- #include file="menu.html" -->
```

In Listing 22.1, you can find the complete source code of the include file for the menu, referenced by all the other HTML files of the project. In Figure 22.16, you can see an example of this menu, across the top of the page with the table generation script mentioned earlier.



### Listing 22.1: The *menu.html* file included in each page of the WSnap2 example

```
<html>
<head>
<title><%= Page.Title %></title>
</head>
<body>
<h2><%= Application.Title %></h2>
<table cellspacing="0" cellpadding="2" border="1" bgcolor="#c0c0c0">
<tr>
<% e = new Enumerator(Pages)
   for (; !e.atEnd(); e.moveNext())
   {
     if (e.item().Published)
     {
       if (Page.Name != e.item().Name)
         Response.Write ('<td><a href="' + e.item().HREF + '">' +
           e.item().Title + '</a></td>')
       else
         Response.Write ('<td>' + e.item().Title + '</td>')
     }
   }
%>
</tr>
</table>
<hr>
<h1><%= Page.Title %></h1>
<p>
```

This script for the menu uses the `Pages` list and the `Page` and `Application` global scripting objects. WebSnap makes available a few other global objects, including `EndUser` and `Session` objects (in case you add the corresponding adapters to the application), the `Modules` object, and the `Producer` object, which allows access to the `Producer` component of the Web page module. The script also has available the `Response` and `Request` objects of the Web module.

## Adapters

Besides these global objects, within a script you can access all the adapters available in the corresponding Web page module. (Adapters in other modules, including shared Web data modules, must be referenced by prefixing their name with the `Modules` object and the corresponding module.) The idea is that adapters allow you to pass information from your compiled Delphi code to the interpreted script, providing a scriptable interface to your Delphi application. Adapters contain fields that represent data and host actions that represent commands. The server-side scripts can access these values and issue these commands, passing specific parameters to them.

### NOTE

Technically, adapters implement an `IDispatch` interface that can be accessed by the script through an Active Scripting engine language, such as JavaScript. The page producer component is responsible for invoking the Active Scripting engine and has a property indicating the language of the script. Because of this, you'll have to register two type libraries (and deploy the corresponding DLLs) to make this work on a machine where Delphi is not installed: `WebBrokerScript.tlb` and `stdole140.dll`. As the first is a type library, it must be installed with Delphi's `TRegSvr` utility (available in the `bin` subfolder) rather than Microsoft's `RegSvr32` program. Of course, the server computer must also have Microsoft Active Scripting Engine installed in order to work.

## Adapter Fields

For simple customizations, you can simply add new fields to the specific adapters. For instance, in the `WSnap2` example, I've added a custom field to the application adapter. After selecting this component, you can either open up its Fields editor (accessible via its local menu) or simply work within the Object TreeView. After adding a new field (called `Count` in the example), you can assign a value to it in its `OnGetValue` event. As I want to count the hits (or requests) on any page of the Web application, I've also handled the `OnBeforePageDispatch` event of the *global* `PageDispatcher` component. Here is the code of the two methods:

```
procedure Thome.PageDispatcherBeforeDispatchPage(Sender: TObject;  
    const PageName: String; var Handled: Boolean);  
begin  
    Inc (HitCount);  
end;  
  
procedure Thome.CountGetValue(Sender: TObject; var Value: Variant);  
begin  
    Value := HitCount;  
end;
```

Of course, I could have used the page name to also count hits on each specific page (and I could have added some support for persistency, as the count is reset every time you run a new instance of the application). Now that I've added a custom field to an existing adapter (corresponding to the `Application` script object), I can access it from within any script, like this:

```
<p>Application hits since last activation:  
<%= Application.Count.Value %></p>
```

## Adapter Components

In the same way, you can also add custom adapters to specific pages. If you need to pass along a few fields, use the generic Adapter component. Other custom adapters (besides the global `ApplicationAdapter` we've already used) include these:

- The `PagedAdapter` component has built-in support for showing its content over multiple pages.
- The `DataSetAdapter` component is used to access a Delphi dataset from a script and is covered in the next section.
- The `StringValuesList` holds a list of name/value pairs, like a string list, and can be used directly or to provide a list of values to an adapter field. The inherited `DataSetValuesList` adapter has the same role but grabs the list of name/value pairs from a dataset, providing support for lookups and other selections.
- User-related adapters, such as the `EndUser`, `EndUserSession`, and `LoginForm` adapters, are used to access user and session information and to build a login form for the application, automatically tied to the users list. I'll cover these adapters in the section "Sessions, Users, and Permissions" later in this chapter.

## Using the AdapterPageProducer

Most of these components are used in conjunction with an `AdapterPageProducer` component. The `AdapterPageProducer`, in fact, can generate portions of script after you visually design the desired result. As an example, I've added to the `WSnap2` application the *inout* page, which has an adapter with two fields, one standard and one Boolean:

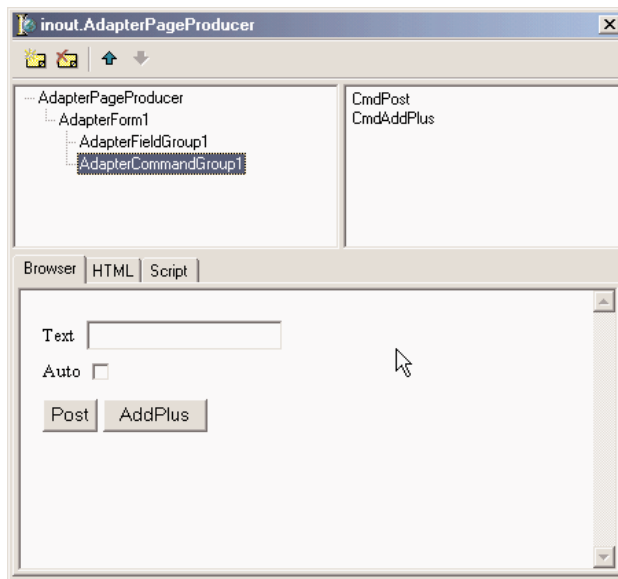
```
object Adapter1: TAdapter  
  OnBeforeExecuteAction = Adapter1BeforeExecuteAction  
  object TAdapterActions  
    object AddPlus: TAdapterAction  
      OnExecute = AddPlusExecute  
    end  
    object Post: TAdapterAction  
      OnExecute = PostExecute  
    end  
  end  
end
```

```
object TAdapterFields
  object Text: TAdapterField
    OnGetValue = TextGetValue
  end
  object Auto: TAdapterBooleanField
    OnGetValue = AutoGetValue
  end
end
end
```

The adapter has also a couple of actions, used to post the current user input and to add a plus sign to the text. The same plus sign is added anyway when the Auto field is enabled. Developing the user interface for this form, and the related scripting, would take some time using plain HTML. But the AdapterPageProducer component (used in this page) has an integrated HTML designer, which Borland calls Web Surface Designer. Using this tool, you can visually add a form to the HTML page and add an AdapterFieldGroup to it. Connect this field group to the adapter to have editors for the two fields automatically displayed. Then you can add an AdapterCommandGroup and connect it to the AdapterFieldGroup, to have buttons for all of the actions of the adapter. You can see an example of this designer in Figure 22.17.

**FIGURE 22.17:**

The Web Surface Designer of Delphi 6 for the *inout* page of the WSnap2 example, at design time



To be more precise, the fields and buttons are automatically displayed if the `AddDefaultFields` and `AddDefaultCommands` properties of the field group and command group are set. The effect of the visual operations I've done to build this form are summarized in the following DFM snippet:

```

object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = Adapter1
      object FldText: TAdapterDisplayField
        FieldName = 'Text'
      end
      object FldAuto: TAdapterDisplayField
        FieldName = 'Auto'
      end
    end
  object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterFieldGroup1
    object CmdPost: TAdapterActionButton
      ActionName = 'Post'
    end
    object CmdAddPlus: TAdapterActionButton
      ActionName = 'AddPlus'
    end
  end
end

```

Now that we have an HTML page with some scripts to move data back and forth and issue commands, we can have a look at the source code required to make this work. First, you'll have to add to the class two local fields to store the adapter fields and manipulate them, and you need to implement the `OnGetValue` event for both, returning the field values. When each of the buttons is pressed, we have to retrieve the text passed by the user, which is not automatically copied into the corresponding adapter field. You can obtain this effect by looking at the `ActionValue` property of these fields, which is set only if something was entered (for this reason, when nothing is entered we set the Boolean field to False). To avoid repeating this code for both actions, I've placed it in the `OnBeforeExecuteAction` event of the Web page module:

```

procedure Tinout.Adapter1BeforeExecuteAction(Sender, Action: TObject;
  Params: TStrings; var Handled: Boolean);
begin
  if Assigned (Text.ActionValue) then
    fText := Text.ActionValue.Values [0];
    fAuto := Assigned (Auto.ActionValue);
  end;

```



Notice that each action can have multiple values (in case of components allowing multiple selections); but this is not the case, so we can simply grab the first element. Finally, I've written the code for the `OnExecute` events of the two actions:

```
procedure Tinout.AddPlusExecute(Sender: TObject; Params: TStrings);  
begin  
    fText := fText + '+';  
end;  
  
procedure Tinout.PostExecute(Sender: TObject; Params: TStrings);  
begin  
    if fAuto then  
        AddPlusExecute (Self, nil);  
end;
```

As an alternative, adapter fields have a public `EchoActionFieldValue` property that you can set to get the value entered by the user and place it again in the resulting form. This technique is typically used in case of errors, to let the user change the input starting with the values already entered.

#### NOTE

The `AdapterPageProducer` component has specific support for cascading style sheets (CSS). You can define the CSS for a page using either the `StylesFile` property or `Styles` string list. Any element of the editor of the items of the producer, at this point, can define a specific style or choose one of the styles of the attached CSS. This last operation (which is the suggested approach) is accomplished using the `StyleRule` property.

## Scripts Rather Than Code?

Even this simple example of the combined use of an adapter and an adapter page producer, with its visual designer, shows the power of this architecture. However, this approach also has a big drawback. By letting the components generate the script (in the HTML, you have only the `<#SERVERSCRIPT>` tag), you save a lot of development time, but at the same time you end up mixing the script with the code, so that changes to the user interface will require updating the program. The division of responsibilities between the Delphi application developer and the HTML/script designer is lost. And, ironically, we end up having to run a script to accomplish something the Delphi program could have done right away, possibly even much faster!

So my opinion is that this is a very powerful architecture and a huge step forward from `WebBroker`, but it has to be used with some care, to avoid misusing some of these technologies just because they are simple and powerful (and they are indeed). For example, it might be worth using the designer of the `AdapterPageProducer` to generate the first version of a page, then grabbing the generated script and copying to the HTML of a plain `PageProducer`, so that a Web designer can modify the script with a specific tool.

For nontrivial applications, I tend to prefer the possibilities offered by XML and XSL, which are available within this architecture even if they don't have a central role. More on this specific topic in the next chapter.

## WebSnap and Databases

One of the areas where Delphi has always shined is database programming. For this reason, it is not surprising to see a lot of support for handling datasets within the WebSnap framework. Specifically, you can use the `DataSetAdapter` component to connect to a dataset and display its values in a form or a table using the visual editor of the `AdapterPageProducer` component.

### A WebSnap Data Module

As an example, I've built a new WebSnap application (called `WSnapTable`) with an `AdapterPageProducer` as its main page to display a table in a grid and another `AdapterPageProducer` in a secondary page to show a form with a single record. I've also added to the application a WebSnap Data Module, as a container of the dataset components. The data module has a `ClientDataSet` wired to a `dbExpress` dataset through a provider and based on an `InterBase` connection, as shown here:

```
object ClientDataSet1: TClientDataSet
  Active = True
  ProviderName = 'DataSetProvider1'
end
object SQLConnection1: TSQLConnection
  Connected = True
  ConnectionName = 'IBLocal1'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText =
    'select CUST_NO, CUSTOMER, ADDRESS_LINE1, CITY, STATE_PROVINCE, ' +
    ' COUNTRY from CUSTOMER'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
```

### The DataSetAdapter

Now that we have a dataset available, we can add a `DataSetAdapter` to the first page, and connect it to the `ClientDataSet` of the Web module. The adapter automatically makes available all of the fields of the dataset and several predefined actions for operating on it (such as

Delete, Edit, and Apply). You can add them explicitly to the Actions and Fields collections to exclude some of them and customize their behavior, but this is not always required.

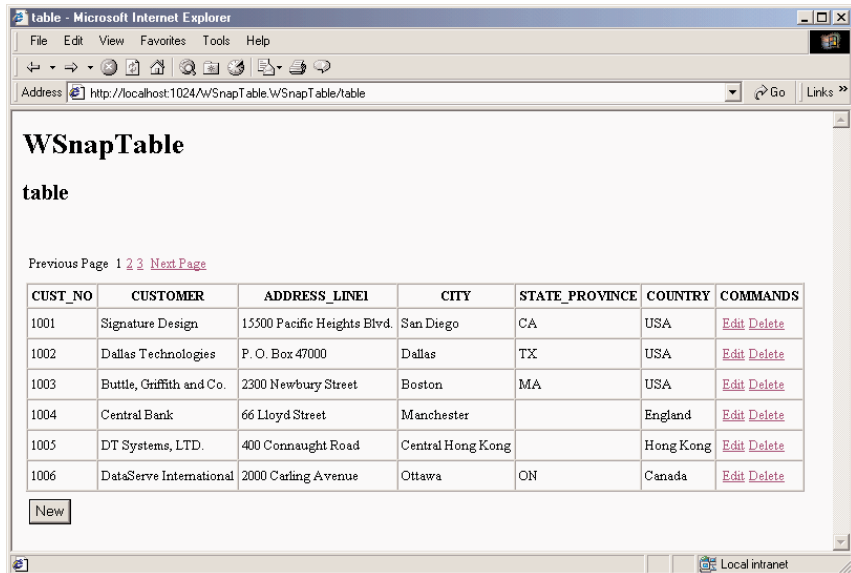
Like the PagedAdapter, the DataSetAdapter has a PageSize property where you can indicate the number of elements to display in each page. The component also has commands that you can use to navigate among pages. This approach is particularly suitable when you want to display a large dataset in a grid. These are the adapter settings for the main page of the WSnapTable example:

```
object DataSetAdapter1: TDataSetAdapter
  DataSet = WebDataModule1.ClientDataSet1
  PageSize = 6
end
```

The corresponding page producer has a form containing two command groups and a grid. The first command group (displayed above the grid) has the predefined commands for handling pages: CmdPrevPage, CmdNextPage, and CmdGotoPage. This last command generates a list of numbers for the pages, so that a user can jump to each of them directly. The AdapterGrid component has the default columns plus an extra one hosting a couple of commands, Edit and Delete. The bottom command group has a button used to create a new record. You can see an example of the output of the table in Figure 22.18 and the complete settings of the AdapterPageProducer in Listing 22.2.

**FIGURE 22.18:**

The page shown by the WSnapTable example at start up includes the initial portion of a *paged* table.



**Listing 22.2:** AdapterPageProducer settings for the WSnapTable main page

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
```

```
object AdapterCommandGroup1: TAdapterCommandGroup
  DisplayComponent = AdapterGrid1
  object CmdPrevPage: TAdapterActionButton
    ActionName = 'PrevPage'
    Caption = 'Previous Page'
  end
  object CmdGotoPage: TAdapterActionButton
    ActionName = 'GotoPage'
  end
  object CmdNextPage: TAdapterActionButton
    ActionName = 'NextPage'
    Caption = 'Next Page'
  end
end
object AdapterGrid1: TAdapterGrid
  TableAttributes.CellSpacing = 0
  TableAttributes.CellPadding = 3
  Adapter = DataSetAdapter1
  AdapterMode = 'Browse'
  object ColCUST_NO: TAdapterDisplayColumn
    FieldName = 'CUST_NO'
  end
  object ColCUSTOMER: TAdapterDisplayColumn
    FieldName = 'CUSTOMER'
  end
  object ColADDRESS_LINE1: TAdapterDisplayColumn
    FieldName = 'ADDRESS_LINE1'
  end
  object ColCITY: TAdapterDisplayColumn
    FieldName = 'CITY'
  end
  object ColSTATE_PROVINCE: TAdapterDisplayColumn
    FieldName = 'STATE_PROVINCE'
  end
  object ColCOUNTRY: TAdapterDisplayColumn
    FieldName = 'COUNTRY'
  end
end
object AdapterCommandColumn1: TAdapterCommandColumn
  Caption = 'COMMANDS'
  object CmdEditRow: TAdapterActionButton
    ActionName = 'EditRow'
    Caption = 'Edit'
    PageName = 'formview'
    DisplayType = ctAnchor
  end
  object CmdDeleteRow: TAdapterActionButton
    ActionName = 'DeleteRow'
    Caption = 'Delete'
    DisplayType = ctAnchor
  end
end
end
```

```
end
object AdapterCommandGroup2: TAdapterCommandGroup
  DisplayComponent = AdapterGrid1
  object CmdNewRow: TAdapterActionButton
    ActionName = 'NewRow'
    Caption = 'New'
    PageName = 'formview'
  end
end
end
end
end
```

---

In this rather long listing, there are a few things to notice. First, the grid has the `Adapter-Mode` property set to `Browse`, other possibilities being `Edit`, `Insert`, and `Query`. This dataset display mode for adapters determines the type of user interface (text or edit boxes and other input controls) and the visibility of other buttons (for example, `Apply` and `Cancel` buttons are only present in the edit view, the opposite for the `Edit` command).

---

**NOTE**

The adapter mode can also be modified using server-side script and accessing `Adapter.Mode`.

Second, I've modified the display of the commands inside the grid, using the `ctAnchor` value for the `DisplayType` property instead of the default button style. Similar properties are available in most components of this architecture to tweak the HTML code they produce.

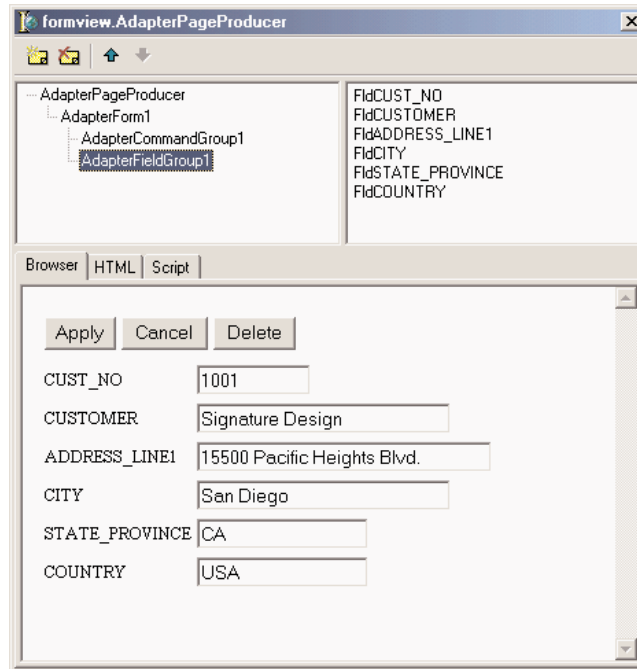
## Editing the Data in a Form

Finally, some of the commands are connected to a different page, the page that is going to be displayed after the commands are invoked. For example, the `edit` command has its `PageName` property set to `formview`. This second page of the application has an `AdapterPageProducer` with components hooked to the same `DataSetAdapter` of the other table, so that all of the request will be automatically synchronized. Selecting the `edit` command, in fact, the program will open the secondary page displaying the data of the record corresponding to the command.

Listing 22.3 shows the details of the page producer of the second page of the program. Again, building the HTML form visually using the Delphi specific designer (see Figure 22.19) was a very fast operation.

**FIGURE 22.19:**

The formview page shown by the WSnapTable example at design time, in the Web Surface Designer (or AdapterPageProducer editor)



### Listing 22.3: AdapterPageProducer settings for the formview page

```

object AdapterPageProducer: TAdapterPageProducer
object AdapterForm1: TAdapterForm
  object AdapterErrorList1: TAdapterErrorList
    Adapter = table.DataSetAdapter1
  end
object AdapterCommandGroup1: TAdapterCommandGroup
  DisplayComponent = AdapterFieldGroup1
  object CmdApply: TAdapterActionButton
    ActionName = 'Apply'
    PageName = 'table'
  end
object CmdCancel: TAdapterActionButton
  ActionName = 'Cancel'
  PageName = 'table'
end
object CmdDeleteRow: TAdapterActionButton
  ActionName = 'DeleteRow'
  Caption = 'Delete'

```

```

        PageName = 'table'
    end
end
object AdapterFieldGroup1: TAdapterFieldGroup
    Adapter = table.DataSetAdapter1
    AdapterMode = 'Edit'
    object FldCUST_NO: TAdapterDisplayField
        DisplayWidth = 10
        FieldName = 'CUST_NO'
    end
    object FldCUSTOMER: TAdapterDisplayField
        DisplayWidth = 27
        FieldName = 'CUSTOMER'
    end
    object FldADDRESS_LINE1...
    object FldCITY...
    object FldSTATE_PROVINCE...
    object FldCOUNTRY...
end
end
end

```

---

In the listing, you can see that all the operations send the user back the main page and that the `AdapterMode` is set to `Edit`, unless there are update errors or conflicts. In this case, the same page is displayed again, with a description of the errors obtained by adding an `AdapterErrorList` component at the top of the form.

The second page is not published, because selecting it without referring to a specific record would make very little sense. To unpublish the page, I've simply commented the corresponding flag in the initialization code. Finally, to make the changes to the database persistent, you can call the `ApplyUpdates` method in the `OnAfterPost` and `OnAfterDelete` events of the `ClientDataSet` component hosted by the data module. Another problem (which I haven't fixed) relates to the fact that the SQL server assigns the ID of each customer, so that when you enter a new record, the data in the `ClientDataSet` and in the actual database are not aligned any more. This can cause Record Not Found errors, a problem I've not fixed in the example.

## Master/Detail in WebSnap

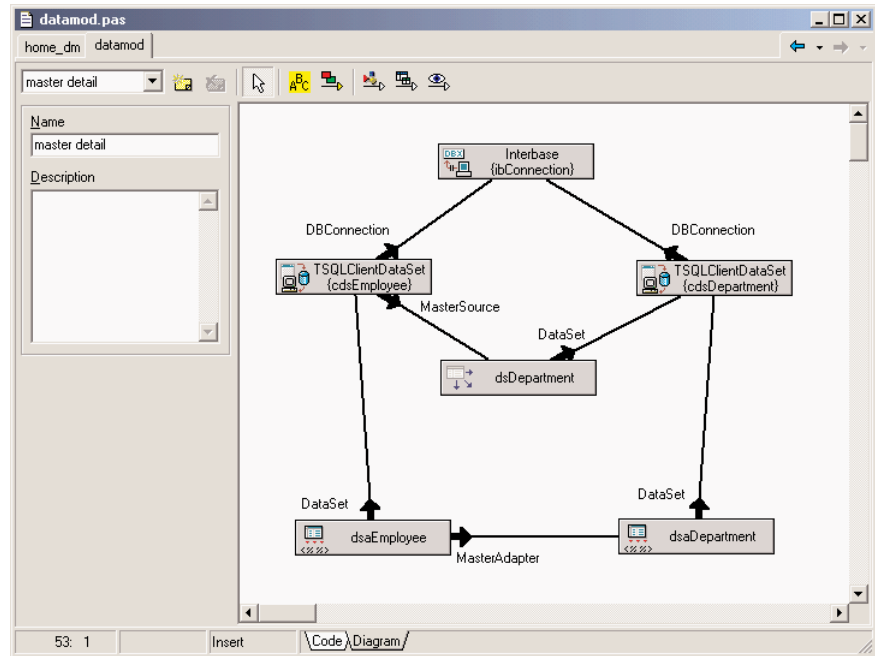
The `DataSetAdapter` component has specific support for master/detail relationships between datasets. After you've created the relationship among the datasets, as usual, define an adapter for each dataset and then connect the `MasterAdapter` property of the adapter of the detail dataset. Setting up the master/detail relationship between the adapters makes them work in a

more seamless way. For example, when you change the work mode of the master, or enter new records, the detail automatically enters into Edit mode or is refreshed.

In the WSnapMD example, I've defined such a relationship using two `SQLClientDataSet` components connected with an `InterBase` database via `dbExpress`. All these components and the related adapters are in a Web data module, which has the structure displayed in the design view in Figure 22.20. I haven't provided a complete listing of the details of these components, as it shouldn't be too difficult for you to rebuild it after looking at the example itself.

**FIGURE 22.20:**

The design view of the Web data module of the WSnapMD example. Both the datasets and the adapters have a master/detail relationship.



The only page of this WebSnap application has an `AdapterPageProducer` component hooked to both dataset adapters. The form of this page, in fact, has both a field group hooked to the master and a grid connected with the detail. Unlike other examples, I've tried to improve the user interface by adding custom attributes for the various elements, as you can see in the following detailed excerpt:

```

object AdapterPageProducer: TAdapterPageProducer
object AdapterForm1: TAdapterForm
    Custom = 'Border="1" CellSpacing="0" CellPadding="10" ' +
            'BgColor="Silver" align="center" '
object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterFieldGroup1
  
```



```

Custom = 'Align="Center"'
object CmdFirstRow: TAdapterActionButton
  ActionName = 'FirstRow'
  Caption = '  First  '
end
object CmdPrevRow: TAdapterActionButton
  ActionName = 'PrevRow'
  Caption = ' Previous '
end
object CmdNextRow: TAdapterActionButton
  ActionName = 'NextRow'
  Caption = '  Next   '
end
object CmdLastRow: TAdapterActionButton
  ActionName = 'LastRow'
  Caption = '  Last   '
end
end
object AdapterFieldGroup1: TAdapterFieldGroup
  Custom = 'BgColor="Silver"'
  Adapter = WDataMod.dsaDepartment
  AdapterMode = 'Browse'
end
object AdapterGrid1: TAdapterGrid
  TableAttributes.BgColor = 'Silver'
  TableAttributes.CellSpacing = 0
  TableAttributes.CellPadding = 3
  HeadingAttributes.BgColor = 'Gray'
  Adapter = WDataMod.dsaEmployee
  AdapterMode = 'Browse'
object ColEMP_NO: TAdapterDisplayColumn...
object ColFIRST_NAME: TAdapterDisplayColumn...
object ColLAST_NAME: TAdapterDisplayColumn...
object ColDEPT_NO: TAdapterDisplayColumn...
object ColJOB_CODE: TAdapterDisplayColumn...
object ColJOB_COUNTRY: TAdapterDisplayColumn...
object ColSALARY: TAdapterDisplayColumn...
end
end
end

```

I've used a gray background, displayed some of the grid borders (HTML grids are used very often by the Web surface designer), centered most of the elements, and added some spacing. Notice that I've added some extra spaces to the button captions, to avoid them being too small. The effect of these settings (and the master/detail structure) is visible at run time in Figure 22.21.

**FIGURE 22.21:**

The WSnapMD example shows a master/detail structure and has some customized output.



## Sessions, Users, and Permissions

Another very interesting area of the WebSnap architecture is its support for sessions and users. Sessions are supported using a classic approach: temporary cookies. These cookies are sent to the browser, so that following requests from the same user can be acknowledged by the system. By adding data to a session instead of an application adapter, you can have data that depends on the specific session or user (although a user can possibly run multiple sessions by opening multiple browser windows on the same computer). For supporting sessions, the application keeps data in memory, so this feature is not available in case of CGI programs.

### Using Sessions

To underline the importance of this type of support, I've built a WebSnap application with a single page showing both the total number of hits and the total number of hits for each session. The program has a SessionService component with default values for its MaxSessions and DefaultTimeout properties. For every new request, the program increases both an nHits private field of the page module and the SessionHits value of the current session:

**procedure** TSessionDemo.WebAppPageModuleBeforeDispatchPage(Sender: TObject);

```

    const PageName: String; var Handled: Boolean);
begin
    // increase application and session hits
    Inc (nHits);
    WebContext.Session.Values ['SessionHits'] :=
        Integer (WebContext.Session.Values ['SessionHits']) + 1;
end;

```

**NOTE**

The `WebContext` object (of type `TWebContext`) is a thread variable, created by `WebSnap` for each request, which provides thread-safe access to other global variables used by program.

The associated HTML displays status information both by using some custom tags evaluated by the `OnTag` event of the page producer and some script, evaluated by the engine. Here is the core portion of the HTML file:

```

<h3>Plain Tags</h3>
<p>Session id: <#SessionID>
<br>Session hits: <#SessionHits></p>
<h3>Script</h3>
<p>Session hits (via application): <%=Application.SessionHits.Value%>
<br>Application hits: <%=Application.Hits.Value%></p>

```

The parameters of the output are provided by the `OnTag` event handler and the `OnGetValue` events of the fields:

```

procedure TSessionDemo.PageProducerHTMLTag(Sender: TObject; Tag: TTag;
    const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    if TagString = 'SessionID' then
        ReplaceText := WebContext.Session.SessionID
    else if TagString = 'SessionHits' then
        ReplaceText := WebContext.Session.Values ['SessionHits']
end;

procedure TSessionDemo.HitsGetValue(Sender: TObject; var Value: Variant);
begin
    Value := nHits;
end;

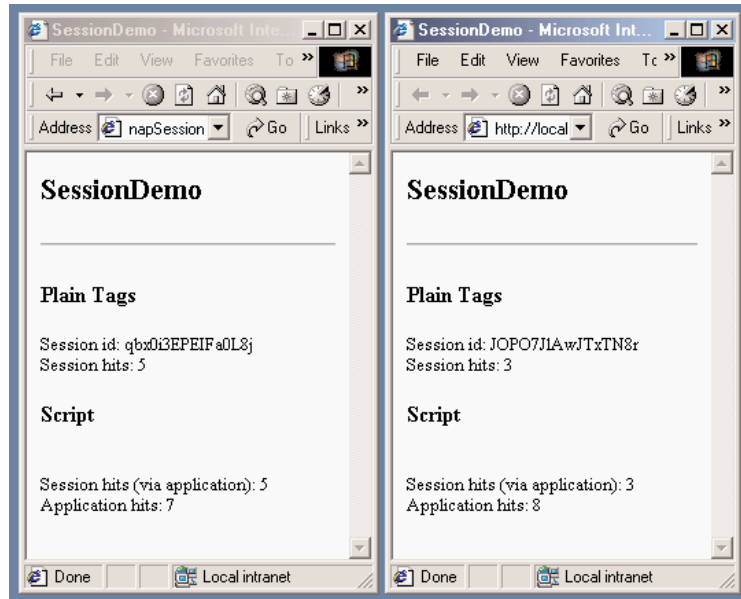
procedure TSessionDemo.SessionHitsGetValue(Sender: TObject; var Value: Variant);
begin
    Value := Integer (WebContext.Session.Values ['SessionHits']);
end;

```

The effect of this program is visible in Figure 22.22, where I've activated two sessions in two different copies of Internet Explorer.

**FIGURE 22.22:**

Two instances of the browser operate on two different sessions of the same WebSnap application.

**TIP**

In this example, I've voluntarily used both the traditional WebBroker tag replacement and the newer WebSnap adapter fields and scripting, so that you can compare the two approaches and keep in mind that they are both available in a WebSnap application.

## Requesting Login

Besides generic sessions, WebSnap also has specific support for users and login-based authorized sessions. You can add to an application a list of users (with the WebUserList component), each with a name and a password. My impression is that this component is rather rudimentary in the data it can store. Instead of filling it with your list of users, however, you can keep the list in a database table (or in some other proprietary format) and use the events of the WebUserList component to retrieve your custom users data and check the user passwords.

You'll generally also add to the application the SessionService and EndUserSession-Adapter components. At this point, you can ask the users to log in, indicating for each page whether it can be viewed by everyone or only by logged-in users. This is accomplished by setting the wpLoginRequired flag in the constructor of the TWebPageModuleFactory and TWebAppPageModuleFactory classes in the initialization code of the Web page unit.

**NOTE**

The reason for having rights and publication information in the factory rather than in the WebPageModule, is that the program can check the access rights and list the pages even without loading the module.

When a user tries to see a page that requires the user identification, the login page indicated in the EndUserSessionAdapter component is displayed. You can create such a page rather easily by creating a new Web page module based on an AdapterPageProducer and adding to it the LoginFormAdapter. In the editor of the page, add a field group within a form, connect the field group to the LoginFormAdapter, and add a command group with the default Login button. The resulting login form will have fields for the username and its password, but also for the requested page. This last value is automatically filled with the requested page, in case this page required authorization and the user wasn't already logged in. This is done so that a user can immediately reach the requested page without being bounced back to a generic menu.

The login form is typically not published, because the corresponding Login command is already available when the user isn't logged into the system; when the user logs in, it is replaced by a Logout command. This is obtained by the standard script of the Web Page Module, and particularly:

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
    <h1>Welcome <%=EndUser.DisplayName %></h1>
  <%   } %>
  <%   if (EndUser.Logout.Enabled) { %>
    <a href="<%=EndUser.Logout.ASHREF%>">Logout</a>
  <%   } %>
  <%   if (EndUser.LoginForm.Enabled) { %>
    <a href="<%=EndUser.LoginForm.ASHREF%>>Login</a>
  <%   } %>
<% } %>
```

There isn't much else to say about the WSnapUsers application, as it has almost no custom code and settings. The access to the users data is demonstrated by the script of the standard template shown above.

## Single Page Access Rights

Besides having pages that require a login for access, you can give specific users the right to see more pages than others. Any user, in fact, has a set of rights separated by semicolons or commas. The user must have all of the rights defined for the requested page generally listed in the ViewAccess and ModifyAccess properties of the adapters, which indicate respectively whether the user can see the given elements while browsing or can even edit them. These settings are very granular, and can be applied to entire adapters or some specific adapter

fields (notice I'm referring to the adapter fields, not the user interface components within the designer). For example, you can hide some of the columns of a table to given users by hiding the corresponding fields (and also in other cases, as specified by the `HideOptions` property).

The global `PageDispatcher` component also has the `OnCanViewPage` and `OnPageAccessDenied` events that can also be used to control the access to the various pages of the program within the program code, allowing for even greater control.

## What's Next?

In this chapter, I've covered Web server applications, using multiple techniques (CGI, ISAPI, Apache dynamic modules) and two different frameworks of the Delphi class library: `WebBroker` and `WebSnap`. This wasn't certainly an in-depth presentation, as one could write an entire book on this topic alone. It was intended as a starting point, and (as usual) I've tried to make the core concepts clear rather than building very complex examples.

If you want to learn more details of the `WebSnap` framework and see different examples in actions, refer to the extensive Delphi demos for this area, in the `\Demos\WebSnap` folder. Some of the other available options, relating to XML, XSL, and client-side scripts, will be examined in the next chapter, where I'll also discuss Web services as a powerful alternative to HTTP/HTML-based distributed applications.

# XML and SOAP

- Introducing XML, Extensible Markup Language
- Working with XML: DOM and SAX
- Delphi 6 and XML: interfaces and mapping
- Internet Express
- Using XSTL
- Web services
- SOAP and WSDL

**B**uilding applications for the Internet means using protocols and creating browser-based user interfaces, as we've done in the preceding two chapters, but also opens up the opportunity of exchanging business documents electronically. The emerging standards for this type of activity all center around the XML document format and include the SOAP transmission protocol, XML schemas for the validation of documents, and XSL for rendering them as HTML.

In this chapter, I'll discuss all of these technologies and the extensive support Delphi 6 offers for them, a series of features collectively known as BizSnap. Since you might not know XML and related technologies, I'll provide a little introduction about each of them, but you should refer to books specifically devoted to each subject to know more. What I won't try to do is to cover *why* this is a revolution for running the IT side of a business and what it opens up. In the conclusion of the chapter, I'll point you to some initiatives you might be interested in tracking.

## Introducing XML

XML, or Extensible Markup Language, is a simplified version of SGML and is getting a lot of attention in the IT world. XML is a *markup language*, meaning it uses symbols to describe its own content—in this case, *tags* consisting of specially defined text enclosed in angle brackets. It is named *extensible* because it allows for free markers (in contrast, for example, to HTML, which has predefined markers). The XML language is a standard promoted by the World Wide Web Consortium (better known as W3C, [www.w3.org](http://www.w3.org)).

### TIP

The XML Recommendation is at [www.w3.org/TR/REC-xml](http://www.w3.org/TR/REC-xml).

XML has been touted as the ASCII of year 2000, to indicate a simple and widespread technology, and also to indicate that XML document is actually a plain text file (optionally with Unicode characters instead of plain ASCII text). The important element of XML is that it is descriptive, as every tag has an almost human-readable name. Here is a small example, in case you've never seen an XML document:

```
<book>
  <title>Mastering Delphi 6</title>
  <author>Cantu</author>
  <publisher>Sybex</publisher>
</book>
```



**WARNING** XML has also a few disadvantages I want to underline from the beginning. The biggest is that without a formal description, a document is worth very little. If you want to exchange documents with another company, you have to agree on what each tag means and also on the semantic meaning of the content. (For example, when you have a quantity, you have to agree on the measurement system or include it in the document.) Another disadvantage is that XML documents are much larger than other formats; using strings for numbers, for example, is far from efficient, and the repeated opening and closing tags eat up a lot of space. The good news is that XML compresses quite well, exactly for the same reasons.

## Core XML Syntax

There are a few technical elements of XML that are worth knowing before discussing its usage within Delphi. Here is a short summary of the key elements of the XML syntax:

- White space (including the space character, carriage return, line feed, and tabs) is generally ignored (as in an HTML document). It is important to format an XML document to make it readable by a human being, but your programs won't care much.
- You can add comments within `<!--` and `-->` markers, which are basically ignored by any XML processor. There are also directives and processing instructions, enclosed within the `<?` and `?>` markers.
- There are a few special or reserved characters you cannot use in the text. The only two symbols you can *never* use are the less-than character (or "left angle bracket," used to delimit a marker) replaced by `&lt;`; and the ampersand character replaced by `&amp;`. Other optional special characters are `&gt;` for the greater-than symbol (right angle bracket), `&apos;` for the single quote, and `&quot;` for the double quote.
- To add non-XML content (for example, binary information or a script), you can use a CDATA section, enclosed within `<![CDATA[` and `]]>`.
- All markers are enclosed by angle brackets, `<` and `>`. Markers are case sensitive (in contrast to HTML).
- For each opening marker, you must have a matching closing marker, denoted by an initial slash character, as in:  

```
<node>value</node>
```
- Markers must not overlap—they must be *properly nested*, as in the first line below (the second line is not correct):

```
<node>xx <nested> yy</nested> </node> // OK  
<node>xx <nested> yy</node> </nested> // WRONG
```

- If a marker has no content (but its presence is important anyway), you can replace the opening and closing markers with a single marker that includes a final or “trailing” slash: `<node/>`.
- Markers can also have attributes, using multiple attribute names followed by a value enclosed within quotes: `<node attrib1="aaa">`.
- Any XML node can have multiple attributes, multiple embedded tags, and only one block of text, representing the value of the node. If it’s technically possible, it is common practice for XML nodes to have either a textual value or embedded tags, and not both. Here is an example of the full syntax of a node:

```
<node attrib1="aaa" attrib2="bbb">
  value1
  <child1>
    value2
  </child1>
</node>
```

- A node can have multiple child nodes with the same tag (tags need not be unique). Attribute names are unique for each node.

## Well-Formed XML

If the elements discussed in the previous section define the syntax of an XML document, they are not enough. A XML document is considered syntactically correct, or *well formed*, if it follows a few extra rules. Notice that this type of check doesn’t guarantee that the content of the document is meaningful, but only the tags are properly laid out.

One of the rules is that each document should have a prologue, indicating that it is indeed an XML document, which version of XML it complies with, and the possibly the type of character encoding. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Possible encodings include various Unicode character sets (such as UTF-8, UTF-16, and UTF-32) or some ISO encodings (such as ISO-10646-xxx or ISO-8859-xxx). The prologue can also include external declarations, the schema used to validate the document, namespace declarations, an associated XSL file, and some internal entity declarations. Refer to XML documentation or books for more information on these topics.

An XML document is well formed if it has a prologue, has a proper syntax (see the rules in the previous section), and has a tree of nodes with a single root. Most tools (including Internet Explorer) check whether a document is well formed when loading it.

**NOTE**

As you can see, XML is more formal and precise than HTML. The W3C is coming along with an XHTML standard that will make HTML documents XML-compliant, for better processing with XML tools. This implies many changes in a typical HTML document, such as avoiding attributes with no values, adding all the closing markers (as `</p>` and `</li>`), adding the slash to stand-alone markers (as `<hr/>` and `<br/>`), proper nesting, and more. An HTML-to-XHTML converter, called HTML Tidy, is hosted by the W3C Web site at [www.w3.org/People/Raggett/tidy/](http://www.w3.org/People/Raggett/tidy/).

## Working with XML

To get acquainted with the format of XML, you can use one of the existing XML editors available on the market. You can also simply type your XML code into Notepad and then try to load it into Internet Explorer to see whether it is correct. In this case, you'll see it within the browser in a tree-like structure.

To speed up this type of operation, I've build the simplest XML editor I could come up with—basically Notepad with some XML syntax-checking and a browser attached to it. The `XmlEditOne` example has a `PageControl` with three pages. The first page, `Settings`, hosts a couple of components where you can insert the path and the name of the file and you want to work with. (The reason for not using a standard dialog will become clear when I show you an extension of the program.) The edit box hosting the complete filename is automatically updated with the path and filename, provided the `AutoUpdate` check box is selected.

The second page hosts a `Memo` control, with the text of the XML file, loaded and saved by clicking the two toolbar buttons. As soon as you load the file, or each time you modify its text, its content is loaded into a `DOM` to let a parser check for its correctness (something that would be quite complex to do with your own code). To parse the code, I've used the `XMLDocument` component available in Delphi 6, which is basically a wrapper around a `DOM` available on the computer and indicated by its `DOMVendor` property. I'll discuss the use of this component in a little more detail in the next section. For the moment, suffice to say you can assign a string list to its `XML` property and activate it to let it parse the XML text and eventually report an error with an exception.

For this specific example, though, this behavior is far from good, because while typing the XML code you'll have temporarily incorrect XML. Still, I prefer not to ask the user to click a button to do the validation, but let it run continuously. As it is not possible to disable the parse exception raised by the `XMLDocument` component, I had to work at a lower level, extracting the `DOMPersist` property (referring to the persistency interface of the `DOM`) after extracting the `IXMLDocumentAccess` interface from the `XMLDocument` component (called `XmlDoc` in this

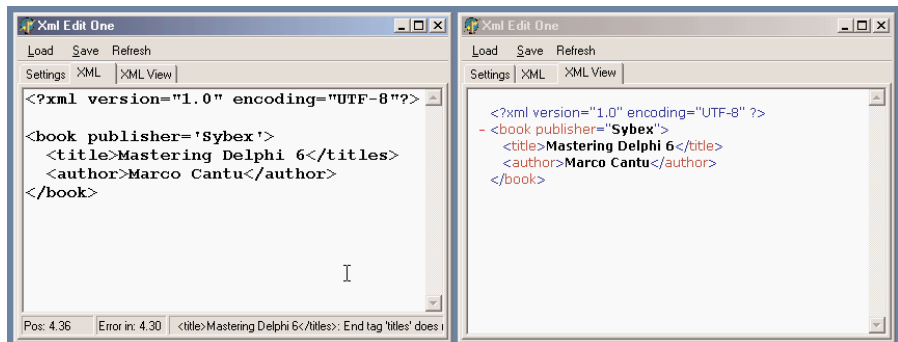
code). At this point, I can also extract the `IDOMParseError` interface from the document component, to display any error message in the status bar:

```
procedure TFormXmlEdit.MemoXmlChange(Sender: TObject);
var
    eParse: IDOMParseError;
begin
    XmlDoc.Active := True;
    xmlBar.Panels[1].Text := 'OK';
    xmlBar.Panels[2].Text := '';
    (XmlDoc as IXMLDocumentAccess).DOMPersist.loadxml(MemoXml.Text);
    eParse := (XmlDoc.DOMDocument as IDOMParseError);
    if eParse.errorCode <> 0 then
        with eParse do
            begin
                xmlBar.Panels[1].Text := 'Error in: ' + IntToStr (Line) + '.' +
                    IntToStr (LinePos);
                xmlBar.Panels[2].Text := SrcText + ': ' + Reason;
            end;
    end;
```

You can see an example of the output of the program in Figure 23.1, alongside the XML tree view provided by the third page (for a correct document). The third page of the program is built using the `WebBrowser` component, which embeds the ActiveX control of Internet Explorer. Unfortunately, there is no direct way to assign a string with the XML text to this control, so you'll have to save the file first and then move to its page to trigger the loading of the XML in the browser (or manually click the Refresh button).

**FIGURE 23.1:**

The `XmlEditOne` example allows you to enter XML text in a memo, indicating errors as you type, and shows the result in the embedded browser.



---

**WARNING** Unicode support in Internet Explorer 5 is quite limited, even on Windows 2000 (which has, in general, rather good Unicode support). If you change the final letter in my last name to an accented letter, as it should be, you won't see any problem when checking the document with the DOM, but you'll see an error when viewing it in the browser. The same happens if you change the format to UTF-16.

## Managing XML Documents

Now that you know the core elements of XML, we can start discussing how to manage XML documents in Delphi programs (or in programs in general, as some of the techniques discussed here go beyond the language used). There are two typical techniques for manipulating XML documents, using a Document Object Model (DOM) interface or using the Simple API for XML (SAX). The two approaches are quite different:

- The DOM loads the entire document into a hierarchical tree of nodes, allowing you to read them and manipulate them to change the document. For this reason, the DOM is suited when you want to navigate the XML structure in memory and edit it, or even for creating new documents from scratch.
- The SAX parses the document firing an event for each element of the document, without building any structure in memory. Once parsed by the SAX, the document is lost, but this operation is generally much faster than the construction of the DOM tree. Using the SAX is good for reading a document once, possibly looking for portion of its data.

There is a third classic way to manipulate (and specifically create) XML documents: string management. Creating a document by adding strings is certainly the fastest operation, particularly if you can do a single pass (and don't need to modify nodes already generated). Even reading documents by means of string functions is very fast, but this can become quite difficult for complex structures.

Finally, Delphi 6 provides two more techniques you should consider. The first is the definition of interfaces mapping the document structure and used to access the document instead of the generic DOM interface. As we'll see, this approach makes for faster coding and more robust applications. Another technique is the development of transformations that allow you to read a generic XML document into a ClientDataSet component or save the dataset into an XML file of a given structure (not the specific XML structure natively supported by the ClientDataSet, or MyBase).

I won't try to fully assess which option is better suited for each type of document and manipulation, but I will try to highlight some of the advantages and disadvantages while discussing examples of each approach in the next sections.

## Programming with the DOM

Since an XML document has a tree-like structure, loading an XML document into a tree in memory is quite a natural fit. This is what the Document Object Model does. The DOM is a standard interface, so that when you have written code that uses a DOM, you can switch DOM implementations without changing your source code (at least, if you haven't used any non-custom extensions).

In Delphi, you can install several DOM implementations, available as COM servers, and use their interfaces. One of the most commonly used DOMs on Windows is the one provided by Microsoft as part of the MSXML SDK. That DOM is also used by Internet Explorer (even if generally in an older version), but the SDK contains some rather detailed documentation, which will probably help you. Other frequently used DOMs are available from IBM and Apache (this one is called Xerces).

---

**Tip**

There are also a couple of native Object Pascal DOM components. One is the open source OpenXML, available at [www.philo.de/xml](http://www.philo.de/xml). Another native Delphi DOM is offered by TurboPower. The advantage of these solutions is that they don't require an external library for the program to execute, because the DOM component gets compiled into your application.

Delphi 6 embeds the DOM implementations into a wrapper component, called XMLDocument. I've just used this component in the preceding example, but here I want to examine its role in a more general way. The idea behind using this component is that, instead of the actual DOM interface, you remain even more independent from the implementations and can work with some simplified methods, or helpers.

The DOM interface, in fact, is quite complex to use. A document is a collection of nodes, each having a name, a text element, a collection of attributes, and a collection of child nodes. Each collection of nodes allows accessing elements by position or searching them by name. Notice that the text within the tags of a node, if any, is rendered as a child of the node and listed in its collection of child nodes. The root node has some extra methods for creating new nodes, values, or attributes.

With Delphi's XMLDocument, you can actually work at two different levels:

- At a lower level, you can use the `DOMDocument` property (of the `IDOMDocument` interface type) to access a standard W3C Document Object Model interface. The official DOM is defined in the `xmldom` unit, and includes interfaces like `IDOMNode`, `IDOMNodeList`,

IDOMAttr, IDOMElement, and IDOMText. With the official DOM interfaces, Delphi supports a lower-level but standard programming model. Notice that the actual DOM implementation will be the one indicated by the XMLDocument component.

- As a higher-level alternative, the XMLDocument component implements also the IXMLDocument interface. This is a custom DOM-like interface defined by Borland in the XMLIntf unit and comprising interfaces like IXMLNode, IXMLNodeList, and IXMLNodeCollection. This Borland interface simplifies some of the DOM operations by replacing multiple method calls, which are repeated quite often in sequence, with a single property or method.

In the following examples (particularly the DomCreate demo), I'll use both approaches so you can have a better idea of the practical differences among the two approaches.

## An XML Document in a TreeView

The starting point is generally loading a document from a file or creating it from a string, but you can also start with a brand new document. As a first example of the use of the DOM, I've built a program that can load an XML document into a DOM and show its structure in a TreeView control. I've also added to the program, called XmlDomTree on the companion CD, a few buttons with sample code used to access to the elements of a sample file, as an example of accessing the DOM data. Loading the document is actually quite simple, while showing it in a tree requires a recursive function that navigates the nodes and subnodes. Here is the code of the two methods:

```
procedure TFormXmlTree.btnLoadClick(Sender: TObject);  
begin  
  OpenFileDialog1.InitialDir := ExtractFilePath (Application.ExeName);  
  if OpenFileDialog1.Execute then  
    begin  
      XMLDocument1.LoadFromFile(OpenDialog1.FileName);  
      Treeview1.Items.Clear;  
      DomToTree (XMLDocument1.DocumentElement, nil);  
      TreeView1.FullExpand;  
    end;  
end;  
  
procedure TFormXmlTree.DomToTree (XmlNode: IXMLNode; TreeNode: TTreeNode);  
var  
  I: Integer;  
  NewTreeNode: TTreeNode;  
  NodeText: string;  
  AttrNode: IXMLNode;  
begin  
  // skip text nodes and other special cases
```

```

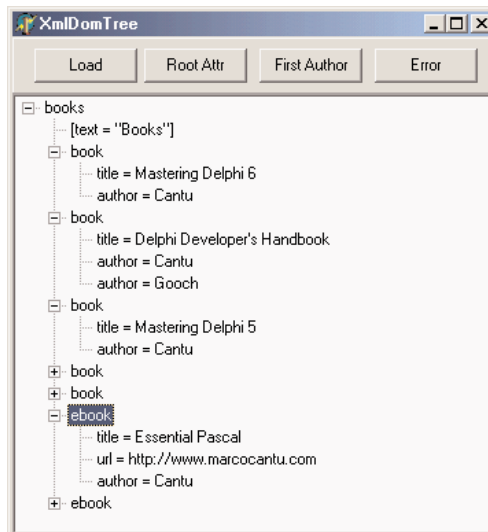
if not (XmlNode.NodeType = ntElement) then
    Exit;
// add the node itself
    NodeText := XmlNode.NodeName;
    if XmlNode.IsTextElement then
        NodeText := NodeText + ' = ' + XmlNode.NodeValue;
    NewTreeNode := TreeView1.Items.AddChild(TreeNode, NodeText);
    // add attributes
    for I := 0 to XmlNode.AttributeNodes.Count - 1 do
    begin
        AttrNode := XmlNode.AttributeNodes.Nodes[I];
        TreeView1.Items.AddChild(NewTreeNode,
            '[' + AttrNode.NodeName + ' = "' + AttrNode.Text + '"']');
    end;
    // add each child node
    if XmlNode.HasChildNodes then
        for I := 0 to XmlNode.ChildNodes.Count - 1 do
            DomToTree (XmlNode.ChildNodes.Nodes [I], NewTreeNode);
    end;

```

This code is quite interesting, as it highlights some of the operations you can do with a DOM. First of all, each node has a `NodeType` property that you can use to determine whether the node is an element, attribute, text node, or special entity (such as CDATA and others). Another aspect is that you cannot access the textual representation of the node, its `NodeValue`, unless it has a text element (notice that the text node will be skipped, as per the initial test). After displaying the name of the item, and then the text value if available, the program (Figure 23.2) shows the content of each attribute directly and of each subnode calling the `DomToTree` method recursively.

**FIGURE 23.2:**

The `XmlDomTree` example can open a generic XML document and show it inside a `TreeView` common control.





Once you have loaded the sample document that accompanies the `XmlDomTree` program (and shown in Listing 23.1) into the `XMLDocument` component, you can use the various methods to access generic nodes, as in tree-building code above, or fetch specific elements. For example, you can grab the value of the attribute `text` of the root node by writing:

```
XMLDocument1.DocumentElement.Attributes ['text']
```

Notice that if there is no attribute called `text`, the call will fail with a rather generic error message, “Invalid variant type conversion,” which helps neither you nor the end user to understand what’s wrong. If you need to access to the first attribute of the root without knowing its name, you can use the following more generic code:

```
XMLDocument1.DocumentElement.AttributeNodes.Nodes[0].NodeValue
```

To access the actual nodes, you use a similar technique, possibly taking advantage of the `ChildValues` array. This is a Delphi extension to the DOM, which allows you to pass as parameter either the name of the element or its numeric position:

```
XMLDocument1.DocumentElement.ChildNodes.Nodes[1].ChildValues['author']
```

This code gets the (first) author of the second book. I cannot use the `ChildValues['book']` expression, as there are multiple nodes with the same name under the root node.



### Listing 23.1: The sample XML document used by examples in this chapter

```
<?xml version="1.0" encoding="UTF-8"?>
<books text="Books">
  <book>
    <title>Mastering Delphi 6</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi Developer's Handbook</title>
    <author>Cantu</author>
    <author>Gooch</author>
  </book>
  <book>
    <title>Mastering Delphi 5</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi COM Programming</title>
    <author>Harmon</author>
  </book>
  <book>
    <title>Thinking in C++</title>
    <author>Eckel</author>
  </book>
</books>
<title>Essential Pascal</title>
```

```

    <url>http://www.marcocantu.com</url>
    <author>Cantu</author>
  </ebook>
  <ebook>
    <title>Thinking in Java</title>
    <url>http://www.mindview.com</url>
    <author>Eckel</author>
  </ebook>
</books>

```

---

## Creating Documents Using the DOM

Although I mentioned earlier that you can create an XML document by chaining together some strings, this is far from a robust technique. Using a DOM to create a document ensures that the XML will be well formed. Also, if the DOM has a schema definition attached, you can validate the structure of the document while adding data to it.

To highlight different cases of document creation, I've built the `DomCreate` example (a program I originally created with the Xerces DOM from the Apache group and changed slightly for supporting Delphi's `XMLDocument`). This program can create XML documents within the DOM, showing the text of them on a memo and optionally in a `TreeView`.

### Tip

The `XMLDocument` component uses the `doAutoIndent` option to improve the output of the XML text to the memo by formatting the XML in a slightly better way. You can choose the type of indentation by setting the `NodeIndentStr` property. For formatting a generic XML text, you can also use the global `FormatXMLData` function using the default setting (2 spaces) as indentation. Oddly, there doesn't seem a way to pass a different parameter to the function.

The first button of the form, `Simple`, creates some simple XML text using the low-level, official DOM interfaces. The program calls the `createElement` method of the document for each node, adding them as children of other nodes:

```

procedure TForm1.btnSimpleClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot, iNode, iNode2, iChild, iAttribute: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('xml'));
  // node "test"
  iNode := iRoot.appendChild (iXml.createElement ('test'));

```

```

iNode.appendChild (iXml.createElement ('test2'));
iChild := iNode.appendChild (iXml.createElement ('test3'));
iChild.appendChild (iXml.createTextNode ('simple value'));
iNode.insertBefore (iXml.createElement ('test4'), iChild);

// node replication
iNode2 := iNode.cloneNode (True);
iRoot.appendChild (iNode2);

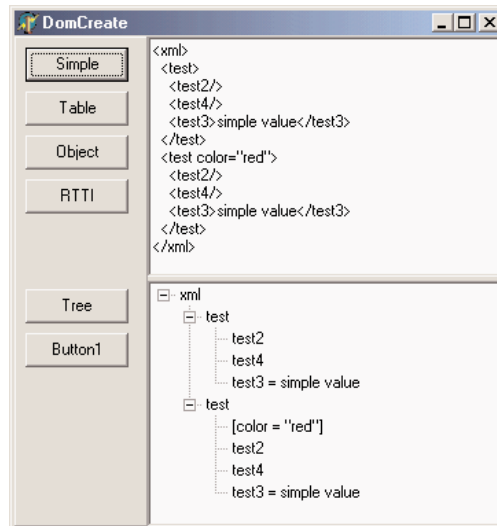
// add an attribute
iAttribute := iXml.createAttribute ('color');
iAttribute.nodeValue := 'red';
iNode2.attributes.setNamedItem (iAttribute);

// show XML in memo
Memo1.Lines.Text := FormatXMLData (XMLDoc.XML.Text);
end;
```

Notice that text nodes are added explicitly, attributes are created with a specific create call, and that the code uses `cloneNode` to replicate an entire branch of the tree. Overall, the code is quite cumbersome to write, but after a while you might get used to this style. The effect of the program is visible (formatted in the memo and in the tree) in Figure 23.3.

**FIGURE 23.3:**

The DomCreate example can generate various types of XML documents using a DOM.



The second example of DOM creation relates to a dataset. I've added to the form a BDE Table component (but any other dataset would have done) and added to a button the call to my custom DataSetToDOM procedure, like this:

```
DataSetToDOM ('customers', 'customer', XMLDoc, Table1);
```

The `DataSetToDOM` procedure creates a root node with the text of the first parameter, then grabs each record of the dataset, defines a node with the second parameter, and adds a subnode for each field of the record, all using extremely generic code:

```
procedure DataSetToDOM (RootName, RecordName: string; XMLDoc: TXMLDocument;
  DataSet: TDataSet);
var
  iNode, iChild: IXMLNode;
  i: Integer;
begin
  DataSet.Open;
  DataSet.First;
  // root
  XMLDoc.DocumentElement := XMLDoc.CreateNode (RootName);

  // add table data
  while not DataSet.EOF do
  begin
    // add a node for each record
    iNode := XMLDoc.DocumentElement.AddChild (RecordName);
    for I := 0 to DataSet.FieldCount - 1 do
    begin
      // add an element for each field
      iChild := iNode.AddChild (DataSet.Fields[i].FieldName);
      iChild.Text := DataSet.Fields[i].AsString;
    end;
    DataSet.Next;
  end;
end;
```

The preceding code uses the simplified DOM access interfaces provided by Borland, which include an `AddChild` node that creates the subnode, and the direct access to the `Text` property for defining a child node with textual content. This apparently simple routine extracts an XML representation of your dataset, also opening up a lot of opportunities for Web publishing, as I'll discuss later in the section on XSL.

Another interesting opportunity is the generation of XML documents describing Delphi objects. The `DomCreate` program has a button used to describe a few selected properties of an object, again using the low-level DOM:

```
procedure AddAttr (iNode: IDOMNode; Name, Value: string);
var
  iAttr: IDOMNode;
begin
  iAttr := iNode.ownerDocument.createAttribute (name);
  iAttr.nodeValue := Value;
  iNode.attributes.setNamedItem (iAttr);
end;
```

```
procedure TForm1.btnObjectClick(Sender: TObject);  
var  
    iXml: IDOMDocument;  
    iRoot: IDOMNode;  
begin  
    // empty the document  
    XMLDoc.Active := False;  
    XMLDoc.XML.Text := '';  
    XMLDoc.Active := True;  
  
    // root  
    iXml := Xm1Doc.DOMDocument;  
    iRoot := iXml.appendChild (iXml.createElement ('Button1'));  
  
    // a few properties as attributes (might also be nodes)  
    AddAttr (iRoot, 'Name', Button1.Name);  
    AddAttr (iRoot, 'Caption', Button1.Caption);  
    AddAttr (iRoot, 'Font.Name', Button1.Font.Name);  
    AddAttr (iRoot, 'Left', IntToStr (Button1.Left));  
    AddAttr (iRoot, 'Hint', Button1.Hint);  
  
    // show XML in memo  
    Memo1.Lines := Xm1Doc.XML;  
end;
```

Of course, it is more interesting to have a generic technique capable of saving the properties of each Delphi component (or persistent object, to be more precise), recursing on persistent subobjects and indicating the names of referenced components. This is what I've done in the `ComponentToDOM` procedure, which uses the low-level RTTI information provided by the `TypeInfo` unit, including the extraction of the list of the properties of a component. Once more, the program uses the simplified Delphi XML interfaces:

```
procedure ComponentToDOM (iNode: IXmlNode; Comp: TPersistent);  
var  
    nProps, i: Integer;  
    PropList: PPropList;  
    Value: Variant;  
    newNode: IXmlNode;  
begin  
    // get list of properties  
    nProps := GetTypeData (Comp.ClassInfo)^.PropCount;  
    GetMem (PropList, nProps * SizeOf(Pointer));  
    try  
        GetPropInfos (Comp.ClassInfo, PropList);  
        for i := 0 to nProps - 1 do  
            begin  
                Value := GetPropValue (Comp, PropList [i].Name);  
                NewNode := iNode.AddChild(PropList [i].Name);  
            end  
    except  
    end
```

```

NewNode.Text := Value;
if (PropList [i].PropType^.Kind = tkClass) and (Value <> 0) then
  if TObject (Integer(Value)) is TComponent then
    NewNode.Text := TComponent (Integer(Value)).Name
  else
    // TPersistent but not TComponent: recurse
    ComponentToDOM (newNode, TObject (Integer(Value)) as TPersistent);
end;
finally
  FreeMem (PropList);
end;
end;

```

These two lines of code, in this case, trigger the creation of the XML document (visible in Figure 23.4):

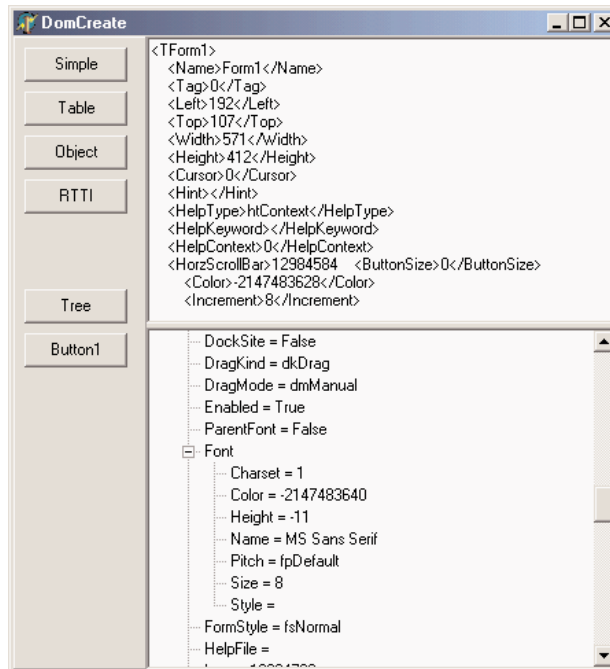
```

XMLDoc.DocumentElement := XMLDoc.CreateNode(self.ClassName);
ComponentToDOM (XMLDoc.DocumentElement, self);

```

**FIGURE 23.4:**

The XML generated to describe the form of the DomCreate program. Notice (in the tree and in the memo text) that properties of class types are further expanded.



## XML Data Binding Interfaces

We have seen that working with the DOM to access or generate a document is rather tedious, because you must use positional information and not logical access to the data. Also, handling

series of repeated nodes of different possible types (as in the XML sample of Listing 23.1, describing books) is far from simple. Moreover, using a DOM, you can create any well-formed document, but (unless you use a validating DOM) you can add any subnode to any node, coming up with almost useless documents, as no one else will be able to manage them.

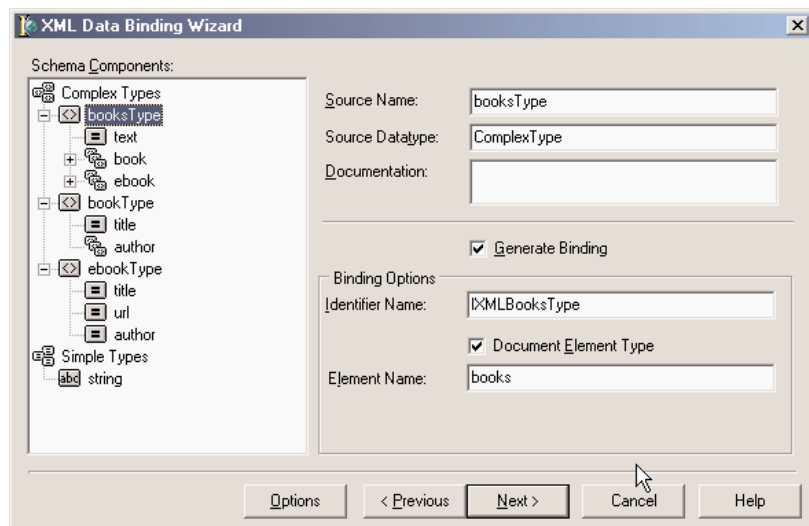
To solve these issues, Borland has added to Delphi 6 an XML Data Binding Wizard, which can examine an XML document or a document definition (a schema, a DTD, or another type of definition) and generate a set of interfaces for manipulating the document. These interfaces are specific to the document and its structure, and allow you to have more readable code, but are certainly less generic as far as the types of documents you can handle with them (and this is far more positive than it might sound at first).

You can activate the XML Data Binding Wizard by using the corresponding icon in the first page of the New Items dialog box of the IDE or by double-clicking directly on the XMLDocument component. (What is quite odd is that the corresponding command is not in the local menu of the component.)

After a first page where you can select a input file, this wizard shows you the structure of the document graphically, as you can see in Figure 23.5 for the sample XML file from Listing 23.1. In this page, you can give a name to each entity of the generated interfaces, in case you don't like the defaults suggested by the wizard. You can actually also change the rules used by the wizard to generate the names, an extended flexibility I'd like to have in other areas of the Delphi IDE. The final page gives you a preview of the generated interfaces and offers options for generating schemas and other definition files.

**FIGURE 23.5:**

Delphi's XML Data Binding Wizard can examine the structure of a document or a schema (or another document definition) to create a set of interfaces for simplified and direct access to the DOM data.



For the sample XML file with the author names, the XML Data Binding Wizard generates an interface of the root node, and four interfaces for the two lists of different elements and the actual elements (books and e-books). These are a few excerpts of the generated code, available in the `XmlIntfDefinition` unit of the `XmlInterface` example:

```

type
  IXMLBooksType = interface(IXMLNode)
    ['{C9A9FB63-47ED-4F27-8ABA-E71F30BA7F11}']
    { Property Accessors }
    function Get_Text: WideString;
    function Get_Book: IXMLBookTypeList;
    function Get_Ebook: IXMLBookTypeList;
    procedure Set_Text(Value: WideString);
    { Methods & Properties }
    property Text: WideString read Get_Text write Set_Text;
    property Book: IXMLBookTypeList read Get_Book;
    property Ebook: IXMLBookTypeList read Get_Ebook;
  end;

  IXMLBookTypeList = interface(IXMLNodeCollection)
    ['{3449E8C4-3222-47B8-B2B2-38EE504790B6}']
    { Methods & Properties }
    function Add: IXMLBookType;
    function Insert(const Index: Integer): IXMLBookType;
    function Get_Item(Index: Integer): IXMLBookType;
    property Items[Index: Integer]: IXMLBookType read Get_Item; default;
  end;

  IXMLBookType = interface(IXMLNode)
    ['{26BF5C51-9247-4D1A-8584-24AE68969935}']
    { Property Accessors }
    function Get_Title: WideString;
    function Get_Author: IXMLString_List;
    procedure Set_Title(Value: WideString);
    { Methods & Properties }
    property Title: WideString read Get_Title write Set_Title;
    property Author: IXMLString_List read Get_Author;
  end;

```

For each interface, the XML Data Binding Wizard also generates an implementation class that provides the code for the interface methods by translating the requests into DOM calls. The unit includes three initialization functions, which can return the interface of the root node from a document loaded in an `XMLDocument` component (or a component providing a generic `IXMLDocument` interface), or return one from a file, or create a brand new DOM:

```

function Getbooks(Doc: XMLDocument): IXMLBooksType;
function Loadbooks(const FileName: WideString): IXMLBooksType;
function Newbooks: IXMLBooksType;

```



After generating these interfaces using the wizard, in the `XmlInterface` example, I've repeated XML document access code similar to the `XmlDomTree` example, but much simpler to write (and to read). For example, you can get the attribute of the root node by writing:

```
procedure TForm1.btnAttrClick(Sender: TObject);  
var  
    Books: IXMLBooksType;  
begin  
    Books := Getbooks (XmlDocument1);  
    ShowMessage (Books.Text);  
end;
```

Simple, isn't it? It is even simpler if you recall that while typing this code, Delphi's code insight can help by listing the available properties of each node, thanks to the fact that the parser can read in the interface definitions (while it cannot understand the format of a generic XML document). Accessing a node of one of the sublists is a matter of writing one of the following statements (possibly the second with the default array property):

```
Books.Book.Items[1].Title // full  
Books.Book[1].Title       // further simplified
```

Similarly simplified code can be used to generate new documents or add new elements, also thanks to the customized `Add` method is available in each list-based interface. Again, if you don't have a predefined structure for the XML document, as in the dataset-based and RTTI-based examples of the previous demonstration, you won't be able to use this approach.

## Validation and Schemas in Short

The XML Data Binding Wizard can work from existing schemas or generate one for an XML document (and eventually save it in a file with the `.XDB` extension). But what is a schema for, and what does it look like? An XML document describes some data, but to exchange this data among companies it has to stick to some agreed structure. A schema is a document definition, against which a document can be checked for correctness, an operation usually indicated with the term *validation*.

The first—and still very widespread—type of validation available for XML was *document type definitions (DTDs)*. These documents describe the structure of the XML but cannot really define the possible content of each node. Also, DTDs are not XML document themselves but use a different, very awkward notation.

At the end of year 2000, the W3C approved the first official draft of XML *schemas* (already available in an incompatible version called XML-Data within Microsoft's DOM). An XML schema is itself a XML document, one that can validate both the structure of the XML tree and the content of the node. A schema is based on the use and definition of simple and complex data types, similar to what happens in an OOP programming language.

A schema defines complex types, indicating for each the possible nodes, their optional sequence (sequence, all), the number of occurrences for each subnode (minOccurs, maxOccurs), and the data type of each specific element. Here is the schema defined by the XML Data Binding Wizard for the usual sample books file:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://www.borland.com/schemas/delphi/6.0/XMLDataBinding">
  <xs:element name="books" type="booksType" />
  <xs:complexType name="booksType">
    <xs:annotation>
      <xs:appinfo xdb:docElement="books" />
    </xs:annotation>
    <xs:sequence>
      <xs:element name="book" type="bookType" maxOccurs="unbounded" />
      <xs:element name="ebook" type="ebookType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="text" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="bookType">
    <xs:annotation>
      <xs:appinfo xdb:repeated="True" />
    </xs:annotation>
    <xs:sequence>
      <xs:element name="title" type="xs:string" />
      <xs:element name="author" type="xs:string" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ebookType">
    <xs:annotation>
      <xs:appinfo xdb:repeated="True" />
    </xs:annotation>
    <xs:sequence>
      <xs:element name="title" type="xs:string" />
      <xs:element name="url" type="xs:string" />
      <xs:element name="author" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

## NOTE

As I write, there are still very few DOM implementations that can be used to validate a document against an XML schema. Apache Xerces DOM has good support for schemas. Another tool I've used for validation is XSV (XML Schema Validator), an open source attempt at a conformant schema-aware processor, which can be used either directly via the Web or after downloading a command-line executable.

## Using the SAX API

The Simple API for XML, or SAX, doesn't create a tree for the XML nodes, but simply parses the node-firing events for each node, attribute, value, and so on. Because it doesn't keep the document in memory, using the SAX allows managing much larger documents. Its approach is also very useful for one-time examination of a document, or retrieval of specific information. This is a list of events fired by the SAX:

- StartDocument and EndDocument for the entire document
- StartElement and EndElement for each node
- Characters for the text within the nodes

It is quite common to use a stack to handle the current path within the nodes tree, and push and pop elements to and from the stack for every StartElement and EndElement event.

Delphi 6 does not include specific support for the SAX interface, but this can be easily obtained by importing Microsoft's XML support (the MSXML library). In particular for the SaxDemo1 example I've used version 2 of MSXML: the Pascal version of its type library is available within the source code of the program, but you must have the COM library registered on your computer to run the program successfully.

To use the SAX, you have to install a SAX event handler within a SAX reader, then load a file and parse it. I've used the SAX reader interface provided by MSXML for VB programmers. In fact, the official (C++) interface had a few errors in its type library that prevented Delphi from importing it properly (the newer MSXML 3 might have fixed this issue by the time you read this).

In the main form of the SaxDemo1 example, I've declared:

```
sax: IVBSAXXMLReader;
```

In the FormCreate method, this is initialized with the actual COM object:

```
sax := CreateComObject (CLASS_SAXXMLReader) as IVBSAXXMLReader;  
sax.ErrorHandler := TMySaxErrorHandler.Create;
```

The code also sets an error handler, which is a class implementing a specific interface, IVBSAXErrorHandler, with three methods that are called depending on the severity of the problem: error, fatalError, and ignorableWarning.

Simplifying the code a little, the SAX parser is activated by calling the parseURL method after assigning a content handler to it:

```
sax.ContentHandler := TMySaxHandler.Create;  
sax.parseURL (filename)
```

So the code ultimately resides in the TMySaxHandler class, which has the SAX events. Because I have multiple SAX content handlers in this example, I've written a base class with

the core code and a few specialized versions for specific processing; this is the code of the base class, which implements both the `IDispatch` and `IVBSAXContentHandler` interfaces:

```

type
  TMySaxHandler = class (TInterfacedObject, IVBSAXContentHandler)
  protected
    stack: TStringList;
  public
    constructor Create;
    destructor Destroy; override;
    // IDispatch
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
      HRESULT; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer):
      HRESULT; stdcall;
    // IVBSAXContentHandler
    procedure Set_documentLocator(const Param1: IVBSAXLocator);
      virtual; safecall;
    procedure startDocument; virtual; safecall;
    procedure endDocument; virtual; safecall;
    procedure startPrefixMapping(var strPrefix: WideString;
      var strURI: WideString); virtual; safecall;
    procedure endPrefixMapping(var strPrefix: WideString); virtual; safecall;
    procedure startElement(var strNamespaceURI: WideString;
      var strLocalName: WideString; var strQName: WideString;
      const oAttributes: IVBSAXAttributes); virtual; safecall;
    procedure endElement(var strNamespaceURI: WideString;
      var strLocalName: WideString; var strQName: WideString);
      virtual; safecall;
    procedure characters(var strChars: WideString); virtual; safecall;
    procedure ignorableWhitespace(var strChars: WideString);
      virtual; safecall;
    procedure processingInstruction(var strTarget: WideString;
      var strData: WideString); virtual; safecall;
    procedure skippedEntity(var strName: WideString); virtual; safecall;
  end;

```

The most interesting portion, of course, is the final list of SAX events. All this base class does is emit information to a log when the parser starts (`startDocument`) and finishes (`endDocument`) and keep track of the current node and all of its parent nodes with a stack:

```

// TMySaxHandler.startElement
stack.Add (strLocalName);
// TMySaxHandler.endElement

```

```
stack.Delete (stack.Count - 1);
```

An actual implementation is provided by the `TMySimpleSaxHandler` class, which overrides the `startElement` event triggered for any new node to output the current position in the tree with the statement:

```
Log.Add (strLocalName + '(' + stack.CommaText + ')');
```

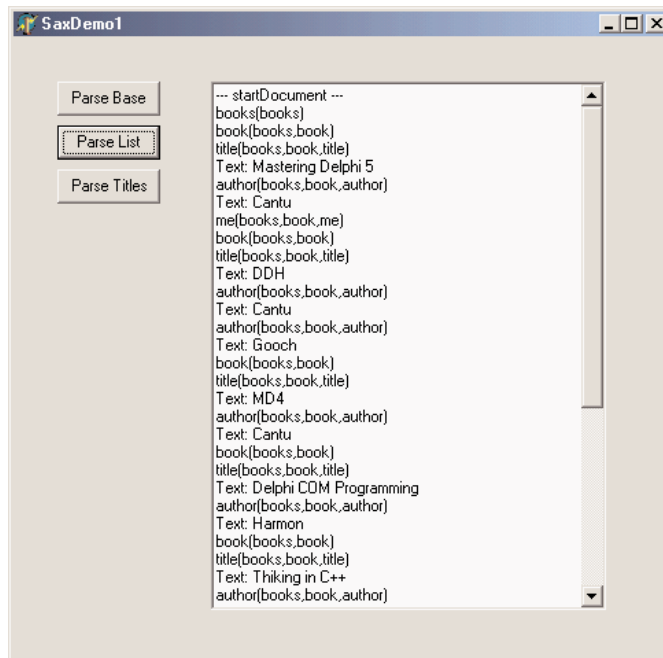
The second method of the class is the `characters` event, triggered when a node value (or a test node) is encountered, to output its content:

```
procedure TMySimpleSaxHandler.characters(var strChars: WideString);  
var  
    str: WideString;  
begin  
    inherited;  
    str := RemoveWhites (strChars);  
    if (str <> '') then  
        Log.Add ('Text: ' + str);  
end;
```

The two methods produce the combined effect of Figure 23.6.

**FIGURE 23.6:**

The log produced by reading an XML document with the SAX in the `SaxDemo1` example



This is still a generic parsing operation affecting the entire XML file. The second derived SAX content handler class, instead, refers to the specific structure of the XML document, extracting only nodes of a give type. In particular, the program looks for nodes of the *title* type. When a node has this type (in `startElement`), the class sets the `isbook` Boolean variable. The text value of the node is considered only right after a node of this type is encountered:

```
procedure TMyBooksListSaxHandler.startElement(var strNamespaceURI,  
    strLocalName, strQName: WideString; const oAttributes: IVBSAXAttributes);  
begin  
    inherited;  
    isbook := (strLocalName = 'title');  
end;  
  
procedure TMyBooksListSaxHandler.characters(var strChars: WideString);  
var  
    str: string;  
begin  
    inherited;  
    if isbook then  
        begin  
            str := RemoveWhites (strChars);  
            if (str <> '') then  
                Log.Add (stack.CommaText + ': ' + str);  
        end;  
end;  
end;
```

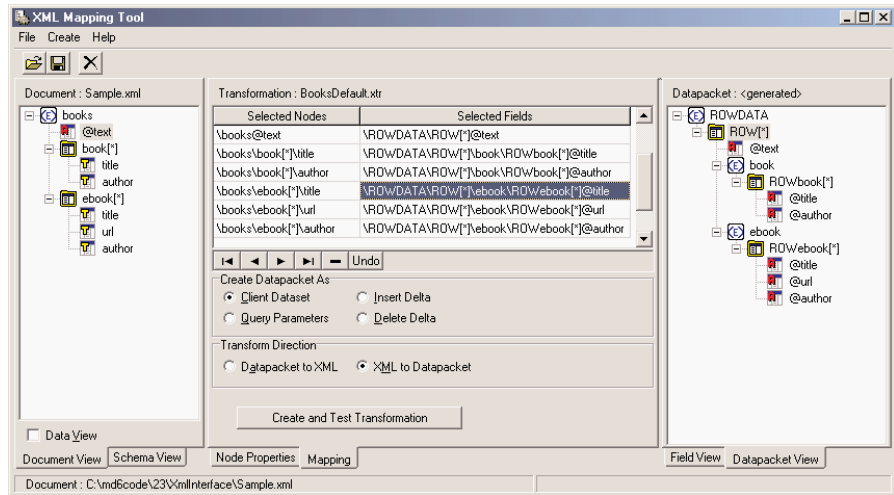
## Mapping XML with Transformations

There is one more technique you can use in Delphi 6 to handle at least some XML documents. You can create a *transformation* to translate the XML of a generic document into the format used natively by the `ClientDataSet` component when saving data to a `MyBase XML` file. In the reverse direction, another transformation can turn a dataset available within a `ClientDataSet` (through a `DataSetProvider` component) into a XML file of a required format (or schema).

Delphi 6 includes a wizard to generate such transformations. Called `XML Mapping Tool`, or `XML Mapper` for short, it is invocable from the `Tools` menu of the IDE or executed as a stand-alone application. The `XML Mapper`, visible in Figure 23.7, is a design-time helper that assists you in defining transformation rules between the nodes of a generic XML document and fields of the data packet of the `ClientDataSet`.

**FIGURE 23.7:**

The XML Mapper shows the two sides of a transformation to define a mapping among them (with the rules indicated in the central portion).



The XML Mapper windows has three areas:

- On the left is the XML document section, which displays information about the structure of the XML document (and eventually its data, if the related check box is active) in the Document View or an XML schema in the Schema View, depending on the selected tab.
- On the right is the data packet section, which displays information about the metadata in the data packet, either in the Field View indicating the dataset structure or in the Datapacket View reporting the XML structure. Notice, in fact, that the XML Mapper can also open files in the native ClientDataSet format.
- The central portion of the windows is used by the mapping section. This contains two pages as well: one for Mapping, where you can see the correspondences between selected elements of the two sides that will be part of the mapping, and one for Node Properties, where you can modify the data types and other details of each of the possible mappings.

The Mapping page of the central pane also hosts the local menu used to generate the transformation, while each other pane and view has specific local menus you can use to perform the various actions (beside the few commands in the main menu).

You can use XML Mapper to map an existing schema (or extract it from a document) to a brand new data packet, an existing data packet to a new schema or document, or an existing data packet into an existing XML document (if a match is reasonable). Besides converting the data of an XML file into a data packet, you can also convert to a delta packet of the ClientDataSet. This is useful for merging a document to an existing table, as if a user had

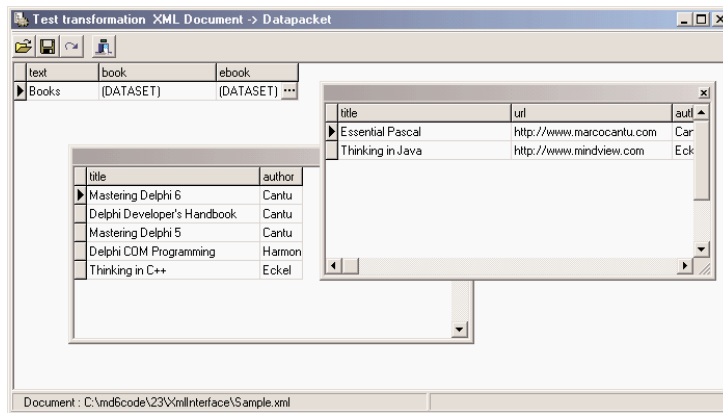
inserted them. In particular, you can transform an XML document into a delta packet for records to be modified, deleted, or inserted.

The result of using the XML Mapper is one or more transformation files, each representing a one-way conversion (so you need at least two transformation files to convert data back and forth). These transformation files are then used at design time and at run time by the XMLTransform, XMLTransformProvider, and XMLTransformClient components.

As an example, I've tried opening the usual "books" XML, which has a structure that doesn't easily match a table, since there are two lists of values of different types (I've skipped easier examples in which the XML has a plain rectangular structure). After opening the Sample.XML file in the XML Document section, I've used its local menu to select all of its elements (Select All) and to create the data packet (Create Datapacket From XML). This automatically fills the right pane with the data packet and the central portion with the proposed transformation. You can also immediately view its effect in a sample program by selecting the Create And Test Transformation button. This opens a generic application that can load a document into the dataset using the transformation you've just created, as you can see in Figure 23.8.

**FIGURE 23.8:**

The Create and Test Transformation button of the XML Mapper tool allows you to immediately verify the effect of the transformations you are building



In this specific case, you can see that the XML Mapper generates a table with two dataset fields, one of each possible list of subelements. This was the only possible standard solution, as the two sublists have different structures, and is the only solution that allows you to edit the data in a DBGrid attached to the ClientDataSet and save it back to a complete XML file, as demonstrated by the XmlMapping example. This program is basically a Windows-based editor of a complex XML document.

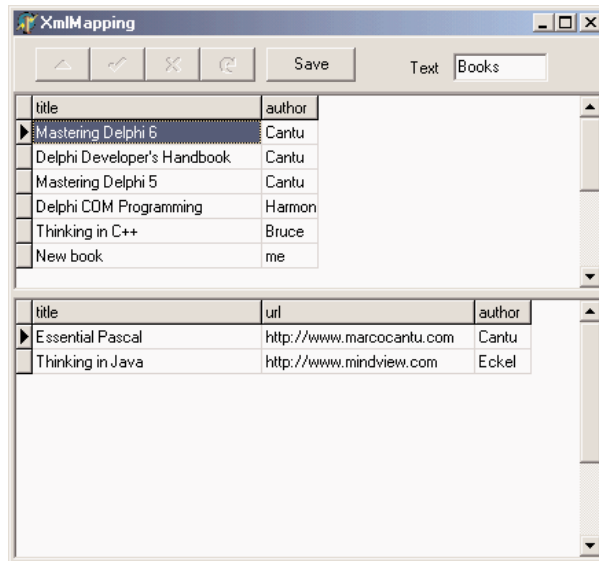
It uses a TransformProvider component, with two transformation files attached, to read in an XML document and make it available to a ClientDataSet. As the name suggests, in fact, this component is a dataset provider. To build the user interface, I haven't connected the



ClientDataSet directly to a grid, as it has a single record with a text field plus two detailed datasets. For this reason, I've added to the program two more ClientDataSet components, attached to the dataset fields and connected to two DBGrid controls. This is probably easier to understand by looking at its DFM source code in the following excerpt and at its output in Figure 23.9.

**FIGURE 23.9:**

The XmlMapping example uses a TransformProvider component to make a complex XML document available for editing within multiple ClientDataSet components.



```

object Form1: TForm1
  Caption = 'XmlMapping'
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'BooksDefault.xtr'
  TransformWrite.TransformationFile = 'BooksDefaultToXml.xtr'
  XMLDataFile = 'Sample.xml'
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'XMLTransformProvider1'
object ClientDataSet1text: TStringField
  FieldName = 'text'
  Size = 5
end
object ClientDataSet1book: TDataSetField
  FieldName = 'book'
end
object ClientDataSet1ebook: TDataSetField
  FieldName = 'ebook'
end
end

```

```
object DataSource1: TDataSource
    DataSet = ClientDataSet1
end
object Panel1: TPanel
    Align = alTop
    object Label2: TLabel
        Caption = 'Text'
        FocusControl = DBEdit2
    end
    object DBNavigator1: TDBNavigator
        VisibleButtons = [nbEdit, nbPost, nbCancel, nbRefresh]
    end
    object DBEdit2: TDBEdit
        DataField = 'text'
        DataSource = DataSource1
    end
    object Button1: TButton
        Caption = 'Save'
        OnClick = Button1Click
    end
end
object ClientDataSet2: TClientDataSet
    DataSetField = ClientDataSet1book
end
object DataSource2: TDataSource
    DataSet = ClientDataSet2
end
object DBGrid1: TDBGrid
    Align = alTop
    DataSource = DataSource2
end
object Splitter1: TSplitter
    Cursor = crVSplit
    Align = alTop
end
object ClientDataSet3: TClientDataSet
    DataSetField = ClientDataSet1ebook
end
object DataSource3: TDataSource
    DataSet = ClientDataSet3
    Left = 232
    Top = 224
end
object DBGrid2: TDBGrid
    Align = alClient
    DataSource = DataSource3
end
end
```

This program allows you to edit the data of the various sublists of nodes, within the grids, modifying them but also adding or deleting records. As you apply the changes to the dataset (clicking the Save button, which calls `ApplyUpdates`), the transform provider saves an updated version of the file to disk.

As an alternative approach, you can also create transformations that map only portions of the XML document into a dataset. As an example, see the `BooksOnly.xtr` file in the folder of the `XmlMapping` example. This can be useful for viewing the data, but the modified XML document you'll generate will have a different structure and content from the original, including only the portion you've selected. So this can be useful for viewing the data, but not for editing it.

**NOTE**

It is not surprising that the transformation files are themselves XML documents, as you can see by opening one in the editor. This XML document uses a custom format.

At the opposite side, we can see how a transformation can be used to take a database table or the result of a query and produce an XML file with a more readable format than the one provided by default by the `ClientDataSet` persistence mechanism. To build the `MapTable` example, I've placed a table component on a form and attached a `DataSetProvider` to it and a `ClientDataSet` to the provider. After opening the table and the client dataset, I saved its content to an XML file.

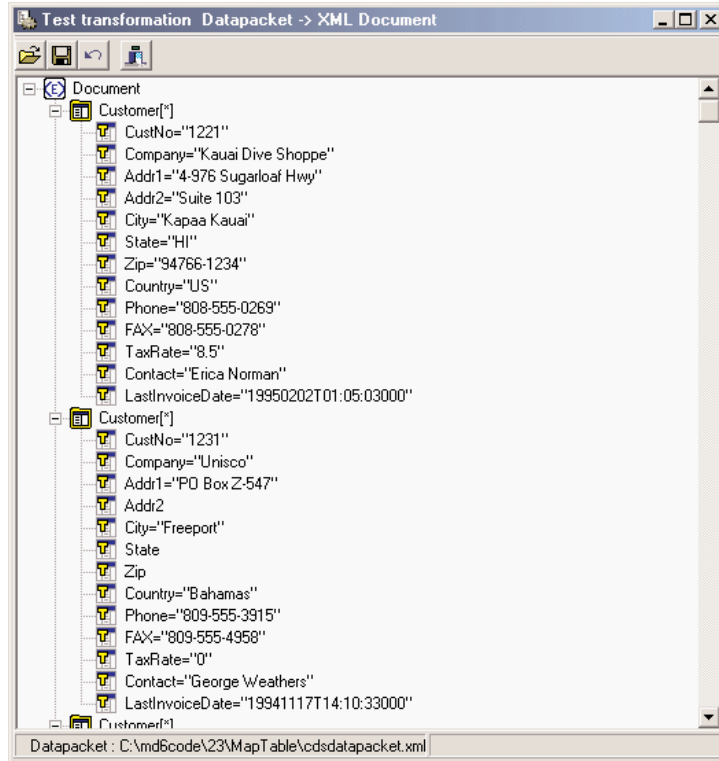
At that point, I opened the XML Mapper, loaded the data packet file into it, selected all of the data packet nodes (with the `Select All` command of its local menu) and invoked the `Create XML From Datapacket` command. In the following dialog box, I accepted the default name mappings for fields and only changed the suggested name for record nodes (*ROW*) into something more readable (*Customer*). If you now test the transformation, the XML Mapper will display the contents of the resulting XML document in a custom tree view, as you can see in Figure 23.10.

After saving the transformation file, I was ready to resume developing the program, adding to it another provider that takes the data from the `ClientDataSet` (as a user might edit in on an attached `DBGrid` before transforming it) and makes it available to an `XMLTransformClient` component. This component has the transformation file connected to it, but not an XML file. In fact, clicking the button shows the XML document within a memo (after a formatting it) instead of saving it to a file, something you can do by calling the `GetDataAsXml` method (even if the `Help` file is far from clear about this):

```
procedure TForm1.btnMapClick(Sender: TObject);  
begin  
    Memo1.Lines.Text := FormatXmlData(XMLTransformClient1.GetDataAsXml(''));  
end;
```

**FIGURE 23.10:**

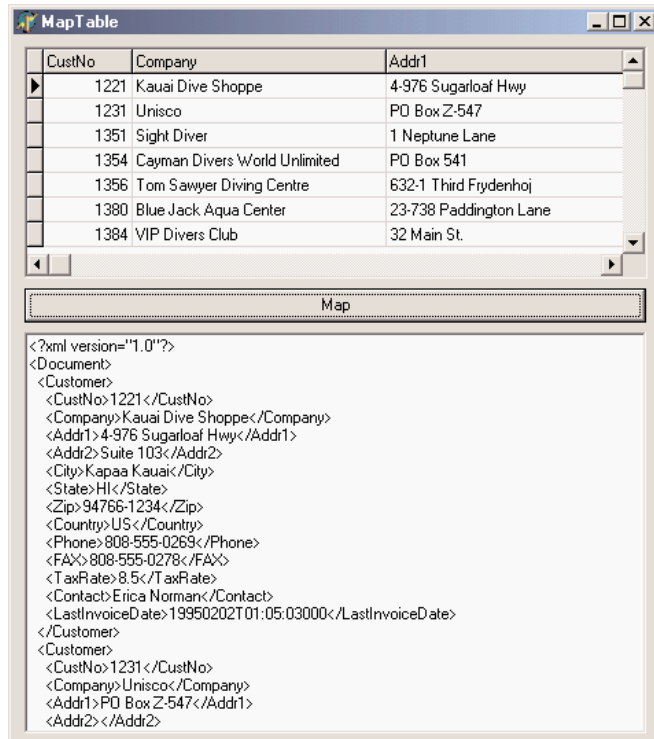
When you convert from a data packet to an XML document, the XML Mapper can preview the document in a tree structure.



This is the only code of the program visible at run time in Figure 23.11. The application has much simpler code than the DomCreate example I used to generate a similar XML document, but requires the design-time definition of the transformation. The DomCreate example, instead, could work on any dataset at run time, without any connection to a specific table, as it had some rather generic code. In theory, it is possible to produce similar dynamic mappings by using the events of the generic XMLTransform component, but I find it much easier to use the DOM-based approach discussed earlier. Notice also that the FormatXmlData call produces much nicer output but slows down the program, because it involves loading the XML into a DOM.

**FIGURE 23.11:**

The MapTable example can generate an XML document from a database table using a custom transformation file. You can see the original dataset in the DBGrid above and the resulting XML document in the memo control below



## XML and Internet Express

Once you have defined the structure of an XML document, you might want to let users see and edit the data in a Windows application or over the Web. This second case is rather interesting, as Delphi provides specific support for it. Delphi 5 already included an architecture called Internet Express, which is now available in Delphi 6 as part of the WebSnap platform. WebSnap offers also support for XSL, which I'll discuss later.

In Chapter 17, "Multitier Database Applications with DataSnap," I discussed the development of DataSnap applications (formerly known as Midas applications). Internet Express provides a client component for this architecture, called XMLBroker, which can be used in place of a client dataset to retrieve data from a middle-tier DataSnap program and make it available to a specific type of page producer, called InetXpageProducer. You can use these components in a standard WebBroker application or in a WebSnap program. The idea behind Internet Express is that you write a Web server extension (CGI or ISAPI or Apache modules, as discussed in the

preceding chapter), which in turn produces Web pages hooked to your DataSnap server. Your custom application acts as a DataSnap client and produces pages for a browser client. Internet Express offers the services required to build this custom application easily.

I know this sounds confusing, but Internet Express is a four-tier architecture: SQL server, application server (the DataSnap server), Web server with a custom application, and finally Web browser. Of course, you can place the database access components within the same application handling the HTTP request and generating the resulting HTML, as in a client/server solution. You can even access a local database or an XML file, in a single-tier structure.

In other words, Internet Express is a technology for building clients based on a browser, which lets you send the entire dataset to the client computer along with the HTML and some JavaScript for manipulating the XML and showing it into the user interface defined by the HTML. It is the JavaScript that enables the browser to show the data and manipulate it.

## The XMLBroker Component

Internet Express uses multiple technologies to accomplish this. The DataSnap data packets are converted into the XML format, to let the program embed this data right into the HTML page for Web client-side manipulation. Actually, the Delta data packet is also represented in XML. These operations are performed by the XMLBroker component, which can handle XML and provide data to the new JavaScript components. Like the ClientDataSet, the XMLBroker has

- A `MaxRecords` property indicating the number of records to add to a single page
- A `Params` property hosting the parameters that components will forward to the remote query through the provider
- A `WebDispatch` property indicating the update request the broker responds to

The `InetXPageProducer` allows you to generate HTML forms from datasets, in a visual way similar to the development of an `AdapterPageProducer` user interface. Actually, the Internet Express architecture, the interfaces it uses internally, and some of its IDE editor can together be considered the parent of the `WebSnap` architecture. With the notable difference of generating scripts to be executed on the server side and on the client side, they both provide an editor for placing visual components and generating such scripts. A notable difference I'm personally not terribly happy about, though, is that the older Internet Express is more XML-oriented than newer `WebSnap`.

---

**TIP**

Another common feature of the `InetXPageProducer` and the `AdapterPageProducer` is the support for Cascading Style Sheets (CSS). These components have the two alternative `Style` and `StylesFile` properties for defining the CSS, and each visual element has a `StyleRule` property where you can select the style name.

## JavaScript Support

To make the editing operations on the client side powerful, the InetXPPageProducer uses special JavaScript components and code. Delphi embeds a rather large JavaScript library, which the browser will have to download. This might seem a nuisance, but it is the only way the browser interface (which is based on dynamic HTML) can be rich enough to support field constraints and other business rules with the browser. This is really impossible with plain HTML. The JavaScript files provided by Borland, and that you should make available on the Web site hosting the application, are the following:

File	Description
Xm1dom.js	DOM-compatible XML parser (for browsers lacking native XML DOM support)
Xm1db.js	JavaScript classes for the HTML controls
Xm1disp.js	JavaScript classes for binding XML data with the HTML controls
Xm1errdisp.js	Classes for reconciling errors
Xm1Show.js	JavaScript functions to display data and delta packets (for debugging purposes)

HTML pages generated by Internet Express usually include references to these JavaScript files, such as:

```
<script language=Javascript type="text/javascript"
  src="IncludePathURL/xm1db.js"></script>
```

You can customize the JavaScript by adding code directly into the HTML pages or by creating a new Delphi components, written to fit with the Internet Express architecture that “emits” JavaScript code (possibly along with HTML). As an example, the sample `TPrompt-QueryButton` class of `INetXCustom` generates the following HTML and JavaScript code:

```
<script language=javascript type="text/javascript">
  function PromptSetField(input, msg) {
    var v = prompt(msg);
    if (v == null || v == "")
      return false;
    input.value = v
    return true;
  }
  var QueryForm3 = document.forms['QueryForm3'];
</script>
<input type=button value="Prompt..."
  onclick="if (PromptSetField(PromptResult, 'Enter some text\n'))
    QueryForm3.submit();">
```

**Tip**

If you plan on using Internet Express, have a look at the `INetXCustom` extra demo components, available in the `\Demos\Midas\InternetExpress\INetXCustom` folder. Follow the detailed instructions in the `readme.txt` file for the installation of these components, which are provided by Borland with no support but allow you to add many more features to your Internet Express applications with little extra effort.

Of course, to deploy this architecture you don't need anything special on the client side, as any browser up to the HTML 4 standard can be used, on any operating system. The Web server, instead, must be a Win32 server (we're waiting for this technology to be available in Kylix) and you must deploy the DataSnap libraries on it (after paying the proper license fee to Borland, still not disclosed at this time).

## Building a First Example

To better understand what I'm talking about, and as a way to cover some more technical details, let me try out a simple demo, called `IeFirst`. To avoid configuration issues, this is a CGI application accessing a dataset directly—in this case, a local table retrieved via the BDE. Later I'll show you how to turn an existing DataSnap Windows client to a browser-based interface. To build `IeFirst`, I've created a new CGI application and added to its data module a `Table` and a `DataSetProvider`. The next step is to add an `XMLBroker` component and connect it to the provider:

```
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'employee.db'
end
object DataSetProvider1: TDataSetProvider
  DataSet = Table1
end
object XMLBroker1: TXMLBroker
  ProviderName = 'DataSetProvider1'
  WebDispatch.MethodType = mtAny
  WebDispatch.PathInfo = 'XMLBroker1'
  ReconcileProducer = PageProducer1
  OnGetResponse = XMLBroker1GetResponse
end
```

The `ReconcileProducer` property is required to show a proper error message in case of an update conflict. As we'll see later, one of the Delphi demos includes some custom code, but in this simple example I've simply connected a traditional `PageProducer` component with a



generic HTML error message. After setting up the XML broker, you can add an InetXPageProducer to the Web data module. This component has a standard HTML skeleton; I've customized to add a title to it, without touching the special tags:

```
<HTML><HEAD>
  <title>IeFirst</title>
</HEAD><BODY>
  <h1>Internet Express First Demo (IeFirst.exe)</h1>
  <#INCLUDES><#STYLES><#WARNINGS><#FORMS><#SCRIPT>
</BODY>
```

The special tags are automatically expanded using the JavaScript files of the directory specified by the IncludePathURL property. You *must* set this property to refer to the Web server directory where these files reside. You can find them in the \Delphi6\Source\WebMidas directory. The five tags have the following effect:

Tag	Effect
<#INCLUDES>	Generates the instructions to include the JavaScript libraries
<#STYLES>	Adds the embedded style sheet definition
<#WARNINGS>	Used at design time to show errors in the InetXPageProducer editor
<#FORMS>	Generates the HTML code produced by the components of the Web page
<#SCRIPT>	Adds a JavaScript block used to start up the client-side script

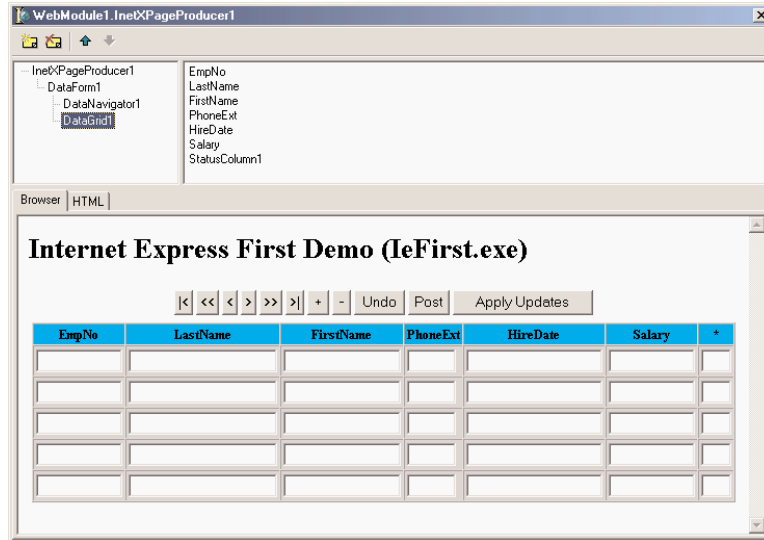
## NOTE

The InetXPageProducer component handles also a few more internal tags. <#BODYELEMENTS> corresponds to all of the five tags of the predefined template. <#COMPONENT Name=WebComponent-Name> is part of the generated HTML code used to declare the components generated visually. <#DATAPACKET XMLBroker=BrokerName> is replaced with the actual XML of the data packet.

To customize the resulting HTML of the InetXPageProducer, you can use its editor, which again is similar to the one for WebSnap server-side scripting. Just double-click the InetXPageProducer component, and Delphi opens up a window like the one in Figure 23.12 (with the final settings of the example). In this editor, you can create complex structures, starting with a query form, data form, or generic layout group. In the data form of my simple example, I've added a DataGrid and a DataNavigator component, without customizing them any further (an operation you do by adding child buttons, columns, and other objects, which fully replace the default ones).

**FIGURE 23.12:**

The InetXPageProducer editor allows you to build complex HTML forms visually, similarly to the AdapterPageProducer.



The DFM code for the InetXPageProducer and its internal components in my example is the following, where you can see the core settings plus some limited graphical customizations:

```

object InetXPageProducer1: TInetXPageProducer
  IncludePathURL = '/jssource/'
  HTMLDoc.Strings = (...)
object DataForm1: TDataForm
  object DataNavigator1: TDataNavigator
    XMLComponent = DataGrid1
    Custom = 'align="center"'
  end
object DataGrid1: TDataGrid
  XMLBroker = XMLBroker1
  DisplayRows = 5
  TableAttributes.BgColor = 'Silver'
  TableAttributes.CellSpacing = 0
  TableAttributes.CellPadding = 2
  HeadingAttributes.BgColor = 'Aqua'
  object EmpNo: TTextColumn...
  object LastName: TTextColumn...
  object FirstName: TTextColumn...
  object PhoneExt: TTextColumn...
  object HireDate: TTextColumn...
  object Salary: TTextColumn...
  object StatusColumn1: TStatusColumn...
end
end
end

```

But the value of these components is in the HTML (and JavaScript) code they generate, which you can preview by selecting the HTML tab of the InetXPPageProducer editor. Here are a few portions of the definitions in the HTML, for the buttons, the data grid heading, and one if its cells:

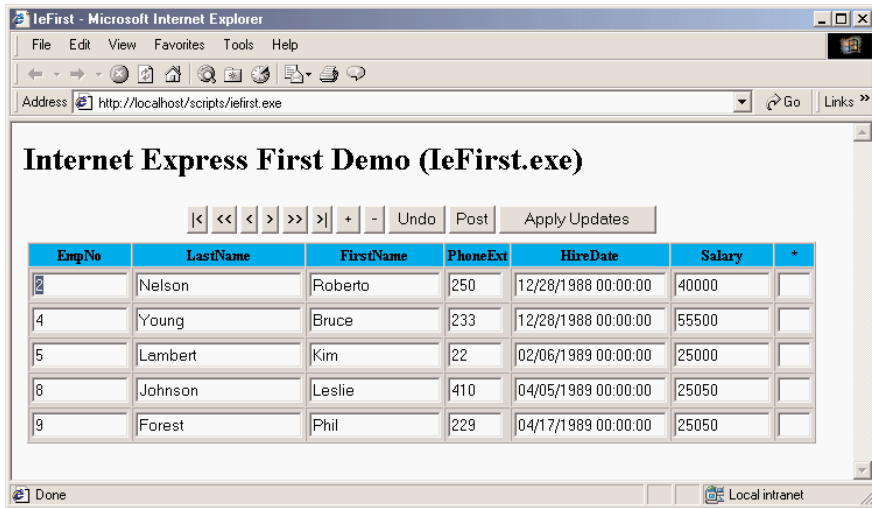
```
// buttons
<table align="center">
  <tr><td colspan="2">
    <input type="button" value="|"<
      onclick='if (xml_ready) DataGrid1_Dispatch.first();'>
    <input type="button" value="<<"
      onclick='if (xml_ready) DataGrid1_Dispatch.pgup();'>
  ...
// data grid heading
<table cellpadding="2" cellspacing="0" border="1" bgcolor="silver">
  <tr bgcolor="aqua">
    <th>EmpNo</th>
    <th>LastName</th>
  ...
</tr>
<tr>
  // a data cell
  <td><div>
    <input type="text" name="EmpNo" size="10"
      onfocus='if(xml_ready)DataGrid1_Dispatch.xfocus(this);'
      onkeydown='if(xml_ready)DataGrid1_Dispatch.keys(this);'>
  </div></td>...
```

When the HTML generator is set up, you can go back to the Web data module, add an action to it, and connect the action with the InetXPPageProducer via the Producer property. This should be enough to make the program work through a browser, as you can see in Figure 23.13.

If you look at the HTML file received by the browser, you'll find the table mentioned in the preceding definition, some JavaScript code here and there, and the database data in the data packet XML format. This data is assembled by the XML broker and passed to the producer component to be embedded in the HTML file. Notice that the number of records sent to the client depends on the XMLBroker, not on the number of lines in the grid. Once the XML data is sent to the browser, in fact, you can use the buttons of the navigator component to move around in the data without requiring further access to the server to fetch more. This is quite different from the paging behavior of WebSnap. Not that one is better than the other; this depends on the specific application you are building.

**FIGURE 23.13:**

The `IeFirst` application sends to the browser some HTML components, an entire XML document, and the JavaScript code to show the data in the visual components.



At the same time, the JavaScript classes in the system allow the user to type in new data, following the rules imposed by the JavaScript code hooked to dynamic HTML events. Notice that the grid, by default, has an extra asterisk column, indicating which records have been modified. The update data is collected in an XML data packet in the browser, and sent back to the server when the user clicks the `Apply Updates` button. At this point, the browser activates the action specified by the `WebDispath.PathInfo` property of the `XMLBroker`. There is no need to export this action from the `Web` data module, as this operation is automatic (although you can disable it by setting `WebDispath.Enable` to `False`).

The `XMLBroker` applies the changes to the server, returning the content of the provider connected to the `ReconcileProvider` property (or raising an exception if this is not defined). When everything works fine, the `XMLBroker` redirects the control to the main page that contains the data. However, I've experienced some problems with this technique, so the `IeFirst` example handles the `OnGetReponse`, indicating this is an update view:

```
procedure TWebModule1.XMLBroker1GetResponse(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
    Response.Content := '<h1>Updated</h1><p>' + InetXPageProducer1.Content;
    Handled := True;
end;
```

## Master/Detail in Internet Express

My second Internet Express example goes a little beyond the basics by providing a master/detail data packet for Web browsing obtained through a `DataSnap` connection. The program

uses the AppPlus server of Chapter 17, which defines the master/detail relationship between two tables. The dataset field embedded in the table will be transformed into a nested XML structure, delivering the same information.

The program uses a combination of XMLBroker, InetXPPageProducer, and DCOMConnection. This time, I've customized the Web components, by adding a LayoutGroup component to obtain multiple columns, and I've also created fields to select the information to display. In Listing 23.2 are some snippets of the DFM file of this Web module: I've removed a lot of extra information, but I think it is worth looking at it.

### Listing 23.2: Portions of the DFM file for the leMd example's Web module

```
object DCOMConnection1: TDCOMConnection
    Connected = False
    ServerName = 'AppSP1us.AppServerP1us'
end
object XMLBroker1: TXMLBroker
    ProviderName = 'ProviderCustomer'
    RemoteServer = DCOMConnection1
    WebDispatch.PathInfo = 'XMLBroker1'
end
object InetXPPageProducer1: TInetXPPageProducer
    IncludePathURL = '/jssource/'
object DataForm1: TDataForm
    object LayoutGroup1: TLayoutGroup
        DisplayColumns = 2
        object DataNavigator1: TDataNavigator
            XMLComponent = FieldGroup1
            object FirstButton1: TFirstButton
                XMLComponent = FieldGroup1
                Caption = '<'
            end
            object PriorButton1: TPriorButton
                XMLComponent = FieldGroup1
                Caption = '<'
            end
            object NextButton1: TNextButton
                XMLComponent = FieldGroup1
                Caption = '>'
            end
            ...
            object ApplyUpdatesButton1: TApplyUpdatesButton
                Caption = 'Apply Updates'
                XMLBroker = XMLBroker1
                XMLUseParent = True
            end
        end
    end
object FieldGroup1: TFieldGroup
    XMLBroker = XMLBroker1
```

```

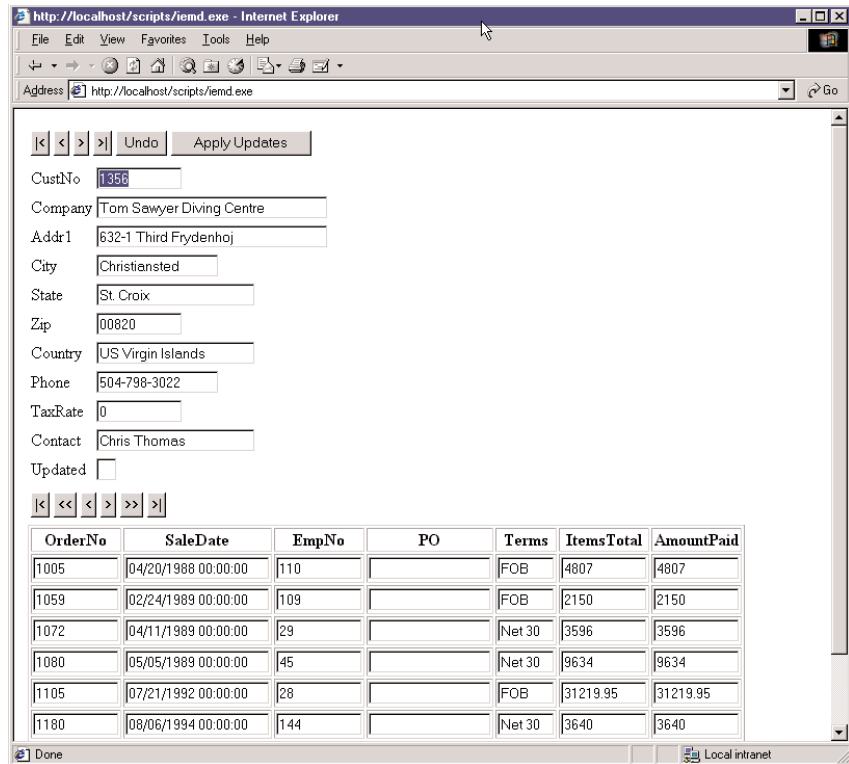
    object CustNo: TFieldText
      DisplayWidth = 10
      Caption = 'CustNo'
      FieldName = 'CustNo'
    end
    object Company: TFieldText
      DisplayWidth = 30
      Caption = 'Company'
      FieldName = 'Company'
    end
    ...
  end
  object DataNavigator2: TDataNavigator
    XMLComponent = DataGrid1
    object FirstButton2: TFirstButton
      XMLComponent = DataGrid1
      Caption = '<'
    end
    object PriorPageButton1: TPriorPageButton
      XMLComponent = DataGrid1
      Caption = '<<'
    end
    ...
  end
  object DataGrid1: TDataGrid
    XMLBroker = XMLBroker1
    XMLDataSetField = 'TableOrders'
    DisplayRows = 8
    object OrderNo: TTextColumn
      DisplayWidth = 10
      Caption = 'OrderNo'
      FieldName = 'OrderNo'
    end
    object SaleDate: TTextColumn
      DisplayWidth = 18
      Caption = 'SaleDate'
      FieldName = 'SaleDate'
    end
    ...
  end
end
end
end
end

```

Once the structure is set up, you can deploy the CGI executable on the Web server and see the effect illustrated in Figure 23.14 directly in a browser. Notice that the HTML you receive is rather large, as it includes the entire master/detail structure. Once you've received it, however, you can browse the master table and the detail grid without having to ask the server for any more data.

**FIGURE 23.14:**

A master/detail relationship displayed in a browser by the leMd example



Obviously, much more could be said about the capabilities of the Internet Express to build a Web front end for a DataSnap server, as a possible alternative to the server-side scripting offered by WebSnap. Server-side scripting certainly allows for wider support of browsers, although most browsers have the minimal JavaScript required by Internet Express. Also, you should consider cases in which you prefer to have immediate feedback on the server for every action of the user or when you prefer to let the user prepare a large delta packet, even working in a disconnected situation, and then receive the entire batch of updates at once.

## Using XSLT

Another approach for generating HTML starting from an XML document is the use of the Extensible Stylesheet Language (XSL) or, to be more precise, its XSL Transformations (XSLT) subset. XSLT uses other XML technologies, notably XPath and XPointer to identify portions of documents.

XPath defines a set of rules to locate one or more nodes within a document. The rules are based on a path-like structure of the node within the XML tree, so that `/books/book` identifies any *book* node under the *books* document root. XPath uses a few special symbols to identify nodes:

- An asterisk (\*) stands for any node; for example, `book/*` indicates any subnode under a *book* node.
- A dot (.) stands for the current node.
- The pipe symbol (|) indicates alternatives, as in `book|ebook`.
- A double slash (//) stands for any path, so that `//title` indicates all of the title nodes, whatever their parent nodes, and `books//author` indicates any author node under a *books* node regardless of the nodes in between.
- The at sign (@) indicates an attribute instead of a node, as in the hypothetical `author/@lastname`. A similar notation can be used to choose only nodes having a given value for an attribute—for example, all authors with a given first name: `author[@name="marco"]`.

There are many other cases, but this short introduction to the rules of XPath should at least get you started and help you understand the following examples. An XSTL document is an XML document that works on the structure of a source XML document and generates in output another XML document, possibly an XHTML document you can view within a Web browser.

**NOTE**

Commonly used XSLT processors include MS-XML, Xalan from the Apache XML project ([xml.apache.org](http://xml.apache.org)), and the Java-based Xt of James Clarke.

The structure of an XSL file is quite simple, although its content can become extremely complex to understand. At the root should be a node like:

```
<xsl:stylesheet version="1.0" xmlns:xsl="...">
```

This node is followed by some of the base commands, such as the definition of a template to operate on a given type of nodes (`xsl:template match`), the activation of a template for a given node (`xsl:apply-templates select`), or the extraction of a value from an XML document (`xsl:value-of select`). There are also specific instructions you can use, including `xsl:for-each`, `xsl:if`, `xsl:choose`, `xsl:sort` (not available in MSXML the last time I checked), and `xsl:number`.



## XSTL in Practice

After this short and probably unclear explanation, let me discuss a couple of examples. As a starting point, you should study XSL by itself, and then focus on its activation from within a Delphi application.

For your initial tests, you can connect an XSL file directly into an XML file. As you load the XML file in Internet Explorer, this will show you the resulting transformation, usually the HTML. The connection is indicated in the heading of the XML document with a command like:

```
<?xml-stylesheet type="text/xsl" href="sample1embedded.xsl"?>
```

This is what I've done in the `sample1embedded.xml` file available in the `XslEmbed` project. The related XSL embeds various XSL snippets that I cannot discuss in detail. For example, it grabs the entire list of authors or filters a specific group of them with this code:

```
<h2>All Authors</h2>
<ul>
  <xsl:for-each select="books//author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
<h3>E-Authors</h3>
<ul>
  <xsl:for-each select="books/ebook/author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
```

Some rather more complex code is used to extract nodes only when a specific value is present in a subnode or attribute, regardless of the higher-level nodes. This final XSL snippet also has an `if` statement and produces an attribute in the resulting node, as a way to build an href hyperlink in the HTML:

```
<h3>Marco's works (books + ebooks)</h3>
<ul>
  <xsl:for-each select="books/*[author = 'Cantu']">
    <li> <xsl:value-of select="title"/>
      <xsl:if test="url">
        (<a><xsl:attribute name="href"><xsl:value-of select="url"/>
          </xsl:attribute>Jump to document</a>)
      </xsl:if>
    </li>
  </xsl:for-each>
</ul>
```

Again, I suggest you get some more documentation on the topic, but these examples should at least get you started.

## XSLT with WebSnap

Within the code of a program, you can execute the `TransformNode` method of a DOM node, passing to it another DOM hosting the XSL document. Instead of using this low-level approach, however, we can let WebSnap help us to create an XSL-based example. In fact, you can create a new WebSnap application (I've built a CGI program called `XslCust` in this case) and choose an `XSLPageProducer` component for its main page, to let Delphi help you start with the application code. Actually, Delphi also includes a skeleton XSL file for manipulating a `ClientDataSet` data packet and adds to the editor many new views. The XSL Text replaces the HTML file; the XML Tree shows the data, if any; the XSL tree shows the XSL within the Internet Explorer ActiveX; the HTML result shows the code produced by the transformation; and (finally) the Preview page shows what a user will see in a browser.

To make this actually work, you must provide some data to the `XSLPageProducer` component, via its `XMLData` property. This property can be hooked up to an `XMLDocument` but also directly to an `XMLBroker` component, as I've done in this case. The broker takes its data from a provider connected to a local table, attached to the `Customers` table of the classic `DBDEMOS`.

The effect is that, with the following Delphi-generated XSL, you get (even at design time) the output of Figure 23.15.

**FIGURE 23.15:**

The result of an XSLT transformation generated (even at design time) by the `XSLPageProducer` component in the `XslCust` example

CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Phone	FAX
1221	Kauai Dive Shoppe	4.976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	808-555-0269	808-555-0278
1231	Unisco	PO Box Z-547	Freeport	Bahamas	809-555-3915	809-555-4938	0	George Weathers	19941117T14:10:33000
1351	Sight Diver	1 Neptune Lane	Kato Paphos	Cyprus	357-6-876708	357-6-870943	0	Phyllis Spooner	19941018T19:20:30000
1354	Cayman Divers World Unlimited	PO Box 541	Grand Cayman	British West Indies	011-5-697044	011-5-697064	0	Joe Bailey	19920130T02:00:56000
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	Christiansted	St. Croix	00820	US Virgin Islands	504-798-3022	504-798-7772	0
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310	Waipahu	HI	99776	US	401-609-7623	401-609-9403
1384	VIP Divers Club	32 Main St.	Christiansted	St. Croix	02800	US Virgin Islands	809-453-5976	809-453-5932	0
1510	Ocean Paradise	PO Box 8745	Kailua-Kona	HI	94756	US	808-555-8231	808-555-8450	0
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	Bogota	Columbia	057-1-773434	057-1-773421	0	Susan Wong	19940718T17:17:01000

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>

  <xsl:template match="DATAPACKET">
    <table border="1">
      <xsl:apply-templates select="METADATA/FIELDS"/>
      <xsl:apply-templates select="ROWDATA/ROW"/>
    </table>
  </xsl:template>

  <xsl:template match="FIELDS">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

  <xsl:template match="FIELD">
    <th><xsl:value-of select="@attrname"/></th>
  </xsl:template>

  <xsl:template match="ROWDATA/ROW">
    <tr><xsl:for-each select="@*">
      <td><xsl:value-of/></td>
    </xsl:for-each></tr>
  </xsl:template>
</xsl:stylesheet>
```

This code, based heavily on XSL templates, generates an HTML table made of the expansion of field metadata and row data. The fields are used to generate the table heading, with a `<th>` cell for each entry in a single row. The row data is used to fill in the other rows of the table with the value of each attribute (`select="@*"`). At this point, you should be able to modify this XSL file and change the output of the program.

## Direct XSL Transformations with the DOM

Using the `XSLPageProducer` can certainly be handy, but generating multiple pages based on the same data just to handle different possible XSL styles with `WebSnap` doesn't seem to be the best approach. I've rather built a plain CGI application, called `CdsXstl`, that can transform a `ClientDataSet` data packet into different types of HTML, depending on the name of the XSL file passed as parameter to it. The advantage is that I can not only modify but even add new XSL files to the system without having to recompile the program.

To obtain the XSL transformation, the program loads both the XML and the XSL files into two XMLDocument components, called `xmlDom` and `XslDom`. Then it invokes the `transformNode` method of the XML document, passing the XSL document as parameter and filling in a third XMLDocument component, called `HtmlDom`:

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  xslfile: string;
  attr: IDOMAttr;
begin
  // open the client dataset and load its XML in a DOM
  ClientDataSet1.Open;
  XmlDom.Xml.Text := ClientDataSet1.XMLData;
  XmlDom.Active := True;
  // load the requested xsl file
  xslfile := Request.QueryFields.Values ['style'];
  if xslfile = '' then
    xslfile := 'customer.xsl';
  xslDom.LoadFromFile ('c:\websites\xsl\' + xslfile);
  XSLDom.Active := True;
  if xslfile = 'single.xsl' then
    begin
      attr := xslDom.DOMDocument.createAttribute('select');
      attr.value := '//ROW[@CustNo="' + Request.QueryFields.Values ['id'] + '"]';
      xslDom.DOMDocument.getElementsByTagName ('xsl:apply-templates').
        item[0].attributes.setNamedItem(attr);
    end;
  // do the transformation
  HTMLDom.Active := True;
  xmlDom.DocumentElement.transformNode (xslDom.DocumentElement, HTMLDom);
  Response.Content := HTMLDom.XML.Text;
end;

```

The code uses the DOM to modify the XSL document for displaying a single record, adding the XPath statement for selecting the record indicated by the `id` query field. This `id` is added to the hyperlink by the XSL with the list of records, but here I'd rather skip listing more XSL files. You can study them, as they are available in the XSL subfolder of the folder for this example.

### WARNING

To run this program, the XSL files should be deployed to the folder `c:\websites\xsl\`, or anywhere you like after fixing the source code accordingly.

## Web Services

Of all of the new features of Delphi 6, one stands out quite clearly: the support for Web services built into the product. The fact I'm discussing it at the end of the book has nothing to do with its importance, but only with the logical flow of the text, and with the fact that this is indeed not the starting point to learn Delphi programming.

But what is a Web service? It is a rapidly emerging technology that has the potential to change the way the Internet works for businesses. Browsing Web pages to enter your orders is fine for individuals (so-called B2C or business-to-consumer applications) but not for companies (so-called B2B or business-to-business applications). If you want to buy a few books, going to a book vendor Web site and punching in your requests is probably fine. But if you run a bookstore and want to place hundreds of orders a day, this is far from a good approach, particularly if you have a program that helps you track your sales and determine reorders. Grabbing the output of this program and reentering it into another application is really ridiculous.

The idea of Web services is to solve this issue: The program used to track sales can automatically create a request and send it to a Web service, which can immediately return information about the order. The next step might be to ask for a tracking number for the shipment. At this point, your program can use another Web service to track the shipment until it is at its destination, so you can tell your customers how long they have to wait. As the shipment arrives, your program can send a reminder via SMS or pager to the people with pending orders, issue a payment with a bank Web service, and ... I could continue but I think I've given you an idea. Web services are for computer interoperability as much as the Web and email let people interact.

### SOAP and WSDL

If the idea of Web services should be clear by now, what makes them possible is the Simple Object Access Protocol (SOAP). SOAP is built over standard HTTP, so that a Web server can handle the SOAP requests and the related data packets can pass through firewalls. SOAP defines an XML-based notation for requesting the execution of a method by an object on the server, passing parameters to it, and a notation to define the format of a response.

---

**NOTE**

SOAP was originally developed by DevelopMentor (the training company of COM expert Don Box) and Microsoft, to overcome weaknesses of using DCOM inside Web servers. Submitted to the W3C for standardization, it is being embraced by many companies, with a particular push from IBM. It is too early to see whether there will be a real standardization to let software programs from Microsoft, IBM, Sun, Oracle, and many others truly interoperate or whether some of these vendors will try to push a private version of the standard. In any case, SOAP is a cornerstone of Microsoft's dotNet architecture but also of the current proposals by Sun and Oracle.

SOAP is going to replace COM invocation, at least between different types of computers. Similarly, the definition of a SOAP service in the Web Services Description Language (WSDL) format is going to replace the IDL and type libraries used by COM and COM+. WSDL documents are another type of XML document that provides the metadata definition of a SOAP request. As you get a file in this format (generally published to define a service), you'll be able to create a program to call it.

Specifically, Delphi 6 provides a bidirectional mapping between WSDL and interfaces. This means you can grab a WSDL file and generate an interface for it. At this point, you can create a client program embedding SOAP requests via these interfaces and use a special Delphi component that allows you to convert your local interface requests into SOAP calls (as I doubt you want to manually generate the XML required for a SOAP request).

The other way around, you can define an interface (or use an existing one) and let a Delphi component generate a WSDL description for it. Another component provides you with a SOAP-to-Pascal mapping, so that by embedding this component and an object implementing the interface within a server-side program, you can have your Web service up and running in matter of minutes.

## BabelFish Translations

As a first example of the use of Web service, I'm going to build a simple client for the BabelFish translation service offered by AltaVista. Because this is an experimental service, like most others, there is no guarantee that the service will be working by the time you read this. You can find this and other services for experiments on the XMethods Web site ([www.xmethods.com](http://www.xmethods.com)) and also look for sample Web services provided by my own site ([www.marcocantu.com](http://www.marcocantu.com)).

After downloading the WSDL description of this service from XMethods (also available on the CD-ROM), I invoked Delphi's Web Services Importer in the Web Services page of the New items dialog box and selected the file. Delphi generated an Object Pascal interface for the Web service, as follows:

```
unit babelintf;  
  
interface  
  
uses  
    Types, XSBuiltIns;  
  
type  
    BabelFishPortType = interface(IInvokable)  
        ['{DF96B8F8-DD8E-43A1-9276-4F821D9EA3FA}']  
        function BabelFish(const translationmode: String;  
            const sourcedata: String): String; stdcall;
```

```
end;
```

### implementation

```
uses InvokeRegistry;
```

### initialization

```
  InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType), '', '');  
end.
```

Notice first that the interface inherits from the `IInvokable` interface. This doesn't add anything in terms of methods to the `IInterface` base interface of Delphi, but is compiled with the flag used to enable RTTI generation, `{$M+}`, like the `TPersistent` class. In this code, you can also see that the interface is registered in the global invocation registry (or `InvRegistry`), passing the type information reference of the interface type.

## NOTE

Having RTTI information for interfaces is actually the most important technological advance underlying SOAP invocation. Not that SOAP-to-Pascal mapping isn't important, as it is crucial to simplify the process, but having RTTI for an interface is what makes the entire architecture powerful and robust.

Once you have converted a WSDL definition into an easy-to-use interface, you need a component translating from interface call to SOAP call, and also handling the response and possible errors. This role can be played by the `HTTPRIO` component, which implements the idea of a Remote Invocation Object (RIO) over HTTP. Delphi 6 was built opening up the road for SOAP, but also keeping it open for other remote invocation mechanisms.

In the `BabelFish` example, the `HTTPRIO` component has the following settings, obtained by choosing the WSDL file first and then selecting the only available service and port from it, directly in the drop-down lists of the Object Inspector:

```
object HTTPRIO1: THHTTPRIO  
  WSDLLocation = 'C:\md6code\23\BabelFish\BabelFishService.xml'  
  Service = 'BabelFish'  
  Port = 'BabelFishPort'  
  HTTPWebNode.Agent = 'Borland SOAP 1.1'  
  Converter.Options = [soSendMultiRefObj, soTryAllSchema]  
end
```

At this point, there is very little left to do. We have information about the service that can be used for its invocation, and we know the parameters required by the only available method. The two elements are merged by extracting the interface you want to call directly from the `HTTPRIO` component, with an expression like `HTTPRIO1 as BabelFishPortType`. It might

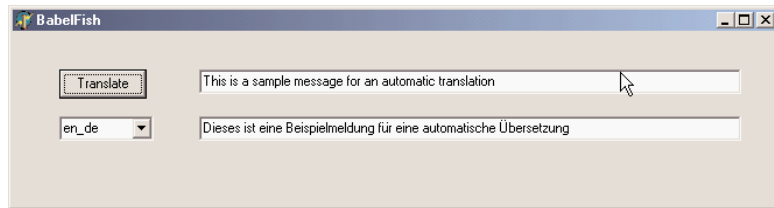
seem rather astonishing at first, but it is also outrageously simple. This is the Web service call done by the example:

```
EditTarget.Text := (HTTPRIO1 as BabelFishPortType).
  BabelFish(ComboBoxType.Text, EditSource.Text);
```

The resulting output of the program, depicted in Figure 23.16, allows you to learn foreign languages (although the teacher here has its shortcomings!). I haven't replicated the same example with stock options, currencies, weather forecasts, and the many other services available, as they would look much the same.

**FIGURE 23.16:**

An example of a translation from English to German obtained by AltaVista's BabelFish via a Web service



## Building a Web Service

If calling a Web service in Delphi 6 is very straightforward, the same can be said of the development of an actual service. If you go into the Web Services page of the New items dialog box, you can see the SOAP Server Application option. Selecting it, Delphi presents you a list quite similar to selection of a WebBroker application. A Web service, in fact, is typically hosted by a Web server, using one of the available Web server extension technologies (CGI, ISAPI, Apache modules, etc.). After completing this step, Delphi will add three components to the resulting Web module, which is just a plain Web module, with no special additions:

- The HTTPSoapDispatcher component has the role of receiving the Web request, as any other HTTP dispatcher does.
- The HTTPSoapPascalInvoker component does the reverse operation of the HTTPRIO component, as it can translate SOAP requests into calls of Pascal interfaces (instead of shifting interface method calls into SOAP requests).
- The WSDLHTMLPublish component can be used to extract the WSDL definition of the service from the interfaces it support, and performs the opposite role of the Web Services Importer Wizard. Technically, this is another HTTP dispatcher.

Once this framework is in place—something you can also do by adding the three components above to an existing Web module—we can start writing a service. As an example I've taken the euro conversion example of Chapter 4, "The Run-Time Library," and transformed



it into a Web service called ConvertService. First of all, I've added to the program a unit defining the interface of the service, as follows:

```
type
  IConvert = interface(IInvokable)
    ['{FF1EAA45-0B94-4630-9A18-E768A91A78E2}']
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
end;
```

Defining an interface directly in code, without having to use a tool such as the Type Library Editor, provides a great advantage. Notice that I've given a GUID to the interface, as usual, and used the `stdcall` calling convention, as the SOAP converter does not support the default register calling convention.

In the same unit defining the interface of the service, we should also register it, an operation which will be useful on both the client and server sides of the program, as we will be able to include this interface definition unit in both.

```
uses InvokeRegistry;
```

#### **initialization**

```
  InvRegistry.RegisterInterface(TypeInfo(IConvert));
```

Now that we have an interface we can expose to the public, we have to provide an implementation for it, again by means of the standard Pascal code (and with the help of the predefined `TInvokableClass` class:

```
type
  TConvert = class (TInvokableClass, IConvert)
  protected
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
end;
```

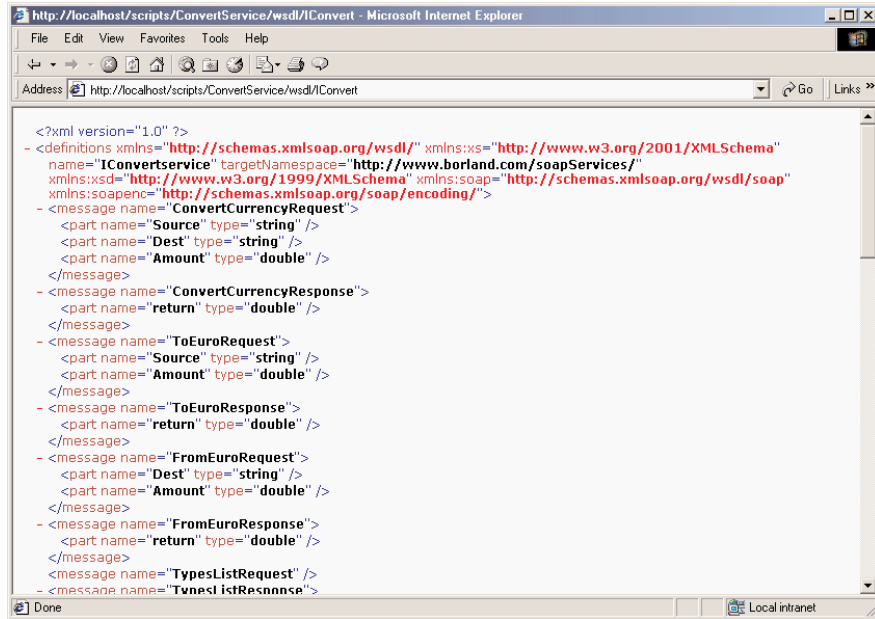
The implementation of these functions, which call in the code of the euro conversion system of Chapter 4, is not discussed here as it has little to do with the development of the service itself. What is important to notice, instead, is that this implementation unit also has a registration call in its initialization section:

```
  InvRegistry.RegisterInvokableClass (TConvert);
```

This is basically all. By registering the interface, we'll make it possible for the program to generate a WSDL description, as you can see in Figure 23.17, where I've used a browser to connect to the *wSDL* action of the service, implemented by the WSDLHTMLPublish component.

**FIGURE 23.17:**

The WSDL file generated automatically from the interface published by the ConvertService program



Of course, you cannot call the service from a browser, as the role of a Web service is not to display data on the Web but rather to let applications interoperate (still, you can reasonably have a Web application calling a service). Before I discuss the client application, though, let me cover another interesting element. For debugging purposes, I've added to the Web module an actual action, connected with code to generate information about the registered interfaces and servers:

```
Response.Content :=
  '<h3>GetMethExternalName - ToEuro</h3><p>' +
  InvRegistry.GetMethExternalName(TypeInfo(IConvert), 'ToEuro</p>') +
  '<h3>GetInterfaceExternalName - IConvert</h3><p>' +
  InvRegistry.GetInterfaceExternalName(TypeInfo(IConvert)) + '</p>' +
  '<h3>GetNamespaceByGUID - IConvert</h3><p>' +
  InvRegistry.GetNamespaceByGUID (IConvert) + '</p>';
```

The first call verifies whether a given method is properly registered and really exists, by calling the `GetMethExternalName` of the invocation registry. The result, unsurprisingly, is the same string passed as parameter, `ToEuro`. The second call, `GetInterfaceExternalName`,

should return the external name of the interface, but I haven't been able to make it work properly (I left it in anyway, as it is supposed to work). The last call, `GetNamespaceByGUID`, returns the XML namespace of the interface, `urn:ConvertIntf-IConvert`. There are other similar calls you can make against the registry, which are quite interesting and demonstrate the power of this approach and of RTTI for interfaces.

Having said this, let me move to the client application, calling the service. This time I don't really need to start from the WSDL file, as I already have the Pascal interface. This time the form doesn't even have the `HTTPRIO` component, which is created in code:

```
private
    Invoker: THTTPRIO;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Invoker := THTTPRIO.Create(nil);
    Invoker.URL := 'http://localhost/scripts/ConvertService.exe/soap/convert';
    ConvIntf := Invoker as IConvert;
end;
```

As an alternative to using a WSDL file, the SOAP invoker component can be associated with an URL. Once this association has been done and the required interface has been extracted from the component, you can start writing straight Pascal code to invoke the service, as we saw earlier.

A user can fill the two combo boxes, calling the `TypesList` method, which returns a list of available currencies within a string (separated by semicolons). This list is extracted by replacing any semicolon with a line break and then assigning the multiline string directly to the combo items:

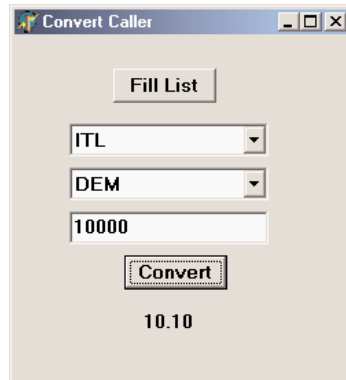
```
procedure TForm1.Button2Click(Sender: TObject);
var
    TypeNames: string;
begin
    TypeNames := ConvIntf.TypesList;
    ComboBoxFrom.Items.Text := StringReplace (TypeNames, ';', sLineBreak,
        [rfReplaceAll]);
    ComboBoxTo.Items := ComboBoxFrom.Items;
end;
```

At this point, after selecting two currencies, you can perform the conversion, with this code and the result of Figure 23.18:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    LabelResult.Caption := Format ('%n', [(ConvIntf.ConvertCurrency(
        ComboBoxFrom.Text, ComboBoxTo.Text, StrToFloat(EditAmount.Text))]);
end;
```

**FIGURE 23.18:**

The ConvertCaller client of the ConvertService Web service shows how few German marks you used to get for so many Italian liras, before the euro changed all this.



## DataSnap over SOAP

Now that we have a reasonably good idea of how to build a SOAP server and a SOAP client, we can have a look at how to use this technology in building a multitier DataSnap application. We'll use a Soap Server Data Module to create the new Web service and the SoapConnection component to connect a client application to it.

Let's look at the server side first. You have to move to the Web Services page of the New Items dialog box and use the Soap Server Application icon first to create a new Web service, and then use the Soap Server Data Module icon to add a DataSnap server-side data module to the SOAP server. This is what I've done in the SoapDataServer example (which uses the Web App Debugger architecture for testing purposes). From this point on, all you do is write a *normal* DataSnap server (or actually a middle-tier DataSnap application) as discussed in Chapter 17. In this specific case, I've added to the program InterBase access by means of dbExpress, resulting in the following structure:

```

object SoapTestDm: TSoapTestDm
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
  end
end

```

The data module built for a SOAP-based DataSnap server defines a custom interface (so you can add methods to it) inheriting from `IAppServer`, which is now defined as a published interface (even though it doesn't inherit from `IInvokable`). The implementation class, `TSoapTestDm`, is the data module itself, as in other DataSnap types of servers. This is the code Delphi generated for me:

```
type
  ISoapTestDm = interface(IAppServer)
    ['{1F109687-6D8B-4F85-9BF5-EFFC87A9F10F}']
  end;

  TSoapTestDm = class(TSoapDataModule, ISoapTestDm, IAppServer)
    DataSetProvider1: TDataSetProvider;
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
  end;
```

The base `TSoapDataModule` doesn't inherit from `TInvokableClass`. This is not a problem as long as you provide an extra procedure to create the object (which is what `TInvokableClass` does for you) and add it to the registration code:

```
procedure TSoapTestDmCreateInstance(out obj: TObject);
begin
  obj := TSoapTestDm.Create(nil);
end;

initialization
  InvRegistry.RegisterInvokableClass(TSoapTestDm, TSoapTestDmCreateInstance);
  InvRegistry.RegisterInterface(TypeInfo(ISoapTestDm));
```

The server application actually also publishes the `IAppServer` interface, thanks to the only line of code in the `SOAPMidas` unit.

---

**WARNING** Web service applications should not include more than one SOAP Data Module, as the registration cannot distinguish between multiple implementations of the same `IAppServer` interface.

To build the client application, called `SoapDataClient`, I've started with a plain program and added a `SoapConnection` component to it (from the Web Services page of the palette), hooking it to the URL of the DataSnap Web service, referring to the specific interface we are looking for:

```
object SoapConnection1: TSoapConnection
  Agent = 'Borland SOAP 1.1'
  URL = 'http://localhost:1024/SoapDataServer.SoopDataServer/Soap/ISoapTestDm'
end
```

From this point on, I've proceeded as usual, adding a `ClientDataSet` component, a `DataSource`, and a `DBGrid` to the program, choosing the only available provider for the client dataset, and hooking the rest as usual. Not surprisingly, for this simple example, the client application has little custom code: a single call to open the connection when a button is clicked (to avoid startup errors) and an `ApplyUpdates` call to send changes back to the database.

Regardless of the apparent similarity of this program to all of the other `DataSnap` client and server programs built in Chapter 17, there is a very important difference worth underlining: The `SoapDataServer` and `SoapDataClient` programs do not use COM for exposing or calling the `IAppServer` interface. Quite the opposite—the socket- and HTTP-based connections of `DataSnap` still rely on local COM objects and a registration of the server in the Windows Registry. The native SOAP-based support provided by Delphi 6, instead, allows for a totally custom solution independent from COM and with many more chances to be ported to other operating systems (Linux being certainly the first, with a future release of `Kylix`).

## What's Next?

In this final chapter of the book, I've covered XML and related technologies, including XSLT, SOAP, WSDL, XML schemas, XPath, and a few more. We've seen how Delphi 6 provides simplified DOM programming, XML access using interfaces, and XML transformations. I've also covered Internet Express and the development of Web services.

Besides tracking what goes on in the area of SOAP and WSDL, particularly in terms of standards conformance by the major players, there are a few interesting initiatives you should probably keep track of if you're interested in the development in business-to-business services. One of them is the UDDI proposal ([www.uddi.org](http://www.uddi.org)), pushed by Microsoft, IBM, Ariba, and many other companies, to create a universal registry of services. Another is ebXML ([www.ebxml.org](http://www.ebxml.org)), a proposal by the U.N. office that defined the EDI standards for an XML-based global business exchange.

Of course, I don't want to delve too much into these nontechnical issues, but I thought it was worth mentioning them at the end of this book, as I try to give a few hints at a sort of "what's next" for Delphi programmers. Delphi is indeed a strong player in both the Windows and Linux client markets, in the client/server and enterprise application markets, and now takes a bold step in the areas of Web development and Web services.

Just as Borland wants to provide the best tools to developers, I hope this book has helped you master Delphi, the most successful tool Borland has brought to the market in the last few years. Remember to check from time to time the reference, foundations, and advanced material I've collected on my Web site ([www.marcocantu.com](http://www.marcocantu.com)). Check my site also for eventual updates and integration of the material in the book, and feel free to use the newsgroups

hosted there for your questions about the book and about Delphi in general. Much of this material could not be included in the book, simply because of space constraints. Some of this extra material is actually already available on the companion CD, where you can continue reading about other aspects of Delphi programming.



# Developer's Guide

---



Borland®  
**Delphi™ 6**  
for Windows

Borland Software Corporation  
100 Enterprise Way, Scotts Valley, CA 95066-3249



Refer to the DEPLOY document located in the root directory of your Delphi 6 product for a complete list of files that you can distribute in accordance with the Delphi 6 License Statement and Limited Warranty.

Inprise may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 2001 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

HDE1350WW21001 1E0R0501

0102030405-9 8 7 6 5 4 3 2 1

D3

# Contents

<b>Chapter 1</b>	
<b>Introduction</b>	<b>1-1</b>
What's in this manual? . . . . .	1-1
Manual conventions . . . . .	1-3
Developer support services. . . . .	1-3
Ordering printed documentation . . . . .	1-3
<b>Part I</b>	
<b>Programming with Delphi</b>	
<hr/>	
<b>Chapter 2</b>	
<b>Developing applications with Delphi</b>	<b>2-1</b>
Integrated development environment. . . . .	2-1
Designing applications . . . . .	2-2
Developing applications . . . . .	2-3
Creating projects . . . . .	2-3
Editing code. . . . .	2-4
Compiling applications . . . . .	2-4
Debugging applications . . . . .	2-5
Deploying applications . . . . .	2-5
<b>Chapter 3</b>	
<b>Using the component libraries</b>	<b>3-1</b>
Understanding the component libraries. . . . .	3-1
Properties, methods, and events . . . . .	3-2
Properties . . . . .	3-2
Methods . . . . .	3-3
Events . . . . .	3-3
User events . . . . .	3-3
System events . . . . .	3-4
Object Pascal and the class libraries . . . . .	3-4
Using the object model . . . . .	3-4
What is an object?. . . . .	3-5
Examining a Delphi object. . . . .	3-5
Changing the name of a component . . . . .	3-7
Inheriting data and code from an object. . . . .	3-8
Scope and qualifiers . . . . .	3-8
Private, protected, public, and published declarations . . . . .	3-9
Using object variables . . . . .	3-10
Creating, instantiating, and destroying objects . . . . .	3-11
Components and ownership . . . . .	3-11
Objects, components, and controls. . . . .	3-12
TObject branch . . . . .	3-14
TPersistent branch. . . . .	3-14
TComponent branch . . . . .	3-15
TControl branch . . . . .	3-16
TWinControl/TWidgetControl branch . . . . .	3-17
Properties common to TControl . . . . .	3-18
Action properties . . . . .	3-18
Position, size, and alignment properties . . . . .	3-19
Display properties . . . . .	3-19
Parent properties. . . . .	3-19
A navigation property. . . . .	3-19
Drag-and-drop properties . . . . .	3-20
Drag-and-dock properties (VCL only) . . . . .	3-20
Standard events common to TControl . . . . .	3-20
Properties common to TWinControl and TWidgetControl . . . . .	3-21
General information properties . . . . .	3-21
Border style display properties. . . . .	3-22
Navigation properties. . . . .	3-22
Drag-and-dock properties (VCL only) . . . . .	3-22
Events common to TWinControl and TWidgetControl . . . . .	3-22
Creating the application user interface . . . . .	3-23
Using Delphi components . . . . .	3-23
Setting component properties . . . . .	3-24
Using the Object Inspector . . . . .	3-24
Using property editors . . . . .	3-25
Setting properties at runtime . . . . .	3-25
Calling methods. . . . .	3-25
Working with events and event handlers. . . . .	3-25
Generating a new event handler . . . . .	3-26
Generating a handler for a component's default event . . . . .	3-26
Locating event handlers. . . . .	3-26
Associating an event with an existing event handler. . . . .	3-27
Associating menu events with event handlers. . . . .	3-28
Deleting event handlers. . . . .	3-28
VCL and CLX components . . . . .	3-28
Adding custom components to the Component palette . . . . .	3-30
Text controls . . . . .	3-31
Text control properties. . . . .	3-31

Properties of memo and rich text controls . . . . .	3-31	Using helper objects . . . . .	3-46
Rich text controls (VCL only) . . . . .	3-32	Working with lists . . . . .	3-47
Specialized input controls . . . . .	3-32	Working with string lists . . . . .	3-47
Scroll bars . . . . .	3-32	Loading and saving string lists . . . . .	3-48
Track bars . . . . .	3-33	Creating a new string list . . . . .	3-48
Up-down controls (VCL only) . . . . .	3-33	Manipulating strings in a list . . . . .	3-50
Spin edit controls (CLX only) . . . . .	3-33	Associating objects with a string list . . . . .	3-52
Hot key controls (VCL only) . . . . .	3-33	Windows registry and INI files . . . . .	3-52
Splitter controls . . . . .	3-34	Using TIniFile (VCL only) . . . . .	3-52
Buttons and similar controls . . . . .	3-34	Using TRegistry . . . . .	3-53
Button controls . . . . .	3-34	Using TRegIniFile . . . . .	3-53
Bitmap buttons . . . . .	3-35	Creating drawing spaces . . . . .	3-54
Speed buttons . . . . .	3-35	Printing . . . . .	3-54
Check boxes . . . . .	3-35	Using streams . . . . .	3-55
Radio buttons . . . . .	3-35		
Toolbars . . . . .	3-36	<b>Chapter 4</b>	
Cool bars (VCL only) . . . . .	3-36	<b>Common programming tasks</b>	<b>4-1</b>
Handling lists . . . . .	3-36	Understanding classes . . . . .	4-1
List boxes and check-list boxes . . . . .	3-37	Defining classes . . . . .	4-2
Combo boxes . . . . .	3-37	Handling exceptions . . . . .	4-4
Tree views . . . . .	3-38	Protecting blocks of code . . . . .	4-4
List views . . . . .	3-38	Responding to exceptions . . . . .	4-5
Date-time pickers and month calendars (VCL only) . . . . .	3-39	Exceptions and the flow of control . . . . .	4-6
Grouping components . . . . .	3-39	Nesting exception responses . . . . .	4-6
Group boxes and radio groups . . . . .	3-39	Protecting resource allocations . . . . .	4-7
Panels . . . . .	3-39	What kind of resources need protection? . . . . .	4-7
Scroll boxes . . . . .	3-40	Creating a resource protection block . . . . .	4-8
Tab controls . . . . .	3-40	Handling RTL exceptions . . . . .	4-9
Page controls . . . . .	3-40	What are RTL exceptions? . . . . .	4-9
Header controls . . . . .	3-41	Creating an exception handler . . . . .	4-10
Providing visual feedback . . . . .	3-41	Exception handling statements . . . . .	4-11
Labels and static text components . . . . .	3-41	Using the exception instance . . . . .	4-11
Status bars . . . . .	3-42	Scope of exception handlers . . . . .	4-12
Progress bars . . . . .	3-42	Providing default exception handlers . . . . .	4-12
Help and hint properties . . . . .	3-42	Handling classes of exceptions . . . . .	4-13
Grids . . . . .	3-43	Reraising the exception . . . . .	4-13
Draw grids . . . . .	3-43	Handling component exceptions . . . . .	4-14
String grids . . . . .	3-43	Exception handling with external sources . . . . .	4-15
Value list editors (VCL only) . . . . .	3-43	Silent exceptions . . . . .	4-15
Displaying graphics . . . . .	3-44	Defining your own exceptions . . . . .	4-16
Images . . . . .	3-44	Declaring an exception object type . . . . .	4-16
Shapes . . . . .	3-44	Raising an exception . . . . .	4-17
Bevels . . . . .	3-45	Using interfaces . . . . .	4-17
Paint boxes . . . . .	3-45	Interfaces as a language feature . . . . .	4-18
Animation control (VCL only) . . . . .	3-45	Implementing interfaces across the hierarchy . . . . .	4-18
Developing dialog boxes . . . . .	3-45		
Using open dialog boxes . . . . .	3-46		

Using interfaces with procedures . . . . .	4-20	Passing a local variable as a PChar . . . . .	4-48
Implementing IInterface . . . . .	4-20	Compiler directives for strings. . . . .	4-49
TInterfacedObject . . . . .	4-21	Strings and characters: related topics . . . . .	4-50
Using the as operator . . . . .	4-21	Working with files . . . . .	4-50
Reusing code and delegation . . . . .	4-22	Manipulating files . . . . .	4-50
Using implements for delegation . . . . .	4-22	Deleting a file. . . . .	4-50
Aggregation . . . . .	4-23	Finding a file . . . . .	4-51
Memory management of interface		Renaming a file. . . . .	4-52
objects . . . . .	4-24	File date-time routines . . . . .	4-52
Using reference counting . . . . .	4-24	Copying a file . . . . .	4-53
Not using reference counting . . . . .	4-25	File types with file I/O . . . . .	4-53
Using interfaces in distributed		Using file streams . . . . .	4-54
applications (VCL only) . . . . .	4-26	Creating and opening files . . . . .	4-54
Defining custom variants . . . . .	4-27	Using the file handle . . . . .	4-55
Storing a custom variant type's data . . . . .	4-28	Reading and writing to files . . . . .	4-55
Creating a class to enable the custom		Reading and writing strings . . . . .	4-56
variant type . . . . .	4-28	Seeking a file . . . . .	4-56
Enabling casting . . . . .	4-29	File position and size . . . . .	4-57
Implementing binary operations . . . . .	4-30	Copying. . . . .	4-57
Implementing comparison operations . . . . .	4-32	Converting measurements. . . . .	4-58
Implementing unary operations . . . . .	4-34	Performing conversions . . . . .	4-58
Copying and clearing custom		Performing simple conversions . . . . .	4-58
variants. . . . .	4-34	Performing complex conversions . . . . .	4-58
Loading and saving custom variant		Adding new measurement types . . . . .	4-59
values. . . . .	4-35	Creating a simple conversion family	
Using the TCustomVariantType		and adding units. . . . .	4-59
descendant. . . . .	4-36	Using a conversion function . . . . .	4-60
Writing utilities to work with a custom		Using a class to manage	
variant type . . . . .	4-36	conversions . . . . .	4-62
Supporting properties and methods in		Defining data types. . . . .	4-64
custom variants . . . . .	4-37		
Using TInvokeableVariantType . . . . .	4-37	<b>Chapter 5</b>	
Using TPublishableVariantType. . . . .	4-39	<b>Building applications, components,</b>	
Working with strings . . . . .	4-39	<b>and libraries</b>	<b>5-1</b>
Character types . . . . .	4-39	Creating applications . . . . .	5-1
String types . . . . .	4-40	GUI applications. . . . .	5-1
Short strings . . . . .	4-40	User interface models . . . . .	5-2
Long strings . . . . .	4-41	SDI applications . . . . .	5-2
WideString. . . . .	4-41	MDI applications. . . . .	5-2
PChar types . . . . .	4-42	Setting IDE, project, and compilation	
OpenString . . . . .	4-42	options . . . . .	5-3
Runtime library string handling routines . . . . .	4-42	Programming templates . . . . .	5-3
Wide character routines . . . . .	4-43	Console applications . . . . .	5-3
Commonly used long string routines. . . . .	4-43	Service applications . . . . .	5-4
Declaring and initializing strings. . . . .	4-46	Service threads . . . . .	5-6
Mixing and converting string types . . . . .	4-47	Service name properties. . . . .	5-7
String to PChar conversions. . . . .	4-47	Debugging services . . . . .	5-8
String dependencies . . . . .	4-47	Creating packages and DLLs . . . . .	5-9
Returning a PChar local variable . . . . .	4-48	When to use packages and DLLs . . . . .	5-9

Writing database applications . . . . .	5-10	Registering Help selectors . . . . .	5-27
Distributing database applications . . . . .	5-11	Using Help in a VCL Application . . . . .	5-28
Creating Web server applications . . . . .	5-11	How TApplication processes VCL	
Using Web Broker . . . . .	5-11	Help . . . . .	5-28
Creating WebSnap applications . . . . .	5-13	How VCL controls process Help . . . . .	5-28
Using InternetExpress . . . . .	5-13	Using Help in a CLX Application . . . . .	5-29
Creating Web Services applications . . . . .	5-13	How TApplication processes CLX	
Writing applications using COM . . . . .	5-14	Help . . . . .	5-29
Using COM and DCOM . . . . .	5-14	How CLX controls process Help . . . . .	5-29
Using MTS and COM+ . . . . .	5-14	Calling a Help system directly . . . . .	5-30
Using data modules . . . . .	5-15	Using IHelpSystem . . . . .	5-30
Creating and editing standard data		Customizing the IDE Help system . . . . .	5-30
modules . . . . .	5-15		
Naming a data module and its unit			
file . . . . .	5-16		
Placing and naming components . . . . .	5-17		
Using component properties and			
events in a data module . . . . .	5-17		
Creating business rules in a data			
module . . . . .	5-18		
Accessing a data module from a form . . . . .	5-18		
Adding a remote data module to an			
application server project . . . . .	5-19		
Using the Object Repository . . . . .	5-19		
Sharing items within a project . . . . .	5-19		
Adding items to the Object Repository . . . . .	5-19		
Sharing objects in a team environment . . . . .	5-20		
Using an Object Repository item in			
a project . . . . .	5-20		
Copying an item . . . . .	5-20		
Inheriting an item . . . . .	5-20		
Using an item . . . . .	5-21		
Using project templates . . . . .	5-21		
Modifying shared items . . . . .	5-21		
Specifying a default project, new form,			
and main form . . . . .	5-21		
Enabling Help in applications . . . . .	5-22		
Help system interfaces . . . . .	5-22		
Implementing ICustomHelpViewer . . . . .	5-23		
Communicating with the Help			
Manager . . . . .	5-23		
Asking the Help Manager for			
information . . . . .	5-24		
Displaying keyword-based Help . . . . .	5-24		
Displaying tables of contents . . . . .	5-25		
Implementing IExtendedHelpViewer . . . . .	5-26		
Implementing IHelpSelector . . . . .	5-26		
Registering Help system objects . . . . .	5-27		
Registering Help viewers . . . . .	5-27		

## Chapter 6 Developing the application user interface 6-1

Controlling application behavior . . . . .	6-1
Using the main form . . . . .	6-1
Adding forms . . . . .	6-2
Linking forms . . . . .	6-2
Avoiding circular unit references . . . . .	6-2
Hiding the main form . . . . .	6-3
Working at the application level . . . . .	6-3
Handling the screen . . . . .	6-3
Managing layout . . . . .	6-4
Responding to event notification . . . . .	6-5
Using forms . . . . .	6-5
Controlling when forms reside in	
memory . . . . .	6-6
Displaying an auto-created form . . . . .	6-6
Creating forms dynamically . . . . .	6-6
Creating modeless forms such as	
windows . . . . .	6-7
Using a local variable to create a form	
instance . . . . .	6-7
Passing additional arguments to forms . . . . .	6-8
Retrieving data from forms . . . . .	6-9
Retrieving data from modeless forms . . . . .	6-9
Retrieving data from modal forms . . . . .	6-10
Reusing components and groups of	
components . . . . .	6-12
Creating and using component templates . . . . .	6-13
Working with frames . . . . .	6-13
Creating frames . . . . .	6-14
Adding frames to the component	
palette . . . . .	6-14
Using and modifying frames . . . . .	6-14
Sharing frames . . . . .	6-15

Organizing actions for toolbars and menus . . . . .	6-16	Merging menus . . . . .	6-40
What is an action? . . . . .	6-17	Specifying the active menu: Menu property. . . . .	6-40
Setting up action bands . . . . .	6-18	Determining the order of merged menu items: GroupIndex property . . . . .	6-40
Creating toolbars and menus . . . . .	6-18	Importing resource files . . . . .	6-41
Adding color, patterns, or pictures to menus, buttons, and toolbars . . . . .	6-20	Designing toolbars and cool bars . . . . .	6-41
Adding icons to menus and toolbars . . . . .	6-21	Adding a toolbar using a panel component . . . . .	6-42
Creating toolbars and menus that users can customize. . . . .	6-21	Adding a speed button to a panel . . . . .	6-43
Hiding unused items and categories in action bands . . . . .	6-22	Assigning a speed button's glyph . . . . .	6-43
Using action lists . . . . .	6-23	Setting the initial condition of a speed button . . . . .	6-43
Setting up action lists . . . . .	6-23	Creating a group of speed buttons. . . . .	6-44
What happens when an action fires . . . . .	6-24	Allowing toggle buttons . . . . .	6-44
Responding with events . . . . .	6-24	Adding a toolbar using the toolbar component . . . . .	6-44
How actions find their targets. . . . .	6-26	Adding a tool button . . . . .	6-45
Updating actions . . . . .	6-26	Assigning images to tool buttons . . . . .	6-45
Predefined action classes . . . . .	6-26	Setting tool button appearance and initial conditions . . . . .	6-46
Writing action components . . . . .	6-27	Creating groups of tool buttons . . . . .	6-46
Registering actions . . . . .	6-28	Allowing toggled tool buttons . . . . .	6-46
Creating and managing menus. . . . .	6-29	Adding a cool bar component . . . . .	6-47
Opening the Menu Designer . . . . .	6-29	Setting the appearance of the cool bar . . . . .	6-47
Building menus. . . . .	6-31	Responding to clicks . . . . .	6-48
Naming menus . . . . .	6-31	Assigning a menu to a tool button. . . . .	6-48
Naming the menu items . . . . .	6-31	Adding hidden toolbars . . . . .	6-48
Adding, inserting, and deleting menu items . . . . .	6-32	Hiding and showing toolbars . . . . .	6-48
Adding separator bars . . . . .	6-32	Demo programs . . . . .	6-49
Specifying accelerator keys and keyboard shortcuts . . . . .	6-33		
Creating submenus. . . . .	6-33		
Creating submenus by demoting existing menus . . . . .	6-34		
Moving menu items . . . . .	6-34		
Adding images to menu items . . . . .	6-35		
Viewing the menu . . . . .	6-35		
Editing menu items in the Object Inspector. . . . .	6-35		
Using the Menu Designer context menu . . . . .	6-36		
Commands on the context menu . . . . .	6-36		
Switching between menus at design time. . . . .	6-37		
Using menu templates. . . . .	6-37		
Saving a menu as a template . . . . .	6-38		
Naming conventions for template menu items and event handlers . . . . .	6-39		
Manipulating menu items at runtime . . . . .	6-40		
		<b>Chapter 7</b>	
		<b>Working with controls</b>	<b>7-1</b>
		Implementing drag-and-drop in controls . . . . .	7-1
		Starting a drag operation. . . . .	7-1
		Accepting dragged items. . . . .	7-2
		Dropping items . . . . .	7-2
		Ending a drag operation . . . . .	7-3
		Customizing drag and drop with a drag object . . . . .	7-3
		Changing the drag mouse pointer. . . . .	7-4
		Implementing drag-and-dock in controls . . . . .	7-4
		Making a windowed control a docking site. . . . .	7-4
		Making a control a dockable child. . . . .	7-4
		Controlling how child controls are docked . . . . .	7-5

Controlling how child controls are unlocked . . . . .	7-6
Controlling how child controls respond to drag-and-dock operations . . . . .	7-6
Working with text in controls. . . . .	7-6
Setting text alignment . . . . .	7-7
Adding scroll bars at runtime. . . . .	7-7
Adding the clipboard object. . . . .	7-8
Selecting text . . . . .	7-8
Selecting all text . . . . .	7-9
Cutting, copying, and pasting text . . . . .	7-9
Deleting selected text . . . . .	7-9
Disabling menu items . . . . .	7-10
Providing a pop-up menu. . . . .	7-10
Handling the OnPopup event. . . . .	7-11
Adding graphics to controls . . . . .	7-11
Indicating that a control is owner-drawn . . . . .	7-12
Adding graphical objects to a string list . . . . .	7-12
Adding images to an application . . . . .	7-13
Adding images to a string list . . . . .	7-13
Drawing owner-drawn items . . . . .	7-13
Sizing owner-draw items . . . . .	7-14
Drawing owner-draw items. . . . .	7-15

## Chapter 8 Working with graphics and multimedia

**8-1**

Overview of graphics programming. . . . .	8-1
Refreshing the screen . . . . .	8-2
Types of graphic objects . . . . .	8-3
Common properties and methods of Canvas . . . . .	8-4
Using the properties of the Canvas object . . . . .	8-5
Using pens. . . . .	8-5
Using brushes . . . . .	8-8
Reading and setting pixels. . . . .	8-9
Using Canvas methods to draw graphic objects . . . . .	8-9
Drawing lines and polylines. . . . .	8-10
Drawing shapes. . . . .	8-11
Handling multiple drawing objects in your application . . . . .	8-12
Keeping track of which drawing tool to use . . . . .	8-12
Changing the tool with speed buttons . . . . .	8-13
Using drawing tools . . . . .	8-13
Drawing on a graphic . . . . .	8-16

Making scrollable graphics . . . . .	8-16
Adding an image control . . . . .	8-17
Loading and saving graphics files. . . . .	8-18
Loading a picture from a file . . . . .	8-19
Saving a picture to a file. . . . .	8-19
Replacing the picture . . . . .	8-20
Using the clipboard with graphics . . . . .	8-21
Copying graphics to the clipboard. . . . .	8-21
Cutting graphics to the clipboard . . . . .	8-21
Pasting graphics from the clipboard. . . . .	8-22
Rubber banding example. . . . .	8-23
Responding to the mouse. . . . .	8-23
Responding to a mouse-down action . . . . .	8-24
Adding a field to a form object to track mouse actions . . . . .	8-26
Refining line drawing . . . . .	8-27
Working with multimedia . . . . .	8-28
Adding silent video clips to an application . . . . .	8-29
Example of adding silent video clips . . . . .	8-30
Adding audio and/or video clips to an application . . . . .	8-30
Example of adding audio and/or video clips (VCL only) . . . . .	8-32

## Chapter 9 Writing multi-threaded applications

**9-1**

Defining thread objects. . . . .	9-1
Initializing the thread . . . . .	9-2
Assigning a default priority . . . . .	9-2
Indicating when threads are freed . . . . .	9-3
Writing the thread function . . . . .	9-4
Using the main VCL/CLX thread . . . . .	9-4
Using thread-local variables . . . . .	9-5
Checking for termination by other threads . . . . .	9-5
Handling exceptions in the thread function . . . . .	9-6
Writing clean-up code. . . . .	9-6
Coordinating threads. . . . .	9-7
Avoiding simultaneous access . . . . .	9-7
Locking objects. . . . .	9-7
Using critical sections . . . . .	9-7
Using the multi-read exclusive-write synchronizer . . . . .	9-8
Other techniques for sharing memory. . . . .	9-8
Waiting for other threads. . . . .	9-9
Waiting for a thread to finish executing . . . . .	9-9

Waiting for a task to be completed . . . . .	9-9
Executing thread objects . . . . .	9-10
Overriding the default priority . . . . .	9-11
Starting and stopping threads . . . . .	9-11
Debugging multi-threaded applications . . . . .	9-12

## Chapter 10 Using CLX for cross-platform development 10-1

Creating cross-platform applications . . . . .	10-1
Porting VCL applications to CLX . . . . .	10-2
Porting techniques . . . . .	10-3
Platform-specific ports . . . . .	10-3
Cross-platform ports . . . . .	10-3
Windows emulation ports . . . . .	10-3
Porting your application. . . . .	10-4
CLX versus VCL . . . . .	10-5
What CLX does differently . . . . .	10-6
Look and feel . . . . .	10-6
Styles . . . . .	10-6
Variants . . . . .	10-7
Registry . . . . .	10-7
Other differences . . . . .	10-7
Missing in CLX . . . . .	10-8
Features that will not port. . . . .	10-8
CLX and VCL unit comparison. . . . .	10-9
Differences in CLX object constructors . . . . .	10-13
Sharing source files between Windows and Linux . . . . .	10-13
Environmental differences between Windows and Linux . . . . .	10-14
Directory structure on Linux . . . . .	10-16
Writing portable code . . . . .	10-17
Using conditional directives . . . . .	10-18
Terminating conditional directives . . . . .	10-19
Emitting messages . . . . .	10-20
Including inline assembler code. . . . .	10-20
Messages and system events . . . . .	10-21
Programming differences on Linux . . . . .	10-22
Cross-platform database applications . . . . .	10-23
dbExpress differences . . . . .	10-23
Component-level differences . . . . .	10-24
User interface-level differences . . . . .	10-25
Porting database applications to Linux. . . . .	10-25
Updating data in dbExpress applications . . . . .	10-27
Cross-platform Internet applications . . . . .	10-29

Porting Internet applications to Linux . . . . .	10-29
--	-------

## Chapter 11 Working with packages and components 11-1

Why use packages? . . . . .	11-2
Packages and standard DLLs . . . . .	11-2
Runtime packages . . . . .	11-2
Using packages in an application . . . . .	11-3
Dynamically loading packages . . . . .	11-4
Deciding which runtime packages to use . . . . .	11-4
Custom packages . . . . .	11-4
Design-time packages . . . . .	11-5
Installing component packages . . . . .	11-5
Creating and editing packages . . . . .	11-6
Creating a package . . . . .	11-6
Editing an existing package . . . . .	11-7
Editing package source files manually . . . . .	11-8
Understanding the structure of a package . . . . .	11-8
Naming packages . . . . .	11-8
Requires clause. . . . .	11-8
Contains clause. . . . .	11-9
Compiling packages . . . . .	11-10
Package-specific compiler directives . . . . .	11-10
Using the command-line compiler and linker . . . . .	11-12
Package files created by a successful compilation . . . . .	11-12
Deploying packages . . . . .	11-13
Deploying applications that use packages . . . . .	11-13
Distributing packages to other developers . . . . .	11-13
Package collection files . . . . .	11-13

## Chapter 12 Creating international applications 12-1

Internationalization and localization . . . . .	12-1
Internationalization . . . . .	12-1
Localization . . . . .	12-2
Internationalizing applications . . . . .	12-2
Enabling application code . . . . .	12-2
Character sets . . . . .	12-2
OEM and ANSI character sets . . . . .	12-3
Multibyte character sets. . . . .	12-3
Wide characters . . . . .	12-4



Including bi-directional functionality in applications . . . . .	12-4
BiDiMode property . . . . .	12-6
Locale-specific features . . . . .	12-8
Designing the user interface . . . . .	12-9
Text . . . . .	12-9
Graphic images . . . . .	12-9
Formats and sort order . . . . .	12-10
Keyboard mappings . . . . .	12-10
Isolating resources . . . . .	12-10
Creating resource DLLs . . . . .	12-10
Using resource DLLs . . . . .	12-12
Dynamic switching of resource DLLs . . . . .	12-13
Localizing applications . . . . .	12-13
Localizing resources . . . . .	12-13

## Chapter 13 Deploying applications **13-1**

Deploying general applications . . . . .	13-1
Using installation programs . . . . .	13-2
Identifying application files . . . . .	13-2
Application files . . . . .	13-3
Package files . . . . .	13-3
Merge modules . . . . .	13-3
ActiveX controls . . . . .	13-5
Helper applications . . . . .	13-5
DLL locations . . . . .	13-5
Deploying CLX applications . . . . .	13-6
Deploying database applications . . . . .	13-6
Deploying dbExpress database applications . . . . .	13-7
Deploying BDE applications . . . . .	13-8
Borland Database Engine . . . . .	13-8
SQL Links . . . . .	13-8
Deploying multi-tiered database applications (DataSnap) . . . . .	13-9
Deploying Web applications . . . . .	13-9
Deployment on Apache . . . . .	13-10
Programming for varying host environments . . . . .	13-11
Screen resolutions and color depths . . . . .	13-11
Considerations when not dynamically resizing . . . . .	13-11
Considerations when dynamically resizing forms and controls . . . . .	13-12
Accommodating varying color depths . . . . .	13-13
Fonts . . . . .	13-13
Operating systems versions . . . . .	13-14

Software license requirements . . . . .	13-14
DEPLOY . . . . .	13-14
README . . . . .	13-15
No-nonsense license agreement . . . . .	13-15
Third-party product documentation . . . . .	13-15

## Part II Developing database applications

---

### Chapter 14 Designing database applications **14-1**

Using databases . . . . .	14-1
Types of databases . . . . .	14-2
Database security . . . . .	14-3
Transactions . . . . .	14-4
Referential integrity, stored procedures, and triggers . . . . .	14-5
Database architecture . . . . .	14-5
General structure . . . . .	14-6
The user interface form . . . . .	14-6
The data module . . . . .	14-6
Connecting directly to a database server . . . . .	14-7
Using a dedicated file on disk . . . . .	14-9
Connecting to another dataset . . . . .	14-10
Connecting a client dataset to another dataset in the same application . . . . .	14-11
Using a multi-tiered architecture . . . . .	14-12
Combining approaches . . . . .	14-14
Designing the user interface . . . . .	14-15
Analyzing data . . . . .	14-15
Writing reports . . . . .	14-16

### Chapter 15 Using data controls **15-1**

Using common data control features . . . . .	15-2
Associating a data control with a dataset . . . . .	15-3
Changing the associated dataset at runtime . . . . .	15-3
Enabling and disabling the data source . . . . .	15-4
Responding to changes mediated by the data source . . . . .	15-4
Editing and updating data . . . . .	15-5
Enabling editing in controls on user entry . . . . .	15-5
Editing data in a control . . . . .	15-5

Disabling and enabling data display . . . . .	15-6
Refreshing data display . . . . .	15-6
Enabling mouse, keyboard, and timer events . . . . .	15-7
Choosing how to organize the data . . . . .	15-7
Displaying a single record. . . . .	15-7
Displaying data as labels. . . . .	15-8
Displaying and editing fields in an edit box. . . . .	15-8
Displaying and editing text in a memo control . . . . .	15-8
Displaying and editing text in a rich edit memo control. . . . .	15-9
Displaying and editing graphics fields in an image control . . . . .	15-9
Displaying and editing data in list and combo boxes . . . . .	15-10
Handling Boolean field values with check boxes . . . . .	15-12
Restricting field values with radio controls. . . . .	15-13
Displaying multiple records. . . . .	15-14
Viewing and editing data with TDBGrid . . . . .	15-15
Using a grid control in its default state . . . . .	15-15
Creating a customized grid . . . . .	15-16
Understanding persistent columns . . . . .	15-16
Creating persistent columns. . . . .	15-17
Deleting persistent columns . . . . .	15-18
Arranging the order of persistent columns . . . . .	15-19
Setting column properties at design time. . . . .	15-19
Defining a lookup list column. . . . .	15-20
Putting a button in a column . . . . .	15-21
Restoring default values to a column . . . . .	15-21
Displaying ADT and array fields. . . . .	15-21
Setting grid options . . . . .	15-23
Editing in the grid . . . . .	15-25
Controlling grid drawing . . . . .	15-25
Responding to user actions at runtime. . . . .	15-25
Creating a grid that contains other data-aware controls . . . . .	15-26
Navigating and manipulating records. . . . .	15-28
Choosing navigator buttons to display . . . . .	15-28
Hiding and showing navigator buttons at design time . . . . .	15-29

Hiding and showing navigator buttons at runtime . . . . .	15-29
Displaying fly-over help . . . . .	15-30
Using a single navigator for multiple datasets . . . . .	15-30

## Chapter 16 Using decision support components 16-1

Overview . . . . .	16-1
About crosstabs . . . . .	16-2
One-dimensional crosstabs. . . . .	16-2
Multidimensional crosstabs . . . . .	16-3
Guidelines for using decision support components . . . . .	16-3
Using datasets with decision support components . . . . .	16-4
Creating decision datasets with TQuery or TTable . . . . .	16-5
Creating decision datasets with the Decision Query editor. . . . .	16-6
Using decision cubes . . . . .	16-7
Decision cube properties and events . . . . .	16-7
Using the Decision Cube editor . . . . .	16-7
Viewing and changing dimension settings . . . . .	16-8
Setting the maximum available dimensions and summaries. . . . .	16-8
Viewing and changing design options . . . . .	16-8
Using decision sources . . . . .	16-9
Properties and events . . . . .	16-9
Using decision pivots. . . . .	16-9
Decision pivot properties. . . . .	16-10
Creating and using decision grids . . . . .	16-10
Creating decision grids . . . . .	16-10
Using decision grids . . . . .	16-11
Opening and closing decision grid fields. . . . .	16-11
Reorganizing rows and columns in decision grids. . . . .	16-11
Drilling down for detail in decision grids. . . . .	16-11
Limiting dimension selection in decision grids. . . . .	16-12
Decision grid properties . . . . .	16-12
Creating and using decision graphs . . . . .	16-13
Creating decision graphs . . . . .	16-13
Using decision graphs . . . . .	16-13

The decision graph display . . . . .	16-15	Opening and closing datasets . . . . .	18-4
Customizing decision graphs . . . . .	16-15	Navigating datasets. . . . .	18-5
Setting decision graph template		Using the First and Last methods . . . . .	18-6
defaults. . . . .	16-16	Using the Next and Prior methods . . . . .	18-6
Customizing decision graph		Using the MoveBy method. . . . .	18-7
series . . . . .	16-17	Using the Eof and Bof properties . . . . .	18-7
Decision support components at		Eof. . . . .	18-7
runtime . . . . .	16-18	Bof. . . . .	18-8
Decision pivots at runtime . . . . .	16-18	Marking and returning to records . . . . .	18-9
Decision grids at runtime . . . . .	16-18	The Bookmark property. . . . .	18-9
Decision graphs at runtime . . . . .	16-19	The GetBookmark method . . . . .	18-9
Decision support components and		The GotoBookmark and Bookmark	
memory control . . . . .	16-19	Valid methods . . . . .	18-9
Setting maximum dimensions,		The CompareBookmarks method . . . . .	18-9
summaries, and cells . . . . .	16-19	The FreeBookmark method. . . . .	18-9
Setting dimension state . . . . .	16-19	A bookmarking example . . . . .	18-10
Using paged dimensions . . . . .	16-20	Searching datasets . . . . .	18-10
		Using Locate . . . . .	18-10
		Using Lookup . . . . .	18-11
		Displaying and editing a subset of data	
		using filters . . . . .	18-12
		Enabling and disabling filtering . . . . .	18-12
		Creating filters. . . . .	18-13
		Setting the Filter property. . . . .	18-13
		Writing an OnFilterRecord event	
		handler . . . . .	18-14
		Switching filter event handlers at	
		runtime . . . . .	18-15
		Setting filter options. . . . .	18-15
		Navigating records in a filtered	
		dataset . . . . .	18-16
		Modifying data . . . . .	18-16
		Editing records. . . . .	18-17
		Adding new records . . . . .	18-18
		Inserting records . . . . .	18-19
		Appending records . . . . .	18-19
		Deleting records . . . . .	18-19
		Posting data . . . . .	18-20
		Canceling changes. . . . .	18-20
		Modifying entire records . . . . .	18-21
		Calculating fields . . . . .	18-22
		Types of datasets . . . . .	18-23
		Using table-type datasets . . . . .	18-24
		Advantages of using table-type	
		datasets . . . . .	18-25
		Sorting records with indexes . . . . .	18-25
		Obtaining information about	
		indexes . . . . .	18-26
<b>Chapter 17</b>			
<b>Connecting to databases</b>	<b>17-1</b>		
Using implicit connections . . . . .	17-2		
Controlling connections. . . . .	17-2		
Connecting to a database server . . . . .	17-3		
Disconnecting from a database server . . . . .	17-3		
Controlling server login. . . . .	17-4		
Managing transactions . . . . .	17-5		
Starting a transaction . . . . .	17-6		
Ending a transaction . . . . .	17-7		
Ending a successful transaction . . . . .	17-8		
Ending an unsuccessful transaction . . . . .	17-8		
Specifying the transaction isolation			
level . . . . .	17-9		
Sending commands to the server . . . . .	17-10		
Working with associated datasets . . . . .	17-11		
Closing all datasets without dis-			
connecting from the server . . . . .	17-12		
Iterating through the associated			
datasets . . . . .	17-12		
Obtaining metadata . . . . .	17-12		
Listing available tables. . . . .	17-13		
Listing the fields in a table . . . . .	17-13		
Listing available stored procedures . . . . .	17-13		
Listing available indexes . . . . .	17-14		
Listing stored procedure parameters. . . . .	17-14		
<b>Chapter 18</b>			
<b>Understanding datasets</b>	<b>18-1</b>		
Using TDataSet descendants . . . . .	18-2		
Determining dataset states . . . . .	18-3		

Specifying an index with IndexName . . . . .	18-26	Setting up parameters at design time . . . . .	18-50
Creating an index with Index FieldNames . . . . .	18-27	Using parameters at runtime . . . . .	18-52
Using Indexes to search for records . . . .	18-27	Preparing stored procedures . . . . .	18-52
Executing a search with Goto methods . . . . .	18-28	Executing stored procedures that don't return a result set . . . . .	18-53
Executing a search with Find methods . . . . .	18-28	Fetching multiple result sets . . . . .	18-53
Specifying the current record after a successful search . . . . .	18-29		
Searching on partial keys . . . . .	18-29	<b>Chapter 19</b>	
Repeating or extending a search . . . .	18-29	<b>Working with field components</b>	<b>19-1</b>
Limiting records with ranges . . . . .	18-30	Dynamic field components . . . . .	19-2
Understanding the differences between ranges and filters . . . . .	18-30	Persistent field components . . . . .	19-3
Specifying Ranges . . . . .	18-30	Creating persistent fields . . . . .	19-4
Modifying a range . . . . .	18-33	Arranging persistent fields . . . . .	19-5
Applying or canceling a range . . . . .	18-33	Defining new persistent fields . . . . .	19-5
Creating master/detail relationships. . . .	18-34	Defining a data field . . . . .	19-6
Making the table a detail of another dataset . . . . .	18-34	Defining a calculated field . . . . .	19-7
Using nested detail tables . . . . .	18-36	Programming a calculated field . . . . .	19-7
Controlling Read/write access to tables . . . . .	18-37	Defining a lookup field . . . . .	19-8
Creating and deleting tables . . . . .	18-37	Defining an aggregate field . . . . .	19-10
Creating tables . . . . .	18-37	Deleting persistent field components . . . .	19-10
Deleting tables . . . . .	18-40	Setting persistent field properties and events . . . . .	19-10
Emptying tables . . . . .	18-40	Setting display and edit properties at design time. . . . .	19-11
Synchronizing tables . . . . .	18-40	Setting field component properties at runtime. . . . .	19-12
Using query-type datasets . . . . .	18-41	Creating attribute sets for field components. . . . .	19-12
Specifying the query . . . . .	18-42	Associating attribute sets with field components. . . . .	19-13
Specifying a query using the SQL property . . . . .	18-42	Removing attribute associations . . . . .	19-14
Specifying a query using the CommandText property . . . . .	18-43	Controlling and masking user input. . . . .	19-14
Using parameters in queries . . . . .	18-43	Using default formatting for numeric, date, and time fields . . . . .	19-14
Supplying parameters at design time. . . . .	18-44	Handling events . . . . .	19-15
Supplying parameters at runtime. . . . .	18-45	Working with field component methods at runtime . . . . .	19-16
Establishing master/detail relationships using parameters . . . . .	18-46	Displaying, converting, and accessing field values. . . . .	19-17
Preparing queries. . . . .	18-47	Displaying field component values in standard controls . . . . .	19-17
Executing queries that don't return a result set . . . . .	18-47	Converting field values. . . . .	19-17
Using unidirectional result sets . . . . .	18-48	Accessing field values with the default dataset property . . . . .	19-19
Using stored procedure-type datasets . . . .	18-48	Accessing field values with a dataset's Fields property. . . . .	19-19
Working with stored procedure parameters. . . . .	18-50		

Accessing field values with a dataset's FieldByName method . . . . .	19-20
Setting a default value for a field. . . . .	19-20
Working with constraints . . . . .	19-21
Creating a custom constraint . . . . .	19-21
Using server constraints. . . . .	19-21
Using object fields . . . . .	19-22
Displaying ADT and array fields. . . . .	19-23
Working with ADT fields . . . . .	19-23
Using persistent field components . . .	19-24
Using the dataset's FieldByName method . . . . .	19-24
Using the dataset's FieldValues property . . . . .	19-24
Using the ADT field's FieldValues property . . . . .	19-24
Using the ADT field's Fields property . . . . .	19-25
Working with array fields. . . . .	19-25
Using persistent fields . . . . .	19-25
Using the array field's FieldValues property . . . . .	19-25
Using the array field's Fields property . . . . .	19-26
Working with dataset fields. . . . .	19-26
Displaying dataset fields. . . . .	19-26
Accessing data in a nested dataset . . .	19-26
Working with reference fields. . . . .	19-27
Displaying reference fields. . . . .	19-27
Accessing data in a reference field . . .	19-27

## Chapter 20 Using the Borland Database Engine

<b>Engine</b>	<b>20-1</b>
BDE-based architecture . . . . .	20-1
Using BDE-enabled datasets . . . . .	20-2
Associating a dataset with database and session connections . . . . .	20-3
Caching BLOBs . . . . .	20-4
Obtaining a BDE handle . . . . .	20-4
Using TTable . . . . .	20-4
Specifying the table type for local tables . . . . .	20-5
Controlling read/write access to local tables . . . . .	20-6
Specifying a dBASE index file. . . . .	20-6
Renaming local tables . . . . .	20-7
Importing data from another table . . .	20-8
Using TQuery. . . . .	20-8

Creating heterogeneous queries . . . . .	20-9
Obtaining an editable result set . . . . .	20-10
Updating read-only result sets . . . . .	20-11
Using TStoredProc . . . . .	20-11
Binding parameters . . . . .	20-12
Working with Oracle overloaded stored procedures . . . . .	20-12
Connecting to databases with	
TDatabase . . . . .	20-12
Associating a database component with a session. . . . .	20-13
Understanding database and session component interactions . . . . .	20-13
Identifying the database . . . . .	20-13
Opening a connection using TDatabase. . . . .	20-15
Using database components in data modules . . . . .	20-16
Managing database sessions . . . . .	20-16
Activating a session . . . . .	20-17
Specifying default database connection behavior . . . . .	20-18
Managing database connections . . . . .	20-19
Working with password-protected Paradox and dBASE tables . . . . .	20-21
Specifying Paradox directory locations. . . . .	20-24
Working with BDE aliases . . . . .	20-24
Retrieving information about a session. . . . .	20-26
Creating additional sessions . . . . .	20-27
Naming a session . . . . .	20-28
Managing multiple sessions . . . . .	20-28
Using transactions with the BDE . . . . .	20-30
Using passthrough SQL . . . . .	20-30
Using local transactions . . . . .	20-31
Using the BDE to cache updates . . . . .	20-32
Enabling BDE-based cached updates . . .	20-33
Applying BDE-based cached updates. . .	20-33
Applying cached updates using a database. . . . .	20-35
Applying cached updates with dataset component methods. . . . .	20-35
Creating an OnUpdateRecord event handler. . . . .	20-36
Handling cached update errors . . . . .	20-37
Using update objects to update a dataset . . . . .	20-39

Creating SQL statements for update components . . . . .	20-40
Using multiple update objects. . . . .	20-43
Executing the SQL statements. . . . .	20-44
Using TBatchMove. . . . .	20-47
Creating a batch move component . . . . .	20-47
Specifying a batch move mode . . . . .	20-49
Appending records . . . . .	20-49
Updating records . . . . .	20-49
Appending and updating records . . . . .	20-49
Copying datasets . . . . .	20-49
Deleting records. . . . .	20-50
Mapping data types . . . . .	20-50
Executing a batch move . . . . .	20-51
Handling batch move errors . . . . .	20-51
The Data Dictionary . . . . .	20-52
Tools for working with the BDE . . . . .	20-53

## Chapter 21

### Working with ADO components 21-1

Overview of ADO components . . . . .	21-1
Connecting to ADO data stores . . . . .	21-2
Connecting to a data store using TADOConnection. . . . .	21-3
Accessing the connection object. . . . .	21-4
Fine-tuning a connection . . . . .	21-4
Forcing asynchronous connections . . . . .	21-5
Controlling timeouts . . . . .	21-5
Indicating the types of operations the connection supports . . . . .	21-6
Specifying whether the connection automatically initiates transactions . . . . .	21-6
Accessing the connection's commands . . . . .	21-7
ADO connection events . . . . .	21-7
Events when establishing a connection . . . . .	21-7
Events when disconnecting . . . . .	21-8
Events when managing transactions . . . . .	21-8
Other events. . . . .	21-8
Using ADO datasets . . . . .	21-9
Connecting an ADO dataset to a data store . . . . .	21-9
Working with record sets . . . . .	21-10
Filtering records based on bookmarks . . . . .	21-10
Fetching records asynchronously . . . . .	21-11
Using batch updates . . . . .	21-12

Loading data from and saving data to files . . . . .	21-14
Using TADODataset . . . . .	21-15
Using Command objects . . . . .	21-16
Specifying the command . . . . .	21-17
Using the Execute method . . . . .	21-17
Canceling commands . . . . .	21-18
Retrieving result sets with commands . . . . .	21-18
Handling command parameters. . . . .	21-19

## Chapter 22

### Using unidirectional datasets 22-1

Types of unidirectional datasets . . . . .	22-2
Connecting to the database server . . . . .	22-2
Setting up TSQLConnection . . . . .	22-3
Identifying the driver . . . . .	22-3
Specifying connection parameters . . . . .	22-4
Naming a connection description . . . . .	22-4
Using the Connection Editor . . . . .	22-5
Specifying what data to display. . . . .	22-5
Representing the results of a query . . . . .	22-6
Representing the records in a table . . . . .	22-6
Representing a table using TSQLDataSet . . . . .	22-6
Representing a table using TSQLTable. . . . .	22-7
Representing the results of a stored procedure. . . . .	22-7
Fetching the data . . . . .	22-8
Preparing the dataset . . . . .	22-8
Fetching multiple datasets . . . . .	22-9
Executing commands that do not return records . . . . .	22-9
Specifying the command to execute. . . . .	22-9
Executing the command . . . . .	22-10
Creating and modifying server metadata. . . . .	22-10
Setting up master/detail linked cursors . . . . .	22-12
Accessing schema information . . . . .	22-12
Fetching metadata into a unidirectional dataset . . . . .	22-12
Fetching data after using the dataset for metadata . . . . .	22-13
The structure of metadata datasets . . . . .	22-13
Debugging dbExpress applications . . . . .	22-17
Using TSQLMonitor to monitor SQL commands . . . . .	22-17
Using a callback to monitor SQL commands . . . . .	22-18

## Chapter 23

### Using client datasets **23-1**

Working with data using a client dataset . . . . .	23-2
Navigating data in client datasets . . . . .	23-2
Limiting what records appear. . . . .	23-2
Editing data. . . . .	23-5
Undoing changes . . . . .	23-5
Saving changes . . . . .	23-6
Constraining data values . . . . .	23-6
Specifying custom constraints. . . . .	23-7
Sorting and indexing. . . . .	23-7
Adding a new index . . . . .	23-8
Deleting and switching indexes. . . . .	23-9
Using indexes to group data. . . . .	23-9
Representing calculated values . . . . .	23-10
Using internally calculated fields in client datasets . . . . .	23-10
Using maintained aggregates . . . . .	23-11
Specifying aggregates . . . . .	23-11
Aggregating over groups of records . . . . .	23-12
Obtaining aggregate values . . . . .	23-13
Copying data from another dataset . . . . .	23-13
Assigning data directly. . . . .	23-13
Cloning a client dataset cursor . . . . .	23-14
Adding application-specific information to the data . . . . .	23-14
Using a client dataset to cache updates . . . . .	23-15
Overview of using cached updates. . . . .	23-16
Choosing the type of dataset for caching updates . . . . .	23-17
Indicating what records are modified . . . . .	23-18
Updating records. . . . .	23-19
Applying updates. . . . .	23-19
Intervening as updates are applied. . . . .	23-20
Reconciling update errors . . . . .	23-22
Using a client dataset with a provider. . . . .	23-23
Specifying a provider . . . . .	23-24
Requesting data from the source dataset or document. . . . .	23-25
Incremental fetching . . . . .	23-25
Fetch-on-demand . . . . .	23-26
Getting parameters from the source dataset . . . . .	23-26
Passing parameters to the source dataset . . . . .	23-27
Sending query or stored procedure parameters. . . . .	23-27
Limiting records with parameters . . . . .	23-28

Handling constraints from the server. . . . .	23-28
Refreshing records. . . . .	23-29
Communicating with providers using custom events . . . . .	23-30
Overriding the source dataset . . . . .	23-31
Using a client dataset with file-based data. . . . .	23-31
Creating a new dataset . . . . .	23-32
Loading data from a file or stream . . . . .	23-32
Merging changes into data . . . . .	23-33
Saving data to a file or stream . . . . .	23-33

## Chapter 24

### Using provider components **24-1**

Determining the source of data . . . . .	24-2
Using a dataset as the source of the data . . . . .	24-2
Using an XML document as the source of the data . . . . .	24-2
Communicating with the client dataset . . . . .	24-3
Choosing how to apply updates using a dataset provider. . . . .	24-4
Controlling what information is included in data packets. . . . .	24-4
Specifying what fields appear in data packets . . . . .	24-4
Setting options that influence the data packets . . . . .	24-5
Adding custom information to data packets . . . . .	24-6
Responding to client data requests . . . . .	24-7
Responding to client update requests . . . . .	24-8
Editing delta packets before updating the database . . . . .	24-9
Influencing how updates are applied. . . . .	24-9
Screening individual updates . . . . .	24-11
Resolving update errors on the provider. . . . .	24-11
Applying updates to datasets that do not represent a single table . . . . .	24-11
Responding to client-generated events. . . . .	24-12
Handling server constraints . . . . .	24-12

## Chapter 25

### Creating multi-tiered applications **25-1**

Advantages of the multi-tiered database model. . . . .	25-2
Understanding provider-based multi-tiered applications . . . . .	25-2
Overview of a three-tiered application . . . . .	25-3

The structure of the client application . . . . .	25-4		
The structure of the application server. . . . .	25-5		
The contents of the remote data module . . . . .	25-6		
Using transactional data modules . . . . .	25-6		
Pooling remote data modules . . . . .	25-8		
Choosing a connection protocol . . . . .	25-8		
Using DCOM connections . . . . .	25-8		
Using Socket connections . . . . .	25-9		
Using Web connections. . . . .	25-9		
Using SOAP connections. . . . .	25-10		
Using CORBA connections . . . . .	25-10		
Building a multi-tiered application . . . . .	25-11		
Creating the application server. . . . .	25-11		
Setting up the remote data module. . . . .	25-13		
Configuring TRemoteData-Module. . . . .	25-13		
Configuring TMTSDataModule. . . . .	25-14		
Configuring TSoapDataModule. . . . .	25-15		
Configuring TCorbaDataModule . . . . .	25-15		
Extending the application server's interface . . . . .	25-16		
Adding callbacks to the application server's interface . . . . .	25-17		
Extending a transactional application server's interface . . . . .	25-18		
Managing transactions in multi-tiered applications . . . . .	25-18		
Supporting master/detail relationships. . . . .	25-19		
Supporting state information in remote data modules . . . . .	25-19		
Using multiple remote data modules . . . . .	25-21		
Registering the application server . . . . .	25-22		
Creating the client application . . . . .	25-23		
Connecting to the application server. . . . .	25-23		
Specifying a connection using DCOM . . . . .	25-24		
Specifying a connection using sockets . . . . .	25-25		
Specifying a connection using HTTP . . . . .	25-26		
Specifying a connection using SOAP . . . . .	25-26		
Specifying a connection using CORBA. . . . .	25-27		
Brokering connections . . . . .	25-27		
Managing server connections. . . . .	25-28		
Connecting to the server. . . . .	25-28		
Dropping or changing a server connection . . . . .	25-28		
Calling server interfaces . . . . .	25-29		
Connecting to an application server that uses multiple data modules . . . . .	25-30		
Writing Web-based client applications . . . . .	25-31		
Distributing a client application as an ActiveX control. . . . .	25-32		
Creating an Active Form for the client application . . . . .	25-33		
Building Web applications using InternetExpress . . . . .	25-33		
Building an InternetExpress application . . . . .	25-34		
Using the javascript libraries . . . . .	25-35		
Granting permission to access and launch the application server. . . . .	25-36		
Using an XML broker . . . . .	25-36		
Fetching XML data packets. . . . .	25-36		
Applying updates from XML delta packets . . . . .	25-37		
Creating Web pages with an InternetExpress page producer . . . . .	25-38		
Using the Web page editor . . . . .	25-39		
Setting Web item properties . . . . .	25-40		
Customizing the InternetExpress page producer template . . . . .	25-41		
<b>Chapter 26</b>			
<b>Using XML in database applications</b>		<b>26-1</b>	
Defining transformations . . . . .	26-1		
Mapping between XML nodes and data packet fields . . . . .	26-2		
Using XMLMapper . . . . .	26-4		
Loading an XML schema or data packet . . . . .	26-4		
Defining mappings . . . . .	26-4		
Generating transformation files . . . . .	26-5		
Converting XML documents into data packets . . . . .	26-6		
Specifying the source XML document . . . . .	26-6		
Specifying the transformation . . . . .	26-7		
Obtaining the resulting data packet. . . . .	26-7		
Converting user-defined nodes . . . . .	26-7		
Using an XML document as the source for a provider . . . . .	26-8		
Using an XML document as the client of a provider . . . . .	26-9		



Fetching an XML document from a provider . . . . .	26-9
Applying updates from an XML document to a provider . . . . .	26-10

## Part III

# Writing Internet applications

---

### Chapter 27

## Creating Internet applications **27-1**

About Web Broker and WebSnap . . . . .	27-1
Terminology and standards. . . . .	27-2
Parts of a Uniform Resource Locator. . . . .	27-3
URI vs. URL . . . . .	27-3
HTTP request header information . . . . .	27-4
HTTP server activity. . . . .	27-4
Composing client requests . . . . .	27-4
Serving client requests . . . . .	27-5
Responding to client requests. . . . .	27-5
Types of Web server applications . . . . .	27-6
ISAPI and NSAPI . . . . .	27-6
Apache . . . . .	27-6
CGI stand-alone. . . . .	27-6
Win-CGI stand-alone. . . . .	27-7
Debugging server applications. . . . .	27-7
Using the Web Application Debugger . . . . .	27-7
Launching your application with the Web Application Debugger . . . . .	27-7
Converting your application to another type of Web server application . . . . .	27-8
Debugging Web applications that are DLLs . . . . .	27-8
Debugging under Windows NT. . . . .	27-9
Debugging under Windows 2000 . . . . .	27-9

### Chapter 28

## Using Web Broker **28-1**

Creating Web server applications with Web Broker . . . . .	28-1
The Web module . . . . .	28-2
The Web Application object. . . . .	28-3
The structure of a Web Broker application . . . . .	28-3
The Web dispatcher . . . . .	28-4
Adding actions to the dispatcher . . . . .	28-4
Dispatching request messages . . . . .	28-5
Action items . . . . .	28-5
Determining when action items fire . . . . .	28-6
The target URL . . . . .	28-6

The request method type . . . . .	28-6
Enabling and disabling action items. . . . .	28-6
Choosing a default action item. . . . .	28-7
Responding to request messages with action items. . . . .	28-7
Sending the response . . . . .	28-8
Using multiple action items . . . . .	28-8
Accessing client request information . . . . .	28-8
Properties that contain request header information. . . . .	28-9
Properties that identify the target . . . . .	28-9
Properties that describe the Web client. . . . .	28-9
Properties that identify the purpose of the request . . . . .	28-9
Properties that describe the expected response. . . . .	28-10
Properties that describe the content. . . . .	28-10
The content of HTTP request messages . . . . .	28-10
Creating HTTP response messages . . . . .	28-10
Filling in the response header . . . . .	28-11
Indicating the response status . . . . .	28-11
Indicating the need for client action . . . . .	28-11
Describing the server application . . . . .	28-12
Describing the content . . . . .	28-12
Setting the response content . . . . .	28-12
Sending the response . . . . .	28-12
Generating the content of response messages . . . . .	28-13
Using page producer components. . . . .	28-13
HTML templates. . . . .	28-13
Specifying the HTML template. . . . .	28-14
Converting HTML-transparent tags . . . . .	28-14
Using page producers from an action item . . . . .	28-15
Chaining page producers together . . . . .	28-16
Using database information in responses. . . . .	28-17
Adding a session to the Web module . . . . .	28-17
Representing database information in HTML . . . . .	28-18
Using dataset page producers . . . . .	28-18
Using table producers. . . . .	28-18
Specifying the table attributes . . . . .	28-18

Specifying the row attributes . . . . .	28-19	Script objects . . . . .	29-11
Specifying the columns. . . . .	28-19	Dispatching requests . . . . .	29-12
Embedding tables in HTML documents . . . . .	28-19	WebContext . . . . .	29-13
Setting up a dataset table producer . . . . .	28-20	Dispatcher components. . . . .	29-13
Setting up a query table producer . . . . .	28-20	Adapter dispatcher operation . . . . .	29-13
<b>Chapter 29</b>		Using adapter components to generate content. . . . .	29-13
<b>Using WebSnap</b>	<b>29-1</b>	Adapter requests and responses . . . . .	29-15
Creating Web server applications with WebSnap . . . . .	29-2	Action request . . . . .	29-15
Server type . . . . .	29-2	Action response . . . . .	29-15
Web application module types . . . . .	29-3	Image request . . . . .	29-16
Web application module options . . . . .	29-3	Image response. . . . .	29-17
Application components . . . . .	29-4	Dispatching action items . . . . .	29-17
Web modules . . . . .	29-5	Page dispatcher operation . . . . .	29-18
Web data modules . . . . .	29-5	WebSnap tutorial . . . . .	29-18
Structure of a Web data module unit . . . . .	29-5	Create a new application . . . . .	29-19
Interfaces implemented by a Web data module . . . . .	29-6	Step 1. Start the WebSnap application wizard . . . . .	29-19
Web page modules . . . . .	29-6	Step 2. Save the generated files and project . . . . .	29-19
Page producer component . . . . .	29-6	Step 3. Specify the application title . . . . .	29-19
Page name . . . . .	29-6	Create a CountryTable page . . . . .	29-20
Producer template . . . . .	29-6	Step 1. Add a new module . . . . .	29-20
Interfaces that the Web page module implements . . . . .	29-7	Step 2. Save the new module . . . . .	29-20
Web application modules . . . . .	29-7	Add data components to the CountryTable module . . . . .	29-20
Interfaces implemented by a Web application data module . . . . .	29-7	Step 1. Add data-aware components . . . . .	29-20
Interfaces implemented by a Web application page module. . . . .	29-7	Step 2. Specify a key field . . . . .	29-21
Adapters . . . . .	29-8	Step 3. Add an adapter component . . . . .	29-21
Fields. . . . .	29-8	Create a grid to display the data. . . . .	29-22
Actions. . . . .	29-8	Step 1. Add a grid . . . . .	29-22
Errors . . . . .	29-8	Step 2. Add editing commands to the grid . . . . .	29-22
Records . . . . .	29-8	Add an edit form . . . . .	29-23
Page producers. . . . .	29-9	Step 1. Add a new module . . . . .	29-23
Templates . . . . .	29-9	Step 2. Save the new module . . . . .	29-23
Server-side scripting in WebSnap . . . . .	29-9	Step 3. Use the CountryTableU unit . . . . .	29-23
Active scripting. . . . .	29-9	Step 4. Add input fields. . . . .	29-23
Script engine . . . . .	29-10	Step 5. Add buttons . . . . .	29-24
Script blocks. . . . .	29-10	Step 6. Link form actions to the grid page . . . . .	29-24
Creating script . . . . .	29-10	Step 7. Link grid actions to the form page . . . . .	29-24
Wizard templates . . . . .	29-10	Add error reporting . . . . .	29-25
TAdapterPageProducer . . . . .	29-10	Step 1. Add error support to the grid . . . . .	29-25
Editing and viewing script . . . . .	29-10		
Including script in a page . . . . .	29-11		

Step 2. Add error support to the form . . . . .	29-25
Step 3. Test the error-reporting mechanism. . . . .	29-26
Run the completed application . . . . .	29-26

## Chapter 30

### **Working with XML documents** **30-1**

Using the Document Object Model . . . . .	30-2
Working with XML components . . . . .	30-3
Using XmlDocument . . . . .	30-3
Working with XML nodes . . . . .	30-4
Working with a node's value . . . . .	30-4
Working with a node's attributes . . . . .	30-5
Adding and deleting child nodes . . . . .	30-5
Abstracting XML documents with the Data Binding wizard . . . . .	30-5
Using the XML Data Binding wizard . . . . .	30-7
Using code that the XML Data Binding wizard generates . . . . .	30-8

## Chapter 31

### **Using Web Services** **31-1**

Writing Servers that support Web Services . . . . .	31-2
Building a Web Service server . . . . .	31-2
Defining invocable interfaces . . . . .	31-3
Using complex types in invocable interfaces . . . . .	31-5
Creating and registering the implementation . . . . .	31-6
Creating custom exception classes for Web Services. . . . .	31-7
Generating WSDL documents for a Web Service application . . . . .	31-7
Writing clients for Web Services . . . . .	31-8
Importing WSDL documents . . . . .	31-8
Calling invocable interfaces. . . . .	31-9

## Chapter 32

### **Working with sockets** **32-1**

Implementing services . . . . .	32-1
Understanding service protocols . . . . .	32-2
Communicating with applications . . . . .	32-2
Services and ports . . . . .	32-2
Types of socket connections. . . . .	32-2
Client connections . . . . .	32-3
Listening connections . . . . .	32-3
Server connections . . . . .	32-3

Describing sockets . . . . .	32-3
Describing the host . . . . .	32-4
Choosing between a host name and an IP address . . . . .	32-4
Using ports. . . . .	32-5
Using socket components . . . . .	32-5
Getting information about the connection . . . . .	32-6
Using client sockets . . . . .	32-6
Specifying the desired server. . . . .	32-6
Forming the connection. . . . .	32-6
Getting information about the connection . . . . .	32-6
Closing the connection . . . . .	32-7
Using server sockets . . . . .	32-7
Specifying the port. . . . .	32-7
Listening for client requests . . . . .	32-7
Connecting to clients . . . . .	32-7
Closing server connections . . . . .	32-7
Responding to socket events. . . . .	32-8
Error events . . . . .	32-8
Client events . . . . .	32-8
Server events. . . . .	32-9
Events when listening. . . . .	32-9
Events with client connections . . . . .	32-9
Reading and writing over socket connections . . . . .	32-9
Non-blocking connections . . . . .	32-9
Reading and writing events . . . . .	32-10
Blocking connections . . . . .	32-10

## Part IV

### **Developing COM-based applications**

## Chapter 33

### **Overview of COM technologies** **33-1**

COM as a specification and implementation . . . . .	33-1
COM extensions . . . . .	33-2
Parts of a COM application . . . . .	33-3
COM interfaces . . . . .	33-3
The fundamental COM interface, IUnknown . . . . .	33-4
COM interface pointers . . . . .	33-4
COM servers. . . . .	33-5
CoClasses and class factories . . . . .	33-6
In-process, out-of-process, and remote servers. . . . .	33-6
The marshaling mechanism . . . . .	33-8

Aggregation . . . . .	33-9
COM clients . . . . .	33-9
COM extensions . . . . .	33-10
Automation servers . . . . .	33-12
Active Server Pages . . . . .	33-12
ActiveX controls . . . . .	33-13
Active Documents . . . . .	33-13
Transactional objects . . . . .	33-14
Type libraries . . . . .	33-15
The content of type libraries . . . . .	33-15
Creating type libraries . . . . .	33-16
When to use type libraries . . . . .	33-16
Accessing type libraries . . . . .	33-16
Benefits of using type libraries . . . . .	33-17
Using type library tools . . . . .	33-18
Implementing COM objects with wizards . . . . .	33-18
Code generated by wizards . . . . .	33-21

## Chapter 34 Working with type libraries **34-1**

Type Library editor . . . . .	34-2
Parts of the Type Library editor . . . . .	34-3
Toolbar . . . . .	34-3
Object list pane . . . . .	34-5
Status bar . . . . .	34-5
Pages of type information . . . . .	34-6
Type library elements . . . . .	34-8
Interfaces . . . . .	34-8
Dispinterfaces . . . . .	34-9
CoClasses . . . . .	34-9
Type definitions . . . . .	34-10
Modules . . . . .	34-10
Using the Type Library editor . . . . .	34-11
Valid types . . . . .	34-11
Using Object Pascal or IDL syntax . . . . .	34-13
Creating a new type library . . . . .	34-19
Opening an existing type library . . . . .	34-19
Adding an interface to the type library . . . . .	34-20
Modifying an interface using the type library . . . . .	34-20
Adding properties and methods to an interface or dispinterface . . . . .	34-21
Adding a CoClass to the type library . . . . .	34-22
Adding an interface to a CoClass . . . . .	34-23

Adding an enumeration to the type library . . . . .	34-23
Adding an alias to the type library . . . . .	34-23
Adding a record or union to the type library . . . . .	34-24
Adding a module to the type library . . . . .	34-24
Saving and registering type library information . . . . .	34-24
Apply Updates dialog . . . . .	34-25
Saving a type library . . . . .	34-25
Refreshing the type library . . . . .	34-26
Registering the type library . . . . .	34-26
Exporting an IDL file . . . . .	34-26
Deploying type libraries . . . . .	34-27

## Chapter 35 Creating COM clients **35-1**

Importing type library information . . . . .	35-2
Using the Import Type Library dialog . . . . .	35-3
Using the Import ActiveX dialog . . . . .	35-4
Code generated when you import type library information . . . . .	35-5
Controlling an imported object . . . . .	35-6
Using component wrappers . . . . .	35-6
ActiveX wrappers . . . . .	35-6
Automation object wrappers . . . . .	35-7
Using data-aware ActiveX controls . . . . .	35-8
Example: Printing a document with Microsoft Word . . . . .	35-9
Step 1: Prepare Delphi for this example . . . . .	35-9
Step 2: Import the Word type library . . . . .	35-10
Step 3: Use a VTable or dispatch interface object to control Microsoft Word . . . . .	35-10
Step 4: Clean up the example . . . . .	35-11
Writing client code based on type library definitions . . . . .	35-12
Connecting to a server . . . . .	35-12
Controlling an Automation server using a dual interface . . . . .	35-12
Controlling an Automation server using a dispatch interface . . . . .	35-13
Handling events in an automation controller . . . . .	35-13
Creating Clients for servers that do not have a type library . . . . .	35-15

## Chapter 36

### Creating simple COM servers **36-1**

Overview of creating a COM object . . . . .	36-2
Designing a COM object . . . . .	36-2
Using the COM object wizard . . . . .	36-2
Using the Automation object wizard . . . . .	36-4
COM object instancing types . . . . .	36-5
Choosing a threading model . . . . .	36-6
Writing an object that supports the free threading model. . . . .	36-7
Writing an object that supports the apartment threading model . . . . .	36-8
Writing an object that supports the neutral threading model . . . . .	36-9
Defining a COM object's interface . . . . .	36-9
Adding a property to the object's interface . . . . .	36-9
Adding a method to the object's interface . . . . .	36-10
Exposing events to clients . . . . .	36-10
Managing events in your Automation object . . . . .	36-12
Automation interfaces . . . . .	36-12
Dual interfaces . . . . .	36-13
Dispatch interfaces . . . . .	36-14
Custom interfaces . . . . .	36-14
Marshaling data . . . . .	36-15
Automation compatible types . . . . .	36-15
Type restrictions for automatic marshaling . . . . .	36-16
Custom marshaling . . . . .	36-16
Registering a COM object. . . . .	36-16
Registering an in-process server . . . . .	36-17
Registering an out-of-process server . . . . .	36-17
Testing and debugging the application . . . . .	36-17

## Chapter 37

### Creating an Active Server Page **37-1**

Creating an Active Server Object. . . . .	37-2
Using the ASP intrinsics. . . . .	37-3
Application . . . . .	37-4
Request. . . . .	37-4
Response. . . . .	37-5
Session . . . . .	37-5
Server . . . . .	37-6
Creating ASPs for in-process or out-of-process servers . . . . .	37-7
Registering an Active Server Object . . . . .	37-7
Registering an in-process server . . . . .	37-7

Registering an out-of-process server . . . . .	37-8
Testing and debugging the Active Server Page application. . . . .	37-8

## Chapter 38

### Creating an ActiveX control **38-1**

Overview of ActiveX control creation . . . . .	38-2
Elements of an ActiveX control . . . . .	38-2
VCL control. . . . .	38-3
ActiveX wrapper. . . . .	38-3
Type library. . . . .	38-3
Property page . . . . .	38-3
Designing an ActiveX control . . . . .	38-4
Generating an ActiveX control from a VCL control . . . . .	38-4
Generating an ActiveX control based on a VCL form. . . . .	38-5
Licensing ActiveX controls. . . . .	38-6
Customizing the ActiveX control's interface . . . . .	38-7
Adding additional properties, methods, and events . . . . .	38-8
Adding properties and methods . . . . .	38-8
Adding events . . . . .	38-9
Enabling simple data binding with the type library. . . . .	38-10
Creating a property page for an ActiveX control . . . . .	38-11
Creating a new property page . . . . .	38-12
Adding controls to a property page. . . . .	38-12
Associating property page controls with ActiveX control properties . . . . .	38-13
Updating the property page . . . . .	38-13
Updating the object . . . . .	38-13
Connecting a property page to an ActiveX control. . . . .	38-14
Registering an ActiveX control . . . . .	38-14
Testing an ActiveX control. . . . .	38-14
Deploying an ActiveX control on the Web . . . . .	38-15
Setting options. . . . .	38-16

## Chapter 39

### Creating MTS or COM+ objects **39-1**

Understanding transactional objects . . . . .	39-2
Requirements for a transactional object . . . . .	39-3
Managing resources . . . . .	39-3
Accessing the object context . . . . .	39-4

Just-in-time activation . . . . .	39-4
Resource pooling . . . . .	39-5
Database resource dispensers . . . . .	39-5
Shared property manager . . . . .	39-6
Releasing resources . . . . .	39-7
Object pooling . . . . .	39-8
MTS and COM+ transaction support . . . . .	39-8
Transaction attributes . . . . .	39-9
Setting the transaction attribute . . . . .	39-10
Stateful and stateless objects . . . . .	39-11
Influencing how transactions end . . . . .	39-11
Initiating transactions . . . . .	39-12
Setting up a transaction object on the client side . . . . .	39-12
Setting up a transaction object on the server side . . . . .	39-13
Transaction timeout . . . . .	39-14
Role-based security . . . . .	39-14
Overview of creating transactional objects . . . . .	39-15
Using the Transactional Object wizard . . . . .	39-15
Choosing a threading model for a transactional object . . . . .	39-16
Activities . . . . .	39-17
Generating events under COM+ . . . . .	39-18
Using the Event Object wizard . . . . .	39-19
Firing events using a COM+ event object . . . . .	39-20
Passing object references . . . . .	39-20
Using the SafeRef method . . . . .	39-20
Callbacks . . . . .	39-21
Debugging and testing transactional objects . . . . .	39-21
Installing transactional objects . . . . .	39-22
Administering transactional objects . . . . .	39-23

## Part V

### Creating custom components

#### Chapter 40

#### **Overview of component creation 40-1**

VCL and CLX . . . . .	40-1
Components and classes . . . . .	40-2
How do you create components? . . . . .	40-2
Modifying existing controls . . . . .	40-3
Creating windowed controls . . . . .	40-3
Creating graphic controls . . . . .	40-4
Subclassing Windows controls . . . . .	40-4

Creating nonvisual components . . . . .	40-5
What goes into a component? . . . . .	40-5
Removing dependencies . . . . .	40-5
Properties, methods, and events . . . . .	40-6
Properties . . . . .	40-6
Events . . . . .	40-6
Methods . . . . .	40-6
Graphics encapsulation . . . . .	40-7
Registration . . . . .	40-8
Creating a new component . . . . .	40-8
Using the Component wizard . . . . .	40-9
Creating a component manually . . . . .	40-11
Creating a unit file . . . . .	40-11
Deriving the component . . . . .	40-11
Registering the component . . . . .	40-12
Testing uninstalled components . . . . .	40-12
Testing installed components . . . . .	40-14

#### Chapter 41

#### **Object-oriented programming for component writers 41-1**

Defining new classes . . . . .	41-1
Deriving new classes . . . . .	41-2
To change class defaults to avoid repetition . . . . .	41-2
To add new capabilities to a class . . . . .	41-2
Declaring a new component class . . . . .	41-3
Ancestors, descendants, and class hierarchies . . . . .	41-3
Controlling access . . . . .	41-4
Hiding implementation details . . . . .	41-4
Defining the component writer's interface . . . . .	41-5
Defining the runtime interface . . . . .	41-6
Defining the design-time interface . . . . .	41-6
Dispatching methods . . . . .	41-7
Static methods . . . . .	41-7
An example of static methods . . . . .	41-7
Virtual methods . . . . .	41-8
Overriding methods . . . . .	41-8
Abstract class members . . . . .	41-9
Classes and pointers . . . . .	41-9

#### Chapter 42

#### **Creating properties 42-1**

Why create properties? . . . . .	42-1
Types of properties . . . . .	42-2
Publishing inherited properties . . . . .	42-2

Defining properties . . . . .	42-3
The property declaration . . . . .	42-3
Internal data storage . . . . .	42-4
Direct access. . . . .	42-4
Access methods. . . . .	42-5
The read method . . . . .	42-6
The write method. . . . .	42-6
Default property values . . . . .	42-7
Specifying no default value . . . . .	42-7
Creating array properties . . . . .	42-8
Creating properties for subcomponents. . . . .	42-8
Creating properties for interfaces . . . . .	42-10
Storing and loading properties. . . . .	42-11
Using the store-and-load mechanism . . . . .	42-11
Specifying default values . . . . .	42-11
Determining what to store. . . . .	42-12
Initializing after loading. . . . .	42-13
Storing and loading unpublished properties . . . . .	42-13
Creating methods to store and load property values . . . . .	42-14
Overriding the DefineProperties method. . . . .	42-14

## Chapter 43 Creating events 43-1

What are events? . . . . .	43-1
Events are method pointers . . . . .	43-2
Events are properties. . . . .	43-2
Event types are method-pointer types. . . . .	43-3
Event-handler types are procedures . . . . .	43-3
Event handlers are optional. . . . .	43-4
Implementing the standard events. . . . .	43-4
Identifying standard events. . . . .	43-4
Standard events for all controls . . . . .	43-4
Standard events for standard controls. . . . .	43-5
Making events visible . . . . .	43-5
Changing the standard event handling . . . . .	43-6
Defining your own events . . . . .	43-6
Triggering the event . . . . .	43-7
Two kinds of events . . . . .	43-7
Defining the handler type. . . . .	43-7
Simple notifications. . . . .	43-7
Event-specific handlers. . . . .	43-7
Returning information from the handler. . . . .	43-8

Declaring the event . . . . .	43-8
Event names start with "On". . . . .	43-8
Calling the event. . . . .	43-8

## Chapter 44 Creating methods 44-1

Avoiding dependencies . . . . .	44-1
Naming methods . . . . .	44-2
Protecting methods . . . . .	44-2
Methods that should be public. . . . .	44-3
Methods that should be protected. . . . .	44-3
Abstract methods . . . . .	44-3
Making methods virtual . . . . .	44-4
Declaring methods . . . . .	44-4

## Chapter 45 Using graphics in components 45-1

Overview of graphics. . . . .	45-1
Using the canvas . . . . .	45-3
Working with pictures . . . . .	45-3
Using a picture, graphic, or canvas . . . . .	45-3
Loading and storing graphics . . . . .	45-4
Handling palettes . . . . .	45-5
Specifying a palette for a control. . . . .	45-5
Off-screen bitmaps . . . . .	45-6
Creating and managing off-screen bitmaps . . . . .	45-6
Copying bitmapped images . . . . .	45-6
Responding to changes. . . . .	45-7

## Chapter 46 Handling messages 46-1

Understanding the message-handling system . . . . .	46-1
What's in a Windows message? . . . . .	46-2
Dispatching messages. . . . .	46-2
Tracing the flow of messages . . . . .	46-3
Changing message handling. . . . .	46-3
Overriding the handler method . . . . .	46-3
Using message parameters. . . . .	46-4
Trapping messages . . . . .	46-4
Creating new message handlers. . . . .	46-5
Defining your own messages . . . . .	46-5
Declaring a message identifier . . . . .	46-6
Declaring a message-record type. . . . .	46-6
Declaring a new message-handling method . . . . .	46-7

<b>Chapter 47</b>	
<b>Making components available at design time</b>	<b>47-1</b>
Registering components . . . . .	47-1
Declaring the Register procedure . . . . .	47-2
Writing the Register procedure . . . . .	47-2
Specifying the components . . . . .	47-2
Specifying the palette page . . . . .	47-3
Using the RegisterComponents function . . . . .	47-3
Adding palette bitmaps . . . . .	47-3
Providing Help for your component . . . . .	47-4
Creating the Help file . . . . .	47-4
Creating the entries . . . . .	47-4
Making component help context-sensitive . . . . .	47-6
Adding component help files . . . . .	47-6
Adding property editors . . . . .	47-6
Deriving a property-editor class . . . . .	47-7
Editing the property as text . . . . .	47-8
Displaying the property value . . . . .	47-8
Setting the property value . . . . .	47-8
Editing the property as a whole . . . . .	47-9
Specifying editor attributes . . . . .	47-10
Registering the property editor . . . . .	47-11
Property categories . . . . .	47-12
Registering one property at a time . . . . .	47-13
Registering multiple properties at once . . . . .	47-13
Specifying property categories . . . . .	47-14
Using the IsPropertyInCategory function . . . . .	47-15
Adding component editors . . . . .	47-15
Adding items to the context menu . . . . .	47-16
Specifying menu items . . . . .	47-16
Implementing commands . . . . .	47-16
Changing the double-click behavior . . . . .	47-17
Adding clipboard formats . . . . .	47-18
Registering the component editor . . . . .	47-18
Compiling components into packages . . . . .	47-19

<b>Chapter 48</b>	
<b>Modifying an existing component</b>	<b>48-1</b>
Creating and registering the component . . . . .	48-1
Modifying the component class . . . . .	48-2
Overriding the constructor . . . . .	48-2
Specifying the new default property value . . . . .	48-3

<b>Chapter 49</b>	
<b>Creating a graphic component</b>	<b>49-1</b>
Creating and registering the component . . . . .	49-1
Publishing inherited properties . . . . .	49-2
Adding graphic capabilities . . . . .	49-3
Determining what to draw . . . . .	49-3
Declaring the property type . . . . .	49-3
Declaring the property . . . . .	49-4
Writing the implementation method . . . . .	49-4
Overriding the constructor and destructor . . . . .	49-4
Changing default property values . . . . .	49-4
Publishing the pen and brush . . . . .	49-5
Declaring the class fields . . . . .	49-5
Declaring the access properties . . . . .	49-6
Initializing owned classes . . . . .	49-6
Setting owned classes' properties . . . . .	49-7
Drawing the component image . . . . .	49-8
Refining the shape drawing . . . . .	49-9

<b>Chapter 50</b>	
<b>Customizing a grid</b>	<b>50-1</b>
Creating and registering the component . . . . .	50-1
Publishing inherited properties . . . . .	50-2
Changing initial values . . . . .	50-3
Resizing the cells . . . . .	50-4
Filling in the cells . . . . .	50-5
Tracking the date . . . . .	50-5
Storing the internal date . . . . .	50-6
Accessing the day, month, and year . . . . .	50-6
Generating the day numbers . . . . .	50-8
Selecting the current day . . . . .	50-9
Navigating months and years . . . . .	50-10
Navigating days . . . . .	50-11
Moving the selection . . . . .	50-11
Providing an OnChange event . . . . .	50-11
Excluding blank cells . . . . .	50-12

<b>Chapter 51</b>	
<b>Making a control data aware</b>	<b>51-1</b>
Creating a data-browsing control . . . . .	51-1
Creating and registering the component . . . . .	51-2
Making the control read-only . . . . .	51-2
Adding the ReadOnly property . . . . .	51-3
Allowing needed updates . . . . .	51-3
Adding the data link . . . . .	51-4
Declaring the class field . . . . .	51-4



Declaring the access properties . . . . .	.51-5
An example of declaring access properties . . . . .	.51-5
Initializing the data link . . . . .	.51-6
Responding to data changes . . . . .	.51-6
Creating a data-editing control. . . . .	.51-7
Changing the default value of FReadOnly . . . . .	.51-8
Handling mouse-down and key-down messages. . . . .	.51-8
Responding to mouse-down messages . . . . .	.51-8
Responding to key-down messages . . .	.51-9
Updating the field datalink class . . . . .	.51-10

Modifying the Change method . . . . .	.51-11
Updating the dataset . . . . .	.51-11

## Chapter 52

### **Making a dialog box a component 52-1**

Defining the component interface. . . . .	52-1
Creating and registering the component . . . .	52-2
Creating the component interface. . . . .	52-3
Including the form unit. . . . .	52-3
Adding interface properties . . . . .	52-3
Adding the Execute method . . . . .	52-4
Testing the component . . . . .	52-6

## Index

**I-1**

# Tables

1.1	Typefaces and symbols . . . . .	1-3	8.5	Multimedia device types and their functions . . . . .	8-32
3.1	Important base classes . . . . .	3-13	9.1	Thread priorities . . . . .	9-3
3.2	Graphic controls . . . . .	3-17	9.2	WaitFor return values . . . . .	9-10
3.3	Component palette pages . . . . .	3-29	10.1	Porting techniques . . . . .	10-3
3.4	Text control properties . . . . .	3-31	10.2	CLX parts . . . . .	10-6
3.5	Components for creating and managing lists . . . . .	3-47	10.3	Changed or different features . . . . .	10-9
4.1	RTL exceptions . . . . .	4-9	10.4	VCL and equivalent CLX units . . . . .	10-10
4.2	Object Pascal character types . . . . .	4-40	10.5	Units in CLX, not VCL . . . . .	10-11
4.3	String comparison routines . . . . .	4-44	10.6	VCL-only units . . . . .	10-11
4.4	Case conversion routines . . . . .	4-44	10.7	Differences in the Linux and Windows operating environments . . . . .	10-14
4.5	String modification routines . . . . .	4-45	10.8	Common Linux directories . . . . .	10-16
4.6	Sub-string routines . . . . .	4-45	10.9	TWidgetControl protected methods for responding to system events . . . . .	10-21
4.7	String handling routines . . . . .	4-45	10.10	Comparable data-access components . . . . .	10-24
4.8	Compiler directives for strings . . . . .	4-49	10.11	Properties, methods, and events for cached updates . . . . .	10-28
4.9	Attribute constants and values . . . . .	4-51	11.1	Compiled package files . . . . .	11-2
4.10	File types for file I/O . . . . .	4-53	11.2	Package-specific compiler directives . . . . .	11-10
4.11	Open modes . . . . .	4-54	11.3	Package-specific command-line compiler switches . . . . .	11-12
4.12	Shared modes . . . . .	4-54	11.4	Compiled package files . . . . .	11-12
4.13	Shared modes available for each open mode . . . . .	4-55	12.1	VCL objects that support BiDi . . . . .	12-4
5.1	Compiler directives for libraries . . . . .	5-9	12.2	Estimating string lengths . . . . .	12-9
5.2	Database pages on the Component palette . . . . .	5-10	13.1	Application files . . . . .	13-3
5.3	Web server applications . . . . .	5-12	13.2	Merge modules and their dependencies . . . . .	13-4
5.4	Context menu options for data modules . . . . .	5-16	13.3	dbExpress deployment as standalone executable . . . . .	13-7
5.5	Help methods in TApplication . . . . .	5-28	13.4	dbExpress deployment, driver DLLs . . . . .	13-7
6.1	Action setup terminology . . . . .	6-16	13.5	SQL database client software files . . . . .	13-9
6.2	Default values of the action manager's PrioritySchedule property . . . . .	6-22	15.1	Data controls . . . . .	15-2
6.3	Action classes . . . . .	6-27	15.2	Column properties . . . . .	15-19
6.4	Sample captions and their derived names . . . . .	6-31	15.3	Expanded TColumn Title properties . . . . .	15-20
6.5	Menu Designer context menu commands . . . . .	6-36	15.4	Properties that affect the way composite fields appear . . . . .	15-23
6.6	Setting speed buttons' appearance . . . . .	6-44	15.5	Expanded TDBGrid Options properties . . . . .	15-24
6.7	Setting tool buttons' appearance . . . . .	6-46	15.6	Grid control events . . . . .	15-26
6.8	Setting a cool button's appearance . . . . .	6-47	15.7	Selected database control grid properties . . . . .	15-27
7.1	Properties of selected text . . . . .	7-8	15.8	TDBNavigator buttons . . . . .	15-28
7.2	Fixed vs. variable owner-draw styles . . . . .	7-12	17.1	Database connection components . . . . .	17-1
8.1	Graphic object types . . . . .	8-3			
8.2	Common properties of the Canvas object . . . . .	8-4			
8.3	Common methods of the Canvas object . . . . .	8-4			
8.4	Mouse-event parameters . . . . .	8-24			

18.1	Values for the dataset State property . . .	18-3	22.5	Columns in tables of metadata listing parameters. . . . .	22-16
18.2	Navigational methods of datasets . . . . .	18-5	23.1	Filter support in client datasets . . . . .	23-3
18.3	Navigational properties of datasets . . . . .	18-6	23.2	Summary operators for maintained aggregates . . . . .	23-11
18.4	Comparison and logical operators that can appear in a filter . . . . .	18-14	23.3	Specialized client datasets for caching updates. . . . .	23-17
18.5	FilterOptions values. . . . .	18-15	24.1	AppServer interface members. . . . .	24-3
18.6	Filtered dataset navigational methods. . . . .	18-16	24.2	Provider options . . . . .	24-5
18.7	Dataset methods for inserting, updating, and deleting data . . . . .	18-17	24.3	UpdateStatus values . . . . .	24-9
18.8	Methods that work with entire records . . . . .	18-21	24.4	UpdateMode values . . . . .	24-10
18.9	Index-based search methods . . . . .	18-27	24.5	ProviderFlags values . . . . .	24-10
19.1	TFloatField properties that affect data display . . . . .	19-1	25.1	Components used in multi-tiered applications . . . . .	25-3
19.2	Special persistent field kinds . . . . .	19-6	25.2	Connection components . . . . .	25-5
19.3	Field component properties . . . . .	19-11	25.3	Javascript libraries . . . . .	25-35
19.4	Field component formatting routines . . . . .	19-15	27.1	Web Broker versus WebSnap . . . . .	27-2
19.5	Field component events. . . . .	19-15	27.2	Web server application components . . . . .	27-6
19.6	Selected field component methods . . . . .	19-16	28.1	MethodType values. . . . .	28-6
19.7	Special conversion results. . . . .	19-19	33.1	COM object requirements . . . . .	33-11
19.8	Types of object field components . . . . .	19-22	33.2	Delphi wizards for implementing COM, Automation, and ActiveX objects. . . . .	33-19
19.9	Common object field descendant properties. . . . .	19-23	33.3	DAX Base classes for generated implementation classes . . . . .	33-21
20.1	Table types recognized by the BDE based on file extension . . . . .	20-5	34.1	Type Library editor files . . . . .	34-2
20.2	TableType values. . . . .	20-5	34.2	Type Library editor parts . . . . .	34-3
20.3	BatchMove import modes . . . . .	20-8	34.3	Type library pages . . . . .	34-6
20.4	Database-related informational methods for session components . . . . .	20-26	34.4	Valid types. . . . .	34-12
20.5	TSessionList properties and methods. . . . .	20-29	34.5	Attribute syntax. . . . .	34-14
20.6	Properties, methods, and events for cached updates. . . . .	20-32	36.1	Threading models for COM objects . . . . .	36-6
20.7	UpdateKind values . . . . .	20-38	37.1	IApplicationObject interface members. . . . .	37-4
20.8	Batch move modes. . . . .	20-49	37.2	IRequest interface members . . . . .	37-4
20.9	Data Dictionary interface . . . . .	20-52	37.3	IResponse interface members . . . . .	37-5
21.1	ADO components . . . . .	21-2	37.4	ISessionObject interface members . . . . .	37-6
21.2	ADO connection modes . . . . .	21-6	37.5	IServer interface members . . . . .	37-6
21.3	Execution options for ADO datasets . . . . .	21-12	39.1	IObjectContext methods for transaction support. . . . .	39-11
21.4	Comparison of ADO and client dataset cached updates. . . . .	21-12	39.2	Threading models for transactional objects . . . . .	39-17
22.1	Columns in tables of metadata listing tables . . . . .	22-14	39.3	Call synchronization options . . . . .	39-18
22.2	Columns in tables of metadata listing stored procedures . . . . .	22-14	40.1	Component creation starting points . . . . .	40-3
22.3	Columns in tables of metadata listing fields . . . . .	22-15	41.1	Levels of visibility within an object. . . . .	41-4
22.4	Columns in tables of metadata listing indexes . . . . .	22-16	42.1	How properties appear in the Object Inspector. . . . .	42-2
			45.1	Canvas capability summary . . . . .	45-3
			45.2	Image-copying methods . . . . .	45-7
			47.1	Predefined property-editor types . . . . .	47-7
			47.2	Methods for reading and writing property values . . . . .	47-8
			47.3	Property-editor attribute flags. . . . .	47-10
			47.4	Property categories . . . . .	47-14

# Figures

3.1	A simple form . . . . .	3-6	15.5	TDBCtrlGrid at design time . . . . .	15-27
3.2	Objects, components, and controls. . . . .	3-12	15.6	Buttons on the TDBNavigator control . . . . .	15-28
3.3	A simplified hierarchy diagram . . . . .	3-13	16.1	Decision support components at design time . . . . .	16-2
3.4	Three views of the track bar component . . . . .	3-33	16.2	One-dimensional crosstab . . . . .	16-3
3.5	A progress bar . . . . .	3-42	16.3	Three-dimensional crosstab . . . . .	16-3
6.1	A frame with data-aware controls and a data source component . . . . .	6-15	16.4	Decision graphs bound to different decision sources. . . . .	16-14
6.2	The Action Manager editor. . . . .	6-20	20.1	Components in a BDE-based application. . . . .	20-2
6.3	Menu terminology. . . . .	6-29	25.1	Web-based multi-tiered database application. . . . .	25-31
6.4	MainMenu and PopupMenu components . . . . .	6-30	27.1	Parts of a Uniform Resource Locator. . . . .	27-3
6.5	Menu Designer for a pop-up menu . . . . .	6-31	28.1	Structure of a Server Application . . . . .	28-3
6.6	Menu Designer for a main menu . . . . .	6-31	29.1	Generating Content Flow . . . . .	29-14
6.7	Nested menu structures. . . . .	6-34	29.2	Action request and response . . . . .	29-16
8.1	Bitmap-dimension dialog box from the BMPDlg unit. . . . .	8-20	29.3	Image response to a request . . . . .	29-17
12.1	TListBox set to bdLeftToRight . . . . .	12-7	29.4	Dispatching a page . . . . .	29-18
12.2	TListBox set to bdRightToLeft . . . . .	12-7	33.1	A COM interface . . . . .	33-3
12.3	TListBox set to bdRightToLeft-NoAlign. . . . .	12-7	33.2	Interface vtable . . . . .	33-5
12.4	TListBox set to bdRightToLeft-ReadingOnly . . . . .	12-7	33.3	In-process server . . . . .	33-7
14.1	Generic Database Architecture . . . . .	14-6	33.4	Out-of-process and remote servers . . . . .	33-8
14.2	Connecting directly to the database server . . . . .	14-8	33.5	COM-based technologies . . . . .	33-11
14.3	A file-based database application . . . . .	14-9	33.6	Simple COM object interface . . . . .	33-18
14.4	Architecture combining a client dataset and another dataset . . . . .	14-12	33.7	Automation object interface . . . . .	33-19
14.5	Multi-tiered database architecture . . . . .	14-13	33.8	ActiveX object interface . . . . .	33-19
15.1	TDBGrid control . . . . .	15-15	33.9	Delphi ActiveX framework . . . . .	33-21
15.2	TDBGrid control with ObjectView set to False . . . . .	15-22	34.1	Type Library editor . . . . .	34-3
15.3	TDBGrid control with Expanded set to False . . . . .	15-23	34.2	Object list pane . . . . .	34-5
15.4	TDBGrid control with Expanded set to True. . . . .	15-23	36.1	Dual interface VTable . . . . .	36-13
			38.1	Mask Edit property page in design mode . . . . .	38-12
			40.1	Visual Component Library class hierarchy. . . . .	40-2
			40.2	Component wizard . . . . .	40-9



# Introduction

The *Developer's Guide* describes intermediate and advanced development topics, such as building client/server database applications, writing custom components, and creating Internet Web server applications. It allows you to build applications that meet many industry-standard specifications such as SOAP, TCP/IP, COM+, and ActiveX. Many of the advanced features that support Web development, advanced XML technologies, and database development require components or wizards that are not available in all versions of Delphi.

The *Developer's Guide* assumes you are familiar with using Delphi and understand fundamental Delphi programming techniques. For an introduction to Delphi programming and the integrated development environment (IDE), see the Quick Start manual or the online Help.

## What's in this manual?

---

This manual contains five parts, as follows:

- **Part I, "Programming with Delphi,"** describes how to build general-purpose Delphi applications. This part provides details on programming techniques you can use in any Delphi application. For example, it describes how to use common Visual Component Library (VCL) or Component Library for Cross-platform (CLX) objects that make user interface programming easy. Objects are available for handling strings, manipulating text, implementing common dialogs, and so on. This section also includes chapters on working with graphics, error and exception handling, using DLLs, OLE automation, and writing international applications.

A chapter describes how to use objects in the Borland Component Library for Cross-Platform (CLX) to develop applications that can be compiled and run on either Windows or Linux platforms.

The chapter on deployment details the tasks involved in deploying your application to your application users. For example, it includes information on effective compiler options, using InstallShield Express, licensing issues, and how to determine which packages, DLLs, and other libraries to use when building the production-quality version of your application.

- **Part II, “Developing database applications,”** describes how to build database applications using database tools and components. Delphi lets you access many types of databases, including local databases such as Paradox and dBASE, and network SQL server databases like InterBase, Oracle, and Sybase. You can choose from a variety of data access mechanisms, including dbExpress, the Borland Database Engine, InterbaseExpress, and ADO. To implement the more advanced database applications, you need the Delphi features that are not available in all versions.
- **Part III, “Writing Internet applications,”** describes how to create applications that are distributed over the internet. Delphi includes a wide array of tools for writing Web server applications, including the Web Broker architecture, which lets you create cross-platform server applications, WebSnap, which lets you design Web pages in a GUI environment, support for working with XML documents, and an architecture for using SOAP-based Web Services. For lower-level support that underlies much of the messaging in Internet applications, this section also describes how to work with socket components. The components that implement many of these features are not available in all versions of Delphi.
- **Part IV, “Developing COM-based applications,”** describes how to build applications that can interoperate with other COM-based API objects on the system such as Windows Shell extensions or multimedia applications. Delphi contains components that support the ActiveX, COM+, and a COM-based library for COM controls that can be used for general-purpose and Web-based applications. Support for COM controls is not available in all editions of Delphi. To create ActiveX controls, you need the Professional or Enterprise edition.
- **Part V, “Creating custom components,”** describes how to design and implement your own components, and how to make them available on the Component palette of the IDE. A component can be almost any program element that you want to manipulate at design time. Implementing custom components entails deriving a new class from an existing class type in the VCL or CLX class libraries.

# Manual conventions

---

This manual uses the typefaces and symbols described in Table 1.1 to indicate special text.

**Table 1.1** Typefaces and symbols

Typeface or symbol	Meaning
Monospace type	Monospaced text represents text as it appears on screen or in Object Pascal code. It also represents anything you must type.
[ ]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
<b>Boldface</b>	Boldfaced words in text or code listings represent Object Pascal keywords or compiler options.
<i>Italics</i>	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
<i>Keycaps</i>	This typeface indicates a key on your keyboard. For example, "Press <i>Esc</i> to exit a menu."

---

## Developer support services

---

Inprise also offers a variety of support options to meet the needs of its diverse developer community. To find out about support offerings for Delphi, refer to <http://www.borland.com/devsupport/delphi>.

Additional Delphi Technical Information documents and answers to Frequently Asked Questions (FAQs) are also available at this Web site.

From the Web site, you can access many newsgroups where Delphi developers exchange information, tips, and techniques. The site also includes a list of books about Delphi.

## Ordering printed documentation

---

For information about ordering additional documentation, refer to the Web site at [shop.borland.com](http://shop.borland.com).





# Programming with Delphi

The chapters in “Programming with Delphi” introduce concepts and skills necessary for creating Delphi applications using any edition of the product. They also introduce the concepts discussed in later sections of the *Developer’s Guide*.



# Developing applications with Delphi

Borland Delphi is an object-oriented, visual programming environment for rapid development of 32-bit applications for deployment on Windows and Linux. Using Delphi, you can create highly efficient applications with a minimum of manual coding.

Delphi provides a comprehensive class library called the *Visual Component Library* (VCL), Borland Component Library for Cross Platform (CLX), and a suite of Rapid Application Development (RAD) design tools, including application and form templates, and programming wizards. Delphi supports truly object-oriented programming:

- the VCL class library includes objects that encapsulate the Windows API as well as other useful programming techniques (Windows)
- the CLX class library includes objects that encapsulate the Qt library (Windows or Linux)

This chapter briefly describes the Delphi development environment and how it fits into the development life cycle. The rest of this manual provides technical details on developing general-purpose, database, Internet and Intranet applications, and includes information on creating ActiveX and COM controls and writing your own components.

## Integrated development environment

---

When you start Delphi, you are immediately placed within the integrated development environment, also called the IDE. This environment provides all the tools you need to design, develop, test, debug, and deploy applications.

Delphi's development environment includes a visual form designer, Object Inspector, Object TreeView, Component palette, Project Manager, source code editor, and debugger among other tools. Some tools may not be included in all versions of the product. You can move freely from the visual representation of an object (in the

form designer), to the Object Inspector to edit the initial runtime state of the object, to the source code editor to edit the execution logic of the object. Changing code-related properties, such as the name of an event handler, in the Object Inspector automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the Object Inspector.

The IDE supports application development throughout the stages of the product life cycle—from design to deployment. Using the tools in the IDE allows for rapid prototyping and shortens development time.

A more complete overview of the development environment is presented in the *Quick Start* manual included with the product. In addition, the online Help system provides help on all menus, dialogs, and windows.

## Designing applications

---

Delphi includes all the tools necessary to start designing applications:

- A blank window, known as a *form*, on which to design the UI for your application.
- Extensive class libraries with many reusable objects.
- An Object Inspector for examining and changing object traits.
- A Code editor that provides direct access to the underlying program logic.
- A Project Manager for managing the files that make up one or more projects.
- Many other tools such as an image editor on the toolbar and an integrated debugger on menus to support application development in the IDE.
- Command-line tools including compilers, linkers, and other utilities.

You can use Delphi to design any kind of 32-bit application—from general-purpose utilities to sophisticated data access programs or distributed applications. Delphi's database tools and data-aware components let you quickly develop powerful desktop database and client/server applications. Using Delphi's data-aware controls, you can view live data while you design your application and immediately see the results of database queries and changes to the application interface.

Chapter 5, “Building applications, components, and libraries” introduces Delphi's support for different types of applications.

Many of the objects provided in the class library are accessible in the IDE from the Component palette. The Component palette shows all of the controls, both visual and nonvisual, that you can place on a form. Each tab contains components grouped by functionality. By convention, the names of objects in the class library begin with a T, such as *TStatusBar*.

One of the revolutionary things about Delphi is that you can create your own components using Object Pascal. Most of the components provided are written in Object Pascal. You can add components that you write to the Component palette and customize the palette for your use by including new tabs if needed.

You can also use Delphi for cross platform development on both Linux and Windows by using CLX. CLX contains a set of classes that, if used instead of those in the VCL, allow your program to port between Windows and Linux.

# Developing applications

---

As you visually design the user interface for your application, Delphi generates the underlying Object Pascal code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment as well.

## Creating projects

---

All of Delphi's application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Delphi.

At the top of the project hierarchy, is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the Project Manager. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the Project Manager, the project file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of a Delphi application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Delphi automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

## Editing code

---

The Delphi Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the Object Inspector. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changed and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor and continue adjusting the form from there.

The Delphi code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the VCL objects, CLX objects, RTL sources, all of the Delphi project files can be viewed and edited in the Code editor.

## Compiling applications

---

When you have finished designing your application interface on the form, writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether or not they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to use Build when you've changed global compiler directives, to ensure that all code compiles in the proper state. You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application.

If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose Project | Compile All Projects or Project | Build All Projects with the project group selected in the Project Manager.

## Debugging applications

---

Delphi provides an integrated debugger that helps you find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging.

The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the values of variables, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed. The debugger is described in online Help.

You can also use exception handling to recognize, locate, and deal with errors. Exceptions in Delphi are classes, like other classes in Delphi, except, by convention, they begin with an E rather than the initial T for other classes.

## Deploying applications

---

Delphi includes add-on tools to help with application deployment. For example, InstallShield Express (not available in all versions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. Refer to Chapter 13, “Deploying applications” for specific information on deployment.

**Note** Not all versions of Delphi have deployment capabilities.

TeamSource software (not available in all versions) is also available for tracking application updates.





# Using the component libraries

This chapter presents an overview of the component libraries and introduces some of the components that you can use while developing applications. Delphi includes both the Visual Component Library (VCL) and the Borland Component Library for Cross-Platform (CLX). The VCL is for Windows development and CLX is for cross-platform development on Windows and Linux. They are two different class libraries but they have many similarities. Objects, properties, methods, and events that are not in CLX are marked “VCL only.”

## Understanding the component libraries

---

VCL and CLX are class libraries made up of objects, some of which are also components or controls, that you use when developing applications. Both libraries look very similar and contain many of the same objects. Some objects in the VCL implement features that are available on Windows only such as objects that appear on the ADO, BDE, QReport, COM+, Web Services, and Servers tabs on the Component palette. Virtually all CLX objects are available on both Windows and Linux.

VCL and CLX objects are active entities that contain all necessary data and the “methods” (code) that modify the data. The data is stored in the fields and properties of the objects, and the code is made up of methods that act upon the field and property values. Each object is declared as a “class.” All VCL and CLX objects descend from the ancestor object *TObject* including objects that you develop in Object Pascal.

A subset of objects are components. Components are objects that you can place on a form or data module and manipulate at design time. Components appear on the Component palette. You can specify their properties without writing code. All VCL or CLX components descend from the *TComponent* object.

Components are objects in the true object-oriented programming (OOP) sense because they

- Encapsulate a set of data and data-access functions
- Inherit data and behavior from the objects they are derived from
- Operate interchangeably with other objects derived from a common ancestor, through a concept called *polymorphism*

Unlike most components, objects do not appear on the Component palette. Instead, a default instance variable is declared in the unit of the object, or you have to declare one yourself.

Controls are a special kind of component that is visible to users at runtime. Controls are a subset of components. Controls are visual components that you can see when your application is running. All controls have properties in common that specify their visual attributes, such as *Height* and *Width*. The properties, methods, and events that all controls have in common are inherited from *TControl*.

Refer to Chapter 10, “Using CLX for cross-platform development” for details about cross-platform programming and the differences between the Windows and Linux environments. Detailed reference material on all of the objects in the VCL or CLX is accessible using online Help while you are programming. From within the code editor, place the cursor anywhere on the object and press F1 to display help on VCL or CLX components.

If you are using Kylix while developing cross-platform applications, Kylix also includes a *Developer’s Guide* that is tailored for the Linux environment. You can refer to the manual both in the Kylix online Help or the printed manual provided with the Kylix product.

## Properties, methods, and events

---

Both the VCL and CLX form hierarchies of objects that are tied to the Delphi IDE, where you can develop applications quickly. The objects in both component libraries are based on properties, methods, and events. Each object includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). The VCL is written in Object Pascal, whereas CLX is based on Qt, a C++ class library.

### Properties

*Properties* are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *Visible* property determines whether an object can be seen or not in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at runtime, you can see and change properties at design time and get immediate feedback as the components change in the IDE.

- Properties can be accessed in the Object Inspector where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.
- The actual calls to get and set the values are methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.
- You can implement logic that triggers events or modifies other data during the access of the property. For example, changing the value of one property may require the modification of another. You can make the change in the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing a one property on one object could effect several objects. For example, setting the *Checked* property on a radio button effects all of the radio buttons in the group.

## Methods

A *method* is a procedure that is always associated with a class. Methods define the behavior of an object. Class methods can access all the *public*, *protected*, and *private* properties and data members of the class and are commonly referred to as member functions.

## Events

An *event* is an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform next. They may choose a menu item, click a button, or mark some text. You can write code to handle the events you're interested in, rather than writing code that always executes in the same restricted order.

Regardless of how an event is called, Delphi looks to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

The kinds of events that can occur can be divided into two main categories:

- User events
- System events

Regardless of how the event was called, Delphi looks to see if you have assigned any code to handle that event. If you have, then that code is executed; otherwise, nothing is done.

## User events

User events are actions that are initiated by the user. Examples of user events are *OnClick* (the user clicked the mouse), *OnKeyPress* (the user pressed a key on the keyboard), and *OnDblClick* (the user double-clicked a mouse button). These events are always tied to a user's actions.

## System events

System events are events that the operating system fires for you. For example, the *OnTimer* event (the *Timer* component issues one of these events whenever a predefined interval has elapsed), the *OnCreate* event (the component is being created), the *OnPaint* event (a component or window needs to be redrawn), and so on. Usually, system events are not directly initiated by a user action.

## Object Pascal and the class libraries

---

Object Pascal, a set of object-oriented extensions to standard Pascal, is the language of Delphi. Using Delphi's Component palette and Object Inspector, you can place VCL or CLX components on forms and manipulate their properties without writing code.

All objects descend from *TObject*, an abstract class whose methods encapsulate fundamental behavior like construction, destruction, and message handling. *TObject* is the immediate ancestor of many simple classes.

*Components* in the VCL or CLX descend from the abstract class *TComponent*. Components are objects that you can manipulate on forms at design time. Visual components—that is, components like *TForm* and *TSpeedButton* that appear on the screen at runtime—are called *controls*, and they descend from *TControl*.

In addition to the visual components, the component libraries contain many nonvisual objects. The IDE allows you to add many nonvisual components to your programs by dropping them onto forms. For example, if you were writing an application that connects to a database, you might place a *TDataSource* component on a form. Although *TDataSource* is nonvisual, it is represented on the form by an icon (which doesn't appear at runtime). You can manipulate the properties and events of *TDataSource* in the Object Inspector just as you would those of a visual control.

When you write classes of your own in Object Pascal, they should descend from *TObject* in the class library that you plan to use. Use VCL if you're writing a Windows application or CLX if writing a cross-platform application. By deriving new classes from the appropriate base class (or one of its descendants), you provide your classes with essential functionality and ensure that they work with the other classes in the class library.

## Using the object model

---

Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you create an object (or, more formally, a class), you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

If you want to create new components and put them on the Component palette, see Chapter 40, "Overview of component creation."

## What is an object?

An object, or *class*, is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements.

You can begin to understand objects if you understand Object Pascal *records* or *structures* in C. Records are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of VCL and CLX objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

## Examining a Delphi object

When you create a new project, Delphi displays a new form for you to customize. In the Code editor, Delphi declares a new class type for the form and produces the code that creates the new form instance. The code generated for a new Windows application looks like this:

```

unit Unit1;
interface

uses Windows, Classes, Graphics, Forms, Controls, Dialogs;

type
  TForm1 = class(TForm){ The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end;{ The type declaration of the form ends here }

var
  Form1: TForm1;

implementation{ Beginning of implementation part }
{$R *.DFM}
end.{ End of implementation part and unit}

```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no

fields or methods, because you haven't added to the form any components (the fields of the new object) and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

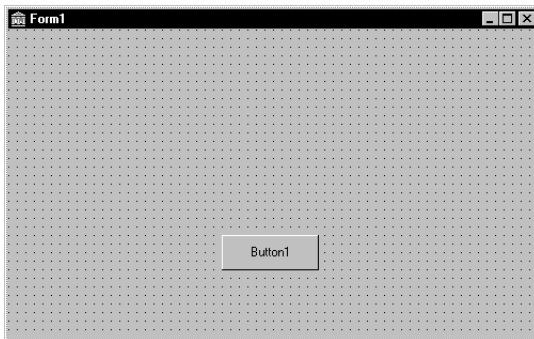
```
var
  Form1: TForm1;
```

*Form1* represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete Delphi application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:

**Figure 3.1** A simple form



When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls;
```

```

type
  TForm1 = class(TForm)
    Button1: TButton; { New data field }
    procedure Button1Click(Sender: TObject); { New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
  {$R *.DFM}

  procedure TForm1.Button1Click(Sender: TObject); { The code of the new method }
  begin
    Form1.Color := clGreen;
  end;

end.

```

*TForm1* has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write in Delphi are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared within the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the **implementation** part of the unit.

## Changing the name of a component

You should always use the Object Inspector to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorBox*. When you change the form's *Name* property in the Object Inspector, the new name is automatically reflected in the form's *.dfm* or *.xfm* file (which you usually don't edit manually) and in the Object Pascal source code that Delphi generates:

```

unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls;

type
  TColorBox = class(TForm) { Changed from TForm1 to TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var

```



```
ColorBox: TColorBox; { Changed from Form1 to ColorBox }  
  
implementation  
{ $R *.DFM }  
  
procedure TColorBox.Button1Click(Sender: TObject);  
begin  
    Form1.Color := clGreen; { The reference to Form1 didn't change! }  
end;  
  
end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorBox.Button1Click(Sender: TObject);  
begin  
    ColorBox.Color := clGreen;  
end;
```

## Inheriting data and code from an object

---

The *TForm1* object described seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the component *TForm*. When you add a new form to your project, you start with *TForm* and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, Delphi automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived object inherits all the properties, events, and methods of the object it derives from. The derived object is called a *descendant* and the object it derives from is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. An object can have only one immediate ancestor, but it can have many direct descendants.

## Scope and qualifiers

---

*Scope* determines the accessibility of an object's fields, properties, and methods. All members declared within an object are available to that object and its descendants. Although a method's implementation code appears outside of the object declaration, the method is still within the scope of the object because it is declared within the object's declaration.

When you write code to implement a method that refers to properties, methods, or fields of the object where the method is declared, you don't need to preface those identifiers with the name of the object. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Color := clFuchsia;
    Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

Delphi creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the **uses** clause of *Form1*'s unit.

The scope of an object extends to the object's descendants. You can, however, redeclare a field, property, or method within a descendant object. Such redeclarations either hide or override the inherited member.

For more information about scope, inheritance, and the **uses** clause, see the *Object Pascal Language Guide*.

## Private, protected, public, and published declarations

When you declare a field, property, or method, the new member has a *visibility* indicated by one of the keywords **private**, **protected**, **public**, or **published**. The visibility of a member determines its accessibility to other objects and units.

- A private member is accessible only within the unit where it is declared. Private members are often used within a class to implement other (public or published) methods and properties.
- A protected member is accessible within the unit where its class is declared and within any descendant class, regardless of the descendant class's unit.
- A public member is accessible from wherever the object it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.

- A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the Object Inspector at design time.

For more information about visibility, see the *Object Pascal Language Guide*.

## Using object variables

---

You can assign one object variable to another object variable if the variables are of the same type or assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable you are assigning to is an ancestor of the type of the variable being assigned. For example, here is a *TDataForm* type declaration (VCL only) and a variable declaration section declaring two variables, *AForm* and *DataForm*:

```

type
  TDataForm = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Databasel: TDatabase;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AForm: TForm;
  DataForm: TDataForm;

```

*AForm* is of type *TForm*, and *DataForm* is of type *TDataForm*. Because *TDataForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := DataForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  :
end;

```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word **is**. For example,

```

if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;

```

## Creating, instantiating, and destroying objects

---

Many of the objects you use in Delphi, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and datasource components, have no visual representation at runtime.

You may want to create your own objects. For example, you could create a *TEmployee* object that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the **interface** or **implementation** part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically by Delphi. But if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a *destructor*) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a **nil** reference before calling *Destroy*. For example,

```
Employee.Free
```

destroys the *Employee* object and deallocates its memory.

## Components and ownership

---

Delphi has a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's

memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

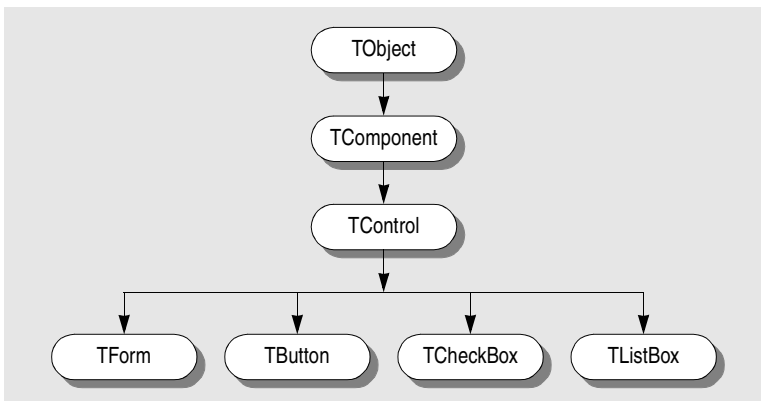
**Note** Don't confuse a component's *owner* with its *parent*. See "Parent properties" on page 3-19".

## Objects, components, and controls

---

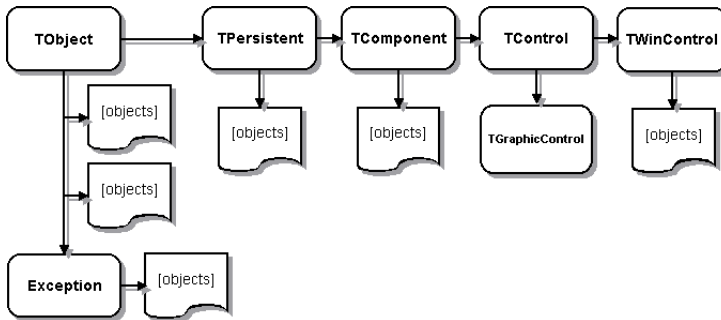
Figure 3.2 is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.

**Figure 3.2** Objects, components, and controls



Every object inherits from *TObject*, and many objects inherit from *TComponent*. Controls, which inherit from *TControl*, have the ability to display themselves at runtime. A control like *TCheckBox* inherits all the functionality of *TObject*, *TComponent*, and *TControl*, and adds specialized capabilities of its own.

Figure 3.3 is an overview of the Visual Component Library (VCL) that shows the major branches of the inheritance tree. The Borland Component Library for Cross-Platform (CLX) look very much the same at this level but *TWinControl* is replaced by *TWidgetControl*.

**Figure 3.3** A simplified hierarchy diagram

Several important base classes are shown in the figure, and they are described in the following table:

**Table 3.1** Important base classes

Class	Description
<i>TObject</i>	Signifies the base class and ultimate ancestor of everything in the VCL or CLX. <i>TObject</i> encapsulates the fundamental behavior common to all VCL/CLX objects by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.
<i>Exception</i>	Specifies the base class of all classes that relate to exceptions. <i>Exception</i> provides a consistent interface for error conditions, and enables applications to handle error conditions gracefully.
<i>TPersistent</i>	Specifies the base class for all objects that implement properties. Classes under <i>TPersistent</i> deal with sending data to streams and allow for the assignment of classes.
<i>TComponent</i>	Specifies the base class for all nonvisual components such as <i>TApplication</i> . <i>TComponent</i> is the common ancestor of all components. This class allows a component to be displayed on the Component palette, lets the component own other components, and allows the component to be manipulated directly on a form.
<i>TControl</i>	Represents the base class for all controls that are visible at runtime. <i>TControl</i> is the common ancestor of all visual components and provides standard visual controls like position and cursor. This class also provides events that respond to mouse actions.
<i>TWinControl</i>	Specifies the base class of all user interface objects also called widgets. Controls under <i>TWinControl</i> are windowed controls that can capture keyboard input. (In CLX, <i>TWidgetControl</i> replaces <i>TWinControl</i> .)

The next few sections present a general description of the types of classes that each branch contains. For a complete overview of the VCL object hierarchy, refer to the VCL Object Hierarchy wall chart that is included with this product. For details on CLX, refer to the CLX Object Hierarchy wall chart included with the product and the Kylix documentation.

## TObject branch

---

The *TObject* branch includes all objects that descend from *TObject* but not from *TPersistent*. All VCL or CLX objects descend from *TObject*, an abstract class whose methods define fundamental behavior like construction, destruction, and message or system event handling. Much of the powerful capability of VCL and CLX objects are established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all objects in the VCL and CLX by introducing methods that provide:

- The ability to respond when object instances are created or destroyed.
- Class type and instance information on an object, and runtime type information (RTTI) about its published properties.
- Support for message-handling (VCL only).

*TObject* is the immediate ancestor of many simple classes. Classes that are contained within this branch have one common, important characteristic: they are transitory. What this means is that these classes do not have a method to save the state that they are in prior to destruction; they are not persistent.

One of the main groups of classes in this branch is the *Exception* class. This class provides a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions.

Another type of group in the *TObject* branch are classes that encapsulate data structures, such as:

- *TBits*, a class that stores an “array” of Boolean values
- *TList*, a linked list class
- *TStack*, a class that maintains a last-in first-out array of pointers
- *TQueue*, a class that maintains a first-in first-out array of pointers

In the VCL, you can also find wrappers for external objects like *TPrinter*, which encapsulates the Windows printer interface, and *TRegistry*, a low-level wrapper for the system registry and functions that operate on the registry. These are specific to the Windows environment.

*TStream* is good example of another type of class in this branch. *TStream* is the base class type for stream objects that can read from or write to various kinds of storage media, such as disk files, dynamic memory, and so on.

So you can see, this branch includes many different types of classes that are very useful to you as a developer.

## TPersistent branch

---

Objects in this branch of the VCL and CLX descend from *TPersistent* but not from *TComponent*. *TPersistent* adds persistence to objects. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory.

Objects in this branch implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. This branch introduces the *GetOwner* function which lets you determine the owner of the property.

Objects in this branch are also the first to include a published section where properties can be automatically loaded and saved. A *DefineProperties* method also allows you to indicate how to load and save properties.

Following are some of the other classes in the *TPersistent* branch of the hierarchy:

- *TGraphicsObject*, an abstract base class for graphics objects such as: *TBrush*, *TFont*, and *TPen*.
- *TGraphic*, an abstract base class for objects such as icons and bitmaps that can store and display visual images: *TBitmap* and *TIcon* (and for Windows development only: *TMetafile*).
- *TStrings*, a base class for objects that represent a list of strings.
- *TClipboard*, a class that contains text or graphics that have been cut or copied from an application.
- *TCollection*, *TOwnedCollection*, and *TCollectionItem*, classes that maintain indexed collections of specially defined items.

## TComponent branch

---

*TComponent* branch contains objects that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time. They are persistent objects that can do the following:

- Appear on the Component palette and can be changed in the form designer.
- Own and manage other components.
- Load and save themselves.

Several methods in *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming is introduced in this branch of the VCL and CLX. Delphi handles most streaming chores automatically. Properties are persistent if they are published and published properties are automatically streamed.

The *TComponent* class also introduces the concept of ownership that is propagated throughout the VCL and CLX. Two properties support ownership: *Owner* and *Components*. Every component has an *Owner* property that references another component as its owner. A component may own other components. In this case, all owned components are referenced in the component's *Array* property.

A component's constructor takes a single parameter that is used to specify the new component's owner. If the passed-in owner exists, the new component is added to the owner's *Components* list. Aside from using the *Components* list to reference owned components, this property also provides for the automatic destruction of owned components. As long as the component has an owner, it will be destroyed when the owner is destroyed. For example, since *TForm* is a descendant of *TComponent*, all components owned by the form are destroyed and their memory



freed when the form is destroyed. This assumes that all of the components on the form clean themselves up properly when their destructors are called.

If a property type is a *TComponent* or a descendant, the streaming system creates an instance of that type when reading it in. If a property type is *TPersistent* but not *TComponent*, the streaming system uses the existing instance available through the property and read values for that instance's properties.

When creating a form file (a file used to store information about the components on the form), the form designer loops through its components array and saves all the components on the form. Each component "knows" how to write its changed properties out to a stream (in this case, a text file). Conversely, when loading the properties of components in the form file, the form designer loops through the components array and loads each component.

The types of classes you'll find in this branch include:

- *TMainMenu*, a class that provides a menu bar and its accompanying drop-down menus for a form.
- *TTimer*, a class that includes the timer functions.
- *TOpenDialog*, *TSaveDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog*, and so on, provide commonly used dialog boxes.
- *TActionList*, a class that maintains a list of actions used with components and controls, such as menu items and buttons.
- *TScreen*, a class that keeps track of what forms and data modules have been instantiated by the application, the active form, and the active control within that form, the size and resolution of the screen, and the cursors and fonts available for the application to use.

Components that do not need a visual interface can be derived directly from *TComponent*. To make a tool such as a *TTimer* device, you can derive from *TComponent*. This type of component resides on the Component palette but performs internal functions that are accessed through code rather than appearing in the user interface at runtime.

In CLX, the *TComponent* branch also includes *THandleComponent*. This is the base class for nonvisual components that require a handle to an underlying Qt object such as dialogs and menus.

## TControl branch

---

The *TControl* branch consists of components that descend from *TControl* but not *TWinControl* (*TWidgetControl* in CLX). Objects in this branch are controls that are visual objects which the application user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position, the cursor associated with the control's window (or widget in CLX), methods to paint or move the control, and events to respond to mouse actions. Controls can never receive keyboard input.

Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls. This includes drawing routines, standard events, and containership.

There are two basic types of control:

- Those that have a window (or widget) of their own
- Those that use the window (or widget) of their “parent”

Controls that have their own window are called “windowed” controls (VCL) or “widget-based” controls (CLX) and descend from *TWinControl* (*TWidgetControl* in CLX). Buttons and check boxes fall into this class.

Controls that use a parent window (or widget) are called “graphic” controls and descend from *TGraphicControl*. Image and label controls fall into this class. In the VCL, the main difference between these types of components is that graphic controls do not maintain a window handle, and thus cannot receive the input focus. In CLX, the main difference between these types of components is that graphic controls do not have an associated widget, and thus cannot receive the input focus nor can they contain other controls. Because a graphic control does not need a handle, its demand on system resources is lessened, and painting a graphic control is quicker than painting a widget-based control.

*TGraphicControl* controls must draw themselves and include controls such as:

**Table 3.2** Graphic controls

Control	Description
<i>TImage</i>	Displays graphical images.
<i>TLabel</i>	Displays text on a form.
<i>TBevel</i>	Represents a beveled outline.
<i>TPaintBox</i>	Provides a canvas that applications can use for drawing or rendering an image.

Notice that these include common paint routines (*Repaint*, *Invalidate*, and so on) that never need to receive focus.

## TWinControl/TWidgetControl branch

The *TWinControl* branch (*TWidgetControl* replaces *TWinControl* in CLX) includes all controls that descend from *TWinControl*. *TWinControl* is the base class for all windowed controls, including many of the items that you will use in the user interface of an application.

*TWidgetControl* is the base class for all widget-based controls or *widgets*. The term widget comes from combining “window” and “gadget.” A widget is almost anything you use in the user interface of an application. Examples of widgets are buttons, labels, and scroll bars.

The following are features of windowed and widget-based controls:

- Both can receive focus while an application is running.
- Other controls may display data, but the user can use the keyboard to interact with windowed or widget-based controls.
- Windowed or widget-based controls can contain other controls.

- A control that contains other controls is called a parent. Only a windowed or widget-based control can be a parent of one or more child controls.
- Windowed controls have a window handle. Widget-based controls have an associated widget.

Descendants of *TWinControl* (*TWidgetControl* in CLX) are controls that can receive focus, meaning they can receive keyboard input from the application user. This implies that many more standard events apply to them.

This branch includes both controls that are drawn automatically (including *TEdit*, *TListBox*, *TComboBox*, *TPageControl*, and so on) and custom controls that Delphi must draw (such as *TDBNavigator*, *TMediaPlayer* (VCL only), *TGauge* (VCL only), and so on). Direct descendants of *TWinControl* (*TWidgetControl* in CLX) typically implement standard controls, like an edit field, a combo box, list box, or page control, and, therefore, already know how to paint themselves.

The *TCustomControl* class is provided for components that require a window handle but do not encapsulate a standard control that includes the ability to repaint itself. You never have to worry about how the controls render themselves or how they respond to events—Delphi completely encapsulates this behavior for you.

The following sections provide an overview of controls. Refer to Chapter 7, “Working with controls” for more information on using controls.

## Properties common to TControl

---

All visual controls (descendants of *TControl*) share certain properties including:

- Action properties
- Position, size, and alignment properties
- Display properties
- Parent properties
- A navigation property
- Drag-and-drop properties
- Drag-and-dock properties (VCL only)

While these properties are inherited from *TControl*, they are published—and hence appear in the Object Inspector—only for components to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays.

### Action properties

Actions let you share common code for performing actions (for example, when a tool bar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

- *Action* designates the action associated with the control.
- *ActionLink* contains the action link object associated with the control.

## Position, size, and alignment properties

This set of properties defines the position and size of a control on the parent control:

- *Height* sets the vertical size.
- *Width* sets the horizontal size.
- *Top* positions the top edge.
- *Left* positions the left edge.
- *AutoSize* specifies whether the control sizes itself automatically to accommodate its contents.
- *Align* determines how the control aligns within its container (parent control).
- *Anchor* specifies how the control is anchored to its parent (VCL only).

This set of properties determine the height, width, and overall size of the control's client area:

- *ClientHeight* specifies the height of the control's client area in pixels.
- *ClientWidth* specifies the width of the control's client area in pixels.

These properties aren't accessible in nonvisual components, but Delphi does keep track of where you place the component icons on your forms. Most of the time you'll set and alter these properties by manipulating the control's image on the form or using the Alignment palette. You can, however, alter them at runtime.

## Display properties

The following properties govern the general appearance of a control:

- *Color* changes the background color of a control.
- *Font* changes the color, type family, style, or size of text.
- *Cursor* specifies the image used to represent the mouse pointer when it passes into the region covered by the control.
- *DesktopFont* specifies whether the control uses the Windows icon font when writing text (VCL only).

## Parent properties

To maintain a consistent appearance across your application, you can make any control look like its container—called its *parent*—by setting the parent properties to *True*.

- *ParentColor* determines where a control looks for its color information.
- *ParentFont* determines where a control looks for its font information.
- *ParentShowHint* determines where a control looks to find out if its Help Hint should be shown.

## A navigation property

The following property determines how users navigate among the controls in a form:

- *Caption* contains the text string that labels a component. To underline a character in a string, include an ampersand (&) before the character. This type of character is called an accelerator key. The user can then select the control or menu item by pressing *Alt* while typing the underlined character.

## Drag-and-drop properties

Two component properties affect drag-and-drop behavior:

- *DragMode* determines how dragging starts. By default, *DragMode* is *dmManual*, and the application must call the *BeginDrag* method to start dragging. When *DragMode* is *dmAutomatic*, dragging starts as soon as the mouse button goes down.
- *DragCursor* determines the shape of the mouse pointer when it is over a draggable component (VCL only).

## Drag-and-dock properties (VCL only)

The following properties control drag-and-dock behavior:

- *Floating* indicates whether the control is floating.
- *DragKind* specifies whether the control is being dragged normally or for docking.
- *DragMode* determines how the control initiates drag-and-drop or drag-and-dock operations.
- *FloatingDockSiteClass* specifies the class of the temporary control that hosts the control when it is floating.
- *DragCursor* is the cursor that is shown while dragging.
- *DockOrientation* specifies how the control is docked relative to other controls docked in the same parent.
- *HostDockSite* specifies the control in which the control is docked.

For more information, see “Implementing drag-and-dock in controls” on page 7-4.

## Standard events common to TControl

---

The VCL defines a set of standard events for its controls. The following events are declared as part of the *TControl* class, and are therefore available for all classes derived from *TControl*:

- *OnClick* occurs when the user clicks the control.
- *OnContextPopup* occurs when the user right-clicks the control or otherwise invokes the popup menu (such as using the keyboard).
- *OnCanResize* occurs when an attempt is made to resize the control.
- *OnResize* occurs immediately after the control is resized.
- *OnConstrainedResize* occurs immediately after *OnCanResize*.
- *OnStartDock* occurs when the user begins to drag a control with a *DragKind* of *dkDock* (VCL only).
- *OnEndDock* occurs when the dragging of an object ends, either by docking the object or by canceling the dragging (VCL only).
- *OnStartDrag* occurs when the user begins to drag the control or an object it contains by left-clicking on the control and holding the mouse button down.

- *OnEndDrag* occurs when the dragging of an object ends, either by dropping the object or by canceling the dragging.
- *OnDragDrop* occurs when the user drops an object being dragged.
- *OnMouseMove* occurs when the user moves the mouse pointer while the mouse pointer is over a control.
- *OnDbClick* occurs when the user double-clicks the primary mouse button when the mouse pointer is over the control.
- *OnDragOver* occurs when the user drags an object over a control (VCL only).
- *OnMouseDown* occurs when the user presses a mouse button with the mouse pointer over a control.
- *OnMouseUp* occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

## Properties common to TWinControl and TWidgetControl

---

All windowed controls (descendants of *TWinControl* in the VCL and *TWidgetControl* in CLX) share certain properties including:

- Information about the control
- Border style display properties
- Navigation properties
- Drag-and-dock properties (VCL only)

While these properties are inherited from *TWinControl* and *TWidgetControl*, they are published—and hence appear in the Object Inspector—only for controls to which they are applicable.

### General information properties

The general information properties contain information about the appearance of the *TWinControl* and *TWidgetControl*, client area size and origin, windows assigned information, and help context information.

- *ClientOrigin* specifies the screen coordinates (in pixels) of the top left corner of a control's client area. The screen coordinates of a control that is descended from *TControl* and not *TWinControl* are the screen coordinates of the control's parent added to its *Left* and *Top* properties.
- *ClientRect* returns a rectangle with its *Top* and *Left* properties set to zero, and its *Bottom* and *Right* properties set to the control's *Height* and *Width*, respectively. *ClientRect* is equivalent to `Rect(0, 0, ClientWidth, ClientHeight)`.
- *Brush* determines the color and pattern used for painting the background of the control.
- *HelpContext* provides a context number for use in calling context-sensitive online Help.
- *Handle* provides access to the window or widget handle of the control.

## Border style display properties

The bevel properties control the appearance of the beveled lines, boxes, or frames on the forms and windowed controls in your application.

Many more objects in the VCL publish these properties; they are not all available in CLX and the border style properties are published on fewer objects.

- *InnerBevel* specifies whether the inner bevel has a raised, lowered, or flat look (VCL only).
- *BevelKind* specifies the type of bevel if the control has beveled edges (VCL only).
- *BevelOuter* specifies whether the outer bevel has a raised, lowered, or flat look.
- *BevelWidth* specifies the width, in pixels, of the inner and outer bevels.
- *BorderWidth* is used to get or set the width of the control's border.
- *BevelEdges* is used to get or set which edges of the control are beveled.

## Navigation properties

Two additional properties determine how users navigate among the controls on a form:

- *TabOrder* indicates the position of the control in its parent's tab order, the order in which controls receive focus when the user presses the *Tab* key. Initially, tab order is the order in which the components are added to the form, but you can change this by changing *TabOrder*. *TabOrder* is meaningful only if *TabStop* is *True*.
- *TabStop* determines whether the user can tab to a control. If *TabStop* is *True*, the control is in the tab order.

## Drag-and-dock properties (VCL only)

The following properties manage drag-and-dock behavior in VCL objects:

- *UseDockManager* specifies whether the dock manager is used in drag-and-dock operations.
- *VisibleDockClientCount* specifies the number of visible controls that are docked on the windowed control.
- *DockManager* specifies the control's dock manager interface.
- *DockClients* lists the controls that are docked to the windowed control.
- *DockSite* specifies whether the control can be the target of drag-and-dock operations.

For more information, see “Implementing drag-and-dock in controls” on page 7-4.

## Events common to TWinControl and TWidgetControl

---

The following events exist for all controls derived from *TWinControl* in the VCL (this also includes all the controls that Windows defines) and in *TWidgetControl* in CLX. These events are in addition to those that exist in all controls.

- *OnEnter* occurs when the control is about to receive focus.
- *OnKeyDown* occurs on the down stroke of a key press.
- *OnKeyPress* occurs when a user presses a single character key.
- *OnKeyUp* occurs when the user releases a key that has been pressed.

- *OnExit* occurs when the input focus shifts away from one control to another.
- *OnMouseWheel* occurs when the mouse wheel is rotated.
- *OnMouseWheelDown* occurs when the mouse wheel is rotated downward.
- *OnMouseWheelUp* occurs when the mouse wheel is rotated upward.

The following events relate to docking and are available in the VCL only:

- *OnUnDock* occurs when the application tries to undock a control that is docked to a windowed control (VCL only).
- *OnDockDrop* occurs when another control is docked to the control (VCL only).
- *OnDockOver* occurs when another control is dragged over the control (VCL only).
- *OnGetSiteInfo* returns the control's docking information (VCL only).

## Creating the application user interface

---

All visual design work in Delphi takes place on forms. When you open Delphi or create a new project, a blank form is displayed on the screen. You can use it to start building your application interface including windows, menus, and common dialogs.

You design the look and feel of the graphical user interface for an application by placing and arranging visual components such as buttons and list boxes on the form. Delphi takes care of the underlying programming details. You can also place invisible components on forms to capture information from databases, perform calculations, and manage other interactions.

Chapter 6, "Developing the application user interface" provides details on using forms such as creating modal forms dynamically, passing parameters to forms, and retrieving data from forms.

## Using Delphi components

---

Many visual components are provided in the development environment itself on the Component palette. All visual design work in Delphi takes place on forms. When you open Kylix or create a new project, a blank form is displayed on the screen. You select components from the Component palette and drop them onto the form. You design the look and feel of the application's user interface by arranging the visual components such as buttons and list boxes on the form. Once a visual component is on the form, you can adjust its position, size, and other design-time properties. Delphi takes care of the underlying programming details.

Delphi components are grouped functionally on different pages of the Component palette. For example, commonly used components such as those to create menus, edit boxes, or buttons are located on the Standard page of the Component palette. Handy VCL controls such as a timer, paint box, media player, and OLE container are on the System page.



At first glance, Delphi's components appear to be just like any other classes. But there are differences between components in Delphi and the standard class hierarchies that many programmers work with. Some differences are described here:

- All Delphi components descend from *TComponent*.
- Components are most often used as is and are changed through their properties, rather than serving as "base classes" to be subclassed to add or change functionality. When a component is inherited, it is usually to add specific code to existing event handling member functions.
- Components can only be allocated on the heap, not on the stack.
- Properties of components intrinsically contain runtime type information.
- Components can be added to the Component palette in the Delphi user interface and manipulated on a form.

Components often achieve a better degree of encapsulation than is usually found in standard classes. For example, consider the use of a dialog containing a push button. In a Windows program developed using VCL components, when a user clicks on the button, the system generates a `WM_LBUTTONDOWN` message. The program must catch this message (typically in a **switch** statement, a message map, or a response table) and dispatch it to a routine that will execute in response to the message.

Most Windows messages (VCL) or system events (CLX) are handled by Delphi components. When you want to respond to a message, you only need to provide an event handler.

## Setting component properties

---

Published properties can be set at design time in the Object Inspector and, in some cases, with special property editors.

To set properties at runtime, assign them new values in your application source code.

For information about the properties of each component, see the online Help.

## Using the Object Inspector

When you select a component on a form, the Object Inspector displays its published properties and (when appropriate) allows you to edit them. Use the *Tab* key to toggle between the Value column and the Property column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column.

If a plus (+) symbol appears next to a property name, clicking the plus symbol or typing '+' when the property has focus displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol or typing '-' hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the Object Inspector and choose View. For more information, see “property categories” in the online Help.

When more than one component is selected, the Object Inspector displays all properties—except *Name*—that are shared by the selected components. If the value for a shared property differs among the selected components, the Object Inspector displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

## Using property editors

Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the Object Inspector. To open the property editor, double-click in the Value column, click the ellipsis mark, or type *Ctrl+Enter* when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

## Setting properties at runtime

Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

```
Form1.Caption := MyString;
```

## Calling methods

---

Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control’s image on the screen. You could call the *Repaint* method in a draw-grid object like this:

```
DrawGrid1.Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for example, to repaint a form within an event handler of one of the form’s child controls, you don’t have to prepend the name of the form to the method call:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
```

For more information about scope, see “Scope and qualifiers” on page 3-8.

## Working with events and event handlers

---

In Delphi, almost all the code you write is executed, directly or indirectly, in response to *events*. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is an Object Pascal procedure. The sections that follow show how to

- Generate a new event handler
- Generate a handler for a component's default event
- Locate event handlers
- Associate an event with an existing event handler
- Associate menu events with event handlers
- Delete event handlers

### Generating a new event handler

Delphi can generate skeleton event handlers for forms and other components. To create an event handler,

- 1 Select a component.
- 2 Click the Events tab in the Object Inspector. The Events page of the Object Inspector displays all events defined for the component.
- 3 Select the event you want, then double-click the Value column or press *Ctrl+Enter*. Delphi generates the event handler in the Code editor and places the cursor inside the **begin...end** block.
- 4 Inside the **begin...end** block, type the code that you want to execute when the event occurs.

### Generating a handler for a component's default event

Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as *TBevel*, don't respond to any events. Other components respond differently when you double-click on them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

### Locating event handlers

If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that's not the default,

- 1 In the form, select the component whose event handler you want to locate.
- 2 In the Object Inspector, click the Events tab.
- 3 Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor at the beginning of the event-handler body.

## Associating an event with an existing event handler

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler,

- 1 On the form, select the component whose event you want to handle.
- 2 On the Events page of the Object Inspector, select the event to which you want to attach a handler.
- 3 Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The procedure above is an easy way to reuse event handlers. *Action lists* and in the VCL, *action bands*, however, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications, whereas action bands cannot. For more information about action lists and action bands, see "Organizing actions for toolbars and menus" on page 6-16.

## Using the Sender parameter

In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an *if...then...else* statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  if Sender = Button1 then
    AboutBox.Caption := 'About ' + Application.Title
  else
    AboutBox.Caption := '';
  AboutBox.ShowModal;
end;
```

## Displaying and coding shared events

When components share events, you can display their shared events in the Object Inspector. First, select the components by holding down the *Shift* key and clicking on them in the Form Designer; then choose the Events tab in the Object Inspector. From the Value column in the Object Inspector, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.

## Associating menu events with event handlers

The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This section explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see “Creating and managing menus” on page 6-29.

To create an event handler for a menu item,

- 1 Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
- 2 Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item’s *Name* property.
- 3 From the Menu Designer, double-click the menu item. Delphi generates an event handler in the Code editor and places the cursor inside the **begin...end** block.
- 4 Inside the **begin...end** block, type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing *OnClick* event handler,

- 1 Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
- 2 Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item’s *Name* property.
- 3 On the Events page of the Object Inspector, click the down arrow in the Value column next to *OnClick* to open a list of previously written event handlers. (The list includes only event handlers written for *OnClick* events on this form.) Select from the list by clicking an event handler name.

## Deleting event handlers

When you delete a component from a form using the Form Designer, Delphi removes the component from the form’s type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method’s forward declaration (in the **interface** section of the unit) and its implementation (in the **implementation** section); otherwise you’ll get a compiler error when you build your project.

## VCL and CLX components

---

The Component palette contains a selection of components that handle a wide variety of programming tasks. You can add, remove, and rearrange components on the palette, and you can create component *templates* and *frames* that group several components.

The components on the palette are arranged in pages according to their purpose and functionality. Which pages appear in the default configuration depends on the version of Delphi you are running. Table 3.3 lists typical default pages and

components available for creating applications. Some of the tabs and components are not cross platform and the table points them out. You can use some VCL-specific nonvisual components in Windows-only CLX applications, however, the applications will not be cross-platform unless you isolate these portions of the code.

**Table 3.3** Component palette pages

Page name	Description	Cross platform?
Standard	Standard controls, menus	Yes
Additional	Specialized controls	Yes except ApplicationEvents and CustomizeDlg
Win32	Windows common controls	Many of the same components are on the Common Controls tab that appears instead when creating CLX applications; RichEdit, UpDown, HotKey, Animate, DateTimePicker, MonthCalendar, Coolbar, PageScroller, and ComboBoxEx are not cross-platform
System	Components and controls for system-level access, including timers, multimedia, and DDE	Timer is but PaintBox, MediaPlayer, OleContainer, and the Dde components are not
Data Access	Components for working with database data that are not tied to any particular data access mechanism	Yes
Data Controls	Visual, data-aware controls	Yes except for DBRichEdit, DBCtrlGrid, and DBChart
dbExpress	Database controls that use dbExpress, a cross-platform, database-independent layer that provides methods for dynamic SQL processing. It defines a common interface for accessing SQL servers.	Yes
DataSnap	Components used for creating multi-tiered database applications	No but can be used in Windows CLX applications
BDE	Components that provide data access through the Borland Database Engine	No but can be used in Windows CLX applications
ADO	Components that provide data access through the ADO framework	No but can be used in Windows CLX applications
InterBase	Components that provide direct access to InterBase	Yes
InternetExpress	Components that are simultaneously a Web Server application and the client of a multi-tiered database application	No but can be used in Windows CLX applications
Internet	Components for Internet communication protocols and Web applications	Yes except for ClientSocket, ServerSocket, QueryTableProducer, XMLDoc, and WebBrowser
WebSnap	Components for building Web server applications	No but can be used in Windows CLX applications
FastNet	NetMasters Internet controls	No but can be used in Windows CLX applications

**Table 3.3** Component palette pages (continued)

Page name	Description	Cross platform?
QReport	QuickReport components for creating embedded reports	No but can be used in Windows CLX applications
Dialogs	Commonly used dialog boxes	Yes except for OpenPictureDialog, SavePictureDialog, PrinterSetupDialog, and PageSetupDialog
Win 3.1	Old style Win 3.1 components	No
Samples	Sample custom components	No
ActiveX	Sample ActiveX controls; see Microsoft documentation (msdn.microsoft.com)	No
COM+	Component for handling COM+ events	No but can be used in Windows CLX applications
WebServices	Components for writing applications that implement or use SOAP-based Web services	No but can be used in Windows CLX applications
Servers	COM Server examples for Microsoft Excel, Word, and so on (see Microsoft MSDN documentation)	No but can be used in Windows CLX applications
Indy Clients	Cross-platform Internet components for the client (open source Winshoes Internet components)	Yes
Indy Servers	Cross-platform Internet components for the server (open source Winshoes Internet components)	Yes
Indy Misc	Additional cross-platform Internet components (open source Winshoes Internet components)	Yes

The online Help provides information about the components on the Component palette. Some of the components on the ActiveX, Servers, and Samples pages, however, are provided as examples only and are not documented.

## Adding custom components to the Component palette

You can install custom components—written by yourself or third parties—on the Component palette and use them in your applications. To write a component, see Part V, “Creating custom components”. To install an existing component, see “Installing component packages” on page 11-5.

## Text controls

---

Many applications present text to the user or allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

Use this component:	When you want users to do this:
<i>TEdit</i>	Edit a single line of text
<i>TMemo</i>	Edit multiple lines of text
<i>TMaskEdit</i>	Adhere to a particular format, such as a postal code or phone number
<i>TRichEdit</i>	Edit multiple lines of text using rich text format (VCL only)

*TEdit* and *TMaskEdit* are simple text controls that include a single line text edit box in which you can type information. When the edit box has focus, a blinking insertion point appears.

You can include text in the edit box by assigning a string value to its *Text* property. You control the appearance of the text in the edit box by assigning values to its *Font* property. You can specify the typeface, size, color, and attributes of the font. The attributes affect all of the text in the edit box and cannot be applied to individual characters.

An edit box can be designed to change its size depending on the size of the font it contains. You do this by setting the *AutoSize* property to *True*. You can limit the number of characters an edit box can contain by assigning a value to the *MaxLength* property.

*TMaskEdit* is a special edit control that validates the text entered against a mask that encodes the valid forms the text can take. The mask can also format the text that is displayed to the user.

*TMemo* is for adding several lines of text.

### Text control properties

Following are some of the important properties of text controls:

**Table 3.4** Text control properties

Property	Description
<i>Text</i>	Determines the text that appears in the edit box or memo control.
<i>Font</i>	Controls the attributes of text written in the edit box or memo control.
<i>AutoSize</i>	Enables the edit box to dynamically change its height depending on the currently selected font.
<i>ReadOnly</i>	Specifies whether the user is allowed to change the text.
<i>MaxLength</i>	Limits the number of characters in simple text controls.

### Properties of memo and rich text controls

Memo and rich text controls, which handle multiple lines of text, have several properties in common. Note that rich text controls are not cross-platform.



*TMemo* is another type of edit box, which handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property.

Memo and rich text controls include other properties such as the following:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the *Modified* property.
- *Lines* contains the text as a list of strings.
- *OEMConvert* determines whether the text is temporarily converted from ANSI to OEM as it is entered. This is useful for validating file names (VCL only).
- *WordWrap* determines whether the text will wrap at the right margin.
- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.
- *SelText* contains the currently selected (highlighted) part of the text.
- *SelStart* and *SelLength* indicate the position and length of the selected part of the text.

At runtime, you can select all the text in the memo with the *SelectAll* method.

## Rich text controls (VCL only)

The rich edit (*TRichEdit*) component is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering.

## Specialized input controls

---

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
<i>TScrollBar</i>	Select values on a continuous range
<i>TTrackBar</i>	Select values on a continuous range (more visually effective than a scroll bar)
<i>TUpDown</i>	Select a value from a spinner attached to an edit component (VCL only)
<i>THotKey</i>	Enter <i>Ctrl/Shift/Alt</i> keyboard sequences (VCL only)
<i>TSpinEdit</i>	Select a value from a spinner widget (CLX only)

## Scroll bars

The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don't require additional coding. For

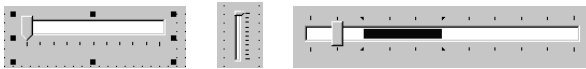
example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBar*.

## Track bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

- Use the *Max* and *Min* properties to set the upper and lower range of the track bar.
- Use *SelEnd* and *SelStart* to highlight a selection range. See Figure 3.4.
- The *Orientation* property determines whether the track bar is vertical or horizontal.
- By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTick* method.

**Figure 3.4** Three views of the track bar component



- *Position* sets a default position for the track bar and tracks the position at runtime.
- By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.
- Set *PageSize* to determine the number of ticks moved when the user presses *Page Up* and *Page Down*.

## Up-down controls (VCL only)

An up-down control (*TUpDown*) consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.

## Spin edit controls (CLX only)

A spin edit control (*TSpinEdit*) is also called an up-down widget, little arrows widget, or spin button. This control lets the application user change an integer value in fixed increments, either by clicking the up or down arrow buttons to increase or decrease the value currently displayed, or by typing the value directly into the spin box.

The current value is given by the *Value* property; the increment, which defaults to 1, is specified by the *Increment* property.

## Hot key controls (VCL only)

Use the hot key component (*THotKey*) to assign a keyboard shortcut that transfers focus to any control. The *HotKey* property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*.

The hot key component can be assigned as the *ShortCut* property of a menu item. Then, when a user enters the key combination specified by the *HotKey* and *Modifiers* properties, Windows activates the menu item.

## Splitter controls

A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment* to *alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to *True* to give the splitter's edge a 3D look.

## Buttons and similar controls

---

Aside from menus, buttons provide the most common way to invoke a command in an application. Delphi offers several button-like controls:

Use this component:	To do this:
<i>TButton</i>	Present command choices on buttons with text
<i>TBitBtn</i>	Present command choices on buttons with text and glyphs
<i>TSpeedButton</i>	Create grouped toolbar buttons
<i>TCheckBox</i>	Present on/off options
<i>TRadioButton</i>	Present a set of mutually exclusive choices
<i>TToolBar</i>	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions
<i>TCoolBar</i>	Display a collection of windowed controls within movable, resizable bands (VCL only)

---

## Button controls

Users click button controls with the mouse to initiate actions. Buttons are labeled with text that represent the action. The text is specified by assigning a string value to the *Caption* property. Most buttons can also be selected by pressing a key on the keyboard as a keyboard shortcut. The shortcut is shown as an underlined letter on the button.

Users click button controls to initiate actions. You can assign an action to a *TButton* component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set *Cancel* to *True* if you want the button to trigger its *OnClick* event when the user presses *Esc*.
- Set *Default* to *True* if you want the *Enter* key to trigger the button's *OnClick* event.

## Bitmap buttons

A bitmap button (*BitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph is to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered in the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

## Speed buttons

Speed buttons, which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

- To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.
- By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*.
- If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio group.

For more information on speed buttons, refer to subtopics in the section “Adding a toolbar using a panel component” on page 6-43.

## Check boxes

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*.

- Set *Checked* to *True* to make the box appear checked by default.
- Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed.
- The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

**Note** Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

## Radio buttons

Radio buttons present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See “Grouping components” on page 3-39 for more information.

A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value `True` or `False` to the `Checked` property to change the radio button's visual state.

## Toolbars

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the `ToolBar` component, then right-click and choose `New Button` to add buttons to the toolbar.

The `TToolBar` component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and `TToolBar` offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

You can use a centralized set of actions on toolbars and menus, by using *action lists* or *action bands*. See "Using action lists" on page 6-23 for details on how to use action lists with buttons and toolbars.

Toolbars can also parent other controls such as edit boxes, combo boxes, and so on.

## Cool bars (VCL only)

A cool bar contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band.

The cool bar requires version 4.70 or later of `COMCTL32.DLL` (usually located in the `Windows\System` or `Windows\System32` directory) at both design time and runtime. Cool bars cannot be used in cross-platform applications.

- The `Bands` property holds a collection of `TCoolBand` objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the `Bands` property in the Object Inspector, then double-click in the Value column to the right, or click the ellipsis (...) button. You can also create bands by adding new windowed controls from the palette.
- The `FixedOrder` property determines whether users can reorder the bands.
- The `FixedSize` property determines whether the bands maintain a uniform height.

## Handling lists

---

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
<code>TListBox</code>	A list of text strings
<code>TCheckListBox</code>	A list with a check box in front of each item
<code>TComboBox</code>	An edit box with a scrollable drop-down list
<code>TTreeView</code>	A hierarchical list

Use this component:	To display:
<i>TListView</i>	A list of (draggable) items with optional icons, columns, and headings
<i>TDateTimePicker</i>	A list box for entering dates or times (VCL only)
<i>TMonthCalendar</i>	A calendar for selecting dates (VCL only)

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see “Working with string lists” on page 3-47.

## List boxes and check-list boxes

List boxes (*TListBox*) and check-list boxes display lists from which users can select items.

- *Items* uses a *TStrings* object to fill the control with values.
- *ItemIndex* indicates which item in the list is selected.
- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *IntegralHeight* specifies whether the list box shows only entries that fit completely in the vertical space (VCL only).
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see “Adding graphics to controls” on page 7-11.

To create a simple list box,

- 1 Within your project, drop a list box component from the Component palette onto a form.
- 2 Size the list box and set its alignment as needed.
- 3 Double-click the right side of the *Items* property or choose the ellipsis button to display the String List Editor.
- 4 Use the editor to enter free form text arranged in lines for the contents of the list box.
- 5 Then choose OK.

To let users select multiple items in the list box, you can use the *ExtendedSelect* and *MultiSelect* properties.

## Combo boxes

A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data.

Three types of combo boxes are: standard, drop-down (the default), and drop-down list.

- Use the *Style* property to select the type of combo box you need.
- Use *csDropDown* if you want an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list). Set the *DropDownCount* property to change the number of items displayed in the list.
- Use *csSimple* to create a combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are displayed.
- Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see “Adding graphics to controls” on page 7-11.

At runtime, CLX combo boxes work differently than VCL combo boxes. In CLX (but not in the VCL combo box), you can add a item to a drop down by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to *ciNone*. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

## Tree views

A tree view (*TTreeView*) displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of '+' and '-' buttons to indicate whether an item can be expanded.
- *ShowLines* enables display of connecting lines to show hierarchical relationships (VCL only).
- *ShowRoot* determines whether lines connecting the top-level items are displayed (VCL only).

To add items to a tree view control at design time, double-click on the control to display the TreeView Items editor. The items you add become the value of the *Items* property. You can change the items at runtime by using the methods of the *Items* property, which is an object of type *TTreeNode*. *TTreeNode* has methods for adding, deleting, and navigating the items in the tree view.

Tree views can display columns and subitems similar to list views in vsReport mode.

## List views

List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window (VCL only).
- *vsList* displays items as labeled icons that cannot be dragged.

- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

## Date-time pickers and month calendars (VCL only)

The *DateTimePicker* component displays a list box for entering dates or times, while the *MonthCalendar* component presents a calendar for entering dates or ranges of dates. To use these components, you must have version 4.70 or later of *COMCTL32.DLL* (usually located in the *Windows\System* or *Windows\System32* directory) at both design time and runtime. They are not available for use in cross-platform applications.

## Grouping components

---

A graphical interface is easier to use when related controls and information are presented in groups. Delphi provides several components for grouping components:

Use this component:	When you want this:
<i>TGroupBox</i>	A standard group box with a title
<i>TRadioGroup</i>	A simple group of radio buttons
<i>TPanel</i>	A more visually flexible group of controls
<i>TScrollBox</i>	A scrollable region containing controls
<i>TTabControl</i>	A set of mutually exclusive notebook-style tabs
<i>TPageControl</i>	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls
<i>THeaderControl</i>	Resizable column headers

## Group boxes and radio groups

A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the Component palette and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the Object Inspector; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

## Panels

The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is



resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel *alTop* aligned so that its position will remain in place even if the form is resized.

The look of the panel can be changed to a raised or lowered look by using the *BevelOuter* and *BevelInner* properties. You can vary the values of these properties to create different visual 3-D effects. Note that if you merely want a raised or lowered bevel, you can use the less resource intensive *TBevel* control instead.

You can also use one or more panels to build various status bars or information display areas.

## Scroll boxes

Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents.

Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

Another use of a scroll box is to restrict scrolling in areas of a window, such as a toolbar or status bar (*TPanel* components). To prevent a toolbar and status bar from scrolling, hide the scroll bars, and then position a scroll box in the client area of the window between the toolbar and status bar. The scroll bars associated with the scroll box will appear to belong to the window, but will scroll only the area inside the scroll box.

## Tab controls

The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the Object Inspector; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

## Page controls

The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control.

To create a new page in a page control at design time, right-click the control and choose **New Page**. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

To access the active page, use the *ActivePage* property. To change the active page, you can set either the *ActivePage* or the *ActivePageIndex* property.

## Header controls

A header control (*THeaderControl*) is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).

## Providing visual feedback

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime.

Use this component or property:	To do this:
<i>TLabel</i> and <i>TStaticText</i>	Display non-editable text
<i>TStatusBar</i>	Display a status region (usually at the bottom of a window)
<i>TProgressBar</i>	Show the amount of work completed for a particular task
<i>Hint</i> and <i>ShowHint</i>	Activate fly-by or “tooltip” help
<i>HelpContext</i> and <i>HelpFile</i>	Link context-sensitive online Help

## Labels and static text components

Labels (*TLabel*) display text and are usually placed next to other controls. You place a label on a form when you need to identify or annotate another component such as an edit box or when you want to include text on a form. The standard label component, *TLabel*, is a non-windowed control (not widget-based in CLX), so it cannot receive focus; when you need a label with a window handle, use *TStaticText* instead.

Label properties include the following:

- *Caption* contains the text string for the label.
- *Font*, *Color*, and other properties determine the appearance of the label. Each label can use only one typeface, size, and color.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.

- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined and enables an accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

Labels usually display read-only static text that cannot be changed by the application user. The application can change the text while it is executing by assigning a new value to the *Caption* property. To add a text object to a form that a user can scroll or edit, use *TEdit*.

## Status bars

Although you can use a panel to make a status bar, it is simpler to use the status bar component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar.

You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the Object Inspector, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. Each panel's *Text* property contains the text displayed in the panel.

## Progress bars

When your application performs a time-consuming operation, you can use a progress bar to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right.

**Figure 3.5** A progress bar



The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

## Help and hint properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.

## Grids

---

Grids display information in rows and columns. If you're writing a database application, use the *TDBGrid* or *TDBCtrlGrid* component described in Chapter 15, "Using data controls." Otherwise, use a standard draw grid or string grid.

### Draw grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

- The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.
- The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.
- You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.
- You can choose to have fixed or non-scrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.
- The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

### String grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

### Value list editors (VCL only)

---

*TValueListEditor* is a specialized grid for editing string lists that contain name/value pairs in the form Name=Value. The names and values are stored as a *TStrings* descendant that is the value of the *Strings* property. You can look up the value for any name using the *Values* property. *TValueListEditor* is not available for cross-platform programming.

The grid contains two columns, one for the names and one for the values. By default, the Name column is named "Key" and the Value column is named "Value". You can change these defaults by setting the *TitleCaptions* property. You can omit these titles using the *DisplayOptions* property (which also controls resize when you resize the control.)

You can control whether users can edit the Name column using the *KeyOptions* property. *KeyOptions* contains separate options to allow editing, adding new names, deleting names, and controlling whether new names must be unique.

You can control how users edit the entries in the Value column using the *ItemProps* property. Each item has a separate *TItemProp* object that lets you

- Supply an edit mask to limit the valid input.
- Specify a maximum length for values.
- Mark the value as read-only.
- Specify that the value list editor displays a drop-down arrow that opens a pick list of values from which the user can choose or an ellipsis button that triggers an event you can use for displaying a dialog in which users enter values.

If you specify that there is a drop-down arrow, you must supply the list of values from which the user chooses. These can be a static list (the *PickList* property of the *TItemProp* object) or they can be dynamically added at runtime using the value list editor's *OnGetPickList* event. You can also combine these approaches and have a static list that the *OnGetPickList* event handler modifies.

If you specify that there is an ellipsis button, you must supply the response that occurs when the user clicks that button (including the setting of a value, if appropriate). You provide this response by writing an *OnEditButtonClick* event handler.

## Displaying graphics

---

The following components make it easy to incorporate graphics into an application.

Use this component:	To display:
<i>TImage</i>	Graphics files
<i>TShape</i>	Geometric shapes
<i>TBevel</i>	3-D lines and frames
<i>TPaintBox</i>	Graphics drawn by your program at runtime
<i>TAnimate</i>	AVI files (VCL only)

### Images

The image component displays a graphical image, like a bitmap, icon, or metafile. The *Picture* property determines the graphic to be displayed. Use *Center*, *AutoSize*, *Stretch*, and *Transparent* to set display options. For more information, see “Overview of graphics programming” on page 8-1.

### Shapes

The shape component displays a geometric shape. It is a nonwindowed control (not widget-based in CLX) and therefore, cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

## Bevels

The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

## Paint boxes

The paint box (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see “Overview of graphics programming” on page 8-1.

## Animation control (VCL only)

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip. An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI files or AVI clips compressed using run-length encoding (RLE). Animation control cannot be used in cross-platform programming.

Following are some of the properties of an animation component:

- *ResHandle* is the Windows handle for the module that contains the AVI clip as a resource. Set *ResHandle* at runtime to the instance handle or module handle of the module that includes the animation resource. After setting *ResHandle*, set the *ResID* or *ResName* property to specify which resource in the indicated module is the AVI clip that should be displayed by the animation control.
- Set *AutoSize* to *True* to have the animation control adjust its size to the size of the frames in the AVI clip.
- *StartFrame* and *StopFrame* specify in which frames to start and stop the clip.
- Set *CommonAVI* to display one of the common Windows AVI clips provided in *Shell32.DLL*.
- Specify when to start and interrupt the animation by setting the *Active* property to *True* and *False*, respectively, and how many repetitions to play by setting the *Repetitions* property.
- The *Timers* property lets you display the frames using a timer. This is useful for synchronizing the animation sequence with other actions, such as playing a sound track.

## Developing dialog boxes

---

The dialog box components on the Dialogs page of the Component palette make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening, saving, and printing files. Dialog boxes display and/or obtain data.

Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns *True*; if the user chooses Cancel to escape from the dialog box without making or saving changes, *Execute* returns *False*.

If you are developing cross-platform applications, you can use the dialogs provided with CLX in the QDialogs unit. For operating systems that have native dialog box types for common tasks, such as for opening or saving a file or for changing font or color, you can use the *UseNativeDialog* property. Set *UseNativeDialog* to *True* if your application will run in such an environment, and if you want it to use the native dialogs instead of the Qt dialogs.

### Using open dialog boxes

One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files using a wildcard character and navigate through directories.

The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box.

When the user chooses OK in the dialog box, the user's file is stored in the *TOpenDialog.FileName* property, which you can then process as you want.

The following code snippet can be placed in an *Action* and linked to the *Action* property of a *TMainMenu* subitem or be placed in the subitem's *OnClick* event:

```
if OpenDialog1.Execute then
    filename := OpenDialog1.FileName;
```

This code will show the dialog box and if the user presses the OK button, it will copy the name of the file into a previously declared *AnsiString* variable named *filename*.

## Using helper objects

---

The VCL and CLX include a variety of nonvisual objects that simplify common programming tasks. This section describes a few Helper objects that make it easier to perform the following tasks:

- Working with lists
- Working with string lists
- Changing the Windows registry and .INI files
- Using streams

## Working with lists

---

The following objects provide functionality for creating and managing lists:

**Table 3.5** Components for creating and managing lists

Object	Maintains
<i>TList</i>	A list of pointers
<i>TObjectList</i>	A memory-managed list of instance objects
<i>TComponentList</i>	A memory-managed list of components (that is, instances of classes descended from <i>TComponent</i> )
<i>TQueue</i>	A first-in first-out list of pointers
<i>TStack</i>	A last-in first-out list of pointers
<i>TObjectQueue</i>	A first-in first-out list of objects
<i>TObjectStack</i>	A last-in first-out list of objects
<i>TClassList</i>	A list of class types
<i>TCollection</i> , <i>TOwnedCollection</i> , and <i>TCollectionItem</i>	Indexed collections of specially defined items
<i>TStringList</i>	A list of strings

For more information about these objects, see the online reference.

## Working with string lists

---

Applications often need to manage lists of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The VCL and CLX provide a common interface to any list of strings through an object called *TStrings* and its descendant *TStringList*. *TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to

- Sort the strings in the list.
- Prohibit duplicate strings in sorted lists.
- Respond to changes in the contents of the list.

In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are an instance of *TStrings*) and then use these lines as items in a combo box (also an instance of *TStrings*).

A string-list property appears in the Object Inspector with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists
- Creating a new string list
- Manipulating strings in a list
- Associating objects with a string list



## Loading and saving string lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the WIN.INI file into a memo field and makes a backup copy called WIN.BAK.

```

procedure EditWinIni;
var
  FileName: string;{ storage for file name }
begin
  FileName := 'C:\WINDOWS\WIN.INI';{ set the file name }
  with Form1.Memo1.Lines do
    begin
      LoadFromFile(FileName);{ load from file }
      SaveToFile(ChangeFileExt(FileName, '.BAK'));{ save into backup file }
    end;
  end;

```

## Creating a new string list

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

### Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a **try...finally** block to ensure that the memory is freed even if an exception occurs.

- 1 Construct the string-list object.
- 2 In the **try** part of a **try...finally** block, use the string list.
- 3 In the **finally** part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

```

procedure TForm1.Button1Click(Sender: TObject);
var
  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try
    { use the string list }
  finally
    TempList.Free;
  end;

```

```

    finally
        TempList.Free;{ destroy the list object }
    end;
end;

```

### Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

- 1 In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.
- 2 Write an event handler for the main form's *constructor*, which executes before the form appears. It should create a string list and assign it to the field you declared in the first step.
- 3 Write an event handler that frees the string list for the form's *OnClose* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

```

unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
{For CLX: uses SysUtils, Classes, QGraphics, QControls, QForms, Qialogs;}

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ClickList := TStringList.Create;{ construct the list }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.LOG'));{ save the list }

```

```
ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;

end.
```

## Manipulating strings in a list

Operations commonly performed on string lists include:

- Counting the strings in a list
- Accessing a particular string
- Finding the position of a string in the list
- Iterating through strings in a list
- Adding a string to a list
- Moving a string within a list
- Deleting a string from a list
- Copying a complete string list

### Counting the strings in a list

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

### Accessing a particular string

The *Strings* array property contains the strings in the list, referenced by a zero-based index. Because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
StringList1.Strings[0] := 'This is the first string.';
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
```

### Locating items in a string list

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns  $-1$  if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the *Items* of a list box:

```
if FileListBox1.Items.IndexOf('WIN.INI') > -1 ...
```

### Iterating through strings in a list

To iterate through the strings in a list, use a **for** loop that runs from zero to *Count*  $- 1$ .

This example converts each string in a list box to uppercase characters.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    Index: Integer;
begin
    for Index := 0 to ListBox1.Items.Count - 1 do
        ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
    end;

```

### Adding a string to a list

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string “Three” the third string in a list, you would use:

```
Insert(2, 'Three');
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2); { append the strings from StringList2 to StringList1 }
```

### Moving a string within a list

To move a string in a string list, call the *Move* method, passing two parameters: the current index of the string and the index you want assigned to it. For example, to move the third string in a list to the fifth position, you would use:

```
Move(2, 4)
```

### Deleting a string from a list

To delete a string from a string list, call the list’s *Delete* method, passing the index of the string you want to delete. If you don’t know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

This example uses *IndexOf* and *Delete* to find and delete a string:

```

with ListBox1.Items do
begin
    BIndex := IndexOf('bureaucracy');
    if BIndex > -1 then
        Delete(BIndex);
end;

```

### Copying a complete string list

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
Memo1.Lines.Assign(ComboBox1.Items); { overwrites original strings }
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items); { appends strings to end }
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you assign one string-list variable to another—

```
StringList1 := StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

## Associating objects with a string list

In addition to the strings stored in its *Strings* property, a string list can maintain references to objects, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear*, and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

## Windows registry and INI files

---

The Windows system registry is a hierarchical database where applications store configuration information. The VCL class *TRegistry* supplies methods that read and write to the registry.

Until Windows 95, most applications stored configuration information in initialization files, usually named with the extension .INI. The VCL provides the following classes to facilitate maintenance and migration of programs that use INI files:

- *TRegistry* to work with the registry (VCL only).
- *TIniFile* (VCL only) or *TMemIniFile* to work with INI files.
- *TRegistryIniFile* when you want to work with both the registry and INI files (VCL only). *TRegistryIniFile* has properties and methods similar to those of *TIniFile*, but it reads and writes to the system registry. By using a variable of type *TCustomIniFile* (the common ancestor of *TIniFile*, *TMemIniFile*, and *TRegistryIniFile*), you can write generic code that accesses either the registry or an INI file, depending on where it is called.

Only *TMemIniFile* can be used in cross-platform programming.

### Using TIniFile (VCL only)

The INI file format is still popular, many of the Delphi configuration files (such as the DSK Desktop settings file) are in this format. Because this file format was and is prevalent, VCL provides a class to make reading and writing these files very easy. *TIniFile* is not available for cross-platform programming.

When you instantiate the *TIniFile* object, you pass as a parameter to the constructor the name of the INI file. If the file does not exist, it is automatically created. You are then free to read values using *ReadString*, *ReadInteger*, or *ReadBool*. Alternatively, if you want to read an entire section of the INI file, you can use the *ReadSection* method. Similarly, you can write values using *WriteBool*, *WriteInteger*, or *WriteString*.

Each of the Read routines takes three parameters. The first parameter identifies the section of the INI file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the INI file. Similarly, the Write routines will create the section and/or value if they do not exist. The example code creates an INI file the first time it is run that looks like this:

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

On subsequent execution of this application, the INI values are read in during creation of the form and written back out in the *OnClose* event.

## Using TRegistry

Most 32-bit applications store their information in the registry instead of INI files because the registry is hierarchical, more robust, and doesn't suffer from the size limitations of INI files. The *TRegistry* object contains methods to open, close, save, move, copy, and delete keys.

*TRegistry* is not available for cross-platform programming.

For more information, see the *TRegistry* topic in the online help.

## Using TRegIniFile

If you are accustomed to using INI files and want to move your configuration information to the registry instead, you can use the *TRegIniFile* class. *TRegIniFile* is designed to make registry entries look like INI file entries. All the methods from *TIniFile* (read and write) exist in *TRegIniFile*.

When you construct a *TRegIniFile* object, the parameter you pass (the filename for an *IniFile* object) becomes a key value under the user key in the registry, and all sections and values branch from that root. In fact, this object simplifies the registry interface considerably, so you may want to use it instead of the *TRegistry* component even if you aren't porting existing code.

*TRegIniFile* is not available for cross-platform programming.

For more information, see the *TRegIniFile* topic in the VCL online reference.

## Creating drawing spaces

---

The *TCanvas* encapsulates a Windows device context in the VCL and a paint device (Qt painter) in CLX, which handles all drawing for both forms, visual containers (such as panels) and the printer object (covered in “Printing” on page 3-54 “).

Using the canvas object, you no longer have to worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you.

*TCanvas* includes a large number of primitive graphics routines to draw lines, shapes, polygons, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to the middle of the form and outputs some raw text onto the form:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Canvas.Pen.Color := clBlue;
    Canvas.MoveTo( 10, 10 );
    Canvas.LineTo( 100, 100 );
    Canvas.Brush.Color := clBtnFace;
    Canvas.Font.Name := 'Arial';
    Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y, 'This is the end of the line' );
end;

```

In Windows applications, the *TCanvas* object also protects you against common Windows graphics errors, such as restoring device contexts, pens, brushes, and so on to the value they had before the drawing operation. *TCanvas* is used everywhere in Delphi that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

See *TCanvas* in the online help reference for a complete listing of properties and methods.

## Printing

---

The VCL *TPrinter* object encapsulates details of Windows printers. To get a list of installed and available printers, use the *Printers* property. The CLX *TPrinter* object is a paint device that paints on a printer. It generates postscript and sends that to *lpr*, *lp*, or another print command.

Both printer objects use a *TCanvas* (which is identical to the form's *TCanvas*) which means that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method.

This example uses a button and a memo on a form. When the user clicks the button, the content of the memo is printed with a 200-pixel border around the page.

To run this example successfully, add `Printers` to your `uses` clause.

```

procedure TForm1.Button1Click(Sender: TObject);
var
    r: TRect;
    i: Integer;
begin
    with Printer do
        begin
            r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
            BeginDoc;
            for i := 0 to Memo1.Lines.Count do
                Canvas.TextOut(200,200 + (i *
Canvas.TextHeight(Memo1.Lines.Strings[i])),
                                Memo1.Lines.Strings[i]);
                Canvas.Brush.Color := clBlack;
                Canvas.FrameRect(r);
            EndDoc;
        end;
    end;

```

For more information on the use of the *TPrinter* object, look in the online help under *TPrinter*.

## Using streams

---

Streams are just ways of reading and writing data. Streams provide a common interface for reading and writing to different media such as memory, strings, sockets, and blob streams.

In the following streaming example, one file is copied to another one using streams. The application includes two edit controls (From and To) and a Copy File button.

```

procedure TForm1.CopyFileClick(Sender: TObject);
var
    stream1, stream2:TStream;
begin
    stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
    try
        stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
        try
            stream2.CopyFrom(Stream1,Stream1.Size);
        finally
            stream2.Free;
        finally
            stream1.Free
    end;

```

Use specialized stream objects to read or write to storage media. Each descendant of *TStream* implements methods for accessing a particular medium, such as disk files, dynamic memory, and so on. *TStream* descendants include *TFileStream*, *TStringStream*, and *TMemoryStream*. In addition to methods for reading and writing, these objects permit applications to seek to an arbitrary position in the stream. Properties of *TStream* provide information about the stream, such as size and current position.





# Common programming tasks

This chapter discusses how to perform some of the common programming tasks in Delphi:

- Understanding classes
- Defining classes
- Handling exceptions
- Using interfaces
- Defining custom variants
- Working with strings
- Working with files
- Converting measurements

## Understanding classes

---

A *class* is an abstract definition of properties, methods, events, and class members (such as variables local to the class). When you create an instance of a class, this instance is called an object. The term object is often used more loosely in the Delphi documentation and where the distinction between a class and an instance of the class is not important, the term “object” may also refer to a class.

Although Delphi includes many classes in its object hierarchy, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants. A class type declaration contains three possible sections that control the accessibility of its fields and methods:

```
Type
TClassName = Class(TObject)
public
    {public fields}
    {public methods}
```

```
    protected
        {protected fields}
        {protected methods}
    private
        {private fields}
        {private methods}
end;
```

- The public section declares fields and methods with no access restrictions; class instances and descendant classes can access these fields and methods.
- The protected section includes fields and methods with some access restrictions; descendant classes can access these fields and methods.
- The private section declares fields and methods that have rigorous access restrictions; they cannot be accessed by class instances or descendant classes.

The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior.

The general syntax of a descendant class is as follows:

```
Type
TClassName = Class (TParentClass)
    public
        {public fields}
        {public methods}
    protected
        {protected fields}
        {protected methods}
    private
        {private fields}
        {private methods}
end;
```

If no parent class name is specified, the class inherits directly from *TObject*. *TObject* defines only a handful of methods, including a basic constructor and destructor.

For more information about the syntax, language definitions, and rules for classes, see the *Object Pascal Language Guide* online Help on Class types.

## Defining classes

---

Delphi allows you to declare classes that implement the programming features you need to use in your application. Some versions of Delphi include a feature called class completion that simplifies the work of defining and implementing new classes by generating skeleton code for the class members you declare.

To define a class,

- 1 In the IDE, start with a project open and choose File | New | Unit to create a new unit where you can define the new class.

- 2 Add the **uses** clause and **type** section to the **interface** section.
- 3 In the **type** section, write the class declaration. You need to declare all the member variables, properties, methods, and events.

```
TMyClass = class; {This implicitly descends from TObject}
public
  .
  .
  .
private
  .
  .
published {If descended from TPersistent or below}
  .
  .
```

**Note** The object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {\$M+} compiler directive, or descend from *TPersistent* or below.

If you want the class to descend from a specific class, you need to indicate that class in the definition:

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

For example:

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

If your version of Delphi includes class completion: place the cursor within a method definition in the **interface** section and press Ctrl+Shift+C (or right-click and select Complete Class at Cursor). Delphi completes any unfinished property declarations and creates the empty methods you need in the **implementation** section. (If you do not have class completion, you'll need to write the code yourself, completing property declarations and writing the methods.)

Given the example above, if you have class completion, Delphi adds **read** and **write** specifiers to your interface declaration, including any supporting fields or methods:

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

It also adds the following code to the **implementation** section of the unit.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
```

## Handling exceptions

```
end;  
procedure TMyButton.SetSize(const Value: Integer);  
begin  
    FSize := Value;  
end;
```

- 4 Fill in the methods. For example, to make it so the button beeps when you call the DoSomething method, add the Beep between **begin** and **end**.

```
{ TMyButton }  
procedure TMyButton.DoSomething;  
begin  
    Beep;  
end;  
  
procedure TMyButton.SetSize(const Value: Integer);  
begin  
    if fsize < > value then  
    begin  
        FSize := Value;  
        DoSomething;  
    end;  
end;
```

Note that the button also beeps when you call SetSize to change the size of the button.

For more information about the syntax, language definitions, and rules for classes and methods, see the *Object Pascal Language Guide* online Help on Class types and methods.

## Handling exceptions

---

Delphi provides a mechanism to handle errors in a consistent manner. Exception handling allows the application to recover from errors if possible and to shut down if need be, without losing data or resources. Error conditions in Delphi are indicated by exceptions. This section describes the following tasks for using exceptions to create safe applications:

- Protecting blocks of code
- Protecting resource allocations
- Handling RTL exceptions
- Handling component exceptions
- Exception handling with external sources
- Silent exceptions
- Defining your own exceptions

### Protecting blocks of code

---

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application will present a message box describing the error. Your job, then, is to recognize places

where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

To protect blocks of code you need to understand

- Responding to exceptions
- Exceptions and the flow of control
- Nesting exception responses

## Responding to exceptions

When an error condition occurs, the application raises an exception, meaning it creates an exception object. Once an exception is raised, your application can execute cleanup code, handle the exception, or both.

### Executing cleanup code

The simplest way to respond to an exception is to guarantee that some cleanup code is executed. This kind of response doesn't correct the condition that caused the error but lets you ensure that your application doesn't leave its environment in an unstable state. You typically use this kind of response to ensure that the application frees allocated resources, regardless of whether errors occur.

### Handling an exception

This is a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are easy to correct and retry, while others, such as running out of memory, might be more difficult for the application or the user to correct.

To handle exceptions effectively, you need to understand the following:

- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

## Exceptions and the flow of control

Object Pascal makes it easy to incorporate error handling into your applications because exceptions don't get in the way of the normal flow of your code. In fact, by moving error checking and error handling out of the main flow of your algorithms, exceptions can simplify the code you write.

When you declare a protected block, you define specific responses to exceptions that might occur within that block. When an exception occurs in that block, execution immediately jumps to the response you defined, then leaves the block.

**Example** The following code that includes a protected block. If any exception occurs in the protected block, execution jumps to the exception-handling part, which beeps. Execution resumes outside the block.

```

try
  AssignFile(F, FileName);
  Reset(F);
  :
except
  on Exception do Beep;
end;
  : { execution resumes here, outside the protected block }

```

## Nesting exception responses

Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks of code inside other blocks, you can customize responses even within blocks that already contain customized responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:

```

      { allocate first resource }
      try
        { allocate second resource }
        try
          { code that uses both resources }
          finally
            { release second resource }
          end;
        finally
          { release first resource }
        end;

```

You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:

```

exception-handling block
┌─── try
│   { protected code }
│   ┌─── try
│   │   { specially protected code }
│   │   └─── except
│   │       { local exception handling }
│   │       └─── end;
│   └─── except
│       { global exception handling }
└─── end;

```

You can also mix different kinds of exception-response blocks, nesting resource protections within exception handling blocks and vice versa.

## Protecting resource allocations

---

One key to having a robust application is ensuring that if it allocates resources, it also releases them, even if an exception occurs. For example, if your application allocates memory, you need to make sure it eventually releases the memory, too. If it opens a file, you need to make sure it closes the file later.

Keep in mind that exceptions don't come just from your code. A call to an RTL routine, for example, or another component in your application might raise an exception. Your code needs to ensure that if these conditions occur, you release allocated resources.

To protect resources effectively, you need to understand the following:

- What kind of resources need protection?
- Creating a resource protection block

### What kind of resources need protection?

Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are:

- Files
- Memory
- Windows resources (VCL only)
- Objects



**Example** The following event handler allocates memory, then generates an error, so it never executes the code to free the memory:

```

procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an error }
  FreeMem(APointer, 1024);{ it never gets here }
end;

```

Although most errors are not that obvious, the example illustrates an important point: When the division-by-zero error occurs, execution jumps out of the block, so the *FreeMem* statement never gets to free the memory.

To guarantee that the *FreeMem* gets to free the memory allocated by *GetMem*, you need to put the code in a resource-protection block.

## Creating a resource protection block

To ensure that you free allocated resources, even in case of an exception, you embed the resource-using code in a protected block, with the resource-freeing code in a special part of the block. Here's an outline of a typical protected resource allocation:

```

{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;

```

The key to the **try..finally** block is that the application always executes any statements in the **finally** part of the block, even if an exception occurs in the protected block. When any code in the **try** part of the block (or any routine called by code in the **try** part) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the **finally** part, which is called the cleanup code. After the finally part is executed, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the **try** part.

**Example** The following code illustrates an event handler that allocates memory and generates an error, but still frees the allocated memory:

```

procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an error }
  finally
    FreeMem(APointer, 1024);{ it never gets here }
  end;

```

```

finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the error }
end;
end;

```

The statements in the **finally** block do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the **finally** block.

## Handling RTL exceptions

---

When you write code that calls routines in the runtime library (RTL), such as mathematical functions or file-handling procedures, the RTL reports errors back to your application in the form of exceptions. By default, RTL exceptions generate a message that the application displays to the user. You can define your own exception handlers to handle RTL exceptions in other ways.

There are also silent exceptions that do not, by default, display a message.

RTL exceptions are handled like any other exceptions. To handle RTL exceptions effectively, you need to understand the following:

- What are RTL exceptions?
- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

### What are RTL exceptions?

The runtime library's exceptions are defined in the *SysUtils* unit, and they all descend from a generic exception-object type called *Exception*. *Exception* provides the string for the message that RTL exceptions display by default.

Several kinds of exceptions can be raised by the RTL, as described in the following table.

**Table 4.1** RTL exceptions

Error type	Cause	Meaning
Input/output	Error accessing a file or I/O device	Most I/O exceptions are related to error codes returned when accessing a file.
Heap	Error using dynamic memory	Heap errors can occur when there is insufficient memory available, or when an application disposes of a pointer that points outside the heap.
Integer math	Illegal operation on integer-type expressions	Errors include division by zero, numbers or expressions out of range, and overflows.

**Table 4.1** RTL exceptions (continued)

Error type	Cause	Meaning
Floating-point math	Illegal operation on real-type expressions	Floating-point errors can come from either a hardware coprocessor or the software emulator. Errors include invalid instructions, division by zero, and overflow or underflow.
Typecast	Invalid typecasting with the <code>as</code> operator	Objects can only be typecast to compatible types.
Conversion	Invalid type conversion	Type-conversion functions such as <code>IntToStr</code> , <code>StrToInt</code> , and <code>StrToFloat</code> raise conversion exceptions when the parameter cannot be converted to the desired type.
Hardware	System condition	Hardware exceptions indicate that either the processor or the user generated some kind of error condition or interruption, such as an access violation, stack overflow, or keyboard interrupt.
Variant	Illegal type coercion	Errors can occur when referring to variants in expressions where the variant cannot be coerced into a compatible type.

For a list of the RTL exception types, see the code in the *SysUtils* unit.

## Creating an exception handler

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. In cross-platform programming, it is very rare that you will need to write an exception handler. Most exceptions can be handled using `try..finally` blocks as described in “Protecting blocks of code” on page 4-4 and “Protecting resource allocations” on page 4-7.

To define an exception handler, embed the code you want to protect in an exception-handling block and specify the exception handling statements in the **except** part of the block. Here is an outline of a typical exception-handling block:

```
try
  { statements you want to protect }
except
  { exception-handling statements }
end;
```

The application executes the statements in the **except** part only if an exception occurs during execution of the statements in the **try** part. Execution of the **try** part statements includes routines called by code in the **try** part. That is, if code in the **try** part calls a routine that doesn't define its own exception handler, execution returns to the exception-handling block, which handles the exception.

When a statement in the **try** part raises an exception, execution immediately jumps to the **except** part, where it steps through the specified exception-handling statements, or exception handlers, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

## Exception handling statements

Each **on** statement in the **except** part of a **try..except** block defines code for handling a particular kind of exception. The form of these exception-handling statements is as follows:

```
on <type of exception> do <statement>;
```

**Example** You can define an exception handler for division by zero to provide a default result:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems; { handle the normal case }
  except
    on EDivByZero do Result := 0; { handle the exception only if needed }
  end;
end;
```

Note that this is clearer than having to test for zero every time you call the function. Here's an equivalent function that doesn't take advantage of exceptions:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  if NumberOfItems <> 0 then { always test }
    Result := Sum div NumberOfItems { use normal calculation }
  else Result := 0; { handle exceptional case }
end;
```

The difference between these two functions really defines the difference between programming with exceptions and programming without them. This example is quite simple, but you can imagine a more complex calculation involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid.

By using exceptions, you can spell out the “normal” expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every single time to make sure you're allowed to proceed with each step in the calculation.

## Using the exception instance

Most of the time, an exception handler doesn't need any information about an exception other than its type, so the statements following **on..do** are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of **on..do** that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance.

**Example** If you create a new project that contains a single form, you can add a scroll bar and a command button to the form. Double-click the button and add the following line to its click-event handler:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

That line raises an exception because the maximum value of a scroll bar must always exceed the minimum value. The default exception handler for the application opens a

dialog box containing the message in the exception object. You can override the exception handling in this handler and create your own message box containing the exception's message string:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

The temporary variable (E in this example) is of the type specified after the colon (*EInvalidOperation* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

**Note** Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

### Scope of exception handlers

You do not need to provide handlers for every kind of exception in every block. In fact, you only need handlers for exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains the block (or returns to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

### Providing default exception handlers

You can provide a single default exception handler to handle any exceptions you haven't provided specific handlers for. To do that, you add an else part to the **except** part of the exception-handling block:

```
try
  { statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from the containing block.

**Caution** It is not advisable to use this all-encompassing default exception handler. The **else** clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more

information about the exception and how to handle it, then you can do so use an enclosing **try..finally** block:

```
try
  try
    { statements }
  except
    on ESomething do { specific exception-handling code };
  end;
finally
  {cleanup code };
end;
```

For another approach to augmenting exception handling, see Reraising the exception.

## Handling classes of exceptions

Because exception objects are part of a hierarchy, you can specify handlers for entire parts of the hierarchy by providing a handler for the exception class from which that part of the hierarchy descends.

**Example** The following block outlines an example that handles all integer math exceptions specially:

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

You can still specify specific handlers for more specific exceptions, but you need to place those handlers above the generic handler, because the application searches the handlers in the order they appear in, and executes the first applicable handler it finds. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError*.

## Reraising the exception

Sometimes when you handle an exception locally, you actually want to augment the handling in the enclosing block, rather than replacing it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond.

**Example** When an exception occurs, you might want to display a message to the user or record the error in a log file, then proceed with the standard handling. To do that, you declare a local exception handler that displays the message then calls the reserved word `raise`. This is called reraising the exception, as shown in this example:

```

try
  { statements }
  try
    { special statements }
  except
    on ESomething do
      begin
        { handling for only the special statements }
        raise; { reraise the exception }
      end;
    end;
  except
    on ESomething do ...; { handling you want in all cases }
  end;

```

If code in the `{ statements }` part raises an *ESomething* exception, only the handler in the outer **except** part executes. However, if code in the `{ special statements }` part raises *ESomething*, the handling in the inner **except** part is executed, followed by the more general handling in the outer **except** part.

By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

## Handling component exceptions

---

Delphi's components raise exceptions to indicate error conditions. Most component exceptions indicate programming errors that would otherwise generate a runtime error. The mechanics of handling component exceptions are no different than handling RTL exceptions.

**Example** A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a "List index out of bounds" exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string'); { add a string to list box }
  ListBox1.Items.Add('another string'); { add another string... }
  ListBox1.Items.Add('still another string'); { ...and a third string }
  try
    Caption := ListBox1.Items[3]; { set form caption to fourth string in list box }
  except
    on EStringListError do
      MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
    end;
  end;

```

If you click the button once, the list box has only three strings, so accessing the fourth string (`Items[3]`) raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

## Exception handling with external sources

---

*HandleException* provides default handling of exceptions for the application. Normally when developing cross-platform applications, you do not need to call *TApplication.HandleException*. However, you may need it when writing shared object files or callback functions. You can use *TApplication.HandleException* to block an exception from escaping from your code particularly when the code is being called from an external source that does not support exceptions.

For example, if an exception passes through all the **try** blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating that an error has occurred. You can use *HandleException* in this fashion:

```
try
  { statements }
except
  Application.HandleException(Self);
end;
```

For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. Therefore, if you want to both handle the exception, and provide this default behavior as the built-in components do, you can add a call to *HandleException* to your code:

```
try
  { special statements }
except
  on ESomething do
  begin
    { handling for only the special statements }
    Application.HandleException(Self); { call HandleException }
  end;
end;
```

**Note** Do not call *HandleException* from within a thread's exception handling code.

For more information, search for exception handling routines in the Help index.

## Silent exceptions

---

Delphi applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the



*Break* or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for Delphi VCL and CLX applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

**Note** For console applications, an error-message dialog is displayed on any unhandled *EAbort* exceptions.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which will break out of the current operation without displaying an error message.

**Example** The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 10 do{ loop ten times }
  begin
    ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
    if I = 7 then Abort;{ abort after the seventh one }
  end;
end;
```

## Defining your own exceptions

---

In addition to protecting your code from exceptions generated by the runtime library and various components, you can use the same mechanism to manage exception conditions in your own code.

To use exceptions in your code, you need to complete these steps:

- Declaring an exception object type
- Raising an exception

### Declaring an exception object type

Because exceptions are objects, defining a new kind of exception is as simple as declaring a new object type. Although you can raise any object instance as an exception, the standard exception handlers handle only exceptions descended from *Exception*.

As a convention, new exception types should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by a specific exception handler for that exception, one of the standard handlers will handle it instead.

**Example** For example, consider the following declaration:

```
type
  EMyException = class(Exception);
```

If you raise *EMyException* but don't provide a specific handler for it, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

## Raising an exception

To indicate a disruptive error condition in an application, you can raise an exception that involves constructing an instance of that type and calling the reserved word **raise**.

To raise an exception, call the reserved word **raise**, followed by an instance of an exception object. This allows you to establish an exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

Raising an exception address set the *ErrorAddr* variable in the *System* unit to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user where the error occurred. You can also specify a value in the **raise** clause which will appear in *ErrorAddr* when an exception occurs.

**Warning** Do not assign a value to *ErrorAddr* yourself. It is intended as read-only.

To specify an error address for an exception, add the reserved word **at** after the exception instance, followed by an address expression such as an identifier.

For example, given the following declaration,

```
type
  EPasswordInvalid = class(Exception);
```

you can raise a "password invalid" exception at any time by calling **raise** with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered');
```

## Using interfaces

---

Delphi's **interface** keyword allows you to create and use interfaces in your application. Interfaces are a way extending the single-inheritance model of Object Pascal by allowing a single class to implement more than one interface, and by allowing several classes descended from different bases to share the same interface. Interfaces are useful when the same sets of operations, such as streaming, are used across a broad range of objects. Interfaces are also a fundamental aspect of the COM (the Component Object Model) and CORBA (Common Object Request Broker Architecture) distributed object models.

## Interfaces as a language feature

---

An interface is like a class that contains only abstract methods and a clear definition of their functionality. Strictly speaking, interface method definitions include the number and types of their parameters, their return type, and their expected behavior. Interface methods are usually named to indicate the purpose of the interface. It is the convention to name interfaces according to their behavior and to preface them with a capital *I*. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file.

An interface has the following syntax:

```
IMyObject = interface
  procedure MyProcedure;
end;
```

A simple example of declaring an interface is:

```
type
  IEdit = interface
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
  end;
```

Like abstract classes, interfaces themselves can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, you must define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

### Implementing interfaces across the hierarchy

Using interfaces offers a design approach to separating the way a class is used from the way it is implemented. Two classes can implement the same interface without requiring that they descend from the same base class. This polymorphic invocation of the same methods on unrelated objects is possible as long as the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from say, *TFigure* which implemented a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. The above *IPaint* would be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

which makes sense for the rectangle to support but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern used to fill the circle without having to add rotation to the simple circle.

**Note** For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IInterface* that manage reference counting. For more information, see “Implementing *IInterface*” on page 4-20 and “Memory management of interface objects” on page 4-24.

## Using interfaces with procedures

Interfaces also allow you to write generic procedures that can handle objects without requiring the objects to descend from a particular base class. Using the above *IPaint* and *IRotate* interfaces you can write the following procedures,

```

procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;

```

*RotateObjects* does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate. This allows the above generic procedures to be used more often than if they were written to only work against a *TFigure* class.

For details about the syntax, language definitions and rules for interfaces, see the *Object Pascal Language Guide* online Help section on Object interfaces.

## Implementing Interface

---

All interfaces derive either directly or indirectly from the *IInterface* interface. This interface provides the essential functionality of an interface, that is, dynamic querying and lifetime management. This functionality is established in the three *IInterface* methods:

- *QueryInterface* provides a method for dynamically querying a given object and obtaining interface references for the interfaces the object supports.
- *\_AddRef* is a reference counting method that increments the count each time the call to *QueryInterface* succeeds. While the reference count is nonzero the object must remain in memory.
- *\_Release* is used with *\_AddRef* to enable an object to track its own lifetime and to determine when it is safe to delete itself. Once the reference count reaches zero, the object is freed from memory.

Every class that implements interfaces must implement the three *IInterface* methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of life-time management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

## TInterfacedObject

---

Delphi defines a simple class, *TInterfacedObject*, that serves as a convenient base because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:

```

type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;

```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```

type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;

```

Because it implements the methods of *IInterface*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see “Memory management of interface objects” on page 4-24, which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

## Using the as operator

---

Classes that implement interfaces can use the **as** operator for dynamic binding on the interface. In the following example:

```

procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;

begin
  X := P as IPaint;
  { statements }
end;

```

the variable *P* of type *TInterfacedObject*, can be assigned to the variable *X*, which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of *P*'s *IInterface* interface. This is because the compiler cannot tell from *P*'s declared type whether *P*'s instance actually supports *IPaint*. At runtime, *P* either resolves to an *IPaint* reference or an exception is raised. In either case, assigning *P* to *X* will not generate a compile-time error as it would if *P* was of a class type that did not implement *IInterface*.

When you use the **as** operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IInterface*: Although all interfaces derive from *IInterface*, it is not sufficient, if you want to use the **as** operator, for a class to simply implement the methods of *IInterface*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IInterface* in its interface list.
- Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the **as** operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the *Ctrl+Shift+G* editor shortcut key.

## Reusing code and delegation

---

One approach to reusing code with interfaces is to have an object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, Object Pascal has a keyword **implements**, that makes it easy to write code to delegate all or part of the implementation of an interface to a subobject. Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object contains an inner object that implements interfaces which are exposed only by the outer object. The VCL and CLX have classes that support aggregation.

### Using implements for delegation

Many classes have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword **implements** to specify that the methods of that interface are delegated to the object or interface reference which is the property instance. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list.

By default using the keyword **implements** delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods as a way of overriding this default behavior.

The following example uses the `implements` keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```

unit cadapt;

type
IRGB8bit = interface
  ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;

IColorRef = interface
  ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;

{ TRGB8ColorRefAdapter  map an IRGB8bit to an IColorRef }
TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
  end;
end.

```

For more information about the syntax, implementation details, and language rules of the `implements` keyword, see the *Object Pascal Language Guide* online Help section on Object interfaces.

## Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that define the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. The only requirement is that it implement *Interface*. The inner object, or objects, can implement one or more interfaces, however only the outer object exposes the



interfaces. These include both the interfaces it implements and the ones implemented by its contained objects. Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The implementation model for aggregation defines explicit rules for implementing *IInterface* using delegation. The inner object must implement an *IInterface* on itself, that controls the inner object's reference count. This implementation of *IInterface* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *IInterface*. The inner object also implements a second *IInterface* for all the interfaces it implements. These are the interfaces exposed by the outer object. This second *IInterface* delegates calls to *QueryInterface*, *AddRef*, and *Release* to the outer object. The outer *IInterface* is referred to as the "controlling Unknown."

Refer to the MS online help for the rules about creating an aggregation. When writing your own aggregation classes, you can also refer to the implementation details of *IInterface* in *TComObject*. *TComObject* is a COM class that supports aggregation. If you are writing COM applications, you can also use *TComObject* directly as a base class.

## Memory management of interface objects

---

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *\_AddRef* and *\_Release* methods of *IInterface* provide a way to implement this lifetime management. *\_AddRef* and *\_Release* track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero.

If you are creating COM objects for distributed applications (in the Windows environment only), then you should strictly adhere to the reference counting rules. However, if you are using interfaces only internally in your application, then you have a choice that depends upon the nature of your object and how you decide to use it.

### Using reference counting

Delphi provides most of the *IInterface* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from these classes. *TInterfacedObject* is the non-CoClass that provides this behavior. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);
function test_func()
var
```

```

    y: ITest;
begin
    y := TTest.Create; // because y is of type ITest, the reference count is one
    beep(y); // the act of calling the beep function increments the reference count
              // and then decrements it when it returns
    y.something; // object is still here with a reference count of one
end;

```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```

function test_func()
var
    x: TTest;
begin
    x := TTest.Create; // no count on the object yet
    beep(x as ITest); // count is incremented by the act of calling beep
                      // and decremented when it returns
    x.something; // surprise, the object is gone
end;

```

**Note** In the examples above, the *beep* procedure, as it is declared, increments the reference count (call *\_AddRef*) on the parameter, whereas either of the following declarations do not:

```

procedure beep(const x: ITest);

```

or

```

procedure beep(var x: ITest);

```

These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting because the lifetime of the object is not dictated by its interfaces.

## Not using reference counting

If your object is a component or a control that is owned by another component, then your object is part of a different memory management system that is based in *TComponent*. You should not mix the object lifetime management approaches of VCL or CLX components and interface reference counting. If you want to create a component that supports interfaces, you can implement the *IInterface* *\_AddRef* and *\_Release* methods as empty functions to bypass the interface reference counting mechanism:

```

function TMyObject._AddRef: Integer;
begin
    Result := -1;
end;

function TMyObject._Release: Integer;
begin
    Result := -1;
end;

```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object.

Note that, because you do implement *QueryInterface*, you can still use the **as** operator for interfaces on components, as long as you create an interface identifier (IID). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the “controlling Unknown.” It is at the level of the outer, component object that the decision is made to circumvent the *\_AddRef* and *\_Release* methods, and to handle memory management via the component-based approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a component as its containing outer object.

**Note** The “controlling Unknown” is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. *IUnknown* is the same as *IInterface*, but is used instead in COM-based applications (Windows only). For more information distinguishing the various implementations of the *IUnknown* or *IInterface* interface by the inner and outer objects, see “Aggregation” on page 4-23 and the Microsoft online Help topics on the “controlling Unknown.”

## Using interfaces in distributed applications (VCL only)

---

Interfaces are a fundamental element in the COM, SOAP, and CORBA distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IInterface* interface methods.

When using COM, classes and interfaces are defined in terms of *IUnknown* rather than *IInterface*. There is no semantic difference between *IUnknown* and *IInterface*, the use of *IUnknown* is simply a way to adapt Delphi interfaces to the COM definition. COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class instances via CLSIDs. The CLSIDs are used to register and manipulate COM classes. COM classes that have class factories and class identifiers are called CoClasses. CoClasses take advantage of the versioning capabilities of *QueryInterface*, so that when a software module is updated *QueryInterface* can be invoked at runtime to query the current capabilities of an object.

New versions of old interfaces, as well as any new interfaces or features of an object, can become immediately available to new clients. At the same time, objects retain complete compatibility with existing client code; no recompilation is necessary because interface implementations are hidden (while the methods and parameters remain constant). In COM applications, developers can change the implementation to improve performance, or for any internal reason, without breaking any client code that relies on that interface. For more information about COM interfaces, see Chapter 33, “Overview of COM technologies.”

When distributing an application using SOAP, interfaces are required to carry their own runtime type information (RTTI). The compiler only adds RTTI to an interface when it is compiled using the `{M+}` switch. Such interfaces are called *invokable interfaces*. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, client

applications can only call the methods defined in the invocable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be called by clients.

The easiest way to define invocable interfaces is to define your interface so that it descends from *IInvokable*. *IInvokable* is the same as *IInterface*, except that it is compiled using the {SM+} switch. For more information about Web Service applications that are distributed using SOAP, and about invocable interfaces, see Chapter 31, "Using Web Services."

Another distributed application technology is CORBA. The use of interfaces in CORBA applications is mediated by stub classes on the client and skeleton classes on the server. These stub and skeleton classes handle the details of marshaling interface calls so that parameter values and return values can be transmitted correctly. Applications must use either a stub or skeleton class, or employ the Dynamic Invocation Interface (DII) which converts all parameters to special variants (so that they carry their own type information).

## Defining custom variants

---

One powerful built-in type of the Object Pascal language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions.

By default, variants can't hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create a descendant of the *TCustomVariantType* class that indicates how the Variant type performs standard operations.

To create a Variant type,

- 1 Map the storage of the variant's data on to the *TVarData* record.
- 2 Declare a class that descends from *TCustomVariantType*. Implement all required behavior (including type conversion rules) in the new class.
- 3 Write utility methods for creating instances of your custom variant and recognizing its type.

The above steps extend the Variant type so that the standard operators work with your new type and the new Variant type can be cast to other data types. You can further enhance your new Variant type so that it supports properties and methods that you define. When creating a Variant type that supports properties or methods, you use *TInvokeableVariantType* or *TPublishableVariantType* as a base class rather than *TCustomVariantType*.

## Storing a custom variant type's data

---

Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first Word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type.

For example, the `VarConv` unit defines a custom variant type that represents a measurement. The data for this type includes the units (*TConvType*) of measurement, as well as the value (a double). The `VarConv` unit defines its own type to represent such a value:

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  Value: Double;
end;
```

This type is exactly the same size as the *TVarData* record. When working with a custom variant of the new type, the variant (or its *TVarData* record) can be cast to *TConvertVarData*, and the custom Variant type simply works with the *TVarData* record as if it were a *TConvertVarData* type.

**Note** When defining a record that maps onto the *TVarData* record in this way, be sure to define it as a packed record.

If your new custom Variant type needs more than 14 bytes to store its data, you can define a new record type that includes a pointer or object instance. For example, the `VarCmplx` unit uses an instance of the class *TComplexData* to represent the data in a complex-valued variant. It therefore defines a record type the same size as *TVarData* that includes a reference to a *TComplexData* object:

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Object references are actually pointers (two Words), so this type is the same size as the *TVarData* record. As before, a complex custom variant (or its *TVarData* record), can be cast to *TComplexVarData*, and the custom variant type works with the *TVarData* record as if it were a *TComplexVarData* type.

## Creating a class to enable the custom variant type

---

Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of *TCustomVariantType*. This involves overriding the appropriate virtual methods of *TCustomVariantType*.

## Enabling casting

One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts.

There are two methods for you to implement that enable the custom Variant type to perform typecasts: *Cast*, which converts another variant type to your custom variant, and *CastTo*, which converts your custom Variant type to another type of Variant.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

For example, the following *Cast* method, from the *TComplexVariantType* class uses the type *Double* as an intermediate type:

```

procedure TComplexVariantType.Cast (var Dest: TVarData; const Source: TVarData);
var
    LSource, LTemp: TVarData;
begin
    VarDataInit(LSource);
    try
        VarDataCopyNoInd(LSource, Source);
        if VarDataIsStr(LSource) then
            TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
        else
            begin
                VarDataInit(LTemp);
                try
                    VarDataCastTo(LTemp, LSource, varDouble);
                    TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
                finally
                    VarDataClear(LTemp);
                end;
            end;
            Dest.VType := VarType;
        finally
            VarDataClear(LSource);
        end;
    end;

```

In addition to the use of *Double* as an intermediate Variant type, there are a few things to note in this implementation:

- The last step of this method sets the *VType* member of the returned *TVarData* record. This member gives the Variant type code. It is set to the *VarType* property of *TComplexVariantType*, which is the Variant type code assigned to the custom variant.
- The custom variant's data (*Dest*) is typecast from *TVarData* to the record type that is actually used to store its data (*TComplexVarData*). This makes the data easier to work with.
- The method makes a local copy of the source variant rather than working directly with its data. This prevents side effects that may affect the source data.

When casting from a complex variant to another type, the *CastTo* method also uses an intermediate type of *Double* (for any destination type other than a string):

```

procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
const AVarType: TVarType);
var
    LTemp: TVarData;
begin
    if Source.VType = VarType then
        case AVarType of
            varOleStr:
                VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
            varString:
                VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
        else
            VarDataInit(LTemp);
            try
                LTemp.VType := varDouble;
                LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
                VarDataCastTo(Dest, LTemp, AVarType);
            finally
                VarDataClear(LTemp);
            end;
        end
    else
        RaiseCastError;
    end;

```

Note that the *CastTo* method includes a case where the source variant data does not have a type code that matches the *VarType* property. This case only occurs for empty (unassigned) source variants.

## Implementing binary operations

To allow the custom variant type to work with standard binary operators (+, -, \*, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the *BinaryOp* method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand.

For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the *VarCmplx* unit:

```

procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
const Operator: TVarOp);
begin
    if Right.VType = VarType then
        case Left.VType of
            varString:
                case Operator of
                    opAdd: Variant(Left) := Variant(Left) + TComplexVarData(Right).VComplex.AsString;
                else
                    RaiseInvalidOp;
                end;
        end;

```

```

else
  if Left.VType = VarType then
    case Operator of
      opAdd:
        TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
      opSubtract:
        TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
      opMultiply:
        TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
      opDivide:
        TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
      else
        RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
  end
else
  RaiseInvalidOp;
end;

```

There are several things to note in this implementation:

This method only handles the case where the variant on the right side of the operator is a custom variant that represents a complex number. If the left-hand operand is a complex variant and the right-hand operand is not, the complex variant forces the right-hand operand first to be cast to a complex variant. It does this by overriding the *RightPromotion* method so that it always requires the type in the *VarType* property:

```

function TComplexVariantType.RightPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { Complex Op TypeX }
  RequiredVarType := VarType;
  Result := True;
end;

```

The addition operator is implemented for a string and a complex number (by casting the complex value to a string and concatenating), and the addition, subtraction, multiplication, and division operators are implemented for two complex numbers using the methods of the *TComplexData* object that is stored in the complex variant's data. This is accessed by casting the *TVarData* record to a *TComplexVarData* record and using its *VComplex* member.

Attempting any other operator or combination of types causes the method to call the *RaiseInvalidOp* method, which causes a runtime error. The *TCustomVariantType* class includes a number of utility methods such as *RaiseInvalidOp* that can be used in the implementation of custom variant types.

*BinaryOp* only deals with a limited number of types: strings and other complex variants. It is possible, however, to perform operations between complex numbers and other numeric types. For the *BinaryOp* method to work, the operands must be cast to complex variants before the values are passed to this method. We have already seen (above) how to use the *RightPromotion* method to force the right-hand operand to be a complex variant if the left-hand operand is complex. A similar



method, *LeftPromotion*, forces a cast of the left-hand operand when the right-hand operand is complex:

```
function TComplexVariantType.LeftPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { TypeX Op Complex }
  if (Operator = opAdd) and VarDataIsStr(V) then
    RequiredVarType := varString
  else
    RequiredVarType := VarType;
  Result := True;
end;
```

This *LeftPromotion* method forces the left-hand operand to be cast to another complex variant, unless it is a string and the operation is addition, in which case *LeftPromotion* allows the operand to remain a string.

## Implementing comparison operations

There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method.

The *Compare* method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the *VarConv* unit implements the following *Compare* method:

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
  var Relationship: TVarCompareResult);
const
  CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
    (crLessThan, crEqual, crGreaterThan);
var
  LValue: Double;
  LType: TConvType;
  LRelationship: TValueRelationship;
begin
  // supports...
  //   convvar cmp number
  //   Compare the value of convvar and the given number
  //   convvar1 cmp convvar2
  //   Compare after converting convvar2 to convvar1's unit type
  //   The right can also be a string. If the string has unit info then it is
  //   treated like a varConvert else it is treated as a double
  LRelationship := EqualsValue;
  case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
```

```

TConvertVarData(Left).VConvType, LValue, LType)
    else
        RaiseCastError;
varDouble:
    LRelationship := CompareValue(TConvertVarData(Left).VValue, TVarData(Right).VDouble);
else
    if Left.VType = VarType then
        LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
            TConvertVarData(Left).VConvType, TConvertVarData(Right).VValue,
            TConvertVarData(Right).VConvType)
    else
        RaiseInvalidOp;
end;
Relationship := CRelationshipToRelationship[LRelationship];
end;
end;

```

If the custom type does not support the concept of “greater than” or “less than,” only “equal” or “not equal,” however, it is difficult to implement the *Compare* method, because *Compare* must return *crLessThan*, *crEqual*, or *crGreaterThan*. When the only valid response is “not equal,” it is impossible to know whether to return *crLessThan* or *crGreaterThan*. Thus, for types that do not support the concept of ordering, you can override the *CompareOp* method instead.

*CompareOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the comparison operator. Implement this method to perform the operation and return a boolean that indicates whether the comparison is *True*. You can then call the *RaiseInvalidOp* method when the comparison makes no sense.

For example, the following *CompareOp* method comes from the *TComplexVariantType* object in the *VarCmplx* unit. It supports only a test of equality or inequality:

```

function TComplexVariantType.CompareOp(const Left, Right: TVarData;
    const Operator: Integer): Boolean;
begin
    Result := False;
    if (Left.VType = VarType) and (Right.VType = VarType) then
        case Operator of
            opCmpEQ:
                Result := TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
            opCmpNE:
                Result := not TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
        else
            RaiseInvalidOp;
        end
    else
        RaiseInvalidOp;
    end;
end;

```

Note that the types of operands that both these implementations support are very limited. As with binary operations, you can use the *RightPromotion* and *LeftPromotion* methods to limit the cases you must consider by forcing a cast before *Compare* or *CompareOp* is called.

## Implementing unary operations

To allow the custom variant type to work with standard unary operators (`-`, `not`), you must override the *UnaryOp* method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand.

For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the *VarCmplx* unit:

```

procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Operator of
      opNegate:
        TComplexVarData(Right).VComplex.DoNegate;
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
  end;

```

Note that for the logical **not** operator, which does not make sense for complex values, this method calls *RaiseInvalidOp* to cause a runtime error.

## Copying and clearing custom variants

In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type.

To indicate how to copy the variant's value, implement the *Copy* method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant's value:

```

procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
  const Indirect: Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
    with TComplexVarData(Dest) do
      begin
        VType := VarType;
        VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
      end;
    end;

```

**Note** The *Indirect* parameter in the *Copy* method signals that the copy must take into account the case when the variant holds only an indirect reference to its data.

**Tip** If your custom variant type does not allocate any memory to hold its data (if the data fits entirely in the *TVarData* record), your implementation of the *Copy* method can simply call the *SimplisticCopy* method.

To indicate how to clear the variant's value, implement the *Clear* method. As with the *Copy* method, the only tricky thing about doing this is ensuring that you free any resources allocated to store the variant's data:

```

procedure TComplexVariantType.Clear(var V: TVarData);
begin
    V.VType := varEmpty;
    FreeAndNil(TComplexVarData(V).VComplex);
end;

```

You will also need to implement the *IsClear* method. This way, you can detect any invalid values or special values that represent "blank" data:

```

function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
    Result := (TComplexVarData(V).VComplex = nil) or
              TComplexVarData(V).VComplex.IsZero;
end;

```

## Loading and saving custom variant values

By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the *IVarStreamable* interface from *Classes.pas*.

*IVarStreamable* defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to the stream. For example, *TComplexVariantType*, in the *VarCmplx* unit, implements the *IVarStreamable* methods as follows:

```

procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
    with TReader.Create(Stream, 1024) do
        try
            with TComplexVarData(Dest) do
                begin
                    VComplex := TComplexData.Create;
                    VComplex.Real := ReadFloat;
                    VComplex.Imaginary := ReadFloat;
                end;
            finally
                Free;
            end;
        end;
end;

procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream: TStream);
begin
    with TWriter.Create(Stream, 1024) do
        try
            with TComplexVarData(Source).VComplex do
                begin
                    WriteFloat(Real);
                    WriteFloat(Imaginary);
                end;
        end;
end;

```

```

    end;
  finally
    Free;
  end;
end;

```

Note how these methods create a Reader or Writer object for the *Stream* parameter to handle the details of reading or writing values.

## Using the *TCustomVariantType* descendant

In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the *VarCmplx* unit:

```

initialization
  ComplexVariantType := TComplexVariantType.Create;

```

In the finalization section of the unit that defines your *TCustomVariantType* descendant, free the instance of your class. This automatically unregisters the variant type. Here is the finalization section of the *VarCmplx* unit:

```

finalization
  FreeAndNil(ComplexVariantType);

```

## Writing utilities to work with a custom variant type

---

Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be.

For example, without a utility function, the only way to create an instance of your custom variant type is to use the global *VarCast* procedure on a source variant of another type. It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out the structure you defined to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:

```

function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
  TComplexVarData(Result).VType := ComplexVariantType.VarType;
  TComplexVarData(ADest).VComplex := TComplexData.Create(AReal, AImaginary);
end;

```

This function does not actually exist in the *VarCmplx* unit, but is a synthesis of methods that do exist, provided to simplify the example. Note that the returned variant is cast to the record that was defined to map onto the *TVarData* structure (*TComplexVarData*), and then filled out.

Another useful utility to create is one that returns the variant type code for your new Variant type. This type code is not a constant. It is automatically generated when you instantiate your *TCustomVariantType* descendant. It is therefore useful to provide a

way to easily determine the type code for your custom variant type. The following function from the `VarCmplx` unit illustrates how to write one, by simply returning the *VarType* property of the *TCustomVariantType* descendant:

```
function VarComplex: TVarType;
begin
    Result := ComplexVariantType.VarType;
end;
```

Two other standard utilities provided for most custom variants check whether a given variant is of the custom type and cast an arbitrary variant to the new custom type. Here is the implementation of those utilities from the `VarCmplx` unit:

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
    Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;

function VarAsComplex(const AValue: Variant): Variant;
begin
    if not VarIsComplex(AValue) then
        VarCast(Result, AValue, VarComplex)
    else
        Result := AValue;
end;
```

Note that these use standard features of all variants: the *VType* member of the *TVarData* record and the *VarCast* function, which works because of the methods implemented in the *TCustomVariantType* descendant for casting data.

In addition to the standard utilities mentioned above, you can write any number of utilities specific to your new custom variant type. For example, the `VarCmplx` unit defines a large number of functions that implement mathematical operations on complex-valued variants.

## Supporting properties and methods in custom variants

---

Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.

### Using *TInvokeableVariantType*

To provide support for properties and methods, the class you create to enable the new custom variant type should descend from *TInvokeableVariantType* instead of directly from *TCustomVariantType*.

*TInvokeableVariantType* defines four methods:

- *DoFunction*
- *DoProcedure*
- *GetProperty*
- *SetProperty*

that you can implement to support properties and methods on your custom variant type.

For example, the `VarConv` unit uses `TInvokeableVariantType` as the base class for `TConvertVariantType` so that the resulting custom variants can support properties. The following example shows the property getter for these properties:

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
    const V: TVarData; const Name: String): Boolean;
var
    LType: TConvType;
begin
    // supports...
    // 'Value'
    // 'Type'
    // 'TypeName'
    // 'Family'
    // 'FamilyName'
    // 'As[Type]'
    Result := True;
    if Name = 'VALUE' then
        Variant(Dest) := TConvertVarData(V).WValue
    else if Name = 'TYPE' then
        Variant(Dest) := TConvertVarData(V).VConvType
    else if Name = 'TYPENAME' then
        Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
    else if Name = 'FAMILY' then
        Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
    else if Name = 'FAMILYNAME' then
        Variant(Dest) := ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
    else if System.Copy(Name, 1, 2) = 'AS' then
        begin
            if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
                System.Copy(Name, 3, MaxInt), LType) then
                VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).WValue,
                    TConvertVarData(V).VConvType, LType), LType)
            else
                Result := False;
        end
    else
        Result := False;
end;
```

The `GetProperty` method checks the `Name` parameter to determine what property is wanted. It then retrieves the information from the `TVarData` record of the `Variant (V)`, and returns it as a `Variant (Dest)`. Note that this method supports properties whose names are dynamically generated at runtime (`As[Type]`), based on the current value of the custom variant.

Similarly, the `SetProperty`, `DoFunction`, and `DoProcedure` methods are sufficiently generic that you can dynamically generate method names, or respond to variable numbers and types of parameters.

## Using `TPublishableVariantType`

If the custom variant type stores its data using an object instance, then there is an easier way to implement properties, as long as they are also properties of the object that represents the variant's data. If you use `TPublishableVariantType` as the base class for your custom variant type, then you need only implement the `GetInstance` method, and all the published properties of the object that represents the variant's data are automatically implemented for the custom variants.

For example, as was seen in “Storing a custom variant type's data” on page 4-28, `TComplexVariantType` stores the data of a complex-valued variant using an instance of `TComplexData`. `TComplexData` has a number of published properties (*Real*, *Imaginary*, *Radius*, *Theta*, and *FixedTheta*), that provide information about the complex value. `TComplexVariantType` descends from `TPublishableVariantType`, and implements the `GetInstance` method to return the `TComplexData` object (in `TypeInfo.pas`) that is stored in a complex-valued variant's `TVarData` record:

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
    Result := TComplexVarData(V).VComplex;
end;
```

`TPublishableVariantType` does the rest. It overrides the `GetProperty` and `SetProperty` methods to use the runtime type information (RTTI) of the `TComplexData` object for getting and setting property values.

**Note** For `TPublishableVariantType` to work, the object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the `{$M+}` compiler directive, or descend from `TPersistent`.

## Working with strings

---

Delphi has a number of different character and string types that have been introduced throughout the development of the Object Pascal language. This section is an overview of these types, their purpose, and usage. For language details, see the Object Pascal Language online Help on String types.

### Character types

---

Delphi has three character types: `Char`, `AnsiChar`, and `WideChar`.

The `Char` character type came from standard Pascal, and was used in Turbo Pascal and then in Object Pascal. Later Object Pascal added `AnsiChar` and `WideChar` as specific character types that were used to support standards for character representation on the Windows operating system. `AnsiChar` was introduced to support an 8-bit character ANSI standard, and `WideChar` was introduced to support a 16-bit Unicode standard. The name `WideChar` is used because Unicode characters are also known as wide characters. Wide characters are two bytes instead of one, so that the character set can represent many more different characters. When `AnsiChar` and `WideChar` were implemented, `Char` became the default character type representing



the currently recommended implementation. If you use *Char* in your application, remember that its implementation is subject to change in future versions of Delphi.

**Note** For cross-platform programming: The Linux `wchar_t` widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal widechars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal widechar data must be widened to 32 bits per character before it can be passed to an OS function as `wchar_t`.

The following table summarizes these character types:

**Table 4.2** Object Pascal character types

Type	Bytes	Contents	Purpose
Char	1	A single character	Default character type
AnsiChar	1	A single character	8-bit characters
WideChar	2	A single Unicode character	16-bit Unicode standard.

For more information about using these character types, see the *Object Pascal Language Guide* online Help on Character types. For more information about Unicode characters, see the *Object Pascal Language Guide* online Help on About extended character sets.

## String types

Delphi has three categories of types that you can use when working with strings:

- Character pointers
- String types
- String classes

This section summarizes string types, and discusses using them with character pointers. For information about using string classes, see the online Help on TStrings.

Delphi has three string implementations: short strings, long strings, and wide strings. Several different string types represent these implementations. In addition, there is a reserved word **string** that defaults to the currently recommended string implementation.

### Short strings

**String** was the first string type used in Turbo Pascal. **String** was originally implemented as a short string. Short strings are an allocation of between 1 and 256 bytes, of which the first byte contains the length of the string and the remaining bytes contain the characters in the string:

```
S: string[0..n]// the original string type
```

When long strings were implemented, **string** was changed to a long string implementation by default and *ShortString* was introduced as a backward compatibility type. *ShortString* is a predefined type for a maximum length string:

```
S: string[255]// the ShortString type
```

The size of the memory allocated for a *ShortString* is static, meaning that it is determined at compile time. However, the location of the memory for the *ShortString* can be dynamically allocated, for example if you use a *PShortString*, which is a pointer to a *ShortString*. The number of bytes of storage occupied by a short string type variable is the maximum length of the short string type plus one. For the *ShortString* predefined type the size is 256 bytes.

Both short strings, declared using the syntax **string**[0..n], and the *ShortString* predefined type exist primarily for backward compatibility with earlier versions of Delphi and Borland Pascal.

A compiler directive, \$H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {\$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {\$H-} directive. The {\$H-} state is mostly useful for using code from versions of Object Pascal that used short strings by default. However, short strings can be useful in data structures where you need a fixed-size component or in DLLs when you don't want to use the *ShareMem* unit (see also the online Help on Memory Management). You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to **string**[255] or *ShortString*, which are unambiguous and independent of the \$H setting.

For details about short strings and the *ShortString* type, see the *Object Pascal Language Guide* online Help on Short strings.

## Long strings

Long strings are dynamically allocated strings with a maximum length of 2 Gigabytes, but the practical limit is usually dependent on the amount of available memory. Like short strings, long strings use 8-bit Ansi characters and have a length indicator. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the *Length* standard function, and to set the length of a long string you must use the *SetLength* standard procedure. Long strings are also reference-counted and, like *PChars*, long strings are null-terminated. For details about the implementation of long strings, see the *Object Pascal Language Guide* online Help on Long strings.

Long strings are denoted by the reserved word **string** and by the predefined identifier *AnsiString*. For new applications, it is recommended that you use the long string type. All components in the VCL are compiled in this state, typically using **string**. If you write components, they should also use long strings, as should any code that receives data from string-type properties. If you want to write specific code that always uses a long string, then you should use *AnsiString*. If you want to write flexible code that allows you to easily change the type as new string implementations become standard, then you should use **string**.

## WideString

The *WideChar* type allows wide character strings to be represented as arrays of *WideChars*. Wide strings are strings composed of 16-bit Unicode characters. As with long strings, wide strings are dynamically allocated with a maximum length of two Gigabytes, but the practical limit is usually dependent on the amount of available

memory. In Delphi, wide strings are not reference-counted. Every assignment of a wide string to a wide string var creates a copy of the string data. In Kylix, `WideStrings` are reference counted.

The dynamically allocated memory that contains the string is deallocated when the wide string goes out of scope. In all other respects wide strings possess the same attributes as long strings. The *WideString* type is denoted by the predefined identifier *WideString*.

Since the 32-bit version of OLE (Windows only) uses Unicode for all strings, strings must be of wide string type in any OLE automated properties and method parameters. Also, most OLE API functions use null-terminated wide strings.

For more information, see the *Object Pascal Language Guide* topic on `WideString`.

## PChar types

A *PChar* is a pointer to a null-terminated string of characters of the type *Char*. Each of the three character types also has a built-in pointer type:

- A *PChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PAnsiChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PWideChar* is a pointer to a null-terminated string of 16-bit characters.

*PChars* are, with short strings, one of the original Object Pascal string types. They were created primarily as a C language and Windows API compatibility type.

## OpenString

An *OpenString* is obsolete, but you may see it in older code. It is for 16-bit compatibility and is allowed only in parameters. *OpenString* was used, before long strings were implemented, to allow a short string of an unspecified length string to be passed as a parameter. For example, this declaration:

```
procedure a(v : openstring);
```

will allow any length string to be passed as a parameter, where normally the string length of the formal and actual parameters must match exactly. You should not have to use *OpenString* in any new applications you write.

Refer also to the `{$P+/-}` switch in “Compiler directives for strings” on page 4-49.

## Runtime library string handling routines

---

The runtime library provides many specialized string handling routines specific to a string type. These are routines for wide strings, long strings, and null-terminated strings (meaning *PChars*). Routines that deal with *PChar* types use the null-termination to determine the length of the string. For more details about null-terminated strings, see Working with null-terminated strings in the *Object Pascal Language Guide* or online Help.

The runtime library also includes a category of string formatting routines. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the *Low* and *High* standard functions.

Because wide strings and long strings are the commonly used types, the remaining sections discuss these routines.

## Wide character routines

When working with strings you should make sure that the code in your application can handle the strings it will encounter in the various target locales. Sometimes you will need to use wide characters and wide strings. In fact, one approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. The runtime library includes the following wide character string functions for converting between standard single-byte character strings (or MBCS strings) and Unicode strings:

- `StringToWideChar`
- `WideCharLenToString`
- `WideCharLenToStrVar`
- `WideCharToString`
- `WideCharToStrVar`

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

A disadvantage of working with wide characters is that Windows 95 does not support wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require tremendous amounts of extra code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

**Note** Typically, CLX components represent string values as wide strings.

## Commonly used long string routines

The long string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

Where appropriate, the tables also provide columns indicating whether or not a routine satisfies the following criteria.

- **Uses case sensitivity:** If locale settings are used, it determines the definition of case. If the routine does not use locale settings, analyses are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper and lower case characters that is determined by a predefined pattern.

- Uses locale settings: Locale settings allow you to customize your application for specific locales, in particular, for Asian language environments. Most locale settings consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Routines that use the Windows locale are typically prefaced with *Ansi* (that is, *AnsiXXX*).
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented as a mix of one- and two-byte character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS are written parse one- and two-byte characters.

*ByteType* and *StrByteType* determine whether a particular byte is the lead byte of a two-byte character. Be careful when using multi-byte characters not to truncate a string by cutting a two-byte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a character or string. For more information about MBCS, see “Enabling application code” on page 12-2 of Chapter 12, “Creating international applications.”

**Table 4.3** String comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
<i>AnsiCompareStr</i>	yes	yes	yes
<i>AnsiCompareText</i>	no	yes	yes
<i>AnsiCompareFileName</i>	no	yes	yes
<i>CompareStr</i>	yes	no	no
<i>CompareText</i>	no	no	no

**Table 4.4** Case conversion routines

Routine	Uses locale settings	Supports MBCS
<i>AnsiLowerCase</i>	yes	yes
<i>AnsiLowerCaseFileName</i>	yes	yes
<i>AnsiUpperCaseFileName</i>	yes	yes
<i>AnsiUpperCase</i>	yes	yes
<i>LowerCase</i>	no	no
<i>UpperCase</i>	no	no

The routines used for string file names: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the Windows locale. You should always use file names that are portable because the locale (character set) used for file names can and might differ from the default user interface.

**Table 4.5** String modification routines

Routine	Case-sensitive	Supports MBCS
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
StringReplace	optional by flag	yes
Trim	NA	yes
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

**Table 4.6** Sub-string routines

Routine	Case-sensitive	Supports MBCS
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes
LastDelimiter	yes	yes
QuotedStr	no	no

**Table 4.7** String handling routines

Routine	Case-sensitive	Supports MBCS
AnsiContainsText	no	yes
AnsiEndsText	no	no
AnsiIndexText	no	yes
AnsiMatchText	no	yes
AnsiResemblesText	no	no
AnsiStartsText	no	yes
IfThen	NA	yes
LeftStr	yes	no
RightStr	yes	no
SoundEx	NA	no
SoundExInt	NA	no
DecodeSoundExInt	NA	no
SoundExWord	NA	no
DecodeSoundExWord	NA	no
SoundExSimilar	NA	no
SoundExCompare	NA	no

## Declaring and initializing strings

---

When you declare a long string:

```
S: string;
```

you do not need to initialize it. Long strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
S = EmptyStr;
```

or test against an empty string:

```
S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access **nil** and will result in an access violation:

```
var
  S: string;
begin
  S[i]; // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a **nil** pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle **nil**:

```
var
  S: string; // empty string
begin
  proc(PChar(S)); // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
S := 'No longer nil';
proc(PChar(S)); // proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
SetLength(S, 100); // sets the dynamic length of S to 100
proc(PChar(S)); // proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, *S* is guaranteed to reference a unique string, that is a string with a reference count of one. To obtain the length of a string, use the *Length* function.

Remember when declaring a **string** that:

```
S: string[n];
```

implicitly declares a short string, not a long string of *n* length. To declare a long string of specifically *n* length, declare a variable of type **string** and use the *SetLength* procedure.

```
S: string;
SetLength(S, n);
```

## Mixing and converting string types

---

Short, long, and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

Additional functions (*CopyQStringListToTstrings*, *CopyTStringsToQStringList*, *QStringListToTStringList*) are provided for converting underlying Qt string types and CLX string types. These functions are located in *Qtypes.pas*.

## String to PChar conversions

---

Long string to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.
- Long strings are null-terminated and also contain the length of the string, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in this section.

## String dependencies

Sometimes you will need convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. For example:

```
procedure my_func(x: string);
begin
    // do something with x
    some_proc(PChar(x)); // cast the string to a PChar
    // you now need to guarantee that the string remains
    // as long as the some_proc procedure needs to use it
end;
```



## Returning a PChar local variable

A common error when working with *PChars* is to store in a data structure, or return as a value, a local variable. When your routine ends, the *PChar* will disappear because it is simply a pointer to memory, and is not a reference counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

## Passing a local variable as a PChar

Consider that you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:

```
// VCL version
// assume MAXSIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := GetModuleFilename(0, @buf, SizeOf(buf)); // treats @buf as a PChar
  S := buf;
  //statements
end;
```

Or, for cross-platform programs, the code is nearly identical:

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
  . . .
end;

// assume MAX_SIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, @buf, SizeOf(buf)); // treats @buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an **array of char** and a **string** is automatic. When *GetModuleFilename* (or *FillBuffer* in the cross-platform version) returns, the *Length* of the string correctly indicates the number of bytes written to *buf*.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an **array of char** to a **string**. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
  SetLength(S, MAX_SIZE; // when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // statements
end;
```

## Compiler directives for strings

---

The following compiler directives affect character and string types.

**Table 4.8** Compiler directives for strings

Directive	Description
{H+/-}	A compiler directive, \$H, controls whether the reserved word <b>string</b> represents a short string or a long string. In the default state, {H+}, <b>string</b> represents a long string. You can change it to a <i>ShortString</i> by using the {H-} directive.
{P+/-}	The \$P directive is meaningful only for code compiled in the {H-} state, and is provided for backwards compatibility. \$P controls the meaning of variable parameters declared using the string keyword in the {H-} state.  In the {\$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {\$P+} state, they are open string parameters. Regardless of the setting of the \$P directive, the <i>OpenString</i> identifier can always be used to declare open string parameters.
{V+/-}	The \$V directive controls type checking on short strings passed as variable parameters. In the {\$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types.  In the {\$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:  <pre>var S: string[3];  procedure Test(var T: string); begin   T := '1234'; end;  begin   Test(S); end.</pre>
{X+/-}	The {\$X+} compiler directive enables support for null-terminated strings by activating the special rules that apply to the built-in <i>PChar</i> type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with <i>Write</i> , <i>WriteLn</i> , <i>Val</i> , <i>Assign</i> , and <i>Rename</i> from the System unit.)

## Strings and characters: related topics

---

The following *Object Pascal Language Guide* topics discuss strings and character sets. Also see Chapter 12, “Creating international applications.”

- About extended character sets (Discusses international character sets)
- Working with null-terminated strings (Contains information about character arrays)
- Character strings
- Character pointers
- String operators

## Working with files

---

This section describes working with files and distinguishes between manipulating files on disk, and input/output operations such as reading and writing to files. The first section discusses the runtime library and Windows API routines you would use for common programming tasks that involve manipulating files on disk. The next section is an overview of file types used with file I/O. The last section focuses on the recommended approach to working with file I/O, which is to use file streams.

Although the Object Pascal language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications.

**Note** Previous versions of the Object Pascal language performed operations on files themselves, rather than on the filename parameters commonly used now. With these file types you had to locate a file and assign it to a file variable before you could, for example, rename the file.

## Manipulating files

---

Several common file operations are built into Object Pascal's runtime library. The procedures and functions for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead. Object Pascal provides routines for most file manipulation. When it does not, alternative routines are discussed.

**Caution** Although the Object Pascal language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications.

### Deleting a file

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm deletions of files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

*DeleteFile* returns *True* if it deleted the file and *False* if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

## Finding a file

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminate a *FindFirst/FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns *True* if the file exists, *False* otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. The declaration for *TSearchRec* is:

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer; //Time contains the time stamp of the file.
    Size: Integer; //Size contains the size of the file in bytes.
    Attr: Integer; //Attr represents the file attributes of the file.
    Name: TFileName; //Name contains the filename and extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData; //FindData contains additional information such as
    //file creation time, last access time, long and short filenames.
  end;
```

If a file is found, the fields of the *TSearchRec* type parameter are modified to describe the found file. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

**Table 4.9** Attribute constants and values

Constant	Value	Description
faReadOnly	\$00000001	Read-only files
faHidden	\$00000002	Hidden files
faSysFile	\$00000004	System files
faVolumeID	\$00000008	Volume ID files
faDirectory	\$00000010	Directory files
faArchive	\$00000020	Archive files
faAnyFile	\$0000003F	Any file

To test for an attribute, combine the value of the *Attr* field with the attribute constant with the **and** operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*: (*SearchRec.Attr* and *faHidden* > 0). Attributes can be combined by OR'ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass (*faReadOnly* or *faHidden*) the *Attr* parameter.

**Example:** This example uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption. Each time the user clicks the *Again* button, the next matching filename and size is displayed in the label:

```

var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\delphi6\bin\*..*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
end;

procedure TForm1.AgainClick(Sender: TObject);
begin
  if (FindNext(SearchRec) = 0)
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
  else
    FindClose(SearchRec);
end;

```

In cross-platform applications, you should replace any hardcoded pathnames such as `c:\Program Files\delphi6\bin\*..*` with the correct pathname for the system or use environment variables (on the Environment Variables page when you choose Tools | Environment Options) to represent them.

## Renaming a file

To change a filename, simply use the *RenameFile* function:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

which changes a filename, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file, for example, if a file called *NewFileName* already exists, it returns *False*. For example:

```

if not RenameFile('OLDNAME.TXT', 'NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');

```

You cannot rename (move) a file across drives using *RenameFile*. You would need to first copy the file and then delete the old one.

**Note** *RenameFile* in the VCL is a wrapper around the Windows API *MoveFile* function, so *MoveFile* will not work across drives either.

## File date-time routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or -1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, take a *Handle* type as a parameter. To get access to a Windows file *Handle* either

- Call the Windows API *CreateFile* function. *CreateFile* is a 32-bit only function that creates or opens a file and returns a *Handle* that can be used to access the file.
- Instantiate *TFileStream* to create or open a file. Then use the *Handle* property as you would a Windows' file *Handle*. See "Using file streams" on page 4-54 for more information.

## Copying a file

The runtime library does not provide any routines for copying a file. However, if you are writing Windows-only applications, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the Delphi runtime library file routines, *CopyFile* takes a filename as a parameter, not a *Handle*. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the Delphi *RenameFile* function nor the Windows API *MoveFile* function can rename/move files across drives. For more information, see the Microsoft Windows online Help.

## File types with file I/O

---

You can use three file types when working with file I/O: Pascal file types, file handles, and file stream objects. The following table summarizes these types.

**Table 4.10** File types for file I/O

File type	Description
Pascal file types	In the System unit. These types are used with file variables, usually of the format "F: Text:" or "F: File". The files have three types: typed, text, and untyped. A number of file-handling routines, such as <i>AssignPrn</i> and <i>writeln</i> , use them. These file types are obsolete and are incompatible with Windows file handles. If you need to work with them, see the <i>Object Pascal Language Guide</i> .
File handles	In the Sysutils unit. A number of routines use a handle to identify the file. You get the handle when you open or create the file (for example, using <i>FileOpen</i> or <i>FileCreate</i> ). Once you have the handle, there are routines to work with the contents of the file given its handle (write a line, read text, and so on).  In Windows programming, the Object Pascal file handles are wrappers for the Windows file handle type. The runtime library file-handling routines that use Windows file Handles are typically wrappers around Windows API functions. For example, the <i>FileRead</i> calls the Windows <i>ReadFile</i> function. Because the Delphi functions use Object Pascal syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Using these routines is straightforward, and if you are familiar and comfortable with the Windows API file routines, you may want to use them when working with file I/O.
File streams	File streams are object instances of the <i>TFileStream</i> class used to access information in disk files. File streams are a portable and high-level approach to file I/O. <i>TFileStream</i> has a <i>Handle</i> property that lets you access the file handle. The next section discusses <i>TFileStream</i> .

## Using file streams

---

*TFileStream* is a class that enables applications to read from and write to a file on disk. It is used for high-level object representations of file streams. *TFileStream* offers multiple functionality: persistence, interaction with other streams, and file I/O.

- *TFileStream* is a descendant of the stream classes. As such, one advantage of using file streams is that they inherit the ability to persistently store component properties. The stream classes work with the *TFile* classes, *TReader*, and *TWriter*, to stream objects out to disk. Therefore, when you have a file stream, you can use that same code for the component streaming mechanism. For more information about using the component streaming system, see the online Help on the *TStream*, *TFile*, *TReader*, *TWriter*, and *TComponent* classes.
- *TFileStream* can interact easily with other stream classes. For example, if you want to dump a dynamic memory block to disk, you can do so using a *TFileStream* and a *TMemoryStream*.
- *TFileStream* provides the basic methods and properties for file I/O. The remaining sections focus on this aspect of file streams

### Creating and opening files

To create or open a file and get access to a handle for the file, you simply instantiate a *TFileStream*. This opens or creates a named file and provides methods to read from or write to it. If the file cannot be opened, *TFileStream* raises an exception.

```
constructor Create(const filename: string; Mode: Word);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode or'ed together. The open mode must be one of the following values:

**Table 4.11** Open modes

Value	Meaning
fmCreate	TFileStream a file with the given name. If a file with the given name exists, open the file in write mode.
fmOpenRead	Open the file for reading only.
fmOpenWrite	Open the file for writing only. Writing to the file completely replaces the current contents.
fmOpenReadWrite	Open the file to modify the current contents rather than replace them.

The share mode can be one of the following values with the restrictions listed below:

**Table 4.12** Shared modes

Value	Meaning
fmShareCompat	Sharing is compatible with the way FCBs are opened.
fmShareExclusive	Other applications can not open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.
fmShareDenyRead	Other applications can open the file for writing but not for reading.
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

Note that which share mode you can use depends on which open mode you used. The following table shows shared modes that are available for each open mode.

**Table 4.13** Shared modes available for each open mode

Open Mode	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	Can't use	Can't use	Available	Can't use	Available
fmOpenWrite	Available	Available	Can't use	Available	Available
fmOpenReadWrite	Available	Available	Available	Available	Available

The file open and share mode constants are defined in the SysUtils unit.

## Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. *Handle* is read-only and indicates the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a window's file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a window's file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

## Reading and writing to files

*TFileStream* has several different methods for reading from and writing to files. These are distinguished by whether they perform the following:

- Return the number of bytes read or written.
- Require the number of bytes is known.
- Raise an exception on error.

*Read* is a function that reads up to *Count* bytes from the file associated with the file stream, starting at the current *Position*, into *Buffer*. *Read* then advances the current position in the file by the number of bytes actually transferred. The prototype for *Read* is

```
function Read(var Buffer; Count: Longint): Longint; override;
```

*Read* is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the end of file marker is encountered.

*Write* is a function that writes *Count* bytes from the *Buffer* to the file associated with the file stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint; override;
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered.



The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception on error (*EReadError* and *EWriteError*) while the *Read* and *Write* methods do not. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

These methods call the *Read* and *Write* methods, to perform the actual reading and writing.

## Reading and writing strings

If you are passing a string to a read or write function, you need to be aware of the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in only the pointer element. What you need to do is first cast the string to a *Pointer* or *PChar*, and then dereference it. For example:

```
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s)); // this will give you garbage
  fs.Write(PChar(s)^, Length(s)); // this is the correct way
end;
```

## Seeking a file

Most typical file I/O mechanisms have a process of seeking a file in order to read from or write to a particular location within it. For this purpose, *TFileStream* has a *Seek* method. The prototype for *Seek* is:

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

Value	Meaning
soFromBeginning	Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be $\geq 0$ .
soFromCurrent	Offset is from the current position in the resource. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the resource. Offset must be $\leq 0$ to indicate a number of bytes before the end of the file.

*Seek* resets the current *Position* of the stream, moving it by the indicated offset. *Seek* returns the new value of the *Position* property, the new current position in the resource.

## File position and size

*TFileStream* has properties that hold the current position and size of the file. These are used by the *Seek*, *read*, and *write* methods.

The *Position* property of *TFileStream* is used to indicate the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Longint;
```

The *Size* property indicates the size in bytes of the stream. It is used as an end of file marker to truncate the file. The declaration for *Size* is:

```
property Size: Longint;
```

*Size* is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the file. If the *Size* of the file cannot be changed, an exception is raised. For example, trying to change the *Size* of a file that was opened in *fmOpenRead* mode raises an exception.

## Copying

*CopyFrom* copies a specified number of bytes from one (file) stream to another.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Using *CopyFrom* eliminates the need to create, read into, write from, and free a buffer when copying data.

*CopyFrom* copies *Count* bytes from *Source* into the stream. *CopyFrom* then moves the current position by *Count* bytes, and returns the number of bytes copied. If *Count* is 0, *CopyFrom* sets *Source* position to 0 before reading and then copies the entire contents of *Source* into the stream. If *Count* is greater than or less than 0, *CopyFrom* reads from the current position in *Source*.

## Converting measurements

---

The `ConvUtils` unit declares a general-purpose *Convert* function that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*. The units you're converting must be in the same conversion family. For information on doing conversions, see the next section *Performing conversions* and refer to *Convert* in the online Help.

The `StdConvs` unit defines several conversion families and measurement units within each family. In addition, you can create customized conversion families and associated units using the *RegisterConversionType* and *RegisterConversionFamily* functions. For information on extending conversion and conversion units, see the section *Adding new measurement types* and refer to *Convert* in the online Help.

### Performing conversions

---

You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.

#### Performing simple conversions

You can use the *Convert* function to convert a measurement from one set of units to another. The *Convert* function converts between units that measure the same type of thing (distance, area, time, temperature, and so on).

To use *Convert*, you must specify the units from which to convert and to which to convert. You use the *TConvType* type to identify the units of measurement.

For example, this converts a temperature from degrees Fahrenheit to degrees Kelvin:

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

#### Performing complex conversions

You can also use the *Convert* function to perform more complex conversions between the ratio of two measurement types. Examples of when you might need to use this are when converting miles per hour to meters per minute for calculating speed or when converting gallons per minute to liters per hour for calculating flow.

For example, the following call converts miles per gallon to kilometers per liter:

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

The units you're converting must be in the same conversion family (they must measure the same thing). If the units are not compatible, *Convert* raises an *EConversionError* exception. You can check whether two *TConvType* values are in the same conversion family by calling *CompatibleConversionTypes*.

The `StdConvs` unit defines several families of *TConvType* values. See Conversion family variables in the online Help for a list of the predefined families of measurement units and the measurement units in each family.

## Adding new measurement types

---

If you want to perform conversions between measurement units not already defined in the `StdConvs` unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values.

You first need to obtain *TConvFamily* values by registering a conversion family using the *RegisterConversionFamily* function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the `StdConvs` unit), you can use the *RegisterConversionType* function to add the new units to the conversion family. The following examples show how to do this.

For more examples, refer to the source code for the standard conversions unit (`stdconvs.pas`). (Note that the source is not included in all versions of Delphi.)

### Creating a simple conversion family and adding units

One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur.

To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not an exact conversion between days and months, days and years, and so on. Months have different lengths; years have correction factors for leap years, leap seconds, and so on.

If you are only using units of measurement greater than or equal to months, you can create a more accurate conversion family with years as its base unit. This example creates a new conversion family called *cbLongTime*.

### Declare variables

First, you need to declare variables for the identifiers. The identifiers are used in the new `LongTime` conversion family, and the units of measurement that are its members:

```
var
  cbLongTime: TConvFamily;
  ltMonths: TConvType;
  ltYears: TConvType;
  ltDecades: TConvType;
  ltCenturies: TConvType;
  ltMillennia: TConvType;
```

## Register the conversion family

Next, register the conversion family:

```
cbLongTime := RegisterConversionFamily ('Long Times');
```

Although an *UnregisterConversionFamily* procedure is provided, you don't need to unregister conversion families unless the unit that defines them is removed at runtime. They are automatically cleaned up when your application shuts down.

## Register measurement units

Next, you need to register the measurement units within the conversion family that you just created. You use the *RegisterConversionType* function, which registers units of measurement within a specified family. You need to define the base unit which in the example is years, and the other units are defined using a factor that indicates their relation to the base unit. So, the factor for *ltMonths* is 1/12 because the base unit for the LongTime family is years. You also include a description of the units to which you are converting.

The code to register the measurement units is shown here:

```
ltMonths:=RegisterConversionType(cbLongTime,'Months',1/12);  
ltYears:=RegisterConversionType(cbLongTime,'Years',1);  
ltDecades:=RegisterConversionType(cbLongTime,'Decades',10);  
ltCenturies:=RegisterConversionType(cbLongTime,'Centuries',100);  
ltMillennia:=RegisterConversionType(cbLongTime,'Millennia',1000);
```

## Use the new units

You can now use the newly registered units to perform conversions. The global *Convert* function can convert between any of the conversion types that you registered with the *cbLongTime* conversion family.

So instead of using the following *Convert* call,

```
Convert (StrToFloat (Edit1.Text),tuMonths,tuMillennia);
```

you can now use this one for greater accuracy:

```
Convert (StrToFloat (Edit1.Text),ltMonths,ltMillennia);
```

## Using a conversion function

For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins.

This example, which comes from the StdConvs unit, shows how to register a conversion type by providing functions to convert to and from the base units.

**Declare variables**

First, declare variables for the identifiers. The identifiers are used in the *cbTemperature* conversion family, and the units of measurement are its members:

```
var
    cbTemperature: TConvFamily;
    tuCelsius: TConvType;
    tuKelvin: TConvType;
    tuFahrenheit: TConvType;
```

**Note** The units of measurement listed here are a subset of the temperature units actually registered in the *StdConv*s unit.

**Register the conversion family**

Next, register the conversion family:

```
cbTemperature := RegisterConversionFamily ('Temperature');
```

**Register the base unit**

Next, define and register the base unit of the conversion family, which in the example is degrees Celsius. Note that in the case of the base unit, we can use a simple conversion factor, because there is no actual conversion to make:

```
tuCelsius := RegisterConversionType(cbTemperature, 'Celsius', 1);
```

**Write methods to convert to and from the base unit**

You need to write the code that performs the conversion from each temperature scale to and from degrees Celsius, because these do not rely on a simple conversion factor. These functions are taken from the *StdConv*s unit:

```
function FahrenheitToCelsius(const AValue: Double): Double;
begin
    Result := ((AValue - 32) * 5) / 9;
end;

function CelsiusToFahrenheit(const AValue: Double): Double;
begin
    Result := ((AValue * 9) / 5) + 32;
end;

function KelvinToCelsius(const AValue: Double): Double;
begin
    Result := AValue - 273.15;
end;

function CelsiusToKelvin(const AValue: Double): Double;
begin
    Result := AValue + 273.15;
end;
```

**Register the other units**

Now that you have the conversion functions, you can register the other measurement units within the conversion family. You also include a description of the units.

The code to register the other units in the family is shown here:

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
CelsiusToKelvin);
tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit', FahrenheitToCelsius,
CelsiusToFahrenheit);
```

### Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the conversion types that you registered with the *cbTemperature* conversion family. For example the following code converts a value from degrees Fahrenheit to degrees Kelvin.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

### Using a class to manage conversions

You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing.

If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros), you can't use a simple conversion factor approach to properly convert between European currencies for two reasons:

- The conversion must round to a currency-specific number of digits.
- The conversion factor approach uses an inverse factor to the one specified by the standard Euro conversions.

However, this can all be handled by the conversion functions such as the following:

```
function FromEuro(const AValue: Double, Factor, FRound): Double;
begin
    Result := RoundTo(AValue * Factor, FRound);
end;

function ToEuro(const AValue: Double, Factor): Double;
begin
    Result := AValue / Factor;
end;
```

The problem is, this approach requires extra parameters on the conversion function, which means you can't simply register the same function with every European currency. In order to avoid having to write two new conversion functions for every European currency, you can make use of the same two functions by making them the members of a class.

### Creating the conversion class

The class must be a descendant of *TConvTypeFactor*. *TConvTypeFactor* defines two methods, *ToCommon* and *FromCommon*, for converting to and from the base units of a

conversion family (in this case, to and from Euros). Just as with the functions you use directly when registering a conversion unit, these methods have no extra parameters, so you must supply the number of digits to round off and the conversion factor as private members of your conversion class. This is shown in the EuroConv example in the demos\ConvertIt directory (see euroconv.pas):

```

type
  TConvTypeEuroFactor = class(TConvTypeFactor)
  private
    FRound: TRoundToRange;
  public
    constructor Create(const AConvFamily: TConvFamily;
      const ADescription: string; const AFactor: Double;
      const ARound: TRoundToRange);
    function ToCommon(const AValue: Double): Double; override;
    function FromCommon(const AValue: Double): Double; override;
  end;
end;

```

The constructor assigns values to those private members:

```

constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
  const ADescription: string; const AFactor: Double;
  const ARound: TRoundToRange);
begin
  inherited Create(AConvFamily, ADescription, AFactor);
  FRound := ARound;
end;

```

The two conversion functions simply use these private members:

```

function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
  Result := SimpleRoundTo(AValue * Factor, FRound);
end;

function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
begin
  Result := AValue / Factor;
end;

```

## Declare variables

Now that you have a conversion class, begin as with any other conversion family, by declaring identifiers:

```

var
  euEUR: TConvType; { EU euro }
  euBEF: TConvType; { Belgian francs }
  euDEM: TConvType; { German marks }
  euGRD: TConvType; { Greek drachmas }
  euESP: TConvType; { Spanish pesetas }
  euFFR: TConvType; { French francs }
  euIEP: TConvType; { Irish pounds }
  euITL: TConvType; { Italian lire }
  euLUF: TConvType; { Luxembourg francs }
  euNLG: TConvType; { Dutch guilders }

```



## Defining data types

```
euATS: TConvType; { Austrian schillings }
euPTE: TConvType; { Portuguese escudos }
euFIM: TConvType; { Finnish marks }
euUSD: TConvType; { US dollars }
euGBP: TConvType; { British pounds }
euJPY: TConvType; { Japanese yen }
```

### Register the conversion family and the other units

Now you are ready to register the conversion family and the European monetary units, using your new conversion class:

```
cbEuro := RegisterConversionFamily ('European currency');
...
// Euro's various conversion types
euEUR := RegisterEuroConversionType(cbEuro, SEURDescription, EURToEUR, EURSubUnit);
euBEF := RegisterEuroConversionType(cbEuro, SBEFDescription, BEFToEUR, BEFSubUnit);
euDEM := RegisterEuroConversionType(cbEuro, SDEMDescription, DEMToEUR, DEMSubUnit);
euGRD := RegisterEuroConversionType(cbEuro, SGRDDescription, GRDToEUR, GRDSubUnit);
euESP := RegisterEuroConversionType(cbEuro, SESPDescription, ESPToEUR, ESPSubUnit);
euFFR := RegisterEuroConversionType(cbEuro, SFFRDescription, FFRToEUR, FFRSubUnit);
euIEP := RegisterEuroConversionType(cbEuro, SIEPDescription, IEPToEUR, IEPSubUnit);
euITL := RegisterEuroConversionType(cbEuro, SITLDescription, ITLToEUR, ITLSubUnit);
euLUF := RegisterEuroConversionType(cbEuro, SLUFDescription, LUFToEUR, LUFSubUnit);
euNLG := RegisterEuroConversionType(cbEuro, SNLGDescription, NLGToEUR, NLGSubUnit);
euATS := RegisterEuroConversionType(cbEuro, SATSDescription, ATSToEUR, ATSSubUnit);
euPTE := RegisterEuroConversionType(cbEuro, SPTEDescription, PTEToEUR, PTESubUnit);
euFIM := RegisterEuroConversionType(cbEuro, SFIMDescription, FIMToEUR, FIMSubUnit);
euUSD := RegisterEuroConversionType(cbEuro, SUSDDescription,
    ConvertUSDToEUR, ConvertEURToUSD);
euGBP := RegisterEuroConversionType(cbEuro, SGBPDescription,
    ConvertGBPToEUR, ConvertEURToGBP);
euJPY := RegisterEuroConversionType(cbEuro, SJPYDescription,
    ConvertJPYToEUR, ConvertEURToJPY);
```

Note that *RegisterEuroConversionType* is a wrapper function that simplifies the registering of the monetary types. See the example code for details.

### Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the European currencies you have registered with the new *cbEuro* family. For example, the following code converts a value from Italian Lire to German Marks:

```
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
```

## Defining data types

---

Object Pascal has many predefined data types. You can use these predefined types to create new types that meet the specific needs of your application. For an overview of types, see the *Object Pascal Language Guide*.

# Building applications, components, and libraries

This chapter provides an overview of how to use Delphi to create applications, libraries, and components.

## Creating applications

---

The main use of Delphi is designing and building the following types of applications:

- GUI applications
- Console applications
- Service applications (for Windows applications only)
- Packages and DLLs

GUI applications generally have an easy-to-use interface. Console applications run from a console window. Service applications are run as Windows services. These types of applications compile as executables with start-up code.

You can create other types of projects such as packages and DLLs that result in creating packages or dynamically linkable libraries. These applications produce executable code without start-up code. Refer to “Creating packages and DLLs” on page 5-9.

## GUI applications

---

A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can

extend the application by calling DLLs, packages, and other support files from the executable.

Delphi offers two application UI models:

- Single document interface (SDI)
- Multiple document interface (MDI)

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE.

## User interface models

Any form can be implemented as a multiple document interface (MDI) or single document interface (SDI) form. In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors. An SDI application, in contrast, normally contains a single document view. To make your form an SDI application, set the *FormStyle* property of your *Form* object to *fsNormal*.

For more information on developing the UI for an application, see Chapter 6, “Developing the application user interface.”

## SDI applications

To create a new SDI application,

- 1 Select File | New | Other to bring up the New Items dialog.
- 2 Click on the Projects page and select SDI Application.
- 3 Click OK.

By default, the *FormStyle* property of your *Form* object is set to *fsNormal*, so Delphi assumes that all new applications are SDI applications.

## MDI applications

To create a new MDI application,

- 1 Select File | New | Other to bring up the New Items dialog.
- 2 Click on the Projects page and select MDI Application.
- 3 Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIForm*) or main form (*fsMDIChild*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form’s properties.

## Setting IDE, project, and compilation options

Choose Project | Options to specify various options for your project. For more information, see the online Help.

### Setting default project options

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the Default box at the bottom right of the window. All new projects will use the current options selected by default.

## Programming templates

---

Programming templates are commonly used “skeleton” structures that you can add to your source code and then fill in. Some standard code templates such as those for array, class, and function declarations, and many statements, are included with Delphi.

You can also write your own templates for coding structures that you often use. For example, if you want to use a **for** loop in your code, you could insert the following template:

```
for := to do
begin

end;
```

To insert a code template in the Code editor, press *Ctrl-J* and select the template you want to use. You can also add your own templates to this collection. To add a template:

- 1 Select Tools | Editor Options.
- 2 Click the Code Insight tab.
- 3 In the Templates section, click Add.
- 4 Type a name for the template after Shortcut name and enter a brief description of the new template.
- 5 Add the template code to the Code text box.
- 6 Click OK.

## Console applications

---

Console applications are 32-bit programs that run without a graphical interface, usually in a console window. These applications typically don't require much user input and perform a limited set of functions.

To create a new console application,

- 1 Choose File | New | Other and select Console Application from the New Items dialog box.

Delphi then creates a project file for this type of source file and displays the code editor.

**Note** When you create a new console application, the IDE does not create a new form. Only the code editor is displayed.

## Service applications

---

Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A web, FTP, or e-mail server is an example of a service application.

To create an application that implements a Win32 service, Choose File | New, and select Service Application from the New Items page. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.

Once you have created a service application, you will see a window in the designer that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the Object Inspector. You can add additional services to your service application by choosing Service from the new items dialog. Do not add services to an application that is not a service application. While a *TService* object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.

Once your service application is built, you can install its services with the Service Control Manager (SCM). Other applications can then launch your services by sending requests to the SCM.

To install your application's services, run it using the `/INSTALL` option. The application installs its services and exits, giving a confirmation message if the services are successfully installed. You can suppress the confirmation message by running the service application using the `/SILENT` option.

To uninstall the services, run it from the command line using the `/UNINSTALL` option. (You can also use the `/SILENT` option to suppress the confirmation message when uninstalling).

**Example** This service has a *TServerSocket* whose port is set to 80. This is the default port for Web Browsers to make requests to Web Servers and for Web Servers to make responses to Web Browsers. This particular example produces a text document in the C:\Temp directory called WebLogxxx.log (where xxx is the ThreadID). There should be only one Server listening on any given port, so if you have a web server, you should make sure that it is not listening (the service is stopped).

To see the results: open up a web browser on the local machine and for the address, type 'localhost' (with no quotes). The Browser will time out eventually, but you should now have a file called weblogxxx.log in the C:\temp directory.

1 To create the example, choose File | New and select Service Application from the New Items dialog. You will see a window appear named Service1. From the Internet page of the Component palette, add a ServerSocket component to the service window (Service1).

- 2 Next, add a private data member of type *TMemoryStream* to the *TService1* class. The interface section of your unit should now look like this:

```

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;
type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;
var
  Service1: TService1;

```

- 3 Next, select *ServerSocket1*, the component you added in step 1. In the Object Inspector, double click the *OnClientRead* event and add the following event handler:

```

procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;
begin
  Buffer := nil;
while Socket.ReceiveLength > 0 do begin
  Buffer := AllocMem(Socket.ReceiveLength);
  try
    Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
    Stream.Write(Buffer^, StrLen(Buffer));
  finally
    FreeMem(Buffer);
  end;
  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
end;
end;

```

- 4 Finally, select *Service1* by clicking in the window's client area (but not on the *ServiceSocket*). In the Object Inspector, double click the *OnExecute* event and add the following event handler:

```

procedure TService1.Service1Execute(Sender: TService);
begin
  Stream := TMemoryStream.Create;
  try

```

```
ServerSocket1.Port := 80; // WWW port
ServerSocket1.Active := True;

while not Terminated do begin
    ServiceThread.ProcessRequests(True);
end;

ServerSocket1.Active := False;
finally
    Stream.Free;
end;
end;
```

When writing your service application, you should be aware of:

- Service threads
- Service name properties
- Debugging services

### Service threads

Each service has its own thread (*TServiceThread*), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TService.OnExecute* event handler. The service thread has its own *Execute* method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests.

Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread (derived from *TThread*, not *TServiceThread*) for each request and move the implementation of that service to the new thread's *Execute* method. This allows the service thread's *Execute* loop to process new requests continually without having to wait for the service's *OnExecute* handler to finish. The following example demonstrates.

**Example** This service beeps every 500 milliseconds from within the standard thread. It handles pausing, continuing, and stopping of the thread when the service is told to pause, continue, or stop.

- 1 Choose File | New | Other and select Service Application from the New Items dialog. You will see a window appear named Service1.
- 2 In the interface section of your unit, declare a new descendant of *TThread* named *TSparkyThread*. This is the thread that does the work for your service. The declaration should appear as follows:

```
TSparkyThread = class(TThread)
public
    procedure Execute; override;
end;
```

- 3 Next, in the implementation section of your unit, create a global variable for a *TSparkyThread* instance:

```
var
    SparkyThread: TSparkyThread;
```

- 4 Add the following code to the implementation section for the `TSparkyThread` `Execute` method (the thread function):

```
procedure TSparkyThread.Execute;
begin
  while not Terminated do
  begin
    Beep;
    Sleep(500);
  end;
end;
```

- 5 Select the Service window (`Service1`), and double-click the `OnStart` event in the Object Inspector. Add the following `OnStart` event handler:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
```

- 6 Double-click the `OnContinue` event in the Object Inspector. Add the following `OnContinue` event handler:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
  SparkyThread.Resume;
  Continued := True;
end;
```

- 7 Double-click the `OnPause` event in the Object Inspector. Add the following `OnPause` event handler:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
```

- 8 Finally, double-click the `OnStop` event in the Object Inspector and add the following `OnStop` event handler:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

When developing server applications, choosing to spawn a new thread depends on the nature of the service being provided, the anticipated number of connections, and the expected number of processors on the computer running the service.

## Service name properties

The VCL provides classes for creating service applications on the Windows platform (not available for cross-platform applications). These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This section describes the differences.



Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name* property. The following sections summarize the name properties:

### **TDependency properties**

The *TDependency DisplayName* is both a display name and the actual name of the service. It is nearly always the same as the *TDependency Name* property.

### **TService name properties**

The *TService Name* property is inherited from *TComponent*. It is the name of the component, and is also the name of the service. For dependencies that are services, this property is the same as the *TDependency Name* and *DisplayName* properties.

*TService's DisplayName* is the name displayed in the Service Manager window. This often differs from the actual service name (*TService.Name*, *TDependency.DisplayName*, *TDependency.Name*). Note that the *DisplayName* for the Dependency and the *DisplayName* for the Service usually differ.

Service start names are distinct from both the service display names and the actual service names. A *ServiceStartName* is the user name input on the Start dialog selected from the Service Control Manager.

### **Debugging services**

Debugging service applications can be tricky, because it requires short time intervals:

- 1 First, launch the application in the debugger. Wait a few seconds until it has finished loading.

- 2 Quickly start the service from the control panel or from the command line:  
`start MyServ`

You must launch the service quickly (within 15-30 seconds of application startup) because the application will terminate if no service is launched.

Another approach is to attach to the service application process when it is already running. (That is, starting the service first, and then attaching to the debugger). To attach to the service application process, choose Run | Attach To Process, and select the service application in the resulting dialog.

In some cases, this second approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

- 1 First create a key called **Image File Execution Options** in the following registry location:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`

- 2 Create a subkey with the same name as your service (for example, MYSERV.EXE). To this subkey, add a value of type REG\_SZ, named Debugger. Use the full path to Delphi32.exe as the string value.

- 3 In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

## Creating packages and DLLs

---

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs. However, on Linux, DLLs (and packages) recompile as shared objects.

Packages are special DLLs used by Delphi applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

The following compiler directives can be placed in library project files:

**Table 5.1** Compiler directives for libraries

Compiler Directive	Description
{\$LIBPREFIX 'string'}	Adds a specified prefix to the output file name. For example, you could specify {\$LIBPREFIX 'dcl'} for a design-time package, or use {\$LIBPREFIX ''} to eliminate the prefix entirely.
{\$LIBSUFFIX 'string'}	Adds a specified suffix to the output file name before the extension. For example, use {\$LIBSUFFIX '-2.1.3'} in something.pas to generate something-2.1.3.bpl.
{\$LIBVERSION 'string'}	Adds a second extension to the output file name after the .bpl extension. For example, use {\$LIBVERSION '2.1.3'} in something.pas to generate something.bpl.2.1.3.

For more information on packages, see Chapter 11, “Working with packages and components.”

## When to use packages and DLLs

---

For most applications written in Delphi, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

You cannot pass runtime type information (RTTI) across DLLs or from a DLL to an executable. That’s because DLLs all maintain their own symbol information. If you need to pass a *TStrings* object from a DLL then using an **is** or **as** operator, you need to create a package rather than a DLL. Packages share symbol information.

## Writing database applications

---

**Note** Not all versions of Delphi include database support.

One of Delphi's strengths is its support for creating advanced database applications. Delphi supports tools that allow you to connect to SQL servers and databases such as Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, and DB2 while providing transparent data sharing between applications.

Delphi includes many components for accessing databases and representing the information they contain. On the Component palette, the database components are grouped according to the data access mechanism and function.

**Table 5.2** Database pages on the Component palette

Palette page	Contents
BDE	Components that use the Borland Database Engine (BDE), a large API for interacting with databases. The BDE supports the broadest range of functions and comes with the most supporting utilities including Database Desktop, Database Explorer, SQL Monitor, and BDE Administrator. See Chapter 20, "Using the Borland Database Engine" for details.
ADO	Components that use ActiveX Data Objects (ADO), developed by Microsoft, to access database information. Many ADO drivers are available for connecting to different database servers. ADO-based components let you integrate your application into an ADO-based environment. See Chapter 21, "Working with ADO components" for details.
dbExpress	Cross-platform components that use dbExpress to access database information. dbExpress drivers provide fast access to databases but need to be used with <i>TClientDataSet</i> and <i>TDataSetProvider</i> to perform updates. See Chapter 22, "Using unidirectional datasets" for details.
InterBase	Components that access InterBase databases directly, without going through a separate engine layer. For more information about using the InterBase components, see the online Help.
Data Access	Components that can be used with any data access mechanism such as <i>TClientDataSet</i> and <i>TDataSetProvider</i> . See Chapter 23, "Using client datasets" for information about client datasets. See Chapter 24, "Using provider components" for information about providers.
Data Controls	Data-aware controls that can access information from a data source. See Chapter 15, "Using data controls" for details.

When designing a database application, you must decide which data access mechanism to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

See Part II, "Developing database applications" in this manual for details on how to use Delphi to create both database client applications and application servers. Refer to "Deploying database applications" on page 13-6 for deployment information.

## Distributing database applications

---

Delphi provides support for creating distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including DCOM, CORBA, TCP/IP, and SOAP.

For more information about building distributed database applications, see Chapter 25, “Creating multi-tiered applications.”

Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see “Deploying database applications” on page 13-6.

## Creating Web server applications

---

Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests.

You can create several different types of Web server applications using the following Delphi technologies:

- Web Broker
- WebSnap
- InternetExpress
- Web Services

### Using Web Broker

---

You can use Web Broker (also called NetCLX architecture) to create Web server applications such as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Components on the Internet page of the Component palette enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client.

To create a new Web server application using the Web Broker architecture, select File | New | Other and select Web Server Application in the New Items dialog box. Then select the Web server application type:

**Table 5.3** Web server applications

Web server application type	Description
ISAPI and NSAPI Dynamic Link Library	<p>ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread.</p> <p>Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.</p>
CGI Stand-alone executable	<p>CGI Web server applications are console applications that receive requests from clients on standard input, process those requests, and sends back the results to the server on standard output to be sent to the client.</p> <p>Selecting this type of application adds the required entries to the <b>uses</b> clause of the project file and adds the appropriate \$APPTYPE directive to the source.</p>
Win-CGI Stand-alone executable	<p>Win-CGI Web server applications are Windows applications that receive requests from clients from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application.</p> <p>Selecting this type of application adds the required entries to the <b>uses</b> clause of the project file and adds the appropriate \$APPTYPE directive to the source.</p>
Apache Shared Module (DLL)	<p>Selecting this type of application sets up your project as a DLL. Apache Web server applications are DLLs loaded by the Web server. Information is passed to the DLL, processed, and returned to the client by the Web server.</p>
Web App Debugger Stand-alone executable	<p>Selecting this type of application sets up an environment for developing and testing Web server applications. Web App Debugger applications are executable files loaded by the Web server. This type of application is not intended for deployment.</p>

CGI and Win-CGI applications use more system resources on the server, so complex applications are better created as ISAPI , NSAPI , or Apache DLL applications. If writing cross-platform applications, you should select CGI stand-alone or Apache Shared Module (DLL) for Web server development. These are also the same options you see when creating WebSnap and Web Service applications.

For more information on building Web server applications, see Chapter 27, “Creating Internet applications.”

## Creating WebSnap applications

---

WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other mime content for Web pages. WebSnap is for server side development. WebSnap cannot be used in cross-platform applications at this time.

To create a new WebSnap application, select File | New | Other and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (ISAPI/NSAPI, CGI, Win-CGI, Apache). See Table 5.3, “Web server applications” for details.

For more information on WebSnap, see Chapter 29, “Using WebSnap.”

## Using InternetExpress

---

InternetExpress is a set of components that extends the basic Web server application architecture to act as the client of an application server. You use InternetExpress for applications wherein browser-based clients can fetch data from a provider, resolve updates to the provider, while executing on a client.

InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML determines the layout and appearance of the pages displayed in end-user browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in the XML data packets on the client machine.

For more information on InternetExpress, see “Building Web applications using InternetExpress” on page 25-33.

## Creating Web Services applications

---

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language).

Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

You can use Delphi to build servers to implement Web Services and clients that call on those services. You can write clients for arbitrary servers to implement Web Services that respond to SOAP messages, and Delphi servers to publish Web Services for use by arbitrary clients.

Refer to Chapter 31, “Using Web Services” for more information on Web Services.

## Writing applications using COM

---

COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine. You can also use COM+, ActiveX and Active Server Pages.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

### Using COM and DCOM

---

Delphi has classes and wizards that make it easy to create COM, OLE, or ActiveX applications. You can create COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms. COM also serves as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories.

Using Delphi to create COM-based applications offers a wide range of possibilities, from improving software design by using interfaces internally in an application, to creating objects that can interact with other COM-based API objects on the system, such as the Win9x Shell extensions and DirectX multimedia support. Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

For more information on COM and Active X controls, see Chapter 33, “Overview of COM technologies,” Chapter 38, “Creating an ActiveX control,” and “Distributing a client application as an ActiveX control” on page 25-32.

For more information on DCOM, see “Using DCOM connections” on page 25-8.

### Using MTS and COM+

---

COM applications can be augmented with special services for managing objects in a large distributed environment. These services include transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) on versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later).

For more information on MTS and COM+, see Chapter 39, “Creating MTS or COM+ objects” and “Using transactional data modules” on page 25-6.

## Using data modules

---

A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are several types of data modules, including standard, remote, Web modules, applet modules, and services, depending on which edition of Delphi you have. Each type of data module serves a special purpose.

- Standard data modules are particularly useful for single- and two-tiered database applications, but can be used to organize the nonvisual components in any application. For more information, see “Creating and editing standard data modules” on page 5-15.
- Remote data modules form the basis of an application server in a multi-tiered database application. They are not available in all editions. In addition to holding the nonvisual components in the application server, remote data modules expose the interface that clients use to communicate with the application server. For more information about using them, see “Adding a remote data module to an application server project” on page 5-19.
- Web modules form the basis of Web server applications. In addition to holding the components that create the content of HTTP response messages, they handle the dispatching of HTTP messages from client applications. See Chapter 27, “Creating Internet applications” for more information about using Web modules.
- Applet modules form the basis of control panel applets. In addition to holding the nonvisual controls that implement the control panel applet, they define the properties that determine how the applet’s icon appears in the control panel and include the events that are called when users execute the applet. For more information about applet modules, see the online Help.
- Services encapsulate individual services in an NT service application. In addition to holding any nonvisual controls used to implement a service, services include the events that are called when the service is started or stopped. For more information about services, see “Service applications” on page 5-4.

### Creating and editing standard data modules

---

To create a standard data module for a project, choose File | New | Data Module. Delphi opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project.

At design time a data module looks like a standard Delphi form with a white background and no alignment grid. As with forms, you can place nonvisual components from the Component palette onto a module, and edit their properties in the Object Inspector. You can resize a data module to accommodate the components you add to it.



You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a data module.

**Table 5.4** Context menu options for data modules

Menu item	Purpose
<i>Edit</i>	Displays a context menu with which you can cut, copy, paste, delete, and select the components in the data module.
<i>Position</i>	Aligns nonvisual components to the module's invisible grid ( <i>Align To Grid</i> ) or according to criteria you supply in the Alignment dialog box ( <i>Align</i> ).
<i>Tab Order</i>	Enables you to change the order that the focus jumps from component to component when you press the tab key.
<i>Creation Order</i>	Enables you to change the order that data access components are created at start-up.
<i>Revert to Inherited</i>	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
<i>Add to Repository</i>	Stores a link to the data module in the Object Repository.
<i>View as Text</i>	Displays the text representation of the data module's properties.
<i>View DFM</i>	Toggles between the formats (binary or text) in which this particular form file is saved.

For more information about data modules, see the online Help.

## Naming a data module and its unit file

The title bar of a data module displays the module's name. The default name for a data module is "DataModuleN" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2." The corresponding unit file for *DataModule2* defaults to "Unit2."

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module:

- 1 Select the module.
- 2 Edit the *Name* property for the module in the Object Inspector.

The new name for the module appears in the title bar when the *Name* property in the Object Inspector no longer has focus.

Changing the name of a data module at design time changes its variable name in the interface section of code. It also changes any use of the type name in procedure declarations. You must manually change any references to the data module in code you write.

To rename a unit file for a data module:

- 1 Select the unit file.

## Placing and naming components

You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate page of the Component palette, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, Delphi assigns it a generic name that identifies what kind of component it is, followed by a *1*. For example, the *TDataSource* component adopts the name *DataSource1*. This makes it easy to select specific components whose properties and methods you want to work with.

You may still want to name a component a different name that reflects the type of component and what it is used for.

To change the name of a component in a data module:

- 1 Select the component.
- 2 Edit the component's *Name* property in the Object Inspector.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the Object Inspector no longer has focus.

For example, suppose your database application uses the CUSTOMER table. To access the table, you need a minimum of two data access components: a data source component (*TDataSource*) and a table component (*TClientDataSet*). When you place these components in your data module, Delphi assigns them the names *DataSource1* and *ClientDataSet1*. To reflect the type of component and the database they access, CUSTOMER, you could change these names to *CustomerSource* and *CustomerTable*.

## Using component properties and events in a data module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to *True* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1*, to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms. The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a *TDataSource* component has three possible events: *OnDataChange*, *OnStateChange*, and *OnUpdateData*. A *TClientDataSet* component has over 20 potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

## Creating business rules in a data module

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module. The prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:

```

type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
    Orders: TClientDataSet;
    :
  private
    { Private declarations }
  public
    { Public declarations }
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;

var
  CustomerData: TCustomerData;

```

The procedures and functions you write should follow in the implementation section of the code for the module.

## Accessing a data module from a form

---

To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways:

- In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section.
- Click the form's unit file, choose File | Use Unit, and enter the name of the module or pick it from the list box in the Use Unit dialog.
- For database components, in the data module click a dataset or query component to open the Fields editor and drag any existing fields from the editor onto the form. Delphi prompts you to confirm that you want to add the module to the form's **uses** clause, then creates controls (such as edit boxes) for the fields.

For example, if you've added the *TClientDataSet* component to your data module, double-click it to open the Fields editor. Select a field and drag it to the form. An edit box component appears.

Because the data source is not yet defined, Delphi adds a new data source component, *DataSource1*, to the form and sets the edit box's *DataSource* property to *DataSource1*. The data source automatically sets its *DataSet* property to the dataset component, *ClientDataSet1*, in the data module.

You can define the data source *before* you drag a field to the form by adding a *TDataSource* component to the data module. Set the data source's *DataSet* property to *ClientDataSet1*. After you drag a field to the form, the edit box appears with its *TDataSource* property already set to *DataSource1*. This method keeps your data access model cleaner.

## Adding a remote data module to an application server project

---

Some editions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.

To add a remote data module to a project:

- 1 Choose File | New | Other.
- 2 Select the Multitier page in the New Items dialog box.
- 3 Double-click the desired type of module (CORBA Data Module, Remote Data Module, or Transactional Data Module) to open the Remote Data Module wizard.

Once you add a remote data module to a project, you use it just like a standard data module.

For more information about multi-tiered database applications, see Chapter 25, "Creating multi-tiered applications."

## Using the Object Repository

---

The Object Repository (Tools | Repository) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The repository is maintained in DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

### Sharing items within a project

---

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (File | New | Other), you'll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

### Adding items to the Object Repository

---

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository. To add an item to the Object Repository,

- 1 If the item is a project or is in a project, open the project.

- 2 For a project, choose Project | Add To Repository. For a form or data module, right-click the item and choose Add To Repository.
- 3 Type a description, title, and author.
- 4 Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, Delphi creates a new page.
- 5 Choose Browse to select an icon to represent the object in the Object Repository.
- 6 Choose OK.

## Sharing objects in a team environment

---

You can share objects with your workgroup or development team by making a repository available over a network. To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

- 1 Choose Tools | Environment Options.
- 2 On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the repository, Delphi creates a DELPHI32.DRO file in the Shared Repository directory if one doesn't exist already.

## Using an Object Repository item in a project

---

To access items in the Object Repository, choose File | New | Other. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- Copy
- Inherit
- Use

### Copying an item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

### Inheriting an item

Choose Inherit to derive a new class from the selected item in the Object Repository and add the new class to your project. When you recompile your project, any changes that have been made to the item in the Object Repository will be reflected in your

derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

## Using an item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

## Using project templates

---

Templates are predesigned projects that you can use as starting points for your own work. To create a new project from a template,

- 1 Choose File | New | Other to display the New Items dialog box.
- 2 Choose the Projects tab.
- 3 Select the project template you want and choose OK.
- 4 In the Select Directory dialog, specify a directory for the new project's files.

Delphi copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

## Modifying shared items

---

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with other developers.

## Specifying a default project, new form, and main form

---

By default, when you choose File | New | Application or File | New | Form, Delphi displays a blank form. You can change this behavior by reconfiguring the Repository:

- 1 Choose Tools | Repository
- 2 If you want to specify a default project, select the Projects page and choose an item under Objects. Then select the New Project check box.

- 3 If you want to specify a default form, select a Repository page (such as Forms), then choose a form under Objects. To specify the default new form (File | New | Form), select the New Form check box. To specify the default main form for new projects, select the Main Form check box.
- 4 Click OK.

## Enabling Help in applications

---

Both the VCL and CLX support displaying Help from applications using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers. To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager.

The VCL provides to all applications an instance of *TWinHelpViewer*, which implements all of these interfaces and provides a link between applications and WinHelp; CLX requires that application developers provide their own implementation.

The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer if it can provide support for a particular Help keyword or context, and then it passes the Help request on to the viewer which says it can provide such support. (If more than one viewer supports the keyword, as would be the case in an application which had registered viewers for both Man and Info, the Help Manager can display a selection box through which the user of the application can determine which Help viewer to invoke. Otherwise, it displays the first responding Help system encountered).

### Help system interfaces

---

The Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in *HelpIntfs.pas*, which also contains the implementation of the Help Manager.

*ICustomHelpViewer* provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer.

*IExtendedHelpViewer* provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, "IntToStr" might be a keyword in the Help system, but "String manipulation routines" could be the name of a topic).

*ISpecialWinHelpViewer* provides support for responding to specialized WinHelp messages that an application running under Windows may receive and which are not easily generalizable. In general, only applications operating in the Windows environment need to implement this interface, and even then it is only required for applications that make extensive use of non-standard WinHelp messages.

*IHelpManager* provides a mechanism for the Help viewer to communicate back to the application's Help Manager and request additional information. An *IHelpManager* is obtained at the time the Help viewer registers itself.

*IHelpSystem* provides a mechanism through which *TApplication* passes Help requests on to the Help system. *TApplication* obtains an instance of an object which implements both *IHelpSystem* and *IHelpManager* at application load time and exports that instance as a property; this allows other code within the application to file Help requests directly when appropriate.

*IHelpSelector* provides a mechanism through which the Help system can invoke the user interface to ask which Help viewer should be used in cases where more than one viewer is capable of handling a Help request, and to display a Table of Contents. This display capability is not built into the Help Manager directly to allow the Help Manager code to be identical regardless of which widget set or class library is in use.

## Implementing ICustomHelpViewer

---

The *ICustomHelpViewer* interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents.

## Communicating with the Help Manager

---

*ICustomHelpViewer* provides four functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

The Help Manager calls through these functions in the following circumstances:

- *ICustomHelpViewer.GetViewerName* : *String* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* is called **immediately** following registration to provide the viewer with a unique cookie that identifies it. This information must be stored off for later use; if the viewer shuts down on its own (as opposed to in response to a notification from the Help Manager), it must provide the Help Manager with the identifying cookie so that the Help Manager can release all references to the viewer. (Failing to provide the cookie, or providing the wrong one, causes the Help Manager to potentially release references to the wrong viewer.)



- *ICustomHelpViewer.ShutDown* is called by the Help Manager to notify the Help viewer that the Manager is shutting down and that any resources the Help viewer has allocated should be freed. It is recommended that all resource freeing be delegated to this method.
- *ICustomHelpViewer.SoftShutDown* is called by the Help Manager to ask the Help viewer to close any externally visible manifestations of the help system (for example, windows displaying help information) without unloading the viewer.

## Asking the Help Manager for information

---

Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things: a request for the window handle of the currently active control; a request for the name of the Help file which the Help Manager believes should contain help for the currently active control; a request for the path to that Help file; and a notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that it do so.

*IHelpManager.GetHandle* : *LongInt* is called by the Help viewer if it needs to know the handle of the currently active control; the result is a window handle.

*IHelpManager.GetHelpFile* : *String* is called by the Help viewer if it wishes to know the name of the Help file which the currently active control believes contains its help.

*IHelpManager.Release* is called to notify the Help Manager when a Help viewer is disconnecting. It should *never* be called in response to a request through *ICustomHelpViewer.ShutDown*; it is only used to notify the Help Manager of unexpected disconnects.

## Displaying keyword-based Help

---

Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier. (Numeric help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system; while CLX supports them, they are not recommended for use in CLX applications because most Linux Help systems do not understand them.) *ICustomHelpViewer* implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests.

*ICustomHelpViewer* provides three methods for handling keyword-based Help:

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.UnderstandsKeyword(const HelpString: String): Integer
```

is the first of the three methods called by the Help Manager, which will call *each* registered Help viewer with the same string to ask if the viewer provides help for that string; the viewer is expected to respond with an integer indicating how many different Help pages it can display in response to that Help request. The viewer can use any method it wants to determine this — inside the IDE, the HyperHelp viewer maintains its own index and searches it. If the viewer does not support help on this keyword, it should return zero. Negative numbers are currently interpreted as meaning zero, but this behavior is not guaranteed in future releases.

```
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
```

is called by the Help Manager if more than one viewer can provide help on a topic. The viewer is expected to return a *TStringList*. The strings in the returned list should map to the pages available for that keyword, but the characteristics of that mapping can be determined by the viewer. In the case of the HyperHelp viewer, the string list always contains exactly one entry (HyperHelp provides its own indexing, and duplicating that elsewhere would be pointless duplication); in the case of the Man page viewer, the string list consists of multiple strings, one for each section of the manual which contains a page for that keyword.

```
ICustomHelpViewer.ShowHelp(const HelpString: String)
```

is called by the Help Manager if it needs the Help viewer to display help for a particular keyword. This is the last method call in the operation; it is guaranteed to never be called unless *CanShowKeyword* is invoked first.

## Displaying tables of contents

---

*ICustomHelpViewer* provides two methods relating to displaying tables of contents:

- *CanShowTableOfContents*
- *ShowTableOfContents*

The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling *ICustomHelpViewer.CanShowTableOfContents : Boolean* and then invokes a particular Help viewer by calling *ICustomHelpViewer.ShowTableOfContents*.

It is reasonable for a particular viewer to refuse to allow requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports a table of contents, on the other hand, by passing the request to display a table of contents directly to HyperHelp. It is *not* reasonable, however, for an implementation of *ICustomHelpViewer* to respond to queries through *CanShowTableOfContents* with the answer *true*, and then ignore requests through *ShowTableOfContents*.

## Implementing IExtendedHelpViewer

---

*ICustomHelpViewer* only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself.

Applications written in CLX can talk to systems requiring context-based Help by extending the object which implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking to Help systems that allow you to jump directly to high-level topics instead of using keyword searches.

*IExtendedHelpViewer* exposes four functions. Two of them — *UnderstandsContext* and *DisplayHelpByContext* — are used to support context-based Help; the other two — *UnderstandsTopic* and *DisplayTopic* — are used to support topics.

When an application user presses F1, the Help Manager calls

```
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
```

and the currently activated control supports context-based, rather than keyword-based Help. As with *ICustomHelpViewer.CanShowKeyword*, the Help Manager queries all registered Help viewers iteratively. Unlike the case with *ICustomHelpViewer.CanShowKeyword*, however, if more than one viewer supports a specified context, the *first* registered viewer with support for a given context is invoked.

The Help Manager calls

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
```

after it has polled the registered Help viewers.

The topic support functions work the same way:

```
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
```

is used to poll the Help viewers asking if they support a topic;

```
IExtendedHelpViewer.DisplayTopic(const Topic: String)
```

is used to invoke the first registered viewer which reports that it is able to provide help for that topic.

## Implementing IHelpSelector

---

*IHelpSelector* is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims

to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application.

To override the decision of the Help Manager in such cases, an application must register a class that provides an implementation of the *IHelpSelector* interface. *IHelpSelector* exports two functions: *SelectKeyword*, and *TableOfContents*. Both take as arguments a *TStrings* containing, one by one, either the possible keyword matches or the names of the viewers claiming to provide a table of contents. The implementor is required to return the index (in the *TStrings*) that represents the selected string.

**Note** The Help Manager may get confused if the strings are re-arranged; it is recommended that implementors of *IHelpSelector* refrain from doing this. The Help system only supports *one* HelpSelector; when new selectors are registered, any previously existing selectors are disconnected.

## Registering Help system objects

---

For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer*, *IExtendedHelpViewer*, *ISpecialWinHelpViewer*, and *IHelpSelector* must register with the Help Manager.

To register Help system objects with the Help Manager, you need to

- Register the Help viewer
- Register the Help Selector

## Registering Help viewers

The unit that contains the object implementation must use *HelpIntfs*. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must assign the instance variable and pass it to the function *RegisterViewer*. *RegisterViewer* is a flat function exported by *HelpIntfs.pas* which takes as an argument an *ICustomHelpViewer* and returns an *IHelpManager*. The *IHelpManager* should be stored for future use.

## Registering Help selectors

The unit that contains the object implementation must use *HelpIntfs* and *QForms*. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must register the Help selector through the *HelpSystem* property of the global Application object:

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

This procedure does not return a value.

## Using Help in a VCL Application

---

The following sections explain how to use Help within a VCL application.

- How TApplication processes VCL Help
- How VCL controls process Help
- Calling a Help system directly
- Using IHelpSystem

### How TApplication processes VCL Help

---

*TApplication* in the VCL provides four methods that are accessible from application code:

**Table 5.5** Help methods in TApplication

HelpCommand	Takes a Windows Help style HELP_COMMAND and passes it off to WinHelp. Help requests forwarded through this mechanism are passed only to implementations of ISpecialWinHelpViewer.
HelpContext	Invokes the Help System with a request for context-based Help.
HelpKeyword	Invokes the HelpSystem with a request for keyword-based Help.
HelpJump	Requests the display of a particular topic.

All four functions take the data passed to them and forward it through a data member of *TApplication* which represents the Help System. That data member is directly accessible through the property *HelpSystem*.

### How VCL controls process Help

---

All controls that derive from *TControl* expose three properties which are used by the Help system: *HelpSystem*, *HelpType*, *HelpContext*, and *HelpKeyword*.

The *HelpType* property contains an instance of an enumerated type that determines if the control's designer expects help to be provided via keyword-based Help or context-based Help. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, that can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of Help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWinControl* calls *InvokeHelp*.

# Using Help in a CLX Application

---

The following sections explain how to use Help within a CLX application.

- How *TApplication* processes CLX Help
- How CLX controls process Help
- Calling a Help system directly
- Using *IHelpSystem*

## How *TApplication* processes CLX Help

---

*TApplication* in CLX provides two methods that are accessible from application code:

- *ContextHelp*, which invokes the Help system with a request for context-based Help
- *KeywordHelp*, which invokes the Help system with a request for keyword-based Help

Both functions take as an argument the context or keyword being passed and forward the request on through a data member of *TApplication*, which represents the Help system. That data member is directly accessible through the read-only property *HelpSystem*.

## How CLX controls process Help

---

All controls that derive from *TControl* expose four properties which are used by the Help system: *HelpType*, *HelpFile*, *HelpContext*, and *HelpKeyword*. *HelpFile* is supposed to contain the name of the file in which the control's help is located; if the help is located in an external Help system that does not care about file names (say, for example, the Man page system), then the property should be left blank.

The *HelpType* property contains an instance of an enumerated type which determines if the control's designer expects help to be provided via keyword-based Help or context-based Help; the other two properties are linked to it. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, which can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWidgetControl* calls *InvokeHelp*.

## Calling a Help system directly

---

For additional Help system functionality not provided by the VCL or CLX, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.

## Using IHelpSystem

---

*IHelpSystem* allows a VCL or CLX application to do three things:

- Provides path information to the Help Manager
- Provides a new Help selector
- Asks the Help Manager to display help

Assigning a Help selector allows the Help Manager to delegate decision-making in cases where multiple external Help systems can provide help for the same keyword. For more information, see the section “Implementing *IHelpSelector*” on page 5-26.

*IHelpSystem* exports four procedures and one function to request the Help Manager to display help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

*Hook* is intended entirely for WinHelp compatibility and should not be used in a CLX application; it allows processing of WM\_HELP messages that cannot be mapped directly onto requests for keyword-based, context-based, or topic-based Help. The other methods each take two arguments: the keyword, context ID, or topic for which help is being requested, and the Help file in which it is expected that help can be found.

In general, unless you are asking for topic-based help, it is equally effective and more clear to pass help requests to the Help Manager through the *InvokeHelp* method of your control.

## Customizing the IDE Help system

---

The Delphi IDE supports multiple Help viewers in exactly the same way that a VCL or CLX application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same *WinHelpViewer* that the VCL uses.

To install a new Help viewer in the IDE, you do exactly what you would do in a CLX application, with one difference. You write an object that implements *ICustomHelpViewer* (and, if desired, *IExtendedHelpViewer*) to forward Help requests to the external viewer of your choice, and you register the *ICustomHelpViewer* with the IDE.

To register a custom Help viewer with the IDE,

- 1 Make sure that the unit implementing the Help viewer contains `HelpIntfs.pas`.
- 2 Build the unit into a design-time package registered with the IDE, and build the package with runtime packages turned on. (This is necessary to ensure that the Help Manager instance used by the unit is the same as the Help Manager instance used by the IDE.)
- 3 Make sure that the Help viewer exists as a global instance within the unit.
- 4 In the initialization section of the unit, make sure that the instance is passed to the `RegisterHelpViewer` function.





# Developing the application user interface

With Delphi, you design a user interface (UI) by selecting components from the component palette and dropping them onto forms. You get the components to do what you want by setting their properties and coding their event handlers.

## Controlling application behavior

---

*TApplication*, *TScreen*, and *TForm* are the classes that form the backbone of all Delphi applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information such as screen resolution and available display fonts. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

## Using the main form

---

*TForm* is the key class for creating GUI applications. When you open Delphi displaying a default project or when you create a new project, a form is displayed on which you can begin your UI design.

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form,

- 1 Choose Project | Options and select the Forms page.
- 2 In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed.

## Adding forms

---

To add a form to your project, select File | New Form. You can see all your project's forms and their associated units listed in the Project Manager (View | Project Manager) and you can display a list of the forms alone by choosing View | Forms.

## Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form,

- 1 Select the form that needs to refer to another.
- 2 Choose File | Use Unit.
- 3 Select the name of the form unit for the form to be referenced.
- 4 Choose OK.

Linking a form to another just means that the **uses** clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

## Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the File | Use Unit command does.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both **uses** clauses in the **interface** parts of their respective unit files. This will generate the "Circular reference" error at compile time.

## Hiding the main form

---

You can prevent the main form from displaying when your application first starts up. To do so, you must use the global *Application* variable (described in the next topic).

To hide the main form at startup,

- 1 Choose Project | View Source to display the main project file.
- 2 Add the following lines after the call to `Application.CreateForm` and before the call to `Application.Run`.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

**Note** You can set the form's *Visible* property to *False* using the Object Inspector at design time rather than setting it at runtime as shown above.

## Working at the application level

---

The global variable *Application*, of type *TApplication*, is in every VCL or CLX based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* would handle how you would call a help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the Project | Options Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnMessage* event lets you intercept Windows messages (on Windows only), the *OnEvent* event lets you intercept events, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

## Handling the screen

---

A global variable of type *TScreen* called *Screen* is created when you create a project. *Screen* encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying

- the look of the cursor
- the size of the window in which your application is running
- a list of fonts available to the screen device
- multiple screen behavior (not available for cross-platform)

If your Windows application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.

If using CLX for cross-platform programming, the default behavior is that applications create a screen component based on information about the current screen device and assign it to *Screen*.

## Managing layout

---

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

*TControl* introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
    MaxHeight: Integer) of object;
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as *var* parameters which can be changed inside the event handler. *OnConstrainedResize* is published for container objects (*TForm*, *TScrollBar*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

## Responding to event notification

---

The operating system will notify your application when an event has occurred (such as a mouse click, keystrokes entered, and so on) while it is running. The underlying way that event notifications are handled by VCL and CLX objects is different, but the way you work with event notifications at the component level is typically the same. Components have events and methods built-in for the most commonly occurring events. You can use the methods provided with the component in most cases. If you need to write additional event handling, you can override an existing method to write your own. Unless you are writing your own components, you do not need to change the underlying event notification schema.

**VCL** If developing applications for Windows only, you need to understand that Windows is a message-based operating system. System messages are handled by a message handler that translates the message to an event or event handler. The message itself is a record passed to a control by Windows. For instance, when you click a mouse button on a dialog box, Windows sends a message to the active control and the application containing that control reacts to this new event. If the click occurs over a button, the *OnClick* event could be activated upon receipt of the message. If the click occurs just in the form, the application can ignore the message.

The record type passed to the application by Windows is called a *TMsg*. Windows predefines a constant for each message, and these values are stored in the message field of the *TMsg* record. Each of these constants begin with the letters *wm*.

The VCL automatically handles messages unless you override the message handling system and create your own message handlers. For more information on messages and message handling, see “Understanding the message-handling system” on page 46-1, “Changing message handling” on page 46-3, and “Creating new message handlers” on page 46-5.

**CLX** For cross-platform programming: The operating system notification that an event occurred is sent to the underlying Qt widget layer where it is translated into an event and eventually into event objects by *HookEvents*. *EventFilter* is called automatically when a CLX control needs to handle a Qt mouse or keyboard event.

*EventFilter* responds to event notifications by performing the default response. Typically, this involves dispatching the event to the appropriate virtual method (such as the *Click* method, which generates an *OnClick* event).

**CLX Note** When overriding the *EventFilter* method, you need to call the inherited method so that the default event processing can occur.

## Using forms

---

When you create a form in Delphi from the IDE, Delphi automatically creates the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don’t have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default Delphi behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

## Controlling when forms reside in memory

---

By default, Delphi automatically creates the application's main form in memory by including the following code in the application's main entry point:

```
Application.CreateForm(TForm1, Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form's class and is used to reference the form while the application is running. Any unit that includes the form's unit in its **uses** clause can access the form via this variable.

All forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

### Displaying an auto-created form

If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
    ResultsForm.ShowModal;
end;
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

### Creating forms dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE, you:

- 1 Select the File | New Form from the main menu to display the new form.

- 2 Remove the form from the Auto-create forms list of the Project | Options | Forms page.

This removes the form's invocation. As an alternative, you can manually remove the following line from program's main entry point:

```
Application.CreateForm(TResultsForm, ResultsForm);
```

- 3 Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm:=TResultForm.Create(self);
  try
    ResultsForm.ShowModal;
  finally
    ResultsForm.Free;
  end;
```

In the above example, note the use of **try..finally**. Putting in the line `ResultsForm.Free;` in the **finally** clause ensures that the memory for the form is freed even if the form raises an exception.

The event handler in the example deletes the form after it is closed, so the form would need to be recreated if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

**Note** If you create a form using its constructor, be sure to check that the form is not in the Auto-create forms list on the Project Options | Forms page. Specifically, if you create the new form without deleting the form of the same name from the list, Delphi creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global variable no longer points to a valid form. Any attempt to use the global variable will likely crash the application.

## Creating modeless forms such as windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

## Using a local variable to create a form instance

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not



matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

```

procedure TMainForm.Button1Click(Sender: TObject);
var
    RF:TResultForm;
begin
    RF:=TResultForm.Create(self)
    RF.ShowModal;
    RF.Free;
end;

```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

## Passing additional arguments to forms

---

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be *nil*.

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

```

TResultsForm = class(TForm)
    ResultsLabel: TLabel;
    OKButton: TButton;
    procedure OKButtonClick(Sender: TObject);
private
public
    constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;

```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

```

constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
    inherited Create(Owner);
    case whichButton of

```

```

1: ResultsLabel.Caption := 'You picked the first button.';
2: ResultsLabel.Caption := 'You picked the second button.';
3: ResultsLabel.Caption := 'You picked the third button.';
end;
end;

```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

```

procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;

```

## Retrieving data from forms

---

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

### Retrieving data from modeless forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red", "Green", "Blue", and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

```

TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;

```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor*, to store the

actual value for the property in the private data member *FColor*:

```

procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
    Index: Integer;
begin
    Index := ColorListBox.ItemIndex;
    if Index >= 0 then
        CurrentColor := ColorListBox.Items[Index]
    else
        CurrentColor := '';
    end;

```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton*) on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```

procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
    MainColor: String;
begin
    if Assigned(ColorForm) then
        begin
            MainColor := ColorForm.CurrentColor;
            {do something with the string MainColor}
        end;
    end;

```

The event handler first verifies that *ColorForm* exists using the *Assigned* function. It then gets the value of *ColorForm*'s *CurrentColor* property.

Alternatively, if *ColorForm* had a public function named *GetColor*, another form could get the current color without using the *CurrentColor* property (for example, `MainColor := ColorForm.GetColor;`). In fact, there's nothing to prevent another form from getting the *ColorForm*'s currently selected color by checking the listbox selection directly:

```

with ColorForm.ColorListBox do
    MainColor := Items[Index];

```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

## Retrieving data from modal forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* argument. Presumably, this *Pointer* points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

The constructor saves the pointer to a private data member *FColor* and initializes the string to an empty string.

**Note** To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see “Controlling when forms reside in memory” on page 6-6.

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
    end;
  Close;
end;
```

Notice that the event handler stores the selected color name in the string referenced by the pointer that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked.

The event handler would look as follows:

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;

procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

*UpdateButtonClick* creates a String called *MainColor*. The address of *MainColor* is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to *MainColor* as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in *MainColor*, assuming that a color was selected. Otherwise, *MainColor* contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having *MainColor* default to an empty string).

## Reusing components and groups of components

---

Delphi offers several ways to save and reuse work you've done with components:

- *Component templates* provide a simple, quick way of configuring and saving groups of components. See "Creating and using component templates" on page 6-13.
- You can save forms, data modules, and projects in the *Repository*. This gives you a central database of reusable elements and lets you use form inheritance to propagate changes. See "Using the Object Repository" on page 5-19.
- You can save *frames* on the component palette or in the repository. Frames use form inheritance and can be embedded into forms or other frames. See "Working with frames" on page 6-13.
- Creating a *custom component* is the most complicated way of reusing code, but it offers the greatest flexibility. See Chapter 40, "Overview of component creation."

## Creating and using component templates

---

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the component palette, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template,

- 1 Place and arrange components on a form. In the Object Inspector, set their properties and events as desired.
- 2 Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.
- 3 Choose Component | Create Component Template.
- 4 Specify a name for the component template in the Component Name edit box. The default proposal is the component type of the first component selected in step 2 followed by the word "Template". For example, if you select a label and then an edit box, the proposed name will be "TLabelTemplate". You can change this name, but be careful not to duplicate existing component names.
- 5 In the Palette Page edit box, specify the component palette page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.
- 6 Under Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.
- 7 Click OK.

To remove templates from the component palette, choose Component | Configure Palette.

## Working with frames

---

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties.

In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the component palette for easy reuse, and they can be nested

within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Frames are useful to organize groups of controls that are used in multiple places in your application. For example, if you have a bitmap that is used on multiple forms, you can put it in a frame and only one copy of that bitmap is included in the resources of your application. You could also describe a set of edit fields that are intended to edit a table with a frame and use that whenever you want to enter data into the table.

## Creating frames

---

To create an empty frame, choose **File | New | Frame**, or choose **File | New** and double-click on **Frame**. You can then drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose **File | New | Application**, close the new form and unit without saving them, then choose **File | New | Frame** and save the project.

**Note** When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing **View | Forms** and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame's form file by right-clicking and choosing **View as Form** or **View as Text**.

## Adding frames to the component palette

---

Frames are added to the component palette as component templates. To add a frame to the component palette, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click on the frame, and choose **Add to Palette**. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

## Using and modifying frames

---

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

- Select a frame from the component palette and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.

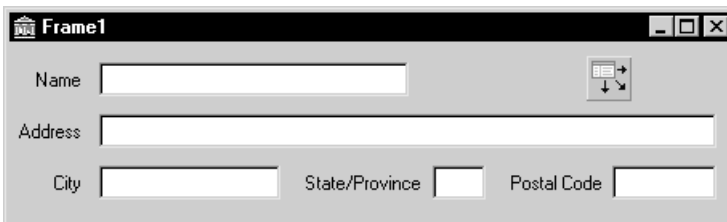
- Select *Frames* from the Standard page of the component palette and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Delphi declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Delphi declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed.

If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

**Figure 6.1** A frame with data-aware controls and a data source component



In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the *Picture* property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Image* object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *TImage.Picture* once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

## Sharing frames

---

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.pas) and form (.dfm or .xfm) files.



To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository For more information, see “Using the Object Repository” on page 5-19.

If you send a frame’s unit and form files to other developers, they can open them and add them to the component palette. If the frame has other frames embedded in it, they will have to open it as part of a project.

## Organizing actions for toolbars and menus

---

Delphi provides several features that simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon.

Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.

On the Windows platform, tools are provided to make it easy to define and group actions, create different layouts, and customize menus at design time or runtime. These tools are known collectively as ActionBand tools, and the menus and toolbars you create with them are known as action bands. In general, you can create an ActionBand user interface as follows:

- Build the action list to create a set of actions that will be available for your application (use the Action Manager, *TActionManager*)
- Add the user interface elements to the application (use ActionBand components such as *TActionMainMenuBar* and *TActionToolBar*)
- Drag and drop actions from the Action Manager onto the user interface elements

The following table defines the terminology related to setting up menus and toolbars:

**Table 6.1** Action setup terminology

Term	Definition
Action	A response to something a user does, such as clicking a menu item. Many standard actions that are frequently required are provided for you to use in your applications as is. For example, file operations such as File Open, File Save As, File Run, and File Exit are included along with many others for editing, formatting, searches, help, dialogs, and window actions. You can also program custom actions and access them using action lists and the Action Manager.
Action band	A container for a set of actions associated with a customizable menu or toolbar. The ActionBand components for main menus and toolbars ( <i>TActionMainMenuBar</i> and <i>TActionToolBar</i> ) are examples of action bands.
Action category	Lets you group actions and drop them as a group onto a menu or toolbar. For example, one of the standard action categories is Search which includes Find, FindFirst, FindNext, and Replace actions all at once.

**Table 6.1** Action setup terminology (continued)

Term	Definition
Action classes	Classes that perform the actions used in your application. All of the standard actions are defined in action classes such as <i>TEditCopy</i> , <i>TEditCut</i> , and <i>TEditUndo</i> . You can use these classes by dragging and dropping them from the Customize dialog onto an action band.
Action client	Most often represents a menu item or a button that receives a notification to initiate an action. When the client receives a user command (such as a mouse click), it initiates an associated action.
Action list	Maintains a list of actions that your application can take in response to something a user does.
Action Manager	Groups and organizes logical sets of actions that can be reused on ActionBand components. See <i>TActionManager</i> .
Menu	Lists commands that the user of the application can execute by clicking on them. You can create menus by using the ActionBand menu class <i>TActionMainMenuBar</i> , or by using cross-platform components such as <i>TMainMenu</i> or <i>TPopupMenu</i> .
Target	Represents the item an action does something to. The target is usually a control, such as a memo or a data control. Not all actions require a target. For example, the standard help actions ignore the target and simply launch the help system.
Toolbar	Displays a visible row of button icons which, when clicked, cause the program to perform some action, such as printing the current document. You can create toolbars by using the ActionBand toolbar component <i>TActionToolBar</i> , or by using the cross-platform component <i>TToolBar</i> .

If you are doing cross-platform development, refer to “Using action lists” on page 6-23.

## What is an action?

As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, Tools | Customize).

An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond when a control fires, for example, when the application user clicks a button or chooses a menu item. You can create a repertoire of actions that are available to your application through menus, through buttons, through toolbars, context menus, and so on.

Actions are associated with other components:

- **Clients:** One or more clients use the action. The client most often represents a menu item or a button (for example, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton*, and so on). Actions also reside on ActionBand components such as *TActionMainMenuBar* and *TActionToolBar*. When the client receives a user command (such as a mouse click), it initiates an associated action. Typically, a client’s *OnClick* event is associated with its action’s *Execute* event.

- **Target:** The action acts on the target. The target is usually a control, such as a memo or a data control. Component writers can create actions specific to the needs of the controls they design and use, and then package those units to create more modular applications. Not all actions use a target. For example, the standard help actions ignore the target and simply launch the help system.

A target can also be a component. For example, data controls change the target to an associated dataset.

The client influences the action—the action responds when a client fires the action. The action also influences the client—action properties dynamically update the client properties. For example, if at runtime an action is disabled (by setting its *Enabled* property to *False*), every client of that action is disabled, appearing grayed.

You can add, delete, and rearrange actions using the Action Manager or the Action List editor (displayed by double-clicking an action list object, *TActionList*). These actions are later connected to client controls.

## Setting up action bands

---

Because actions do not maintain any “layout” (either appearance or positional) information, Delphi provides action bands which are capable of storing this data. Action bands provide a mechanism that allows you to specify layout information and a set of controls. You can render actions as UI elements such as toolbars and menus.

You organize sets of actions using the Action Manager (*TActionManager*). You can use standard actions provided or create new actions of your own.

You then create the action bands:

- Use *TActionMainMenuBar* to create a main menu.
- Use *TActionToolBar* to create a toolbar.

The action bands act as containers that hold and render sets of actions. You can drag and drop items from the Action Manager editor onto the action band at design time. At runtime, application users can also customize the application’s menus or toolbars using a dialog box similar to the Action Manager editor.

## Creating toolbars and menus

---

**Note** This section describes the recommended method for creating menus and toolbars in Windows applications. For cross-platform development, you need to use *TToolBar* and the menu components, such as *TMainMenu*, organizing them using action lists (*TActionList*). See “Setting up action lists” on page 6-23.

You use the Action Manager to automatically generate toolbars and main menus based on the actions contained in your application. The Action Manager manages standard actions and any custom actions that you have written. You then create UI elements based on these actions and use action bands to render the actions items as either menu items or as buttons on a toolbar.

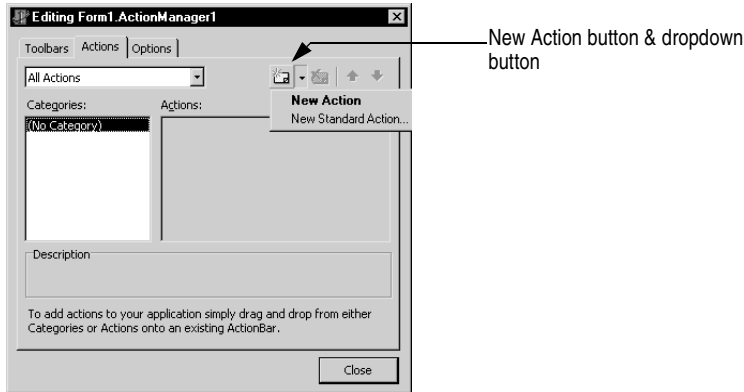
The general procedure for creating menus, toolbars, and other action bands involves these steps:

- Drop an Action Manager onto a form.
- Add actions to the Action Manager, which organizes them into appropriate action lists.
- Create the action bands (that is, the menu or the toolbar) for the user interface.
- Drag and drop the actions into the application interface.

The following procedure explains these steps in more detail.

To create menus and toolbars using action bands:

- 1** From the Additional page of the component palette, drop an Action Manager component (*TActionManager*) onto the form where you want to create the toolbar or menu.
- 2** If you want images on the menu or toolbar, drop an ImageList component from the Win32 page of the component palette onto a form. (You need to add the images you want to use to the ImageList or use the one provided.)
- 3** From the Additional page of the component palette, drop one or more of the following action bands onto the form:
  - *TActionMainMenuBar* (for designing main menus)
  - *TActionToolBar* (for designing toolbars)
- 4** Connect the ImageList to the Action Manager: with focus on the Action Manager and in the Object Inspector, select the name of the ImageList from the Images property.
- 5** Add actions to the Action Manager editor's action pane:
  - Double-click the Action Manager to display the Action Manager editor.
  - Click the drop-down arrow next to the New Action button (the leftmost button at the top right corner of the Actions tab, as shown in Figure 6.2) and select "New Action..." or "New Standard Action...". A tree view is displayed. Add one or more actions or categories of actions to the Action Manager's actions pane. The Action Manager adds the actions to its action lists.

**Figure 6.2** The Action Manager editor.

- 6 Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar you are designing.

To add user-defined actions, create a new *TAction* by pressing the New Action button and writing an event handler that defines how it will respond when fired. See “What happens when an action fires” on page 6-24 for details. Once you’ve defined the actions, you can drag and drop them onto menus or toolbars like the standard actions.

### Adding color, patterns, or pictures to menus, buttons, and toolbars

You can use the *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on a menu item or button. These properties also let you set up a banner the runs up the left or right side of a menu.

You assign backgrounds and layouts to subitems from their action client objects. If you want to set the background of the items in a menu, in the form designer click on the menu item that contains the items. For example, selecting File lets you change the background of items appearing on the File menu. You can assign a color, pattern, or bitmap in the *Background* property in the Object Inspector.

Use the *BackgroundLayout* property to describe how to place the background on the element. Colors or images can be placed behind the caption normally, stretched to fit the item area, or tiled in small squares to cover the area.

Items with normal (*blNormal*), stretched (*blStretch*), or tiled (*blTile*) backgrounds are rendered with a transparent background. If you create a banner, the full image is placed on the left (*blLeftBanner*) or the right (*blRightBanner*) of the item. You need to make sure it is the correct size because it is not stretched or shrunk to fit.

To change the background of an action band (that is, on a main menu or toolbar), select the action band and choose the *TActionClientBar* through the action band collection editor. You can set *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on the entire toolbar or menu.

## Adding icons to menus and toolbars

You can add icons next to menu items or replace captions on toolbars with icons. You organize bitmaps or icons using an `ImageList`.

- 1 Drop an `ImageList` component from the Win32 page of the component palette onto a form.
- 2 Add the images you want to use to the `ImageList`: Double-click the `ImageList`. Click Add and navigate to the images you want to use and click OK when done. Some sample images are included in Program Files\Common Files\Borland Shared\Images. (The buttons images include two views of each for active and inactive buttons.)
- 3 From the Additional page of the component palette, drop one or more of the following action bands onto the form:
  - `TActionMainMenuBar` (for designing main menus)
  - `TActionToolBar` (for designing toolbars)
- 4 Connect the `ImageList` to the Action Manager. First, set the focus on the Action Manager. Next, in the Object Inspector, select the name of the `ImageList` from the Images property.
- 5 Use the Action Manager editor to add actions to the Action Manager. You can associate an image with an action by setting its `ImageIndex` property to its number in the `ImageList`.
- 6 Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar.
- 7 For toolbars where you only want to display the icon and no caption: select the Toolbar action band and double-click its Items property. In the collection editor, you can select one or more items and set their Caption properties.
- 8 The images automatically appear on the menu or toolbar.

## Creating toolbars and menus that users can customize

You can use action bands with the Action Manager to create customizable toolbars and menus. At runtime, users of your application can customize the toolbars and menus (action bands) in the application user interface using a customization dialog similar to the Action Manager editor.

To allow the user of your application to customize an action band in your application:

- 1 Drop an Action Manager component onto a form.
- 2 Drop your action band components (`TActionMainMenuBar`, `TActionToolBar`).
- 3 Double-click the Action Manager to display the Action Manager editor:
  - Add the actions you want to use in your application. Also add the Customize action, which appears at the bottom of the standard actions list.

- Drop a *TCustomizeDlg* component from the Dialogs tab onto the form, and connect it to the Action Manager using its *ActionManager* property. You specify a filename for where to stream customizations made by users.
- Drag and drop the actions onto the action band components. (Make sure you add the *Customize* action to the toolbar or menu.)

#### 4 Complete your application.

When you compile and run the application, users can access a *Customize* command that displays a customization dialog box similar to the Action Manager editor. They can drag and drop menu items and create toolbars using the same actions you supplied in the Action Manager.

### Hiding unused items and categories in action bands

One benefit of using *ActionBands* is that unused items and categories can be hidden from the user. Over time, the action bands become customized for the application users, showing only the items that they use and hiding the rest from view. Hidden items can become visible again when the user presses a dropdown button. Also, the user can restore the visibility of all action band items by resetting the usage statistics from the customization dialog. Item hiding is the default behavior of action bands, but that behavior can be changed to prevent hiding of individual items, all the items in a particular collection (like the File menu), or all of the items in a given action band.

The action manager keeps track of the number of times an action has been called by the user, which is stored in the associated *TActionClientItem*'s *UsageCount* field. The action manager also records the number of times the application has been run, which we shall call the session number, as well as the session number of the last time an action was used. The value of *UsageCount* is used to look up the maximum number of sessions the item can go unused before it becomes hidden, which is then compared with the difference between the current session number and the session number of the last use of the item. If that difference is greater than the number determined in *PrioritySchedule*, the item is hidden. The default values of *PrioritySchedule* are shown in the table below:

**Table 6.2** Default values of the action manager's *PrioritySchedule* property

Number of sessions in which an action band item was used	Number of sessions an item will remain unhidden after its last use
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29
25 or more	31

It is possible to disable item hiding at design time. To prevent a specific action (and all the collections containing it) from becoming hidden, find its `TActionClientItem` object and set its `UsageCount` to -1. To prevent hiding for an entire collection of items, such as the File menu or even the main menu bar, find the `TActionClients` object associated with the collection and set its `HideUnused` property to `False`.

## Using action lists

---

**Note** The contents of this section apply to setting up toolbars and menus for cross-platform development. For Windows development you can also use the methods described here. However, using action bands instead is simpler and offers more options. The action lists will be handled automatically by the Action Manager. See “Organizing actions for toolbars and menus” on page 6-16 for information on using action bands and the Action Manager.

Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for performing actions (for example, when a toolbar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

### Setting up action lists

---

Setting up action lists is fairly easy once you understand the basic steps involved:

- Create the action list.
- Add actions to the action list.
- Set properties on the actions.
- Attach clients to the action.

Here are the steps in more detail:

- 1 Drop a `TActionList` object onto your form or data module. (`ActionList` is on the Standard page of the component palette.)
- 2 Double-click the `TActionList` object to display the Action List editor.
  - a Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.
  - b The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Action Classes dialog box. Select all the standard actions you want to add to the action list and click OK.
 

or
  - c Create a new action of your own: right-click and choose New Action.



- 3 Set the properties of each action in the Object Inspector. (The properties you set affect every client of the action.)

The *Name* property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible*, and *Execute*) correspond to the properties and events of its client controls. The client's corresponding properties are typically, but not necessarily, the same name as the corresponding client property. For example, an action's *Enabled* property corresponds to a *TToolButton*'s *Enabled* property. However, an action's *Checked* property corresponds to a *TToolButton*'s *Down* property.

- 4 If you use the predefined actions, the action includes a standard response that occurs automatically. If creating your own action, you need to write an event handler that defines how the action responds when fired. See “What happens when an action fires” on page 6-24 for details.
- 5 Attach the actions in the action list to the clients that require them:
  - Click on the control (such as the button or menu item) on the form or data module. In the Object Inspector, the *Action* property lists the available actions.
  - Select the one you want.

The standard actions, such as *TEditDelete* or *TDataSetPost*, all perform the action you would expect. You can look at the online reference Help for details on how all of the standard actions work if you need to. If writing your own actions, you'll need to understand more about what happens when the action is fired.

## What happens when an action fires

---

When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.

### Responding with events

When a client component or control is clicked or otherwise acted on, a series of events occurs to which you can respond. For example, the following code illustrates the event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
  { Toggle Toolbar1's visibility }
  Toolbar1.Visible := not Toolbar1.Visible;
end;
```

- Note** For general information about events and event handlers, see “Working with events and event handlers” on page 25.

You can supply an event handler that responds at one of three different levels: the action, the action list, or the application. This is only a concern if you are using a new generic action rather than a predefined standard action. You do not have to worry about this if using the standard actions because standard actions have built-in behavior that executes when these events occur.

The order in which the event handlers will respond to events is as follows:

- Action list
- Application
- Action

When the user clicks on a client control, Delphi calls the action's `Execute` method which defers first to the action list, then the `Application` object, then the action itself if neither action list nor `Application` handles it. To explain this in more detail, Delphi follows this dispatching sequence when looking for a way to respond to the user action:

- 1 If you supply an *OnExecute* event handler for the action list and it handles the action, the application proceeds.

The action list's event handler has a parameter called *Handled*, that returns *False* by default. If the handler is assigned and it handles the event, it returns *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    Handled := True;
end;
```

If you don't set *Handled* to *True* in the action list event handler, then processing continues.

- 2 If you did not write an *OnExecute* event handler for the action list or if the event handler doesn't handle the action, the application's *OnActionExecute* event handler fires. If it handles the action, the application proceeds.

The global *Application* object receives an *OnActionExecute* event if any action list in the application fails to handle an event. Like the action list's *OnExecute* event handler, the *OnActionExecute* handler has a parameter *Handled* that returns *False* by default. If an event handler is assigned and handles the event, it returns *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
    { Prevent execution of all actions in Application }
    Handled := True;
end;
```

- 3 If the application's *OnExecute* event handler doesn't handle the action, the action's *OnExecute* event handler fires.

You can use built-in actions or create your own action classes that know how to operate on specific target classes (such as edit controls). When no event handler is found at any level, the application next tries to find a target on which to execute the action. When the application locates a target that the action knows how to address, it invokes the action. See the next section for details on how the application locates a target that can respond to a predefined action class.

## How actions find their targets

“What happens when an action fires” on page 6-24 describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

The application looks for the target using the following sequence:

- 1 **Active control:** The application looks first for an active control as a potential target.
- 2 **Active form:** If the application does not find an active control or if the active control can't act as a target, it looks at the screen's *ActiveForm*.
- 3 **Controls on the form:** If the active form is not an appropriate target, the application looks at the other controls on the active form for a target.

If no target is located, nothing happens when the event is fired.

Some controls can expand the search to defer the target to an associated component; for example, data-aware controls defer to the associated dataset component. Also, some predefined actions do not use a target; for example, the File Open dialog.

## Updating actions

---

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is “checked” when the toolbar is visible:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
    { Indicate whether ToolBar1 is currently visible }
    (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

**Warning** Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

## Predefined action classes

---

You can add predefined actions to your application by right-clicking on the Action Manager and choosing New Standard Action. The New Standard Action Classes dialog box is displayed listing the predefined action classes and the associated standard actions. These are actions that are included with Delphi and they are objects

that automatically perform actions. The predefined actions are organized within the following classes:

**Table 6.3** Action classes

Class	Description
Edit	Standard edit actions: Used with an edit control target. <i>TEditAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> method to implement copy, cut, and paste tasks by using the clipboard.
Format	Standard formatting actions: Used with rich text to apply text formatting options such as bold, italic, underline, strikeout, and so on. <i>TRichEditAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> and <i>UpdateTarget</i> methods to implement formatting of the target.
Help	Standard Help actions: Used with any target. <i>THelpAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> method to pass the command onto a Help system.
Window	Standard window actions: Used with forms as targets in an MDI application. <i>TWindowAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
File	File actions: Used with operations on files such as File Open, File Run, or File Exit.
Search	Search actions: Used with search options. <i>TSearchAction</i> implements the common behavior for actions that display a modeless dialog where the user can enter a search string for searching an edit control.
Tab	Tab control actions: Used to move between tabs on a tab control such as the Prev and Next buttons on a wizard.
List	List control actions: Used for managing items in a list view.
Dialog	Dialog actions: Used with dialog components. <i>TDialogAction</i> implements the common behavior for actions that display a dialog when executed. Each descendant class represents a specific dialog.
Internet	Internet actions: Used for functions such as Internet browsing, downloading, and sending mail.
DataSet	DataSet actions: Used with a dataset component target. <i>TDataSetAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> and <i>UpdateTarget</i> methods to implement navigation and editing of the target. <i>TDataSetAction</i> introduces a <i>DataSource</i> property that ensures actions are performed on that dataset. If <i>DataSource</i> is nil, the currently focused data-aware control is used.
Tools	Tools: Additional tools such as <i>TCustomizeActionBars</i> for automatically displaying the customization dialog for action bands.

All of the action objects are described under the action object names in the online reference Help. Refer to the Help for details on how they work.

## Writing action components

You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of

object. Then, you can use your custom actions in the same way you use pre-defined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler.

Component writers can use the classes in the `QStdActns` and `DBActns` units as examples for deriving their own action classes to implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction*, *TWindowAction*, and so on) generally override *HandlesTarget*, *UpdateTarget*, and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task. These methods are described here:

Method	Description
<i>HandlesTarget</i>	Called automatically when the user invokes an object (such as a toolbutton or menu item) that is linked to the action. The <i>HandlesTarget</i> method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the <i>Target</i> parameter as a “target”. See “How actions find their targets” on page 6-26 for details.
<i>UpdateTarget</i>	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of <i>OnUpdateAction</i> . See “Updating actions” on page 6-26 for details.
<i>ExecuteTarget</i>	Called automatically when the action fires in response to a user action in place of <i>OnExecute</i> (for example, when the user selects a menu item or presses a tool button that is linked to this action). See “What happens when an action fires” on page 6-24 for details.

## Registering actions

When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the `Actnlist` unit:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);
```

```
procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

When you call *RegisterActions*, the actions you register appear in the Action List editor for use by your applications. You can supply a category name to organize your actions, as well as a *Resource* parameter that lets you supply default property values.

For example, the following code registers the standard actions with the IDE:

```
{ Standard action registration }
RegisterActions('', [TAction], nil);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

When you call *UnRegisterActions*, the actions no longer appear in the Action List editor.

## Creating and managing menus

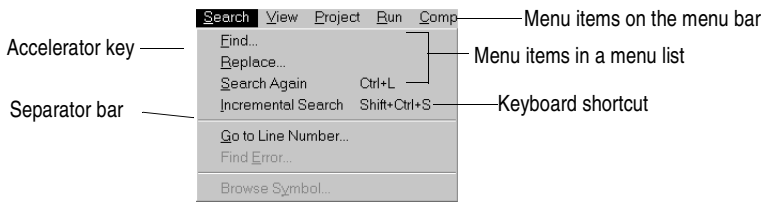
Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This chapter explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer
- Building menus
- Editing menu items in the Object Inspector
- Using the Menu Designer context menu
- Using menu templates
- Saving a menu as a template
- Adding images to menu items

**Figure 6.3** Menu terminology

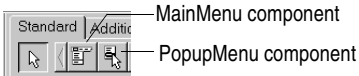


For information about hooking up menu items to the code that executes when they are selected, see “Associating menu events with event handlers” on page 3-28.

## Opening the Menu Designer

You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a MainMenu or PopupMenu component to your form. Both menu components are located on the Standard page of the component palette.

**Figure 6.4** MainMenu and PopupMenu components



A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then either:

- Double-click the menu component.

or

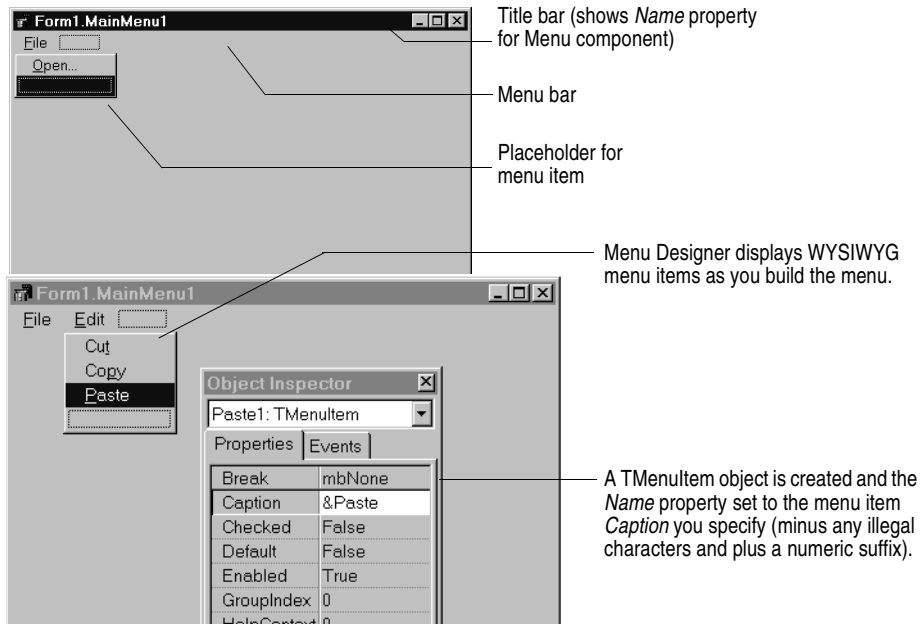
- From the Properties page of the Object Inspector, select the *Items* property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

The Menu Designer appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the Object Inspector.

**Figure 6.5** Menu Designer for a pop-up menu



**Figure 6.6** Menu Designer for a main menu



## Building menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

This section discusses the basics of creating a menu at design time. For more information about menu templates, see “Using menu templates” on page 6-38.



## Naming menus

As with all components, when you add a menu component to the form, Delphi gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows Object Pascal naming conventions.

Delphi adds the menu name to the form's type declaration, and the menu name then appears in the Component list.

## Naming the menu items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type the value for the *Name* property.
- Type the value for the *Caption* property first, and let Delphi derive the *Name* property from the caption.

For example, if you give a menu item a *Caption* property value of *File*, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi leaves the *Caption* property blank until you type a value.

**Note** If you enter characters in the *Caption* property that are not valid for Object Pascal identifiers, Delphi modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Delphi precedes the number with a character to derive the *Name* property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

**Table 6.4** Sample captions and their derived names

Component caption	Derived name	Explanation
&File	File1	Removes ampersand
&File (2nd occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numerical order
1234 (2nd occurrence)	N12342	Adds a number to disambiguate the derived name
\$@@@#	N1	Removes all non-standard characters, adding preceding letter and numerical order
- (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, Delphi adds any menu item names to the form's type declaration, and those names then appear in the Component list.

## Adding, inserting, and deleting menu items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time,

- 1 Select the position where you want to create the menu item.

If you've just opened the Menu Designer, the first position on the menu bar is already selected.

- 2 Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the Object Inspector and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.

- 3 Press *Enter*.

The next placeholder for a menu item is selected.

If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Delphi has filled in the *Name* property based on the value you entered for the caption. (See "Naming the menu items" on page 6-32.)

- 4 Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press *Esc* to return to the menu bar.

Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press *Enter* to complete an action. To return to the menu bar, press *Esc*.

To insert a new, blank menu item,

- 1 Place the cursor on a menu item.
- 2 Press *Ins*.

Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command,

- 1 Place the cursor on the menu item you want to delete.
- 2 Press *Del*.

**Note** You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

## Adding separator bars

Separator bars insert a line between menu items. You can use separator bars to indicate groupings within the menu list, or simply to provide a visual break in a list.

To make the menu item a separator bar, type a hyphen (-) for the caption.

## Specifying accelerator keys and keyboard shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing *Alt+* the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Delphi automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual*.

To specify an accelerator,

- Add an ampersand in front of the appropriate letter.

For example, to add a Save menu command with the *S* as an accelerator key, type &Save.

Keyboard shortcuts enable the user to perform the action without using the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut,

- Use the Object Inspector to enter a value for the *Shortcut* property, or select a key combination from the drop-down list.

This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

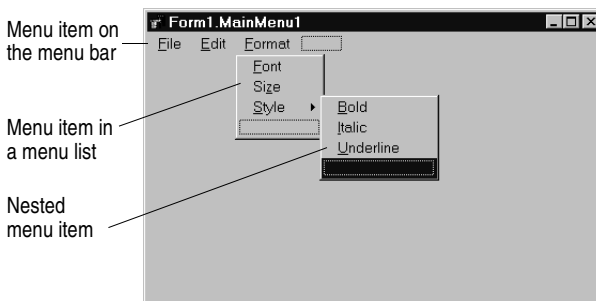
**Caution** Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

## Creating submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Delphi supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

**Figure 6.7** Nested menu structures



To create a submenu,

- 1 Select the menu item under which you want to create a submenu.
- 2 Press *Ctrl*→ to create the first placeholder, or right-click and choose Create Submenu.
- 3 Type a name for the submenu item, or drag an existing menu item into this placeholder.

- 4 Press *Enter*, or ↓, to create the next placeholder.
- 5 Repeat steps 3 and 4 for each item you want to create in the submenu.
- 6 Press *Esc* to return to the previous menu level.

### Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well—moving a menu item into an existing submenu just creates one more level of nesting.

### Moving menu items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a different menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a *different* menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

- 1 Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.
- 2 Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list,

- 1 Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

This causes the menu to open, enabling you to drag the item to its new location.

- 2 Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

### Adding images to menu items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list.

To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu item.

To add an image to a menu item using an image list:

- 1 Drop a *TMainMenu* or *TPopupMenu* object on a form.
- 2 Drop a *TImageList* object on the form.
- 3 Open the ImageList editor by double clicking on the *TImageList* object.
- 4 Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.
- 5 Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.
- 6 Create your menu items and submenu items as described previously.
- 7 Select the menu item you want to have an image in the Object Inspector and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

**Note** Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

## Viewing the menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu,

- 1 If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.
- 2 If the form has more than one menu, select the menu you want to view from the form's *Menu* property drop-down list.

The menu appears in the form exactly as it will when you run the program.

## Editing menu items in the Object Inspector

---

This section has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the Object Inspector, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list in the Object Inspector and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items,

- 1 Switch focus from the Menu Designer window to the Object Inspector by clicking the properties page of the Object Inspector.
- 2 Close the Menu Designer as you normally would.

The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

## Using the Menu Designer context menu

---

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to “Using menu templates” on page 6-38.)

To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.

### Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

**Table 6.5** Menu Designer context menu commands

Menu command	Action
Insert	Inserts a placeholder above or to the left of the cursor.
Delete	Deletes the selected menu item (and all its sub-items, if any).
Create Submenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item.
Select Menu	Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu.
Save As Template	Opens the Save Template dialog box, where you can save a menu for future reuse.
Insert From Template	Opens the Insert Template dialog box, where you can select a template to reuse.
Delete Templates	Opens the Delete Templates dialog box, where you can choose to delete any existing templates.
Insert From Resource	Opens the Insert Menu from Resource file dialog box, where you can choose an .mnu file to open in the current form.

### Switching between menus at design time

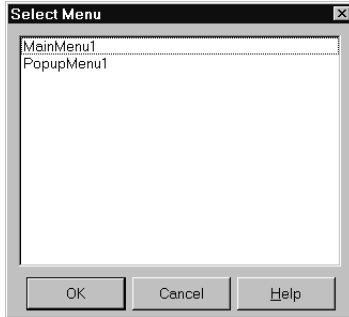
If you’re designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch between menus in a form,

- 1 Right-click in the Menu Designer and choose Select Menu.

The Select Menu dialog box appears.

**Figure 6.8** Select Menu dialog box



This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

- 2 From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form,

- 1 Give focus to the form whose menus you want to choose from.
- 2 From the Component list, select the menu you want to edit.
- 3 On the Properties page of the Object Inspector, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

## Using menu templates

---

Delphi provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates shipped with Delphi are stored in the BIN subdirectory in a default installation. These files have a .DMT (Delphi menu template) extension.

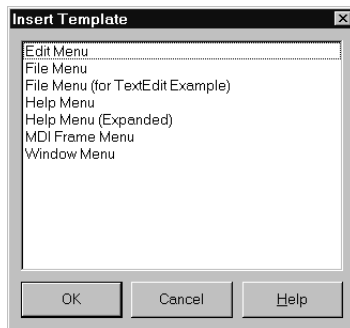
You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

To add a menu template to your application,

- 1 Right-click the Menu Designer and choose Insert From Template.  
(If there are no templates, the Insert From Template option appears dimmed in the context menu.)

The Insert Template dialog box opens, displaying a list of available menu templates.

**Figure 6.9** Sample Insert Template dialog box for menus



- 2 Select the menu template you want to insert, then press *Enter* or choose OK.

This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

- 1 Right-click the Menu Designer and choose Delete Templates.

(If there are no templates, the Delete Templates option appears dimmed in the context menu.)

The Delete Templates dialog box opens, displaying a list of available templates.

- 2 Select the menu template you want to delete, and press *Del*.

Delphi deletes the template from the templates list and from your hard disk.

## Saving a menu as a template

---

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in your BIN subdirectory as .DMT files.

To save a menu as a template,

- 1 Design the menu you want to be able to reuse.

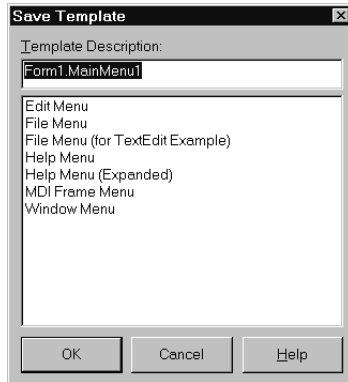
This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.

- 2 Right-click in the Menu Designer and choose Save As Template.



The Save Template dialog box appears.

**Figure 6.10** Save Template dialog box for menus



- 3 In the Template Description edit box, type a brief description for this menu, and then choose OK.

The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

**Note** The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

### Naming conventions for template menu items and event handlers

When you save a menu as a template, Delphi does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Delphi names it *File1*. If you insert it into a menu with an existing menu item named *File1*, Delphi names it *File2*.

Delphi also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Delphi still generates the event handler name.

You can easily associate items in the menu template with existing *OnClick* event handlers in the form. For more information, see “Associating an event with an existing event handler” on page 3-27.

### Manipulating menu items at runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can

insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the *Enabled* property.

For examples that use the menu item's *Visible* and *Enabled* properties, see "Disabling menu items" on page 7-10.

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. The following section discusses this in more detail.

## Merging menus

---

For MDI applications, such as the text editor sample application, and for OLE client applications, your application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*. Note that OLE technology is limited to Windows applications only and is not available for use in cross-platform programming.

You prepare menus for merging by specifying values for two properties:

- *Menu*, a property of the form
- *GroupIndex*, a property of menu items in the menu

### Specifying the active menu: Menu property

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
Form1.Menu := SecondMenu;
```

### Determining the order of merged menu items: GroupIndex property

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:

- Lower numbers appear first (farther left) in the menu.

For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.

- To replace items in the main menu, give items on the child menu the same *GroupIndex* value.

This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well.

Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.

- To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.

For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

## Importing resource files

---

Delphi supports use of menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your Delphi project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load existing .RC menu files,

- 1 In the Menu Designer, place your cursor where you want the menu to appear.

The imported menu can be part of a menu you are designing, or an entire menu in itself.

- 2 Right-click and choose Insert From Resource.

The Insert Menu From Resource dialog box appears.

- 3 In the dialog box, select the resource file you want to load, and choose OK.

The menu appears in the Menu Designer window.

**Note** If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

## Designing toolbars and cool bars

---

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added.

**Note** Cool bars are not available in CLX for cross-platform applications.

You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

You can add a toolbar to a form in several ways:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.

- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.
- Use a cool bar (*TCoolBar*) component and add controls to it. The cool bar displays controls on independently movable and resizable bands.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar and cool bar components, you are ensuring that your application has the look and feel of a Windows application because you are using the native Windows controls. If these operating system controls change in the future, your application could change as well. Also, since the toolbar and cool bar rely on common components in Windows, your application requires the COMCTL32.DLL. Toolbars and cool bars are not supported in WinNT 3.51 applications.

The following sections describe how to

- Add a toolbar and corresponding speed button controls using the panel component
- Add a toolbar and corresponding tool button controls using the Toolbar component
- Add a cool bar using the cool bar component
- Respond to clicks
- Add hidden toolbars and cool bars
- Hide and show toolbars and cool bars

## Adding a toolbar using a panel component

---

To add a toolbar to a form using the panel component,

- 1 Add a panel component to the form (from the Standard page of the component palette).
- 2 Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.
- 3 Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked

- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Add a speed button to a toolbar panel
- Assign a speed button's glyph
- Set the initial condition of a speed button
- Create a group of speed buttons
- Allow toggle buttons

### Adding a speed button to a panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional page of the component palette) on the panel.

The panel, rather than the form, “owns” the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

### Assigning a speed button's glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time,

- 1 Select the speed button.
- 2 In the Object Inspector, select the *Glyph* property.
- 3 Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

### Setting the initial condition of a speed button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

Table 6.6 lists some actions you can set to change a speed button's appearance:

**Table 6.6** Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	<i>GroupIndex</i> property to a value other than zero and its <i>Down</i> property to <i>True</i> .

**Table 6.6** Setting speed buttons' appearance (continued)

To make a speed button:	Set the toolbar's:
Appear disabled	<i>Enabled</i> property to <i>False</i> .
Have a left margin	<i>Indent</i> property to a value greater than 0.

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to *True*.

## Creating a group of speed buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

## Allowing toggle buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a *toggle*. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*.

Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

## Adding a toolbar using the toolbar component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not. To add a toolbar to a form using the toolbar component,

- 1 Add a toolbar component to the form (from the Win32 page of the component palette). The toolbar automatically aligns to the top of the form.
- 2 Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement tool buttons on a toolbar, do the following:

- Add a tool button
- Assign images to tool buttons
- Set the tool buttons' appearance
- Create a group of tool buttons
- Allow toggled tool buttons

### Adding a tool button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar “owns” the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the component palette onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

### Assigning images to tool buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled. To assign images to tool buttons at design time,

- 1 Select the toolbar on which the buttons appear.
- 2 In the Object Inspector, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.
- 3 Select a tool button.
- 4 In the Object Inspector, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

### Setting tool button appearance and initial conditions

Table 6.7 lists some actions you can set to change a tool button's appearance:

**Table 6.7** Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	(on tool button) <i>Style</i> property to <i>tbsCheck</i> and <i>Down</i> property to <i>True</i> .
Appear disabled	<i>Enabled</i> property to <i>False</i> .

**Table 6.7** Setting tool buttons' appearance (continued)

To make a tool button:	Set the toolbar's:
Have a left margin	<i>Indent</i> property to a value greater than 0.
Appear to have "pop-up" borders, thus making the toolbar appear transparent	<i>Flat</i> property to <i>True</i> .

**Note** Using the *Flat* property of *TToolBar* requires version 4.70 or later of COMCTL32.DLL. To force a new row of controls after a specific tool button, select the tool button that you want to appear last in the row and set its *Wrap* property to *True*. To turn off the auto-wrap feature of the toolbar, set the toolbar's *Wrapable* property to *False*.

### Creating groups of tool buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is *False*.
- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

### Allowing toggled tool buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*.

As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.



## Adding a cool bar component

---

**Note** The *TCoolBar* component requires version 4.70 or later of COMCTL32.DLL and is not available in CLX.

The cool bar component (*TCoolBar*)—also called a *rebar*—displays windowed controls on independently movable, resizable bands. The user can position the bands by dragging the resizing grips on the left side of each band.

To add a cool bar to a form in a Windows application,

- 1 Add a cool bar component to the form (from the Win32 page of the component palette). The cool bar automatically aligns to the top of the form.
- 2 Add windowed controls from the component palette to the bar.

Only VCL components that descend from *TWinControl* are windowed controls. You can add graphic controls—such as labels or speed buttons—to a cool bar, but they will not appear on separate bands.

### Setting the appearance of the cool bar

The cool bar component offers several useful configuration options. Table 6.8 lists some actions you can set to change a tool button's appearance:

**Table 6.8** Setting a cool button's appearance

To make the cool bar:	Set the toolbar's:
Resize automatically to accommodate the bands it contains	<i>AutoSize</i> property to <i>True</i> .
Bands maintain a uniform height	<i>FixedSize</i> property to <i>True</i> .
Reorient to vertical rather than horizontal	<i>Vertical</i> property to <i>True</i> . This changes the effect of the <i>FixedSize</i> property.
Prevent the <i>Text</i> properties of the bands from displaying at runtime	<i>ShowText</i> property to <i>False</i> . Each band in a cool bar has its own <i>Text</i> property.
Remove the border around the bar	<i>BandBorderStyle</i> to <i>bsNone</i> .
Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.)	<i>FixedOrder</i> to <i>True</i> .
Create a background image for the cool bar	<i>Bitmap</i> property to <i>TBitmap</i> object.
Choose a list of images to appear on the left of any band	<i>Images</i> property to <i>TImageList</i> object.

To assign images to individual bands, select the cool bar and double-click on the *Bands* property in the Object Inspector. Then select a band and assign a value to its *ImageIndex* property.

## Responding to clicks

---

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For more information, see “Working with events and event handlers” on page 3-25 and “Generating a handler for a component’s default event” on page 3-26.

### Assigning a menu to a tool button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

- 1 Select the tool button.
- 2 In the Object Inspector, assign a pop-up menu (*TPopupMenu*) to the tool button’s *DropDownMenu* property.

If the menu’s *AutoPopup* property is set to *True*, it will appear automatically when the button is pressed.

## Adding hidden toolbars

---

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar,

- 1 Add a toolbar, cool bar, or panel component to the form.
- 2 Set the component’s *Visible* property to *False*.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

## Hiding and showing toolbars

---

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To hide or show a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
    PenBar.Visible := PenButton.Down;
end;
```

## Demo programs

---

For examples of Windows applications that use actions and action lists, refer to Demos\RichEdit. In addition, the Application wizard (File | New Project page), MDI Application, SDI Application, and Winx Logo Applications can use the action and action list objects. For examples of cross-platform applications, refer to Demos\CLX.

# Working with controls

Controls are visual components that the user can interact with at runtime. This chapter describes a variety of features common to many controls.

## Implementing drag-and-drop in controls

---

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another.

- Starting a drag operation
- Accepting dragged items
- Dropping items
- Ending a drag operation
- Customizing drag and drop with a drag object
- Changing the drag mouse pointer

### Starting a drag operation

---

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and, optionally, an integer parameter called *Threshold*. If you pass *True* for *Immediate*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse the number of pixels specified by *Threshold*. Calling

```
BeginDrag False)
```

allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

```

procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    if Button = mbLeft then { drag only if left button pressed }
        with Sender as TFileListBox do { treat Sender as TFileListBox }
            begin
                if ItemAtPos(Point(X, Y), True) >= 0 then { is there an item here? }
                    BeginDrag(False); { if so, drag it }
            end;
    end;

```

## Accepting dragged items

---

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. If *Accept* is *True*, the application sends a drag-and-drop event to the control.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drop. In the following example, a directory tree view accepts dragged items only if they come from a file list box.

```

procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
    Y: Integer; State: TDragState; var Accept: Boolean);
begin
    if Source is TFileListBox then
        Accept := True
    else
        Accept := False;
    end;

```

## Dropping items

---

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components as they are passed over.

In the following example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

```

procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
end;

```

## Ending a drag operation

---

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the item was dragged. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is **nil**, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In this example, a file list box handles an end-drag event by refreshing its file list.

```

procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;

```

## Customizing drag and drop with a drag object

---

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* or *TDragObjectEx* and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-and-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

*TDragObjectEx* descendants are freed automatically whereas descendants of *TDragObject* are not. If you have *TDragObject* descendants that you are not explicitly freeing, you can change them so they descend from *TDragObjectEx* instead to prevent memory loss.

Drag objects let you drag items between a form implemented in the application's main executable file and a form implemented using a DLL, or between forms that are implemented using different DLLs.

## Changing the drag mouse pointer

---

You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property (VCL only).

## Implementing drag-and-dock in controls

---

**Note** Drag and dock properties are available in the VCL but not CLX.

Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel.

- Making a windowed control a docking site
- Making a control a dockable child
- Controlling how child controls are docked
- Controlling how child controls are undocked
- Controlling how child controls respond to drag-and-dock operations

### Making a windowed control a docking site

---

**Note** Drag and dock properties are available in the VCL but not CLX.

To make a windowed control a docking site,

- 1 Set the *DockSite* property to *True*.
- 2 If the dock site object should not appear except when it contains a docked client, set its *AutoSize* property to *True*. When *AutoSize* is *True*, the dock site is sized to 0 until it accepts a child control for docking. Then it resizes to fit around the child control.

### Making a control a dockable child

---

**Note** Drag and dock properties are available in the VCL but not CLX.

To make a control a dockable child,

- 1 Set its *DragKind* property to *dkDock*. When *DragKind* is *dkDock*, dragging the control moves the control to a new docking site or undocks the control so that it becomes a floating window. When *DragKind* is *dkDrag* (the default), dragging the control starts a drag-and-drop operation which must be implemented using the *OnDragOver*, *OnEndDrag*, and *OnDragDrop* events.

- 2 Set its *DragMode* to *dmAutomatic*. When *DragMode* is *dmAutomatic*, dragging (for drag-and-drop or docking, depending on *DragKind*) is initiated automatically when the user starts dragging the control with the mouse. When *DragMode* is *dmManual*, you can still begin a drag-and-dock (or drag-and-drop) operation by calling the *BeginDrag* method.
- 3 Set its *FloatingDockSiteClass* property to indicate the *TWinControl* descendant that should host the control when it is undocked and left as a floating window. When the control is released and not over a docking site, a windowed control of this class is created dynamically, and becomes the parent of the dockable child. If the dockable child control is a descendant of *TWinControl*, it is not necessary to create a separate floating dock site to host the control, although you may want to specify a form in order to get a border and title bar. To omit a dynamic container window, set *FloatingDockSiteClass* to the same class as the control, and it will become a floating window with no parent.

## Controlling how child controls are docked

---

**Note** Drag and dock properties are available in the VCL but not CLX.

A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control).

Three events allow docking sites to further constrain how child controls are docked:

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var InfluenceRect:
TRect; var CanDock: Boolean) of object;
```

*OnGetSiteInfo* occurs on the docking site when the user drags a dockable child over the control. It allows the site to indicate whether it will accept the control specified by the *DockClient* parameter as a child, and if so, where the child must be to be considered for docking. When *OnGetSiteInfo* occurs, *InfluenceRect* is initialized to the screen coordinates of the docking site, and *CanDock* is initialized to *True*. A more limited docking region can be created by changing *InfluenceRect* and the child can be rejected by setting *CanDock* to *False*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer; State:
TDragState; var Accept: Boolean) of object;
```

*OnDockOver* occurs on the docking site when the user drags a dockable child over the control. It is analogous to the *OnDragOver* event in a drag-and-drop operation. Use it to signal that the child can be released for docking, by setting the *Accept* parameter. If the dockable control is rejected by the *OnGetSiteInfo* event handler (perhaps because it is the wrong type of control), *OnDockOver* does not occur.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y: Integer) of object;
```



*OnDockDrop* occurs on the docking site when the user releases the dockable child over the control. It is analogous to the *OnDragDrop* event in a normal drag-and-drop operation. Use this event to perform any necessary accommodations to accepting the control as a child control. Access to the child control can be obtained using the *Control* property of the *TDockObject* specified by the *Source* parameter.

## Controlling how child controls are undocked

---

**Note** Drag and dock properties are available in the VCL but not CLX.

A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:

```
property OnUnDock: TUnDockEvent;  
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean) of  
object;
```

The *Client* parameter indicates the child control that is trying to undock, and the *Allow* parameter lets the docking site (*Sender*) reject the undocking. When implementing an *OnUnDock* event handler, it can be useful to know what other children (if any) are currently docked. This information is available in the read-only *DockClients* property, which is an indexed array of *TControl*. The number of dock clients is given by the read-only *DockClientCount* property.

## Controlling how child controls respond to drag-and-dock operations

---

**Note** Drag and dock properties are available in the VCL but not CLX.

Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.

## Working with text in controls

---

The following sections explain how to use various features of rich edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment
- Adding scrollbars at runtime
- Adding the clipboard object
- Selecting text
- Selecting all text
- Cutting, copying, and pasting text
- Deleting selected text
- Disabling menu items
- Providing a pop-up menu
- Handling the *OnPopup* event

## Setting text alignment

---

In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to.

For example, the following code attaches an *OnClick* event handler to the Character | Left menu item, then attaches the same event handler to both the Right and Center menu items on the Character menu.

```

procedure TEditForm.AlignClick(Sender: TObject);
begin
  Left1.Checked := False; { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True; { check the item clicked }
  with Editor do { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;

```

## Adding scroll bars at runtime

---

Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word-wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word-wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime,

- 1 Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.
- 2 Set the rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a Character | WordWrap menu item.

```

procedure TEditForm.WordWrap1Click(Sender: TObject);
begin
  with Editor do
    begin
      WordWrap := not WordWrap; { toggle word-wrapping }
      if WordWrap then
        ScrollBars := ssVertical { wrapped requires only vertical }
      else

```

```

    ScrollBars := ssBoth; { unwrapped might need both }
    WordWrap1.Checked := WordWrap; { check menu item to match property }
end;
end;

```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

## Adding the clipboard object

---

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. The *Clipboard* object in Delphi encapsulates a clipboard (such as the Windows Clipboard) and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the *Clipbrd* unit.

To add the *Clipboard* object to an application,

- 1 Select the unit that will use the clipboard.
- 2 Search for the `implementation` reserved word.
- 3 Add *Clipbrd* to the `uses` clause below `implementation`.
  - If there is already a `uses` clause in the `implementation` part, add *Clipbrd* to the end of it.
  - If there is not already a `uses` clause, add one that says

```
uses Clipbrd;
```

For example, in an application with a child window, the `uses` clause in the unit's `implementation` part might look like this:

```
uses
  MDIframe, Clipbrd;
```

## Selecting text

---

Before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

Table 7.1 lists properties commonly used to handle selected text.

**Table 7.1** Properties of selected text

Property	Description
<i>SelText</i>	Contains a string representing the selected text in the component.
<i>SelLength</i>	Contains the length of a selected string.
<i>SelStart</i>	Contains the starting position of a string.

## Selecting all text

---

The *SelectAll* method selects the entire contents of the rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a rich edit or memo control, call the *RichEdit1* control's *SelectAll* method.

For example,

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
    RichEdit1.SelectAll; { select all text in RichEdit }
end;
```

## Cutting, copying, and pasting text

---

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard. (See "Using the clipboard with graphics" on page 8-21 for information on using the clipboard with graphics.)

To cut, copy, or paste text with the clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the Edit | Cut, Edit | Copy, and Edit | Paste commands, respectively:

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
    Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
    Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
    Editor.PasteFromClipboard;
end;
```

## Deleting selected text

---

You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

```
procedure TEditForm.Delete(Sender: TObject);
begin
    RichEdit1.ClearSelection;
end;
```

## Disabling menu items

---

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to *False*.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *RichEdit1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

```

procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean; { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT); {enable or disable the Paste
                                                    menu item}
  HasSelection := Editor.SelLength > 0; { True if text is selected }
  Cut1.Enabled := HasSelection; { enable menu items if HasSelection is True }
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;

```

The *HasFormat* method of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. By calling *HasFormat* with the parameter *CF\_TEXT*, you can determine whether the clipboard contains any text, and enable or disable the Paste item as appropriate.

Chapter 8, “Working with graphics and multimedia” provides more information about using the clipboard with graphics.

## Providing a pop-up menu

---

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's *PopupMenu* property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form,

- 1 Place a pop-up menu component on the form.

- 2 Use the Menu Designer to define the items for the pop-up menu.
- 3 Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.
- 4 Attach handlers to the *OnClick* events of the pop-up menu items.

## Handling the OnPopup event

---

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu, as described in “Disabling menu items” on page 7-10.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

To adjust menu items on a pop-up menu before displaying them,

- 1 Select the pop-up menu component.
- 2 Attach an event handler to its *OnPopup* event.
- 3 Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *Edit1Click* event handler described previously in “Disabling menu items” on page 7-10 is attached to the pop-up menu component’s *OnPopup* event. A line of code is added to *Edit1Click* for each item in the pop-up menu.

```

procedure TEditForm.Edit1Click(Sender: TObject);
var
    HasSelection: Boolean;
begin
    Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
    Paste2.Enabled := Paste1.Enabled;{Add this line}
    HasSelection := Editor.SelLength <> 0;
    Cut1.Enabled := HasSelection;
    Cut2.Enabled := HasSelection;{Add this line}
    Copy1.Enabled := HasSelection;
    Copy2.Enabled := HasSelection;{Add this line}
    Delete1.Enabled := HasSelection;
end;

```

## Adding graphics to controls

---

Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and tool bars. Instead of using the standard method of drawing a control or its items, the control’s owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see “Adding images to menu items” on page 6-35.

All owner-draw controls contain lists of items. Usually, those lists are lists of strings that are displayed as text, or lists of objects that contain strings that are displayed as text. You can associate an object with each item in a list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in Delphi involves these steps:

- 1 Indicating that a control is owner-drawn
- 2 Adding graphical objects to a string list
- 3 Drawing owner-drawn items

## Indicating that a control is owner-drawn

---

To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and tool bars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as "OnCustomDraw" or "OnAdvancedCustomDraw".

Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing. List views and tab controls have a property called *OwnerDraw* that enables or disabled the default drawing.

List boxes and combo boxes have additional owner-draw styles, called *fixed* and *variable*, as Table 7.2 describes. Other controls are always fixed, although the size of the item that contains the text may vary, the size of each item is determined before drawing the control.

**Table 7.2** Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the <i>ItemHeight</i> property.	<i>lbOwnerDrawFixed</i> , <i>csOwnerDrawFixed</i>
Variable	Each item might have a different height, determined by the data at runtime.	<i>lbOwnerDrawVariable</i> , <i>csOwnerDrawVariable</i>

## Adding graphical objects to a string list

---

Every string list has the ability to hold a list of objects in addition to its list of strings.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

## Adding images to an application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example, you can store bitmaps for owner-draw controls in hidden image controls, like this:

- 1 Add image controls to the main form.
- 2 Set their *Name* properties.
- 3 Set the *Visible* property for each image control to *False*.
- 4 Set the *Picture* property of each image to the desired bitmap using the Picture editor from the Object Inspector.

The image controls are invisible when you run the application.

## Adding images to a string list

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate* event handler looks like this:

```

procedure TFMForm.FormCreate(Sender: TObject);
var
    Drive: Char;
    AddedIndex: Integer;
begin
    for Drive := 'A' to 'Z' do { iterate through all possible drives }
        begin
            case GetDriveType(Drive + ':/') of { positive values mean valid drives }
                DRIVE_REMOVABLE: { add a tab }
                    AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
                DRIVE_FIXED: { add a tab }
                    AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
                DRIVE_REMOTE: { add a tab }
                    AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
            end;
            if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then { current drive? }
                DriveTabSet.TabIndex := AddedIndex; { then make that current tab }
            end;
        end;
end;

```

## Drawing owner-drawn items

When you indicate that a control is owner-drawn, either by setting a property or supplying a custom draw event handler, the control is no longer drawn on the screen. Instead, the operating system generates events for each visible item in the control. Your application handles the events to draw the items.



To draw the items in an owner-draw control, do the following for each visible item in the control. Use a single event handler for all items.

- 1 Size the item, if needed.

Items of the same size (for example, with a list box style of *lsOwnerDrawFixed*), do not require sizing.

- 2 Draw the item.

## Sizing owner-draw items

---

Before giving your application the chance to draw each item in a variable owner-draw control, the operating system generates a measure-item event. The measure-item event tells the application where the item appears on the control.

Delphi determines the size of the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to be the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be big enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. Depending on the control, the name of the event can vary. List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the size of that item. The size is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

```

procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer); { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;

```

**Note** You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

## Drawing owner-draw items

---

When an application needs to draw or redraw an owner-draw control, the operating system generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a whole or subitems.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing typically start with one of the following:

- *OnDraw*, such as *OnDrawItem* or *OnDrawCell*
- *OnCustomDraw*, such as *OnCustomDrawItem*
- *OnAdvancedCustomDraw*, such as *OnAdvancedCustomDrawItem*

The draw-item event contains parameters identifying the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

```

procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
    begin
      Draw(R.Left, R.Top + 4, Bitmap); { draw bitmap }
      TextOut(R.Left + 2 + Bitmap.Width, { position text }
        R.Top + 2, DriveTabSet.Tabs[Index]); { and draw it to the right of the
        bitmap }
    end;
end;

```



# Working with graphics and multimedia

Graphics and multimedia elements can add polish to your applications. Delphi offers a variety of ways to introduce these features into your application. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, Delphi includes special components that can play audio and video clips. Note that multimedia components are not available for cross-platform programming.

## Overview of graphics programming

---

The VCL graphics components defined in the Graphics unit encapsulate the Windows Graphics Device Interface (GDI), making it easy to add graphics to your Windows applications. CLX graphics components defined in the QGraphics unit encapsulate the Qt graphics widgets for adding graphics to cross-platform applications.

To draw graphics in a Delphi application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it takes care of device context, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles (drawings in CLX). Canvases are available only at runtime, so you do all your work with canvases by writing code.

**VCL Note** Since *TCanvas* is a wrapper resource manager around the Windows device context, you can also use all Windows GDI functions on the canvas. The *Handle* property of the canvas is the device context Handle.

**CLX Note** *TCanvas* is a wrapper resource manager around a Qt painter. The *Handle* property of the canvas is typed pointer to an instance of a Qt painter object. Having this instance pointer exposed allows you to use low-level Qt graphics library functions that require an instance pointer to a painter object.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* message (VCL) or event (CLX).

When working with graphics, you often encounter the terms *drawing* and *painting*:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

The examples in the beginning of this chapter demonstrate how to draw various graphics, but they do so in response to *OnPaint* events. Later sections show how to do the same kind of drawing in response to other events.

## Refreshing the screen

---

At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates WM\_PAINT messages on Windows, which the VCL routes to *OnPaint* events. (If you are using CLX for cross-platform development, a paint event is generated, which CLX routes to *OnPaint* events.) If you have written an *OnPaint* event handler for that object, it is called when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics or if using the VCL, you want to paint a background on a form.

While some operating systems automatically handle the redrawing of the client area of a window that has been invalidated, Windows does not. In the Windows operating system anything drawn on the screen is permanent. When a form or control is temporarily obscured, for example during window dragging, the form or control must repaint the obscured area when it is re-exposed. For more information about the WM\_PAINT message, see the Windows online Help.

If you use the *TImage* control to display a graphical image on a form, the painting and refreshing of the graphic contained in the *TImage* is handled automatically. The *Picture* property specifies the actual bitmap, drawing, or other graphic object that

*TImage* displays. You can also set the *Proportional* property to ensure that the image can be fully displayed in the image control without any distortion. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*'s canvas maps directly onto the screen device (VCL) or the painter (CLX), so that anything drawn to the *PaintBox*'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for the image to be repainted each time the client area is invalidated.

## Types of graphic objects

---

The VCL/CLX provides the graphic objects shown in Table 8.1. These objects have methods to draw on the canvas, which are described in "Using Canvas methods to draw graphic objects" on page 8-9 and to load and save to graphics files, as described in "Loading and saving graphics files" on page 8-18.

**Table 8.1** Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the <i>Picture Register</i> method. Use this to handle arbitrary files such as displaying images in an image control.
Bitmap	A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the <i>handle</i> is copied, not the image.
Clipboard	Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the clipboard; manage and manipulate formats for objects in the clipboard.
Icon	Represents the value loaded from an icon file (::ICO file).
Metafile (VCL only) Drawing (CLX only)	Contains a file that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Metafiles or drawings are extremely scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, metafiles and drawings do not display as fast as bitmaps. Use a metafile or drawing when versatility or precision is more important than performance.

---

## Common properties and methods of Canvas

---

Table 8.2 lists the commonly used properties of the Canvas object. For a complete list of properties and methods, see the *TCanvas* component in online Help.

**Table 8.2** Common properties of the Canvas object

Properties	Descriptions
Font	Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font.
Brush	Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas.
Pen	Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen.
PenPos	Specifies the current drawing position of the pen.
Pixels	Specifies the color of the area of pixels within the current ClipRect.

These properties are described in more detail in “Using the properties of the Canvas object” on page 8-5.

Table 8.3 is a list of several methods you can use:

**Table 8.3** Common methods of the Canvas object

Method	Descriptions
Arc	Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle.
Chord	Draws a closed figure represented by the intersection of a line and an ellipse.
CopyRect	Copies part of an image from another canvas into the canvas.
Draw	Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y).
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
FloodFill (VCL only)	Fills an area of the canvas using the current brush.
FrameRect	Draws a rectangle using the Brush of the canvas to draw the border.
LineTo	Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y).
MoveTo	Changes the current drawing position to the point (X,Y).
Pie	Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas.

**Table 8.3** Common methods of the Canvas object (continued)

Method	Descriptions
Polygon	Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point.
Polyline	Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points.
Rectangle	Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use <i>Rectangle</i> to draw a box using Pen and fill it using Brush.
RoundRect	Draws a rectangle with rounded corners on the canvas.
StretchDraw	Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.
TextHeight, TextWidth	Returns the height and width, respectively, of a string in the current font. Height includes leading between lines.
TextOut	Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string.
TextRect	Writes a string inside a region; any portions of the string that fall outside the region do not appear.

These methods are described in more detail in “Using Canvas methods to draw graphic objects” on page 8-9.

## Using the properties of the Canvas object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This section describes

- Using pens
- Using brushes
- Reading and setting pixels

### Using pens

The *Pen* property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change: *Color*, *Width*, *Style*, and *Mode*.

- Color property: Changes the pen color
- Width property: Changes the pen width
- Style property: Changes the pen style
- Mode property: Changes the pen mode



The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

You can use *TPenRecall* for quick saving off and restoring the properties of pens.

## Changing the pen color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
    Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

## Changing the pen width

A pen's width determines the thickness, in pixels, of the lines it draws.

**Note** When the thickness is greater than 1, Windows 95/98 always draw solid lines, regardless of the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's *Width* property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
    Canvas.Pen.Width := PenWidth.Position; { set the pen width directly }
    PenSize.Caption := IntToStr(PenWidth.Position); { convert to string for caption }
end;
```

## Changing the pen style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

**VCL Note** If developing a cross-platform application for deployment under Windows, Windows 95/98 does not support dashed or dotted line styles for pens wider than one pixel and makes all larger pens solid, no matter what style you specify.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

- 1 Select all six pen-style buttons and select the Object Inspector | Events | *OnClick* event and in the Handler column, type *SetPenStyle*.

Delphi generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.

- 2 Fill in the click-event handler by setting the pen's style depending on the value of *Sender*, which is the control that sent the click event:

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
    else if Sender = DashPen then Style := psDash
    else if Sender = DotPen then Style := psDot
    else if Sender = DashDotPen then Style := psDashDot
    else if Sender = DashDotDotPen then Style := psDashDotDot
    else if Sender = ClearPen then Style := psClear;
  end;
end;
```

## Changing the pen mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on. See *TPen* in online Help for details.

## Getting the pen position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its *PenPos* property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the *MoveTo* method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

**Note** Drawing a line with the *LineTo* method also moves the current position to the endpoint of the line.

## Using brushes

The *Brush* property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- Color property: Changes the fill color
- Style property: Changes the brush style
- Bitmap property: Uses a bitmap as a brush pattern

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

You can use *TBrushRecall* for quick saving off and restoring the properties of brushes.

### Changing the brush color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar (see "Changing the pen color" on page 8-6):

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
    Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

### Changing the brush style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its *Style* property to one of the predefined values: *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, or *bsDiagCross*.

This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight buttons are selected, the Object Inspector | Events | *OnClick* is set, and the *OnClick* handler is named *SetBrushStyle*. Here is the handler code:

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
    with Canvas.Brush do
    begin
        if Sender = SolidBrush then Style := bsSolid
        else if Sender = ClearBrush then Style := bsClear
```

```

else if Sender = HorizontalBrush then Style := bsHorizontal
else if Sender = VerticalBrush then Style := bsVertical
else if Sender = FDiagonalBrush then Style := bsFDiagonal
else if Sender = BDiagonalBrush then Style := bsBDiagonal
else if Sender = CrossBrush then Style := bsCross
else if Sender = DiagCrossBrush then Style := bsDiagCross;
end;
end;

```

## Setting the Brush Bitmap property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

```

var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;

```

**Note** The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remains valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

## Reading and setting pixels

You will notice that every canvas has an indexed *Pixels* property that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color.

**Note** Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the *TBitmap.ScanLine* property.

## Using Canvas methods to draw graphic objects

---

This section shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines

- Drawing shapes
- Drawing rounded rectangles
- Drawing polygons

## Drawing lines and polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

### Drawing lines

To draw a straight line on a canvas, use the *LineTo* method of the canvas.

*LineTo* draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    MoveTo(0, 0);
    LineTo(ClientWidth, ClientHeight);
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
  end;
end;
```

### Drawing polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the *Polyline* method of the canvas.

The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the *MoveTo* method and the *LineTo* method because it eliminates a lot of call overhead.

The following method, for example, draws a rhombus in a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
    Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
  end;
```

This example takes advantage of Delphi's ability to create an open-array parameter on-the-fly. You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter. For more information, see online Help.

## Drawing shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This section covers:

- Drawing rectangles and ellipses
- Drawing rounded rectangles
- Drawing polygons

### Drawing rectangles and ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's *Rectangle* method or *Ellipse* method, passing the coordinates of a bounding rectangle.

The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
  Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

### Drawing rounded rectangles

To draw a rounded rectangle on a canvas, call the canvas's *RoundRect* method.

The first four parameters passed to *RoundRect* are a bounding rectangle, just as for the *Rectangle* method or the *Ellipse* method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

The following method, for example, draws a rounded rectangle in a form's upper left quadrant, rounding the corners as sections of a circle with a diameter of 10 pixels:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

### Drawing polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

*Polygon* takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

For example, the following code draws a right triangle in the lower left half of a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
    Point(ClientWidth, ClientHeight)]);
end;
```

## Handling multiple drawing objects in your application

---

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use
- Changing the tool with speed buttons
- Using drawing tools

### Keeping track of which drawing tool to use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time. You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won't be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Object Pascal provides a means to handle both of these shortcomings. You can declare an enumerated type.

An enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Object Pascal's type-checking to ensure that you assign only those specific values.

To declare an enumerated type, use the reserved `work` type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

For example, the following code declares an enumerated type for each drawing tool available in a graphics application:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

By convention, type identifiers begin with the letter *T*, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for “drawing tool”).

The declaration of the `TDrawingTool` type is equivalent to declaring a group of constants:

```
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

The main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use Object Pascal's type-checking to prevent many errors. A variable of type `TDrawingTool` can be assigned only one of the constants `dtLine`..`dtRoundRect`. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form's drawing tool:

```

type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
  TForm1 = class(TForm)
    ...{ method declarations }
  public
    Drawing: Boolean;
    Origin, MovePt: TPoint;
    DrawingTool: TDrawingTool;{ field to hold current tool }
  end;

```

## Changing the tool with speed buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

```

procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;

```

## Using drawing tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.



This section describes

- Drawing shapes
- Sharing code among event handlers

### Drawing shapes

Drawing shapes is just as easy as drawing lines: Each one takes a single statement; you just need the coordinates.

Here's a rewrite of the *OnMouseDown* event handler that draws shapes for all four tools:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState;
                                X, Y: Integer);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
        Canvas.LineTo(X, Y)
      end;
    dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
    dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
    dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                                   (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
  Drawing := False;
end;
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
    begin
      Canvas.Pen.Mode := pmNotXor;
      case DrawingTool of
        dtLine: begin
                    Canvas.MoveTo(Origin.X, Origin.Y);
                    Canvas.LineTo(MovePt.X, MovePt.Y);
                    Canvas.MoveTo(Origin.X, Origin.Y);
                    Canvas.LineTo(X, Y);
                  end;
        dtRectangle: begin
                    Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                    Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                  end;
        dtEllipse: begin
                    Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                    Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                  end;
        dtRoundRect: begin
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                  end;
      end;
    end;
```

```

        end;
    end;
    MovePt := Point(X, Y);
end;
Canvas.Pen.Mode := pmCopy;
end;

```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next section shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

## Sharing code among event handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form,

### 1 Add the method declaration to the form object.

You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.

### 2 Write the method implementation in the implementation part of the form unit.

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

The following code adds a method to the form called *DrawShape* and calls it from each of the handlers. First, the declaration of *DrawShape* is added to the form object's declaration:

```

type
    TForm1 = class(TForm)
        ...{ fields and methods declared here}
    public
        { Public declarations }
        procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
    end;

```

Then, the implementation of *DrawShape* is written in the implementation part of the unit:

```

implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
    with Canvas do
    begin
        Pen.Mode := AMode;
        case DrawingTool of
            dtLine:
                begin
                    MoveTo(TopLeft.X, TopLeft.Y);

```

```
        LineTo(BottomRight.X, BottomRight.Y);
    end;
    dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
    dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
    dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
        (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
    end;
end;
end;
```

The other event handlers are modified to call *DrawShape*.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
    Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if Drawing then
    begin
        DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
        MovePt := Point(X, Y);{ record the current point }
        DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
    end;
end;
```

## Drawing on a graphic

---

You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses an image control component to display the graphic on a form.

Once you move the application's drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes that have been installed such as jpeg graphics.

**Note** Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

## Making scrollable graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it,

you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

## Adding an image control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

**Note** “Adding graphics to controls” on page 7-11 shows how to use graphics in controls.

## Placing the control

You can place an image control anywhere on a form. If you take advantage of the image control’s ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form’s client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image’s picture. Then set the image control’s properties.

## Setting the initial bitmap size

When you place an image control, it is simply a container. However, you can set the image control’s *Picture* property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in “Loading and saving graphics files” on page 8-18.

To create a blank bitmap when the application starts,

- 1 Attach a handler to the *OnCreate* event for the form that contains the image.
- 2 Create a bitmap object, and assign it to the image control’s *Picture.Graphic* property.

In this example, the image is in the application’s main form, *Form1*, so the code attaches a handler to *Form1*’s *OnCreate* event:

```

procedure TForm1.FormCreate(Sender: TObject);
var
    Bitmap: TBitmap; { temporary variable to hold the bitmap }
begin
    Bitmap := TBitmap.Create; { construct the bitmap object }
    Bitmap.Width := 200; { assign the initial width... }
    Bitmap.Height := 200; { ...and the initial height }
    Image.Picture.Graphic := Bitmap; { assign the bitmap to the image control }
    Bitmap.Free; {We are done with the bitmap, so free it }
end;

```

Assigning the bitmap to the picture’s *Graphic* property copies the bitmap to the picture object. However, the picture object does not take ownership of the bitmap, so after making the assignment, you must free it.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

### Drawing on the bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

```
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
  x,y : integer;
  Bitmap : TBitmap;
  P : PByteArray;
begin
  Bitmap := TBitmap.create;
  try
    Bitmap.LoadFromFile('C:\Program Files\Borland\Delphi 4\Images\Splash\256color\
factory.bmp');
    for y := 0 to Bitmap.height -1 do
      begin
        P := Bitmap.ScanLine[y];
        for x := 0 to Bitmap.width -1 do
          P[x] := y;
        end;
      canvas.draw(0,0,Bitmap);
    finally
      Bitmap.free;
    end;
  end;
```

## Loading and saving graphics files

---

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again.

The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in the following sections:

- Loading a picture from a file
- Saving a picture to a file
- Replacing the picture

## Loading a picture from a file

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:

```

procedure TForm1.Open1Click(Sender: TObject);
begin
    if OpenPictureDialog1.Execute then
        begin
            CurrentFile := OpenPictureDialog1.FileName;
            Image.Picture.LoadFromFile(CurrentFile);
        end;
end;

```

## Saving a picture to a file

The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next section.

The following pair of event handlers, attached to the File | Save and File | Save As menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

```

procedure TForm1.Save1Click(Sender: TObject);
begin
    if CurrentFile <> '' then
        Image.Picture.SaveToFile(CurrentFile){ save if already named }
    else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
    if SaveDialog1.Execute then{ get a file name }
        begin

```

```

    CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
    Save1Click(Sender);{ then save normally }
end;
end;

```

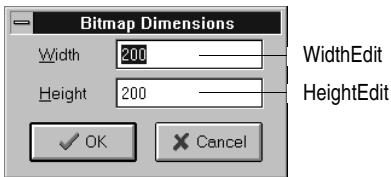
## Replacing the picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic (see "Setting the initial bitmap size" on page 8-17), but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box, such as the one in Figure 8.1.

**Figure 8.1** Bitmap-dimension dialog box from the BMPDlg unit.



This particular dialog box is created in the *BMPDlg* unit included with the *GraphEx* project (in the *EXAMPLES\DOC\GRAPHEX* directory).

With such a dialog box in your project, add it to the uses clause in the unit for your main form. You can then attach an event handler to the File | New menu item's *OnClick* event. Here's an example:

```

procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
  with NewBMPForm do
    begin
      ActiveControl := WidthEdit;{ make sure focus is on width field }
      WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
      HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
      if ShowModal <> idCancel then{ continue if user doesn't cancel dialog box }
        begin
          Bitmap := TBitmap.Create;{ create fresh bitmap object }
          Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
          Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
          Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
          CurrentFile := '';{ indicate unnamed file }
          Bitmap.Free;
        end;
    end;
end;
end;

```

**Note** Assigning a new bitmap to the picture object's *Graphic* property causes the picture object to copy the new graphic, but it does not take ownership of it. The picture object maintains its own internal graphic object. Because of this, the previous code frees the bitmap object after making the assignment.

## Using the clipboard with graphics

---

You can use the Windows clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the clipboard object in your application, you must add the `Clipbrd` (`QClipbrd` in `CLX`) unit to the **uses** clause of any unit that needs to access clipboard data.

For cross-platform applications, data that is stored on the clipboard when using `CLX` is stored as a mime type with an associated *TStream* object. `CLX` provides the following predefined mime source and mime type string constants for the following `CLX` objects:

- `TBitmap` = 'image/delphi.bitmap'
- `TComponent` = 'application/delphi.component'
- `TPicture` = 'image/delphi.picture'
- `TDrawing` = 'image/delphi.drawing'

### Copying graphics to the clipboard

You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications.

To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an Edit | Copy menu item:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

### Cutting graphics to the clipboard

Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original.



In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the Edit | Cut menu item:

```

procedure TForm1.Cut1Click(Sender: TObject);
var
    ARect: TRect;
begin
    Copy1Click(Sender); { copy picture to clipboard }
    with Image.Canvas do
        begin
            CopyMode := cmWhiteness; { copy everything as white }
            ARect := Rect(0, 0, Image.Width, Image.Height); { get bitmap rectangle }
            CopyRect(ARect, Image.Canvas, ARect); { copy bitmap over itself }
            CopyMode := cmSrcCopy; { restore normal mode }
        end;
    end;

```

## Pasting graphics from the clipboard

If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the clipboard,

- 1 Call the clipboard's *HasFormat* method (if using the VCL) or *Provides* method (if using CLX) to see whether the clipboard contains a graphic.

*HasFormat* (or *Provides* in CLX) is a Boolean function. It returns *True* if the clipboard contains an item of the type specified in the parameter. To test for graphics on the Windows platform, you pass *CF\_BITMAP*. In cross-platform applications, you pass *SDelphiBitmap*.

- 2 Assign the clipboard to the destination.

This code shows how to paste a picture from the clipboard into an image control in response to a click on an Edit | Paste menu item:

```

procedure TForm1.PasteButtonClick(Sender: TObject);
var
    Bitmap: TBitmap;
begin
    if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Windows clipboard? }
        begin
            Image1.Picture.Bitmap.Assign(Clipboard);
        end;
    end;

```

The same example in CLX for cross-platform development would look as follows:

```

procedure TForm1.PasteButtonClick(Sender: TObject);
var
    Bitmap: TBitmap;
begin
    if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? }

```

```
begin
    Image1.Picture.Bitmap.Assign(Clipboard);
end;
end;
```

The graphic on the clipboard could come from this application, or it could have been copied from another application, such as Microsoft Paint. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

## Rubber banding example

---

This example describes the details of implementing the “rubber banding” effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The example code in this section is taken from a sample application located in the Demos\DOC\Graphexdirectory. The application draws lines and shapes on a window’s canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

The following topics describe the example:

- Responding to the mouse
- Adding a field to a form object to track mouse actions
- Refining line drawing

### Responding to the mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This section covers:

- What’s in a mouse event
- Responding to a mouse-down action
- Responding to a mouse-up action
- Responding to a mouse move

#### What’s in a mouse event?

The VCL has three mouse events: *OnMouseDown* event, *OnMouseMove* event, and *OnMouseUp* event.

When an application detects a mouse action, it calls whatever event handler you’ve defined for the corresponding event, passing five parameters. Use the information in

those parameters to customize your responses to the events. The five parameters are as follows:

**Table 8.4** Mouse-event parameters

Parameter	Meaning
<i>Sender</i>	The object that detected the mouse action
<i>Button</i>	Indicates which mouse button was involved: <i>mbLeft</i> , <i>mbMiddle</i> , or <i>mbRight</i>
<i>Shift</i>	Indicates the state of the <i>Alt</i> , <i>Ctrl</i> , and <i>Shift</i> keys at the time of the mouse action
<i>X, Y</i>	The coordinates where the event occurred

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check *Button* to determine which mouse button caused the event.

**Note** Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default “primary” and “secondary” mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record *mbLeft* as the value of the *Button* parameter.

### Responding to a mouse-down action

Whenever the user presses a button on the mouse, an *OnMouseDown* event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

The VCL generates an empty handler for a mouse-down event on the form:

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
end;
    
```

### Responding to a mouse-down action

The following code displays the string 'Here!' at the location on a form clicked with the mouse:

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Canvas.TextOut(X, Y, 'Here!');{ write text at (X, Y) }
end;
    
```

When the application runs, you can press the mouse button down with the mouse cursor on the form and have the string, “Here!” appear at the point clicked. This code sets the current drawing position to the coordinates where the user presses the button:

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Canvas.MoveTo(X, Y);{ set pen position }
end;
    
```

Pressing the mouse button now sets the pen position, setting the line's starting point. To draw a line to the point where the user releases the button, you need to respond to a mouse-up event.

### Responding to a mouse-up action

An *OnMouseUp* event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

Here's a simple *OnMouseUp* event handler that draws a line to the point of the mouse-button release:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
```

This code lets a user draw lines by clicking, dragging, and releasing. In this case, the user cannot see the line until the mouse button is released.

### Responding to a mouse move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line to current position }
end;
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

## Adding a field to a form object to track mouse actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Delphi adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Delphi "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

The following code gives a form a field called *Drawing* of type Boolean, in the form object's declaration. It also adds two fields to store points *Origin* and *MovePt* of type TPoint.

```

type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  public
    Drawing: Boolean; { field to track whether button was pressed }
    Origin, MovePt: TPoint; { fields to store points }
  end;

```

When you have a *Drawing* field to track whether to draw, set it to *True* when the user presses the mouse button, and *False* when the user releases it:

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True; { set the Drawing flag }
  Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
  Drawing := False; { clear the Drawing flag }
end;

```

Then you can modify the *OnMouseMove* event handler to draw only when *Drawing* is *True*:

```

procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);

```

```

begin
  if Drawing then{ only draw if Drawing flag is set }
    Canvas.LineTo(X, Y);
end;

```

This results in drawing only between the mouse-down and mouse-up events, but you still get a scribbled line that tracks the mouse movements instead of a straight line.

The problem is that each time you move the mouse, the mouse-move event handler calls *LineTo*, which moves the pen position, so by the time you release the button, you've lost the point where the straight line was supposed to start.

## Refining line drawing

With fields in place to track various points, you can refine an application's line drawing.

### Tracking the origin point

When drawing lines, track the point where the line starts with the *Origin* field.

*Origin* must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
  Canvas.LineTo(X, Y);
  Drawing := False;
end;

```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions--the application does not yet support "rubber banding."

### Tracking movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position*, not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

```

procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
    begin
      Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }

```

```
    Canvas.LineTo(X, Y);  
end;  
end;
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

*MovePt* must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    Drawing := True;  
    Canvas.MoveTo(X, Y);  
    Origin := Point(X, Y);  
    MovePt := Point(X, Y); { keep track of where this move was }  
end;  
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Drawing then  
        begin  
            Canvas.Pen.Mode := pmNotXor; { use XOR mode to draw/erase }  
            Canvas.MoveTo(Origin.X, Origin.Y); { move pen back to origin }  
            Canvas.LineTo(MovePt.X, MovePt.Y); { erase the old line }  
            Canvas.MoveTo(Origin.X, Origin.Y); { start at origin again }  
            Canvas.LineTo(X, Y); { draw the new line }  
        end;  
        MovePt := Point(X, Y); { record point for next move }  
        Canvas.Pen.Mode := pmCopy;  
    end;
```

Now you get a “rubber band” effect when you draw the line. By changing the pen’s mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you’re actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final drawing when you release the mouse button.

## Working with multimedia

---

Delphi allows you to add multimedia components to your applications. To do this, you can use either the *TAnimate* component on the Win32 page or the *TMediaPlayer* component on the System page of the Component palette. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application.

For more information on the *TAnimate* and *TMediaPlayer* components, see the VCL on-line help.

The following topics are discussed in this section:

- Adding silent video clips to an application
- Adding audio and/or video clips to an application

## Adding silent video clips to an application

---

The animation control in Delphi allows you to add silent video clips to your application.

To add a silent video clip to an application:

- 1 Double-click the animate icon on the Win32 page of the Component palette. This automatically puts an animation control on the form window in which you want to display the video clip.
- 2 Using the Object Inspector, select the *Name* property and enter a new *name* for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Delphi identifiers).

Always work directly with the Object Inspector when setting design time properties and creating event handlers.

- 3 Do one of the following:
  - Select the *Common AVI* property and choose one of the AVIs available from the drop down list; or
  - Select the *FileName* property and click the ellipsis (...) button, choose an AVI file from any available local or network directories and click Open in the Open AVI dialog; or
  - Select the resource of an AVI using the *ResName* or *ResID* properties. Use *ResHandle* to indicate the module that contains the resource identified by *ResName* or *ResID*.

This loads the AVI file into memory. If you want to display the first frame of the AVI clip on-screen until it is played using the *Active* property or the *Play* method, then set the *Open* property to *True*.

- 4 Set the *Repetitions* property to the number of times you want to the AVI clip to play. If this value is 0, then the sequence is repeated until the *Stop* method is called.
- 5 Make any other changes to the animation control settings. For example, if you want to change the first frame displayed when animation control opens, then set the *StartFrame* property to the desired frame value.
- 6 Set the *Active* property to *True* using the drop down list or write an event handler to run the AVI clip when a specific event takes place at runtime. For example, to activate the AVI clip when a button object is clicked, write the button's *OnClick* event specifying that. You may also call the *Play* method to specify when to play the AVI.



**Note** If you make any changes to the form or any of the components on the form after setting *Active* to *True*, the *Active* property becomes *False* and you have to reset it to *True*. Do this either just before runtime or at runtime.

## Example of adding silent video clips

Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.

To run this example, create a new project and save the `Unit1.pas` file as `Frmlogo.pas` and save the `Project1.dpr` file as `Logo.dpr`. Then:

- 1 Double-click the animate icon from the Win32 page of the Component palette.
- 2 Using the Object Inspector, set its Name property to `Logo1`.
- 3 Select its `FileName` property, click the ellipsis (...) button, choose the `cool.avi` file from your `..\Demos\Coolstuff` directory. Then click Open in the Open AVI dialog. This loads the `cool.avi` file into memory.
- 4 Position the animation control box on the form by clicking and dragging it to the top right hand side of the form.
- 5 Set its `Repetitions` property to 5.
- 6 Click the form to bring focus to it and set its Name property to `LogoForm1` and its `Caption` property to `Logo Window`. Now decrease the height of the form to right-center the animation control on it.
- 7 Double-click the form's `OnActivate` event and write the following code to run the AVI clip when the form is in focus at runtime:

```
Logo1.Active := True;
```

- 8 Double-click the Label icon on the Standard page of the Component palette. Select its `Caption` property and enter `Welcome to Cool Images 4.0`. Now select its `Font` property, click the ellipsis (...) button and choose `Font Style: Bold`, `Size: 18`, `Color: Navy` from the Font dialog and click OK. Click and drag the label control to center it on the form.
- 9 Click the animation control to bring focus back to it. Double-click its `OnStop` event and write the following code to close the form when the AVI file stops:

```
LogoForm1.Close;
```

- 10 Select Run | Run to execute the animated logo window.

## Adding audio and/or video clips to an application

---

The media player component in Delphi allows you to add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software.

**Note** Audio and video clip support is not provided for cross-platform programming.

To add an audio and/or video clip to an application:

- 1 Double-click the media player icon on the System page of the Component palette. This automatically put a media player control on the form window in which you want the media feature.
- 2 Using the Object Inspector, select the *Name* property and enter a new name for your media player control. You will use this when you call the media player control. (Follow the standard rules for naming Delphi identifiers.)

Always work directly with the Object Inspector when setting design time properties and creating event handlers.

- 3 Select the *DeviceType* property and choose the appropriate device type to open using the *AutoOpen* property or the *Open* method. (If *DeviceType* is *dtAutoSelect* the device type is selected based on the file extension of the media file specified by the *FileName* property.) For more information on device types and their functions, see Table 8.5.
- 4 If the device stores its media in a file, specify the name of the media file using the *FileName* property. Select the *FileName* property, click the ellipsis (...) button, and choose a media file from any available local or network directories and click Open in the Open dialog. Otherwise, insert the hardware the media is stored in (disk, cassette, and so on) for the selected media device, at runtime.
- 5 Set the *AutoOpen* property to *True*. This way the media player automatically opens the specified device when the form containing the media player control is created at runtime. If *AutoOpen* is *False*, the device must be opened with a call to the *Open* method.
- 6 Set the *AutoEnable* property to *True* to automatically enable or disable the media player buttons as required at runtime; or, double-click the *EnabledButtons* property to set each button to *True* or *False* depending on which ones you want to enable or disable.

The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the media player component. The device can also be controlled by the methods that correspond to the buttons (Play, Pause, Stop, Next, Previous, and so on).

- 7 Position the media player control bar on the form by either clicking and dragging it to the appropriate place on the form or by selecting the *Align* property and choosing the appropriate align position from the drop down list.

If you want the media player to be invisible at runtime, set the *Visible* property to *False* and control the device by calling the appropriate methods (*Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *Start Recording*, *Eject*).

- 8 Make any other changes to the media player control settings. For example, if the media requires a display window, set the *Display* property to the control that

displays the media. If the device uses multiple tracks, set the *Tracks* property to the desired track.

**Table 8.5** Multimedia device types and their functions

Device Type	Software/Hardware used	Plays	Uses Tracks	Uses a Display Window
dtAVIVideo	AVI Video Player for Windows	AVI Video files	No	Yes
dtCDAudio	CD Audio Player for Windows or a CD Audio Player	CD Audio Disks	Yes	No
dtDAT	Digital Audio Tape Player	Digital Audio Tapes	Yes	No
dtDigitalVideo	Digital Video Player for Windows	AVI, MPG, MOV files	No	Yes
dtMMMovie	MM Movie Player	MM film	No	Yes
dtOverlay	Overlay device	Analog Video	No	Yes
dtScanner	Image Scanner	N/A for Play (scans images on Record)	No	No
dtSequencer	MIDI Sequencer for Windows	MIDI files	Yes	No
dtVCR	Video Cassette Recorder	Video Cassettes	No	Yes
dtWaveAudio	Wave Audio Player for Windows	WAV files	No	No

### Example of adding audio and/or video clips (VCL only)

This example runs an AVI video clip of a multimedia advertisement for Delphi. To run this example, create a new project and save the Unit1.pas file to FrmAd.pas and save the Project1.dpr file to DelphiAd.dpr. Then:

- 1 Double-click the media player icon on the System page of the Component palette.
- 2 Using the Object Inspector, set the Name property of the media player to *VideoPlayer1*.
- 3 Select its DeviceType property and choose dtAVIVideo from the drop down list.
- 4 Select its FileName property, click the ellipsis (...) button, choose the speedis.avi file from your ..\Demos\Coolstuf directory. Click Open in the Open dialog.
- 5 Set its AutoOpen property to *True* and its Visible property to *False*.
- 6 Double-click the Animate icon from the Win32 page of the Component palette. Set its AutoSize property to *False*, its Height property to *175* and Width property to *200*. Click and drag the animation control to the top left corner of the form.
- 7 Click the media player to bring back focus to it. Select its Display property and choose Animate1 from the drop down list.

- 8 Click the form to bring focus to it and select its Name property and enter *Delphi\_Ad*. Now resize the form to the size of the animation control.
- 9 Double-click the form's *OnActivate* event and write the following code to run the AVI video when the form is in focus:

```
VideoPlayer1.Play;
```

- 10 Choose Run | Run to execute the AVI video.



# Writing multi-threaded applications

Delphi provides several objects that make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by

- **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.
- **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

**Note** Not all operating systems implement true multi-processing, even when it is supported by the underlying hardware. For example, Windows 9x only simulates multiprocessing, even if the underlying hardware supports it.

## Defining thread objects

---

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

**Note** Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the *BeginThread* function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in “Coordinating threads” on page 9-7. For more information on using *BeginThread*, see the online help.

To use a thread object in your application, you must create a new descendant of *TThread*. To create a descendant of *TThread*, choose File | New from the main menu. In the new objects dialog box, select Thread Object. You are prompted to provide a class name for your new thread object. After you provide the name, Delphi creates a new unit file to implement the thread.

**Note** Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog does not automatically prepend a ‘T’ to the front of the class name you provide.

The automatically generated unit file contains the skeleton code for your new thread object. If you named your thread *TMyThread*, it would look like the following:

```

unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.

```

You must fill in the code for the *Execute* method. These steps are described in the following sections.

## Initializing the thread

---

If you want to write initialization code for your new thread class, you must override the *Create* method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

### Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority

thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the *Priority* property.

If writing a Windows application, *Priority* values fall along a seven-point scale, as described in Table 9.1:

**Table 9.1** Thread priorities

Value	Priority
<code>tpIdle</code>	The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with <i>tpIdle</i> priority.
<code>tpLowest</code>	The thread's priority is two points below normal.
<code>tpLower</code>	The thread's priority is one point below normal.
<code>tpNormal</code>	The thread has normal priority.
<code>tpHigher</code>	The thread's priority is one point above normal.
<code>tpHighest</code>	The thread's priority is two points above normal.
<code>tpTimeCritical</code>	The thread gets highest priority.

**Note** If writing a cross-platform application, you must use separate code for assigning priorities on Windows and Linux. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See the CLX version of *TThread* and *Priority* online Help for details.

**Warning** Boosting the thread priority of a CPU intensive operation may “starve” other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application's performance:

```

constructor TMyThread.Create(CreateSuspended: Boolean);
begin
    inherited Create(CreateSuspended);
    Priority := tpIdle;
end;

```

## Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the *FreeOnTerminate* property to *True*.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to *False* and then explicitly freeing the first thread from the second.



## Writing the thread function

---

The *Execute* method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

### Using the main VCL/CLX thread

When you use objects from the VCL or CLX object hierarchies, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main thread is set aside for access of VCL and CLX objects. This is the thread that handles all Windows messages received by components in your application.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's *Synchronize* method. For example:

```

procedure TMyThread.PushTheButton;
begin
    Button1.Click;
end;
:
procedure TMyThread.Execute;
begin
    :
    Synchronize(PushTheButton);
    :
end;

```

*Synchronize* waits for the main thread to enter the message loop and then executes the passed method.

**Note** Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to VCL or CLX objects in console applications.

You do not always need to use the main thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the VCL or CLX thread to enter its message loop. You do not need to use the *Synchronize* method in the following situations:

- Data access components are thread-safe as follows: For BDE-enabled datasets, each thread must have its own database session component. The one exception to this is when you are using Access drivers, which are built using a Microsoft library that is not thread-safe. For dbDirect, as long as the vendor client library is thread-

safe, the dbDirect components will be thread-safe. ADO and InterbaseExpress components are thread-safe.

When using data access components, you must still wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset.

For more information about using database sessions with threads in BDE-enabled applications, see "Managing multiple sessions" on page 20-28.

- VisualCLX objects are not thread-safe.
- DataCLX objects are thread-safe.
- Graphics objects are thread-safe. You do not need to use the main VCL or CLX thread to access *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile* (VCL only), *TDrawing* (CLX only), or *TIcon*. Canvas objects can be used outside the *Synchronize* method by locking them (see "Locking objects" on page 9-7).
- While list objects are not thread-safe, you can use a thread-safe version, *TThreadList*, instead of *TList*.

Call the *CheckSynchronize* routine periodically within the main thread of your application so that background threads can synchronize their execution with the main thread. The best place to call *CheckSynchronize* is when the application is idle (for example, from an *OnIdle* event handler). This ensures that it is safe to make method calls in the background thread.

## Using thread-local variables

Your *Execute* method and any of the routines it calls have their own local variables, just like any other Object Pascal routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section. For example,

```
threadvar
    x : integer;
```

declares an integer type variable that is private to each thread in the application, but global within each thread.

The **threadvar** section can only be used for global variables. Pointer and Function variables can't be thread variables. Types that use copy-on-write semantics, such as long strings don't work as thread variables either.

## Checking for termination by other threads

Your thread begins running when the *Execute* method is called (see "Executing thread objects" on page 9-10) and continues until *Execute* finishes. This reflects the

model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the *Terminated* property. When another thread tries to terminate your thread, it calls the *Terminate* method. *Terminate* sets your thread's *Terminated* property to *True*. It is up to your *Execute* method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

```
procedure TMyThread.Execute;
begin
    while not Terminated do
        PerformSomeTask;
    end;
```

## Handling exceptions in the thread function

The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running.

To catch the exceptions that occur inside your thread function, add a **try...except** block to the implementation of the *Execute* method:

```
procedure TMyThread.Execute;
begin
    try
        while not Terminated do
            PerformSomeTask;
        except
            { do something with exceptions }
        end;
    end;
```

## Writing clean-up code

---

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an *OnTerminate* event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the *Execute* method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main VCL or CLX thread of your application. This has two implications:

- You can't use any thread-local variables in an *OnTerminate* event handler (unless you want the main VCL or CLX thread values).

- You can safely access any components and VCL or CLX objects from the *OnTerminate* event handler without worrying about clashing with other threads.

For more information about the main VCL or CLX thread, see “Using the main VCL/CLX thread” on page 9-4.

## Coordinating threads

---

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

### Avoiding simultaneous access

---

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

#### Locking objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a *Lock* method that prevents other threads from accessing the canvas until the *Unlock* method is called.

The VCL and CLX also both include a thread-safe list object, *TThreadList*. Calling *TThreadList.LockList* returns the list object while also blocking other execution threads from using the list until the *UnlockList* method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

#### Using critical sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of *TCriticalSection*. *TCriticalSection* has two methods, *Acquire* (which blocks other threads from executing the section) and *Release* (which removes the block).

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the *Acquire* method to ensure that no other thread is using it. When finished, threads call the *Release* method so that other threads can access the global memory by calling *Acquire*.

**Warning** Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling *Acquire* can introduce problems of simultaneous access.

For example, consider an application that has a global critical section variable, *LockXY*, that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

## Using the multi-read exclusive-write synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using *TMultiReadExclusiveWriteSynchronizer*. This object acts like a critical section, but allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the *BeginRead* method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the *EndRead* method. Any thread that writes to the protected memory must call *BeginWrite* first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the *EndWrite* method, so that threads waiting to read the memory can begin.

**Warning** Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

## Other techniques for sharing memory

When using objects in the VCL or CLX, use the main thread to execute your code. Using the main thread ensures that the object does not indirectly access any memory that is also used by VCL or CLX objects in other threads. See “Using the main VCL/CLX thread” on page 9-4 for more information on the main thread.

If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See “Using thread-local variables” on page 9-5 for more information about thread-local variables.

## Waiting for other threads

---

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either wait for another thread to completely finish executing, or you can wait for another thread to signal that it has completed a task.

### Waiting for a thread to finish executing

To wait for another thread to finish executing, use the *WaitFor* method of that other thread. *WaitFor* doesn't return until the other thread terminates, either by finishing its own *Execute* method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

```

if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
    end;
    ThreadList1.UnlockList;
  end;
end;

```

In the previous example, the list items were only accessed when the *WaitFor* method indicated that the list was successfully filled. This return value must be assigned by the *Execute* method of the thread that was waited for. However, because threads that call *WaitFor* want to know the result of thread execution, not code that calls *Execute*, the *Execute* method does not return any value. Instead, the *Execute* method sets the *ReturnValue* property. *ReturnValue* is then returned by the *WaitFor* method when it is called by other threads. Return values are integers. Your application determines their meaning.

### Waiting for a task to be completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (*TEvent*) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. *SetEvent* turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the *ResetEvent* method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don't know which thread will finish last, you can't simply use the *WaitFor* method of one of the threads. Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the *OnTerminate* event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

```

procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
  ⋮
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter); { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
    CounterGuard.Release; { release the lock on the counter }
  ⋮
end;

```

The main thread initializes the *Counter* variable, launches the task threads, and waits for the signal that they are all done by calling the *WaitFor* method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from Table 9.2.

**Table 9.2** WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the timeout period elapsed.
wrError	An error occurred while waiting.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```

Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ now continue with the main thread. All task threads have finished }

```

**Note** If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

## Executing thread objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the *Resume* method. To create a thread so that it starts up

immediately, set the constructor's *CreateSuspended* parameter to *False*. For example, the following line creates a thread and starts its execution:

```
SecondProcess := TMyThread.Create(false); {create and run the thread }
```

**Warning** Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

## Overriding the default priority

---

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in "Initializing the thread" on page 9-2. However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

```
SecondProcess := TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.Resume; { now run the thread }
```

**Note** If writing a cross-platform application, you must use separate code for assigning priorities on Windows and Linux. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See the CLX version of *TThread* and *Priority* online Help for details.

## Starting and stopping threads

---

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its *Suspend* method. When it is safe for the thread to resume, call its *Resume* method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*.

You can request that a thread end execution prematurely by calling the *Terminate* method. *Terminate* sets the thread's *Terminated* property to *True*. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is *True*.



## Debugging multi-threaded applications

---

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose View | Threads from the main menu.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

For additional details on the Thread Status box, see online Help.

# Using CLX for cross-platform development

You can use Delphi to develop cross-platform 32-bit applications that run on both the Windows and Linux operating systems. To do this, you can start with an existing Windows application and modify it, or you can create a new application by following the recommended practices for writing platform-independent code. Kylix is Borland's Delphi for Linux software that allows you to compile and develop applications on Linux. If you want to develop and deploy applications on Linux and Windows, you'll need to use Kylix as well as Delphi.

This chapter describes how to change Delphi applications so they will compile on Linux and includes information on the differences between developing applications on Windows and Linux. It also provides guidelines for writing code that is portable between the different environments.

**Note** Most applications developed using CLX (with no operating system specific API calls) will run on both Linux and Windows platforms. The application must be compiled on the platform on which you want it to run.

## Creating cross-platform applications

---

You create cross-platform applications much as you create any Delphi application. You need to use CLX visual components, and you should not use operating system specific APIs if you want the application to be completely cross-platform. (See "Writing portable code" on page 10-17 for tips on writing cross-platform applications.)

To create a cross-platform application:

- 1 In the IDE, choose File | New | CLX application.  
The Component palette shows components that can be used in CLX applications.

**Note** Some Windows only nonvisual components can be used in CLX applications. The Component palette includes ADO, BDE, System, DataSnap, InterBase, Site Express, FastNet, QReport, COM+, BizSnap, and Servers tabs which include functionality that will only work in Windows CLX applications. If you plan to compile your application on Linux as well, do not use the components on these tabs or use **\$IFDEFs** to mark these sections of the code as Windows only.

- 2 Develop your application within the IDE. Remember to use only CLX components in your application.
- 3 Compile and test the application on each platform on which you want to run the application. Review any error messages to see where additional changes need to be made.

When moving an application to Kylix, you need to reset your project options. That's because the .dof file which stores the project options is recreated on Kylix and called .kof (with the default options set). You can also store many of the compiler options with the application by typing Ctrl+O+O. The options are placed at the beginning of the currently open file.

The form file in cross-platform applications will have an extension of xfm instead of dfm. This is to distinguish cross-platform forms that use CLX components from forms that use VCL components. An xfm form file will work on both Windows or Linux but a dfm form only works on Windows.

You could also begin development of a cross-platform application by starting on Kylix instead of Delphi:

- 1 Develop, compile and test the application on Linux using Kylix.
- 2 Move the application source files over to Windows.
- 3 Reset your project options.
- 4 Recompile the application on Windows using Delphi.

For information on writing platform-independent database or internet applications, see "Cross-platform database applications" on page 10-23 and "Cross-platform Internet applications" on page 10-29.

## Porting VCL applications to CLX

---

If you have Delphi applications that were written for the Windows environment, you can make them cross platform. How easy it will be depends on the nature and complexity of the application and how many Windows dependencies there are.

The following sections describe some of the major differences between the Windows and Linux environments and provide guidelines on how to get started porting an application.

## Porting techniques

---

The following are different approaches you can take to port an application from one platform to another:

**Table 10.1** Porting techniques

Technique	Description
Platform-specific port	Targets an operating system and underlying APIs
Cross-platform port	Targets a cross-platform API
Windows emulation	Leave the code alone and port the API it uses

### Platform-specific ports

Platform-specific ports tend to be time-consuming, expensive, and only produce a single targeted result. They create different code bases, which makes them particularly difficult to maintain. However, each port is designed for the specific operating system and can take advantage of platform-specific functionality. So, the application typically runs faster.

### Cross-platform ports

Cross-platform ports generally provide the quickest technique and the ported applications target multiple platforms. In reality, the amount of work involved in developing cross-platform applications is highly dependent on the existing code. If code has been developed without regard for platform independence, you may run into scenarios where platform-independent “logic” and platform-dependent “implementation” are mixed together.

The cross-platform approach is the preferable approach because business logic is expressed in platform-independent terms. Some services are abstracted behind an internal interface that looks the same on all platforms, but has a specific implementation on each. Delphi’s runtime library is an example of this: The interface is very similar on both platforms, although the implementation may be vastly different. You should separate cross-platform parts, then implement specific services on top. In the end, this approach is the least expensive solution, because of reduced maintenance costs due to a largely shared source base and an improved application architecture.

### Windows emulation ports

Windows emulation is the most complex method and it can be very costly, but the resulting Linux application will look most similar to an existing Windows application. This approach involves implementing Windows functionality on Linux. From an engineering point of view, this solution is very hard to maintain.

Where you want to emulate Windows APIs, you can include two distinct sections using `$IFDEFs` to indicate sections of the code that apply specifically to Windows or Linux.

## Porting your application

---

If you are porting an application that you want to run on both Windows and Linux, you need to modify your code or use **\$IFDEFs** to indicate sections of the code that apply specifically to Windows or Linux.

Follow these general steps to port your VCL application to CLX:

- 1 Open the project containing the application you want to change in Delphi.
- 2 Copy .dfm files to .xfm files of the same name (for example, rename unit1.dfm to unit1.xfm). Rename (or **\$IFDEF**) the reference to the .dfm file in the unit file(s) from `{$R *.dfm}` to `{$R *.xfm}`. (The .xfm file will work in both Kylix and Delphi.)

For example, change the form reference in the **implementation** section from

```
{$R *.dfm}
```

to

```
{$R *.xfm}
```

- 3 Change (or **\$IFDEF**) all **uses** clauses so they refer to the correct units in CLX. (See “CLX and VCL unit comparison” on page 10-9 for information.)

For example, change the following **uses** clause in a Windows application

```
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
```

to the following for a CLX application:

```
uses Windows, Messages, SysUtils, Variants, Classes, QForms, QControls, QStdCtrls;
```

- 4 Save the project and reopen it. Now the Component palette shows components that can be used in CLX applications.

**Note** Some Windows only nonvisual components can be used in CLX applications. The Component palette includes ADO, BDE, System, DataSnap, InterBase, Internet Express, Site Express, FastNet, QReport, COM+, Web Services, and Servers tabs which include functionality that will only work in Windows CLX applications. If you plan to compile your application on Linux as well, do not use the components on these tabs or use **\$IFDEFs** to mark these sections of the code as Windows only.

- 5 Rewrite any code that does not require Windows dependencies making the code more platform-independent. Do this using the runtime library routines and constants. (See “Writing portable code” on page 10-17 for information.)
- 6 Find equivalent functionality for features that are different on Linux. Use **\$IFDEFs** (sparingly) to delimit Windows-specific information. (See “Using conditional directives” on page 10-18 for information.)

For example, you can **\$IFDEF** platform-specific code in your source files:

```
[$IFDEF MSWINDOWS]
IniFile.LoadFromFile('c:\x.txt');
[$ENDIF]

[$IFDEF LINUX]
IniFile.LoadFromFile('/home/name/x.txt');
[$ENDIF]
```

**7** Search for references to pathnames in all the project files.

- Pathnames in Linux use a forward slash / as a delimiter (for example, /usr/lib) and files may be located in different directories on the Linux system. Use the PathDelim constant (in SysUtils) to specify the path delimiter that is appropriate for the system. Determine the correct location for any files on Linux.
- Change references to drive letters (for example, C:\) and code that looks for drive letters by looking for a colon at position 2 in the string. Use the DriveDelim constant (in SysUtils) to specify the location in terms that are appropriate for the system.
- In places where you specify multiple paths, change the path separator from semicolon (;) to colon (:). Use the PathSep constant (in SysUtils) to specify the path separator that is appropriate for the system.
- Because file names are case-sensitive in Linux, make sure that your application doesn't change the case of file names or assume a certain case.

**8** Compile, text and debug your application.

To transfer the application to Linux:

- 1 Move your Delphi Windows application source files and other project-related files onto your Linux computer. (You can share source files between Linux and Windows if you want the program to run on both platforms. Or you can transfer the files using a tool such as ftp using the ASCII mode.)

Source files should include your unit files (.pas files), project file (.dpr file), and any package files (.dpk files). Project-related files include form files (.xfm files), resource files (.res files), and project options files (.dof files—in Kylix these change to .kof files). If you want to compile your application from the command line only (rather than using the IDE), you'll need the configuration file (.cfg file—in Kylix this changes to .conf).

- 2 Open the project in Kylix. You will receive warnings on Windows-specific features that are in use.
- 3 Compile the project using Kylix. Review any error messages to see where additional changes need to be made.

## CLX versus VCL

---

Kylix uses the Borland Component Library for Cross Platform (CLX) in place of the Visual Component Library (VCL). Within the VCL, many controls provide an easy way to access Windows controls. Similarly, CLX provides access to Qt widgets (from window + gadget) in the Qt shared libraries. Delphi includes both CLX and the VCL.

CLX looks much like the VCL. Most of the component names are the same, many properties have the same names. In addition, CLX, as well as the VCL, will be available on Windows (check the latest release of Delphi to determine availability).

CLX components can be grouped into the following parts:

**Table 10.2** CLX parts

Part	Description
VisualCLX	Native cross-platform GUI components and graphics. The components in this area may differ on Linux and Windows.
DataCLX	Client data-access components. The components in this area are a subset of the local, client/server, and n-tier based on client datasets. The code is the same on Linux and Windows.
NetCLX	Internet components including Apache DSO and CGI Web Broker. These are the same on Linux and Windows.
RTL	Runtime Library up to and including <code>Classes.pas</code> . The code is the same on Linux and Windows.

Widgets in VisualCLX replace Windows controls. In CLX, *TWidgetControl* replaces the VCL's *TWinControl*. Other components (such as *TScrollingWidget*) have corresponding names. However, you do not need to change occurrences of *TWinControl* to *TWidgetControl*. Type declarations, such as the following

```
TWinControl = TWidgetControl;
```

appear in the `QControls.pas` source file to simplify sharing of source code. *TWidgetControl* and its descendants all have a *Handle* property that is a reference to the Qt object; and a *Hooks* property, which is a reference to the hook objects that handle the event mechanism.

Unit names and locations of some classes are different for CLX. You will need to modify **uses** clauses to eliminate references to units that don't exist in CLX and to change the names to CLX units. (Most project files and the interface sections of most units contain a **uses** clause. The implementation section of a unit can also contain its own **uses** clause.)

## What CLX does differently

Although much of CLX is implemented so that it is consistent with the VCL, some features are implemented differently. This section provides an overview of some of the differences between CLX and VCL implementations to be aware of when writing cross-platform applications.

### Look and feel

The visual environment in Linux looks somewhat different than it does in Windows. The look of dialogs may differ depending on which window manager is in use (for example, if using KDE or Gnome).

### Styles

Application-wide "styles" can be used in addition to the *OwnerDraw* properties. You can use the *TApplication.Style* property to specify the look and feel of an application's graphical elements. Using styles, a widget or an application can take on a whole new look. You can still use owner draw on Linux but using styles is recommended.

## Variants

All of the variant/safe array code that was in System is in two new units:

- Variants.pas
- VarUtils.pas

The operating system dependent code is now isolated in VarUtils.pas, and it also contains generic versions of everything needed by Variants.pas. If you are converting a VCL application that included Windows calls to a CLX application, you need to replace these calls to calls into VarUtils.pas.

If you want to use variants, you must include the Variants unit to your **uses** clause.

*VarIsEmpty* does a simple test against *varEmpty* to see if a variant is clear, and on Linux you need to use the *VarIsClear* function to clear a variant.

## Custom variant data handler

You can define custom data types for variants. This introduces operator overloading while the type is assigned to the variant. To create a new variant type, descend from the class, *TCustomVariantType*, and instantiate your new variant type.

For an example, see VarCmplx.pas. This unit implements complex mathematics support via custom variants. It supports the following variant operations: addition, subtraction, multiplication, division (not integer division), and negation. It also handles conversion to and from: SmallInt, Integer, Single, Double, Currency, Date, Boolean, Byte, OleStr, and String. Any of the float/ordinal conversion will lose any imaginary portion of the complex value.

## Registry

Linux does not use a registry to store configuration information. Instead, you use text configuration files and environment variables instead of using the registry. System configuration files on Linux are often located in /etc, for example, /etc/hosts. Other user profiles are located in hidden files (preceded with a dot), such as .bashrc, which holds bash shell settings or .XDefaults, which is used to set defaults for X programs.

Registry-dependent code may be changed to using a local configuration text file instead stored, for example, in the same directory as the application. Writing a unit containing all the registry functions but diverting all output to a local configuration file is one way you could handle a former dependency on the registry.

To place information in a global location on Linux, you could store a global configuration file in the root directory. This makes it so all of your applications can access the same configuration file. However, you must be sure that the file permissions and access rights are set up correctly.

You can also use ini files in cross-platform applications. However, in CLX, you need to use *TMemIniFile* instead of *TRegIniFile*.

## Other differences

CLX implementation also has some other differences that affect the way your application works. This section describes some of those differences.



*ToggleButton* doesn't get toggled by the Enter key. Pressing Enter doesn't simulate a click event on Kylix as it does in Delphi.

*TColorDialog* does not have a *TColorDialog.Options* property to set. Therefore, you cannot customize the appearance and functionality of the color selection dialog. Also, *TColorDialog* is not always modal. You can manipulate the title bar of an application with a modal dialog on Kylix (that is, you can select the parent form of the color dialog and do things like maximizing it while the color dialog is open).

At runtime, combo boxes work differently on Kylix than they do in Delphi. On Kylix (but not on Delphi), you can add a item to a drop down by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to *ciNone*. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

*TCustomEdit* does not implement *Undo*, *ClearUndo*, or *CanUndo*. So there is no way to programmatically undo edits. But application users can undo their edits in an edit box (*TEdit*) at runtime by right-clicking on the edit box and choosing the Undo command.

The key value in a *OnKeyDown* event or *KeyUp* event for the Enter key on Windows is 13. On Linux, this value is 4100. If you check for a hardcoded numeric value for a key, such as checking for a value of 13 for the Enter key, you need to change this when porting a Delphi application to Kylix.

Additional differences exist. Refer to the CLX online documentation for details on all of the CLX objects or in versions of Delphi that include the source code you can refer to the code, it is located in `..\Delphi6\Source\VCL\CLX`.

## Missing in CLX

---

When using CLX instead of the VCL, many of the objects are the same. However, the objects may be missing some features (such as properties, methods, or events). The following general features are missing in CLX:

- Bi-directional properties (*BidiMode*) for right-to-left text output or input
- Generic bevel properties on common controls (note that some objects still have bevel properties)
- Docking properties and methods
- Backward compatibility features such components on the Win3.1 tab and *Ctl3D*
- *DragCursor* and *DragKind* (but drag and drop is included)

## Features that will not port

---

Some Windows-specific features supported on Delphi will not transport directly to Linux environments. Features, such as COM, ActiveX, OLE, BDE, and ADO are dependent on Windows technology and are not available in Kylix. The following

table lists features that are different on the two platforms and lists the equivalent Kylix feature, if one is available.

**Table 10.3** Changed or different features

Delphi/Windows feature	Kylix/Linux feature
ADO components	Regular database components
Automation Servers	Not available
BDE	dbExpress and regular database components
COM+ components (including ActiveX)	Not available
DataSnap	Not yet available
FastNet	Not available
Internet Express	Not yet available
Legacy components (such as items on the Win 3.1 Component palette tab)	Not available
Messaging Application Programming Interface (MAPI) includes a standard library of Windows messaging functions.	SMTP/POP3 let you send, receive, and save email messages
Quick Reports	Not available
Web Services (SOAP)	Not yet available
WebSnap	Not yet available
Windows API calls	CLX methods, Qt calls, libc calls, or calls to other system libraries
Windows messaging	Qt events
Winsock	BSD sockets

The Linux equivalent of Windows DLLs are shared object libraries (.so files), which contain position-independent code (PIC). This has the following consequences:

- Variables referring to an absolute address in memory (using the **absolute** directive) are not allowed.
- Global memory references and calls to external functions are made relative to the EBX register, which must be preserved across calls.

You only need to worry about global memory references and calls to external functions if using assembler—Kylix or Delphi generates the correct code. (For information, see “Including inline assembler code” on page 10-20.)

Kylix library modules and packages are implemented using .so files.

## CLX and VCL unit comparison

All of the objects in the VCL or CLX are defined in unit files (.pas source files). For example, you can find the implementation of *TObject* in the System unit, and the Classes unit defines the base *TComponent* class. When you drop an object onto a form

or use an object within your application, the name of the unit is added to the **uses** clause which tells the compiler which units to link into the project.

This section provides tables that list the CLX units and the comparable VCL unit, list the units that are for CLX only, and list the units that are for VCL only.

The following table lists VCL units and the comparable CLX units:

**Table 10.4** VCL and equivalent CLX units

VCL units	CLX units
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts, and RTLConsts
Contnrs	Contnrs
Controls	QControls
DateUtils	DateUtils
DB	DB
DBActns	QDBActns
DBClient	DBClient
DBCommon	DBCommon
DBConnAdmin	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs
DSIntf	DSIntf
ExtCtrls	QExtCtrls
FMTBCD	FMTBCD
Forms	QForms
Graphics	QGraphics
Grids	QGrids
HelpIntfs	HelpIntfs
ImgList	QImgList
IniFiles	IniFiles
Mask	QMask
MaskUtils	MaskUtils

**Table 10.4** VCL and equivalent CLX units (continued)

VCL units	CLX units
Masks	Masks
Math	Math
Menus	QMenus
Midas	Midas
MidConst	MidConst
Printers	QPrinters
Provider	Provider
Qt	Qt
Search	QSearch
Sockets	Sockets
StdActns	QStdActns
StdCtrls	QStdCtrls
SqlConst	SqlConst
SqlExpr	SqlExpr
SqlTimSt	SqlTimSt
SyncObjs	SyncObjs
SysConst	SysConst
SysInit	SysInit
System	System
SysUtils	SysUtils
Types	Types and QTypes
TypeInfo	TypeInfo
Variants	Variants
VarUtils	VarUtils

The following units are in CLX but not VCL:

**Table 10.5** Units in CLX, not VCL

Unit	Description
DirSel	Directory selection
QStyle	GUI look and feel

The following Windows VCL units are not included in CLX mostly because they concern Windows-specific features that are not available on Linux such as ADO, COM, and the BDE. The reason for the unit's exclusion is listed.

**Table 10.6** VCL-only units

Unit	Reason for exclusion
ADOCnst	No ADO feature
ADODB	No ADO feature
AppEvnts	No TApplicationEvent object

**Table 10.6** VCL-only units (continued)

Unit	Reason for exclusion
AxCtrls	No COM feature
BdeConst	No BDE feature
ComStrs	No COM feature
ConvUtils	New feature for Delphi 6
CorbaCon	No Corba feature
CorbaStd	No Corba feature
CorbaVCL	No Corba feature
CtlPanel	No Windows Control Panel support
DataBkr	May appear later in upsell
DBCGrids	No BDE feature
DBExcept	No BDE feature
DBInpReq	No BDE feature
DBLookup	Obsolete
DbOleCtl	No COM feature
DBPWDlg	No BDE feature
DBTables	No BDE feature
DdeMan	No DDE feature
DRTable	No BDE feature
ExtActns	New feature to Delphi 6
ExtDlgs	No picture dialogs
FileCtrl	Obsolete
ListActns	New feature to Delphi 6
MConnect	No COM feature
Messages	Windows-specific area
MidasCon	Obsolete
MPlayer	Windows-specific media player
Mtsobj	No COM feature
MtsRdm	No COM feature
Mtx	No COM feature
mxConsts	No COM feature
ObjBrkr	May appear later in upsell
OleConstMay	No COM feature
OleCtnrs	No COM feature
OleCtrls	No COM feature
OLEDDB	No COM feature
OleServer	No COM feature
Outline	Obsolete
Registry	Windows-specific registry support
ScktCnst	Replaced by Sockets
ScktComp	Replaced by Sockets

**Table 10.6** VCL-only units (continued)

Unit	Reason for exclusion
SConnect	Unsupported connection protocols
StdConvs	New feature to Delphi 6
SvcMgr	NT Services support
Tabnotbk	Obsolete
Tabs	Obsolete
ToolWin	No docking feature
VarCmplx	New feature to Delphi 6
VarConv	New feature to Delphi 6
VCLCom	No COM feature
WebConst	Windows-specific constants
Windows	Windows-specific (API)

## Differences in CLX object constructors

---

When a CLX object is created, either implicitly in the Forms Designer by placing that object on the form or explicitly in code by using the *Create* method of the object, an instance of the underlying associated widget is created also. The instance of the widget is owned by this CLX object. When the CLX object is deleted by calling the *Free* method or automatically deleted by the CLX object's parent container, the underlying widget is also deleted. This is the same type of functionality that you see in the VCL in Windows applications.

When you explicitly create a CLX object in code, by calling into the Qt interface library such as *QWidget\_Create()*, you are creating an instance of a Qt widget that is not owned by a CLX object. This passes the instance of an existing Qt widget to the CLX object to use during its construction. This CLX object does not own the Qt widget that is passed to it. Therefore, when you call the *Free* method after creating the object in this manner, only the CLX object is destroyed and not the underlying Qt widget instance. This is different from the VCL.

Some CLX objects let you assume ownership of the underlying widget using the *OwnHandle* method. After calling *OwnHandle*, if you delete the CLX object, the underlying widget is destroyed as well.

## Sharing source files between Windows and Linux

---

If you want your application to run on both Windows and Linux, you can share the source files making them accessible to both operating systems. You can do this many ways such as placing the source files on a server that is accessible to both computers or by using Samba on the Linux machine to provide access to files through Microsoft network file sharing for both Linux and Windows. You can choose to keep the source on Linux and create a shared drive on Linux. Or you can keep the source on Windows and create a share on Windows for the Linux machine to access.

You can continue to develop and compile the file on Kylix using objects that are supported by both VCL and CLX. When you are finished, you can compile on both Linux and Windows.

Form files (.dfm files in Delphi) are called .xfm files in Kylix. If you create a new CLX application in Delphi or Kylix, an .xfm is created instead of a .dfm. If you plan to write cross-platform applications, the .xfm will work both on Delphi and Kylix.

## Environmental differences between Windows and Linux

---

Currently, cross-platform means an application that can run virtually unchanged on both the Windows and Linux operating systems. The following table lists some of the differences between Linux and the Windows operating environments.

**Table 10.7** Differences in the Linux and Windows operating environments

Difference	Description
File name case sensitivity	In Linux, a capital letter is <i>not</i> the same as a lowercase letter. The file Test.txt is <i>not</i> the same file as test.txt. You need to pay close attention to capitalization of file names on Linux.
Line ending characters	On Windows, lines of text are terminated by CR/LF (that is, ASCII 13 + ASCII 10), but on Linux it is LF. While the code editor in Kylix can handle the difference, you should be aware of this when importing code from Windows.
End of file character	In DOS and Windows, the character value #26 (Ctrl-Z) is treated as the end of the text file, even if there is data in the file after that character. Linux has no special end of file character; the text data ends at the end of the file.
Batch files/shell scripts	The Linux equivalent of .bat files are shell scripts. A script is a text file containing instructions, saved and made executable with the command, <code>chmod +x &lt;scriptfile&gt;</code> . To execute it, type its name. (The scripting language depends on the shell you are using on Linux. Bash is commonly used.)
Command confirmation	In DOS or Windows, if you try to delete a file or folder, it asks for confirmation ("Are you sure you want to do that?"). Generally, Linux won't ask; it will just do it. This makes it easy to accidentally destroy a file or the entire file system. There is no way to undo a deletion on Linux unless a file is backed up on another media.
Command feedback	If a command succeeds on Linux, it redisplay the command prompt without a status message.
Command switches	Linux uses a dash (-) to indicate command switches or a double dash (--) for multiple character options where DOS uses a slash (/) or dash (-).

**Table 10.7** Differences in the Linux and Windows operating environments (continued)

Difference	Description
Configuration files	<p>On Windows, configuration is done in the registry or in files such as <code>autoexec.bat</code>.</p> <p>On Linux, configuration files are created as hidden files starting with a dot (<code>.</code>). Many are placed in the <code>/etc</code> directory and your home directory.</p> <p>Linux also uses environment variables such as <code>LD_LIBRARY_PATH</code> (search path for libraries). Other important environment variables:</p> <p><code>HOME</code> Your home directory (<code>/home/sam</code>)</p> <p><code>TERM</code> Terminal type (<code>xterm</code>, <code>vt100</code>, <code>console</code>)</p> <p><code>SHELL</code> Path to your shell (<code>/bin/bash</code>)</p> <p><code>USER</code> Your login name (<code>sfuller</code>)</p> <p><code>PATH</code> List to search for programs</p> <p>They are specified in the shell or in <code>rc</code> files such as the <code>.bashrc</code>.</p>
DLLs	<p>On Linux, you use shared object files (<code>.so</code>). In Windows, these are dynamic link libraries (DLLs).</p>
Drive letters	<p>Linux doesn't have drive letters. An example Linux pathname is <code>/lib/security</code>. See <code>DriveDelim</code> in the runtime library.</p>
Exceptions	<p>Operating system exceptions are called signals on Linux.</p>
Executable files	<p>On Linux, executable files require no extension. On Windows, executable files have an <code>exe</code> extension.</p>
File name extensions	<p>Linux does not use file name extensions to identify file types or to associate files with applications.</p>
File permissions	<p>On Linux, files (and directories) are assigned read, write, and execute permissions for the file owner, group, and others. For example, <code>-rwxr-xr-x</code> means, from left to right:</p> <p>- is the file type (- = ordinary file, <code>d</code> = directory, <code>l</code> = link); <code>rwx</code> are the permissions for the file owner (read, write, execute); <code>r-x</code> are the permissions for the group of the file owner (read, execute); and <code>r-x</code> are the permissions for all other users (read, execute). The root user (superuser) can override these permissions.</p> <p>You need to make sure that your application runs under the correct user and has proper access to required files.</p>
Make utility	<p>Borland's make utility is not available on the Linux platform. Instead, you can use Linux's own GNU make utility.</p>
Multitasking	<p>Linux fully supports multitasking. You can run several programs (in Linux, called processes) at the same time. You can launch processes in the background (using <code>&amp;</code> after the command) and continue working straight away. Linux also lets you have several sessions.</p>
Pathnames	<p>Linux uses a forward slash (<code>/</code>) wherever DOS uses a backslash (<code>\</code>). A <code>PathDelim</code> constant can be used to specify the appropriate character for the platform. See <code>PathDelim</code> in the runtime library.</p>
Search path	<p>When executing programs, Windows always checks the current directory first, then looks at the <code>PATH</code> environment variable. Linux never looks in the current directory but searches only the directories listed in <code>PATH</code>. To run a program in the current directory, you usually have to type <code>./</code> before it.</p> <p>You can also modify your <code>PATH</code> to include <code>./</code> as the first path to search.</p>



**Table 10.7** Differences in the Linux and Windows operating environments (continued)

Difference	Description
Search path separator	Windows uses the semicolon as a search path separator. Linux uses a colon. See PathDelim in the runtime library.
Symbolic links	On Linux, a symbolic link is a special file that points to another file on disk. Place symbolic links in the global bin directory that points to your application's main files and you don't have to modify the system search path. A symbolic link is created with the ln (link) command.  Windows has shortcuts for the GUI desktop. To make a program available at the command line, Windows install programs typically modify the system search path.

## Directory structure on Linux

Directories are different in Linux. Any file or device can be mounted anywhere on the file system.

**Note** Linux pathnames use forward slashes as opposed to Windows use of backslashes. The initial slash stands for the root directory.

Following are some commonly used directories in Linux.

**Table 10.8** Common Linux directories

Directory	Contents
/	The root or top directory of the entire Linux file system
/root	The root file system; the Superuser's home directory
/bin	Commands, utilities
/sbin	System utilities
/dev	Devices shown as files
/lib	Libraries
/home/username	Files owned by the user where username is the user's login name.
/opt	Optional
/boot	Kernel that gets called when the system starts up
/etc	Configuration files
/usr	Applications, programs. Usually includes directories like /usr/spool, /usr/man, /usr/include, /usr/local
/mnt	Other media mounted on the system such as a CD or a floppy disk drive
/var	Logs, messages, spool files
/proc	Virtual file system and reporting system statistics
/tmp	Temporary files

**Note** Different distributions of Linux sometimes place files in different locations. A utility program may be placed in /bin in a Red Hat distribution but in /usr/local/bin in a Debian distribution.

Refer to [www.pathname.com](http://www.pathname.com) for additional details on the organization of the UNIX/Linux hierarchical file system and to read the *Filesystem Hierarchy Standard*.

## Writing portable code

---

If you are writing cross-platform applications that are meant to run on Windows and Linux, you can write code that compiles under different conditions. Using conditional compilation, you can maintain your Windows coding, yet also make allowances for Linux operating system differences.

To create applications that are easily portable between Windows and Linux, remember to

- reduce or isolate calls to platform-specific (Win32 or Linux) APIs; use CLX methods instead.
- eliminate Windows messaging (PostMessage, SendMessage) constructs within an application.
- use *TMemIniFile* instead of *TRegIniFile*.
- observe and preserve case-sensitivity in file and directory names.
- port any external assembler TASM code. The GNU assembler, “as,” does not support the TASM syntax. (See “Including inline assembler code” on page 10-20.)

Try to write the code to use platform-independent runtime library routines and use constants found in System, SysUtils, and other runtime library units. For example, use the PathDelim constant to insulate your code from ‘/’ versus ‘\’ platform differences.

Another example involves the use of multibyte characters on both platforms. Windows code traditionally expects only 2 bytes per multibyte character. In Linux, multibyte character encoding can have many more bytes per char (up to 6 bytes for UTF-8). Both platforms can be accommodated using the StrNextChar function in SysUtils. Existing Windows code such as the following

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    inc(p);
  inc(p);
end;
```

can be replaced with platform-independent code like this:

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

This example is platform portable and supports multibyte characters longer than 2 bytes, but still avoids the performance cost of a procedure call for non-multibyte locales.

If using runtime library functions is not a workable solution, try to isolate the platform-specific code in your routine into one chunk or into a subroutine. Try to limit the number of `$IFDEF` blocks to maintain source code readability and portability. The conditional symbol `WIN32` is not defined on Linux. The conditional symbol `LINUX` is defined, indicating the source code is being compiled for the Linux platform.

## Using conditional directives

Using `$IFDEF` compiler directives is a reasonable way to conditionalize your code for the Windows and Linux platforms. However, because `$IFDEFs` make source code harder to understand and maintain, you need to understand when it is reasonable to use `$IFDEFs`. When considering the use of `$IFDEFs`, the top questions should be “Why does this code require an `$IFDEF`?” and “Can this be written without an `$IFDEF`?”

Follow these guidelines for using `$IFDEFs` within cross-platform applications:

- Try not to use `$IFDEFs` unless absolutely necessary. `$IFDEFs` in a source file are only evaluated when source code is compiled. Unlike C/C++, Delphi does not require unit sources (header files) to compile a project. Full rebuilds of all source code is an uncommon event for most Delphi projects.
- Do not use `$IFDEFs` in package (.dpc) files. Limit their use to source files. Component writers need to create two design-time packages when doing cross-platform development, not one package using `$IFDEFs`.
- In general, use `$IFDEF MSWINDOWS` to test for any Windows platform including `WIN32`. Reserve the use of `$IFDEF WIN32` for distinguishing between specific Windows platforms, such as 32-bit versus 64-bit Windows. Don't limit your code to `WIN32` unless you know for sure that it will not work in `WIN64`.
- Avoid negative tests like `$IFNDEF` unless absolutely required. `$IFNDEF LINUX` is *not* equivalent to `$IFDEF MSWINDOWS`.
- Avoid `$IFNDEF/$ELSE` combinations. Use a positive test instead (`$IFDEF`) for better readability.
- Avoid `$ELSE` clauses on platform-sensitive `$IFDEFs`. Use separate `$IFDEF` blocks for `LINUX`- and `MSWINDOWS`-specific code instead of `$IFDEF LINUX/$ELSE` or `$IFDEF MSWINDOWS/$ELSE`.

For example, old code may contain

```
{IFDEF WIN32}
    (32-bit Windows code)
{ELSE}
    (16-bit Windows code)  ///! By mistake, Linux could fall into this code.
{ENDIF}
```

For any non-portable code in `$IFDEFs`, it is better for the source code to fail to compile than to have the platform fall into an `$ELSE` clause and fail mysteriously at runtime. Compile failures are easier to find than runtime failures.

- Use the **\$IF** syntax for complicated tests. Replace nested **\$IFDEFs** with a boolean expression in an **\$IF** directive. You should terminate the **\$IF** directive using **\$IFEND**, not **\$ENDIF**. This allows you to place **\$IF** expressions within **\$IFDEFs** to hide the new **\$IF** syntax from previous compilers.

All of the conditional directives are documented in the online Help. Also see, the topic “Conditional Compilation” in Help for more information.

## Terminating conditional directives

Use the **\$IFEND** directive to terminate **\$IF** and **\$ELSEIF** conditional directives. This allows **\$IF/\$IFEND** blocks to be hidden from older compilers inside of using **\$IFDEF/\$ENDIF**. Older compilers won't recognize the **\$IFEND** directive. **\$IF** can only be terminated with **\$IFEND**. You can only terminate old-style directives (**\$IFDEF**, **\$IFNDEF**, **\$IFOPT**) with **\$ENDIF**.

**Note** When nesting an **\$IF** inside of **\$IFDEF/\$ENDIF**, do not use **\$ELSE** with the **\$IF**. Older compilers will see the **\$ELSE** and think it is part of the **\$IFDEF**, producing a compile error down the line. You can use **{\$ELSEIF True}** as a substitute for **{\$ELSE}** in this situation, since the **\$ELSEIF** won't be taken if the **\$IF** is taken first, and the older compilers won't know **\$ELSEIF**. Hiding **\$IF** for backwards compatibility is primarily an issue for third party vendors and application developers who want their code to run on several different versions.

**\$ELSEIF** is a combination of **\$ELSE** and **\$IF**. The **\$ELSEIF** directive allows you to write multi-part conditional blocks where only one of the conditional blocks will be taken. For example:

```
{$IFDEF doit}
  do_doit
{$ELSEIF RTLVersion >= 14}
  goforit
{$ELSEIF somestring = 'yes'}
  beep
{$ELSE}
  last chance
{$IFEND}
```

Of these four cases, only one is taken. If none of the first three conditions is true, the **\$ELSE** clause is taken. **\$ELSEIF** must be terminated by **\$IFEND**. **\$ELSEIF** cannot appear after **\$ELSE**. Conditions are evaluated top to bottom like a normal **\$IF...\$ELSE** sequence. In the example, if `doit` is not defined, `RTLVersion` is 15, and `somestring = 'yes'`, only the “goforit” block will be taken not the “beep” block, even though the conditions for both are true.

If you forget to use an **\$ENDIF** to end one of your **\$IFDEFs**, the compiler reports the following error message at the end of the source file:

```
Missing ENDIF
```

If you have more than a few **\$IF/\$IFDEF** directives in your source file, it can be difficult to determine which one is causing the problem. Kylix or Delphi reports the

following error message on the source line of the last **\$IF/\$IFDEF** compiler directive with no matching **\$ENDIF/\$IFEND**:

```
Unterminated conditional directive
```

You can start looking for the problem at that location.

## Emitting messages

The **\$MESSAGE** compiler directive allows source code to emit hints, warnings, and errors just as the compiler does.

```
{$MESSAGE HINT|WARN|ERROR|FATAL 'text string' }
```

The message type is optional. If no message type is indicated, the default is HINT. The text string is required and must be enclosed in single quotes.

Examples:

```
{$MESSAGE 'Boo!'} emits a hint.
```

```
{$Message Hint 'Feed the cats'} emits a hint.
```

```
{$Message Warn 'Looks like rain.'} emits a warning.
```

```
{$Message Error 'Not implemented'} emits an error, continues compiling.
```

```
{$Message Fatal 'Bang. Yer dead.'} emits an error, terminates the compiler.
```

## Including inline assembler code

If you include inline assembler code in your Windows applications, you may not be able to use the same code on Linux because of position-independent code (PIC) requirements on Linux. Linux shared object libraries (DLL equivalents) require that all code be relocatable in memory without modification. This primarily affects inline assembler routines that use global variables or other absolute addresses, or that call external functions.

For units that contain only Object Pascal code, the compiler automatically generates PIC when required. PIC units have a `.dpu` extension (instead of `.dcu`). It's a good idea to compile every Pascal unit source file into both PIC and non-PIC formats; use the `-p` compiler switch to generate PIC. Precompiled units are available in both forms.

You may want to code assembler routines differently depending on whether you'll be compiling to an executable or a shared library; use `{IFDEF PIC}` to branch the two versions of your assembler code. Or you can consider rewriting the routine in Object Pascal to avoid the issue.

Following are the PIC rules for inline assembler code:

- PIC requires all memory references be made relative to the EBX register, which contains the current module's base address pointer (in Linux called the Global Offset Table or GOT). So, instead of

```
MOV EAX,GlobalVar
```

```
use
```

```
MOV EAX,[EBX].GlobalVar
```

- PIC requires that you preserve the EBX register across calls into your assembly code (same as on Win32), and also that you restore the EBX register *before* making calls to external functions (different from Win32).
- While PIC code will work in base executables, it may slow the performance and generate more code. You don't have any choice in shared objects, but in executables you probably still want to get the highest level of performance that you can.

## Messages and system events

---

Message loops and events work differently on Linux and in CLX, but this primarily affects component writing. Most component and property editors port easily. *TObject.Dispatch* and message method syntax on classes work fine on Linux; under Linux, however, operating system notifications are handled using system events rather than messages.

To create an event handler in a cross-platform application, you can override one of the methods described in Table 10.9 to write your own custom message instead of responding to Windows messages. In the override, call the inherited method so any default processes still take place.

**Table 10.9** TWidgetControl protected methods for responding to system events

Method	Description
<i>ChangeBounds</i>	Used when a <i>TWidgetControl</i> is resized. Roughly analogous to WM_SIZE or WM_MOVE in Windows. Qt sets the “geometry” of a widget based on the client area, VCL uses the entire control size, which includes what Qt refers to as the frame.
<i>ChangeScale</i>	Called automatically when resizing controls. Used to change the scale of a form and all its controls for a different screen resolution or font size. Because <i>ChangeScale</i> modifies the control’s Top, Left, Width, and Height properties, it changes the position of the control and its children as well as their size.
<i>ColorChanged</i>	Called when the color of the control has been changed.
<i>CursorChanged</i>	Called when the cursor changes shape. The mouse cursor assumes this shape when it’s over this widget.
<i>EnabledChanged</i>	Called when an application changes the enabled state of a window or control.
<i>FontChanged</i>	Called when the collection of font resources changed. It sets the font for the widget and informs all children about the change. Roughly analogous to the WM_FONTCHANGE message.
<i>PaletteChanged</i>	Called when the system palette has been changed. .
<i>ShowHintChanged</i>	Called when Help hints are displayed or hidden on a control.
<i>StyleChanged</i>	Called when the window or control’s GUI styles have changed.
<i>TabStopChanged</i>	Called when the tab order on the form has been changed.
<i>VisibleChanged</i>	Called when a control is hidden or shown.
<i>WidgetDestroyed</i>	Called when a widget underlying a control is destroyed.

Qt is a C++ toolkit, so all of its widgets are C++ objects. CLX is written in Object Pascal, and Object Pascal does not interact directly with C++ objects. In addition, Qt uses multiple inheritance in a few places. So Delphi includes an interface layer that converts all of the Qt classes to a series of straight C functions. These are then wrapped in a shared object in Linux and a DLL in Windows.

Every *TWidgetControl* has *CreateWidget*, *InitWidget*, and *HookEvents* virtual methods that almost always have to be overridden. *CreateWidget* creates the Qt widget, and assigns the Handle to the FHandle private field variable. *InitWidget* gets called after the widget is constructed, and the Handle is valid.

Some property assignments in Delphi CLX have moved from the Create constructor to *InitWidget*. This will allow delayed construction of the Qt object until it's really needed. For example, say you have a property named *Color*. In *SetColor*, you can check with *HandleAllocated* to see if you have a Qt handle. If the Handle is allocated, you can make the proper call to Qt to set the color. If not, you can store the value in a private field variable, and, in *InitWidget*, you set the property.

Linux supports two types of events: Widget and System. *HookEvents* is a virtual method that hooks the CLX controls event methods to a special hook object that communicates with the Qt object. The hook object is really just a set of method pointers. System events on Kylix go through *EventHandler*, which is basically a replacement for *WndProc*.

## Programming differences on Linux

---

The Linux `wchar_t` widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal widechars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal widechar data must be widened to 32 bits per character before it can be passed to an OS function as `wchar_t`.

In Linux, widestrings are reference counted like long strings (in Windows, they're not).

Multibyte handling differs in Linux. In Windows, multibyte characters (MBCS) are represented as 1- and 2-byte char codes. In Linux, they are represented in 1 to 6 bytes.

AnsiStrings can carry multibyte character sequences, dependent upon the user's locale settings. The Linux encoding for multibyte characters such as Japanese, Chinese, Hebrew, and Arabic may not be compatible with the Windows encoding for the same locale. Unicode is portable, whereas multibyte is not.

In Linux, you cannot use variables on absolute addresses. The syntax `var X: Integer absolute $1234;` is not supported in PIC and is not allowed in Delphi.

## Cross-platform database applications

---

On Windows, Delphi provides several choices for how to access database information. These include using ADO, the Borland Database Engine (BDE), and InterBase Express. These three choices are not available on Kylix, however. Instead, you can use *dbExpress*, a new, cross-platform data access technology, which is also available on Windows, starting with Delphi version 6.

Before you port a database application to *dbExpress* so that it will run on Linux, you should understand the differences between using *dbExpress* and the data access mechanism you were using. These differences occur at different levels.

- At the lowest level, there is a layer that communicates between your application and the database server. This could be ADO, the BDE, or the InterBase client software. This layer is replaced by *dbExpress*, which is a set of lightweight drivers for dynamic SQL processing.
- The low-level data access is wrapped in a set of components that you add to data modules or forms. These components include database connection components, which represent the connection to a database server, and datasets, which represent the data fetched from the server. Although there are some very important differences, due to the unidirectional nature of *dbExpress* cursors, the differences are less pronounced at this level, because datasets all share a common ancestor, as do database connection components.
- At the user-interface level, there are the fewest differences. CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. The major differences at the user interface level arise from changes needed to accommodate the use of cached updates.

For information on porting existing database applications to *dbExpress*, see “Porting database applications to Linux” on page 10-25. For information on designing new *dbExpress* applications, see Chapter 14, “Designing database applications.”

### dbExpress differences

---

On Linux, *dbExpress* manages the communication with database servers. *dbExpress* consists of a set of lightweight drivers that implement a set of common interfaces. Each driver is a shared object (.so file) that must be linked to your application. Because *dbExpress* is designed to be cross-platform, it will also be available on Windows as a set of dynamic-link libraries (.dlls).

As with any data-access layer, *dbExpress* requires the client-side software provided by the database vendor. In addition, it uses a database-specific driver, plus two configuration files, *dbxconnections* and *dbxdrivers*. This is markedly less than you need for, say, the BDE, which requires the main Borland Database Engine library (*Idapi32.dll*) plus a database-specific driver and a number of other supporting libraries.



Here are some other differences between *dbExpress* and the other data-access layers from which you need to port your application:

- *dbExpress* allows for a simpler and faster path to remote databases. As a result, you can expect a noticeable performance increase for simple, straight-through data access.
- *dbExpress* can process queries and stored procedures, but does not support the concept of opening tables.
- *dbExpress* returns only unidirectional cursors.
- *dbExpress* has no built-in update support other than the ability to execute an INSERT, DELETE, or UPDATE query.
- *dbExpress* does no metadata caching, and the design time metadata access interface is implemented using the core data-access interface.
- *dbExpress* executes only queries requested by the user, thereby optimizing database access by not introducing any extra queries.
- *dbExpress* manages a record buffer or a block of record buffers internally. This differs from the BDE, where clients are required to allocate the memory used to buffer records.
- *dbExpress* does not support local tables that are not SQL-based (such as Paradox, dBase, or FoxPro).
- *dbExpress* drivers exist for InterBase, Oracle, DB2, and MySQL. If you are using a different database server, you must either port your data to one of these databases, write a *dbExpress* driver for the database server you are using, or obtain a third-party *dbExpress* driver for your database server.

## Component-level differences

---

When you write a *dbExpress* application, it requires a different set of data-access components than those used in your existing database applications. The *dbExpress* components share the same base classes as other data-access components (*TDataSet* and *TCustomConnection*), which means that many of the properties, methods, and events are the same as the components used in your existing applications.

Table 10.10 lists some of the important database components used in InterBase Express, BDE, and ADO in the Windows environment and shows the comparable *dbExpress* components for use on Linux and in cross-platform applications.

**Table 10.10** Comparable data-access components

InterBase Express components	BDE components	ADO components	dbExpress components
<i>TIBDatabase</i>	<i>TDatabase</i>	<i>TADOConnection</i>	<i>TSQLConnection</i>
<i>TIBTable</i>	<i>TTable</i>	<i>TADOTable</i>	<i>TSQLTable</i>
<i>TIBQuery</i>	<i>TQuery</i>	<i>TADOQuery</i>	<i>TSQLQuery</i>
<i>TIBStoredProc</i>	<i>TStoredProc</i>	<i>TADOStoredProc</i>	<i>TSQLStoredProc</i>
<i>TIBDataSet</i>		<i>TADODataset</i>	<i>TSQLDataSet</i>

The *dbExpress* datasets (*TSQLTable*, *TSQLQuery*, *TSQLStoredProc*, and *TSQLDataSet*) are more limited than their counterparts, however, because they do not support editing and only allow forward navigation. For details on the differences between the *dbExpress* datasets and the other datasets that are available on Windows, see Chapter 22, “Using unidirectional datasets.”

Because of the lack of support for editing and navigation, most *dbExpress* applications do not work directly with the *dbExpress* datasets. Rather, they connect the *dbExpress* dataset to a client dataset, which buffers records in memory and provides support for editing and navigation. For more information about this architecture, see “Database architecture” on page 14-5.

**Note** For very simple applications, you can use *TSQLClientDataSet* instead of a *dbExpress* dataset connected to a client dataset. This has the benefit of simplicity, because there is a 1:1 correspondence between the dataset in the application you are porting and the dataset in the ported application, but is less flexible that explicitly connecting a *dbExpress* dataset to a client dataset. For most applications, it is recommended that you use a *dbExpress* dataset connected to a *TClientDataSet* component.

## User interface-level differences

---

CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. As a result, porting the user-interface portion of your database applications introduces few additional considerations beyond those involved in porting any Windows application to CLX.

The major differences at the user interface level arise from differences in the way *dbExpress* datasets or client datasets supply data.

If you are using only *dbExpress* datasets, then you must adjust your user interface to accommodate the fact that the datasets do not support editing and only support forward navigation. Thus, for example, you may need to remove controls that allow users to move to a previous record. Because *dbExpress* datasets do not buffer data, you can't display data in a data-aware grid: only one record can be displayed at a time.

If you have connected the *dbExpress* dataset to a client dataset, then the user interface elements associated with editing and navigation should still work. You need only reconnect them to the client dataset. The main consideration in this case is handling how updates are written to the database. By default, most datasets on Windows write updates to the database server automatically when they are posted (for example, when the user moves to a new record). Client datasets, on the other hand, always cache updates in memory. For information on how to accommodate this difference, see “Updating data in *dbExpress* applications” on page 10-27.

## Porting database applications to Linux

---

Porting your database application to *dbExpress* allows you to create a cross-platform application that runs both on Windows and Linux. The porting process involves making changes to your application because the technology is different. How difficult it is to port depends on the type of application it is, how complex it is, and

what it needs to accomplish. An application that heavily uses Windows-specific technologies such as ADO will be more difficult to port than one that uses Delphi database technology.

Follow these general steps to port your Windows/VCL database application to Kylix/CLX:

- 1 Consider where database data is stored. *dbExpress* provides drivers for Oracle, Interbase, DB2, and MySQL. The data needs to reside on one of these SQL servers.

Some versions of Delphi include the Data Pump utility which you can use to move local database data from platforms such as Paradox, dBase, and FoxPro onto one of the supported platforms. (See the `datapump.hlp` file in Program Files\Common Files\Borland\Shared\BDE for information on using the utility.)

- 2 If you have not isolated your user interface forms from data modules containing the datasets and connection components, you may want to consider doing so before you start the port. That way, you isolate the portions of your application that require a completely new set of components into data modules. Forms that represent the user interface can then be ported like any other application. For details, see “Porting your application” on page 10-4.

The remaining steps assume that your datasets and connection components are isolated in their own data modules.

- 3 Create a new data module to hold the CLX versions of your datasets and connection components.
- 4 For each dataset in the original application, add a *dbExpress* dataset, *TDataSetProvider* component, and *TClientDataSet* component. Use the correspondences in Table 10.10 to decide which *dbExpress* dataset to use. Give these components meaningful names.
  - Set the *ProviderName* property of the *TClientDataSet* component to the name of the *TDataSetProvider* component.
  - Set the *DataSet* property of the *TDataSetProvider* component to the *dbExpress* dataset.
  - Change the *DataSet* property of any data source components that referred to the original dataset so that it now refers to the client dataset.
- 5 Set properties on the new dataset to match the original dataset:
  - If the original dataset was a *TTable*, *TADOTable*, or *TIBTable* component, set the new *TSQLTable*'s *TableName* property to the original dataset's *TableName*. Also copy any properties used to set up master/detail relationships or specify indexes. Properties specifying ranges and filters should be set on the client dataset rather than the new *TSQLTable* component.
  - If the original dataset was a *TQuery*, *TADOQuery*, or *TIBQuery* component, set the new *TSQLQuery* component's *SQL* property to the original dataset's *SQL* property. Set the *Params* property of the new *TSQLQuery* to match the value of the original dataset's *Params* or *Parameters* property. If you have set the *DataSource* property to establish a master/detail relationship, copy this as well.

- If the original dataset was a *TStoredProc*, *TADOStoredProc*, or *TIBStoredProc* component, set the new *TSQLStoredProc* component's *StoredProcName* to the *StoredProcName* or *ProcedureName* property of the original dataset. Set the *Params* property of the new *TSQLStoredProc* to match the value of the original dataset's *Params* or *Parameters* property.
- 6 For any database connection components in the original application (*TDatabase*, *TIBDatabase*, or *TADOConnection*), add a *TSQLConnection* component to the new data module. You must also add a *TSQLConnection* component for every database server to which you connected without a connection component (for example, by using the *ConnectionString* property on an ADO dataset or by setting the *DatabaseName* property of a BDE dataset to a BDE alias).
  - 7 For each *dbExpress* dataset placed in step 4, set its *SQLConnection* property to the *TSQLConnection* component that corresponds to the appropriate database connection.
  - 8 On each *TSQLConnection* component, specify the information needed to establish a database connection. To do so, double-click the *TSQLConnection* component to display the Connection Editor and set parameter values to indicate the appropriate settings. If you had to transfer data to a new database server in step 1, then specify settings appropriate to the new server. If you are using the same server as before, you can look up some of this information on the original connection component:
    - If the original application used *TDatabase*, you must transfer the information that appears in the *Params* and *TransIsolation* properties.
    - If the original application used *TADOConnection*, you must transfer the information that appears in the *ConnectionString* and *IsolationLevel* properties.
    - If the original application used *TIBDatabase*, you must transfer the information that appears in the *DatabaseName* and *Params* properties.
    - If there was no original connection component, you must transfer the information associated with the BDE alias or that appeared in the dataset's *ConnectionString* property.

You may want to save this set of parameters under a new connection name. For more details on this process, see "Controlling connections" on page 17-2.

## Updating data in dbExpress applications

---

*dbExpress* applications use client datasets to support editing. When you post edits to a client dataset, the changes are written to the client dataset's in-memory snapshot of the data, but are not automatically written to the database server. If your original application used a client dataset for caching updates, then you do not need to change anything to support editing on Linux. However, if you relied on the default behavior of most datasets on Windows, which is to write edits to the database server when you post records, you must make changes to accommodate the use of a client dataset.

There are two ways to convert an application that did not previously cache updates:

- You can mimic the behavior of the dataset on Windows by writing code to apply each updated record to the database server as soon as it is posted. To do this, supply the client dataset with an *AfterPost* event handler that applies update to the database server:

```

procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
    with DataSet as TClientDataSet do
        ApplyUpdates(1);
end;
    
```

- You can adjust your user interface to deal with cached updates. This approach has certain advantages, such as reducing the amount of network traffic and minimizing transaction times. However, if you switch to using cached updates, you must decide when to apply those updates back to the database server, and probably make user interface changes to let users initiate the application of updates or inform provide them with feedback about whether their edits have been written to the database. Further, because update errors are not detected when the user posts a record, you will need to change the way you report such errors to the user, so that they can see which update caused a problem as well as what type of problem occurred.

If your original application used the support provided by the BDE or ADO for caching updates, you will need to make some adjustments in your code to switch to using a client dataset. The following table lists the properties, events, and methods that support cached updates on BDE and ADO datasets, and the corresponding properties, methods and events on *TClientDataSet*:

**Table 10.11** Properties, methods, and events for cached updates

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
<i>CachedUpdates</i>	<i>LockType</i>	Not needed, client datasets always cache updates.	Determines whether cached updates are in effect.
Not supported.	<i>CursorType</i>	Not supported.	Specifies how isolated the dataset is from changes on the server.
<i>UpdatesPending</i>	Not supported.	<i>ChangeCount</i>	Indicates whether the local cache contains updated records that need to be applied to the database.
<i>UpdateRecordTypes</i>	<i>FilterGroup</i>	<i>StatusFilter</i>	Indicates the kind of updated records to make visible when applying cached updates.
<i>UpdateStatus</i>	<i>RecordStatus</i>	<i>UpdateStatus</i>	Indicates if a record is unchanged, modified, inserted, or deleted.
<i>OnUpdateError</i>	Not supported.	<i>OnReconcileError</i>	An event for handling update errors on a record-by-record basis.
<i>ApplyUpdates</i> (on dataset or database)	UpdateBatch	<i>ApplyUpdates</i>	Applies records in the local cache to the database.

**Table 10.11** Properties, methods, and events for cached updates (continued)

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
<i>CancelUpdates</i>	CancelUpdates or CancelBatch	<i>CancelUpdates</i>	Removes pending updates from the local cache without applying them.
<i>CommitUpdates</i>	Handled automatically	<i>Reconcile</i>	Clears the update cache following successful application of updates.
<i>FetchAll</i>	Not supported	<i>GetNextPacket</i> (and <i>PacketRecords</i> )	Copies database records to the local cache for editing and updating.
<i>RevertRecord</i>	CancelBatch	<i>RevertRecord</i>	Undoes updates to the current record if updates are not yet applied.

## Cross-platform Internet applications

An Internet application is a client/server application that uses standard Internet protocols for connecting the client to the server. Because your applications use standard Internet protocols for client/server communications, you can make your applications cross-platform. For example, a server-side program for an Internet application communicates with the client through the Web server software for the machine. The server application is typically written for Linux or Windows, but can also be cross-platform. The clients can be on either platform.

You can use Delphi or Kylix to create Web server applications as CGI or Apache applications for deployment on Linux. On Windows, you can create other types of Web servers such as Microsoft Server DLLs (ISAPI), Netscape Server DLLs (NSAPI), and Windows CGI applications. Only straight CGI applications and some applications that use Web Broker will run on both Windows and Linux.

### Porting Internet applications to Linux

If you have existing Internet applications that you want to make cross-platform, you should consider whether you want to port your Web server application or if you want to create a new application on Linux. See Chapter 27, “Creating Internet applications” for information on writing Web servers. If your application uses Web Broker and writes to the Web Broker interface and does not use native API calls, it will not be as difficult to port it to Linux.

If your application writes to ISAPI, NSAPI, Windows CGI, or other Web APIs, it will be more difficult to port. You will need to search through your source files and translate these API calls into Apache (see `httpd.pas` in the Internet directory for function prototypes for Apache APIs) or CGI calls. You also need to make all other suggested changes described in “Porting VCL applications to CLX” on page 10-2.



## Working with packages and components

A *package* is a special dynamic-link library used by Delphi applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland package library) extension.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used Delphi components reside in a package called vcl. Each time you create an application, it automatically uses vcl. When you compile an application created this way, the application's executable image contains only the code and data unique to it; the common code is in the runtime package called vcl60.bpl. A computer with several package-enabled applications installed on it needs only a single copy of vcl60.bpl, which is shared by all the applications and the Delphi IDE itself.

Delphi ships with several precompiled runtime packages that encapsulate VCL and CLX components. Delphi also uses design-time packages to manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write Delphi components, you can compile your components into design-time packages before installing them.



## Why use packages?

---

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Delphi itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller—saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

### Packages and standard DLLs

---

Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL when you want to build a library that can be called from any application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages:

**Table 11.1** Compiled package files

File extension	Contents
dpk	The source file listing the units contained in the package.
dcp	A binary image containing a package header and the concatenation of all dcu files in the package, including all symbol information required by the compiler. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file. You must have a .dcp file to build an application with packages.
dcu	A binary image for a unit file contained in a package. One dcu is created, when necessary, for each unit file.
bpl	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the bpl is the base name of the dpk source file.

You can include VCL or CLX or both types of components in a package. Packages meant to be cross-platform should include CLX components only.

**Note** Packages share their global data with other modules in an application.

For more information about DLLs and packages, see the *Object Pascal Language Guide*.

## Runtime packages

---

Runtime packages are deployed with Delphi applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's executable file and all the packages (.bpl files) that the application uses.

The .bpl files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required .bpls.

## Using packages in an application

---

To use packages in an application,

- 1 Load or create a project in the IDE.
- 2 Choose Project | Options.
- 3 Choose the Packages tab.
- 4 Select the “Build with Runtime Packages” check box, and enter one or more package names in the edit box underneath. (Runtime packages associated with installed design-time packages are already listed in the edit box.)
- 5 To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog. To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog.

If you edit the Search Path edit box in the Add Runtime Package dialog, you will be changing Delphi’s global Library Path.

You do not need to include file extensions with package names (or the number representing the Delphi release); that is, vcl60.bpl is written as vcl. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with semicolons. For example:

```
rtl;vcl;vclldb;vclado;vclx;Vclbde;
```

Packages listed in the Runtime Packages edit box are automatically linked to your application when you compile. Duplicate package names are ignored, and if the edit box is empty the application is compiled without packages.

Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the “Defaults” check box at the bottom of the dialog.

**Note** When you create an application with packages, you still need to include the names of the original Delphi units in the **uses** clause of your source files. For example, the source file for your main form might begin like this:

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
```

The units referenced in this example are contained in the vcl and rtl packages. Nonetheless, you must keep these references in the **uses** clause, even if you use vcl and rtl in your application, or you will get compiler errors. In generated source files, Delphi adds these units to the **uses** clause automatically.

## Dynamically loading packages

---

To load a package at runtime, call the *LoadPackage* function. *LoadPackage* loads the package, checks for duplicate units, and calls the initialization blocks of all units contained in the package. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

## Deciding which runtime packages to use

---

Delphi ships with several precompiled runtime packages, including rtl and vcl, which supply basic language and component support.

The vcl package contains the most commonly used components; the rtl package includes all the non-component system functions and Windows interface elements. It does not include database or other special components, which are available in separate packages.

To create a client/server database application that uses packages, you need at least three runtime packages: vcl and vcldb. If you want to use Outline components in your application, you also need vclx. To use these packages, choose Project | Options, select the Packages tab, and enter the following list in the Runtime Packages edit box.

```
rtl;vcl;Vcldb;vclx;
```

Actually, you don't have to include vcl and rtl, because they are referenced in the Requires clause of vcldb. (See "Requires clause" on page 11-8.) Your application will compile just the same whether or not vcl and rtl are included in the Runtime Packages edit box.

## Custom packages

---

A custom package is either a bpl you code and compile yourself or a precompiled package from a third-party vendor. To use a custom runtime package with an application, choose Project | Options and add the name of the package to the Runtime Packages edit box on the Packages page. For example, suppose you have a statistical package called stats.bpl. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:

```
rtl;vcl;vcldb;stats
```

If you create your own packages, you can add them to the list as needed.

## Design-time packages

---

Design-time packages are used to install components on the IDE's Component palette and to create special property editors for custom components.

Delphi ships with many design-time component packages preinstalled in the IDE. Which ones are installed depends on which version of Delphi you are using and whether or not you have customized it. You can view a list of what packages are installed on your system by choosing Component | Install Packages.

The design-time packages work by calling runtime packages, which they reference in their Requires clauses. (See "Requires clause" on page 11-8.) For example, `dclstd` references `vcl`. `Dclstd` itself contains additional functionality that makes most of the standard components available on the Component palette.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The `dclusr` design-time package is provided as a default container for new components.

### Installing component packages

---

All components are installed in the IDE as packages. If you've written your own components, create and compile a package that contains them. (See "Creating and editing packages" on page 11-6.) Your component source code must follow the model described in Part V, "Creating custom components".

To install or uninstall your own components, or components from a third-party vendor, follow these steps:

- 1 If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with `.bpl`, `.dcp`, and `.dcu` files, be sure to copy all of them. (For information about these files, see "Package files created by a successful compilation.")

The directory where you store the `.dcp` file—and the `.dcu` files, if they are included with the distribution—must be in the Delphi Library Path.

If the package is shipped as a `.dpc` (package collection) file, only the one file needs to be copied; the `.dpc` file contains the other files. (For more information about package collection files, see "Package collection files" on page 11-13.)

- 2 Choose Component | Install Packages from the IDE menu, or choose Project | Options and click the Packages tab.
- 3 A list of available packages appears under "Design packages."
  - To install a package in the IDE, select the check box next to it.
  - To uninstall a package, deselect its check box.
  - To see a list of components included in an installed package, select the package and click Components.

- To add a package to the list, click Add and browse in the Open Package dialog box for the directory where the .bpl or .dpc file resides (see step 1). Select the .bpl or .dpc file and click Open. If you select a .dpc file, a new dialog box appears to handle the extraction of the .bpl and other files from the package collection.
- To remove a package from the list, select the package and click Remove.

#### 4 Click OK.

The components in the package are installed on the Component palette pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the default settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the Packages tab of the Project Options dialog box.

To remove components from the Component palette without uninstalling a package, select Component | Configure Palette, or select Tools | Environment Options and click the Palette tab. The Palette options tab lists each installed component along with the name of the Component palette page where it appears. Selecting any component and clicking Hide removes the component from the palette.

## Creating and editing packages

---

Creating a package involves specifying

- A *name* for the package.
- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which contain the functionality of the compiled .bpl. The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

Package source files, which end with the .dpc extension, are generated by the Package editor.

### Creating a package

---

To create a package, follow the procedure below. Refer to “Understanding the structure of a package” on page 11-8 for more information about the steps outlined here.

**Note** Do not use IFDEFs in a package file (.dpc) such as when doing cross-platform development. You can use them in the source code, however.

- 1 Choose File | New, select the Package icon, and click OK.

- 2 The generated package is displayed in the Package editor.
- 3 The Package editor shows a *Requires* node and a *Contains* node for the new package.
- 4 To add a unit to the **contains** clause, click the Add to package speed button. In the Add unit page, type a .pas file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the Package editor. You can add additional units by repeating this step.
- 5 To add a package to the **requires** clause, click the Add to package speed button. In the Requires page, type a .dcp file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the Package editor. You can add additional packages by repeating this step.
- 6 Click the Options speed button, and decide what kind of package you want to build.
  - To create a design-time only package (a package that cannot be used at runtime), select the Designtime only radio button. (Or add the `{$DESIGNONLY}` compiler directive to the dpk file.)
  - To create a runtime-only package (a package that cannot be installed), select the Runtime only radio button. (Or add the `{$RUNONLY}` compiler directive to the dpk file.)
  - To create a package that is available at both design time and runtime, select the Designtime and runtime radio button.
- 7 In the Package editor, click the Compile package speed button to compile your package.

## Editing an existing package

---

You can open an existing package for editing in several ways:

- Choose File | Open (or File | Reopen) and select a dpk file.
- Choose Component | Install Packages, select a package from the Design Packages list, and click the Edit button.
- When the Package editor is open, select one of the packages in the Requires node, right-click, and choose Open.

To edit a package's description or set usage options, click the Options speed button in the Package editor and select the Description tab.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the `defproj.dof` file.

## Editing package source files manually

---

Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi package) extension to avoid confusion with other files containing Object Pascal source code.

To open a package source file in the Code editor,

- 1 Open the package in the Package editor.
- 2 Right-click in the Package editor and select View Source.
  - The **package** heading specifies the name for the package.
  - The **requires** clause lists other, external packages used by the current package. If a package does not contain any units that use units in another package, then it doesn't need a **requires** clause.
  - The **contains** clause identifies the unit files to be compiled and bound into the package. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains clause (the compiler will give a warning).

For example, the following code declares the vcldb package (in the source file vcldb60.bpl):

```
package vcldb;
  requires vcldb;
  contains rtl, vcl, Db, DBActns, DBOLECtl, Dbcgrids, dbCommon, dbConsts, Dbctrls,
  Dbgrids, Dblogdlg, SQLTimSt, FmtBcd;
end.
```

## Understanding the structure of a package

---

Packages include the following parts:

- Package name
- Requires clause
- Contains clause

### Naming packages

Package names must be unique within a project. If you name a package STATS, the Package editor generates a source file for it called STATS.dpk; the compiler generates an executable and a binary image called STATS.bpl and STATS.dcp, respectively. Use STATS to refer to the package in the **requires** clause of another package, or when using the package in an application.

### Requires clause

The **requires** clause specifies other, external packages that are used by the current package. An external package included in the **requires** clause is automatically linked

at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's **requires** clause or you should add them. If the other packages are omitted from the **requires** clause, the compiler will import them into your package 'implicitly contained units'.

**Note** Most packages that you create will require rtl. If using VCL components, you'll also need to include the vcl package. If using CLX components for cross-platform programming, you need to include VisualCLX.

### Avoiding circular package references

Packages cannot contain circular references in their **requires** clause. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

### Handling duplicate package references

Duplicate references in a package's **requires** clause—or in the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

### Contains clause

The **contains** clause identifies the unit files to be bound into the package. If you are writing your own package, put your source code in pas files and include them in the **contains** clause.

### Avoiding redundant source code uses

A package cannot appear in the **contains** clause of another package.

All units included directly in a package's **contains** clause, or included indirectly in any of those units, are bound into the package at compile time.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, *including the Delphi IDE*. This means that if you create a package that contains one of the units in vcl you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's **requires** clause.



## Compiling packages

---

You can compile a package from the IDE or from the command line. To recompile a package by itself from the IDE,

- 1 Choose File | Open.
- 2 Select Delphi package (\*.dpk) from the Files of Type drop-down list.
- 3 Select a .dpk file in the dialog.
- 4 When the Package editor opens, click the Compile speed button.

You can insert compiler directives into your package source code. For more information, see “Package-specific compiler directives”, below.

If you compile from the command line, several package-specific switches are available. For more information, see “Using the command-line compiler and linker” on page 11-12.

### Package-specific compiler directives

The following table lists package-specific compiler directives that you can insert into your source code.

**Table 11.2** Package-specific compiler directives

Directive	Purpose
{\$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{\$G-} or {IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{\$WEAKPACKAGEUNIT ON}	Packages unit “weakly.” See “Weak packaging” on page 11-11 below.
{\$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{\$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{\$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

**Note** Including **{\$DENYPACKAGEUNIT ON}** in your source code prevents the unit file from being packaged. Including **{\$G-}** or **{IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{\$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the IDE. Other compiler directives may be included, if appropriate, in package source code. See Compiler directives in the online help for information on compiler directives not discussed here.

Refer to “Creating packages and DLLs” on page 5-9 for additional directives that can be used in all libraries.

## Weak packaging

The **\$WEAKPACKAGEUNIT** directive affects the way a .dcp file is stored in a package's .dcp and .bpl files. (For information about files generated by the compiler, see "Package files created by a successful compilation" on page 11-12.) If **{WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged."

For example, suppose you've created a package called PACK that contains only one unit, UNIT1. Suppose UNIT1 does not use any further units, but it makes calls to RARE.dll. If you put **{WEAKPACKAGEUNIT ON}** in UNIT1.pas when you compile your package, UNIT1 will not be included in PACK.bpl; you will not have to distribute copies of RARE.dll with PACK. However, UNIT1 will still be included in PACK.dcp. If UNIT1 is referenced by another package or application that uses PACK, it will be copied from PACK.dcp and compiled directly into the project.

Now suppose you add a second unit, UNIT2, to PACK. Suppose that UNIT2 uses UNIT1. This time, even if you compile PACK with **{WEAKPACKAGEUNIT ON}** in UNIT1.pas, the compiler will include UNIT1 in PACK.bpl. But other packages or applications that reference UNIT1 will use the (non-packaged) copy taken from PACK.dcp.

**Note** Unit files containing the **{WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **\$WEAKPACKAGEUNIT** directive is an advanced feature intended for developers who distribute their packages to other Delphi programmers. It can help you to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, Delphi's PenWin unit references PenWin.dll. Most projects don't use PenWin, and most computers don't have PenWin.dll installed on them. For this reason, the PenWin unit is weakly packaged in vcl. When you compile a project that uses PenWin and the vcl package, PenWin is copied from VCL60.dcp and bound directly into your project; the resulting executable is statically linked to PenWin.dll.

If PenWin were not weakly packaged, two problems would arise. First, vcl itself would be statically linked to PenWin.dll, and so you could not load it on any computer which didn't have PenWin.dll installed. Second, if you tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both vcl and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of vcl.

## Using the command-line compiler and linker

When you compile from the command line, you can use the package-specific switches listed in the following table.

**Table 11.3** Package-specific command-line compiler switches

Switch	Purpose
<code>-\$G-</code>	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
<code>-LEpath</code>	Specifies the directory where the package bpl file will be placed.
<code>-LNpath</code>	Specifies the directory where the package dcp file will be placed.
<code>-LUpackage</code>	Use packages.
<code>-Z</code>	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

**Note** Using the `-$G-` switch may prevent a package from being used in the same application with other packages. Other command-line options may be used, if appropriate, when compiling packages. See “The Command-line compiler” in the online help for information on command-line options not discussed here.

## Package files created by a successful compilation

To create a package, you compile a source file that has a `.dpk` extension. The base name of the `.dpk` file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called `traypak.dpk`, the compiler creates a package called `traypak.bpl`.

The following table lists the files produced by the successful compilation of a package.

**Table 11.4** Compiled package files

File extension	Contents
<code>dcp</code>	A binary image containing a package header and the concatenation of all <code>dcu</code> files in the package. A single <code>dcp</code> file is created for each package. The base name for the <code>dcp</code> is the base name of the <code>dpk</code> source file.
<code>dcu</code>	A binary image for a unit file contained in a package. One <code>dcu</code> is created, when necessary, for each unit file.
<code>bpl</code>	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the <code>bpl</code> is the base name of the <code>dpk</code> source file.

When compiled, the `bpi`, `bpl`, and `lib` files are generated by default in the directories specified in Library page of the Tools | Environment Options dialog. You can override the default settings by clicking the Options speed button in the Package editor to display the Project Options dialog; make any changes on the Directories/Conditionals page.

# Deploying packages

---

You deploy packages much like you deploy other applications. For general deployment information, refer to Chapter 13, “Deploying applications”.

## Deploying applications that use packages

---

When distributing an application that uses runtime packages, make sure that your users have the application’s .exe file as well as all the library (.bpl or .dll) files that the application calls. If the library files are in a different directory from the .exe file, they must be accessible through the user’s Path. You may want to follow the convention of putting library files in the Windows\System directory. If you use InstallShield Express, your installation script can check the user’s system for any packages it requires before blindly reinstalling them.

## Distributing packages to other developers

---

If you distribute runtime or design-time packages to other Delphi developers, be sure to supply both .dcp and .bpl files. You will probably want to include .dcu files as well.

## Package collection files

---

Package collections (.dpc files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including bpls and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .pce container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.

To create a package collection,

- 1 Choose Tools | Package Collection Editor to open the Package Collection editor.
- 2 Click the Add a Package speed button, then select a bpl in the Select Package dialog and click Open. To add more bpls to the collection, click the Add a Package speed button again. A tree diagram on the left side of the Package editor displays the bpls as you add them. To remove a package, select it and click the Remove Package speed button.
- 3 Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields will appear:
  - In the Author/Vendor Name edit box, you can enter optional information about your package collection that will appear in the Installation dialog when users install packages.

- Under **Directory List**, list the default directories where you want the files in your package collection to be installed. Use the **Add**, **Edit**, and **Delete** buttons to edit this list. For example, suppose you want all source code files to be copied to the same directory. In this case, you might enter `Source` as a **Directory Name** with `C:\MyPackage\Source` as the **Suggested Path**. The **Installation** dialog box will display `C:\MyPackage\Source` as the suggested path for the directory.
- 4 In addition to `bpls`, your package collection can contain `.dcp`, `.dcu`, and `.pas` (unit) files, documentation, and any other files you want to include with the distribution. Ancillary files are placed in file groups associated with specific packages (`bpls`); the files in a group are installed only when their associated `bpl` is installed. To place ancillary files in your package collection, select a `bpl` in the tree diagram and click the **Add File Group** speed button; type a name for the file group. Add more file groups, if desired, in the same way. When you select a file group, new fields will appear on the right in the **Package Collection** editor,
    - In the **Install Directory** list box, select the directory where you want files in this group to be installed. The drop-down list includes the directories you entered under **Directory List** in step 3, above.
    - Check the **Optional Group** check box if you want installation of the files in this group to be optional.
    - Under **Include Files**, list the files you want to include in this group. Use the **Add**, **Delete**, and **Auto** buttons to edit the list. The **Auto** button allows you to select all files with specified extensions that are listed in the **contains** clause of the package; the **Package Collection** editor uses Delphi's global **Library Path** to search for these files.
  - 5 You can select installation directories for the packages listed in the **requires** clause of any package in your collection. When you select a `bpl` in the tree diagram, four new fields appear on the right side of the **Package Collection** editor:
    - In the **Required Executables** list box, select the directory where you want the `.bpl` files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under **Directory List** in step 3, above.) The **Package Collection** Editor searches for these files using Delphi's global **Library Path** and lists them under **Required Executable Files**.
    - In the **Required Libraries** list box, select the directory where you want the `.dcp` files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under **Directory List** in step 3, above.) The **Package Collection** Editor searches for these files using Delphi's global **Library Path** and lists them under **Required Library Files**.
  - 6 To save your package collection source file, choose **File | Save**. Package collection source files should be saved with the `.pce` extension.
  - 7 To build your package collection, click the **Compile** speed button. The **Package Collection** editor generates a `.dpc` file with the same name as your source (`.pce`) file. If you have not yet saved the source file, the editor queries you for a file name before compiling.

To edit or recompile an existing `.pce` file, select **File | Open** in the **Package Collection** editor and locate the file you want to work with.

# Creating international applications

This chapter discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

## Internationalization and localization

---

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization
- Localization

If your version of Delphi includes the Translation Tools, you can use the them to manage localization. For more information, see the online Help for the Translation Tools (ETM.hlp).

### Internationalization

---

Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Windows supports a large set of locales, each of which is described by a language and country pair.

## Localization

---

Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified to be aware of the different tax laws in different countries.

## Internationalizing applications

---

You need to complete the following steps to create internationalized applications:

- You must enable your code to handle strings from international character sets.
- You need to design your user interface so that it can accommodate the changes that result from localization.
- You need to isolate all resources that need to be localized.

## Enabling application code

---

You must make sure that the code in your application can handle the strings it will encounter in the various target locales.

### Character sets

The United States edition of Windows uses the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-JIS character set (code page 932), which represents Japanese characters as multibyte character codes.

There are generally three types of characters sets:

- Single-byte
- Multibyte
- Fixed-width multibyte

Windows and Linux both support single-byte and multibyte character sets as well as Unicode. With a single-byte character set, each byte in a string represents one character. The ANSI character set used by many Western operating systems is a single-byte character set.

In a multibyte character set, some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages, while the UTF-8 character set used by Linux is a multibyte encoding of Unicode.

## OEM and ANSI character sets

It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).

## Multibyte character sets

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) *char* type. These languages have too many characters to be represented using the 1-byte *char*. Instead, a multibyte string can contain one or more bytes per character. AnsiStrings can contain a mix of single-byte and multibyte characters.

The lead byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent bytes can sometimes be the same as the character code for a separate 1-byte character, or it can fall in the range reserved for the first byte of multibyte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or is part of a multibyte character is to read the string, starting at the beginning, parsing it into 2 or more byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into multibyte characters. Delphi provides you with many runtime library functions that allow you to do this, many of which are listed here:

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrIComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrLIComp	CharToByteLen	

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.



## Wide characters

Another approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. Unicode characters and strings are also called wide characters and wide character strings. In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words.

The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi/Kylix supports UCS-2 on both platforms. Because wide characters are two bytes instead of one, the character set can represent many more different characters.

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

The biggest disadvantage of working with wide characters is that Windows 9x only supports a few wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require additional code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

## Including bi-directional functionality in applications

Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional.

*TApplication* has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties. The following table lists VCL objects that have these properties:

**Table 12.1** VCL objects that support BiDi

Component palette page	VCL object
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel

**Table 12.1** VCL objects that support BiDi (continued)

Component palette page	VCL object
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar
Additional	TActionMainMenuBar
	TActionToolBar
	TBitBtn
	TCheckBox
	TColorBox
	TDrawGrid
	TLabeledEdit
	TMaskEdit
	TScrollBar
	TSpeedButton
	TStaticLabel
	TStaticText
	TStringGrid
	TValueListEditor
Win32	TComboBoxEx
	TDateTimePicker
	THeaderControl
	THotKey
	TListView
	TMonthCalendar
	TPageControl
	TRichEdit
	TStatusBar
	TTreeView
Data Controls	TDBCheckBox
	TDBComboBox
	TDBEdit
	TDBGrid
	TDBListBox
	TDBLookupComboBox
	TDBLookupListBox
	TDBMemo

**Table 12.1** VCL objects that support BiDi (continued)

Component palette page	VCL object
	TDBRadioGroup
	TDBRichEdit
	TDBText
QReport	TQRDBText
	TQRExpr
	TQRLabel
	TQRMemo
	TQRPreview
	TQRSysData
Other classes	TApplication (has no <i>ParentBiDiMode</i> )
	TBoundLabel
	TControl (has no <i>ParentBiDiMode</i> )
	TCustomHeaderControl (has no <i>ParentBiDiMode</i> )
	TForm
	TFrame
	THeaderSection
	THintWindow (has no <i>ParentBiDiMode</i> )
	TMenu
	TStatusPanel
	TTabControl
	TValueListEditor

**Notes** *THintWindow* picks up the *BiDiMode* of the control that activated the hint.

### Bi-directional properties

The objects listed in Table 12.1, “VCL objects that support BiDi,” on page 12-4 have the properties *BiDiMode* and *ParentBiDiMode*. These properties, along with *TApplication*’s *BiDiKeyboard* and *NonBiDiKeyboard*, support bi-directional localization.

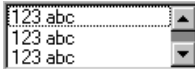
**Note** Bi-directional properties are not available in CLX for cross-platform programming.

### BiDiMode property

The property *BiDiMode* is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly*.

### bdLeftToRight

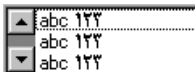
*bdLeftToRight* draws text using left to right reading order, and the alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

**Figure 12.1** TListBox set to bdLeftToRight**bdRightToLeft**

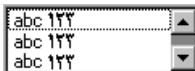
*bdRightToLeft* draws text using right to left reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left-to-right.

**Figure 12.2** TListBox set to bdRightToLeft**bdRightToLeftNoAlign**

*bdRightToLeftNoAlign* draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

**Figure 12.3** TListBox set to bdRightToLeftNoAlign**bdRightToLeftReadingOnly**

*bdRightToLeftReadingOnly* draws text using right to left reading order, and the alignment and scroll bars are not changed.

**Figure 12.4** TListBox set to bdRightToLeftReadingOnly**ParentBiDiMode property**

*ParentBiDiMode* is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from *Application*. If all the *ParentBiDiMode* properties are *True*, when you change *Application's BiDiMode* property, all forms and controls in the project are updated with the new setting.

**FlipChildren method**

The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupBox*. *FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped.

Delphi flips the controls by changing the *Left* property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control,

after it is flipped the edit control's right side is five pixels from the right edge of the parent control. If the edit control is left aligned, calling *FlipChildren* will make the control right aligned.

To flip a control at design-time select Edit | Flip Children and select either All or Selected, depending on whether you want to flip all the controls, or just the children of the selected control. You can also flip a control by selecting the control on the form, right-clicking, and selecting Flip Children from the context menu.

**Note** Selecting an edit control and issuing a Flip Children | Selected command does nothing. This is because edit controls are not containers.

### Additional methods

There are several other methods useful for developing applications for bi-directional users.

Method	Description
OkToChangeFieldAlignment	Used with database controls. Checks to see if the alignment of a control can be changed.
DBUseRightToLeftAlignment	A wrapper for database controls for checking alignment.
ChangeBiDiModeAlignment	Changes the alignment parameter passed to it. No check is done for <i>BiDiMode</i> setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone.
IsRightToLeft	Returns <i>True</i> if any of the right to left options are selected. If it returns <i>False</i> the control is in left to right mode.
UseRightToLeftReading	Returns <i>True</i> if the control is using right to left reading.
UseRightToLeftAlignment	Returns <i>True</i> if the control is using right to left alignment. It can be overridden for customization.
UseRightToLeftScrollBar	Returns <i>True</i> if the control is using a left scroll bar.
DrawTextBiDiModeFlags	Returns the correct draw text flags for the <i>BiDiMode</i> of the control.
DrawTextBiDiModeFlagsReadingOnly	Returns the correct draw text flags for the <i>BiDiMode</i> of the control, limiting the flag to its reading order.
AddBiDiModeExStyle	Adds the appropriate <i>ExStyle</i> flags to the control that is being created.

### Locale-specific features

You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings.

VCL components offer support in programming the IME. Most windowed controls that work directly with text input have an *ImeName* property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an *ImeMode* property that specifies how the IME should convert

keyboard input. *TWinControl* introduces several protected methods that you can use to control the IME from classes you define. In addition, the global *Screen* variable provides information about the IMEs available on the user's system.

The global *Screen* variable (available in VCL and CLX) also provides information about the keyboard mapping installed on the user's system. You can use this to obtain locale-specific information about the environment in which your application is running.

## Designing the user interface

---

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

### Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. Table 12.2 provides a rough estimate of how much expansion you should plan for given the length of your English strings:

**Table 12.2** Estimating string lengths

Length of English string (in characters)	Expected increase
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

### Graphic images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if your

application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

## Formats and sort order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, if you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, 2-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

## Keyboard mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

## Isolating resources

---

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Delphi automatically creates a .dfm (.xfm in CLX applications) file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the Object Pascal Language Guide. It is best to include all resource strings in a single, separate unit.

## Creating resource DLLs

---

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL.

Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the

project, and generate a project for a resource only DLL that contains the relevant forms and the created RES file. The RES file is compiled from the new RC file.

You should create a resource DLL for each translation you want to support. Each resource DLL should have a file name extension specific to the target locale. The first two characters indicate the target language, and the third character indicates the country of the locale. If you use the Resource DLL wizard, this is handled for you. Otherwise, use the following code to obtain the locale code for the target translation:

```

unit locales;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleInfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;

{ Called for each supported locale. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin

```



```

with Languages do
begin
  for I := 0 to Count - 1 do
  begin
    ListBox1.Items.Add(Name[I]);
  end;
end;
end;

```

## Using resource DLLs

---

The executable, DLLs, and packages that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your EXE, DLL, or BPL files.

When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, DLL, or package. If there is not a resource module that matches both the language and the country, your application will try to locate a resource module that matches just the language. If there is no resource module that matches the language, your application will use the resources compiled with the executable, DLL, or package.

If you want your application to use a different resource module than the one that matches the locale of the local system, you can set a locale override entry in the Windows registry. Under the HKEY\_CURRENT\_USER\Software\Borland\Locales key, add your application's path and file name as a string value and set the data value to the extension of your resource DLLs. At startup, the application will look for resource DLLs with this extension before trying the system locale. Setting this registry entry allows you to test localized versions of your application without changing the locale on your system.

For example, the following procedure can be used in an install or setup program to set the registry key value that indicates the locale to use when loading Delphi applications:

```

procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software\Borland\Locales', True) then
      Reg.WriteString(LocaleOverride, FileName);
  finally
    Reg.Free;
  end;

```

Within your application, use the global *FindResourceHInstance* function to obtain the handle of the current resource module. For example:

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource DLLs.

## Dynamic switching of resource DLLs

---

In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the *ReInit* unit in your **uses** statement. (*ReInit* is located in the *Richedit* sample in the *Demos* directory.) To switch languages, you should call *LoadResourceModule*, passing the LCID for the new language, and then call *ReinitializeForms*.

For example, the following code switches the interface language to French:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

The advantage of this technique is that the current instance of the application and all of its forms are used. It is not necessary to update the registry settings and restart the application or reacquire resources required by the application, such as logging in to database servers.

When you switch resource DLLs the properties specified in the new DLL overwrite the properties in the running instances of the forms.

**Note** Any changes made to the form properties at runtime will be lost. Once the new DLL is loaded, default values are not reset. Avoid code that assumes that the form objects are reinitialized to the their startup state, apart from differences due to localization.

## Localizing applications

---

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

### Localizing resources

---

Ideally, your resources have been isolated into a resource DLL that contains form files (.dfm or .xfm) and a resource file. You can open your forms in the IDE and translate the relevant properties.

**Note** In a resource DLL project, you cannot add or delete components. It is possible, however, to change properties in ways that could cause runtime errors, so be careful to modify only those properties that require translation. To avoid mistakes, you can configure the Object Inspector to display only localizable properties; to do so, right-click in the Object Inspector and use the View menu to filter out unwanted property categories.

You can open the RC file and translate relevant strings. Use the StringTable editor by opening the RC file from the Project Manager.

## Deploying applications

Once your Delphi application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. What is required by a given application varies, depending on the type of application. The following sections describe considerations when deploying different types of applications:

- Deploying general applications
- Deploying CLX applications
- Deploying database applications
- Deploying Web applications
- Programming for varying host environments
- Software license requirements

**Note** Information included in these sections is for deploying applications on Windows. If writing cross-platform applications for deployment on Linux, you need to refer to deployment information provided in your Kylix documentation.

### Deploying general applications

---

Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. These are the main deployment concerns common to nearly all types of applications:

- Using installation programs
- Identifying application files

Delphi applications that access databases and those that run across the Web require additional installation steps beyond those that apply to general applications. For additional information on installing database applications, see “Deploying database applications” on page 13-6. For more information on installing Web applications, see “Deploying Web applications” on page 13-9. For more information on installing ActiveX controls, see “Deploying an ActiveX control on the Web” on page 38-15.

## Using installation programs

---

Simple Delphi applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for BDE database applications.

InstallShield Express is a setup toolkit that is bundled with Delphi. InstallShield Express is certified for use with Delphi and the Borland Database Engine. It is based on Windows Installer (MSI) technology.

InstallShield Express is not automatically installed when Delphi is installed, so it must be manually installed if you want to use it to create installation programs. Run the installation program from the Delphi CD to install InstallShield Express. For more information on using InstallShield Express to create installation programs, see the InstallShield Express online help.

Other setup toolkits are available. However, if deploying BDE database applications, you should only use toolkits based on MSI technology and those which are certified to deploy the Borland Database Engine.

## Identifying application files

Besides the executable file, a number of other files may need to be distributed with an application.

- Application files
- Package files
- Merge modules
- ActiveX controls

## Application files

The following types of files may need to be distributed with an application.

**Table 13.1** Application files

Type	File name extension
Program files	.exe and .dll
Package files	.bpl and .dcp
Help files	.hlp, .cnt, and .toc (if used) or any other help files your application supports
ActiveX files	.ocx (sometimes supported by a DLL)
Local table files	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

## Package files

If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. You can also use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express. See the next section for details.

Borland recommends installing the runtime package files supplied by Borland in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you created, it is recommended that you install them in the same directory as the application. Only the .BPL files need to be distributed.

**Note** If deploying packages with CLX applications, you need to include clx60.bpl rather than vcl60.bpl.

If you are distributing packages to other developers, supply both the .BPL and the .DCP files.

## Merge modules

InstallShield Express 3.0 is based on Windows Installer (MSI) technology. That is why Delphi includes merge modules. Merge modules provide a standard method that you can use to deliver shared code, files, resources, Registry entries, and setup logic to applications as a single compound file. You can use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express.

The runtime libraries have some interdependencies because of the way they are grouped together. The result of this is that when one package is added to an install project, the install tool will automatically add or report a dependency on one or more other packages. For example, if you add the VCLInternet merge module to an install project, the install tool should also automatically add or report a dependency on the VCLDatabase and StandardVCL modules.

The dependencies for each merge module are listed in the table below. The various install tools may react to these dependencies differently. The InstallShield for Windows Installer automatically adds the required modules if it can find them.

Other tools may simply report a dependency or may generate a build failure if all required modules are not included in the project.

**Table 13.2** Merge modules and their dependencies

Merge module	BPLs included	Dependencies
ADORTL	adortl60.bpl	DatabaseRTL, BaseRTL
BaseClientDataSet	cds60.bpl	DatabaseRTL, BaseRTL, DataSnap, dbExpress
BaseRTL	rtl60.bpl	No dependencies
BaseVCL	vcl60.bpl, vclx60.bpl	BaseRTL
BDEClientDataSet	bdecds60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress, BDERTL
BDEInternet	inetdbbde60.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl60.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl60.bpl	BaseRTL
DatabaseVCL	vcldb60.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsn60.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsncon60.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapCorba	dsnacrba60.bpl	DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DataSnapEntera	dsnarent60.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx60.bpl	DatabaseVCL, BaseVCL, BaseRTL
dbExpress	dbexpress60.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress60.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss60.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
FastNet	nmfast60.bpl	BaseVCL, BaseRTL
InterbaseVCL	vclib60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress, BaseVCL
Internet	inet60.bpl, inetdb60.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy60.bpl	BaseVCL, BaseRTL
Office2000Components	dcloffice2k60.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
QuickReport	qrpt60.bpl	BaseVCL, BaseRTL, BDERTL, DatabaseRTL
SampleVCL	vclsmpl60.bpl	BaseVCL, BaseRTL
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	BaseVCL, BaseRTL
VCLIE	vclie60.bpl	BaseVCL, BaseRTL
VisualCLX	visualclx60.bpl	BaseRTL
WebDataSnap	webdsn60.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL

**Table 13.2** Merge modules and their dependencies (continued)

Merge module	BPLs included	Dependencies
WebSnap	websnap60.bpl, vcljpg60.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmrtl60.bpl	Internet, DatabaseRTL, BaseRTL

## ActiveX controls

Certain components bundled with Delphi are ActiveX controls. The component wrapper is linked into the application's executable file (or a runtime package), but the .OCX file for the component also needs to be deployed with the application. These components include

- Chart FX, copyright SoftwareFX Inc.
- VisualSpeller Control, copyright Visual Components, Inc.
- Formula One (spreadsheet), copyright Visual Components, Inc.
- First Impression (VtChart), copyright Visual Components, Inc.
- Graph Custom Control, copyright Bits Per Second Ltd.

ActiveX controls of your own creation need to be registered on the deployment computer before use. Installation programs such as InstallShield Express automate this registration process. To manually register an ActiveX control, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows versions).

DLLs that support an ActiveX control also need to be distributed with an application.

## Helper applications

Helper applications are separate programs without which your Delphi application would be partially or completely unable to function. Helper applications may be those supplied with the operating system, by Borland, or they might be third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the documentation for the helper for specific information.

## DLL locations

You can install .dll files used only by a single application in the same directory as the application. DLLs that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community DLLs is to place them either in the Windows or the Windows\System directory. A better way is to create a dedicated directory for the common .dll files, similar to the way the Borland Database Engine is installed.



## Deploying CLX applications

---

If you are writing cross-platform applications that will be deployed on both Windows and Linux, you need to compile and deploy the applications on both platforms. The steps for deploying CLX applications are the same as those for general applications. For information on deploying general applications, see “Deploying general applications” on page 13-1. For information on installing database CLX applications, see “Deploying database applications” on page 13-6.

**Note** When deploying CLX applications on Windows, you need to include `qtintf.dll` with the application to include the CLX runtime. If deploying packages with CLX applications, you need to include `clx60.bpl` rather than `vcl60.bpl`.

See Chapter 10, “Using CLX for cross-platform development” for information on writing CLX applications.

## Deploying database applications

---

Applications that access databases involve special installation considerations beyond copying the application’s executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application’s executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require even more specialized handling on installation, because the files that make up the application are typically located on multiple computers.

Since several different database technologies (ADO, BDE, dbExpress, and InterBase Express) are supported, deployment requirements differ for each. Regardless of which you are using, you need to make sure that the client side software is installed on the system where you plan to run the database application. BDE, ADO, and dbExpress also require drivers to interact with the client-side software of the database. InterBase does not require drivers because the IBX components communicate directly with the database.

Specific information on how to deploy dbExpress, BDE, and multi-tiered database applications is described in the following sections:

- Deploying dbExpress database applications
- Deploying BDE applications
- Deploying multi-tiered database applications (DataSnap)

Database applications that use client datasets such as *TClientDataSet* or *TSQLClientDataSet* or dataset providers require you to include `libmidas.dcu` and `crtl.dcu` (for static linking when providing a standalone executable); if you are packaging your application (with the executable and any needed DLLs), you need to include `Midas.dll`.

If deploying database applications that use ADO, you need to be sure that MDAC version 2.1 or later is installed on the system where you plan to run the application. MDAC is automatically installed with software such as Windows 2000 and Internet Explorer version 5 or later. You also need to be sure the drivers for the database

server you want to connect to are installed on the client. No other deployment steps are required.

If deploying database applications that use InterBase Express, you need to be sure that the InterBase client is installed on the system where you plan to run the application. InterBase requires `gd32.dll` and `interbase.msg` to be located in an accessible directory. No other deployment steps are required. InterBase Express components communicate directly with the database and do not require additional drivers. For more information, refer to the Embedded Installation Guide posted on the Borland Web site.

In addition to the technologies described here, you can also use third-party database engines to provide database access for Delphi applications. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

## Deploying dbExpress database applications

---

dbExpress is a set of drivers that provide fast access to database information. dbExpress components support cross-platform development because they are also available on Linux. Refer to Chapter 22, “Using unidirectional datasets” for more information about using the dbExpress components.

You can deploy dbExpress applications either as a stand-alone executable file or as an executable file that includes associated dbExpress driver DLLs.

To deploy dbExpress applications as standalone executable files, the dbExpress object files must be statically linked into your executable. You do this by including the following DCUs, located in the `lib` directory:

**Table 13.3** dbExpress deployment as standalone executable

Database unit	When to include
dbExpInt	Applications connecting to InterBase databases
dbExpOra	Applications connecting to Oracle databases
dbExpDb2	Applications connecting to DB2 databases
dbExpMy	Applications connecting to MySQL databases
Crtl, MidasLib	Required by dbExpress executables that use client datasets such as <i>TSQLClientDataSet</i>

If you are not deploying a standalone executable, you can deploy associated dbExpress drivers and DataSnap DLLs with your executable. The following table lists the appropriate DLLs and when to include them:

**Table 13.4** dbExpress deployment with driver DLLs

Database DLL	When to deploy
dbexpint.dll	Applications connecting to InterBase databases
dbexpora.dll	Applications connecting to Oracle databases
dbexpdb2.dll	Applications connecting to DB2 databases
dbexpmy.dll	Applications connecting to MySQL databases
Midas.dll	Required by database applications that use client datasets

## Deploying BDE applications

---

The Borland Database Engine (BDE) defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables.

Database access for an application is provided by various database engines. An application can use the BDE or a third-party database engine. SQL Links is provided (not available in all versions) to enable native access to SQL database systems. The following sections describe installation of the database access elements of an application:

- Borland Database Engine
- SQL Links

### Borland Database Engine

For standard Delphi data components to have database access, the Borland Database Engine (BDE) must be present and accessible. See the BDEDEPLOY document for specific rights and limitations on redistributing the BDE.

Borland recommends use of InstallShield Express (or other certified installation program) for installing the BDE. InstallShield Express will create the necessary registry entries and define any aliases the application may require. Using a certified installation program to deploy the BDE files and subsets is important because:

- Improper installation of the BDE or BDE subsets can cause other applications using the BDE to fail. Such applications include not only Borland products, but many third-party programs that use the BDE.
- Under Windows 9x and Windows NT, BDE configuration information is stored in the Windows registry instead of .INI files, as was the case under 16-bit Windows. Making the correct entries and deletions for install and uninstall is a complex task.

It is possible to install only as much of the BDE as an application actually needs. For instance, if an application only uses Paradox tables, it is only necessary to install that portion of the BDE required to access Paradox tables. This reduces the disk space needed for an application. Certified installation programs, like InstallShield Express, are capable of performing partial BDE installations. Be sure to leave BDE system files that are not used by the deployed application, but that are needed by other programs.

### SQL Links

SQL Links provides the drivers that connect an application (through the Borland Database Engine) with the client software for an SQL database. See the DEPLOY document for specific rights and limitations on redistributing SQL Links. As is the case with the Borland Database Engine (BDE), SQL Links must be deployed using InstallShield Express (or other certified installation program).

**Note** SQL Links only connects the BDE to the client software, not to the SQL database itself. It is still necessary to install the client software for the SQL database system

used. See the documentation for the SQL database system or consult the vendor that supplies it for more information on installing and configuring client software.

Table 13.5 shows the names of the driver and configuration files SQL Links uses to connect to the different SQL database systems. These files come with SQL Links and are redistributable in accordance with the Delphi license agreement.

**Table 13.5** SQL database client software files

Vendor	Redistributable files
Oracle 7	SQLORA32.DLL and SQL_ORA.CNF
Oracle8	SQLORA8.DLL and SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL and SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL and SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL and SQL_MSS.CNF
Informix 7	SQLINF32.DLL and SQL_INF.CNF
Informix 9	SQLINF9.DLL and SQL_INF9.CNF
DB/2	SQLDB232.DLL and SQL_DB2.CNF
InterBase	SQLINT32.DLL and SQL_INT.CNF

Install SQL Links using InstallShield Express or other certified installation program. For specific information concerning the installation and configuration of SQL Links, see the help file SSQLNK32.HLP, by default installed into the main BDE directory.

## Deploying multi-tiered database applications (DataSnap)

DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

Install DataSnap along with a multi-tier application using InstallShield Express (or other Borland-certified installation scripting utility). See the DEPLOY document (found in the main Delphi directory) for details on the files that need to be redistributed with an application. Also see the REMOTE document for related information on what DataSnap files can be redistributed and how.

## Deploying Web applications

Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension DLLs (ISAPI and Apache), CGI applications, and ActiveForms.

The steps for deploying Web applications are the same as those for general applications, except the application's files are deployed on the Web server. For information on installing general applications, see "Deploying general applications" on page 13-1. For information on deploying database Web applications, see "Deploying database applications" on page 13-6.

Here are some special considerations for deploying Web applications:

- For BDE database applications, the Borland Database Engine (or alternate database engine) is installed with the application files on the Web server.
- For dbExpress applications, the dbExpress DLLs must be included in the path. If included, the dbExpress driver is installed with the application files on the Web server.
- Security for the directories should be set so that the application can access all needed database files.
- The directory containing an application must have read and execute attributes.
- The application should not use hard-coded paths for accessing database or other files.
- The location of an ActiveX control is indicated by the CODEBASE parameter of the <OBJECT> HTML tag.

Deployment on Apache is described in the next section.

## Deployment on Apache

---

WebBroker supports Apache version 1.3.9 and later for DLLs and CGI applications. Apache is configured by files in the conf directory.

If creating Apache DLLs, you need to be sure to set appropriate directives in the Apache server configuration file, called httpd.conf. The DLL should be physically located in the Modules subdirectory of the Apache software.

If creating CGI applications, the physical directory (specified in the Directory directive of the httpd.conf file) must have the ExecCGI option set to allow execution of programs so the CGI script can be executed. To ensure that permissions are set up properly, you need to either use the ScriptAlias directive or set Options ExecCGI to on.

The ScriptAlias directive creates a virtual directory on your server and marks the target directory as containing CGI scripts. For example, you could add the following line to your httpd.conf file:

```
ScriptAlias /cgi-bin "c:\inetpub\cgi-bin"
```

This would cause requests such as /cgi-bin/mycgi to be satisfied by running the script c:\inetpub\cgi-bin\mycgi.

You can also set Options to All or to ExecCGI using the Directory directive in httpd.conf. The Options directive controls which server features are available in a particular directory. Directory directives are used to enclose a set of directives that apply to the named directory and its subdirectories. An example of the Directory directive is shown below:

```
<Directory <apache-root-dir>\cgi-bin>
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

In this example, Options is set to ExecCGI permitting execution of CGI scripts in the directory cgi-bin.

**Note** Apache executes locally on the server within the account specified in the User directive in the httpd.conf file. Make sure that the user has the appropriate rights to access the resources needed by the application.

Information concerning the deployment of Apache software can be found in the Apache LICENSE file, which is included in the Apache distribution. You can also find configuration information on the Apache Web site at [www.apache.org](http://www.apache.org).

## Programming for varying host environments

---

Due to the nature of various operating system environments, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths
- Fonts
- Operating systems versions
- Helper applications
- DLL locations

### Screen resolutions and color depths

---

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

### Considerations when not dynamically resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution.

Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.

## Considerations when dynamically resizing forms and controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

- The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen.Height* or the *TScreen.Width* property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.
- Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *CustomForm.Scaled* property to *True* and calling the *TWinControl.ScaleBy* method (*TWidgetControl.ScaleBy* for cross-platform applications). The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.
- The controls on a form can be resized manually, instead of automatically with the *TWinControl.ScaleBy* method (*TWidgetControl.ScaleBy* for cross-platform applications), by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.
- If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 ( $640 / 1024 = 0.625$ ). The original font size of 8 is reduced to 5 ( $8 * 0.625 = 5$ ). Text in the application appears jagged and unreadable as it is displayed in the reduced font size.

- Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen resolutions. This effect is offset by setting the *AutoSize* property of these controls to *False*.
- Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

## Accommodating varying color depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

## Fonts

---

The Windows and Linux operating systems come with a standard sets of fonts. When designing an application to be deployed on other computers, realize that not all computers will have fonts outside the standard sets.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Windows has a safety measure to account for attempts to use a font that does not exist on the computer. It substitutes another, existing font that it considers the closest match. While this may circumvent errors concerning missing fonts, the end result may be a degradation of the visual appearance of the application. It is better to prepare for this eventuality at design time.

To make a nonstandard font available to a Windows application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .fot file for the nonstandard font with the application.



## Operating systems versions

---

When using operating system APIs or accessing areas of the operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different operating system versions.

To account for this possibility, you have a few options:

- Specify in the application's system requirements the versions of the operating system on which the application can run. It is the user's responsibility to install and use the application only under compatible operating system versions.
- Check the version of the operating system as the application is installed. If an incompatible version of the operating system is present, either halt the installation process or at least warn the installer of the problem.
- Check the operating system version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of the operating system is present, abort the process and alert the user. Alternately, provide different code to run dependent on different operating system versions. For example, some operations are performed differently on Windows 95/98 than on Windows NT/2000. Use the Windows API function *GetVersionEx* to determine the Windows version.

## Software license requirements

---

The distribution of some files associated with Delphi applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:

- DEPLOY
- README
- No-nonsense license agreement
- Third-party product documentation

### DEPLOY

---

DEPLOY covers the some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with a Delphi application. DEPLOY is a document installed in the main Delphi directory. The topics covered include

- .exe, .dll, and .bpl files
- Components and design-time packages
- Borland Database Engine (BDE)
- ActiveX controls
- Sample Images
- SQL Links

## **README**

---

README contains last minute information about Delphi, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. README is a document installed into the main Delphi directory.

## **No-nonsense license agreement**

---

The Delphi no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Delphi.

## **Third-party product documentation**

---

Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Delphi applications prior to distribution.



## Developing database applications

The chapters in “Developing Database Applications” present concepts and skills necessary for creating Delphi database applications.

**Note** You need the Professional or Enterprise edition of Delphi to develop database applications. To implement more advanced Client/Server databases, you need the Delphi features available in the Enterprise edition.



# Designing database applications

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify existing data.

This chapter introduces some common considerations for designing a database application and the decisions involved in designing a user interface.

## Using databases

---

Delphi includes many components for accessing databases and representing the information they contain. They are grouped according to the data access mechanism:

- The BDE page of the component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the BDE components, see Chapter 20, “Using the Borland Database Engine.”
- The ADO page of the component palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. There is a broad range of ADO drivers available for connecting to different database servers. Using ADO-based components lets you

integrate your application into an ADO-based environment (for example, making use of ADO-based application servers). For more information about using the ADO components, see Chapter 21, “Working with ADO components”.

- The dbExpress page of the component palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. In addition, dbExpress components support cross-platform development because they are also available on Linux. However, dbExpress database components also support the narrowest range of data manipulation functions. For more information about using the dbExpress components, see Chapter 22, “Using unidirectional datasets”.
- The InterBase page of the Component palette contains components that access InterBase databases directly, without going through a separate engine layer.
- The Data Access page of the component palette contains components that can be used with any data access mechanism. This page includes *TClientDataset*, which can work with data stored on disk or, using the *TDataSetProvider* component also on this page, with components from one of the other groups. For more information about using client datasets, see Chapter 23, “Using client datasets”. For more information about *TDataSetProvider*, see Chapter 24, “Using provider components”.

**Note** Different versions of Delphi include different drivers for accessing database servers using the BDE, ADO, or dbExpress.

When designing a database application, you must decide which set of components to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

In addition to choosing a data access mechanism, you must choose a database server. There are different types of databases and you will want to consider the advantages and disadvantages of each type before settling on a particular database server.

All types of databases contain tables which store information. In addition, most (but not all) servers support additional features such as

- Database security
- Transactions
- Referential integrity, stored procedures, and triggers

## Types of databases

---

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Delphi provides support for two types of relational database server:

- **Remote database servers** reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote

Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique “dialect” of SQL. Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered applications** because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions. Some local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.
- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.
- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are less expensive to operate because they do not require separately installed servers or expensive site licenses.

## Database security

---

Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBASE, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.



Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see “Controlling server login” on page 17-4.

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can't be breached by reading the password from the application.

If you require your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password at the point when you will eventually log in to the SQL database, rather than when opening individual tables.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password that is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use HTTPs, CORBA, or COM+ to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

## Transactions

---

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover

the durable state after disk media failures. Transactions also form the basis of multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user's commands can't disrupt the unity of another user's transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is not part of most local databases, although it is provided by local InterBase. In addition, the BDE drivers provide limited transaction support for some local databases. Database transaction support is provided by the component that represents the database connection. For details on managing transactions using a database connection component, see "Managing transactions" on page 17-5.

In multi-tiered applications, you can create transactions that include actions other than database operations or that span multiple databases. For details on using transactions in multi-tiered applications, see "Managing transactions in multi-tiered applications" on page 25-18.

## Referential integrity, stored procedures, and triggers

---

All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

## Database architecture

---

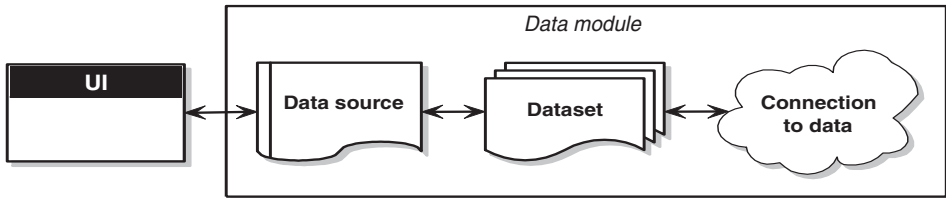
Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

## General structure

---

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in Figure 14.1:

**Figure 14.1** Generic Database Architecture



### The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see “Designing the user interface” on page 14-15.

### The data module

If you have isolated your user interface into its own form, you can use a data module to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

### The data source

The first item in the data module is a data source. The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

## The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Chapter 18, “Understanding datasets”.

## The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. There are four basic mechanisms for connecting to the data:

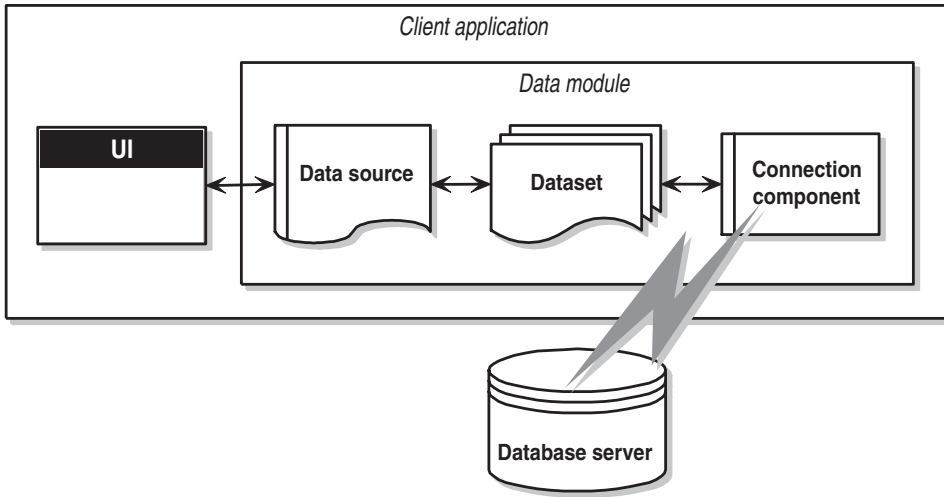
- Connecting directly to a database server. Most datasets use a descendant of *TCustomConnection* to represent the connection to a database server.
- Using a dedicated file on disk. Client datasets support the ability to work with a dedicated file on disk. No separate connection component is needed when working with a dedicated file because the client dataset itself knows how to read from and write to the file.
- Connecting to another dataset. Client datasets can work with data provided by another dataset. A *TDataSetProvider* component serves as an intermediary between the client dataset and its source dataset. This dataset provider can reside in the same data module as the client dataset, or it can be part of an application server running on another machine. If the provider is part of an application server, you also need a special descendant of *TCustomConnection* to represent the connection to the application server.
- Obtaining data from an RDS DataSpace object. ADO datasets can use a *TRDSConnection* component to marshal data in multi-tier database applications that are built using ADO-based application servers.

Sometimes, these mechanisms can be combined in a single application.

## Connecting directly to a database server

---

The most common database architecture is the one where the dataset uses a connection component to establish a connection to a database server. The dataset then fetches data directly from the server and posts edits directly to the server. This is illustrated in Figure 14.2.

**Figure 14.2** Connecting directly to the database server

Each type of dataset uses its own type of connection component, which represents a single data access mechanism:

- If the dataset is a BDE dataset such as *TTable*, *TQuery*, or *TStoredProc*, the connection component is a *TDataBase* object. You connect the dataset to the database component by setting its *Database* property. You do not need to explicitly add a database component when using BDE dataset. If you set the dataset's *DatabaseName* property, a database component is created for you automatically at runtime.
- If the dataset is an ADO dataset such as *TADODataset*, *TADOTable*, *TADOQuery*, or *TADOStoredProc*, the connection component is a *TADOConnection* object. You connect the dataset to the ADO connection component by setting its *ADOConnection* property. As with BDE datasets, you do not need to explicitly add the connection component: instead you can set the dataset's *ConnectionString* property.
- If the dataset is a dbExpress dataset such as *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, or *TSQLStoredProc*, the connection component is a *TSQLConnection* object. You connect the dataset to the SQL connection component by setting its *SQLConnection* property. When using dbExpress datasets, you must explicitly add the connection component. Another difference between dbExpress datasets and the other datasets is that dbExpress datasets are always read-only and unidirectional: This means you can only navigate by iterating through the records in order, and you can't use the dataset methods that support editing.
- If the dataset is an InterBase Express dataset such as *TIBDataSet*, *TIBTable*, *TIBQuery*, or *TIBStoredProc*, the connection component is a *TIBDatabase* object. You connect the dataset to the IB database component by setting its *Database* property. As with dbExpress datasets, you must explicitly add the connection component.

In addition to the components listed above, you can use a specialized client dataset such as *TBDEClientDataSet*, *TSQLClientDataSet*, or *TIBClientDataSet* with a database connection component. When using one of these client datasets, specify the appropriate type of connection component as the value of the *DBConnection* property.

Although each type of dataset uses a different connection component, they all perform many of the same tasks and surface many of the same properties, methods, and events. For more information on the commonalities among the various database connection components, see Chapter 17, “Connecting to databases”.

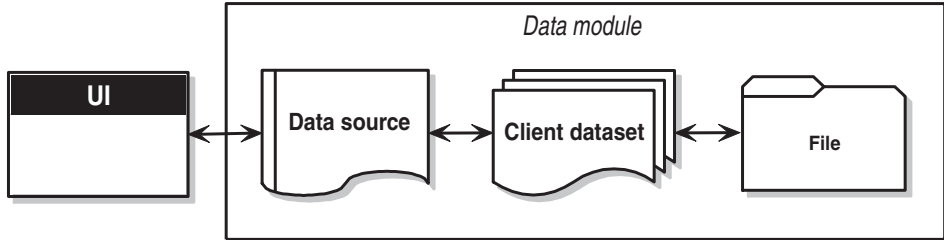
This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

**Note** The connection components or drivers needed to create two-tiered applications are not available in all version of Delphi.

## Using a dedicated file on disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses MyBase, the ability of client datasets to save themselves to a file and to load the data from a file. This architecture is illustrated in Figure 14.3:

**Figure 14.3** A file-based database application



When using this file-based approach, your application writes changes to disk using the client dataset’s *SaveToFile* method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the *LoadFromFile* method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the *FileName* property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (If you do not statically link in `midaslib.dcu`, the client dataset does require `midas.dll`). There is no need for site licenses or database administration.

In addition, some versions of Delphi let you convert between arbitrary XML documents and the data packets that are used by a client dataset. Thus, the file-based approach can be used to work with XML documents as well as dedicated datasets. For information about converting between XML documents and client dataset data packets, see Chapter 26, “Using XML in database applications”.

The file-based approach offers no support for multiple users. The dataset should be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other’s data files.

For more information about using a client dataset with data stored on disk, see “Using a client dataset with file-based data” on page 23-31.

## Connecting to another dataset

---

There are specialized client datasets that use the BDE or *dbExpress* to connect to a database server. These specialized client datasets are, in fact, composite components that include another dataset internally to access the data and an internal provider component to package the data from the source dataset and to apply updates back to the database server. These composite components require some additional overhead, but provide certain benefits:

- Client datasets provide the most robust way to work with cached updates. By default, other types of datasets post edits directly to the database server. You can reduce network traffic by using a dataset that caches updates locally and applies them all later in a single transaction. For information on the advantages of using client datasets to cache updates, see “Using a client dataset to cache updates” on page 23-15.
- Client datasets can apply edits directly to a database server when the dataset is read-only. When using *dbExpress*, this is the only way to edit the data in the dataset (it is also the only way to navigate freely in the data when using *dbExpress*). Even when not using *dbExpress*, the results of some queries and all stored procedures are read-only. Using a client dataset provides a standard way to make such data editable.
- Because client datasets can work directly with dedicated files on disk, using a client dataset can be combined with a file-based model to allow for a flexible “briefcase” application. For information on the briefcase model, see “Combining approaches” on page 14-14.

In addition to these specialized client datasets, there is a generic client dataset (*TClientDataSet*), which does not include an internal dataset and dataset provider. Although *TClientDataSet* has no built-in database access mechanism, you can connect it to another, external, dataset from which it fetches data and to which it sends updates. Although this approach is a bit more complicated, there are times when it is preferable:

- Because the source dataset and dataset provider are external, you have more control over how they fetch data and apply updates. For example, the provider component surfaces a number of events that are not available when using a specialized client dataset to access data.
- When the source dataset is external, you can link it in a master/detail relationship with another dataset. An external provider automatically converts this arrangement into a single dataset with nested details. When the source dataset is internal, you can't create nested detail sets this way.
- Connecting a client dataset to an external dataset is an architecture that easily scales up to multiple tiers. Because the development process can get more involved and expensive as the number of tiers increases, you may want to start developing your application as a single-tiered or two-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. If you think you may eventually use a multi-tiered architecture, it can be worthwhile to start by using a client dataset with an external source dataset. This way, when you move the data access and manipulation logic to a middle tier, you protect your development investment because the code can be reused as your application grows.
- *TClientDataSet* can link to any source dataset. This means you can use custom datasets (third-party components) for which there is no corresponding specialized client dataset. Some versions of Delphi even include special provider components that connect a client dataset to an XML document rather than another dataset. (This works the same way as connecting a client dataset to another (source) dataset, except that the XML provider uses an XML document rather than a dataset. For information about these XML providers, see "Using an XML document as the source for a provider" on page 26-8.)

There are two versions of the architecture that connects a client dataset to an external dataset:

- Connecting a client dataset to another dataset in the same application.
- Using a multi-tiered architecture.

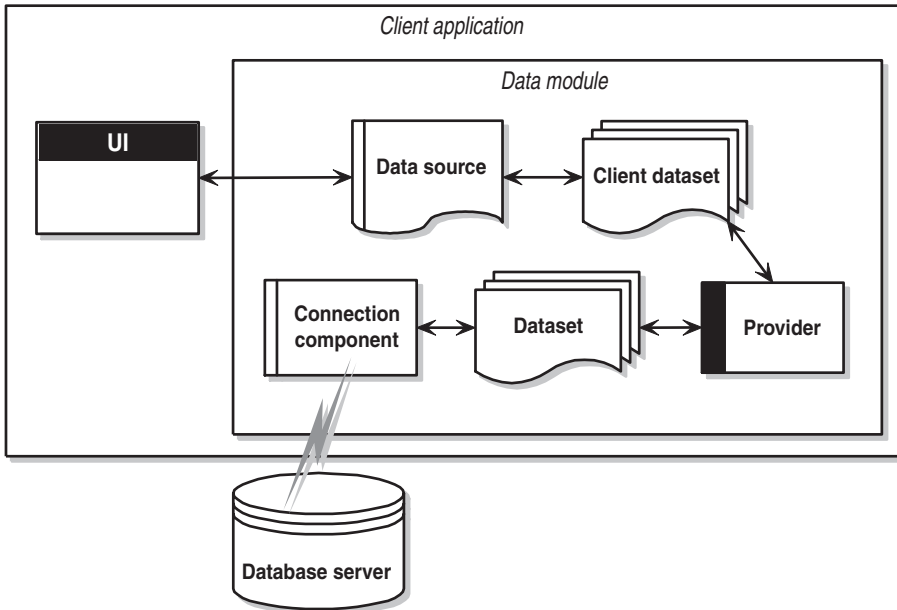
### **Connecting a client dataset to another dataset in the same application**

By using a provider component, you can connect *TClientDataSet* to another (source) dataset. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets



(which client datasets create) back to a database server. The architecture for this is illustrated in Figure 14.4:

**Figure 14.4** Architecture combining a client dataset and another dataset



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

To link the client dataset to the provider, set its *ProviderName* property to the name of the provider component. The provider must be in the same data module as the client dataset. To link the provider to the source dataset, set its *DataSet* property.

Once the client dataset is linked to the provider and the provider is linked to the source dataset, these components automatically handle all the details necessary for fetching, displaying, and navigating through the database records (assuming the source dataset is connected to a database). To apply user edits back to the database, you need only call the client dataset's *ApplyUpdates* method.

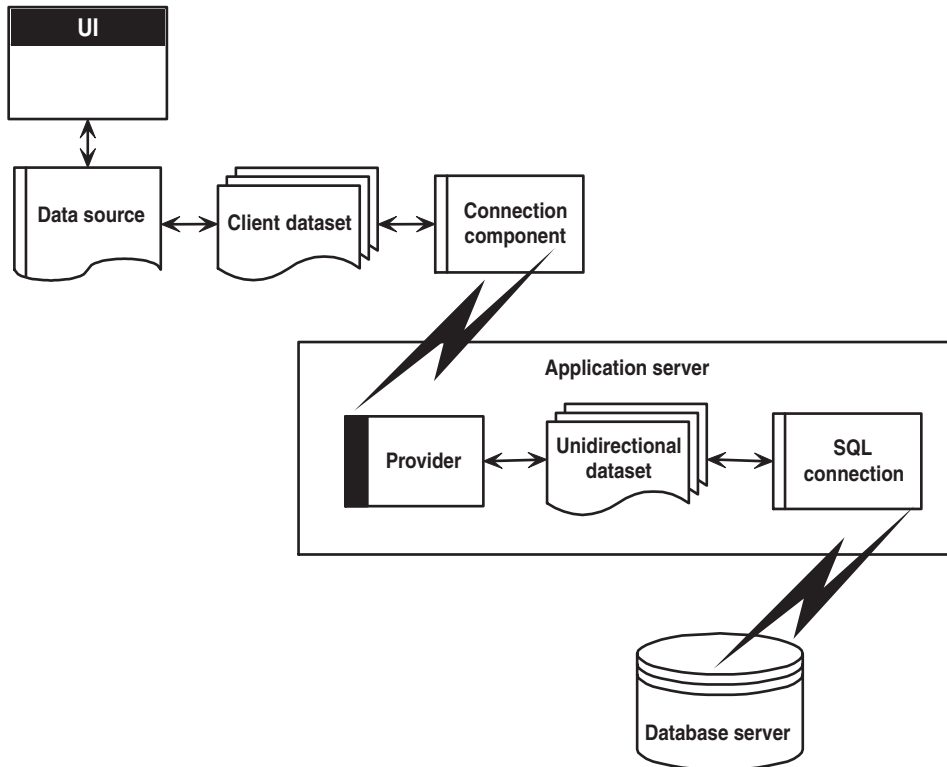
For more information on using a client dataset with a provider, see "Using a client dataset with a provider" on page 23-23.

### Using a multi-tiered architecture

When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a multi-tiered

application. Multi-tiered applications have middle tiers between the client application and database server. The architecture for this is illustrated in Figure 14.5:

**Figure 14.5** Multi-tiered database architecture



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data, while ensuring consistent data logic. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading data-processing over several systems.

The multi-tiered architecture is very similar to the previous model. It differs mainly in that source dataset that connects to the database server and the provider that acts as an intermediary between that source dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the “remote data broker”).

Because the provider has moved to a separate application, the client dataset can no longer connect to the source dataset by simply setting its *ProviderName* property. In addition, it must use some type of connection component to locate and connect to the application server.

There are several types of connection components that can connect a client dataset to an application server. They are all descendants of *TCustomRemoteServer*, and differ primarily in the communication protocol they use (TCP/IP, HTTP, DCOM, SOAP, or CORBA). Link the client dataset to its connection component by setting the *RemoteServer* property.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

For more information about connecting a client dataset to an application server, see Chapter 25, “Creating multi-tiered applications”.

## Combining approaches

---

The previous sections describe several architectures you can use when writing database applications. There is no reason, however, why you can't combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in “Using a dedicated file on disk” on page 14-9 with another approach such as those described in “Connecting a client dataset to another dataset in the same application” on page 14-11 or “Using a multi-tiered architecture” on page 14-12. These combinations are easy because all models use a client dataset to represent the data that appears in the user interface. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An onsite company database contains customer contact data that sales representatives can use and update in the field. While onsite, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales reps return onsite, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The client dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back onsite, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

# Designing the user interface

---

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see Chapter 15, "Using data controls."

In addition to the basic data controls, you may also want to introduce other elements into your user interface:

- You may want your application to analyze the data contained in a database. Applications that analyze data do more than just display the data in a database, they also summarize the information in useful formats to help users grasp the impact of that data.
- You may want to print reports that provide a hard copy of the information displayed in your user interface.
- You may want to create a user interface that can be viewed from Web browsers. The simplest Web-based database applications are described in "Using database information in responses" on page 28-17. In addition, you can combine the Web-based approach with the multi-tiered architecture, as described in "Writing Web-based client applications."

## Analyzing data

---

Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data.

The *TDBChart* component on the Data Controls page of the Component palette lets you present database information in a graphical format that enables users to quickly grasp the import of database information.

In addition, some versions of Delphi include a Decision Cube page on the Component palette. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information about using the Decision Cube components, see Chapter 16, "Using decision support components".

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates with a client dataset. For more information about using maintained aggregates, see "Using maintained aggregates" on page 23-11.

## Writing reports

---

If you want to let your users print database information from the datasets in your application, you can use the report components on the QReport page of the Component palette. Using these components you can visually build banded reports to present and summarize the information in your database tables. You can add summaries to group headers or footers to analyze the data based on grouping criteria.

Start a report for your application by selecting the QuickReport icon from the New Items dialog. Select File | New from the main menu, and go to the page labeled Business. Double-click the QuickReport Wizard icon to launch the wizard.

**Note** See the QuickReport demo that ships with Delphi for an example of how to use the components on the QReport page.

## Using data controls

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in “Displaying a single record” on page 15-7.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. “Choosing how to organize the data” on page 15-7 describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator. For more information about using a data navigator, see “Navigating and manipulating records” on page 15-28.

**Note** More complex data-aware controls for decision support are discussed in Chapter 16, “Using decision support components.”

Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described below.

## Using common data control features

---

The following tasks are common to most data controls:

- Associating a data control with a dataset
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display
- Enabling mouse, keyboard, and timer events

Data controls let you display and edit fields of data associated with the current record in a dataset. Table 15.1 summarizes the data controls that appear on the Data Controls page of the Component palette.

**Table 15.1** Data controls

Data control	Description
<i>TDBGrid</i>	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
<i>TDBNavigator</i>	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
<i>TDBText</i>	Displays data from a field as a label.
<i>TDBEdit</i>	Displays data from a field in an edit box.
<i>TDBMemo</i>	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
<i>TDBImage</i>	Displays graphics from a data field in a graphics box.
<i>TDBListBox</i>	Displays a list of items from which to update a field in the current data record.
<i>TDBComboBox</i>	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
<i>TDBCheckBox</i>	Displays a check box that indicates the value of a Boolean field.
<i>TDBRadioGroup</i>	Displays a set of mutually exclusive options for a field.
<i>TDBLookupListBox</i>	Displays a list of items looked up from another dataset based on the value of a field.
<i>TDBLookupComboBox</i>	Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box.
<i>TDBCtrlGrid</i>	Displays a configurable, repeating set of data-aware controls within a grid.
<i>TDBRichEdit</i>	Displays formatted data from a field in an edit box.

Data controls are data-aware at design time. When you associate the data control with an active dataset while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see “Creating persistent fields” on page 19-4.

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

## Associating a data control with a dataset

---

Data controls connect to datasets by using a data source. A data source component (*TDataSource*) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

**Note** Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset,

- 1 Place a dataset in a data module (or on a form), and set its properties as appropriate.
- 2 Place a data source in the same data module (or form). Using the Object Inspector, set its *DataSet* property to the dataset you placed in step 1.
- 3 Place a data control from the Data Access page of the Component palette onto a form.
- 4 Using the Object Inspector, set the *DataSource* property of the control to the data source component you placed in step 2.
- 5 Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid*, *TDBCtrlGrid*, and *TDBNavigator* because they access all available fields in the dataset.
- 6 Set the *Active* property of the dataset to *True* to display data in the control.

## Changing the associated dataset at runtime

In the preceding example, the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
```



```

else
    DataSet := Customers;
end;

```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

```

procedure TForm2.FormCreate (Sender : TObject);
begin
    DataSource1.Dataset := Form1.Table1;
end;

```

## Enabling and disabling the data source

The data source has an *Enabled* property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's *DisableControls* and *EnableControls* methods because they affect all attached data sources.

## Responding to changes mediated by the data source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The *OnDataChange* event occurs whenever the data in a record may have changed, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The *OnUpdateData* event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the underlying database server or local cache.

The *OnStateChange* event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's *State* property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

```

procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
    CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
    CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
    CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
    :
end;

```

**Note** For more information about dataset states, see “Determining dataset states” on page 18-3.

## Editing and updating data

---

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

**Note** Unidirectional datasets never permit users to edit and update data.

### Enabling editing in controls on user entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source’s *AutoEdit* property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a *TDBNavigator* control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see “Navigating and manipulating records” on page 15-28. Alternately, you can write code that calls the dataset’s *Edit* method when you want to put the dataset into edit mode.

### Editing data in a control

A data control can only post edits to its associated dataset if the dataset’s *CanModify* property is *True*. *CanModify* is always *False* for unidirectional datasets. Some datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*.

**Note** Whether a dataset can update data depends on whether the underlying database table permits updates.

Even if the dataset’s *CanModify* property is *True*, the *Enabled* property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is *True* (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property of the data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. Clearly, you will want to ensure that the control’s *ReadOnly* property is *True* when the dataset’s *CanModify* property is *False*. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying dataset when you *Tab* from the control. If you press *Esc* before you *Tab* from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In *TDBGrid*, modifications are posted when you move to a different record; you can press *Esc* in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware controls associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

**Note** If your application caches updates (for example, using a client dataset), all modifications are posted to an internal cache. These modifications are not applied to the underlying database table until you call the dataset's *ApplyUpdates* method.

## Disabling and enabling data display

---

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

*DisableControls* is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...finally** statement so that you can re-enable controls even if an exception occurs during processing. The **finally** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    :
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

## Refreshing data display

---

The *Refresh* method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. If you are using cached updates, before you refresh the dataset you must apply any updates the dataset has currently cached.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

## Enabling mouse, keyboard, and timer events

---

The *Enabled* property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

## Choosing how to organize the data

---

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records.

In addition, you will want to add controls to navigate and manipulate records. The *TDBNavigator* control provides built-in support for many of the functions you may want to perform.

## Displaying a single record

---

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. These controls are typically data-aware versions of other controls that are available on the component palette. For example, the *TDBEdit* control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field.

## Displaying data as labels

*TDBText* is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

*TDBText* gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

**Note** When you place a *TDBText* component on a form, make sure its *AutoSize* property is *True* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is *False*, and the control is too small, data display is clipped.

## Displaying and editing fields in an edit box

*TDBEdit* is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open *TClientDataSet* called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

## Displaying and editing text in a memo control

*TDBMemo* is a data-aware component—similar to the standard *TMemo* component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the memo control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the *ScrollBars* property. To prevent

word wrap, set the *WordWrap* property to *False*. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the *Font* property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

## Displaying and editing text in a rich edit memo control

*TDBRichEdit* is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBRichEdit* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well.

**Note** While *TDBRichEdit* provides properties and methods to enter and work with rich text, it does not provide any user interface components to make these formatting options available to the user. Your application must implement the user interface to surface rich text capabilities.

By default, *TDBRichEdit* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the rich edit control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Because the *TDBRichEdit* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBRichEdit* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBRichEdit* displays the field name rather than actual data. Double-click inside the control to view the actual data.

## Displaying and editing graphics fields in an image control

*TDBImage* is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch* property to *True* to resize the graphic to fit within an image control as it is resized.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should automatically be displayed. If you set *AutoDisplay* to *False*, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

## Displaying and editing data in list and combo boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list and combo box controls:

- *TDBListBox*, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- *TDBComboBox*, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- *TDBLookupListBox*, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- *TDBLookupComboBox*, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

**Note** At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. *Backspace* and *Esc* cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

### Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items* property in the Object Inspector. Then type in the items that you want to have appear in the list. At runtime, use the methods of the *Items* property to manipulate its string list.

When a *TDBListBox* or *TDBComboBox* control is linked to a field through its *DataField* property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, *TDBComboBox* displays the current value for the field in its edit box, regardless of whether it appears in the *Items* list.

For *TDBListBox*, the *Height* property determines how many items are visible in the list box at one time. The *IntegralHeight* property controls how the last item can appear. If *IntegralHeight* is *False* (the default), the bottom of the list box is determined

by the *ItemHeight* property, and the bottom item may not be completely displayed. If *IntegralHeight* is *True*, the visible bottom item in the list box is fully displayed.

For *TDBComboBox*, the *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the *Items* list is displayed at runtime:

- *Style* determines the display style of the component:
  - *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
  - *csSimple*: Combines an edit control with a fixed size list of items that is always displayed. When setting *Style* to *csSimple*, be sure to increase the *Height* property so that the list is displayed.
  - *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
  - *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.
- *DropDownCount*: the maximum number of items displayed in the list. If the number of *Items* is greater than *DropDownCount*, the user can scroll the list. If the number of *Items* is less than *DropDownCount*, the list will be just large enough to display all the *Items*.
- *ItemHeight*: The height of each item when style is *csOwnerDrawFixed*.
- *Sorted*: If *True*, then the *Items* list is displayed in alphabetical order.

### Displaying and editing data in lookup list and combo boxes

Lookup list boxes and lookup combo boxes (*TDBLookupListBox* and *TDBLookupComboBox*) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

- **A lookup field defined for a dataset.**  
To specify list box items using a lookup field, the dataset to which you link the



control must already define a lookup field. (This process is described in “Defining a lookup field” on page 19-8). To specify the lookup field for the list box items,

- 1 Set the *DataSource* property of the list box to the data source for the dataset containing the lookup field to use.
- 2 Choose the lookup field to use from the drop-down list for the *DataField* property.

When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

- **A secondary data source, data field, and key.**

If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item. To specify a secondary data source for list box items,

- 1 Set the *DataSource* property of the list box to the data source for the control.
- 2 Choose a field into which to insert looked-up values from the drop-down list for the *DataField* property. The field you choose cannot be a lookup field.
- 3 Set the *ListSource* property of the list box to the data source for the dataset that contain the field whose values you want to look up.
- 4 Choose a field to use as a lookup key from the drop-down list for the *KeyField* property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
- 5 Choose a field whose values to return from the drop-down list for the *ListField* property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the *RowCount* property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the *DropDownRows* property instead.

**Note** You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see “Defining a lookup list column” on page 15-20.

## Handling Boolean field values with check boxes

*TDBCheckBox* is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of *ValueChecked* and *ValueUnchecked* properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

**Note** The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to “true,” but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the field for the current record contains values of “true,” “Yes,” or “On,” then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to “false,” but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

## Restricting field values with radio controls

*TDBRadioGroup* is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items* property determines the radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button’s label.

If the current value of a field associated with a radio group matches one of the strings in the *Items* property, that radio button is selected. For example, if three strings, “Red,” “Yellow,” and “Blue,” are listed for *Items*, and the field for the current record contains the value “Blue,” then the third button in the group appears selected.

**Note** If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the *Values* property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains “Red,” “Yellow,” and “Blue,” and *Values* contains “Magenta,” “Yellow,” and “Cyan.” If a user selects the button labeled “Red,” “Magenta” is posted to the database.

If strings for *Values* are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

## Displaying multiple records

---

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application’s user interface more compelling and effective. They are discussed in “Viewing and editing data with TDBGrid” on page 15-15 and “Creating a grid that contains other data-aware controls” on page 15-26.

**Note** You can’t display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

- **Master-detail forms:** You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see “Creating master/detail relationships” on page 18-34 and “Establishing master/detail relationships using parameters” on page 18-46.
- **Drill-down forms:** In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

**Tip** It is generally not a good idea to combine these two approaches on a single form. It is usually confusing for users to understand the data relationships in such forms.

## Viewing and editing data with TDBGrid

A *TDBGrid* control lets you view and edit records in a dataset in a tabular grid format.

**Figure 15.1** TDBGrid control

The diagram shows a TDBGrid control displaying a table of data. The table has four columns: VendorName, Address1, City, and State. The first row is highlighted, and a record indicator points to it. Labels above the table indicate the current field and column titles.

Record indicator	Current field	Column titles		
	VendorName	Address1	City	State
→	Cacor Corporation	161 Southfield Rd	Southfield	OH
	Underwater	50 N 3rd Street	Indianapolis	IN
	J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
	Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
	Divers' Supply Shop	5208 University Dr	Macon	GA
	Techniques	52 Dolphin Drive	Redwood City	CA
	Perry Scuba	3443 James Ave	Hapeville	GA

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see “Creating a customized grid” on page 15-16.
- Creation of persistent field components for the dataset displayed in the grid. For more information about creating persistent field components using the Fields editor, see Chapter 19, “Working with field components.”
- The dataset’s *ObjectView* property setting for grids displaying ADT and array fields. See “Displaying ADT and array fields” on page 15-21.

A grid control has a *Columns* property that is itself a wrapper on a *TDBGridColumns* object. *TDBGridColumns* is a collection of *TColumn* objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the *Columns* property of the grid to access the properties, events, and methods of *TDBGridColumns* at runtime.

### Using a grid control in its default state

The *State* property of the grid’s *Columns* property indicates whether persistent column objects exist for the grid. *Columns.State* is a runtime-only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid’s dataset, or, if there are no persistent field components, by a default set of display characteristics.

When the grid’s *Columns.State* property is *csDefault*, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with

a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Paradox table at one moment, then switch to display the results of an SQL query when the grid's *DataSource* property changes or when the *DataSet* property of the data source itself is changed.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the *Width* property of a column changes the *DisplayWidth* property of the field associated with that column. Changes made to column properties that are not based on field properties, such as *Font*, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

**Note** Changing a grid's *Columns.State* property to *csDefault* at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.

## Creating a customized grid

---

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

## Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a

column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property. If you modify the *DisplayLabel* property, the column title reflects that change immediately. If you then assign a string to the column title's caption, the title caption becomes independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of the record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

**Note** Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's *FieldCount* property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's *SelectedField* property is NULL and the *SelectedIndex* property is -1.

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (...) in a cell that can be clicked upon to launch special data viewers or dialogs related to the current cell.

## Creating persistent columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the *State* property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control,

- 1 Select the grid component in the form.
- 2 Invoke the Columns editor by double clicking on the grid's *Columns* property in the Object Inspector.

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis. To create persistent columns for all fields:

- 1 Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.
- 2 If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.
- 3 Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually:

- 1 Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).
- 2 To associate a field with this new column, set the *FieldName* property in the Object Inspector.
- 3 To set the title for the new column, expand the *Title* property in the Object Inspector and set its *Caption* property.
- 4 Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

## Deleting persistent columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display. To remove a persistent column from a grid,

- 1 Double-click the grid to display the Columns editor.
- 2 Select the field to remove in the Columns list box.
- 3 Click Delete (you can also use the context menu or *Del* key, to remove a column).

**Note** If you delete all the columns from a grid, the *Columns.State* property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
```

## Arranging the order of persistent columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column,

- 1 Select the column in the Columns list box.
- 2 Drag it to a new location in the list box.

You can also change the column order by clicking on the column title of the actual grid and dragging the column to a new position, just as you can at runtime.

**Note** Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

**Important** You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's *OnColumnMoved* event fires after a column has been moved.

## Setting column properties at design time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component (called the *default source*) such as a grid or an associated field component.

To set a column's properties, select the column in The Columns editor and set its properties in the Object Inspector. The following table summarizes key column properties you can set.

**Table 15.2** Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: <i>TField.Alignment</i> .
ButtonStyle	<i>cbsAuto</i> : (default) Displays a drop-down list if the associated field is a lookup field, or if the column's <i>PickList</i> property contains data. <i>cbsEllipsis</i> : Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's <i>OnEditButtonClick</i> event. <i>cbsNone</i> : The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. Default source: <i>TDBGrid.Color</i> . (For text foreground color, see the Font property.)



**Table 15.2** Column properties (continued)

Property	Purpose
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.
Expanded	Specifies whether the column is expanded. Only applies to columns representing ADT or array fields.
FieldName	Specifies the field name associated with this column. This can be blank.
ReadOnly	<i>True</i> : The data in the column cannot be edited by the user. <i>False</i> : (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default source: <i>TField.DisplayWidth</i> .
Font	Specifies the font type, size, and color used to draw text in the column. Default source: <i>TDBGrid.Font</i> .
PickList	Contains a list of values to display in a drop-down list in the column.
Title	Sets properties for the title of the selected column.

The following table summarizes the options you can specify for the *Title* property.

**Table 15.3** Expanded TColumn Title properties

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default source: <i>TField.DisplayLabel</i> .
Color	Specifies the background color used to draw the column title cell. Default source: <i>TDBGrid.FixedColor</i> .
Font	Specifies the font type, size, and color used to draw text in the column title. Default source: <i>TDBGrid.TitleFont</i> .

## Defining a lookup list column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control. To specify that the column acts like a combo box, set the column's *ButtonStyle* property to *cbsAuto*. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field in the dataset. For information about creating lookup fields, see “Defining a lookup field” on page 19-8. Once the lookup field is defined, set the column's *FieldName* to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the *PickList* property for the column in the Object Inspector. This brings up the String List editor, where you can enter the values that populate the pick list for the column.

By default, the drop-down list displays 7 values. You can change the length of this list by setting the *DropDownRows* property.

**Note** To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

## Putting a button in a column

A column can display an ellipsis button (...) to the right of the normal cell editor. *Ctrl+Enter* or a mouse click fires the grid's *OnEditButtonClick* event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column:

- 1 Select the column in the *Columns* list box.
- 2 Set *ButtonStyle* to *cbsEllipsis*.
- 3 Write an *OnEditButtonClick* event handler.

## Restoring default values to a column

At runtime you can test a column's *AssignedValues* property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's *RestoreDefaults* method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1.Columns.RestoreDefaults;
```

## Displaying ADT and array fields

---

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" the field so that each of the simpler types that make up the field appears as a separate field in the dataset. When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

To display composite fields as if they were flattened out, set the dataset's *ObjectView* property to *False*. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.

- It can display composite fields in a single column, reflecting the fact that they are a single field. When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expanded* property of the column:
  - When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
  - When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to *True*. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse

Figure 15.2 shows a grid with an ADT field and an array field. The dataset's *ObjectView* property is set to *False* so that each child field has a column.

**Figure 15.2** TDBGrid control with *ObjectView* set to *False*

ID_KEY	ADT child fields			Array child fields		
	NAME_ADT.FIRST	NAME_ADT.MIDDLE	NAME_ADT.LAST	TELNDS_ARRAY[0]	TELNDS_ARRAY[1]	TELNDS_ARR[2]
1	Stephan		Wright	415-908-9875	902-786-1245	
2	Whitney	N	Long			
3	Chris	T	Scanlan	234-232-1343		

Figure 15.3 and 15.4 show the grid with an ADT field and an array field. Figure 15.3 shows the fields collapsed. In this state they cannot be edited. Figure 15.4 shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.

**Figure 15.3** TDBGrid control with Expanded set to False

ID_KEY	NAME_ADT	TELNOS_ARRAY
1	(Stephan, Wright)	(415-908-9875, 902-786-1245, ...)
2	(Whitney, N, Long)	(... 510-454-7234, ...)
3	(Chris, T, Scanlan)	(234-232-1343, ...)

**Figure 15.4** TDBGrid control with Expanded set to True

ID_KEY	ADT child field columns			Array child field columns			
	FIRST	MIDDLE	LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARRAY[2]	TELNO
1	Stephan		Wright	415-908-9875	902-786-1245		
2	Whitney	N	Long				510-454-7234
3	Chris	T	Scanlan	234-232-1343			

The following table lists the properties that affect the way ADT and array fields appear in a *TDBGrid*:

**Table 15.4** Properties that affect the way composite fields appear

Property	Object	Purpose
Expandable	TColumn	Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only)
Expanded	TColumn	Specifies whether the column is expanded.
MaxTitleRows	TDBGrid	Specifies the maximum number of title rows that can appear in the grid
ObjectView	TDataSet	Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed.
ParentColumn	TColumn	Refers to the TColumn object that owns the child field's column.

**Note** In addition to ADT and array fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as "(DataSet)" or "(Reference)", respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field's value. For reference fields, this grid contains a single row that displays the record from another dataset.

## Setting grid options

You can use the grid *Options* property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of

Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

**Table 15.5** Expanded TDBGrid Options properties

Option	Purpose
dgEditing	<i>True</i> : (Default). Enables editing, inserting, and deleting records in the grid. <i>False</i> : Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	<i>True</i> : When a field is selected, it is in Edit state. <i>False</i> : (Default). A field is not automatically in Edit state when selected.
dgTitles	<i>True</i> : (Default). Displays field names across the top of the grid. <i>False</i> : Field name display is turned off.
dgIndicator	<i>True</i> : (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam. <i>False</i> : The indicator column is turned off.
dgColumnResize	<i>True</i> : (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying <i>TField</i> component. <i>False</i> : Columns cannot be resized in the grid.
dgColLines	<i>True</i> : (Default). Displays vertical dividing lines between columns. <i>False</i> : Does not display dividing lines between columns.
dgRowLines	<i>True</i> : (Default). Displays horizontal dividing lines between records. <i>False</i> : Does not display dividing lines between records.
dgTabs	<i>True</i> : (Default). Enables tabbing between fields in records. <i>False</i> : Tabbing exits the grid control.
dgRowSelect	<i>True</i> : The selection bar spans the entire width of the grid. <i>False</i> : (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	<i>True</i> : (Default). The selection bar in the grid is always visible, even if another control has focus. <i>False</i> : The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	<i>True</i> : (Default). Prompt for confirmation to delete records ( <i>Ctrl+Del</i> ). <i>False</i> : Delete records without confirmation.
dgCancelOnExit	<i>True</i> : (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records. <i>False</i> : Permits pending inserts.
dgMultiSelect	<i>True</i> : Allows user to select noncontiguous rows in the grid using <i>Ctrl+Shift</i> or <i>Shift+ arrow</i> keys. <i>False</i> : (Default). Does not allow user to multi-select rows.

## Editing in the grid

---

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The *CanModify* property of the *Dataset* is *True*.
- The *ReadOnly* property of grid is *False*.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus is changed to another control on a form, the grid does not post changes until another the cursor for the dataset is moved to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, the grid raises an exception, and does not modify the record.

**Note** If your application caches updates, posting record changes only adds them to an internal cache. They are not posted back to the underlying database table until your application applies the updates.

You can cancel all edits for a record by pressing *Esc* in any field before moving to another record.

## Controlling grid drawing

---

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat* properties of the field component associated with the column.

You can augment the default grid display logic with code in a grid's *OnDrawColumnCell* event. If the grid's *DefaultDrawing* property is *True*, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special graphics in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to *False* and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* inside your *OnDrawColumnCell* event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

## Responding to user actions at runtime

---

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and

records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the Object Inspector.

**Table 15.6** Grid control events

Event	Purpose
OnClick	Occurs when a user clicks on a cell in the grid.
OnColEnter	Occurs when a user moves into a column on the grid.
OnColExit	Occurs when a user leaves a column on the grid.
OnColumnMoved	Occurs when the user moves a column to a new location.
OnDblClick	Occurs when a user double clicks in the grid.
OnDragDrop	Occurs when a user drags and drops in the grid.
OnDragOver	Occurs when a user drags over the grid.
OnDrawColumnCell	Occurs when application needs to draw individual cells.
OnDrawDataCell	(obsolete) Occurs when application needs to draw individual cells if <i>State</i> is <i>csDefault</i> .
OnEditButtonClick	Occurs when the user clicks on an ellipsis button in a column.
OnEndDrag	Occurs when a user stops dragging on the grid.
OnEnter	Occurs when the grid gets focus.
OnExit	Occurs when the grid loses focus.
OnKeyDown	Occurs when a user presses any key or key combination on the keyboard when in the grid.
OnKeyPress	Occurs when a user presses a single alphanumeric key on the keyboard when in the grid.
OnKeyUp	Occurs when a user releases a key when in the grid.
OnStartDrag	Occurs when a user starts dragging on the grid.
OnTitleClick	Occurs when a user clicks the title for a column.

There are many uses for these events. For example, you might write a handler for the *OnDblClick* event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the *SelectedField* property to determine to current row and column.

## Creating a grid that contains other data-aware controls

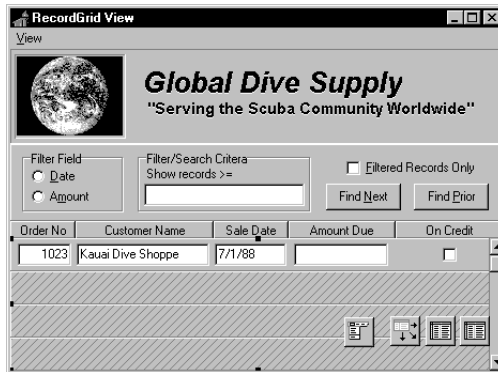
A *TDBCtrlGrid* control displays multiple fields in multiple records in a tabular grid format. Each cell in a grid displays multiple fields from a single row. To use a database control grid:

- 1 Place a database control grid on a form.
- 2 Set the grid's *DataSource* property to the name of a data source.

- 3 Place individual data controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.
- 4 Set the *DataField* property for each data control to the name of a field. The data source for these data controls is already set to the data source of the database control grid.
- 5 Arrange the controls within the cell as desired.

When you compile and run an application containing a database control grid, the arrangement of data controls you set in the design cell at runtime is replicated in each cell of the grid. Each cell displays a different record in a dataset.

**Figure 15.5** TDBCtrGrid at design time



The following table summarizes some of the unique properties for database control grids that you can set at design time:

**Table 15.7** Selected database control grid properties

Property	Purpose
AllowDelete	<i>True</i> (default): Permits record deletion. <i>False</i> : Prevents record deletion.
AllowInsert	<i>True</i> (default): Permits record insertion. <i>False</i> : Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.
Orientation	<i>goVertical</i> (default): Display records from top to bottom. <i>goHorizontal</i> : Displays records from left to right.
PanelHeight	Sets the height for an individual panel. Default = 72.
PanelWidth	Sets the width for an individual panel. Default = 200.
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	<i>True</i> (default): Displays a focus rectangle around the current record's panel at runtime. <i>False</i> : Does not display a focus rectangle.

For more information about database control grid properties and methods, see the online *VCL Reference*.

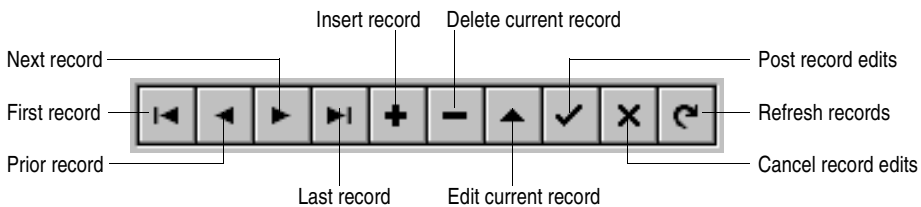


## Navigating and manipulating records

*TDBNavigator* provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

Figure 15.6 shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator enables you to hide or show a subset of these buttons dynamically.

**Figure 15.6** Buttons on the TDBNavigator control



The following table describes the buttons on the navigator.

**Table 15.8** TDBNavigator buttons

Button	Purpose
First	Calls the dataset's <i>First</i> method to set the current record to the first record.
Prior	Calls the dataset's <i>Prior</i> method to set the current record to the previous record.
Next	Calls the dataset's <i>Next</i> method to set the current record to the next record.
Last	Calls the dataset's <i>Last</i> method to set the current record to the last record.
Insert	Calls the dataset's <i>Insert</i> method to insert a new record before the current record, and set the dataset in Insert state.
Delete	Deletes the current record. If the <i>ConfirmDelete</i> property is <i>True</i> it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

### Choosing navigator buttons to display

When you first place a *TDBNavigator* on a form at design time, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the

*First*, *Next*, and *Refresh* buttons are meaningful. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

## Hiding and showing navigator buttons at design time

The *VisibleButtons* property in the Object Inspector is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The list of buttons that can be turned on or off appears in the Object Inspector below the *VisibleButtons* property. The + sign changes to a – (minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to *True*, the button appears in the *TDBNavigator*. If *False*, the button is removed from the navigator at design time and runtime.

**Note** As button values are set to *False*, they are removed from the *TDBNavigator* on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

## Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The *VisibleButtons* property controls which buttons are displayed in the navigator. Here's one way you might code the *OnEnter* event handler:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    begin
      DBNavigatorAll.DataSource := CustomerCompany.DataSource;
      DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
    end
  else
    begin
      DBNavigatorAll.DataSource := OrderNum.DataSource;
      DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
        nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
    end;
end;
```

## Displaying fly-over help

---

To display fly-over help for each navigator button at runtime, set the navigator *ShowHint* property to *True*. When *ShowHint* is *True*, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default.

The *Hints* property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

## Using a single navigator for multiple datasets

---

As with other data-aware controls, a navigator's *DataSource* property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two edit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the edit control connected to *CustomersSource*, the navigator should also use *CustomersSource*, and when the user enters the edit control connected to *OrdersSource*, the navigator should switch to *OrdersSource* as well. You can code an *OnEnter* event handler for one of the edit controls, and then share that event with the other edit control. For example:

```

procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;

```

# Using decision support components

The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see “About crosstabs” on page 16-2.

## Overview

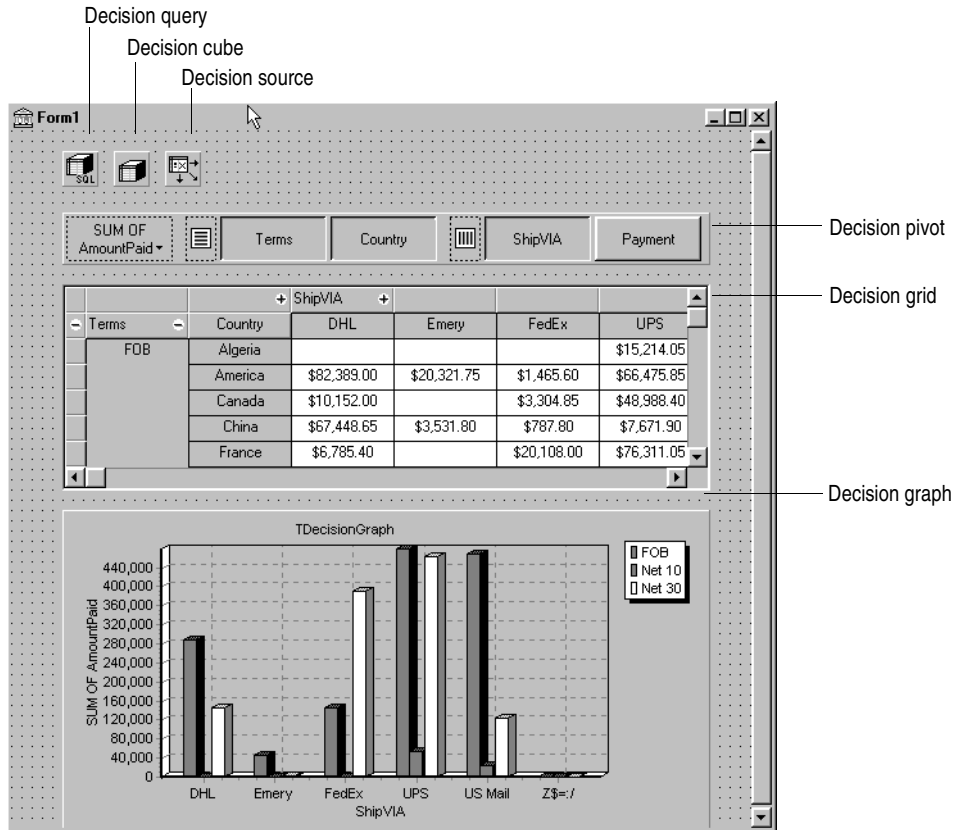
---

The decision support components appear on the Decision Cube page of the component palette:

- The decision cube, *TDecisionCube*, is a multidimensional data store.
- The decision source, *TDecisionSource*, defines the current pivot state of a decision grid or a decision graph.
- The decision query, *TDecisionQuery*, is a specialized form of *TQuery* used to define the data in a decision cube.
- The decision pivot, *TDecisionPivot*, lets you open or close decision cube dimensions, or fields, by pressing buttons.
- The decision grid, *TDecisionGrid*, displays single- and multidimensional data in table form.
- The decision graph, *TDecisionGraph*, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified.

Figure 16.1 shows all the decision support components placed on a form at design time.

**Figure 16.1** Decision support components at design time



## About crosstabs

Cross-tabulations, or crosstabs, are a way of presenting subsets of data so that relationships and trends are more visible. Table fields become the dimensions of the crosstab while field values define categories and summaries within a dimension.

You can use the decision support components to set up crosstabs in forms. *TDecisionGrid* shows data in a table, while *TDecisionGraph* charts it graphically. *TDecisionPivot* has buttons that make it easier to display and hide dimensions and move them between columns and rows.

Crosstabs can be one-dimensional or multidimensional.

### One-dimensional crosstabs

One-dimensional crosstabs show a summary row (or column) for the categories of a single dimension. For example, if *Payment* is the chosen column dimension and

Amount Paid is the summary category, the crosstab in Figure 16.2 shows the amount paid using each method.

**Figure 16.2** One-dimensional crosstab

+	Payment					
+	AmEx	Cash	Check	COD	Credit	MC
	\$134,753.40	\$164,003.65	\$270,492.15	\$33,776.55	\$1,332,430.25	\$250,163.25

## Multidimensional crosstabs

Multidimensional crosstabs use additional dimensions for the rows and/or columns. For example, a two-dimensional crosstab could show amounts paid by payment method for each country.

A three-dimensional crosstab could show amounts paid by payment method and terms by country, as shown in Figure 16.3.

**Figure 16.3** Three-dimensional crosstab

	+	Country	Check	COD	Credit	MC
-	Terms					
	FOB	Algeria	\$2,577.85		\$1,400.00	\$13,814.05
		America			\$356,816.20	\$20,881.35
		Canada			\$24,485.00	\$3,304.85
		China	\$61,936.90		\$6,641.55	

## Guidelines for using decision support components

The decision support components listed on page 16-1 can be used together to present multidimensional data as tables and graphs. More than one grid or graph can be attached to each dataset. More than one instance of *TDecisionPivot* can be used to display the data from different perspectives at runtime.

To create a form with tables and graphs of multidimensional data, follow these steps:

- 1 Create a form.
- 2 Add these components to the form and use the Object Inspector to bind them as indicated:
  - A dataset, usually *TDecisionQuery* (for details, see “Creating decision datasets with the Decision Query editor” on page 16-6) or *TQuery*

- A decision cube, *TDecisionCube*, bound to the dataset by setting its *DataSet* property to the dataset's name
  - A decision source, *TDecisionSource*, bound to the decision cube by setting its *DecisionCube* property to the decision cube's name
- 3 Add a decision pivot, *TDecisionPivot*, and bind it to the decision source with the Object Inspector by setting its *DecisionSource* property to the appropriate decision source name. The decision pivot is optional but useful; it lets the form developer and end users change the dimensions displayed in decision grids or decision graphs by pushing buttons.

In its default orientation, horizontal, buttons on the left side of the decision pivot apply to fields on the left side of the decision grid (rows); buttons on the right side apply to fields at the top of the decision grid (columns).

You can determine where the decision pivot's buttons appear by setting its *GroupLayout* property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default). For more information on decision pivot properties, see "Using decision pivots" on page 16-9.
  - 4 Add one or more decision grids and graphs, bound to the decision source. For details, see "Creating and using decision grids" on page 16-10 and "Creating and using decision graphs" on page 16-13.
  - 5 Use the Decision Query editor or *SQL* property of *TDecisionQuery* (or *TQuery*) to specify the tables, fields, and summaries to display in the grid or graph. The last field of the SQL SELECT should be the summary field. The other fields in the SELECT must be GROUP BY fields. For instructions, see "Creating decision datasets with the Decision Query editor" on page 16-6.
  - 6 Set the *Active* property of the decision query (or alternate dataset component) to *True*.
  - 7 Use the decision grid and graph to show and chart different data dimensions. See "Using decision grids" on page 16-11 and "Using decision graphs" on page 16-13 for instructions and suggestions.

For an illustration of all decision support components on a form, see Figure 16.1 on page 16-2.

## Using datasets with decision support components

---

The only decision support component that binds directly to a dataset is the decision cube, *TDecisionCube*. *TDecisionCube* expects to receive data with groups and summaries defined by an SQL statement of an acceptable format. The GROUP BY phrase must contain the same non-summarized fields (and in the same order) as the SELECT phrase, and summary fields must be identified.

The decision query component, *TDecisionQuery*, is a specialized form of *TQuery*. You can use *TDecisionQuery* to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes (*TDecisionCube*). You can also use an ordinary *TQuery* or other BDE-enabled dataset

as a dataset for *TDecisionCube*, but the correct setup of the dataset and *TDecisionCube* are then the responsibility of the designer.

To work correctly with the decision cube, all projected fields in the dataset must either be dimensions or summaries. The summaries should be additive values (like sum or count), and should represent totals for each combination of dimension values. For maximum ease of setup, sums should be named "Sum..." in the dataset while counts should be named "Count...".

The Decision Cube can pivot, subtotal, and drill-in correctly only for summaries whose cells are additive. (SUM and COUNT are additive, while AVERAGE, MAX, and MIN are not.) Build pivoting crosstab displays only for grids that contain only additive aggregators. If you are using non-additive aggregators, use a static decision grid that does not pivot, drill, or subtotal.

Since averages can be calculated using SUM divided by COUNT, a pivoting average is added automatically when SUM and COUNT dimensions for a field are included in the dataset. Use this type of average in preference to an average calculated using an AVERAGE statement.

Averages can also be calculated using COUNT(\*). To use COUNT(\*) to calculate averages, include a "COUNT(\*) COUNTALL" selector in the query. If you use COUNT(\*) to calculate averages, the single aggregator can be used for all fields. Use COUNT(\*) only in cases where none of the fields being summarized include blank values, or where a COUNT aggregator is not available for every field.

## Creating decision datasets with TQuery or TTable

---

If you use an ordinary *TQuery* component as a decision dataset, you must manually set up the SQL statement, taking care to supply a GROUP BY phrase which contains the same fields (and in the same order) as the SELECT phrase.

The SQL should look similar to this:

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
       ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

The ordering of the SELECT fields should match the ordering of the GROUP BY fields.

With *TTable*, you must supply information to the decision cube about which of the fields in the query are grouping fields, and which are summaries. To do this, Fill in the Dimension Type for each field in the *DimensionMap* of the Decision Cube. You must indicate whether each field is a dimension or a summary, and if a summary, you must provide the summary type. Since pivoting averages depend on SUM/COUNT calculations, you must also provide the base field name to allow the decision cube to match pairs of SUM and COUNT aggregators.



## Creating decision datasets with the Decision Query editor

---

All data used by the decision support components passes through the decision cube, which accepts a specially formatted dataset most easily produced by an SQL query. See “Using datasets with decision support components” on page 16-4 for more information.

While both *TTable* and *TQuery* can be used as decision datasets, it is easier to use *TDecisionQuery*; the Decision Query editor supplied with it can be used to specify tables, fields, and summaries to appear in the decision cube and will help you set up the SELECT and GROUP BY portions of the SQL correctly.

To use the Decision Query editor:

- 1 Select the decision query component on the form, then right-click and choose Decision Query editor. The Decision Query editor dialog box appears.
- 2 Choose the database to use.
- 3 For single-table queries, click the Select Table button.  
For more complex queries involving multi-table joins, click the Query Builder button to display the SQL Builder or type the SQL statement into the edit box on the SQL tab page.
- 4 Return to the Decision Query editor dialog box.
- 5 In the Decision Query editor dialog box, select fields in the Available Fields list box and assign them to be either Dimensions or Summaries by clicking the appropriate right arrow button. As you add fields to the Summaries list, select from the menu displayed the type of summary to use: sum, count, or average.
- 6 By default, all fields and summaries defined in the *SQL* property of the decision query appear in the Active Dimensions and Active Summaries list boxes. To remove a dimension or summary, select it in the list and click the left arrow beside the list, or double-click the item to remove. To add it back, select it in the Available Fields list box and click the appropriate right arrow.

Once you define the contents of the decision cube, you can further manipulate dimension display with its *DimensionMap* property and the buttons of *TDecisionPivot*. For more information, see the next section, “Using decision cubes,” “Using decision sources” on page 16-9, and “Using decision pivots” on page 16-9.

**Note** When you use the Decision Query editor, the query is initially handled in ANSI-92 SQL syntax, then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays only ANSI standard SQL. The dialect translation is automatically assigned to the *TDecisionQuery*'s SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather than the SQL property.

## Using decision cubes

---

The decision cube component, *TDecisionCube*, is a multidimensional data store that fetches its data from a dataset (typically a specially structured SQL statement entered through *TDecisionQuery* or *TQuery*). The data is stored in a form that makes its easy to pivot (that is, change the way in which the data is organized and summarized) without needing to run the query a second time.

### Decision cube properties and events

---

The *DimensionMap* properties of *TDecisionCube* not only control which dimensions and summaries appear but also let you set date ranges and specify the maximum number of dimensions the decision cube may support. You can also indicate whether or not to display data during design. You can display names, (categories) values, subtotals, or data. Display of data at design time can be time consuming, depending on the data source.

When you click the ellipsis next to *DimensionMap* in the Object Inspector, the Decision Cube editor dialog box appears. You can use its pages and controls to set the *DimensionMap* properties.

The *OnRefresh* event fires whenever the decision cube cache is rebuilt. Developers can access the new dimension map and change it at that time to free up memory, change the maximum summaries or dimensions, and so on. *OnRefresh* is also useful if users access the Decision Cube editor; application code can respond to user changes at that time.

### Using the Decision Cube editor

---

You can use the Decision Cube editor to set the *DimensionMap* properties of decision cubes. You can display the Decision Cube editor through the Object Inspector, as described in the previous section, or by right-clicking a decision cube on a form at design time and choosing Decision Cube editor.

The Decision Cube Editor dialog box has two tabs:

- **Dimension Settings**, used to activate or disable available dimensions, rename and reformat dimensions, put dimensions in a permanently drilled state, and set date ranges to display.
- **Memory Control**, used to set the maximum number of dimensions and summaries that can be active at one time, to display information about memory usage, and to determine the names and data that appear at design time.

## Viewing and changing dimension settings

To view the dimension settings, display the Decision Cube editor and click the Dimension Settings tab. Then, select a dimension or summary in the Available Fields list. Its information appears in the boxes on the right side of the editor:

- To change the dimension or summary name that appears in the decision pivot, decision grid, or decision graph, enter a new name in the Display Name edit box.
- To determine whether the selected field is a dimension or summary, read the text in the Type edit box. If the dataset is a *TTable* component, you can use Type to specify whether the selected field is a dimension or summary.
- To disable or activate the selected dimension or summary, change the setting in the Active Type drop-down list box: Active, As Needed, or Inactive. Disabling a dimension or setting it to As Needed saves memory.
- To change the format of that dimension or summary, enter a format string in the Format edit box.
- To display that dimension or summary by Year, Quarter, or Month, change the setting in the Binning drop-down list box. Note that you can choose Set in the Binning list box to put the selected dimension or summary in a permanently “drilled down” state. This can be useful for saving memory when a dimension has many values. For more information, see “Decision support components and memory control” on page 16-19.
- To determine the starting value for ranges, or the drill-down value for a “Set” dimension, first choose the appropriate Grouping value in the Grouping drop-down, and then enter the starting range value or permanent drill-down value in the Initial Value drop-down list.

## Setting the maximum available dimensions and summaries

To determine the maximum number of dimensions and summaries available for decision pivots, decision grids, and decision graphs bound to the selected decision cube, display the Decision Cube editor and click the Memory Control tab. Use the edit controls to adjust the current settings, if necessary. These settings help to control the amount of memory required by the decision cube. For more information, see “Decision support components and memory control” on page 16-19.

## Viewing and changing design options

To determine how much information appears at design time, display the Decision Cube editor and click the Memory Control tab. Then, check the setting that indicates which names and data to display. Display of data or field names at design time can cause performance delays in some cases because of the time needed to fetch the data.

## Using decision sources

---

The decision source component, *TDecisionSource*, defines the current pivot state of decision grids or decision graphs. Any two objects which use the same decision source also share pivot states.

### Properties and events

---

The following are some special properties and events that control the appearance and behavior of decision sources:

- The *ControlType* property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections).
- The *SparseCols* and *SparseRows* properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed.
- *TDecisionSource* has the following events:
  - *OnLayoutChange* occurs when the user performs pivots or drill-downs that reorganize the data.
  - *OnNewDimensions* occurs when the data is completely altered, such as when the summary or dimension fields are altered.
  - *OnSummaryChange* occurs when the current summary is changed.
  - *OnStateChange* occurs when the Decision Cube activates or deactivates.
  - *OnBeforePivot* occurs when changes are committed but not yet reflected in the user interface. Developers have an opportunity to make changes, for example, in capacity or pivot state, before application users see the result of their previous action.
  - *OnAfterPivot* fires after a change in pivot state. Developers can capture information at that time.

## Using decision pivots

---

The decision pivot component, *TDecisionPivot*, lets you open or close decision cube dimensions, or fields, by pressing buttons. When a row or column is opened by pressing a *TDecisionPivot* button, the corresponding dimension appears on the *TDecisionGrid* or *TDecisionGraph* component. When a dimension is closed, its detailed data doesn't appear; it collapses into the totals of other dimensions. A dimension may also be in a "drilled" state, where only the summaries for a particular value of the dimension field appear.

You can also use the decision pivot to reorganize dimensions displayed in the decision grid and decision graph. Just drag a button to the row or column area or reorder buttons within the same area.

For illustrations of decision pivots at design time, see Figures 16.1, 16.2, and 16.3.

## Decision pivot properties

---

The following are some special properties that control the appearance and behavior of decision pivots:

- The first properties listed for *TDecisionPivot* define its overall behavior and appearance. You might want to set *ButtonAutoSize* to *False* for *TDecisionPivot* to keep buttons from expanding and contracting as you adjust the size of the component.
- The *Groups* property of *TDecisionPivot* defines which dimension buttons appear. You can display the row, column, and summary selection button groups in any combination. Note that if you want more flexibility over the placement of these groups, you can place one *TDecisionPivot* on your form which contains only rows in one location, and a second which contains only columns in another location.
- Typically, *TDecisionPivot* is added above *TDecisionGrid*. In its default orientation, horizontal, buttons on the left side of *TDecisionPivot* apply to fields on the left side of *TDecisionGrid* (rows); buttons on the right side apply to fields at the top of *TDecisionGrid* (columns).
- You can determine where *TDecisionPivot*'s buttons appear by setting its *GroupLayout* property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default, described in the previous paragraph).

## Creating and using decision grids

---

Decision grid components, *TDecisionGrid*, present cross-tabulated data in table form. These tables are also called crosstabs, described on page 16-2. Figure 16.1 on page 16-2 shows a decision grid on a form at design time.

### Creating decision grids

---

To create a form with one or more tables of cross-tabulated data,

- 1 Follow steps 1–3 listed under “Guidelines for using decision support components” on page 16-3.
- 2 Add one or more decision grid components (*TDecisionGrid*) and bind them to the decision source, *TDecisionSource*, with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.

- 3** Continue with steps 5–7 listed under “Guidelines for using decision support components.”

For a description of what appears in the decision grid and how to use it, see “Using decision grids” on page 16-11.

To add a graph to the form, follow the instructions in “Creating decision graphs” on page 16-13.

## Using decision grids

---

The decision grid component, *TDecisionGrid*, displays data from decision cubes (*TDecisionCube*) bound to decision sources (*TDecisionSource*).

By default, the grid appears with dimension fields at its left side and/or top, depending on the grouping instructions defined in the dataset. Categories, one for each data value, appear under each field. You can

- Open and close dimensions
- Reorganize, or pivot, rows and columns
- Drill down for detail
- Limit dimension selection to a single dimension for each axis

For more information about special properties and events of the decision grid, see “Decision grid properties” on page 16-12.

### Opening and closing decision grid fields

If a plus sign (+) appears in a dimension or summary field, one or more fields to its right are closed (hidden). You can open additional fields and categories by clicking the sign. A minus sign (-) indicates a fully opened (expanded) field. When you click the sign, the field closes. This outlining feature can be disabled; see “Decision grid properties” on page 16-12 for details.

### Reorganizing rows and columns in decision grids

You can drag row and column headings to new locations within the same axis or to the other axis. In this way, you can reorganize the grid and see the data from new perspectives as the data groupings change. This pivoting feature can be disabled; see “Decision grid properties” on page 16-12 for details.

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See “Using decision pivots” on page 16-9 for instructions.

### Drilling down for detail in decision grids

You can drill down to see more detail in a dimension.

For example, if you right-click a category label (row heading) for a dimension with others collapsed beneath it, you can choose to drill down and only see data for that category. When a dimension is drilled, you do not see the category labels for that dimension displayed on the grid, since only the records for a single category value

are being displayed. If you have a decision pivot on the form, it displays category values and lets you change to other values if you want.

To drill down into a dimension,

- Right-click a category label and choose Drill In To This Value, or
- Right-click a pivot button and choose Drilled In.

To make the complete dimension active again,

- Right-click the corresponding pivot button, or right-click the decision grid in the upper-left corner and select the dimension.

## Limiting dimension selection in decision grids

You can change the *ControlType* property of the decision source to determine whether more than one dimension can be selected for each axis of the grid. For more information, see “Using decision sources” on page 16-9.

## Decision grid properties

---

The decision grid component, *TDecisionGrid*, displays data from the *TDecisionCube* component bound to *TDecisionSource*. By default, data appears in a grid with category fields on the left side and top of the grid.

The following are some special properties that control the appearance and behavior of decision grids:

- *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the Object Inspector, then select a dimension. Its properties then appear in the Object Inspector: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines the color of category labels, *FieldName* displays the name of the active dimension, *Format* can hold any standard format for that data type, and *Subtotals* indicates whether to display subtotals for that dimension. With summary fields, these same properties are used to changed the appearance of the data that appears in the summary area of the grid. When you’re through setting dimension properties, either click a component in the form or choose a component in the drop-down list box at the top of the Object Inspector.
- The *Options* property of *TDecisionGrid* lets you control display of grid lines (*cgGridLines = True*), enabling of outline features (collapse and expansion of dimensions with + and - indicators; *cgOutliner = True*), and enabling of drag-and-drop pivoting (*cgPivotable = True*).
- The *OnDecisionDrawCell* event of *TDecisionGrid* gives you a chance to change the appearance of each cell as it is drawn. The event passes the *String*, *Font*, and *Color* of the current cell as reference parameters. You are free to alter those parameters to achieve effects such as special colors for negative values. In addition to the *DrawState* which is passed by *TCustomGrid*, the event passes *TDecisionDrawState*, which can be used to determine what type of cell is being drawn. Further information about the cell can be fetched using the *Cells*, *CellValueArray*, or *CellDrawState* functions.

- The *OnDecisionExamineCell* event of *TDecisionGrid* lets you hook the right-click-on-event to data cells, and is intended to allow a program to display information (such as detail records) about that particular data cell. When the user right-clicks a data cell, the event is supplied with all the information which is used to compose the data value, including the currently active summary value and a *ValueArray* of all the dimension values which were used to create the summary value.

## Creating and using decision graphs

---

Decision graph components, *TDecisionGraph*, present cross-tabulated data in graphic form. Each decision graph shows the value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see page 16-2. For illustrations of decision graphs at design time, see Figure 16.1 on page 16-2 and Figure 16.4 on page 16-14.

### Creating decision graphs

---

To create a form with one or more decision graphs,

- 1 Follow steps 1–3 listed under “Guidelines for using decision support components” on page 16-3.
- 2 Add one or more decision graph components (*TDecisionGraph*) and bind them to the decision source, *TDecisionSource*, with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.
- 3 Continue with steps 5–7 listed under “Guidelines for using decision support components.”
- 4 Finally, right-click the graph and choose Edit Chart to modify the appearance of the graph series. You can set template properties for each graph dimension, then set individual series properties to override these defaults. For details, see “Customizing decision graphs” on page 16-15.

For a description of what appears in the decision graph and how to use it, see the next section, “Using decision graphs.”

To add a decision grid—or crosstab table—to the form, follow the instructions in “Creating and using decision grids” on page 16-10.

### Using decision graphs

---

The decision graph component, *TDecisionGraph*, displays fields from the decision source (*TDecisionSource*) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (*TDecisionPivot*).



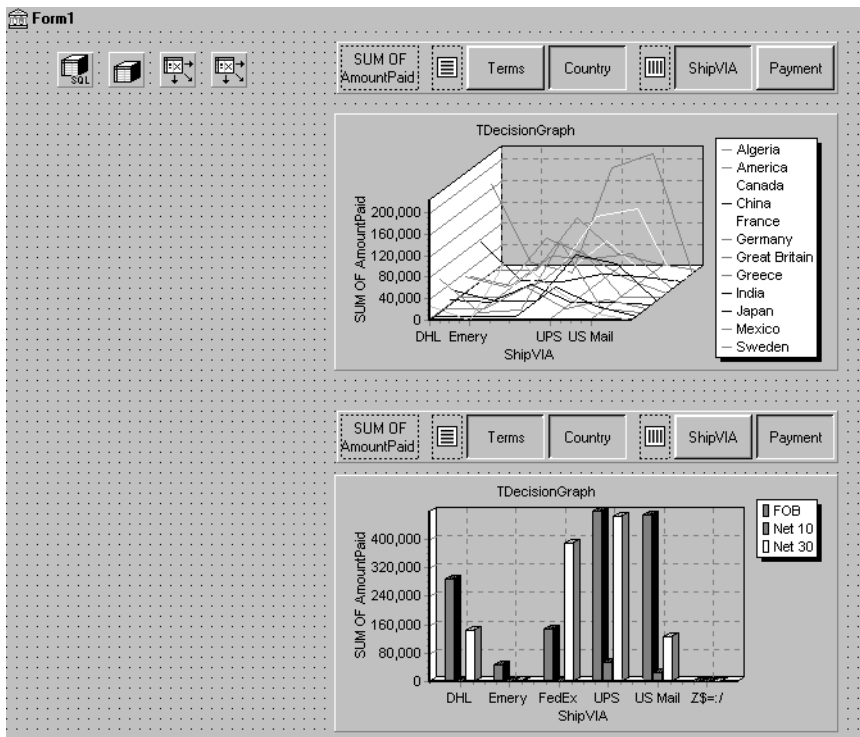
Graphed data comes from a specially formatted dataset such as *TDecisionQuery*. For an overview of how the decision support components handle and arrange this data, see page 16-1.

By default, the first row dimension appears as the x-axis and the first column dimension appears as the y-axis.

You can use decision graphs instead of or in addition to decision grids, which present cross-tabulated data in tabular form. Decision grids and decision graphs that are bound to the same decision source present the same data dimensions. To show different summary data for the same dimensions, you can bind more than one decision graph to the same decision source. To show different dimensions, bind decision graphs to different decision sources.

For example, in Figure 16.4 the first decision pivot and graph are bound to the first decision source and the second decision pivot and graph are bound to the second. So, each graph can show different dimensions.

**Figure 16.4** Decision graphs bound to different decision sources



For more information about what appears in a decision graph, see the next section, “The decision graph display.”

To create a decision graph, see the previous section, “Creating decision graphs.”

For a discussion of decision graph properties and how to change the appearance and behavior of decision graphs, see “Customizing decision graphs” on page 16-15.

## The decision graph display

---

By default, the decision graph plots summary values for categories in the first active row field (along the y-axis) against values in the first active column field (along the x-axis). Each graphed category appears as a separate series.

If only one dimension is selected—for example, by clicking only one *TDecisionPivot* button—only one series is graphed.

If you used a decision pivot, you can push its buttons to determine which decision cube fields (dimensions) are graphed. To exchange graph axes, drag the decision pivot dimension buttons from one side of the separator space to the other. If you have a one-dimensional graph with all buttons on one side of the separator space, you can use the Row or Column icon as a drop target for adding buttons to the other side of the separator and making the graph multidimensional.

If you only want one column and one row to be active at a time, you can set the *ControlType* property for *TDecisionSource* to *xtRadio*. Then, there can be only one active field at a time for each decision cube axis, and the decision pivot's functionality will correspond to the graph's behavior. *xtRadioEx* works the same as *xtRadio*, but does not allow the state where all row or all columns dimensions are closed.

When you have both a decision grid and graph connected to the same *TDecisionSource*, you'll probably want to set *ControlType* back to *xtCheck* to correspond to the more flexible behavior of *TDecisionGrid*.

## Customizing decision graphs

---

The decision graph component, *TDecisionGraph*, displays fields from the decision source (*TDecisionSource*) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (*TDecisionPivot*). You can change the type, colors, marker types for line graphs, and many other properties of decision graphs.

To customize a graph,

- 1 Right-click it and choose Edit Chart. The Chart Editing dialog box appears.
- 2 Use the Chart page of the Chart Editing dialog box to view a list of visible series, select the series definition to use when two or more are available for the same series, change graph types for a template or series, and set overall graph properties.

The Series list on the Chart page shows all decision cube dimensions (preceded by Template:) and currently visible categories. Each category, or series, is a separate object. You can:

- Add or delete series derived from existing decision-graph series. Derived series can provide annotations for existing series or represent values calculated from other series.
- Change the default graph type, and change the title of templates and series.

For a description of the other Chart page tabs, search for the following topic in online Help: “Chart page (Chart Editing dialog box).”

- 3 Use the Series page to establish dimension templates, then customize properties for each individual graph series.

By default, all series are graphed as bar graphs and up to 16 default colors are assigned. You can edit the template type and properties to create a new default. Then, as you pivot the decision source to different states, the template is used to dynamically create the series for each new state. For template details, see “Setting decision graph template defaults” on page 16-16.

To customize individual series, follow the instructions in “Customizing decision graph series” on page 16-17.

For a description of each Series page tab, search for the following topic in online Help: “Series page (Chart Editing dialog box).”

## Setting decision graph template defaults

Decision graphs display the values from two dimensions of the decision cube: one dimension is displayed as an axis of the graph, and the other is used to create a set of series. The template for that dimension provides default properties for those series (such as whether the series are bar, line, area, and so on). As users pivot from one state to another, any required series for the dimension are created using the series type and other defaults specified in the template.

A separate template is provided for cases where users pivot to a state where only one dimension is active. A one-dimensional state is often represented with a pie chart, so a separate template is provided for this case.

You can

- Change the default graph type.
- Change other graph template properties.
- View and set overall graph properties.

## Changing the default decision graph type

To change the default graph type,

- 1 Select a template in the Series list on the Chart page of the Chart Editing dialog box.
- 2 Click the Change button.
- 3 Select a new type and close the Gallery dialog box.

## Changing other decision graph template properties

To change color or other properties of a template,

- 1 Select the Series page at the top of the Chart Editing dialog box.
- 2 Choose a template in the drop-down list at the top of the page.
- 3 Choose the appropriate property tab and select settings.

## Viewing overall decision graph properties

To view and set decision graph properties other than type and series,

- 1 Select the Chart page at the top of the Chart Editing dialog box.
- 2 Choose the appropriate property tab and select settings.

## Customizing decision graph series

The templates supply many defaults for each decision cube dimension, such as graph type and how series are displayed. Other defaults, such as series color, are defined by *TDecisionGraph*. If you want you can override the defaults for each series.

The templates are intended for use when you want the program to create the series for categories as they are needed, and discard them when they are no longer needed. If you want, you can set up custom series for specific category values. To do this, pivot the graph so its current display has a series for the category you want to customize. When the series is displayed on the graph, you can use the Chart editor to

- Change the graph type.
- Change other series properties.
- Save specific graph series that you have customized.

To define series templates and set overall graph defaults, see “Setting decision graph template defaults” on page 16-16.

## Changing the series graph type

By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See “Changing the default decision graph type” on page 16-16 for instructions.

To change the graph type for a single series,

- 1 Select a series in the Series list on the Chart page of the Chart editor.
- 2 Click the Change button.
- 3 Select a new type and close the Gallery dialog box.
- 4 Check the Save Series check box.

## Changing other decision graph series properties

To change color or other properties of a decision graph series,

- 1 Select the Series page at the top of the Chart Editing dialog box.
- 2 Choose a series in the drop-down list at the top of the page.
- 3 Choose the appropriate property tab and select settings.
- 4 Check the Save Series check box.

## Saving decision graph series settings

By default, only settings for templates are saved at design time. Changes made to specific series are only saved if the Save box is checked for that series in the Chart Editing dialog box.

Saving series can be memory intensive, so if you don't need to save them you can uncheck the Save box.

## Decision support components at runtime

---

At runtime, users can perform many operations by left-clicking, right-clicking, and dragging visible decision support components. These operations, discussed earlier in this chapter, are summarized below.

### Decision pivots at runtime

---

Users can:

- Left-click the summary button at the left end of the decision pivot to display a list of available summaries. They can use this list to change the summary data displayed in decision grids and decision graphs.
- Right-click a dimension button and choose to:
  - Move it from the row area to the column area or the reverse.
  - Drill In to display detail data.
- Left-click a dimension button following the Drill In command and choose:
  - Open Dimension to move back to the top level of that dimension.
  - All Values to toggle between displaying just summaries and summaries plus all other values in decision grids.
  - From a list of available categories for that dimension, a category to drill into for detail values.
- Left-click a dimension button to open or close that dimension.
- Drag and drop dimension buttons from the row area to the column area and the reverse; they can drop them next to existing buttons in that area or onto the row or column icon.

### Decision grids at runtime

---

Users can:

- Right-click within the decision grid and choose to:
  - Toggle subtotals on and off for individual data groups, for all values of a dimension, or for the whole grid.
  - Display the Decision Cube editor, described on page 16-7.
  - Toggle dimensions and summaries open and closed.

- Click + and – within the row and column headings to open and close dimensions.
- Drag and drop dimensions from rows to columns and the reverse.

## Decision graphs at runtime

---

Users can drag from side to side or up and down in the graph grid area to scroll through off-screen categories and values.

## Decision support components and memory control

---

When a dimension or summary is loaded into the decision cube, it takes up memory. Adding a new summary increases memory consumption linearly: that is, a decision cube with two summaries uses twice as much memory as the same cube with only one summary, a decision cube with three summaries uses three times as much memory as the same cube with one summary, and so on. Memory consumption for dimensions increases more quickly. Adding a dimension with 10 values increases memory consumption by a factor of 10. Adding a dimension with 100 values increases memory consumption 100 times. Thus adding dimensions to a decision cube can have a dramatic effect on memory use, and can quickly lead to performance problems. This effect is especially pronounced when adding dimensions that have many values.

The decision support components have a number of settings to help you control how and when memory is used. For more information on the properties and techniques mentioned here, look up *TDecisionCube* in the online Help.

### Setting maximum dimensions, summaries, and cells

---

The decision cube's *MaxDimensions* and *MaxSummaries* properties can be used with the *CubeDim.ActiveFlag* property to control how many dimensions and summaries can be loaded at a time. You can set the maximum values on the Cube Capacity page of the Decision Cube editor to place some overall control on how many dimensions or summaries can be brought into memory at the same time.

Limiting the number of dimensions or summaries provides a rough limit on the amount of memory used by the decision cube. However, it does not distinguish between dimensions with many values and those with only a few. For greater control of the absolute memory demands of the decision cube, you can also limit the number of cells in the cube. Set the maximum number of cells on the Cube Capacity page of the Decision Cube editor.

### Setting dimension state

---

The *ActiveFlag* property controls which dimensions get loaded. You can set this property on the Dimension Settings tab of the Decision Cube editor using the Activity Type control. When this control is set to *Active*, the dimension is loaded

unconditionally, and will always take up space. Note that the number of dimensions in this state must always be less than *MaxDimensions*, and the number of summaries set to *Active* must be less than *MaxSummaries*. You should set a dimension or summary to *Active* only when it is critical that it be available at all times. An *Active* setting decreases the ability of the cube to manage the available memory.

When *ActiveFlag* is set to *AsNeeded*, a dimension or summary is loaded only if it can be loaded without exceeding the *MaxDimensions*, *MaxSummaries*, or *MaxCells* limit. The decision cube will swap dimensions and summaries that are marked *AsNeeded* in and out of memory to keep within the limits imposed by *MaxCells*, *MaxDimensions*, and *MaxSummaries*. Thus, a dimension or summary may not be loaded in memory if it is not currently being used. Setting dimensions that are not used frequently to *AsNeeded* results in better loading and pivoting performance, although there will be a time delay to access dimensions which are not currently loaded.

## Using paged dimensions

---

When Binning is set to Set on the Dimension Settings tab of the Decision cube editor and Start Value is not NULL, the dimension is said to be “paged,” or “permanently drilled down.” You can access data for just a single value of that dimension at a time, although you can programmatically access a series of values sequentially. Such a dimension may not be pivoted or opened.

It is extremely memory intensive to include dimensional data for dimensions that have very large numbers of values. By making such dimensions paged, you can display summary information for one value at a time. Information is usually easier to read when displayed this way, and memory consumption is much easier to manage.

# Connecting to databases

Most dataset components can connect directly to a database server. Once connected, the dataset communicates with the server automatically. When you open the dataset, it populates itself with data from the server, and when you post records, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection.

Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table lists these data access mechanisms and the associated connection components:

**Table 17.1** Database connection components

Data access mechanism	Connection component
The Borland Database Engine (BDE).	TDatabase
ActiveX Data Objects (ADO).	TADOConnection
dbExpress.	TSQLConnection
InterBase Express.	TIBDatabase

**Note** For a discussion of some pros and cons of each of these mechanisms, see “Using databases” on page 14-1.

The connection component provides all the information necessary to establish a database connection. This information is different for each type of connection component:

- For information about describing a BDE-based connection, see “Identifying the database” on page 20-13.
- For information about describing an ADO-based connection, see “Connecting to a data store using TADOConnection” on page 21-3
- For information about describing a dbExpress connection, see “Setting up TSQLConnection” on page 22-3



- For information about describing an InterBase Express connection, see the online help for *TIBDatabase*.

Although each type of dataset uses a different connection component, they are all descendants of *TCustomConnection*. They all perform many of the same tasks and surface many of the same properties, methods, and events. This chapter discusses many of these common tasks.

## Using implicit connections

---

No matter what data access mechanism you are using, you can always create the connection component explicitly and use it to manage the connection to and communication with a database server. For BDE-enabled and ADO-based datasets, you also have the option of describing the database connection through properties of the dataset and letting the dataset generate an implicit connection. For BDE-enabled datasets, you specify an implicit connection using the *DatabaseName* property. For ADO-based datasets, you use the *ConnectionString* property.

When using an implicit connection, you do not need to explicitly create a connection component. This can simplify your application development, and the default connection you specify can cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own connection components to tune each database connection to your application's needs. Explicit connection components give you greater control. For example, you need to access the connection component to perform the following tasks:

- Customize database server login support. (Implicit connections display a default login dialog to prompt the user for a user name and password.)
- Control transactions and specify transaction isolation levels.
- Execute SQL commands on the server without using a dataset.
- Perform actions on all open datasets that are connected to the same database.

In addition, if you have multiple datasets that all use the same server, it can be easier to use an connection component, so that you only have to specify the server to use in one place. That way, if you later change the server, you do not need to update several dataset components: only the connection component.

## Controlling connections

---

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server. Each type of connection component surfaces a different set of properties to let you identify the server. In general, however, they all provide a way for you to name the server you want and supply a set of connection parameters that control how the connection is formed. Connection parameters vary from server to server. They can include information such as user name and password, the maximum size of BLOB fields, SQL roles, and so on.

Once you have identified the desired server and any connection parameters, you can use the connection component to explicitly open or close a connection. The connection component generates events when it opens or closes a connection that you can use to customize the response of your application to changes in the database connection.

## Connecting to a database server

---

There are two ways to connect to a database server using a connection component:

- Call the *Open* method.
- Set the *Connected* property to *True*.

Calling the *Open* method sets *Connected* to *True*.

**Note** When a connection component is not connected to a server and an application attempts to open one of its associated datasets, the dataset automatically calls the connection component's *Open* method.

When you set *Connected* to *True*, the connection component first generates a *BeforeConnect* event, where you can perform any initialization. For example, you can use this event to alter connection parameters.

After the *BeforeConnect* event, the connection component may display a default login dialog, depending on how you choose to control server login. It then passes the user name and password to the driver, opening a connection.

Once the connection is open, the connection component generates an *AfterConnect* event, where you can perform any tasks that require an open connection.

**Note** Some connection components generate additional events as well when establishing a connection.

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, the connection component drops the connection. Some connection components surface a *KeepConnection* property that allows the connection to remain open even if all the datasets that use it are closed. If *KeepConnection* is *True*, the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting *KeepConnection* to *True* reduces network traffic and speeds up the application. If *KeepConnection* is *False*, the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

## Disconnecting from a database server

---

There are two ways to disconnect a server using a connection component:

- Set the *Connected* property to *False*.
- Call the *Close* method.

Calling *Close* sets *Connected* to *False*.

When *Connected* is set to *False*, the connection component generates a *BeforeDisconnect* event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, the connection component closes all open datasets and disconnects from the server.

Finally, the connection component generates an *AfterDisconnect* event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

**Note** Calling *Close* or setting *Connected* to *False* disconnects from a database server even if the connection component has a *KeepConnection* property that is *True*.

## Controlling server login

---

Most remote database servers include security features to prohibit unauthorized access. Usually, the server requires a user name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

- Let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt* property of the connection component to *True* (the default) and add *DBLogDlg* to the *uses* clause of the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.
- Supply the login information before the login attempt. Each type of connection component uses a different mechanism for specifying the user name and password:
  - For BDE, dbExpress, and InterBase express datasets, the user name and password connection parameters can be accessed through the *Params* property. (For BDE datasets, the parameter values can also be associated with a BDE alias, while for dbExpress datasets, they can also be associated with a connection name).
  - For ADO datasets, the user name and password can be included in the *ConnectionString* property (or provided as parameters to the *Open* method).

If you specify the user name and password before the server requests them, be sure to set the *LoginPrompt* to *False*, so that the default login dialog does not appear. For example, the following code sets the user name and password on a SQL connection component in the *BeforeConnect* event handler, decrypting an encrypted password that is associated with the current connection name:

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
    with Sender as TSQLConnection do
    begin
```

```

if LoginPrompt = False then
begin
  Params.Values['User_Name'] := 'SYSDBA';
  Params.Values['Password'] := Decrypt(Params.Values['Password']);
end;
end;
end;

```

Note that setting the user name and password at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. This still leaves them easy to find, compromising server security.

- Provide your own custom handling for the login event. The connection component generates an event when it needs the user name and password.
  - For *TDatabase*, *TSQLConnection*, and *TIBDatabase*, this is an *OnLogin* event. The event handler has two parameters, the connection component, and a local copy of the user name and password parameters in a string list. (*TSQLConnection* includes the database parameter as well). You must set the *LoginPrompt* property to *True* for this event to occur. Having a *LoginPrompt* value of *False* and assigning a handler for the *OnLogin* event creates a situation where it is impossible to log in to the database because the default dialog does not appear and the *OnLogin* event handler never executes.
  - For *TADOConnection*, the event is an *OnWillConnect* event. The event handler has five parameters, the connection component and four parameters that return values to influence the connection (including two for user name and password). This event always occurs, regardless of the value of *LoginPrompt*.

Write an event handler for the event in which you set the login parameters. Here is an example where the values for the USER NAME and PASSWORD parameters are provided from a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*)

```

procedure TForm1.Database1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;

```

As with the other methods of providing login parameters, when writing an *OnLogin* or *OnWillConnect* event handler, avoid hard coding the password in your application code. It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

## Managing transactions

---

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database. Most databases provide their own transaction management model, although some have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, connection components provide a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (The BDE also provides limited transaction support for local tables with no server transaction support. When not using the BDE, trying to start transactions on a database that does not support them causes connection components to raise an exception.)

**Warning** When a dataset provider component applies updates, it implicitly generates transactions for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

## Starting a transaction

---

When you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or (in the case of overlapping transactions) until another transaction is started. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

While the transaction is in process, your view of the data in database tables is determined by your transaction isolation level. For information about transaction isolation levels, see “Specifying the transaction isolation level” on page 17-9.

For *TADOConnection*, start a transaction by calling the *BeginTrans* method:

```
Level := ADOConnection1.BeginTrans;
```

*BeginTrans* returns the level of nesting for the transaction that started. A nested transaction is one that is nested within another, parent, transaction. After the server starts the transaction, the ADO connection receives an *OnBeginTransComplete* event.

For *TDatabase*, use the *StartTransaction* method instead. *TDataBase* does not support nested or overlapped transactions: If you call a *TDatabase* component’s *StartTransaction* method while another transaction is underway, it raises an

exception. To avoid calling *StartTransaction*, you can check the *InTransaction* property:

```
if not Database1.InTransaction then
    Database1.StartTransaction;
```

*TSQLConnection* also uses the *StartTransaction* method, but it uses a version that gives you a lot more control. Specifically, *StartTransaction* takes a transaction descriptor, which lets you manage multiple simultaneous transactions and specify the transaction isolation level on a per-transaction basis. (For more information on transaction levels, see “Specifying the transaction isolation level” on page 17-9.) In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested (as they can be when using ADO) or they can be overlapped.

```
var
    TD: TTransactionDesc;
begin
    TD.TransactionID := 1;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
```

By default, with overlapped transactions, the first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. If you are using *TSQLConnection* with an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its *TransactionLevel* property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

**Note** Unlike *TADOConnection*, *TSQLConnection* and *TDatabase* do not receive any events when the transactions starts.

InterBase express offers you even more control than *TSQLConnection* by using a separate transaction component rather than starting transactions using the connection component. You can, however, use *TIBDatabase* to start a default transaction:

```
if not IBDatabase1.DefaultTransaction.InTransaction then
    IBDatabase1.DefaultTransaction.StartTransaction;
```

You can have overlapped transactions by using two separate transaction components. Each transaction component has a set of parameters that let you configure the transaction. These let you specify the transaction isolation level, as well as other properties of the transaction.

## Ending a transaction

---

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

## Ending a successful transaction

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by committing the transaction. For *TDatabase*, you commit a transaction using the *Commit* method:

```
MyOracleConnection.Commit;
```

For *TSQLConnection*, you also use the *Commit* method, but you must specify which transaction you are committing by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Commit(TD);
```

For *TIBDatabase*, you commit a transaction object using its *Commit* method:

```
IBDatabase1.DefaultTransaction.Commit;
```

For *TADODConnection*, you commit a transaction using the *CommitTrans* method:

```
ADODConnection1.CommitTrans;
```

**Note** It is possible for a nested transaction to be committed, only to have the changes rolled back later if the parent transaction is rolled back.

After the transaction is successfully committed, an ADO connection component receives an *OnCommitTransComplete* event. Other connection components do not receive any similar events.

A call to commit the current transaction is usually attempted in a **try...except** statement. That way, if the transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

## Ending an unsuccessful transaction

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. Discarding these changes is called rolling back the transaction.

For *TDatabase*, you roll back a transaction by calling the *Rollback* method:

```
MyOracleConnection.Rollback;
```

For *TSQLConnection*, you also use the *Rollback* method, but you must specify which transaction you are rolling back by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Rollback(TD);
```

For *TIBDatabase*, you roll back a transaction object by calling its *Rollback* method:

```
IBDatabase1.DefaultTransaction.Rollback;
```

For *TADODConnection*, you roll back a transaction by calling the *RollbackTrans* method:

```
ADODConnection1.RollbackTrans;
```

After the transaction is successfully rolled back, an ADO connection component receives an *OnRollbackTransComplete* event. Other connection components do not receive any similar events.

A call to roll back the current transaction usually occurs in

- Exception handling code when you can't recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

## Specifying the transaction isolation level

---

Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction “sees” of other transactions' changes to a table.

Each server type supports a different set of possible transaction isolation levels. There are three possible transaction isolation levels:

- *DirtyRead*: When the isolation level is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle, Sybase, MS-SQL, and InterBase).
- *ReadCommitted*: When the isolation level is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading. This level is available for all transactions except local transactions managed by the BDE.
- *RepeatableRead*: When the isolation level is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions. This level is not available on some servers, such as Sybase and MS-SQL and is unavailable on local transactions managed by the BDE.

In addition, *TSQLConnection* lets you specify database-specific custom isolation levels. Custom isolation levels are defined by the *dbExpress* driver. See your driver documentation for details.

**Note** For a detailed description of how each isolation level is implemented, see your server documentation.

*TDatabase* and *TADOConnection* let you specify the transaction isolation level by setting the *TransIsolation* property. When you set *TransIsolation* to a value that is not supported by the database server, you get the next highest level of isolation (if available). If there is no higher level available, the connection component raises an exception when you try to start a transaction.

When using *TSQLConnection*, transaction isolation level is controlled by the *IsolationLevel* field of the transaction descriptor.

When using InterBase express, transaction isolation level is controlled by a transaction parameter.



## Sending commands to the server

---

All database connection components except *TIBDatabase* let you execute SQL statements on the associated server by calling the *Execute* method. Although *Execute* can return a cursor when the statement is a SELECT statement, this use is not recommended. The preferred method for executing statements that return data is to use a dataset.

The *Execute* method is very convenient for executing simple SQL statements that do not return any records. Such statements include Data Definition Language (DDL) statements, which operate on or create a database's metadata, such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Some Data Manipulation Language (DML) SQL statements also do not return a result set. The DML statements that perform an action on data but do not return a result set are: INSERT, DELETE, and UPDATE.

The syntax for the *Execute* method varies with the connection type:

- For *TDatabase*, *Execute* takes four parameters: a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, a boolean that indicates whether the statement should be cached because you will call it again, and a pointer to a BDE cursor that can be returned (It is recommended that you pass nil).
- For *TADOConnection*, there are two versions of *Execute*. The first takes a *WideString* that specifies the SQL statement and a second parameter that specifies a set of options that control whether the statement is executed asynchronously and whether it returns any records. This first syntax returns an interface for the returned records. The second syntax takes a *WideString* that specifies the SQL statement, a second parameter that returns the number of records affected when the statement executes, and a third that specifies options such as whether the statement executes asynchronously. Note that neither syntax provides for passing parameters.
- For *TSQLConnection*, *Execute* takes three parameters: a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, and a pointer that can receive a *TCustomSQLDataSet* that is created to return records.

**Note** *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters on a *TSQLConnection* component:

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
```

```

    '( ' +
    ' CustNo INTEGER, ' +
    ' Company CHAR(40), ' +
    ' State CHAR(2), ' +
    ' PRIMARY KEY (CustNo) ' +
    ')';
SQLConnection1.Execute(SQLstmt, nil, nil);
end;

```

To use parameters, you must create a *TParams* object. For each parameter value, use the *TParams.CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which uses *TDatabase* to execute an INSERT statement. The INSERT statement has a single parameter named *:StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of "CA" for that parameter.

```

procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
    SQLstmt: String;
    stmtParams: TParams;
begin
    stmtParams := TParams.Create;
    try
        Database1.Connected := True;
        stmtParams.CreateParam(ftString, 'StateParam', ptInput);
        stmtParams.ParamByName('StateParam').AsString := 'CA';
        SQLstmt := 'INSERT INTO "Custom.db" '+
            '(CustNo, Company, State) ' +
            'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
        Database1.Execute(SQLstmt, stmtParams, False, nil);
    finally
        stmtParams.Free;
    end;
end;

```

If the SQL statement includes a parameter but you do not supply a *TParam* object to provide its value, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

## Working with associated datasets

---

All database connection components maintain a list of all datasets that use them to connect to a database. A connection component uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific connection component to connect to a particular database.

## Closing all datasets without disconnecting from the server

---

The connection component automatically closes all datasets when you close its connection. There may be times, however, when you want to close all datasets without disconnecting from the database server.

To close all open datasets without disconnecting from a server, you can use the *CloseDataSets* method.

For *TADOConnection* and *TIBDatabase*, calling *CloseDataSets* always leaves the connection open. For *TDatabase* and *TSQLConnection*, you must also set the *KeepConnection* property to *True*.

## Iterating through the associated datasets

---

To perform any actions (other than closing them all) on all the datasets that use a connection component, use the *DataSets* and *DataSetCount* properties. *DataSets* is an indexed array of all datasets that are linked to the connection component. For all connection components except *TADOConnection*, this list includes only the active datasets. *TADOConnection* lists the inactive datasets as well. *DataSetCount* is the number of datasets in this array.

**Note** When you use a specialized client dataset to cache updates (as opposed to the generic client dataset, *TClientDataSet*), the *DataSets* property lists the internal dataset owned by the client dataset, not the client dataset itself.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

```
var
  I: Integer;
begin
  with MyDBConnection do
    begin
      for I := 0 to DataSetCount - 1 do
        DataSets[I].DisableControls;
      end;
    end;
end;
```

**Note** *TADOConnection* supports command objects as well as datasets. You can iterate through these much like you iterate through the datasets, by using the *Commands* and *CommandCount* properties.

## Obtaining metadata

---

All database connection components can retrieve lists of metadata on the database server, although they vary in the types of metadata they retrieve. The methods that retrieve metadata fill a string list with the names of various entities available on the server. You can then use this information, for example, to let your users dynamically select a table at runtime.

You can use a *TADOConnection* component to retrieve metadata about the tables and stored procedures available on the ADO data store. You can then use this information, for example, to let your users dynamically select a table or stored procedure at runtime.

## Listing available tables

---

The *GetTableNames* method copies a list of table names to an already-existing string list object. This can be used, for example, to fill a list box with table names that the user can then use to choose a table to open. The following line fills a listbox with the names of all tables on the database:

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

*GetTableNames* has two parameters: the string list to fill with table names, and a boolean that indicates whether the list should include system tables, or ordinary tables. Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.

**Note** For most database connection components, *GetTableNames* returns a list of all available non-system tables when the second parameter is *False*. For *TSQLConnection*, however, you have more control over what type is added to the list when you are not fetching only the names of system tables. When using *TSQLConnection*, the types of names added to the list are controlled by the *TableScope* property. *TableScope* indicates whether the list should contain any or all of the following: ordinary tables, system tables, synonyms, and views.

## Listing the fields in a table

---

The *GetFieldNames* method fills an existing string list with the names of all fields (columns) in a specified table. *GetFieldNames* takes two parameters, the name of the table for which you want to list the fields, and an existing string list to be filled with field names:

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

## Listing available stored procedures

---

To get a listing of all of the stored procedures contained in the database, use the *GetProcedureNames* method. This method takes a single parameter: an already-existing string list to fill:

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

**Note** *GetProcedureNames* is only available for *TADOConnection* and *TSQLConnection*.

## Listing available indexes

---

To get a listing of all indexes defined for a specific table, use the *GetIndexNames* method. This method takes two parameters: the table whose indexes you want, and an already-existing string list to fill:

```
SqlConnection1.GetIndexNames('Employee', ListBox1.Items);
```

**Note** *GetIndexNames* is only available for *TSQLConnection*, although most table-type datasets have an equivalent method.

## Listing stored procedure parameters

---

To get a list of all parameters defined for a specific stored procedure, use the *GetProcedureParams* method. *GetProcedureParams* fills a *TList* object with pointers to parameter description records, where each record describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on.

*GetProcedureParams* takes two parameters: the name of the stored procedure, and an already-existing *TList* object to fill:

```
SqlConnection1.GetProcedureParams('GetInterestRate', List1);
```

You can convert the parameter descriptions that are added to the list into the more familiar *TParams* object by calling the global *LoadParamListItems* procedure. Because *GetProcedureParams* dynamically allocates the individual records, your application must free them when it is finished with the information. The global *FreeProcParams* routine can do this for you.

**Note** *GetProcedureParams* is only available for *TSQLConnection*.

## Understanding datasets

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure.

All dataset objects that you use in your database applications descend from *TDataSet*, and they inherit data fields, properties, events, and methods from this class. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you use in your database applications. You need to understand this shared functionality to use any dataset object.

*TDataSet* is a virtualized dataset, meaning that many of its properties and methods are **virtual** or **abstract**. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of the built-in *TDataSet* descendants and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its **abstract** methods.

*TDataSet* defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For information about *TField* components, see Chapter 19, “Working with field components.”

This chapter describes how to use the common database functionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* descendants implement them. In particular, unidirectional datasets implement only a limited subset.

## Using TDataSet descendants

---

*TDataSet* has several immediate descendants, each of which corresponds to a different data access mechanism. You do not work directly with any of these descendants. Rather, each descendant introduces the properties and methods for using a particular data access mechanism. These properties and methods are then exposed by descendant classes that are adapted to different types of server data. The immediate descendants of *TDataSet* include

- *TBDEDataSet*, which uses the Borland Database Engine (BDE) to communicate with the database server. The *TBDEDataSet* descendants you use are *TTable*, *TQuery*, *TStoredProc*, and *TNestedTable*. The unique features of BDE-enabled datasets are described in Chapter 20, “Using the Borland Database Engine”.
- *TCustomADODataset*, which uses ActiveX Data Objects (ADO) to communicate with an OLEDB data store. The *TCustomADODataset* descendants you use are *TADODataset*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. The unique features of ADO-based datasets are described in Chapter 21, “Working with ADO components”.
- *TCustomSQLDataSet*, which uses dbExpress to communicate with a database server. The *TCustomSQLDataSet* descendants you use are *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*. The unique features of dbExpress datasets are described in Chapter 22, “Using unidirectional datasets”.
- *TIBCustomDataSet*, which communicates directly with an InterBase database server. The *TIBCustomDataSet* descendants you use are *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TIBStoredProc*.
- *TCustomClientDataSet*, which represents the data from another dataset component or the data from a dedicated file on disk. The *TCustomClientDataSet* descendants you use are *TClientDataSet*, which can connect to an external (source) dataset, and the client datasets that are specialized to a particular data access mechanism (*TBDEClientDataSet*, *TSQLClientDataSet*, and *TIBClientDataSet*), which use an internal source dataset. The unique features of client datasets are described in Chapter 23, “Using client datasets”.

Some pros and cons of the various data access mechanisms employed by these *TDataSet* descendants are described in “Using databases” on page 14-1.

In addition to the built-in datasets, you can create your own custom *TDataSet* descendants — for example to supply data from a process other than a database server, such as a spreadsheet. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see Chapter 40, “Overview of component creation.”

Although each *TDataSet* descendant has its own unique properties and methods, some of the properties and methods introduced by descendant classes are the same as those introduced by other descendant classes that use another data access mechanism. For example, there are similarities between the “table” components (*TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*). For information about the commonalities introduced by *TDataSet* descendants, see “Types of datasets” on page 18-23.

## Determining dataset states

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset’s read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

**Table 18.1** Values for the dataset *State* property

Value	State	Meaning
<i>dsInactive</i>	Inactive	DataSet closed. Its data is unavailable.
<i>dsBrowse</i>	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
<i>dsEdit</i>	Edit	DataSet open. The current row can be modified. (not supported on unidirectional datasets)
<i>dsInsert</i>	Insert	DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets)
<i>dsSetKey</i>	SetKey	DataSet open. Enables setting of ranges and key values used for ranges and <i>GotoKey</i> operations. (not supported by all datasets)
<i>dsCalcFields</i>	CalcFields	DataSet open. Indicates that an <i>OnCalcFields</i> event is under way. Prevents changes to fields that are not calculated.
<i>dsCurValue</i>	CurValue	DataSet open. Indicates that the <i>CurValue</i> property of fields is being fetched for an event handler that responds to errors in applying cached updates.
<i>dsNewValue</i>	NewValue	DataSet open. Indicates that the <i>NewValue</i> property of fields is being fetched for an event handler that responds to errors in applying cached updates.
<i>dsOldValue</i>	OldValue	DataSet open. Indicates that the <i>OldValue</i> property of fields is being fetched for an event handler that responds to errors in applying cached updates.
<i>dsFilter</i>	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets)
<i>dsBlockRead</i>	Block Read	DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes.



**Table 18.1** Values for the dataset State property (continued)

Value	State	Meaning
<i>dsInternalCalc</i>	Internal Calc	DataSet open. An <i>OnCalcFields</i> event is underway for calculated values that are stored with the record. (client datasets only)
<i>dsOpening</i>	Opening	DataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching.

Typically, an application checks the dataset state to determine when to perform certain tasks. For example, you might check for the *dsEdit* or *dsInsert* state to ascertain whether you need to post updates.

**Note** Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see "Responding to changes mediated by the data source" on page 15-4.

## Opening and closing datasets

To read or write data in a dataset, an application must first open it. You can open a dataset in two ways,

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at runtime:

```
CustTable.Active := True;
```

- Call the *Open* method for the dataset at runtime,

```
CustQuery.Open;
```

When you open the dataset, the dataset first receives a *BeforeOpen* event, then it opens a cursor, populating itself with data, and finally, it receives an *AfterOpen* event.

The newly-opened dataset is in browse mode, which means your application can read the data and navigate through it.

You can close a dataset in two ways,

- Set the *Active* property of the dataset to *False*, either at design time in the Object Inspector, or in code at runtime,

```
CustQuery.Active := False;
```

- Call the *Close* method for the dataset at runtime,

```
CustTable.Close;
```

Just as the dataset receives *BeforeOpen* and *AfterOpen* events when you open it, it receives a *BeforeClose* and *AfterClose* event when you close it. handlers that respond to the *Close* method for a dataset. You can use these events, for example, to prompt the

user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```

procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
    if (CustTable.State in [dsEdit, dsInsert]) then begin
        case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0) of
            mrYes:    CustTable.Post;    { save the changes }
            mrNo:    CustTable.Cancel; { abandon the changes}
            mrCancel: Abort;             { abort closing the dataset }
        end;
    end;
end;

```

**Note** You may need to close a dataset when you want to change certain of its properties, such as *TableName* on a *TTable* component. When you reopen the dataset, the new property value takes effect.

## Navigating datasets

---

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

**Table 18.2** Navigational methods of datasets

Method	Moves the cursor to
<i>First</i>	The first row in a dataset.
<i>Last</i>	The last row in a dataset. (not available for unidirectional datasets)
<i>Next</i>	The next row in a dataset.
<i>Prior</i>	The previous row in a dataset. (not available for unidirectional datasets)
<i>MoveBy</i>	A specified number of rows forward or back in a dataset.

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime. For information about the navigator component, see “Navigating and manipulating records” on page 15-28.

Whenever you change the current record using one of these methods (or by other methods that navigate based on a search criterion), the dataset receives two events: *BeforeScroll* (before leaving the current record) and *AfterScroll* (after arriving at the new record). You can use these events to update your user interface (for example, to update a status bar that indicates information about the current record).

*TDataSet* also defines two boolean properties that provide useful information when iterating through the records in a dataset.

**Table 18.3** Navigational properties of datasets

Property	Description
<i>Bof</i> (Beginning-of-file)	<i>True</i> : the cursor is at the first row in the dataset. <i>False</i> : the cursor is not known to be at the first row in the dataset
<i>Eof</i> (End-of-file)	<i>True</i> : the cursor is at the last row in the dataset. <i>False</i> : the cursor is not known to be at the first row in the dataset

## Using the First and Last methods

The *First* method moves the cursor to the first row in a dataset and sets the *Bof* property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable.First;
```

The *Last* method moves the cursor to the last row in a dataset and sets the *Eof* property to *True*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable.Last;
```

**Note** The *Last* method raises an exception in unidirectional datasets.

**Tip** While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons call the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see “Navigating and manipulating records” on page 15-28.

## Using the Next and Prior methods

The *Next* method moves the cursor forward one row in the dataset and sets the *Bof* property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable.Next;
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* to *False* if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
CustTable.Prior;
```

**Note** The *Prior* method raises an exception in unidirectional datasets.

## Using the MoveBy method

---

*MoveBy* lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the *Bof* and *Eof* properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

**Note** *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

*MoveBy* returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable.MoveBy(-2);
```

**Note** If your application uses *MoveBy* in a multi-user database environment, keep in mind that datasets are fluid. A record that was five records back a moment ago may now be four, six, or even an unknown number of records back if several users are simultaneously accessing the database and changing its data.

## Using the Eof and Bof properties

---

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful when you want to iterate through all records in a dataset.

### Eof

When *Eof* is *True*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *True* when an application

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset).
- Calls *SetRange* on an empty range or dataset.

*Eof* is set to *False* in all other cases; you should assume *Eof* is *False* unless one of the conditions above is met *and* you test the property directly.

*Eof* is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call *First*) *Eof* is

*False*. To iterate through the dataset a record at a time, create a loop that steps through each record by calling *Next*, and terminates when *Eof* is *True*. *Eof* remains *False* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets Eof False }
  while not CustTable.Eof do { Cycle until Eof is True }
  begin
    { Process each record here }
    :
    CustTable.Next; { Eof False on success; Eof True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

**Tip** This example also shows how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

## Bof

When *Bof* is *True*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *True* when an application

- Opens a dataset.
- Calls a dataset's *First* method.
- Calls a dataset's *Prior* method, and the method fails (because the cursor is currently at the first row in the dataset).
- Calls *SetRange* on an empty range or dataset.

*Bof* is set to *False* in all other cases; you should assume *Bof* is *False* unless one of the conditions above is met *and* you test the property directly.

Like *Eof*, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.Bof do { Cycle until Bof is True }
  begin
    { Process each record here }
    :
  end;
```

```

    CustTable.Prior; { Bof False on success; Bof True when Prior fails on first record }
end;
finally
    CustTable.EnableControls; { Display new current row in controls }
end;

```

## Marking and returning to records

---

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. *TDataSet* introduces a bookmarking feature that consists of a *Bookmark* property and five bookmark methods.

*TDataSet* implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. *TDataSet* descendants vary in the level of support they provide for bookmarks. None of the dbExpress datasets add any support for bookmarks. ADO datasets can support bookmarks, depending on the underlying database tables. BDE datasets, InterBase express datasets, and client datasets always support bookmarks.

### The Bookmark property

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

### The GetBookmark method

To create a bookmark, you must declare a variable of type *TBookmark* in your application, then call *GetBookmark* to allocate storage for the variable and set its value to a particular location in a dataset. The *TBookmark* type is a Pointer.

### The GotoBookmark and BookmarkValid methods

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. Before calling *GotoBookmark*, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record.

### The CompareBookmarks method

You can also call *CompareBookmarks* to see if a bookmark you want to move to is different from another (or the current) bookmark. If the two bookmarks refer to the same record (or if both are *nil*), *CompareBookmarks* returns 0.

### The FreeBookmark method

*FreeBookmark* frees the memory allocated for a specified bookmark when you no longer need it. You should also call *FreeBookmark* before reusing an existing bookmark.

## A bookmarking example

The following code illustrates one use of bookmarking:

```

procedure DoSomething (const Tbl: TTable)
var
    Bookmark: TBookmark;
begin
    Bookmark := Tbl.GetBookmark; { Allocate memory and assign a value }
    Tbl.DisableControls; { Turn off display of records in data controls }
    try
        Tbl.First; { Go to first record in table }
        while not Tbl.Eof do {Iterate through each record in table }
        begin
            { Do your processing here }
            :
            Tbl.Next;
        end;
    finally
        Tbl.GotoBookmark(Bookmark);
        Tbl.EnableControls; { Turn on display of records in data controls, if necessary }
        Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
    end;
end;

```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

## Searching datasets

---

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods. These methods enable you to search on any type of columns in any dataset.

**Note** Some *TDataSet* descendants introduce an additional family of methods for searching based on an index. For information about these additional methods, see “Using Indexes to search for records” on page 18-27.

### Using Locate

---

*Locate* moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is “Professional Divers, Ltd.”:

```

var
    LocateSuccess: Boolean;
    SearchOptions: TLocateOptions;
begin

```

```

SearchOptions := [loPartialKey];
LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
end;

```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *True* if it finds a matching record, *False* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```

with CustTable do
    Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);

```

*Locate* uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

## Using Lookup

---

*Lookup* searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is “Professional Divers, Ltd.”, and returns the company name, a contact person, and a phone number for the company:

```

var
    LookupResults: Variant;
begin
    LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
        'Company;Contact; Phone');
end;

```

*Lookup* returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant. For more information about Variant arrays, see the online help.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing



multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
  with CustTable do
    LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
      'Company; Addr1; Addr2; State; Zip');
  end;
```

Like *Locate*, *Lookup* uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

## Displaying and editing a subset of data using filters

---

An application is frequently interested in only a subset of records from a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. In each case, you can use filters to restrict an application's access to a subset of all records in the dataset.

With unidirectional datasets, you can only limit the records in the dataset by using a query that restricts the records in the dataset. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's *Filter* property or coded into its *OnFilterRecord* event handler. Filter conditions are based on the values in any specified number of fields in a dataset, regardless of whether those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

**Note** Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

### Enabling and disabling filtering

---

Enabling filters on a dataset is a three-step process:

- 1 Create a filter.
- 2 Set filter options for string-based filter tests, if necessary.
- 3 Set the *Filtered* property to *True*.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to *False*.

## Creating filters

---

There are two ways to create a filter for a dataset:

- Specify simple filter conditions in the *Filter* property. *Filter* is especially useful for creating and applying filters at runtime.
- Write an *OnFilterRecord* event handler for simple or complex filter conditions. With *OnFilterRecord*, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

### Setting the Filter property

To create a filter using the *Filter* property, set the value of the property to a string that contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

You can also supply a value for *Filter* based on text supplied by the user. For example, the following statement assigns the text in from edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
```

You can, of course, create a string based on both hard-coded text and user-supplied data:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

Blank field values do not appear unless they are explicitly included in the filter:

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

**Note** After you specify a value for *Filter*, to apply the filter to the dataset, set the *Filtered* property to *True*.

Filters can compare field values to literals and to constants using the following comparison and logical operators:

**Table 18.4** Comparison and logical operators that can appear in a filter

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both <i>True</i>
NOT	Tests that the following statement is not <i>True</i>
OR	Tests that at least one of two statements is <i>True</i>
+	Adds numbers, concatenates strings, adds numbers to date/time values (only available for some drivers)
-	Subtracts numbers, subtracts dates, or subtracts a number from a date (only available for some drivers)
*	Multiplies two numbers (only available for some drivers)
/	Divides two numbers (only available for some drivers)
*	wildcard for partial comparisons ( <i>FilterOptions</i> must include <i>foPartialCompare</i> )

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

**Note** When filtering is on, user edits to a record may mean that the record no longer meets a filter’s test conditions. The next time the record is retrieved from the dataset, it may therefore “disappear.” If that happens, the next record that passes the filter condition becomes the current record.

### Writing an OnFilterRecord event handler

You can write code to filter records using the *OnFilterRecord* events generated by the dataset for each record it retrieves. This event handler implements a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your *OnFilterRecord* handler sets its *Accept* parameter to *True* to include a record, or *False* to exclude it. For

example, the following filter displays only those records with the State field set to "CA":

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
    Accept := DataSet['State'].AsString = 'CA';
end;
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter’s conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly-coded as possible to avoid adversely affecting the performance.

### Switching filter event handlers at runtime

You can code any number of *OnFilterRecord* event handlers and switch among them at runtime. For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

## Setting filter options

---

The *FilterOptions* property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

**Table 18.5** FilterOptions values

Value	Meaning
<i>foCaseInsensitive</i>	Ignore case when comparing strings.
<i>foNoPartialCompare</i>	Disable partial string matching; that is, don’t match strings that end with an asterisk (*).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

## Navigating records in a filtered dataset

---

There are four dataset methods that navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

**Table 18.6** Filtered dataset navigational methods

Method	Purpose
<i>FindFirst</i>	Move to the first record that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
<i>FindLast</i>	Move to the last record that matches the current filter criteria.
<i>FindNext</i>	Moves from the current record in the filtered dataset to the next one.
<i>FindPrior</i>	Move from the current record in the filtered dataset to the previous one.

For example, the following statement finds the first filtered record in a dataset:

```
DataSet1.FindFirst;
```

Provided that you set the *Filter* property or create an *OnFilterRecord* event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

**Note** If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return *False*. You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset and you call *FindNext*, the method returns *False*, and the current record is unchanged.

## Modifying data

---

You can use the following dataset methods to insert, update, and delete data if the read-only *CanModify* property is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write

privileges, or some other factor intervenes. (Intervening factors include the *ReadOnly* property on some datasets or the *RequestLive* property on *TQuery* components.)

**Table 18.7** Dataset methods for inserting, updating, and deleting data

Method	Description
<i>Edit</i>	Puts the dataset into <i>dsEdit</i> state if it is not already in <i>dsEdit</i> or <i>dsInsert</i> states.
<i>Append</i>	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in <i>dsInsert</i> state.
<i>Insert</i>	Posts any pending data, and puts the dataset in <i>dsInsert</i> state.
<i>Post</i>	Attempts to post the new or altered record to the database. If successful, the dataset is put in <i>dsBrowse</i> state; if unsuccessful, the dataset remains in its current state.
<i>Cancel</i>	Cancels the current operation and puts the dataset in <i>dsBrowse</i> state.
<i>Delete</i>	Deletes the current record and puts the dataset in <i>dsBrowse</i> state.

## Editing records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*.

When a dataset transitions to *dsEdit* mode, it first receives a *BeforeEdit* event. After the transition to edit mode is successfully completed, the dataset receives an *AfterEdit* event. Typically, these events are used for updating the user interface to indicate the current state of the dataset. If the dataset can't be put into edit mode for some reason, an *OnEditError* event occurs, where you can inform the user of the problem or try to correct the situation that prevented the dataset from entering edit mode.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

**Note** Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you have a navigator component on your form, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, *Edit* and *Post* are often used together. For example,

```
with CustTable do
begin
  Edit;
```

```

    FieldValues['CustNo'] := 1234;
    Post;
end;

```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record. If you are not caching updates, posting writes the change back to the database. If you are caching updates, the change is written to a temporary buffer, where it stays until the dataset's *ApplyUpdates* method is called.

## Adding new records

---

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*.

When a dataset transitions to *dsInsert* mode, it first receives a *BeforeInsert* event. After the transition to insert mode is successfully completed, the dataset receives first an *OnNewRecord* event hand then an *AfterInsert* event. You can use these events, for example, to provide initial values to newly inserted records:

```

procedure TForm1.OrdersTableNewRecord(DataSet: TDataSet);
begin
    DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;

```

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

**Note** Even if a dataset is in *dsInsert* state, adding records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

*Post* writes the new record to the database, or, if you are caching updates, *Post* writes the record to an in-memory cache. To write cached inserts and appends to the database, call the dataset's *ApplyUpdates* method.

## Inserting records

*Insert* opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when using cached updates), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at its current position.
- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

## Appending records

*Append* opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when using cached updates), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

## Deleting records

---

Use the *Delete* method to delete the current record in an active dataset. When the *Delete* method is called,

- The dataset receives a *BeforeDelete* event.
- The dataset attempts to delete the current record.
- The dataset returns to the *dsBrowse* state.
- The dataset receives an *AfterDelete* event.

If want to prevent the deletion in the *BeforeDelete* event handler, you can call the global *Abort* procedure:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```



If *Delete* fails, it generates an *OnDeleteError* event. If the *OnDeleteError* event handler can't correct the problem, the dataset remains in *dsEdit* state. If *Delete* succeeds, the dataset reverts to the *dsBrowse* state and the record that followed the deleted record becomes the current record.

If you are caching updates, the deleted record is not removed from the underlying database table until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

## Posting data

---

After you finish editing a record, you must call the *Post* method to write out your changes. The *Post* method behaves differently, depending on the dataset's state and on whether you are caching updates.

- If you are not caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to the database and returns the dataset to the *dsBrowse* state.
- If you are caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to an internal cache and returns the dataset to the *dsBrowse* state. The edits are not written to the database until you call *ApplyUpdates*.
- If the dataset is in the *dsSetKey* state, *Post* returns the dataset to the *dsBrowse* state.

Regardless of the initial state of the dataset, *Post* generates *BeforePost* and *AfterPost* events, before and after writing the current changes. You can use these events to update the user interface, or prevent the dataset from posting changes by calling the *Abort* procedure. If the call to *Post* fails, the dataset receives an *OnPostError* event, where you can inform the user of the problem or attempt to correct it.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The *Append* and *Insert* methods also implicitly post any pending data.

**Warning** The *Close* method does not call *Post* implicitly. Use the *BeforeClose* event to post any pending edits explicitly.

## Canceling changes

---

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the *Cancel* method for the dataset. A call to *Cancel* always returns a dataset to *dsBrowse* state.

If the dataset was in *dsEdit* or *dsInsert* mode when your application called *Cancel*, it receives *BeforeCancel* and *AfterCancel* events before and after the current record is restored to its original values.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

## Modifying entire records

---

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

**Table 18.8** Methods that work with entire records

Method	Description
<i>AppendRecord</i> ([array of values])	Appends a record with the specified column values at the end of a table; analogous to <i>Append</i> . Performs an implicit <i>Post</i> .
<i>InsertRecord</i> ([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to <i>Insert</i> . Performs an implicit <i>Post</i> .
<i>SetFields</i> ([array of values])	Sets the values of the corresponding fields; analogous to assigning values to <i>TFields</i> . The application must perform an explicit <i>Post</i> .

These methods take an array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, *AppendRecord* adds a record to the end of the dataset and *InsertRecord* inserts a record after the current cursor position. For indexed datasets, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

*SetFields* assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a *Post*.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, *SetFields* assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a *TTable* component called *CountryTable*

were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of “Japan”.

To update the record, an application could use the following code:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

This code assigns values to the Area and Population fields and then posts them to the database. The three NULL pointers act as place holders for the first three columns to preserve their current contents.

## Calculating fields

---

Using the Fields editor, you can define calculated fields for your datasets. When a dataset contains calculated fields, you provide the code to calculate those field’s values in an *OnCalcFields* event handler. For details on how to define calculated fields using the Fields editor, see “Defining a calculated field” on page 19-7.

The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when

- A dataset is opened.
- The dataset enters edit mode.
- A record is retrieved from the database.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a record are edited (the fourth condition above).

**Caution** *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or a linked dataset if it is part of a master-detail relationship), because this leads to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, causing another *Post*, and so on.

When *OnCalcFields* executes, a dataset enters *dsCalcFields* mode. This state prevents modifications or additions to the records except for the calculated fields the handler is designed to modify. The reason for preventing other modifications is because *OnCalcFields* uses the values in other fields to derive calculated field values. Changes to those other fields might otherwise invalidate the values assigned to calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

## Types of datasets

---

“Using TDataSet descendants” on page 18-2 classifies *TDataSet* descendants by the method they use to access their data. Another useful way to classify *TDataSet* descendants is to consider the type of server data they represent. Viewed this way, there are three basic classes of datasets:

- **Table-type datasets:** Table-type datasets represent a single table from the database server, including all of its rows and columns. Table-type datasets include *TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*.

Table-type datasets let you take advantage of indexes defined on the server. Because there is a one-to-one correspondence between database table and dataset, you can use server indexes that are defined for the database table. Indexes allow your application to sort the records in the table, speed searches and lookups, and can form the basis of a master/detail relationship. Some table-type datasets also take advantage of the one-to-one relationship between dataset and database table to let you perform table-level operations such as creating and deleting database tables.

- **Query-type datasets:** Query-type datasets represent a single SQL command, or query. Queries can represent the result set from executing a command (typically a SELECT statement), or they can execute a command that does not return any records (for example, an UPDATE statement). Query-type datasets include *TQuery*, *TADOQuery*, *TSQLQuery*, and *TIBQuery*.

To use a query-type dataset effectively, you must be familiar with SQL and your server’s SQL implementation, including limitations and extensions to the SQL-92 standard. If you are new to SQL, you may want to purchase a third party book that covers SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

- **Stored procedure-type datasets:** Stored procedure-type datasets represent a stored procedure on the database server. Stored procedure-type datasets include *TStoredProc*, *TADOStoredProc*, *TSQLStoredProc*, and *TIBStoredProc*.

A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. They typically handle frequently-repeated database-related tasks, and are especially useful for operations that act on large numbers of records or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

- Taking advantage of the server’s usually greater processing power and speed.
- Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a `SELECT` query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

**Note** You can usually use a query-type dataset to execute stored procedures because most servers provide extensions to SQL for working with stored procedures. Each server, however, uses its own syntax for this. If you choose to use a query-type dataset instead of a stored procedure-type dataset, see your server documentation for the necessary syntax.

In addition to the datasets that fall neatly into these three categories, *TDataSet* has some descendants that fit into more than one category:

- *TADODataset* and *TSQLDataset* have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are most similar to query-type datasets, although *TADODataset* lets you specify an index like a table-type dataset.
- *TClientDataSet* represents the data from another dataset. As such, it can represent a table, query, or stored procedure. *TClientDataSet* behaves most like a table-type dataset, because of its index support. However, it also has some of the features of queries and stored procedures: the management of parameters and the ability to execute without retrieving a result set.
- Some other client datasets (*TBDEClientDataSet* and *TSQLClientDataSet*) have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are like *TClientDataSet*, including parameter support, indexes, and the ability to execute without retrieving a result set.
- *TIBDataSet* can represent both queries and stored procedures. In fact, it can represent multiple queries and stored procedures simultaneously, with separate properties for each.

## Using table-type datasets

---

To use a table-type dataset,

- 1 Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
- 2 Identify the database server that contains the table you want to use. Each table-type dataset does this differently, but typically you specify a database connection component:
  - For *TTable*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.

- For *TADOTable*, specify a *TADOConnection* component using the *Connection* property.
- For *TSQTable*, specify a *TSQLConnection* component using the *SQLConnection* property.
- For *TIBTable*, specify a *TIBConnection* component using the *Database* property.

For information about using database connection components, see Chapter 17, “Connecting to databases”.

- 3 Set the *TableName* property to the name of the table in the database. You can select tables from a drop-down list if you have already identified a database connection component.
- 4 Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the dataset. The data source component is used to pass a result set from the dataset to data-aware components for display.

## Advantages of using table-type datasets

---

The main advantage of using table-type datasets is the availability of indexes. Indexes enable your application to

- Sort the records in the dataset.
- Locate records quickly.
- Limit the records that are visible.
- Establish master/detail relationships.

In addition, the one-to-one relationship between table-type datasets and database tables enables many of them to be used for

- Controlling Read/write access to tables
- Creating and deleting tables
- Emptying tables
- Synchronizing tables

## Sorting records with indexes

---

An index determines the display order of records in a table. Typically, records appear in ascending order based on a primary, or default, index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.
- A list of columns on which to sort (not available on servers that aren't SQL-based).

Indexes let you present the data from a table in different orders. On SQL-based tables, this sort order is implemented by using the index to generate an ORDER BY clause in a query that fetches the table's records. On other tables (such as Paradox and dBASE tables), the index is used by the data access mechanism to present records in the desired order.

## Obtaining information about indexes

Your application can obtain information about server-defined indexes from all table-type datasets. To obtain a list of available indexes for the dataset, call the *GetIndexNames* method. *GetIndexNames* fills a string list with valid index names. For example, the following code fills a listbox with the names of all indexes defined for the *CustomersTable* dataset:

```
CustomersTable.GetIndexNames(ListBox1.Items);
```

**Note** For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. You can still change the index back to a primary index on a Paradox table after using an alternative index, however, by setting the *IndexName* property to a blank string.

To obtain information about the fields of the current index, use the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list the field components for the columns that comprise the index.

The following code illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20] of string;
begin
  with CustomersTable do
    begin
      for I := 0 to IndexFieldCount - 1 do
        ListOfIndexFields[I] := IndexFields[I].FieldName;
      end;
    end;
end;
```

**Note** *IndexFieldCount* is not valid for a dBASE table opened on an expression index.

## Specifying an index with IndexName

Use the *IndexName* property to cause an index to be active. Once active, an index determines the order of records in the dataset. (It can also be used as the basis for a master-detail link, an index-based search, or index-based filtering.)

To activate an index, set the *IndexName* property to the name of the index. In some database systems, primary indexes do not have names. To activate one of these indexes, set *IndexName* to a blank string.

At design-time, you can select an index from a list of available indexes by clicking the property's ellipsis button in the Object Inspector. At runtime set *IndexName* using a *String* literal or variable. You can obtain a list of available indexes by calling the *GetIndexNames* method.

The following code sets the index for *CustomersTable* to *CustDescending*:

```
CustomersTable.IndexName := 'CustDescending';
```

## Creating an index with `IndexFieldNames`

If there is no defined index that implements the sort order you want, you can create a pseudo-index using the `IndexFieldNames` property.

**Note** `IndexName` and `IndexFieldNames` are mutually exclusive. Setting one property clears values set for the other.

The value of `IndexFieldNames` is a string. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for `PhoneTable` based on `LastName`, then `FirstName`:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

**Note** If you use `IndexFieldNames` on Paradox and dBASE tables, the dataset attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

## Using Indexes to search for records

You can search against any dataset using the `Locate` and `Lookup` methods of `TDataSet`. However, by explicitly using indexes, some table-type datasets can improve over the searching performance provided by the `Locate` and `Lookup` methods.

ADO datasets all support the `Seek` method, which moves to a record based on a set of field values for fields in the current index. `Seek` lets you specify where to move the cursor relative to the first or last matching record.

`TTable` and all types of client dataset support similar indexed-based searches, but use a combination of related methods. The following table summarizes the six related methods provided by `TTable` and client datasets to support index-based searches:

**Table 18.9** Index-based search methods

Method	Purpose
<code>EditKey</code>	Preserves the current contents of the search key buffer and puts the dataset into <code>dsSetKey</code> state so your application can modify existing search criteria prior to executing a search.
<code>FindKey</code>	Combines the <code>SetKey</code> and <code>GotoKey</code> methods in a single method.
<code>FindNearest</code>	Combines the <code>SetKey</code> and <code>GotoNearest</code> methods in a single method.
<code>GotoKey</code>	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
<code>GotoNearest</code>	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
<code>SetKey</code>	Clears the search key buffer and puts the table into <code>dsSetKey</code> state so your application can specify new search criteria prior to executing a search.



*GotoKey* and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

*GotoNearest* and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

## Executing a search with Goto methods

To execute a search using *Goto* methods, follow these general steps:

- 1 Specify the index to use for the search. This is the same index that sorts the records in the dataset (see “Sorting records with indexes” on page 18-25). To specify the index, use the *IndexName* or *IndexFieldNames* property.
- 2 Open the dataset.
- 3 Put the dataset in *dsSetKey* state by calling the *SetKey* method.
- 4 Specify the value(s) to search on in the *Fields* property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
- 5 Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button’s *OnClick* event, uses the *GotoKey* method to move to the first record where the first field in the index has a value that exactly matches the text in an edit box:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

*GotoNearest* is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

If a record exists with “Sm” as the first two characters of the first indexed field’s value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns *False*.

## Executing a search with Find methods

The *Find* methods do the same thing as the *Goto* methods, except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search. To execute a search using *Find* methods, follow these general steps:

- 1 Specify the index to use for the search. This is the same index that sorts the records in the dataset (see “Sorting records with indexes” on page 18-25). To specify the index, use the *IndexName* or *IndexFieldNames* property.
- 2 Open the dataset.
- 3 Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

**Note** *FindNearest* can only be used for string fields.

## Specifying the current record after a successful search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

## Searching on partial keys

If the dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the dataset’s current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For table-type datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

## Repeating or extending a search

Each time you call *SetKey* or *FindKey*, the method clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*.

For example, suppose you have already executed a search of the *Employee* table based on the *City* field of the “*CityIndex*” index. Suppose further that “*CityIndex*” includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

```
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
```

## Limiting records with ranges

---

You can temporarily view and edit a subset of data for any dataset by using filters (see “Displaying and editing a subset of data using filters” on page 18-12). Some table-type datasets support an additional way to access a subset of available records, called ranges.

Ranges only apply to *TTable* and to client datasets. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges.

### Understanding the differences between ranges and filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than “Jones” and less than “Smith”. Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to sort records, you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query-type dataset to select data. For details on specifying a query, see “Using query-type datasets” on page 18-41.

### Specifying Ranges

There are two mutually exclusive ways to specify a range:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.
- Specify both endpoints at once using *SetRange*.

#### Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the *Fields* property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a *TSQLClientDataSet* component named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first

column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

**Tip** To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 18-33.

### Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

**Warning** Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName* method. For example,

```
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 18-33.

### Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

*SetRange* takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statement establishes a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

### Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith.” The value specification could also be:

```
Contacts['LastName'] := 'Sm';
```

This statement includes records that have *LastName* greater than or equal to “Sm.”

### Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a dataset to *True* to exclude records equal to ending range. For example,

```
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

## Modifying a range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

The process for editing and applying a range involves these general steps:

- 1 Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
- 2 Modifying the ending index value for the range.
- 3 Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

### Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the *Fields* property overwrite the current index values to use when applying the range.

**Tip** If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range. For more information about ranges based on partial keys, see “Specifying a range based on partial keys” on page 18-32.

### Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range.

## Applying or canceling a range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

### Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* method. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

### Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```

:
MyTable.CancelRange;
:
{later on, use the same range again. No need to call SetRangeStart, etc.}
MyTable.ApplyRange;
:

```

## Creating master/detail relationships

---

Table-type datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Table-type datasets support master/detail relationships in two very distinct ways:

- All table-type datasets can act as the detail of another dataset by linking cursors. This process is described in “Making the table a detail of another dataset” below.
- *TTable*, *TSQLTable*, and all client datasets can act as the master in a master/detail relationship that uses nested detail tables. This process is described in “Using nested detail tables” on page 18-36.

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is any type of dataset. With nested details, the type of dataset that can act as the detail table is limited, but they provide for more options in how to display the data. If the master is a client dataset, nested details provide a more robust mechanism for applying cached updates.

### Making the table a detail of another dataset

A table-type dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the table gets data from the master table. This data source can be linked to any type of dataset. For instance, by specifying a query's data source in this property, you can link a client dataset as the detail of the query, so that the client dataset tracks events occurring in the query.

The dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the detail dataset, first specify the index in the detail dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you specify the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the index fields in the detail table. To link datasets on multiple column names, separate field names with semicolons:

```
Parts.MasterFields := 'OrderNo;ItemNo';
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the *CustomersTable*, and the detail table is *OrdersTable*. The example uses the BDE-based *TTable* component, but you can use the same methods to link any table-type datasets.

- 1 Place two *TTable* components and two *TDataSource* components in a data module.
- 2 Set the properties of the first *TTable* component as follows:
  - *DatabaseName*: DBDEMOS
  - *TableName*: CUSTOMER
  - *Name*: CustomersTable
- 3 Set the properties of the second *TTable* component as follows:
  - *DatabaseName*: DBDEMOS
  - *TableName*: ORDERS
  - *Name*: OrdersTable
- 4 Set the properties of the first *TDataSource* component as follows:
  - *Name*: CustSource
  - *DataSet*: CustomersTable
- 5 Set the properties of the second *TDataSource* component as follows:
  - *Name*: OrdersSource
  - *DataSet*: OrdersTable
- 6 Place two *TDBGrid* components on a form.
- 7 Choose File | Use Unit to specify that the form should use the data module.
- 8 Set the *DataSource* property of the first grid component to "CustSource", and set the *DataSource* property of the second grid to "OrdersSource".
- 9 Set the *MasterSource* property of *OrdersTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).



- 10 Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:
  - In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
  - Select *CustNo* in both the Detail Fields and Master Fields field lists.
  - Click the Add button to add this join condition. In the Joined Fields list, “CustNo -> CustNo” appears.
  - Choose OK to commit your selections and exit the Field Link Designer.
- 11 Set the *Active* properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.
- 12 Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

## Using nested detail tables

A nested table is a detail dataset that is the value of a single dataset field in another (master) dataset. For datasets that represent server data, a nested detail dataset can only be used for a dataset field on the server. *TClientDataSet* components do not represent server data, but they can also contain dataset fields if you create a dataset for them that contains nested details, or if they receive data from a provider that is linked to the master table of a master/detail relationship.

**Note** For *TClientDataSet*, using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server.

To use nested detail sets, the *ObjectView* property of the master dataset must be *True*. When your table-type dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on how this works, see “Displaying dataset fields” on page 19-26.

Alternately, you can display and edit detail datasets in data-aware controls by using a separate dataset component for the detail set. At design time, create persistent fields for the fields in your (master) dataset, using the Fields Editor: right click the master dataset and choose Fields Editor. Add a new persistent field to your dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of the detail table. You must also add persistent fields for any other fields used in your master dataset.

The dataset component for the detail table is a dataset descendant of a type allowed by the master table. *TTable* components only allow *TNestedDataSet* components as nested datasets. *TSQLTable* components allow other *TSQLTable* components. *TClientDataSet* components allow other client datasets. Choose a dataset of the appropriate type from the Component palette and add it to your form or data module. Set this detail dataset’s *DataSetField* property to the persistent DataSet field in the master dataset. Finally, place a data source component on the form or data module and set its *DataSet* property to the detail dataset. Data-aware controls can use this data source to access the data in the detail set.

## Controlling Read/write access to tables

---

By default when a table-type dataset is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

**Note** This is not true for *TClientDataSet*, which determines whether users can edit data from information that the dataset provider supplies with data packets. It is also not true for *TSQLTable*, which is a unidirectional dataset, and hence always read-only.

When the table opens, you can check the *CanModify* property to ascertain whether the underlying database (or the dataset provider) allows users to edit the data in the table. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided the table's *ReadOnly* property is *False*.

*ReadOnly* determines whether a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening the table.

**Note** *ReadOnly* is implemented on all table-type datasets except *TSQLTable*, which is always read-only.

## Creating and deleting tables

---

Some table-type datasets let you create and delete the underlying tables at design time or at runtime. Typically, database tables are created and deleted by a database administrator. However, it can be handy during application development and testing to create and destroy database tables that your application can use.

### Creating tables

*TTable* and *TIBTable* both let you create the underlying database table without using SQL. Similarly, *TClientDataSet* lets you create a dataset when you are not working with a dataset provider. Using *TTable* and *TClientDataSet*, you can create the table at design time or runtime. *TIBTable* only lets you create tables at runtime.

Before you can create the table, you must be set properties to specify the structure of the table you are creating. In particular, you must specify

- The database that will host the new table. For *TTable*, you specify the database using the *DatabaseName* property. For *TIBTable*, you must use a *TIBDataBase* component, which is assigned to the *Database* property. (Client datasets do not use a database.)
- The type of database (*TTable* only). Set the *TableType* property to the desired type of table. For Paradox, dBASE, or ASCII tables, set *TableType* to *ttParadox*, *ttDBase*, or *ttASCII*, respectively. For all other table types, set *TableType* to *ttDefault*.

- The name of the table you want to create. Both *TTable* and *TIBTable* have a *TableName* property for the name of the new table. Client datasets do not use a table name, but you should specify the *FileName* property before you save the new table. If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered. To avoid overwriting an existing table, you can check the *Exists* property at runtime. *Exists* is only available on *TTable* and *TIBTable*.
- The fields for the new table. There are two ways to do this:
  - You can add field definitions to the *FieldDefs* property. At design time, double-click the *FieldDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the *AddFieldDef* method to add each new field definition. For each new field definition, set the properties of the *TFieldDef* object to specify the desired attributes of the field.
  - You can use persistent field components instead. At design time, double-click on the dataset to bring up the *Fields* editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.
- Indexes for the new table (optional). At design time, double-click the *IndexDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of index definitions. At runtime, clear any existing index definitions, and then use the *AddIndexDef* method to add each new index definition. For each new index definition, set the properties of the *TIndexDef* object to specify the desired attributes of the index.

**Note** You can't define indexes for the new table if you are using persistent field components instead of field definition objects.

To create the table at design time, right-click the dataset and choose Create Table (*TTable*) or Create Data Set (*TClientDataSet*). This command does not appear on the context menu until you have specified all the necessary information.

To create the table at runtime, call the *CreateTable* method (*TTable* and *TIBTable*) or the *CreateDataSet* method (*TClientDataSet*).

**Note** You can set up the definitions at design time and then call the *CreateTable* (or *CreateDataSet*) method at runtime to create the table. However, to do so you must indicate that the definitions specified at runtime should be saved with the dataset component. (by default, field and index definitions are generated dynamically at runtime). Specify that the definitions should be saved with the dataset by setting its *StoreDefs* property to *True*.

**Tip** If you are using *TTable*, you can preload the field definitions and index definitions of an existing table at design time. Set the *DatabaseName* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *DatabaseName* and *TableName* to specify the table you want to create, canceling any prompts to rename the existing table.

**Note** When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, and dataset fields).

The following code creates a new table at runtime and associates it with the DBDEMOS alias. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```

var
  TableFound: Boolean;
begin
  with TTable.Create(nil) do // create a temporary TTable component
    begin
      try
        { set properties of the temporary TTable component }
        Active := False;
        DatabaseName := 'DBDEMOS';
        TableName := Edit1.Text;
        TableType := ttDefault;
        { define fields for the new table }
        FieldDefs.Clear;
        with FieldDefs.AddFieldDef do begin
          Name := 'First';
          DataType := ftString;
          Size := 20;
          Required := False;
        end;
        with FieldDefs.AddFieldDef do begin
          Name := 'Second';
          DataType := ftString;
          Size := 30;
          Required := False;
        end;
        { define indexes for the new table }
        IndexDefs.Clear;
        with IndexDefs.AddIndexDef do begin
          Name := '';
          Fields := 'First';
          Options := [ixPrimary];
        end;
        TableFound := Exists; // check whether the table already exists
        if TableFound then
          if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
            mtConfirmation, mbYesNoCancel, 0) = mrYes then
            TableFound := False;
          if not TableFound then
            CreateTable; // create the table
        finally
          Free; // destroy the temporary TTable when done
        end;
      end;
    end;
  end;
end;

```

## Deleting tables

*TTable* and *TIBTable* let you delete tables from the underlying database table without using SQL. To delete a table at runtime, call the dataset's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
```

**Caution** When you delete a table with *DeleteTable*, the table and all its data are gone forever.

If you are using *TTable*, you can also delete tables at design time: Right-click the table component and select Delete Table from the context menu. The Delete Table menu pick is only present if the table component represents an existing database table (the *DatabaseName* and *TableName* properties specify an existing table).

## Emptying tables

---

Many table-type datasets supply a single method that lets you delete all rows of data in the table.

- For *TTable* and *TIBTable*, you can delete all the records by calling the *EmptyTable* method at runtime:

```
PhoneTable.EmptyTable;
```

- For *TADOTable*, you can use the *DeleteRecords* method.

```
PhoneTable.DeleteRecords;
```

- For *TSQLTable*, you can use the *DeleteRecords* method as well. Note, however, that the *TSQLTable* version of *DeleteRecords* never takes any parameters.

```
PhoneTable.DeleteRecords;
```

- For client datasets, you can use the *EmptyDataSet* method.

```
PhoneTable.EmptyDataSet;
```

**Note** For tables on SQL servers, these methods only succeed if you have DELETE privilege for the table.

**Caution** When you empty a dataset, the data you delete is gone forever.

## Synchronizing tables

---

If you have two or more datasets that represent the same database table but do not share a data source component, then each dataset has its own view on the data and its own current record. As users access records through each datasets, the components' current records will differ.

If the datasets are all instances of *TTable*, or all instances of *TIBTable*, or all client datasets, you can force the current record for each of these datasets to be the same by calling the *GotoCurrent* method. *GotoCurrent* sets its own dataset's current record to the current record of a matching dataset. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

**Tip** If your application needs to synchronize datasets in this manner, put the datasets in a data module and add the unit for the data module to the uses clause of each unit that accesses the tables.

To synchronize datasets from separate forms, you must add one form's unit to the uses clause of the other, and you must qualify at least one of the dataset names with its form name. For example:

```
CustomerTableOne.GotoCurrent (Form2.CustomerTableTwo);
```

## Using query-type datasets

---

To use a query-type dataset,

- 1 Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
- 2 Identify the database server to query. Each query-type dataset does this differently, but typically you specify a database connection component:
  - For *TQuery*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
  - For *TADOQuery*, specify a *TADOConnection* component using the *Connection* property.
  - For *TSQLQuery*, specify a *TSQLConnection* component using the *SQLConnection* property.
  - For *TIBQuery*, specify a *TIBConnection* component using the *Database* property.

For information about using database connection components, see Chapter 17, "Connecting to databases".

- 3 Specify an SQL statement in the *SQL* property of the dataset, and optionally specify any parameters for the statement. For more information, see "Specifying the query" on page 18-42 and "Using parameters in queries" on page 18-43.
- 4 If the query data is to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the query-type dataset. The data source component forwards the results of the query (called a *result set*) to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
- 5 Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. To execute queries that only perform an action on a table and return no result set, use the *ExecSQL* method at runtime. If you plan to execute the query more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the query. For information about preparing a query, see "Preparing queries" on page 18-47.

## Specifying the query

---

For true query-type datasets, you use the *SQL* property to specify the SQL statement for the dataset to execute. Some datasets, such as *TADODataSet*, *TSQLDataSet*, and client datasets, use a *CommandText* property to accomplish the same thing.

Most queries that return records are *SELECT* commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

Queries that do not return records include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than *SELECT* statements (For example, *INSERT*, *DELETE*, *UPDATE*, *CREATE INDEX*, and *ALTER TABLE* commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution. In most cases, the SQL command must be only one complete SQL statement, although that statement can be as complex as necessary (for example, a *SELECT* statement with a *WHERE* clause that uses several nested logical operators such as *AND* and *OR*). Some servers also support “batch” syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements when you specify the query.

The SQL statements used by queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user’s view of and access to data on the fly at runtime without having to alter the SQL statement. For more information about parameterized queries, see “Using parameters in queries” on page 18-43.

### Specifying a query using the SQL property

When using a true query-type dataset (*TQuery*, *TADOQuery*, *TSQLQuery*, or *TIBQuery*), assign the query to the *SQL* property. The *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to modify and debug the query if you divide the statement into logical units:

```
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT CustNo, OrderNO, SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER BY SaleDate');
MyQuery.Open;
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the `ORDER BY` clause already exists on the third line of the statement. It is referenced via the `SQL` property using an index of 2.

```
MyQuery.SQL[2] := 'ORDER BY OrderNo';
```

**Note** The dataset must be closed when you specify or modify the `SQL` property.

At design time, use the String List editor to specify the query. Click the ellipsis button by the `SQL` property in the Object Inspector to display the String List editor.

**Note** With some versions of Delphi, if you are using `TQuery`, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select the query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use SQL Builder, open it and use its online help.

Because the `SQL` property is a `TStrings` object, you can load the text of the query from a file by calling the `TStrings.LoadFromFile` method:

```
MyQuery.SQL.LoadFromFile('custquery.sql');
```

You can also use the `Assign` method of the `SQL` property to copy the contents of a string list object into the `SQL` property. The `Assign` method automatically clears the current contents of the `SQL` property before copying the new statement:

```
MyQuery.SQL.Assign(Memo1.Lines);
```

## Specifying a query using the `CommandText` property

When using `TADODataSet`, `TSQLDataSet`, or a client dataset, assign the text of the query statement to the `CommandText` property:

```
MyQuery.CommandText := 'SELECT CustName, Address FROM Customer';
```

At design time, you can type the query directly into the Object Inspector, or, if the dataset already has an active connection to the database, you can click the ellipsis button by the `CommandText` property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

## Using parameters in queries

---

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a `WHERE` clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following `INSERT` statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, `:Name`, `:Capital`, and `:Population` are placeholders for actual values supplied to the statement at runtime by your application. Note that the names of parameters begin with a colon. The colon is required so that the parameter names



can be distinguished from literal values. You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. *TQuery*, *TIBQuery*, *TSQLQuery*, and client datasets use the *Params* property to store these values. *TADOQuery* uses the *Parameters* property instead. *Params* (or *Parameters*) is a collection of parameter objects (*TParam* or *TParameter*), where each object represents a single parameter. When you specify the text for the query, the dataset generates this set of parameter objects, and (depending on the dataset type) initializes any of their properties that it can deduce from the query.

**Note** You can suppress the automatic generation of parameter objects in response to changing the query text by setting the *ParamCheck* property to *False*. This is useful for data definition language (DDL) statements that contain parameters as part of the DDL statement that are not parameters for the query itself. For example, the DDL statement to create a stored procedure may define parameters that are part of the stored procedure. By setting *ParamCheck* to *False*, you prevent these parameters from being mistaken for parameters of the query.

Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

**Tip** It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is “Number,” then its corresponding parameter would be “:Number”. Using matching names is especially important if the dataset uses a datasource to obtain parameter values from another dataset. This process is described in “Establishing master/detail relationships using parameters” on page 18-46.

## Supplying parameters at design time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the Object Inspector. If the SQL statement does not contain any parameters, no objects are listed in the collection editor.

**Note** The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for query parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the Object Inspector to modify its properties.

When using the *Params* property (*TParam* objects), you will want to inspect or modify the following:

- The *DataType* property lists the data type for the parameter’s value. For some datasets, this value may be correctly initialized. If the dataset did not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.

The *DataType* property lists the logical data type for the parameter. In general, these data types conform to server data types. For specific logical type-to-server data type mappings, see the documentation for the data access mechanism (BDE, dbExpress, InterBase).

- The *ParamType* property lists the type of the selected parameter. For queries, this is always *ptInput*, because queries can only contain input parameters. If the value of *ParamType* is *ptUnknown*, change it to *ptInput*.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

When using the *Parameters* property (*TParameter* objects), you will want to inspect or modify the following:

- The *DataType* property lists the data type for the parameter's value. For some data types, you must provide additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *Direction* property lists the type of the selected parameter. For queries, this is always *pdInput*, because queries can only contain input parameters.
- The *Attributes* property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

## Supplying parameters at runtime

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name (not available for *TADOQuery*)
- *Params* or *Parameters* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* or *Parameters.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the *:Capital* parameter:

```
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the *:Capital* parameter is the first parameter in the SQL statement):

```
SQLQuery1.Params[0].AsString := Edit1.Text;
```

The command line below sets three parameters at once, using the *Params.ParamValues* property:

```
Query1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Note that *ParamValues* uses *Variants*, avoiding the need to cast values.

## Establishing master/detail relationships using parameters

---

To set up a master/detail relationship where the detail set is a query-type dataset, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, all query-type datasets except *TIBQuery* provide an easier mechanism using the *DataSource* property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, query-type datasets attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

The first step is to set up the Customer dataset:

- 1 Add a table-type dataset to your application and bind it to the Customer table.
- 2 Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.
- 3 Add a query-type dataset and set its *SQL* property to

```
SELECT CustID, OrderNo, SaleDate
FROM Orders
WHERE CustID = :ID
```

Note that the name of the parameter is the same as the name of the field in the master (Customer) table.

- 4 Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail dataset is not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by *CustomerSource*. *CustomerSource* gets its data

from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called “ID,” the value from the *ID* field in the current record of the master dataset is assigned to the *:ID* parameter for the detail dataset’s SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset’s SELECT statement executes to retrieve all orders based on the current customer id.

## Preparing queries

---

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the data access layer and the database server for parsing, resource allocation, and optimization. In some datasets, the dataset may perform additional setup operations when preparing the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by setting the *Prepared* property to *True*. If you do not prepare a query before executing it, the dataset automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though the dataset prepares queries for you, you can improve performance by explicitly preparing the dataset before you open it the first time.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you add a parameter).

**Note** When you change the text of the *SQL* property for a query, the dataset automatically closes and unprepares the query.

## Executing queries that don’t return a result set

---

When a query returns a set of records (such as a SELECT query), you execute the query the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often SQL commands do not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records).

For all query-type datasets, you can execute a query that does not return a result set by calling *ExecSQL*:

```
CustomerQuery.ExecSQL; { query does not return a result set }
```

**Tip** If you are executing the query multiple times, it is a good idea to set the *Prepared* property to *True*.

Although the query does not return any records, you may want to know the number of records it affected (for example, the number of records deleted by a DELETE query). The *RowsAffected* property gives the number of affected records after a call to *ExecSQL*.

**Tip** When you do not know at design time whether the query returns a result set (for example, if the user supplies the query dynamically at runtime), you can code both types of query execution statements in a **try...except** block. Put a call to the *Open* method in the **try** clause. An action query is executed when the query is activated with the *Open* method, but an exception occurs in addition to that. Check the exception, and suppress it if it merely indicates the lack of a result set. (For example, *TQuery* indicates this by an *ENoResultSet* exception.)

## Using unidirectional result sets

---

When a query-type dataset returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source. Unless you are using *dbExpress*, this cursor is bi-directional by default. A bi-directional cursor can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries.

If you do not need to be able to navigate backward through a result set, *TQuery* and *TIBQuery* let you improve query performance by requesting a unidirectional cursor instead. To request a unidirectional cursor, set the *UniDirectional* property to *True*.

Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
if not (CustomerQuery.Prepared) then
begin
    CustomerQuery.UniDirectional := True;
    CustomerQuery.Prepared := True;
end;
CustomerQuery.Open; { returns a result set with a one-way cursor }
```

**Note** Do not confuse the *UniDirectional* property with a unidirectional dataset. Unidirectional datasets (*TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*) use *dbExpress*, which only returns unidirectional cursors. In addition to restricting the ability to navigate backwards, unidirectional datasets do not buffer records, and so have additional limitations (such as the inability to use filters).

## Using stored procedure-type datasets

---

How your application uses a stored procedure depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

In general terms, to access a stored procedure on a server, an application must:

- 1 Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
- 2 Identify the database server that defines the stored procedure. Each stored procedure-type dataset does this differently, but typically you specify a database connection component:
  - For *TStoredProc*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
  - For *TADOStoredProc*, specify a *TADOConnection* component using the *Connection* property.
  - For *TSQLStoredProc*, specify a *TSQLConnection* component using the *SQLConnection* property.
  - For *TIBStoredProc*, specify a *TIBConnection* component using the *Database* property.

For information about using database connection components, see Chapter 17, “Connecting to databases”.

- 3 Specify the stored procedure to execute. For most stored procedure-type datasets, you do this by setting the *StoredProcName* property. The one exception is *TADOStoredProc*, which has a *ProcedureName* property instead.
- 4 If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the stored procedure-type dataset. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
- 5 Provide input parameter values for the stored procedure, if necessary. If the server does not provide information about all stored procedure parameters, you may need to provide additional input parameter information, such as parameter names and data types. For information about working with stored procedure parameters, see “Working with stored procedure parameters” on page 18-50.
- 6 Execute the stored procedure. For stored procedures that return a cursor, use the *Active* property or the *Open* method. To execute stored procedures that do not return any results or that only return output parameters, use the *ExecProc* method at runtime. If you plan to execute the stored procedure more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the stored procedure. For information about preparing a query, see “Executing stored procedures that don’t return a result set” on page 18-53.
- 7 Process any results. These results can be returned as result and output parameters, or they can be returned as a result set that populates the stored procedure-type dataset. Some stored procedures return multiple cursors. For details on how to access the additional cursors, see “Fetching multiple result sets” on page 18-53.

## Working with stored procedure parameters

---

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.
- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- *A result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, on MS-SQL Server and Sybase stored procedures always return a result parameter, but the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by the *Params* property (in *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*) or the *Parameters* property (in *TADOStoredProc*). When you assign a value to the *StoredProcName* (or *ProcedureName*) property, the dataset automatically generates objects for each parameter of the stored procedure. For some datasets, if the stored procedure name is not specified until runtime, objects for each parameter must be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* or *TParameter* objects allows a single dataset to be used with any number of available stored procedures.

**Note** Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters.

### Setting up parameters at design time

You can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the Object Inspector.

**Important** You can assign values to input parameters by selecting them in the parameter collection editor and using the Object Inspector to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add, you must fully describe the parameter. Even if you do not need to add any parameters, you should check the properties of individual parameter objects to ensure that they are correct.

If the dataset has a *Params* property (*TParam* objects), the following properties must be correctly specified:

- The *Name* property indicates the name of the parameter as it is defined by the stored procedure.
- The *DataType* property gives the data type for the parameter's value. When using *TSQLStoredProc*, some data types require additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *ParamType* property indicates the type of the selected parameter. This can be *ptInput* (for input parameters), *ptOutput* (for output parameters), *ptInputOutput* (for input/output parameters) or *ptResult* (for result parameters).
- The *Value* property specifies a value for the selected parameter. You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

If the dataset uses a *Parameters* property (*TParameter* objects), the following properties must be correctly specified:

- The *Name* property indicates the name of the parameter as it is defined by the stored procedure.
- The *DataType* property gives the data type for the parameter's value. For some data types, you must provide additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *Direction* property gives the type of the selected parameter. This can be *pdInput* (for input parameters), *pdOutput* (for output parameters), *pdInputOutput* (for input/output parameters) or *pdReturnValue* (for result parameters).
- The *Attributes* property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The *Value* property specifies a value for the selected parameter. Do not set values for output and result parameters. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.



## Using parameters at runtime

With some datasets, if the name of the stored procedure is not specified until runtime, no *TParam* objects are automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method:

```
var
  P1, P2: TParam;
begin
  ...
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[1].Name := 'PROJ_ID';
      ParamByName('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByName('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ...
end;
```

Even if you do not need to add the individual parameter objects at runtime, you may want to access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. You can use the dataset's *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/output parameter, executes the stored procedure, and retrieves the returned value:

```
with SQLStoredProc1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

## Preparing stored procedures

---

As with query-type datasets, stored procedure-type datasets must be prepared before they execute the stored procedure. Preparing a stored procedure tells the data access layer and the database server to allocate resources for the stored procedure and to bind parameters. These operations can improve performance.

If you attempt to execute a stored procedure before preparing it, the dataset automatically prepares it for you, and then unprepares it after it executes. If you plan

to execute a stored procedure a number of times, it is more efficient to explicitly prepare it by setting the *Prepared* property to *True*.

```
MyProc.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the stored procedure are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change the parameters when using Oracle overloaded procedures).

## Executing stored procedures that don't return a result set

---

When a stored procedure returns a cursor, you execute it the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often stored procedures do not return any data, or only return results in output parameters. You can execute a stored procedure that does not return a result set by calling *ExecProc*. After executing the stored procedure, you can use the *ParamByName* method to read the value of the result parameter or of any output parameters:

```
MyStoredProcedure.ExecProc; { does not return a result set }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
```

**Note** *TADOStoredProc* does not have a *ParamByName* method. To obtain output parameter values when using ADO, access parameter objects using the *Parameters* property.

**Tip** If you are executing the procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

## Fetching multiple result sets

---

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. If you are using *TSQLStoredProc* or *TADOStoredProc*, you can access the other sets of records by calling the *NextRecordSet* method:

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

In *TSQLStoredProc*, *NextRecordSet* returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. In *TADOStoredProc*, *NextRecordset* returns an interface that can be assigned to the *RecordSet* property of an existing ADO dataset. For either class, the method returns the number of records in the returned dataset as an output parameter.

The first time you call *NextRecordSet*, it returns the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional cursors, *NextRecordSet* returns **nil**.



## Working with field components

This chapter describes the properties, events, and methods common to the *TField* object and its descendants. Field components represent individual fields (columns) in datasets. This chapter also describes how to use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a *TField* object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as *TDBEdit* and *TDBGrid* access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

**Table 19.1** TFloatField properties that affect data display

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications. The following sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

## Dynamic field components

---

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact *TField* descendant created for each column is determined by field type information received from the database or (for *TClientDataSet*) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool such as SQL explorer, you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application:

- 1 Place datasets and data sources in a data module.
- 2 Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
- 3 Associate the data sources with the datasets.
- 4 Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any field name you specify will exist when the dataset is opened.
- 5 Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

## Persistent field components

---

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.
- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

**Note** When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see “Dynamic field components” on page 19-2.

**Note** One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see “Creating a customized grid” on page 15-16.

## Creating persistent fields

---

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset:

- 1 Place a dataset in a data module.
- 2 Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, if you are using *TADODataset*, you can set the *Connection* property to a properly configured *TADOConnection* component and set the *CommandText* property to a valid query.
- 3 Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed.

The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

- 4 Choose Add Fields from the Fields editor context menu.
- 5 Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open the dataset.

## Arranging persistent fields

---

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields:

- 1 Select the fields. You can select and order one or more fields at a time.
- 2 Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use *Ctrl+Up* and *Ctrl+Dn* to change an individual field's order in the list.

## Defining new persistent fields

---

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's *Name* property and is only provided for informational purposes (*Name* is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.



- The Type combo box in the Field properties group lets you specify the field component’s data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

**Table 19.2** Special persistent field kinds

Field kind	Purpose
Data	Replaces an existing field (for example to change its data type)
Calculated	Displays values calculated at runtime by a dataset’s <i>OnCalcFields</i> event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify. (not supported by unidirectional datasets)
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Aggregate	Displays a value summarizing the data in a set of records from a client dataset.

The Lookup definition group box is only used to create lookup fields. This is described more fully in “Defining a lookup field” on page 19-8.

### Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field’s data type directly, you must define a new field to replace it.

**Important** Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset, follow these steps:

- 1 Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
- 2 In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
- 3 Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.

- 4 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
- 5 Select Data in the Field type radio group if it is not already selected.
- 6 Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 19-10.

### Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

- 1 Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
- 2 Choose a data type for the field from the Type combo box.
- 3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
- 4 Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.
- 5 Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's **type** declaration.
- 6 Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field" on page 19-7.

**Note** To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 19-10.

### Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

- 1 Select the dataset component from the Object Inspector drop-down list.
- 2 Choose the Object Inspector Events page.
- 3 Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the dataset component.
- 4 Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
+ ' ' + CustomersZip.Value;
```

**Note** When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's *State* property and only recomputing the value when *State* is *dsInternalCalc*. See "Using internally calculated fields in client datasets" on page 23-10 for details.

## Defining a lookup field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

**Note** Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box:

- 1 Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
- 2 Choose a data type for the field from the Type combo box.
- 3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

- 4 Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
- 5 Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
- 6 Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.
- 7 Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.
- 8 Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the *LookupCache* property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to *True* to cache the values of a lookup field when the *LookupDataSet* is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields* values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

**Tip** You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Be sure that the *LookupDataSet* property is *nil*. Then, use the *LookupList* property's *Add* method to fill it with lookup values. Set the *LookupCache* property to *True*. The field will use the supplied lookup list without overwriting it with values from a lookup dataset.

If every record of *DataSet* has different values for *KeyFields*, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call *RefreshLookupList* to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the *LookupResultField* for every set of *LookupKeyFields* values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

## Defining an aggregate field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See “Using maintained aggregates” on page 23-11 for details about maintained aggregates.

To create an aggregate field in the New Field dialog box:

- 1 Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
- 2 Choose aggregate data type for the field from the Type combo box.
- 3 Select Aggregate in the Field type radio group.
- 4 Choose OK. The newly defined aggregate field is automatically added to the client dataset and its *Aggregates* property is automatically updated to include the appropriate aggregate specification.
- 5 Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see “Specifying aggregates” on page 23-11.

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the underlying client data set.

## Deleting persistent field components

---

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table. To remove one or more persistent field components for a dataset:

- 1 Select the field(s) to remove in the Fields editor list box.
- 2 Press *Del*.

**Note** You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see “Creating persistent fields” on page 19-4.

**Note** If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

## Setting persistent field properties and events

---

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value

can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

## Setting display and edit properties at design time

To edit the display properties of a selected field component, switch to the Properties page on the Object Inspector window. The following table summarizes display properties that can be edited.

**Table 19.3** Field component properties

Property	Purpose
<i>Alignment</i>	Left justifies, right justifies, or centers a field contents within a data-aware component.
<i>ConstraintErrorMessage</i>	Specifies the text to display when edits clash with a constraint condition.
<i>CustomConstraint</i>	Specifies a local constraint to apply to data during editing.
<i>Currency</i>	Numeric fields only. <i>True</i> : displays monetary values. <i>False</i> (default): does not display monetary values.
<i>DisplayFormat</i>	Specifies the format of data displayed in a data-aware component.
<i>DisplayLabel</i>	Specifies the column name for a field in a data-aware grid component.
<i>DisplayWidth</i>	Specifies the width, in characters, of a grid column that display this field.
<i>EditFormat</i>	Specifies the edit format of data in a data-aware component.
<i>EditMask</i>	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
<i>FieldKind</i>	Specifies the type of field to create.
<i>FieldName</i>	Specifies the actual name of a column in the table from which the field derives its value and data type.
<i>HasConstraints</i>	Indicates whether there are constraint conditions imposed on a field.
<i>ImportedConstraint</i>	Specifies an SQL constraint imported from the Data Dictionary or an SQL server.
<i>Index</i>	Specifies the order of the field in a dataset.
<i>LookupDataSet</i>	Specifies the table used to look up field values when <i>Lookup</i> is <i>True</i> .
<i>LookupKeyFields</i>	Specifies the field(s) in the lookup dataset to match when doing a lookup.
<i>LookupResultField</i>	Specifies the field in the lookup dataset from which to copy values into this field.
<i>MaxValue</i>	Numeric fields only. Specifies the maximum value a user can enter for the field.
<i>MinValue</i>	Numeric fields only. Specifies the minimum value a user can enter for the field.
<i>Name</i>	Specifies the component name of the field component within Delphi.
<i>Origin</i>	Specifies the name of the field as it appears in the underlying database.
<i>Precision</i>	Numeric fields only. Specifies the number of significant digits.

**Table 19.3** Field component properties (continued)

Property	Purpose
<i>ReadOnly</i>	<i>True</i> : Displays field values in data-aware controls, but prevents editing. <i>False</i> (the default): Permits display and editing of field values.
<i>Size</i>	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of <i>TBytesField</i> and <i>TVarBytesField</i> fields.
<i>Tag</i>	Long integer bucket available for programmer use in every component as needed.
<i>Transliterate</i>	<i>True</i> (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. <i>False</i> : Locale translation does not occur.
<i>Visible</i>	<i>True</i> (the default): Permits display of field in a data-aware grid. <i>False</i> : Prevents display of field in a data-aware grid component. User-defined components can make display decisions based on this property.

Not all properties are available for all field components. For example, a field component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type *TFloatField* does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see “Controlling and masking user input” on page 19-14.

### Setting field component properties at runtime

You can use and manipulate the properties of field component at runtime. Access persistent field components by name, where the name can be obtained by concatenating the field name to the dataset name.

For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly := True;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
```

### Creating attribute sets for field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

**Note** Attribute sets and the Data Dictionary are only available for BDE-enabled datasets.

To create an attribute set based on a field component in a dataset:

- 1 Double-click the dataset to invoke the Fields editor.
- 2 Select the field for which to set properties.
- 3 Set the desired properties for the field in the Object Inspector.
- 4 Right-click the Fields editor list box to invoke the context menu.
- 5 Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save Attributes As instead of Save Attributes from the context menu.

Once you have created a new attribute set and added it to the Data Dictionary, you can then associate it with other persistent field components. Even if you later remove the association, the attribute set remains defined in the Data Dictionary.

**Note** You can also create attribute sets directly from the SQL Explorer. When you create an attribute set using SQL Explorer, it is added to the Data Dictionary, but not applied to any fields. SQL Explorer lets you specify two additional attributes: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) that is automatically placed on a form when a field based on the attribute set is dragged to the form. For more information, see the online help for the SQL Explorer.

## Associating attribute sets with field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

- 1 Double-click the dataset to invoke the Fields editor.
- 2 Select the field for which to apply an attribute set.
- 3 Invoke the context menu and choose Associate Attributes.
- 4 Select or enter the attribute set to apply from the Associate Attributes dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

**Important** If the attribute set in the Data Dictionary is changed at a later date, you must reapply the attribute set to each field component that uses it. You can invoke the Fields editor and multi-select field components within a dataset when reapplying attributes.



## Removing attribute associations

If you change your mind about associating an attribute set with a field, you can remove the association by following these steps:

- 1 Invoke the Fields editor for the dataset containing the field.
- 2 Select the field or fields from which to remove the attribute association.
- 3 Invoke the context menu for the Fields editor and choose Unassociate Attributes.

**Important** Unassociating an attribute set does not change any field properties. A field retains the settings it had when the attribute set was applied to it. To change these properties, select the field in the Fields editor and set its properties in the Object Inspector.

## Controlling and masking user input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, and *TDateTimeField*, and *TSQLTimeStampField* components. You can use existing masks or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the Object Inspector.

**Note** For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component:

- 1 Select the component in the Fields editor or Object Inspector.
- 2 Click the Properties page in the Object Inspector.
- 3 Double-click the values column for the *EditMask* field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

## Using default formatting for numeric, date, and time fields

Delphi provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, and *TTimeField*, and *TSQLTimeStampField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

**Table 19.4** Field component formatting routines

Routine	Used by . . .
<i>FormatFloat</i>	<i>TFloatField</i> , <i>TCurrencyField</i>
<i>FormatDateTime</i>	<i>TDateField</i> , <i>TTimeField</i> , <i>TDateTimeField</i> ,
<i>SQLTimeStampToString</i>	<i>TSQLTimeStampField</i>
<i>FormatCurr</i>	<i>TCurrencyField</i> , <i>TBCDField</i>
<i>BcdToStrF</i>	<i>TFMTBcdField</i>

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Control Panel. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to \$1234.56, while the *EditFormat* is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and *EditFormat* properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime.

## Handling events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

**Table 19.5** Field component events

Event	Purpose
<i>OnChange</i>	Called when the value for a field changes.
<i>OnGetText</i>	Called when the value for a field component is retrieved for display or editing.
<i>OnSetText</i>	Called when the value for a field component is set.
<i>OnValidate</i>	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

*OnGetText* and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component:

- 1 Select the component.
- 2 Select the Events page in the Object Inspector.
- 3 Double-click the Value field for the event handler to display its source code window.
- 4 Create or edit the handler code.

## Working with field component methods at runtime

---

Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see the entries for *TField* and its descendants in the online *VCL Reference*.

**Table 19.6** Selected field component methods

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

## Displaying, converting, and accessing field values

---

Data-aware controls such as *TDBEdit* and *TDBGrid* automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Chapter 15, “Using data controls.”

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls, your application is responsible for tracking when to update controls because field values change. If the dataset has a *datasource* component, you can use its events to help you do this. In particular, the *OnDataChange* event lets you know when you may need to update a control's value and the *OnStateChange* event can help you determine when to enable or disable controls. For more information on these events, see “Responding to changes mediated by the data source” on page 15-4.

The following topics discuss how to work with field values so that you can display them in standard controls.

### Displaying field component values in standard controls

---

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a *TEdit* control because the value of the *CustomersCompany* field may have changed:

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
    Edit3.Text := CustomersCompany.Value;
end;
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in properties for handling conversions.

**Note** You can also use Variants to access and set field values. For more information about using variants to access and set field values, see “Accessing field values with the default dataset property” on page 19-19.

### Converting field values

---

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations.

The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

	<b>AsVariant</b>	<b>AsString</b>	<b>AsInteger</b>	<b>AsFloat</b> <b>AsCurrency</b> <b>AsBCD</b>	<b>AsDateTime</b> <b>AsSQLTimeStamp</b>	<b>AsBoolean</b>
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		
TWordField	yes	yes	yes	yes		
TLargeIntField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		
TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes		yes	yes	
TDateField	yes	yes		yes	yes	
TTimeField	yes	yes		yes	yes	
TSQLTimeStampField	yes	yes		yes	yes	
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any data types not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of

days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. Table 19.7 lists permissible conversions that produce special results:

**Table 19.7** Special conversion results

Conversion	Result
<i>String to Boolean</i>	Converts “True,” “False,” “Yes,” and “No” to Boolean. Other values raise exceptions.
<i>Float to Integer</i>	Rounds float value to nearest integer value.
<i>DateTime or SQLTimeStamp to Float</i>	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
<i>Boolean to String</i>	Converts any Boolean value to “True” or “False.”

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

## Accessing field values with the default dataset property

The most general method for accessing a field’s value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Because the *FieldValues* property is of type Variant, it automatically converts other datatypes into a Variant value.

For more information about Variants, see the online help.

## Accessing field values with a dataset’s Fields property

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each

field component's conversion properties. For more information about field component conversion properties, see "Converting field values" on page 19-17.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
begin
    Customers.Edit;
    Customers.Fields[6].AsString := Edit1.Text;
    Customers.Post;
end;
```

## Accessing field values with a dataset's FieldByName method

---

You can also access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
    Customers.Edit;
    Customers.FieldByName('CustNo').AsString := Edit2.Text;
    Customers.Post;
end;
```

## Setting a default value for a field

---

You can specify how a default value for a field in a client dataset or a BDE-enabled dataset should be calculated at runtime using the *DefaultExpression* property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

**Note** If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the dataset posts the record containing the field, before the edited record is applied to the database server.

## Working with constraints

---

Field components in client datasets or BDE-enabled datasets can use SQL server constraints. In addition, your applications can create and use custom constraints for these datasets that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store.

### Creating a custom constraint

---

A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a prevalidation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

*CustomConstraint* is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

**Note** Custom constraints are only available in BDE-enabled and client datasets.

Custom constraints are imposed in addition to any constraints to the field's value that come from the server. To see the constraints imposed by the server, read the *ImportedConstraint* property.

### Using server constraints

---

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than 0 and less than 150. While you could replicate such conditions in your client applications, client datasets and BDE-enabled datasets offer the *ImportedConstraint* property to propagate a server's constraints locally.

*ImportedConstraint* is a read-only property that specifies an SQL clause that limits field values in some manner. For example:

```
Value > 0 and Value < 100
```

Do not change the value of *ImportedConstraint*, except to edit nonstandard or server-specific SQL that has been imported as a comment because it cannot be interpreted by the database engine.



To add additional constraints on the field value, use the *CustomConstraint* property. Custom constraints are imposed in addition to the imported constraints. If the server constraints change, the value of *ImportedConstraint* also changed but constraints introduced in the *CustomConstraint* property persist.

Removing constraints from the *ImportedConstraint* property will not change the validity of field values that violate those constraints. Removing constraints results in the constraints being checked by the server instead of locally. When constraints are checked locally, the error message supplied as the *ConstraintErrorMessage* property is displayed when violations are found, instead of displaying an error message from the server.

## Using object fields

---

Object fields are fields that represent a composite of other, simpler data types. These include ADT (Abstract Data Type) fields, Array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields are fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

**Table 19.8** Types of object field components

Component name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's *ObjectView* property to *True*, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

**Table 19.9** Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classifies the object field.
FieldCount	Number of child fields belonging to the object field.
FieldValues	Provides access to the values of the child fields.

## Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as *TDBEdit* that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed and edited just like any other normal data field.

A *TDBGrid* control displays ADT and array field data differently, depending on the value of the dataset's *ObjectView* property. When *ObjectView* is *False*, each child field appears in a single column. When *ObjectView* is *True*, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma-delimited string containing the child fields.

## Working with ADT fields

ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called *CityEdit*, and use the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

## Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the *Customer* table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
CityEdit.Text := CustomerAddrCity.AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's *ObjectView* property to *True*.

## Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's *FieldByName* method by qualifying the name of the child field with the ADT field's name:

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

## Using the dataset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
CityEdit.Text := Customer['Address.City'];
```

Note that you can omit the property name (*FieldValues*) because *FieldValues* is the dataset's default property.

**Note** Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is *False*.

## Using the ADT field's FieldValues property

You can access the value of a child field with the *TADTField*'s *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

Because *FieldValues* is the default property of *TADTField*, the property name (*FieldValues*) can be omitted. Thus, the following statement is equivalent to the one above:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

## Using the ADT field's Fields property

Each ADT field has a *Fields* property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

or by name:

```
CityEdit.Text :=
  TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

## Working with array fields

---

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The *SparseArrays* property of *TDataSet* determines whether a unique *TField* object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's *ObjectView* property must be set to *True* before you can access the elements of an array field.

## Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos\_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component represent the *TelNos\_Array* field and its six elements:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

## Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

Because *FieldValues* is the default property of *TArrayField*, this can also be written

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

## Using the array field's Fields property

*TArrayField* has a *Fields* property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

## Working with dataset fields

---

Dataset fields provide access to data stored in a nested dataset. The *NestedDataSet* property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.

### Displaying dataset fields

*TDBGrid* controls enable the display of data stored in data set fields. In a *TDBGrid* control, a dataset field is indicated in each cell of a dataset column with the string "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a dataset field, the following code will display the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

### Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested data set is just that, a data set, the means to get at its data is via a *TDataSet* descendant. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a BDE-enabled dataset uses *TNestedTable* to represent the data in its dataset fields, while client datasets use other client datasets.

To access the data in a dataset field,

- 1 Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
- 2 Create a dataset to represent the values in that dataset field. It must be of a type compatible with the parent dataset.
- 3 Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Before inserting records into a nested dataset, you should be sure to post the corresponding record in the master table, if it has just been inserted. If the inserted record is not posted, it will be automatically posted before the nested dataset posts.

## Working with reference fields

---

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the *Fields* property on the *TReferenceField*.

### Displaying reference fields

In a *TDBGrid* control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

### Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset:

- 1 Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
- 2 Create a dataset to represent the value of that dataset field.
- 3 Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's *Fields* property to access the data in a reference field. For example, the following lines are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

When data in a reference field is edited, it is actually the referenced data that is modified.

## Using object fields

To assign a reference field, you need to first use a **SELECT** statement to select the reference from the table, and then assign. For example:

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
  Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

# Using the Borland Database Engine

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. Depending on your version of Delphi, you can use the drivers for local databases (Paradox, dBASE, FoxPro, and Access), SQL Links drivers for remote database servers such as InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2, and an ODBC adapter that lets you supply your own ODBC drivers.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broad range of support for database manipulation. Although you can use the BDE's API directly in your application, the components on the BDE page of the Component palette wrap most of this functionality for you.

**Note** For information on the BDE API, see its online help file, BDE32.hlp, which is installed in the directory where you install the Borland Database Engine.

## BDE-based architecture

---

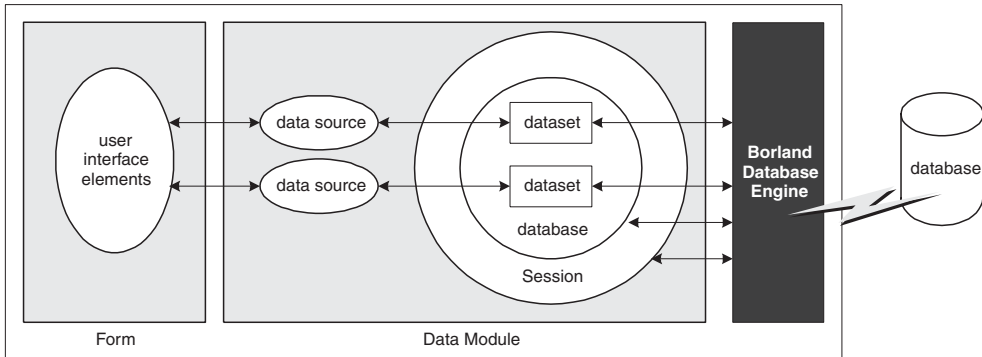
When using the BDE, your application uses a variation of the general database architecture described in "Database architecture" on page 14-5. In addition to the user interface elements, datasource, and datasets common to all Delphi database applications, A BDE-based application can include

- One or more database components to control transactions and to manage database connections.
- One or more session components to isolate data access operations such as database connections, and to manage groups of databases.



The relationships between the components in a BDE-based application are illustrated in Figure 20.1:

**Figure 20.1** Components in a BDE-based application



## Using BDE-enabled datasets

BDE-enabled datasets use the Borland Database Engine (BDE) to access data. They inherit the common dataset capabilities described in Chapter 18, “Understanding datasets,” using the BDE to provide the implementation. In addition, all BDE datasets add properties, events, and methods for

- Associating a dataset with database and session connections.
- Caching BLOBs.
- Obtaining a BDE handle.

There are three BDE-enabled datasets:

- *TTable*, a table-type dataset that represents all of the rows and columns of a single database table. See “Using table-type datasets” on page 18-24 for a description of features common to table-type datasets. See “Using TTable” on page 20-4 for a description of features unique to *TTable*.
- *TQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See “Using query-type datasets” on page 18-41 for a description of features common to query-type datasets. See “Using TQuery” on page 20-8 for a description of features unique to *TQuery*.
- *TStoredProc*, a stored procedure-type dataset that executes a stored procedure that is defined on a database server. See “Using stored procedure-type datasets” on page 18-48 for a description of features common to stored procedure-type datasets. See “Using TStoredProc” on page 20-11 for a description of features unique to *TStoredProc*.

**Note** In addition to the three types of BDE-enabled datasets, there is a BDE-based client dataset (*TBDEClientDataSet*) that can be used for caching updates. For information on caching updates, see “Using a client dataset to cache updates” on page 23-15.

## Associating a dataset with database and session connections

In order for a BDE-enabled dataset to fetch data from a database server it needs to use both a database and a session.

- Databases represent connections to specific database servers. The database identifies a BDE driver, a particular database server that uses that driver, and a set of connection parameters for connecting to that database server. Each database is represented by a *TDatabase* component. You can either associate your datasets with a *TDatabase* component you add to a form or data module, or you can simply identify the database server by name and let Delphi generate an implicit database component for you. Using an explicitly-created *TDatabase* component is recommended for most applications, because the database component gives you greater control over how the connection is established, including the login process, and lets you create and use transactions.

To associate a BDE-enabled dataset with a database, use the *DatabaseName* property. *DatabaseName* is a string that contains different information, depending on whether you are using an explicit database component and, if not, the type of database you are using:

- If you are using an explicit *TDatabase* component, *DatabaseName* is the value of the *DatabaseName* property of the database component.
  - If you want to use an implicit database component and the database has a BDE alias, you can specify a BDE alias as the value of *DatabaseName*. A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on). Use the BDE Administration tool or the SQL explorer to create and manage BDE aliases.
  - If you want to use an implicit database component for a Paradox or dBASE database, you can also use *DatabaseName* to simply specify the directory where the database tables are located.
- A session provides global management for a group of database connections in an application. When you add BDE-enabled datasets to your application, your application automatically contains a session component, named *Session*. As you add database and dataset components to the application, they are automatically associated with this default session. It also controls access to password protected Paradox files, and it specifies directory locations for sharing Paradox files over a network. You can control database connections and access to Paradox files using the properties, events, and methods of the session.

You can use the default session to control all database connections in your application. Alternatively, you can add additional session components at design time or create them dynamically at runtime to control a subset of database connections in an application. To associate your dataset with an explicitly created session component, use the *SessionName* property. If you do not use explicit session components in your application, you do not have to provide a value for this property. Whether you use the default session or explicitly specify a session using the *SessionName* property, you can access the session associated with a dataset by reading the *DBSession* property.

**Note** If you use a session component, the *SessionName* property of a dataset must match the *SessionName* property for the database component with which the dataset is associated.

For more information about *TDatabase* and *TSession*, see “Connecting to databases with TDatabase” on page 20-12 and “Managing database sessions” on page 20-16.

## Caching BLOBs

BDE-enabled datasets all have a *CacheBlobs* property that controls whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records.

In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to *False* to ensure that your application always sees the latest version of a BLOB field.

## Obtaining a BDE handle

You can use BDE-enabled datasets without ever needing to make direct API calls to the Borland Database Engine. The BDE-enabled datasets, in combination with database and session components, encapsulate much of the BDE functionality. However, if you need to make direct API calls to the BDE, you may need BDE handles for resources managed by the BDE. Many BDE APIs require these handles as parameters.

All BDE-enabled datasets include three read-only properties for accessing BDE handles at runtime:

- *Handle* is a handle to the BDE cursor that accesses the records in the dataset.
- *DBHandle* is a handle to the database that contains the underlying tables or stored procedure.
- *DBLocale* is a handle to the BDE language driver for the dataset. The locale controls the sort order and character set used for string data.

These properties are automatically assigned to a dataset when it is connected to a database server through the BDE. For more information about the BDE API, see the online help file, BDE32.HLP.

## Using TTable

---

*TTable* encapsulates the full structure of and data in an underlying database table. It implements all of the basic functionality introduced by *TDataSet*, as well as all of the special features typical of table-type datasets. Before looking at the unique features introduced by *TTable*, you should familiarize yourself with the common database features described in “Understanding datasets,” including the section on table-type datasets that starts on page 18-24.

Because *TTable* is a BDE-enabled dataset, it must be associated with a database and a session. “Associating a dataset with database and session connections” on page 20-3 describes how you form these associations. Once the dataset is associated with a database and session, you can bind it to a particular database table by setting the *TableName* property and, if you are using a Paradox, dBASE, FoxPro, or comma-delimited ASCII text table, the *TableType* property.

**Note** The table must be closed when you change its association to a database, session, or database table, or when you set the *TableType* property. However, before you close the table to change these properties, first post or discard any pending changes. If cached updates are enabled, call the *ApplyUpdates* method to write the posted changes to the database.

*TTable* components are unique in the support they offer for local database tables (Paradox, dBASE, FoxPro, and comma-delimited ASCII text tables). The following topics describe the special properties and methods that implement this support.

In addition, *TTable* components can take advantage of the BDE’s support for batch operations (table level operations to append, update, delete, or copy entire groups of records). This support is described in “Importing data from another table” on page 20-8.

## Specifying the table type for local tables

If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the *TableType* property to determine the table’s type (its expected structure). *TableType* is not used when *TTable* represents an SQL-based table on a database server.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table’s type from its filename extension. Table 20.1 summarizes the file extensions recognized by the BDE and the assumptions it makes about a table’s type:

**Table 20.1** Table types recognized by the BDE based on file extension

Extension	Table type
No file extension	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII text

If your local Paradox, dBASE, and ASCII text tables use the file extensions as described in Table 20.1, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. Table 20.2 indicates the values you can assign to *TableType*:

**Table 20.2** *TableType* values

Value	Table type
ttDefault	Table type determined automatically by the BDE
ttParadox	Paradox

**Table 20.2** TableType values (continued)

Value	Table type
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Comma-delimited ASCII text

## Controlling read/write access to local tables

Like any table-type dataset, *TTable* lets you control read and write access by your application using the *ReadOnly* property.

In addition, for Paradox, dBASE, and FoxPro tables, *TTable* can let you control read and write access to tables by other applications. The *Exclusive* property controls whether your application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. Your request for exclusive access is not honored if the table is already in use when you attempt to open it.

The following statements open a table for exclusive access:

```
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
```

**Note** You can attempt to set *Exclusive* on SQL tables, but some servers do not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

## Specifying a dBASE index file

For most servers, you use the methods common to all table-type datasets to specify an index. These methods are described in "Sorting records with indexes" on page 18-25.

For dBASE tables that use non-production index files or dBASE III PLUS-style indexes (\*.NDX), however, you must use the *IndexFiles* and *IndexName* properties instead. Set the *IndexFiles* property to the name of the non-production index file or list the .NDX files. Then, specify one index in the *IndexName* property to have it actively sorting the dataset.

At design time, click the ellipsis button in the *IndexFiles* property value in the Object Inspector to invoke the Index Files editor. To add one non-production index file or .NDX file: click the Add button in the Index Files dialog and select the file from the Open dialog. Repeat this process once for each non-production index file or .NDX file. Click the OK button in the Index Files dialog after adding all desired indexes.

This same operation can be performed programmatically at runtime. To do this, access the *IndexFiles* property using properties and methods of string lists. When adding a new set of indexes, first call the *Clear* method of the table's *IndexFiles*

property to remove any existing entries. Call the *Add* method to add each non-production index file or .NDX file:

```
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
```

After adding any desired non-production or .NDX index files, the names of individual indexes in the index file are available, and can be assigned to the *IndexName* property. The index tags are also listed when using the *GetIndexNames* method and when inspecting index definitions through the *TIndexDef* objects in the *IndexDefs* property. Properly listed .NDX files are automatically updated as data is added, changed, or deleted in the table (regardless of whether a given index is used in the *IndexName* property).

In the example below, the *IndexFiles* for the *AnimalsTable* table component is set to the non-production index file ANIMALS.MDX, and then its *IndexName* property is set to the index tag called "NAME":

```
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
```

Once you have specified the index file, using non-production or .NDX indexes works the same as any other index. Specifying an index name sorts the data in the table and makes it available for indexed-based searches, ranges, and (for non-production indexes) master-detail linking. See "Using table-type datasets" on page 18-24 for details on these uses of indexes.

There are two special considerations when using dBASE III PLUS-style .NDX indexes with *TTable* components. The first is that .NDX files cannot be used as the basis for master-detail links. The second is that when activating a .NDX index with the *IndexName* property, you must include the .NDX extension in the property value as part of the index name:

```
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
```

## Renaming local tables

To rename a Paradox or dBASE table at design time, right-click the table component and select Rename Table from the context menu.

To rename a Paradox or dBASE table at runtime, call the table's *RenameTable* method. For example, the following statement renames the Customer table to CustInfo:

```
Customer.RenameTable('CustInfo');
```

## Importing data from another table

You can use a table component's *BatchMove* method to import data from another table. *BatchMove* can

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

*BatchMove* takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. Table 20.3 describes the possible settings for the mode specification:

**Table 20.3** BatchMove import modes

Value	Meaning
batAppend	Append all records from the source table to the end of this table.
batAppendUpdate	Append all records from the source table to the end of this table and update existing records in this table with matching records from the source table.
batCopy	Copy all records from the source table into this table.
batDelete	Delete all records in this table that also appear in the source table.
batUpdate	Update existing records in this table with matching records from the source table.

For example, the following code updates all records in the current table with records from the *Customer* table that have the same values for fields in the current index:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

*BatchMove* returns the number of records it imports successfully.

**Caution** Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

*BatchMove* performs only some of the batch operations supported by the BDE. Additional functions are available using the *TBatchMove* component. If you need to move a large amount of data between or among tables, use *TBatchMove* instead of calling a table's *BatchMove* method. For information about using *TBatchMove*, see "Using TBatchMove" on page 20-47.

## Using TQuery

*TQuery* represents a single Data Definition Language (DDL) or Data Manipulation Language (DML) statement (For example, a SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, or ALTER TABLE command). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. *TQuery* implements all of the basic functionality introduced by *TDataSet*, as well as all of the special features typical of query-type datasets. Before looking at the unique features introduced by *TQuery*, you should familiarize yourself with the

common database features described in “Understanding datasets,” including the section on query-type datasets that starts on page 18-41.

Because *TQuery* is a BDE-enabled dataset, it must usually be associated with a database and a session. (The one exception is when you use the *TQuery* for a heterogeneous query.) “Associating a dataset with database and session connections” on page 20-3 describes how you form these associations. You specify the SQL statement for the query by setting the *SQL* property.

A *TQuery* component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL is a subset of the SQL-92 specification. Most DML is supported and enough DDL syntax to work with these types of tables. See the local SQL help, LOCALSQL.HLP, for details on supported SQL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase’s SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference*.
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase. You must install the appropriate SQL Link driver and client software (vendor-supplied) specific to the database server to access a remote server. Any standard SQL syntax supported by these servers is allowed. For information on SQL syntax, limitations, and extensions, see the documentation for your particular server.

## Creating heterogeneous queries

*TQuery* supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query, follow these steps:

- 1 Define separate BDE aliases for each database accessed in the query using the BDE BDE Administration tool or the SQL explorer.
- 2 Leave the *DatabaseName* property of the *TQuery* blank; the names of the databases used will be specified in the SQL statement.
- 3 In the *SQL* property, specify the SQL statement to execute. Precede each table name in the statement with the BDE alias for the table’s database, enclosed in colons. This whole reference is then enclosed in quotation marks.
- 4 Set any parameters for the query in the *Params* property.
- 5 Call *Prepare* to prepare the query for execution prior to executing it for the first time.
- 6 Call *Open* or *ExecSQL* depending on the type of query you are executing.



For example, suppose you define an alias called *Oracle1* for an Oracle database that has a CUSTOMER table, and *Sybase1* for a Sybase database that has an ORDERS table. A simple query against these two tables would be:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM " :Oracle1:CUSTOMER"
JOIN " :Sybase1:ORDERS"
ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

As an alternative to using a BDE alias to specify the database in a heterogeneous query, you can use a *TDatabase* component. Configure the *TDatabase* as normal to point to the database, set the *TDatabase.DatabaseName* to an arbitrary but unique value, and then use that value in the SQL statement instead of a BDE alias name.

## Obtaining an editable result set

To request a result set that users can edit in data-aware controls, set a query component's *RequestLive* property to *True*. Setting *RequestLive* to *True* does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether the query uses the local SQL parser or a server's SQL parser.

- Queries where table names are preceded by a BDE database alias (as in heterogeneous queries) and queries executed against Paradox or dBASE are parsed by the BDE using Local SQL. When queries use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. When using Local SQL, a live result set for a query against a single table or view is returned if the query does not contain any of the following:
  - DISTINCT in the SELECT clause
  - Joins (inner, outer, or UNION)
  - Aggregate functions with or without GROUP BY or HAVING clauses
  - Base tables or views that are not updatable
  - Subqueries
  - ORDER BY clauses not based on an index
- Queries against a remote database server are parsed by the server. If the *RequestLive* property is set to *True*, the SQL statement must abide by Local SQL standards in addition to any server-imposed restrictions because the BDE needs to use it for conveying data changes to the table. A live result set for a query against a single table or view is returned if the query does not contain any of the following:
  - A DISTINCT clause in the SELECT statement
  - Aggregate functions, with or without GROUP BY or HAVING clauses
  - References to more than one base table or updatable views (joins)
  - Subqueries that reference the table in the FROM clause or other tables

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *True*. Even if the query returns a live result set, you may not be able to update the result set directly if it contains linked fields or you switch indexes before attempting an update. If these conditions exist, you should treat the result set as a read-only result set, and update it accordingly.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for SQL queries made against a remote server.

## Updating read-only result sets

Applications can update data returned in a read-only result set if they are using cached updates.

If you are using a client dataset to cache updates, the client dataset or its associated provider can automatically generate the SQL for applying updates unless the query represents multiple tables. If the query represents multiple tables, you must indicate how to apply the updates:

- If all updates are applied to a single database table, you can indicate the underlying table to update in an *OnGetTableName* event handler.
- If you need more control over applying updates, you can associate the query with an update object (*TUpdateSQL*). A provider automatically uses this update object to apply updates:
  - 1 Associate the update object with the query by setting the query's *UpdateObject* property to the *TUpdateSQL* object you are using.
  - 2 Set the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties to SQL statements that perform the appropriate updates for your query's data.

If you are using the BDE to cache updates, you must use an update object.

**Note** For more information on using update objects, see "Using update objects to update a dataset" on page 20-39.

## Using TStoredProc

---

*TStoredProc* represents a stored procedure. It implements all of the basic functionality introduced by *TDataSet*, as well as most of the special features typical of stored procedure-type datasets. Before looking at the unique features introduced by *TStoredProc*, you should familiarize yourself with the common database features described in "Understanding datasets," including the section on stored procedure-type datasets that starts on page 18-48.

Because *TStoredProc* is a BDE-enabled dataset, it must be associated with a database and a session. "Associating a dataset with database and session connections" on page 20-3 describes how you form these associations. Once the dataset is associated with a database and session, you can bind it to a particular stored procedure by setting the *StoredProcName* property.

*TStoredProc* differs from other stored procedure-type datasets in the following ways:

- It gives you greater control over how to bind parameters.
- It provides support for Oracle overloaded stored procedures.

## Binding parameters

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

*TStoredProc* lets you use the *ParamBindMode* property to specify how parameters should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters.

Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set *ParamBindMode* to *pbByNumber*.

**Tip** If you want to set *ParamBindMode* to *pbByNumber*, you need to specify the correct parameter types in the correct order. You can view a server's stored procedure source code in the SQL Explorer to determine the correct order and type of parameters to specify.

## Working with Oracle overloaded stored procedures

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's *Overload* property enables an application to specify the procedure to execute.

If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on.

**Note** Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.

## Connecting to databases with TDatabase

---

When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a *TDatabase* component. A database component represents the connection to a single database in the context of a BDE session.

*TDatabase* performs many of the same tasks as and shares many common properties, methods, and events with other database connection components. These commonalities are described in Chapter 17, "Connecting to databases."

In addition to the common properties, methods, and events, *TDatabase* introduces many BDE-specific features. These features are described in the following topics.

## Associating a database component with a session

All database components must be associated with a BDE session. Use the *SessionName*, establish this association. When you first create a database component at design time, *SessionName* is set to “Default”, meaning that it is associated with the default session component that is referenced by the global *Session* variable.

Multi-threaded or reentrant BDE applications may require more than one session. If you need to use multiple sessions, add *TSession* components for each session. Then, associate your dataset with a session component by setting the *SessionName* property to a session component’s *SessionName* property.

At runtime, you can access the session component with which the database is associated by reading the *Session* property. If *SessionName* is blank or “Default”, then the *Session* property references the same *TSession* instance referenced by the global *Session* variable. *Session* enables applications to access the properties, methods, and events of a database component’s parent session component without knowing the session’s actual name.

For more information about BDE sessions, see “Managing database sessions” on page 20-16.

If you are using an implicit database component, the session for that database component is the one specified by the dataset’s *SessionName* property.

## Understanding database and session component interactions

In general, session component properties provide global, default behaviors that apply to all implicit database components created at runtime. For example, the controlling session’s *KeepConnections* property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPassword* event for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box.

Session methods apply somewhat differently. *TSession* methods affect all database components, regardless of whether they are explicitly created or instantiated implicitly by a dataset. For example, the session method *DropConnections* closes all datasets belonging to a session’s database components, and then drops all database connections, even if the *KeepConnection* property for individual database components is *True*.

Database component methods apply only to the datasets associated with a given database component. For example, suppose the database component *Database1* is associated with the default session. *Database1.CloseDataSets()* closes only those datasets associated with *Database1*. Open datasets belonging to other database components within the default session remain open.

## Identifying the database

*AliasName* and *DriverName* are mutually exclusive properties that identify the database server to which the *TDatabase* component connects.

- *AliasName* specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

You create and edit BDE aliases using the Database Explorer or the BDE Administration utility. For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

- *DriverName* is the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local BDE alias for a database component using the *DatabaseName* property. If you specify *DriverName*, any value already assigned to *AliasName* is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in *AliasName*.

*DatabaseName* lets you provide your own name for a database connection. The name you supply is in addition to *AliasName* or *DriverName*, and is local to your application. *DatabaseName* can be a BDE alias, or, for Paradox and dBASE files, a fully-qualified path name. Like *AliasName*, *DatabaseName* appears in subsequent drop-down lists for dataset components to let you link them to database components.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

You can enter a *DatabaseName* in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the *Alias* property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the *DriverName* property, or you can choose from existing driver names in the drop-down list.

**Note** The Database Properties editor also lets you view and set BDE connection parameters, and set the states of the *LoginPrompt* and *KeepConnection* properties. For information on connection parameters, see “Setting BDE alias parameters” below. For information on *LoginPrompt*, see “Controlling server login” on page 17-4. For information on *KeepConnection* see “Opening a connection using TDatabase” on page 20-15.

### Setting BDE alias parameters

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer or BDE Administration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the Object Inspector to invoke the String List editor.
- Double-click a database component in a data module or form to invoke the Database Properties editor.

All of these methods edit the *Params* property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

At runtime, an application can set alias parameters only by editing the *Params* property directly. For more information about parameters specific to using SQL Links drivers with the BDE, see your online SQL Links help file.

## Opening a connection using TDatabase

As with all database connection components, to connect to a database using *TDatabase*, you set the *Connected* property to *True* or call the *Open* method. This process is described in “Connecting to a database server” on page 17-3. Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, the connection is dropped unless the database component’s *KeepConnection* property is *True*.

When you connect to a remote database server from an application, the application uses the BDE and the Borland SQL Links driver to establish the connection. (The BDE can also communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the *Params* property of a database component. For information about SQL Links parameters, see the online *SQL Links User’s Guide*. To edit the *Params* property, see “Setting BDE alias parameters” on page 20-14.

## Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver’s configuration options. In most cases, network protocol configuration is handled using a server’s client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server’s client-side connection properly configured?
- Are the DLLs for your connection and database drivers in the search path?
- If you are using TCP/IP:
  - Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?
  - Is the server’s IP address registered in the client’s HOSTS file?

- Is the Domain Name Services (DNS) properly configured?
- Can you ping the server?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

### Using ODBC

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- The BDE Administration utility.

To set up a BDE alias for an ODBC driver connection, use the BDE Administration utility. For more information, see the BDE Administration utility's online help file.

### Using database components in data modules

You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the *HandleShared* property of the database component to *True* to prevent global name space conflicts.

## Managing database sessions

---

An BDE-based application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application.

All BDE-based database applications automatically include a default session component, named *Session*, that encapsulates the default BDE session. When database components are added to the application, they are automatically associated with the default session (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are implicit (created by the session at runtime when you open a dataset that is not associated with a database component you create) or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

To use the default session, you need write no code unless your application must

- Explicitly activate or deactivate a session, enabling or disabling the session's databases' ability to open.
- Modify the properties of the session, such as specifying default properties for implicitly generated database components.

- Execute a session's methods, such as managing database connections (for example opening and closing database connections in response to user actions).
- Respond to session events, such as when the application attempts to access a password-protected Paradox or dBASE table.
- Set Paradox directory locations such as the *NetFileDir* property to access Paradox tables on a network and the *PrivateDir* property to a local hard drive to speed performance.
- Manage the BDE aliases that describe possible database connection configurations for databases and datasets that use the session.

Whether you add database components to an application at design time or create them dynamically at runtime, they are automatically associated with the default session unless you specifically assign them to a different session. If you open a dataset that is not associated with a database component, Delphi automatically

- Creates a database component for it at runtime.
- Associates the database component with the default session.
- Initializes some of the database component's key properties based on the default session's properties. Among the most important of these properties is *KeepConnections*, which determines when database connections are maintained or dropped by an application.

The default session provides a widely applicable set of defaults that can be used as is by most applications. You need only associate a database component with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions, where each thread has its own session.

Applications can create additional session components as needed. BDE-based database applications automatically include a session list component, named *Sessions*, that you can use to manage all of your session components. For more information about managing multiple sessions see, "Managing multiple sessions" on page 20-28.

You can safely place session components in data modules. If you put a data module that contains one or more session components into the Object Repository, however, make sure to set the *AutoSessionName* property to *True* to avoid namespace conflicts when users inherit from it.

## Activating a session

*Active* is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is



associated with a session is opened and there are currently no other open databases or datasets). Setting *Active* to *True* triggers a session's *OnStartup* event, registers the paradox directory locations with the BDE, and registers the *ConfigMode* property, which determines what BDE aliases are available within the session. You can write an *OnStartup* event handler to initialize the *NetFileDir*, *PrivateDir*, and *ConfigMode* properties before they are registered with the BDE, or to perform other specific session start-up activities. For information about the *NetFileDir* and *PrivateDir* properties, see "Specifying Paradox directory locations" on page 20-24. For information about *ConfigMode*, see "Working with BDE aliases" on page 20-24.

Once a session is active, you can open its database connections by calling the *OpenDatabase* method.

For session components you place in a data module or form, setting *Active* to *False* when there are open databases or datasets closes them. At runtime, closing databases and datasets may trigger events associated with them.

**Note** You cannot set *Active* to *False* for the default session at design time. While you can close the default session at runtime, it is not recommended.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following single line of code closes all open databases and datasets for a session:

```
Session1.Close;
```

This code sets *Session1*'s *Active* property to *False*. When a session's *Active* property is *False*, any subsequent attempt by the application to open a database or dataset resets *Active* to *True* and calls the session's *OnStartup* event handler if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
Session1.Open;
```

**Note** If a session is active you can also open and close individual database connections. For more information, see "Closing database connections" on page 20-19.

## Specifying default database connection behavior

*KeepConnections* provides the default value for the *KeepConnection* property of implicit database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed.

**Note** Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a database component always overrides the *KeepConnections* property of the session. For more information about controlling individual database connections within a session, see "Managing database connections" on page 20-19.

*KeepConnections* should be set to *True* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces

network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or reestablishes a connection, it incurs the overhead of attaching and detaching the database.

**Note** Even when *KeepConnections* is *True* for a session, you can close and free inactive database connections for all implicit database components by calling the *DropConnections* method. For more information about *DropConnections*, see “Dropping inactive database connections” on page 20-20.

## Managing database connections

You can use a session component to manage the database connections within it. The session component includes properties and methods you can use to

- Open database connections.
- Close database connections.
- Close and free all inactive temporary database connections.
- Locate specific database connections.
- Iterate through all open database connections.

### Opening database connections

To open a database connection within a session, call the *OpenDatabase* method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:

```
var
    DBDemosDatabase: TDatabase;
begin
    DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
    ...
```

*OpenDatabase* activates the session if it is not already active, and then checks if the specified database name matches the *DatabaseName* property of any database components for the session. If the name does not match an existing database component, *OpenDatabase* creates a temporary database component using the specified name. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server. Each call to *OpenDatabase* increments a reference count for the database by 1. As long as this reference count remains greater than 0, the database is open.

### Closing database connections

To close an individual database connection, call the *CloseDatabase* method. When you call *CloseDatabase*, the reference count for the database, which is incremented when you call *OpenDatabase*, is decremented by 1. When the reference count for a database is 0, the database is closed. *CloseDatabase* takes one parameter, the database to close. If you opened the database using the *OpenDatabase* method, this parameter can be set to the return value of *OpenDatabase*.

```
Session.CloseDatabase(DBDemosDatabase);
```

If the specified database name is associated with a temporary (implicit) database component, and the session's *KeepConnections* property is *False*, the database component is freed, effectively closing the connection.

**Note** If *KeepConnections* is *False* temporary database components are closed and freed automatically when the last dataset associated with the database component is closed. An application can always call *CloseDatabase* prior to that time to force closure. To free temporary database components when *KeepConnections* is *True*, call the database component's *Close* method, and then call the session's *DropConnections* method.

**Note** Calling *CloseDatabase* for a persistent database component does not actually close the connection. To close the connection, call the database component's *Close* method directly.

There are two ways to close all database connections within the session:

- Set the *Active* property for the session to *False*.
- Call the *Close* method for the session.

When you set *Active* to *False*, Delphi automatically calls the *Close* method. *Close* disconnects from all active databases by freeing temporary database components and calling each persistent database component's *Close* method. Finally, *Close* sets the session's BDE handle to **nil**.

### Dropping inactive database connections

If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the *DropConnections* method. For example, the following code frees all inactive, temporary database components for the default session:

```
Session.DropConnections;
```

Temporary database components for which one or more datasets are active are not dropped or freed by this call. To free these components, call *Close*.

### Searching for a database connection

Use a session's *FindDatabase* method to determine whether a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, it can also be a fully-qualified path name.

*FindDatabase* returns the database component if it finds a match. Otherwise it returns **nil**.

The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

```
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
```

```

if (DB = nil) then                                { database doesn't exist for session so,}
    DB := Session.OpenDatabase('DBDEMOS');      { create and open it}
if Assigned(DB) and DB.Connected then begin
    DB.StartTransaction;
    ...
end;
end;

```

### Iterating through a session's database components

You can use two session component properties, *Databases* and *DatabaseCount*, to cycle through all the active database components associated with a session.

*Databases* is an array of all currently active database components associated with a session. *DatabaseCount* is the number of databases in that array. As connections are opened or closed during a session's life-span, the values of *Databases* and *DatabaseCount* change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```

var
    MaxDbCount: Integer;
begin
    with Session do
        if (DatabaseCount > 0) then
            for MaxDbCount := 0 to (DatabaseCount - 1) do
                Databases[MaxDbCount].KeepConnection := True;
end;

```

### Working with password-protected Paradox and dBASE tables

A session component can store passwords for password-protected Paradox and dBASE tables. Once you add a password to the session, your application can open tables protected by that password. Once you remove the password from the session, your application can't open tables that use the password until you add it again.

#### Using the AddPassword method

The *AddPassword* method provides an optional way for an application to provide a password for a session prior to opening an encrypted Paradox or dBASE table that requires a password for access. If you do not add the password to the session, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

*AddPassword* takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords (one at a time) to access tables protected with different passwords.

```

var
  Passwr: String;
begin
  Passwr := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwr);
  try
    Table1.Open;
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;

```

**Note** Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form.

The Add button of the *PasswordDialog* function dialog has the same effect as the *AddPassword* method.

```

if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;

```

### Using the RemovePassword and RemoveAllPasswords methods

*RemovePassword* deletes a previously added password from memory.

*RemovePassword* takes one parameter, a string containing the password to delete.

```
Session.RemovePassword('secret');
```

*RemoveAllPasswords* deletes all previously added passwords from memory.

```
Session.RemoveAllPasswords;
```

### Using the GetPassword method and OnPassword event

The *OnPassword* event allows you to control how your application supplies passwords for Paradox and dBASE tables when they are required. Provide a handler for the *OnPassword* event if you want to override the default password handling behavior. If you do not provide a handler, Delphi presents a default dialog for entering a password and no special behavior is provided—the table open attempt either succeeds or an exception is raised.

If you provide a handler for the *OnPassword* event, do two things in the event handler: call the *AddPassword* method and set the event handler's *Continue* parameter to *True*. The *AddPassword* method passes a string to the session to be used as a password for the table. The *Continue* parameter indicates to Delphi that no further password prompting need be done for this table open attempt. The default value for *Continue* is *False*, and so requires explicitly setting it to *True*. If *Continue* is *False* after the event handler has finished executing, an *OnPassword* event fires again—even if a valid password has been passed using *AddPassword*. If *Continue* is *True* after execution of the event handler and the string passed with *AddPassword* is not the valid password, the table open attempt fails and an exception is raised.

*OnPassword* can be triggered by two circumstances. The first is an attempt to open a password-protected table (dBASE or Paradox) when a valid password has not already been supplied to the session. (If a valid password for that table has already been supplied, the *OnPassword* event does not occur.)

The other circumstance is a call to the *GetPassword* method. *GetPassword* either generates an *OnPassword* event, or, if the session does not have an *OnPassword* event handler, displays a default password dialog. It returns *True* if the *OnPassword* event handler or default dialog added a password to the session, and *False* if no entry at all was made.

In the following example, the *Password* method is designated as the *OnPassword* event handler for the default session by assigning it to the global *Session* object's *OnPassword* property.

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Session.OnPassword := Password;
end;

```

In the *Password* method, the *InputBox* function prompts the user for a password. The *AddPassword* method then programmatically supplies the password entered in the dialog to the session.

```

procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
    Passwrđ: String;
begin
    Passwrđ := InputBox('Enter password', 'Password:', '');
    Continue := (Passwrđ > '');
    Session.AddPassword(Passwrđ);
end;

```

The *OnPassword* event (and thus the *Password* event handler) is triggered by an attempt to open a password-protected table, as demonstrated below. Even though the user is prompted for a password in the handler for the *OnPassword* event, the table open attempt can still fail if they enter an invalid password or something else goes wrong.

```

procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
    CRLF = #13 + #10;
begin
    try
        Table1.Open; { this line triggers the OnPassword event }
    except
        on E:Exception do begin { exception if cannot open table }
            ShowMessage('Error!' + CRLF + { display error explaining what happened }
                E.Message + CRLF +
                'Terminating application...');
            Application.Terminate; { end the application }
        end;
    end;
end;

```

## Specifying Paradox directory locations

Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables.

*NetFileDir* specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server). Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so be sure to validate the new value.

At design time, you can specify a value for *NetFileDir* in the Object Inspector. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
Session.NetFileDir := ExtractFilePath(Application.EXENAME);
```

**Note** *NetFileDir* can only be changed when an application does not have any open Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

*PrivateDir* specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements. If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you can improve application performance at runtime by setting *PrivateDir* to a user's local hard drive before opening the database.

**Note** Do not set *PrivateDir* at design time and then open the session in the IDE. Doing so generates a Directory is busy error when running your application from the IDE.

The following code changes the setting of the default session's *PrivateDir* property to a user's C:\TEMP directory:

```
Session.PrivateDir := 'C:\TEMP';
```

**Important** Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

## Working with BDE aliases

Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables). A session can create, modify, and delete aliases during its lifetime.

The *AddAlias* method creates a new BDE alias for an SQL database server. *AddAlias* takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For example, the following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
```

```

try
  with AliasParams do begin
    Add('OPEN MODE=READ');
    Add('USER NAME=TOMSTOPPARD');
    Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
  end;
  Session.AddAlias('CATS', 'INTRBASE', AliasParams);
  ...
finally
  AliasParams.Free;
end;
end;

```

*AddStandardAlias* creates a new BDE alias for Paradox, dBASE, or ASCII tables. *AddStandardAlias* takes three string parameters: the name for the alias, the fully-qualified path to the Paradox or dBASE table to access, and the name of the default driver to use when attempting to open a table that does not have an extension. For example, the following statement uses *AddStandardAlias* to create a new alias for accessing a Paradox table:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

When you add an alias to a session, the BDE stores a copy of the alias in memory, where it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. *ConfigMode* is a set that describes which types of aliases can be used by the databases in the session. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent*, *cfmSession*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session (*cfmSession*), all aliases in the BDE configuration file on a user's system (*cfmPersistent*), and all aliases that the BDE maintains in memory (*cfmVirtual*). You can change *ConfigMode* to restrict what BDE aliases the databases in a session can use. For example, setting *ConfigMode* to *cfmSession* restricts a session's view of aliases to those created within the session. All other aliases in the BDE configuration file and in memory are not available.

To make a newly created alias available to all sessions and to other applications, use the session's *SaveConfigFile* method. *SaveConfigFile* writes aliases in memory to the BDE configuration file where they can be read and used by other BDE-enabled applications.

After you create an alias, you can make changes to its parameters by calling *ModifyAlias*. *ModifyAlias* takes two parameters: the name of the alias to modify and a string list containing the parameters to change and their values. For example, the following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```

var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...

```



To delete an alias previously created in a session, call the *DeleteAlias* method. *DeleteAlias* takes one parameter, the name of the alias to delete. *DeleteAlias* makes an alias unavailable to the session.

**Note** *DeleteAlias* does not remove an alias from the BDE configuration file if the alias was written to the file by a previous call to *SaveConfigFile*. To remove the alias from the configuration file after calling *DeleteAlias*, call *SaveConfigFile* again.

Session components provide five methods for retrieving information about a BDE aliases, including parameter information and driver information. They are:

- *GetAliasNames*, to list the aliases to which a session has access.
- *GetAliasParams*, to list the parameters for a specified alias.
- *GetAliasDriverName*, to return the name of the BDE driver used by the alias.
- *GetDriverNames*, to return a list of all BDE drivers available to the session.
- *GetDriverParams*, to return driver parameters for a specified driver.

For more information about using a session's informational methods, see "Retrieving information about a session" below. For more information about BDE aliases and the SQL Links drivers with which they work, see the BDE online help, BDE32.HLP.

## Retrieving information about a session

You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method retrieves the names of tables associated with a specific database component used by the session. Table 20.4 summarizes the informational methods to a session component:

**Table 20.4** Database-related informational methods for session components

<i>Method</i>	<i>Purpose</i>
<i>GetAliasDriverName</i>	Retrieves the BDE driver for a specified alias of a database.
<i>GetAliasNames</i>	Retrieves the list of BDE aliases for a database.
<i>GetAliasParams</i>	Retrieves the list of parameters for a specified BDE alias of a database.
<i>GetConfigParams</i>	Retrieves configuration information from the BDE configuration file.
<i>GetDatabaseNames</i>	Retrieves the list of BDE aliases and the names of any <i>TDatabase</i> components currently in use.
<i>GetDriverNames</i>	Retrieves the names of all currently installed BDE drivers.
<i>GetDriverParams</i>	Retrieves the list of parameters for a specified BDE driver.
<i>GetStoredProcNames</i>	Retrieves the names of all stored procedures for a specified database.
<i>GetTableNames</i>	Retrieves the names of all tables matching a specified pattern for a specified database.
<i>GetFieldNames</i>	Retrieves the names of all fields in a specified table in a specified database.

Except for *GetAliasDriverName*, these methods return a set of values into a string list declared and maintained by your application. (*GetAliasDriverName* returns a single string, the name of the current BDE driver for a particular database component used by the session.)

For example, the following code retrieves the names of all database components and aliases known to the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

## Creating additional sessions

You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the Object Inspector, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime.

**Note** Creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.

To enable dynamic creation of a session component at runtime, follow these steps:

- 1 Declare a *TSession* variable.
- 2 Instantiate a new session by calling the *Create* method. The constructor sets up an empty list of database components for the session, sets the *KeepConnections* property to *True*, and adds the session to the list of sessions maintained by the application's session list component.
- 3 Set the *SessionName* property for the new session to a unique name. This property is used to associate database components with the session. For more information about the *SessionName* property, see "Naming a session" on page 20-28.
- 4 Activate the session and optionally adjust its properties.

You can also create and open sessions using the *OpenSession* method of *TSessionList*. Using *OpenSession* is safer than calling *Create*, because *OpenSession* only creates a session if it does not already exist. For information about *OpenSession*, see "Managing multiple sessions" on page 20-28.

The following code creates a new session component, assigns it a name, and opens the session for database operations that follow (not shown here). After use, it is destroyed with a call to the *Free* method.

**Note** Never delete the default session.

```
var
  SecondSession: TSession;
begin
  SecondSession := TSession.Create(Form1);
  with SecondSession do
    try
```

```

    SessionName := 'SecondSession';
    KeepConnections := False;
    Open;
    ...
  finally
    SecondSession.Free;
  end;
end;

```

## Naming a session

A session's *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is "Default." For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the *SessionName* of a session component you create.

The following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to "InterBaseSession," activate the session, and associate an existing database component *Database1* with that session:

```

var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;

```

## Managing multiple sessions

If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The BDE page on the Component palette contains a session component that you can place in a data module or on a form at design time.

**Important** When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the *OpenSession* method of the global *Sessions* object at runtime.

*OpenSession* requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

*Sessions* is a variable of type *TSessionList* that is automatically instantiated for BDE-based database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. Table 20.5 summarizes the properties and methods of the *TSessionList* component:

**Table 20.5** TSessionList properties and methods

Property or Method	Purpose
<i>Count</i>	Returns the number of sessions, both active and inactive, in the session list.
<i>FindSession</i>	Searches for a session with a specified name and returns a pointer to it, or <b>nil</b> if there is no session with the specified name. If passed a blank session name, <i>FindSession</i> returns a pointer to the default session, <i>Session</i> .
<i>GetSessionNames</i>	Populates a string list with the names of all currently instantiated session components. This procedure always adds at least one string, "Default" for the default session.
<i>List</i>	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
<i>OpenSession</i>	Creates and activates a new session or reactivates an existing session for a specified session name.
<i>Sessions</i>	Accesses the session list by ordinal value.

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

If you are replicating a data module that contains a session in a multi-threaded application, where each thread contains its own copy of the data module, you can use the *AutoSessionName* property to make sure that all datasets in the data module use the correct session. Setting *AutoSessionName* to *True* causes the session to generate its own unique name dynamically when it is created at runtime. It then assigns this name to every dataset in the data module, overriding any explicitly set session names. This ensures that each thread has its own session, and each dataset uses the session in its own data module.

## Using transactions with the BDE

---

By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly.

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the database component to control transactions. The main advantage to using the methods and properties of a database component is that it provides a clean, portable application that is not dependent on a particular database or server. This type of transaction control is supported by all database connection components, and described in “Managing transactions” on page 17-5
- Use passthrough SQL in a query component to pass SQL statements directly to remote SQL or ODBC servers. The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server’s transaction management model, see your database server documentation. For more information about using passthrough SQL, see “Using passthrough SQL” below.

When working with local databases, you can only use the database component to create explicit transactions (local databases do not support passthrough SQL). However, there are limitations to using local transactions. For more information on using local transactions, see “Using local transactions” on page 20-31.

**Note** You can minimize the number of transactions you need by caching updates. For more information about cached updates, see “Using a client dataset to cache updates” on page 23-15 and “Using the BDE to cache updates” on page 20-32.

### Using passthrough SQL

---

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To use passthrough SQL to control a transaction, you must

- Install the proper SQL Links drivers. If you chose the “Typical” installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol. See your network administrator for more information.
- Have access to a database on a remote server.
- Set `SQLPASSTHRU MODE` to `NOT SHARED` using the SQL Explorer. `SQLPASSTHRU MODE` specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, `SQLPASSTHRU MODE` is set to `SHARED AUTOCOMMIT`. However, you can’t share database connections when using transaction control statements. For more information about `SQLPASSTHRU` modes, see the help file for the BDE Administration utility.

**Note** When `SQLPASSTHRU MODE` is `NOT SHARED`, you must use separate database components for datasets that pass SQL transaction statements to the server and datasets that do not.

## Using local transactions

---

The BDE supports local transactions against Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

**Note** When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Local transactions are more limited than transactions against SQL servers or ODBC drivers. In particular, the following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- Transactions cannot be run against temporary tables.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Only a limited number of records can be locked and modified. With Paradox tables, you are limited to 255 records. With dBASE the limit is 100.

- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
  - Several tables are open.
  - The cursor is closed on a table to which no changes were made.

## Using the BDE to cache updates

The recommended approach for caching updates is to use a client dataset (*TBDEClientDataSet*) or to connect the BDE-dataset to a client dataset using a dataset provider. The advantages of using a client dataset are discussed in “Using a client dataset to cache updates” on page 23-15.

For simple cases, however, you may choose to use the BDE to cache updates instead. BDE-enabled datasets and *TDatabase* components provide built-in properties, methods, and events for handling cached updates. Most of these correspond directly to the properties, methods, and events that you use with client datasets and dataset providers when using a client dataset to cache updates. The following table lists these properties, events, and methods and the corresponding properties, methods and events on *TBDEClientDataSet*:

**Table 20.6** Properties, methods, and events for cached updates

On BDE-enabled datasets (or TDatabase)	On TBDEClientDataSet	Purpose
<i>CachedUpdates</i>	Not needed for client datasets, which always cache updates.	Determines whether cached updates are in effect for the dataset.
<i>UpdateObject</i>	Use a <i>BeforeUpdateRecord</i> event handler, or, if using <i>TClientDataSet</i> , use the <i>UpdateObject</i> property on the BDE-enabled source dataset.	Specifies the update object for updating read-only datasets.
<i>UpdatesPending</i>	<i>ChangeCount</i>	Indicates whether the local cache contains updated records that need to be applied to the database.
<i>UpdateRecordTypes</i>	<i>StatusFilter</i>	Indicates the kind of updated records to make visible when applying cached updates.
<i>UpdateStatus</i>	<i>UpdateStatus</i>	Indicates if a record is unchanged, modified, inserted, or deleted.
<i>OnUpdateError</i>	<i>OnReconcileError</i>	An event for handling update errors on a record-by-record basis.
<i>OnUpdateRecord</i>	<i>BeforeUpdateRecord</i>	An event for processing updates on a record-by-record basis.
<i>ApplyUpdates</i> <i>ApplyUpdates</i> (database)	<i>ApplyUpdates</i>	Applies records in the local cache to the database.

**Table 20.6** Properties, methods, and events for cached updates (continued)

On BDE-enabled datasets (or TDatabase)	On TBDEClientDataSet	Purpose
<i>CancelUpdates</i>	<i>CancelUpdates</i>	Removes all pending updates from the local cache without applying them.
<i>CommitUpdates</i>	<i>Reconcile</i>	Clears the update cache following successful application of updates.
<i>FetchAll</i>	<i>GetNextPacket</i> (and <i>PacketRecords</i> )	Copies database records to the local cache for editing and updating.
<i>RevertRecord</i>	<i>RevertRecord</i>	Undoes updates to the current record if updates are not yet applied.

For an overview of the cached update process, see “Overview of using cached updates” on page 23-16.

**Note** Even if you are using a client dataset to cache updates, you may want to read the section about update objects on page 20-39. You can use update objects in the *BeforeUpdateRecord* event handler of *TBDEClientDataSet* or *TDataSetProvider* to apply updates from stored procedures or multi-table queries.

## Enabling BDE-based cached updates

To use the BDE for cached updates, the BDE-enabled dataset must indicate that it should cache updates. This is specified by setting the *CachedUpdates* property to *True*. When you enable cached updates, a copy of all records is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the application applies those changes to the database server. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

The dataset caches all updates until you set *CachedUpdates* to *False*. Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory. Canceling the updates by calling *CancelUpdates* removes all the changes currently in the cache, but does not stop the dataset from caching any subsequent changes.

**Note** If you disable cached updates by setting *CachedUpdates* to *False*, any pending changes that you have not yet applied are discarded without notification. To prevent losing changes, test the *UpdatesPending* property before disabling cached updates.

## Applying BDE-based cached updates

Applying updates is a two-phase process that should occur in the context of a database component’s transaction so that your application can recover gracefully from errors. For information about transaction handling with database components, see “Managing transactions” on page 17-5.



When applying updates under database transaction control, the following events take place:

- 1 A database transaction starts.
- 2 Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.
- 3 The transaction is committed if writes are successful or rolled back if they are not:
  - If the database write is successful:
    - Database changes are committed, ending the database transaction.
    - Cached updates are committed, clearing the internal cache buffer (phase 2).
  - If the database write is unsuccessful:
    - Database changes are rolled back, ending the database transaction.
    - Cached updates are not committed, remaining intact in the internal cache.

For information about creating and using an *OnUpdateRecord* event handler, see “Creating an *OnUpdateRecord* event handler” on page 20-36. For information about handling update errors that occur when applying cached updates, see “Handling cached update errors” on page 20-37.

**Note** Applying cached updates is particularly tricky when you are working with multiple datasets linked in a master/detail relationship because the order in which you apply updates to each dataset is significant. Usually, you must update master tables before detail tables, except when handling deleted records, where this order must be reversed. Because of this difficulty, it is strongly recommended that you use client datasets when caching updates in a master/detail form. Client datasets automatically handle all ordering issues with master/detail relationships.

There are two ways to apply BDE-based updates:

- You can apply updates using a database component by calling its *ApplyUpdates* method. This method is the simplest approach, because the database handles all details of managing a transaction for the update process and of clearing the dataset’s cache when updating is complete.
- You can apply updates for a single dataset by calling the dataset’s *ApplyUpdates* and *CommitUpdates* methods. When applying updates at the dataset level you must explicitly code the transaction that wraps the update process as well as explicitly call *CommitUpdates* to commit updates from the cache.

**Important** To apply updates from a stored procedure or an SQL query that does not return a live result set, you must use *TUpdateSQL* to specify how to perform updates. For updates to joins (queries involving two or more tables), you must provide one *TUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. See “Using update objects to update a dataset” on page 20-39 for details.

## Applying cached updates using a database

To apply cached updates to one or more datasets in the context of a database connection, call the database component's *ApplyUpdates* method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```

procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
    // for local databases such as Paradox, dBASE, and FoxPro
    // set TransIsolation to DirtyRead
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
        Database1.TransIsolation := tiDirtyRead;
    Database1.ApplyUpdates([CustomersQuery]);
end;

```

The above sequence writes cached updates to the database in the context of an automatically-generated transaction. If successful, it commits the transaction and then commits the cached updates. If unsuccessful, it rolls back the transaction and leaves the update cache unchanged. In this latter case, you should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see "Handling cached update errors" on page 20-37.

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TDBDataSet*. For example, the following code applies updates for two queries:

```

if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);

```

## Applying cached updates with dataset component methods

You can apply updates for individual BDE-enabled datasets directly using the dataset's *ApplyUpdates* and *CommitUpdates* methods. Each of these methods encapsulate one phase of the update process:

- 1 *ApplyUpdates* writes cached changes to a database (phase 1).
- 2 *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset:

```

procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
    Database1.StartTransaction;
    try
        if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
            Database1.TransIsolation := tiDirtyRead;
        CustomerQuery.ApplyUpdates;           { try to write the updates to the database }
        Database1.Commit;                     { on success, commit the changes }
    except
        Database1.Rollback;                   { on failure, undo any changes }
    end;

```

```

        raise;                { raise the exception again to prevent a call to CommitUpdates }
    end;
    CustomerQuery.CommitUpdates;    { on success, clear the internal cache }
end;

```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The *raise* statement inside the *try...except* block *reraises* the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

## Creating an OnUpdateRecord event handler

When a BDE-enabled dataset applies its cached updates, it iterates through the changes recorded in its cache, attempting to apply them to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, special parameter substitution, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```

procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
    UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    { perform updates here... }
end;

```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update that needs to be performed for the current record. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. If you are using an update object, you need to pass this parameter to the update object when applying the update. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update.

The *UpdateAction* parameter indicates whether you applied the update. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. If your event handler successfully applies the update, change this parameter to *uaApplied* before exiting. If you decide not to update the current record, change the value to *uaSkip* to preserve unapplied changes in the cache. If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted and an exception is raised. You can suppress the error message (raising a silent exception) by changing *UpdateAction* to *uaAbort*.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. *OldValue* gives the original field value that was fetched from the database. It can be useful in locating the database record to update. *NewValue* is the edited value in the update you are trying to apply.

**Important** An *OnUpdateRecord* event handler, like an *OnUpdateError* or *OnCalcFields* event handler, should never call any methods that change the current record in a dataset.

The following example illustrates how to use these parameters and properties. It uses a *TTable* component named *UpdateTable* to apply updates. In practice, it is easier to use an update object, but using a table illustrates the possibilities more clearly.

```

procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
    UpdateAction := uaApplied;
end;

```

## Handling cached update errors

The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports any errors. The dataset component's *OnUpdateError* event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Here is the skeleton code for an *OnUpdateError* event handler:

```

procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;

```

*DataSet* references the dataset to which updates are applied. You can use this dataset to access new and old values during error handling. The original values for fields in each record are stored in a read-only *TField* property called *OldValue*. Changed values are stored in the analogous *TField* property *NewValue*. These values provide the only way to inspect and change update values in the event handler.

**Warning** Do not call any dataset methods that change the current record (such as *Next* and *Prior*). Doing so causes the event handler to enter an endless loop.

The *E* parameter is usually of type *EDBEngineError*. From this exception type, you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
ErrorLabel.Caption := E.Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it.

The *UpdateKind* parameter describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

**Table 20.7** UpdateKind values

Value	Meaning
<i>ukModify</i>	Editing an existing record caused an error.
<i>ukInsert</i>	Inserting a new record caused an error.
<i>ukDelete</i>	Deleting an existing record caused an error.

*UpdateAction* tells the BDE how to proceed with the update process when your event handler exits. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler:

- If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.
- When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.
- Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

The following code shows an *OnUpdateError* event handler that checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip           { key violation, just skip this record }
    else
      UpdateAction := uaAbort;         { don't know what's wrong, abort the update }
  end;
```

**Note** If an error occurs during the application of cached updates, an exception is *raised* and an error message displayed. Unless the *ApplyUpdates* is called from within a *try...except* construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

## Using update objects to update a dataset

---

When the BDE-enabled dataset represents a stored procedure or a query that is not “live”, it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates. Whether you are using the BDE or a client dataset to cache updates, you can handle these problem datasets by using an update object:

- 1 If you are using a client dataset, use an external provider component with *TClientDataSet* rather than *TBDEClientDataSet*. This is so you can set the *UpdateObject* property of the BDE-enabled source dataset (step 3).
- 2 Add a *TUpdateSQL* component to the same data module as the BDE-enabled dataset.
- 3 Set the BDE-enabled dataset component’s *UpdateObject* property to the *TUpdateSQL* component in the data module.
- 4 Specify the SQL statements needed to perform updates using the update object’s *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. You can use the Update SQL editor to help you compose these statements.
- 5 Close the dataset.
- 6 Set the dataset component’s *CachedUpdates* property to *True* or link the dataset to the client dataset using a dataset provider.
- 7 Reopen the dataset.

**Note** Sometimes, you need to use multiple update objects. For example, when updating a multi-table join or a stored procedure that represents data from multiple datasets, you must provide one *TUpdateSQL* object for each table you want to update. When using multiple update objects, you can’t simply associate the update object with the dataset by setting the *UpdateObject* property. Instead, you must manually call the update object from an *OnUpdateRecord* event handler (when using the BDE to cache updates) or a *BeforeUpdateRecord* event handler (when using a client dataset).

The update object actually encapsulates three *TQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties for which you supply SQL statements:

- *ModifySQL* specifies the UPDATE statement.
- *InsertSQL* specifies the INSERT statement.
- *DeleteSQL* specifies the DELETE statement.

At runtime, when the update component is used to apply updates, it:

- 1 Selects an SQL statement to execute based on whether the current record is modified, inserted, or deleted.
- 2 Provides parameter values to the SQL statement.
- 3 Prepares and executes the SQL statement to perform the specified update.

## Creating SQL statements for update components

To update a record in an associated dataset, an update object uses one of three SQL statements. Each update object can only update a single table, so the object's update statements must each reference the same base table.

The three SQL statements delete, insert, and modify records cached for update. You must provide these statements as update object's *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. You can provide these values at design time or at runtime. For example, the following code specifies a value for the *DeleteSQL* property at runtime:

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

At design time, you can use the Update SQL editor to help you compose the SQL statements that apply updates.

Update objects provide automatic parameter binding for parameters that reference the dataset's original and updated field values. Typically, therefore, you insert parameters with specially formatted names when you compose the SQL statements. For information on using these parameters, see "Understanding parameter substitution in update SQL statements" on page 20-41.

## Using the Update SQL editor

To create the SQL statements for an update component,

- 1 Using the Object Inspector, select the name of the update object from the drop-down list for the dataset's *UpdateObject* property. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.
- 2 Right-click the update object and select UpdateSQL Editor from the context menu. This displays the Update SQL editor. The editor creates SQL statements for the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. For Paradox, dBASE, and FoxPro the columns you specify here must correspond to an existing index, but this is not a requirement for remote SQL databases. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. In most cases you will want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

**Important** Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather than using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

### Understanding parameter substitution in update SQL statements

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string



“OLD\_”, then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the “:OLD\_FieldName” syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer’s last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with “Smith” as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

**Note** If you create SQL statements that contain parameters that do not refer the edited or original field values, the update object does not know how to bind their values. You can, however, do this manually, using the update object’s *Query* property. See “Using an update component’s Query property” on page 20-46 for details.

### Composing update SQL statements

At design time, you can use the Update SQL editor to write the SQL statements for the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. If you do not use the Update SQL editor, or if you want to modify the generated statements, you should keep in mind the following guidelines when writing statements to delete, insert, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with “OLD\_”, the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some table types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update

fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
      ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with "OLD\_". In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD\_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

## Using multiple update objects

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with a dataset by setting its *DataSet* property to the name of the dataset.

**Tip** When using multiple update objects, you can use *TBDEClientDataSet* instead of *TClientDataSet* with an external provider. This is because you do not need to set the source dataset's *UpdateObject* property.

The *DataSet* property for update objects is not available at design time in the Object Inspector. You can only set this property at runtime.

```
UpdateSQL1.DataSet := Query1;
```

The update object uses this dataset to obtain original and updated field values for parameter substitution and, if it is a BDE-enabled dataset, to identify the session and database to use when applying the updates. So that parameter substitution will work correctly, the update object's *DataSet* property must be the dataset that contains the updated field values. When using the BDE-enabled dataset to cache updates, this is the BDE-enabled dataset itself. When using a client dataset, this is a client dataset that is provided as a parameter to the *BeforeUpdateRecord* event handler.

When the update object has not been assigned to the dataset's *UpdateObject* property, its SQL statements are not automatically executed when you call *ApplyUpdates*. To update records, you must manually call the update object from an *OnUpdateRecord* event handler (when using the BDE to cache updates) or a *BeforeUpdateRecord* event handler (when using a client dataset). In the event handler, the minimum actions you need to take are

- If you are using a client dataset to cache updates, you must be sure that the updates object's *DatabaseName* and *SessionName* properties are set to the *DatabaseName* and *SessionName* properties of the source dataset.
- The event handler must call the update object's *ExecSQL* or *Apply* method. This invokes the update object for each record that requires updating. For more information about executing update statements, see "Executing the SQL statements" below.
- Set the event handler's *UpdateAction* parameter to *uaApplied* (*OnUpdateRecord*) or the *Applied* parameter to *True* (*BeforeUpdateRecord*).

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

**Warning** If you call an update object's *ExecSQL* or *Apply* method in an *OnUpdateRecord* event handler, be sure that you do not set the dataset's *UpdateObject* property to that update object. Otherwise, this will result in a second attempt to apply each record's update.

## Executing the SQL statements

When you use multiple update objects, you do not associate the update objects with a dataset by setting its *UpdateObject* property. As a result, the appropriate statements are not automatically executed when you apply updates. Instead, you must explicitly invoke the update object in code.

There are two ways to invoke the update object. Which way you choose depends on whether the SQL statement uses parameters to represent field values:

- If the SQL statement to execute uses parameters, call the *Apply* method.

- If the SQL statement to execute does not use parameters, it is more efficient to call the *ExecSQL* method.

**Note** If the SQL statement uses parameters other than the built-in types (for the original and updated field values), you must manually supply parameter values instead of relying on the parameter substitution provided by the *Apply* method. See “Using an update component’s Query property” on page 20-46 for information on manually providing parameter values.

For information about the default parameter substitution for parameters in an update object’s SQL statements, see “Understanding parameter substitution in update SQL statements” on page 20-41.

### Calling the Apply method

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

- 1 Initial and edited field values for the record are bound to parameters in the appropriate SQL statement.
- 2 The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. The *Apply* method is most often called from within a handler for the dataset’s *OnUpdateRecord* event or from a provider’s *BeforeUpdateRecord* event handler.

**Warning** If you use the dataset’s *UpdateObject* property to associate dataset and update object, *Apply* is called automatically. In that case, do not call *Apply* in an *OnUpdateRecord* event handler as this will result in a second attempt to apply the current record’s update.

*OnUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *Apply* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```

procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
    DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
    with UpdateSQL1 do
        begin
            DataSet := DeltaDS;
            DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
            SessionName := (SourceDS as TDBDataSet).SessionName;
            Apply(UpdateKind);
            Applied := True;
        end;
    end;
end;

```

### Calling the ExecSQL method

The *ExecSQL* method for an update component manually applies updates for the current record. Unlike the *Apply* method, *ExecSQL* does not bind parameters in the SQL statement before executing it. The *ExecSQL* method is most often called from

within a handler for the *OnUpdateRecord* event (when using the BDE) or the *BeforeUpdateRecord* event (when using a client dataset).

Because *ExecSQL* does not bind parameter values, it is used primarily when the update object's SQL statements do not include parameters. You can use *Apply* instead, even when there are no parameters, but *ExecSQL* is more efficient because it does not check for parameters.

If the SQL statements include parameters, you can still call *ExecSQL*, but only after explicitly binding parameters. If you are using the BDE to cache updates, you can explicitly bind parameters by setting the update object's *DataSet* property and then calling its *SetParams* method. When using a client dataset to cache updates, you must supply parameters to the underlying query object maintained by *TUpdateSQL*. For information on how to do this, see "Using an update component's Query property" on page 20-46.

**Warning** If you use the dataset's *UpdateObject* property to associate dataset and update object, *ExecSQL* is called automatically. In that case, do not call *ExecSQL* in an *OnUpdateRecord* or *BeforeUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

*OnUpdateRecord* and *BeforeUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *ExecSQL* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```

procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
    DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
    with UpdateSQL1 do
        begin
            DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
            SessionName := (SourceDS as TDBDataSet).SessionName;
            ExecSQL(UpdateKind);
            Applied := True;
        end;
    end;

```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

### Using an update component's Query property

The *Query* property of an update component provides access to the query components that implement its *DeleteSQL*, *InsertSQL*, and *ModifySQL* statements. In most applications, there is no need to access these query components directly: you can use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to specify the statements these queries execute, and execute them by calling the update object's *Apply* or *ExecSQL* method. There are times, however, when you may need to directly manipulate the query component. In particular, the *Query* property is useful when you want to supply your own values for parameters in the SQL statements rather than relying on the update object's automatic parameter binding to old and new field values.

**Note** The *Query* property is only accessible at runtime.

The *Query* property is indexed on a *TUpdateKind* value:

- Using an index of *ukModify* accesses the query that updates existing records.
- Using an index of *ukInsert* accesses the query that inserts new records.
- Using an index of *ukDelete* accesses the query that deletes records.

The following shows how to use the *Query* property to supply parameter values that can't be bound automatically:

```

procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
    DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
    UpdateSQL1.DataSet := DeltaDS; { required for the automatic parameter substitution }
    with UpdateSQL1.Query[UpdateKind] do
        begin
            { Make sure the query has the correct DatabaseName and SessionName }
            DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
            SessionName := (SourceDS as TDBDataSet).SessionName;
            ParamByName('TimeOfUpdate').Value = Now;
        end;
        UpdateSQL1.Apply(UpdateKind); { now perform automatic substitutions and execute }
        Applied := True;
    end;

```

## Using TBatchMove

---

*TBatchMove* encapsulates Borland Database Engine (BDE) features that let you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

### Creating a batch move component

---

To create a batch move component:

- 1 Place a table or query component for the dataset from which you want to import records (called the *Source* dataset) on a form or in a data module.
- 2 Place the dataset to which to move records (called the *Destination* dataset) on the form or data module.

- 3 Place a *TBatchMove* component from the BDE page of the Component palette in the data module or form, and set its *Name* property to a unique value appropriate to your application.
- 4 Set the *Source* property of the batch move component to the name of the table from which to copy, append, or update records. You can select tables from the drop-down list of available dataset components.
- 5 Set the *Destination* property to the dataset to create, append to, or update. You can select a destination table from the drop-down list of available dataset components.
  - If you are appending, updating, or deleting, *Destination* must represent an existing database table.
  - If you are copying a table and *Destination* represents an existing table, executing the batch move overwrites all of the current data in the destination table.
  - If you are creating an entirely new table by copying an existing table, the resulting table has the name specified in the *Name* property of the table component to which you are copying. The resulting table type will be of a structure appropriate to the server specified by the *DatabaseName* property.
- 6 Set the *Mode* property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For information about these modes, see “Specifying a batch move mode” on page 20-49.
- 7 Optionally set the *Transliterate* property. If *Transliterate* is *True* (the default), character data is translated from the *Source* dataset’s character set to the *Destination* dataset’s character set as necessary.
- 8 Optionally set column mappings using the *Mappings* property. You need not set this property if you want batch move to match columns based on their position in the source and destination tables. For more information about mapping columns, see “Mapping data types” on page 20-50.
- 9 Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used. For more information about handling batch move errors, see “Handling batch move errors” on page 20-51.

## Specifying a batch move mode

---

The *Mode* property specifies the operation a batch move component performs:

**Table 20.8** Batch move modes

Property	Purpose
batAppend	Append records to the destination table.
batUpdate	Update records in the destination table with matching records from the source table. Updating is based on the current index of the destination table.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table.
batCopy	Create the destination table based on the structure of the source table. If the destination table already exists, it is dropped and recreated.
batDelete	Delete records in the destination table that match records in the source table.

### Appending records

To append data, the destination dataset must represent an existing table. During the append operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary. If a conversion is not possible, an exception is thrown and the data is not appended.

### Updating records

To update data, the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. During the update operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

### Appending and updating records

To append and update data the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. Otherwise, data from the source dataset is appended to the destination dataset. During append and update operations, the BDE converts data to appropriate data types and sizes for the destination dataset, if necessary.

### Copying datasets

To copy a source dataset, the destination dataset should not represent an exist table. If it does, the batch move operation overwrites the existing table with a copy of the source dataset.

If the source and destination datasets are maintained by different types of database engines, for example, Paradox and InterBase, the BDE creates a destination dataset



with a structure as close as possible to that of the source dataset and automatically performs data type and size conversions as necessary.

**Note** *TBatchMove* does not copy metadata structures such as indexes, constraints, and stored procedures. You must recreate these metadata objects on your database server or through the SQL Explorer as appropriate.

## Deleting records

To delete data in the destination dataset, it must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are deleted in the destination table.

## Mapping data types

---

In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the *Mappings* property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a “best fit”. It trims character data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of ‘5’ to the corresponding integer value. Values that cannot be converted generate errors. For more information about errors, see “Handling batch move errors” on page 20-51.

When moving data between different table types, a batch move component translates data types as appropriate based on the dataset’s server types. See the BDE online help file for the latest tables of mappings among server types.

**Note** To batch move data to an SQL server database, you must have that database server and a version of Delphi with the appropriate SQL Link installed, or you can use ODBC if you have the proper third party ODBC drivers installed.

## Executing a batch move

---

Use the *Execute* method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
BatchMoveAdd.Execute;
```

You can also execute a batch move at design time by right clicking the mouse on a batch move component and choosing *Execute* from the context menu.

The *MovedCount* property keeps track of the number of records that are moved when a batch move executes.

The *RecordCount* property specifies the maximum number of records to move. If *RecordCount* is zero, all records are moved, beginning with the first record in the source dataset. If *RecordCount* is a positive number, a maximum of *RecordCount* records are moved, beginning with the current record in the source dataset. If *RecordCount* is greater than the number of records between the current record in the source dataset and its last record, the batch move terminates when the end of the source dataset is reached. You can examine *MoveCount* to determine how many records were actually transferred.

## Handling batch move errors

---

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. *TBatchMove* has a number of properties that report on and control error handling.

The *AbortOnProblem* property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems.

The *AbortOnKeyViol* property indicates whether to abort the operation when a Paradox key violation occurs.

The *ProblemCount* property indicates the number of records that could not be handled in the destination table without a loss of data. If *AbortOnProblem* is *True*, this number is one, since the operation is aborted when an error occurs.

The following properties enable a batch move component to create additional tables that document the batch move operation:

- *ChangedTableName*, if specified, creates a local Paradox table containing all records in the destination table that changed as a result of an update or delete operation.
- *KeyViolTableName*, if specified, creates a local Paradox table containing all records from the source table that caused a key violation when working with a Paradox table. If *AbortOnKeyViol* is *True*, this table will contain at most one entry since the operation is aborted on the first problem encountered.

- *ProblemTableName*, if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *True*, there is at most one record in this table since the operation is aborted on the first problem encountered.

**Note** If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table.

## The Data Dictionary

---

When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a client/server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see “Creating attribute sets for field components” on page 19-12. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

A programming interface to the Data Dictionary is available in the `drntf` unit (located in the `lib` directory). This interface supplies the following methods:

**Table 20.9** Data Dictionary interface

Routine	Use
<code>DictionaryActive</code>	Indicates if the data dictionary is active.
<code>DictionaryDeactivate</code>	Deactivates the data dictionary.
<code>IsNullID</code>	Indicates whether a given ID is a null ID
<code>FindDatabaseID</code>	Returns the ID for a database given its alias.
<code>FindTableID</code>	Returns the ID for a table in a specified database.
<code>FindFieldID</code>	Returns the ID for a field in a specified table.
<code>FindAttrID</code>	Returns the ID for a named attribute set.
<code>GetAttrName</code>	Returns the name an attribute set given its ID.
<code>GetAttrNames</code>	Executes a callback for each attribute set in the dictionary.

**Table 20.9** Data Dictionary interface (continued)

Routine	Use
GetAttrID	Returns the ID of the attribute set for a specified field.
NewAttr	Creates a new attribute set from a field component.
UpdateAttr	Updates an attribute set to match the properties of a field.
CreateField	Creates a field component based on stored attributes.
UpdateField	Changes the properties of a field to match a specified attribute set.
AssociateAttr	Associates an attribute set with a given field ID.
UnassociateAttr	Removes an attribute set association for a field ID.
GetControlClass	Returns the control class for a specified attribute ID.
QualifyTableName	Returns a fully qualified table name (qualified by user name).
QualifyTableNameByName	Returns a fully qualified table name (qualified by user name).
HasConstraints	Indicates whether the dataset has constraints in the dictionary.
UpdateConstraints	Updates the imported constraints of a dataset.
UpdateDataset	Updates a dataset to the current settings and constraints in the dictionary.

## Tools for working with the BDE

One advantage of using the BDE as a data access mechanism is the wealth of supporting utilities that ship with Delphi. These utilities include:

- **SQL Explorer** and **Database Explorer**: Delphi ships with one of these two applications, depending on which version you have purchased. Both Explorers enable you to
  - Examine existing database tables and structures. The SQL Explorer lets you examine and query remote SQL databases.
  - Populate tables with data
  - Create extended field attribute sets in the Data Dictionary or associate them with fields in your application.
  - Create and manage BDE aliases.

SQL Explorer lets you do the following as well:

- Create SQL objects such as stored procedures on remote database servers.
- View the reconstructed text of SQL objects on remote database servers.
- Run SQL scripts.
- **SQL Monitor**: SQL Monitor lets you watch all of the communication that passes between the remote database server and the BDE. You can filter the messages you want to watch, limiting them to only the categories of interest. SQL Monitor is most useful when debugging your application.

- **BDE Administration utility:** The BDE Administration utility lets you add new database drivers, configure the defaults for existing drivers, and create new BDE aliases.
- **Database Desktop:** If you are using Paradox or dBASE tables, Database Desktop lets you view and edit their data, create new tables, and restructure existing tables. Using Database Desktop affords you more control than using the methods of a *TTable* component (for example, it allows you to specify validity checks and language drivers). It provides the only mechanism for restructuring Paradox and dBASE tables other than making direct calls the BDE's API.

## Working with ADO components

The *ADOExpress* components provide data access through the ADO framework. ADO, (Microsoft ActiveX Data Objects) is a set of COM objects that access data through an OLE DB provider. The Delphi *ADOExpress* components encapsulate these ADO objects in the Delphi database architecture.

The ADO layer of an ADO-based application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are wrapped by the *TADOConnection*, *TADOCommand*, and ADO dataset components. The ADO framework includes other “helper” objects, like the Field and Properties objects, but these are typically not used directly in Delphi applications and are not wrapped by dedicated components.

This chapter presents the *ADOExpress* components and discusses the unique features they add to the common Delphi database architecture. Before reading about the features peculiar to the *ADOExpress* components, you should familiarize yourself with the common features of database connection components and datasets described in Chapter 17, “Connecting to databases” and Chapter 18, “Understanding datasets.”

### Overview of ADO components

---

The ADO page of the component palette hosts the *ADOExpress* components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most *ADOExpress* components have direct counterparts in the components available for other data access mechanisms: a database connection component (*TADOConnection*) and various types of datasets. In addition, *ADOExpress* includes *TADOCommand*, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The following table lists the ADO components.

**Table 21.1** ADO components

Component	Use
<i>TADOConnection</i>	A database connection component that establishes a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
<i>TADODataSet</i>	The primary dataset for retrieving and operating on data; <i>TADODataSet</i> can retrieve data from a single or multiple tables; can connect directly to a data store or use a <i>TADOConnection</i> component.
<i>TADOTable</i>	A table-type dataset for retrieving and operating on a recordset produced by a single database table; <i>TADOTable</i> can connect directly to a data store or use a <i>TADOConnection</i> component.
<i>TADOQuery</i>	A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement; <i>TADOQuery</i> can also execute data definition language (DDL) SQL statements. It can connect directly to a data store or use a <i>TADOConnection</i> component.
<i>TADOStoredProc</i>	A stored procedure-type dataset for executing stored procedures; <i>TADOStoredProc</i> executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a <i>TADOConnection</i> component.
<i>TADOCommand</i>	A simple component for executing commands (SQL statements that do not return result sets); <i>TADOCommand</i> can be used with a supporting dataset component, or retrieve a dataset from a table; It can connect directly to a data store or use a <i>TADOConnection</i> component.

## Connecting to ADO data stores

Delphi ADO-based applications use Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider that connects to a data store and accesses its data. One of the items a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows.

An ADO provider represents one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must be installed on the client computer. OLE DB drivers for various database systems are supplied by the database vendor or by a third-party. If the application uses an SQL database, such as Microsoft SQL Server or Oracle, the client software for that database system must also be installed on the client computer. Client software is supplied by the database vendor and installed from the database systems CD (or disk).

To connect your application with the data store, use an ADO connection component (*TADOConnection*). Configure the ADO connection component to use one of the available ADO providers. Although *TADOConnection* is not strictly required, because ADO command and dataset components can establish connections directly using their *ConnectionString* property, you can use *TADOConnection* to share a single connection among several ADO components. This can reduce resource consumption, and allows you to create transactions that span multiple datasets.

Like other database connection components, *TADOConnection* provides support for

- Controlling connections
- Controlling server login
- Managing transactions
- Working with associated datasets
- Sending commands to the server
- Obtaining metadata

In addition to these features that are common to all database connection components, *TADOConnection* provides its own support for

- A wide range of options you can use to fine-tune the connection.
- The ability to list the command objects that use the connection.
- Additional events when performing common tasks.

## Connecting to a data store using TADOConnection

---

One or more ADO dataset and command components can share a single connection to a data store by using *TADOConnection*. To do so, associated dataset and command components with the connection component through their *Connection* properties. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the Object Inspector. At runtime, assign the reference to the *Connection* property. For example, the following line associates a *TADODataset* component with a *TADOConnection* component.

```
ADODataset1.Connection := ADOConnection1;
```

The connection component represents an ADO connection object. Before you can use the connection object to establish a connection, you must identify the data store to which you want to connect. Typically, you provide information using the *ConnectionString* property. *ConnectionString* is a semicolon delimited string that lists one or more named connection parameters. These parameters identify the data store by specifying either the name of a file that contains the connection information or the name of an ADO provider and a reference identifying the data store. Use the following, predefined parameter names to supply this information:

Parameter	Description
<i>Provider</i>	The name of a local ADO provider to use for the connection.
<i>Data Source</i>	The name of the data store.
<i>File name</i>	The name of a file containing connection information.
<i>Remote Provider</i>	The name of an ADO provider that resides on a remote machine.
<i>Remote Server</i>	The name of the remote server when using a remote provider.



Thus, a typical value of *ConnectionString* has the form

```
Provider=MSDASQL.1;Data Source=MQIS
```

**Note** The connection parameters in *ConnectionString* do not need to include the *Provider* or *Remote Provider* parameter if you specify an ADO provider using the *Provider* property. Similarly, you do not need to specify the *Data Source* parameter if you use the *DefaultDatabase* property.

In addition, to the parameters listed above, *ConnectionString* can include any connection parameters peculiar to the specific ADO provider you are using. These additional connection parameters can include user ID and password if you want to hardcode the login information.

At design-time, you can use the Connection String Editor to build a connection string by selecting connection elements (like the provider and server) from lists. Click the ellipsis button for the *ConnectionString* property in the Object Inspector to launch the Connection String Editor, which is an ActiveX property editor supplied by ADO.

Once you have specified the *ConnectionString* property (and, optionally, the *Provider* property), you can use the ADO connection component to connect to or disconnect from the ADO data store, although you may first want to use other properties to fine-tune the connection. When connecting to or disconnecting from the data store, *TADOConnection* lets you respond to a few additional events beyond those common to all database connection components. These additional events are described in “Events when establishing a connection” on page 21-7 and “Events when disconnecting” on page 21-8.

**Note** If you do not explicitly activate the connection by setting the connection component’s *Connected* property to *True*, it automatically establishes the connection when the first dataset component is opened or the first time you use an ADO command component to execute a command.

## Accessing the connection object

Use the *ConnectionObject* property of *TADOConnection* to access the underlying ADO connection object. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object.

Using the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in particular. It is not recommended that you use the Connection object unless you are familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

## Fine-tuning a connection

---

One advantage of using *TADOConnection* for establishing the connection to a data store instead of simply supplying a connection string for your ADO command and dataset components, is that it provides a greater degree of control over the conditions and attributes of the connection.

## Forcing asynchronous connections

Use the *ConnectOptions* property to force the connection to be asynchronous. Asynchronous connections allow your application to continue processing without waiting for the connection to be completely opened.

By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*.

The example routines below enable and disable asynchronous connections in the specified connection component:

```

procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
    with ADOConnection1 do begin
        Close;
        ConnectOptions := coAsyncConnect;
        Open;
    end;
end;

procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
    with ADOConnection1 do begin
        Close;
        ConnectOptions := coConnectUnspecified;
        Open;
    end;
end;

```

## Controlling timeouts

You can control the amount of time that can elapse before attempted commands and connections are considered failed and are aborted using the *ConnectionTimeout* and *CommandTimeout* properties.

*ConnectionTimeout* specifies the amount of time, in seconds, before an attempt to connect to the data store times out. If the connection does not successfully compile prior to expiration of the time specified in *ConnectionTimeout*, the connection attempt is canceled:

```

with ADOConnection1 do begin
    ConnectionTimeout := 10 {seconds};
    Open;
end;

```

*CommandTimeout* specifies the amount of time, in seconds, before an attempted command times out. If a command initiated by a call to the *Execute* method does not successfully complete prior to expiration of the time specified in *CommandTimeout*, the command is canceled and ADO generates an exception:

```

with ADOConnection1 do begin
    CommandTimeout := 10 {seconds};
    Execute('DROP TABLE Employee1997', cmdText, []);
end;

```

## Indicating the types of operations the connection supports

ADO connections are established using a specific mode, similar to the mode you use when opening a file. The connection mode determines the permissions available to the connection, and hence the types of operations (such as reading and writing) that can be performed using that connection.

Use the *Mode* property to indicate the connection mode. The possible values are listed in Table 21.2:

**Table 21.2** ADO connection modes

Connect Mode	Meaning
cmUnknown	Permissions are not yet set for the connection or cannot be determined.
cmRead	Read-only permissions are available to the connection.
cmWrite	Write-only permissions are available to the connection.
cmReadWrite	Read/write permissions are available to the connection.
cmShareDenyRead	Prevents others from opening connections with read permissions.
cmShareDenyWrite	Prevents others from opening connection with write permissions.
cmShareExclusive	Prevents others from opening connection.
cmShareDenyNone	Prevents others from opening connection with any permissions.

The possible values for *Mode* correspond to the *ConnectModeEnum* values of the *Mode* property on the underlying ADO connection object. See the Microsoft Data Access SDK help for more information on these values.

## Specifying whether the connection automatically initiates transactions

Use the *Attributes* property to control the connection component's use of retaining commits and retaining aborts. When the connection component uses retaining commits, then every time your application commits a transaction, a new transaction is automatically started. When the connection component uses retaining aborts, then every time your application rolls back a transaction, a new transaction is automatically started.

*Attributes* is a set that can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. When *Attributes* contains *xaCommitRetaining*, the connection uses retaining commits. When *Attributes* contains *xaAbortRetaining*, it uses retaining aborts.

Check whether either retaining commits or retaining aborts is enabled using the *in* operator. Enable retaining commits or aborts by adding the appropriate value to the *attributes* property; disable them by subtracting the value. The example routines below respectively enable and disable retaining commits in an ADO connection component.

```

procedure TForm1.RetainingCommitsOnClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
  end

```

```

    Open;
  end;
end;

procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
end;

```

## Accessing the connection's commands

---

Like other database connection components, you can access the datasets associated with the connection using the *DataSets* and *DataSetCount* properties. However, *ADOExpress* also includes *TADOCommand* objects, which are not datasets, but which maintain a similar relationship to the connection component.

You can use the *Commands* and *CommandCount* properties of *TADOConnection* to access the associated ADO command objects in the same way you use the *DataSets* and *DataSetCount* properties to access the associated datasets. Unlike *DataSets* and *DataSetCount*, which only list active datasets, *Commands* and *CommandCount* provide references to all *TADOCommand* components associated with the connection component.

*Commands* is a zero-based array of references to ADO command components. *CommandCount* provides a total count of all of the commands listed in *Commands*. You can use these properties together to iterate through all the commands that use a connection component, as illustrated in the following code:

```

var
  i: Integer;
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
  end;
end;

```

## ADO connection events

---

In addition to the usual events that occur for all database connection components, *TADOConnection* generates a number of additional events that occur during normal usage.

### Events when establishing a connection

In addition to the *BeforeConnect* and *AfterConnect* events that are common to all database connection components, *TADOConnection* also generates an *OnWillConnect* and *OnConnectComplete* event when establishing a connection. These events occur after the *BeforeConnect* event.

- *OnWillConnect* occurs before the ADO provider establishes a connection. It lets you make last minute changes to the connection string, provide a user name and password if you are handling your own login support, force an asynchronous connection, or even cancel the connection before it is opened.
- *OnConnectComplete* occurs after the connection is opened. Because *TADOConnection* can represent asynchronous connections, you should use *OnConnectComplete*, which occurs after the connection is opened or has failed due to an error condition, instead of the *AfterConnect* event, which occurs after the connection component instructs the ADO provider to open a connection, but not necessarily after the connection is opened.

## Events when disconnecting

In addition to the *BeforeDisconnect* and *AfterDisconnect* events common to all database connection components, *TADOConnection* also generates an *OnDisconnect* event after closing a connection. *OnDisconnect* occurs after the connection is closed but before any associated datasets are closed and before the *AfterDisconnect* event.

## Events when managing transactions

The ADO connection component provides a number of events for detecting when transaction-related processes have been completed. These events indicate when a transaction process initiated by a *BeginTrans*, *CommitTrans*, and *RollbackTrans* method has been successfully completed at the data store.

- The *OnBeginTransComplete* event occurs when the data store has successfully started a transaction after a call to the *BeginTrans* method.
- The *OnCommitTransComplete* event occurs after a transaction is successfully committed due to a call to *CommitTrans*.
- The *OnRollbackTransComplete* event occurs after a transaction is successfully aborted due to a call to *RollbackTrans*.

## Other events

ADO connection components introduce two additional events you can use to respond to notifications from the underlying ADO connection object:

- The *OnExecuteComplete* event occurs after the connection component executes a command on the data store (for example, after calling the *Execute* method). *OnExecuteComplete* indicates whether the execution was successful.
- The *OnInfoMessage* event occurs when the underlying connection object provides detailed information after an operation is completed. The *OnInfoMessage* event handler receives the interface to an ADO Error object that contains the detailed information and a status code indicating whether the operation was successful.

## Using ADO datasets

---

ADO dataset components encapsulate the ADO Recordset object. They inherit the common dataset capabilities described in Chapter 18, “Understanding datasets,” using ADO to provide the implementation. In order to use an ADO dataset, you must familiarize yourself with these common features.

In addition to the common dataset features, all ADO datasets add properties, events, and methods for

- Connecting to an ADO data store.
- Accessing the underlying Recordset object.
- Filtering records based on bookmarks.
- Fetching records asynchronously.
- Performing batch updates (caching updates).
- Using files on disk to store data.

There are four ADO datasets:

- *TADOTable*, a table-type dataset that represents all of the rows and columns of a single database table. See “Using table-type datasets” on page 18-24 for information on using *TADOTable* and other table-type datasets.
- *TADOQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See “Using query-type datasets” on page 18-41 for information on using *TADOQuery* and other query-type datasets.
- *TADOStoredProc*, a stored procedure-type dataset that executes a stored procedure defined on a database server. See “Using stored procedure-type datasets” on page 18-48 for information on using *TADOStoredProc* and other stored procedure-type datasets.
- *TADODataSet*, a general-purpose dataset that includes the capabilities of the other three types. See “Using TADODataSet” on page 21-15 for a description of features unique to *TADODataSet*.

**Note** When using ADO to access database information, you do not need to use a dataset such as *TADOQuery* to represent SQL commands that do not return a cursor. Instead, you can use *TADOCommand*, a simple component that is not a dataset. For details on *TADOCommand*, see “Using Command objects” on page 21-16.

### Connecting an ADO dataset to a data store

ADO datasets can connect to an ADO data store either collectively or individually.

When connecting datasets collectively, set the *Connection* property of each dataset to a *TADOConnection* component. Each dataset then uses the ADO connection component’s connection.

```
ADODataset1.Connection := ADOConnection1;
ADODataset2.Connection := ADOConnection1;
...
```

Among the advantages of connecting datasets collectively are:

- The datasets share the connection object's attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The datasets can participate in transactions.

For more information on using *TADOConnection* see "Connecting to ADO data stores" on page 21-2.

When connecting datasets individually, set the *ConnectionString* property of each dataset. Each dataset that uses *ConnectionString* establishes its own connection to the data store, independent of any other dataset connection in the application.

The *ConnectionString* property of ADO datasets works the same way as the *ConnectionString* property of *TADOConnection*: it is a set of semicolon-delimited connection parameters such as the following:

```
ADODataset1.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +  
    'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +  
    'Initial Catalog=Employee';
```

At design time you can use the Connection String Editor to help you build the connection string. For more information about connection strings, see "Connecting to a data store using *TADOConnection*" on page 21-3.

## Working with record sets

The *Recordset* property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *Recordset* to directly access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. Using the recordset object directly is not recommended unless you are familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

The *RecordsetState* property indicates the current state of the underlying recordset object. *RecordsetState* corresponds to the *State* property of the ADO recordset object. The value of *RecordsetState* is either *stOpen*, *stExecuting*, or *stFetching*. (*TObjectState*, the type of the *RecordsetState* property, defines other values, but only *stOpen*, *stExecuting*, and *stFetching* pertain to recordsets.) A value of *stOpen* indicates that the recordset is currently idle. A value of *stExecuting* indicates that it is executing a command. A value of *stFetching* indicates that it is fetching rows from the associated table (or tables).

Use *RecordsetState* values to perform actions dependent on the current state of the dataset. For example, a routine that updates data might check the *RecordsetState* property to see whether the dataset is active and not in the process of other activities such as connecting or fetching data.

## Filtering records based on bookmarks

ADO datasets support the common dataset feature of using bookmarks to mark and return to specific records. Also like other datasets, ADO datasets let you use filters to

limit the available records in the dataset. ADO datasets provide an additional feature that combines these two common dataset features: the ability to filter on a set of records identified by bookmarks.

To filter on a set of bookmarks,

- 1 Use the *Bookmark* method to mark the records you want to include in the filtered dataset.
- 2 Call the *FilterOnBookmarks* method to filter the dataset so that only the bookmarked records appear.

This process is illustrated below:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    BM1, BM2: TBookmarkStr;
begin
    with ADODataset1 do begin
        BM1 := Bookmark;
        BMList.Add(Pointer(BM1));
        MoveBy(3);
        BM2 := Bookmark;
        BMList.Add(Pointer(BM2));
        FilterOnBookmarks([BM1, BM2]);
    end;
end;

```

Note that the example above also adds the bookmarks to a list object named BMList. This is necessary so that the application can later free the bookmarks when they are no longer needed.

For details on using bookmarks, see “Marking and returning to records” on page 18-9. For details on other types of filters, see “Displaying and editing a subset of data using filters” on page 18-12.

## Fetching records asynchronously

Unlike other datasets, ADO datasets can fetch their data asynchronously. This allows your application to continue performing other tasks while the dataset populates itself with data from the data store.

To control whether the dataset fetches data asynchronously, if it fetches data at all, use the *ExecuteOptions* property. *ExecuteOptions* governs how the dataset fetches its records when you call *Open* or set *Active* to *True*. If the dataset represents a query or stored procedure that does not return any records, *ExecuteOptions* governs how the query or stored procedure is executed when you call *ExecSQL* or *ExecProc*.



*ExecuteOptions* is a set that includes zero or more of the following values:

**Table 21.3** Execution options for ADO datasets

Execute Option	Meaning
eoAsyncExecute	The command or data fetch operation is executed asynchronously.
eoAsyncFetch	The dataset first fetches the number of records specified by the <i>CacheSize</i> property synchronously, then fetches any remaining rows asynchronously.
eoAsyncFetchNonBlocking	Asynchronous data fetches or command execution do not block the current thread of execution.
eoExecuteNoRecords	A command or stored procedure that does not return data. If any rows are retrieved, they are discarded and not returned.

## Using batch updates

One approach for caching updates is to connect the ADO dataset to a client dataset using a dataset provider. This approach is discussed in “Using a client dataset to cache updates” on page 23-15.

However, ADO dataset components provide their own support for cached updates, which they call batch updates. The following table lists the correspondences between caching updates using a client dataset and using the batch updates features:

**Table 21.4** Comparison of ADO and client dataset cached updates

ADO dataset	TClientDataSet	Description
LockType	Not used: client datasets always cache updates	Specifies whether the dataset is opened in batch update mode.
CursorType	Not used: client datasets always work with an in-memory snapshot of data	Specifies how isolated the ADO dataset is from changes on the server.
RecordStatus	UpdateStatus	Indicates what update, if any, has occurred on the current row. <i>RecordStatus</i> provides more information than <i>UpdateStatus</i> .
FilterGroup	StatusFilter	Specifies which type of records are available. <i>FilterGroup</i> provides a wider variety of information.
UpdateBatch	ApplyUpdates	Applies the cached updates back to the database server. Unlike <i>ApplyUpdates</i> , <i>UpdateBatch</i> lets you limit the types of updates to be applied.
CancelBatch	CancelUpdates	Discards pending updates, reverting to the original values. Unlike <i>CancelUpdates</i> , <i>CancelBatch</i> lets you limit the types of updates to be canceled.

Using the batch updates features of ADO dataset components is a matter of:

- Opening the dataset in batch update mode
- Inspecting the update status of individual rows
- Filtering multiple rows based on update status
- Applying the batch updates to base tables
- Canceling batch updates

## Opening the dataset in batch update mode

To open an ADO dataset in batch update mode, it must meet these criteria:

- 1 The component's *CursorType* property must be *ctKeySet* (the default property value) or *ctStatic*.
- 2 The *LockType* property must be *ltBatchOptimistic*.
- 3 The command must be a SELECT query.

Before activating the dataset component, set the *CursorType* and *LockType* properties as indicated above. Assign a SELECT statement to the component's *CommandText* property (for *TADODataset*) or the *SQL* property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that returns a result set. These properties can be set at design-time through the Object Inspector or programmatically at runtime. The example below shows the preparation of a *TADODataset* component for batch update mode.

```
with ADODataset1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

After a dataset has been opened in batch update mode, all changes to the data are cached rather than applied directly to the base tables.

## Inspecting the update status of individual rows

Determine the update status of a given row by making it current and then inspecting the *RecordStatus* property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unchanged record';
  rsModified:   StatusBar1.Panels[0].Text := 'Changed record';
  rsDeleted:   StatusBar1.Panels[0].Text := 'Deleted record';
  rsNew:       StatusBar1.Panels[0].Text := 'New record';
end;
```

## Filtering multiple rows based on update status

Filter a recordset to show only those rows that belong to a group of rows with the same update status using the *FilterGroup* property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

**Note** For the *FilterGroup* property to have an effect, the ADO dataset component's *Filtered* property must be set to *True*.

### Applying the batch updates to base tables

Apply pending data changes that have not yet been applied or canceled by calling the *UpdateBatch* method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache.

Used alone with no parameter, *UpdateBatch* applies all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *arAll* is passed, only a subset of the pending changes are applied. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be applied. The example below applies only the currently active row to be applied:

```
ADODataset1.UpdateBatch(arCurrent);
```

### Canceling batch updates

Cancel pending data changes that have not yet been canceled or applied by calling the *CancelBatch* method. When you cancel pending batch updates, field values on rows that have been changed revert to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes.

Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *CancelBatch*. If any value except *arAll* is passed, only a subset of the pending changes are canceled. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
ADODataset1.CancelBatch;
```

### Loading data from and saving data to files

The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. The data is saved in one of two proprietary formats: ADTG or XML. These two file formats are the only formats supported by ADO. However, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the version you are using to determine what save file formats are supported.

Save the data to a file using the *SaveToFile* method. *SaveToFile* takes two parameters, the name of the file to which data is saved, and, optionally, the format (ADTG or XML) in which to save the data. Indicate the format for the saved file by setting the *Format* parameter to *pfADTG* or *pfXML*. If the file specified by the *FileName* parameter already exists, *SaveToFile* raises an *EOleException*.

Retrieve the data from file using the *LoadFromFile* method. *LoadFromFile* takes a single parameter, the name of the file to load. If the specified file does not exist, *LoadFromFile* raises an *EOleException* exception. On calling the *LoadFromFile* method, the dataset component is automatically activated.

In the example below, the first procedure saves the dataset retrieved by the *TADODataset* component *ADODataset1* to a file. The target file is an ADTG file named *SaveFile*, saved to a local drive. The second procedure loads this saved file into the *TADODataset* component *ADODataset2*.

```

procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    begin
      DeleteFile('c:\SaveFile');
      StatusBar1.Panels[0].Text := 'Save file deleted!';
    end;
  ADODataset1.SaveToFile('c:\SaveFile', pfADTG);
end;

procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataset2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Save file does not exist!';
  end;

```

The datasets that save and load the data need not be on the same form as above, in the same application, or even on the same computer. This allows for the briefcase-style transfer of data from one computer to another.

## Using TADODataset

---

*TADODataset* is a general-purpose dataset for working with data from an ADO data store. Unlike the other ADO dataset components, *TADODataset* is not a table-type, query-type, or stored procedure-type dataset. Instead, it can function as any of these types:

- Like a table-type dataset, *TADODataset* lets you represent all of the rows and columns of a single database table. To use it in this way, set the *CommandType* property to *cmdTable* and the *CommandText* property to the name of the table. *TADODataset* supports table-type tasks such as
  - Assigning indexes to sort records or form the basis of record-based searches. In addition to the standard index properties and methods described in “Sorting records with indexes” on page 18-25, *TADODataset* lets you sort using temporary indexes by setting the *Sort* property. Indexed-based searches performed using the *Seek* method use the current index.
  - Emptying the dataset. The *DeleteRecords* method provides greater control than related methods in other table-type datasets, because it lets you specify what records to delete.

The table-type tasks supported by *TADODataset* are available even when you are not using a *CommandType* of *cmdTable*.

- Like a query-type dataset, *TADODataset* lets you specify a single SQL command that is executed when you open the dataset. To use it in this way, set the *CommandType* property to *cmdText* and the *CommandText* property to the SQL command you want to execute. At design time, you can double-click on the *CommandText* property in the Object Inspector to use the Command Text editor for help in constructing the SQL command. *TADODataset* supports query-type tasks such as
  - Using parameters in the query text. See “Using parameters in queries” on page 18-43 for details on query parameters.
  - Setting up master/detail relationships using parameters. See “Establishing master/detail relationships using parameters” on page 18-46 for details on how to do this.
  - Preparing the query in advance to improve performance by setting the *Prepared* property to *True*.
- Like a stored procedure-type dataset, *TADODataset* lets you specify a stored procedure that is executed when you open the dataset. To use it in this way, set the *CommandType* property to *cmdStoredProc* and the *CommandText* property to the name of the stored procedure. *TADODataset* supports stored procedure-type tasks such as
  - Working with stored procedure parameters. See “Working with stored procedure parameters” on page 18-50 for details on stored procedure parameters.
  - Fetching multiple result sets. See “Fetching multiple result sets” on page 18-53 for details on how to do this.
  - Preparing the stored procedure in advance to improve performance by setting the *Prepared* property to *True*.

In addition, *TADODataset* lets you work with data stored in files by setting the *CommandType* property to *cmdFile* and the *CommandText* property to the file name.

Before you set the *CommandText* and *CommandType* properties, you should link the *TADODataset* to a data store by setting the *Connection* or *ConnectionString* property. This process is described in “Connecting an ADO dataset to a data store” on page 21-9. As an alternative, you can use an RDS DataSpace object to connect the *TADODataset* to an ADO-based application server. To use an RDS DataSpace object, set the *RDSConnection* property to a *TRDSConnection* object.

## Using Command objects

---

In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language.

Although you can always execute commands using *TADOQuery*, you may not want the overhead of using a dataset component, especially if the command does not return a result set. As an alternative, you can use the *TADOCommand* component, which is a lighter-weight object designed to execute commands, one command at a time. *TADOCommand* is intended primarily for executing those commands that do not return result sets, such as Data Definition Language (DDL) SQL statements. Through an overloaded version of its *Execute* method, however, it is capable of returning a result set that can be assigned to the *RecordSet* property of an ADO dataset component.

In general, working with *TADOCommand* is very similar to working with *TADODataSet*, except that you can't use the standard dataset methods to fetch data, navigate records, edit data, and so on. *TADOCommand* objects connect to a data store in the same way as ADO datasets. See "Connecting an ADO dataset to a data store" on page 21-9 for details.

The following topics provide details on how to specify and execute commands using *TADOCommand*.

## Specifying the command

---

Specify commands for a *TADOCommand* component using the *CommandText* property. Like *TADODataSet*, *TADOCommand* lets you specify the command in different ways, depending on the *CommandType* property. Possible values for *CommandType* include: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the Object Inspector. At runtime, assign a value of type *TCommandType* to the *CommandType* property.

```
with ADOCommand1 do begin
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
  ...
end;
```

If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*.

*CommandText* can contain the text of an SQL query that includes parameters or the name of a stored procedure that uses parameters. You must then supply parameter values, which are bound to the parameters before executing the command. See "Handling command parameters" on page 21-19 for details.

## Using the Execute method

---

Before *TADOCommand* can execute its command, it must have a valid connection to a data store. This is established just as with an ADO dataset. See "Connecting an ADO dataset to a data store" on page 21-9 for details.

To execute the command, call the *Execute* method. *Execute* is an overloaded method that lets you choose the most appropriate way to execute the command.

For commands that do not require any parameters and for which you do not need to know how many records were affected, call *Execute* without any parameters:

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

Other versions of *Execute* let you provide parameter values using a Variant array, and to obtain the number of records affected by the command.

For information on executing commands that return a result set, see “Retrieving result sets with commands” on page 21-18.

## Canceling commands

---

If you are executing the command asynchronously, then after calling *Execute* you can abort the execution by calling the *Cancel* method:

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;

procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

The *Cancel* method only has an effect if there is a command pending and it was executed asynchronously (*eoAsynchExecute* is in the *ExecuteOptions* parameter of the *Execute* method). A command is said to be pending if the *Execute* method has been called but the command has not yet been completed or timed out.

A command times out if it is not completed or canceled before the number of seconds specified in the *CommandTimeout* property expire. By default, commands time out after 30 seconds.

## Retrieving result sets with commands

---

Unlike *TADOQuery* components, which use different methods to execute depending on whether they return a result set, *TADOCommand* always uses the *Execute* command to execute the command, regardless of whether it returns a result set. When the command returns a result set, *Execute* returns an interface to the *ADO\_RecordSet* interface.

The most convenient way to work with this interface is to assign it to the *RecordSet* property of an ADO dataset.

For example, the following code uses *TADOCCommand* (*ADOCCommand1*) to execute a SELECT query, which returns a result set. This result set is then assigned to the *RecordSet* property of a *TADODataset* component (*ADODataset1*).

```
with ADOCCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataset1.Recordset := Execute;
end;
```

As soon as the result set is assigned to the ADO dataset's *Recordset* property, the dataset is automatically activated and the data is available.

## Handling command parameters

---

There are two ways in which a *TADOCCommand* object may use parameters:

- The *CommandText* property can specify a query that includes parameters. Working with parameterized queries in *TADOCCommand* works like using a parameterized query in an ADO dataset. See “Using parameters in queries” on page 18-43 for details on parameterized queries.
- The *CommandText* property can specify a stored procedure that uses parameters. Stored procedure parameters work much the same using *TADOCCommand* as with an ADO dataset. See “Working with stored procedure parameters” on page 18-50 for details on stored procedure parameters.

There are two ways to supply parameter values when working with *TADOCCommand*: you can supply them when you call the *Execute* method, or you can specify them ahead of time using the *Parameters* property.

The *Execute* method is overloaded to include versions that take a set of parameter values as a Variant array. This is useful when you want to supply parameter values quickly without the overhead of setting up the *Parameters* property:

```
ADOCCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
```

When working with stored procedures that return output parameters, you must use the *Parameters* property instead. Even if you do not need to read output parameters, you may prefer to use the *Parameters* property, which lets you supply parameters at design time and lets you work with *TADOCCommand* properties in the same way you work with the parameters on datasets.

When you set the *CommandText* property, the *Parameters* property is automatically updated to reflect the parameters in the query or those used by the stored procedure. At design-time, you can use the Parameter Editor to access parameters, by clicking the ellipsis button for the *Parameters* property in the Object Inspector. At runtime, use properties and methods of *TParameter* to set (or get) the values of each parameter.



## Using Command objects

```
with ADOCommand1 do begin  
    CommandText := 'INSERT INTO Talley ' +  
        '(Counter) ' +  
        'VALUES (:NewValueParam)';  
    CommandType := cmdText;  
    Parameters.ParamByName('NewValueParam').Value := 57;  
    Execute  
end;
```

## Using unidirectional datasets

*dbExpress* is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy a database application that uses *dbExpress*, you need only include a dll (the server-specific driver) with the application files you build.

*dbExpress* lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by *TDataSet* are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the *First* and *Next* methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The *CanModify* property is always *False*, so attempts to put the dataset into edit mode always fail. You can, however, use unidirectional datasets to update data using an SQL UPDATE command or provide conventional editing support by using a *dbExpress*-enabled client dataset or connecting the dataset to a client dataset (see “Connecting to another dataset” on page 14-10).
- There is no support for filters, because filters work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.

- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are the fastest data access mechanism, and very simple to use and deploy.

## Types of unidirectional datasets

---

The *dbExpress* page of the component palette contains four types of unidirectional dataset: *TSQLDataSet*, *TSQLQuery*, *TSQLTable*, and *TSQLStoredProc*.

*TSQLDataSet* is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

*TSQLQuery* is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See “Using query-type datasets” on page 18-41 for information on using query-type datasets.

*TSQLTable* is a table-type dataset that represents all of the rows and columns of a single database table. See “Using table-type datasets” on page 18-24 for information on using table-type datasets.

*TSQLStoredProc* is a stored procedure-type dataset that executes a stored procedure defined on a database server. See “Using stored procedure-type datasets” on page 18-48 for information on using stored procedure-type datasets.

**Note** The *dbExpress* page also includes *TSQLClientDataSet*, which is not a unidirectional dataset. Rather, it is a client dataset that uses a unidirectional dataset internally to access its data

## Connecting to the database server

---

The first step when working with a unidirectional dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the Object Inspector can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the Object Inspector can list what stored procedures are available on the server.

The connection to a database server is represented by a separate *TSQLConnection* component. You work with *TSQLConnection* like any other database connection component. For information about database connection components, see Chapter 17, “Connecting to databases”.

To use *TSQLConnection* to connect a unidirectional dataset to a database server, set the *SQLConnection* property. At design time, you can choose the SQL connection

component from a drop-down list in the Object Inspector. If you make this assignment at runtime, be sure that the connection is active:

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers. However, you may want to use a separate connection for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the *MaxStmtsPerConn* property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the *AutoClone* property to *False*.

Before you assign the *SQLConnection* property, you will need to set up the *TSQLConnection* component so that it identifies the database server and any required connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on).

## Setting up TSQLConnection

---

In order to describe a database connection in sufficient detail for *TSQLConnection* to open a connection, you must identify both the driver to use and a set of connection parameters that are passed to that driver.

### Identifying the driver

The driver is identified by the *DriverName* property, which is the name of an installed *dbExpress* driver, such as INTERBASE, ORACLE, MYSQL, or DB2. The driver name is associated with two files

- The *dbExpress* driver. This can be either a dynamic-link library with a name like *dbexpint.dll*, *dbexpora.dll*, *dbexpmys.dll*, or *dbexpdb2.dll*, or a compiled unit that you can statically link into your application (*dbexptint.dcu*, *dbexpora.dcu*, *dbexpmys.dcu*, or *dbexpdb2.dcu*).
- The dynamic-link library provided by the database vendor for client-side support.

The relationship between these two files and the database name is stored in a file called *dbxdrivers.ini*, which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in *dbxdrivers.ini* when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the *LibraryName* and *VendorLib* properties to the names of the associated dlls. Once *LibraryName* and *VendorLib* have been set, your application does not need to rely on *dbxdrivers.ini*. (That is, you do not need to deploy *dbxdrivers.ini* with your application unless you set the *DriverName* property at runtime.)

## Specifying connection parameters

The *Params* property is a string list that lists name/value pairs. Each pair has the form *Name=Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with ORACLE it is the entry in TNSNames.ora, while with DB2, it is the client-side node name.

Other typical parameters include the *User\_Name* (the name to use when logging in), *Password* (the password for *User\_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the Object Inspector to edit the parameters using the String List editor. At runtime, use the *Params.Values* property to assign values to individual parameters.

## Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. You can name *dbExpress* database and parameter combinations, which are then saved in a file called *dbxconnections.ini*. The name of each combination is called a connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the *ConnectionName* property to a valid connection name. Setting *ConnectionName* automatically sets the *DriverName* and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the *dbxconnections.ini* file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of *dbxconnections.ini*. Then, when you deploy your application, it loads these values from a separate version of *dbxconnections.ini* that uses the “real” database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the *LoadParamsOnConnect* property to *True*. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in *dbxconnections.ini* when the connection is opened.

- Call the *LoadParamsFromIniFile* method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in *dbxconnections.ini* (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

## Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the *dbxconnections.ini* file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in *dbxconnections.ini*:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to *dbxconnections.ini*.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to *dbxconnections.ini*.
- Click the Delete Connection button to delete the currently selected named connection from *dbxconnections.ini*.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

## Specifying what data to display

---

There are a number of ways to specify what data a unidirectional dataset represents. Which method you choose depends on the type of unidirectional dataset you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a *TSQLDataSet* component, use the *CommandType* property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.

- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

**Note** You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see “Fetching metadata into a unidirectional dataset” on page 22-12.

## Representing the results of a query

---

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

When using *TSQLDataSet*, set the *CommandType* property to *ctQuery* and assign the text of the query statement to the *CommandText* property. When using *TSQLQuery*, assign the query to the *SQL* property instead. These properties work the same way for all general-purpose or query-type datasets. “Specifying the query” on page 18-42 discusses them in greater detail.

When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. Using parameters in queries and supplying values for those parameters is discussed in “Using parameters in queries” on page 18-43.

SQL defines queries such as UPDATE queries that perform actions on the server but do not return records. Such queries are discussed in “Executing commands that do not return records” on page 22-9.

## Representing the records in a table

---

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

**Note** If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This can result in slightly slower performance on the server. The wildcard (\*) in automatically-generated queries is more robust to changes in the fields on the server.

## Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the *CommandType* property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- *CommandText* specifies the name of the database table that the *TSQLDataSet* object should represent.
- *SortFieldNames* lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

*TSQLDataSet* generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

## Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the *TableName* property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the *IndexName* property to the name of an index defined on the server that imposes the order you want.
- Set the *IndexFieldNames* property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

## Representing the results of a stored procedure

---

Stored procedures are sets of SQL statements that are named and stored on an SQL server. How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

When using *TSQLDataSet*, to specify a stored procedure:

- Set the *CommandType* property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the *CommandText* property:

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

When using *TSQLStoredProc*, you need only specify the name of the stored procedure as the value of the *StoredProcName* property.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure. See “Working with stored procedure parameters” on page 18-50 for information about working with stored procedure parameters.



## Fetching the data

---

Once you have specified the source of the data, you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a unidirectional dataset:

- Set the *Active* property to *True*, either at design time in the Object Inspector, or in code at runtime:

```
CustQuery.Active := True;
```

- Call the *Open* method at runtime,

```
CustQuery.Open;
```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

## Preparing the dataset

---

Before a query or stored procedure can execute on the server, it must first be “prepared”. Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to *True* or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its *Prepared* property to *True*.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

## Fetching multiple datasets

---

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the *NextRecordSet* method:

```
var
  DataSet2: TSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLDataSet1.NextRecordSet (nRows);
  ...
```

*NextRecordSet* returns a newly created *TSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns **nil**.

## Executing commands that do not return records

---

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

**Note** If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See “Sending commands to the server” on page 17-10 for details.

## Specifying the command to execute

---

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using *TSQLDataSet*, use the *CommandType* and *CommandText* properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.

- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using *TSQLQuery*, use the *SQL* property to specify the SQL statement to pass to the server.

When using *TSQLStoredProc*, use the *StoredProcName* property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters or stored procedure parameters the same way as with queries and stored procedures that return records. See “Using parameters in queries” on page 18-43 and “Working with stored procedure parameters” on page 18-50 for details.

## Executing the command

---

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

- The *ExecSQL* method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';  
FixTicket.ExecSQL;
```

- The *ExecProc* method if the dataset is an instance of *TSQLStoredProc*.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';  
SQLStoredProc1.ExecProc;
```

**Tip** If you are executing the query or stored procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

## Creating and modifying server metadata

---

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset and let it handle all the generation of all SQL commands concerned with editing (see “Connecting a client dataset to another dataset in the same application” on page 14-11). In fact, this is the recommended approach because data-aware controls are designed to perform edits through a dataset such as *TClientDataSet*.

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET\_EMP\_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP\_NO and :PROJ\_ID) with a parameter of the query that creates the stored procedure.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
    'RETURNS (PROJ_ID CHAR(5)) AS ' +
    'BEGIN ' +
    'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
    'WHERE EMP_NO = :EMP_NO ' +
    'INTO :PROJ_ID ' +
    'DO SUSPEND; ' +
    'END';
  ExecSQL;
end;
```

## Setting up master/detail linked cursors

---

There are two ways to use linked cursors to set up a master/detail relationship with a unidirectional dataset as the detail set. Which method you use depends on the type of unidirectional dataset you are using. Once you have set up such a relationship, the unidirectional dataset (the “many” in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the “one” in the one-to-many relationship).

*TSQLDataSet* and *TSQLQuery* require you to use a parameterized query to establish a master/detail relationship. This is the technique for creating such relationships on all query-type datasets. For details on creating master/detail relationships with query-type datasets, see “Establishing master/detail relationships using parameters” on page 18-46.

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the *MasterSource* and *MasterFields* properties, just as you would with any other table-type dataset. For details on creating master/detail relationships with table-type datasets, see “Establishing master/detail relationships using parameters” on page 18-46.

## Accessing schema information

---

There are two ways to obtain information about what is available on the server. This information, called schema information or metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

The simplest way to obtain this metadata is to use the methods of *TSQLConnection*. These methods fill an existing string list or list object with the names of tables, stored procedures, fields, or indexes, or with parameter descriptors. This technique is the same as the way you fill lists with metadata for any other database connection component. These methods are described in “Obtaining metadata” on page 17-12.

If you require more detailed schema information, you can populate a unidirectional dataset with metadata. Instead of a simple list, the unidirectional dataset is filled with schema information, where each record represents a single table, stored procedure, index, field, or parameter.

## Fetching metadata into a unidirectional dataset

---

To populate a unidirectional datasets with metadata from the database server, you must first indicate what data you want to see, using the *SetSchemaInfo* method. *SetSchemaInfo* takes three parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a

list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see “The structure of metadata datasets” on page 22-13.

- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil.
- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as ‘Cust%’, which uses the wildcards ‘%’ (to match a string of arbitrary characters of any length) and ‘\_’ (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or \_\_). If you do not want to use a pattern, this parameter can be nil.

**Note** If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection’s *TableScope* property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, instead, you want to obtain a list of input parameters for a stored procedure named ‘MyProc’. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters (‘inName’, ‘outValue’ and so on). You could call *SetSchemaInfo* as follows:

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

## Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the *CommandText* property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

## The structure of metadata datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

### Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

**Table 22.1** Columns in tables of metadata listing tables

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table. This is the same as the <i>Database</i> parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the table.
TABLE_NAME	ftString	The name of the table. This field determines the sort order of the dataset.
TABLE_TYPE	ftInteger	Identifies the type of table. It is a sum of one or more of the following values: 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table.

### Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

**Table 22.2** Columns in tables of metadata listing stored procedures

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the <i>Database</i> parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure. This field determines the sort order of the dataset.
PROC_TYPE	ftInteger	Identifies the type of stored procedure. It is a sum of one or more of the following values: 1: Procedure 2: Function 4: Package 8: System procedure
IN_PARAMS	ftSmallint	The number of input parameters
OUT_PARAMS	ftSmallint	The number of output parameters.

### Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

**Table 22.3** Columns in tables of metadata listing fields

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table whose fields you listing. This is the same as the <i>Database</i> parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the field.
TABLE_NAME	ftString	The name of the table that contains the fields.
COLUMN_NAME	ftString	The name of the field. This value determines the sort order of the dataset.
COLUMN_POSITION	ftSmallint	The position of the column in its table.
COLUMN_TYPE	ftInteger	Identifies the type of value in the field. It is a sum of one or more of the following: 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	The datatype of the column. This is one of the logical field type constants defined in <i>sqlinks.pas</i> .
COLUMN_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in <i>COLUMN_DATATYPE</i> and <i>COLUMN_SUBTYPE</i> , but in a form used in some DDL statements.
COLUMN_SUBTYPE	ftSmallint	A subtype for the column's datatype. This is one of the logical subtype constants defined in <i>sqlinks.pas</i> .
COLUMN_PRECISION	ftInteger	The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on).
COLUMN_SCALE	ftSmallint	The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields.
COLUMN_LENGTH	ftInteger	The number of bytes required to store field values.
COLUMN_NULLABLE	ftSmallint	A Boolean that indicates whether the field can be left blank (0 means the field requires a value).



### Information about indexes

When you request information about the indexes on a table (`stIndexes`), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

**Table 22.4** Columns in tables of metadata listing indexes

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the index. This is the same as the <i>Database</i> parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the index.
TABLE_NAME	ftString	The name of the table for which the index is defined.
INDEX_NAME	ftString	The name of the index. This field determines the sort order of the dataset.
PKEY_NAME	ftString	Indicates the name of the primary key.
COLUMN_NAME	ftString	The name of the field (column) in the index.
COLUMN_POSITION	ftSmallint	The position of this field in the index.
INDEX_TYPE	ftSmallint	Identifies the type of index. It is a sum of one or more of the following values: 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	Indicates that the index is ascending (a) or descending (d).
FILTER	ftString	Describes a filter condition that limits the indexed records.

### Information about stored procedure parameters

When you request information about the parameters of a stored procedure (`stProcedureParams`), the resulting dataset includes a record for each parameter. It has the following columns:

**Table 22.5** Columns in tables of metadata listing parameters

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the <i>Database</i> parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure that contains the parameter.
PARAM_NAME	ftString	The name of the parameter. This field determines the sort order of the dataset.
PARAM_TYPE	ftSmallint	Identifies the type of parameter. This is the same as a <i>TParam</i> object's <i>ParamType</i> property.

**Table 22.5** Columns in tables of metadata listing parameters (continued)

Column name	Field type	Contents
PARAM_DATATYPE	ftSmallint	The datatype of the parameter. This is one of the logical field type constants defined in <code>sqllinks.pas</code> .
PARAM_SUBTYPE	ftSmallint	A subtype for the parameter's datatype. This is one of the logical subtype constants defined in <code>sqllinks.pas</code> .
PARAM_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in <code>PARAM_DATATYPE</code> and <code>PARAM_SUBTYPE</code> , but in a form used in some DDL statements.
PARAM_PRECISION	ftInteger	The maximum number of digits in floating-point values or bytes (for strings and Bytes fields).
PARAM_SCALE	ftSmallint	The number of digits to the right of the decimal on floating-point values.
PARAM_LENGTH	ftInteger	The number of bytes required to store parameter values.
PARAM_NULLABLE	ftSmallint	A Boolean that indicates whether the parameter can be left blank (0 means the parameter requires a value).

## Debugging dbExpress applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

### Using TSQLMonitor to monitor SQL commands

*TSQLConnection* uses a companion component, *TSQLMonitor*, to intercept these messages and save them in a string list. *TSQLMonitor* works much like the SQL monitor utility that you can use with the BDE, except that it monitors only those commands involving a single *TSQLConnection* component rather than all commands managed by *dbExpress*.

To use *TSQLMonitor*,

- 1 Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.
- 2 Set its *TSQLConnection* property to the *TSQLConnection* component.
- 3 Set the SQL monitor's *Active* property to *True*.

As SQL commands are sent to the server, the SQL monitor's *TraceList* property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the *FileName* property and then setting the *AutoSave* property to *True*. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time it logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the *OnLogTrace* event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of *TraceList* every 10th message, clearing the log after saving it so that the list never gets too long:

```

procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CInfo: Pointer);
var
    LogFileName: string;
begin
    with Sender as TSQLMonitor do
        begin
            if TraceCount = 10 then
                begin
                    LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
                    Tag := Tag + 1; {ensure next log file has a different name }
                    SaveToFile(LogFileName);
                    TraceList.Clear; { clear list }
                end;
            end;
        end;
end;

```

**Note** If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

## Using a callback to monitor SQL commands

---

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's *SetTraceCallbackEvent* method. *SetTraceCallbackEvent* takes two parameters: a callback of type *TSQLCallbackEvent*, and a user-defined value that is passed to the callback function.

The callback function takes two parameters: *CallType* and *CInfo*:

- *CallType* is reserved for future use.
- *CInfo* is a pointer to a record that includes the category (the same as *CallType*), the text of the SQL command, and the user-defined value that is passed to the *SetTraceCallbackEvent* method.

The callback returns a value of type *CBRTYPE*, typically *cbrUSEDEF*.

The *dbExpress* driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

**Warning** Do not call *SetTraceCallbackEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

## Using client datasets

Client datasets are specialized datasets that hold all their data in memory. The support for manipulating the data they store in memory is provided by `midaslib.dcu` or `midas.dll`. The format client datasets use for storing data is self-contained and easily transported, which allows client datasets to

- Read from and write to dedicated files on disk, acting as a file-based dataset. Properties and methods supporting this mechanism are described in “Using a client dataset with file-based data” on page 23-31.
- Cache updates for data from a database server. Client dataset features that support cached updates are described in “Using a client dataset to cache updates” on page 23-15.
- Represent the data in the client portion of a multi-tiered application. To function in this way, the client dataset must work with an external provider, as described in “Using a client dataset with a provider” on page 23-23. For information about multi-tiered database applications, see Chapter 25, “Creating multi-tiered applications.”
- Represent the data from a source other than a dataset. Because a client dataset can use the data from an external provider, specialized providers can adapt a variety of information sources to work with client datasets. For example, you can use an XML provider to enable a client dataset to represent the information in an XML document.

Whether you use client datasets for file-based data, caching updates, data from an external provider (such as working with an XML document or in a multi-tiered application), or a combination of these approaches such as a “briefcase model” application, you can take advantage of broad range of features client datasets support for working with data.

## Working with data using a client dataset

---

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Chapter 15, “Using data controls” for information on how to display database information in data-aware controls.

Client datasets implement all the properties and methods inherited from *TDataSet*. For a complete introduction to this generic dataset behavior, see Chapter 18, “Understanding datasets.”

In addition, client datasets implement many of the features common to table-type datasets such as

- Sorting records with indexes.
- Using Indexes to search for records.
- Limiting records with ranges.
- Creating master/detail relationships.
- Controlling read/write access
- Creating the underlying dataset
- Emptying the dataset
- Synchronizing client datasets

For details on these features, see “Using table-type datasets” on page 18-24.

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for some database functions can involve additional capabilities or considerations. This chapter describes some of these common functions and the differences introduced by client datasets.

### Navigating data in client datasets

---

If an application uses standard data-aware controls, then a user can navigate through a client dataset’s records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see “Navigating datasets” on page 18-5.

Unlike most datasets, client datasets can also position the cursor at a specific record in the dataset by using the *RecNo* property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a particular record number to make that record the current one.

### Limiting what records appear

---

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see “Displaying and editing a subset of data using filters” on page 18-12. For more information about ranges, see “Limiting records with ranges” on page 18-30.

With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because of this, the SQL dialect of the server limits what operations are used in filter strings. Client datasets implement their own filter support, which includes more operations than that of other datasets. For example, when using a client dataset, filter expressions can include string operators that return substrings, operators that parse date/time values, and much more. Client datasets also allow filters on BLOB fields or complex field types such as ADT fields and array fields.

The various operators and functions that client datasets can use in filters, along with a comparison to other datasets that support filters, is given below:

**Table 23.1** Filter support in client datasets

Operator or function	Example	Supported by other datasets	Comment
<b>Comparisons</b>			
=	State = 'CA'	Yes	
<>	State <> 'CA'	Yes	
>=	DateEntered >= '1/1/1998'	Yes	
<=	Total <= 100,000	Yes	
>	Percentile > 50	Yes	
<	Field1 < Field2	Yes	
BLANK	State <> 'CA' or State = BLANK	Yes	Blank records do not appear unless explicitly included in the filter.
IS NULL	Field1 IS NULL	No	
IS NOT NULL	Field1 IS NOT NULL	No	
<b>Logical operators</b>			
and	State = 'CA' and Country = 'US'	Yes	
or	State = 'CA' or State = 'MA'	Yes	
not	not (State = 'CA')	Yes	
<b>Arithmetic operators</b>			
+	Total + 5 > 100	Depends on driver	Applies to numbers, strings, or date (time) + number.
-	Field1 - 7 <> 10	Depends on driver	Applies to numbers, dates, or date (time) - number.
*	Discount * 100 > 20	Depends on driver	Applies to numbers only.
/	Discount > Total / 5	Depends on driver	Applies to numbers only.

**Table 23.1** Filter support in client datasets (continued)

Operator or function	Example	Supported by other datasets	Comment
<b>String functions</b>			
Upper	Upper(Field1) = 'ALWAYS'	No	
Lower	Lower(Field1 + Field2) = 'josp'	No	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	No	Value goes from position of second argument to end or number of chars in third argument. First char has position 1.
Trim	Trim(Field1 + Field2) Trim(Field1, '-')	No	Removes third argument from front and back. If no third argument, trims spaces.
TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '\$') <> "	No	See Trim.
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> "	No	See Trim.
<b>DateTime functions</b>			
Year	Year(DateField) = 2000	No	
Month	Month(DateField) <> 12	No	
Day	Day(DateField) = 1	No	
Hour	Hour(DateField) < 16	No	
Minute	Minute(DateField) = 0	No	
Second	Second(DateField) = 30	No	
GetDate	GetDate - DateField > 7	No	Represents current date and time.
Date	DateField = Date(GetDate)	No	Returns the date portion of a datetime value.
Time	TimeField > Time(GetDate)	No	Returns the time portion of a datetime value.
<b>Miscellaneous</b>			
Like	Memo LIKE '%filters%'	No	Works like SQL-92 without the ESC clause. When applied to BLOB fields, FilterOptions determines whether case is considered.
In	Day(DateField) in (1,7)	No	Works like SQL-92. Second argument is a list of values all with the same type.
*	State = 'M*'	Yes	Wildcard for partial comparisons.

When applying ranges or filters, the client dataset still stores all of its records in memory. The range or filter merely determines which records are available to controls that navigate or display data from the client dataset.

**Note** When fetching data from a provider, you can also limit the data that the client dataset stores by supplying parameters to the provider. For details, see “Limiting records with parameters” on page 23-28.

## Editing data

---

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset’s *Data* property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the *Delta* property. Using a change log serves two purposes:

- The change log is required for applying updates to a database server or external provider component.
- The change log provides sophisticated support for undoing changes.

The *LogChanges* property lets you disable logging. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in file-based applications if you do not want the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

**Note** Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

## Undoing changes

Even though a record’s original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record’s previous state:

- To remove the last change to a record, call *UndoLastChange*. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (*True*), or to leave the cursor on the current record (*False*). If there are several changes to a record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure. If the removal occurs, *UndoLastChange* returns *True*. Use the *ChangeCount* property to check whether there are more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.



- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call *RevertRecord*. *RevertRecord* removes any changes to the current record from the change log.
- To restore a deleted record, first set the *StatusFilter* property to *[usDeleted]*, which makes the deleted records “visible.” Next, navigate to the record you want to restore and call *RevertRecord*. Finally, restore the *StatusFilter* property to *[usModified, usInserted, usUnmodified]* so that the edited version of the dataset (now containing the restored record) is again visible.
- At any point during edits, you can save the current state of the change log using the *SavePoint* property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling *CancelUpdates*. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

## Saving changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client datasets stores its data in a file or represents data obtained through a provider. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. “Merging changes into data” on page 23-33 describes this process.

You can’t use *MergeChangeLog* if you are using the client dataset to cache updates or to represent the data from an external provider component. The information in the change log is required for resolving updated records with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which attempts to write the modifications to the database server or source dataset, and updates the *Data* property only when the modifications have been successfully committed. See “Applying updates” on page 23-19 for more information about this process.

## Constraining data values

---

Client datasets can enforce constraints on the edits a user makes to data. These constraints are applied when the user tries to post changes to the change log. You can always supply custom constraints. These let you provide your own, application-defined limits on what values users post to a client dataset.

In addition, when client datasets represent server data that is accessed using the BDE, they also enforce data constraints imported from the database server. If the client dataset works with an external provider component, the provider can control whether those constraints are sent to the client dataset, and the client dataset can control whether it uses them. For details on how the provider controls whether constraints are included in data packets, see “Handling server constraints” on page 24-12. For details on how and why client dataset can turn off enforcement of server constraints, see “Handling constraints from the server” on page 23-28.

## Specifying custom constraints

You can use the properties of the client dataset’s field components to impose your own constraints on what data users can enter. Each field component has two properties that you can use to specify constraints:

- The *DefaultExpression* property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset’s version takes precedence because it is assigned before the update is applied back to the database server or source dataset.
- The *CustomConstraint* property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints defined this way are applied in addition to any constraints imported from the server. For more information about working with custom constraints on field components, see “Creating a custom constraint” on page 19-21.

In addition, you can create record-level constraints using the client dataset’s *Constraints* property. *Constraints* is a collection of *TCheckConstraint* objects, where each object represents a separate condition. Use the *CustomConstraint* property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

## Sorting and indexing

---

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application set up relationships with other datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset represents server data or uses an external provider, it inherits a default index and sort order based on the data it receives. The default index is called `DEFAULT_ORDER`. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called `CHANGEINDEX`, on the changed records stored in the change log (*Delta* property). `CHANGEINDEX` orders all records in the client dataset as they would appear if the

changes specified in *Delta* were applied. `CHANGEINDEX` is based on the ordering inherited from `DEFAULT_ORDER`. As with `DEFAULT_ORDER`, you cannot change or delete the `CHANGEINDEX` index.

You can use other existing indexes, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets.

**Note** You may also want to review the material on indexes in table-type datasets, which also applies to client datasets. This material is in “Sorting records with indexes” on page 18-25 and “Limiting records with ranges” on page 18-30.

## Adding a new index

There are three ways to add indexes to a client dataset:

- To create a temporary index at runtime that sorts the records in the client dataset, you can use the *IndexFieldNames* property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index.

This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping. These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.

- To create an index at runtime that can be used for grouping, call *AddIndex*. *AddIndex* lets you specify the properties of the index, including
  - The name of the index. This can be used for switching indexes at runtime.
  - The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields.
  - How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can set options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order.
  - The default level of grouping support for the index.

Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can't call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

- The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call *CreateDataSet*. See “Creating and deleting tables” on page 18-37 for more information about creating client datasets.

As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others. Indexes created this way always persist and are saved when you save the client dataset to a file.

**Tip** You can index and sort on internally calculated fields with client datasets.

## Deleting and switching indexes

To remove an index you created for a client dataset, call *DeleteIndex* and specify the name of the index to remove. You cannot remove the `DEFAULT_ORDER` and `CHANGEINDEX` indexes.

To use a different index when more than one index is available, use the *IndexName* property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the Object Inspector.

## Using indexes to group data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the `SalesRep` and `Customer` fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the `SalesRep` column are duplicated. Within the records for `SalesRep` 1, adjacent values in the `Customer` column are duplicated. That is, the data is grouped by `SalesRep`, and within the `SalesRep` group it is grouped by `Customer`. Each grouping has an associated level. In this case, the `SalesRep` group has level 1 (because it is not nested in any other groups) and the `Customer` group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to display a field value only if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

To determine where the current record falls within any group, use the *GetGroupState* method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

## Representing calculated values

---

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record. For more information about using calculated fields, see "Defining a calculated field" on page 19-7.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields. For more information on internally calculated fields, see "Using internally calculated fields in client datasets" below.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates. For more information on maintained aggregates, see "Using maintained aggregates" on page 23-11.

### Using internally calculated fields in client datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an *OnCalcFields* event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnCalcFields* event handler. However, you can optimize your event handler by checking the *State* property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. Depending on whether you use persistent fields or field definitions, you do this in one of the following ways:

- If you use persistent fields, define fields as internally calculated by selecting *InternalCalc* in the Fields editor.
- If you use field definitions, set the *InternalCalcField* property of the relevant field definition to *True*.

**Note** Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the BDE or remote database server.

## Using maintained aggregates

---

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a “maintained aggregate.”

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

### Specifying aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the *Aggregates* property. *Aggregates* is a collection of aggregate specifications (*TAggregate*). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the *Add* method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

**Note** When you create aggregated fields, the appropriate aggregate objects are added to the client dataset’s *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields. For details on creating aggregated persistent fields, see “Defining an aggregate field” on page 19-10.

For each aggregate, the *Expression* property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in Table 23.2

**Table 23.2** Summary operators for maintained aggregates

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can’t nest summary operators, however.) You can create expressions by using operators on summarized

values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

```
Sum(Qty * Price)           {legal -- summary of an expression on fields }
Max(Field1) - Max(Field2) {legal -- expression on summaries }
Avg(DiscountRate) * 100   {legal -- expression of summary and constant }
Min(Sum(Field1))          {illegal -- nested summaries }
Count(Field1) - Field2    {illegal -- expression of summary and field }
```

## Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in the client dataset. However, you can specify that you want to summarize over the records in a group instead. This lets you provide intermediate summaries such as subtotals for groups of records that share a common field value.

Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping. See "Using indexes to group data" on page 23-9 for information on grouping support.

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName  := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

## Obtaining aggregate values

To get the value of a maintained aggregate, call the *Value* method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the *GetGroupState* method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification. The *AggFields* property contains the new aggregated field component, and the *FindField* method returns it.

## Copying data from another dataset

---

To copy the data from another dataset at design time, right click the client dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one whose data and structure you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

### Assigning data directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an *OleVariant*. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a client dataset that represents server data or that uses an external provider component, data packets are automatically assigned to *Data*.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```



**Note** When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

If you are copying from a dataset other than a client dataset, you can create a dataset provider component, link it to the source dataset, and then copy its data:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

**Note** When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

If you want to merge changes from another dataset, rather than copying its data, you must use a provider component. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

## Cloning a client dataset cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

*CloneCursor* takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

## Adding application-specific information to the data

---

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the *SetOptionalParam* method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the *GetOptionalParam* method, passing in the name that was used when the information was stored.

## Using a client dataset to cache updates

---

By default, when you edit data in most datasets, every time you delete or post a record, the dataset generates a transaction, deletes or writes that record to the database server, and commits the transaction. If there is a problem writing changes to the database, your application is notified immediately: the dataset raises an exception when you post the record.

If your dataset uses a remote database server, this approach can degrade performance due to network traffic between your application and the server every time you move to a new record after editing the current record. To minimize the network traffic, you may want to cache updates locally. When you cache updates, your application retrieves data from the database, caches and edits it locally, and then applies the cached updates to the database in a single transaction. When you cache updates, changes to a dataset (such as posting changes or deleting records) are stored locally instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also can't see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

Although the BDE and ADO provide alternate mechanisms for caching updates, using a client dataset for caching updates has several advantages:

- Applying updates when datasets are linked in master/detail relationships is handled for you. This ensures that updates to multiple linked datasets are applied in the correct order.
- Client datasets give you the maximum of control over the update process. You can set properties to influence the SQL that is generated for updating records, specify the table to use when updating records from a multi-table join, or even apply updates manually from a *BeforeUpdateRecord* event handler.
- When errors occur applying cached updates to the database server, only client datasets (and dataset providers) provide you with information about the current record value on the database server in addition to the original (unedited) value from your dataset and the new (edited) value of the update that failed.
- Client datasets let you specify the number of update errors you want to tolerate before the entire update is rolled back.

## Overview of using cached updates

---

To use cached updates, the following order of processes must occur in an application:

- 1 Indicate the data you want to edit.** How you do this depends on the type of client dataset you are using:
  - If you are using *TClientDataSet*, Specify the provider component that represent the data you want to edit. This is described in “Specifying a provider” on page 23-24.
  - If you are using a client dataset associated with a particular data access mechanism, you must
    - Identify the database server by setting the *DBConnection* property to an appropriate connection component.
    - Indicate what data you want to see by specifying the *CommandText* and *CommandType* properties. *CommandType* indicates whether *CommandText* is an SQL statement to execute, the name of a stored procedure, or the name of a table. If *CommandText* is a query or stored procedure, use the *Params* property to provide any input parameters.
    - Optionally, use the *Options* property to indicate whether nested detail sets and BLOB data should be included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) should be disabled, whether a single update can affect multiple server records, and whether the client dataset’s records are refreshed when it applies updates. *Options* is identical to a provider’s *Options* property. As a result, it allows you to set options that are not relevant or appropriate. For example, there is no reason to include *poIncFieldProps*, because the client dataset does not fetch its data from a dataset with persistent fields. Conversely, you do not want to exclude *poAllowCommandText*, which is included by default, because that would disable the *CommandText* property, which the client dataset uses to specify what data it wants. For information on the provider’s *Options* property, see “Setting options that influence the data packets” on page 24-5.
- 2 Display and edit the data,** permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory. This process is described in “Editing data” on page 23-5.
- 3 Fetch additional records as necessary.** By default, client datasets fetch all records and store them in memory. If a dataset contains many records or records with large BLOB fields, you may want to change this so that the client dataset fetches only enough records for display and re-fetches as needed. For details on how to control the record-fetching process, see “Requesting data from the source dataset or document” on page 23-25.
- 4 Optionally, refresh the records.** As time passes, other users may modify the data on the database server. This can cause the client dataset’s data to deviate more and more from the data on the server, increasing the chance of errors when you apply updates. To mitigate this problem, you can refresh records that have not already been edited. See “Refreshing records” on page 23-29 for details.

- 5 Apply the locally cached records to the database** or cancel the updates. For each record written to the database, a *BeforeUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache. For more information about applying updates to the database, see “Updating records” on page 23-19.

Instead of applying updates, an application can cancel the updates, emptying the change log without writing the changes to the database. You can cancel the updates by calling *CancelUpdates* method. All deleted records in the cache are undeleted, modified records revert to original values, and newly inserted record simply disappear.

## Choosing the type of dataset for caching updates

---

Delphi includes some specialized client dataset components for caching updates. Each client dataset is associated with a particular data access mechanism. These are listed in Table 23.3:

**Table 23.3** Specialized client datasets for caching updates

Client dataset	Data access mechanism
TBDEClientDataSet	Borland Database Engine
TSQLClientDataSet	dbExpress
TIBClientDataSet	InterBase Express

In addition, you can cache updates using the generic client dataset (*TClientDataSet*) with an external provider and source dataset. For information about using *TClientDataSet* with an external provider, see “Using a client dataset with a provider” on page 23-23.

**Note** The specialized client datasets associated with each data access mechanism actually use a provider and source dataset as well. However, both the provider and the source dataset are internal to the client dataset.

It is simplest to use one of the specialized client datasets to cache updates. However, there are times when it is preferable to use *TClientDataSet* with an external provider:

- If you are using a data access mechanism that does not have a specialized client dataset, you must use *TClientDataSet* with an external provider component. For example, if the data comes from an XML document or custom dataset.
- If you are working with tables that are related in a master/detail relationship, you should use *TClientDataSet* and connect it, using a provider, to the master table of two source datasets linked in a master/detail relationship. The client dataset sees the detail dataset as a nested dataset field. This approach is necessary so that updates to master and detail tables can be applied in the correct order.

- If you want to code event handlers that respond to the communication between the client dataset and the provider (for example, before and after the client dataset fetches records from the provider), you must use *TClientDataSet* with an external provider component. The specialized client datasets publish the most important events for applying updates (*OnReconcileError*, *BeforeUpdateRecord* and *OnGetTableName*), but do not publish the events surrounding communication between the client dataset and its provider, because they are intended primarily for multi-tiered applications.
- When using the BDE, you may want to use an external provider and source dataset if you need to use an update object. Although it is possible to code an update object from the *BeforeUpdateRecord* event handler of *TBDEClientDataSet*, it can be simpler just to assign the *UpdateObject* property of the source dataset. For information about using update objects, see “Using update objects to update a dataset” on page 20-39.

## Indicating what records are modified

---

While the user edits a client dataset, you may find it useful to provide feedback about the edits that have been made. This is especially useful if you want to allow the user to undo specific edits, for example, by navigating to them and clicking an “Undo” button.

The *UpdateStatus* method and *StatusFilter* properties are useful when providing feedback on what updates have occurred:

- *UpdateStatus* indicates what type of update, if any, has occurred for the current record. It can be any of the following values:
  - *usUnmodified* indicates that the current record is unchanged.
  - *usModified* indicates that the current record has been edited.
  - *usInserted* indicates a record that was inserted by the user.
  - *usDeleted* indicates a record that was deleted by the user.
- *StatusFilter* controls what type of updates in the change log are visible. *StatusFilter* works on cached records in much the same way as filters work on regular data. *StatusFilter* is a set, so it can contain any combination of the following values:
  - *usUnmodified* indicates an unmodified record.
  - *usModified* indicates a modified record.
  - *usInserted* indicates an inserted record.
  - *usDeleted* indicates a deleted record.

By default, *StatusFilter* is the set [*usModified*, *usInserted*, *usUnmodified*]. You can add *usDeleted* to this set to provide feedback about deleted records as well.

**Note** *UpdateStatus* and *StatusFilter* are also useful in *BeforeUpdateRecord* and *OnReconcileError* event handlers. For information about *BeforeUpdateRecord*, see “Intervening as updates are applied” on page 23-20. For information about *OnReconcileError*, see “Reconciling update errors” on page 23-22.

The following example shows how to provide feedback about the update status of records using the *UpdateStatus* method. It assumes that you have changed the

*StatusFilter* property to include *usDeleted*, allowing deleted records to remain visible in the dataset. It further assumes that you have added a calculated field to the dataset called “Status.”

```

procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
    with ClientDataSet1 do begin
        case UpdateStatus of
            usUnmodified: FieldByName('Status').AsString := '';
            usModified: FieldByName('Status').AsString := 'M';
            usInserted: FieldByName('Status').AsString := 'I';
            usDeleted: FieldByName('Status').AsString := 'D';
        end;
    end;
end;

```

## Updating records

---

The contents of the change log are stored as a data packet in the client dataset’s *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database (or source dataset or XML document).

When a client applies updates to the server, the following steps occur:

- 1 The client application calls the *ApplyUpdates* method of a client dataset object. This method passes the contents of the client dataset’s *Delta* property to the (internal or external) provider. *Delta* is a data packet that contains a client dataset’s updated, inserted, and deleted records.
- 2 The provider applies the updates, caching any problem records that it can’t resolve itself. See “Responding to client update requests” on page 24-8 for details on how the provider applies updates.
- 3 The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
- 4 The client dataset attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

## Applying updates

Changes made to the client dataset’s local copy of data are not sent to the database server (or XML document) until the client application calls the *ApplyUpdates* method. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the provider. (Note that, when using most client datasets, the provider is internal to the client dataset.)

*ApplyUpdates* takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is 0, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset’s change log remains intact. If *MaxErrors* is -1, any number of errors is tolerated, and

the change log contains all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

*ApplyUpdates* returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

- 1 It indirectly calls the provider's *ApplyUpdates* method. The provider's *ApplyUpdates* method writes the updates to the database, source dataset, or XML document and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
- 2 The client dataset's *ApplyUpdates* method then attempts to reconcile these problem records by calling the *Reconcile* method. *Reconcile* is an error-handling routine that calls the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see "Reconciling update errors" on page 23-22.
- 3 Finally, *Reconcile* removes successfully applied changes from the change log and updates *Data* to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

**Important** In some cases, the provider can't determine how to apply updates (for example, when applying updates from a stored procedure or multi-table join). Client datasets and provider components generate events that let you handle these situations. See "Intervening as updates are applied" below for details.

**Tip** If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. *TClientDataSet* receives a *BeforeApplyUpdates* event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), *TClientDataSet* receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

## Intervening as updates are applied

When a client dataset applies its updates, the provider determines how to handle writing the insertions, deletions, and modifications to the database server or source dataset. When you use *TClientDataSet* with an external provider component, you can use the properties and events of that provider to influence the way updates are applied. These are described in "Responding to client update requests" on page 24-8.

When the provider is internal, however, as it is for any client dataset associated with a data access mechanism, you can't set its properties or provide event handlers. As a result, the client dataset publishes one property and two events that let you influence how the internal provider applies updates.

- *UpdateMode* controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see "Influencing how updates are applied" on page 24-9.
- *OnGetTableName* lets you supply the provider with the name of the database table to which it should apply updates. This lets the provider generate the SQL statements for updates when it can't identify the database table from the stored procedure or query specified by *CommandText*. For example, if the query executes a multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates.

An *OnGetTableName* event handler has three parameters: the internal provider component, the internal dataset that fetched the data from the server, and a parameter to return the table name to use in the generated SQL.

- *BeforeUpdateRecord* occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.)

A *BeforeUpdateRecord* event handler has five parameters: the internal provider component, the internal dataset that fetched the data from the server, a delta packet that is positioned on the record that is about to be updated, an indication of whether the update is an insertion, deletion, or modification, and a parameter that returns whether the event handler performed the update. The use of these is illustrated in the following event handler. For simplicity, the example assumes the SQL statements are available as global variables that only need field values:

```

procedure TForm1.SQLClientDataSet1BeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
  var Applied Boolean);
var
  SQL: string;
  Connection: TSQLConnection;
begin
  Connection := (SourceDS as TCustomSQLDataSet).SQLConnection;
  case UpdateKind of
    ukModify:
      begin
        { 1st dataset: update Fields[1], use Fields[0] in where clause }
        SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
        Connection.Execute(SQL, nil, nil);
        { 2nd dataset: update Fields[2], use Fields[3] in where clause }
        SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
        Connection.Execute(SQL, nil, nil);
      end;
    ukDelete:
      begin
        { 1st dataset: use Fields[0] in where clause }
        SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
        Connection.Execute(SQL, nil, nil);
      end;
  end;

```



## Using a client dataset to cache updates

```
{ 2nd dataset: use Fields[3] in where clause }
SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
Connection.Execute(SQL, nil, nil);
end;
ukInsert:
begin
{ 1st dataset: values in Fields[0] and Fields[1] }
SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue, DeltaDS.Fields[1].NewValue]);
Connection.Execute(SQL, nil, nil);
{ 2nd dataset: values in Fields[2] and Fields[3] }
SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
Connection.Execute(SQL, nil, nil);
end;
end;
Applied := True;
end;
```

## Reconciling update errors

There are two events that let you handle errors that occur during the update process:

- During the update process, the internal provider generates an *OnUpdateError* event every time it encounters an update that it can't handle. If you correct the problem in an *OnUpdateError* event handler, then the error does not count toward the maximum number of errors passed to the *ApplyUpdates* method. This event only occurs for client datasets that use an internal provider. If you are using *TClientDataSet*, you can use the provider component's *OnUpdateError* event instead.
- After the entire update operation is finished, the client dataset generates an *OnReconcileError* event for every record that the provider could not apply to the database server.

You should always code an *OnReconcileError* or *OnUpdateError* event handler, even if only to discard the records returned that could not be applied. The event handlers for these two events work the same way. They include the following parameters:

- *DataSet*: A client dataset that contains the updated record which couldn't be applied. You can use this dataset's methods to get information about the problem record and to edit the record in order to correct any problems. In particular, you will want to use the *CurValue*, *OldValue*, and *NewValue* properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in your event handler.
- *E*: An object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.
- *UpdateKind*: The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).

- *Action*: A **var** parameter that indicates what action to take when the event handler exits. In your event handler, you set this parameter to
  - Skip this record, leaving it in the change log. (rrSkip or raSkip)
  - Stop the entire reconcile operation. (rrAbort or raAbort)
  - Merge the modification that failed into the corresponding record from the server. (rrMerge or raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
  - Replace the current update in the change log with the value of the record in the event handler, which has presumably been corrected. (rrApply or raCorrect)
  - Ignore the error completely. (rrIgnore) This possibility only exists in the *OnUpdateError* event handler, and is intended for the case where the event handler applies the update back to the database server. The updated record is removed from the change log and merged into *Data*, as if the provider had applied the update.
  - Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel) This possibility only exists in the *OnReconcileError* event handler.
  - Update the current record value to match the record on the server. (raRefresh) This possibility only exists in the *OnReconcileError* event handler.

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit which ships in the objrepos directory. (To use this dialog, add RecError to your uses clause.)

```

procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
    Action := HandleReconcileError(DataSet, UpdateKind, E);
end;

```

## Using a client dataset with a provider

---

A client dataset uses a provider to supply it with data and apply updates when

- It caches updates from a database server or another dataset.
- It represents the data in an XML document.
- It stores the data in the client portion of a multi-tiered application.

For any client dataset other than *TClientDataSet*, this provider is internal, and so not directly accessible by the application. With *TClientDataSet*, the provider is an external component that links the client dataset to an external source of data.

An external provider component can reside in the same application as the client dataset, or it can be part of a separate application running on another system. For more information about provider components, see Chapter 24, "Using provider components." For more information about applications where the provider is in a

separate application on another system, see Chapter 25, “Creating multi-tiered applications.”

When using an (internal or external) provider, the client dataset always caches any updates. For information on how this works, see “Using a client dataset to cache updates” on page 23-15.

The following topics describe additional properties and methods of the client dataset that enable it to work with a provider.

## Specifying a provider

---

Unlike the client datasets that are associated with a data access mechanism, *TClientDataSet* has no internal provider component to package data or apply updates. If you want it to represent data from a source dataset or XML document, therefore, you must associated the client dataset with an external provider component.

The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

- If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This works as long as the provider has the same *Owner* as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset’s *SetProvider* method

If you think you may eventually scale up to a remote provider, or if you want to make calls directly to the *IAppServer* interface, you can also set the *RemoteServer* property to a *TLocalConnection* component. If you use *TLocalConnection*, the *TLocalConnection* instance manages the list of all providers that are local to the application, and handles the client dataset’s *IAppServer* calls. If you do not use *TLocalConnection*, Delphi creates a hidden object that handles the *IAppServer* calls from the client dataset.

- If the provider is on a remote application server, then, in addition to the *ProviderName* property, you need to specify a component that connects the client dataset to the application server. There are two properties that can handle this task: *RemoteServer*, which specifies the name of a connection component from which to get a list of providers, or *ConnectionBroker*, which specifies a centralized broker that provides an additional level of indirection between the client dataset and the connection component. The connection component and, if used, the connection broker, reside in the same data module as the client dataset. The connection component establishes and maintains a connection to an application server, sometimes called a “data broker”. For more information, see “The structure of the client application” on page 25-4.

At design time, after you specify *RemoteServer* or *ConnectionBroker*, you can select a provider from the drop-down list for the *ProviderName* property in the Object

Inspector. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.

**Note** If the connection component is an instance of *TDCOMConnection*, the application server must be registered on the client machine.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

## Requesting data from the source dataset or document

---

Client datasets can control how they fetch their data packets from a provider. By default, they retrieve all records from the source dataset. This is true whether the source dataset and provider are internal components (as with *TBDEClientDataSet*, *TSQClientDataSet*, and *TIBClientDataSet*), or separate components that supply the data for *TClientDataSet*.

You can change how the client dataset fetches records using the *PacketRecords* and *FetchOnDemand* properties.

### Incremental fetching

By changing the *PacketRecords* property, you can specify that the client dataset fetches data in smaller chunks. *PacketRecords* specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
```

This process of fetching records in batches is called “incremental fetching”. Client datasets use incremental fetching when *PacketRecords* is greater than zero.

To fetch each batch of records, the client dataset calls *GetNextPacket*. Newly fetched packets are appended to the end of the data already in the client dataset. *GetNextPacket* returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

**Warning** Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server. See “Supporting state information in remote data modules” on page 25-19 for information on how to use incremental fetching with stateless remote data modules.

**Note** You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

## Fetch-on-demand

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is *True* (the default), the client dataset automatically fetches records as needed. To prevent automatic fetching of records, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *False*, the application must explicitly call *GetNextPacket* to fetch records.

For example, Applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the server.

The provider controls whether the records in data packets include BLOB data and nested detail datasets. If the provider excludes this information from records, the *FetchOnDemand* property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If *FetchOnDemand* is *False*, and the provider does not include BLOB data and detail datasets with records, you must explicitly call the *FetchBlobs* or *FetchDetails* method to retrieve this information.

## Getting parameters from the source dataset

---

There are two circumstances when the client dataset needs to fetch parameter values:

- The application needs the value of output parameters on a stored procedure.
- The application wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

Client datasets store parameter values in their *Params* property. These values are refreshed with any output parameters when the client dataset fetches data from the source dataset. However, there may be times a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the *FetchParams* method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

**Note** There is never a need to call *FetchParams* when the client dataset uses an internal provider and source dataset, because the *Params* property always reflects the parameters of the internal source dataset. With *TClientDataSet*, the *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a table-type dataset, there are no parameters to fetch.

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a

stateless application server, use the *Execute* method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

## Passing parameters to the source dataset

---

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Input parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the source dataset at design time or at runtime. At design time, select the client dataset and double-click the *Params* property in the Object Inspector. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the Object Inspector to edit the properties of that parameter.

At runtime, use the *CreateParam* method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code adds an input parameter named *CustNo* with a value of 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
  AsInteger := 605;
```

If the client dataset is not active, you can send the parameters to the application server and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

## Sending query or stored procedure parameters

When the client dataset's *CommandType* property is *ctQuery* or *ctStoredProc*, or, if the client dataset is a *TClientDataSet* instance, when the associated provider represents the results of a query or stored procedure, you can use the *Params* property to specify parameter values. When the client dataset requests data from the source dataset or uses its *Execute* method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

**Note** Parameter names should match the names of the corresponding parameters on the source dataset.

## Limiting records with parameters

If the client dataset is

- a *TClientDataSet* instance whose associated provider represents a *TTable* or *TSQLTable* component
- a *TSQLClientDataSet* or *TBDEClientDataSet* instance whose *CommandType* property is *ctTable*

then it can use the *Params* property to limit the records that it caches in memory. Each parameter represents a field value that must be matched before a record can be included in the client dataset's data. This works much like a filter, except that with a filter, the records are still cached in memory, but unavailable.

Each parameter name must match the name of a field. When using *TClientDataSet*, these are the names of fields in the *TTable* or *TSQLTable* component associated with the provider. When using *TSQLClientDataSet* or *TBDEClientDataSet*, these are the names of fields in the table on the database server. The data in the client dataset then includes only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named *CustID* (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset, it passes this parameter value. The provider then sends only the records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

## Handling constraints from the server

---

When a database server defines constraints on what data is valid, it is useful if the client dataset knows about them. That way, the client dataset can ensure that user edits never violate those server constraints. As a result, such violations are never passed to the database server where they would be rejected. This means fewer updates generate error conditions during the updating process.

Regardless of the source of data, you can duplicate such server constraints by explicitly adding them to the client dataset. This process is described in "Specifying custom constraints" on page 23-7.

It is more convenient, however, if the server constraints are automatically included in data packets. Then you need not explicitly specify default expressions and constraints, and the client dataset changes the values it enforces when the server constraints change. By default, this is exactly what happens: if the source dataset is aware of server constraints, the provider automatically includes them in data packets and the client dataset enforces them when the user posts edits to the change log.

**Note** Only datasets that use the BDE can import constraints from the server. This means that server constraints are only included in data packets when using *TBDEClientDataSet* or *TClientDataSet* with a provider that represents a BDE-based

dataset. For more information on how to import server constraints and how to prevent a provider from including them in data packets, see “Handling server constraints” on page 24-12.

**Note** For more information on working with the constraints once they have been imported, see “Using server constraints” on page 19-21.

While importing server constraints and expressions is an extremely valuable feature that helps an application preserve data integrity, there may be times when it needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value of a field, but the client dataset uses incremental fetching, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call the *DisableConstraints* method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset’s *EnableConstraints* method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

**Tip** Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

## Refreshing records

---

Client datasets work with an in-memory snapshot of the data from the source dataset. If the source dataset represents server data, then as time elapses other users may modify that data. The data in the client dataset becomes a less accurate picture of the underlying data.

Like any other dataset, client datasets have a *Refresh* method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client datasets can also update the data while leaving the change log intact. To do this, call the *RefreshRecord* method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

**Warning** It is not always appropriate to call *RefreshRecord*. If the user’s edits conflict with changes made to the underlying dataset by other users, calling *RefreshRecord* masks this conflict. When the client dataset applies its updates, no reconcile error occurs and the application can’t resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following *AfterScroll*



refreshes the current record every time the user moves to a new record (ensuring the most up-to-date value), but only when it is safe to do so.:

```

procedure TForm1.ClientDataSet1AfterScroll(DataSet: TDataSet);
begin
    if ClientDataSet1.UpdateStatus = usUnModified then
        ClientDataSet1.RefreshRecord;
end;

```

## Communicating with providers using custom events

---

Client datasets communicate with a provider component through a special interface called *IAppServer*. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server

*TClientDataSet* provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

- 1 The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
- 2 The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.
- 3 The provider goes through its normal process of assembling a data packet (including all the accompanying events).
- 4 The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.
- 5 The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between client dataset and provider.

In addition, the client dataset has a special method, *DataRequest*, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

## Overriding the source dataset

---

The client datasets that are associated with a particular data access mechanism use the *CommandText* and *CommandType* properties to specify the data they represent. When using *TClientDataSet*, however, the data is specified by the source dataset, not the client dataset. Typically, this source dataset has a property that specifies an SQL statement to generate the data or the name of a database table or stored procedure.

If the provider allows, *TClientDataSet* can override the property on the source dataset that indicates what data it represents. That is, if the provider permits, the client dataset's *CommandText* property replaces the property on the provider's dataset that specifies what data it represents. This allows *TClientDataSet* to specify dynamically what data it wants to see.

By default, external provider components do not let client datasets use the *CommandText* value in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider. Otherwise, the value of *CommandText* is ignored.

**Note** Never remove *poAllowCommandText* from the *Options* property of *TSQLClientDataSet*, *TBDEClientDataSet*, or *TIBClientDataSet*. The client dataset's *Options* property is forwarded to the internal provider, so removing *poAllowCommandText* prevents the client dataset from specifying what data to access.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the *AppServer* property. This property represents the interface through which the client dataset communicates with its provider.

## Using a client dataset with file-based data

---

Client datasets can work with dedicated files on disk as well as server data. This allows them to be used in file-based database applications and "briefcase model" applications. The special files that client datasets use for their data are called MyBase.

**Tip** All client datasets are appropriate for a briefcase model application, but for a pure MyBase application (one that does not use a provider), it is preferable to use *TClientDataSet*, because it involves less overhead.

In a pure MyBase application, the client application cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables

- Load saved data
- Merge edits into its data
- Save data

## Creating a new dataset

---

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset using persistent fields or field and index definitions. This follows the same scheme as creating any table-type dataset. See “Creating and deleting tables” on page 18-37 for details.
- You can copy an existing dataset (at design or runtime). See “Copying data from another dataset” on page 23-13 for more information about copying existing datasets.
- You can create a client dataset from an arbitrary XML document. See “Converting XML documents into data packets” on page 26-6 for details.

Once the dataset is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you start with the metadata for your client dataset already defined, making it easier to set up the user interface.

## Loading data from a file or stream

---

To load data from a file, call a client dataset’s *LoadFromFile* method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset’s data from the same file, you can use the *FileName* property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset’s *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream*) must have previously been saved in a client dataset’s data format by this or another client dataset using the *SaveToFile* (*SaveToStream*) method, or generated from an XML document. For more information about saving data to a file or stream, see “Saving data to a file or stream” on page 23-33. For information about creating client dataset data from an XML document, see Chapter 26, “Using XML in database applications.”

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property. However, the only indexes that are read from the file are those that were created with the dataset.

## Merging changes into data

---

When you edit the data in a client dataset, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the *MergeChangeLog* method. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, *Data* contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the *ChangeCount* property to 0.

**Warning** Do not call *MergeChangeLog* for client datasets that use a provider. In this case, call *ApplyUpdates* to write changes to the database. For more information, see “Applying updates” on page 23-19.

**Note** It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see “Assigning data directly” on page 23-13.

If you do not want to use the extended undo capabilities of the change log, you can set the client dataset’s *LogChanges* property to *False*. When *LogChanges* is *False*, edits are automatically merged when you post records and there is no need to call *MergeChangeLog*.

## Saving data to a file or stream

---

Even when you have merged changes into the data of a client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the *SaveToFile* method.

*SaveToFile* takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

**Note** *SaveToFile* does not preserve any indexes you added to the client dataset at runtime, only indexes that were added when you created the client dataset.

If you always save the data to the same file, you can use the *FileName* property instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the *SaveToStream* method. *SaveToStream* takes one parameter, a stream object that receives the data.

**Note** If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component on the application server.

## Using provider components

Provider components (*TDataSetProvider* and *TXMLTransformProvider*) supply the most common mechanism by which client datasets obtain their data. Providers

- Receive data requests from a client dataset (or XML broker), fetch the requested data, package the data into a transportable data packet, and return the data to the client dataset (or XML broker). This activity is called “providing.”
- Receive updated data from a client dataset (or XML broker), apply updates to the database server, source dataset, or source XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called “resolving.”

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in a dataset or XML document or to apply updates. However, provider components include a number of events and properties that allow your application more direct control over what information is packaged for clients and how your application responds to client requests.

When using *TBDEClientDataSet*, *TSQLClientDataSet*, or *TIBClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet* or *TXMLBroker*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. The client datasets that have internal providers surface some of the internal provider’s properties and events as their own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset (or XML broker), or it can reside on an application server as part of a multi-tiered application.

This chapter describes how to use a provider component to control the interaction with client datasets or XML brokers.

## Determining the source of data

---

When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. Depending on your version of Delphi, you can specify the source as one of the following:

- To provide the data from a dataset, use *TDataSetProvider*.
- To provide the data from an XML document, use *TXMLTransformProvider*.

### Using a dataset as the source of the data

---

If the provider is a dataset provider (*TDataSetProvider*), set the *DataSet* property of the provider to indicate the source dataset. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

*TDataSetProvider* interacts with the source dataset using the *IProviderSupport* interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don't do anything or that raise exceptions.

The dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, *dbExpress* datasets, and InterBase Express datasets) override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets don't add anything to the inherited *IProviderSupport* implementation, but can still be used as a source dataset as long as the *ResolveToDataSet* property of the provider is *True*.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider only provides data packets on a read-only basis (that is, if it does not apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

### Using an XML document as the source of the data

---

If the provider is an XML provider, set the *XMLDataFile* property of the provider indicate the source document.

XML providers must transform the source document into data packets, so in addition to indicating the source document, you must also specify how to transform that document into data packets. This transformation is handled by the provider's *TransformRead* property. *TransformRead* represents a *TXMLTransform* object. You can set its properties to specify what transformation to use, and use its events to provide your own input to the transformation. For more information on using XML providers, see "Using an XML document as the source for a provider" on page 26-8.

## Communicating with the client dataset

All communication between a provider and a client dataset or XML broker takes place through an *IAppServer* interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you, or by a *TLocalConnection* component. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module.

Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset or XML broker. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of a client dataset.

Table 24.1 lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an *OleVariant* parameter called *OwnerData* that allows a client dataset and a provider to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

**Table 24.1** AppServer interface members

<b>IAppServer</b>	<b>provider component</b>	<b>TClientDataSet</b>
AS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event.
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method.
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event.
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetParams event, AfterGetParams event.
AS_GetProviderNames method	Used to identify all available providers.	Used to create a design-time list for ProviderName property.
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event



## Choosing how to apply updates using a dataset provider

---

*TXMLTransformProvider* components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server).

However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you use may not support the use of SQL statements (for example if you are providing from a *TClientDataSet* component).

*TDataSetProvider* lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the *ResolveToDataSet* property. When this property is *True*, updates are applied to the dataset. When it is *False*, updates are applied directly to the underlying database server.

## Controlling what information is included in data packets

---

When working with a dataset provider, there are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets
- Setting options that influence the data packets
- Adding custom information to data packets

**Note** These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.

## Specifying what fields appear in data packets

---

When using a dataset provider, you can control what fields are included in data packets by creating persistent fields on the dataset that the provider uses to build data packets. The provider then includes only these fields. Fields whose values are generated dynamically by the source dataset (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields. For information about persistent fields, see “Persistent field components” on page 19-3.

If the client dataset will be editing the data and applying updates, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields provided only

to ensure uniqueness, set the *ProviderFlags* property for those fields to include *pfHidden*.

**Note** Including enough fields to avoid duplicate records is also a consideration when the provider's source dataset represents a query. You must specify the query so that it includes enough fields to ensure all records are unique, even if your application does not use all the fields.

## Setting options that influence the data packets

The *Options* property of a dataset provider lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

**Table 24.2** Provider options

Value	Meaning
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poReadOnly	The client dataset can't apply updates to the provider.
poDisableEdits	Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises exception. (This does not affect the client dataset's ability to insert or delete records).
poDisableInserts	Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data)
poDisableDeletes	Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records)
poFetchBlobsOnDemand	BLOB field values are not included in data packets. Instead, client datasets must request these values on an as-needed basis. If the client dataset's <i>FetchOnDemand</i> property is <i>True</i> , it requests these values automatically. Otherwise, the application must call the client dataset's <i>FetchBlobs</i> method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider's dataset represents the master of a master/detail relationship, nested detail values are not included in data packets. Instead, client datasets request these on an as-needed basis. If the client dataset's <i>FetchOnDemand</i> property is <i>True</i> , it requests these values automatically. Otherwise, the application must call the client dataset's <i>FetchDetails</i> method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): <i>Alignment</i> , <i>DisplayLabel</i> , <i>DisplayWidth</i> , <i>Visible</i> , <i>DisplayFormat</i> , <i>EditFormat</i> , <i>MaxValue</i> , <i>MinValue</i> , <i>Currency</i> , <i>EditMask</i> , <i>DisplayValues</i> .

**Table 24.2** Provider options (continued)

Value	Meaning
poCascadeDeletes	When the provider's dataset represents the master of a master/detail relationship, the server automatically deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.
poCascadeUpdates	When the provider's dataset represents the master of a master/detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poNoReset	Client datasets can't specify that the provider should reposition the cursor on the first record before providing data.
poPropagateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client can override the associated dataset's SQL text or the name of the table or stored procedure it represents.
poRetainServerOrder	The client dataset should not re-sort the records in the dataset to enforce a default order.

---

## Adding custom information to data packets

---

Dataset providers can add application-defined information to data packets using the *OnGetDataSetProperties* event. This information is encoded as an *OleVariant*, and stored under a name you specify. Client datasets can then retrieve the information using their *GetOptionalParam* method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client dataset may never be aware of the information, but the provider can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Add multiple attributes by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number

of records in the source dataset. Only the time the data was provided is returned when client datasets apply updates:

```

procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
    Properties := VarArrayCreate([0,1], varVariant);
    Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
    Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;

```

When the client dataset applies updates, the time the original records were provided can be read in the provider's *OnUpdateData* event:

```

procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
    WhenProvided: TDateTime;
begin
    WhenProvided := DataSet.GetOptionalParam('TimeProvided');
    ...
end;

```

## Responding to client data requests

---

Usually client requests for data are handled automatically. A client dataset or XML broker requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset or XML document, creating a data packet, and sending the packet to the client.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client, write an *OnGetData* event handler. *OnGetData* event handlers provide the data packet as a parameter in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client.

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers. For a discussion of persistent state information in application servers, see "Supporting state information in remote data modules" on page 25-19.

## Responding to client update requests

---

A provider applies updates to database records based on a *Delta* data packet received from a client dataset or XML broker. The client requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *ApplyUpdates*. This communication takes place using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers. For a discussion of persistent state information in application servers, see “Supporting state information in remote data modules” on page 25-19.

If you are using a dataset provider, a number of additional events allow you more control:

When a dataset provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdateData* event, the provider writes the changes to the database or source dataset.

The provider performs the update on a record-by-record basis. Before the dataset provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error. Usually errors occur because the change violates a server constraint or a database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset’s request to apply updates.

Update errors can be processed by either the dataset provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can’t resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won’t be using. To filter the client dataset on the update status of its records, set its *StatusFilter* property.

**Note** Applications must supply extra support when the updates are directed at a dataset that does not represent a single table. For details on how to do this, see “Applying updates to datasets that do not represent a single table” on page 24-11.

## Editing delta packets before updating the database

---

Before a dataset provider applies updates to the database, it generates an *OnUpdateData* event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the *UpdateStatus* property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in Table 24.3.

**Table 24.3** UpdateStatus values

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
  end;
end;
```

## Influencing how updates are applied

---

The *OnUpdateData* event also gives your dataset provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the *UpdateMode* property of the provider. *UpdateMode* can be assigned any of the following values:

**Table 24.4** UpdateMode values

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereKeyOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the *ProviderFlags* property. *ProviderFlags* is a set that can include any of the values in Table 24.5

**Table 24.5** ProviderFlags values

Value	Description
pfInWhere	The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when <i>UpdateMode</i> is <i>upWhereAll</i> or <i>upWhereChanged</i> .
pfInUpdate	The field appears in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of generated statements when <i>UpdateMode</i> is <i>upWhereKeyOnly</i> .
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

**Note** You can use the *UpdateFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

## Screening individual updates

---

Immediately before each update is applied, a dataset provider receives a *BeforeUpdateRecord* event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

## Resolving update errors on the provider

---

When an error condition arises as the dataset provider tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the *NewValue*, *OldValue*, and *CurValue* properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

## Applying updates to datasets that do not represent a single table

---

When a dataset provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as table-type datasets or "live" *TQuery* components. Automatic updates are a problem however, if the provider must apply updates to the data underlying a stored procedure with a result set or a multi-table query. There is no easy way to obtain the name of the table to which updates should be applied.



If the query or stored procedure is a BDE-enabled dataset (*TQuery* or *TStoredProc*) and it has an associated update object, the provider uses the update object. However, if there is no update object, you can supply the table name programmatically in an *OnGetTableName* event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the *BeforeUpdateRecord* event of the provider. Once this event handler has applied an update, you can set the event handler's *Applied* parameter to *True* so that the provider does not generate an error.

**Note** If the provider is associated with a BDE-enabled dataset, you can use an update object in the *BeforeUpdateRecord* event handler to apply updates using customized SQL statements. See "Using update objects to update a dataset" on page 20-39 for details.

## Responding to client-generated events

---

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the *OnDataRequest* event.

*OnDataRequest* is not part of the normal functioning of the provider. It is simply a hook to allow your client datasets to communicate directly with providers. The event handler takes an *OleVariant* as an input parameter and returns an *OleVariant*. By using *OleVariants*, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, the client application calls the *DataRequest* method of the client dataset.

## Handling server constraints

---

Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints:

- NOT NULL, to guarantee that a value supplied to a column has a value.
- NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record.
- CHECK, to guarantee that a value supplied to a column falls within a certain range, or is one of a limited number of possible values.
- CONSTRAINT, a table-wide check constraint that applies to multiple columns.

- PRIMARY KEY, to designate one or more columns as the table's primary key for indexing purposes.
- FOREIGN KEY, to designate one or more columns in a table that reference another table.

**Note** This list is not exclusive. Your database server may support some or all of these constraints in part or in whole, and may support additional constraints. For more information about supported constraints, see your server documentation.

Database server constraints obviously duplicate many kinds of data checks that traditional desktop database applications manage. You can take advantage of server constraints in multi-tiered database applications without having to duplicate the constraints in application server or client application code.

If the provider is working with a BDE-enabled dataset, the *Constraints* property lets you replicate and apply server constraints to data passed to and received from client datasets. When *Constraints* is *True* (the default), server constraints stored in the source dataset are included in data packets and affect client attempts to update data.

**Important** Before the provider can pass constraint information on to client datasets, it must retrieve the constraints from the database server. To import database constraints from the server, use SQL Explorer to import the database server's constraints and default expressions into the Data Dictionary. Constraints and default expressions in the Data Dictionary are automatically made available to BDE-enabled datasets.

There may be times when you do not want to apply server constraints to data sent to a client dataset. For example, a client dataset that receives data in packets and permits local updating of records prior to fetching more records may need to disable some server constraints that might be triggered because of the temporarily incomplete set of data. To prevent constraint replication from the provider to a client dataset, set *Constraints* to *False*. Note that client datasets can disable and enable constraints using the *DisableConstraints* and *EnableConstraints* methods. For more information about enabling and disabling constraints from the client dataset, see "Handling constraints from the server" on page 23-28.



## Creating multi-tiered applications

This chapter describes how to create a multi-tiered, client/server database application. A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the “three-tiered model,” a multi-tiered application is partitioned into thirds:

- **Client application:** provides a user interface on the user’s machine.
- **Application server:** resides in a central networking location accessible to all clients and provides common data services.
- **Remote database server:** provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a “data broker.” With Delphi you usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Delphi’s support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. This chapter focuses on creating a three-tiered database application. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

## Advantages of the multi-tiered database model

---

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier.** Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications.** Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the Borland Database Engine and the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- **Distributed data processing.** Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security.** You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTP, CORBA, or COM+, you can take advantage of the security models they support.

## Understanding provider-based multi-tiered applications

---

Delphi's support for multi-tiered applications use the components on the DataSnap page and the Data Access page of the component palette, plus a remote data module that is created by a wizard on the Multitier page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets.

The components needed for a multi-tiered application are described in Table 25.1:

**Table 25.1** Components used in multi-tiered applications

Component	Description
Remote data modules	Specialized data modules that can act as a COM Automation server, SOAP server, or CORBA server to give client applications access to any providers they contain. Used on the application server.
Provider component	A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
Client dataset component	A specialized dataset that uses <i>midas.dll</i> or <i>midaslib.dcu</i> to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server.
Connection components	A family of components that locate the server, form connections, and make the <i>IAppServer</i> interface available to client datasets. Each connection component is specialized to use a particular communications protocol.

The provider and client dataset components require *midas.dll* or *midaslib.dcu*, which manages datasets stored as data packets. (Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using *midas.dll*, you must deploy it on both application server and client application.)

If you are using BDE-enabled datasets, the application server may also require SQL Explorer to help in database administration and to import server constraints into the Data Dictionary so that they can be checked at any level of the multi-tiered application.

**Note** You must purchase server licenses for deploying your application server.

An overview of the architecture into which these components fit is described in “Using a multi-tiered architecture” on page 14-12.

## Overview of a three-tiered application

The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application:

- 1 A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface from the application server.
- 2 The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).
- 3 The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, field display characteristics) can be included in the metadata of the data packet. This process of packaging data into data packets is called “providing.”

- 4 The client decodes the data packet and displays the data to the user.
- 5 As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.
- 6 Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.
- 7 The application server decodes the package and posts updates (in the context of a transaction if appropriate). If a record can't be posted (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client's changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called "resolving."
- 8 When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.
- 9 The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.
- 10 The client refreshes its data from the server.

## The structure of the client application

---

To the end user, the client application of a multi-tiered application looks and behaves no differently than a traditional two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a *TClientDataSet* component. For detailed information about using the properties, events, and methods of client datasets, see Chapter 23, "Using client datasets."

*TClientDataSet* fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Chapter 24, "Using provider components". For details about client dataset features that facilitate its communication with a provider, see "Using a client dataset with a provider" on page 23-23

The client dataset communicates with the provider through the *IAppServer* interface. It gets this interface from a connection component. The connection component establishes the connection to the application server. Different connection components

are available for using different communications protocols. These connection components are summarized in the following table:

**Table 25.2** Connection components

Component	Protocol
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP (HTTP and XML)
TCorbaConnection	CORBA (IIOP)

**Note** Delphi also includes a connection component that does not connect to an application server at all, but instead supplies an *IAppServer* interface for client datasets to use when communicating with providers in the same application. This component, *TLocalConnection*, is not required, but makes it easier to later scale up to a multi-tiered application.

For more information about using connection components, see “Connecting to the application server” on page 25-23.

## The structure of the application server

When you set up and run an application server, it does not establish any connection with client applications. Instead, connection is initiated and maintained by client applications. The client application uses its connection component to establish a connection to the application server, which it uses to communicate with its selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

The basis of an application server is a remote data module, which is a specialized data module that supports the *IAppServer* interface. Client applications use the *IAppServer* interface to communicate with providers on the application server.

There are four types of remote data modules:

- **TRemoteDataModule:** This is a dual-interface Automation server. Use this type of remote data module if clients use DCOM, HTTP, sockets, or OLE to connect to the application server, unless you want to install the application server with MTS.
- **TMTSDataModule:** This is a dual-interface Automation server. Use this type of remote data module if you are creating the application server as an Active Library (.DLL) that is installed with MTS or COM+. You can use MTS remote data modules with DCOM, HTTP, sockets, or OLE.
- **TCorbaDataModule:** This is a CORBA server. Use this type of remote data module to provide data to CORBA clients.
- **TSoapDataModule:** This is a data module that implements an *IAppServer* descendant as an invocable interface in a Web Service application. Use this type of remote data module to provide data to clients that access data as a Web Service.



If the application server is to be deployed under MTS or COM+, the remote data module includes events for when the application server is activated or deactivated. This allows it to acquire database connections when activated and release them when deactivated.

## The contents of the remote data module

As with any data module, you can include any nonvisual component in the remote data module. There are certain components, however, that you must include:

- If the remote data module is exposing information from a database server, it must include a dataset component to represent the records from that database server. Other components, such as a database connection component of some type, may be required to allow the dataset to interact with a database server. For information about datasets, see Chapter 18, “Understanding datasets.” For information about database connection components, see Chapter 17, “Connecting to databases.”

For every dataset that the remote data module exposes to clients, it must include a dataset provider. A dataset provider packages data into data packets that are sent to client datasets and applies updates received from client datasets back to a source dataset or a database server. For more information about dataset providers, see Chapter 24, “Using provider components.”

- For every XML document that the remote data module exposes to clients, it must include an XML provider. An XML provider acts like a dataset provider, except that it fetches data from and applies updates to an XML document rather than a database server. For more information about XML providers, see “Using an XML document as the source for a provider” on page 26-8.

**Note** Do not confuse database connection components, which connect datasets to a database server, with the connection components used by client applications in a multi-tiered application. The connection components in multi-tiered applications can be found on the DataSnap page of the Component palette.

## Using transactional data modules

You can write an application server that takes advantage of special services for distributed applications that are supplied by MTS (before Windows 2000) or COM+ (under Windows 2000 and later). To do so, you create a transactional data module instead of an ordinary remote data module.

When you use a transactional data module, your application can take advantage of the following special services:

- **Security.** MTS and COM+ provide role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module’s interface. The MTS data module implements the *IsCallerInRole* method, which you let you check the role of the currently connected client and conditionally allow certain functions based on that role. For more information about MTS and COM+ security, see “Role-based security” on page 39-14.
- **Database handle pooling.** Transactional data modules automatically pool database connections that are made via ADO or (if you are using MTS and turn on MTS POOLING) the BDE. When one client is finished with a database connection,

another client can reuse it. This cuts down on network traffic, because your middle tier does not need to log off of the remote database server and then log on again. When pooling database handles, your database connection component should set its *KeepConnection* property to *False*, so that your application maximizes the sharing of connections. For more information about pooling database handles, see “Database resource dispensers” on page 39-5.

- **Transactions.** When using a transactional data module, you can provide enhanced transaction support beyond that available with a single database connection. Transactional data modules can participate in transactions that span multiple databases, or include functions that do not involve databases at all. For more information about the transaction support provided by transactional objects such as transactional data modules, see “Managing transactions in multi-tiered applications” on page 25-18.
- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your server so that remote data module instances are activated and deactivated on an as-needed basis. When using just-in-time activation and as-soon-as-possible deactivation, your remote data module is instantiated only when it is needed to handle client requests. This prevents it from tying up resources such as database handles when they are not in use.

Using just-in-time activation and as-soon-as-possible deactivation provides a middle ground between routing all clients through a single remote data module instance, and creating a separate instance for every client connection. With a single remote data module instance, the application server must handle all database calls through a single database connection. This acts as a bottleneck, and can impact performance when there are many clients. With multiple instances of the remote data module, each instance can maintain a separate database connection, thereby avoiding the need to serialize database access. However, this monopolizes resources because other clients can’t use the database connection while it is associated with another client’s remote data module.

To take advantage of transactions, just-in-time activation, and as-soon-as-possible deactivation, remote data module instances must be stateless. This means you must provide additional support if your client relies on state information. For example, the client must pass information about the current record when performing incremental fetches. For more information about state information and remote data modules in multi-tiered applications, see “Supporting state information in remote data modules” on page 25-19.

By default, all automatically generated calls to a transactional data module are transactional (that is, they assume that when the call exits, the data module can be deactivated and any current transactions committed or rolled back). You can write a transactional data module that depends on persistent state information by setting the *AutoComplete* property to *False*, but it will not support transactions, just-in-time activation, or as-soon-as-possible deactivation unless you use a custom interface.

**Warning** Application servers containing transactional data modules should not open database connections until the data module is activated. While developing your application, be sure that all datasets are not active and the database is not connected before running your application. In the application itself, add code to open database connections when the data module is activated and close them when it is deactivated.

## Pooling remote data modules

Object pooling allows you to create a cache of remote data modules that are shared by their clients, thereby conserving resources. How this works depends on the type of remote data module and on the connection protocol.

If you are creating a transactional data module that will be installed to COM+, you can use the COM+ Component Manager to install the application server as a pooled object. See “Object pooling” on page 39-8 for details.

Even if you are not using a transactional data module, you can take advantage of object pooling if the connection is formed using HTTP (*TWebConnection*). Under this second type of object pooling, you limit the number of instances of your remote data module that are created. This limits the number of database connections that you must hold, as well as any other resources used by the remote data module.

When the Web Server application (which passes calls to your remote data module) receives client requests, it passes them on to the first available remote data module in the pool. If there is no available remote data module, it creates a new one (up to a maximum number that you specify). This provides a middle ground between routing all clients through a single remote data module instance (which can act as a bottleneck), and creating a separate instance for every client connection (which can consume many resources).

If a remote data module instance in the pool does not receive any client requests for a while, it is automatically freed. This prevents the pool from monopolizing resources unless they are used.

To set up object pooling when using a Web connection (HTTP), your remote data module must override the *UpdateRegistry* method. In the overridden method, call *RegisterPooled* when the remote data module registers and *UnregisterPooled* when the remote data module unregisters. When using either method of object pooling, your remote data module must be stateless. This is because a single instance potentially handles requests from several clients. If it relied on persistent state information, clients could interfere with each other. See “Supporting state information in remote data modules” on page 25-19 for more information on how to ensure that your remote data module is stateless.

## Choosing a connection protocol

---

Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future development plans.

### Using DCOM connections

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server. However, because DCOM is not included with Windows 95, some older client machines may not have DCOM installed.

DCOM provides the only approach that lets you use security services when writing a transactional data module. These security services are based on assigning roles to the callers of transactional objects. When using DCOM, DCOM identifies the caller to the system that calls your application server (MTS or COM+). Therefore, it is possible to accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes COM calls into the application server on behalf of the client. Because of this, it is impossible to assign roles to separate clients: The runtime executable is, effectively, the only client. For more information about security and transactional objects, see “Role-based security” on page 39-14.

## Using Socket connections

TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application, you can't be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about sockets, see Chapter 32, “Working with sockets.”

Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe on the server are responsible for marshaling *IAppServer* calls.

**Note** ScktSrvr.exe can run as an NT service application. Register it with the Service manager by starting it using the -install command line option. You can unregister it using the -uninstall command line option.

Before you can use a socket connection, the application server must register its availability to clients using a socket connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableSocketTransport* in the *UpdateRegistry* method. You can remove this call to prevent socket connections to your application server.

**Note** Because older servers did not add this registration, you can disable the check for whether an application server is registered by unchecking the Connections | Registered Objects Only menu item on ScktSrvr.exe.

When using sockets, there is no protection on the server against client systems failing before they release a reference to interfaces on the application server. While this results in less message traffic than when using DCOM (which sends periodic keep-alive messages), this can result in an application server that can't release its resources because it is unaware that the client has gone away.

## Using Web connections

HTTP lets you create clients that can communicate with an application server that is protected by a firewall. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Chapter 27, “Creating Internet applications.”

Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) that accepts client requests and instantiates the remote data module using COM. Because of this, they are also called Web connections. The connection component on the client and httpsrvr.dll on the server are responsible for marshaling *IAppServer* calls.

Web connections can take advantage of the SSL security provided by wininet.dll (a library of Internet utilities that runs on the client system). Once you have configured the Web server on the server system to require authentication, you can specify the user name and password using the properties of the Web connection component.

As an additional security measure, the application server must register its availability to clients using a Web connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableWebTransport* in the *UpdateRegistry* method. You can remove this call to prevent Web connections to your application server.

Web connections can take advantage of object pooling. This allows your server to create a limited pool of remote data module instances that are available for client requests. By pooling the remote data modules, your server does not consume the resources for the data module and its database connection except when they are needed. For more information on object pooling, see “Pooling remote data modules” on page 25-8.

Unlike most other connection components, you can't use callbacks when the connection is formed via HTTP.

## Using SOAP connections

SOAP is the protocol that underlies Delphi's support for Web Service applications. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol.

SOAP connections have the advantage that they work in cross-platform applications because they are supported on both the Windows and Linux. Because SOAP connections use HTTP, they have the same advantages as Web connections: HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a “firewall”. For more information about using SOAP to distribute applications in Delphi, see Chapter 31, “Using Web Services.”

As with HTTP connections, you can't use callbacks when the connection is formed via SOAP. SOAP connections also limit you to a single remote data module in the application server.

## Using CORBA connections

CORBA lets you integrate your multi-tiered database applications into an environment that is standardized on CORBA. For example, when working with a client application written in Java, only the CORBA connection is available. Because CORBA (and Java) is available on multiple platforms, this allows you to write cross-platform multi-tiered applications.

By using CORBA, your application automatically gets the benefits of load-balancing, location transparency, and fail-over from the ORB runtime software. In addition, you can add hooks to take advantage of other CORBA services.

## Building a multi-tiered application

---

The general steps for creating a multi-tiered database application are

- 1 Create the application server.
- 2 Register or install the application server.
- 3 Create a client application.

The order of creation is important. You should create and run the application server before you create a client. At design time, you can then connect to the application server to test your client. You can, of course, create a client without specifying the application server at design time, and only supply the server name at runtime. However, doing so prevents you from seeing if your application works as expected when you code at design time, and you will not be able to choose servers and providers using the Object Inspector.

**Note** If you are not creating the client application on the same system as the server, and you are using a DCOM connection, you may want to register the application server on the client system. This makes the connection component aware of the application server at design time so that you can choose server names and provider names from a drop-down list in the Object Inspector. (If you are using a Web connection, SOAP connection, or socket connection, the connection component fetches the names of registered servers from the server machine.)

## Creating the application server

---

You create an application server very much as you create most database applications. The major difference is that the application server uses a remote data module.

To create an application server, follow these steps:

- 1 Start a new project:
  - If you are using SOAP as a transport protocol, this should be a new Web Service application. Choose File | New | Other, and on the Web Services page of the new items dialog, choose Web Service application.
  - For any other transport protocol, you need only choose File | New | Application.Save the new project.
- 2 Add a new remote data module to the project. From the main menu, choose File | New | Other, and on the Multitier page of the new items dialog, select
  - **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, or sockets.

- **Transactional Data Module** if you are creating a remote data module that runs under MTS or COM+. Connections can be formed using DCOM, HTTP, or sockets. However, only DCOM supports the security services.
- **CORBA Data Module** if you are creating a CORBA server.
- **SOAP Data Module** if you are creating a SOAP server in a Web Service application.

For more detailed information about setting up a remote data module, see “Setting up the remote data module” on page 25-13.

**Note**

Remote data modules are more than simple data modules. The CORBA data module acts as a CORBA server. The SOAP data module implements an invocable interface in a Web Service application. Other data modules are COM Automation objects.

- 3 Place the appropriate dataset components on the data module and set them up to access the database server.
- 4 Place a *TDataSetProvider* component on the data module for each dataset. This provider is required for brokering client requests and packaging data. Set the *DataSet* property for each provider component to the name of the dataset to access. You can set additional properties for the provider. See Chapter 24, “Using provider components” for more detailed information about setting up a provider.

If you are working with data from XML documents, you can use a *TXMLTransformProvider* component instead of a dataset and *TDataSetProvider* component. When using *TXMLTransformProvider*, set the *XMLDataFile* property to specify the XML document from which data is provided and to which updates are applied.

- 5 Write application server code to implement events, shared business rules, shared data validation, and shared security. When writing this code, you may want to
  - Extend the application server’s interface to provide additional ways for the client application to call the server. Extending the application server’s interface is described in “Extending the application server’s interface” on page 25-16.
  - Provide transaction support beyond the transactions automatically created when applying updates. Transaction support in multi-tiered database applications is described in “Managing transactions in multi-tiered applications” on page 25-18.
  - Create master/detail relationships between the datasets in your application server. Master/detail relationships are described in “Supporting master/detail relationships” on page 25-19.
  - Ensure your application server is stateless. Handling state information is described in “Supporting state information in remote data modules” on page 25-19.
  - Divide your application server into multiple remote data modules. Using multiple remote data modules is described in “Using multiple remote data modules” on page 25-21.

- 6 Save, compile, and register or install the application server. Registering an application server is described in “Registering the application server” on page 25-22.
- 7 If your server application does not use DCOM or SOAP, you must install the runtime software that receives client messages, instantiates the remote data module, and marshals interface calls.
  - For TCP/IP sockets this is a socket dispatcher application, `Scktsrvr.exe`.
  - For HTTP connections this is `httpsrvr.dll`, an ISAPI/NSAPI DLL that must be installed with your Web server.
  - For CORBA, this is the VisiBroker ORB.

## Setting up the remote data module

---

When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See “The structure of the application server” on page 25-5 for information on what type of remote data module you need.

### Configuring `TRemoteDataModule`

To add a `TRemoteDataModule` component to your application, choose File | New | Other and select Remote Data Module from the Multitier page of the new items dialog. You will see the Remote Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of `TRemoteDataModule` that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name `MyDataServer`, the wizard creates a new unit declaring `TMyDataServer`, a descendant of `TRemoteDataModule`, which implements `IMyDataServer`, a descendant of `IAppServer`.

**Note** You can add your own properties and methods to the new interface. For more information, see “Extending the application server’s interface” on page 25-16.

You must specify the threading model in the Remote Data Module wizard. You can choose Single-threaded, Apartment-threaded, Free-threaded, or Both.

- If you choose Single-threaded, COM ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment-threaded, COM ensures that any instance of your remote data module services one request at a time. When writing code in an Apartment-threaded library, you must guard against thread conflicts if you use global variables or objects not contained in the remote data module. This is the recommended model if you are using BDE-enabled datasets. (Note that you will need a session component with its `AutoSessionName` property set to `True` to handle threading issues on BDE-enabled datasets).



- If you choose Free-threaded, your application can receive simultaneous client requests on several threads. You are responsible for ensuring your application is thread-safe. Because multiple clients can access your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables. This is the recommended model if you are using ADO datasets.
- If you choose Both, your library works the same as when you choose Free-threaded, with one exception: all callbacks (calls to client interfaces) are serialized for you.
- If you choose Neutral, the remote data module can receive simultaneous calls on separate threads, as in the Free-threaded model, but COM guarantees that no two threads access the same method at the same time.

If you are creating an EXE, you must also specify what type of instancing to use. You can choose Single instance or Multiple instance (Internal instancing applies only if the client code is part of the same process space.)

- If you choose Single instance, each client connection launches its own instance of the executable. That process instantiates a single instance of the remote data module, which is dedicated to the client connection.
- If you choose Multiple instance, a single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

## Configuring TMTSDataModule

To add a *TMTSDataModule* component to your application, choose File | New | Other and select Transactional Data Module from the Multitier page of the new items dialog. You will see the Transactional Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

**Note** You can add your own properties and methods to your new interface. For more information, see “Extending the application server’s interface” on page 25-16.

You must specify the threading model in the Transactional Data Module wizard. Choose Single, Apartment, or Both.

- If you choose Single, client requests are serialized so that your application services only one at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment, the system ensures that any instance of your remote data module services one request at a time, and calls always use the same thread. You must guard against thread conflicts if you use global variables or objects not contained in the remote data module. Instead of using global variables, you can use the shared property manager. For more information on the shared property manager, see “Shared property manager” on page 39-6.

- If you choose Both, MTS calls into the remote data module's interface in the same way as when you choose Apartment. However, any callbacks you make to client applications are serialized, so that you don't need to worry about them interfering with each other.

**Note** The Apartment model under MTS or COM+ is different from the corresponding model under DCOM.

You must also specify the transaction attributes of your remote data module. You can choose from the following options:

- Requires a transaction. When you select this option, every time a client uses your remote data module's interface, that call is executed in the context of a transaction. If the caller supplies a transaction, a new transaction need not be created.
- Requires a new transaction. When you select this option, every time a client uses your remote data module's interface, a new transaction is automatically created for that call.
- Supports transactions. When you select this option, your remote data module can be used in the context of a transaction, but the caller must supply the transaction when it invokes the interface.
- Does not support transactions. When you select this option, your remote data module can't be used in the context of transactions.

## Configuring TSoapDataModule

To add a *TSoapDataModule* component to your application, choose File | New | Other and select SOAP Data Module from the Multitier page of the new items dialog. The SOAP data module wizard appears.

You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendant that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TSoapDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

You may want to edit the definitions of the generated interface and *TSoapDataModule* descendant, adding your own properties and methods. These properties and methods are not called automatically, but client applications that request your new interface by name can use any of the properties and methods that you add.

**Note** To use *TSoapDataModule*, the new data module should be added to a Web Service application. The *IAppServer* interface is an invocable interface, which is registered in the initialization section of the new unit. This allows the invoker component in the main Web module to forward all incoming calls to your data module.

## Configuring TCorbaDataModule

To add a *TCorbaDataModule* component to your application, choose File | New and select CORBA Data Module from the Multitier page of the new items dialog. You will see the CORBA Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TCorbaDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TCorbaDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

**Note** You can add your own properties and methods to your new interface. For more information on adding to your data module's interface, see "Extending the application server's interface" on page 25-16.

The CORBA Data Module wizard lets you specify how you want your server application to create instances of the remote data module. You can choose either shared or instance-per-client.

- When you choose shared, your application creates a single instance of the remote data module that handles all client requests. This is the model used in traditional CORBA development.
- When you choose instance-per-client, a new remote data module instance is created for each client connection. This instance persists until its timeout period elapses with no messages from the client. This allows the server to free instances when they are no longer used by clients, but holds the risk that the server may be freed prematurely if the client does not use the server's interface for a long time.

**Note** Unlike instancing for COM servers, where the model determines the number of instances of the process that run, with CORBA, instancing determines the number of instances created of your object. They are all created within a single instance of the server executable.

In addition to the instancing model, you must specify the threading model in the CORBA Data Module wizard. You can choose Single- or Multi-threaded.

- If you choose Single-threaded, each remote data module instance is guaranteed to receive only one client request at a time. You can safely access the objects contained in your remote data module. However, you must guard against thread conflicts when you use global variables or objects not contained in the remote data module.
- If you choose Multi-threaded, each client connection has its own dedicated thread. However, your application may be called by multiple clients simultaneously, each on a separate thread. You must guard against simultaneous access of instance data as well as global memory. Writing Multi-threaded servers is tricky when you are using a shared remote data module instance, because you must protect all use of objects contained in your remote data module.

## Extending the application server's interface

---

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is

created for you automatically by the wizard when you create the remote data module.

To add to the remote data module's interface, you can

- Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you are adding a procedure, function, or property, and enter its syntax. When you click OK, you will be positioned in the code editor on the implementation of your new interface member.
- Use the type library editor. Select the interface for your application server in the type library editor, and click the tool button for the type of interface member (method or property) that you are adding. Give your interface member a name in the Attributes page, specify parameters and type in the Parameters page, and then refresh the type library. See Chapter 34, "Working with type libraries" for more information about using the type library editor. Note that many of the features you can specify in the type library editor (such as help context, version, and so on) do not apply to CORBA interfaces. Any values you specify for these in the type library editor are ignored.

**Note** Neither of these approaches works if you are implementing *TSoapDataModule*. For *TSoapDataModule* descendants, you must edit the server interface directly.

What Delphi does when you add new entries to the interface using the type library editor or the Add To Interface command depends on whether you are creating a COM-based (*TRemoteDataModule* or *TMTSDDataModule*) or CORBA (*TCorbaDataModule*) server.

- When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).
- When you add to a CORBA interface, your changes are reflected in your unit source code and the automatically generated `_TLB` unit. The `_TLB` unit is added to the `uses` clause of your unit. You must add this unit to the `uses` clause in your client application if you want to take advantage of early binding. In addition, you can save an .IDL file from the type library editor using the Export to IDL button. The .IDL file is needed for registering the interface with the Interface Repository and Object Activation Daemon.

**Note** You must explicitly save the TLB file by choosing Refresh in the type library editor and then saving the changes from the IDE.

Once you have added to your remote data module's interface, locate the properties and methods that were added to your remote data module's implementation. Add code to finish this implementation by filling in the bodies of the new methods.

Client applications call your interface extensions using the *AppServer* property of their connection component. For more information on how to do this, see "Calling server interfaces" on page 25-29.

## Adding callbacks to the application server's interface

You can allow the application server to call your client application by introducing a callback. To do this, the client application passes an interface to one of the application server's methods, and the application server later calls this method as needed.

However, if your extensions to the remote data module's interface include callbacks, you can't use an HTTP or SOAP-based connection. *TWebConnection* does not support callbacks. If you are using a socket-based connection, client applications must indicate whether they are using callbacks by setting the *SupportCallbacks* property. All other types of connection automatically support callbacks.

## Extending a transactional application server's interface

When using transactions or just-in-time activation, you must be sure all new methods call *SetComplete* to indicate when they are finished. This allows transactions to complete and permits the remote data module to be deactivated.

Furthermore, you can't return any values from your new methods that allow the client to communicate directly with objects or interfaces on the application server unless they provide a safe reference. If you are using a stateless MTS data module, neglecting to use a safe reference can lead to crashes because you can't guarantee that the remote data module is active. For more information on safe references, see "Passing object references" on page 39-20.

## Managing transactions in multi-tiered applications

---

When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back.

In addition, you can add transaction support to your server application by adding a database connection component or managing the transaction directly by sending SQL to the database server. This works the same way that you would manage transactions in a two-tiered application. For more information about this sort of transaction control, see "Managing transactions" on page 17-5.

If you have a transactional data module, you can broaden your transaction support by using MTS or COM+ transactions. These transactions can include any of the business logic on your application server, not just the database access. In addition, because they support two-phase commits, they can span multiple databases.

Only the BDE- and ADO-based data access components support two-phase commit. Do not use InterbaseExpress or dbExpress components if you want to have transactions that span multiple databases.

**Warning** When using the BDE, two-phase commit is fully implemented only on Oracle7 and MS-SQL databases. If your transaction involves multiple databases, and some of them are remote servers other than Oracle7 or MS-SQL, your transaction runs a small risk of only partially succeeding. Within any one database, however, you will always have transaction support.

By default, all *IAppServer* calls on a transactional data module are transactional. You need only set the transaction attribute of your data module to indicate that it must participate in transactions. In addition, you can extend the application server's interface to include method calls that encapsulate transactions that you define.

If your transaction attribute indicates that the remote data module requires a transaction, then every time a client calls a method on its interface, it is automatically wrapped in a transaction. All client calls to your application server are then enlisted in that transaction until you indicate that the transaction is complete. These calls either succeed as a whole or are rolled back.

**Note** Do not combine MTS or COM+ transactions with explicit transactions created by a database connection component or using explicit SQL commands. When your transactional data module is enlisted in a transaction, it automatically enlists all of your database calls in the transaction as well.

For more information about using MTS and COM+ transactions, see “MTS and COM+ transaction support” on page 39-8.

## Supporting master/detail relationships

---

You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see “Creating master/detail relationships” on page 18-34.

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see “Limiting records with parameters” on page 23-28.)
- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database connection component to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this when providing from datasets, set up a master/detail relationship between the datasets on the application server. Then set the *DataSet* property of your provider component to the master table. To use nested tables to represent master/detail relationships when providing from XML documents, use a transformation file that defines the nested detail sets.

When clients call the *GetRecords* method of the provider, it automatically includes the detail dataset as a *DataSet* field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

## Supporting state information in remote data modules

---

The *IAppServer* interface, which controls all communication between client datasets and providers on the application server, is mostly stateless. When an application is stateless, it does not “remember” anything that happened in previous calls by the

client. This stateless quality is useful if you are pooling database connections in a transactional data module, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with just-in-time activation, object pooling, or typical CORBA servers. SOAP data modules must be stateless.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching, the provider on the application server must “remember” information from previous calls (the current record).

Before and after any calls to the *IAppServer* interface that the client dataset makes (*AS\_ApplyUpdates*, *AS\_Execute*, *AS\_GetParams*, *AS\_GetRecords*, or *AS\_RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless.

For example, consider a dataset that represents the following parameterized query:

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

To enable incremental fetching in a stateless application server, you can do the following:

- When the provider packages a set of records in a data packet, it notes the value of *CUST\_NO* on the last record in the packet:

```
TRemoteDataModule1.DataSetProvider1GetData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
    DataSet.Last; { move to the last record }
    with Sender as TDataSetProvider do
        Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
    end;
```

- The provider sends this last *CUST\_NO* value to the client after sending the data packet:

```
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender: TObject;
            var OwnerData: OleVariant);
begin
    with Sender as TDataSetProvider do
        OwnerData := Tag; {send the last value of CUST_NO }
    end;
```

- On the client, the client dataset saves this last value of *CUST\_NO*:

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
    with Sender as TClientDataSet do
        Tag := OwnerData; {save the last value of CUST_NO }
    end;
```

- Before fetching a data packet, the client sends the last value of CUST\_NO it received:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    begin
      if not Active then Exit;
      OwnerData := Tag; { Send last value of CUST_NO to application server }
    end;
  end;
end;
```

- Finally, on the server, the provider uses the last CUST\_NO sent as a minimum value in the query:

```
TRemoteDataModule1.DataSetProvider1BeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
        begin
          Params.ParamValues['MinVal'] := OwnerData;
          Refresh; { force the query to reexecute }
        end;
      end;
    end;
  end;
end;
```

## Using multiple remote data modules

---

You may want to structure your application server so that it uses multiple remote data modules. Using multiple remote data modules lets you partition your code, organizing a large application server into multiple self-contained units, where each unit is relatively self-contained.

Although you can always create multiple remote data modules on the application server that function independently, Delphi provides support for a model where you have one main “parent” remote data module that dispatches connections from clients to other “child” remote data modules.

To create the parent remote data module, you must extend its *IAppServer* interface, adding properties that expose the interfaces of the child remote data modules. That is, for each child remote data module, add a property to the parent data module’s interface whose value is the *IAppServer* interface for the child data module. The property getter should look something like the following:

```
function ParentRDM.GetChildRDM: IChildRDM;
begin
  {note the parent RDM uses a factory component defined in the child RDM’s unit.
  This is more efficient if it must create several children for different clients }
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.ParentRDM := Self;
end;
```



For information about extending the parent remote data module's interface, see "Extending the application server's interface" on page 25-16.

**Tip** You may also want to extend the interface for each child data module, exposing the parent data module's interface, or the interfaces of the other child data modules. This lets the various data modules in your application server communicate more freely with each other.

Once you have added properties that represent the child remote data modules to the main remote data module, client applications do not need to form separate connections to each remote data module on the application server. Instead, they share a single connection to the parent remote data module, which then dispatches messages to the "child" data modules. Because each client application uses the same connection for every remote data module, the remote data modules can share a single database connection, conserving resources. For information on how child applications share a single connection, see "Connecting to an application server that uses multiple data modules" on page 25-30.

## Registering the application server

---

Before client applications can locate and use an application server, it must be registered or installed. (This is not strictly true for CORBA application servers, although registration is still recommended.)

- If the application server uses DCOM, HTTP, or sockets as a communication protocol, it acts as an Automation server and must be registered like any other COM server. For information about registering a COM server, see "Registering a COM object" on page 36-16.
- If you are using a transactional data module, you do not register the application server. Instead, you install it with MTS or COM+. For information about installing transactional objects, see "Installing transactional objects" on page 39-22.
- When the application server uses SOAP, the application must be a Web Service application. As such, it must be registered with your Web Server, so that it receives incoming HTTP messages. In addition, if you want clients that are not written using Delphi to access any of the interfaces in your application, you can publish a WSDL document that describes the invocable interfaces in your application. For information about exporting a WSDL document for a Web Service application, see "Generating WSDL documents for a Web Service application" on page 31-7.
- When the application server uses CORBA, registration is optional. If you want to allow client applications to use dynamic binding to your interface, you must install the server's interface in the Interface Repository. In addition, if you want to allow client applications to launch the application server when it is not already running, it must be registered with the OAD (Object Activation Daemon).

## Creating the client application

---

In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.

To create a multi-tiered client application, start a new project and follow these steps:

- 1 Add a new data module to the project.
- 2 Place a connection component on the data module. The type of connection component you add depends on the communication protocol you want to use. See “The structure of the client application” on page 25-4 for details.
- 3 Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see “Connecting to the application server” on page 25-23.
- 4 Set the other connection component properties as needed for your application. For example, you might set the *ObjectBroker* property to allow the connection component to choose dynamically from several servers. For more information about using the connection components, see “Managing server connections” on page 25-28.
- 5 Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see Chapter 23, “Using client datasets.”
- 6 Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property’s drop-down list.
- 7 Continue in the same way you would create any other database application. There are a few additional features available to clients of multi-tiered applications:
  - Your application may want to make direct calls to the application server. “Calling server interfaces” on page 25-29 describes how to do this.
  - You may want to use the special features of client datasets that support their interaction with the provider components. These are described in “Using a client dataset with a provider” on page 23-23.

## Connecting to the application server

---

To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the DataSnap page of the Component palette.

Use a connection component to

- Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See “Choosing a connection protocol” on page 25-8 for details on the benefits and limitations of the available protocols.
- Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol.
- Identify the application server on the server machine.

If you are not using CORBA, identify the server using the *ServerName* or *ServerGUID* property. *ServerName* identifies the base name of the class you specify when creating the remote data module on the application server. See “Setting up the remote data module” on page 25-13 for details on how this value is specified on the server. If the server is registered or installed on the client machine, or if the connection component is connected to the server machine, you can set the *ServerName* property at design time by choosing from a drop-down list in the Object Inspector. *ServerGUID* specifies the GUID of the remote data module’s interface. You can look up this value using the type library editor.

If you are using CORBA, identify the server using the *RepositoryID* property. *RepositoryID* specifies the Repository ID of the application server’s factory interface, which appears as the third argument in the call to *TCorbaVCLComponentFactory.Create* that is automatically added to the initialization section of the CORBA server’s implementation unit. You can also set this property to the base name of the CORBA data module’s interface (the same string as the *ServerName* property for other connection components), and it is automatically converted into the appropriate Repository ID for you.

- Manage server connections. Connection components can be used to create or drop connections and to call application server interfaces.

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the connection component to identify the application server by name, specify a server machine, and use the application server interface.

## Specifying a connection using DCOM

When using DCOM to communicate with the application server, client applications include a *TDCOMConnection* component for connecting to the application server. *TDCOMConnection* uses the *ComputerName* property to identify the machine on which the server resides.

When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*.

- Note** Even when there is a system registry entry for the application server, you can specify *ComputerName* to override this entry. This can be especially useful during development, testing, and debugging.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *ComputerName*. For more information, see “Brokering connections” on page 25-27.

If you supply the name of a host computer or server that cannot be found, the DCOM connection component raises an exception when you try to open the connection.

## Specifying a connection using sockets

You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a *TSocketConnection* component for connecting to the application server.

*TSocketConnection* identifies the server machine using the IP Address or host name of the server system, and the port number of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see “Describing sockets” on page 32-3.

Three properties of *TSocketConnection* specify this information:

- *Address* specifies the IP Address of the server.
- *Host* specifies the host name of the server.
- *Port* specifies the port number of the socket dispatcher program on the application server.

*Address* and *Host* are mutually exclusive. Setting one unsets the value of the other. For information on which one to use, see “Describing the host” on page 32-4.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *Address* or *Host*. For more information, see “Brokering connections” on page 25-27.

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher programs supplied with Delphi. If the socket dispatcher has been configured to use a different port, set the *Port* property to match that value.

**Note** You can configure the port of the socket dispatcher while it is running by right-clicking the Borland Socket Server tray icon and choosing Properties.

Although socket connections do not provide for using security protocols, you can customize the socket connection to add your own encryption. To do this

- 1 Create a COM object that supports the *IDataIntercept* interface. This is an interface for encrypting and decrypting data.
- 2 Use *TPacketInterceptFactory* as the class factory for this object. If you are using a wizard to create the COM object in step 1, replace the line in the initialization section that says `TComponentFactory.Create(...)` with `TPacketInterceptFactory.Create(...)`.
- 3 Register your new COM server on the client machine.

- 4 Set the *InterceptName* or *InterceptGUID* property of the socket connection component to specify this COM object. If you used *TPacketInterceptFactory* in step 2, your COM server appears in the drop-down list of the Object Inspector for the *InterceptName* property.
- 5 Finally, right click the Borland Socket Server tray icon, choose Properties, and on the properties tab set the Intercept Name or Intercept GUID to the ProgId or GUID for the interceptor.

This mechanism can also be used for data compression and decompression.

## Specifying a connection using HTTP

You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a *TWebConnection* component for connecting to the application server.

The Web connection component establishes a connection to the Web server application (*httpsrvr.dll*), which in turn communicates with the application server. *TWebConnection* locates *httpsrvr.dll* using a Uniform Resource Locator (URL). The URL specifies the protocol (*http* or, if you are using SSL security, *https*), the host name for the machine that runs the Web server and *httpsrvr.dll*, and the path to the Web server application (*httpsrvr.dll*). Specify this value using the *URL* property.

**Note** When using *TWebConnection*, *wininet.dll* must be installed on the client machine. If you have IE3 or higher installed, *wininet.dll* can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the *UserName* and *Password* properties so that the connection component can log on.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *URL*. For more information, see “Brokering connections” on page 25-27.

## Specifying a connection using SOAP

You can establish a connection to a SOAP application server using the *TSoapConnection* component. *TSoapConnection* is very similar to *TWebConnection*, because it also uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security to communicate with a server that is protected by a firewall.

The SOAP connection component establishes a connection to a Web server application that implements the *IAppServer* interface as a Web Service. *TSoapConnection* locates this Web Server application using a Uniform Resource Locator (URL). The URL specifies the protocol (*http* or, if you are using SSL security, *https*), the host name for the machine that runs the Web server, the name of the Web Service application, and a path that matches the path name of the *THTTPSoapDispatcher* on the application server. Specify this value using the *URL* property.

**Note** When using *TSoapConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the *UserName* and *Password* properties so that the connection component can log on.

## Specifying a connection using CORBA

Only the *RepositoryID* property is necessary in order to specify a CORBA connection. This is because a Smart Agent on the local network automatically locates an available server for your CORBA client.

However, you can limit the possible servers to which your client application connects by the other properties of the CORBA connection component. If you want to specify a particular server machine, rather than letting the CORBA Smart Agent locate any available server, use the *HostName* property. If there is more than one object instance that implements your server interface, you can specify which object you want to use by setting the *ObjectName* property.

The *TCorbaConnection* component obtains an interface to the CORBA data module on the application server in one of two ways:

- If you are using early (static) binding, you must add the *\_TLB.pas* file (generated by the type library editor) to your client application. Early binding is highly recommended, both for compile-time type checking and because it is much faster than late (dynamic) binding.
- If you are using late (dynamic) binding, the interface must be registered with the Interface Repository.

For more information on early vs. late binding, see “Calling server interfaces” on page 25-29.

## Brokering connections

If you have multiple servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so on, until a connection is formed.

Once the connection component has formed a connection with a name supplied by the broker, it saves that name as the value of the appropriate property (*ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL*). If the connection component closes the connection later, and then needs to reopen the connection, it tries using this property value, and only requests a new name from the broker if the connection fails.

Use an Object Broker by specifying the *ObjectBroker* property of your connection component. When the *ObjectBroker* property is set, the connection component does not save the value of *ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL* to disk.

**Note** Do not use the *ObjectBroker* property with CORBA connections. CORBA has its own brokering mechanism.

## Managing server connections

---

The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface.

### Connecting to the server

To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in "Connecting to the application server" on page 25-23. Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the connection component, you can open the connection by setting the *Connected* property of the connection component to *True*.

Before a connection component establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

### Dropping or changing a server connection

A connection component drops a connection to the application server when you

- set the *Connected* property to *False*.
- free the connection component. A connection object is automatically freed when a user closes the client application.
- change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

**Note** Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions

prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

## Calling server interfaces

---

Applications do not need to call the *IAppServer* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface, you need a way to call those extensions using the connection created by your connection component. Unless you are using SOAP, you can do this using the *AppServer* property of the connection component. For information about extending the application server's interface, see "Extending the application server's interface" on page 25-16.

*AppServer* is a Variant that represents the application server's interface. You can call an interface method using *AppServer* by writing a statement such as

```
MyConnection.AppServer.SpecialMethod(x,y);
```

However, this technique provides late (dynamic) binding of the interface call. That is, the *SpecialMethod* procedure call is not bound until runtime when the call is executed. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the server to set up interface calls before they are invoked.

When you are using DCOM or CORBA as a communications protocol, you can use early binding of *AppServer* calls. Use the **as** operator to cast the *AppServer* variable to the *IAppServer* descendant you created when you created the remote data module. For example:

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

To use early binding under DCOM, the server's type library must be registered on the client machine. You can use *TRegsvr.exe*, which ships with Delphi to register the type library.

**Note** See the *TRegSvr* demo (which provides the source for *TRegsvr.exe*) for an example of how to register the type library programmatically.

To use early binding with CORBA, you must add the *\_TLB* unit that is generated by the type library editor to your project. To do this, add this unit to the **uses** clause of your unit.

When you are using TCP/IP or HTTP, you can't use true early binding, but because the remote data module uses a dual interface, you can use the application server's dispinterface to improve performance over simple late-binding. The dispinterface has the same name as the remote data module's interface, with the string 'Disp'



appended. You can assign the *AppServer* property to a variable of this type to obtain the dispinterface. Thus:

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
  ...
  TempInterface.SpecialMethod(x,y);
  ...
end;
```

**Note** To use the dispinterface, you must add the `_TLB` unit that is generated when you save the type library to the `uses` clause of your client module.

If you are using SOAP, you can't use the *AppServer* property. Instead, you must use a remote interfaced object (*THHTTPRIO*) and make early-bound calls. As with all early-bound calls, the client application must know the application server's interface declaration at compile time. You can add this to your client application either by adding the same unit the server uses to declare and register the interface to the client's `uses` clause, or you can reference a WSDL document that describes the interface. For information on importing a WSDL document that describes the server interface, see "Importing WSDL documents" on page 31-8.

**Note** The unit that declares the server interface must also register it with the invocation registry. For details on how to register invocable interfaces, see "Defining invocable interfaces" on page 31-3.

Once your application uses the server unit that declares and registers the interface, or you have imported a WSDL document to generate such a unit, create an instance of *THHTTPRIO* for the desired interface:

```
X := THHTTPRIO.Create(nil);
```

Next, assign the URL that your connection component uses to the remote interfaced object:

```
X.URL := SoapConnection1.URL;
```

You can then use the `as` operator to cast the instance of *THHTTPRIO* to the application server's interface:

```
InterfaceVariable := X as IMyAppServer;
InterfaceVariable.SpecialMethod(x,y);
```

## Connecting to an application server that uses multiple data modules

If the application server uses a main "parent" remote data module and several child remote data modules, as described in "Using multiple remote data modules" on page 25-21, then you need a separate connection component for every remote data module on the application server. Each connection component represents the connection to a single remote data module.

While it is possible to have your client application form independent connections to each remote data module on the application server, it is more efficient to use a single

connection to the application server that is shared by all the connection components. That is, you add a single connection component that connects to the “main” remote data module on the application server, and then, for each “child” remote data module, add an additional component that shares the connection to the main remote data module.

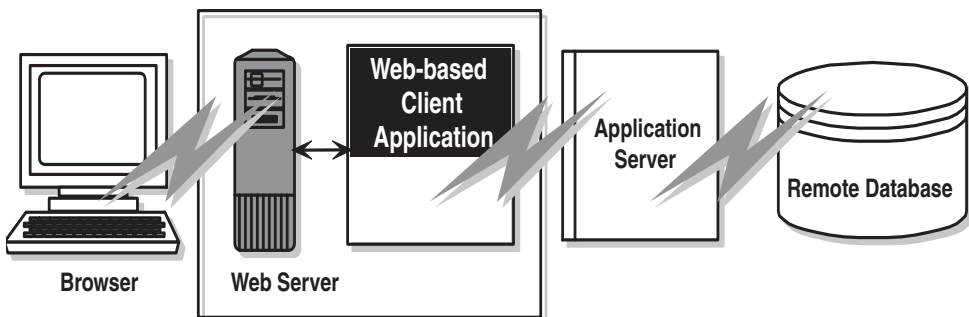
- 1 For the connection to the main remote data module, add and set up a connection component as described in “Connecting to the application server” on page 25-23. The only limitation is that you can’t use a CORBA or SOAP connection.
- 2 For each child remote data module, use a *TSharedConnection* component.
  - Set its *ParentConnection* property to the connection component you added in step 1. The *TSharedConnection* component shares the connection that this main connection establishes.
  - Set its *ChildName* property to the name of the property on the main remote data module’s interface that exposes the interface of the desired child remote data module.

When you assign the *TSharedConnection* component placed in step 2 as the value of a client dataset’s *RemoteServer* property, it works as if you were using an entirely independent connection to the child remote data module. However, the *TSharedConnection* component uses the connection established by the component you placed in step 1.

## Writing Web-based client applications

If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web application that acts simultaneously as a client to an application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in Figure 25.1.

**Figure 25.1** Web-based multi-tiered database application



There are two approaches that you can take to build the Web application:

- You can combine the multi-tiered database architecture with Delphi’s ActiveX support to distribute the client application as an ActiveX control. This allows any browser that supports ActiveX to run your client application as an in-process server.

- You can use XML data packets to build an InternetExpress application. This allows browsers that supports javascript to interact with your client application through html pages.

These two approaches are very different. Which one you choose depends on the following considerations:

- Each approach relies on a different technology (ActiveX vs. javascript and XML). Consider what systems your end users will use. The first approach requires a browser to support ActiveX (which limits clients to a Windows platform). The second approach requires a browser to support javascript and the DHTML capabilities introduced by Netscape 4 and Internet Explorer 4.
- ActiveX controls must be downloaded to the browser to act as an in-process server. As a result, the clients using an ActiveX approach require much more memory than the clients of an HTML-based application.
- The InternetExpress approach can be integrated with other HTML pages. An ActiveX client must run in a separate window.
- The InternetExpress approach uses standard HTTP, thereby avoiding any firewall issues that confront an ActiveX application.
- The ActiveX approach provides greater flexibility in how you program your application. You are not limited by the capabilities of the javascript libraries. The client datasets used in the ActiveX approach surface more features (such as filters, ranges, aggregation, optional parameters, delayed fetching of BLOBs or nested details, and so on) than the XML brokers used in the InternetExpress approach.

**Caution** Your Web client application may look and act differently when viewed from different browsers. Test your application with the browsers you expect your end-users to use.

## Distributing a client application as an ActiveX control

---

The multi-tiered database architecture can be combined with Delphi's ActiveX features to distribute a client application as an ActiveX control.

When you distribute your client application as an ActiveX control, create the application server as you would for any other multi-tiered application. The only limitation is that you will want to use DCOM, HTTP, SOAP, or sockets as a communications protocol, because you can't count on client machines having installed the CORBA runtime software. For details on creating the application server, see "Creating the application server" on page 25-11.

When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See "Creating an Active Form for the client application" for details.

Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your client application, the Web server must be running on the machine that has the client application.

If the client application uses DCOM to communicate between the client application and the application server, the machine with the Web browser must be enabled to work with DCOM. If the machine with the Web browser is a Windows 95 machine, it must have installed DCOM95, which is available from Microsoft.

## Creating an Active Form for the client application

- 1 Because the client application will be deployed as an ActiveX control, you must have a Web server that runs on the same system as the client application. You can use a ready-made server such as Microsoft's Personal Web server or you can write your own using the socket components described in Chapter 32, "Working with sockets."
- 2 Create the client application following the steps described in "Creating the client application" on page 25-23, except start by choosing File | New | Active Form, rather than beginning the client project as an ordinary Delphi project.
- 3 If your client application uses a data module, add a call to explicitly create the data module in the active form initialization.
- 4 When your client application is finished, compile the project, and select Project | Web Deployment Options. In the Web Deployment Options dialog, you must do the following:
  - 1 On the Project page, specify the Target directory, the URL for the target directory, and the HTML directory. Typically, the Target directory and the HTML directory will be the same as the projects directory for your Web Server. The target URL is typically the name of the server machine that is specified in the Windows Network | DNS settings.
  - 2 On the Additional Files page, include midas.dll with your client application.
- 5 Finally, select Project | WebDeploy to deploy the client application as an active form.

Any Web browser that can run Active forms can run your client application by specifying the .HTM file that was created when you deployed the client application. This .HTM file has the same name as your client application project, and appears in the directory specified as the Target directory.

## Building Web applications using InternetExpress

---

A client application can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries of database functions, and Delphi's Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. These applications make up Delphi's InternetExpress support.

Before building an InternetExpress application, you should understand Delphi's Web server application architecture. This is described in Chapter 27, "Creating Internet applications."

An InternetExpress application extends the basic Web server application architecture to act as the client of an application server. InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML governs the layout and appearance of the pages seen by end users in their browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in these XML data packets on the client machine.

If the InternetExpress application uses DCOM to connect to the application server, you must take additional steps to ensure that the application server grants access and launch permissions to its clients. See “Granting permission to access and launch the application server” on page 25-36 for details.

**Tip** You can create an InternetExpress application to provide Web browsers with “live” data even if you do not have an application server. Simply add the provider and its dataset to the Web module.

## Building an InternetExpress application

---

The following steps describe one way to build a Web application using InternetExpress. The result is an application that creates HTML pages that let users interact with the data from an application server via a javascript-enabled Web browser. You can also build an InternetExpress application using the Site Express architecture by using the InternetExpress page producer (*TInetXPageProducer*).

- 1 Choose File | New to display the New Items dialog box, and on the New page select Web Server application. This process is described in “Creating Web server applications with Web Broker” on page 28-1.
- 2 From the DataSnap page of the component palette, add a connection component to the Web Module that appears when you create a new Web server application. The type of connection component you add depends on the communication protocol you want to use. See “Choosing a connection protocol” on page 25-8 for details.
- 3 Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see “Connecting to the application server” on page 25-23.
- 4 Instead of a client dataset, add an XML broker from the InternetExpress page of the component palette to the Web module. Like *TClientDataSet*, *TXMLBroker* represents the data from a provider on the application server and interacts with the application server through its *IAppServer* interface. However, unlike client datasets, XML brokers request data packets as XML instead of as OleVariants and interact with InternetExpress components instead of data controls.
- 5 Set the *RemoteServer* property of the XML broker to point to the connection component you added in step 2. Set the *ProviderName* property to indicate the provider on the application server that provides data and applies updates. For more information about setting up the XML broker, see “Using an XML broker” on page 25-36.

- 6 Add an InternetExpress page producer (*TInetXPageProducer*) to the Web module for each separate page that users will see in their browsers. For each page producer, you must set the *IncludePathURL* property to indicate where it can find the javascript libraries that augment its generated HTML controls with data management capabilities.
- 7 Right-click a Web page and choose Action Editor to display the Action editor. Add action items for every message you want to handle from browsers. Associate the page producers you added in step 6 with these actions by setting their *Producer* property or writing code in an *OnAction* event handler. For more information on adding action items using the Action editor, see “Adding actions to the dispatcher” on page 28-4.
- 8 Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipsis button in the Object Inspector next to the *WebPageItems* property.) In this editor you can add Web Items to design the pages that users see in their browsers. For more information about designing Web pages for your InternetExpress application, see “Creating Web pages with an InternetExpress page producer” on page 25-38.
- 9 Build your Web application. Once you install this application with your Web server, browsers can call it by specifying the name of the application as the scriptname portion of the URL and the name of the Web Page component as the pathinfo portion.

## Using the javascript libraries

The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship with Delphi:

**Table 25.3** Javascript libraries

Library	Description
xmlDOM.js	This library is a DOM-compatible XML parser written in javascript. It allows parsers that do not support XML to use XML data packets. Note that this does not include support for XML Islands, which are supported by IE5 and later.
xmlDB.js	This library defines data access classes that manage XML data packets and XML delta packets.
xmlDisp.js	This library defines classes that associate the data access classes in xmlDB with HTML controls in the HTML page.
xmlErrDisp.js	This library defines classes that can be used when reconciling update errors. These classes are not used by any of the built-in InternetExpress components, but are useful when writing a Reconcile producer.
xmlShow.js	This library includes functions to display formatted XML data packets and XML delta packets. This library is not used by any of the InternetExpress components, but is useful when debugging.

These libraries can be found in the Source/Webmidas directory. Once you have installed these libraries, you must set the *IncludePathURL* property of all InternetExpress page producers to indicate where they can be found.

It is possible to write your own HTML pages using the javascript classes provided in these libraries instead of using Web items to generate your Web pages. However, you must ensure that your code does not do anything illegal, as these classes include minimal error checking (so as to minimize the size of the generated Web pages).

The classes in the javascript libraries are an evolving standard, and are updated regularly. If you want to use them directly rather than relying on Web items to generate the javascript code, you can get the latest versions and documentation of how to use them from CodeCentral available through [community.borland.com](http://community.borland.com).

## Granting permission to access and launch the application server

Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR\_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with EOLE\_ACCESS\_ERROR.

**Note** Because the application server runs under this guest account, it can't be shut down by other accounts.

To grant the Web application access and launch permissions, run DCOMCnfg.exe, which is located in the System32 directory of the machine that runs the application server. The following steps describe how to configure your application server:

- 1 When you run DCOMCnfg, select your application server in the list of applications on the Applications page.
- 2 Click the Properties button. When the dialog changes, select the Security page.
- 3 Select Use Custom Access Permissions, and press the Edit button. Add the name IUSR\_computername to the list of accounts with access permission, where computername is the name of the machine that runs the Web application.
- 4 Select Use Custom Launch Permissions, and press the Edit button. Add IUSR\_computername to this list as well.
- 5 Click the Apply button.

## Using an XML broker

---

The XML broker serves two major functions:

- It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application.
- It receives updates in the form of XML delta packets from browsers and applies them to the application server.

### Fetching XML data packets

Before the XML broker can supply XML data packets to the components that generate HTML pages, it must fetch them from the application server. To do this, it

uses the *IAppServer* interface of the application server, which it acquires through a connection component. You must set the following properties so that the XML producer can use the application server's *IAppServer* interface:

- Set the *RemoteServer* property to the connection component that establishes the connection to the application server and gets its *IAppServer* interface. At design time, you can select this value from a drop-down list in the object inspector.
- Set the *ProviderName* property to the name of the provider component on the application server that represents the dataset for which you want XML data packets. This provider both supplies XML data packets and applies updates from XML delta packets. At design time, if the *RemoteServer* property is set and the connection component has an active connection, the Object Inspector displays a list of available providers. (If you are using a DCOM connection the application server must also be registered on the client machine).

Two properties let you indicate what you want to include in data packets:

- You can limit the number of records that are added to the data packet by setting the *MaxRecords* property. This is especially important for large datasets because InternetExpress applications send the entire data packet to client Web browsers. If the data packet is too large, the download time can become prohibitively long.
- If the provider on the application server represents a query or stored procedure, you may want to provide parameter values before obtaining an XML data packet. You can supply these parameter values using the *Params* property.

The components that generate HTML and javascript for the InternetExpress application automatically use the XML broker's XML data packet once you set their *XMLBroker* property. To obtain the XML data packet directly in code, use the *RequestRecords* method.

**Note** When the XML broker supplies a data packet to another component (or when you call *RequestRecords*), it receives an *OnRequestRecords* event. You can use this event to supply your own XML string instead of the data packet from the application server. For example, you could fetch the XML data packet from the application server using *GetXMLRecords* and then edit it before supplying it to the emerging Web page.

## Applying updates from XML delta packets

When you add the XML broker to the Web module (or data module containing a *TWebDispatcher*), it automatically registers itself with the Web dispatcher as an auto-dispatching object. This means that, unlike other components, you do not need to create an action item for the XML broker in order for it to respond to update messages from a Web browser. These messages contain XML delta packets that should be applied to the application server. Typically, they originate from a button that you create on one of the HTML pages produced by the Web client application.

So that the dispatcher can recognize messages for the XML broker, you must describe them using the *WebDispatch* property. Set the *PathInfo* property to the path portion of the URL to which messages for the XML broker are sent. Set *MethodType* to the value of the method header of update messages addressed to that URL (typically *mtPost*). If you want to respond to all messages with the specified path, set *MethodType* to *mtAny*. If you don't want the XML broker to respond directly to update messages (for



example, if you want to handle them explicitly using an action item), set the *Enabled* property to *False*. For more information on how the Web dispatcher determines which component handles messages from the Web browser, see “Dispatching request messages” on page 28-5.

When the dispatcher passes an update message on to the XML broker, it passes the updates on to the application server and, if there are update errors, receives an XML delta packet describing all update errors. Finally, it sends a response message back to the browser, which either redirects the browser to the same page that generated the XML delta packet or sends it some new content.

A number of events allow you to insert custom processing at all steps of this update process:

- 1 When the dispatcher first passes the update message to the XML broker, it receives a *BeforeDispatch* event, where you can preprocess the request or even handle it entirely. This event allows the XML broker to handle messages other than update messages.
- 2 If the *BeforeDispatch* event handler does not handle the message, the XML broker receives an *OnRequestUpdate* event, where you can apply the updates yourself rather than using the default processing.
- 3 If the *OnRequestUpdate* event handler does not handle the request, the XML broker applies the updates and receives a delta packet containing any update errors.
- 4 If there are no update errors, the XML broker receives an *OnGetResponse* event, where you can create a response message that indicates the updates were successfully applied or sends refreshed data to the browser. If the *OnGetResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker sends a response that redirects the browser back to the document that generated the delta packet.
- 5 If there are update errors, the XML broker receives an *OnGetErrorResponse* event instead. You can use this event to try to resolve update errors or to generate a Web page that describes them to the end user. If the *OnGetErrorResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker calls on a special content producer called the *ReconcileProducer* to generate the content of the response message.
- 6 Finally, the XML broker receives an *AfterDispatch* event, where you can perform any final actions before sending a response back to the Web browser.

## Creating Web pages with an InternetExpress page producer

---

Each InternetExpress page producer generates an HTML document that appears in the browsers of your application’s clients. If your application includes several separate Web documents, use a separate page producer for each of them.

The InternetExpress page producer (*TInetXPageProducer*) is a special page producer component. As with other page producers, you can assign it as the *Producer* property of an action item or call it explicitly from an *OnAction* event handler. For more information about using content producers with action items, see “Responding to

request messages with action items” on page 28-7. For more information about page producers, see “Using page producer components” on page 28-13.

Unlike most page producers, the InternetExpress page producer has a default template as the value of its *HTMLDoc* property. This template contains a set of HTML-transparent tags that the InternetExpress page producer uses to assemble an HTML document (with embedded javascript and XML) including content produced by other components. Before it can translate all of the HTML-transparent tags and assemble this document, you must indicate the location of the javascript libraries used for the embedded javascript on the page. This location is specified by setting the *IncludePathURL* property.

You can specify the components that generate parts of the Web page using the Web page editor. Display the Web page editor by double-clicking the Web page component or clicking the ellipsis button next to the *WebPageItems* property in the Object Inspector.

The components you add in the Web page editor generate the HTML that replaces one of the HTML-transparent tags in the InternetExpress page producer’s default template. These components become the value of the *WebPageItems* property. After adding the components in the order you want them, you can customize the template to add your own HTML or change the default tags.

## Using the Web page editor

The Web page editor lets you add Web items to your InternetExpress page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a InternetExpress page producer component.

**Note** You must have Internet Explorer 4 or better installed to use the Web page editor.

The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of the Web items, indicating how they are nested. On the right, you can see the Web items included by the currently selected item. When you select a component in the top of the Web page editor, you can set its properties using the Object Inspector.

Click the New Item button to add a subitem to the currently selected item. The Add Web Component dialog lists only those items that can be added to the currently selected item.

The InternetExpress page producer can contain one of two types of item, each of which generates an HTML form:

- *TDataForm*, which generates an HTML form for displaying data and the controls that manipulate that data or submit updates.

Items you add to *TDataForm* display data in a multi-record grid (*TDataGrid*) or in a set of controls each of which represents a single field from a single record (*TFieldGroup*). In addition, you can add a set of buttons to navigate through data or post updates (*TDataNavigator*), or a button to apply updates back to the Web client (*TApplyUpdatesButton*). Each of these items contains subitems to represent

individual fields or buttons. Finally, as with most Web items, you can add a layout grid (*TLayoutGroup*), that lets you customize the layout of any items it contains.

- *TQueryForm*, which generates an HTML form for displaying or reading application-defined values. For example, you can use this form for displaying and submitting parameter values.

Items you add to *TQueryForm* display application-defined values (*TQueryFieldGroup*) or a set of buttons to submit or reset those values (*TQueryButtons*). Each of these items contains subitems to represent individual values or buttons. You can also add a layout grid to a query form, just as you can to a data form.

The bottom of the Web page editor displays the generated HTML code and lets you see what it looks like in a browser (IE4).

## Setting Web item properties

The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML.

When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the *XMLBroker* property associates the Web item with the XML broker that manages the data packet. You can further specify a dataset that is contained in a dataset field of that data packet using the *XMLDataSetField* property. If the Web item represents a specific field or parameter value, the Web item has a *FieldName* or *ParamName* property.

You can apply a style attribute to any Web item, thereby influencing the overall appearance of all the HTML it generates. Styles and style sheets are part of the HTML 4 standard. They allow an HTML document to define a set of display attributes that apply to a tag and everything in its scope. Web items offer a flexible selection of ways to use them:

- The simplest way to use styles is to define a style attribute directly on the Web item. To do this, use the *Style* property. The value of *Style* is simply the attribute definition portion of a standard HTML style definition, such as  
`color: red.`
- You can also define a style sheet that defines a set of style definitions. Each definition includes a style selector (the name of a tag to which the style always applies or a user-defined style name) and the attribute definition in curly braces:  
`H2 B {color: red}  
.MyStyle {font-family: arial; font-weight: bold; font-size: 18px }`

The entire set of definitions is maintained by the InternetExpress page producer as its *Styles* property. Each Web item can then reference the styles with user-defined names by setting its *StyleRule* property.

- If you are sharing a style sheet with other applications, you can supply the style definitions as the value of the InternetExpress page producer's *StylesFile* property instead of the *Styles* property. Individual Web items still reference styles using the *StyleRule* property.

Another common property of Web items is the *Custom* property. *Custom* includes a set of options that you add to the generated HTML tag. HTML defines a different set of options for each type of tag. The VCL reference for the *Custom* property of most Web items gives an example of possible options. For more information on possible options, use an HTML reference.

## Customizing the InternetExpress page producer template

The template of an InternetExpress page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the page producer generates a default template as the value of the *HTMLDoc* property. This default template has the form

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

The HTML-transparent tags in the default template are translated as follows:

**<#INCLUDES>** generates the statements that include the javascript libraries. These statements have the form

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </SCRIPT>
```

**<#STYLES>** generates the statements that defines a style sheet from definitions listed in the *Styles* or *StylesFile* property of the InternetExpress page producer.

**<#WARNINGS>** generates nothing at runtime. At design time, it adds warning messages for problems detected while generating the HTML document. You can see these messages in the Web page editor.

**<#FORMS>** generates the HTML produced by the components that you add in the Web page editor. The HTML from each component is generated in the order it appears in *WebPageItems*.

**<#SCRIPT>** generates a block of javascript declarations that are used in the HTML generated by the components added in the Web page editor.

You can replace the default template by changing the value of *HTMLDoc* or setting the *HTMLFile* property. The customized HTML template can include any of the HTML-transparent tags that make up the default template. The InternetExpress page producer automatically translates these tags when you call the *Content* method. In

addition, The InternetExpress page producer automatically translates three additional tags:

<#BODYELEMENTS> is replaced by the same HTML as results from the 5 tags in the default template. It is useful when generating a template in an HTML editor when you want to use the default layout but add additional elements using the editor.

<#COMPONENT Name=WebComponentName> is replaced by the HTML that the component named *WebComponentName* generates. This component can be one of the components added in the Web page editor, or it can be any component that supports the *IWebContent* interface and has the same Owner as the InternetExpress page producer.

<#DATAPACKET XMLBroker=BrokerName> is replaced with the XML data packet obtained from the XML broker specified by *BrokerName*. When, in the Web page editor, you see the HTML that the InternetExpress page producer generates, you see this tag instead of the actual XML data packet.

In addition, the customized template can include any other HTML-transparent tags that you define. When the InternetExpress page producer encounters a tag that is not one of the seven types it translates automatically, it generates an *OnHTMLTag* event, where you can write code to perform your own translations. For more information about HTML templates in general, see “HTML templates” on page 28-13.

- Tip** The components that appear in the Web page editor generate static code. That is, unless the application server changes the metadata that appears in data packets, the HTML is always the same, no matter when it is generated. You can avoid the overhead of generating this code dynamically at runtime in response to every request message by copying the generated HTML in the Web page editor and using it as a template. Because the Web page editor displays a <#DATAPACKET> tag instead of the actual XML, using this as a template still allows your application to fetch data packets from the application server dynamically.

## Using XML in database applications

In addition to the support for connecting to database servers, Delphi lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on Delphi's support for working directly with XML documents, see Chapter 30, "Working with XML documents."

Delphi's support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset) into XML documents and convert XML documents into data packets. In order to use these components, you must first define the transformation between the XML document and the data packet. Once you have defined the transformation, you can use special components to

- convert XML documents into data packets.
- provide data from and resolve updates to an XML document.
- use an XML document as the client of a provider.

### Defining transformations

---

Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document. A description of this relationship is stored in a special XML document called a transformation.

Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or from the metadata in a data packet to an XML schema. Often, you create transformation files

in pairs: one that maps from XML to data packet, and one that maps from data packet to XML.

In order to create the transformation files for a mapping, use the XMLMapper utility that ships in the bin directory.

## Mapping between XML nodes and data packet fields

---

XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset.

Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):

```
<?xml version="1.0" standalone='yes' ?>
<email>
  <head>
    <from>
      <name>Dave Boss</name>
      <address>dboss@MyCo.com</address>
    </from>
    <to>
      <name>Joe Engineer</name>
      <address>jengineer@MyCo.com</address>
    </to>
    <cc>
      <name>Robin Smith</name>
      <address>rsmith@MyCo.com</address>
    </cc>
    <cc>
      <name>Leonard Devon</name>
      <address>ldevon@MyCo.com</address>
    </cc>
  </head>
  <body>
    <subject>XML components</subject>
    <content>
      Joe,
      Attached is the specification for the new XML component support in Delphi.
      This looks like a good solution to our buisness-to-buisness application!
      Also attached, please find the project schedule. Do you think its reasonable?
      Dave.
    </content>
    <attachment attachfile="XMLSpec.txt"/>
    <attachment attachfile="Schedule.txt"/>
  </body>
</email>
```

One natural mapping between this document and a dataset would map each email message to a single record. The record would have fields for the sender's name and address. Because an email message can have multiple recipients, the recipient (<to>) would map to a nested dataset. Similarly, the cc list maps to a nested dataset. The subject line would map to a string field while the message itself (<content>) would probably be a memo field. The names of attachment files would map to a nested dataset because one message can have several attachments. Thus, the email above would map to a dataset something like the following:

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

where the nested dataset in the "To" field is

Name	Address
Joe Engineer	jengineer@MyCo.Com

the nested dataset in the "CC" field is

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

and the nested dataset in the "Attach" field is

Attachfile
XMLSpec.txt
Schedule.txt

Defining such a mapping involves identifying those nodes of the XML document that can be repeated and mapping them to nested datasets. Tagged elements that have values and appear only once (such as <content>...</content>) map to fields whose datatype reflects the type of data that can appear as the value. Attributes of a tag (such as the AttachFile attribute of the attachment tag) also map to fields.

Note that not all tags in the XML document appear in the corresponding dataset. For example, the <head>...</head> element has no corresponding element in the resulting dataset. Typically, only elements that have values, elements that can be repeated, or the attributes of a tag map to the fields (including nested dataset fields) of a dataset. The exception to this rule is when a parent node in the XML document maps to a field whose value is built up from the values of the child nodes. For example, an XML document might contain a set of tags such as

```
<FullName>
  <Title> Mr. </Title>
  <FirstName> John </FirstName>
  <LastName> Smith </LastName>
</FullName>
```

which could map to a single dataset field with the value

```
Mr. John Smith
```



## Using XMLMapper

---

The XML mapper utility, `xmlmapper.exe`, lets you define mappings in three ways:

- From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema.
- From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system.
- Between an existing XML schema and an existing data packet. This is useful when you have an XML schema and a database that both describe the same information and you want to make them work together.

Once you define the mapping, you can generate the transformation files that are used to convert XML documents to data packets and to convert data packets to XML documents. Note that only the transformation file is directional: a single mapping can be used to generate both the transformation from XML to data packet and from data packet to XML.

**Note** XML mapper relies on two .DLLs (`midas.dll` and `msxml.dll`) to work correctly. Be sure that you have both of these .DLLs installed before you try to use `xmlmapper.exe`. In addition, `msxml.dll` must be registered as a COM server. You can register it using `Regsvr32.exe`.

### Loading an XML schema or data packet

Before you can define a mapping and generate a transformation file, you must first load descriptions of the XML document and the data packet between which you are mapping.

You can load an XML document or schema by choosing **File | Open** and selecting the document or schema in the resulting dialog.

You can load a data packet by choosing **File | Open** and selecting a data packet file in the resulting dialog. (The data packet is simply the file generated when you call a client dataset's `SaveToFile` method.) If you have not saved the data packet to disk, you can fetch the data packet directly from the application server of a multi-tiered application by right-clicking in the **Datapacket** pane and choosing **Connect To Remote Server**.

You can load only an XML document or schema, only a data packet, or you can load both. If you load only one side of the mapping, XML mapper can generate a natural mapping for the other side.

### Defining mappings

The mapping between an XML document and a data packet need not include all of the fields in the data packet or all of the tagged elements in the XML document. Therefore, you must first specify those elements that are mapped. To specify these elements, first select the **Mapping** page in the central pane of the dialog.

To specify the elements of an XML document or schema that are mapped to fields in a data packet, select the Sample or Structure tab of the XML document pane and double-click on the nodes for elements that map to data packet fields.

To specify the fields of the data packet that are mapped to tagged elements or attributes in the XML document and double-click on the nodes for those fields in the Datapacket pane.

If you have only loaded one side of the mapping (the XML document or the data packet), you can generate the other side after you have selected the nodes that are mapped.

- If you are generating a data packet from an XML document, you first define attributes for the selected nodes that determine the types of fields to which they correspond in the data packet. In the center pane, select the Node Repository page. Select each node that participates in the mapping and indicate the attributes of the corresponding field. If the mapping is not straightforward (for example, a node with subnodes that corresponds to a field whose value is built from those subnodes), check the User Defined Translation check box. You will need to write an event handler later to perform the transformation on user defined nodes.

Once you have specified the way nodes are to be mapped, choose Create | Datapacket from XML. The corresponding data packet is automatically generated and displayed in the Datapacket pane.

- If you are generating an XML document from a data packet, choose Create | XML from Datapacket. A dialog appears where you can specify the names of the tags and attributes in the XML document that correspond to fields, records, and datasets in the data packet. For field values, you specify whether they map to a tagged element with a value or to an attribute by the way you name them. Names that begin with an @ symbol map to attributes of the tag that corresponds to the record, while names that do not begin with an @ symbol map to tagged elements that have values and that are nested within the element for the record.
- If you have loaded both an XML document and a data packet (client dataset file), be sure you select corresponding nodes in the same order. The corresponding nodes should appear next to each other in the table at the top of the Mapping page.

Once you have loaded or generated both the XML document and the data packet and selected the nodes that appear in the mapping, the table at the top of the Mapping page should reflect the mapping you have defined.

## Generating transformation files

To generate a transformation file, use the following steps:

- 1 First select the radio button that indicates what the transformation creates:
  - Choose the Datapacket to XML button if the mapping goes from data packet to XML document.
  - Choose the XML to Datapacket button if the mapping goes from XML document to data packet.

- 2 If you are generating a data packet, you will also want to use the radio buttons in the Create Datapacket As section. These buttons let you specify how the data packet will be used: as a dataset, as a delta packet for applying updates, or as the parameters to supply to a provider before fetching data.
- 3 Click Create and Test Transformation to generate an in-memory version of the transformation. XML mapper displays the XML document that would be generated for the data packet in the Datapacket pane or the data packet that would be generated for the XML document in the XML Document pane.
- 4 Finally, choose File | Save | Transformation to save the transformation file. The transformation file is a special XML file (with the .xtr extension) that describes the transformation you have defined.

## Converting XML documents into data packets

---

Once you have created a transformation file that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application.

The *TXMLTransform* component transforms an XML document into a data packet according to the mapping in a transformation file.

**Note** You can also use *TXMLTransform* to convert a data packet that appears in XML format into an arbitrary XML document.

## Specifying the source XML document

---

There are three ways to specify the source XML document:

- If the source document is a .xml file on disk, you can use the *SourceXmlFile* property.
- If the source document is an in-memory string of XML, you can use the *SourceXml* property.
- If you have an IDOMDocument interface for the source document, you can use the *SourceXmlDocument* property.

*TXMLTransform* checks these properties in the order listed above. That is, it first checks for a file name in the *SourceXmlFile* property. Only if *SourceXmlFile* is an empty string does it check the *SourceXml* property. Only if *SourceXml* is an empty string does it then check the *SourceXmlDocument* property.

## Specifying the transformation

---

There are two ways to specify the transformation that converts the XML document into a data packet:

- Set the *TransformationFile* property to indicate a transformation file that was created using `xmlmapper.exe`.
- Set the *TransformationDocument* property if you have an *IDOMDocument* interface for the transformation.

*TXMLTransform* checks these properties in the order listed above. That is, it first checks for a file name in the *TransformationFile* property. Only if *TransformationFile* is an empty string does it check the *TransformationDocument* property.

## Obtaining the resulting data packet

---

To cause *TXMLTransform* to perform its transformation and generate a data packet, you need only read the *Data* property. For example, the following code uses an XML document and transformation file to generate a data packet, which is then assigned to a client dataset:

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

## Converting user-defined nodes

---

When you define a transformation using `xmlmapper.exe`, you can specify that some of the nodes in the XML document are “user-defined”. User-defined nodes are nodes for which you want to provide the transformation in code rather than relying on a straightforward node-value-to-field-value translation.

You can provide the code to translate user-defined nodes using the *OnTranslate* event. *OnTranslate* is called every time the *TXMLTransform* component encounters a user-defined node in the XML document. In the *OnTranslate* event handler, you can read the source document and specify the resulting value for the field in the data packet.

For example, the following *OnTranslate* event handler converts a node in the XML document with the following form

```
<FullName>
  <Title> </Title>
  <FirstName> </FirstName>
  <LastName> </LastName>
</FullName>
```

into a single field value:

```
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
```

```

CurNode: IDOMNode;
begin
  if Id = 'FullName' then
  begin
    Value = '';
    if SrcNode.hasChildNodes then
    begin
      CurNode := SrcNode.firstChild;
      Value := Value + CurNode.nodeValue;
      while CurNode <> SrcNode.lastChild do
      begin
        CurNode := CurNode.nextSibling;
        Value := Value + ' ';
        Value := Value + CurNode.nodeValue;
      end;
    end;
  end;
end;
end;
end;

```

## Using an XML document as the source for a provider

---

The *TXMLTransformProvider* component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Chapter 24, “Using provider components.” For information on using provider components with client datasets, see “Using a client dataset with a provider” on page 23-23.

You can specify the XML document from which the XML provider provides data and to which it applies updates using the *XMLDataFile* property.

*TXMLTransformProvider* components use internal *TXMLTransform* components to translate between data packets and the source XML document: one to translate the XML document into data packets, and one to translate data packets back into the XML format of the source document after applying updates. These two *TXMLTransform* components can be accessed using the *TransformRead* and *TransformWrite* properties, respectively.

When using *TXMLTransformProvider*, you must specify the transformations that these two *TXMLTransform* components use to translate between data packets and the source XML document. You do this by setting the *TXMLTransform* component’s *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component.

In addition, if the transformation includes any user-defined nodes, you must supply an *OnTranslate* event handler to the internal *TXMLTransform* components.

You do not need to specify the source document on the *TXMLTransform* components that are the values of *TransformRead* and *TransformWrite*. For *TransformRead*, the source is the file specified by the provider’s *XMLDataFile* property (although, if you set *XMLDataFile* to an empty string, you can supply the source document using *TransformRead.XmlSource* or *TransformRead.XmlSourceDocument*). For *TransformWrite*, the source is generated internally by the provider when it applies updates.

## Using an XML document as the client of a provider

---

The *TXMLTransformClient* component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransform* client lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents.

To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the *ProviderName* property. As with the *ProviderName* property of a client dataset, *ProviderName* can be the name of a provider on a remote application server or it can be a local provider in the same form or data module as the *TXMLTransformClient* object. For information about providers, see Chapter 24, “Using provider components.”

If the provider is on a remote application server, you must use a *DataSnap* connection component to connect to that application server. Specify the connection component using the *RemoteServer* property. For information on *DataSnap* connection components, see “Connecting to the application server” on page 25-23.

### Fetching an XML document from a provider

---

*TXMLTransformClient* uses an internal *TXMLTransform* component to translate data packets from the provider into an XML document. You can access this *TXMLTransform* component as the value of the *TransformGetData* property.

Before you can create an XML document that represents the data from a provider, you must specify the transformation file that *TransformGetData* uses to translate the data packet into the appropriate XML format. You do this by setting the *TXMLTransform* component’s *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component. If that transformation includes any user-defined nodes, you will want to supply *TransformGetData* with an *OnTranslate* event handler as well.

There is no need to specify the source document for *TransformGetData*, *TXMLTransformClient* fetches that from the provider. However, if the provider expects any input parameters, you may want to set them before fetching the data. Use the *SetParams* method to supply these input parameters before you fetch data from the provider. *SetParams* takes two arguments: a string of XML from which to extract parameter values, and the name of a transformation file to translate that XML into a data packet. *SetParams* uses the transformation file to convert the string of XML into a data packet, and then extracts the parameter values from that data packet.

**Note** You can override either of these arguments if you want to specify the parameter document or transformation in another way. Simply set one of the properties on *TransformSetParams* property to indicate the document that contains the parameters or the transformation to use when converting them, and then set the argument you want to override to an empty string when you call *SetParams*. For details on the

properties you can use, see “Converting XML documents into data packets” on page 26-6.

Once you have configured *TransformGetData* and supplied any input parameters, you can call the *GetDataAsXml* method to fetch the XML. *GetDataAsXml* sends the current parameter values to the provider, fetches a data packet, converts it into an XML document, and returns that document as a string. You can save this string to a file:

```
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransformSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
  finally
    XMLDoc.Free;
  end;
end;
```

## Applying updates from an XML document to a provider

---

*TXMLTransformClient* also lets you insert all of the data from an XML document into the provider’s dataset or to delete all of the records in an XML document from the provider’s dataset. To perform these updates, call the *ApplyUpdates* method, passing in

- A string whose value is the contents of the XML document with the data to insert or delete.
- The name of a transformation file that can convert that XML data into an insert or delete delta packet. (When you define the transformation file using the XML mapper utility, you specify whether the transformation is for an insert or delete delta packet.)
- The number of update errors that can be tolerated before the update operation is aborted. If fewer than the specified number of records can’t be inserted or deleted, *ApplyUpdates* returns the number of actual failures. If more than the specified number of records can’t be inserted or deleted, the entire update operation is rolled back, and no update is performed.

The following call transforms the XML document *Customers.xml* into a delta packet and applies all updates regardless of the number of errors:

```
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```

## Writing Internet applications

The chapters in “Writing Internet applications” present concepts and skills necessary for building applications that are distributed over the Internet.

**Note** The components described in this section are not available in all editions of Delphi.





# Creating Internet applications

Web server applications extend the functionality and capability of existing Web servers. A Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Any operation that you can perform with a Delphi application can be incorporated into a Web server application.

Delphi provides two different architectures for developing Web server applications: Web Broker and WebSnap. Although these two architectures are different, WebSnap and Web Broker have many common elements. The WebSnap architecture acts as a superset of Web Broker. It provides additional components, and new features such as the WebSnap Surface Designer—which allows the content of a page to be displayed without the developer having to run the application. Applications developed with WebSnap can include Web Broker components, whereas applications developed with Web Broker cannot include WebSnap components.

This chapter describes the features of the Web Broker and WebSnap technologies and provides general information on Internet-based client/server applications.

## About Web Broker and WebSnap

---

The first step in building a Web server application is choosing which architecture you want to use. Both approaches provide many of the same features, including

- Support for many types of Web server applications, including ISAPI, NSAPI, CGI, Win CGI, and Apache. These are described in “Types of Web server applications” on page 27-6.
- Multithreading support so that incoming client requests are handled on separate threads.
- Caching of Web modules for quicker responses.

However, each approach has certain advantages and disadvantages. The major differences between these two approaches are outlined in the following table:

**Table 27.1** Web Broker versus WebSnap

Web Broker	WebSnap
Backward compatible	Although WebSnap applications can use any Web Broker components that produce content, the Web modules and dispatcher that contain these are new.
Available in cross-platform (CLX) applications.	At present, WebSnap is only available on Windows.
Only one Web module allowed in an application.	Multiple Web modules can partition the application into units, allowing multiple developers to work on the same project with fewer conflicts.
Only one Web dispatcher allowed in the application.	Multiple, special-purpose dispatchers handle different types of requests.
Specialized components for creating content include page producers, InternetExpress components, and Web Services components.	Supports all the content producers that can appear in Web broker applications, plus many others designed to let you quickly build complex data-driven Web pages.
No scripting support.	Support for server-side scripting (JScript or VBScript) allows HTML generation logic to be separated from the business logic.
No built-in support for named pages.	Named pages can be automatically retrieved by a page dispatcher and addressed from server-side scripts.
No session support.	Sessions store information about an end user that is needed for a short period of time. This can be used for such tasks as login/logout support.
Every request must be explicitly handled, using either an action item or an auto-dispatching component.	Dispatch components automatically respond to a variety of requests.
Only a few specialized components provide previews of the content they produce. Most development is not visual.	The WebSnap surface designer lets you build Web pages visually, and view the results at design time. Previews are available for all components.

For more information on Web Broker, see Chapter 28, "Using Web Broker." For more information on WebSnap, see Chapter 29, "Using WebSnap."

## Terminology and standards

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development

arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications:

- RFC822, “Standard for the format of ARPA Internet text messages,” describes the structure and content of message headers.
- RFC1521, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies,” describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, “Hypertext Transfer Protocol — HTTP/1.0,” describes a transfer mechanism used to distribute collaborative hypermedia documents.

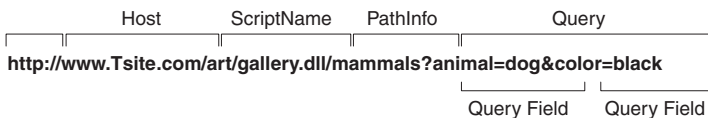
The IETF maintains a library of the RFCs on their Web site, [www.ietf.cnri.reston.va.us](http://www.ietf.cnri.reston.va.us)

## Parts of a Uniform Resource Locator

---

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in Figure 27.1:

**Figure 27.1** Parts of a Uniform Resource Locator



The first portion (not technically part of the URL) identifies the protocol (`http`). This portion can specify other protocols such as `https` (secure `http`), `ftp`, and so on.

The **Host** portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The **ScriptName** portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the **pathinfo**. This identifies the destination of the message within the Web server application. Path info values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The **Query** portion contains a set of named values. These values and their names are defined by the Web server application.

### URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many

sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

### HTTP request header information

---

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as “Host” followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case `/art/gallery.dll/animals?animal=dog&color=black`). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

## HTTP server activity

---

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests.

### Composing client requests

---

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

## Serving client requests

---

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application:

- If the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the content directly back to the server.
- If the program is WinCGI, the server opens a file and writes out the request information. It then executes the Win-CGI program, passing the location of the file containing the client information and the location of a file that the Win-CGI program should use to create content. The server waits while the program executes. When the program exits, the server reads the data from the content file written by the Win-CGI program.
- If the program is a dynamic-link library (DLL), the server loads the DLL (if necessary) and passes the information contained in the request to the DLL as a structure. The server waits while the program executes. When the DLL exits, it passes the content directly back to the server.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

## Responding to client requests

---

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent also differs based on the type of program:

- When a Win-CGI script finishes it constructs a page of HTML, writes it to a file, writes any response information to another file, and passes the locations of both

files back to the server. The server opens both files and passes the HTML page back to the client.

- When a DLL finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client.

Creating a Web server application as a DLL reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual request.

## Types of Web server applications

---

Whether you use Web Broker or WebSnap, you can create five types of Web server applications. In addition, you can create a Web Application Debugger executable, which integrates the Web server into your application so that you can debug your application logic. The Web Application Debugger executable is intended only for debugging. When you deploy your application, you should migrate to one of the other five types.

Each of the five types of Web server application uses a type-specific descendant of *TWebApplication*, *TWebRequest*, and *TWebResponse*:

**Table 27.2** Web server application components

Application Type	Application Object	Request Object	Response Object
Microsoft Server DLL (ISAPI)	<i>TISAPIApplication</i>	<i>TISAPIRequest</i>	<i>TISAPIResponse</i>
Netscape Server DLL (NSAPI)	<i>TISAPIApplication</i>	<i>TISAPIRequest</i>	<i>TISAPIResponse</i>
Apache Server DLL	<i>TApacheApplication</i>	<i>TApacheRequest</i>	<i>TApacheResponse</i>
Console CGI application	<i>TCGIApplication</i>	<i>TCGIRequest</i>	<i>TCGIResponse</i>
Windows CGI application	<i>TCGIApplication</i>	<i>TWinCGIRequest</i>	<i>TWinCGIResponse</i>

### ISAPI and NSAPI

An ISAPI or NSAPI Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by *TISAPIApplication*, which creates *TISAPIRequest* and *TISAPIResponse* objects. Each request message is automatically handled in a separate execution thread.

### Apache

An Apache Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by *TApacheApplication*, which creates *TApacheRequest* and *TApacheResponse* objects. Each request message is automatically handled in a separate execution thread.

### CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by *TCGIApplication*, which creates

*TCGIRequest* and *TCGIResponse* objects. Each request message is handled by a separate instance of the application.

### Win-CGI stand-alone

A Win-CGI stand-alone Web server application is a Windows application that receives client request information from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by *TCGIApplication*, which creates *TWinCGIRequest* and *TWinCGIResponse* objects. Each request message is handled by a separate instance of the application.

## Debugging server applications

---

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting.

The following topics describe techniques you can use to debug Web server applications.

### Using the Web Application Debugger

---

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application. This creates a Web server application that is also a COM server.

For information on how to write this Web server application using Web Broker, see Chapter 28, "Using Web Broker". For more information on using WebSnap, see Chapter 29, "Using WebSnap".

### Launching your application with the Web Application Debugger

Once you have developed your Web server application, you can run and debug it as follows:

- 1 With your project loaded in the IDE, set any breakpoints so that you can debug your application just like any other executable.
- 2 Choose Run | Run. This displays the console window of the COM server that is your Web server application. The first time you run your application, it registers your COM server so that the Web App debugger can access it.



- 3 Select Tools | Web App Debugger.
- 4 Click the Start button. This displays the ServerInfo page in your default Browser.
- 5 The ServerInfo page provides a drop-down list of all registered Web Application Debugger executables. Select your application from the drop-down list. If you do not find your application in this drop-down list, try running your application as an executable. Your application must be run once so that it can register itself. If you still do not find your application in the drop-down list, try refreshing the Web page. (Sometimes the Web browser caches this page, preventing you from seeing the most recent changes.)
- 6 Once you have selected your application in the drop-down list, press the Go button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

### Converting your application to another type of Web server application

When you have finished debugging your Web server application, you will need to convert it to another type of Web application before you install it with a commercial Web server. The following steps describe how to make these changes:

- 1 In the IDE, choose Project | Add New Project. This opens a new project without closing down the current one (you Web server application).
- 2 Launch the wizard to start a Web Broker or WebSnap application and choose the type of application you want to create.
- 3 Choose View | Project Manager to display the project manager.
- 4 In the project manager, drag all the units that make up your Web Server application from the old project to the one you just added. Omit the unit that implements the console window.

You now have a Web server application of the appropriate type.

### Debugging Web applications that are DLLs

---

ISAPI, NSAPI, and Apache applications are actually DLLs that contain predefined entry points. The Web server passes request messages to the application by making calls to these entry points. Because these applications are DLLs, you can debug them by setting your application's run parameters to launch the server.

To set up your application's run parameters, choose Run | Parameters and set the Host Application and Run Parameters to specify the executable for the Web server and any parameters it requires when you launch it. For details about these values on your Web server, see the documentation provided by your Web server vendor.

**Note** Some Web Servers require additional changes before you have the rights to launch the Host Application in this way. See your Web server vendor for details.

**Tip** If you are using Windows 2000 with IIS 5, details on all of the changes you need to make to set up your rights properly are described at the following Web site:

<http://community.borland.com/article/0,1410,23024,00.html>

Once you have set the Host Application and Run Parameters, you can set up your breakpoints so that when the server passes a request message to your DLL, you hit one of your breakpoints, and can debug normally.

**Note** Before launching the Web server using your application's run parameters, make sure that the server is not already running.

### **Debugging under Windows NT**

Under Windows NT, you must have the correct user rights to debug a DLL. In the User Manager, add your name to the lists granting rights for

- Log on as Service
- Act as part of the operation system
- Generate security audits

**Tip** Insert a hard-coded `_asm int 3` into your code where you wish to begin debugging. Recreate your DLL and use Just-In-Time Debugging (Tools | Options | Debug).

### **Debugging under Windows 2000**

Under Windows 2000, you grant the same rights as follows:

- 1 In the Administrative Tools portion of the Control Panel, click on Local Security Policy. Expand Local Policies and click on User Rights Assignment. Double-click on Act as part of the operating system in the right-hand panel.
- 2 Select Add to add a user to the list. Add your current user.
- 3 Reboot so the changes take effect.



## Using Web Broker

Web Broker components (located on the Internet tab of the component palette) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections.

The following sections of this chapter explain how you use the Web Broker components to create a Web server application.

### Creating Web server applications with Web Broker

---

To create a new Web server application using the Web Broker architecture:

- 1 Select File | New | Other.
- 2 In the New Items dialog box, select the New tab and choose Web Server Application.
- 3 A dialog box appears, where you can select one of the Web server application types:
  - ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
  - Apache: Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Apache Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.

- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application, and adds the required entries to the uses clause of the project file.
- Web Application Debugger stand-alone executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components and containing an empty Web Module.

## The Web module

---

The Web module (*TWebModule*) is a descendant of *TDataModule* and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer* and *TInetXPageProducer*, or descendants of *TCustomContentProducer* that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a multi-tiered database application.

In addition to storing non-visual components and business rules, the Web module also acts as a dispatcher, matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a *TWebDispatcher* component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from *TCustomWebDispatcher*, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the *TWebDispatcher* component that you add to a data module that replaces the Web module. If a second data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

**Note** The Web module that you set up at design time is actually a template. In ISAPI and NSAPI applications, each request message spawns a separate thread, and separate instances of the Web module and its contents are created dynamically for each thread.

**Warning** The Web module in a DLL-based Web server application is cached for later reuse to increase response time. The state of the dispatcher and its action list is not reinitialized between requests. Enabling or disabling action items during execution may cause unexpected results when that module is used for subsequent client requests.

## The Web Application object

The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* (either *TISAPIApplication* or *TCGIApplication*) that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

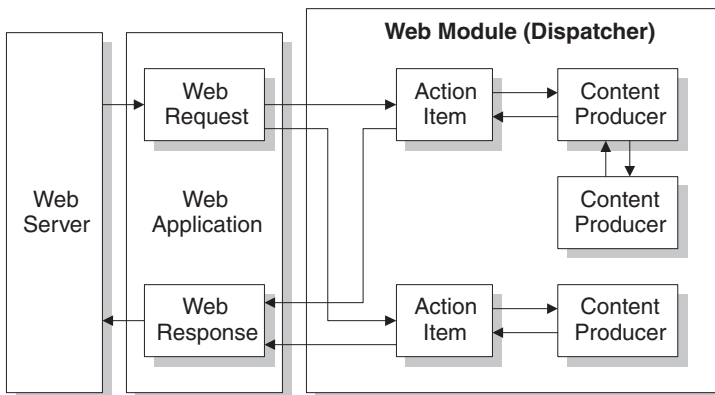
**Warning** Do not include the forms unit in the project **uses** clause after the CGIApp or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

## The structure of a Web Broker application

When the Web application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

The Web dispatcher controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message. It is described more fully in the section “The Web dispatcher” on page 28-4.

**Figure 28.1** Structure of a Server Application



The action items are responsible for reading the request and assembling a response message. Specialized content producer components aid the action items in dynamically generating the content of response messages, which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message. For more information on content producers, see “Generating the content of response messages” on page 28-13.

If you are creating the Web Client in a multi-tiered database application, your Web server application may include additional, auto-dispatching components that represent database information encoded in XML and database manipulation classes encoded in javascript. If you are creating a server that implements a Web Service, your Web server application may include an auto-dispatching component that passes SOAP-based messages on to an invoker that interprets and executes them. The dispatcher calls on these auto-dispatching components to handle the request message after it has tried all of its action items.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

## The Web dispatcher

---

If you are using a Web module, it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (*TWebDispatcher*) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it chooses one or more action items to respond to the request.

### Adding actions to the dispatcher

---

Open the action editor from the Object Inspector by clicking the ellipsis on the *Actions* property of the dispatcher. Action items can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

Action items are discussed in further detail in “Action items” on page 28-5.

## Dispatching request messages

---

When the dispatcher receives the client request, it generates a *BeforeDispatch* event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items.

Next, the dispatcher looks through its list of action items for one that matches the *pathinfo* portion of the request message's target URL and that can provide the service specified as the method of the request message. It does this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

After checking all its action items, if the message is not handled the dispatcher checks any specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications, which are described in "Building Web applications using InternetExpress" on page 25-33

If, after checking all the action items and any specially registered auto-dispatching components, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an *AfterDispatch* event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

## Action items

---

Each action item (*TWebActionItem*) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.



## Determining when action items fire

---

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the Object Inspector and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the Object Inspector.

### The target URL

The dispatcher compares the *PathInfo* property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

and assuming that the `/gallery.dll` part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

### The request method type

The *MethodType* property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

**Table 28.1** MethodType values

Value	Meaning
<i>mtGet</i>	The request is asking for the information associated with the target URI to be returned in a response message.
<i>mtHead</i>	The request is asking for the header properties of a response, as if servicing an <i>mtGet</i> request, but omitting the content of the response.
<i>mtPost</i>	The request is providing information to be posted to the Web application.
<i>mtPut</i>	The request asks that the resource associated with the target URI be replaced by the content of the request message.
<i>mtAny</i>	Matches any request method type, including <i>mtGet</i> , <i>mtHead</i> , <i>mtPut</i> , and <i>mtPost</i> .

### Enabling and disabling action items

Each action item has an *Enabled* property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A *BeforeDispatch* event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

**Caution** Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. If the Web server application is a DLL that caches Web modules, the initial state will not be reinitialized for the next request. Use the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

## Choosing a default action item

Only one of the action items can be the default action item. The default action item is selected by setting its *Default* property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the *Enabled* property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its *Enabled* property to *False*.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

**Caution** Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to *True*, that action will not be re-evaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

## Responding to request messages with action items

---

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its *Producer* property, that producer automatically assigns the *Content* of the response message using its *Content* method. The Internet page of the component palette includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an *OnAction* event. The *OnAction* event handler is passed the *TWebRequest* object that represents the HTTP request message and a *TWebResponse* object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see "Generating the content of response messages" on page 28-13.

### **Sending the response**

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

### **Using multiple action items**

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to *False*.

If many action items divide up the work of responding to request messages, each setting *Handled* to *False* so that others can continue, make sure the default action item leaves the *Handled* parameter set to *True*. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

## **Accessing client request information**

---

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of a *TWebResponse* object. In NSAPI and ISAPI applications, the request message is encapsulated by a *TISAPIRequest* object. Console CGI applications use *TCGIRequest* objects, and Windows CGI applications use *TWinCGIRequest* objects.

The properties of the request object are read-only. You can use them to gather all of the information available in the client request.

## Properties that contain request header information

---

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the *GetFieldByName* method.

### Properties that identify the target

The full target of the request message is given by the *URL* property. Usually, this is a URL that can be broken down into the protocol (HTTP), *Host* (server system), *ScriptName* (server application), *PathInfo* (location on the host), and *Query*.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the *QueryFields* property.

### Properties that describe the Web client

The request also includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the *From* property), to the URI where the message originated (the *Referer* or *RemoteHost* property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the *DerivedFrom* property. You can also determine the IP address of the client (the *RemoteAddr* property), and the name and version of the application that sent the request (the *UserAgent* property).

### Properties that identify the purpose of the request

The *Method* property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

Value	What the message requests
<i>OPTIONS</i>	Information about available communication options.
<i>GET</i>	Information identified by the <i>URL</i> property.
<i>HEAD</i>	Header information from an equivalent GET message, without the content of the response.
<i>POST</i>	The server application to post the data included in the <i>Content</i> property, as appropriate.
<i>PUT</i>	The server application to replace the resource indicated by the <i>URL</i> property with the data included in the <i>Content</i> property.
<i>DELETE</i>	The server application to delete or hide the resource identified by the <i>URL</i> property.
<i>TRACE</i>	The server application to send a loop-back to confirm receipt of the request.

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The *MethodType* property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

### Properties that describe the expected response

The *Accept* property indicates the media types the Web client will accept as the content of the response message. The *IfModifiedSince* property specifies whether the client only wants information that has changed recently. The *Cookie* property includes state information (usually added previously by your application) that can modify the response.

### Properties that describe the content

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the *ContentType* property, and its length in the *ContentLength* property. If the content of the message was encoded (for example, for data compression), this information is in the *ContentEncoding* property. The name and version number of the application that produced the content is specified by the *ContentVersion* property. The *Title* property may also provide information about the content.

## The content of HTTP request messages

---

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database maintained by the Web server application.

The unprocessed value of the content is given by the *Content* property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the *ContentFields* property.

## Creating HTTP response messages

---

When the Web server application creates a *TWebRequest* object for an incoming HTTP request message, it also creates a corresponding *TWebResponse* object to represent the response message that will be sent in return. In NSAPI and ISAPI applications, the response message is encapsulated by a *TISAPIResponse* object. Console CGI applications use *TCGIResponse* objects, and Windows CGI applications use *TWinCGIResponse* objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

## Filling in the response header

---

Most of the properties of the *TWebResponse* object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

## Indicating the response status

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the *StatusCode* property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the *ReasonString* property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

## Indicating the need for client action

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the *Location* property. If the client must provide a password before you can proceed, set the *WWWAuthenticate* property.

## Describing the server application

Some of the response header properties describe the capabilities of the Web server application. The *Allow* property indicates the methods to which the application can respond. The *Server* property gives the name and version number of the application used to generate the response. The *Cookies* property can hold state information about the client's use of the server application which is included in subsequent request messages.

## Describing the content

Several properties describe the content of the response. *ContentType* gives the media type of the response, and *ContentVersion* is the version number for that media type. *ContentLength* gives the length of the response. If the content is encoded (such as for data compression), indicate this with the *ContentEncoding* property. If the content came from another URI, this should be indicated in the *DerivedFrom* property. If the value of the content is time-sensitive, the *LastModified* property and the *Expires* property indicate whether the value is still valid. The *Title* property can provide descriptive information about the content.

## Setting the response content

---

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the *Content* property or the *ContentStream* property.

The *Content* property is a string. Delphi strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a *TFileStream* object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

**Note** If the value of the *ContentStream* property is not **nil**, the *Content* property is ignored.

## Sending the response

---

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an *OnAction* event handler. The response object provides two methods for sending a response: *SendResponse* and *SendRedirect*. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

## Generating the content of response messages

---

Delphi provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from *TCustomContentProducer* or one of its descendants.

*TCustomContentProducer* provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in the following section.
- Table producers create HTML commands based on the information in a dataset. They are described in “Using database information in responses” on page 28-17.

## Using page producer components

---

Page producers (*TPageProducer* and its descendants) take an HTML template and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

### HTML templates

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName=Value*, where there is no space between the parameter name, the equals symbol (=) and the value. The parameters are separated by whitespace.



The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the *TTag* data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

Tag Name	TTag value	What the tag should be converted to
<i>Link</i>	<i>tgLink</i>	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag.
<i>Image</i>	<i>tgImage</i>	A graphic image. The result is an HTML <IMG> tag.
<i>Table</i>	<i>tgTable</i>	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
<i>ImageMap</i>	<i>tgImageMap</i>	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.
<i>Object</i>	<i>tgObject</i>	An embedded ActiveX object. The result is an HTML sequence beginning with an <OBJECT> tag and ending with an </OBJECT> tag.
<i>Embed</i>	<i>tgEmbed</i>	A Netscape-compliant add-in DLL. The result is an HTML sequence beginning with an <EMBED> tag and ending with an </EMBED> tag.

Any other tag name is associated with *tgCustom*. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

**Note** The predefined tag names are case insensitive.

## Specifying the HTML template

Page producers provide you with many choices in how to specify the HTML template. You can set the *HTMLFile* property to the name of a file that contains the HTML template. You can set the *HTMLDoc* property to a *TStrings* object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the *Content* method.

In addition, you can call the *ContentFromString* method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the *ContentFromStream* method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands, reading the template directly from a *TBlobStream* object.

## Converting HTML-transparent tags

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers

the *OnHTMLTag* event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

## Using page producers from an action item

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
begin
    PageProducer1.HTMLFile := 'Greeting.html';
    Response.Content := PageProducer1.Content;
end;
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello <#UserName>! Welcome to our web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (<#UserName>) in the HTML during execution:

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
    const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
    if CompareText(TagString,'UserName') = 0 then
        ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

If the content of the request message was the string *Mr. Ed*, the value of *Response.Content* would be

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello Mr. Ed! Welcome to our web site.
</BODY>
</HTML>
```

**Note** This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer's *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item's *Producer* property to accomplish the same effect as the *OnAction* event handler above.

## Chaining page producers together

The replacement text from an *OnHTMLTag* event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the *ContentFromString* method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:



The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces `<#MonthlyImage>` with `<#Image Month=January Year=1997>`.
- Replaces `<#TitleLine>` with `<#Calendar Month=December Year=1996 Size=Small> January 1997 <#Calendar Month=February Year=1997 Size=Small>`.
- Replaces `<#MainBody>` with `<#Calendar Month=January Year=1997 Size=Large>`.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the `<#Image Month=January Year=1997>` tag with the appropriate HTML `<IMG>` tag. Yet another page producer resolves the `#Calendar` tags with appropriate HTML tables.

## Using database information in responses

---

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

As an alternate approach, special components on the InternetExpress page of the component palette let you build Web servers that are part of a multi-tiered database application. See “Building Web applications using InternetExpress” on page 25-33 for details.

### Adding a session to the Web module

---

Both console CGI applications and Win-CGI applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application. Each instance of the application has its own distinct, default session.

When writing an ISAPI application or an NSAPI application, however, each request message is handled in a separate thread of a single application instance. To prevent the database connections from different threads from interfering with each other, you must give each thread its own session.

Each request message in an ISAPI or NSAPI application spawns a new thread. The Web module for that thread is generated dynamically at runtime. Add a *TSession* object to the Web module to handle the database connections for the thread that contains the Web module.

Separate instances of the Web module are generated for each thread at runtime. Each of those modules contains the session object. Each of those sessions must have a separate name, so that the threads that handle separate request messages do not interfere with each other’s database connections. To cause the session objects in each module to dynamically generate unique names for themselves, set the *AutoSessionName* property of the session object. Each session object will dynamically generate a unique name for itself and set the *SessionName* property of all datasets in the module to refer to that unique name. This allows all interaction with the database for each request thread to proceed without interfering with any of the other request messages. For more information on sessions, see “Managing database sessions” on page 20-16.

## Representing database information in HTML

---

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer, which formats the fields of a dataset into the text of an HTML document.
- Table producers, which format the records of a dataset as an HTML table.

### Using dataset page producers

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tagname which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see “Using page producer components” on page 28-13.

To use a dataset page producer, add a *TDataSetPageProducer* component to your web module and set its *DataSet* property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

### Using table producers

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers, which format the fields of a dataset into the text of an HTML document.
- Query table producers, which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table’s color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

### Specifying the table attributes

Table producers use the *THTMLTableAttributes* object to describe the visual appearance of the HTML table that displays the records from the dataset. The

*THTMLTableAttributes* object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the Object Inspector. Select the table producer object in the Object Inspector and expand the *TableAttributes* property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

## Specifying the row attributes

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The *RowAttributes* property is a *THTMLTableRowAttributes* object.

At design time, specify these properties using the Object Inspector by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the *MaxRows* property.

## Specifying the columns

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the Object Inspector after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate *THTMLTableColumn* objects and using the methods of the *Columns* property to add them to the table. If you do not set up the *Columns* property, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

## Embedding tables in HTML documents

You can embed the HTML table that represents your dataset in a larger document by using the *Header* and *Footer* properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the *Caption* and *CaptionAlignment* properties to give your table a caption.

## Setting up a dataset table producer

*TDataSetTableProducer* is a table producer that creates an HTML table for a dataset. Set the *DataSet* property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

## Setting up a query table producer

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the *TQuery* object that uses those parameters as the *Query* property of a *TQueryTableProducer* component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the *Content* method of *TQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

## Using WebSnap

WebSnap augments WebBroker with new components, wizards, and views—making it easier to build Web applications that contain complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripts makes team development easier.

The dispatcher components automatically handle requests for page content, HTML form submissions, and requests for dynamic images. New components called adapters provide a means to define a scriptable interface to the business rules of your application. For example, the *TDataSetAdapter* object is used to make data-set components scriptable. You can use new producer components to quickly build complex, data-driven forms and tables, or to use XSL to generate a page. You can use the session component to keep track of end-users. You can use the user list component to provide access to user names, passwords, and access rights.

The Web application wizard allows you to quickly build an application that is customized with the components that you will need. The Web page module wizard allows you to create a module that defines a new page in your application. Or use the Web data module wizard to create a container for components that are shared across your Web application.

The page module views make it possible to see the result of server-side script without running the application. The Preview tab shows the page in an embedded browser. The HTML Result view shows the generated HTML. The XSL Tree and XML Tree views make it easier to work with XML and XSL.

To support a team of developers, you can use WebSnap's multiple-module support to partition the application into units that can be worked on independently. When creating a new page module, you can have the page module wizard create an external template file. You can edit the template file outside of the IDE and test it without recompiling the application.

The following sections of this chapter explain how you use the WebSnap components to create a Web server application.



## Creating Web server applications with WebSnap

---

To create a new Web server application using the WebSnap architecture:

- 1 Select File | New | Other.
- 2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.
- 3 A dialog box appears which requires the following types of information:
  - Server type
  - Web application module types
  - Web application module options
  - Application components

### Server type

---

Select one of the following types of Web server application, depending on your application's type of Web server:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Web server. It adds the library header to the project file, and adds the required entries to the uses list and to the exports clause of the project file.
- Apache: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Apache Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application and adds the required entries to the uses clause of the project file.
- Web Application Debugger executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web server application that communicates with the type of Web server your application will use.

## Web application module types

---

The Web application module provides centralized control for business rules and non-visual components in the Web application. There are two types of Web application modules:

- **Page Module:** Selecting this type of module creates a content page. The page module contains a page producer which is responsible for generating the content of a page. The page producer displays its associated page when the HTTP request pathinfo matches the page name. The page can act as the default page when the pathinfo is blank.
- **Data Module:** Selecting this type of module does not create a content page. This module is used as a container for components shared by other modules—for example, database components used by two Web Page modules.

## Web application module options

---

If the selected application module type is page module, you can associate a name with the page by entering a name in the Page Name field in the dialog box. At runtime, the instance of this module can be either kept in cache, or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options through the Page Options button. You can set the following categories:

- **Producer:** The producer type for the page can be set to one of *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, or *XSLPageProducer*. If the selected page producer supports scripting, then use the Script Engine drop-down list to select the language used to script the page.

**Note** The *AdapterPageProducer* supports only JScript.

- **HTML:** When the selected producer uses an HTML template this group will be visible.
- **XSL:** When the selected producer uses an XSL template, such as *TXSLPageProducer*, this group will be visible.
- **New File:** Check New File if you want a template file to be created and managed as part of the unit. A managed template file will appear in the project manager and have the same file name and location as the unit source file. Uncheck New File if you want to use the properties of the producer component (typically the *HTMLDoc* or *HTMLFile* property).
- **Template:** When New File is checked, choose the default content for the template file from the Template drop-down. The “Default” template displays the title of the application, the title of the page, and hyperlinks to published pages.
- **Page:** Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application’s logic, whereas the title is the name that the end user will see when the page is displayed in a browser.

- **Published:** Check **Published** to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.
- **Login Required:** Check **Login Required** to require the user to log on before the page can be accessed.

## Application components

---

Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page.

Selecting the **Components** button displays a dialog box where you can select one or more of the Application components:

- **Application Adapter:** Contains information about the application, such as the title. The default type is *TApplicationAdapter*.
- **End User Adapter:** Contains information about the user, such as their name, access rights, and whether they are logged in. The default type is *TEndUserAdapter*. *TEndUserSessionAdapter* may also be selected.
- **Page Dispatcher:** Examines the HTTP request's pathinfo, and calls the appropriate page module to return the content of a page. The default type is *TPageDispatcher*.
- **Adapter Dispatcher:** Automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components. The default type is *TAdapterDispatcher*.
- **Dispatch Actions:** Allows you to define a collection of action items to handle requests based on pathinfo and method type. Action items call user-defined events, or request the content of page-producer components. The default type is *TWebDispatcher*.
- **Locate File Service:** Provides control over the loading of template files, and script include files, when the Web application is running. The default type is *TLocateFileService*.
- **Sessions Service:** Used to store information about an end-users that is needed for a short period of time. For example, you can use sessions to keep track of logged-in users, and to automatically log a user out after a period of inactivity. The default type is *TSessionService*.
- **User List Service:** Keeps track of authorized users, and their passwords and access rights. The default type is *TWebUserList*.

For each of the above components, the component types listed are the default types shipped with the Delphi software product. Users can create their own component types or use third-party component types.

## Web modules

---

There are four Web module types:

- *TWebAppPageModule*
- *TWebAppDataModule*
- *TWebPageModule*
- *TWebDataModule*

The Web application module (*TWebAppPageModule* or *TWebAppDataModule*) is a container for the application components that perform functions for the application as a whole—such as dispatching requests, managing sessions, and maintaining user lists. Your project can contain only one of these types of application modules.

The Web page module (*TWebPageModule*) provides content to a page and the Web data module (*TWebDataModule*) acts as a container for components shared across your application. You can optionally include one or more Web page and Web data modules in the Web application module.

### Web data modules

---

Like standard data modules, a Web data module is a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. When the application is running, a data module does not create a window.

The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements.

Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both:

- a page module that displays a grid, and
- a page module that displays an input form.

#### Structure of a Web data module unit

Standard data modules have a variable called the form variable, which is used to access the data module object. Web data modules replace this with a function. The purpose is the same. However, because WebSnap applications may be multi-threaded and may have multiple instances of a particular module that service multiple requests concurrently, this function is implemented to return the correct instance.

The unit also registers a factory. The factory specifies how the module should be managed by the WebSnap application. For example, flags indicate whether to cache the module and reuse it for other requests, or to destroy the module after a request has been serviced.

## Interfaces implemented by a Web data module

A Web data module implements the following interfaces:

*INotifyWebActivate*: This interface is called when the module is activated to service a Web request, and before the module is deactivated after a Web request is serviced.

*IWebVariablesContainer*: This interface is called during script execution to resolve variable references. The adapter dispatcher also calls this interface to locate adapter actions and fields that are referenced in an HTTP request.

*IGetScriptObject*: This interface is called to retrieve the module's IDispatch implementation. The returned object is the interface between the module and the active scripting engine.

*IteratorObjectSupport*: This interface is used to iterate through all of the objects within the module that can be accessed by active scripting.

## Web page modules

---

The page module has a page producer component associated with it. When a request is received, the page dispatcher analyses the request and calls the appropriate page module to process the request and return the content of the page.

Like Web data modules, Web page modules are containers for components. The difference between a Web data module and a Web page module is that a Web page module is used to produce a Web page.

### Page producer component

Web page modules have a property that identifies the page producer component responsible for generating content for the page. The WebSnap wizards automatically add a producer when creating a Web page module. You can change the page producer component later by dropping in a different producer from the WebSnap palette. However, if the page module has a template file, be sure that the content of this file is compatible with the producer component.

### Page name

Web page modules have a page name that can be used to reference the page in an HTTP request or within the application's logic. A factory in the Web page module's unit specifies the page name for the Web page module.

### Producer template

Most page producers use a template. HTML templates typically contain some static HTML mixed in with transparent tags or server-side script. When page producers create their content, they replace the transparent tags with appropriate values and execute the server-side script to produce the HTML that is displayed by a client browser. The XSLPageProducer is an exception to this. It uses XSL templates, which contain XSL rather than HTML. The XSL templates do not support transparent tags or server-side script.

Web page modules may have an associated template file that is managed as part of the unit. A managed template file appears in the project manager and has the same base file name and location as the unit service file. If the Web page module does not have an associated template file then the properties of the page producer component specify the template.

### Interfaces that the Web page module implements

A Web page module implements all of the interfaces of a Web data module, in addition to the following:

*IDefaultPageFileName*: The WebSnap surface designer uses this interface to ensure that a page module uses the proper template file.

*ISetWebContentOptions*: The WebSnap surface designer uses this interface to control the content that the page module generates. For example, an option can be used to retrieve the content before the Active Script executes.

*IGetProducerComponent*: The WebSnap surface designer uses this interface to retrieve the producer component associated with the page module.

*IProducerEditorViewSupport*: The WebSnap surface designer uses this interface to retrieve information about the editor views that it should display when the page module is active. The available editor views include HTML Script, Preview, HTML Result, XSL Tree, and XML Tree.

*IPageResult*: This interface provides access to the result that a page module produces. This interface supports three types of results: HTTP Content, HTTP Redirection, and Page Include.

*IGetDefaultAction*: This interface retrieves the default adapter action (if any) that is associated with the page module.

## Web application modules

---

The Web application module implements interfaces that are not implemented by *TWebPageModule* or *TWebDataModule*.

### Interfaces implemented by a Web application data module

A Web application data module implements all the interfaces of a Web data module, in addition to the following:

*IGetWebAppServices*: Retrieves the interface to the application's Web request handler.

*IGetWebAppComponents*: Retrieves the interface to the application-level components, including the AdapterDispatcher, PageDispatcher, SessionsService, and EndUserAdapter.

### Interfaces implemented by a Web application page module

A Web application page module implements all the interfaces of a Web Application data module and a Web page module.

## Adapters

---

Adapters provide a way to create an interface to the application data. They allow you to insert scripting languages into a page, and to retrieve information by making calls from your script code to the adapters.

For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the values of those data fields.

### Fields

---

Fields are components that the page producer uses to retrieve data from your application and to display the content on a Web page. Fields can also be used to retrieve an image. In this case, the field returns the address of the image written to the Web page. When a page displays its content, a request sent to the Web application, invokes the adapter dispatcher to retrieve the actual image from the field component.

### Actions

---

Actions are components that execute commands on behalf of the adapter. When a page producer generates its page, the scripting language calls adapter action components to return the name of the action along with any parameters necessary to execute the command. For example, consider clicking a button on an HTML form to delete a row from a table. This returns, in the HTTP request, the action name associated with the button and a parameter indicating the row number. The adapter dispatcher locates the named action component and passes the row number as a parameter to the action.

### Errors

---

Adapters keep a list of errors that occur while executing an action. Page producers can access this list of errors and display them in the Web page that the application returns to the end user.

### Records

---

Some adapter components, such as *TDataSetAdapter*, represent multiple rows. The adapter provides a scripting interface which allows iteration through the rows. Some adapters support paging, and iterate over only the rows on the current page.

## Page producers

---

You use page producers to generate content on behalf of a Web page module. You can also use producers in the same way as they are used in WebBroker applications, by associating the producer with a Web dispatcher action item. The advantages of using the Web page module are:

- the ability to preview the page's layout without running the application, and
- the ability to associate a page name with the module, so that the page dispatcher can call the page producer automatically.

## Templates

---

Producers provide the following functionality:

- They generate HTML content.
- They can reference an external file using the HTMLfile property, or the internal string using the HTMLDoc property.
- When the producers are used in conjunction with a Web page module, the template can be a file associated with a unit.
- Producers dynamically generate HTML which can be inserted into the template using transparent tags or active scripting. Transparent tags can be used in the same way as WebBroker applications. For more details, see "Converting HTML-transparent tags" on page 28-14. Active scripting support allows you to embed JavaScript or VBScript inside the HTML page.

## Server-side scripting in WebSnap

---

Page producer templates can include scripting languages such as JScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser).

### Active scripting

---

WebSnap relies on *active scripting* to implement server-side script. Active scripting is a technology created by Microsoft to allow a scripting language to be used with application objects through COM interfaces. Microsoft ships two active scripting languages, VBScript and JScript. Support for other languages is available through third parties.



## Script engine

---

The page producer's *ScriptEngine* property identifies the active scripting engine that evaluates the script within a template.

## Script blocks

---

Script blocks are delimited by `<%` and `%>`. The script engine evaluates any text inside script blocks. The result becomes part of the page producer's content. The page producer writes text outside of a script block after translating any embedded transparent tags. Script blocks can also enclose text, allowing conditional logic and loops to dictate the output of text. For example, the following JScript block generates a list of five numbered lines:

```
<ul>
  <% for (i=0;i<5;i++) { %>
    <li>Item <% Response.Write(i) %></li>
  <% } %>
</ul>
```

The following script block is equivalent:

```
<ul>
  <% for (i=0;i<5;i++) { %>
    <li>Item <%=i %></li>
  <% } %>
</ul>
```

The `<%=` delimiter is short for *Response.Write*.

## Creating script

---

Developers can take advantage of WebSnap features to automatically generate script.

### Wizard templates

When they create a new WebSnap application or page module, WebSnap wizards provide a template field that is used to select the initial content for the page module template. For example, the template called "Default" generates JScript to display the application title, page name, and links to published pages.

### TAdapterPageProducer

The *TAdapterPageProducer* builds forms and tables by generating HTML and JScript. The generated JScript calls adapter objects to retrieve field values, field image parameters, and action parameters.

## Editing and viewing script

---

The WebSnap surface designer provides a view of your Web Page modules which lets you preview a scripted page. Use the HTML Result tab to view the HTML

resulting from the executed script. Use the Preview tab to view the result in a browser. The HTML Script tab is available when the Web Page module uses *TAdapterPageProducer*. The HTML Script tab displays the HTML and JScript generated by the *TAdapterPageProducer* object. Consult this view to see how to write script that builds HTML forms to display adapter fields and execute adapter actions.

## Including script in a page

---

A template can include script from a file or from another page. To include script from a file, use the following code statement:

```
<!-- #include file="filename.html" -->
```

When the template includes script from another page, the script is evaluated by the including page, use the following code statement to include the unevaluated content of page1.

```
<!-- #include page="page1" -- >
```

## Script objects

---

Script objects are either VCL or CLX objects that can be referenced by script. You make VCL or CLX objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting:

- **Application**—The application object (which may be null) provides access to the application adapter of the Web Application module. The following JScript block writes the application title:
- **EndUser**—The EndUser object provides access to the end-user adapter of the Web Application module. The following JScript block writes the end-user name:
- **Session**—The session object provides access to the session object of the Web Application module. The following JScript block writes the session ID:
- **Pages**—The pages object (*Pages*) provides access to the application pages. The following JScript block writes links to all published pages:

```
<%= Application.Title %>
```

```
<%= EndUser.DisplayName %>
```

```
<%= Session.SessionID %>
```

```
<%
  e = new Enumerator(Pages)
  for (; !e.atEnd(); e.moveNext())
  {
    if (e.item().Published)
    {
      Response.Write('<A HREF="' + e.item().HREF + '>' + e.item().Title + '</A>')
    }
  }
%>
```

Note that the editor's Preview tab will not display the proper result of this script block. The pages object is always empty at design time because the Web page module factories have not been registered.

- **Modules**—The modules object provides access to the application modules. The following JScript block writes the content of an adapter field in a module called DM.

```
<%= Modules.DM.Adapter1.Field1.DisplayText %>
```

- **Page**—The Page object provides access to the current page. The following JScript block writes the title of the current page:

```
<%= Page.Title %>
```

- **Producer**—The Producer object provides access to the page producer of the Web Page module. The following JScript block evaluates a transparent tag before writing the content:

```
<% Producer.Write('Here is a tag <#TAG>') %>
```

Note that the editor's Preview tab will probably not display the proper result of this script block. The event handlers that usually replace transparent tags do not execute unless the application is running.

- **Response**—The Response object provides access to the WebResponse. Use this object when tag replacement is not desired.

```
<% Response.Write('Hello World!') %>
```

- **Request**—The Request object provides access to the WebRequest. The following JScript block displays the pathinfo.

```
<%= Request.PathInfo %>
```

- **Adapter Objects**—All of the adapter's components on the current page can be referenced without qualification. Adapter's in other modules must be qualified using the Modules objects. The following JScript block displays the text value of the *FirstName* field from of all rows of Adapter1:

```
<% e = new Enumerator(Adapter1.Records) %>  
<% for (; !e.atEnd(); e.moveNext()) %>  
<% { %>  
    <p><%= Adapter1.FirstName.DisplayText %>  
<% } %>
```

## Dispatching requests

---

When the WebSnap application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned.

## WebContext

---

Before handling the request, the Web application module initializes the WebContext object (of type *TWebContext*). The WebContext is a thread variable that provides global access to variables used by components servicing the request. For example, the WebContext contains the *TWebResponse* and *TWebRequest* objects, as well as the adapter request and adapter response objects discussed later in this section.

## Dispatcher components

---

The dispatcher components within the Web Application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request.

The adapter dispatcher component (*TAdapterDispatcher*) looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it will pass control to that component.

The Web Dispatcher component (*TWebDispatcher*) maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The Web dispatcher looks for an action item that matches the request. If it finds one, it passes control to that action item. The Web dispatcher also looks for auto-dispatching components that can handle the request.

The page dispatcher component (*TPageDispatcher*) examines the pathInfo property of the *TWebRequest* object, looking for the name of a registered Web page module. If the dispatcher finds a Web page module name, then it will pass control to that Web page module.

## Adapter dispatcher operation

---

The adapter dispatcher component automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

## Using adapter components to generate content

In order for WebSnap applications to automatically execute adapter actions and retrieve dynamic images from adapter fields, the HTML content must be properly constructed. If the HTML content is not properly constructed, then the resulting HTTP request will not contain the information that the adapter dispatcher needs to call adapter action and field components.

To reduce errors in constructing the HTML page, adapter components indicate what the names and values of HTML elements must be. Adapter components have methods that retrieve the names and values of hidden fields that must appear on an HTML form used to update adapter fields. Typically, page producers use server-side scripts to retrieve names and values from adapter components and generates HTML

using these names and values. For example, the following script constructs an <IMG> element that references the field called Graphic from Adapter1:

```
">
```

When the Web application evaluates the script, the HTML src attribute will contain the information necessary to identify the field and any parameters that the field component needs to retrieve the image. The resulting HTML might look like this:

```

```

When the browser sends an HTTP request to retrieve this image to the Web application, the adapter dispatcher will be able to determine that the Graphic field of Adapter1, in the module DM, should be called with the parameter "Species No=90090". The adapter dispatcher will call the Graphic field to write an appropriate HTTP response.

The following script constructs an <A> element referencing the EditRow action of Adapter1 and the page called "Details":

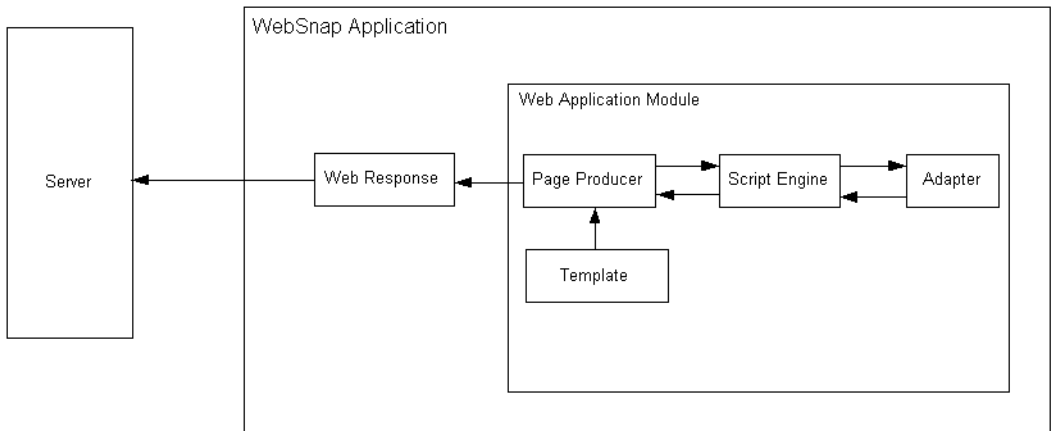
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

The resulting HTML might look like this:

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

When the end-user clicks this hyperlink and the browser sends an HTTP request, the adapter dispatcher will be able to determine that the EditRow action of Adapter1, in the module DM, should be called with the parameter "Species No=903010". The adapter dispatcher will also indicate that the Edit page is to be displayed if the action executes successfully, and that the Grid page is to be displayed if action execution fails. It will then call the EditRow action to locate the row to be edited, and the page named Edit will be called to generate an HTTP response. Figure 29.1 shows how adapter components are used to generate content.

**Figure 29.1** Generating Content Flow



## Adapter requests and responses

When the adapter dispatcher receives the client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about the HTTP request. The adapter request and adapter response objects are stored in the WebContext to allow access during the processing of the request.

The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field.

The adapter response object is used by the adapter component to indicate the response to an adapter action or adapter image request. There are two types of adapter response objects, action and image.

### Action request

The action request object is responsible for breaking the HTTP request down into information needed to execute an adapter action. The types of information needed for executing an adapter action may include:

- **Component Name**—Identifies the adapter action component.
- **Adapter Mode**—Adapters can define a mode. For example, *TDataSetAdapter* supports Edit, Insert, and Browse modes. An adapter action may execute differently depending on the mode. For example, the *TDataSetAdapter* Apply action adds a new record when in Insert mode, and updates a record when in Edit mode.
- **Success Page**—The success page identifies the page displayed after successful execution of the action.
- **Failure Page**—The failure page identifies the page displayed if an error occurs during execution of the action.
- **Action Request Parameters**—This identifies the parameters need by the adapter action. For example, the *TDataSetAdapter* Apply action will include the key values identifying the record to be updated.
- **Adapter Field Values**—These are the values for the adapter fields passed in the HTTP request when an HTML form is submitted. A field value can include new values entered by the end-user, the original values of the adapter field, and uploaded files.
- **Record Keys**—If an HTML form submits changes to multiple records, keys used by the adapter action component are required to uniquely identify each record so that the adapter action can be performed on each record. For example, when the *TDataSetAdapter* Apply action is performed on multiple records, the record keys are used to locate each record in the dataset before updating the dataset fields.

### Action response

The Action Response object generates an HTTP response on behalf of an adapter action component. The adapter action indicates the type of response by setting

properties within the object, or by calling methods in the Action Response object. The properties include:

- *Redirect Options*—The redirect options indicate whether to perform an HTTP redirect instead of returning HTML content.
- *Execution Status*—Setting the status to success causes the default action response to be the content of the success page identified in the Action Request.

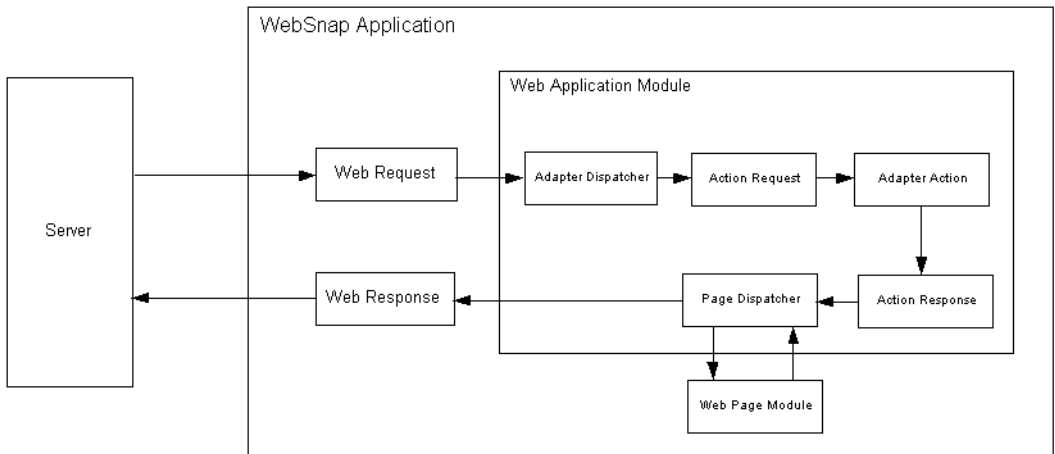
The Action response methods include:

- *RespondWithPag* —The adapter action calls this method when a particular Web page module should generate the response.
- *RespondWithComponent*—The adapter action calls this method when the response should come from the Web page module containing this component.
- *RespondWithURL*—The adapter action calls this method when the response is a redirect to a specified URL.

When responding with a page, the Action response object attempts to use the page dispatcher to generate page content. If it does not find the page dispatcher, it calls the Web Page module directly.

Figure 29.4 illustrates how action request and action response objects handle a request.

**Figure 29.2** Action request and response



### Image request

The Image Request object is responsible for breaking the HTTP request down into the information required by the adapter image field to generate an image. The types of information represented by the Image Request include:

- **Component Name** - Identifies the adapter field component.

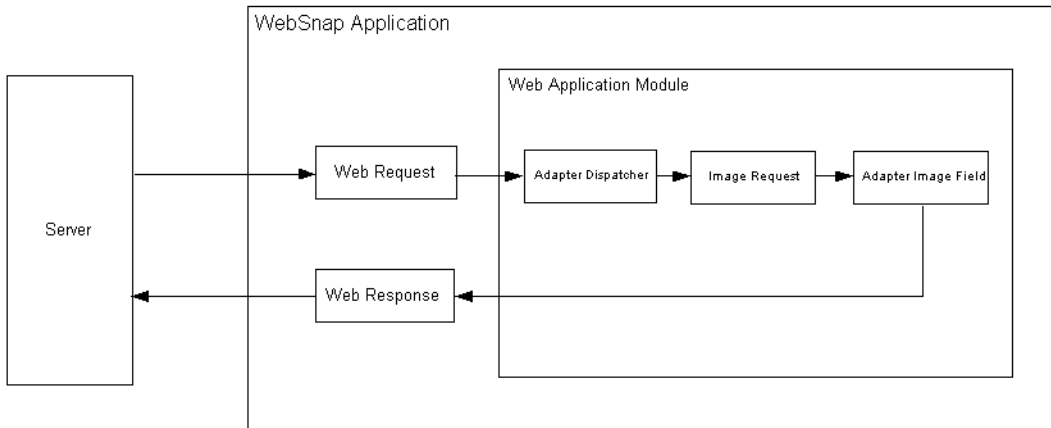
- Image Request Parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

## Image response

The Image response object contains the *TWebResponse* object. Adapter fields respond to an adapter request by writing an image to the Web response object.

Figure 29.3 illustrates how adapter image fields respond to a request.

**Figure 29.3** Image response to a request



## Dispatching action items

The Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that:

- matches the *PathInfo* portion of the target URL's request message, and
- can provide the service specified as the method of the request message.

It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response, or signals that the request has been completely handled.
- Adds to the response, and then allows other action items to complete the job.
- Defers the request to other action items.

After the dispatcher has checked all of its action items, if the message has not been handled correctly, the dispatcher checks for specially registered auto-dispatching components that do not use action items. These components are specific to multi-



tiered database applications. If the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

## Page dispatcher operation

---

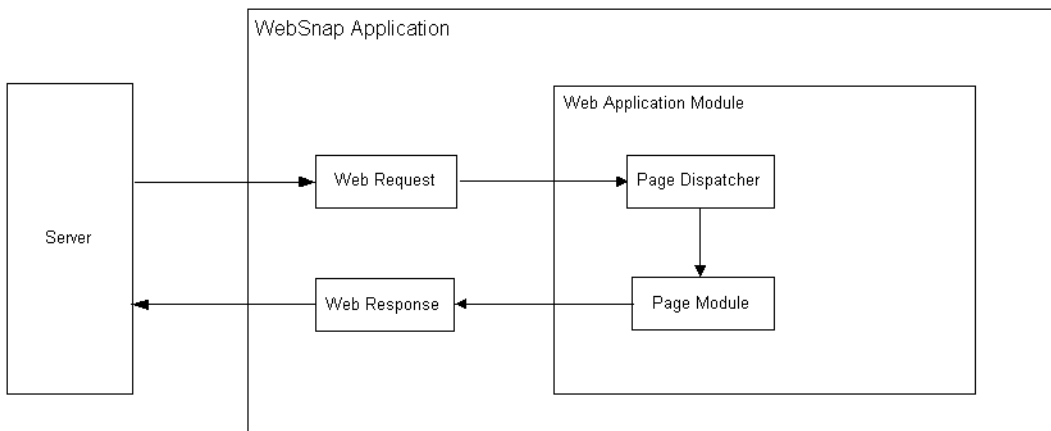
When the page dispatcher receives a client request, it determines the page name by checking the Pathinfo portion of the target URL's request message. If the pathinfo portion is not blank, the page dispatcher uses the ending word of pathinfo as the page name. If the pathinfo portion is blank, the page dispatcher tries to determine a default page name.

If the page dispatcher's *DefaultPage* property contains a page name, then the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank, and the Web application module is a page module, then the page dispatcher uses the name of the Web application module as the default page name.

If the page name is not blank, the page dispatcher searches for a Web page module with a matching name. If it finds a Web page module, then it calls that module to generate a response. If the page name is blank, or if the page dispatcher does not find a Web page module, the page dispatcher raises an exception.

Figure 29.4 shows how the page dispatcher responds to a request.

**Figure 29.4** Dispatching a page



## WebSnap tutorial

---

The following section describes the steps to build a WebSnap application. Completing the tutorial will familiarize users with the WebSnap architecture and new concepts, by incorporating the new dispatcher and adapter components into the new Web Page module. The WebSnap application demonstrates how to use WebSnap HTML components to build an application that edits the content of a table.

## Create a new application

---

To create a new WebSnap application:

### Step 1. Start the WebSnap application wizard

- 1 Run the Delphi application and select File | New | Other.
- 2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.
- 3 In the New WebSnap Application dialog box:
  - Select the Web App Debugger Executable.
  - In the CoClass field type **CountryTutorial**.
  - Select Page Module as the component type.
  - In the Page Name field type **Home**.
- 4 Click OK.

### Step 2. Save the generated files and project

To save the pascal unit file and project:

- 1 Select File | Save All.
- 2 In the File name field enter **HomeU.pas** and Click OK.
- 3 In the File name field enter **CountryU.pas** and Click OK.
- 4 In the File name field enter **CountryTutorial.dpr** and Click OK.
- 5 Click OK.

**Note** Save the unit and the project to the same directory since the application will look for the HomeU.html file in the same directory as the executable.

### Step 3. Specify the application title

The application title is the name displayed to the end user. To specify the application title:

- 1 Select View | Project Manager.
- 2 In the Project Manager window expand CountryTutorial.exe and double click the HomeU entry.
- 3 In the Object Inspector window (bottom left), select ApplicationAdapter from the pull down list.
- 4 In the Properties tab, enter **Country Tutorial** in the ApplicationTitle field.
- 5 Click on the Preview tab in the editor window. The application title is displayed at the top of the page.

## Create a CountryTable page

---

A Web page module is used to define a published page, and as a container for data components.

### Step 1. Add a new module

To add a new module:

- 1 Select File | New | Other.
- 2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.
- 3 In the dialog box, set the Producer Type to AdapterPageProducer from the list.
- 4 In the Page Name field enter **CountryTable**.
- 5 Leave the rest of the fields and selection at their default values.
- 6 Click OK.

### Step 2. Save the new module

Save the unit to the directory of the project file. When the application runs, it searches for the CountryTableU.html file in the same directory as the executable.

- 1 Select File | Save.
- 2 In the File name field, enter **CountryTableU.pas** and Click OK.

## Add data components to the CountryTable module

---

A *TTable* component provides the data for the HTML table. The *TDataSetAdapter* component allows server side script to access the *TTable* component.

### Step 1. Add data-aware components

- 1 Select View | Project Manager.
- 2 In the Project Manager window expand CountryTutorial.exe and double click the CountryTableU entry.
- 3 Select View | Object TreeView. The Object TreeView window (left hand side) becomes active.
- 4 Select the Data Access tab in the tool palette.
- 5 Select a Table component (left-click and hold) and drag the component to the Object TreeView window.
- 6 Select a Session component (left-click and hold) and drag the component to the Object TreeView window.
- 7 Select the Session component in the Object TreeView window. This displays the Session component values in the Object Inspector window.

- 8 In the Object Inspector window, set the `AutoSessionName` property to `True`.
- 9 Select the Table component in the Object TreeView window. This displays the Table component values in the Object Inspector window.
- 10 In the Object Inspector window, change the following properties:
  - Set the `Active` property to `True`.
  - Set the `DatabaseName` property to `DBDEMOS`.
  - In the `Name` property, type **Country**.
  - Set the `TableName` property to `country.db`.

**Note** The Session component is required because we are using the BDE component (`TTable`) in a multi threaded application.

## Step 2. Specify a key field

The key field is used to identify records within a table. This becomes important when we add an edit page to the application. To specify a key field:

- 1 In the Object Tree View window, expand the Session and `DBDemos` node, and select the `country.db` node. This node is the Country Table component.
- 2 Right-click on the `country.db` node and select Fields Editor.
- 3 Right-click in the `CountryTable.Country` editor window and select the Add All Fields command.
- 4 Select the Name field from the list of added fields.
- 5 In the Object Inspector window, expand the `ProviderFlags` property.
- 6 Set the `pfInKey` property value to `True`.

## Step 3. Add an adapter component

To expose the data in the `TTable` server-side scripting, you must include a `DataSetAdapter` (`TDataSetAdapter`) component. To add such a component:

- 1 Select the WebSnap tab in the tool palette.
- 2 Select the `DataSetAdapter` component (left-click and hold) and drag the component to the Object TreeView window.
- 3 In the Object Inspector window, change the following properties:
  - Set the `DataSet` field to `Country`.
  - In the `Name` field type **Adapter**.

## Create a grid to display the data

---

The *AdapterPageProducer* leverages server-side script to quickly build an HTML table.

### Step 1. Add a grid

To add a grid to display the data from the Country table:

- 1 Select View | Project Manager.
- 2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryTableU entry.
- 3 Select View | Object TreeView. The Object TreeView window (left-hand side) becomes active.
- 4 Expand the *AdapterPageProducer* component.
- 5 Right-click on WebPageItems entry and select New Component.
- 6 In the Add Web Component window, select AdapterForm, then Click OK. An AdapterForm1 component appears in the Object TreeView window.
- 7 Right-click on AdapterForm1 and select New Component.
- 8 In the Add Web Component window, select AdapterGrid then click OK. An AdapterGrid1 component appears in the Object TreeView window.
- 9 In the Object Inspector window, set the Adapter property to Adapter.
- 10 To preview the Page, select the CountryTableU.pas tab in the code editor window, and select the Preview tab at the bottom. If the Preview tab is not shown, use the right arrow at the bottom to scroll through the tabs.
- 11 Select the HTML Script tab to view the JScript generated by the WebSnap components.

### Step 2. Add editing commands to the grid

Users may need to update the content of the table. To allow users to make such updates, such as deleting or inserting a row, add command components.

To add command components:

- 1 In the Object TreeView window for the CountryTable, expand the *AdapterPageProducer* component and all its branches.
- 2 Right-click on the AdapterGrid1 component and select Add All Columns.
- 3 Right-click on the AdapterGrid1 component and select New Component. An AdapterCommandColumn1 entry is added to the AdapterGrid1 component.
- 4 Right-click on AdapterCommandColumn1 and choose Add Commands.
- 5 Multi-select the DeleteRow, EditRow, and NewRow commands; then click OK.
- 6 To preview the Page, click on the Preview tab at the bottom of the code editor.

## Add an edit form

---

Create a Web page module to be the Edit form for the country table.

### Step 1. Add a new module

To add a new WebSnap page module:

- 1 Select File | New | Other.
- 2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.
- 3 In the dialog box, set the Producer Type to AdapterPageProducer from the list.
- 4 In the Page Name field, enter **CountryForm**.
- 5 Uncheck the Published box.
- 6 Leave the rest of the fields and selections at their default values.
- 7 Click OK.

### Step 2. Save the new module

Save the unit to the directory as the project file. When the application runs, it will look for the CountryFormU.html file in the same directory as the executable.

- 1 Select File | Save.
- 2 In the File name field enter **CountryFormU.pas** and Click OK.

### Step 3. Use the CountryTableU unit

Add CountryTableU unit to the **uses** clause to allow the module access to the Adapter component.

- 1 Select File | Use Unit.
- 2 Select CountryTableU from the list then click OK.

### Step 4. Add input fields

Add components to the *AdapterPageProducer* component to generate data entry fields in the HTML form.

To add input fields:

- 1 Select View | Project Manager.
- 2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryFormU entry.
- 3 Select View | Object TreeView. The Object TreeView window (left-hand side) becomes active.
- 4 In the Object TreeView window, expand the *AdapterPageProducer* component, right-click on WebPageItems, and select New Component.

- 5 Select `AdapterForm`, then click OK. An `AdapterForm1` entry appears in the Object TreeView window.
- 6 Right-click on `AdapterForm1` and select `New Component`.
- 7 Select `AdapterFieldGroup` then click OK. An `AdapterFieldGroup1` entry appears in the Object TreeView window.
- 8 In the Object Inspector window, set the `Adapter` property to `CountryTable.Adapter`.
- 9 To preview the Page, click the `Preview` tab at the bottom of the code editor.

### Step 5. Add buttons

Add components to the `AdapterPageProducer` component to generate the submit buttons in the HTML form. To add components:

- 1 In the Object TreeView, expand the `AdapterPageProducer` component and all its branches.
- 2 Right-click on `AdapterForm1` entry and select `New Component`.
- 3 Select `AdapterCommandGroup` then click OK. An `AdapterCommandGroup1` entry appears in the Object TreeView window.
- 4 In the Object Inspector window, set the `DisplayComponent` property to `AdapterFieldGroup1`.
- 5 Right-click on `AdapterCommandGroup1` entry and select `Add Commands`.
- 6 Multi-select the `Cancel`, `Apply`, and `Refresh Row` commands; then click OK.
- 7 To preview the Page, click the `Preview` tab at the bottom of the code editor window. If the preview does not show the country form, click on the `Code` tab and then re-click the `Preview` tab.

### Step 6. Link form actions to the grid page

When the user clicks a button, an adapter action is executed. To specify which page to display after an adapter action is executed:

- 1 In the Object TreeView, expand `AdapterCommandGroup1` to show the `CmdCancel`, `CmdApply`, and `CmdRefreshRow` entries.
- 2 Select `CmdCancel`. In the Object Inspector window, type **CountryTable** in the `PageName` property.
- 3 Select `CmdApply`. In the Object Inspector window, type **CountryTable** in the `PageName` property.

### Step 7. Link grid actions to the form page

To specify which page to display after an adapter action is executed by pushing a button in the grid:

- 1 Select `View | Project Manager`.

- 2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryTableU entry.
- 3 In the Object TreeView window, expand the *AdapterPageProducer* component and all its branches, to show the CmdNewRow, CmdEditRow, and CmdDeleteRow entries. These entries appear under the AdapterCommandColumn1 entry.
- 4 Select CmdNewRow. In the Object Inspector window, type **CountryForm** in the PageName property.
- 5 Select CmdEditRow. In the Object Inspector window, type **CountryForm** in the PageName property.
- 6 To verify that the application is working and that all buttons perform some action, run the application.

**Note** There will be no indication of database errors, such as an invalid type. For example, try adding a new country with an invalid value (for example, 'abc') in the Area field.

## Add error reporting

---

To report errors to the end user, an AdapterErrorList component is used to display errors that occur while executing adapter actions that edit the country table.

### Step 1. Add error support to the grid

- 1 In the Object TreeView for CountryTable, expand the *AdapterPageProducer* component and all its branches to show AdapterForm1.
- 2 Right-click on AdapterForm1 and select New Component.
- 3 Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Object TreeView window.
- 4 Move AdapterErrorList1 above AdapterGrid1 (either by dragging it or by using the upward-pointing arrow in the Object TreeView toolbar).
- 5 In the Object Inspector window, set the Adapter property to Adapter.

### Step 2. Add error support to the form

- 1 In the Object TreeView for CountryForm, expand the *AdapterPageProducer* component and all its branches to show AdapterForm1.
- 2 Right-click on AdapterForm1 and select New Component.
- 3 Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Object TreeView window.
- 4 Move AdapterErrorList1 above AdapterGrid1 (either by dragging it or by using the upward-pointing arrow in the Object TreeView toolbar).
- 5 In the Object Inspector window, set the Adapter property to CountryTable.Adapter.



### Step 3. Test the error-reporting mechanism

To test Grid Errors:

- 1 Run the application, and browse to the CountryTable page.
- 2 Start up another instance of your browser and browse to the CountryTable page.
- 3 Click the DeleteRow button on the first row in the grid.
- 4 Without refreshing the second browser, click the DeleteRow button on the first row in the grid.
- 5 An error message will be displayed above the grid.

To test Form Errors:

- 1 Run the application, and browse to the CountryTable page.
- 2 Click on the EditRow Button.
- 3 The CountryForm page is displayed.
- 4 Change the area field to 'abc', and click the Apply Button.
- 5 An error message will be displayed above the first field.

### Run the completed application

---

To run the completed application:

- 1 Select Run | Run. You will see a form displayed. Web App Debugger executable Web applications are COM servers, and the form you see is the console window for the COM server. The first time that you run the project, it registers the COM object that can be accessed directly by Web App Debugger.
- 2 Select Tools | Web App Debugger.
- 3 Click on the default URL link to display the ServerInfo page. The ServerInfo page displays the names of all registered Web Application Debugger executables.
- 4 Select CountryTutorial in the drop-down list and click on the Go button.

## Working with XML documents

XML (Extensible Markup Language) is a markup language for describing structured data. It is similar to HTML, except that the tags describe the structure of information rather than its display characteristics. XML documents provide a simple, text-based way to store information so that it is easily searched or edited. They are often used as a standard, transportable format for data in Web applications, business-to-business communication, and so on.

XML documents provide a hierarchical view of a body of data. Tags in the XML document describe the role or meaning of each data element, as illustrated in the following document, which describes a collection of stock holdings:

```
<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM "sth.dtd">
<StockList>
  <Stock exchange=NASDAQ>
    <name>Inprise (Borland)</name>
    <price>15.375</price>
    <symbol>INPR</symbol>
    <shares>100</shares>
  </Stock>
  <Stock exchange=NYSE>
    <name>Pfizer</name>
    <price>42.75</price>
    <symbol>PFE</symbol>
    <shares type=preferred>25</shares>
  </Stock>
</StockList>
```

This example illustrates a number of typical elements in an XML document. The first line is a processing instruction. It provides information on how to interpret the file, but does not include any data.

The second line, which begins with the `<!DOCTYPE>` tag, is a document type declaration. It names the structure of the document and references another file (`sth.dtd`) that describes that structure. In this case, the structure is described by a

Document Type Definition (DTD) file. Other types of files that describe the structure of an XML document include Reduced XML Data (XDR) and XML schemas (XSD).

The remaining lines are organized into a hierarchy with a single root node (the <StockList> tag). Each node in this hierarchy contains either a set of child nodes, or a text value. Some of the tags (the <Stock> and <shares> tags) include attributes, which are Name=Value pairs that provide details on how to interpret the tag.

Although it is possible to work directly with the text in an XML document, typically applications use some sort of additional tools for parsing and editing the data. W3C defines a set of standard interfaces for representing a parsed XML document called the Document Object Model (DOM). A number of vendors provide XML parsers that implement the DOM interfaces to let you interpret and edit XML documents more easily.

Delphi provides a number of additional tools for working with XML documents. These tools use a DOM parser that is provided by another vendor, and make it even easier to work with XML documents. This chapter describes those tools.

**Note** In addition to the tools described in this chapter, Delphi comes with tools and components for converting XML documents to data packets that integrate into the Delphi database architecture. For details on tools for integrating XML documents into database applications, see Chapter 26, “Using XML in database applications.”

## Using the Document Object Model

---

The Document Object Model (DOM) is a set of standard interfaces for representing a parsed XML document. Delphi ships with two DOM implementations (from Microsoft and from IBM). In addition, it includes a registration mechanism that lets you integrate additional DOM implementations by other vendors into the Delphi XML framework.

The XMLDOM unit includes declarations for all the DOM interfaces defined in the W3C XML DOM level 2 specification. Each DOM vendor provides an implementation for these interfaces.

- To use the Microsoft implementation, include the MSXMLDOM unit in your uses clause. Because the Microsoft implementation is COM-based, you must also register the msxml.dll library as a COM server. You can use Regsvr32.exe to register this DLL.
- To use the IBM implementation, include the IBMXMLDOM unit in your uses clause.
- To use another DOM implementation, you must create a unit that includes a function to return the top-level interface (*IDOMImplementation*). Your unit should register this function by calling the global *RegisterDOMImplementation* procedure.

Some vendors supply extensions to the standard DOM interfaces. To allow you to use these extensions, the XMLDOM unit also defines an *IDOMNodeEx* interface. *IDOMNodeEx* is a descendant of the standard *IDOMNode* that includes the most useful of these extensions.

You can work directly with the DOM interfaces to parse and edit XML documents. Simply call the *GetDOM* function to obtain an *IDOMImplementation* interface, which you can use as a starting point.

**Note** For detailed descriptions of the DOM interfaces, see the declarations in the XMLDOM unit, the documentation supplied by your DOM Vendor, or the specifications provided on the W3C web site ([www.w3.org](http://www.w3.org)).

You may find it more convenient to use Delphi's XML classes rather than working directly with the DOM interfaces. These are described below.

## Working with XML components

---

Delphi defines a number of classes and interfaces for working with XML documents. These simplify the process of loading, editing, and saving XML documents.

### Using TXMLDocument

---

The starting point for working with an XML document is the *TXMLDocument* component. The following steps describe how to use *TXMLDocument* to work directly with an XML document:

- 1 Add a *TXMLDocument* component into your form or data module.  
*TXMLDocument* appears on the Internet page of the Component palette.
- 2 Set the *DOMVendor* property to specify the DOM implementation you want the component to use for parsing and editing an XML document. The Object Inspector lists all the currently registered DOM vendors. For information on DOM implementations, see "Using the Document Object Model" on page 30-2.
- 3 Depending on your implementation, you may want to set the *ParseOptions* property to configure how the underlying DOM implementation parses the XML document.
- 4 If you are working with an existing XML document, specify the document:
  - If the XML document is stored in a file, set the *FileName* property to the name of that file.
  - You can specify the XML document as a string instead by using the *XML* property.
- 5 Set the *Active* property to *True*.

Once you have an active *TXMLDocument* object, you can traverse the hierarchy of its nodes, reading or setting their values. The root node of this hierarchy is available as the *DocumentElement* property.

## Working with XML nodes

---

Once an XML document has been parsed by a DOM implementation, the data it represents is available as a hierarchy of nodes. Each node corresponds to a tagged element in the document. For example, given the following XML:

```
Component palette<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM "sth.dtd">
<StockList>
  <Stock exchange=NASDAQ>
    <name>Inprise (Borland)</name>
    <price>15.375</price>
    <symbol>INPR</symbol>
    <shares>100</shares>
  </Stock>
  <Stock exchange=NYSE>
    <name>Pfizer</name>
    <price>42.75</price>
    <symbol>PFE</symbol>
    <shares type=preferred>25</shares>
  </Stock>
</StockList>
```

*TXMLDocument* would generate a hierarchy of nodes as follows: The root of the hierarchy would be the *StockList* node. *StockList* would have two child nodes, which correspond to the two *Stock* tags. Each of these two child nodes would have four child nodes of its own (*name*, *price*, *symbol*, and *shares*). Those four child nodes would act as leaf nodes. The text they contain would appear as the value of each of the leaf nodes.

**Note** This division into nodes differs slightly from the way a DOM implementation generates nodes for an XML document. In particular, a DOM parser treats all tagged elements as internal nodes. Additional nodes (of type text node) would be created for the values of the *name*, *price*, *symbol*, and *shares* nodes. These text nodes would then appear as the children of the *name*, *price*, *symbol*, and *shares* nodes.

Each node is accessed through an *IXMLNode* interface, starting with the root node, which is the value of the XML document component's *DocumentElement* property.

### Working with a node's value

Given an *IXMLNode* interface, you can check whether it represents an internal node or a leaf node by checking the *IsTextElement* property.

- If it represents a leaf node, you can read or set its value using the *Text* property.
- If it represents an internal node, you can access its child nodes using the *ChildNodes* property.

Thus, for example, using the XML document above, you can read the price of Inprise's stock as follows:

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
Price := InpriseStock.ChildNodes['price'].Text;
```

## Working with a node's attributes

If the node includes any attributes, you can work with them using the *Attributes* property. You can read or change an attribute value by specifying an existing attribute name. You can add new attributes by specifying a new attribute name when you set the *Attributes* property:

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
InpriseStock.ChildNodes['shares'].Attributes['type'] := 'common';
```

## Adding and deleting child nodes

You can add child nodes using the *AddChild* method. *AddChild* creates new nodes that correspond to tagged elements in the XML document. Such nodes are called element nodes.

To create a new element node, specify the name that appears in the new tag and, optionally, the position where the new node should appear. For example, the following code adds a new stock listing to the document above:

```
var
  NewStock: IXMLNode;
  ValueNode: IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems';
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
  ValueNode.Text := '25';
end;
```

An overloaded version of *AddChild* lets you specify the namespace in which the tag name is defined.

You can delete child nodes using the methods of the *ChildNodes* property. *ChildNodes* is an *IXMLNodeList* interface, which manages the children of a node. You can use its *Delete* method to delete a single child node that is identified by position or by name. For example, the following code deletes the last stock listed in the document above:

```
StockList := XMLDocument1.DocumentElement;
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
```

# Abstracting XML documents with the Data Binding wizard

---

Although it is possible to work with an XML document using only the *TXMLDocument* component and the *IXMLNode* interface it surfaces for the nodes in that document, or even to work exclusively with the DOM interfaces (avoiding even *TXMLDocument*), you can write code that is much simpler and more readable by using the XML Data Binding wizard.

The Data Binding wizard takes an XML schema or data file and generates a set of interfaces that map on top of it. For example, given XML data that looks like the following:

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

The Data Binding wizard generates the following interface (along with a class to implement it):

```
ICustomer = interface(IXMLNode)
  property id: Integer read Getid write Setid;
  property name: DOMString read Getname write Setname;
  property phone: DOMString read Getphone write Setphone;
  function Getid: Integer;
  function Getname: DOMString;
  function Getphone: DOMString;
  procedure Setid(Value: Integer);
  procedure Setname(Value: DOMString);
  procedure Setphone(Value: DOMString);
end;
```

Every child node is mapped to a property whose name matches the tag name of the child node and whose value is the interface of the child node (if the child is an internal node) or the value of the child node (for leaf nodes). Every node attribute is also mapped to a property, where the property name is the attribute name and the property value is the attribute value.

In addition to creating interfaces (and implementation classes) for each tagged element in the XML document, the wizard creates a global function for obtaining the interface to the root node. For example, if the XML above came from a document whose root node had the tag <Customers>, the Data Binding wizard would create the following global routine:

```
function GetCustomers(XMLDoc: TXMLDocument): ICustomers;
```

Using the generated interfaces simplifies your code, because they reflect the structure of the XML document more directly. For example, instead of writing code such as the following:

```
CustName := XMLDocument1.DocumentElement.ChildNodes[0].ChildNodes['name'].Value;
```

Your code would look as follows:

```
CustName := GetCustomers(XMLDocument1)[0].Name;
```

Note that the interfaces generated by the Data Binding wizard all descend from *IXMLNode*. This means you can still add and delete child nodes in the same way as when you do not use the Data Binding wizard. (See “Adding and deleting child nodes” on page 30-5.) In addition, when child nodes represent repeating elements (when all of the children of a node are of the same type), the parent node is given two methods, *Add*, and *Insert*, for adding additional repeats. These methods are simpler than using *AddChild*, because you do not need to specify the type of node to create.

## Using the XML Data Binding wizard

---

To use the Data Binding wizard,

- 1 Choose File | New | Other and select the icon labeled XML Data Binding from the bottom of the New page.
- 2 The XML Data Binding wizard appears.
- 3 On the first page of the wizard
  - specify the XML document or schema for which you want to generate interfaces. This can be a sample XML document, a Document Type Definition (.dtd) file, a Reduced XML Data (.xdr) file, or an XML schema (.xsd) file.
- 4 Click the Options button to specify the naming strategies you want the wizard to use when generating interfaces and implementation classes and the default mapping of types defined in the schema to Pascal data types.
- 5 Move to the second page of the wizard. This page lets you provide detailed information about every node type in the document or schema. At the left is a tree view that shows all of the node types in the document. For complex nodes (nodes that have children), the tree view can be expanded to display the child elements. When you select a node in this tree view, the right-hand side of the dialog displays information about that node and lets you specify how you want the wizard to treat that node.
  - The Source Name control displays the name of the node type in the XML schema.
  - The Source Type control displays the type of the node's value, as specified in the XML schema.
  - The Documentation control lets you add comments to the schema describing the use or purpose of the node.
  - If the wizard generates code for the selected node (that is, if it is a complex type for which the wizard generates an interface and implementation class, or if it is one of the child elements of a complex type for which the wizard generates a property on the complex type's interface), you can use the Generate Binding check box to specify whether you want the wizard to generate code for the node. If you uncheck Generate Binding, the wizard does not generate the interface or implementation class for a complex type, or does not create a property in the parent interface for a child element or attribute.
  - The Binding Options section lets you influence the code that the wizard generates for the selected element. For any node, you can specify the Identifier Name (the name of the generated interface or property). In addition, for interfaces, you must indicate which one represents the root node of the document. For nodes that represent properties, you can specify the type of the property and, if the property is not an interface, whether it is a read-only property.



- 6 Once you have specified what code you want the wizard to generate for each node, move to the third page. This page lets you choose some global options about how the wizard generates its code and lets you preview the code that will be generated, and lets you tell the wizard how to save your choices for future use.
  - To preview the code the wizard generates, select an interface in the Binding Summary list and view the resulting interface definition in the Code Preview control.
  - Use the Data Binding Settings to indicate how the wizard should save your choices. You can store the settings as annotations in a schema file that is associated with the document (the schema file specified on the first page of the dialog), or you can name an independent schema file that is used only by the wizard.
- 7 When you click Finish, the Data Binding wizard generates a new unit that defines interfaces and implementation classes for all of the node types in your XML document. In addition, it creates a global function that takes a *TXMLDocument* object and returns the interface for the root node of the data hierarchy.

## Using code that the XML Data Binding wizard generates

---

Once the wizard has generated a set of interfaces and implementation classes, you can use them to work with XML documents that match the structure of the document or schema you supplied to the wizard. Just as when you are using only the built-in XML components that ship with Delphi, your starting point is the *TXMLDocument* component that appears on the Internet page of the Component palette.

To work with an XML document, use the following steps:

- 1 Place a *TXMLDocument* component in your form or data module.
- 2 Bind the *TXMLDocument* to an XML document by setting the *FileName* property. (As an alternative approach, you can use a string of XML by setting the *XML* property at runtime.)
- 3 In your code, call the global function that the wizard created to obtain an interface for the root node of the XML document. For example, if the root element of the XML document was the tag `<StockList>`, by default, the wizard generates a function *GetStockListType*, which returns an *IStockListType* interface:
 

```
var
    StockList: IStockListType;
begin
    StockList := GetStockListType(XMLDocument1);
```
- 4 This interface has properties that correspond to the subnodes of the document's root element, as well as properties that correspond to that root element's attributes. You can use these to traverse the hierarchy of the XML document, modify the data in the document, and so on.
- 5 To save any changes you make using the interfaces generated by the wizard, call the *TXMLDocument* component's *SaveToFile* method or read its *XML* property.

## Using Web Services

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided.

Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as well.

Delphi's support for Web Services is designed to work using SOAP (Simple Object Access Protocol). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. It uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. For more information about SOAP, see the SOAP specification available at

<http://www.w3.org/TR/SOAP/>

**Note** Although Delphi's support for Web Services is based on SOAP and HTTP, the framework is sufficiently general that it can be expanded to use other encoding and communications protocols.

Delphi's SOAP-based technology is available on Windows and will later be implemented on Linux, so that it can form the basis of cross-platform distributed applications. There is no special client runtime software to install, as you must have when distributing applications using CORBA. Because this technology is based on HTTP messages, it has the advantage that it is widely available on a variety of machines. Support for Web Services is built on top of Delphi's cross-platform Web server application architecture.

You can use Delphi to build both servers that implement Web Services and clients that call on those services. If you use Delphi to create both the server and client applications, you can share a single unit that defines the interfaces for the Web Services. In addition, you can write Delphi clients for arbitrary servers that implement Web Services that respond to SOAP messages, and Delphi servers that publish Web Services that can be used by arbitrary clients.

When either the client or server is not written using Delphi, you can publish or import information on what interfaces are available and how to call them using a WSDL (Web Service Definition Language) document. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard can import a published WSDL document, providing you with the interface definitions and connection information you need.

## Writing Servers that support Web Services

---

In Delphi, servers that support Web Services are built using invocable interfaces. Invokable interfaces are interfaces that are compiled to include runtime type information (RTTI). This RTTI is used when interpreting incoming method calls from clients so that they can be correctly marshaled.

In addition to the invocable interfaces, and the classes that implement them, your server requires two components: a dispatcher and an invoker. The dispatcher (*THTTPSoapDispatcher*) is an auto-dispatching component that receives incoming SOAP messages and passes them on to the invoker. The invoker (*THTTPSoapPascalInvoker*) interprets the SOAP message, identifies the invocable interface it calls, executes the call and assembles the response message.

**Note** *THTTPSoapDispatcher* and *THTTPSoapPascalInvoker* are designed to respond to HTTP messages containing a SOAP request. The underlying architecture is sufficiently general, however, that it can support other protocols with the substitution of different dispatcher and invoker components.

Once you register your invocable interfaces and their implementation classes, the dispatcher and invoker automatically handle any messages that identify those interfaces in the SOAP Action header of the HTTP request message.

### Building a Web Service server

---

Use the following steps to build a server application that implements a Web Service:

- 1 Define the interfaces that make up your Web Service. These interface definitions must be invocable interfaces. It is a good idea to create your interface definitions in their own units, separate from the unit that contains the implementation classes. In this way, the unit that defines the interfaces can be included in both the server and client applications. In the initialization section of this unit, add code to register the interfaces. For details on writing and registering invocable interfaces, see “Defining invocable interfaces” on page 31-3.
- 2 If your interface uses any complex (non-scalar) types, you must make sure these can be marshaled correctly. The Web Service application can only handle these using special objects that contain runtime type information (RTTI) that describes their structure. For details on creating and registering objects to represent complex types, see “Using complex types in invocable interfaces” on page 31-5.

- 3 Define and implement classes that implement the invocable interfaces you defined in step 1. For each implementation class, you may also need to create a factory procedure that instantiates the class. In the initialization section of this unit, add code to register the implementation class. This process is described in “Creating and registering the implementation” on page 31-6.
- 4 If your application raises an exception when attempting to execute a SOAP request, the exception will be automatically encoded in a SOAP fault packet, which is returned instead of the results of the method call. If you want to convey more information than a simple error message, you can create your own exception classes that are encoded and passed to the client. This is described in “Creating custom exception classes for Web Services” on page 31-7.
- 5 Choose File | New | Other, and on the Web Services page, double-click the Web Service application icon. Choose the type of Web server application you want to have implement your Web Service. For information about different types of Web Server applications, see “Types of Web server applications” on page 27-6.
- 6 The wizard generates a new Web Service application that includes an invoker component (*THHTTPSOAPPascalInvoker*) and a dispatcher component (*THHTTPSoapDispatcher*). The invoker converts between SOAP messages and the methods of any interfaces you registered in step 1. The dispatcher automatically responds to incoming SOAP messages and forwards them to the invoker. You can use its *WebDispatch* property to identify the HTTP request messages to which your application responds. This involves setting the *PathInfo* property to indicate the path portion of any URL directed to your application, and the *MethodType* property to indicate the method header for request messages.
- 7 Choose Project | Add To Project, and add the units you created in steps 1 through 4 to your Web server application.
- 8 If you want your application to work with clients that are not written using Delphi, publish a WSDL document that defines your interfaces and how to call them. For details on how to generate a WSDL document that describes your Web Service application, see “Generating WSDL documents for a Web Service application” on page 31-7.

## Defining invocable interfaces

To create an invocable interface, you need only compile an interface with the {\$M+} compiler option. The descendant of any invocable interface is also invocable. However, if an invocable interface descends from another interface that is not invocable, clients of your Web Service server can only call the methods defined in the invocable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be called by clients.

Delphi defines a base invocable interface, *IInvokable*, that can be used as the basis of any interface exposed to clients by a Web Service server. *IInvokable* is the same as the base interface (*IInterface*), except that it is compiled using the {\$M+} compiler option so that it and all its descendants are compiled to include RTTI.

For example, the following code defines an invocable interface that contains two methods for encoding and decoding numeric values:

```
IEncodeDecode = interface(IInvokable)
    ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
```

Before a Web Service application can use this invocable interface, it must be registered with the invocation registry. On the server, the invocation registry entry allows the invoker component (*THHTPSOAPPascalInvoker*) to identify an implementation class to use for executing interface calls. On client applications, an invocation registry entry allows components to look up information that identifies the invocable interface and supplies information on how to call it.

In the initialization section of the unit that defines the interface, add code to register the interface with the invocation registry. To access the invocation registry, add the *InvokeRegistry* unit to the uses clause of your unit. The *InvokeRegistry* unit declares a global variable, *InvRegistry*, which maintains in memory a catalog of all registered invocable interfaces, their implementation classes, and the factories that create instances of the implementation classes.

When you are finished, the unit that defines the interface should look something like the following:

```
unit EncodeDecode;

interface
type
IEncodeDecode = interface(IInvokable)
    ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;

implementation
uses InvokeRegistry;

initialization
InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
```

Because the interfaces of Web Services must have a namespace to identify them among all the interfaces in all possible Web Services, when you register an interface the invocation registry automatically generates a namespace for the interface. The default namespace is built from a string that uniquely identifies the application (the *AppNamespacePrefix* variable), the interface name, and the name of the unit in which it is defined.

**Tip** It is a good idea to keep the unit that defines your invocable interfaces separate from the unit in which you write the classes that implement them. This unit can then be included in both the client and the server application. Because the generated namespace includes the name of the unit in which the interface is defined, sharing the same unit in both client and server applications enables them to automatically use the same namespace.

## Using complex types in invocable interfaces

The invoker component (*THTTPSOAPascalInvoker*) automatically knows how to marshal scalar types on invocable interfaces. It can also handle dynamic arrays, as long as they are registered with the remotable class registry (see below). However, you must provide additional support if you want to transmit data in more complex types such as static arrays, interfaces, records, sets, or classes. This support must take the form of a class that includes runtime type information (RTTI), which the invoker can use to convert between data in the SOAP stream and type values.

Use *TRemotable* as a base class when defining a class to represent a complex data type on an invocable interface. For example, in the case where you would ordinarily pass a record as a parameter, you would instead define a *TRemotable* descendant where every member of the record is a published property on your new class.

If the value of your new *TRemotable* descendant represents to a scalar type in a WSDL document that does not correspond to an Object Pascal scalar type, you should use *TRemotableXS* as a base class instead. *TRemotableXS* is a *TRemotable* descendant that introduces two methods for converting between your new class and its string representation. Provide these methods by overriding the *XSToNative* and *NativeToXS* methods.

In the initialization section of the unit that defines the *TRemotable* descendant, you must register this class with the remotable class registry. Access the remotable class registry by adding the *InvokeRegistry* unit to the uses clause. This unit declares a global variable, *RemClassRegistry*, which maintains a catalog of all registered remotable classes, and an indication of whether their values can be transmitted as strings. For example, the following line comes from the *XSBuiltIns* unit. It registers *TXSDateTime*, a *TRemotable* descendant that represents *TDateTime* values:

```
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNamespace, 'dateTime', True);
```

The first parameter is the name of the *TRemotable* descendant. The second is a uniform resource identifier (URI) that uniquely identifies the namespace of the new class. If you supply an empty string, the registry can generate a URI for you. The third parameter is the name of the data type your class represents. If you supply an empty string, the registry simply uses the class name. The last parameter indicates whether the value of class instances can be transmitted as a string (whether you implemented the *XSToNative* and *NativeToXS* methods).

**Tip** It is a good idea to implement and register *TRemotable* descendants in a separate unit from the rest of your server application, including from the units that declare and register invocable interfaces. In this way, you can use the unit that defines your type in both the client and server, and you can use the type for more than one interface.

If you are using dynamic arrays for parameters, you do not need to create a remotable class to represent them, but you do have to register them with the remotable class registry. Thus, for example, if your interface uses a type such as the following:

```
type
  TDateTimeArray = array of TXSDateTime;
```

You must add the following registration to the initialization section of the unit where you declare this dynamic array:

```
RemClassRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNamespace, 'D TArray', False);
```

The parameters are the same as those used by *RegisterXSClass*, except for the first which takes a pointer to the type information of the dynamic array rather than a class reference.

## Creating and registering the implementation

The simplest way to write an implementation for an invocable interface is to create a class that descends from *TInvokableClass*. Add the class declaration, including the invocable interfaces you are supporting, and then type *Ctrl+Shift+C* to invoke class completion. The interface members appear in your class declaration, and empty methods appear in the implementation section of the unit.

For example, the declaration for an implementation class implement the interface declared in “Defining invocable interfaces” above might look like the following:

```
TEncodeDecode = class(TInvokableClass, IEncodeDecode)
protected
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
```

In the implementation section of the unit that declares this class, fill in the *EncodeValue* and *DecodeValue* methods.

Once you have created an implementation class, you must register this class with the invocation registry. The invocation registry uses this to identify the class that implements a registered interface and to make it available to the invoker component when the invoker needs to call the interface. To register the implementation class, add a call the *RegisterInvokableClass* method of the global *InvRegistry* variable to the initialization section of your implementation unit:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode);
```

You can also create implementation classes that do not descend from *TInvokableClass*. In this case, however, you must provide a factory procedure that the invocation registry can call to create instances of your class.

The factory procedure must be of type *TCreateInstanceProc*. It returns an instance of your implementation class. If the procedure creates a new instance, the object should free itself when the reference count on its interface drops to zero, as the invocation registry does not explicitly free object instances. As an alternative, the factory procedure can return a reference to a global instance that is shared by all callers. The following code illustrates this latter approach:

```
procedure CreateEncodeDecode(out obj: TObject);
begin
    if FEncodeDecode = nil then
    begin
        FEncodeDecode := TEncodeDecode.Create;
        {save a reference to the interface so that the global instance doesn't free itself }
        FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
    end;
    obj := FEncodeDecode; { return global instance }
end;
```

When using a factory procedure, supply the factory procedure as a second parameter to the *RegisterInvokableClass* method:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
```

## Creating custom exception classes for Web Services

When your Web Service application raises an exception in the course of trying to execute a SOAP request, it automatically encodes information about that exception in a SOAP fault packet, which it returns instead of the results of the method call. The client application then raises the exception.

By default, the client application merely raises a generic exception (*Exception*) with the error message in the SOAP fault packet. However, you can transmit additional exception information by using an exception class that descends from *ERemotableException*. The values of any published properties you add to your exception class are included in the SOAP fault packet so that the client can raise an equivalent exception.

To use an *ERemotableException* descendant, you must register it with the remotable class registry. Thus, in the unit that defines your *ERemotableException* descendant, you must add the *InvokeRegistry* unit to the uses clause and add a call to the *RegisterXSClass* method of the global *RemClassRegistry* variable.

If the client uses the same unit that defines and registers your *ERemotableException* descendant, then when it receives the SOAP fault packet, it automatically raises an instance of the appropriate exception class, with all properties set to the values in the SOAP fault packet.

## Generating WSDL documents for a Web Service application

If you include the same units that define and register your invokable interfaces, the classes that represent complex type information, and your remotable exceptions in a Delphi client application, it can generate calls to use your Web Service. All you need to do is supply the URL where you install your Web Service application.

However, you may want to make your Web Service available to a wider range of clients. For example, you may have clients that are not written in Delphi. If you are deploying several versions of your server application, you may not want to use a single hard-coded URL for the server, but rather let the client look up the server location dynamically. For these cases, you may want to publish a WSDL document that describes the types and interfaces in your Web Service, with information on how to call them.

To publish a WSDL document that describes your Web Service, simply add a *TWSDLHTMLPublish* component to your Web Module. *TWSDLHTMLPublish* is an auto-dispatching component, which means it automatically responds to incoming messages that request a list of WSDL documents for your Web Service. Use the *WebDispatch* property to specify the path information of the URL clients must use to access the list of WSDL documents. The Web browser can then request the list of WSDL documents by specifying an URL that is made up of the location of the server application followed by the path in the *WebDispatch* property. This URL looks something like the following:

```
http://www.myco.com/MyService.dll/WSDL
```



**Tip** If you want a physical WSDL file instead, you can display the WSDL document in your Web browser and then save it to generate a WSDL document file.

It is not necessary to publish the WSDL document from the same application that implements your Web Service. To create an application that simply publishes the WSDL document, omit the units that contain the implementation objects, and only include the units that define and register invocable interfaces, remotable classes that represent complex types, and any remotable exceptions.

By default, when you publish a WSDL document, it indicates that the services are available at the same URL as the one where you published the WSDL document (but with a different path). If you are deploying multiple versions of your Web Service application, or if you are publishing the WSDL document from a different application than the one that implements the Web Service, you will need to change the WSDL document so that includes updated information on where to locate the Web Service.

To change the URL, use the WSDL administrator. The first step is to enable the administrator. You do this by setting the *AdminEnabled* property of the *TWSDLHTMLPublish* component to *True*. Then, when you use your browser to display the list of WSDL documents, it includes a button to administer them as well. Use the WSDL administrator to specify the locations (URLs) where you have deployed your Web Service application.

## Writing clients for Web Services

---

Delphi provides client-side support for calling Web Services that use a SOAP-based binding. These Web Services can be supplied by a server written in Delphi, or by any other server that defines its Web Service in a WSDL document.

If the server is not written in Delphi, you can first import the WSDL document that describes the server. This process is described below. If the server was written using Delphi, you do not need to use a WSDL document: you can simply add any units that define the invocable interfaces you want to use to your project, as well as any units that define remotable classes that represent complex types and that define remotable exceptions that the Web Service application can raise.

**Note** For information about creating the unit that defines an invocable interface in a Delphi Web Service server, see “Defining invocable interfaces” on page 31-3. For information about creating a unit that defines a remotable class for a complex type, see “Using complex types in invocable interfaces” on page 31-5. For information about creating a unit that defines a remotable exception, see “Creating custom exception classes for Web Services” on page 31-7.

### Importing WSDL documents

---

Before you can use a Web Service that was not written using Delphi, you must import a WSDL document (or XML schema file) that defines the service. The Web Services importer creates a unit that defines and registers the interfaces and types you need to use.

To use the Web Services importer, choose File | New | Other, and on the WebServices page double-click the icon labelled Web Services importer. In the dialog that appears, specify the file name of a WSDL document (or XML schema file) or provide the URL where that document is published. When you click Generate, the importer creates new units that define and register invocable interfaces for the operations defined in the document, and that define and register remotable classes for the types that the document defines.

If the WSDL document or XML schema file uses identifiers that are also Object Pascal keywords, the importer automatically adjusts their names so that the generated code can compile. When complex types are declared inline, the importer adds code to define and register the corresponding remotable class in the same unit as the invocable interface that uses them. Otherwise, types are defined and registered in a separate unit.

## Calling invocable interfaces

---

To call an invocable interface, your client application must include any units that define the invocable interfaces and any remotable classes that implement complex types. If the server is written in Delphi, these should be the same units that the server application uses to define and register these interfaces and classes. It is best to use the same unit, because when you register an invocable interface or remotable class, it is given a uniform resource identifier (URI) that uniquely identifies it. That URI is derived from the name of the interface (or class) and the name of the unit in which it is defined. If the client and server do not register the interface (or class) using the same URI, they can't communicate. If you do not use the same unit, the code that registers the interface and implementation class must explicitly specify a namespace URI to ensure that client and server use the same namespace.

If the server is not written in Delphi, or if you do not want to use the same unit in the client that you used in the server, these units can be created by the Web Services importer.

Once the client application has the declaration of an invocable interface, create an instance of *THHTPRio* for the desired interface:

```
X := THHTPRio.Create(nil);
```

Next, provide the *THHTPRio* object with the information it needs to identify the server interface and locate the server. There are two ways to supply this information:

- If the server is written in Delphi, the identification of the interface on the server is handled automatically, based on the URI that is generated for it when the interface is registered. You need only set the *URL* property to indicate the location of the server. The path portion of this URL should match the path of the dispatcher component in the server's Web Module:

```
X.URL := 'http://www.myco.com/MyService.dll/SOAP/';
```

- If the server is not written in Delphi, *THTTPrIo* must look up the URI for the interface, the information that must be included in the Soap Action header, and the location of the server from a WSDL document. You tell it how to do this using the *WSDLLocation*, *Service*, and *Port* properties:

```
X.WSDLLocation := 'Cryptography.wsdl';  
X.Service := 'Cryptography';  
X.Port := 'SoapEncodeDecode';
```

You can then use the **as** operator to cast the instance of *THTTPrIo* to the invocable interface. When you do this, it creates a vtable for the associated interface dynamically in memory, enabling you to make interface calls:

```
InterfaceVariable := X as IEncodeDecode;  
Code := InterfaceVariable.EncodeValue(5);
```

*THTTPrIo* relies on the invocation registry to obtain information about the invocable interface. If the client application does not have an invocation registry, or if the invocable interface is not registered, *THTTPrIo* can't build its in-memory vtable.

## Working with sockets

This chapter describes the socket components that let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as User Datagram Protocol (UDP), Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

### Implementing services

---

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

## Understanding service protocols

---

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

## Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the clients that use it. You can copy the API of a standard third party server (such as Apache), or you can design and publish your own API.

## Services and ports

---

Most standard services are associated, by convention, with specific port numbers. We will discuss port numbers in greater detail later. For now, consider the port number a numeric code for the service.

If you are implementing a standard service, Linux socket objects provide methods for you to look up the port number for the service. If you are providing a new service, you can specify the associated port number in the `/etc/services` file. See your Linux documentation for more information on the services file.

## Types of socket connections

---

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections.
- Listening connections.
- Server connections.

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same

capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

## Client connections

---

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket to which it wishes to connect. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

## Listening connections

---

Server sockets do not locate clients. Instead, they form passive “half connections” that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

## Server connections

---

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

# Describing sockets

---

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies

- The system on which it is running.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host and the port number.

Before you can make a socket connection, you must fully describe the sockets that form its endpoints. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

## Describing the host

---

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.ASite.com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. You can learn the host name associated with any IP address (if one already exists) by executing the following command from a command prompt:

```
nslookup IPADDRESS
```

where *IPADDRESS* is the IP address you're interested in. If your local IP address doesn't have a host name and you decide you want one, contact your network administrator.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

## Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When

you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

## Using ports

---

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

Earlier, we described port numbers as numeric codes for the services implemented by network applications. This is actually just a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

## Using socket components

---

The Internet palette page includes three socket components that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. These are:

- *TcpServer*
- *TcpClient*
- *UdpSocket*

Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the socket server calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the socket objects.



## Getting information about the connection

---

After completing the connection to a client or server socket, you can use the client or server socket object associated with your socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the local client or server socket, or use the *RemoteHost* and *RemotePort* properties to determine the address and port number used by the remote client or server socket. Use the *GetSocketAddr* method to build a valid socket address based on the host name and port number. You can use the *LookupPort* method to look up the port number. Use the *LookupProtocol* method to look up the protocol number. Use the *LookupHostName* method to look up the host name based on the host machine's IP address.

To view network traffic in and out of the socket, use the *BytesSent* and *BytesReceived* properties.

## Using client sockets

---

Add a *TcpClient* or *UdpSocket* component to your form or data module to turn your application into a TCP/IP or UDP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Each client socket component uses a single client socket object to represent the client endpoint in a connection.

### Specifying the desired server

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the *RemoteHost* property to specify the remote host server by either its host name or IP address.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can use the *RemotePort* property to specify the server port number directly or indirectly by naming the target service.

### Forming the connection

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the *Open* method. If you want your application to form the connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

### Getting information about the connection

After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the *Handle* property to obtain a handle to the socket connection to use when making socket calls.

## Closing the connection

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the *Close* method. The connection may also be closed from the server end. If that is the case, you will receive notification in an *OnDisconnect* event.

## Using server sockets

---

Add a server socket component (*TcpServer* or *UdpSocket*) to your form or data module to turn your application into an IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

Each server socket component uses a single server socket object to represent the server endpoint in a listening connection. It also uses a server client socket object for the server endpoint of each active connection to a client socket that the server accepts.

## Specifying the port

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the *LocalPort* property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.

## Listening for client requests

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the *Open* method. If you want your application to form the listening connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

## Connecting to clients

A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an *OnAccept* event.

## Closing server connections

When you want to shut down the listening connection, call the *Close* method or set the *Active* property to *False*. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When TCP clients shut down their individual connections to your server socket, you are informed by an *OnDisconnect* event.

## Responding to socket events

---

When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf*, *SendStream*, or *SendLn* methods in your program after the socket has been opened. You can read from the socket using the similarly-named methods *ReceiveBuf* and *ReceiveLn*. The *OnSend* and *OnReceive* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive two events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, you must use the *SendBuf* and *ReceiveBuf* methods to respond to these client events or server events.

### Error events

---

Client and server sockets generate *OnError* events when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about

- What socket object received the error notification.
- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

### Client events

---

When a client socket opens a connection, the following events occur:

- The socket is set up and initialized for event notification.
- An *OnCreateHandle* event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection.
- The connection request is accepted by the server and completed by the client socket.
- When the connection is established, the *OnConnect* notification event occurs.

## Server events

---

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

### Events when listening

Just before the listening connection is formed, the *OnListening* event occurs. You can use its *Handle* property to make changes to the socket before it is opened for listening. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListening* event handler.

### Events with client connections

When a server socket accepts a client connection request, the following events occur:

- An *OnAccept* event occurs, passing in the new *TTcpClient* object to the event handler. This is the first point when you can use the properties of *TTcpClient* to obtain information about the server endpoint of the connection to a client.
- If *BlockMode* is *bmThreadBlocking* an *OnGetThread* event occurs. If you want to provide your own customized descendant of *TServerSocketThread*, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerSocketThread*. If you want to perform any initialization of the thread, or make any socket API calls before the thread starts reading or writing over the connection, you should use the *OnGetThread* event handler for these tasks as well.
- The client completes the connection and an *OnAccept* event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

## Reading and writing over socket connections

---

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection. You can also form blocking connections, where your application waits for the reading or writing to be completed before executing the next line of code.

### Non-blocking connections

---

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in your network application. To create a non-blocking connection for client or server sockets, set the *BlockMode* property to *bmNonBlocking*.

When the connection is non-blocking, reading and writing events inform your socket when the socket on the other end of the connection tries to read or write information.

## Reading and writing events

Non-blocking sockets generate reading and writing events when it needs to read or write over the connection. You can respond to these notifications in an *OnReceive* or *OnSend* event handler.

The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the *ReceiveBuf* or *ReceiveLn* method. To write to the socket connection, use the *SendBuf*, *SendStream*, or *SendLn* method.

## Blocking connections

---

When the connection is blocking your socket must initiate reading or writing over the connection rather than waiting passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client or server sockets, set the *BlockMode* property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *BlockMode* property to *bmBlocking* or *bmThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *BlockMode* is *bmThreadBlocking*. When the *BlockMode* is *bmBlocking*, program execution is blocked until a new connection is established.

## Developing COM-based applications

The chapters in “Developing COM-based applications” present concepts necessary for building COM-based applications, including Automation controllers, Automation servers, ActiveX controls, and COM+ applications.

**Note** Support for COM clients is available in all editions of Delphi. However, to create servers, you need the Professional or Enterprise edition.



# Overview of COM technologies

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

**Note** COM components such as those on the ActiveX, COM+, and Servers tabs of the Component palette are not available for use in CLX applications. This technology is for use on Windows only and is not cross-platform.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM). For more information on clients, servers, and interfaces see, “Parts of a COM application,” on page 33-3.

This chapter provides a conceptual overview of the underlying technology on which Automation and ActiveX controls are built. Later chapters provide details on creating Automation objects and ActiveX controls in Delphi.

## COM as a specification and implementation

COM is both a specification and an implementation. The COM specification defines how objects are created and how they communicate with each other. According to this specification, COM objects can be written in different languages, run in different process spaces and on different platforms. As long as the objects adhere to the



written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is built into the Win32 subsystem, which provides a number of core services that support the written specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions designed for the purpose of creating and managing COM objects.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly (such as Active Documents). You can find these wrappers defined in the `ComObj` unit and the API definitions in the `AxCtrls` unit.

**Note** Delphi's interfaces and language follow the COM specification. Delphi implements objects conforming to the COM spec using a set of classes called the Delphi ActiveX framework (DAX). These classes are found in the `AxCtrls`, `OleCtrls`, and `OleServer` units. In addition, the Pascal interface to the COM API is in `ActiveX.pas` and `ComSvcs.pas`.

## COM extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories. For details on COM extensions, see "COM extensions" on page 33-10.

In addition, when working in a large, distributed environment, you can create transactional COM objects. Prior to Windows 2000, these objects were not architecturally part of COM, but rather ran in the Microsoft Transaction Server (MTS) environment. With the advent of Windows 2000, this support is integrated into COM+. Transactional objects are described in detail in Chapter 39, "Creating MTS or COM+ objects."

Delphi provides wizards to easily implement applications that incorporate the above technologies in the Delphi environment. For details, see "Implementing COM objects with wizards" on page 33-18.

## Parts of a COM application

---

When implementing a COM application, you supply the following:

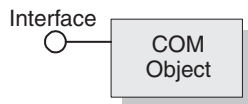
- COM Interface** The way in which an object exposes its services externally to clients. A COM object provides an interface for each set of related methods and properties. Note that COM properties are not identical to properties on VCL objects. COM properties always use read and write access methods.
- COM server** A module, either an EXE, DLL, or OCX, that contains the code for a COM object. Object implementations reside in servers. A COM object implements one or more interfaces.
- COM client** The code that calls the interfaces to get the requested services from the server. Clients know what they want to get from the server (through the interface); clients do not know the internals of how the server provides the services. Delphi eases the process in creating a client by letting you install COM servers (such as a Word document or PowerPoint slide) as components on the Component Palette. This allows you to connect to the server and hook its events through the Object Inspector.

### COM interfaces

---

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients. The standard way to depict a COM interface is shown in Figure 33.1:

**Figure 33.1** A COM interface



For example, every COM object implements the basic interface, *IUnknown*, which tells the client what interfaces are available on the COM object.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to convey to the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces are immutable; that is, they do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.
- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as *IMalloc* or *IPersist*.

- Interfaces are guaranteed to have a unique identification, called a **Globally Unique Identifier (GUID)**, which is a 128-bit randomly generated number. Interface GUIDs are called **Interface Identifiers (IIDs)**. This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer either explicitly or implicitly.
- Interfaces are not objects themselves; they provide a way to access an object. Therefore, clients do not access data directly; clients access data through an interface pointer. Windows 2000 adds an additional layer of indirection known as an interceptor through which it provides COM+ features such as just-in-time activation and object pooling.
- Interfaces are always inherited from the fundamental interface, *IUnknown*.
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection. For more information see , “In-process, out-of-process, and remote servers,” on page 33-6.

## The fundamental COM interface, *IUnknown*

All COM objects must support the fundamental interface, called *IUnknown*, a **typedef** to the base interface type *IInterface*. *IUnknown* contains the following routines:

QueryInterface	Provides pointers to other interfaces that the object supports.
AddRef and Release	Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

Clients obtain pointers to other interfaces through the *IUnknown* method, *QueryInterface*. *QueryInterface* knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any method of the interface.

Objects track their own lifetime through the *IUnknown* methods, *AddRef* and *Release*, which are simple reference counting methods. As long as an object's reference count is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object(s).

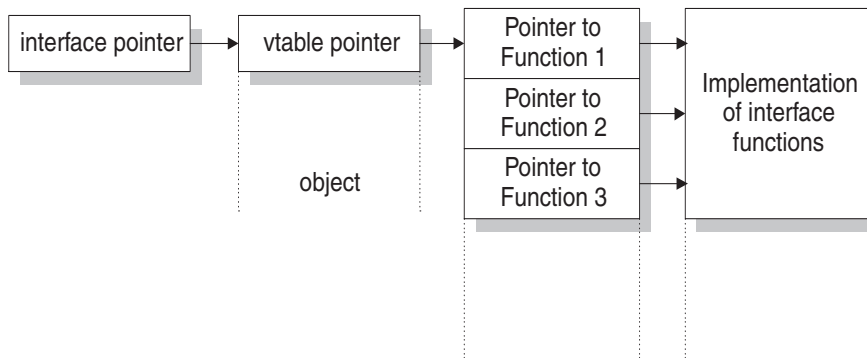
## COM interface pointers

An interface pointer is a 32-bit pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a **vtable**. Vtables are similar to the mechanism used to support virtual functions in Object Pascal. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Object Pascal classes.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's

interface pointer, then, is a pointer to the pointer to the vtable, as shown in the following diagram.

**Figure 33.2** Interface vtable



In Windows 2000 and subsequent versions of Windows, when an object is running under COM+, an added level of indirection is provided between the interface pointer and the vtable pointer. The interface pointer available to the client points at an interceptor, which in turn points at the vtable. This allows COM+ to provide such services as just-in-time activation, whereby the server can be deactivated and reactivated dynamically in a way that is opaque to the client. To achieve this, COM+ guarantees that the interceptor behaves as if it were an ordinary vtable pointer.

## COM servers

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods.

Clients do not know *how* a COM object performs its service; the object's implementation remains encapsulated. An object makes its services available through its interfaces as described previously.

In addition, clients do not need to know *where* a COM object resides. COM provides transparent access regardless of the object's location.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server instantiate an object instance for the client. This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

As a minimum, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).

- Implement a class factory object, which manufactures another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

**Note** Delphi wizards automate the creation of COM objects and servers as described in “Implementing COM objects with wizards” on page 33-18.

## CoClasses and class factories

A COM object is an instance of a **CoClass**, which is a class that implements one or more COM interfaces. The COM object provides the services as defined by its interfaces.

CoClasses are instantiated by a special type of object called a *class factory*. Whenever an object’s services are requested by a client, a class factory creates an object instance for that particular client. Typically, if another client requests the object’s services, the class factory creates another object instance to service the second client. (Clients can also bind to running COM objects that register themselves to support it.)

A CoClass must have a class factory and a class identifier (CLSID) so that it can be instantiated externally, that is, from another module. Using these unique identifiers for CoClasses means that they can be updated whenever new interfaces are implemented in their class. A new interface can modify or add methods without affecting older versions, which is a common problem when using DLLs.

Delphi wizards take care of assigning class identifiers and of implementing and instantiating class factories.

## In-process, out-of-process, and remote servers

With COM, a client does not need to know where an object resides, it simply makes a call to an object’s interface. COM performs the necessary steps to make the call. These steps differ depending on whether the object resides in the same process as the client, in a different process on the client machine, or in a different machine across the network. The different types of servers are known as:

**In-process server** A library (DLL) running in the *same process space* as the client, for example, an ActiveX control embedded in a Web page viewed under Internet Explorer or Netscape. Here, the ActiveX control is downloaded to the client machine and invoked within the same process as the Web browser.

The client communicates with the in-process server using direct calls to the COM interface.

**Out-of-process server (or local server)**

Another application (EXE) running in a *different process space* but on the *same machine* as the client. For example, an Excel spreadsheet embedded in a Word document are two separate applications running on the same machine.

The local server uses COM to communicate with the client.

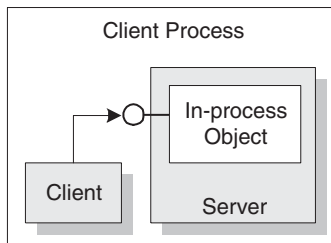
**Remote server**

A DLL or another application running on a *different machine* from that of the client. For example, a Delphi database application is connected to an application server on another machine in the network.

The remote server uses distributed COM (DCOM) to access interfaces and communicate with the application server.

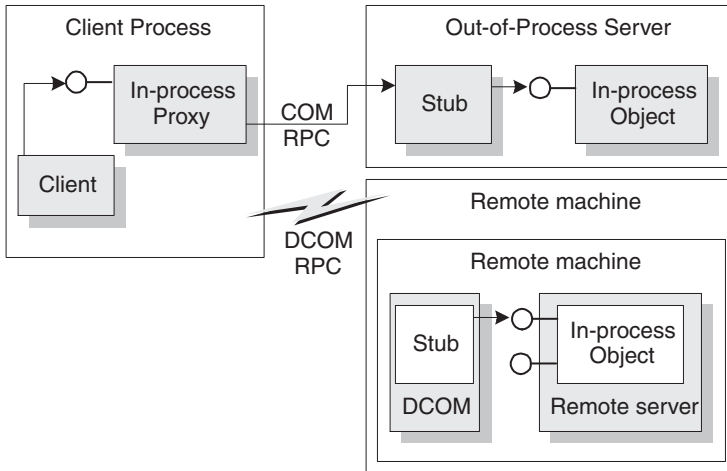
As shown in Figure 33.3, for in-process servers, pointers to the object interfaces are in the same process space as the client, so COM makes direct calls into the object implementation.

**Figure 33.3** In-process server



**Note** This is not always true under COM+. When a client makes a call to an object in a different context, COM+ intercepts the call so that it behaves like a call to an out-of-process server (see below), even if the server is in-process. See Chapter 39, “Creating MTS or COM+ objects” for more information working with COM+.

As shown in Figure 33.4, when the process is either in a different process or in a different machine altogether, COM uses a proxy to initiate remote procedure calls. The **proxy** resides in the same process as the client, so from the client’s perspective, all interface calls look alike. The proxy intercepts the client’s call and forwards it to where the real object is running. The mechanism that enables the client to access objects in a different process space, or even different machine, as if they were in their own process, is called **marshaling**.

**Figure 33.4** Out-of-process and remote servers

The difference between out-of-process and remote servers is the type of interprocess communication used. The proxy uses COM to communicate with an out-of-process server, it uses distributed COM (DCOM) to communicate with a remote machine. DCOM transparently transfers a local object request to the remote object running on a different machine.

**Note** For remote procedure calls, DCOM uses the RPC protocol provided by Open Group's Distributed Computing Environment (DCE). For distributed security, DCOM uses the NT LAN Manager (NTLM) security protocol. For directory services, DCOM uses the Domain Name System (DNS).

## The marshaling mechanism

Marshaling is the mechanism that allows a client to make interface function calls to remote objects in another process or on a different machine. Marshaling

- Takes an interface pointer in the server's process and makes a proxy pointer available to code in the client process.
- Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space.

For any interface call, the client pushes arguments onto a stack and makes a function call through the interface pointer. If the call to the object is not in-process, the call gets passed to the proxy. The proxy packs the arguments into a marshaling packet and transmits the structure to the remote object. The object's stub unpacks the packet, pushes the arguments onto the stack, and calls the object's implementation. In essence, the object recreates the client's call in its own address space.

What type of marshaling occurs depends on what the COM object implements. Objects can use a standard marshaling mechanism provided by the *IDispatch* interface. This is a generic marshaling mechanism that enables communication through a system-standard remote procedure call (RPC). For details on the *IDispatch* interface, see "Automation interfaces" on page 36-12. Even if the object does not

implement *IDispatch*, if it limits itself to automation-compatible types and has a registered type library, COM automatically provides marshaling support.

Applications that do not limit themselves to automation-compatible types or register a type library must provide their own marshaling. Marshaling is provided either through an implementation of the *IMarshal* interface, or by using a separately generated proxy/stub DLL. Delphi does not support the automatic generation of proxy/stub DLLs.

## Aggregation

Sometimes, a server object makes use of another COM object to perform some of its functions. For example, an inventory management object might make use of a separate invoicing object to handle customer invoices. If the inventory management object wants to present the invoice interface to clients, however, there is a problem: Although a client that has the inventory interface can call *QueryInterface* to obtain the invoice interface, when the invoice object was created it did not know about the inventory management object and can't return an invoice interface in response to a call to *QueryInterface*. A client that has the invoice interface can't get back to the inventory interface.

To avoid this problem, some COM objects support **aggregation**. When the inventory management object creates an instance of the invoice object, it passes it a copy of its own *IUnknown* interface. The invoice object can then use that *IUnknown* interface to handle any *QueryInterface* calls that request an interface, such as the inventory interface, that it does not support. When this happens, the two objects together are called an aggregate. The invoice object is called the inner, or contained object of the aggregate, and the inventory object is called the outer object.

**Note** In order to act as the outer object of an aggregate, a COM object must create the inner object using the Windows API *CoCreateInstance* or *CoCreateInstanceEx*, passing its *IUnknown* pointer as a parameter that the inner object can use for *QueryInterface* calls.

In order to create an object that can act as the inner object of an aggregate, it must descend from *TContainedObject*. When the object is created, the *IUnknown* interface of the outer object is passed to the constructor so that it can be used by the *QueryInterface* method on calls that the inner object can't handle.

## COM clients

---

Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the *IDispatch* interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations about the client using its objects. Similarly, clients don't need to know how (or even where) an object provides the services; they simply rely on server objects to provide the services they advertise through their interfaces.

There are two types of COM clients, controllers and containers. Controllers launch the server and interact with it through its interface. They request services from the COM object or drive it as a separate process. Containers host visual controls or



objects that appear in the container's user interface. They use predefined interfaces to negotiate display issues with server objects. It is impossible to have a container relationship over DCOM; for example, visual controls that appear in the container's user interface must be located locally. This is because the controls are expected to paint themselves, which requires that they have access to local GDI resources.

Delphi makes it easier for you to develop COM clients by letting you import a type library or ActiveX control into a component wrapper so that server objects look like other VCL components. For details on this process, see Chapter 35, "Creating COM clients".

## COM extensions

---

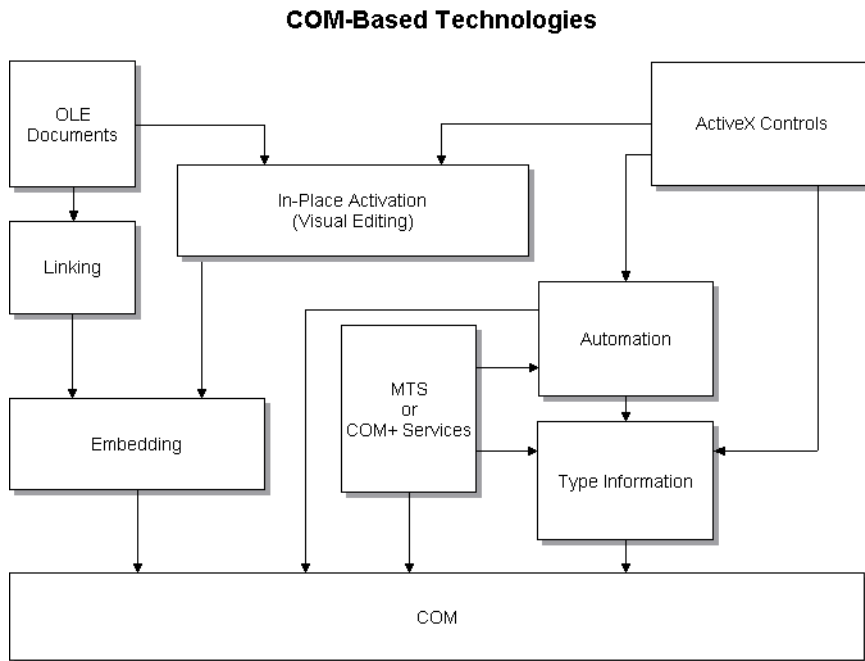
COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes.

The following lists some of the services COM extensions currently provide. Subsequent sections describe these services in greater detail.

- Automation servers** Automation refers to the ability of an application to control the objects in another application programmatically. Automation servers are the objects that can be controlled by other executables at runtime.
- ActiveX controls** ActiveX controls are specialized in-process servers, typically intended for embedding in a client application. The controls offer both design and runtime behaviors as well as events.
- Active Server Pages** Active Server Pages are scripts that generate HTML pages. The scripting language includes constructs for creating and running Automation objects. That is, the Active Server Page acts as an Automation controller.
- Active Documents** Objects that support linking and embedding, drag-and-drop, visual editing, and in-place activation. Word documents and Excel spreadsheets are examples of Active Documents.
- Transactional objects** Objects that include additional support for responding to large numbers of clients. This includes features such as just-in-time activation, transactions, resource pooling, and security services. These features were originally handled by MTS but have been built into COM with the advent of COM+.
- Type libraries** A collection of static data structures, often saved as a resource, that provides detailed type information about an object and its interfaces. Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available.

The following diagram illustrates the relationship of the COM extensions and how they are built upon COM:

**Figure 33.5** COM-based technologies



COM objects can be visual or non-visual. Some must run in the same process space as their clients; others can run in different processes or remote machines, as long as the objects provide marshaling support. Table 33.1 summarizes the types of COM objects that you can create, whether they are visual, process spaces they can run in, how they provide marshaling, and whether they require a type library.

**Table 33.1** COM object requirements

Object	Visual Object?	Process space	Communication	Type library
Active Document	Usually	In-process, or out-of-process	OLE Verbs	No
Automation	Occasionally	In-process, out-of-process, or remote	Automatically marshaled using the <i>IDispatch</i> interface (for out-of process and remote servers)	Required for automatic marshaling
ActiveX Control	Usually	In-process	Automatically marshaled using the <i>IDispatch</i> interface	Required
MTS or COM+	Occasionally	In-process for MTS, any for COM+	Automatically marshaled via a type library	Required

**Table 33.1** COM object requirements (continued)

Object	Visual Object?	Process space	Communication	Type library
In-process custom interface object	Optionally	In-process	No marshaling required for in-process servers	Recommended
Other custom interface object	Optionally	In-process, out-of-process, or remote	Automatically marshaled via a type library; otherwise, manually marshaled using custom interfaces	Recommended

## Automation servers

---

Automation refers to the ability of an application to control the objects in another application programmatically, like a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller.

Automation can be used on in-process, local, and remote servers.

Automation is characterized by two key points:

- The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide information about its interfaces, the interface methods, and those methods' arguments. Typically, this information is available in a type library. The Automation server can also generate type information dynamically when queried via its *IDispatch* interface (see following).
- Automation objects make their methods accessible so that other applications can use them. For this, they implement the *IDispatch* interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space because the Automation *IDispatch* interface automates the marshaling process. Automation does, however, restrict the types that you can use.

For a list of types that are valid for type libraries in general, and Automation interfaces in particular, see "Valid types" on page 34-11.

For information on writing an Automation server, see Chapter 36, "Creating simple COM servers."

## Active Server Pages

---

The Active Server Page (ASP) technology lets you write simple scripts, called Active Server Pages, that can be launched by clients via a Web server. Unlike ActiveX controls, which run on the client, Active Server Pages run on the server, and return a resulting HTML page to clients.

Active Server Pages are written in Jscript or VB script. The script runs every time the server loads the Web page. That script can then launch an embedded Automation server (or Enterprise Java Bean). For example, you can write an Automation server, such as one to create a bitmap or connect to a database, and this server accesses data that gets updated every time a client loads the Web page.

Active Server Pages rely on the Microsoft Internet Information Server (IIS) environment to serve your Web pages.

Delphi wizards let you create an Active Server Object, which is an Automation object specifically designed to work with an Active Server Page. For more information about creating and using these types of objects, see Chapter 37, "Creating an Active Server Page."

## ActiveX controls

---

ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications that need to be downloaded by a client before they are used.

ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as prefabricated OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display issues with their host containers.

ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

One use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX is a standard that targets interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

Delphi wizards allow you to easily create ActiveX controls. For more information about creating and using these types of objects, see Chapter 38, "Creating an ActiveX control."

## Active Documents

---

Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, *TOleContainer*, to support linking and embedding of existing Active Documents.

You can also use *TOleContainer* as a basis for an Active Document container. To create objects for Active Document servers, use the COM object wizard and add the appropriate interfaces, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

**Note** While the specification for Active Documents has built-in support for marshaling in cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine such as window handles, menu handles, and so on.

## Transactional objects

---

Delphi uses the term “transactional objects” to refer to objects that take advantage of the transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) (for versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later). These objects are designed to work in a large, distributed environment.

The transaction services provide robustness so that activities are always completed or rolled back (the server never partially completes an activity). The security services allow you to expose different levels of support to different classes of clients. The resource management allows an object to handle more clients by pooling resources or keeping objects active only when they are in use. To enable the system to provide these services, the object must implement the *IObjectControl* interface. To access the services, transactional objects use an interface called *IObjectContext*, which is created on their behalf by MTS or COM+.

Under MTS, the server object must be built into a library (DLL), which is then installed in the MTS runtime environment. That is, the server object is an in-process server that runs in the MTS runtime process space. Under COM+, this restriction does not apply because all COM calls are routed through an interceptor. To clients, the difference between MTS and COM+ is transparent.

MTS or COM+ servers group transactional objects that run in the same process space. Under MTS, this group is called an MTS package, while under COM+ it is called a COM+ application. A single machine can be running several different MTS packages (or COM+ applications), where each one is running in a separate process space.

To clients, the transactional object may appear like any other COM server object. The client need never know about transactions, security, or just-in-time activation unless it is initiating a transaction itself.

Both MTS and COM+ provide a separate tool for administering transactional objects. This tool lets you configure objects into packages or COM+ applications, view the packages or COM+ applications installed on a computer, view or change the attributes of the included objects, monitor and manage transactions, make objects available to clients, and so on. Under MTS, this tool is the MTS Explorer. Under COM+ it is the COM+ Component Manager.

## Type libraries

---

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available. All of Delphi's wizards generate a type library automatically, although the COM object wizard makes this optional. You can view or edit this type information by using the Type Library Editor as described in Chapter 34, "Working with type libraries."

This section describes what a type library contains, how it is created, when it is used, and how it is accessed. For developers wanting to share interfaces across languages, the section ends with suggestions on using type library tools.

### The content of type libraries

Type libraries contain *type information*, which indicates which interfaces exist in which COM objects, and the types and numbers of arguments to the interface methods. These descriptions include the unique identifiers for the CoClasses (CLSIDs) and the interfaces (IIDs), so that they can be properly accessed, as well as the dispatch identifiers (dispIDs) for Automation interface methods and properties.

Type libraries can also contain the following information:

- Descriptions of custom type information associated with custom interfaces
- Routines that are exported by the Automation or ActiveX server, but that are not interface methods
- Information about enumeration, record (structures), unions, alias, and module data types
- References to type descriptions from other type libraries

## Creating type libraries

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then running that script through a compiler. However, Delphi automatically generates a type library when you create a COM object (including ActiveX controls, Automation objects, remote data modules, and so on) using any of the wizards on the ActiveX or Multitier page of the new items dialog. (You can opt not to create a type library when using the COM object wizard.) You can also create a type library by choosing from the main menu, File | New | Other, select the ActiveX tab, and choose Type Library.

You can view the type library using Delphi's Type Library editor. You can easily edit your type library using the Type Library editor and Delphi automatically updates the corresponding .tlb file (binary type library file) when the type library is saved. For any changes to Interfaces and CoClasses that were created using a wizard, the Type Library editor also updates your implementation files. For more information on using the Type Library editor to write interfaces and CoClasses, see Chapter 34, "Working with type libraries."

## When to use type libraries

It is important to create a type library for each set of objects that is exposed to external users, for example,

- ActiveX controls require a type library, which must be included as a resource in the DLL that contains the ActiveX controls.
- Exposed objects that support vtable binding of custom interfaces must be described in a type library because vtable references are bound at compile time. Clients import information about the interfaces from the type library and use that information to compile. For more information about vtable and compile time binding, see "Automation interfaces" on page 36-12.
- Applications that implement Automation servers should provide a type library so that clients can early bind to it.
- Objects instantiated from classes that support the *IProvideClassInfo* interface, such as all descendants of the VCL *TTypedComObject* class, must have a type library.
- Type libraries are not required, but are useful for identifying the objects used with OLE drag-and-drop.

When defining interfaces for internal use only (within an application) you do not need to create a type library.

## Accessing type libraries

The binary type library is normally a part of a resource file (.res) or a stand-alone file with a .tlb file-name extension. When included in a resource file, the type library can be bound into a server (.dll, .ocx, or .exe).

Once a type library has been created, object browsers, compilers, and similar tools can access type libraries through special type interfaces:

Interface	Description
<i>ITypeLib</i>	Provides methods for accessing a library of type descriptions.
<i>ITypeLib2</i>	Augments <i>ITypeLib</i> to include support for documentation strings, custom data, and statistics about the type library.
<i>ITypeInfo</i>	Provides descriptions of individual objects contained in a type library. For example, a browser uses this interface to extract information about objects from the type library.
<i>ITypeInfo2</i>	Augments <i>ITypeInfo</i> to access additional type library information, including methods for accessing custom data elements.
<i>ITypeComp</i>	Provides a fast way to access information that compilers need when binding to an interface.

Delphi can import and use type libraries from other applications by choosing Project | Import Type Library. Most of the VCL classes used for COM applications support the essential interfaces that are used to store and retrieve type information from type libraries and from running instances of an object. The VCL class *TTypedComObject* supports interfaces that provide type information, and is used as a foundation for the ActiveX object framework.

## Benefits of using type libraries

Even if your application does not require a type library, you can consider the following benefits of using one:

- Type checking can be performed at compile time.
- You can use early binding with Automation, and controllers that do not support vtables or dual interfaces can encode dispIDs at compile time, improving runtime performance.
- Type browsers can scan the library, so clients can see the characteristics of your objects.
- The *RegisterTypeLib* function can be used to register your exposed objects in the registration database.
- The *UnRegisterTypeLib* function can be used to completely uninstall an application's type library from the system registry.
- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.



## Using type library tools

The tools for working with type libraries are listed below.

- The TLIBIMP (Type Library Import) tool, which takes existing type libraries and creates Delphi Interface files (`_TLB.pas` files), is incorporated into the Type Library editor. TLIBIMP provides additional configuration options not available inside the Type Library editor.
- TRegSvr is a tool for registering and unregistering servers and type libraries, which comes with Delphi. The source to TRegSvr is available as an example in the Demos directory.
- The Microsoft IDL compiler (MIDL) compiles IDL scripts to create a type library.
- RegSvr32.exe is a tool for registering and unregistering servers and type libraries, which is a standard Windows utility.
- OLEView is a type library browser tool, found on Microsoft's Web site.

## Implementing COM objects with wizards

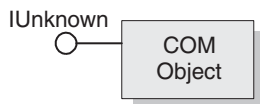
---

Delphi makes it easier to write COM server applications by providing wizards that handle many of the details involved. Delphi provides separate wizards to create the following:

- A simple COM object
- An Automation object
- An Active Server Object (for embedding in an Active Server page)
- An ActiveX control
- An ActiveX Form
- A transactional object
- A Property page
- A Type library
- An ActiveX library

The wizards handle many of the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. As shown in Figure 33.6, with a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.

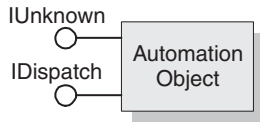
**Figure 33.6** Simple COM object interface



The COM object wizard also provides an implementation for *IDispatch* if you specify that you are creating an object that supports an *IDispatch* descendant.

As shown in Figure 33.7, for Automation and Active Server objects, the wizard implements *IUnknown* and *IDispatch*, which provides automatic marshaling.

**Figure 33.7** Automation object interface



As shown in Figure 33.8, for ActiveX control objects and ActiveX forms, the wizard implements all the required ActiveX control interfaces, from *IUnknown*, *IDispatch*, *IObject*, *IObjectControl*, and so on. For a complete list of interfaces, see the reference page for *TActiveXObject* object.

**Figure 33.8** ActiveX object interface

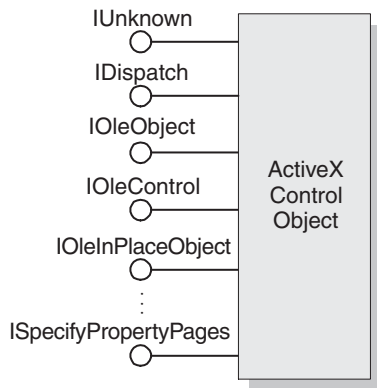


Table 33.2 lists the various wizards and the interfaces they implement:

**Table 33.2** Delphi wizards for implementing COM, Automation, and ActiveX objects

Wizard	Implemented interfaces	What the wizard does
COM server	<i>IUnknown</i> (and <i>IDispatch</i> if you select a default interface that descends from <i>IDispatch</i> )	Exports routines that handle server registration, class registration, loading and unloading the server, and object instantiation. Creates and manages class factories for objects implemented on the server. Provides registry entries for the object that specify the selected threading model. Declares the methods that implement a selected interface, providing skeletal implementations for you to complete. Provides a type library, if requested. Allows you to select an arbitrary interface that is registered in the type library and implement it. If you do this, you must use a type library.

**Table 33.2** Delphi wizards for implementing COM, Automation, and ActiveX objects (continued)

Wizard	Implemented interfaces	What the wizard does
Automation server	<i>IUnknown, IDispatch</i>	<p>Performs the tasks of a COM server wizard (described above), plus:</p> <p>Implements the interface that you specify, either dual or dispatch. Provides server-side support for generating events, if requested.</p> <p>Provides a type library automatically.</p>
Active Server Object	<i>IUnknown, IDispatch, IASPObject</i>	<p>Performs the tasks of an Automation object wizard (described above) and optionally generates an .ASP page which can be loaded into a Web browser. It leaves you in the Type Library editor so that you can modify the object's properties and methods if needed.</p> <p>Surfaces the ASP intrinsics as properties so that you can easily obtain information about the ASP application and the HTTP messages that launched it.</p>
ActiveX Control	<i>IUnknown, IDispatch, IPersistStreamInit, IOleInPlaceActiveObject, IPersistStorage, IViewObject, IOleObject, IViewObject2, IOleControl, IPerPropertyBrowsing, IOleInPlaceObject, ISpecifyPropertyPages</i>	<p>Performs the tasks of the Automation server wizard (described above), plus:</p> <p>Generates a CoClass that corresponds to the VCL control on which the ActiveX control is based and which implements all the ActiveX interfaces.</p> <p>Leaves you in the source code editor so that you can modify the implementation class.</p>
ActiveForm	Same interfaces as ActiveX Control	<p>Performs the tasks of the ActiveX control wizard, plus:</p> <p>Creates a <i>TActiveForm</i> descendant that takes the place of the pre-existing VCL class in the ActiveX control wizard. This new class lets you design the Active Form the same way you design a form in a Windows application.</p>
Transactional object	<i>IUnknown, IDispatch, IObjectControl</i>	<p>Adds a new unit to the current project containing the MTS or COM+ object definition. It inserts proprietary GUIDs into the type library so that Delphi can install the object properly, and leaves you in the Type Library editor so that you can define the interface that the object exposes to clients. You must install the object separately after it is built.</p>
Property Page	<i>IUnknown, IPropertyPage</i>	<p>Creates a new property page that you can design in the Forms designer.</p>
COM+ Event object	None, by default	<p>Creates a COM+ event object that you can define using the Type Library editor. Unlike the other object wizards, the COM+ Event object wizard does not create an implementation unit because event objects have no implementation (it is provided by client sinks).</p>

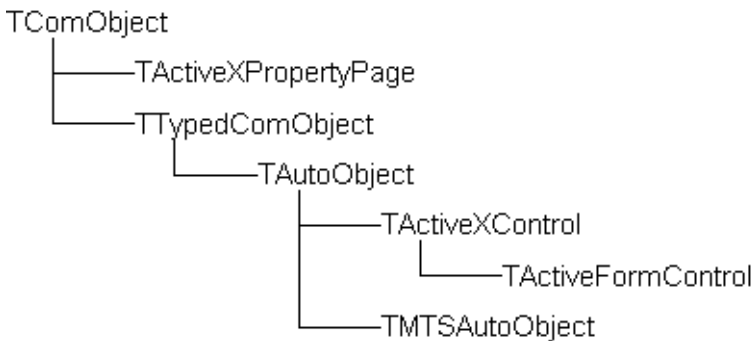
**Table 33.2** Delphi wizards for implementing COM, Automation, and ActiveX objects (continued)

Wizard	Implemented interfaces	What the wizard does
Type Library	None, by default	Creates a new type library and associates it with the active project.
ActiveX library	None, by default	Creates a new ActiveX or Com server DLL and exposes the necessary export functions.

You can add additional COM objects or reimplement an existing implementation. To add a new object, it is easiest to use the wizard a second time. This is because the wizard sets up an association between the type library and an implementation class, so that changes you make in the type library editor are automatically applied to your implementation object.

## Code generated by wizards

Delphi's wizards generate classes that are derived from the Delphi ActiveX framework (DAX). Despite its name, the Delphi ActiveX framework supports all types of COM objects, not just ActiveX controls. The classes in this framework provide the underlying implementation of the standard COM interfaces for the objects you create using a wizard. Figure 33.9 illustrates the objects in the Delphi ActiveX framework:

**Figure 33.9** Delphi ActiveX framework

Each wizard generates an implementation unit that implements your COM server object. The COM server object (the implementation object) descends from one of the classes in DAX:

**Table 33.3** DAX Base classes for generated implementation classes

Wizard	Base class from DAX	Inherited support
COM server	TTypedCOMObject	Support for <i>IUnknown</i> and <i>ISupportErrorInfo</i> interfaces. Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces. Support for reading type library information.

**Table 33.3** DAX Base classes for generated implementation classes (continued)

Wizard	Base class from DAX	Inherited support
Automation server Active Server Object	TAutoObject	Everything provided by <i>TTypedCOMObject</i> , plus: Support for the <i>IDispatch</i> interface. Auto-marshaling support.
ActiveX Control	TActiveXControl	Everything provided by <i>TAutoObject</i> , plus: Support for embedding in a container. Support for in-place activation. Support for properties and property pages. The ability to delegate to an associated windowed control that it creates.
ActiveForm	TActiveFormControl	Everything provided by <i>TAutoObject</i> , except that it works with a descendant of <i>TActiveForm</i> rather than another windowed control class.
MTS object	TMTSAutoObject	Everything provided by <i>TAutoObject</i> , plus: Support for the <i>IObjectControl</i> interface.
Property Page	TPropertyPage (uses TActiveXPropertyPage internally)	Support for <i>IUnknown</i> and <i>ISupportErrorInfo</i> interfaces. Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces. Support for the <i>IPropertyPage</i> interface.

Corresponding to the classes in Figure 33.9 is a hierarchy of class factory objects that handle the creation of these COM objects. The wizard adds code to the initialization section of your implementation unit that instantiates the appropriate class factory for your implementation class.

The wizards also generate a type library and its associated unit, which has a name of the form `Project1_TLB`. The `Project1_TLB` unit includes the definitions your application needs to use the type definitions and interfaces defined in the type library. For more information on the contents of this file, see “Code generated when you import type library information” on page 35-5.

You can modify the interface generated by the wizard using the type library editor. When you do this, the implementation class is automatically updated to reflect those changes. You need only fill in the bodies of the generated methods to complete the implementation.

## Working with type libraries

This chapter describes how to create and edit type libraries using Delphi's Type Library editor. Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by a COM object. They provide a way to identify what types of objects and interfaces are available on a server. For a detailed overview on why and when to use type libraries, see "Type libraries" on page 33-15.

A type library can contain any and all of the following:

- Information about custom data types such as aliases, enumerations, structures, and unions.
- Descriptions of one or more COM elements, such as an interface, dispinterface, or CoClass. Each of these descriptions is commonly referred to as *type information*.
- Descriptions of constants and methods defined in external units.
- References to type descriptions from other type libraries.

By including a type library with your COM application or ActiveX library, you make information about the objects in your application available to other applications and programming tools through COM's type library tools and interfaces.

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then run that script through a compiler. The Type Library editor automates some of this process, easing the burden of creating and modifying your own type libraries.

When you create a COM server of any type (ActiveX control, Automation object, remote data module, and so on) using Delphi's wizards, the wizard automatically generates a type library for you (although in the case of the COM object wizard, this is optional). Most of the work you do in customizing the generated object starts with the type library, because that is where you define the properties and methods it exposes to clients: you change the interface of the CoClass generated by the wizard, using the Type Library editor. The Type Library editor automatically updates the implementation unit for your object, so that all you need do is fill in the bodies of the generated methods.

You can also use the Delphi Type Library Editor in the development of Common Object Request Broker Architecture (CORBA) applications. With traditional CORBA tools, you must define object interfaces separately from your application, using the CORBA Interface Definition Language (IDL). You then run a utility that generates stub-and-skeleton code from that definition. However, Delphi generates the stub, skeleton, and IDL for you automatically. You can easily edit your interface using the Type Library editor and Delphi automatically updates the appropriate source files.

## Type Library editor

---

The Type Library editor enables developers to examine and create type information for COM objects. Using the Type Library editor can greatly simplify the task of developing COM objects by centralizing the tasks of defining interfaces, CoClasses, and types, obtaining GUIDs for new interfaces, associating interfaces with CoClasses, updating implementation units, and so on.

**Note** The Type Library editor is also used to define CORBA interfaces in projects that use the CORBA Object or CORBA Data Module wizard.

The Type Library editor outputs two types of file that represent the contents of the type library:

**Table 34.1** Type Library editor files

File	Description
.TLB file	<p>The binary type library file. By default, you do not need to use this file, because the type library is automatically compiled into the application as a resource. However, you can use this file to explicitly compile the type library into another project or to deploy the type library separately from the .exe or .ocx. For more information, see “Opening an existing type library” on page 34-19 and “Deploying type libraries” on page 34-27.</p> <p><b>Note:</b> When using the Type Library editor for CORBA interfaces, the Type Library editor does not create the .tlb file.</p>
_TLB unit	<p>This unit interprets the contents of the type library for use by your application. It contains all the declarations your application needs to use the elements defined in the type library. Although you can open this file in the code editor, you should never edit it—it is maintained by the Type Library editor, so any changes you make will be overwritten by the Type Library editor. For more details on the contents of this file, see “Code generated when you import type library information” on page 35-5.</p> <p><b>Note:</b> When using the Type Library editor for CORBA interfaces, this unit defines the stub and skeleton objects required by the CORBA application.</p>

## Parts of the Type Library editor

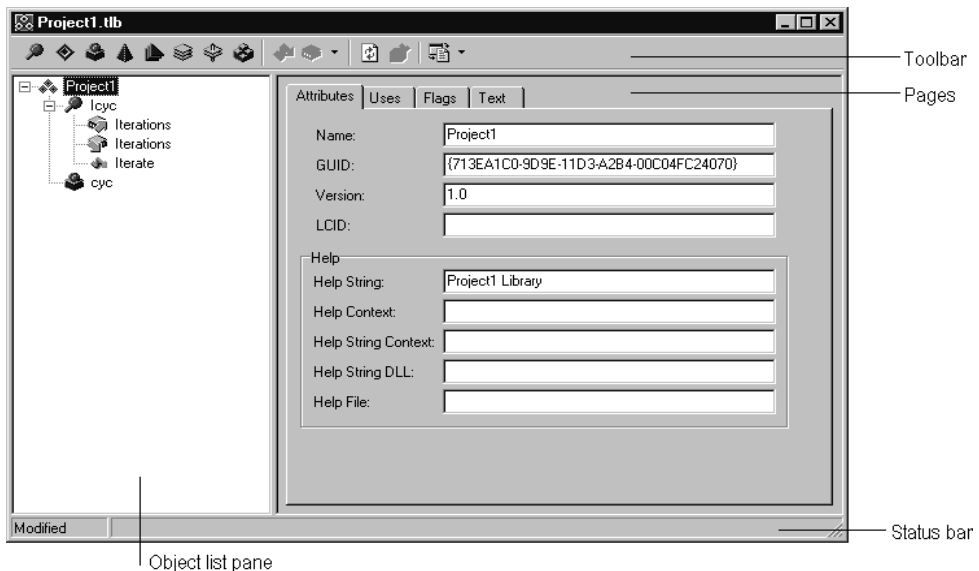
The main elements of the Type Library editor are described in Table 34.2:

**Table 34.2** Type Library editor parts

Part	Description
Toolbar	Includes buttons to add new types, CoClasses, interfaces, and interface members to your type library. The toolbar also includes buttons for refreshing your implementation unit, registering the type library, and saving an IDL file with the information in your type library.
Object list pane	Displays all the existing elements in the type library. When you click on an item in the object list pane, it displays pages valid for that object.
Status bar	Displays syntax errors if you try to add invalid types to your type library.
Pages	Display information about the selected object. Which pages appear here depends on the type of object selected.

These parts are illustrated in Figure 34.1, which shows the Type Library editor displaying type information for a COM object named `cyc`.

**Figure 34.1** Type Library editor











### Toolbar

The Type Library editor's toolbar located at the top of the Type Library Editor, contains buttons that you click to add new objects into your type library.

The first group of buttons let you add elements to the type library. When you click a toolbar button, the icon for that element appears in the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you select, different pages of information appear to the right.










The following table lists the elements you can add to your type library:

Icon	Meaning
	An interface description.
	A dispinterface description. (not used for CORBA interface definitions)
	A CoClass.
	An enumeration.
	An alias.
	A record.
	A union.
	A module.

When you select one of the elements listed above in the object list pane, the second group of buttons displays members that are valid for that element. For example, when you select Interface, the Method and Property icons in the second box become enabled because you can add methods and properties to your interface definition. When you select Enum, the second group of buttons changes to display the Const member, which is the only valid member for Enum type information.

The following table lists the members that can be added to elements in the object list pane:

Icon	Meaning
	A method of the interface, dispinterface, or an entry point in a module.
	A property on an interface or dispinterface.
	A write-only property. (available from the drop-down list on the property button)
	A read-write property. (available from the drop-down list on the property button)
	A read-only property. (available from the drop-down list on the property button)
	A field in a record or union.
	A constant in an enum or a module.

In the third box, you can choose to refresh, register, or export your type library (save it as an IDL file), as described in “Saving and registering type library information” on page 34-24.

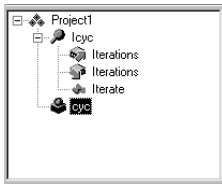
## Object list pane

The Object list pane displays all the elements of the current type library in a tree view. The root of the tree represents the type library itself, and appears as the following icon:



Descending from the type library node are the elements in the type library:

**Figure 34.2** Object list pane



When you select any of these elements (including the type library itself), the pages of type information to the right change to reflect only the relevant information for that element. You can use these pages to edit the definition and properties of the selected element.

You can manipulate the elements in the object list pane by right clicking to get the object list pane context menu. This menu includes commands that let you use the Windows clipboard to move or copy existing elements as well as commands to add new elements or customize the appearance of the Type Library editor.

## Status bar

When editing or saving a type library, syntax, translation errors, and warnings are listed in the Status bar pane.

For example, if you specify a type that the Type Library editor does not support, you will get a syntax error. For a complete list of types supported by the Type Library editor, see “Valid types” on page 34-11.

## Pages of type information

When you select an element in the object list pane, pages of type information appear in the Type Library editor that are valid for the selected element. Which pages appear depends on the element selected in the object list panel, as follows:

**Table 34.3** Type library pages

Type Info element	Page of type information	Contents of page
<b>Type library</b>	Attributes	Name, version, and GUID for the type library, as well as information linking the type library to help.
	Uses	List of other type libraries that contain definitions on which this one depends.
	Flags	Flags that determine how other applications can use the type library.
	Text	All definitions and declarations defining the type library itself (see discussion below).
<b>Interface</b>	Attributes	Name, version, and GUID for the interface, the name of the interface from which it descends, and information linking the interface to help.
	Flags	Flags that indicate whether the interface is hidden, dual, Automation-compatible, and/or extensible.
	Text	The definitions and declarations for the Interface (see discussion below).
<b>Dispinterface</b>	Attributes	Name, version, and GUID for the interface, and information linking it to help.
	Flags	Flags that indicate whether the Dispinterface is hidden, dual, and/or extensible.
	Text	The definitions and declarations for the Dispinterface. (see discussion below).
<b>CoClass</b>	Attributes	Name, version, and GUID for the CoClass, and information linking it to help.
	Implements	A List of interfaces that the CoClass implements, as well as their attributes.
	COM+	The attributes of transactional objects, such as the transaction model, call synchronization, just-in-time activation, object pooling, and so on. Also includes the attributes of COM+ event objects.
	Flags	Flags that indicate various attributes of the CoClass, including how clients can create and use instances, whether it is visible to users in a browser, whether it is an ActiveX control, and whether it can be aggregated (act as part of a composite).
	Text	The definitions and declarations for the CoClass (see discussion below).
<b>Enumeration</b>	Attributes	Name, version, and GUID for the enumeration, and information linking it to help.
	Text	The definitions and declarations for the enumerated type (see discussion below).
<b>Alias</b>	Attributes	Name, version, and GUID for the enumeration, the type the alias represents, and information linking it to help.

**Table 34.3** Type library pages (continued)

Type Info element	Page of type information	Contents of page
	Text	The definitions and declarations for the alias (see discussion below).
<b>Record</b>	Attributes	Name, version, and GUID for the record, and information linking it to help.
	Text	The definitions and declarations for the record (see discussion below).
<b>Union</b>	Attributes	Name, version, and GUID for the union, and information linking it to help.
	Text	The definitions and declarations for the union (see discussion below).
<b>Module</b>	Attributes	Name, version, GUID, and associated DLL for the module, and information linking it to help.
	Text	The definitions and declarations for the module (see discussion below).
<b>Method</b>	Attributes	Name, dispatch ID or DLL entry point, and information linking it to help.
	Parameters	Method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the method, whether this is a default method for the interface, and whether it is replaceable.
	Text	The definitions and declarations for the method (see discussion below).
<b>Property</b>	Attributes	Name, dispatch ID, type of property access method (getter vs. setter), and information linking it to help.
	Parameters	Property access method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the property, whether this is a default for the interface, whether the property is replaceable, bindable, and so on.
	Text	The definitions and declarations for the property access method (see discussion below).
<b>Const</b>	Attributes	Name, value, type (for module consts), and information linking it to help.
	Flags	Flags to indicate how clients can view and use the constant, whether this represents a default value, whether the constant is bindable, and so on.
	Text	The definitions and declarations for the constant (see discussion below).
<b>Field</b>	Attributes	Name, type, and information linking it to help.
	Flags	Flags to indicate how clients can view and use the field, whether this represents a default value, whether the field is bindable, and so on.
	Text	The definitions and declarations for the field (see discussion below).

**Note** For more detailed information about the various options you can set on type information pages, see the online Help for the Type Library editor.

You can use each of the pages of type information to view or edit the values it displays. Most of the pages organize the information into a set of controls so that you can type in values or select them from a list without requiring that you know the syntax of the corresponding declarations. This can prevent many small mistakes such as typographic errors when specifying values from a limited set. However, you may find it faster to type in the declarations directly. To do this, use the Text page.

All type library elements have a text page that displays the syntax for the element. This syntax appears in an IDL subset of Microsoft Interface Definition Language, or Object Pascal. Any changes you make in other pages of the element are reflected on the text page. If you add code directly in the text page, changes are reflected in the other pages of the Type Library editor.

The Type Library editor generates syntax errors if you add identifiers that are currently not supported by the editor; the editor currently supports only those identifiers that relate to type library support (not RPC support or constructs used by the Microsoft IDL compiler for C++ code generation or marshaling support).

## Type library elements

---

The Type Library interface can seem overwhelmingly complicated at first. This is because it represents information about a great number of elements, each of which has its own characteristics. However, many of these characteristics are common to all elements. For example, every element (including the type library itself) has the following:

- A Name, which is used to describe the element and which is used when referring to the element in code.
- A GUID (globally unique identifier), which is a globally unique 128-bit value that COM uses to identify the element. This should always be supplied for the type library itself and for CoClasses and interfaces. It is optional otherwise.
- A Version number, which distinguishes between multiple versions of the element. This is always optional, but should be provided for CoClasses and interfaces, because some tools can't use them without a version number.
- Information linking the element to a Help topic. These include a Help String, and Help Context or Help String Context value. The Help Context is used for a traditional Windows Help system where the type library has a stand-alone Help file. The Help String Context is used when help is supplied by a separate DLL instead. The Help Context or Help String Context refers to a Help file or DLL that is specified on the type library's Attributes page. This is always optional.

## Interfaces

An interface describes the methods (and any properties expressed as 'get' and 'set' functions) for an object that must be accessed through a virtual function table (VTable). If an interface is flagged as dual, a dispinterface is also implied and can be

accessed through OLE automation. By default, the type library flags all interfaces you add as dual.

Interfaces can be assigned members: methods and properties. These appear in the object list pane as children of the interface node. Properties for interfaces are represented by the 'get' and 'set' methods used to read and write the property's underlying data. They are represented in the tree view using special icons that indicate their purpose.

**Note** When a property is specified as Write By Reference, it means it is passed as a pointer rather than by value. Some applications, such as Visual Basic, use Write By Reference, if it is present, to optimize performance. To pass the property only by reference rather than by value, use the property type *By Reference Only*. To pass the property by reference as well as by value, select Read | Write | Write By Ref. To invoke this menu, go to the toolbar and select the arrow next to the property icon.

Once you add the properties or methods using the toolbar button or the object list pane context menu, you describe their syntax and attributes by selecting the property or method and using the pages of type information.

The Attributes page lets you give the property or method a name and dispatch ID (so that it can be called using IDispatch). For properties, you also assign a type. The function signature is created using the Parameters page, where you can add, remove, and rearrange parameters, set their type and any modifiers, and specify function return types.

**Note** Members of interfaces that need to raise exceptions should return an HRESULT and specify a return value parameter (PARAM\_RETVAL) for the actual return value. Declare these methods using the **safecall** calling convention.

Note that when you assign properties and methods to an interface, they are implicitly assigned to its associated CoClass. This is why the Type Library editor does not let you add properties and methods directly to a CoClass.

## Dispinterfaces

Interfaces are more commonly used than dispinterfaces to describe the properties and methods of an object. Dispinterfaces are only accessible through dynamic binding, while interfaces can have static binding through a vtable.

You can add methods and properties to dispinterfaces in the same way you add them to interfaces. However, when you create a property for a dispinterface, you can't specify a function kind or parameter types.

## CoClasses

A CoClass describes a unique COM object that implements one or more interfaces. When defining a CoClass, you must specify which implemented interface is the default for the object, and optionally, which dispinterface is the default source for events. Note that you do not add properties or methods to a CoClass in the Type Library editor. Properties and methods are exposed to clients by interfaces, which are associated with the CoClass using the Implements page.

## Type definitions

Enumerations, aliases, records, and unions all declare types that can then be used elsewhere in the type library.

Enums consist of a list of constants, each of which must be numeric. Numeric input is usually an integer in decimal or hexadecimal format. The base value is zero by default. You can add constants to your enumeration by selecting the enumeration in the object list pane and clicking the Const button on the toolbar or selecting New | Const command from the object list pane context menu.

**Note** It is strongly recommended that you provide help strings for your enumerations to make their meaning clearer. The following is a sample entry of an enumeration type for a mouse button and includes a help string for each enumeration element.

```
mbLeft = 0 [helpstring 'mbLeft'];
mbRight = 1 [helpstring 'mbRight'];
mbMiddle = 3 [helpstring 'mbMiddle'];
```

An alias creates an alias (type definition) for a type. You can use the alias to define types that you want to use in other type info such as records or unions. Associate the alias with the underlying type definition by setting the Type attribute on the Attributes page.

A record consists of a list of structure members or fields. A union is a record with only a variant part. Like a record, a union consists of a list of structure members or fields. However, unlike the members of records, each member of a union occupies the same physical address, so that only one logical value can be stored.

Add the fields to a record or union by selecting it in the object list pane and clicking the field button in the toolbar or right clicking and choosing field from the object list pane context menu. Each field has a name and a type, which you assign by selecting the field and assigning values using the Attributes page. Records and unions can be defined with an optional tag.

Members can be of any built-in type, or you can specify a type using alias before you define the record.

## Modules

A module defines a group of functions, typically a set of DLL entry points. You define a module by

- Specifying a DLL that it represents on the attributes page.
- Adding methods and constants using the toolbar or the object list pane context menu. For each method or constant, you must then define its attributes by selecting the it in the object list pane and setting the values on the Attributes page.

For module methods, you must assign a name and DLL entry-point using the attributes page. Declare the function's parameters and return type using the parameters page.

For module constants, use the Attributes page to specify a name, type, and value.

**Note** The Type Library editor does not generate any declarations or implementation related to a module. The specified DLL must be created as a separate project.

## Using the Type Library editor

---

Using the type library editor, you can create new type libraries or edit existing ones. Typically, an application developer uses a wizard to create the objects that are exposed in the type library, letting Delphi generate the type library automatically. Then, the automatically-generated type library is opened in the Type Library editor so that the interfaces can be defined (or modified), type definitions added, and so on.

However, even if you are not using a wizard to define the objects, you can use the Type Library editor to define a new type library. In this case, you must create any implementation classes yourself, because the Type Library editor does not generate code for CoClasses that were not associated with a type library by a wizard.

The editor supports a subset of valid types in a type library as described below.

The final topics in this section describe how to:

- Create a new type library
- Open an existing type library
- Add an interface to the type library
- Modify an interface
- Add properties and methods to the type library
- Add a CoClass to the type library
- Add an interface to a CoClass
- Add an enumeration to the type library
- Add an alias to the type library
- Add a record or union to the type library
- Add a module to the type library
- Save and register type library information

### Valid types

In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Object Pascal. Specify the language you want to use in the Environment options dialog.

The following types are valid in a type library for COM development. The Automation compatible column specifies whether the type can be used by an interface that has its Automation or Dispinterface flag checked.



These are the types that COM can marshal via the type library automatically.

**Table 34.4** Valid types

Pascal type	IDL type	variant type	Automation compatible	Description
Smallint	short	VT_I2	Yes	2-byte signed integer
Integer	long	VT_I4	Yes	4-byte signed integer
Single	single	VT_R4	Yes	4-byte real
Double	double	VT_R8	Yes	8-byte real
Currency	CURRENCY	VT_CY	Yes	currency
TDateTime	DATE	VT_DATE	Yes	date
WideString	BSTR	VT_BSTR	Yes	binary string
IDispatch	IDispatch	VT_DISPATCH	Yes	pointer to IDispatch interface
SCODE	SCODE	VT_ERROR	Yes	Ole Error Code
WordBool	VARIANT_BOOL	VT_BOOL	Yes	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	Yes	Ole Variant
IUnknown	IUnknown	VT_UNKNOWN	Yes	pointer to IUnknown interface
Shortint	byte	VT_I1	No	1 byte signed integer
Byte	unsigned char	VT_UI1	Yes	1 byte unsigned integer
Word	unsigned short	VT_UI2	Yes*	2 byte unsigned integer
LongWord	unsigned long	VT_UI4	Yes*	4 byte unsigned integer
Int64	__int64	VT_I8	No	8 byte signed integer
Largeuint	uint64	VT_UI8	No	8 byte unsigned integer
SYSINT	int	VT_INT	Yes*	system dependent integer (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	Yes*	system dependent unsigned integer
HResult	HRESULT	VT_HRESULT	No	32 bit error code
Pointer		VT_PTR -> VT_VOID	No	untyped pointer
SafeArray	SAFEARRAY	VT_SAFEARRAY	No	OLE Safe Array
PChar	LPSTR	VT_LPSTR	No	pointer to Char
PWideChar	LPWSTR	VT_LPWSTR	No	pointer to WideChar

\* Word, LongWord, SYSINT, and SYSUINT are Automation-compatible in most applications, but in older applications they may not be.

**Note** The Byte (VT\_UI1) is Automation-compatible, but is not allowed in a Variant or OleVariant since many Automation servers do not handle this value correctly.

Besides these IDL types, any interfaces and types defined in the library or defined in referenced libraries can be used in a type library definition.

The Type Library editor stores type information in the generated type library (.TLB) file in binary form.

If a parameter type is specified as a Pointer type, the Type Library editor usually translates that type into a variable parameter. When the type library is saved, the variable parameter's associated ElemDesc's IDL flags are marked IDL\_FIN or IDL\_FOUT.

Often, ElemDesc IDL flags are not marked by IDL\_FIN or IDL\_FOUT when the type is preceded with a Pointer. Or, in the case of dispinterfaces, IDL flags are not typically used. In these cases, you may see a comment next to the variable identifier such as {IDL\_None} or {IDL\_In}. These comments are used when saving a type library to correctly mark the IDL flags.

### SafeArrays

COM requires that arrays be passed via a special data type known as a *SafeArray*. You can create and destroy *SafeArrays* by calling special COM functions to do so, and all elements within a *SafeArray* must be valid automation-compatible types. The Delphi compiler has built-in knowledge of COM *SafeArrays* and automatically calls the COM API to create, copy, and destroy *SafeArrays*.

In the Type Library editor, a *SafeArray* must specify the type of its elements. For example, the following line from the text page declares a method with a parameter that is a *SafeArray* with an element type of Integer:

```
procedure HighLightLines(Lines: SafeArray of Integer);
```

**Note** Although you must specify the element type when declaring a *SafeArray* type in the Type Library editor, the declaration in the generated `_TLB` unit does not indicate the element type.

### Using Object Pascal or IDL syntax

The Text page of the Type Library editor displays your type information in one of two ways:

- Using an extension of Object Pascal syntax.
- Using the Microsoft IDL.

**Note** When working on a CORBA object, you use neither of these on the text page. Instead, you must use the CORBA IDL.

You can select which language you want to use by changing the setting in the Environment Options dialog. Choose Tools | Environment Options, and specify either Pascal or IDL as the Language on the Type Library page of the dialog.

**Note** The choice of Object Pascal or IDL syntax also affects the choices available on the parameters attributes page.

Like Object Pascal applications in general, identifiers in type libraries are case insensitive. They can be up to 255 characters long, and must begin with a letter or an underscore (`_`).

### Attribute specifications

Object Pascal has been extended to allow type libraries to include attribute specifications. Attribute specifications appear enclosed in square brackets and separated by commas. Each attribute specification consists of an attribute name followed (if appropriate) by a value.

The following table lists the attribute names and their corresponding values.

**Table 34.5** Attribute syntax

Attribute name	Example	Applies to
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo
bindable	[bindable]	members except CoClass members
control	[control]	type library, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	anything
default	[default]	CoClass members
defaultbind	[defaultbind]	members except CoClass members
defaultcollection	[defaultcollection]	members except CoClass members
defaultvtbl	[defaultvtbl]	CoClass members
dispid	[dispid]	members except CoClass members
displaybind	[displaybind]	members except CoClass members
dllname	[dllname 'Helper.dll']	module typeinfo
dual	[dual]	interface typeinfo
helpfile	[helpfile 'c:\help\myhelp.hlp']	type library
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	type library
helpcontext	[helpcontext 2005]	anything except CoClass members and parameters
helpstring	[helpstring 'payroll interface']	anything except CoClass members and parameters
helpstringcontext	[helpstringcontext \$17]	anything except CoClass members and parameters
hidden	[hidden]	anything except parameters
immediatebind	[immediatebind]	members except CoClass members
lcid	[lcid \$324]	type library
licensed	[licensed]	type library, CoClass typeinfo
nonbrowsable	[nonbrowsable]	members except CoClass members
nonextensible	[nonextensible]	interface typeinfo
oleautomation	[oleautomation]	interface typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	members except CoClass members
propput	[propput]	members except CoClass members

**Table 34.5** Attribute syntax (continued)

Attribute name	Example	Applies to
propputref	[propputref]	members except CoClass members
public	[public]	alias typeinfo
readonly	[readonly]	members except CoClass members
replaceable	[replaceable]	anything except CoClass members and parameters
requestedit	[requestedit]	members except CoClass members
restricted	[restricted]	anything except parameters
source	[source]	all members
uidefault	[uidefault]	members except CoClass members
usesgetlasterror	[usesgetlasterror]	members except CoClass members
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	type library, typeinfo (required)
vararg	[vararg]	members except CoClass members
version	[version 1.1]	type library, typeinfo

## Interface syntax

The Object Pascal syntax for declaring interface type information has the form

```
interfacename = interface[(baseinterface)] [attributes]
functionlist
[propertymethodlist]
end;
```

For example, the following text declares an interface with two methods and one property:

```
Interface1 = interface (IDispatch)
[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
function Calculate(optional seed:Integer=0): Integer;
procedure Reset;
procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',version 1.0]
interface Interface1 :IDispatch
{
long Calculate([in, optional, defaultvalue(0)] long seed);
void Reset(void);
[propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
[propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

## Dispatch interface syntax

The Object Pascal syntax for declaring dispinterface type information has the form

```
dispinterfacename = dispinterface [attributes]
functionlist
[propertylist]
end;
```

For example, the following text declares a dispinterface with the same methods and property as the previous interface:

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
 version 1.0,
 helpstring 'dispatch interface for MyObj']
function Calculate(seed:Integer): Integer [dispid 1];
procedure Reset [dispid 2];
property Range: Integer [dispid 3];
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
 version 1.0,
 helpstring "dispatch interface for MyObj"]
dispinterface Interface1
{
 methods:
 [id(1)] int Calculate([in] int seed);
 [id(2)] void Reset(void);
 properties:
 [id(3)] int Value;
};
```

## CoClass syntax

The Object Pascal syntax for declaring CoClass type information has the form

```
classname = coclass(interfacename[interfaceattributes], ...); [attributes];
```

For example, the following text declares a coclass for the interface *IMyInt* and dispinterface *DmyInt*:

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 version 1.0,
 helpstring 'A class',
 appobject]
coclass myapp
{
```

```

methods:
[source] interface IMyInt;
dispinterface DMyInt;
};

```

## Enum syntax

The Object Pascal syntax for declaring Enum type information has the form

```
enumname = ([attributes] enumlist);
```

For example, the following text declares an enumerated type with three values:

```

location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
            helpstring 'location of booth']
  Inside = 1 [helpstring 'Inside the pavillion'];
  Outside = 2 [helpstring 'Outside the pavillion'];
  Offsite = 3 [helpstring 'Not near the pavillion']);;

```

The corresponding syntax in Microsoft IDL is

```

[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
 helpstring 'location of booth']
typedef enum
{
  [helpstring 'Inside the pavillion'] Inside = 1,
  [helpstring 'Outside the pavillion'] Outside = 2,
  [helpstring 'Not near the pavillion'] Offsite = 3
} location;

```

## Alias syntax

The Object Pascal syntax for declaring Alias type information has the form

```
aliasname = basetype[attributes];
```

For example, the following text declares DWORD as an alias for integer:

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

## Record syntax

The Object Pascal syntax for declaring Record type information has the form

```
recordname = record [attributes] fieldlist end;
```

For example, the following text declares a record:

```

Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
             helpstring 'Task description']
  ID: Integer;
  StartDate: TDate;
  EndDate: TDate;
  Ownername: WideString;
  Subtasks: safearray of Integer;
end;

```

## The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    helpstring 'Task description']
typedef struct
{
    long ID;
    DATE StartDate;
    DATE EndDate;
    BSTR Ownername;
    SAFEARRAY (int) Subtasks;
} Tasks;
```

## Union syntax

The Object Pascal syntax for declaring Union type information has the form

```
unionname = record [attributes]
case Integer of
    0: field1;
    1: field2;
    ...
end;
```

For example, the following text declares a union:

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    helpstring 'item description']
case Integer of
    0: (Name: WideString);
    1: (ID: Integer);
    3: (Value: Double);
end;
```

## The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    helpstring 'item description']
typedef union
{
    BSTR Name;
    long ID;
    double Value;
} MyUnion;
```

## Module syntax

The Object Pascal syntax for declaring Module type information has the form

```
modulename = module constants entrypoints end;
```

For example, the following text declares the type information for a module:

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    dllname 'circle.dll']
    PI: Double = 3.14159;
    function area(radius: Double): Double [ entry 1 ]; stdcall;
    function circumference(radius: Double): Double [ entry 2 ]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
    dllname("circle.dll")]
module MyModule
{
    double PI = 3.14159;
    [entry(1)] double _stdcall area([in] double radius);
    [entry(2)] double _stdcall circumference([in] double radius);
};
```

## Creating a new type library

You may want to create a type library that is independent of a particular COM object. For example, you might want to define a type library that contains type definitions that you use in several other type libraries. You can then create a type library of basic definitions and add it to the uses page of other type libraries.

You can also create a type library for an object that is not yet implemented. Once the type library contains the interface definition, you can use the COM object wizard to generate a CoClass and implementation.

To create a new type library,

- 1 Choose File | New | Other to open the New Items dialog box.
- 2 Choose the ActiveX page.
- 3 Select the Type Library icon.
- 4 Choose OK.

The Type Library editor opens with a prompt to enter a name for the type library.

- 5 Enter a name for the type library. Continue by adding elements to your type library.

## Opening an existing type library

When you use the wizards to create an ActiveX control, Automation object, Active form, Active Server Page object, COM object, transactional object, remote data module, or transactional data module, a type library is automatically created with an implementation unit. In addition, you may have type libraries that are associated with other products (servers) that are available on your system.

To open a type library that is not currently part of your project,

- 1 Choose File | Open from the main menu in the IDE.
- 2 In the Open dialog box, set the File Type to type library.
- 3 Navigate to the desired type library files and choose Open.

To open a type library associated with the current project,

- 1 Choose View | Type Library.



**Note** When you use the CORBA Object wizard, you can also choose View | Type Library to edit the CORBA Object interfaces. What you see is not, technically speaking, a type library, but you can use it in much the same way.

Now, you can add interfaces, CoClasses, and other elements of the type library such as enumerations, properties, and methods.

**Note** Changes you make to any type library information with the Type Library editor can be automatically reflected in the associated implementation class. If you want to review the changes before they are added, be sure that the Apply Updates dialog is on. It is on by default and can be changed in the setting, “Display updates before refreshing,” on the Tools | Environment Options | Type Library page. For more information, see “Apply Updates dialog” on page 34-25.

**Tip** When writing client applications, you do not need to open the type library. You only need the *Project\_TLB* unit that the Type Library editor creates from a type library, not the type library itself. You can add this file directly to a client project, or, if the type library is registered on your system, you can use the Import Type Library dialog (Project | Import Type Library).

## Adding an interface to the type library

To add an interface,

**1** On the toolbar, click on the interface icon.

An interface is added to the object list pane prompting you to add a name.

**2** Type a name for the interface.

The new interface contains default attributes that you can modify as needed.

You can add properties (represented by getter/setter functions) and methods to suit the purpose of the interface.

## Modifying an interface using the type library

There are several ways to modify an interface or dispinterface once it is created.

- You can change the interface’s attributes using the page of type information that contains the information you want to change. Select the interface in the object list pane and then use the controls on the appropriate page of type information. For example, you may want to change the parent interface using the attributes page, or use the flags page to change whether or not it is a dual interface.
- You can edit the interface declaration directly by selecting the interface in the object list pane and then editing the declarations on the Text page.
- You can Add properties and methods to the interface (see the next section).
- You can modify the properties and methods already in your interface by changing their type information.
- You can associate it with a CoClass by selecting the CoClass in the object list pane, right-clicking on the Implements page, and choosing Insert Interface.

**Note** When using the type library to add a CORBA interface, most of the information on the attributes page is irrelevant. You will also not need the Flags page.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the Type Library editor to apply your changes to the implementation file by clicking the Refresh button on the toolbar. If you have the Apply Updates dialog enabled, the Type Library editor notifies you before updating the sources and warns you of potential problems. For example, if you rename an event interface by mistake, you may get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You also get a TODO comment in your source file immediately above it.

**Warning** If you ignore this warning and TODO comment, the code will not compile.

## Adding properties and methods to an interface or dispinterface

To add properties or methods to an interface or dispinterface,

- 1 Select the interface, and choose either a property or method icon from the toolbar. If you are adding a property, you can click directly on the property icon to create a read/write property (with both a getter and a setter), or click the down arrow to display a menu of property types.

The property access method members or method member is added to the object list pane, prompting you to add a name.

- 2 Type a name for the member.

The new member contains default settings on its attributes, parameters, and flags pages that you can modify to suit the member. For example, you will probably want to assign a type to a property on the attributes page. If you are adding a method, you will probably want to specify its parameters on the parameters page.

As an alternate approach, you can add properties and methods by typing directly into the text page using Pascal or IDL syntax. For example, if you are working in Pascal syntax, you can type the following property declarations into the text page of an interface:

```
Interfacel = interface(IDispatch)
  [ uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
    version 1.0,
    dual,
    oleautomation ]
  function AutoSelect: Integer [propget, dispid $00000002]; safecall; // Add this
  function AutoSize: WordBool [propget, dispid $00000001]; safecall; // And this
  procedure AutoSize(Value: WordBool) [propput, dispid $00000001]; safecall; // And this
end;
```

If you are working in IDL, you can add the same declarations as follows:

```
[
    uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
    version(1.0),
    dual,
    oleautomation
]
interface Interface1: IDispatch
{ // Add everything between the curly braces
[propget, id(0x00000002)]
    HRESULT _stdcall AutoSelect([out, retval] long Value );
    [propget, id(0x00000003)]
    HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
    [propput, id(0x00000003)]
    HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

After you have added members to an interface using the interface text page, the members appear as separate items in the object list pane, each with its own attributes, flags, and parameters pages. You can modify each new property or method by selecting it in the object list pane and using these pages, or by making edits directly in the text page.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the Type Library editor to apply your changes to the implementation file by clicking the Refresh button on the toolbar. The Type Library editor adds new methods to your implementation class to reflect the new members. You can then locate the new methods in implementation unit's source code and fill in their bodies to complete the implementation.

If you have the Apply Updates dialog enabled, the Type Library editor notifies you of all changes before updating the sources and warns you of potential problems.

## Adding a CoClass to the type library

The easiest way to add a CoClass to your project is to choose File | New | Other from the main menu in the IDE and use the appropriate wizard on the ActiveX or Multitier page of the New Items dialog. The advantage to this approach is that, in addition to adding the CoClass and its interface to the type library, the wizard adds an implementation unit and updates the project file to include the new implementation unit in its uses clause.

If you are not using a wizard, however, you can create a CoClass by clicking the CoClass icon on the toolbar and then specifying its attributes. You will probably want to give the new CoClass a name (on the Attributes page), and may want to use the Flags page to indicate information such as whether the CoClass is an application object, whether it represents an ActiveX control, and so on.

**Note** When you add a CoClass to a type library using the toolbar instead of a wizard, you must generate the implementation for the CoClass yourself and update it by hand every time you change an element on one of the CoClass's interfaces. You can't add members directly to a CoClass. Instead, you implicitly add members when you add an interface to the CoClass.

## Adding an interface to a CoClass

CoClasses are defined by the interfaces they present to clients. While you can add any number of properties and methods to the implementation class of a CoClass, clients can only see those properties and methods that are exposed by interfaces associated with the CoClass.

To associate an interface with a CoClass, right-click in the Implements page for the class and choose Insert Interface to display a list of interfaces from which you can choose. The list includes interfaces that are defined in the current type library and those defined in any type libraries that the current type library references. Choose an interface you want the class to implement. The interface is added to the page with its GUID and other attributes.

If the CoClass was generated by a wizard, the Type Library editor automatically updates the implementation class to include skeletal methods for the methods (including property access methods) of any interfaces you add this way. If you have the Apply Updates dialog enabled, the Type Library editor notifies you before updating the sources and warns you of potential problems.

## Adding an enumeration to the type library

To add enumerations to a type library,

- 1 On the toolbar, click on the enum icon.

An enum type is added to the object list pane prompting you to add a name.

- 2 Type a name for the enumeration.

The new enum is empty and contains default attributes in its attributes page for you to modify.

Add values to the enum by clicking on the New Const button. Then, select each enumerated value and assign it a name (and possibly a value) using the attributes page.

Once you have added an enumeration, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the enumeration as the type for a property or parameter.

## Adding an alias to the type library

To add an alias to a type library,

- 1 On the toolbar, click on the alias icon.

An alias type is added to the object list pane prompting you to add a name.

- 2 Type a name for the alias.

By default, the new alias stands for an Integer type. Use the Attributes page to change this to the type you want the alias to represent.

Once you have added an alias, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the alias as the type for a property or parameter.

## Adding a record or union to the type library

To add a record or union to a type library,

- 1 On the toolbar, click on the record icon or the union icon.

The selected type element is added to the object list pane prompting you to add a name.

- 2 Type a name for the record or union.

At this point, the new record or union contains no fields.

- 3 With the record or union selected in the object list pane, click on the field icon in the toolbar. Specify the field's name and type, using the Attributes page.
- 4 Repeat step 3 for as many fields as you need.

Once you have defined the record or union, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the record or union as the type for a property or parameter.

## Adding a module to the type library

To add a module to a type library,

- 1 On the toolbar, click on the module icon.

The selected module is added to the object list pane prompting you to add a name.

- 2 Type a name for the module.

- 3 On the Attributes page, specify the name of the DLL whose entry points the Module represents.

- 4 Add any methods from the DLL you specified in step 3 by clicking on the Method icon in the toolbar and then using the attributes pages to describe the method.

- 5 Add any constants you want the module to define by clicking on the Const icon on the toolbar. For each constant, specify a name, type, and value.

## Saving and registering type library information

After modifying your type library, you'll want to save and register the type library information.

Saving the type library automatically updates:

- The binary type library file (.tlb extension).
- The *Project\_TLB* unit that represents its contents
- The implementation code for any CoClasses that were generated by a wizard.

**Note** The type library is stored as a separate binary (.TLB) file, but is also linked into the server (.EXE, DLL, or .OCX).

**Note** When using the Type Library editor for CORBA interfaces, the *Project\_TLB.pas* unit defines the stub and skeleton objects required by the CORBA application.

The Type Library editor gives you options for storing your type library information. Which way you choose depends on what stage you are at in implementing the type library:

- **Save**, to save both the .TLB and the *Project\_TLB* unit to disk.
- **Refresh**, to update the type library units in memory only.
- **Register**, to add an entry for the type library in your system's Windows registry. This is done automatically when the server with which the .TLB is associated is itself registered.
- **Export**, to save a .IDL file that contains the type and interface definitions in IDL syntax.

All the above methods perform syntax checking. When you refresh, register, or save the type library, Delphi automatically updates the implementation unit of any CoClasses that were created using a wizard. Optionally, you can review these updates before they are committed, if you have the Type Library editor option, **Apply Updates** on.

## Apply Updates dialog

The Apply Updates dialog appears when you refresh, register, or save the type library if you have selected "Display updates before refreshing" in the Tools | Environment Options | Type Library page (which is on by default).

Without this option, the Type Library editor automatically updates the sources of the associated object when you make changes in the editor. With this option, you have a chance to veto the proposed changes when you attempt to refresh, save, or register the type library.

The Apply Updates dialog will warn you about potential errors, and will insert TODO comments in your source file. For example, if you rename an event by mistake, you will get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You will also get a TODO comment in your source file immediately above it.

**Note** If you ignore this warning and TODO comment, the code will not compile.

## Saving a type library

Saving a type library

- Performs a syntax and validity check.
- Saves information out to a .TLB file.
- Saves information out to the *Project\_TLB* unit.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with a CoClass that was generated by a wizard.

To save the type library, choose **File | Save** from the Delphi main menu.

## Refreshing the type library

Refreshing the type library

- Performs a syntax check.
- Regenerates the Delphi type library units in memory only. It does not save any files to disk.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with a CoClass that was generated by a wizard.

To refresh the type library choose the Refresh icon on the Type Library editor toolbar.

**Note** If you have renamed items in the type library, refreshing the implementation may create duplicate entries. In this case, you must move your code to the correct entry and delete any duplicates. Similarly, if you delete items in the type library, refreshing the implementation does not remove them from CoClasses (under the assumption that you are merely removing them from visibility to clients). You must delete these items manually in the implementation unit if they are no longer needed.

## Registering the type library

Typically, you do not need to explicitly register a type library because it is registered automatically when you register your COM server application (see "Registering a COM object" on page 36-16). However, when you create a type library using the Type Library wizard, it is not associated with a server object. In this case, you can register the type library directly using the toolbar.

Registering the type library,

- Performs a syntax check
- Adds an entry to the Windows Registry for the type library

To register the type library, choose the Register icon on the Type Library editor toolbar.

## Exporting an IDL file

Exporting the type library,

- Performs a syntax check.
- Creates an IDL file that contains the type information declarations. This file describes the type information in either CORBA IDL or Microsoft IDL.

To export the type library, choose the Export icon on the Type Library editor toolbar.

## Deploying type libraries

---

By default, when you have a type library that was created as part of an ActiveX or Automation server project, the type library is automatically linked into the .DLL, .OCX, or EXE as a resource.

You can, however, deploy your application with the type library as a separate .TLB, as Delphi maintains the type library, if you prefer.

Historically, type libraries for Automation applications were stored as a separate file with the .TLB extension. Now, typical Automation applications compile the type libraries into the .OCX or .EXE file directly. The operating system expects the type library to be the first resource in the executable (.DLL, .OCX, or .EXE) file.

When you make type libraries other than the primary project type library available to application developers, the type libraries can be in any of the following forms:

- A resource. This resource should have the type TYPELIB and an integer ID. If you choose to build type libraries with a resource compiler, it must be declared in the resource (.RC) file as follows:

```
1 typelib mylib1.tlb
2 typelib mylib2.tlb
```

There can be multiple type library resources in an ActiveX library. Application developers use the resource compiler to add the .TLB file to their own ActiveX library.

- Stand-alone binary files. The .TLB file output by the Type Library editor is a binary file.





## Creating COM clients

COM clients are applications that make use of a COM object implemented by another application or library. The most common types are applications that control an Automation server (Automation controllers) and applications that host an ActiveX control (ActiveX containers).

At first glance these two types of COM client are very different: The typical Automation controller launches an external server EXE and issues commands to make that server perform tasks on its behalf. The Automation server is usually nonvisual and out-of-process. The typical ActiveX client, on the other hand, hosts a visual control, using it much the same way you use any control on the Component palette. ActiveX servers are always in-process servers.

However, the task of writing these two types of COM client is remarkably similar: The client application obtains an interface for the server object and uses its properties and methods. Delphi makes this particularly easy by letting you wrap the server CoClass in a component on the client, which you can even install on the Component palette. Samples of such component wrappers appear on two pages of the Component palette: sample ActiveX wrappers appear on the ActiveX page and sample Automation objects appear on the Servers page.

When writing a COM client, you must understand the interface that the server exposes to clients, just as you must understand the properties and methods of a component from the Component palette to use it in your application. This interface (or set of interfaces) is determined by the server application, and typically published in a type library. For specific information on a particular server application's published interfaces, you should consult that application's documentation.

Even if you do not choose to wrap a server object in a component wrapper and install it on the Component palette, you must make its interface definition available to your application. To do this, you can import the server's type information.

**Note** You can also query the type information directly using COM APIs, but Delphi provides no special support for this.

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. These are discussed in “Creating Clients for servers that do not have a type library” on page 35-15.

## Importing type library information

---

To make information about the COM server available to your client application, you must import the information about the server that is stored in the server’s type library. Your application can then use the resulting generated classes to control the server object.

There are two ways to import type library information:

- You can use the Import Type Library dialog to import all available information about the server types, objects, and interfaces. This is the most general method, because it lets you import information from any type library and can optionally generate component wrappers for all creatable CoClasses in the type library that are not flagged as Hidden, Restricted, or PreDeclID.
- You can use the Import ActiveX dialog if you are importing from the type library of an ActiveX control. This imports the same type information, but only creates component wrappers for CoClasses that represent ActiveX controls.
- You can use the command line utility tlibimp.exe which provides additional configuration options not available from within the IDE.
- A type library generated using a wizard is automatically imported using the same mechanism as the import type library menu item.

Regardless of which method you choose to import type library information, the resulting dialog creates a unit with the name *TypeLibName\_TLB*, where *TypeLibName* is the name of the type library. This file contains declarations for the classes, types, and interfaces defined in the type library. By including it in your project, those definitions are available to your application so that you can create objects and call their interfaces. This file may be recreated by the IDE from time to time; as a result, making manual changes to the file is not recommended.

In addition to adding type definitions to the *TypeLibName\_TLB* unit, the dialog can also create VCL class wrappers for any CoClasses defined in the type library. When you use the Import Type Library dialog, these wrappers are optional. When you use the Import ActiveX dialog, they are always generated for all CoClasses that represent controls.

The generated class wrappers represent the CoClasses to your application, and expose the properties and methods of its interfaces. If a CoClass supports the interfaces for generating events (*IConnectionPointContainer* and *IConnectionPoint*), the VCL class wrapper creates an event sink so that you can assign event handlers for the events as simply as you can for any other component. If you tell the dialog to install the generated VCL classes on the Component palette, you can use the Object Inspector to assign property values and event handlers.

**Note** The Import Type Library dialog does not create class wrappers for COM+ event objects. To write a client that responds to events generated by a COM+ event object, you must create the event sink programmatically. This process is described in “Handling COM+ events” on page 35-14.

For more details about the code generated when you import a type library, see “Code generated when you import type library information” on page 35-5.

## Using the Import Type Library dialog

---

To import a type library,

- 1 Choose Project | Import Type Library.
- 2 Select the type library from the list.

The dialog lists all the libraries registered on this system. If the type library is not in the list, choose the Add button, find and select the type library file, choose OK. This registers the type library, making it available. Then repeat step 2. Note that the type library could be a stand-alone type library file (.tlb, .olb), or a server that provides a type library (.dll, .ocx, .exe).

- 3 If you want to generate a VCL component that wraps a CoClass in the type library, check Generate Component Wrapper. If you do not generate the component, you can still use the CoClass by using the definitions in the *TypeLibName\_TLB* unit. However, you will have to write your own calls to create the server object and, if necessary, to set up an event sink.

The Import Type Library dialog only imports CoClasses that have the CanCreate flag set and that do not have the Hidden, Restricted, or PreDeclID flags set. These flags can be overridden using the command-line utility *tlibimp.exe*.

- 4 If you do not want to install a generated component wrapper on the Component palette, choose Create Unit. This generates the *TypeLibName\_TLB* unit and, if you checked Generate Component Wrapper in step 3, adds the declaration of the component wrapper. This exits the Import Type Library dialog.
- 5 If you want to install the generated component wrapper on the Component palette, select the Palette page on which this component will reside and then choose Install. This generates the *TypeLibName\_TLB* unit, like the Create Unit button, and then displays the Install component dialog, letting you specify the package where the components should reside (either an existing package or a new one). This button is grayed out if no component can be created for the type library.

When you exit the Import Type Library dialog, the new *TypeLibName\_TLB* unit appears in the directory specified by the Unit dir name control. This file contains declarations for the elements defined in the type library, as well as the generated component wrapper if you checked Generate Component Wrapper.

In addition, if you installed the generated component wrapper, a server object that the type library described now resides on the Component palette. You can use the Object Inspector to set properties or write an event handler for the server. If you add

the component to a form or data module, you can right-click on it at design time to see its property page (if it supports one).

**Note** The Servers page of the Component palette contains a number of example Automation servers that were imported this way for you.

## Using the Import ActiveX dialog

---

To import an ActiveX control,

- 1 Choose Component | Import ActiveX Control.
- 2 Select the type library from the list.

The dialog lists all the registered libraries that define ActiveX controls. (This is a subset of the libraries listed in the Import Type Library dialog.) If the type library is not in the list, choose the Add button, find and select the type library file, choose OK. This registers the type library, making it available. Then repeat step 2. Note that the type library could be a stand-alone type library file (.tlb, .olb), or an ActiveX server (.dll, .ocx).

- 3 If you do not want to install the ActiveX control on the Component palette, choose Create Unit. This generates the *TypeLibName\_TLB* unit and adds the declaration of its component wrapper. This exits the Import ActiveX dialog.
- 4 If you want to install the ActiveX control on the Component palette, select the Palette page on which this component will reside and then choose Install. This generates the *TypeLibName\_TLB* unit, like the Create Unit button, and then displays the Install component dialog, letting you specify the package where the components should reside (either an existing package or a new one).

When you exit the Import ActiveX dialog, the new *TypeLibName\_TLB* unit appears in the directory specified by the Unit dir name control. This file contains declarations for the elements defined in the type library, as well as the generated component wrapper for the ActiveX control.

**Note** Unlike the Import Type Library dialog where it is optional, the import ActiveX dialog always generates a component wrapper. This is because, as a visual control, an ActiveX control needs the additional support of the component wrapper so that it can fit in with VCL forms.

If you installed the generated component wrapper, an ActiveX control now resides on the Component palette. You can use the Object Inspector to set properties or write event handlers for this control. If you add the control to a form or data module, you can right-click on it at design time to see its property page (if it supports one).

**Note** The ActiveX page of the Component palette contains a number of example ActiveX controls that were imported this way for you.

## Code generated when you import type library information

---

Once you import a type library, you can view the generated *TypeLibName\_TLB* unit. At the top, you will find the following:

- Constant declarations giving symbolic names to the GUIDS of the type library and its interfaces and CoClasses. The names for these constants are generated as follows:
  - The GUID for the type library has the form *LBID\_TypeLibName*, where *TypeLibName* is the name of the type library.
  - The GUID for an interface has the form *IID\_InterfaceName*, where *InterfaceName* is the name of the interface.
  - The GUID for a dispinterface has the form *DIID\_InterfaceName*, where *InterfaceName* is the name of the dispinterface.
  - The GUID for a CoClass has the form *CLASS\_ClassName*, where *ClassName* is the name of the CoClass.
- Declarations for the CoClasses in the type library. These map each CoClass to its default interface.
- Declarations for the interfaces and dispinterfaces in the type library.
- Declarations for a creator class for each CoClass whose default interface supports Vtable binding. The creator class has two class methods, *Create* and *CreateRemote*, that can be used to instantiate the CoClass locally (*Create*) or remotely (*CreateRemote*). These methods return the default interface for the CoClass.

These declarations provide you with what you need to create instances of the CoClass and access its interface. All you need do is add the generated *TypeLibName\_TLB.pas* file to the uses clause of the unit where you wish to bind to a CoClass and call its interfaces.

**Note** This portion of the *TypeLibName\_TLB* unit is also generated when you use the Type Library editor or the command-line utility TLBIMP.

If you want to use an ActiveX control, you also need the generated VCL wrapper in addition to the declarations described above. The VCL wrapper handles window management issues for the control. You may also have generated a VCL wrapper for other CoClasses in the Import Type Library dialog. These VCL wrappers simplify the task of creating server objects and calling their methods. They are especially recommended if you want your client application to respond to events.

The declarations for generated VCL wrappers appear at the bottom of the interface section. Component wrappers for ActiveX controls are descendants of *TOleControl*. Component wrappers for Automation objects descend from *TOleServer*. The generated component wrapper adds the properties, events, and methods exposed by the CoClass's interface. You can use this component like any other VCL component.

**Warning** You should not edit the generated *TypeLibName\_TLB* unit. It is regenerated each time the type library is refreshed, so any changes will be overwritten.

**Note** For the most up-to-date information about the generated code, refer to the comments in the automatically-generated *TypeLibName\_TLB* unit.

## Controlling an imported object

---

After importing type library information, you are ready to start programming with the imported objects. How you proceed depends in part on the objects, and in part on whether you have chosen to create component wrappers.

### Using component wrappers

---

If you generated a component wrapper for your server object, writing your COM client application is not very different from writing any other application that contains VCL components. The server object's properties, methods, and events are already encapsulated in the VCL component. You need only assign event handlers, set property values, and call methods.

To use the properties, methods, and events of the server object, see the documentation for your server. The component wrapper automatically provides a dual interface where possible. Delphi determines the VTable layout from information in the type library.

In addition, your new component inherits certain important properties and methods from its base class.

### ActiveX wrappers

You should always use a component wrapper when hosting ActiveX controls, because the component wrapper integrates the control's window into the VCL framework.

The properties and methods an ActiveX control inherits from *TOleControl* allow you to access the underlying interface or obtain information about the control. Most applications, however, do not need to use these. Instead, you use the imported control the same way you would use any other VCL control.

Typically, ActiveX controls provide a property page that lets you set their properties. Property pages are similar to the component editors some components display when you double-click on them in the form designer. To display an ActiveX control's property page, right click and choose Properties.

The way you use most imported ActiveX controls is determined by the server application. However, ActiveX controls use a standard set of notifications when they represent the data from a database field. See "Using data-aware ActiveX controls" on page 35-8 for information on how to host such ActiveX controls.

## Automation object wrappers

The wrappers for Automation objects let you control how you want to form the connection to your server object:

- The *ConnectKind* property indicates whether the server is local or remote and whether you want to connect to a server that is already running or if a new instance should be launched. When connecting to a remote server, you must specify the machine name using the *RemoteMachineName* property.
- Once you have specified the *ConnectKind*, there are three ways you can connect your component to the server:
  - you can explicitly connect to the server by calling the component's *Connect* method.
  - You can tell the component to connect automatically when your application starts up by setting the *AutoConnect* property to *True*.
  - You do not need to explicitly connect to the server. The component automatically forms a connection when you use one of the server's properties or methods using the component.

Calling methods or accessing properties is the same as using any other component:

```
TServerComponent1.DoSomething;
```

Handling events is easy, because you can use the Object Inspector to write event handlers. Note, however, that the event handler on your component may have slightly different parameters than those defined for the event in the type library. Specifically, pointer types (var parameters and interface pointers) are changed to Variants. You must explicitly cast var parameters to the underlying type before assigning a value. Interface pointers can be cast to the appropriate interface type using the **as** operator. For example, the following code shows an event handler for the ExcelApplication event, *OnNewWorkBook*. The event handler has a parameter that provides the interface of another CoClass (ExcelWorkbook). However, the interface is not passed as an ExcelWorkbook interface pointer, but rather as an OleVariant.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
  { Note how the OleVariant for the interface must be cast to the correct type }
  ExcelWorkbook1.ConnectTo((iUnknown(Wb) as ExcelWorkbook));
end;
```

In this example, the event handler assigns the workbook to an ExcelWorkbook component (ExcelWorkbook1). This demonstrates how to connect a component wrapper to an existing interface by using the *ConnectTo* method. The *ConnectTo* method is added to the generated code for the component wrapper.

Servers that have an application object expose a *Quit* method on that object to let clients terminate the connection. *Quit* typically exposes functionality that is equivalent to using the File menu to quit the application. Code to call the *Quit* method is generated in your component's *Disconnect* method. If it is possible to call the *Quit* method with no parameters, the component wrapper also has an *AutoQuit* property. *AutoQuit* causes your controller to call *Quit* when the component is freed. If you want to disconnect at some other time, or if the *Quit* method requires parameters, you must call it explicitly. *Quit* appears as a public method on the generated component.



## Using data-aware ActiveX controls

---

When you use a data-aware ActiveX control in a Delphi application, you must associate it with the database whose data it represents. To do this, you need a data source component, just as you need a data source for any data-aware VCL control.

After you place the data-aware ActiveX control in the form designer, assign its *DataSource* property to the data source that represents the desired dataset. Once you have specified a data source, you can use the Data Bindings editor to link the control's data-bound property to a field in the dataset.

To display the Data Bindings editor, right-click the data-aware ActiveX control to display a list of options. In addition to the basic options, the additional Data Bindings item appears. Select this item to see the Data Bindings editor, which lists the names of fields in the dataset and the bindable properties of the ActiveX control.

To bind a field to a property,

- 1 In the ActiveX Data Bindings Editor dialog, select a field and a property name.  
Field Name lists the fields of the database and Property Name lists the ActiveX control properties that can be bound to a database field. The dispID of the property is in parentheses, for example, Value(12).
- 2 Click Bind and OK.

**Note** If no properties appear in the dialog, the ActiveX control contains no data-aware properties. To enable simple data binding for a property of an ActiveX control, use the type library as described in "Enabling simple data binding with the type library" on page 38-10.

The following example walks you through the steps of using a data-aware ActiveX control in the Delphi container. This example uses the Microsoft Calendar Control, which is available if you have Microsoft Office 97 installed on your system.

- 1 From the Delphi main menu, choose Component | Import ActiveX Control.
- 2 Select a data-aware ActiveX control, such as the Microsoft Calendar control 8.0, change its class name to *TCalendarAXControl*, and click Install.
- 3 In the Install dialog, click OK to add the control to the default user package, which makes the control available on the Palette.
- 4 Choose Close All and File | New | Application to begin a new application.
- 5 From the ActiveX tab, drop a *TCalendarAXControl* object, which you just added to the Palette, onto the form.
- 6 From the Data Access tab, drop a *DataSource* and *Table* object onto the form.
- 7 Select the *DataSource* object and set its *DataSet* property to *Table1*.
- 8 Select the *Table* object and do the following:
  - Set the *DatabaseName* property to DBDEMOS
  - Set the *TableName* property to EMPLOYEE.DB
  - Set the *Active* property to *True*

- 9 Select the *TCalendarAXControl* object and set its *DataSource* property to *DataSource1*.
- 10 Select the *TCalendarAXControl* object, right-click, and choose Data Bindings to invoke the ActiveX Control Data Bindings Editor.  
Field Name lists all the fields in the active database. Property Name lists those properties of the ActiveX Control that can be bound to a database field. The dispID of the property is in parentheses.
- 11 Select the *HireDate* field and the *Value* property name, choose Bind, and OK.  
The field name and property are now bound.
- 12 From the Data Controls tab, drop a *DBGrid* object onto the form and set its *DataSource* property to *DataSource1*.
- 13 From the Data Controls tab, drop a *DBNavigator* object onto the form and set its *DataSource* property to *DataSource1*.
- 14 Run the application.
- 15 Test the application as follows:  
With the *HireDate* field displayed in the *DBGrid* object, navigate through the database using the Navigator object. The dates in the ActiveX control change as you move through the database.

## Example: Printing a document with Microsoft Word

---

The following steps show how to create an Automation controller that prints a document using Microsoft Word 8 from Office 97.

**Before you begin:** Create a new project that consists of a form, a button, and an open dialog box (*TOpenDialog*). These controls constitute the Automation controller.

### Step 1: Prepare Delphi for this example

For your convenience, Delphi has provided many common servers, such as Word, Excel, and PowerPoint, on the Component palette. To demonstrate how to import a server, we use Word. Since it already exists on the Component palette, this first step asks you to remove the package containing Word so that you can see how to install it on the palette. Step 4 describes how to return the Component palette to its normal state.

To remove Word from the Component palette,

- 1 Choose Component | Install packages.
- 2 Click Borland Sample Automation Server components and choose Remove.

The Servers page of the Component palette no longer contains any of the servers supplied with Delphi. (If no other servers have been imported, the Servers page also disappears.)

## Step 2: Import the Word type library

To import the Word type library,

- 1 Choose Project | Import Type Library.
- 2 In the Import Type Library dialog,
  - 1 Select Microsoft Office 8.0 Object Library.

If Word (Version 8) is not in the list, choose the Add button, go to Program Files\Microsoft Office\Office, select the Word type library file, MSWord8.olb choose Add, and then select Word (Version 8) from the list.

- 2 For Palette Page, choose Servers.
- 3 Choose Install.

The Install dialog appears. Select the Into New Packages tab and type WordExample to create a new package containing this type library.

- 3 Go to the Servers Palette Page, select WordApplication and place it on a form.
- 4 Write an event handler for the button object as described in the next step.

## Step 3: Use a VTable or dispatch interface object to control Microsoft Word

You can use either a VTable or a dispatch object to control Microsoft Word.

### Using a VTable interface object

By dropping an instance of the WordApplication object onto your form, you can easily access the control using a VTable interface object. You simply call on methods of the class you just created. For Word, this is the *TWordApplication* class.

- 1 Select the button, double-click its *OnClick* event handler and supply the following event handling code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileName: OleVariant;
begin
  if OpenFileDialog1.Execute then
    begin
      FileName := OpenFileDialog1.FileName;

      WordApplication1.Documents.Open(FileName,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam);

      WordApplication1.ActiveDocument.PrintOut(
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam);
    end;
end;
```

- 2 Build and run the program. By clicking the button, Word prompts you for a file to print.

### Using a dispatch interface object

As an alternate, you can use a dispatch interface for late binding. To use a dispatch interface object, you create and initialize the Application object using the `_ApplicationDisp` dispatch wrapper class as follows. Notice that dispinterface methods are “documented” by the source as returning Vtable interfaces, but, in fact, you must cast them to dispatch interfaces.

- 1 Select the button, double-click its *OnQuit* event handler and supply the following event handling code:

```

procedure TForm1.Button1Click(Sender: TObject);
var
  MyWord : _ApplicationDisp;
  FileName : OleVariant;
begin
  if OpenDialog1.Execute then
    begin
      FileName := OpenDialog1.FileName;
      MyWord := CoWordApplication.Create as
        _ApplicationDisp;
      (MyWord.Documents as DocumentsDisp).Open(FileName, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam);
      (MyWord.ActiveDocument as _DocumentDisp).PrintOut(EmptyParam,
        EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam, EmptyParam, EmptyParam,
        EmptyParam, EmptyParam, EmptyParam);
      MyWord.Quit(EmptyParam, EmptyParam, EmptyParam);
    end;
  end;

```

- 2 Build and run the program. By clicking the button, Word prompts you for a file to print.

### Step 4: Clean up the example

After completing this example, you will want to restore Delphi to its original form.

- 1 Delete the objects on this Servers page:
  - Choose Component | Install Packages.
  - From the list, select the WordExample package and click remove.
  - Click Yes to the message box asking for confirmation.
  - Exit the Install Packages dialog by clicking OK.
- 2 Return the Borland Sample Automation Server Components package:
  - Choose Component | Install Packages.
  - Click the Add button.

- In the resulting dialog, choose `dclaxserver50.bpl`.
- Exit the Install Packages dialog by clicking OK.

## Writing client code based on type library definitions

---

Although you must use a component wrapper for hosting an ActiveX control, you can write an Automation controller using only the definitions from the type library that appear in the *TypeLibName\_TLB* unit. This process is a bit more involved than letting a component do the work, especially if you need to respond to events.

### Connecting to a server

Before you can drive an Automation server from your controller application, you must obtain a reference to an interface it supports. Typically, you connect to a server through its main interface. For example, you connect to Microsoft Word through the `WordApplication` component.

If the main interface is a dual interface, you can use the creator objects in the *TypeLibName\_TLB.pas* file. The creator classes have the same name as the `CoClass`, with the prefix “Co” added. You can connect to a server on the same machine by calling the *Create* method, or a server on a remote machine using the *CreateRemote* method. Because *Create* and *CreateRemote* are class methods, you do not need an instance of the creator class to call them.

```
MyInterface := CoServerClassName.Create;
MyInterface := CoServerClassName.CreateRemote('Machine1');
```

*Create* and *CreateRemote* return the default interface for the `CoClass`.

If the default interface is a dispatch interface, then there is no Creator class generated for the `CoClass`. Instead, you can call the global *CreateOleObject* function, passing in the GUID for the `CoClass` (there is a constant for this GUID defined at the top of the *\_TLB* unit). *CreateOleObject* returns an `IDispatch` pointer for the default interface.

### Controlling an Automation server using a dual interface

After using the automatically generated creator class to connect to the server, you call methods of the interface. For example,

```
var
  MyInterface : _Application;
begin
  MyInterface := CoWordApplication.Create;
  MyInterface.DoSomething;
```

The interface and creator class are defined in the *TypeLibName\_TLB* unit that is generated automatically when you import a type library.

For information about dual interfaces, see “Dual interfaces” on page 36-13.

## Controlling an Automation server using a dispatch interface

Typically, you use the dual interface to control the Automation server, as described above. However, you may find a need to control an Automation server with a dispatch interface because no dual interface is available.

To call the methods of a dispatch interface,

- 1 Connect to the server, using the global *CreateOleObject* function.
- 2 Use the **as** operator to cast the *IDispatch* interface returned by *CreateOleObject* to the dispinterface for the CoClass. This dispinterface type is declared in the *TypeLibName\_TLB* unit.
- 3 Control the Automation server by calling methods of the dispinterface.

Another way to use dispatch interfaces is to assign them to a *Variant*. By assigning the interface returned by *CreateOleObject* to a *Variant*, you can take advantage of the *Variant* type's built-in support for interfaces. Simply call the methods of the interface, and the *Variant* automatically handles all *IDispatch* calls, fetching the dispatch ID and invoking the appropriate method. The *Variant* type includes built-in support for calling dispatch interfaces, through its **var**.

```
V: Variant;
begin
  V := CreateOleObject('TheServerObject');
  V.MethodName; { calls the specified method }
  ...
```

An advantage of using *Variants* is that you do not need to import the type library, because *Variants* use only the standard *IDispatch* methods to call the server. The trade-off is that *Variants* are slower, because they use dynamic binding at runtime.

For more information on dispatch interfaces, see “Automation interfaces” on page 36-12.

## Handling events in an automation controller

When you generate a Component wrapper for an object whose type library you import, you can respond to events simply using the events that are added to the generated component. If you do not use a Component wrapper, however, (or if the server uses COM+ events), you must write the event sink code yourself.

### Handling Automation events programmatically

Before you can handle events, you must define an event sink. This is a class that implements the event dispatch interface that is defined in the server's type library.

To write the event sink, create an object that implements the event dispatch interface:

```
TServerEventsSink = class(TObject, _TheServerEvents)
...{ declare the methods of _TheServerEvents here }
end;
```

Once you have an instance of your event sink, you must inform the server object of its existence so that the server can call it. To do this, you call the global *InterfaceConnect* procedure, passing it

- The interface to the server that generates events.
- The GUID for the event interface that your event sink handles.
- An IUnknown interface for your event sink.
- A variable that receives a Longint that represents the connection between the server and your event sink.

```
{MyInterface is the server interface you got when you connected to the server }  
InterfaceConnect(MyInterface, DIID_TheServerEvents,  
                MyEventSinkObject as IUnknown, cookievar);
```

After calling *InterfaceConnect*, your event sink is connected and receives calls from the server when events occur.

You must terminate the connection before you free your event sink. To do this, call the global *InterfaceDisconnect* procedure, passing it all the same parameters except for the interface to your event sink (and the final parameter is ingoing rather than outgoing):

```
InterfaceDisconnect(MyInterface, DIID_TheServerEvents, cookievar);
```

**Note** You must be certain that the server has released its connection to your event sink before you free it. Because you don't know how the server responds to the disconnect notification initiated by *InterfaceDisconnect*, this may lead to a race condition if you free your event sink immediately after the call. The easiest way to guard against problems is to have your event sink maintain its own reference count that is not decremented until the server releases the event sink's interface.

### Handling COM+ events

Under COM+, servers use a special helper object to generate events rather than a set of special interfaces (*IConnectionPointContainer* and *IConnectionPoint*). Because of this, you can't use an event sink that descends from *TEventDispatcher*. *TEventDispatcher* is designed to work with those interfaces, not COM+ event objects.

Instead of defining an event sink, your client application defines a subscriber object. Subscriber objects, like event sinks, provide the implementation of the event interface. They differ from event sinks in that they subscribe to a particular event object rather than connecting to a server's connection point.

To define a subscriber object, use the COM Object wizard, selecting the event object's interface as the one you want to implement. The wizard generates an implementation unit with skeletal methods that you can fill in to create your event handlers. For more information about using the COM Object wizard to implement an existing interface, see "Using the COM object wizard" on page 36-2.

**Note** You may need to add the event object's interface to the registry using the wizard if it does not appear in the list of interfaces you can implement.

Once you create the subscriber object, you must subscribe to the event object's interface or to individual methods (events) on that interface. There are three types of subscriptions from which you can choose:

- **Transient subscriptions.** Like traditional event sinks, transient subscriptions are tied to the lifetime of an object instance. When the subscriber object is freed, the subscription ends and COM+ no longer forwards events to it.

- **Persistent subscriptions.** These are tied to the object class rather than a specific object instance. When the event occurs, COM locates or launches an instance of the subscriber object and calls its event handler. In-process objects (DLLs) use this type of subscription.
- **Per-user subscriptions.** These subscriptions provide a more secure version of transient subscriptions. Both the subscriber object and the server object that fires events must be running under the same user account on the same machine.

**Note** Objects that subscribe to COM+ events must be installed in a COM+ application.

## Creating Clients for servers that do not have a type library

---

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. To write clients that host such objects, you can use the *TOleContainer* component. This component appears on the System page of the Component palette.

*TOleContainer* acts as a host site for an Ole2 object. It implements the *IOleClientSite* interface and, optionally, *IOleDocumentSite*. Communication is handled using OLE verbs.

To use *TOleContainer*,

- 1 Place a *TOleContainer* component on your form.
- 2 Set the *AllowActiveDoc* property to *True* if you want to host an Active document.
- 3 Set the *AllowInPlace* property to indicate whether the hosted object should appear in the *TOleContainer*, or in a separate window.
- 4 Write event handlers to respond when the object is activated, deactivated, moved, or resized.
- 5 To bind the *TOleContainer* object at design time, right click and choose Insert Object. In the Insert Object dialog, choose a server object to host.
- 6 To bind the *TOleContainer* object at runtime, you have several methods to choose from, depending on how you want to identify the server object. These include *CreateObject*, which takes a program id, *CreateObjectFromFile*, which takes the name of a file to which the object has been saved, *CreateObjectFromInfo*, which takes a record containing information on how to create the object, or *CreateLinkToFile*, which takes the name of a file to which the object was saved and links to it rather than embeds it.
- 7 Once the object is bound, you can access its interface using the *OleObjectInterface* property. However, because communication with Ole2 objects was based on OLE verbs, you will most likely want to send commands to the server using the *DoVerb* method.
- 8 When you want to release the server object, call the *DestroyObject* method.





## Creating simple COM servers

Delphi provides wizards to help you create various COM objects. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call.

**Note** COM servers and Automation is not available for use in CLX applications. This technology is for use on Windows only and is not cross-platform.

Two wizards, in particular, ease the process of creating simple COM objects:

- The COM Object wizard builds a lightweight COM object whose default interface descends from *IUnknown* or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create.
- The Automation Object wizard creates a simple Automation object whose default interface descends from *IDispatch*. *IDispatch* introduces a standard marshaling mechanism and support for late binding of interface calls.

**Note** COM defines many standard interfaces and mechanisms for handling specific situations. The Delphi wizards automate the most common tasks. However, some tasks, such as custom marshaling, are not supported by any Delphi wizards. For information on that and other technologies not explicitly supported by Delphi, refer to the Microsoft Developer's Network (MSDN) documentation. The Microsoft Web site also provides current information on COM support.

## Overview of creating a COM object

---

Whether you use the Automation object wizard to create a new Automation server or the COM object wizard to create some other type of COM object, the process you follow is the same. It involves these steps:

- 1 Design the COM object.
- 2 Use the COM Object wizard or the Automation Object wizard to create the server object.
- 3 Define the interface that the object exposes to clients.
- 4 Register the COM object.
- 5 Test and debug the application.

## Designing a COM object

---

When designing the COM object, you need to decide what COM interfaces you want to implement. You can write a COM object to implement an interface that has already been defined, or you can define a new interface for your object to implement. In addition, you can have your object support more than one interface. For information about standard COM interfaces that you might want to support, see the MSDN documentation.

- To create a COM object that implements an existing interface, use the COM Object wizard.
- To create a COM object that implements a new interface that you define, use either the COM Object wizard or the Automation Object wizard. The COM object wizard can generate a new default interface that descends from *IUnknown*, and the Automation object gives your object a default interface that descends from *IDispatch*. No matter which wizard you use, you can always use the Type Library editor later to change the parent interface of the default interface that the wizard generates.

In addition to deciding what interfaces to support, you must decide whether the COM object is an in-process server, out-of-process server, or remote server. For in-process servers and for out-of-process and remote servers that use a type library, COM marshals the data for you. Otherwise, you must consider how to marshal the data to out-of-process servers. For information on server types, see, “In-process, out-of-process, and remote servers,” on page 33-6.

## Using the COM object wizard

---

The COM object wizard performs the following tasks:

- Creates a new unit.

- Defines a new class that descends from *TComObject* and sets up the class factory constructor. For more information on the base class, see “Code generated by wizards” on page 33-21.
- Optionally, adds a type library to your project and adds your object and its interface to the type library.

Before you create a COM object, create or open the project for the application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the COM object wizard,

- 1 Choose File | New | Other to open the New Items dialog box.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the COM object icon.

In the wizard, you must specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. The class created to implement your object has this name with a ‘T’ prepended. If you do not choose to implement an existing interface, the wizard gives your CoClass a default interface that has this name with an ‘I’ prepended.
- **Interface to implement:** By default, the wizard gives your object a default interface that descends from *IUnknown*. After exiting the wizard, you can then use the Type Library editor to add properties and methods to this interface. However, you can also select a pre-defined interface for your object to implement. Click the List button in the COM object wizard to bring up the Interface Selection wizard, where you can select any dual or custom interface defined in a type library registered on your system. The interface you select becomes the default interface for your new CoClass. The wizard adds all the methods on this interface to the generated implementation class, so that you only need to fill in the bodies of the methods in the implementation unit. Note that if you select an existing interface, the interface is not added to your project’s type library. This means that when deploying your object, you must also deploy the type library that defines the interface.
- **Instancing:** Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them. For information on the different possibilities, see “COM object instancing types” on page 36-5.
- **Threading Model:** Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on the different possibilities, see “Choosing a threading model” on page 36-6. For information on how to provide thread support to your application, see Chapter 9, “Writing multi-threaded applications.”

- **Type Library:** You can choose whether you want to include a type library for your object. This is recommended for two reasons: it lets you use the Type Library editor to define interfaces, thereby updating much of the implementation, and it gives clients an easy way to obtain information about your object and its interfaces. If you are implementing an existing interface, Delphi requires your project to use a type library. This is the only way to provide access to the original interface declaration. For information on type libraries, see “Type libraries” on page 33-15 and Chapter 34, “Working with type libraries”.
- **Marshaling:** If you have opted to create a type library and are willing to confine yourself to Automation-compatible types, you can let COM handle the marshaling for you when you are not generating an in-process server. By marking your object’s interface as *OleAutomation* in the type library, you enable COM to set up the proxies and stubs for you and handles passing parameters across process boundaries. For more information on this process, see “The marshaling mechanism” on page 33-8. You can only specify whether your interface is Automation-compatible if you are generating a new interface. If you select an existing interface, its attributes are already specified in its type library. If your object’s interface is not marked as *OleAutomation*, you must either create an in-process server or write your own marshaling code.

You can optionally add a description of your COM object. This description appears in the type library for your object if you create one.

## Using the Automation object wizard

---

The Automation object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from *TAutoObject* and sets up the class factory constructor. For more information on the base class, see “Code generated by wizards” on page 33-21.
- Adds a type library to your project and adds your object and its interface to the type library.

Before you create an Automation object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.

To display the Automation wizard:

- 1 Choose File | New | Other.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the Automation Object icon.

In the wizard dialog, specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. Your object's default interface is created with a name based on this CoClass name with an 'I' prepended, and the class created to implement your object has this name with a 'T' prepended.
- **Instancing:** Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them. For information on the different possibilities, see "COM object instancing types" on page 36-5.
- **Threading Model:** Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on the different possibilities, see "Choosing a threading model" on page 36-6. For information on how to provide thread support to your application, see Chapter 9, "Writing multi-threaded applications."
- **Event support:** You must indicate whether you want your object to generate events to which clients can respond. The wizard can provide support for the interfaces required to generate events and the dispatching of calls to client event handlers. For information on how events work and what you need to do when implementing them, see "Exposing events to clients" on page 36-10.

You can optionally add a description of your COM object. This description appears in the type library for your object.

The Automation object implements a **dual interface**, which supports both early (compile-time) binding through the *VTable* and late (runtime) binding through the *IDispatch* interface. For more information, see "Dual interfaces" on page 36-13.

## COM object instancing types

---

Many of the COM wizards require you to specify an instancing mode for the object. Instancing determines how many instances of your object clients can create in a single executable. If you specify a Single Instance model, for example, then COM once a client has instantiated your object, COM removes the application from view so that other clients must launch their own instances of the application. Because this affects the visibility of your application as a whole, the instancing mode must be consistent across all objects in your application that can be instantiated by clients. That is, you should not create one object in your application that uses Single Instance mode and another in the same application that uses Multiple Instance mode.

**Note** Instancing is ignored when your COM object is used only as an in-process server.

When the wizard creates a new COM object, it can have any of the following instancing types:

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly, although your application can create the object and pass an interface for it to clients.
Single Instance	Allows clients to create only a single instance of the object for each executable (application), so creating multiple instances results in launching multiple instances of the application. Each client has its own dedicated instance of the server application. This option is commonly used for multiple document interface (MDI) applications.
Multiple Instances	Specifies that multiple clients can create instances of the object in the same process space. Any time a client requests service, a separate instance of the object is created. (That is, there can be multiple instances in a single executable.)

## Choosing a threading model

When creating an object using a wizard, you select a threading model that your object agrees to support. By adding thread support to your COM object, you can improve its performance, because multiple clients can access your application at the same time.

Table 36.1 lists the different threading models you can specify.

**Table 36.1** Threading models for COM objects

Threading model	Description	Implementation pros and cons
Single	The server provides no thread support. COM serializes client requests so that the application receives one request at a time.	Clients are handled one at a time so no threading support is needed. No performance benefit.
Apartment (or Single-threaded apartment)	COM ensures that only one client thread can call the object at a time. All client calls use the thread in which the object was created.	Objects can safely access their own instance data, but global data must be protected using critical sections or some other form of serialization. The thread's local variables are reliable across multiple calls. Some performance benefits.
Free (also called multi-threaded apartment)	Objects can receive calls on any number of threads at any time.	Objects must protect all instance and global data using critical sections or some other form of serialization. Thread local variables are <i>not</i> reliable across multiple calls.

**Table 36.1** Threading models for COM objects (continued)

Threading model	Description	Implementation pros and cons
Both	This is the same as the Free-threaded model except that outgoing calls (for example, callbacks) are guaranteed to execute in the same thread.	<p>Maximum performance and flexibility.</p> <p>Does not require the application to provide thread support for parameters supplied to outgoing calls.</p>
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict.	<p>You must guard against thread conflicts involving global data and any instance data that is accessed by multiple methods.</p> <p>This model should not be used with objects that have a user interface (visual controls).</p> <p>This model is only available under COM+. Under COM, it is mapped to the Apartment model.</p>

**Note** Local variables (except those in callbacks) are always safe, regardless of the threading model. This is because local variables are stored on the stack and each thread has its own stack. Local variables may not be safe in callbacks when using free-threading.

The threading model you choose in the wizard determines how the object is registered in the system Registry. You must make sure that your object implementation adheres to the threading model you have chosen. For general information on writing thread-safe code, see Chapter 9, "Writing multi-threaded applications."

For in-process servers, setting the threading model in the wizard sets the threading model key in the CLSID registry entry.

Out-of-process servers are registered as EXE, and Delphi initializes COM for the highest threading model required. For example, if an EXE includes a free-threaded object, it is initialized for free threading, which means that it can provide the expected support for any free-threaded or apartment-threaded objects contained in the EXE. To manually override threading behavior in EXEs, use the `CoInitFlags` variable, which is described in the online help.

### Writing an object that supports the free threading model

Use the free threading (or both) model rather than apartment threading whenever the object needs to be accessed from more than one thread. A common example is a client application connected to an object on a remote machine. When the remote client calls a method on that object, the server receives the call on a thread from the thread pool on the server machine. This receiving thread makes the call locally to the actual object; and, because the object supports the free threading model, the thread can make a direct call into the object.

If the object supported the apartment threading model instead, the call would have to be transferred to the thread on which the object was created, and the result would



have to be transferred back into the receiving thread before returning to the client. This approach requires extra marshaling.

To support free threading, you must consider how instance data can be accessed for *each* method. If the method is writing to instance data, you must use critical sections or some other form of serialization, to protect the instance data. Likely, the overhead of serializing critical calls is less than executing COM's marshaling code.

Note that if the instance data is read-only, serialization is not needed.

Free-threaded in-process servers can improve performance by acting as the outer object in an aggregation with the free-threaded marshaler. The free-threaded marshaler provides a shortcut for COM's standard thread handling when a free-threaded DLL is called by a host (client) that is not free-threaded.

To aggregate with the free threaded marshaler, you must

- Call `CoCreateFreeThreadedMarshaler`, passing your object's `IUnknown` interface for the resulting free-threaded marshaler to use:

```
CoCreateFreeThreadedMarshaler(self as IUnknown, FMarshaler);
```

This line assigns the interface for the free-threaded marshaler to a class member, `FMarshaler`.

- Using the Type Library editor, add the `IMarshal` interface to the set of interfaces your `CoClass` implements.
- In your object's `QueryInterface` method, delegate calls for `IDD_IMarshal` to the free-threaded marshaler (stored as `FMarshaler` above).

**Warning** The free-threaded marshaler violates the normal rules of COM marshaling to provide additional efficiency. It should be used with care. In particular, it should only be aggregated with free-threaded objects in in-process servers, and should only be instantiated by the object that uses it (not another thread).

## Writing an object that supports the apartment threading model

To implement the (single-threaded) apartment threading model, you must follow a few rules:

- The first thread in the application that gets created is COM's main thread. This is typically the thread on which `WinMain` was called. This must also be the last thread to uninitialize COM.
- Each thread in the apartment threading model must have a message loop, and the message queue must be checked frequently.
- When a thread gets a pointer to a COM interface, that pointer may only be used in that thread.

The single-threaded apartment model is the middle ground between providing no threading support and full, multi-threading support of the free threading model. A server committing to the apartment model promises that the server has serialized access to all of its global data (such as its object count). This is because different objects may try to access the global data from different threads. However, the object's instance data is safe because the methods are always called on the same thread.

Typically, controls for use in Web browsers use the apartment threading model because browser applications always initialize their threads as apartment.

### Writing an object that supports the neutral threading model

Under COM+, you can use another threading model that is in between free threading and apartment threading: the neutral model. Like the free-threading model, this model allows multiple threads to access your object at the same time. There is no extra marshaling to transfer to the thread on which the object was created. However, your object is guaranteed to receive no conflicting calls.

Writing an object that uses the neutral threading model follows much the same rules as writing an apartment-threaded object, except that you do need to guard instance data against thread conflicts if it can be accessed by different methods in the object's interface. Any instance data that is only accessed by a single interface method is automatically thread-safe.

## Defining a COM object's interface

---

When you use a wizard to create a COM object, the wizard automatically generates a type library (unless you specify otherwise in the COM object wizard). The type library provides a way for host applications to find out what the object can do. It also lets you define your object's interface using the Type Library editor. The interfaces you define in the Type Library editor define what properties, methods, and events your object exposes to clients.

**Note** If you selected an existing interface in the COM object wizard, you do not need to add properties and methods. The definition of the interface is imported from the type library in which it was defined. Instead, simply locate the methods of the imported interface in the implementation unit and fill in their bodies.

### Adding a property to the object's interface

---

When you add a property to your object's interface using the Type Library editor, it automatically adds a method to read the property's value and/or a method to set the property's value. The Type Library editor, in turn, adds these methods to your implementation class, and in your implementation unit creates empty method implementations for you to complete.

To add a property to your object's interface,

- 1 In the type library editor, select the default interface for the object.

The default interface should be the name of the object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."

- 2 To expose a read/write property, click the Property button on the toolbar; otherwise, click the arrow next to the Property button on the toolbar, and then click the type of property to expose.
- 3 In the Attributes pane, specify the name and type of the property.
- 4 On the toolbar, click the Refresh button.  
A definition and skeletal implementations for the property access methods are inserted into the object's implementation unit.
- 5 In the implementation unit, locate the access methods for the property. These have names of the form `Get_PropertyName` and `Set_PropertyName`. Add code that gets or sets the property value of your object. This code may simply call an existing function inside the application, access a data member that you add to the object definition, or otherwise implement the property.

## Adding a method to the object's interface

---

When you add a method to your object's interface using the Type Library editor, the Type Library editor can, in turn, add the methods to your implementation class, and in your implementation unit create empty implementation for you to complete.

To expose a method via your object's interface,

- 1 In the Type Library editor, select the default interface for the object.  
The default interface should be the name of the object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."
- 2 Click the Method button.
- 3 In the Attributes pane, specify the name of the method.
- 4 In the Parameters pane, specify the method's return type and add the appropriate parameters.
- 5 On the toolbar, click the Refresh button.  
A definition and skeletal implementation for the method is inserted into the object's implementation unit.
- 6 In the implementation unit, locate the newly inserted method implementation. The method is completely empty. Fill in the body to perform whatever task the method represents.

## Exposing events to clients

---

There are two types of events that a COM object can generate: traditional events and COM+ events.

- COM+ events require that you create a separate event object using the event object wizard and add code to call that event object from your server object. For more

information about generating COM+ events, see "Generating events under COM+" on page 39-18.

- You can use the wizard to handle much of the work in generating traditional events. This process is described below.

**Note** The COM object wizard does not generate event support code. If you want your object to generate traditional events, you should use the Automation object wizard.

In order for an object to generate events, you need to do the following:

- 1 In the Automation wizard, check the box, Generate event support code.

The wizard creates an object that includes an Events interface as well as the default interface. This Events interface has a name of the form *ICoClassnameEvents*. It is an outgoing (source) interface, which means that it is not an interface your object implements, but rather is an interface that clients must implement and which your object calls. (You can see this by selecting your CoClass, going to the Implements page, and noting that the Source column on the Events interface says *True*.)

In addition to the Events interface, the wizard adds the *IConnectionPointContainer* interface to the declaration of your implementation class, and adds several class members for handling events. Of these new class members, the most important are *FConnectionPoint* and *FConnectionPoints*, which implement the *IConnectionPoint* and *IConnectionPointContainer* interfaces using built-in VCL classes. *FConnectionPoint* is maintained by another method that the wizard adds, *EventSinkChanged*.

- 2 In the Type Library editor, select the outgoing Events interface for your object. (This is the one with a name of the form *ICoClassNameEvents*)
- 3 Click the Method button from the Type Library toolbar. Each method you add to the Events interface represents an event handler that the client must implement.
- 4 In the Attributes pane, specify the name of the event handler, such as *MyEvent*.
- 5 On the toolbar, click the Refresh button.

Your object implementation now has everything it needs to accept client event sinks and maintain a list of interfaces to call when the event occurs. To call these interfaces, you can create a method to generate each event on clients.

- 6 In the Code Editor, add a method to your object for firing each event. For example,

```
unit ev;
interface
uses
  ComObj, AxCtrls, ActiveX, Project1_TLB;
type
  TMyAutoObject = class (TAutoObject, IConnectionPointContainer, IMyAutoObject)
  private
    .
    .
    .
  public
    procedure Initialize; override;
    procedure Fire_MyEvent; { Add a method to fire the event}
```

- 7 Implement the method you added in the last step so that it iterates through all the event sinks maintained by your object's *FConnectionPoint* member:

```

procedure TMyAutoObject.Fire_MyEvent;
var
  I: Integer;
  EventSinkList: TList;
  EventSink: IMyAutoObjectEvents;
begin
  if FConnectionPoint <> nil then
  begin
    EventSinkList := FConnectionPoint.SinkList; {get the list of client sinks }
    for I := 0 to EventSinkList.Count - 1 do
    begin
      EventSink := IUnknown(FEvents[I]) as IMyAutoObjectEvents;
      EventSink.MyEvent;
    end;
  end;
end;

```

- 8 Whenever you need to fire the event so that clients are informed of its occurrence, call the method that dispatches the event to all event sinks:

```

if EventOccurs then Fire_MyEvent; { Call method you created to fire events.}

```

## Managing events in your Automation object

For a server to support traditional COM events, it must provide the definition of an outgoing interface which is implemented by a client. This outgoing interface includes all the event handlers the client must implement to respond to server events.

When a client has implemented the outgoing event interface, it registers its interest in receiving event notification by querying the server's *IConnectionPointContainer* interface. The *IConnectionPointContainer* interface returns the server's *IConnectionPoint* interface, which the client then uses to pass the server a pointer to its implementation of the event handlers (known as a sink).

The server maintains a list of all client sinks and calls methods on them when an event occurs, as described above.

## Automation interfaces

---

The Automation Object wizard implements a dual interface by default, which means that the Automation object supports both

- Late binding at runtime, which is through the *IDispatch* interface. This is implemented as a dispatch interface, or **dispinterface**.
- Early binding at compile-time, which is accomplished through directly calling one of the member functions in the object's virtual function table (VTable). This is referred to as a **custom interface**.

**Note** Any interfaces generated by the COM object wizard that do not descend from *IDispatch* only support VTable calls.

## Dual interfaces

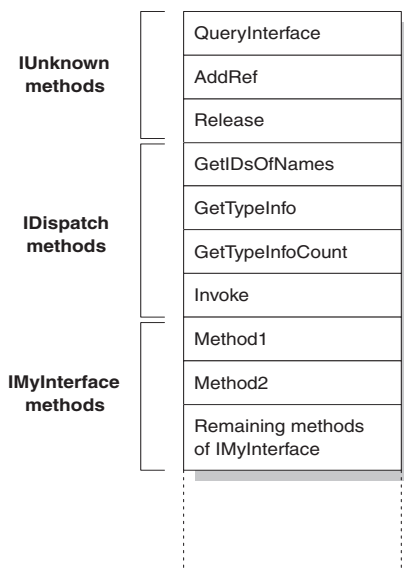
A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch*. For those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time binding, the more efficient VTable interface is used.

Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces:

- For VTable interfaces, the compiler performs type checking and provides more informative error messages.
- For Automation controllers that cannot obtain type information, the dispinterface provides runtime access to the object.
- For in-process servers, you have the benefit of fast access through VTable interfaces.
- For out-of-process servers, COM marshals data for both VTable interfaces and dispinterfaces. COM provides a generic proxy/stub implementation that can marshal the interface based on the information contained in a type library. For more information on marshaling, see, "Marshaling data," on page 36-15.

The following diagram depicts the *IMyInterface* interface in an object that supports a dual interface named *IMyInterface*. The first three entries of the VTable for a dual interface refer to the *IUnknown* interface, the next four entries refer to the *IDispatch* interface, and the remaining entries are COM entries for direct access to members of the custom interface.

**Figure 36.1** Dual interface VTable



## Dispatch interfaces

---

Automation controllers are clients that use the COM *IDispatch* interface to access the COM server objects. The controller must first create the object, then query the object's *IUnknown* interface for a pointer to its *IDispatch* interface. *IDispatch* keeps track of methods and properties internally by a dispatch identifier (dispID), which is a unique identification number for an interface member. Through *IDispatch*, a controller retrieves the object's type information for the dispatch interface and then maps interface member names to specific dispIDs. These dispIDs are available at runtime, and controllers get them by calling the *IDispatch* method, *GetIDsOfNames*.

Once it has the dispID, the controller can then call the *IDispatch* method, *Invoke*, to execute the appropriate code (property or method), packaging the parameters for the property or method into one of the *Invoke* parameters. *Invoke* has a fixed compile-time signature that allows it to accept any number of arguments when calling an interface method.

The Automation object's implementation of *Invoke* must then unpackage the parameters, call the property or method, and be prepared to handle any errors that occur. When the property or method returns, the object passes its return value back to the controller.

This is called late binding because the controller binds to the property or method at runtime rather than at compile time.

**Note** When importing a type library, Delphi will query for dispIDs at the time it generates the code, thereby allowing generated wrapper classes to call *Invoke* without calling *GetIDsOfNames*. This can significantly increase the runtime performance of controllers.

## Custom interfaces

---

Custom interfaces are user-defined interfaces that allow clients to invoke interface methods based on their order in the VTable and knowledge of the argument types. The VTable lists the addresses of all the properties and methods that are members of the object, including the member functions of the interfaces that it supports. If the object does not support *IDispatch*, the entries for the members of the object's custom interfaces immediately follow the members of *IUnknown*.

If the object has a type library, you can access the custom interface through its VTable layout, which you can get using the Type Library editor. If the object has a type library and also supports *IDispatch*, a client can also get the dispIDs of the *IDispatch* interface and bind directly to a VTable offset. Delphi's type library importer (TLIBIMP) retrieves dispIDs at import time, so clients that use dispinterfaces can avoid calls to *GetIDsOfNames*; this information is already in the `_TLB` unit. However, clients still need to call *Invoke*.

## Marshaling data

---

For out-of-process and remote servers, you must consider how COM marshals data outside the current process. You can provide marshaling:

- Automatically, using the *IDispatch* interface.
- Automatically, by creating a type library with your server and marking the interface with the OLE Automation flag. COM knows how to marshal all the **Automation-compatible** types in the type library and can set up the proxies and stubs for you. Some type restrictions apply to enable automatic marshaling.
- Manually by implementing all the methods of the *IMarshal* interface. This is called **custom marshaling**.

**Note** The first method (using *IDispatch*) is only available on Automation servers. The second method is automatically available on all objects that are created by wizards and which use a type library.

### Automation compatible types

---

Function result and parameter types of the methods declared in dual and dispatch interfaces and interfaces that you mark as OLE Automation must be *Automation-compatible* types. The following types are OLE Automation-compatible:

- The predefined valid types such as *Smallint*, *Integer*, *Single*, *Double*, *WideString*. For a complete list, see “Valid types” on page 34-11.
- Enumeration types defined in a type library. OLE Automation-compatible enumeration types are stored as 32-bit values and are treated as values of type *Integer* for purposes of parameter passing.
- Interface types defined in a type library that are OLE Automation safe, that is, derived from *IDispatch* and containing only OLE Automation compatible types.
- Dispinterface types defined in a type library.
- Any custom record type defined within the type library.
- *IFont*, *IStrings*, and *IPicture*. Helper objects must be instantiated to map
  - an *IFont* to a *TFont*
  - an *IStrings* to a *TStrings*
  - an *IPicture* to a *TPicture*

The ActiveX control and ActiveForm wizards create these helper objects automatically when needed. To use the helper objects, call the global routines, *GetOleFont*, *GetOleStrings*, *GetOlePicture*, respectively.



## Type restrictions for automatic marshaling

---

For an interface to support automatic marshaling (also called Automation marshaling or type library marshaling), the following restrictions apply. When you edit your object using the type library editor, the editor enforces these restrictions:

- Types must be compatible for cross-platform communication. For example, you cannot use data structures (other than implementing another property object), unsigned arguments, AnsiStrings, and so on.
- String data types must be transferred as wide strings (BSTR). PChar and AnsiString cannot be marshaled safely.
- All members of a dual interface must pass an HRESULT as the function's return value. If the method is declared using the safecall calling convention, this condition is imposed automatically, with the declared return type converted to an output parameter.
- Members of a dual interface that need to return other values should specify these parameters as **var** or **out**, indicating an output parameter that returns the value of the function.

**Note** One way to bypass the Automation types restrictions is to implement a separate *IDispatch* interface and a custom interface. By doing so, you can use the full range of possible argument types. This means that COM clients have the option of using the custom interface, which Automation controllers can still access. In this case, though, you must implement the marshaling code manually.

## Custom marshaling

---

Typically, you use automatic marshaling in out-of-process and remote servers because it is easier—COM does the work for you. However, you may decide to provide custom marshaling if you think you can improve marshaling performance. When implementing your own custom marshaling, you must support the *IMarshal* interface. For more information, on this approach, see the Microsoft documentation.

## Registering a COM object

---

You can register your server object as an in-process or an out-of-process server. For more information on the server types, see “In-process, out-of-process, and remote servers” on page 33-6.

**Note** Before you remove a COM object from your system, you should unregister it.

## Registering an in-process server

---

To register an in-process server (DLL or OCX),

- Choose Run | Register ActiveX Server.

To unregister an in-process server,

- Choose Run | Unregister ActiveX Server.

## Registering an out-of-process server

---

To register an out-of-process server,

- Run the server with the **/regserver** command-line option.

You can set command-line options with the Run | Parameters dialog box.

You can also register the server by running it.

To unregister an out-of-process server,

- Run the server with the **/unregserver** command-line option.

As an alternative, you can use the **regsvr** command from the command line or run the **regsvr32.exe** from the operating system.

**Note** If the COM server is intended for use under COM+, you should install it in a COM+ application rather than register it. (Installing the object in a COM+ application automatically takes care of registration.) For information on how to install an object in a COM+ application, see “Installing transactional objects” on page 39-22.

## Testing and debugging the application

---

To test and debug your COM server application,

- 1 Turn on debugging information using the Compiler tab on the Project | Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools | Debugger Options dialog.
- 2 For an in-process server, choose Run | Parameters, type the name of the Automation controller in the Host Application box, and choose OK.
- 3 Choose Run | Run.
- 4 Set breakpoints in the Automation server.
- 5 Use the Automation controller to interact with the Automation server.

The Automation server pauses when the breakpoints are reached.

**Note** As an alternate approach, if you are also writing the Automation controller, you can debug into an in-process server by enabling COM cross-process support. Use the General page of the Tools | Debugger Options dialog to enable cross-process support.



## Creating an Active Server Page

If you are using the Microsoft Internet Information Server (IIS) environment to serve your Web pages, you can use Active Server Pages (ASP) to create dynamic Web-based client-server applications. Active Server Pages let you write a script that gets called every time the server loads the Web page. This script can, in turn, call on Automation objects to obtain information that it includes in a generated HTML page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and use this control to access data that gets updated every time the server loads the Web page.

On the client side, the ASP acts like a standard HTML document and can be viewed by users on any platform using any Web Browser.

ASP applications are analogous to applications you write using Delphi's Web broker technology. For more information about the Web broker technology, see Chapter 27, "Creating Internet applications". ASP differs, however, in the way it separates the UI design from the implementation of business rules or complex application logic.

- The UI design is managed by the Active Server Page. This is essentially an HTML document, but it can include embedded script that calls on Active Server objects to supply it with content that reflects your business rules or application logic.
- The application logic is encapsulated by Active Server objects that expose simple methods to the Active Server Page, supplying it with the content it needs.

**Note** Although ASP provides the advantage of separating UI design from application logic, its performance is limited in scale. For Web sites that respond to extremely large numbers of clients, an approach based on the Web broker technology is recommended instead.

The script in your Active Server Pages and the Automation objects you embed in an active server page can make use of the ASP intrinsics (built-in objects that provide information about the current application, HTTP messages from the browser, and so on).

This chapter shows how to create an Active Server Object using the Delphi Active Server Object wizard. This special Automation control can then be called by an Active Server Page and supply it with content.

Here are the steps for creating an Active Server Object:

- Create an Active Server Object for the application.
- Define the Active Server Object's interface.
- Register the Active Server Object.
- Test and debug the application.

## Creating an Active Server Object

---

An Active Server Object is an Automation object that has access to information about the entire ASP application and the HTTP messages it uses to communicate with browsers. It descends from *TASPObj* or *TASPMTSObj* (which is in turn a descendant of *TAutoObj*), and supports Automation protocols, exposing itself for other applications (or the script in the Active Server page) to use. You create an Active Server Object using the Active Server Object wizard.

Your Active Server Object project can be either an executable (exe) or library (dll), depending on your needs. However, you should be aware of the drawbacks of using an out-of-process server. These drawbacks are discussed in "Creating ASPs for in-process or out-of-process servers" on page 37-7.

To display the Active Server Object wizard:

- 1 Choose File | New | Other.
- 2 Select the tab labeled, ActiveX.
- 3 Double-click the Active Server Object icon.

In the wizard, give your new Active Server Object a name, and specify the instancing and threading models you want to support. These details influence the way your object can be called. You must write the implementation so that it adheres to the model (for example, avoiding thread conflicts). The instancing and threading models involve the same choices that you make for other COM objects. For details, see "COM object instancing types" on page 36-5 and "Choosing a threading model" on page 36-6.

The thing that makes an Active Server Object unique is its ability to access information about the ASP application and the HTTP messages that pass between the Active Server page and client Web browsers. This information is accessed using the ASP intrinsics. In the wizard, you can specify how your object accesses these by setting the Active Server Type:

- If you are working with IIS 3 or IIS 4, you use Page Level Event Methods. Under this model, your object implements the methods, *OnStartPage* and *OnEndPage*, which are called when the Active Server page loads and unloads. When your object is loaded, it automatically obtains an *IScriptingContext* interface, which it

uses to access the ASP intrinsics. These interfaces are, in turn, surfaced as properties inherited from the base class (*TASPObject*).

- If you are working with IIS5 or later, you use the Object Context type. Under this model, your object fetches an *IObjectContext* interface, which it uses to access the ASP intrinsics. Again, these interfaces are surfaced as properties in the inherited base class (*TASPMTSObject*). One advantage of this latter approach is that your object has access to all of the other services available through *IObjectContext*. To access the *IObjectContext* interface, simply call *GetObjectContext* (defined in the mtz unit) as follows:

```
ObjectContext := GetObjectContext;
```

For more information about the services available through *IObjectContext*, see Chapter 39, “Creating MTS or COM+ objects”.

You can tell the wizard to generate a simple ASP page to host your new Active Server Object. The generated page provides a minimal script (written in VBScript) that creates your Active Server Object based on its ProgID, and indicates where you can call its methods. This script calls **Server.CreateObject** to launch your Active Server Object.

**Note** Although the generated test script uses VBScript, Active Server Pages also can be written using Jscript.

When you exit the wizard, a new unit is added to the current project that contains the definition for the Active Server Object. In addition, the wizard adds a type library project and opens the Type Library editor. Now you can expose the properties and methods of the interface through the type library as described in “Defining a COM object’s interface” on page 36-9. As you write the implementation of your object’s properties and methods, you can take advantage of the ASP intrinsics (described below) to obtain information about the ASP application and the HTTP messages it uses to communicate with browsers.

The Active Server Object, like any other Automation object, implements a **dual interface**, which supports both early (compile-time) binding through the *VTable* and late (runtime) binding through the *IDispatch* interface. For more information on dual interfaces, see “Dual interfaces” on page 36-13.

## Using the ASP intrinsics

---

The ASP intrinsics are a set of COM objects supplied by ASP to the objects running in an Active Server Page. They let your Active Server Object access information that reflects the messages passing between your application and the Web browser, as well as a place to store information that is shared among Active Server Objects that belong to the same ASP application.

To make these objects easy to access, the base class for your Active Server Object surfaces them as properties. For a complete understanding of these objects, see the Microsoft documentation. However, the following topics provide a brief overview.

## Application

The Application object is accessed through an *ApplicationObject* interface. It represents the entire ASP application, which is defined as the set of all .asp files in a virtual directory and its subdirectories. The Application object can be shared by multiple clients, so it includes locking support that you should use to prevent thread conflicts.

*ApplicationObject* includes the following:

**Table 37.1** IApplicationObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the application using script commands. This interface has two methods, <i>Remove</i> and <i>RemoveAll</i> , that you can use to delete one or all objects from the list.
StaticObjects property	Lists all the objects that were added to the application with the <OBJECT> tag.
Lock method	Prevents other clients from locking the Application object until you call <i>Unlock</i> . All clients should call <i>Lock</i> before accessing shared memory (such as the properties).
Unlock method	Releases the lock that was set using the <i>Lock</i> method.
Application_OnEnd event	Occurs when the application quits, after the <i>Session_OnEnd</i> event. The only intrinsics available are <i>Application</i> and <i>Server</i> . The event handler must be written in VBScript or JScript.
Application_OnStart event	Occurs before the new session is created (before <i>Session_OnStart</i> ). The only intrinsics available are <i>Application</i> and <i>Server</i> . The event handler must be written in VBScript or JScript.

## Request

The Request object is accessed through an *IRequest* interface. It provides information about the HTTP request message that caused the Active Server Page to be opened.

*IRequest* includes the following:

**Table 37.2** IRequest interface members

Property, Method, or Event	Meaning
ClientCertificate property	Indicates the values of all fields in the client certificate that is sent with the HTTP message.
Cookies property	Indicates the values of all Cookie headers on the HTTP message.
Form property	Indicates the values of form elements in the HTTP body. These can be accessed by name.
QueryString property	Indicates the values of all variables in the query string from the HTTP header.
ServerVariables property	Indicates the values of various environment variables. These variables represent most of the common HTTP header variables.
TotalBytes property	Indicates the number of bytes in the request body. This is an upper limit on the number of bytes returned by the <i>BinaryRead</i> method.
BinaryRead method	Retrieves the content of a Post message. Call the method, specifying the maximum number of bytes to read. The resulting content is returned as a Variant array of bytes. After calling <i>BinaryRead</i> , you can't use the <i>Form</i> property.

## Response

The Request object is accessed through an *IResponse* interface. It lets you specify information about the HTTP response message that is returned to the client browser.

*IResponse* includes the following:

**Table 37.3** IResponse interface members

Property, Method, or Event	Meaning
Cookies property	Determines the values of all Cookie headers on the HTTP message.
Buffer property	Indicates whether page output is buffered. When page output is buffered, the server does not send a response to the client until all of the server scripts on the current page are processed.
CacheControl property	Determines whether proxy servers can cache the output in the response.
Charset property	Adds the name of the character set to the content type header.
ContentType property	Specifies the HTTP content type of the response message's body.
Expires property	Specifies how long the response can be cached by a browser before it expires.
ExpiresAbsolute property	Specifies the date and time when the response expires.
IsClientConnected property	Indicates whether the client has disconnected from the server.
Pics property	Set the value for the pics-label field of the response header.
Status property	Indicates the status of the response. This is the value of an HTTP status header.
AddHeader method	Adds an HTTP header with a specified name and value.
AppendToLog method	Adds a string to the end of the Web server log entry for this request.
BinaryWrite method	Writes raw (uninterpreted) information to the body of the response message.
Clear method	Erases any buffered HTML output.
End method	Stops processing the .asp file and returns the current result.
Flush method	Sends any buffered output immediately.
Redirect method	Sends a redirect response message, redirecting the client browser to a different URL.
Write method	Writes a variable to the current HTTP output as a string.

## Session

The Session object is accessed through the *ISessionObject* interface. It allows you to store variables that persist for the duration of a client's interaction with the ASP application. That is, these variables are not freed when the client moves from page to page within the ASP application, but only when the client exits the application altogether.



*ISessionObject* includes the following:

**Table 37.4** ISessionObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the session using the <OBJECT> tag. You can access any variable in the list by name, or call the Contents object's <i>Remove</i> or <i>RemoveAll</i> method to delete values.
StaticObjects property	Lists all the objects that were added to the session with the <OBJECT> tag.
CodePage property	Specifies the code page to use for symbol mapping. Different locales may use different code pages.
LCID property	Specifies the locale identifier to use for interpreting string content.
SessionID property	Indicates the session identifier for the current client.
Timeout property	Specifies the time, in minutes, that the session persists without a request (or refresh) from the client until the application terminates.
Abandon method	Destroys the session and releases its resources.
Session_OnEnd event	Occurs when the session is abandoned or times out. The only intrinsics available are <i>Application</i> , <i>Server</i> , and <i>Session</i> . The event handler must be written in VBScript or JScript.
Session_OnStart event	Occurs when the server creates a new session is created (after <i>Application_OnStart</i> but before running the script on the Active Server Page). All intrinsics are available. The event handler must be written in VBScript or JScript.

## Server

The Server object is accessed through an *IServer* interface. It provides various utilities for writing your ASP application.

*IServer* includes the following:

**Table 37.5** IServer interface members

Property, Method, or Event	Meaning
ScriptTimeout property	Same as the Timeout property on the Session object.
CreateObject method	Instantiates a specified Active Server Object.
Execute method	Executes the script in a specified .asp file.
GetLastError method	Returns an ASPError object that describes the error condition.
HTMLEncode method	Encodes a string for use in an HTML header, replacing reserved characters by the appropriate symbolic constants.
MapPath method	Maps a specified virtual path (an absolute path on the current server or a path relative to the current page) into a physical path.
Transfer method	Sends all of the current state information to another Active Server Page for processing.
URLEncode method	Applies URL encoding rules, including escape characters, to a specified string

## Creating ASPs for in-process or out-of-process servers

---

You can use `Server.CreateObject` in an ASP page to launch either an in-process or out-of-process server, depending on your requirements. However, launching in-process servers is more common.

Unlike most in-process servers, an Active Server Object in an in-process server does not run in the client's process space. Instead, it runs in the IIS process space. This means that the client does not need to download your application (as, for example, it does when you use ActiveX objects). In-process component DLLs are faster and more secure than out-of-process servers, so they are better suited for server-side use.

Because out-of-process servers are less secure, it is common for IIS to be configured to *not* allow out-of-process executables. In this case, creating an out-of-process server for your Active Server Object would result in an error similar to the following:

```
Server object error 'ASP 0196'  
Cannot launch out of process component  
/path/outofprocess_exe.asp, line 11
```

Also, out-of-process components often create individual server processes for each object instance, so they are slower than CGI applications. They do not scale as well as component DLLs.

If performance and scalability are priorities for your site, in-process servers are highly recommended. However, Intranet sites that receive moderate to low traffic may use an out-of-process component without adversely affecting the site's overall performance.

For general information on in-process and out-of-process servers, see, "In-process, out-of-process, and remote servers," on page 33-6.

## Registering an Active Server Object

---

You can register the Active Server Page as an in-process or an out-of-process server. However, in-process servers are more common.

**Note** When you want to remove the Active Server Page object from your system, you should first unregister it, removing its entries from the Windows registry.

### Registering an in-process server

---

To register an in-process server (DLL or OCX),

- Choose Run | Register ActiveX Server.

To unregister an in-process server,

- Choose Run | Unregister ActiveX Server.

## Registering an out-of-process server

---

To register an out-of-process server,

- Run the server with the /regserver command-line option. (You can set command-line options with the Run | Parameters dialog box.)

You can also register the server by running it.

To unregister an out-of-process server,

- Run the server with the /unregserver command-line option.

## Testing and debugging the Active Server Page application

---

Debugging any in-process server such as an Active Server Object is much like debugging a DLL. You choose a host application that loads the DLL, and debug as usual. To test and debug an Active Server Object,

- 1 Turn on debugging information using the Compiler tab on the Project | Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools | Debugger Options dialog.
- 2 Choose Run | Parameters, type the name of your Web Server in the Host Application box, and choose OK.
- 3 Choose Run | Run.
- 4 Set breakpoints in the Active Server Object implementation.
- 5 Use the Web browser to interact with the Active Server Page.

The debugger pauses when the breakpoints are reached.

## Creating an ActiveX control

An ActiveX control is a software component that integrates into and extends the functionality of any host application that supports ActiveX controls, such as C++Builder, Delphi, Visual Basic, Internet Explorer, and (given a plug-in) Netscape Navigator. ActiveX controls implement a particular set of interfaces that allow this integration.

For example, Delphi comes with several ActiveX controls, including charting, spreadsheet, and graphics controls. You can add these controls to the component palette in the IDE, and then use them like any standard VCL component, dropping them on forms and setting their properties using the Object Inspector.

An ActiveX control can also be deployed on the Web, allowing it to be referenced in HTML documents and viewed with ActiveX-enabled Web browsers.

Delphi provides wizards that let you create two types of ActiveX controls:

- **ActiveX controls that wrap VCL classes.** By wrapping a VCL class, you can convert existing components into ActiveX controls or create new ones, test them out locally, and then convert them into ActiveX controls. ActiveX controls are typically intended to be embedded in a larger host application.
- **Active forms.** Active forms let you use the form designer to create a more elaborate control that acts like a dialog or like a complete application. You develop the Active form in much the same way that you develop a typical Delphi application. Active Forms are typically intended for deployment on the Web.

This chapter provides an overview of how to create an ActiveX control in the Delphi environment. It is not intended to provide complete implementation details of writing ActiveX control without using a wizard. For that information, refer to your Microsoft Developer's Network (MSDN) documentation or search the Microsoft Web site for ActiveX information.

## Overview of ActiveX control creation

---

Creating ActiveX controls using Delphi is very similar to creating ordinary controls or forms. This differs markedly from creating other COM objects, where you first define the object's interface and then complete the implementation. To create ActiveX controls (other than Active Forms), you reverse this process, starting with the implementation of a VCL control, and then generating the interface and type library once the control is written. When creating Active Forms, the interface and type library are created at the same time as your form, and then you use the form designer to implement the form.

The completed ActiveX control consists of a VCL control that provides the underlying implementation, a COM object that wraps the VCL control, and a type library that lists the COM object's properties, methods, and events.

To create a new ActiveX control (other than an Active Form), perform the following steps:

- 1 Design and create the custom VCL control that forms the basis of your ActiveX control.
- 2 Use the ActiveX control wizard to create an ActiveX control from the VCL control you created in step 1.
- 3 Use the ActiveX property page wizard to create one or more property pages for the control (optional).
- 4 Associate the property page with the ActiveX control (optional).
- 5 Register the control.
- 6 Test the control with all potential target applications.
- 7 Deploy the ActiveX control on the Web. (optional)

To create a new Active Form, perform the following steps:

- 1 Use the ActiveForm wizard to create an Active Form, which appears as a blank form in the IDE, and an associated ActiveX wrapper for that form.
- 2 Use the form designer to add components to your Active Form and implement its behavior in the same way you create and implement an ordinary form using the form designer.
- 3 Follow steps 3-7 above to give your Active Form a property page, register it, and deploy it on the Web.

### Elements of an ActiveX control

---

An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, a corresponding COM object wrapper that exposes properties, methods, and events, and one or more associated type libraries.

## VCL control

The underlying implementation of an ActiveX control in Delphi is a VCL control. When you create an ActiveX control, you must first design or choose the VCL control from which you will make your ActiveX control.

The underlying VCL control must be a descendant of *TWinControl*, because it must have a window that can be parented by the host application. When you create an Active form, this object is a descendant of *TActiveForm*.

**Note** The ActiveX control wizard lists the available *TWinControl* descendants from which you can choose to make an ActiveX control. This list does not include all *TWinControl* descendants, however. Some controls, such as *THeaderControl*, are registered as incompatible with ActiveX (using the *RegisterNonActiveX* procedure) and do not appear in the list.

## ActiveX wrapper

The actual COM object is an ActiveX wrapper object for the VCL control. For Active forms, this class is always *TActiveFormControl*. For other ActiveX controls, it has a name of the form *TVCLClassX*, where *TVCLClass* is the name of the VCL control class. Thus, for example, the ActiveX wrapper for *TButton* would be named *TButtonX*.

The wrapper class is a descendant of *TActiveXControl*, which provides support for the ActiveX interfaces. The ActiveX wrapper inherits this support, which allows it to forward Windows messages to the VCL control and parent its window in the host application.

The ActiveX wrapper exposes the VCL control's properties and methods to clients via its default interface. The wizard automatically implements most of the wrapper class's properties and methods, delegating method calls to the underlying VCL control. The wizard also provides the wrapper class with methods that fire the VCL control's events on clients and assigns these methods as event handlers on the VCL control.

## Type library

The ActiveX control wizards automatically generate a type library that contains the type definitions for the wrapper class, its default interface, and any type definitions that these require. This type information provides a way for your control to advertise its services to host applications. You can view and edit this information using the Type Library editor. Although this information is stored in a separate, binary type library file (.TLB extension), it is also automatically compiled into the ActiveX control DLL as a resource.

## Property page

You can optionally give your ActiveX control a property page. The property page allows the user of a host (client) application to view and edit your control's properties. You can group several properties on a page, or use a page to provide a dialog-like interface for a property. For information on how to create property pages, see "Creating a property page for an ActiveX control" on page 38-11.

## Designing an ActiveX control

---

When designing an ActiveX control, you start by creating a custom VCL control. This forms the basis of your ActiveX control. For information on creating custom controls, see Part V, “Creating custom components.”

When designing the VCL control, keep in mind that it will be embedded in another application; this control is not an application in itself. For this reason, you probably do not want to use elaborate dialog boxes or other major user-interface components. Your goal is typically to make a simple control that works inside of, and follows the rules of the main application.

In addition, you should make sure that the types for all properties and methods you want your object to expose to clients are Automation-compatible, because the ActiveX control’s interface must support *IDispatch*. The wizard does not add any methods to the wrapper class’s interface that have parameters that are not Automation-compatible. For a list of Automation-compatible types, see “Valid types” on page 34-11.

The wizards implement all the necessary ActiveX interfaces required using the COM wrapper class. They also surface all Automation-compatible properties, methods, and events through the wrapper class’s default interface. Once the wizard has generated the COM wrapper class and its interface, you can use the Type Library editor to modify the default interface or augment the wrapper class by implementing additional interfaces.

## Generating an ActiveX control from a VCL control

---

To generate an ActiveX control from a VCL control, use the ActiveX Control wizard. The properties, methods, and events of the VCL control become the properties, methods, and events of the ActiveX control.

Before using the ActiveX control wizard, you must decide what VCL control will provide the underlying implementation of the generated ActiveX control.

To bring up the ActiveX control wizard,

- 1 Choose File | New | Other to open the New Items dialog box.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the ActiveX Control icon.

In the wizard, select the name of the VCL control that will be wrapped by the new ActiveX control. The dialog lists all available controls, which are descendants of *TWinControl* that are not registered as incompatible with ActiveX using the *RegisterNonActiveX* procedure.

**Tip** If you do not see the control you want in the drop-down list, check whether you have installed it in the IDE or added its unit to your project.

Once you have selected a VCL control, the wizard automatically generates a name for the CoClass, the implementation unit for the ActiveX wrapper, and the ActiveX

library project. (If you currently have an ActiveX library project open, and it does not contain a COM+ event object, the current project is automatically used.) You can change any of these in the wizard (unless you have an ActiveX library project already open, in which case the project name is not editable).

The wizard always specifies Apartment as the threading model. This is not a problem if your ActiveX project usually contains only a single control. However, if you add additional objects to your project, you are responsible for providing thread support.

The wizard also lets you configure various options on your ActiveX control:

- **Enabling licensing:** You can make your control licensed to ensure that users of the control can't open it either for design purposes or at runtime unless they have a license key for the control.
- **Including Version information:** You can include version information, such as a copyright or a file description, in the ActiveX control. This information can be viewed in a browser. Some host clients, such as Visual Basic 4.0, require Version information or they will not host the ActiveX control. Specify version information by choosing Project | Options and selecting the Version Info page.
- **Including an About box:** You can tell the wizard to generate a separate form that implements an About box for your control. Users of the host application can display this About box in a development environment. By default, the About box includes the name of the ActiveX control, an image, copyright information, and an OK button. You can modify this default form, which the wizard adds to your project.

When you exit the wizard, it generates the following:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A type library, which defines and CoClass for your control, the interface it exposes to clients, and any type definitions that these require. For more information about the type library, refer to Chapter 34, "Working with type libraries."
- An ActiveX implementation unit, which defines and implements the ActiveX control, a descendant of *TActiveXControl*. This ActiveX control is a fully-functioning implementation that requires no additional work on your part. However, you can modify this class if you want to customize the properties, methods, and events that the ActiveX control exposes to clients.
- An About box form and unit if you requested them.
- A .LIC file if you enabled licensing.

## Generating an ActiveX control based on a VCL form

---

Unlike other ActiveX controls, Active Forms are not first designed and then wrapped by an ActiveX wrapper class. Instead, the ActiveForm wizard generates a blank form that you design later when the wizard leaves you in the Form Designer.



When an ActiveForm is deployed on the Web, Delphi creates an HTML page to contain the reference to the ActiveForm and specify its location on the page. The ActiveForm can then be displayed and run from a Web browser. Inside the browser, the form behaves just like a stand-alone Delphi form. The form can contain any VCL components or ActiveX controls, including custom-built VCL controls.

To start the ActiveForm wizard,

- 1 Choose File | New | Other to open the New Items dialog box.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the ActiveForm icon.

The Active Form wizard looks just like the ActiveX control wizard, except that you can't specify the name of the VCL class to wrap. This is because Active forms are always based on *TActiveForm*.

As in the ActiveX control wizard, you can change the default names for the CoClass, implementation unit, and ActiveX library project. Similarly, this wizard lets you indicate whether you want your Active Form to require a license, whether it should include version information, and whether you want an About box form.

When you exit the wizard, it generates the following:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A type library, which defines and CoClass for your control, the interface it exposes to clients, and any type definitions that these require. For more information about the type library, refer to Chapter 34, "Working with type libraries."
- A form that descends from *TActiveForm*. This form appears in the form designer, where you can use it to visually design the Active Form that appears to clients. Its implementation appears in the generated implementation unit. In the initialization section of the implementation unit, a class factory is created, setting up *TActiveFormControl* as the ActiveX wrapper for this form.
- An About box form and unit if you requested them.
- A .LIC file if you enabled licensing.

At this point, you can add controls and design the form as you like.

After you have designed and compiled the ActiveForm project into an ActiveX library (which has the OCX extension), you can deploy the project to your Web server and Delphi creates a test HTML page with a reference to the ActiveForm.

## Licensing ActiveX controls

---

Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically for controls created at runtime.

To provide design-time licenses, the ActiveX wizard creates a key for the control, which it stores in a file with the same name as the project with the LIC extension. This .LIC file is added to the project. The user of the control must have a copy of the .LIC

file to open the control in a development environment. Each control in the project that has Make Control Licensed checked has a separate key entry in the LIC file.

To support runtime licenses, the wrapper class implements two methods, *GetLicenseString* and *GetLicenseFilename*. These return the license string for the control and the name of the .LIC file, respectively. When a host application tries to create the ActiveX control, the class factory for the control calls these methods and compares the string returned by *GetLicenseString* with the string stored in the .LIC file.

Runtime licenses for the Internet Explorer require an extra level of indirection because users can view HTML source code for any Web page, and because an ActiveX control is copied to the user's computer before it is displayed. To create runtime licenses for controls used in Internet Explorer, you must first generate a license package file (LPK file) and embed this file in the HTML page that contains the control. The LPK file is essentially an array of ActiveX control CLSIDs and license keys.

**Note** To generate the LPK file, use the utility, LPK\_TOOL.EXE, which you can download from the Microsoft Web site ([www.microsoft.com](http://www.microsoft.com)).

To embed the LPK file in a Web page, use the HTML objects, <OBJECT> and <PARAM> as follows:

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
  <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

The CLSID identifies the object as a license package and PARAM specifies the relative location of the license package file with respect to the HTML page.

When Internet Explorer tries to display the Web page containing the control, it parses the LPK file, extracts the license key, and if the license key matches the control's license (returned by *GetLicenseString*), it renders the control on the page. If more than one LPK is included in a Web page, Internet Explorer ignores all but the first.

For more information, look for Licensing ActiveX Controls on the Microsoft Web site.

## Customizing the ActiveX control's interface

---

The ActiveX Control and ActiveForm wizards generate a default interface for the ActiveX wrapper class. This default interface simply exposes the properties, methods, and events of the original VCL control or form, with the following exceptions:

- Data-aware properties do not appear. Because ActiveX controls have a different mechanism for making controls data-aware than VCL controls, the wizards do not convert properties related to data. See "Enabling simple data binding with the type library" on page 38-10 for information on how to make your ActiveX control data-aware.
- Any property, method, or event that type that is not Automation-compatible does not appear. You may want to add these to the ActiveX control's interface after the wizard has finished.

You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library. You can use the Type Library editor as described in Chapter 34, "Working with type libraries." Remember that when you add events, they should be added to the Events interface, not the ActiveX control's default interface.

**Note** You can add unpublished properties to your ActiveX control's interface. Such properties can be set at runtime and will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value of a property at design time, the changes are not reflected when the control is run. If the source is a VCL object and the property is not already published, you can make properties persistent by creating a descendant of the VCL object and publishing the property in the descendant.

You may also choose not to expose all of the VCL control's properties, methods, and events to host applications. You can use the Type Library editor to remove these from the interfaces that the wizard generated. When you remove properties and methods from an interface using the Type Library editor, the Type Library editor does not remove them from the corresponding implementation class. Edit the ActiveX wrapper class in the implementation unit to remove these after you have changed the interface in the Type Library editor.

**Warning** Any changes you make to the type library will be lost if you regenerate the ActiveX control from the original VCL control or form.

**Tip** It is a good idea to check the methods that the wizard adds to your ActiveX wrapper class. Not only does this give you a chance to note where the wizard omitted any data-aware properties or methods that were not Automation-compatible, it also lets you detect methods for which the wizard could not generate an implementation. Such methods appear with a comment in the implementation that indicates the problem.

## Adding additional properties, methods, and events

---

You can add additional properties, methods, and events to the control using the type library editor. The declaration is automatically added to the control's implementation unit, type library (TLB) file, and type library unit. The specifics of what Delphi supplies depends on whether you have added a property or method or whether you have added an event.

### Adding properties and methods

The ActiveX wrapper class implements properties in its interface using read and write access methods. That is, the wrapper class has COM properties, which appear on an interface as getter and/or setter methods. Unlike VCL properties, you do not see a "property" declaration on the interface for COM properties. Rather, you see methods that are flagged as property access methods. When you add a property to the ActiveX control's default interface, the wrapper class definition (which appears in the `_TLB` unit that is updated by the Type Library editor) gains one or two new methods (a getter and/or setter) that you must implement, just as when you add a method to the interface, the wrapper class gains a corresponding method for you to

implement. Thus, adding properties to the wrapper class's interface is essentially the same as adding methods: the wrapper class definition gains new skeletal method implementations for you to complete.

**Note** For details on what appears in the generated `_TLB` unit, see “Code generated when you import type library information” on page 35-5.

For example, consider a *Caption* property, of type *TCaption* in the underlying VCL object. To Add this property to the object's interface, you enter the following when you add a property to the interface via the type library editor:

```
property Caption: TCaption read Get_Caption write Set_Caption;
```

Delphi adds the following declarations to the wrapper class:

```
function Get_Caption: WideString; safecall;
procedure Set_Caption(const Value: WideString); safecall;
```

In addition, it adds skeletal method implementations for you to complete:

```
function TButtonX.Get_Caption: WideString;
begin
end;

procedure TButtonX.Set_Caption(Value: WideString);
begin
end;
```

Typically, you can implement these methods by simply delegating to the associated VCL control, which can be accessed using the *FDelphiControl* member of the wrapper class:

```
function TButtonX.Get_Caption: WideString;
begin
  Result := WideString(FDelphiControl.Caption);
end;

procedure TButtonX.Set_Caption(const Value: WideString);
begin
  FDelphiControl.Caption := TCaption(Value);
end;
```

In some cases, you may need to add code to convert the COM data types to native Object Pascal types. The preceding example manages this with typecasting.

**Note** Because the Automation interface methods are declared **safecall**, you do not have to implement COM exception code for these methods—the Delphi compiler handles this for you by generating code around the body of **safecall** methods to catch Delphi exceptions and to convert them into COM error info structures and return codes.

## Adding events

The ActiveX control can fire events to its container in the same way that an automation object fires events to clients. This mechanism is described in “Exposing events to clients” on page 36-10.

If the VCL control you are using as the basis of your ActiveX control has any published events, the wizards automatically add the necessary support for managing

a list of client event sinks to your ActiveX wrapper class and define the outgoing dispinterface that clients must implement to respond to events.

You add events to this outgoing dispinterface. To add an event in the type library editor, select the event interface and click on the method icon. Then manually add the list of parameters you want include using the parameter page.

Next, you must declare a method in your wrapper class that is of the same type as the event handler for the event in the underlying VCL control. This is not generated automatically, because Delphi does not know which event handler you are using:

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);
```

Implement this method to use the host application's event sink, which is stored in the wrapper class's *FEvents* member:

```
procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key); {cast to an OleAutomation compatible type }
  if FEvents <> nil then
    FEvents.OnKeyPress(TempKey)
  Key := Char(TempKey);
end;
```

**Note** When firing events in an ActiveX control, you do not need to iterate through a list of event sinks because the control only has a single host application. This is simpler than the process for most Automation servers.

Finally, you must assign this event handler to the underlying VCL control, so that it is called when the event occurs. You make this assignment in the *InitializeControl* method:

```
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
```

## Enabling simple data binding with the type library

---

With simple data binding, you can bind a property of your ActiveX control to a field in a database. To do this, the ActiveX control must communicate with its host application about what value represents field data and when it changes. You enable this communication by setting the property's binding flags using the Type Library editor.

By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies its container (the client host application) that the value has changed and requests that the database record be updated. The container interacts with the database and then notifies the control whether it succeeded or failed to update the record.

**Note** The container application that hosts your ActiveX control is responsible for connecting the data-aware properties you enable in the type library to the database. See “Using data-aware ActiveX controls” on page 35-8 for information on how to write such a container using Delphi.

Use the type library to enable simple data binding,

- 1 On the toolbar, click the property that you want to bind.
- 2 Choose the flags page.
- 3 Select the following binding attributes:

Binding attribute	Description
Bindable	Indicates that the property supports data binding. If marked bindable, the property notifies its container when the property value has changed.
Request Edit	Indicates that the property supports the OnRequestEdit notification. This allows the control to ask the container if its value can be edited by the user.
Display Bindable	Indicates that the container can show users that this property is bindable.
Default Bindable	Indicates the single, bindable property that best represents the object. Properties that have the default bind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Immediate Bindable	Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified. The bindable and request edit attribute bits need to be set for this new bit to have an effect.

- 4 Click the Refresh button on the toolbar to update the type library.

To test a data-binding control, you must register it first.

For example, to convert a *TEdit* control into a data-bound ActiveX control, create the ActiveX control from a *TEdit* and then change the Text property flags to Bindable, Display Bindable, Default Bindable, and Immediate Bindable. After the control is registered and imported, it can be used to display data.

## Creating a property page for an ActiveX control

A property page is a dialog box similar to the Delphi Object Inspector in which users can change the properties of an ActiveX control. A property page dialog allows you to group many properties for a control together to be edited at once. Or, you can provide a dialog box for more complex properties.

Typically, users access the property page by right-clicking the ActiveX control and choosing Properties.

The process of creating a property page is similar to creating a form, you

- 1 Create a new property page.
- 2 Add controls to the property page.
- 3 Associate the controls on the property page with the properties of an ActiveX control.
- 4 Connect the property page to the ActiveX control.

**Note** When adding properties to an ActiveX control or ActiveForm, you must publish the properties that you want to persist. If they are not published in the underlying VCL control, you must make a custom descendant of the VCL control that redeclares the properties as published and then use the ActiveX control wizard to create an ActiveX control from the descendant class.

## Creating a new property page

---

You use the Property Page wizard to create a new property page.

To create a new property page,

- 1 Choose File | New | Other.
- 2 Select the ActiveX tab.
- 3 Double-click the Property Page icon.

The wizard creates a new form and implementation unit for the property page. The form is a descendant of *TPropertyPage*, which lets you associate the form with the ActiveX control whose properties it edits.

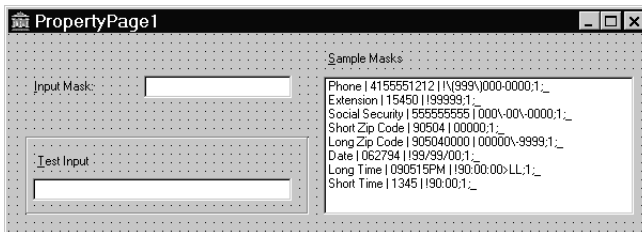
## Adding controls to a property page

---

You must add a control to the property page for each property of the ActiveX control that you want the user to access.

For example, the following illustration shows a property page for setting the MaskEdit property of an ActiveX control.

**Figure 38.1** Mask Edit property page in design mode



The list box allows the user to select from a list of sample masks. The edit controls allow the user to test the mask before applying it to the ActiveX control. You add controls to the property page the same as you would to a form.

## Associating property page controls with ActiveX control properties

---

After adding the controls you need to the property page, you must associate each control with its corresponding property. You make this association by adding code to the property page's *UpdatePropertyPage* and *UpdateObject* methods.

### Updating the property page

Add code to the *UpdatePropertyPage* method to update the control on the property page when the properties of the ActiveX control change. You must add code to the *UpdatePropertyPage* method to update the property page with the current values of the ActiveX control's properties.

You can access the ActiveX control using the property page's *OleObject* property, which is an *OleVariant* that contains the ActiveX control's interface.

For example, the following code updates the property page's edit control (InputMask) with the current value of the ActiveX control's *EditMask* property:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
    { Update your controls from OleObject }
    InputMask.Text := OleObject.EditMask;
end;
```

**Note** It is also possible to write a property page that represents more than one ActiveX control. In this case, you don't use the *OleObject* property. Instead, you must iterate through a list of interfaces that is maintained by the *OleObjects* property.

### Updating the object

Add code to the *UpdateObject* method to update the property when the user changes the controls on the property page. You must add code to the *UpdateObject* method in order to set the properties of the ActiveX control to their new values.

Once again you use the *OleObject* property to access the ActiveX control.

For example, the following code sets the *EditMask* property of the ActiveX control using the value in the property page's edit box control (InputMask):

```
procedure TPropertyPage1.UpdateObject;
begin
    {Update OleObject from your control }
    OleObject.EditMask := InputMask.Text;
end;
```



## Connecting a property page to an ActiveX control

---

To connect a property page to an ActiveX control,

- 1 Add *DefinePropertyPage* with the GUID constant of the property page as the parameter to the *DefinePropertyPages* method implementation in the control's implementation for the unit. For example,

```
procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);  
begin  
    DefinePropertyPage(Class_PropertyPage1);  
end;
```

The GUID constant, *Class\_PropertyPage1*, of the property page can be found in the property pages unit.

The GUID is defined in the property page's implementation unit; it is generated automatically by the Property Page wizard.

- 2 Add the property page unit to the **uses** clause of the controls implementation unit.

## Registering an ActiveX control

---

After you have created your ActiveX control, you must register it so that other applications can find and use it.

To register an ActiveX control:

- Choose Run | Register ActiveX Server.

**Note** Before you remove an ActiveX control from your system, you should unregister it.

To unregister an ActiveX control:

- Choose Run | Unregister ActiveX Server.

As an alternative, you can use the **regsvr** command from the command line or run the **regsvr32.exe** from the operating system.

## Testing an ActiveX control

---

To test your control, add it to a package and import it as an ActiveX control. This procedure adds the ActiveX control to the Delphi component palette. You can drop the control on a form and test as needed.

Your control should also be tested in all target applications that will use the control.

To debug the ActiveX control, select Run | Parameters and type the client name in the Host Application edit box.

The parameters then apply to the host application. Selecting Run | Run will run the host or client application and allow you to set breakpoints in the control.

## Deploying an ActiveX control on the Web

---

Before the ActiveX controls that you create can be used by Web clients, they must be deployed on your Web server. Every time you make a change to the ActiveX control, you must recompile and redeploy it so that client applications can see the changes.

Before you can deploy your ActiveX control, you must have a Web Server that will respond to client messages.

To deploy your ActiveX control, use the following steps:

- 1 Select Project | Web Deployment Options.
- 2 On the Project page, set the Target Dir to the location of the ActiveX control DLL as a path on the Web server. This can be a local path name or a UNC path, for example, C:\INETPUB\wwwroot.
- 3 Set the Target URL to the location as a Uniform Resource Locators (URL) of the ActiveX control DLL (without the file name) on your Web Server, for example, <http://mymachine.inprise.com/>. See the documentation for your Web Server for more information on how to do this.
- 4 Set the HTML Dir to the location (as a path) where the HTML file that contains a reference to the ActiveX control should be placed, for example, C:\INETPUB\wwwroot. This path can be a standard path name or a UNC path.
- 5 Set desired Web deployment options as described in “Setting options” on page 38-16.
- 6 Choose OK.
- 7 Choose Project | Web Deploy.

This creates a deployment code base that contains the ActiveX control in an ActiveX library (with the OCX extension). Depending on the options you specify, this deployment code base can also contain a cabinet (with the CAB extension) or information (with the INF extension).

The ActiveX library is placed in the Target Directory you specified in step 2. The HTML file has the same name as the project file but with the HTM extension. It is created in the HTML Directory specified in step 4. The HTML file contains a URL reference to the ActiveX library at the location specified in step 3.

**Note** If you want to put these files on your Web server, use an external utility such as ftp.

- 8 Invoke your ActiveX-enabled Web browser and view the created HTML page.

When this HTML page is viewed in the Web browser, your form or control is displayed and runs as an embedded application within the browser. That is, the library runs in the same process as the browser application.

## Setting options

---

Before deploying an ActiveX control, specify the Web deployment options that should be followed when creating the ActiveX library.

Web deployment options include settings to allow you to set the following:

- **Including additional files:** If your ActiveX control depends on any packages or other additional files, you can indicate that these should be deployed with the project. By default, these files use the same options that you specify for the entire project, but you can override these settings using the Packages or Additional files tab. When you include packages or additional files, Delphi creates a file with the .INF extension (for INFormation). This file specifies the various files that need to be downloaded and set up for the ActiveX library to run. The syntax of the INF file allows URLs pointing to packages or additional files to download.
- **CAB file compression:** A cabinet is a single file, usually with a CAB file extension, that stores compressed files in a file library. Cabinet compression can dramatically decrease download time (up to 70%) of a file. During installation, the browser decompresses the files stored in a cabinet and copies them to the user's system. Each file that you deploy can be CAB file compressed. You can specify that the ActiveX library use CAB file compression on the Project tab of the Web Deployment options dialog.
- **Version information:** You can specify that you want version information included with your ActiveX control. This information is set in the VersionInfo page of the Project Options dialog. Part of this information is the release number, which you can have automatically updated every time you deploy your ActiveX control. If you include additional packages or files, their Version information resources can get added to the INF file as well.

Depending on whether you include additional files and whether you use CAB file compression, the resulting ActiveX library may be an OCX file, a CAB file containing an OCX file, or an INF file. The following table summarizes the results of choosing different combinations.

Packages and/or additional files	CAB file compression	Result
No	No	An ActiveX library (OCX) file.
No	Yes	A CAB file containing an ActiveX library file.
Yes	No	An INF file, an ActiveX library file, and any additional files and packages.
Yes	Yes	An INF file, a CAB file containing an ActiveX library, and a CAB file each for any additional files and packages.

## Creating MTS or COM+ objects

Delphi uses the term transactional objects to refer to objects that take advantage of the transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) (for versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later). These objects are designed to work in a large, distributed environment. They are not available for use in cross-platform applications due to their dependence on Windows-specific technology.

Delphi provides a wizard that creates transactional objects so that you can take advantage of the benefits of COM+ attributes or the MTS environment. These features make creating COM clients and servers, particularly remote servers, easier to implement.

**Note** For database applications, Delphi also provides a Transactional Data Module. For more information, see Chapter 25, “Creating multi-tiered applications”.

Transactional objects make use of a number of low-level services, such as

- Managing system resources, including processes, threads, and database connections so that your server application can handle many simultaneous users
- Automatically initiating and controlling transactions so that your application is reliable.
- Creating, executing, and deleting server components when needed.
- Providing role-based security so that only authorized users can access your application.
- Managing events so that clients can respond to conditions that arise on the server (COM+ only).

By letting MTS or COM+ provide these underlying services, you can concentrate on developing the specifics for your particular distributed application. Which technology you choose (MTS or COM+) depends on the server on which you choose to run your application. To clients, the difference between the two (or, for that matter, the fact that the server object uses any of these services) is transparent (unless the client explicitly manipulates transactional services via a special interface).

## Understanding transactional objects

---

Typically, transactional objects are small, and are used for discrete business functions. They can implement an application's business rules, providing views and transformations of the application state. Consider, for example, the case of a medical application. Medical records stored in various databases represent the persistent state of the application, such as a patient's health history. Transactional objects update that state to reflect such changes as new patients, test results, and X-ray files.

Transactional objects are distinguished from other COM objects in that they use a set of attributes supplied by MTS or COM+ for handling issues that arise in a distributed computing environment. Some of these attributes require the transactional object to implement the *IObjectControl* interface. *IObjectControl* defines methods that are called when the object is activated or deactivated, where you can manage resources such as database connections. It also is required for object pooling, which is described in "Object pooling" on page 39-8.

**Note** If you are using MTS, your transactional objects must implement *IObjectControl*. Under COM+, *IObjectControl* is not required, but is highly recommended. The Transactional Object wizard provides an object that derives from *IObjectControl*.

A client of a transactional object is called a **base client**. From a base client's perspective, a transactional object looks like any other COM object.

Under MTS, the transactional object must be built into a library (DLL), which is then installed in the MTS runtime environment (the MTS executive, *mtxex.exe*). That is, the server object runs in the MTS runtime process space. The MTS executive can be running in the same process as the base client, as a separate process on the same machine as the base client, or as a remote server process on a separate machine.

Under COM+, the server application need not be an in-process server. Because the various services are integrated into the COM libraries, there is no need for a separate MTS process to intercept calls to the server. Instead, COM itself (or, rather, COM+) provides the resource management, transaction support, and so on. However, the server application must still be installed, this time into a COM+ application.

The connection between the base client and the transactional object is handled by a proxy on the client and a stub on the server, just as with any out-of-process server. Connection information is maintained by the proxy. The connection between the base client and proxy remains open as long as the client requires a connection to the server, so it appears to the client that it has continued access to the server. In reality, though, the proxy may deactivate and reactivate the object, conserving resources so that other clients may use the connection. For details on activating and deactivating, see "Just-in-time activation" on page 39-4.

## Requirements for a transactional object

---

In addition to the COM requirements, a transactional object must meet the following requirements:

- The object must have a standard class factory. This is automatically supplied by the wizard when you create the object.
- The server must expose its class object by exporting the standard *DllGetClassObject* method. Code to do this is supplied by the wizard.
- All object interfaces and CoClasses must be described by a type library, which is created automatically by the wizard. You can add methods and properties to interfaces in the type library by using the Type Library editor. The information in the type library is used by the MTS Explorer or COM+ Component Manager to extract information about the installed components at runtime.
- The server must only export interfaces that use standard COM marshaling. This is automatically supplied by the Transactional Object wizard. Delphi's support of transactional objects does not allow manual marshaling for custom interfaces. All interfaces must be implemented as dual interfaces that use COM's automatic marshaling support.
- The server must export the *DllRegisterServer* function and perform self-registration of its CLSID, ProgID, interfaces, and type library in this routine. This is provided by the Transactional Object wizard.

When using MTS rather than COM+, the following conditions apply as well:

- MTS requires that the server be a dynamic-link library (DLL). Servers that are implemented as executable files (.EXE files) cannot execute in the MTS runtime environment.
- The object must implement the *IObjectControl* interface. Support for this interface is automatically added by the Transactional Object wizard.
- A server running in the MTS process space cannot aggregate with COM objects not running in MTS.

## Managing resources

---

Transactional objects can be administered to better manage the resources used by your application. These resources include everything from the memory for the object instances themselves to any resources they use (such as database connections).

In general, you configure how your application manages resources by the way you install and configure your object. You set your transactional object so that it takes advantage of the following:

- Just-in-time activation
- Resource pooling
- Object pooling (COM+ only)

If you want your object to take full advantage of these services, however, it must use the *IObjectContext* interface to indicate when resources can safely be released.

## Accessing the object context

---

As with any COM object, a transactional object must be created before it is used. COM clients create an object by calling the COM library function, *CoCreateInstance*.

Each transactional object must have a corresponding context object. This context object is implemented automatically by MTS or COM+ and is used to manage the transactional object. The context object's interface is *IObjectContext*. To access most methods of the object context, you can use the *ObjectContext* property of the *TMtsAutoObject* object. For example, you can use the *ObjectContext* property as follows:

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Another way to access the Object context is to use methods in the *TMtsAutoObject* object:

```
if IsCallerInRole ('Manager') ...
```

You can use either of the above methods. However, there is a slight advantage of using the *TMtsAutoObject* methods rather than referencing the *ObjectContext* property when you are testing your application. For a discussion of the differences, see "Debugging and testing transactional objects" on page 39-21.

## Just-in-time activation

---

The ability for an object to be deactivated and reactivated while clients hold references to it is called **just-in-time activation**. From the client's perspective, only a single instance of the object exists from the time the client creates it to the time it is finally released. Actually, it is possible that the object has been deactivated and reactivated many times. By having objects deactivated, clients can hold references to the object for an extended time without affecting system resources. When an object is deactivated, all its resources can be released. For example, when an object is deactivated, it can release its database connection so that other objects can use it.

A transactional object is created in a deactivated state and becomes active upon receiving a client request. When the transactional object is created, a corresponding context object is also created. This context object exists for the entire lifetime of the transactional object, across one or more reactivation cycles. The context object, accessed by the *IObjectContext* interface, keeps track of the object during deactivation and coordinates transactions.

Transactional objects are deactivated as soon as it is safe to do so. This is called **as-soon-as-possible deactivation**. A transactional object is deactivated when any of the following occurs:

- **The object requests deactivation with *SetComplete* or *SetAbort*:** An object calls the *IObjectContext SetComplete* method when it has successfully completed its work and it does not need to save the internal object state for the next call from the client. An object calls *SetAbort* to indicate that it cannot successfully complete its work and its object state does not need to be saved. That is, the object's state rolls back to the state prior to the current transaction. Often, objects can be designed to be **stateless**, which means that objects deactivate upon return from every method.

- **A transaction is committed or aborted:** When an object's transaction is committed or aborted, the object is deactivated. Of these deactivated objects, the only ones that continue to exist are the ones that have references from clients outside the transaction. Subsequent calls to these objects reactivate them and cause them to execute in a new transaction.
- **The last client releases the object:** Of course, when a client releases the object, the object is deactivated, and the object context is also released.

**Note** If you install the transactional object under COM+ from the IDE, you can specify whether object supports just-in-time activation using the COM+ page of the Type Library editor. Just select the object (CoClass) in the Type Library editor, go to the COM+ page, and check or uncheck the box for Just In Time Activation. Otherwise, a system administrator specifies this attribute using the COM+ Component Manager or MTS Explorer. (The system administrator can also override any settings you specify using the Type Library editor.)

## Resource pooling

---

Since idle system resources are freed during a deactivation, the freed resources are available to other server objects. For example, a database connection that is no longer used by a server object can be reused by another client. This is called **resource pooling**. Pooled resources are managed by a resource dispenser.

A resource dispenser caches resources, so that transactional objects that are installed together can share them. The resource dispenser also manages nondurable shared state information. In this way, resource dispensers are similar to resource managers such as the SQL Server, but without the guarantee of durability.

When writing your transactional object, you can take advantage of two types of resource dispenser that are provided for you already:

- Database resource dispensers
- Shared Property Manager

Before other objects can use pooled resources, you must explicitly release them.

### Database resource dispensers

Opening and closing connections to a database can be time-consuming. By using a resource dispenser to pool database connections, your object can reuse existing database connections rather than create new ones. For example, if you have a database lookup and a database update component running in a customer maintenance application, you can install those components together, and then they can share database connections. In this way, your application does not need as many connections and new object instances can access the data more quickly by using a connection that is already open but not in use.

- If you are using BDE components to connect to your data, the resource dispenser is the Borland Database Engine (BDE). This resource dispenser is only available when your transactional object is installed with MTS. To enable the resource



dispenser, use the BDE administrator to turn on MTS POOLING in the System/Init area of the configuration.

- If you are using the ADO database components to connect to your data, the resource dispenser is provided by ADO.

**Note** There is no built-in resource pooling if you are using InterbaseExpress components for your database access.

For remote transactional data modules, connections are automatically enlisted on an object's transactions, and the resource dispenser can automatically reclaim and reuse connections.

## Shared property manager

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. By using the Shared Property Manager, you avoid having to add a lot of code to your application for managing shared data: the Shared Property Manager handles it for you by implementing locks and semaphores to protect shared properties from simultaneous access. The Shared Property Manager eliminates name collisions by providing **shared property groups**, which establish unique name spaces for the shared properties they contain.

To use the Shared Property Manager resource, you first use the *CreateSharedPropertyGroup* helper function to create a shared property group. Then you can write all the properties to that group and read all the properties from that group. By using a shared property group, the state information is saved across all deactivations of a transactional object. In addition, state information can be shared among all transactional objects installed in the same MTS package or COM+ application. You can install transactional objects into a package as described in "Installing transactional objects" on page 39-22.

For objects to share state, they all must run in the same process. If you want instances of different components to share properties, you must install them in the same MTS package or COM+ application. Because there is a risk that administrators may move components from one package to another, it's safest to limit the use of a shared property group to instances of objects that are defined in the same DLL or EXE.

Objects sharing properties must have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same MTS package or COM+ application.

The following example shows how to add code to support the Shared Property Manager in a transactional object:

### **Example: Sharing properties among transactional object instances**

This example creates a property group called MyGroup to contain the properties to be shared among objects and object instances. In this example, there is a Counter property that is shared. It uses the *CreateSharedPropertyGroup* helper function to

create the property group manager and property group, and then uses the *CreateProperty* method of the Group object to create a property called Counter.

To get the value of a property, you use the *PropertyByName* method of the Group object as shown below. You can also use the *PropertyByPosition* method.

```

unit Unit1;
interface
uses
  MtsObj, Mtx, ComObj, Project2_TLB;
type
  Tfoobar = class(TMtsAutoObject, Ifoobar)
  private
    Group: ISharedPropertyGroup;
  protected
    procedure OnActivate; override;
    procedure OnDeactivate; override;
    procedure IncCounter;
  end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
  Exists: WordBool;
  Counter: ISharedProperty;
begin
  Group := CreateSharedPropertyGroup('MyGroup');
  Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
  Counter: ISharedProperty;
begin
  Counter := Group.PropertyByName['Counter'];
  Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
  Group := nil;
end;
initialization
  TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance, tmApartment);
end.

```

## Releasing resources

You are responsible for releasing resources of an object. Typically, you do this by calling the *IObjectContext* methods *SetComplete* and *SetAbort* after servicing a client request. These methods release the resources allocated by the resource dispenser.

At this same time, you must release references to all other resources, including references to other objects (including transactional objects and context objects) and memory held by any instances of the component (freeing the component).

The only time you would not include these calls is if you want to maintain state between client calls. For details, see “Stateful and stateless objects” on page 39-11.

## Object pooling

---

Just as you can pool resources, under COM+ you can also pool objects. When an object is deactivated, COM+ calls the *IObjectControl* interface method, *CanBePooled*, which indicates that the object can be pooled for reuse. If *CanBePooled* returns *True*, then instead of being destroyed on deactivation, the object is moved to the object pool. It remains in the object pool for a specified timeout period, during which time it is available for use to any client requesting it. Only when the object pool is empty is a new instance of the object created. Objects that return *False* or that do not support the *IObjectControl* interface are destroyed when they are deactivated.

Object pooling is not available under MTS. MTS calls *CanBePooled* as described, but no pooling takes place. If your object will only run under COM+ and you want to allow object pooling, set the object’s *Pooled* property to *True*.

Even if an object’s *CanBePooled* method returns *True*, it can be configured so that COM+ does not move it to the object pool. If you install the transactional object under COM+ from the IDE, you can specify whether COM+ tries to pool the object using the COM+ page of the Type Library editor. Just select the object (CoClass) in the type library editor, go to the COM+ page, and check or uncheck the box for Object Pooling. Otherwise, a system administrator specifies this attribute using the COM+ Component Manager or MTS Explorer.

Similarly, you can configure the time a deactivated object remains in the object pool before it is freed. If you are installing from the IDE, you can specify this duration using the Creation Timeout setting on the COM+ page of the type library editor. Otherwise, a system administrator specifies this attribute using the COM+ Component Manager.

## MTS and COM+ transaction support

---

The transaction support that gives transactional objects their name lets you group actions into transactions. For example, in a medical records application, if you had a Transfer component to transfer records from one physician to another, you could include your Add and Delete methods in the same transaction. That way, either the entire Transfer works or it can be rolled back to its previous state. Transactions simplify error recovery for applications that must access *multiple* databases.

Transactions ensure that

- All updates in a single transaction are either committed or get aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial and uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**. Resource managers use transaction-based synchronization protocols to isolate the uncommitted work of active transactions.
- Committed updates to managed resources (such as database records) survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**. Transactional logging allows you to recover the durable state after disk media failures.

An object's associated context object indicates whether the object is executing within a transaction and, if so, the identity of the transaction. When an object is part of a transaction, the services that resource managers and resource dispensers perform on its behalf execute under the transaction as well. Resource dispensers use the context object to provide transaction-based services. For example, when an object executing within a transaction allocates a database connection by using the ADO or BDE resource dispenser, the connection is automatically enlisted on the transaction. All database updates using this connection become part of the transaction, and are either committed or aborted.

Work from multiple objects can be composed into a single transaction. Allowing an object to either live in its own transaction or be part of a larger group of objects that belong to a single transaction is a major advantage of MTS and COM+. It allows an object to be used in various ways, so that application developers can reuse application code in different applications without rewriting the application logic. In fact, developers can determine how objects are used in transactions when installing the transactional object. They can change the transaction behavior simply by adding an object to a different MTS package or COM+ application. For details about installing transactional objects, see "Installing transactional objects" on page 39-22.

## Transaction attributes

---

Every transactional object has a transaction attribute that is recorded in the MTS catalog or that is registered with COM+.

Delphi lets you set the transaction attribute at design time using the Transactional Object wizard or the Type Library editor.

Each transaction attribute can be set to these settings:

<b>Requires a transaction</b>	Objects must execute <i>within the scope of a transaction</i> . When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction context, a new one is automatically created.
<b>Requires a new transaction</b>	Objects must execute <i>within their own transactions</i> . When a new object is created, a new transaction is automatically created for the object, regardless of whether its client has a transaction. An object never runs inside the scope of its client's transaction. Instead, the system always creates independent transactions for the new objects.
<b>Supports transactions</b>	Objects can execute <i>within the scope of their client's transactions</i> . When a new object is created, its object context inherits the transaction from the context of the client. This enables multiple objects to be composed in a single transaction. If the client does not have a transaction, the new context is also created without one.
<b>Transactions Ignored</b>	Objects <i>do not run within the scope of transactions</i> . When a new object is created, its object context is created without a transaction, regardless of whether the client has a transaction. This setting is only available under COM+.
<b>Does not support transactions</b>	The meaning of this setting varies, depending on whether you install the object under MTS or COM+. Under MTS, this setting has the same meaning as Transactions Ignored under COM+. Under COM+, not only is the object context created without a transaction, this setting prevents the object from being activated if the client has a transaction.

## Setting the transaction attribute

You can set a transaction attribute when you first create a transactional object using the Transactional Object wizard.

You can also set (or change) the transaction attribute using the Type Library editor. To change the transaction attribute in the Type Library editor,

- 1 Choose View | Type Library to open the Type Library editor.
- 2 Select the class corresponding to the transactional object.
- 3 Click the COM+ tab and choose the desired transaction attribute.

**Warning** When you set the transaction attribute, Delphi inserts a special GUID for the specified attribute as custom data in the type library. This value is not recognized outside of Delphi. Therefore, it only has an effect if you install the transactional object from the IDE. Otherwise, a system administrator must set this value using the MTS Explorer or COM+ Component Manager.

**Note:** If the transactional object is already installed, you must first uninstall the object and reinstall it when changing the transaction attribute. Use Run | Install MTS objects or Run | Install COM+ objects to do so.

## Stateful and stateless objects

---

Like any COM object, transactional objects can maintain internal state across multiple interactions with a client. For example, the client could set a property value in one call, and expect that property value to remain unchanged when it makes the next call. Such an object is said to be **stateful**. Transactional objects can also be **stateless**, which means the object does not hold any intermediate state while waiting for the next call from a client.

When a transaction is committed or aborted, all objects that are involved in the transaction are deactivated, causing them to lose any state they acquired during the course of the transaction. This helps ensure transaction isolation and database consistency; it also frees server resources for use in other transactions. Completing a transaction enables the resources held by an object to be reclaimed when the object is deactivated. See the following section for information on how to control when the object's state is released.

Maintaining state on an object requires the object to remain activated, holding potentially valuable resources such as database connections.

## Influencing how transactions end

---

A transactional object uses the *IObjectContext* methods as shown in the following table to influence how a transaction completes. These methods, together with the object's transaction attribute, allow you to enlist one or more objects into a single transaction.

**Table 39.1** IObjectContext methods for transaction support

Method	Description
SetComplete	Indicates that the object has successfully completed its work for the transaction. The object is deactivated upon return from the method that first entered the context. The object reactivates on the next call that requires object execution.
SetAbort	Indicates that the object's work can never be committed and the transaction should be rolled back. The object is deactivated upon return from the method that first entered the context. The object reactivates on the next call that requires object execution.

**Table 39.1** IObjectContext methods for transaction support (continued)

Method	Description
EnableCommit	<p>Indicates that the object's work is not necessarily done, but that its transactional updates can be committed in their current form. Use this to retain state across multiple calls from a client while still allowing transactions to complete. The object is not deactivated until it calls SetComplete or SetAbort.</p> <p>EnableCommit is the default state when an object is activated. This is why an object should <i>always</i> call <i>SetComplete</i> or <i>SetAbort</i> before returning from a method, unless you want the object to maintain its internal state for the next call from a client.</p>
DisableCommit	<p>Indicates that the object's work is inconsistent and that it cannot complete its work until it receives further method invocations from the client. Call this before returning control to the client to maintain state across multiple client calls while keeping the current transaction active.</p> <p>DisableCommit prevents the object from deactivating and releasing its resources on return from a method call. Once an object has called DisableCommit, if a client attempts to commit the transaction before the object has called EnableCommit or SetComplete, the transaction will abort.</p>

## Initiating transactions

Transactions can be controlled in three ways:

- They can be controlled by the client.

Clients can have direct control over transactions by using a transaction context object (using the *ITransactionContext* interface).

- They can be controlled by the server.

Servers can control transactions explicitly creating an object context for them. When the server creates an object this way, the created object is automatically enlisted in the current transaction.

- Transactions can occur automatically as a result of the object's transaction attribute.

Transactional objects can be declared so that their objects always execute within a transaction, regardless of how the objects are created. This way, objects do not need to include any logic to handle transactions. This feature also reduces the burden on client applications. Clients do not need to initiate a transaction simply because the component that they are using requires it.

## Setting up a transaction object on the client side

A client-based application can control transaction context through the *ITransactionContextEx* interface. The following code example shows how a client application uses *CreateTransactionContextEx* to create the transaction context. This method returns an interface to this object.

This example wraps the call to the transaction context in a call to *OleCheck* which is necessary because the methods of *IObjectContext* are exposed by Windows directly and are therefore not declared as **safecall**.

```

procedure TForm1.MoveMoneyClick(Sender: TObject);
begin
    Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGUID; Amount: Currency);
var
    TransactionContextEx: ITransactionContextEx;
    CreditAccountIntf, DebitAccountIntf: IAccount;
begin
    TransactionContextEx := CreateTransactionContextEx;
    try
        OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
            IAccount, DebitAccountIntf));
        OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
            IAccount, CreditAccountIntf));
        DebitAccountIntf.Debit(Amount);
        CreditAccountIntf.Credit(Amount);
    except
        TransactionContextEx.Abort;
        raise;
    end;
    TransactionContextEx.Commit;
end;

```

## Setting up a transaction object on the server side

To control transaction context from the server side, you create an instance of *ObjectContext*. In the following example, the Transfer method is in the transactional object. In using *ObjectContext* this way, the instance of the object we are creating will inherit all the transaction attributes of the object that creates it. We wrap the call in a call to *OleCheck* because the methods of *IOBJECTCONTEXT* are exposed by Windows directly and are therefore not declared as **safecall**.

```

procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGUID;
    Amount: Currency);
var
    CreditAccountIntf, DebitAccountIntf: IAccount;
begin
    try
        OleCheck(ObjectContext.CreateInstance(DebitAccountId,
            IAccount, DebitAccountIntf));
        OleCheck(ObjectContext.CreateInstance(CreditAccountId,
            IAccount, CreditAccountIntf));
        DebitAccountIntf.Debit(Amount);
        CreditAccountIntf.Credit(Amount);
    except
        DisableCommit;
        raise;
    end;
    EnableCommit;
end;

```



## Transaction timeout

---

The transaction timeout sets how long (in seconds) a transaction can remain active. The system automatically aborts transactions that are still alive after the timeout. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging transactional objects.

To set the timeout value on your computer,

- 1 In the MTS Explorer or COM+ Component Manager, select Computer, My Computer.

By default, My Computer corresponds to the local computer.

- 2 Right-click and choose Properties and then choose the Options tab.

The Options tab is used to set the computer's transaction timeout property.

- 3 Change the timeout value to 0 to disable transaction timeouts.

- 4 Click OK to save the setting.

For more information on debugging MTS applications, see “Debugging and testing transactional objects” on page 39-21.

## Role-based security

---

MTS and COM+ provide role-based security where you assign a role to a logical group of users. For example, a medical information application might define roles for Physician, X-ray technician, and Patient.

You define authorization for each object and interface by assigning roles. For example, in the physicians' medical application, only the Physician may be authorized to view all medical records; the X-ray Technician may view only X-rays; and Patients may view only their own medical record.

Typically, you define roles during application development and assign roles for each MTS package or COM+ Application. These roles are then assigned to specific users when the application is deployed. Administrators can configure the roles using the MTS Explorer or COM+ Component Manager.

If you want to control access to blocks of code rather than entire objects, you can provide more fine-grained security by using the *IObjectContext* method, *IsCallerInRole*. This method only works if security is enabled, which can be checked by calling the *IObjectContext* method *IsSecurityEnabled*. These methods are automatically added as methods to your transactional object. For example,

```

if IsSecurityEnabled then {check if security is enabled }
begin
  if IsCallerInRole('Physician') then { check caller's role }
  begin
    { execute the call normally }
  end
else

```

```

    { not a physician, do something appropriate }
  end
end
else
  { no security enabled, do something appropriate }
end;

```

**Note** For applications that require stronger security, context objects implement the *ISecurityProperty* interface, whose methods allow retrieval of the Window's security identifier (SID) for the direct caller and creator of the object, as well as the SID for the clients which are using the object.

## Overview of creating transactional objects

---

The process of creating transactional object is as follows:

- 1 Use the Transactional Object wizard to create the transactional object.
- 2 Add methods and properties to the object's interface using the Type Library editor. For details on adding methods and properties using the Type Library editor, see Chapter 34, "Working with type libraries."
- 3 When implementing your object's methods, you can use the *IObjectContext* interface to manage transactions, persistent state, and security. In addition, if you are passing object references, you will need to use extra care so that they are correctly handled. (See "Passing object references" on page 20.)
- 4 Debug and test the transactional object.
- 5 Install the transactional object into an MTS package or COM+ application.
- 6 Administer your objects using the MTS Explorer or COM+ Component Manager.

## Using the Transactional Object wizard

---

Use the Transactional Object wizard to create a COM object that can take advantage of the resource management, transaction processing, and role-based security provided by MTS or COM+.

To bring up the Transactional Object wizard,

- 1 Choose File | New | Other.
- 2 Select the tab labeled Multitier.
- 3 Double-click the Transactional Object icon.

In the wizard, you must specify the following:

- A threading model that indicates how client applications can call your object's interface. The threading model determines how the object is registered. You are responsible for ensuring that the object's implementation adheres to the selected model. For more information on threading models, see "Choosing a threading model for a transactional object" on page 39-16.

- A transaction model
- An indication of whether your object notifies clients of events. Event support is only provided for traditional events, not COM+ events.

When you complete this procedure, a new unit is added to the current project that contains the definition for the transactional object. In addition, the wizard adds a type library to the project and opens it in the Type Library editor. Now you can expose the properties and methods of the interface through the type library. You define the interface as you would define any COM object as described in “Defining a COM object’s interface” on page 36-9.

The transactional object implements a **dual interface**, which supports both early (compile-time) binding through the vtable and late (runtime) binding through the *IDispatch* interface.

The generated transactional object implements the *IObjectControl* interface methods, *Activate*, *Deactivate*, and *CanBePooled*.

It is not strictly necessary to use the transactional object wizard. You can convert any Automation object into a COM+ transactional object (and any in-process Automation object into an MTS transactional object) by using the COM+ page of the Type Library editor and then installing the object into an MTS package or COM+ application. However, the transactional object wizard provides certain benefits:

- It automatically implements the *IObjectControl* interface, adding *OnActivate* and *OnDeactivate* events to the object so that you can create event handlers that respond when the object is activated or deactivated.
- It automatically generates an *ObjectContext* property so that it is easy for your object to access the *IObjectContext* methods to control activation and transactions.

## Choosing a threading model for a transactional object

---

The MTS runtime environment or COM+ manages threads for you. Transactional objects should not create threads. They must also never terminate a thread that calls into a DLL.

When you specify the threading model using the Transactional object wizard, you specify how objects are assigned to threads for method execution.

**Table 39.2** Threading models for transactional objects

Threading model	Description	Implementation pros and cons
Single	<p>No thread support. Client requests are serialized by the calling mechanism.</p> <p>All objects of a single-threaded component execute on the main thread.</p> <p>This is compatible with the default COM threading model, which is used for components that do not have a Threading Model Registry attribute or for COM components that are not reentrant. Method execution is serialized across all objects in the component and across all components in a process.</p>	<p>Allows components to use libraries that are not reentrant.</p> <p>Very limited scalability.</p> <p>Single-threaded, stateful components are prone to deadlocks. You can eliminate this problem by using stateless objects and calling SetComplete before returning from any method.</p>
Apartment (or Single-threaded apartment)	<p>Each object is assigned to a thread apartment, which lasts for the life of the object; however, multiple threads can be used for multiple objects. This is a standard COM concurrency model. Each apartment is tied to a specific thread and has a Windows message pump.</p>	<p>Provides significant concurrency improvements over the single threading model.</p> <p>Two objects can execute concurrently as long as they are not in the same activity.</p> <p>Similar to a COM apartment, except that the objects can be distributed across multiple processes.</p>

**Note** These threading models are similar to those defined by COM objects. However, because the MTS and COM+ provide more underlying support for threads, the meaning of each threading model differs here. Also, the free threading model does not apply to transactional objects due to the built-in support for activities.

## Activities

In addition to the threading model, transactional objects achieve concurrency through **activities**. Activities are recorded in an object's context, and the association between an object and an activity cannot be changed. An activity includes the transactional object created by the base client, as well as any transactional objects created by that object and its descendants. These objects can be distributed across one or more processes, executing on one or more computers.

For example, a physician's medical application may have a transactional object to add updates and remove records to various medical databases, each represented by a different object. This record object may use other objects as well, such as a receipt object to record the transaction. This results in several transactional objects that are either directly or indirectly under the control of a base client. These objects all belong to the same activity.

MTS or COM+ tracks the flow of execution through each activity, preventing inadvertent parallelism from corrupting the application state. This feature results in a single logical thread of execution throughout a potentially distributed collection of objects. By having one logical thread, applications are significantly easier to write.

When a transactional object is created from an existing context, using either a transaction context object or an object context, the new object becomes a member of the same activity. In other words, the new context inherits the activity identifier of the context used to create it.

Only a single logical thread of execution is allowed within an activity. This is similar in behavior to a COM apartment threading model, except that the objects can be distributed across multiple processes. When a base client calls into an activity, all other requests for work in the activity (such as from another client thread) are blocked until after the initial thread of execution returns back to the client.

Under MTS, every transactional object belongs to one activity. Under COM+, you can configure the way the object participates in activities by setting the **call synchronization**. The following options are available:

**Table 39.3** Call synchronization options

Option	Meaning
Disabled	COM+ does not assign activities to the object but it may inherit them with the caller's context. If the caller has no transaction or object context, the object is not assigned to an activity. The result is the same as if the object was not installed in a COM+ application. This option should not be used if any object in the application uses a resource manager or if the object supports transactions or just-in-time activation.
Not Supported	COM+ never assigns the object to an activity, regardless of the status of its caller. This option should not be used if any object in the application uses a resource manager or if the object supports transactions or just-in-time activation.
Supported	COM+ assigns the object to the same activity as its caller. If the caller does not belong to an activity, the object does not either. This option should not be used if any object in the application uses a resource manager or if the object supports transactions or just-in-time activation.
Required	COM+ always assigns the object to an activity, creating one if necessary. This option must be used if the transaction attribute is Supported or Required.
Requires New	COM+ always assigns the object to a new activity, which is distinct from its caller's.

## Generating events under COM+

Before COM+, Automation servers used a set of special interfaces for generating events. COM+, however, introduces a new system for managing events. Instead of the server object managing events, keeping track of clients that need to be notified and calling their interfaces when events occur, the underlying system (COM+) manages this process.

**Note** Transactional objects installed under COM+ can still use the old system for managing events. However, letting COM+ handle the process provides greater flexibility. For example, when COM+ manages events, the client can be an in-process server that is launched by COM+ when the event occurs.

When a COM+ object generates events, it does not do so directly. Rather, it makes use of an associated event object that is specifically created to generate events. The COM+ object calls its event object when it wants to fire an event. When that happens, COM+ calls all clients that have registered an interest in the particular event object.

## Using the Event Object wizard

---

You can create event objects using the Event Object wizard. The wizard first checks whether the current project contains any implementation code, because projects containing COM+ event objects do not include an implementation. They can only contain event object definitions. (You can, however, include multiple COM+ event objects in a single project.)

To bring up the Event Object wizard,

- 1 Choose File | New.
- 2 Select the tab labeled ActiveX.
- 3 Double-click the COM+ Event Object icon.

In the Event Object wizard, specify the name of the event object, the name of the interface that defines the event handlers, and (optionally) a brief description of the events.

When you exit, the wizard creates a project containing a type library that defines your event object and its interface. Use the Type Library editor to define the methods of that interface. These methods are the event handlers that clients implement to respond to events.

The Event object project includes the project file, `_ATL` unit to import the ATL template classes, and the `_TLB` unit to define the type library information. It does not include an implementation unit, however, because COM+ event objects have no implementation. The implementation of the interface is the responsibility of the client. When your server object calls a COM+ event object, COM+ intercepts the call and dispatches it to registered clients. Because COM+ event objects require no implementation object, all you need to do after defining the object's interface in the Type Library editor is compile the project and install it with COM+

COM+ places certain restrictions on the interfaces of event objects. The interface you define in the Type Library editor for your event object must obey the following rules:

- The event object's interface must derive from `IDispatch`.
- All method names must be unique across all interfaces of the event object.
- All methods on the event object's interface must return an `HRESULT` value.
- The modifier for all parameters of methods must be blank.

## Firing events using a COM+ event object

---

When an event occurs, your COM+ object must call the event object and tell it to fire the event on registered clients. It does this by creating an instance of the event object and calling the method that corresponds to the event:

**Note** Objects that fire COM+ events, like the event objects themselves, must be installed in a COM+ application.

## Passing object references

---

**Note** Information on passing object references applies only to MTS, not COM+. This mechanism is needed under MTS because it is necessary to ensure that all pointers to objects running under MTS are routed through interceptors. Because interceptors are built into COM+, you do not need to pass object references.

Under MTS, you can pass object references, (for example, for use as a callback) only in the following ways:

- Through return from an object creation interface, such as *CoCreateInstance* (or its equivalent), *ITransactionContext.CreateInstance*, or *IObjectContext.CreateInstance*.
- Through a call to *QueryInterface*.
- Through a method that has called *SafeRef* to obtain the object reference.

An object reference that is obtained in the above ways is called a **safe reference**. Methods invoked using safe references are guaranteed execute within the correct context.

The MTS runtime environment requires calls to use safe references so that it can manage context switches and allows transactional objects to have lifetimes that are independent of client references. Safe references are not necessary under COM+.

### Using the SafeRef method

An object can use the *SafeRef* function to obtain a reference to itself that is safe to pass outside its context. The unit that defines the *SafeRef* function is Mtx.

*SafeRef* takes as input

- A reference to the interface ID (RIID) of the interface that the current object wants to pass to another object or client.
- A reference to the current object's IUnknown interface.

*SafeRef* returns a pointer to the interface specified in the RIID parameter that is safe to pass outside the current object's context. It returns **nil** if the object is requesting a safe reference on an object other than itself, or the interface requested in the RIID parameter is not implemented.

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a callback), it should always call *SafeRef* first and then pass the reference returned by this call. An object should never pass a **self** pointer, or a self-

reference obtained through an internal call to `QueryInterface`, to a client or to any other object. Once such a reference is passed outside the object's context, it is no longer a valid reference.

Calling `SafeRef` on a reference that is already safe returns the safe reference unchanged, except that the reference count on the interface is incremented.

When a client calls `QueryInterface` on a reference that is safe, the reference returned to the client is also a safe reference.

An object that obtains a safe reference must release the safe reference when finished with it.

For details on `SafeRef` see the `SafeRef` topic in the Microsoft documentation.

## Callbacks

Objects can make callbacks to clients and to other transactional objects. For example, you can have an object that creates another object. The creating object can pass a reference of itself to the created object; the created object can then use this reference to call the creating object.

If you choose to use callbacks, note the following restrictions:

- Calling back to the base client or another package requires access-level security on the client. Additionally, the client must be a DCOM server.
- Intervening firewalls may block calls back to the client.
- Work done on the callback executes in the environment of the object being called. It may be part of the same transaction, a different transaction, or no transaction.
- Under MTS, the creating object must call `SafeRef` and pass the returned reference to the created object in order to call back to itself.

## Debugging and testing transactional objects

---

You can debug local and remote transactional objects. When debugging transactional objects, you may want to turn off transaction timeouts.

The transaction timeout sets how long (in seconds) a transaction can remain active. Transactions that are still alive after the timeout are automatically aborted by the system. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging.

For information on remote debugging, see the `Remote Debugging` topic in Online help.

When testing a transactional object that you intend to run under MTS, you may first want to test your object outside the MTS environment to simplify your test environment.

While developing your server, you cannot rebuild the server when it is still in memory. You may get a compiler error like, "Cannot write to DLL while executable



is loaded.” To avoid this, you can set the MTS package or COM+ application properties to shut down the server when it is idle.

To shut down the server when idle,

- 1 In the MTS Explorer or COM+ Component Manager, right-click the MTS package or COM+ application in which your transactional object is installed and choose Properties.
- 2 Select the Advanced tab.  
The Advanced tab determines whether the server process associated with a package always runs, or whether it shuts down after a certain period of time.
- 3 Change the timeout value to 0, which shuts down the server as soon as no longer has a client to service.
- 4 Click OK to save the setting.

**Note** When testing outside the MTS environment, you do not reference the *ObjectProperty* of *TMtsObject* directly. The *TMtsObject* implements methods such as *SetComplete* and *SetAbort* that are safe to call when the object context is **nil**.

## Installing transactional objects

---

MTS applications consist of a group of in-process MTS objects running in a single instance of the MTS executive (EXE). A group of COM objects that all run in the same process is called a **package**. A single machine can be running several different packages, where each package is running within a separate MTS EXE.

Under COM+, you work with a similar group, called a COM+ application. In a **COM+ application**, the objects need not be in-process, and there is no separate runtime environment.

You can group your application components into a single MTS package or COM+ application to be managed by a single process. You might want to distribute your components into different MTS packages or COM+ applications to partition your application across multiple processes or machines.

To install transactional objects into an MTS package or COM+ application,

- 1 If your system supports COM+, choose Run | Install COM+ objects. If your system does not support COM+ but you have MTS installed on your system, choose Run | Install MTS objects. If your system supports neither MTS nor COM+, you will not see a menu item for installing transactional objects.
- 2 In the Install Object dialog box, check the objects to be installed.
- 3 If you are installing MTS objects, click the Package button to get a list of MTS packages on your system. If you are installing COM+ objects, click the Application button. Indicate the MTS package or COM+ application into which you are installing your objects. You can choose Into New Package or Into New Application to create a new MTS package or COM+ application in which to install the object. You can choose Into Existing Package or Into Existing Application to install the object into an existing listed MTS package or COM+ application.

4 Choose OK to refresh the catalog, which makes the objects available at runtime.

MTS packages can contain components from multiple DLLs, and components from a single DLL can be installed into different packages. However, a single component cannot be distributed among multiple packages.

Similarly, COM+ applications can contain components from multiple executables and different components from a single executable can be installed into different COM+ applications.

**Note** You can also install your transactional object using the COM+ Component Manager or MTS Explorer. Be sure when installing the object with one of these tools that you apply the settings for the object that appear on the COM+ page of the Type Library editor. These settings are not applied automatically when you do not install from the IDE.

## Administering transactional objects

---

Once you have installed transactional objects, you can administer these runtime objects using the MTS Explorer (if they are installed into an MTS package) or the COM+ Component Manager (if they are installed into a COM+ application). Both tools are identical, except that the MTS Explorer operates on the MTS runtime environment and the COM+ Component Manager operates on COM+ objects.

The COM+ Component Manager and MTS Explorer have a graphical user interface for managing and deploying transactional objects. Using one of these tools, you can

- Configure transactional objects, MTS packages or COM+ applications, and roles
- View properties of components in an package or COM+ application and view the MTS packages or COM+ applications installed on a computer
- Monitor and manage transactions for objects that comprise transactions
- Move MTS packages or COM+ applications between computers
- Make a remote transactional object available to a local client

For more details on these tools, see the appropriate *Administrator's Guide* from Microsoft.





## Creating custom components

The chapters in “Creating custom components” present concepts necessary for designing and implementing custom components in Delphi.



## Overview of component creation

This chapter provides an overview of component design and the process of writing components for Delphi applications. The material here assumes that you are familiar with Delphi and its standard components.

- VCL and CLX
- Components and classes
- How do you create components?
- What goes into a component?
- Creating a new component
- Testing uninstalled components
- Testing installed components

For information on installing new components, see “Installing component packages” on page 11-5.

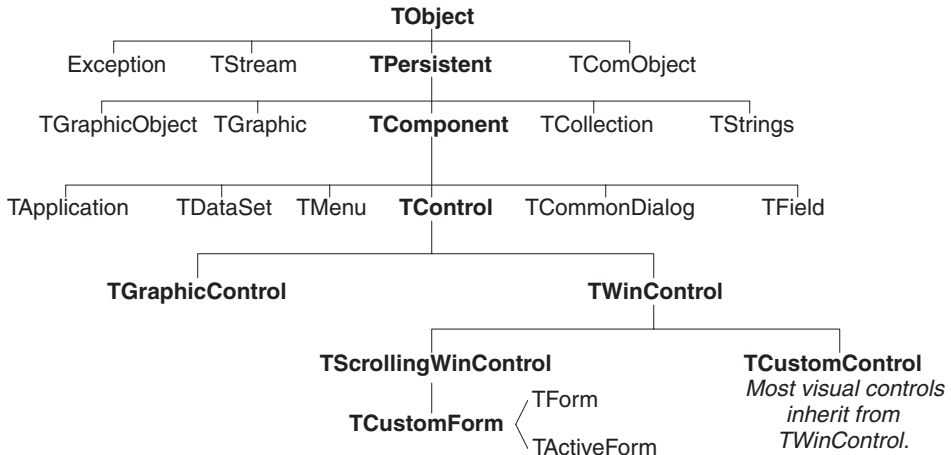
### VCL and CLX

---

Delphi’s components reside in two class hierarchies called the Visual Component Library (VCL) and the Component Library for Cross Platform (CLX). Figure 40.1 shows the relationship of selected classes that make up the VCL. The CLX hierarchy is similar to the VCL but Windows controls are called widgets (therefore *TWinControl* is called *TWidgetControl*, for example), and there are other differences. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Chapter 41, “Object-oriented programming for component writers.” For an overview of how CLX differs from the VCL, see “CLX versus VCL” on page 10-5 and refer to the CLX online reference for details on the components.

The *TComponent* class is the shared ancestor of every component in the VCL and CLX. *TComponent* provides the minimal properties and events necessary for a component to work in Delphi. The various branches of the library provide other, more specialized capabilities.

**Figure 40.1** Visual Component Library class hierarchy



When you create a component, you add to the VCL or CLX by deriving a new class from one of the existing class types in the hierarchy.

## Components and classes

---

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

Briefly, there are two main differences between creating components and using them in applications. When creating components,

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

## How do you create components?

---

A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one. You can derive a new component from any existing component, but the following are the most common ways to create components:

- Modifying existing controls
- Creating windowed controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

Table 40.1 summarizes the different kinds of components and the classes you use as starting points for each.

**Table 40.1** Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as <i>TButton</i> or <i>TListBox</i> , or an abstract component type, such as <i>TCustomListBox</i>
Create a windowed (or widget-based in CLX) control	<i>TWinControl</i> ( <i>TWidgetControl</i> in CLX)
Create a graphic control	<i>TGraphicControl</i>
Subclassing a control	Any Windows (VCL) or widget-based (CLX) control
Create a nonvisual component	<i>TComponent</i>

You can also derive classes that are not components and cannot be manipulated on a form. Delphi includes many such classes, like *TRegIniFile* and *TFont*.

## Modifying existing controls

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided with Delphi.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the VCL and CLX includes an abstract class (with the word “custom” in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWinControl* (or *TWidgetControl* in CLX) class and reinvent all the list box functions, the VCL or CLX provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you publish only the properties you want to make available in your component and leave the rest protected.

Chapter 42, “Creating properties,” explains publishing inherited properties. Chapter 48, “Modifying an existing component,” and Chapter 50, “Customizing a grid,” show examples of modifying existing controls.

## Creating windowed controls

Windowed controls in the VCL and CLX are objects that appear at runtime and that the user can interact with. Each windowed control has a window handle, accessed through its *Handle* property, that lets the operating system identify and operate on the control. If using VCL controls, the handle allows the control to receive input focus and can be passed to Windows API functions. In CLX, these controls are widget-



based controls. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.

All windowed controls descend from the *TWinControl* (*TWidgetControl* in CLX) class. These include most standard windowed controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that's not related to any existing control) directly from *TWinControl* (*TWidgetControl* in CLX), Delphi provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized windowed control that makes it easier to draw complex visual images.

Chapter 50, "Customizing a grid," presents an example of creating a windowed control.

## Creating graphic controls

---

If your control does not need to receive input focus, you can make it a graphic control. Graphic controls are similar to windowed controls, but have no window handles, and therefore consume fewer system resources. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to mouse messages.

Delphi supports the creation of custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on and on Windows, handles *WM\_PAINT* messages; all you need to do is override the *Paint* method.

Chapter 49, "Creating a graphic component," presents an example of creating a graphic control.

## Subclassing Windows controls

---

In traditional Windows programming, you create custom controls by defining a new *window class* and registering it with Windows. The window class (which is similar to the *objects* or *classes* in object-oriented programming) contains information shared among instances of the same sort of control; you can base a new window class on an existing class, which is called *subclassing*. You then put your control in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

Using Delphi, you can create a component "wrapper" around any existing window class. So if you already have a library of custom controls that you want to use in Delphi applications, you can create Delphi components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing Windows controls, see the components in the *StdCtrls* unit that represent standard Windows controls, such as *TEdit*. For CLX examples, see *QStdCtrls*.

## Creating nonvisual components

---

Nonvisual components are used as interfaces for elements like databases (*TDataSet* or *TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TCommonDialog* (VCL) or *TDialog* (CLX) and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components.

## What goes into a component?

---

To make your components reliable parts of the Delphi environment, you need to follow certain conventions in their design. This section discusses the following topics:

- Removing dependencies
- Properties, methods, and events
- Graphics encapsulation
- Registration

### Removing dependencies

---

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An excellent example of removing dependencies is the *Handle* property of *TWinControl*. If you have written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a window or control until you have created it by calling the *CreateWindow* API function. Delphi windowed controls relieve users from this concern by ensuring that a valid window handle is always available when needed. By using a property to represent the window handle, the control can check whether the window has been created; if the handle is not valid, the control creates a window and returns the handle. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating the window, Delphi components allow developers to focus on what they really want to do. Before passing a window handle to an API function, there is no need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

## Properties, methods, and events

---

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a chapter devoted to it in this book, but the discussion that follows explains some of the motivation for their use.

### Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

Chapter 42, “Creating properties,” explains how to add properties to your components.

### Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

Chapter 43, “Creating events,” explains how to implement standard events and how to define new ones.

### Methods

Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component’s constructor method (*Create*) is a class method. Component methods are procedures and functions that operate on the component instances themselves. Application developers use methods to direct a component to perform a specific action or return a value not contained by any property.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

Chapter 44, "Creating methods," explains how to add methods to your components.

## Graphics encapsulation

---

Delphi simplifies Windows graphics by encapsulating various graphic tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is like a Windows device context, but it takes care of all the bookkeeping for you.

If you have written a graphical Windows application, you are familiar with the requirements imposed by Windows' graphics device interface (GDI). For example, GDI limits the number of device contexts available and requires that you restore graphic objects to their initial state before destroying them.

With Delphi, you do not have to worry about these things. To draw on a form or other component, you access the component's *Canvas* property. If you want to customize a pen or brush, you set its color or style. When you finish, Delphi disposes of the resources. Delphi caches resources to avoid recreating them if your application frequently uses the same kinds of resource.

You still have full access to the Windows GDI, but you will often find that your code is simpler and runs faster if you use the canvas built into Delphi components. Graphics features are detailed in Chapter 45, "Using graphics in components."

CLX graphics encapsulation works differently. A canvas is a painter instead. To draw on a form or other component, you access the component's *Canvas* property. *Canvas* is a property and it is also an object called *TCanvas*. *TCanvas* is a wrapper around a Qt painter that is accessible through the *Handle* property. You can use the handle to access low-level Qt graphics library functions.

If you want to customize a pen or brush, you set its color or style. When you finish, Kylix disposes of the resources. CLX also caches the resources.

You can use the canvas built into CLX components by descending from them. How graphics images work in the component depends on the canvas of the object from which your component descends.

## Registration

---

Before you can install your components in the Delphi IDE, you have to register them. Registration tells Delphi where to place the component on the Component palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Chapter 47, “Making components available at design time.”

## Creating a new component

---

You can create a new component two ways:

- Using the Component wizard
- Creating a component manually

You can use either of these methods to create a minimally functional component ready to install on the Component palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the Component palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

- 1 Create a unit for the new component.
- 2 Derive your component from an existing component type.
- 3 Add properties, methods, and events.
- 4 Register your component with Delphi.
- 5 Create a Help file for your component and its properties, methods, and events.
- 6 Create a package (a special dynamic-link library) so that you can install your component in the Delphi IDE.

When you finish, the complete component includes the following files:

- A package (.BPL) or package collection (.DPC) file
- A compiled package (.DCP) file
- A compiled unit (.DCU) file
- A palette bitmap (.DCR) file
- A Help (.HLP) file

Creating a help file to instruct component users on how to use the component is optional.

The chapters in the rest of Part V explain all the aspects of building components and provide several complete examples of writing different kinds of components.

## Using the Component wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify only these things:

- The class from which it is derived
- The class name for the new component
- The Component palette page where you want it to appear
- The name of the unit in which the component is created
- The search path where the unit is found
- The name of the package in which you want to place the component

The Component wizard performs the same tasks you would when creating a component manually:

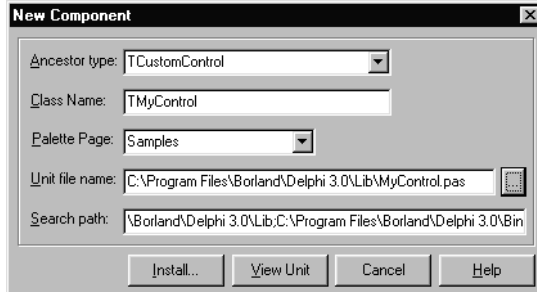
- Creating a unit
- Deriving the component
- Registering the component

The Component wizard cannot add components to an existing unit. You must add components to existing units manually.

To start the Component wizard, choose one of these two methods:

- Choose Component | New Component.
- Choose File | New | Other and double-click on Component

**Figure 40.2** Component wizard



Fill in the fields in the Component wizard:

- 1 In the Ancestor Type field, specify the class from which you are deriving your new component.

**Note**

In the drop-down list, many components are listed twice with different unit names, one for VCL and one for CLX. The CLX-specific units begin with Q (such as QGraphics instead of Graphics). Be sure to descend from the correct component.

- 2 In the Class Name field, specify the name of your new component class.
- 3 In the Palette Page field, specify the page on the Component palette on which you want the new component to be installed.

- 4 In the Unit file name field, specify the name of the unit you want the component class declared in.
- 5 If the unit is not on the search path, edit the search path in the Search Path field as necessary.

To place the component in a new or existing package, click Component | Install and use the dialog box that appears to specify a package.

**Warning** If you derive a component from a VCL or CLX class whose name begins with “custom” (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. Delphi cannot create instance objects of a class that has abstract properties or methods.

To see the source code for your unit, click View Unit. (If the Component wizard is already closed, open the unit file in the Code editor by selecting File | Open.) Delphi creates a new unit containing the class declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Delphi units.

The unit looks like this if descending from *TCustomControl* in the Controls unit:

```
unit MyControl;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TMyControl = class(TCustomControl)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyControl]);
end;

end.
```

If descending from *TCustomControl* in the QControls unit, the only difference is the **uses** clause which looks like this:

```
uses
  Windows, Messages, SysUtils, Classes, QControls;
```

Where CLX uses separate units, they are replaced with units of the same name prefixed with a Q; Controls is replaced by QControls.

## Creating a component manually

---

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

To create a component manually, follow these steps:

- 1 Creating a unit file
- 2 Deriving the component
- 3 Registering the component

### Creating a unit file

A unit is a separately compiled module of Object Pascal code. Delphi uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well.

When you create a component, you either create a new unit for the component or add the new component to an existing unit.

To create a unit, choose File | New | Unit. Delphi creates a new unit file and opens it in the Code editor.

To open an existing unit, choose File | Open and select the source code unit that you want to add your component to.

**Note** When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the Component palette.

Once you have either a new or existing unit for your component, you can derive the component class.

### Deriving the component

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. “How do you create components?” on page 40-2 describes which class to derive different kinds of components from.

Deriving classes is explained in more detail in the section “Defining new classes” on page 41-1.

To derive a component, add an object type declaration to the **interface** part of the unit that will contain the component.

A simple component class is a nonvisual component descended directly from *TComponent*.

To create a simple component class, add the following class declaration to the **interface** part of your component unit:

```
type
  TNewComponent = class(TComponent)
  end;
```

So far the new component does nothing different from *TComponent*. You have created a framework on which to build your new component.



## Registering the component

Registration is a simple process that tells Delphi which components to add to its component library, and on which pages of the Component palette they should appear. For a more detailed discussion of the registration process, see Chapter 47, “Making components available at design time.”

To register a component,

- 1 Add a procedure named *Register* to the **interface** part of the component’s unit. *Register* takes no parameters, so the declaration is very simple:

```
procedure Register;
```

If you are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not need to change the declaration.

- 2 Write the *Register* procedure in the **implementation** part of the unit, calling *RegisterComponents* for each component you want to register. *RegisterComponents* is a procedure that takes two parameters: the name of a Component palette page and a set of component types. If you are adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

To register a component named *TMyControl* and place it on the Samples page of the palette, you would add the following *Register* procedure to the unit that contains *TMyControl*’s declaration:

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewControl]);
end;
```

This *Register* procedure places *TMyControl* on the Samples page of the Component palette.

Once you register a component, you can compile it into a package (see Chapter 47, “Making components available at design time”) and install it on the Component palette.

## Testing uninstalled components

---

You can test the runtime behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette. For information on testing already installed components, see “Testing installed components” on page 40-14.

You test an uninstalled component by emulating the actions performed by Delphi when the component is selected from the palette and placed on a form.

To test an uninstalled component,

- 1 Add the name of component's unit to the form unit's **uses** clause.
- 2 Add an object field to the form to represent the component.

This is one of the main differences between the way you add components and the way Delphi does it. You add the object field to the public part at the bottom of the form's type declaration. Delphi would add it above, in the part of the type declaration that it manages.

Never add fields to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.

- 3 Attach a handler to the form's *OnCreate* event.
- 4 Construct the component in the form's *OnCreate* handler.

When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You will nearly always pass *Self* as the owner. In a method, *Self* is a reference to the object that contains the method. In this case, in the form's *OnCreate* handler, *Self* refers to the form.

- 5 Assign the *Parent* property.

Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that contains the control visually; usually it is the form on which the control appears, but it might be a group box or panel. Normally, you'll set *Parent* to *Self*, that is, the form. Always set *Parent* before setting other properties of the control.

#### Warning

If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating-system problem.

- 6 Set any other component properties as desired.

Suppose you want to test a new component of type *TMyControl* in a unit named *MyControl*. Create a new project, then follow the steps to end up with a form unit that looks like this:

```
unit Unit1;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MyControl;           { 1. Add NewTest to uses clause }

type
  TForm1 = class(TForm)
  procedure FormCreate(Sender: TObject);           { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
```

## Testing installed components

```
    { Public Declarations }
    MyControl1: TMyControl1;           { 2. Add an object field }
end;

var
    Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
    MyControl1 := TMyControl.Create(Self);           { 4. Construct the component }
    MyControl1.Parent := Self;                       { 5. Set Parent property if component is a control }
    MyControl1.Left := 12;                           { 6. Set other properties... }
    :                                               ...continue as needed }
end;
end.
```

## Testing installed components

---

You can test the design-time behavior of a component after you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette. For information on testing components that have not yet been installed, see “Testing uninstalled components” on page 40-12.

Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.

Test an installed component using a second running instance of Delphi:

- 1 From the Delphi IDE menu select Project | Options | and on the Directories/Conditionals page, set the Debug Source Path to the component’s source file.
- 2 Then select Tools | Debugger Options. On the Language Exceptions, page enable the exceptions you want to track.
- 3 Open the component source file and set breakpoints.
- 4 Select Run | Parameters and set the Host Application field to the name and location of the Delphi executable file.
- 5 In the Run Parameters dialog, click the Load button to start a second instance of Delphi.
- 6 Then drop the components to be tested on the form, which should break on your breakpoints in the source.

# Object-oriented programming for component writers

If you have written applications with Delphi, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in this chapter, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes
- Ancestors, descendants, and class hierarchies
- Controlling access
- Dispatching methods
- Abstract class members
- Classes and pointers

## Defining new classes

---

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple

data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

## Deriving new classes

---

There are two reasons to derive a new class:

- To change class defaults to avoid repetition
- To add new capabilities to a class

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

### To change class defaults to avoid repetition

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Component palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Chapter 48, “Modifying an existing component,” shows an example of changing a component’s default properties.

**Note** If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

### To add new capabilities to a class

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component’s ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Chapter 50, “Customizing a grid,” shows an example of customizing an abstract component class.

## Declaring a new component class

---

In addition to standard components, Delphi provides many abstract classes designed as bases for deriving new components. Table 40.1 on page 40-3 shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component’s unit file.

Here is the declaration of a simple graphical component:

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

A finished component declaration usually includes property, event, and method declarations before the **end**. But a declaration like the one above is also valid, and provides a starting point for the addition of component features.

## Ancestors, descendants, and class hierarchies

---

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, Delphi derives your component from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

## Controlling access

---

There are five levels of *access control*—also called *visibility*—on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

Table 41.1 shows the levels of visibility, from most restrictive to most accessible:

**Table 41.1** Levels of visibility within an object

Visibility	Meaning	Used for
private	Accessible only to code in the unit where the class is defined.	Hiding implementation details.
protected	Accessible to code in the unit(s) where the class and its descendants are defined.	Defining the component writer's interface.
public	Accessible to all code.	Defining the runtime interface.
automated	Accessible to all code. Automation type information is generated.	OLE automation only.
published	Accessible to all code and from the Object Inspector.	Defining the design-time interface.

Declare members as **private** if you want them to be available only within the class where they are defined; declare them as **protected** if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

### Hiding implementation details

---

Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

Here is an example that shows how declaring a field as **private** hides it from application developers. The listing shows two VCL form units. Each form has a handler for its *OnCreate* event which assigns a value to a private field. The compiler allows assignment to the field only in the form where it is declared.

```

unit HideInfo;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;
type
  TSecretForm = class(TForm) { declare new form }
  procedure FormCreate(Sender: TObject);
  private { declare private part }
    FSecretCode: Integer; { declare a private field }
  end;
var
  SecretForm: TSecretForm;
implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42; { this compiles correctly }
end;
end. { end of unit }
unit TestHide; { this is the main form file }
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo; { use the unit with TSecretForm }
type
  TTestForm = class(TForm)
  procedure FormCreate(Sender: TObject);
  end;
var
  TestForm: TTestForm;
implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13; { compiler stops with "Field identifier expected" }
end;
end. { end of unit }

```

Although a program using the *HideInfo* unit can use objects of type *TSecretForm*, it cannot access the *FSecretCode* field in any of those objects.

## Defining the component writer's interface

---

Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files).

You can use **protected** declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

**Note** A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the component that receives the event. As a result, they do not have access to the component's protected methods (unless the component is declared in the same unit as the form).



## Defining the runtime interface

---

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

Here is an example that shows two read-only properties declared as part of a component's runtime interface:

```

type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;           { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;           { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
:
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;

```

## Defining the design-time interface

---

Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Read-only properties cannot be part of the design-time interface because the application developer cannot assign values to them directly. Read-only properties should therefore be public, rather than published.

Here is an example of a published property called *Temperature*. Because it is published, it appears in the Object Inspector at design time.

```

type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;           { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;   { writable! }
  end;

```

# Dispatching methods

---

*Dispatch* refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

## Static methods

---

All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

## An example of static methods

In the following code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods inherited from the first component.

```

type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;           { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;           { this is also different }
  end;

```

## Virtual methods

---

Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime.

To make a method virtual, add the directive **virtual** after the method declaration. The **virtual** directive creates an entry in the object's *virtual method table*, or VMT, which holds the addresses of all the virtual methods in an object type.

When you derive a new class from an existing one, the new class gets its own VMT, which includes all the entries from the ancestor's VMT plus any additional virtual methods declared in the new class.

## Overriding methods

*Overriding* a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods.

To override a method in a descendant class, add the directive **override** to the end of the method declaration.

Overriding a method causes a compilation error if

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.
- The declarations are not otherwise identical (number and type of arguments parameters differ).

The following code shows the declaration of two simple components. The first declares three methods, each with a different kind of dispatching. The other, derived from the first, replaces the static method and overrides the virtual methods.

```

type
  TFirstComponent = class(TCustomControl)
    procedure Move;           { static method }
    procedure Flash; virtual; { virtual method }
    procedure Beep; dynamic;  { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;           { declares new method }
    procedure Flash; override; { overrides inherited method }
    procedure Beep; override; { overrides inherited method }
  end;

```

## Dynamic methods

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a

method is called frequently, or if its execution is time-critical, you should probably declare it as `virtual` rather than `dynamic`.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the directive **dynamic** after the method declaration.

## Abstract class members

---

When a method is declared as **abstract** in an ancestor class, you must surface it (by redeclaring and implementing it) in any descendant component before you can use the new component in applications. Delphi cannot create instances of a class that contains abstract members. For more information about surfacing inherited parts of classes, see Chapter 42, "Creating properties," and Chapter 44, "Creating methods."

## Classes and pointers

---

Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.



## Creating properties

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining properties
- Creating array properties
- Storing and loading properties

### Why create properties?

---

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.)

Properties provide more power than simple fields because

- Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, you let Delphi read the values from the Object Inspector. The Object Inspector also validates property assignments as soon as they are made.

- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a field can be a call to a method which implements elaborate processing.
- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to *True* sets *Down* property of all other speed buttons in its group to *False*.

## Types of properties

---

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

**Table 42.1** How properties appear in the Object Inspector

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value ( <b>true</b> if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these properties and edit them individually. Object properties must descend from <i>TPersistent</i> .
Interface	Properties that are interfaces can appear in the Object Inspector as long as the value is an interface that is implemented by a component (a descendant of <i>TComponent</i> ). Interface properties often have their own property editors.
Array	Array properties must have their own property editors; the Object Inspector has no built-in support for editing them. You can specify a property editor when you register your components.

---

## Publishing inherited properties

---

All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published.

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redefining means adding a declaration for the inherited property to the declaration of the descendant class.

If you derive a VCL component from *TWinControl*, for example, it inherits the protected *DockSite* property. By redeclaring *DockSite* in your new component, you can change the level of protection to either public or published.

The following code shows a redeclaration of *DockSite* as published, making it available at design time.

```

type
  TSampleComponent = class(TWinControl)
    published
      property DockSite;
    end;

```

When you redeclare a property, you specify only the property name, not the type and other information described below in “Defining properties”. You can also declare new default values and specify whether to store the property.

Redeclarations can make a property less restricted, but not more restricted. Thus you can make a protected property public, but you cannot hide a public property by redeclaring it as protected.

## Defining properties

---

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

### The property declaration

---

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.
- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a **published** section of the component’s class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read or set in program code.



Here is a typical declaration for a property called *Count*.

```

type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;           { used for internal storage }
    procedure SetCount (Value: Integer); { write method }
  public
    property Count: Integer read FCount write SetCount;
  end;

```

## Internal data storage

---

There are no restrictions on how you store the data for a property. In general, however, Delphi components follow these conventions:

- Property data is stored in class fields.
- The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these fields consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

## Direct access

---

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage field without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be updated when the property value changes.

The following component-type declaration shows a property that uses direct access for both the **read** and the **write** parts.

```

type
  TSampleComponent = class(TComponent)
  private
    FMyProperty: Boolean;           { internal storage is private }
    { declare field to hold property value }
  published
    { make property available at design time }
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;

```

## Access methods

---

You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same read and write access methods:

```

type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ;

```

Because each element of the date (day, month, and year) is an int, and because setting each requires encoding the date when set, the code avoids duplication by sharing the read and write methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the read method that obtains the date element:

```

function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);           { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

```

This is the write method that sets the appropriate date element:

```

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);        { get current date elements }
    case Index of
      1: AYear := Value;                            { set new element depending on Index }
    end;
  end;
end;

```

```
2: AMonth := Value;
3: ADay := Value;
  else Exit;
end;
FDate := EncodeDate(AYear, AMonth, ADay);      { encode the modified date }
Refresh;                                       { update the visible calendar }
end;
end;
```

### The read method

The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property. For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see “Creating array properties” on page 42-8), both of which pass their index values as parameters. (Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the *Object Pascal Language Guide*.)

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

### The write method

The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. (Use index specifiers to create a single write method that is shared by several properties. For more information about index specifiers, see the *Object Pascal Language Guide*.)

If you do not declare a write method, the property is read-only.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is a simple write method for an integer property called *Count* that stores its current value in a field called *FCount*.

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
    begin
```

```

    FCount := Value;
    Update;
  end;
end;

```

## Default property values

---

When you declare a property, you can specify a *default value* for it. Delphi uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, Delphi always stores the property.

To specify a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value. For example,

```
property Cool Boolean read GetCool write SetCool default True;
```

**Note** Declaring a default value does not set the property to that value. The component's constructor method should initialize property values when appropriate. However, since objects always initialize their fields to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to *False*.

## Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append the **nodefault** directive to the property's declaration. For example,

```
property FavoriteFlavor string nodefault;
```

When you declare a property for the first time, there is no need to include **nodefault**. The absence of a declared default value means that there is no default.

Here is the declaration of a component that includes a single Boolean property called *IsTrue* with a default value of *True*. Below the declaration (in the **implementation** section of the unit) is the constructor that initializes the property.

```

type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
:
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { call the inherited constructor }
  FIsTrue := True;                    { set the default value }
end;

```

## Creating array properties

---

Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be fields.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can reference only individual elements of an array property, not the entire range of the property.

The following example shows the declaration of a property that returns a string based on an integer index.

```

type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
:
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
end;

```

## Creating properties for subcomponents

---

By default, when a property's value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property. However, it is also

possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent.

Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents are saved with the component that creates them. In order for this to work, however, the following conditions must be met:

- The *Owner* of the subcomponent must be the component that creates it and uses it as the value of a published property. For subcomponents that are descendants of *TComponent*, you can accomplish this by setting the *Owner* property of the subcomponent. For other subcomponents, you must override the *GetOwner* method of the persistent object so that it returns the creating component.
- If the subcomponent is a descendant of *TComponent*, it must indicate that it is a subcomponent by calling the *SetSubComponent* method. Typically, this call is made either by the owner when it creates the subcomponent or by the constructor of the subcomponent.

Typically, properties whose values are subcomponents are read-only. If you allow a property whose value is a subcomponent to be changed, the property setter must free the subcomponent when another component is assigned as the property value. In addition, the component often re-instantiates its subcomponent when the property is set to nil. Otherwise, once the property is changed to another component, the subcomponent can never be restored at design time. The following example illustrates such a property setter for a property whose value is a *TTimer*:

```

procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
  if Value <> FTimer then
    begin
      if Value <> nil then
        begin
          if (FTimer <> nil and FTimer.Owner = self) then
            FTimer.Free;
            FTimer := Value;
            FTimer.FreeNotification(self);
          end
        else { nil value }
        begin
          if FTimer.Owner <> self then
            {
              FTimer := TTimer.Create(self);
              FTimer.SetSubComponent(True);
              FTimer.FreeNotification(self);
            }
          end;
        end;
      end;
    end;
  end;

```

Note that the property setter above called the *FreeNotification* method of the component that is set as the property value. This call ensures that the component that is the value of the property sends a notification if it is about to be destroyed. It sends

this notification by calling the *Notification* method. You handle this call by overriding the *Notification* method, as follows:

```

procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = FTimer) then
    FTimer := nil;
end;

```

## Creating properties for interfaces

---

You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In the previous topic, the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don't have access to the component that implements that interface. As a result, you can't call its *FreeNotification* method.

To handle this situation, you can call your component's *ReferenceInterface* method:

```

procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;

```

Calling *ReferenceInterface* with a specified interface does the same thing as calling another component's *FreeNotification* method. Thus, after calling *ReferenceInterface* from the property setter, you can override the *Notification* method to handle the notifications from the implementor of the interface:

```

procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.ImplementorOf(MyIntfProp)) then
    MyIntfProp := nil;
end;

```

Note that the *Notification* code assigns **nil** to the *MyIntfProp* property, not to the private field (*FIntfField*). This ensures that *Notification* calls the property setter, which calls *ReferenceInterface* to remove the notification request that was established when the property value was set previously. All assignments to the interface property must be made through the property setter.

## Storing and loading properties

---

Delphi stores forms and their components in form (.dfm in VCL and .xfm in CLX) files. A form file stores the properties of a form and its components. When Delphi developers add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file.

Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties

### Using the store-and-load mechanism

---

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its public and published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

### Specifying default values

---

Delphi components save their property values only if those values differ from the defaults. If you do not specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value.

To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.



**Note** Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value—that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to *False*, pointers to *nil*, and so on. If there is any doubt, assign a value in the constructor method.

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```

type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;      { override to set new default }
  published
    property Align default alBottom;                       { redeclare with new default value }
  end;
:
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                               { perform inherited initialization }
  Align := alBottom;                                     { assign new default value for Align }
end;

```

## Determining what to store

---

You can control whether Delphi stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property.

To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the name of a Boolean function.

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean function:

```

type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
    :
  published
    property Important: Integer stored True;               { always stored }
    property Unimportant: Integer stored False;           { never stored }
    property Sometimes: Integer stored StoreIt;           { storage depends on function value }
  end;

```

## Initializing after loading

---

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

**Note** The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component.

The following code comes from the *TDatabase* component. After loading, the database tries to reestablish any connections that were open at the time it was stored, and specifies how to handle any exceptions that occur while connecting.

```

procedure TDatabase.Loaded;
begin
  inherited Loaded;           { call the inherited method first}
  try
    if FStreamedConnected then Open           { reestablish connections }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then           { at design time... }
      Application.HandleException(Self)           { let Delphi handle the exception }
    else raise;           { otherwise, reraise }
  end;
end;

```

## Storing and loading unpublished properties

---

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Delphi does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Delphi's automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that tells Delphi how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

- 1 Create methods to store and load the property value.
- 2 Override the *DefineProperties* method, passing those methods to a filer object.

## Creating methods to store and load property values

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Delphi's built-in capabilities for saving and loading simple types. If your property value is built out of types that Delphi knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Delphi knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically created component that is the value of a property named *MyCompProperty*:

```

procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
    if Reader.ReadBoolean then
        MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
    Writer.WriteBoolean(MyCompProperty <> nil);
    if MyCompProperty <> nil then
        Writer.WriteComponent(MyCompProperty);
end;

```

## Overriding the DefineProperties method

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Delphi calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

```

procedure TSampleComponent.DefineProperties(Filer: TFiler);
function DoWrite: Boolean;
begin
  if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
  begin
    if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
      Result := MyCompProperty <> nil
    else if MyCompProperty = nil or
      TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
      Result := True
    else Result := False;
  end
  else { no inherited value -- check for default (nil) value }
    Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;

```



## Creating events

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

Events are implemented as properties, so you should already be familiar with the material in Chapter 42, “Creating properties,” before you attempt to create or change a component’s events.

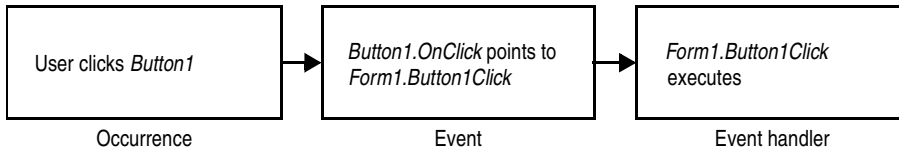
### What are events?

---

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance.

From the application developer’s perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* method. By default, Delphi generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.

To write an event, you need to understand the following:



- Events are method pointers.
- Events are properties.
- Event types are method-pointer types
- Event-handler types are procedures
- Event handlers are optional.

## Events are method pointers

---

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in an instance object. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to an object. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method in a specific instance object. That object is usually the form that contains the component, but it need not be.

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
```

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

## Events are properties

---

Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property.

By convention, the field's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;           { declare a field to hold the method pointer }
    :
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

To learn about *TNotifyEvent* and other event types, see the next section, “Event types are method-pointer types”.

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

## Event types are method-pointer types

---

Because an event is a pointer to an event handler, the type of the event property must be a method-pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of an object.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that is appropriate, or define one of your own.

## Event-handler types are procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer’s code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don’t require the user’s code to change the value.

An example of passing **var** parameters to an event handler is the *OnKeyPress* event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component may want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.



## Event handlers are optional

---

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in “Calling the event” on page 43-8.)

Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component sends numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this should not cause a problem. So the components you create should not require handlers for their events.

Moreover, application developers can write any code they want in an event handler. The components in the VCL and CLX have events written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

## Implementing the standard events

---

The controls that come with Delphi inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events
- Making events visible
- Changing the standard event handling

### Identifying standard events

---

There are two categories of standard events: those defined for all controls and those defined only for the standard windowed controls.

#### Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether windowed, graphical, or custom, inherit these events. The following events are available in all controls:

<i>OnClick</i>	<i>OnDragDrop</i>	<i>OnEndDrag</i>	<i>OnMouseMove</i>
<i>OnDblClick</i>	<i>OnDragOver</i>	<i>OnMouseDown</i>	<i>OnMouseUp</i>

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named *DoEndDrag*.

## Standard events for standard controls

In addition to the events common to all controls, standard windowed controls (those that descend from *TWinControl* in the VCL and *TWidgetControl* in CLX) have the following events:

<i>OnEnter</i>	<i>OnKeyDown</i>	<i>OnKeyPress</i>
<i>OnKeyUp</i>	<i>OnExit</i>	

Like the standard events in *TControl*, the windowed-control events have corresponding methods. The standard key events listed above respond to all normal keystrokes.

**VCL Note** To respond to special keystrokes (such as the Alt key), however, you must respond to the `WM_GETDLGCODE` or `CM_WANTSPECIALKEYS` message from Windows. See Chapter 46, “Handling messages” for information on writing message handlers.

## Making events visible

---

The declarations of the standard events in *TControl* and *TWinControl* (*TWidgetControl* in CLX) are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either public or published.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

For example, to create a component that surfaces the *OnClick* event at design time, you would add the following to the component’s class declaration.

```
type
  TMyControl = class(TCustomControl)
  :
  published
    property OnClick;
  end;
```

## Changing the standard event handling

---

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to them.

Suppose you are writing a component and you want to modify the way it responds to mouse clicks. Instead of assigning a handler to the *OnClick* event as a application developer would, you override the protected method *Click*:

```

procedure click override           { forward declaration }
:
:
procedure TMyControl.Click;
begin
    inherited Click;                 { perform standard handling, including calling handler }
    ...                               { your customizations go here }
end;

```

## Defining your own events

---

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

## Triggering the event

---

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a `WM_LBUTTONDOWN` message to the application. Upon receiving that message, a component calls its `MouseDown` method, which in turn calls any code the user has attached to the `OnMouseDown` event.

But some events are less clearly tied to specific external occurrences. For example, a scroll bar has an `OnChange` event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

### Two kinds of events

There are two kinds of occurrence you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component may need to respond to. State-change events may also be related to messages from Windows (focus changes or enabling, for example), but they can also occur through changes in properties or other code.

You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.

## Defining the handler type

---

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.

### Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type `TNotifyEvent`, which carries only one parameter, the sender of the event. All a handler for a notification “knows” about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

### Event-specific handlers

In some cases, it is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the

handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is likely that the parameters you pass to the event handler come directly from the message parameters.

## Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

## Declaring the event

---

Once you have determined the type of your event handler, you are ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

### Event names start with “On”

The names of most events in Delphi begin with “On.” This is just a convention; the compiler does not enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Developers expect to find events in the alphabetical list of names starting with “On.” Using other kinds of names is likely to confuse them.

**Note** The main exception to this rule is that many events that occur before and after some occurrence begin with “Before” and “After”.

## Calling the event

---

You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid.
- Users can override default handling.

### Empty handlers must be valid

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

An empty handler should produce the same result as no handler at all. So the code for calling an application's event handler should look like this:

```
if Assigned(OnClick) then OnClick(Self);
... { perform default handling }
```

You should *never* have something like this:

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

### Users can override default handling

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

When handling key-press events, for example, application developers can suppress the component's default handling of the keystroke by setting the `var` parameter `Key` to a null character (#0). The logic for supporting this looks like

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ... { perform default handling }
```

The actual code is a little different from this because it deals with Windows messages, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets `Key` to a null character, the component skips the default handling.



## Creating methods

Component methods are procedures and functions built into the structure of a class. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow. These guidelines include

- Avoiding dependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Delphi environment and are accessible at design time.

### Avoiding dependencies

---

At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid



The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if an application calls it when the component is in a bad state, the method corrects the state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

## Naming methods

---

Delphi imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people can use your components.

If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive. Use meaningful verbs.

A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.

- Function names should reflect the nature of what they return.

Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

## Protecting methods

---

All parts of classes, including fields, methods, and properties, have a level of protection or “visibility,” as explained in “Controlling access” on page 41-4. Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

## Methods that should be public

---

Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user.

**Note** Constructors and destructors should always be **public**.

## Methods that should be protected

---

Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

## Abstract methods

---

Sometimes a method is declared as **abstract** in a Delphi component. In the VCL and CLX, abstract methods usually occur in classes whose names begin with “custom,” such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces Component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

## Making methods virtual

---

You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

## Declaring methods

---

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, you do two things:

- Add the declaration to the component's object-type declaration.
- Implement the method in the **implementation** part of the component's unit.

The following code shows a component that defines two new methods, one protected static method and one public virtual method.

```

type
  TSampleComponent = class(TControl)
    protected
      procedure MakeBigger;                { declare protected static method }
    public
      function CalculateArea: Integer; virtual;    { declare public virtual method }
    end;
  :
implementation
  :
  procedure TSampleComponent.MakeBigger;        { implement first method }
  begin
    Height := Height + 5;
    Width := Width + 5;
  end;
  function TSampleComponent.CalculateArea: Integer;    { implement second method }
  begin
    Result := Width * Height;
  end;

```

# Using graphics in components

Windows provides a powerful Graphics Device Interface (GDI) for drawing device-independent graphics. The GDI, however, imposes extra requirements on the programmer, such as managing graphic resources. Delphi takes care of all the GDI drudgery, allowing you to focus on productive work instead of searching for lost handles or unreleased resources.

As with any part of the Windows API, you can call GDI functions directly from your Delphi application. But you will probably find that using Delphi's encapsulation of the graphic functions is faster and easier.

The topics in this section include

- Overview of graphics
- Using the canvas
- Working with pictures
- Off-screen bitmaps
- Responding to changes

## Overview of graphics

---

Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens, brushes, and fonts. After rendering your graphic images, you must restore the device context to its original state before disposing of it.

**CLX Note** GDI functions are Windows-specific and do not apply to CLX or cross-platform applications.

Instead of forcing you to deal with graphics at a detailed level, Delphi provides a simple yet complete interface: your component's *Canvas* property. The canvas ensures that it has a valid device context, and releases the context when you are not

using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all these resources for you, so you need not concern yourself with creating, selecting, and releasing things like pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Delphi manage graphic resources is that it can cache resources for later use, which can speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Delphi caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Delphi uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles thanks to a font cache.

Here are two examples of how simple Delphi's graphics code can be. The first uses standard GDI functions to draw a yellow ellipse outlined in blue on a window, the way you would using other development tools. The second uses a canvas to draw the same ellipse in an application written with Delphi.

```

procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
    PenHandle, OldPenHandle: HPEN;
    BrushHandle, OldBrushHandle: HBRUSH;
begin
    PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));           { create blue pen }
    OldPenHandle := SelectObject(PaintDC, PenHandle);             { tell DC to use blue pen }
    BrushHandle := CreateSolidBrush(RGB(255, 255, 0));            { create a yellow brush }
    OldBrushHandle := SelectObject(PaintDC, BrushHandle);        { tell DC to use yellow brush }
    Ellipse(HDC, 10, 10, 50, 50);                                { draw the ellipse }
    SelectObject(OldBrushHandle);                                 { restore original brush }
    DeleteObject(BrushHandle);                                   { delete yellow brush }
    SelectObject(OldPenHandle);                                 { restore original pen }
    DeleteObject(PenHandle);                                    { destroy blue pen }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
        begin
            Pen.Color := clBlue;                                   { make the pen blue }
            Brush.Color := clYellow;                             { make the brush yellow }
            Ellipse(10, 10, 50, 50);                             { draw the ellipse }
        end;
    end;
end;

```

## Using the canvas

---

The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas; and in the VCL, provides low-level access to the Windows GDI.

Table 45.1 summarizes the capabilities of the canvas.

**Table 45.1** Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as <i>MoveTo</i> , <i>LineTo</i> , <i>Rectangle</i> , and <i>Ellipse</i>
	Displaying and measuring text	<i>TextOut</i> , <i>TextHeight</i> , <i>TextWidth</i> , and <i>TextRect</i> methods
	Filling areas	<i>FillRect</i> and <i>FloodFill</i> methods
Intermediate	Customizing text and graphics	<i>Pen</i> , <i>Brush</i> , and <i>Font</i> properties
	Manipulating pixels	<i>Pixels</i> property.
	Copying and merging images	<i>Draw</i> , <i>StretchDraw</i> , <i>BrushCopy</i> , and <i>CopyRect</i> methods; <i>CopyMode</i> property
Low	Calling Windows GDI functions	<i>Handle</i> property

For detailed information on canvas classes and their methods and properties, see online Help.

## Working with pictures

---

Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling stand-alone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in Delphi:

- Using a picture, graphic, or canvas
- Loading and storing graphics
- Handling palettes

### Using a picture, graphic, or canvas

---

There are three kinds of classes in Delphi that deal with graphics:

- A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.

- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a metafile, or a user-defined graphic type, and an application can access them all in the same way through the picture class. For example, the image control has a property called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas. (The only standard graphic that has a canvas is *TBitmap*.) Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

## Loading and storing graphics

---

All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

**CLX Note** You can also load images from and save them to a Qt MIME source, or a stream object if creating CLX components.

To load an image into a picture from a file, call the picture's *LoadFromFile* method. To save an image from a picture into a file, call the picture's *SaveToFile* method.

*LoadFromFile* and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

To load a bitmap into an image control's picture, for example, pass the name of a bitmap file to the picture's *LoadFromFile* method:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

The picture recognizes *.bmp* as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

## Handling palettes

---

For VCL components, when running on a palette-based device (typically, a 256-color video mode), Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control
- Responding to palette changes

Most controls have no need for a palette, but controls that contain “rich color” graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as *realizing* palettes.

Realizing palettes is the process of ensuring that the foremost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the “real” palette. As windows move in front of one another, Windows continually realizes the palettes.

**Note** Delphi itself provides no specific support for creating or maintaining palettes, other than in bitmaps. If you have a palette handle, however, Delphi controls can manage it for you.

### Specifying a palette for a control

To specify a palette for a VCL control, override the control’s *GetPalette* method to return the handle of the palette.

Specifying the palette for a control does these things for your application:

- It tells the application that your control’s palette needs to be realized.
- It designates the palette to use for realization.

### Responding to palette changes

If your VCL control specifies a palette by overriding *GetPalette*, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*.

The primary role of *PaletteChanged* is to determine whether to realize the control’s palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step further, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.



## Off-screen bitmaps

---

When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in Delphi, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps.
- Copying bitmapped images.

### Creating and managing off-screen bitmaps

---

When creating complex graphic images, you should avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an off-screen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try..finally** block:

```

type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;           { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                       { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;               { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                           { destroy the bitmap object }
  end;
end;

```

### Copying bitmapped images

---

Delphi provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

Table 45.2 summarizes the image-copying methods in canvas objects.

**Table 45.2** Image-copying methods

To create this effect	Call this method
Copy an entire graphic.	Draw
Copy and resize a graphic.	StretchDraw
Copy part of a canvas.	CopyRect
Copy a bitmap with raster operations.	BrushCopy (VCL)
Copy a graphic repeatedly to tile an area.	TiledDraw(CLX)

## Responding to changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes:

```

type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
  :
  implementation
  :
  constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                { construct the pen }
  FPen.OnChange := StyleChanged;      { assign method to OnChange event }
  FBrush := TBrush.Create;           { construct the brush }
  FBrush.OnChange := StyleChanged;   { assign method to OnChange event }
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                       { erase and repaint the component }
end;

```



# Handling messages

One of the keys to traditional Windows programming is handling the *messages* sent by Windows to applications. Delphi handles most of the common ones for you. It is possible, however, that you will need to handle messages that Delphi does not already handle or that you will create your own messages. CLX components do not handle Windows messages but you can create message handlers for your own messages.

There are three aspects to working with messages:

- Understanding the message-handling system
- Changing message handling
- Creating new message handlers

## Understanding the message-handling system

All Delphi classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. (Note that for CLX, the dispatch system does not include `MainWndProc` and `WndProc`.) You should never need to alter this message-dispatch mechanism. All you will need to do is create message-handling methods. See the section “Declaring a new message-handling method” on page 46-7 for more on this subject.

## What's in a Windows message?

---

**Note** This information is applicable when writing VCL components only.

A Windows message is a data record that contains several fields. The most important of these is an integer-size value that identifies the message. Windows defines many messages, and the *Messages* unit declares identifiers for all of them. Other useful information in a message comes in two parameter fields and a result field.

One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as *wParam* and *lParam*, for “word parameter” and “long parameter.” Often, each parameter will contain more than one piece of information, and you see references to names such as *lParamHi*, which refers to the high-order word in the long parameter.

Originally, Windows programmers had to remember or look up in the Windows API what each parameter contained. More recently, Microsoft has named the parameters. This so-called “message cracking” makes it much simpler to understand what information accompanies each message. For example, the parameters to the *WM\_KEYDOWN* message are now called *nVirtKey* and *lKeyData*, which gives much more specific information than *wParam* and *lParam*.

For each type of message, Delphi defines a record type that gives a mnemonic name to each parameter. For example, mouse messages pass the x- and y-coordinates of the mouse event in the long parameter, one in the high-order word, and the other in the low-order word. Using the mouse-message structure, you do not have to worry about which word is which, because you refer to the parameters by the names *XPos* and *YPos* instead of *lParamLo* and *lParamHi*.

## Dispatching messages

---

**Note** This information is applicable when writing VCL components only.

When an application creates a window, it registers a *window procedure* with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that “window” in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure.

Delphi simplifies message dispatching in several ways:

- Each component inherits a complete message-dispatching system.
- The dispatch system has default handling. You define handlers only for messages you need to respond to specially.
- You can modify small parts of the message handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component does not have a handler

defined for the message, the default handling takes care of it, usually by ignoring the message.

## Tracing the flow of messages

Delphi registers a method called *MainWndProc* as the window procedure for each type of component in an application. *MainWndProc* contains an exception-handling block, passing the message structure from Windows to a virtual method called *WndProc* and handling any exceptions by calling the application class's *HandleException* method.

*MainWndProc* is a nonvirtual method that contains no special handling for any particular messages. Customizations take place in *WndProc*, since each component type can override the method to suit its particular needs.

*WndProc* methods check for any special conditions that affect their processing so they can “trap” unwanted messages. For example, while being dragged, components ignore keyboard events, so the *WndProc* method of *TWinControl* passes along keyboard events only if the component is not being dragged. Ultimately, *WndProc* calls *Dispatch*, a nonvirtual method inherited from *TObject*, which determines which method to call to handle the message.

*Dispatch* uses the *Msg* field of the message structure to determine how to dispatch a particular message. If the component defines a handler for that particular message, *Dispatch* calls the method. If the component does not define a handler for that message, *Dispatch* calls *DefaultHandler*.

## Changing message handling

---

**Note** This information is applicable when writing VCL components only.

Before changing the message handling of your components, make sure that is what you really want to do. Delphi translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change message handling in VCL components, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

## Overriding the handler method

---

To change the way a component handles a particular message, you override the message-handling method for that message. If the component does not already handle the particular message, you need to declare a new message-handling method.

To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do *not* use the

**override** directive; you must use the **message** directive and a matching message index.

Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only the message index is significant. For clarity, however, it is best to follow the convention of naming message-handling methods after the messages they handle.

For example, to override a component's handling of the *WM\_PAINT* message, you redeclare the *WMPaint* method:

```
type
  TMyComponent = class(...)
  :
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

## Using message parameters

---

Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the *Result* field for the message: the value returned by the *SendMessage* call that sends the message.

**Note** This information is applicable when writing VCL components only.

Because the type of the *Message* parameter in the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters. If for some reason you need to refer to the message parameters by their old-style names (*WParam*, *LParam*, and so on), you can typecast *Message* to the generic type *TMessage*, which uses those parameter names.

## Trapping messages

---

Under some circumstances, you might want your components to ignore messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message, you override the virtual method *WndProc*.

For VCL components, the *WndProc* method screens messages before passing them to the *Dispatch* method, which in turn determines which method gets to handle the message. By overriding *WndProc*, your component gets a chance to filter out messages before dispatching them. An override of *WndProc* for a control derived from *TWinControl* looks like this:

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);
end;
```

The *TControl* component defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding *WndProc* helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

For CLX, a control might be descended from *TWidgetControl* and you would override *EventFilter* instead of *WndProc*.

Here is part of the *WndProc* method for *TControl*, for example:

```

procedure TControl.WndProc(var Message: TMessage);
begin
  :
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      :                                             { handle others normally }
    end;
  :                                             { otherwise process normally }
end;

```

## Creating new message handlers

---

Because Delphi provides handlers for most common messages, the time you will most likely need to create new message handlers is when you define your own messages. Working with user-defined messages has two aspects:

- Defining your own messages
- Declaring a new message-handling method

CLX components do not handle Windows messages but you can create message handlers for your own messages. Note that you cannot create message handlers for Qt events because they are objects not message IDs.

### Defining your own messages

---

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard messages and notification of state changes. You can define your own messages in both VCL and CLX.

Defining a message is a two-step process. The steps are

- 1 Declaring a message identifier.
- 2 Declaring a message-record type.



## Declaring a message identifier

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant `WM_APP` represents the starting number for user-defined messages. When defining message identifiers, you should base them on `WM_APP`.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the Messages unit to see which messages Windows already defines for that control.

The following code shows two user-defined messages.

```
const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;
```

## Declaring a message-record type

If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message-record is the type of the parameter passed to the message-handling method. If you do not use the message's parameters, or if you want to use the old-style parameter notation (*wParam*, *lParam*, and so on), you can use the default message-record, *TMessage*.

To declare a message-record type, follow these conventions:

- 1 Name the record type after the message, preceded by a *T*.
- 2 Call the first field in the record *Msg*, of type *TMsgParam*.
- 3 Define the next two bytes to correspond to the *Word* parameter, and the next two bytes as unused.

Or

Define the next four bytes to correspond to the *Longint* parameter.

- 4 Add a final field called *Result*, of type *Longint*.

For example, here is the message record for all mouse messages, *TWMMouse*, which uses a variant record to define two sets of names for the same parameters.

```
type
  TWMMouse = record
    Msg: TMsgParam;      ( first is the message ID )
    Keys: Word;         ( this is the wParam )
    case Integer of
      0: {
          XPos: Integer; ( either as x- and y-coordinates... )
          YPos: Integer;
        }
      1: {
          Pos: TPoint;   ( ... or as a single point )
          Result: Longint; ( and finally, the result field )
        }
  end;
```

## Declaring a new message-handling method

---

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that is not already handled by the standard components.
- You have defined your own message for use by your components.

To declare a message-handling method, do the following:

- 1 Declare the method in a **protected** part of the component's class declaration.
- 2 Make the method a procedure.
- 3 Name the method after the message it handles, but without any underline characters.
- 4 Pass a single **var** parameter called *Message*, of the type of the message record.
- 5 Within the message method implementation, write code for any handling specific to the component.
- 6 Call the inherited message handler.

Here is the declaration, for example, of a message handler for a user-defined message called *CM\_CHANGECOLOR*.

```

const
  CM_CHANGECOLOR = WM_APP + 400;

type
  TMyComponent = class(TControl)
  :
protected
  procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;

```



# Making components available at design time

This chapter describes the steps for making the components you create available in the IDE. Making your components available at design time requires several steps:

- Registering components
- Adding palette bitmaps
- Providing Help for your component
- Adding property editors
- Adding component editors
- Compiling components into packages

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see "Installing component packages" on page 11-5.

## Registering components

---

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Component palette.

**Note** If you create your component by choosing Component | New Component in the IDE, the code required to register your component is added automatically.

The steps for manually registering a component are:

- Declaring the Register procedure
- Writing the Register procedure

## Declaring the Register procedure

---

Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Object Pascal) its name is case-sensitive.

The following code shows the outline of a simple unit that creates and registers new components:

```

unit MyBtns;
interface
type
    ...                               { declare your component types here }

procedure Register;                  { this must appear in the interface section }
implementation
    ...                               { component implementation goes here }

procedure Register;
begin
    ...                               { register the components }
end;
end.

```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Component palette. If the unit contains several components, you can register them all in one step.

## Writing the Register procedure

---

Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Component palette. If the unit contains several components, you can register them at the same time.

To register a component, call the *RegisterComponents* procedure once for each page of the Component palette to which you want to add components. *RegisterComponents* involves three important things:

- 1 Specifying the components
- 2 Specifying the palette page
- 3 Using the RegisterComponents function

### Specifying the components

Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);      { two on this page... }
  RegisterComponents('Assorted', [TThird]);                    { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);          { ...and one on the Standard page }
end;
```

## Specifying the palette page

The palette-page name is a string. If the name you give for the palette page does not already exist, Delphi creates a new page with that name. Delphi stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the *LoadStr* function, passing the constant representing the string resource for that page, such as *srSystem* for the System page.

## Using the RegisterComponents function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes two parameters: the name of a Component palette page and the array of component classes.

Set the Page parameter to the name of the page on the component palette where the components should appear. If the named page already exists, the components are added to that page. If the named page does not exist, Delphi creates a new palette page with that name.

Call *RegisterComponents* from the implementation of the *Register* procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the component palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);          {add to system page}
  RegisterComponents('MyCustomPage', [TCustom1, TCustom2]);    { new page}
end;
```

## Adding palette bitmaps

---

Every component needs a bitmap to represent the component on the Component palette. If you don't specify your own bitmap, Delphi uses a default bitmap.

Because the palette bitmaps are needed only at design time, you don't compile them into the component's compilation unit. Instead, you supply them in a Windows resource file with the same name as the unit, but with the extension *.DCR* (dynamic component resource). You can create this resource file using the Image editor in Delphi. Each bitmap should be 24 pixels square.

For each component you want to install, supply a palette bitmap file, and within each palette bitmap file, supply a bitmap for each component you register. The bitmap image has the same name as the component. Keep the palette bitmap file in the same directory with the compiled files, so Delphi can find the bitmaps when it installs the components on the Component palette.

For example, if you create a component named *TMyControl* in a unit named *ToolBox*, you need to create a resource file called *TOOLBOX.DCR* that contains a bitmap called *TMYCONTROL*. The resource names are not case-sensitive, but by convention they are usually in uppercase letters.

## Providing Help for your component

---

When you select a standard component on a form, or a property or event in the Object Inspector, you can press *F1* to get Help on that item. You can provide developers with the same kind of documentation for your components if you create the appropriate Help files.

You can provide a small Help file to describe your components, and your help file becomes part of the user's overall Delphi Help system.

See the section "Creating the Help file" on page 47-4 for information on how to compose the help file for use with a component.

### Creating the Help file

---

You can use any tool you want to create the source file for a Windows Help file (in .rtf format). Delphi includes the Microsoft Help Workshop, which compiles your Help files and provides an online help authoring guide. You can find complete information about creating Help files in the online guide for Help Workshop.

Composing help files for components consists of the steps:

- Creating the entries
- Making component help context-sensitive Adding component help files

#### Creating the entries

To make your component's Help integrate seamlessly with the Help for the rest of the components in the library, observe the following conventions:

##### **1 Each component should have a help topic.**

The component topic should show which unit the component is declared in and briefly describe the component. The component topic should link to secondary windows that describe the component's position in the object hierarchy and list all of its properties, events, and methods. Application developers access this topic by selecting the component on a form and pressing *F1*. For an example of a component topic, place any component on a form and press *F1*.

The component topic must have a # footnote with a value unique to the topic. The # footnote uniquely identifies each topic by the Help system.

The component topic should have a K footnote for keyword searching in the help system Index that includes the name of the component class. For example, the keyword footnote for the *TMemo* component is "TMemo."

The component topic should also have a \$ footnote that provides the title of the topic. The title appears in the Topics Found dialog box, the Bookmark dialog box, and the History window.

## 2 Each component should include the following secondary navigational topics:

- A hierarchy topic with links to every ancestor of the component in the component hierarchy.
- A list of all properties available in the component, with links to entries describing those properties.
- A list of all events available in the component, with links to entries describing those events.
- A list of methods available in the component, with links to entries describing those methods.

Links to object classes, properties, methods, or events in the Delphi help system can be made using Alinks. When linking to an object class, the Alink uses the class name of the object, followed by an underscore and the string "object". For example, to link to the *TCustomPanel* object, use the following:

```
!AL(TCustomPanel_object,1)
```

When linking to a property, method, or event, precede the name of the property, method, or event by the name of the object that implements it and an underscore. For example, to link to the *Text* property which is implemented by *TControl*, use the following:

```
!AL(TControl_Text,1)
```

To see an example of the secondary navigation topics, display the help for any component and click on the links labeled hierarchy, properties, methods, or events.

## 3 Each property, method, and event that is declared within the component should have a topic.

A property, event, or method topic should show the declaration of the item and describe its use. Application developers see these topics either by highlighting the item in the Object Inspector and pressing *F1* or by placing the cursor in the Code editor on the name of the item and pressing *F1*. To see an example of a property topic, select any item in the Object Inspector and press *F1*.

The property, event, and method topics should include a K footnote that lists the name of the property, method, or event, and its name in combination with the name of the component. Thus, the *Text* property of *TControl* has the following K footnote:

```
Text,TControl;TControl,Text;Text,
```

The property, method, and event topics should also include a \$ footnote that indicates the title of the topic, such as *TControl.Text*.



All of these topics should have a topic ID that is unique to the topic, entered as a # footnote.

## Making component help context-sensitive

Each component, property, method, and event topic must have an A footnote. The A footnote is used to display the topic when the user selects a component and presses *F1*, or when a property or event is selected in the Object Inspector and the user presses *F1*. The A footnotes must follow certain naming conventions:

If the Help topic is for a component, the A footnote consists of two entries separated by a semicolon using this syntax:

```
ComponentClass_Object;ComponentClass
```

where *ComponentClass* is the name of the component class.

If the Help topic is for a property or event, the A footnote consists of three entries separated by semicolons using this syntax:

```
ComponentClass_Element;Element_Type;Element
```

where *ComponentClass* is the name of the component class, *Element* is the name of the property, method, or event, and *Type* is the either Property, Method, or Event

For example, for a property named *BackgroundColor* of a component named *TMyGrid*, the A footnote is

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

## Adding component help files

To add your Help file to Delphi, use the OpenHelp utility (called oh.exe) located in the bin directory or accessed using Help | Customize in the IDE.

You will find information about using OpenHelp in the OpenHelp.hlp file, including adding your Help file to the Help system.

## Adding property editors

---

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires these five steps:

- 1 Deriving a property-editor class

- 2 Editing the property as text
- 3 Editing the property as a whole
- 4 Specifying editor attributes
- 5 Registering the property editor

## Deriving a property-editor class

---

Both CLX and the VCL define several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in Table 47.1. The classes in the DesignEditors unit can be used for both VCL and CLX applications. Some of the property-editor classes, however, supply specialized dialogs and so are specialized to either VCL or CLX. These can be found in the WinEditors and CLXEditors units, respectively.

**Note** All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface.

The list in Table 47.1 is not complete. The WinEditors and CLXEditors units also define some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

**Table 47.1** Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from <i>TOrdinalProperty</i> .
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	<i>Char</i> -type and subranges of <i>Char</i> , such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings.
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop-down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

The following example shows the declaration of a simple property editor named *TMyPropertyEditor*:

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

## Editing the property as text

---

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor classes provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits a set of methods used for assigning and reading different sorts of values, as shown in Table 47.2.

**Table 47.2** Methods for reading and writing property values

Property type	Get method	Set method
Floating point	GetFloatValue	SetFloatValue
Method pointer (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue

When you override a *GetValue* method, you will call one of the Get methods, and when you override *SetValue*, you will call one of the Set methods.

### Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns "unknown".

To provide a string representation of your property, override the property editor's *GetValue* method.

If the property is not a string value, *GetValue* must convert the value into a string representation.

### Setting the property value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method. *SetValue* should convert the string and validate the value before calling one of the *Set* methods.

Here are the *GetValue* and *SetValue* methods for *TIntegerProperty*. *Integer* is an ordinal type, so *GetValue* calls *GetOrdValue* and converts the result to a string. *SetValue* converts the string to an integer, performs some range checking, and calls *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // unsigned
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
  end;
end;

procedure TIntegerProperty.SetValue(const Value: string);
  procedure Error(const Args: array of const);
  begin
    raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
  end;
var
  L: Int64;
begin
  L := StrToInt64(Value);
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then
      begin // unsigned compare and reporting needed
        if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
          // bump up to Int64 to get past the %d in the format string
          Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
        end
      else if (L < MinValue) or (L > MaxValue) then
        Error([MinValue, MaxValue]);
        SetOrdValue(L);
      end;
end;
```

The specifics of the particular examples here are less important than the principle: *GetValue* converts the value to a string; *SetValue* converts the string and validates the value before calling one of the “*Set*” methods.

## Editing the property as a whole

---

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

*Edit* methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value “as retrieved.”

When the user clicks the ‘...’ button next to the property or double-clicks the value column, the Object Inspector calls the property editor’s *Edit* method.

Within your implementation of the *Edit* method, follow these steps:

- 1 Construct the editor you are using for the property.
- 2 Read the current value and assign it to the property using a Get method.
- 3 When the user selects a new value, assign that value to the property using a Set method.
- 4 Destroy the editor.

The *Color* properties found in most components use the standard Windows color dialog box as a property editor. Here is the *Edit* method from *TColorProperty*, which invokes the dialog box and uses the result:

```

procedure TColorProperty.Edit;
var
    ColorDialog: TColorDialog;
begin
    ColorDialog := TColorDialog.Create(Application);           { construct the editor }
    try
        ColorDialog.Color := GetOrdValue;                     { use the existing value }
        if ColorDialog.Execute then                          { if the user OKs the dialog... }
            SetOrdValue(ColorDialog.Color);                    { ...use the result to set value }
    finally
        ColorDialog.Free;                                     { destroy the editor }
    end;
end;

```

## Specifying editor attributes

---

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor’s *GetAttributes* method.

*GetAttributes* is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

**Table 47.3** Property-editor attribute flags

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.

**Table 47.3** Property-editor attribute flags (continued)

Flag	Related method	Meaning if included
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.
paFullWidthName	N/A	The value does not need to be displayed. The Object Inspector uses its full width for the property name instead.
paVolatileSubProperties	GetProperties	The Object Inspector refetches the values of all subproperties any time the property value changes.
paReference	GetComponent Value	The value is a reference to something else. When used in conjunction with paSubProperties the referenced object should be displayed as sub properties to this property.

*Color* properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

## Registering the property editor

Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* procedure.

*RegisterPropertyEditor* takes four parameters:

- A type-information pointer for the type of property to edit.  
This is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)`.
- The type of the component to which this editor applies. If this parameter is **nil**, the editor applies to all properties of the given type.

- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;  
begin  
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);  
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',  
    TComponentNameProperty);  
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);  
end;
```

The three statements in this procedure cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you have created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are *nil* and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property (of type *TComponentName*) of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

## Property categories

---

In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit *DesignIntf*.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is included in the *TMiscellaneousCategory* category. Such properties are displayed or hidden in the Object Inspector based on that default categorization.

In addition to these two functions for registering properties, there is an *IsPropertyInCategory* function. This function is useful for creating localization utilities, in which you must determine whether a property is registered in a given property category.

## Registering one property at a time

---

Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation lets you identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

The second variation is much like the first, except that it limits the category to only those properties of the given name that appear on components of a given type. The example below registers (into the 'Help and Hints' category) a property named "HelpContext" of a component of the custom class *TMyButton*.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

The third variation identifies the property using its type rather than its name. The example below registers a property based on its type, *Integer*.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

The final variation uses both the property's type and its name to identify the property. The example below registers a property based on a combination of its type, *TBitmap*, and its name, "Pattern".

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

See the section *Specifying property categories* for a list of the available property categories and a brief description of their uses.

## Registering multiple properties at once

---

Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name "Text" or belongs to a class of type *TEdit* is registered in the category 'Localizable'.

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

The second variation lets you limit the registered properties to those that belong to a specific component. The list of properties to register include only names, not types. For example, the following code registers a number of properties into the 'Help and Hints' category for all components:

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint', 'ParentShowHint', 'ShowHint']);
```



The third variation lets you limit the registered properties to those that have a specific type. As with the second variation, the list of properties to register can include only names:

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

See the section [Specifying property categories](#) for a list of the available property categories and a brief description of their uses.

## Specifying property categories

---

When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in [Table 47.4](#):

**Table 47.4** Property categories

Category	Purpose
<i>Action</i>	Properties related to runtime actions; the <i>Enabled</i> and <i>Hint</i> properties of <i>TEdit</i> are in this category.
<i>Database</i>	Properties related to database operations; the <i>DatabaseName</i> and <i>SQL</i> properties of <i>TQuery</i> are in this category.
<i>Drag, Drop, and Docking</i>	Properties related to drag-n-drop and docking operations; the <i>DragCursor</i> and <i>DragKind</i> properties of <i>TImage</i> are in this category.
<i>Help and Hints</i>	Properties related to using online help or hints; the <i>HelpContext</i> and <i>Hint</i> properties of <i>TMemo</i> are in this category.
<i>Layout</i>	Properties related to the visual display of a control at design-time; the <i>Top</i> and <i>Left</i> properties of <i>TDBEdit</i> are in this category.
<i>Legacy</i>	Properties related to obsolete operations; the <i>Ctl3D</i> and <i>ParentCtl3D</i> properties of <i>TComboBox</i> are in this category.
<i>Linkage</i>	Properties related to associating or linking one component to another; the <i>DataSet</i> property of <i>TDataSource</i> is in this category.
<i>Locale</i>	Properties related to international locales; the <i>BiDiMode</i> and <i>ParentBiDiMode</i> properties of <i>TMainMenu</i> are in this category.
<i>Localizable</i>	Properties that may require modification in localized versions of an application. Many string properties (such as <i>Caption</i> ) are in this category, as are properties that determine the size and position of controls.
<i>Visual</i>	Properties related to the visual display of a control at runtime; the <i>Align</i> and <i>Visible</i> properties of <i>TScrollBar</i> are in this category.
<i>Input</i>	Properties related to the input of data (need not be related to database operations); the <i>Enabled</i> and <i>ReadOnly</i> properties of <i>TEdit</i> are in this category.
<i>Miscellaneous</i>	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the <i>AllowAllUp</i> and <i>Name</i> properties of <i>TSpeedButton</i> are in this category.

## Using the `IsPropertyInCategory` function

---

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the `IsPropertyInCategory` function are available, allowing for different criteria in determining whether a property is in a category.

The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for `IsPropertyInCategory` to return `True`, the property must belong to a `TCustomEdit` descendant, have the name "Text", and be in the property category 'Localizable'.

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

The second variation lets you base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for `IsPropertyInCategory` to return `True`, the property must be a `TCustomEdit` descendant, have the name "Text", and be in the property category 'Localizable'.

```
IsItThere := IsPropertyInCategory('Localizable', 'TCustomEdit', 'Text');
```

## Adding component editors

---

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats.

If you do not give your components a component editor, Delphi uses the default component editor. The default component editor is implemented by the class `TDefaultEditor`. `TDefaultEditor` does not add any new items to a component's context menu. When the component is double-clicked, `TDefaultEditor` searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from `TComponentEditor` and register its use with your component. In your overridden methods, you can use the `Component` property of `TComponentEditor` to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor

## Adding items to the context menu

---

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items
- Implementing commands

### Specifying menu items

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

```
function TMyEditor.GetVerbCount: Integer;
begin
    Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): String;
begin
    case Index of
        0: Result := '&DoThis ...';
        1: Result := 'Do&That';
    end;
end;
```

**Note** Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

### Implementing commands

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous example.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
    MySpecialDialog: TMyDialog;
begin
    case Index of
```

```

0: begin
  MyDialog := TMySpecialDialog.Create(Application);      { instantiate the editor }
  if MySpecialDialog.Execute then;                      { if the user OKs the dialog... }
    MyComponent.FThisProperty := MySpecialDialog.ReturnValue; { ...use the value }
    MySpecialDialog.Free;                               { destroy the editor }
  end;
1: That;                                               { call the That method }
end;
end;
end;

```

## Changing the double-click behavior

---

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example, double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

```

procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
end;

```

**Note** If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of *TComponentEditor*. Then, instead of overriding the *Edit* method, override the protected *TDefaultEditor.EditProperty* method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

```

procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
end;

```

## Adding clipboard formats

---

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Delphi's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Delphi IDE, but can be pasted into other applications.

```

procedure TMyComponent.Copy;
var
    MyFormat : Word;
    AData,APalette : THandle;
begin
    TImage(Component).Picture.Bitmap.SaveToClipboardFormat(MyFormat, AData, APalette);
    Clipboard.SetAsHandle(MyFormat, AData);
end;

```

## Registering the component editor

---

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

Place the call to *RegisterComponentEditor* in the *Register* procedure where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit, the following code registers the component and its association with the component editor.

```

procedure Register;
begin
    RegisterComponents('Miscellaneous', [TMyComponent]);
    RegisterComponentEditor(classes[0], TMyEditor);
end;

```

## Compiling components into packages

---

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Chapter 11, “Working with packages and components”.

To create and compile a package, see “Creating and editing packages” on page 11-6. Put the source-code units for your custom components in the package’s Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see “Installing component packages” on page 11-5.



## Modifying an existing component

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

**Note** To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template* rather than create a new class.

Modifying an existing component takes only two steps:

- Creating and registering the component
- Modifying the component class

### Creating and registering the component

---

Creation of every component begins the same way: you create a unit, derive a component class, register it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Memos*.
- Derive a new component type called *TWrapMemo*, descended from *TMemo*.
- Register *TWrapMemo* on the Samples page of the Component palette.



The resulting unit should look like this:

```
unit Memos;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

If you compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

## Modifying the component class

---

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor.
- Specifying the new default property value.

The constructor actually sets the value of the property. The default tells Delphi what values to store in the form (.dfm for VCL and .xfm for CLX) file. Delphi stores only values that differ from the default, so it is important to perform both steps.

### Overriding the constructor

---

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

**Note** When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see "Overriding methods" on page 41-8.

For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the

constructor override to the forward declaration, then write the new constructor in the **implementation** part of the unit:

```

type
  TWrapMemo = class(TMemo)
  public
    constructor Create(AOwner: TComponent); override; { constructors are always public }
  end;
:
constructor TWrapMemo.Create(AOwner: TComponent); { this goes after implementation }
begin
  inherited Create(AOwner); { ALWAYS do this first! }
  WordWrap := False; { set the new desired value }
end;

```

Now you can install the new component on the Component palette and add it to a form. Note that the *WordWrap* property is now initialized to *False*.

If you change an initial property value, you should also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Delphi cannot store and restore the proper value.

## Specifying the new default property value

---

When Delphi stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Chapter 47, “Making components available at design time.”

To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value. You don’t need to redeclare the entire property, just the name and the default value.

For the word-wrapping memo component, you redeclare the *WordWrap* property in the **published** part of the object declaration, with a default value of *False*:

```

type
  TWrapMemo = class(TMemo)
  :
  published
    property WordWrap default False;
  end;

```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component’s constructor. Redecaring the default ensures that Delphi knows when to write *WordWrap* to the form file.



## Creating a graphic component

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need a window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic component presented in this chapter is *TShape*, the shape component on the Additional page of the Component palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. This chapter calls its shape component *TSampleShape* and shows you all the steps involved in creating the shape component:

- Creating and registering the component
- Publishing inherited properties
- Adding graphic capabilities

### Creating and registering the component

---

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *Shapes*.
- Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
- Register *TSampleShape* on the Samples page of the Component palette.

The resulting unit should look like this:

```

unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.

```

## Publishing inherited properties

---

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in “Publishing inherited properties” on page 42-2 and “Making events visible” on page 43-5. Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the two drag-and-drop properties:

```

type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;           { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;          { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;        { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;

```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

## Adding graphic capabilities

---

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control:

- 1 Determining what to draw.
- 2 Drawing the component image.

In addition, for the shape control example, you will add some properties that enable application developers to customize the appearance of the shape at design time.

### Determining what to draw

---

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

To give your control a property that determines the shape it draws, add a property called *Shape*. This requires

- 1 Declaring the property type.
- 2 Declaring the property.
- 3 Writing the implementation method.

Creating properties is explained in more detail in Chapter 42, “Creating properties.”

### Declaring the property type

When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

Add the following type definition above the shape control class’s declaration.

```
type
    TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
        sstEllipse, sstCircle);
    TSampleShape = class(TGraphicControl) { this is already there }
```

You can now use this type to declare a new property in the class.

## Declaring the property

When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead.

For the shape control, you will declare a field that holds the current shape, then declare a property that reads that field and writes to it through a method call.

Add the following declarations to *TSampleShape*:

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType; { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

Now all that remains is to add the implementation of *SetShape*.

## Writing the implementation method

When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method.

Add the implementation of the *SetShape* method to the **implementation** part of the unit:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then { ignore if this isn't a change }
  begin
    FShape := Value; { store the new value }
    Invalidate; { force a repaint with the new shape }
  end;
end;
```

## Overriding the constructor and destructor

---

To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

## Changing default property values

The default size of a graphic control is fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Chapter 48, “Modifying an existing component.”

In this example, the shape control sets its size to a square 65 pixels on each side.

Add the overridden constructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public
    constructor Create(AOwner: TComponent); override { constructors are always public }
  end; { remember override directive }
```

1 Redeclare the *Height* and *Width* properties with their new default values:

```
type
  TSampleShape = class(TGraphicControl)
  :
  published
    property Height default 65;
    property Width default 65;
  end;
```

2 Write the new constructor in the **implementation** part of the unit:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

## Publishing the pen and brush

---

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires

- 1 Declaring the class fields.
- 2 Declaring the access properties.
- 3 Initializing owned classes.
- 4 Setting owned classes' properties.

### Declaring the class fields

Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor.

Fields for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose.



Add fields for a pen and brush to the shape control:

```

type
  TSampleShape = class(TGraphicControl)
  private
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush; { a field for the brush object }
    :
  end;

```

## Declaring the access properties

You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the class field, but the write part calls a method that enables the component to react to changes in the owned object.

To the shape control, add properties that provide access to the pen and brush fields. You will also declare methods for reacting to changes to the pen or brush.

```

type
  TSampleShape = class(TGraphicControl)
  :
  private
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;

```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```

procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);           { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);           { replace existing pen with parameter }
end;

```

To directly assign the contents of *Value* to *FBrush*...

```
FBrush := Value;
```

...would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

## Initializing owned classes

If you add classes to your component, the component's constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.

Because you have added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

### 1 Construct the pen and brush in the shape control constructor:

```

constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                { construct the pen }
  FBrush := TBrush.Create;           { construct the brush }
end;

```

### 2 Add the overridden destructor to the declaration of the component class:

```

type
  TSampleShape = class(TGraphicControl)
  public                                     { destructors are always public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;          { remember override directive }
  end;

```

### 3 Write the new destructor in the **implementation** part of the unit:

```

destructor TSampleShape.Destroy;
begin
  FPen.Free;                             { destroy the pen object }
  FBrush.Free;                             { destroy the brush object }
  inherited Destroy;                       { always call the inherited destructor, too }
end;

```

## Setting owned classes' properties

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:

```

type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
  :
implementation
  :
  constructor TSampleShape.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner);           { always call the inherited constructor }
    Width := 65;
    Height := 65;
    FPen := TPen.Create;                { construct the pen }
    FPen.OnChange := StyleChanged;     { assign method to OnChange event }
  end;

```

```

    FBrush := TBrush.Create;                                { construct the brush }
    FBrush.OnChange := StyleChanged;                        { assign method to OnChange event }
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
    Invalidate;                                           { erase and repaint the component }
end;

```

With these changes, the component redraws to reflect changes to either the pen or the brush.

## Drawing the component image

---

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user.
- Use the selected shape.
- Adjust coordinates so that squares and circles use the same width and height.

Overriding the *Paint* method requires two steps:

- 1 Add *Paint* to the component's declaration.
- 2 Write the *Paint* method in the **implementation** part of the unit.

For the shape control, add the following declaration to the class declaration:

```

type
    TSampleShape = class(TGraphicControl)
    :
    protected
        procedure Paint; override;
    :
    end;

```

Then write the method in the **implementation** part of the unit:

```

procedure TSampleShape.Paint;
begin
    with Canvas do
    begin
        Pen := FPen;                                { copy the component's pen }
        Brush := FBrush;                            { copy the component's brush }
        case FShape of
            sstRectangle, sstSquare:
                Rectangle(0, 0, Width, Height);    { draw rectangles and squares }
            sstRoundRect, sstRoundSquare:
                RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
            sstCircle, sstEllipse:
                Ellipse(0, 0, Width, Height);      { draw round shapes }
        end;
    end;
end;

```

*Paint* is called whenever the control needs to update its image. Controls are painted when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate*, as the *StyleChanged* method does.

## Refining the shape drawing

---

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is a refined *Paint* method that adjusts for squares and ellipses:

```

procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
    begin
      Pen := FPen;                { copy the component's pen }
      Brush := FBrush;            { copy the component's brush }
      W := Width;                 { use the component width }
      H := Height;                { use the component height }
      if W < H then S := W else S := H;    { save smallest for circles/squares }

      case FShape of                { adjust height, width and position }
        sstRectangle, sstRoundRect, sstEllipse:
          begin
            X := 0;                { origin is top-left for these shapes }
            Y := 0;
          end;
        sstSquare, sstRoundSquare, sstCircle:
          begin
            X := (W - S) div 2;    { center these horizontally... }
            Y := (H - S) div 2;    { ...and vertically }
            W := S;                { use shortest dimension for width... }
            H := S;                { ...and for height }
          end;
      end;

      case FShape of
        sstRectangle, sstSquare:
          Rectangle(X, Y, X + W, Y + H);    { draw rectangles and squares }
        sstRoundRect, sstRoundSquare:
          RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);    { draw rounded shapes }
        sstCircle, sstEllipse:
          Ellipse(X, Y, X + W, Y + H);    { draw round shapes }
      end;
    end;
end;

```



## Customizing a grid

Delphi provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. In this chapter, you will see how to create a small one-month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar involves these tasks:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

The resulting component is similar to the *TCalendar* component on the Samples page of the Component palette.

### Creating and registering the component

---

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *CalSamp*.
- Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
- Register *TSampleCalendar* on the Samples page of the Component palette.

The resulting unit descending from *TCustomGrid* in the VCL should look like this:

```
unit CalSamp;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
  TSampleCalendar = class(TCustomGrid)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

If descending from the CLX version of *TCustomGrid*, only the **uses** clause would differ showing CLX units instead.

If you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

**Note** While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in “Filling in the cells” below.

## Publishing inherited properties

---

The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control.

To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component’s declaration.

For the calendar control, publish the following properties and events, as shown here:

```
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align; { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
```

```

property ParentFont;
property OnClick; { publish events }
property OnDbClick;
property OnDragDrop;
property OnDragOver;
property OnEndDrag;
property OnKeyDown;
property OnKeyPress;
property OnKeyUp;
end;

```

There are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you install the modified calendar component to the Component palette and use it in an application, you will find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

## Changing initial values

---

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual.

Remember that you need to add the constructor to the **public** part of the component's object declaration, then write the new constructor in the **implementation** part of the component's unit. The first statement in the new constructor should always be a call to the inherited constructor.

```

type
  TSampleCalendar = class(TCustomGrid
    public
      constructor Create(AOwner: TComponent); override;
      :
      end;
      :
      constructor TSampleCalendar.Create(AOwner: TComponent);
  begin
    inherited Create(AOwner); { call inherited constructor }
    ColCount := 7; { always seven days/week }
    RowCount := 7; { always six weeks plus the headings }
    FixedCols := 0; { no row labels }
    FixedRows := 1; { one row for day names }
    ScrollBars := ssNone; { no need to scroll }
    Options := Options - [goRangeSelect] + [goDrawFocusSelected]; {disable range selection}
  end;

```



The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

## Resizing the cells

---

- VCL** When a user or application changes the size of a window or control, Windows sends a message called *WM\_SIZE* to the affected window or control so it can adjust any settings needed to later paint its image in the new size. Your VCL component can respond to that message by altering the size of the cells so they all fit inside the boundaries of the control. To respond to the *WM\_SIZE* message, you will add a message-handling method to the component.

Creating a message-handling method is described in detail in “Creating new message handlers” on page 46-5.

In this case, the calendar control needs a response to *WM\_SIZE*, so add a protected method called *WMSize* to the control indexed to the *WM\_SIZE* message, then write the method so that it calculates the proper cell size to allow all cells to be visible in the new size:

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    :
  end;
  :
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;           { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;      { calculate combined size of all lines }
  DefaultColWidth := (Message.Width - GridLines) div 7;    { set new default cell width }
  DefaultRowHeight := (Message.Height - GridLines) div 7;  { and cell height }
end;

```

Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

- CLX** In CLX, changes to the size of a window or control are automatically notified by a call to the protected *BoundsChanged* method. Your CLX component can respond to this notification by altering the size of the cells so they all fit inside the boundaries of the control.

In this case, the calendar control needs to override *BoundsChanged* so that it calculates the proper cell size to allow all cells to be visible in the new size:

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
    :
  end;

```

```

:
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer; { temporary local variable }
begin
  GridLines := 6 * GridLineWidth; { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7; { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7; { and cell height }
  inherited; {now call the inherited method }
end;

```

## Filling in the cells

---

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
    override;
  end;
:
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]); { use RTL strings }
  end;

```

## Tracking the date

---

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. Delphi stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date

- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day

## Storing the internal date

To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.

Adding the internal date to the calendar requires three steps:

- 1 Declare a private field to hold the date:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  ;
```

- 2 Initialize the date field in the constructor:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);          { this is already here }
  ;                                  { other initializations here }
  FDate := Date;                    { get current date from RTL }
end;
```

- 3 Declare a runtime property to allow access to the encoded date.

You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ;
  procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
  begin
    FDate := Value;                  { set new date value }
    Refresh;                         { update the onscreen image }
  end;
```

## Accessing the day, month, and year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

To provide design-time access to the day, month, and year, you do the following:

**1** Declare the three properties, assigning each a unique **index** number:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ;
```

**2** Declare and write the implementation methods, setting different elements for each index value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;           { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ;
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);                       { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                           { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                   { get current date elements }
    case Index of                                           { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);                 { encode the modified date }
    Refresh;                                                  { update the visible calendar }
  end;
end;
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at runtime using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

## Generating the day numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you will need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in a class field, then update that field each time the date changes.

To fill in the days in the proper cells, you do the following:

- 1 Add a month-offset field to the object and a method that updates the field value:

```

type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;           { storage for the offset }
    :
  protected
    procedure UpdateCalendar; virtual; { property for offset access }
  end;
  :
  procedure TSampleCalendar.UpdateCalendar;
  var
    AYear, AMonth, ADay: Word;
    FirstDate: TDateTime;           { date of the first day of the month }
  begin
    if FDate <> 0 then               { only calculate offset if date is valid }
    begin
      DecodeDate(FDate, AYear, AMonth, ADay); { get elements of date }
      FirstDate := EncodeDate(AYear, AMonth, 1); { date of the first }
      FMonthOffset := 2 - DayOfWeek(FirstDate); { generate the offset into the grid }
    end;
    Refresh;                         { always repaint the control }
  end;

```

- 2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```

constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);          { this is already here }
  :                                  { other initializations here }
  UpdateCalendar;                   { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);

```

```

begin
    FDate := Value;                                { this was already here }
    UpdateCalendar;                                { this previously called Refresh }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    :
    FDate := EncodeDate(AYear, AMonth, ADay);      { encode the modified date }
    UpdateCalendar;                                { this previously called Refresh }
end;
end;

```

- 3** Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```

function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
    Result := FMonthOffset + ACol + (ARow - 1) * 7;    { calculate day for this cell }
    if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
        Result := -1;                                  { return -1 if invalid }
    end;
end;

```

Remember to add the declaration of *DayNum* to the component's type declaration.

- 4** Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

```

procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
    TheText: string;
    TempDay: Integer;
begin
    if ARow = 0 then                                  { if this is the header row ... }
        TheText := ShortDayNames[ACol + 1]            { just use the day name }
    else begin
        TheText := '';                                  { blank cell is the default }
        TempDay := DayNum(ACol, ARow);                 { get number for this cell }
        if TempDay <> -1 then TheText := IntToStr(TempDay); { use the number if valid }
    end;
    with ARect, Canvas do
        TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
            Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
    end;
end;

```

Now if you reinstall the calendar component and place one on a form, you will see the proper information for the current month.

## Selecting the current day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    : { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
```

Note that you are now reusing the *ADay* variable previously set by decoding the date.

## Navigating months and years

---

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a “next month” feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year. After incrementing or decrementing *CalendarDate*, decode the date value to fill the *Year*, *Month*, and *Day* properties with corresponding new values.

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
  DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
end;
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

## Navigating days

---

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

The process of navigating days consists of

- Moving the selection
- Providing an OnChange event
- Excluding blank cells

### Moving the selection

---

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*, including the **override** directive afterward.

```

procedure TSampleCalendar.Click;
var
    TempDay: Integer;
begin
    inherited Click;                { remember to call the inherited method! }
    TempDay := DayNum(Col, Row);    { get the day number for the clicked cell }
    if TempDay <> -1 then Day := TempDay;    { change day if valid }
end;

```

### Providing an OnChange event

---

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

Add an *OnChange* event to *TSampleCalendar*.



- 1 Declare the event, a field to store the event, and a dynamic method to call the event:

```

type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
    :
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
    :
  
```

- 2 Write the *Change* method:

```

procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;

```

- 3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                     { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  :                                           { many statements setting element values }
  FDate := EncodeDate(AYear, AMonth, ADay);
  UpdateCalendar;
  Change;                                     { this is new }
end;
end;

```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

## Excluding blank cells

---

As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells.

To control whether a given cell is selectable, override the *SelectCell* method of the grid.

*SelectCell* is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

You can override *SelectCell* to return *False* if the cell does not contain a valid date:

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False           { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);         { otherwise, use inherited value }
end;
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.



## Making a control data aware

When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. Delphi includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Chapter 15, “Using data controls”.

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This chapter first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the *TSampleCalendar* calendar created in Chapter 50, “Customizing a grid”. You can also use the standard calendar control on the Samples page of the Component palette, *TCalendar*.

The chapter then continues with an explanation of how to make the new data-browsing control a data-editing control.

### Creating a data-browsing control

---

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component
- Adding the data link
- Responding to data changes

## Creating and registering the component

---

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *DBCal*.
- Derive a new component class called *TDBCcalendar*, descended from the VCL component *TSampleCalendar*. Chapter 50, “Customizing a grid,” shows you how to create the *TSampleCalendar* component.
- Register *TDBCcalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit DBCal;

interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Grids, Calendar;

type
    TDBCcalendar = class(TSampleCalendar)
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TDBCcalendar]);
end;

end.
```

You can now proceed with making the new calendar a data browser.

## Making the control read-only

---

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves,

- Adding the `ReadOnly` property.
- Allowing needed updates.

Note that if you started with the *TCalendar* component from Delphi’s Samples page instead of *TSampleCalendar*, it already has a `ReadOnly` property, so you can skip these steps.

## Adding the ReadOnly property

By adding a *ReadOnly* property, you will provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unselectable.

- 1 Add the property declaration and a **private** field to hold the value:

```

type
  TDBCcalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;           { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;   { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
:
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor! }
  FReadOnly := True;                  { set the default value }
end;

```

- 2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in “Excluding blank cells” on page 50-12.

```

function TDBCcalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False           { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);   { otherwise, use inherited method }
end;

```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCcalendar*, and append the **override** directive.

If you now add the calendar to a form, you will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

## Allowing needed updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:

```

type
  TDBCcalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;           { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public

```

```

        procedure UpdateCalendar; override;           { remember the override directive }
    end;
    :
function TDBCcalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
    if (not FUpdating) and FReadOnly then Result := False      { allow select if updating }
    else Result := inherited SelectCell(ACol, ARow);          { otherwise, use inherited method }
end;

procedure TDBCcalendar.UpdateCalendar;
begin
    FUpdating := True;                                       { set flag to allow updates }
    try
        inherited UpdateCalendar;                           { update as usual }
    finally
        FUpdating := False;                                  { always clear the flag }
    end;
end;

```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data-browsing ability.

## Adding the data link

---

The connection between a control and a database is handled by a class called a *data link*. The *datalink* class that connects a control with a single field in a database is *TFieldDataLink* (VCL or CLX). There are also data links for entire tables.

A data-aware control *owns* its *datalink* class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Chapter 49, “Creating a graphic component”.

Establishing a data link as an owned class requires these three steps:

- 1 Declaring the class field
- 2 Declaring the access properties
- 3 Initializing the data link

### Declaring the class field

A component needs a field for each of its owned classes, as explained in “Declaring the class fields” on page 49-5. In this case, the calendar needs a field of type *TFieldDataLink* for its data link.

Declare a field for the data link in the calendar:

```

type
    TDBCcalendar = class(TSampleCalendar)
    private
        FDataLink: TFieldDataLink;
    :
    end;

```

Before you can compile the application, you need to add DB and DBCtrls to the unit's **uses** clause.

## Declaring the access properties

Every data-aware control has a *DataSource* property that specifies which data-source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes in the example in Chapter 49, "Creating a graphic component", these access properties do not provide access to the owned classes themselves, but rather to corresponding properties in the owned class. That is, you will create properties that enable the control and its data link to share the same data source and field.

Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as "pass-through" methods to the corresponding properties of the datalink class:

## An example of declaring access properties

```

type
  TDBCcalendar = class(TSampleCalendar)
  private
    ...
    function GetDataField: string;           { returns the name of the data field }
    function GetDataSource: TDataSource;     { returns reference to the data source }
    procedure SetDataField(const Value: string); { assigns name of data field }
    procedure SetDataSource(Value: TDataSource); { assigns new data source }
  published
    { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
:
function TDBCcalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCcalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCcalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCcalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

```



Now that you have established the links between the calendar and its data link, there is one more important step. You must construct the data link class when the calendar control is constructed, and destroy the data link before destroying the calendar.

## Initializing the data link

A data-aware control needs access to its data link throughout its existence, so it must construct the datalink object as part of its own constructor, and destroy the datalink object before it is itself destroyed.

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink object, respectively:

```

type
  TDBCcalendar = class(TSampleCalendar)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    :
  end;
:
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           { always call the inherited constructor first
}
  FDataLink := TFieldDataLink.Create;           { construct the datalink object }
  FDataLink.Control := self;                   {let the datalink know about the calendar }
  FReadOnly := True;                           { this is already here }
end;

destructor TDBCcalendar.Destroy;
begin
  FDataLink.Free;                             { always destroy owned objects first... }
  inherited Destroy;                           { ...then call inherited destructor
}
end;

```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section explains how to do that.

## Responding to data changes

---

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Datalink classes all have events named *OnDataChange*. When the data source indicates a change in its data, the datalink object calls any event handler attached to its *OnDataChange* event.

To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event.

In this case, you will add a method to the calendar, then designate it as the handler for the data link's *OnChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnChange* event in the constructor. In the destructor, detach the *OnChange* handler before destroying the object.

```

type
  TDBCcalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject); { must have proper parameters for event }
  end;
  :
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { always call the inherited constructor first }
  FReadOnly := True; { this is already here }
  FDataLink := TFieldDataLink.Create; { construct the datalink object }
  FDataLink.OnDataChange := DataChange; { attach handler to event }
end;

destructor TDBCcalendar.Destroy;
begin
  FDataLink.OnDataChange := nil; { detach handler before destroying object }
  FDataLink.Free; { always destroy owned objects first... }
  inherited Destroy; { ...then call inherited destructor }
end;

procedure TDBCcalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then { if there is no field assigned... }
  CalendarDate := 0 { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDateTime; { otherwise, set calendar to the date }
end;

```

You now have a data-browsing control.

## Creating a data-editing control

---

When you create a data-editing control, you create and register the component and add the data link just as you do for a data-browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data-editing control described here is the same calendar control described in the first part of the chapter. The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data-editing control involves:

- Changing the default value of `FReadOnly`.
- Handling mouse-down and key-down messages.
- Updating the field datalink class.
- Modifying the `Change` method.
- Updating the dataset.

## Changing the default value of `FReadOnly`

---

Because this is a data-editing control, the `ReadOnly` property should be set to `False` by default. To make the `ReadOnly` property `False`, change the value of `FReadOnly` in the constructor:

```

constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  :
  FReadOnly := False; { set the default value }
  :
end;

```

## Handling mouse-down and key-down messages

---

When the user of the control begins interacting with it, the control receives either mouse-down messages (`WM_LBUTTONDOWN`, `WM_MBUTTONDOWN`, or `WM_RBUTTONDOWN`) or a key-down message (`WM_KEYDOWN`) from Windows. If using CLX, notification is from the operating system in the form of system events. To enable a control to respond to these messages, you must write handlers that respond to these messages.

- Responding to mouse-down messages
- Responding to key-down messages

### Responding to mouse-down messages

A `MouseDown` method is a protected method for a control's `OnMouseDown` event. The control itself calls `MouseDown` in response to a Windows mouse-down message. When you override the inherited `MouseDown` method, you can include code that provides other responses in addition to calling the `OnMouseDown` event.

To override `MouseDown`, add the `MouseDown` method to the `TDBCcalendar` class:

```

type
  TDBCcalendar = class(TSampleCalendar);
  :
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
  :
end;

```

```

procedure TDBCcalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
    begin
      MyMouseDown := OnMouseDown;
      if Assigned(MyMouseDown) then MyMouseDown(Self, Button, Shift, X, Y);
    end;
  end;

```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is *False* and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

## Responding to key-down messages

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override *KeyDown*, follow these steps:

- 1 Add a *KeyDown* method to the *TDBCcalendar* class:

```

type
  TDBCcalendar = class (TSampleCalendar);
  :
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
  :
  end;

```

- 2 Implement the *KeyDown* method:

```

procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
    begin
      MyKeyDown := OnKeyDown;
      if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
    end;
  end;

```

When *KeyDown* responds to a mouse-down message, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is *False*, the key pressed is one of the cursor control keys, and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

## Updating the field datalink class

---

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCcalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field datalink class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

- 1 To reflect a change made to the value in the calendar in the field value, add an *UpdateData* method to the private section of the calendar component:

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
    ;
  end;
```

- 2 Implement the *UpdateData* method:

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;      { set field link to calendar date }
end;
```

- 3 Within the constructor for *TDBCcalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

## Modifying the Change method

---

The *Change* method of the *TDBCcalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code. These are the steps to follow:

- 1 Add a new *Change* method to the *TDBCcalendar* component:

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure Change; override;
    :
  end;
```

- 2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
procedure TDBCcalendar.Change;
begin
  FDataLink.Modified;           { call the Modified method }
  inherited Change;             { call the inherited Change method }
end;
```

## Updating the dataset

---

So far, a change within the data-aware control has changed values in the field *datalink* class. The final step in creating a data-editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the *Tab* key. This process works differently in the VCL and CLX.

- VCL** VCL has defined message control IDs for operations on controls. For example, the *CM\_EXIT* message is sent to the control when the user exits the control. You can write message handlers that respond to the message. In this case, when the user exits the control, the *CMExit* method, the message handler for *CM\_EXIT*, responds by updating the record in the dataset with the changed values in the field *datalink* class. For more information about message handlers, see Chapter 46, “Handling messages.”

To update the dataset within a message handler, follow these steps:

- 1 Add the message handler to the *TDBCcalendar* component:

```
type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWmNoParams); message CM_EXIT;
    :
  end;
```

**2** Implement the *CMExit* method so it looks like this:

```

procedure TDBCcalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;           { tell data link to update database }
  except
    on Exception do SetFocus;       { if it failed, don't let focus leave }
  end;
  inherited;
end;

```

**CLX** In *CLX*, *TWidgetControl* has a protected *DoExit* method that is called when input focus shifts away from the control. This method calls the event handler for the *OnExit* event. You can override this method to update the record in the dataset before generating the *OnExit* event handler.

To update the dataset when the user exits the control, follow these steps:

**1** Add an override for the *DoExit* method to the *TDBCcalendar* component:

```

type
  TDBCcalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
    :
  end;

```

**2** Implement the *DoExit* method so it looks like this:

```

procedure TDBCcalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;           { tell data link to update database }
  except
    on Exception do SetFocus;       { if it failed, don't let focus leave }
  end;
  inherited;                         { let the inherited method generate an OnExit event }
end;

```

## Making a dialog box a component

You will find it convenient to make a frequently used dialog box into a component that you add to the Component palette. Your dialog box components will work just like the components that represent the standard common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time.

Making a dialog box a component requires these steps:

- 1 Defining the component interface
- 2 Creating and registering the component
- 3 Creating the component interface
- 4 Testing the component

The Delphi “wrapper” component associated with the dialog box creates and executes the dialog box at runtime, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this chapter, you will see how to create a wrapper component around the generic About Box form provided in the Delphi Object Repository.

**Note** Copy the files ABOUT.PAS and ABOUT.DFM into your working directory.

There are not many special considerations for designing a dialog box that will be wrapped into a component. Nearly any form can operate as a dialog box in this context.

### Defining the component interface

---

Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it.

For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and



initial control settings, then read back any needed information after the dialog box closes. There is no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component.

The interface must therefore contain enough information that the dialog box form can appear in the way the developer specifies and return any information the application needs. You can think of the properties in the wrapper component as being persistent data for a transient dialog box.

In the case of the About box, you do not need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Because there are four separate fields in the About box that the application might affect, you will provide four string-type properties to provide for them.

## Creating and registering the component

---

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the component palette. This process is outlined in "Creating a new component" on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *AboutDlg*.
- Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*.
- Register *TAboutBoxDlg* on the Samples page of the component palette.

The resulting unit should look like this:

```
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

The new component now has only the capabilities built into *TComponent*. It is the simplest nonvisual component. In the next section, you will create the interface between the component and the dialog box.

# Creating the component interface

---

These are the steps to create the component interface:

- 1 Including the form unit
- 2 Adding interface properties
- 3 Adding the Execute method

## Including the form unit

---

For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the **uses** clause of the wrapper component's unit.

Append *About* to the **uses** clause of the *AboutDlg* unit.

The **uses** clause now looks like this:

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
  About;
```

The form unit always declares an instance of the form class. In the case of the About box, the form class is *TAboutBox*, and the *About* unit includes the following declaration:

```
var
  AboutBox: TAboutBox;
```

So by adding *About* to the **uses** clause, you make *AboutBox* available to the wrapper component.

## Adding interface properties

---

Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's class declaration.

Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you are just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set data at design time so that the wrapper can pass it to the dialog box at runtime.

Declaring an interface property requires two additions to the component's class declaration:

- A private class field, which is a variable the wrapper uses to store the value of the property
- The published property declaration itself, which specifies the name of the property and tells it which field to use for storage

Interface properties of this sort do not need access methods. They use direct access to their stored data. By convention, the class field that stores the property's value has the same name as the property, but with the letter *F* in front. The field and the property *must* be of the same type.

For example, to declare an integer-type interface property called *Year*, you would declare it as follows:

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;           { field to hold the Year-property data }
  published
    property Year: Integer read FYear write FYear;   { property matched with storage }
  end;
```

For this About box, you need four string-type properties—one each for the product name, the version information, the copyright information, and any comments.

```
type
  TAboutBoxDlg = class(TComponent)
  private
    FProductName, FVersion, FCopyright, FComments: string;   { declare fields }
  published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

When you install the component onto the component palette and place the component on a form, you will be able to set the properties, and those values will automatically stay with the form. The wrapper can then use those values when executing the wrapped dialog box.

## Adding the Execute method

---

The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common-dialog-box components, you will use a boolean function called *Execute* that returns *True* if the user clicks OK, or *False* if the user cancels the dialog box.

The declaration for the *Execute* method always looks like this:

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

The minimum implementation for *Execute* needs to construct the dialog box form, show it as a modal dialog box, and return either *True* or *False*, depending on the return value from *ShowModal*.

Here is the minimal *Execute* method for a dialog-box form of type *TMyDialogBox*:

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);           { construct the form }
  try
    Result := (DialogBox.ShowModal = IDOK);              { execute; set result based on how closed }
  finally
    DialogBox.Free;                                       { dispose of the form }
  end;
end;
```

Note the use of a **try..finally** block to ensure that the application disposes of the dialog box object even if an exception occurs. In general, whenever you construct an object this way, you should use a **try..finally** block to protect the block of code and make certain the application frees any resources it allocates.

In practice, there will be more code inside the **try..finally** block. Specifically, before calling *ShowModal*, the wrapper will set some of the dialog box's properties based on the wrapper component's interface properties. After *ShowModal* returns, the wrapper will probably set some of its interface properties based on the outcome of the dialog box execution.

In the case of the About box, you need to use the wrapper component's four interface properties to set the contents of the labels in the About box form. Because the About box does not return any information to the application, there is no need to do anything after calling *ShowModal*. Write the About box wrapper's *Execute* method so that it looks like this:

Within the public part of the *TAboutDlg* class, add the declaration for the *Execute* method:

```
type
  TAboutDlg = class(TComponent)
public
  function Execute: Boolean;
end;

function TAboutBoxDlg.Execute: Boolean;
begin
  AboutBox := TAboutBox.Create(Application);               { construct About box }
  try
    if ProductName = '' then                               { if product name's left blank... }
      ProductName := Application.Title;                   { ...use application title instead }
    AboutBox.ProductName.Caption := ProductName;          { copy product name }
    AboutBox.Version.Caption := Version;                  { copy version info }
    AboutBox.Copyright.Caption := Copyright;              { copy copyright info }
    AboutBox.Comments.Caption := Comments;               { copy comments }
    AboutBox.Caption := 'About ' + ProductName;           { set About-box caption }
    with AboutBox do begin
      ProgramIcon.Picture.Graphic := Application.Icon;    { copy icon }
      Result := (ShowModal = IDOK);                       { execute and set result }
    end;
  finally
    AboutBox.Free;                                         { dispose of About box }
  end;
end;
```

## Testing the component

---

Once you have installed the dialog-box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

For example, if you created an About dialog box, made it a component, and added it to the Component palette, you can test it with the following steps:

- 1 Create a new project.
- 2 Place an About-box component on the main form.
- 3 Place a command button on the form.
- 4 Double-click the command button to create an empty click-event handler.
- 5 In the click-event handler, type the following line of code:

```
AboutBoxDlg1.Execute;
```

- 6 Run the application.

When the main form appears, click the command button. The About box appears with the default project icon and the name Project1. Choose OK to close the dialog box.

You can further test the component by setting the various properties of the About box component and again running the application.

# Index

## Symbols

---

- & (ampersand) character 3-19, 6-33
- ... (ellipsis) buttons 15-21

## A

---

- Abort procedure
  - preventing edits 18-19
- AbortOnKeyViol
  - property 20-51
- AbortOnProblem
  - property 20-51
- About box 52-2, 52-3
  - adding properties 52-4
  - adding to ActiveX controls 38-5
  - executing 52-5
- About unit 52-3
- AboutDlg unit 52-2
- absolute addressing 10-9
- abstract classes 40-3
- accelerators 3-19, 6-33
- Access tables
  - local transactions 20-31
- access violations
  - strings 4-46
- Acquire method 9-7
- Action 6-16
- Action band 6-16
- action bands 6-18
- Action client 6-17
- action editor
  - adding actions 28-4
  - changing actions 28-6
- action items 28-3, 28-4, 28-5 to 28-8
  - adding 28-4
  - caution for changing 28-3
  - chaining 28-8
  - default 28-5, 28-7
  - enabling and disabling 28-6
  - event handlers 28-4
  - page producers and 28-15
  - responding to requests 28-7
  - selecting 28-6, 28-7
  - Web dispatchers 29-17
- Action List editor 6-18
- action lists 3-27, 6-17, 6-18, 6-23 to 6-50
  - Action Manager 6-16, 6-17, 6-18, 6-21
  - Action Manager editor 6-18, 6-19, 6-21
  - Action property 3-18
  - action requests
    - HTML 29-15
  - action responses 29-15
  - ActionBand 6-17
  - ActionLink property 3-18
  - actions 6-23 to 6-50
    - action classes 6-26
    - executing 6-24
    - predefined 6-28
    - registering 6-28
    - updating 6-26
    - Web adapters 29-8
  - Actions property 28-4
  - actions, 6-17
  - activation attribute
    - shared properties 39-6
  - Active Documents 33-10, 33-13 to 33-14
    - See also IOleDocumentSite interface
  - Active property
    - client sockets 32-6
    - datasets 18-4
    - server sockets 32-7
    - sessions 20-17, 20-18
  - active scripting 29-9
  - Active Server Object wizard 37-2 to 37-3
  - Active Server Objects 37-1 to 37-8
    - creating 37-2 to 37-7
    - debugging 37-8
    - in-process servers 37-7
    - out-of-process servers 37-7
    - registering 37-7 to 37-8
  - Active Server Pages 33-10, 33-12 to 33-13, 37-1 to 37-8
    - creating 37-3
    - HTML documents 37-1
    - UI design 37-1
  - ActiveAggs property 23-13
  - ActiveFlag property 16-19
  - ActiveForms 38-1, 38-5 to 38-6
    - as database Web applications 25-33
    - creating 38-2
    - multi-tiered
      - applications 25-32
      - vs. InternetExpress 25-32
    - wizard 38-5 to 38-6
  - ActiveX 33-13, 38-1
    - comparison to ASP 37-7
    - interfaces 33-19
      - vs. InternetExpress 25-32
      - Web applications 33-13, 38-1, 38-15 to 38-16
    - ActiveX controls 13-5, 33-10, 33-13, 33-22, 38-1 to 38-16
      - adding methods 38-8 to 38-9
      - adding properties 38-8 to 38-9
      - component wrappers 35-5, 35-6, 35-8 to 35-9
      - creating 38-2, 38-4 to 38-6
      - data-aware 35-8 to 35-9, 38-7, 38-10 to 38-11
      - debugging 38-14
      - designing 38-4
      - elements 38-2 to 38-3
      - embedding in HTML 28-14
      - event handling 38-9 to 38-10
      - from VCL controls 38-4 to 38-6
      - importing 35-4
      - interfaces 38-7 to 38-11
      - licensing 38-5, 38-6 to 38-7
      - persistent properties 38-12
      - property pages 35-6, 38-3, 38-11 to 38-14
      - registering 38-14
      - threading model 38-5
      - type libraries 33-16, 38-3
      - using Automation-compatible types 38-4, 38-7
      - Web applications 33-13, 38-1, 38-15 to 38-16
      - Web deployment 38-15 to 38-16
      - wizard 38-4 to 38-5
    - ActiveX Data Objects
      - See ADO
    - ActiveX page (Component palette) 3-30, 35-4
    - activities
      - transactional objects 39-17 to 39-18
    - ActnList unit 6-28

- adapter components
  - using 29-13
- adapter dispatcher 29-4
- adapter dispatcher components 29-4
- adapter dispatcher requests 29-15
- AdapterPageProducer 29-3
- adapters
  - actions 29-8
  - errors 29-8
  - fields 29-8
  - records 29-8
  - Web applications 29-8
- Add Fields dialog box 19-4
- Add method
  - menus 6-41
  - persistent columns 15-18
  - strings 3-51
- Add to Interface
  - command 25-17
- Add To Repository
  - command 5-20
- AddAlias method 20-24
- AddFieldDef method 18-38
- AddFontResource
  - function 13-13
- AddIndex method 23-8
- AddIndexDef method 18-38
- Additional page (Component palette) 3-29
- AddObject method 3-52
- AddParam method 18-52
- AddPassword method 20-21
- AddRef method 4-20, 4-24, 4-25, 33-4
- Address property
  - TsocketConnection 25-25
- addresses
  - socket connections 32-3, 32-4
- AddStandardAlias
  - method 20-25
- AddStrings method 3-51
- ADO 14-1, 18-2, 21-1, 21-2
  - components 21-1 to 21-20
    - overview 21-1 to 21-2
  - data stores 21-2, 21-3
  - deployment 13-6
  - implicit transactions 21-6 to 21-7
  - providers 21-3
  - resource dispensers 39-6
- ADO commands 21-7, 21-16 to 21-20
  - asynchronous 21-18
    - canceling 21-18
    - executing 21-17 to 21-18
    - iterating over 17-12
    - parameters 21-19 to 21-20
    - retrieving data 21-18 to 21-19
    - specifying 21-17
- ADO connections 21-2 to 21-8
  - asynchronous 21-5
    - connecting to data stores 21-2 to 21-7
  - events 21-7 to 21-8
  - executing commands 21-5
  - timing out 21-5
- ADO datasets 21-9 to 21-16
  - asynchronous fetching 21-11 to 21-12
    - batch updates 21-12 to 21-14
  - connecting 21-9 to 21-10
  - data files 21-14 to 21-15
  - executing commands 21-20
  - index-based searches 18-27
- ADO objects 21-1
  - Connection object 21-4
  - RDS DataSpace 21-16
  - Recordset 21-9, 21-10
- ADO page (Component palette) 3-29, 14-1, 21-1
- ADOExpress 21-1
- ADT fields 19-22, 19-23 to 19-25
  - displaying 15-21, 19-23
  - flattening 15-21
  - persistent fields 19-24
- ADTG files 21-14
- AfterApplyUpdates
  - event 23-30, 24-8
- AfterCancel event 18-21
- AfterClose event 18-4
- AfterConnect event 17-3, 25-28
- AfterDelete event 18-19
- AfterDisconnect event 17-4, 25-29
- AfterDispatch event 28-5, 28-8
- AfterEdit event 18-17
- AfterGetRecords event 24-7
- AfterInsert event 18-18
- AfterOpen event 18-4
- AfterPost event 18-20
- AfterScroll event 18-5
- AggFields property 23-13
- aggregate fields 19-6, 23-13
  - defining 19-10
  - displaying 19-10
- Aggregates property 23-11, 23-13
- aggregation
  - client datasets 23-11 to 23-13
  - COM 33-9
  - interfaces 4-23
- aliases
  - BDE 20-3, 20-14, 20-24 to 20-26
    - creating 20-24 to 20-25
    - deleting 20-26
    - local 20-25
    - specifying 20-13, 20-14 to 20-15
  - Type Library editor 34-10, 34-17, 34-23
- AliasName property 20-14
- Align property 3-19, 6-4
  - panels 6-43
  - status bars 3-42
  - text controls 7-7
- Alignment property 3-34
  - column headers 15-20
  - data grids 15-19
  - data-aware memo controls 15-9
  - decision grids 16-12
  - fields 19-11
  - memos 3-32
  - rich text controls 3-32
  - status bars 3-42
- AllowAllUp property 3-35
  - speed buttons 6-45
  - tool buttons 6-47
- AllowDelete property 15-27
- AllowGrayed property 3-35
- AllowInsert property 15-27
- alTop constant 6-43
- ampersand (&) character 3-19, 6-33
- analog video 8-32
- ancestor classes 3-5, 3-8, 41-3
  - default 41-3
- Anchor property 3-19
- animation controls 3-45, 8-29 to 8-30
  - example 8-30
- ANSI character sets 4-39, 12-3
- AnsiChar 4-39
- AnsiString 4-41
- Apache applications 27-6
  - creating 28-1, 29-2
  - debugging 27-8
- Apache DLLs 13-9
  - deployment 13-10
- Apache Server DLLs 27-6
  - creating 28-1, 29-2

- apartment threading 36-8 to 36-9
- Append method 18-18, 18-19
  - Insert vs. 18-18
- AppendRecord method 18-21
- application adapter 29-4
- application components, Web applications 29-4
- Application Module Page
  - Options 29-3
- application servers 14-13, 25-1, 25-11 to 25-18
  - callbacks 25-17
  - dropping connections 25-28
  - identifying 25-24
  - interface 25-16 to 25-18
  - interfaces 25-29
  - multiple data modules 25-21 to 25-22
  - opening connections 25-28
  - registering 25-11, 25-22
  - remote data modules 5-19
  - writing 25-12
- Application variable 6-3, 28-3 applications
  - Apache 27-6, 28-1, 29-2
  - bi-directional 12-4
  - CGI stand-alone 29-2
  - client/server 25-1
    - network protocols 20-15
  - COM 5-14, 33-3 to 33-10, 33-18, 36-1 to 36-17
  - creating 3-23
  - cross-platform 10-1 to 10-29
    - creating 10-1
  - database 14-1
    - cross-platform 10-23
  - deploying 13-1
  - files 13-2
  - graphical 40-7, 45-1
  - international 12-1
  - ISAPI 27-6, 28-1, 29-2
  - MDI 5-2
  - MTS 5-14
  - multi-threaded 9-1
  - multi-tiered 25-1 to 25-42
    - overview 25-3 to 25-4
  - NSAPI 27-6, 28-1, 29-2
  - portable 10-1 to 10-29
  - porting 10-17
  - realizing palettes 45-5
  - SDI 5-2
  - service 5-4
  - status information 3-42
  - Web Broker 28-1 to 28-20
  - Web server 5-11, 29-2
  - Web-based client
    - applications 25-31 to 25-42
  - WebSnap 29-1 to 29-26
  - WebSnap tutorial 29-18
  - Win-CGI stand-alone 29-2
- Apply method 20-45
- Apply Updates dialog 34-25
- ApplyRange method 18-34
- ApplyUpdates method 10-28, 20-32
  - BDE datasets 20-35
  - client datasets 21-12, 23-6, 23-19, 23-19 to 23-20, 24-3
  - providers 23-20, 24-3, 24-8
  - TDatabase 20-35
  - TXMLTransformClient 26-10
- AppServer property 23-31, 24-3, 25-17, 25-29
- Arc method 8-4
- architecture
  - BDE-based applications 20-1 to 20-2
  - database applications 14-5 to 14-14, 20-1 to 20-2
    - client 25-4
    - server 25-5
  - multi-tiered 25-4, 25-5
    - Web-based 25-31
  - Web Broker server
    - applications 28-3
- array fields 19-22, 19-25 to 19-26
  - displaying 15-21, 19-23
  - flattening 15-21
  - persistent fields 19-25
- arrays 42-2, 42-8
  - safe 34-13
- as reserved word
  - early binding 25-29
- AS\_ApplyUpdates method 24-3
- AS\_DataRequest method 24-3
- AS\_Execute method 24-3
- AS\_GetParams method 24-3
- AS\_GetProviderNames
  - method 24-3
- AS\_GetRecords method 24-3
- AS\_RowRequest method 24-3
- ASCII tables 20-5
- ASP 33-10, 33-12 to 33-13, 37-1 to 37-8
  - comparison to ActiveX 37-7
  - comparison to Web
    - broker 37-1
  - generating pages 37-3
  - HTML documents 37-1
  - performance limitations 37-1
  - scripting language 33-13, 37-3
  - UI design 37-1
- ASP intrinsics 37-3 to 37-6
  - accessing 37-2 to 37-3
  - Application object 37-4
  - Request object 37-4
  - Response object 37-5
  - Server object 37-6
  - Session object 37-5 to 37-6
- assembler code 10-20
- Assign Local Data
  - command 23-13
- Assign method
  - string lists 3-51
- AssignedValues property 15-21
- assignment statements 42-2
  - object variables 3-10
- AssignValue method 19-16
- Associate property 3-33
- as-soon-as-possible
  - deactivation 25-7, 39-4
- atomicity
  - transactions 14-4, 39-9
- attribute specifications
  - type libraries 34-13
- attributes
  - property editors 47-10
- Attributes property
  - parameters 18-45, 18-51
  - TADODConnection 21-6
- audio clips 8-30
- AutoCalcFields property 18-22
- AutoComplete property 25-7
- AutoDisplay property 15-9, 15-10
- AutoEdit property 15-5
- AutoHotKeys property 6-33
- Automation
  - Active Server Objects 37-2
  - early binding 33-17
  - IDispatch interface 36-14
  - interfaces 36-12 to 36-14
  - late binding 36-14
  - optimizing 33-17
  - type checking 36-13
  - type compatibility 34-11, 36-15 to 36-16
  - type descriptions 33-12
- Automation controllers 33-12, 35-1, 35-12 to 35-15, 36-14
  - creating objects 35-12
  - dispatch interfaces 35-13
  - dual interfaces 35-12



- events 35-13 to 35-15
- example 35-9 to 35-12
- Automation objects 33-12
  - See also* COM objects
  - component wrappers 35-7
    - example 35-9 to 35-12
  - wizard 36-4 to 36-9
- Automation servers 33-10, 33-12
  - See also* COM objects
  - accessing objects 36-14
  - type libraries 33-16
- AutoPopup property 6-49
- AutoSelect property 3-32
- AutoSessionName
  - property 20-17, 20-29, 28-17
- AutoSize property 3-19, 3-31, 6-4, 13-13, 15-8
- averages
  - decision cubes 16-5
- .AVI files 8-32
- AVI clips 3-45, 8-29, 8-32

## B

---

- Background 6-20
- backgrounds 12-9
- Bands property 3-36, 6-48
- base clients 39-2
- base unit 4-59, 4-61
- .bashrc 10-15
- batch files 10-14
- batch operations 20-8, 20-47 to 20-52
  - appending data 20-49
  - copying datasets 20-49
  - deleting records 20-50
  - different databases 20-49
  - error handling 20-51 to 20-52
  - executing 20-51
  - mapping data types 20-50
  - modes 20-8, 20-49
  - setting up 20-47 to 20-48
  - updating data 20-49
- batch updates 21-12 to 21-14
  - applying 21-14
  - canceling 21-14
- BatchMove method 20-8
- BDE
  - See* Borland Database Engine
- BDE Administration
  - utility 20-14, 20-54
- BDE datasets 14-1, 18-2, 20-2 to 20-12
  - applying cached updates 20-35
- batch operations 20-47 to 20-52
  - copying 20-49
  - databases 20-3 to 20-4
  - decision support components and 16-4
  - local database support 20-5 to 20-7
  - sessions 20-3 to 20-4
  - types 20-2
- BDE page (Component palette) 3-29, 14-1
- BeforeApplyUpdates
  - event 23-30, 24-8
- BeforeCancel event 18-21
- BeforeClose event 18-4
- BeforeConnect event 17-3, 25-28
- BeforeDelete event 18-19
- BeforeDisconnect event 17-4, 25-28
- BeforeDispatch event 28-5, 28-7
- BeforeEdit event 18-17
- BeforeGetRecords event 24-7
- BeforeInsert event 18-18
- BeforeOpen event 18-4
- BeforePost event 18-20
- BeforeScroll event 18-5
- BeforeUpdateRecord
  - event 20-32, 20-39, 23-21, 24-11
- BeginDrag method 7-1
- BeginRead method 9-8
- BeginTrans method 17-6
- BeginWrite method 9-8
- Beveled 3-34
- beveled panels 3-45
- BevelKind property 3-22
- bevels 3-45
- bi-directional applications 12-4
  - methods 12-7 to 12-8
  - properties 12-6
- bi-directional cursors 18-48
- bi-directional properties 10-8
- bin directory 10-16
- BinaryOp method 4-30
- bitmap buttons 3-35
- bitmap objects 8-3
- bitmaps 3-44, 8-17 to 8-18, 45-4
  - adding scrollable 8-16
  - adding to components 47-3
  - associating with strings 3-52, 7-12
  - blank 8-17
  - brushes 8-9
  - brushes property 8-8, 8-9
  - destroying 8-21
- drawing on 8-18
- drawing surfaces 45-3
- draw-item events 7-15
- graphical controls vs. 49-3
- in frames 6-15
- internationalizing 12-10
- loading 45-4
- offscreen 45-6 to 45-7
- replacing 8-20
- ScanLine property 8-9
- scrolling 8-17
- setting initial size 8-17
- temporary 8-17
- toolbars 6-46
- when they appear in application 8-2
- BLOB fields 15-2
  - displaying values 15-8, 15-9
  - fetch on demand 24-5
  - getting values 20-4
  - viewing graphics 15-9
- BLOBs 15-8, 15-9
  - caching 20-4
- blocking connections 32-10
  - event handling 32-9
  - non-blocking vs. 32-9
- BlockMode property 32-9, 32-10
- bmBlocking 32-10
- BMPDlg unit 8-20
- bmThreadBlocking 32-9, 32-10
- Bof property 18-6, 18-8
- Bookmark property 18-9
- bookmarks 18-9 to 18-10
  - filtering records 21-10 to 21-11
  - support by dataset types 18-9
- BookmarkValid method 18-9
- Boolean fields 15-2, 15-12
- Boolean values 42-2, 42-12, 51-3
- borders
  - panels 3-40
- BorderWidth property 3-40
- Borland Database Engine 5-10, 14-1, 18-2, 20-1
  - aliases 20-3, 20-14, 20-16, 20-24 to 20-26
  - availability 20-25
  - creating 20-24 to 20-25
  - deleting 20-26
  - heterogeneous queries 20-10
  - specifying 20-13, 20-14 to 20-15
- API calls 20-1, 20-4

- batch operations 20-47 to 20-52
- cached updates 20-32 to 20-47
  - update errors 20-37
- closing connections 20-19
- connecting to
  - databases 20-12 to 20-16
- datasets 20-2
- default connection
  - properties 20-18
- deploying 13-8, 13-14
- driver names 20-14
- drivers 20-1, 20-14
- heterogeneous queries 20-9 to 20-10
- implicit transactions 20-30
- managing connections 20-19 to 20-21
- ODBC drivers 20-16
- opening database
  - connections 20-19
- resource dispensers 39-5
- retrieving data 18-47, 20-2, 20-10
- sessions 20-16
- table types 20-5
- utilities 20-53 to 20-54
- Web applications 13-10
- bounding rectangles 8-11
- .BPL files 11-1, 13-3
- briefcase model 14-14
- brokering connections 25-27
- Brush property 3-44, 8-4, 8-8, 45-3
- BrushCopy method 45-3, 45-7
- brushes 8-8 to 8-9, 49-5
  - bitmap property 8-9
  - changing 49-7
  - colors 8-8
  - styles 8-8
- business rules 25-2, 25-12
  - ASP 37-1
  - transactional objects 39-2
- business-to-business
  - communication
    - XML 30-1
- ButtonAutoSize property 16-10
- buttons 3-34 to 3-35
  - adding to toolbars 6-43 to 6-45, 6-46
  - assigning glyphs to 6-44
  - disabling on toolbars 6-46
  - navigator 15-28
  - toolbars and 6-42

- ButtonStyle property
  - data grids 15-19, 15-20, 15-21
- By Reference Only
  - COM interface
    - properties 34-9
- ByteType 4-44

## C

---

- .CAB files 38-16
- cabinets 38-16
- CacheBlobs property 20-4
- cached updates 23-15 to 23-23
  - ADO 21-12 to 21-14
    - applying 21-14
    - canceling 21-14
  - BDE 20-32 to 20-47
    - applying 20-11, 20-33 to 20-37
      - multiple tables 20-39, 20-43
    - error handling 20-37 to 20-39
    - updating read-only
      - datasets 20-11
- client datasets 14-10 to 14-14, 23-15, 23-19 to 23-23
  - applying 20-11, 23-19 to 23-20
    - multiple tables 20-39, 20-43
  - transactions 17-6
  - update errors 23-22 to 23-23, 24-11
  - updating read-only
    - datasets 20-11
- master/detail
  - relationships 23-17
- overview 23-16 to 23-17
- providers 24-8
- update objects 23-18
- CachedUpdates property 10-28, 20-32
- caching resources 45-2
- calculated fields 18-22 to 18-23, 19-6
  - assigning values 19-7
  - client datasets 23-10 to 23-11
  - defining 19-7 to 19-8
  - lookup fields and 19-9
- calendar components 3-39
- calendars 50-1 to 50-13
  - adding dates 50-5 to 50-10
  - defining properties and
    - events 50-2, 50-6, 50-11
- making read-only 51-2 to 51-4
- moving through 50-10 to 50-13
- resizing 50-4
  - selecting current day 50-9
- call synchronization 39-18
- callbacks
  - multi-tiered
    - applications 25-17
    - limits 25-10
    - transactional objects 39-21
- CanBePooled method 39-8
- Cancel method 18-17, 18-20, 21-18
- Cancel property 3-34
- CancelBatch method 10-29, 21-12, 21-14
- CancelRange method 18-34
- CancelUpdates method 10-29, 20-33, 21-12, 23-6
- CanModify property
  - data grids 15-25
  - datasets 15-5, 18-16, 18-37
  - queries 20-10
- Canvas property 3-45, 40-7
- canvases 40-7, 45-1, 45-3
  - adding shapes 8-11 to 8-12, 8-14
- common properties,
  - methods 8-4
- default drawing tools 49-5
- drawing lines 8-5, 8-10, 8-27 to 8-28
  - changing pen width 8-6
  - event handlers 8-25
- drawing vs. painting 8-4
- overview 8-1 to 8-3
- palettes 45-5
- refreshing the screen 8-2
- Caption property 3-19
- column headers 15-20
- decision grids 16-12
- group boxes and radio
  - groups 3-39
- invalid entries 6-32
- labels 3-41
- TForm 3-41
- cascaded deletes 24-6
- cascaded updates 24-6
- case sensitivity 10-14
  - indexes 23-8
- Cast method 4-29
- CastTo method 4-30
- CDaudio disks 8-32

- CellDrawState function 16-12
- CellRect method 3-43
- cells (grids) 3-43
- Cells function 16-12
- Cells property 3-43
- CellValueArray function 16-12
- CGI applications 13-10
  - creating 29-2
- CGI programs 27-5, 27-6
  - creating 28-2
- change log 23-5, 23-19, 23-33
  - saving changes 23-6
  - undoing changes 23-5
- Change method 51-11
- ChangeBounds property 10-21
- ChangeCount property 10-28, 20-32, 23-5
- ChangedTableName property 20-51
- CHANGEINDEX 23-7
- ChangeScale property 10-21
- Char data type 4-39, 12-3
- character sets 4-43, 12-2, 12-2 to 12-4
  - ANSI 12-3
  - default 12-2
  - international sort orders 12-10
  - multibyte 12-3
  - multibyte conversions 12-3
  - OEM 12-3
- character types 4-39, 12-3
- characters 42-2
- Chart Editing dialog 16-15 to 16-18
- Chart FX 13-5
- check boxes 3-35, 15-2
  - data-aware 15-12 to 15-13
- CHECK constraint 24-12
- Checked property 3-35
- check-list boxes 3-37
- CheckSynchronize routine 9-5
- child controls 3-19
- Chord method 8-4
- circles, drawing 49-9
- circular references 6-2
- class factories 33-6
  - added by wizard 36-3
- class fields 49-4
  - declaring 49-5
  - naming 43-2
- class pointers 41-9
- classes 4-1, 40-2, 40-3, 41-1, 42-2
  - abstract 40-3
  - accessing 41-4 to 41-6, 49-6
  - ancestor 41-3
  - creating 41-1
  - default 41-3
  - defined 41-2
  - defining 40-11
    - static methods and 41-7
    - virtual methods and 41-8
  - derived 41-8
  - deriving new 41-2, 41-8
  - descendant 41-3, 41-8
  - hierarchy 41-3
  - inheritance 41-7
  - instantiating 41-2
  - passing as parameters 41-9
  - properties as 42-2
  - property editors as 47-7
  - protected part 41-5
  - public part 41-6
  - published part 41-6
- Clear method
  - fields 19-16
  - string lists 3-51, 3-52
- ClearSelection method 7-9
- click events 8-24, 8-25, 43-1, 43-2, 43-7
- Click method 43-2
  - overriding 43-6, 50-11
- client applications
  - architecture 25-4
  - as Web server applications 25-31
  - COM 33-3, 33-9, 35-1 to 35-15
  - creating 25-23 to 25-30, 35-1 to 35-15
  - interfaces 32-2
  - multi-tiered 25-2, 25-4
  - network protocols 20-15
  - sockets and 32-1
  - supplying queries 24-6
  - thin 25-2, 25-32
  - transactional objects 39-2
  - type libraries 34-20, 35-2 to 35-6
  - user interfaces 25-1
  - Web Services 31-8 to 31-10
- client connections 32-2, 32-3
  - accepting requests 32-7
  - opening 32-6
  - port numbers 32-5
- client datasets 23-1 to 23-34, 25-3
  - aggregating data 23-11 to 23-13
  - applying updates 23-19 to 23-20
  - calculated fields 23-10 to 23-11
  - connecting to other datasets 14-10 to 14-14, 23-23 to 23-30
  - constraints 23-6 to 23-7, 23-28 to 23-29
    - disabling 23-29
  - copying data 23-13 to 23-14
  - creating tables 23-32
  - deleting indexes 23-9
  - deploying 13-6
  - editing 23-5
  - file-based applications 23-31 to 23-34
  - filtering records 23-2 to 23-5
  - grouping data 23-9 to 23-10
  - index-based searches 18-27
  - indexes 23-7 to 23-10
    - adding 23-8
    - limiting records 23-28
  - loading files 23-32
  - merging changes 23-33
  - merging data 23-14
  - navigation 23-2
  - parameters 23-26 to 23-28
  - providers and 23-23 to 23-30
  - refreshing records 23-29 to 23-30
  - resolving update errors 23-20, 23-22 to 23-23
  - saving changes 23-6
  - saving files 23-33 to 23-34
  - sharing data 23-14
  - specifying providers 23-24 to 23-25
  - supplying queries 23-31
  - switching indexes 23-9
  - types 23-17 to 23-18
  - undoing changes 23-5
  - updating records 23-19 to 23-23
  - with unidirectional datasets 22-10
- client requests 27-4 to 27-6, 28-8
- client sockets 32-3, 32-6 to 32-7
  - assigning hosts 32-4
  - connecting to servers 32-8
  - error messages 32-8
  - event handling 32-8
  - identifying servers 32-6
  - properties 32-6
  - requesting services 32-6

- Windows socket objects 32-6
  - client/server applications 5-10
  - clients *See* client applications
  - clients, action lists 6-17
  - Clipboard 7-8, 7-9, 15-9
    - clearing selection 7-9
    - formats
      - adding 47-15, 47-18
      - graphics and 8-21 to 8-23
      - graphics objects 8-3, 15-9
      - testing contents 7-10
      - testing for images 8-22
  - Clipbrd unit 7-8
  - CloneCursor method 23-14
  - Close method
    - connection components 17-3
    - database connections 20-19
    - datasets 18-4
    - sessions 20-18
  - CloseDatabase method 20-19
  - CloseDataSets method 17-12
  - CLSIDs 33-5, 33-6, 33-15
    - license package file 38-7
  - CLX 3-1
    - applications 10-1
    - deployment 13-6
    - object constructors 10-13
    - vs VCL 10-5
  - clx60.bpl 13-6
  - CM\_EXIT message 51-11
  - CMExit method 51-11
  - CoClasses 33-6
    - ActiveX controls 38-4
    - CLSIDs 33-6
    - component wrappers 35-1, 35-3
      - limitations 35-2
    - creating 33-6, 34-19, 35-5, 35-12
    - declarations 35-5
    - naming 36-3, 36-5
    - Type Library editor 34-9, 34-16, 34-22 to 34-23
    - updating 34-21
  - code 44-4
    - portable 10-15, 10-17
  - Code editor 2-4
    - displaying 47-17
    - event handlers and 3-26
    - opening packages 11-8
    - overview 2-4
  - Code Insight
    - templates 5-3
  - code pages 12-3
  - code templates 5-3
  - ColCount property 15-27
  - color depths 13-11
    - programming for 13-13
  - color grids 8-6
  - Color property 3-19, 3-41, 3-44
    - brushes 8-8
    - column headers 15-20
    - data grids 15-19
    - decision grids 16-12
    - pens 8-5, 8-6
  - ColorChanged property 10-21
  - colors
    - internationalization
      - and 12-10
    - pens 8-6
  - Cols property 3-43
  - column headers 3-41, 15-17, 15-20
  - columns 3-43
    - decision grids 16-11
    - default state 15-15, 15-21
    - deleting 15-16
    - including in HTML
      - tables 28-19
    - persistent 15-15, 15-16 to 15-17
      - creating 15-17 to 15-21
      - deleting 15-18
      - inserting 15-18
      - reordering 15-19
    - properties 15-16, 15-19 to 15-20
      - resetting 15-21
  - Columns editor
    - creating persistent
      - columns 15-17
    - deleting columns 15-18
    - reordering columns 15-19
  - Columns property 3-37, 15-17
    - grids 15-15
    - radio groups 3-39
  - ColWidths property 3-43, 7-14
  - COM 5-14
    - aggregation 33-9
    - applications 33-18
      - distributed 5-14
      - parts 33-3 to 33-10
    - clients 33-3, 33-9, 34-20, 35-1 to 35-15
    - containers 33-9, 35-1
    - controllers 33-9, 35-1
    - definition 33-1 to 33-2
    - early binding 33-16
    - extensions 33-2, 33-10 to 33-12
    - dependencies 33-11
    - interfaces 33-3 to 33-5, 36-3
      - adding to type
        - libraries 34-20
    - Automation 36-12 to 36-14
    - dispatch identifiers 36-14
    - dual interfaces 36-13 to 36-14
    - implementing 33-6, 33-22
    - interface pointer 33-4
    - IUnknown 33-4
    - marshaling 33-8 to 33-9
    - modifying 34-20 to 34-22, 36-9 to 36-12
    - optimizing 33-17
    - type information 33-15
  - overview 33-1 to 33-22
  - proxy 33-7, 33-8
  - specification 33-1, 33-2
  - stubs 33-8
  - technologies 33-11
  - wizards 33-18 to 33-22, 36-1
- COM interfaces, raising
  - exceptions 34-9
- COM library 33-2
- COM objects 33-3, 33-5 to 33-9, 36-1 to 36-17
  - aggregating 33-9
  - component wrappers 35-1, 35-2, 35-3, 35-6 to 35-12
  - creating 36-2 to 36-16
  - debugging 36-17
  - designing 36-2
  - instanting 36-5 to 36-6
  - interfaces 33-3, 36-9 to 36-14
  - lifetime management 4-20
  - registering 36-16 to 36-17
  - threading models 36-6 to 36-9
  - type checking 33-17
  - wizard 36-2 to 36-4, 36-5 to 36-9
- COM servers 33-3, 33-5 to 33-9, 36-1 to 36-17
  - designing 36-2
  - in-process 33-6
  - optimizing 33-17
  - out-of-process 33-7
  - remote 33-7
- COM+ 5-14, 25-6, 33-10, 33-14, 39-1
  - See also* transactional objects
  - call synchronization 39-18
  - configuring activities 39-18

- event objects 39-19
  - creating 39-19
  - wizard 39-19
- events 35-14 to 35-15, 39-18
  - to 39-20
  - firing 39-20
- in-process servers 33-7
- interface pointers 33-5
- object pooling 39-8
- transactional objects 33-14 to 33-15
- transactions 25-18
  - vs. MTS 39-1
- COM+ applications 39-6, 39-22
- COM+ Component Manager 39-23
- combo boxes 3-37, 10-8, 15-2, 15-11
  - data-aware 15-10 to 15-12
  - lookup 15-20
  - owner-draw 7-11
    - measure-item events 7-14
- COMCTL32.DLL 6-43
- command objects 21-16 to 21-20
  - iterating over 17-12
- command switches 10-14
- Command Text editor 18-43
- CommandCount
  - property 17-12, 21-7
- Commands property 17-12, 21-7
- commands, action lists 6-18
- CommandText property 18-43, 21-15, 21-16, 21-17, 21-19, 22-6, 22-7, 23-31
- CommandTimeout
  - property 21-5, 21-18
- CommandType property 21-15, 21-16, 21-17, 22-5, 22-6, 22-7, 23-31
- Commit method 17-8
- CommitTrans method 17-8
- CommitUpdates method 10-29, 20-33, 20-35
- common dialog boxes 3-45, 52-1
  - creating 52-2
  - executing 52-4
- communications 32-1
  - protocols 20-15, 27-2, 32-2
  - standards 27-2
- Compare method 4-32
- CompareBookmarks
  - method 18-9
- CompareOp method 4-33
- compiler directives 10-19
  - libraries 5-9
    - package-specific 11-10
    - strings 4-49
- compiler directives, string and character types 4-49
- compiler options 5-3
- compile-time errors
  - override directive and 41-8
- compiling code 2-4
- component editors 47-15 to 47-18
  - default 47-15
  - registering 47-18
- component interfaces
  - creating 52-3
  - properties, declaring 52-3
- Component palette 3-28
  - ActiveX page 3-30, 35-4
  - adding components 11-6, 47-1, 47-3
  - Additional page 3-29
  - ADO page 3-29, 14-1, 21-1
  - BDE page 3-29, 14-1
  - Data Access page 3-29, 14-2, 25-2
  - Data Controls page 14-15, 15-1, 15-2
  - DataSnap page 3-29, 25-2, 25-6
  - dbDirect page 14-2
  - dbExpress page 3-29, 22-2
  - Decision Cube page 14-15, 16-1
  - Dialogs page 3-30
  - FastNet page 3-29
  - frames 6-14
  - Indy Clients page 3-30
  - Indy Misc page 3-30
  - Indy Servers page 3-30
  - InterBase page 3-29, 14-2
  - Internet page 3-29
  - InternetExpress page 3-29
  - pages listed 3-29
  - QReport page 3-30
  - Samples page 3-30
  - Servers page 3-30
  - Standard page 3-29
  - System page 3-29
  - Win 3.1 page 3-30
  - Win32 page 3-29
- component templates 6-12, 6-13, 41-2
  - and frames 6-14, 6-15
- Component wizard 40-9
- component wrappers 40-4, 52-2
- ActiveX controls 35-6, 35-8
  - to 35-9
- Automation objects 35-7
  - example 35-9 to 35-12
- COM objects 35-1, 35-2, 35-3, 35-6 to 35-12
  - initializing 52-3
- components 3-1, 3-12 to 3-21, 40-1, 41-1, 42-2
  - abstract 40-3
  - adding to Component Palette 47-1
  - adding to existing unit 40-11
  - adding to units 40-11
  - changing 48-1 to 48-3
  - common properties 3-18 to 3-20, 3-21 to 3-22
  - context menus 47-15, 47-16
    - to 47-17
  - creating 40-2, 40-8
  - cross-platform 3-29
  - custom 3-30, 6-12
  - customizing 40-3, 42-1, 43-1
  - data-aware 51-1
  - data-browsing 51-1 to 51-7
  - data-editing 51-7 to 51-11
  - dependencies 40-5
  - derived classes 40-3, 40-11, 49-2
  - dispatcher 29-13
  - double-click 47-15, 47-17
  - grouping 3-39 to 3-41
  - initializing 42-13, 49-6, 51-6
  - installing 3-30, 11-5 to 11-6, 47-19
  - interfaces 41-4, 41-5, 52-1
    - design-time 41-6
    - runtime 41-6
  - memory management 3-11
  - nonvisual 3-46, 40-5, 40-11, 52-2
  - online help 47-4
  - ownership 3-11
  - packages 47-19
  - page producers 29-6
  - palette bitmaps 47-3
  - registering 40-12
  - registration 47-2
  - renaming 3-7 to 3-8
  - resizing 3-34
  - resources, freeing 52-5
  - responding to events 43-6, 43-7, 43-9, 51-6
  - standard 3-28 to 3-30
  - testing 40-12, 40-14, 52-6

- Web applications 29-4
- ComputerName property 25-24
- conditional compilation 10-18, 10-19
- ConfigMode property 20-25
- configuration files 10-15
- connected line segments 8-10
- Connected property 17-3
  - connection components 17-3
- connection components
  - database 14-7 to 14-9, 17-1 to 17-14, 20-3
    - accessing metadata 17-12 to 17-14
    - ADO 21-2 to 21-8
    - BDE 20-12 to 20-16
    - binding 20-13 to 20-15, 21-2 to 21-4, 22-3 to 22-5
    - dbExpress 22-2 to 22-5
    - executing SQL
      - commands 17-10 to 17-11, 21-5
    - implicit 17-2, 20-3, 20-13, 20-19, 20-20, 21-3
    - in remote data
      - modules 25-6
    - statements per
      - connection 22-3
  - DataSnap 14-14, 25-3, 25-4 to 25-5, 25-23, 25-23 to 25-30
    - dropping
      - connections 25-28
    - managing
      - connections 25-28
    - opening
      - connections 25-28
    - protocols 25-8 to 25-11, 25-24
- Connection Editor 22-5
- connection names 22-4 to 22-5
  - changing 22-5
  - defining 22-5
  - deleting 22-5
- connection parameters 20-14 to 20-15
  - ADO 21-3 to 21-4
  - dbExpress 22-4, 22-5
  - login information 17-4, 21-4
- Connection property 21-3, 21-9
- Connection String Editor 21-4
- ConnectionBroker 23-24
- ConnectionString property 22-4
- ConnectionObject property 21-4
- connections
  - client 32-3
  - CORBA 25-10 to 25-11, 25-27
  - database 17-2 to 17-5
    - asynchronous 21-5
    - closing 20-19
    - limiting 25-8
    - managing 20-19 to 20-21
    - naming 22-4 to 22-5
    - network protocols 20-15
    - opening 20-18, 20-19
    - persistent 20-18
    - pooling 25-6, 39-5 to 39-6
    - temporary 20-20
  - database servers 17-3, 20-15
  - DCOM 25-8 to 25-9, 25-24
  - dropping 25-28
  - HTTP 25-9 to 25-10, 25-26
  - opening 25-28, 32-6
  - protocols 25-8 to 25-11, 25-24
  - SOAP 25-10, 25-26
  - TCP/IP 25-9, 25-25, 32-2 to 32-3
    - terminating 32-7
- ConnectionString
  - property 17-2, 17-4, 21-3, 21-10
- ConnectionTimeout
  - property 21-5
- ConnectOptions property 21-5
- consistency
  - transactions 14-4, 39-9
- console applications 5-3
  - CGI 27-6
- Console Wizard 5-3
- CONSTRAINT constraint 24-12
- ConstraintErrorMessage
  - property 19-11, 19-21, 19-22
- constraints
  - controls 6-4
  - data 19-21 to 19-22
    - client datasets 23-6 to 23-7, 23-28 to 23-29
    - creating 19-21
    - disabling 23-29
    - importing 19-21 to 19-22, 23-29, 24-12, 24-13
- Constraints property 6-4, 23-7, 24-13
- constructors 3-11, 40-13, 42-12, 44-3, 50-3, 50-4, 51-6
  - cross-platform 10-22
  - multiple 6-9
  - overriding 48-2
  - owned objects and 49-5, 49-6
- contained objects 33-9
- Contains list (packages) 11-6, 11-7, 11-9, 47-19
- Content method
  - page producers 28-14
- content producers 28-4, 28-13
  - event handling 28-15, 28-16
- Content property
  - Web response objects 28-12
- ContentFromStream method
  - page producers 28-14
- ContentFromString method
  - page producers 28-14
- ContentStream property
  - Web response objects 28-12
- context IDs 5-26
- context menus
  - adding items 47-16 to 47-17
  - Menu designer 6-37
  - toolbars 6-49
- context numbers (Help) 3-42
- ContextHelp 5-29
- controlling Unknown 4-24, 4-26
- controls 3-2, 3-12 to 3-21
  - as ActiveX control
    - implementation 38-3
  - changing 40-3
  - custom 40-4
  - data-aware 15-1 to 15-30
  - data-browsing 51-1 to 51-7
  - data-editing 51-7 to 51-11
  - display options 3-19
  - generating ActiveX
    - controls 38-2, 38-4 to 38-6
  - graphical 45-3, 49-1 to 49-9
    - creating 40-4, 49-3
    - drawing 49-3 to 49-9
    - events 45-7
  - grouping 3-39 to 3-41
  - moving through 3-19, 3-22
  - owner-draw 7-11, 7-13
    - declaring 7-12
  - palettes and 45-5
  - position 3-19
  - receiving focus 40-4
  - repainting 49-7, 49-9, 50-4
  - resizing 45-7, 50-4
  - shape 49-7
  - size 3-19
  - standard
    - displaying data 15-4, 19-17, 19-17
    - windowed 40-3
- ControlType property 16-9, 16-15
- conversion
  - complex 4-60
  - currency 4-62

- conversion class 4-62 to 4-64
  - conversion factor 4-62
  - conversion families 4-59
    - example creating 4-59
  - conversion families, registering 4-60
  - conversion family 4-58
  - conversion utilities 4-58
  - conversions
    - field values 19-16, 19-17 to 19-19
    - PChar 4-47
    - string 4-47
  - Convert function 4-58, 4-59, 4-60, 4-62, 4-64
  - cool bars 3-36, 6-42, 6-43
    - adding 6-48
    - configuring 6-48
    - designing 6-42 to 6-50
    - hiding 6-49
  - coordinates
    - current drawing position 8-24
  - Copy (Object Repository) 5-20
  - CopyFile function 4-53
  - CopyFrom function 4-57
  - CopyMode property 45-3
  - CopyRect method 8-4, 45-3, 45-7
  - CopyToClipboard method 7-9
    - data-aware memo controls 15-9
    - graphics 15-9
  - CORBA 4-17
    - connecting to application servers 25-27
    - multi-tiered database applications 25-10 to 25-11
  - CORBA connections 25-10 to 25-11, 25-27
  - CORBA Data Module wizard 25-15 to 25-16
  - CORBA data modules 25-5
    - instanting 25-16
    - threading models 25-16
  - Count property
    - string lists 3-50
    - TSessionList 20-29
  - Create Data Set command 18-38
  - Create method 3-11
  - Create Submenu command (Menu designer) 6-34, 6-37
  - CREATE TABLE 17-10
  - Create Table command 18-38
  - CreateDataSet method 18-38
  - CreateFile function 4-53
  - CreateObject method 37-3
  - CreateParam method 23-27
  - CreateSharedPropertyGroup 39-6
  - CreateSuspended
    - parameter 9-11
  - CreateTable method 18-38
  - CreateTransactionContextEx
    - example 39-12 to 39-13
  - CreateWidget property 10-22
  - creating a Web page
    - module 29-23
  - creator classes
    - CoClasses 35-5, 35-12
  - critical sections 9-7
    - warning about use 9-8
  - cross-platform
    - applications 3-29, 10-1 to 10-29
    - creating 10-1
  - cross-platform development 6-17, 6-18
  - crosstabs 16-2 to 16-3, 16-10
    - defined 16-2
    - multidimensional 16-3
    - one-dimensional 16-2
    - summary values 16-2, 16-3
  - crtl.dcu 13-6
  - currency
    - formats 12-10
    - internationalizing 12-10
  - currency conversion
    - example 4-62
  - Currency property
    - fields 19-11
  - CursorChanged property 10-21
  - cursor 18-5
    - bi-directional 18-48
    - cloning 23-14
    - linking 18-35, 22-12
    - moving 18-6, 18-7, 18-28, 18-29
      - to first row 18-6, 18-8
      - to last row 18-6, 18-7
      - with conditions 18-10
    - synchronizing 18-40
    - unidirectional 18-48
  - CursorType property 21-12, 21-13
  - CurValue property 24-11
  - custom components 3-30
  - custom controls 40-4
    - libraries 40-4
  - Custom property 25-41
  - custom Variants 4-27 to 4-39
  - binary operations 4-30 to 4-32
  - clearing 4-35
  - comparison operators 4-32 to 4-33
  - copying 4-34
  - creating 4-27, 4-28 to 4-36
  - enabling 4-36
  - loading and saving
    - values 4-35 to 4-36
  - memory 4-34, 4-35
  - methods 4-37 to 4-39
  - properties 4-37 to 4-39
  - storing data 4-28, 4-31, 4-34, 4-35
  - typecasting 4-29 to 4-30, 4-31, 4-32, 4-36
  - unary operators 4-34
  - writing utilities 4-36 to 4-37
  - CustomConstraint
    - property 19-11, 19-21, 23-7
  - customizing components 42-1
  - CutToClipboard method 7-9
    - data-aware memo controls 15-9
    - graphics 15-9
- ## D
- 
- data
    - See also* records
    - accessing 51-1
    - analyzing 14-15, 16-2
    - changing 18-16 to 18-22
    - default values 15-10, 19-20
    - displaying 19-17, 19-17
      - current values 15-8
      - disabling repaints 15-6
      - in grids 15-15, 15-26
    - display-only 15-8
    - entering 18-18
    - formats,
      - internationalizing 12-10
    - graphing 14-15
    - printing 14-16
    - reporting 14-16
    - synchronizing forms 15-4
  - data access
    - components 5-10, 14-1
      - threads 9-4
    - cross-platform 13-7, 14-2
    - mechanisms 5-10, 14-1 to 14-2, 18-2
  - Data Access page (Component palette) 3-29, 14-2, 25-2
  - data binding 38-10

- Data Bindings editor 35-8
- data brokers 23-24, 25-1
- data compression
  - TSocketConnection 25-25
- data constraints *See* constraints
- Data Controls page (Component palette) 3-29, 14-15, 15-1, 15-2
- Data Definition
  - Language 17-10, 18-42, 20-8, 22-10
- Data Dictionary 19-12 to 19-14, 20-52 to 20-53, 25-3
  - constraints 24-13
- data fields 19-6
  - defining 19-6 to 19-7
- data filters 18-12 to 18-15
  - blank fields 18-14
  - client datasets 23-3 to 23-5
    - using parameters 23-28
  - defining 18-13 to 18-15
  - enabling/disabling 18-12
  - operators 18-14
  - queries vs. 18-12
  - setting at runtime 18-15
  - using bookmarks 21-10 to 21-11
- data formats
  - default 19-14
- data grids 15-2, 15-14, 15-15 to 15-26
  - customizing 15-16 to 15-21
  - default state 15-15
    - restoring 15-21
  - displaying data 15-15, 15-16, 15-26
    - ADT fields 15-21
    - array fields 15-21
  - drawing 15-25
  - editing data 15-6, 15-25
  - events 15-25 to 15-26
  - getting values 15-16
  - inserting columns 15-17
  - properties 15-27
  - removing columns 15-16, 15-18
  - reordering columns 15-19
  - runtime options 15-23 to 15-24
- data integrity 14-5, 24-12
- data links 51-4 to 51-6
  - initializing 51-6
- Data Manipulation
  - Language 17-10, 18-42, 20-8
- data members 3-2
- data modules 14-6, 29-6
  - accessing from forms 5-18
  - creating 5-15
  - database components 20-16
  - editing 5-15
  - remote vs. standard 5-15
  - sessions 20-17
  - Web 29-3, 29-5
  - Web applications and 28-2
  - Web Broker
    - applications 28-4
- data packets 26-4
  - application-defined
    - information 23-14, 24-6
  - controlling fields 24-4
  - converting to XML
    - documents 26-1 to 26-8
  - copying 23-13 to 23-14
  - editing 24-7
  - ensuring unique
    - records 24-4
  - fetching 23-25 to 23-26, 24-7
  - including field
    - properties 24-5
  - limiting client edits 24-5
  - mapping to XML
    - documents 26-2
  - read-only 24-5
  - refreshing updated
    - records 24-6
  - XML 25-32, 25-33, 25-36
    - editing 25-37
    - fetching 25-36 to 25-37
- Data property 23-5, 23-13, 23-14, 23-33
- data sources 14-6, 15-3 to 15-5
  - disabling 15-4
  - enabling 15-4
  - events 15-4
- data stores 21-2
- data types
  - persistent fields 19-6
- data-aware controls 14-15, 15-1 to 15-30, 19-17, 51-1
  - associating with
    - datasets 15-3 to 15-4
  - common features 15-2
  - creating 51-1 to 51-12
  - data-browsing 51-1 to 51-7
  - data-editing 51-7 to 51-11
  - destroying 51-6
  - disabling repaints 15-6, 18-8
  - displaying data 15-6 to 15-7
    - current values 15-8
    - in grids 15-15, 15-26
  - displaying graphics 15-9
  - editing 15-5 to 15-6, 18-17
  - entering data 19-14
  - grids 15-14
  - inserting records 18-18
  - list 15-2
  - read-only 15-8
  - refreshing data 15-6
  - representing fields 15-7
  - responding to changes 51-6
- database applications 5-10, 14-1
  - architecture 14-5 to 14-14, 25-31
  - cross-platform 10-23
  - deployment 13-6
  - distributed 5-11
  - file-based 14-9 to 14-10, 21-14 to 21-15, 23-31 to 23-34
  - multi-tiered 25-3 to 25-4
  - porting 10-25
  - scaling 14-11
  - XML and 26-1 to 26-10
- database components 5-10, 20-3, 20-12 to 20-16
  - applying cached
    - updates 20-35
  - identifying databases 20-13 to 20-15
  - sessions and 20-13, 20-20 to 20-21
  - shared 20-16
  - temporary 20-20
    - dropping 20-20
- database connections 17-2 to 17-5
  - dropping 17-3, 17-3 to 17-4
  - limiting 25-8
  - maintaining 17-3
  - persistent 20-18
  - pooling 25-6, 39-5 to 39-6
- Database Desktop 20-54
- database drivers
  - BDE 20-1, 20-3, 20-14
  - dbExpress 22-3
- database engines
  - See also* Borland Database Engine
  - third-party 13-7
- Database Explorer 20-14, 20-53
- database management
  - systems 25-1
- database navigator 15-2, 15-28 to 15-30, 18-5, 18-6
  - buttons 15-28
  - deleting data 18-20



- editing 18-17
  - enabling/disabling
    - buttons 15-28, 15-29
  - help hints 15-30
- Database parameter 22-4
- Database Properties
  - editor 20-14
  - viewing connection
    - parameters 20-15
- database servers 5-10, 17-3, 20-15
  - connecting 14-7 to 14-9
  - constraints 19-21, 19-21 to 19-22, 24-12
  - describing 17-2
  - types 14-2
- DatabaseCount property 20-21
- DatabaseName property 17-2, 20-3, 20-14
  - heterogenous queries 20-9
- databases 5-10, 14-1 to 14-5, 51-1
  - access properties 51-5 to 51-6
  - accessing 18-1
  - adding data 18-21
  - aliases and 20-14
  - choosing 14-3
  - connecting 17-1 to 17-14
  - file-based 14-3
  - generating HTML
    - responses 28-17 to 28-20
  - identifying 20-13 to 20-15
  - implicit connections 17-2
  - logging in 14-4, 17-4 to 17-5
  - naming 20-14
  - relational 14-1
  - security 14-3 to 14-4
  - transactions 14-4 to 14-5
  - types 14-2
  - unauthorized access 17-4
  - Web applications and 28-17
- Databases property 20-21
- DataChange method 51-10
- DataCLX 10-6
- data-entry validation 19-15
- DataField property 15-10, 51-5
  - lookup list and combo boxes 15-12
- DataRequest method 23-30, 24-3
- dataset fields 19-22, 19-26 to 19-27
  - displaying 15-23
  - persistent 18-36
- dataset page producers 28-18
  - converting field values 28-18
- DataSet property
  - data grids 15-16
  - providers 24-2
- dataset providers 14-11
  - See* providers
- DataSetCount property 17-12
- DataSetField property 18-36
- datasets 14-7, 18-1 to 18-53
  - adding records 18-18 to 18-19, 18-21
  - ADO-based 21-9 to 21-16
  - BDE-based 20-2 to 20-12
  - categories 18-23 to 18-24
  - changing data 18-16 to 18-22
  - closing 18-4 to 18-5
    - posting records 18-20
    - w/o disconnecting 17-12
  - creating 18-37 to 18-39
  - current row 18-5
  - cursors 18-5
  - custom 18-2
  - decision components
    - and 16-4 to 16-6
  - deleting records 18-19 to 18-20
  - editing 18-17 to 18-18
  - fields 18-1
  - filtering records 18-12 to 18-15
  - HTML documents 28-19, 28-20
  - iterating over 17-12
  - marking records 18-9 to 18-10
  - modes 18-3 to 18-4
  - navigating 15-28, 18-5 to 18-8, 18-16
  - opening 18-4
  - posting records 18-20
  - providers and 24-2
  - queries 18-23, 18-41 to 18-48
  - read-only
    - updating 20-11
  - searching 18-10 to 18-12
    - extending a search 18-29
    - multiple columns 18-11
    - partial keys 18-29
    - using indexes 18-11, 18-12, 18-27 to 18-29
  - states 18-3 to 18-4
  - stored procedures 18-23, 18-48 to 18-53
  - tables 18-23, 18-24 to 18-41
- undoing changes 18-20 to 18-21
- unidirectional 22-1 to 22-18
- unindexed 18-21
- DataSets property 17-12
- DataSnap page (Component palette) 3-29, 25-2, 25-6
- DataSource property
  - ActiveX controls 35-8
  - data grids 15-16
  - data navigators 15-30
  - data-aware controls 51-5
  - lookup list and combo boxes 15-12
  - queries 18-46
- DataType property
  - parameters 18-44, 18-45, 18-51
- date fields
  - formatting 19-14
- dates
  - calendar components 3-39
  - entering 3-39
  - internationalizing 12-10
- DateTimePicker
  - component 3-39
- DAX 33-2, 33-21 to 33-22
- Day property 50-5
- DB/2 driver
  - deploying 13-9
- dBASE tables 20-5
  - accessing data 20-9
  - adding records 18-19
  - DatabaseName 20-3
  - indexes 20-6
  - local transactions 20-31
  - password protection 20-21 to 20-23
  - renaming 20-7
- DBChart component 14-15
- DBCheckBox component 15-2, 15-12 to 15-13
- DBComboBox component 15-2, 15-10 to 15-11
- DBConnection property 23-16
- DBCtrlGrid component 15-2, 15-26 to 15-27
  - properties 15-27
- dbDirect page (Component palette) 14-2
- DBEdit component 15-2, 15-8
- dbExpress 10-23 to 10-28, 13-7, 14-2, 22-1 to 22-2
  - components 22-1 to 22-18
  - debugging 22-17 to 22-18

- deploying 22-1
- drivers 22-3
- metadata 22-12 to 22-17
- dbExpress applications 13-10
- dbExpress page (Component palette) 3-29, 22-2
- DBGrid component 15-2, 15-15 to 15-26
  - events 15-25
  - properties 15-19
- DBGridColumn component 15-15
- DBImage component 15-2, 15-9 to 15-10
- DBListBox component 15-2, 15-10 to 15-11
- DBLogDlg unit 17-4
- DBLookupComboBox component 15-2, 15-11 to 15-12
- DBLookupListBox component 15-2, 15-11 to 15-12
- DBMemo component 15-2, 15-8 to 15-9
- DBMS 25-1
- DBNavigator component 15-2, 15-28 to 15-30
- DBRadioGroup component 15-2, 15-13 to 15-14
- DBRichEdit component 15-2, 15-9
- DBSession property 20-3
- DBText component 15-2, 15-8
- dbxconnections.ini 22-4, 22-5
- dbxdrivers.ini 22-3
- DCOM 33-7, 33-8
  - connecting to application server 23-25, 25-24
  - distributing applications 5-14
  - InternetExpress applications 25-36
  - multi-tiered applications 25-8 to 25-9
- DCOM connections 25-8 to 25-9, 25-24
- DCOMCnfg.exe 25-36
- .DCP files 11-2, 11-12
- .DCR files 47-3
- .DCU files 11-2, 11-12
- DDL 17-10, 18-42, 18-47, 20-8, 22-10
- debugging
  - Active Server Objects 37-8
  - ActiveX controls 38-14
  - COM objects 36-17
  - dbExpress applications 22-17 to 22-18
  - transactional objects 39-21 to 39-22
  - Web server applications 27-7 to 27-9, 29-2
- debugging code 2-5
- debugging Web server applications 28-2
- Decision Cube Editor 16-7 to 16-8
  - Cube Capacity 16-19
  - Dimension Settings 16-8
  - Memory Control 16-8
- Decision Cube page (Component palette) 14-15, 16-1
- decision cubes 16-7 to 16-8
  - design options 16-8
  - dimension maps 16-5, 16-7, 16-8, 16-19
  - dimensions
    - opening/closing 16-9
    - paged 16-20
  - displaying data 16-9, 16-11
  - drilling down 16-5, 16-9, 16-11, 16-20
  - getting data 16-4
  - memory management 16-8
  - pivoting 16-5, 16-9
  - properties 16-7
  - refreshing 16-7
  - subtotals 16-5
- decision datasets 16-4 to 16-6
- decision graphs 16-13 to 16-18
  - customizing 16-15 to 16-18
  - data series 16-17 to 16-18
  - dimensions 16-14
  - display options 16-15
  - graph types 16-16
  - pivot states 16-9
  - runtime behaviors 16-19
  - templates 16-16
- decision grids 16-10 to 16-13
  - dimensions
    - drilling down 16-11
    - opening/closing 16-11
    - reordering 16-11
    - selecting 16-12
  - events 16-12
  - pivot states 16-9, 16-11
  - properties 16-12
  - runtime behaviors 16-18
- decision pivots 16-9 to 16-10
- dimension buttons 16-10
  - orientation 16-10
  - properties 16-10
  - runtime behaviors 16-18
- decision queries
  - defining 16-6
- Decision Query editor 16-6
- decision sources 16-9
  - events 16-9
  - properties 16-9
- decision support components 14-15, 16-1 to 16-20
  - adding 16-3 to 16-4
  - assigning data 16-4 to 16-6
  - design options 16-8
  - memory management 16-19
  - runtime 16-18 to 16-19
- declarations
  - classes 41-9, 49-5
    - protected 41-5
    - public 41-6
    - published 41-6
  - event handlers 43-5, 43-8, 50-12
  - message handlers 46-4, 46-5, 46-7
  - methods 8-15, 44-4
    - dynamic 41-9
    - public 44-3
    - static 41-7
    - virtual 41-8
  - new component types 41-3
  - properties 42-3, 42-3 to 42-6, 42-7, 42-12, 43-8, 49-4
  - stored 42-12
  - user-defined types 49-3
  - variables
    - example 3-10
- DECnet protocol (Digital) 32-1
- default
  - ancestor class 41-3
  - directive 42-11, 48-3
  - handlers
    - events 43-9
    - message 46-3
    - overriding 43-9
  - project options 5-3
  - property values 42-7
    - changing 48-2, 48-3
    - specifying 42-11 to 42-12
  - reserved word 42-7
  - values 15-10
- Default checkbox 5-3
- Default property
  - action items 28-7

- DEFAULT\_ORDER 23-7
- DefaultColWidth property 3-43
- DefaultDatabase property 21-4
- DefaultDrawing property 7-12, 15-25
- DefaultExpression
  - property 19-20, 23-7
- DefaultHandler method 46-3
- DefaultPage property 29-18
- DefaultRowHeight
  - property 3-43
- delegation 43-1
- Delete command (Menu designer) 6-37
- Delete method 18-19
  - string lists 3-51, 3-52
- DELETE statements 20-39, 20-42, 24-9
- Delete Table command 18-40
- Delete Templates command (Menu designer) 6-37, 6-39
- Delete Templates dialog
  - box 6-39
- DeleteAlias method 20-26
- DeleteFile function 4-50
- DeleteFontResource
  - function 13-13
- DeleteIndex method 23-9
- DeleteRecords method 18-40
- DeleteSQL property 20-40
- DeleteTable method 18-40
- Delphi
  - ActiveX framework (DAX) 33-2, 33-21 to 33-22
- delta packets 24-8, 24-9
  - editing 24-8, 24-9
  - screening updates 24-11
  - XML 25-36, 25-37 to 25-38
- Delta property 23-5, 23-19
- \$DENYPACKAGEUNIT
  - compiler directive 11-10
- DEPLOY 13-8, 13-14
- deploying
  - ActiveX controls 13-5
  - applications 13-1
  - Borland Database Engine 13-8
  - CLX applications 13-6
  - database applications 13-6
  - dbExpress 22-1
  - DLL files 13-5
  - fonts 13-13
  - general applications 13-1
  - MIDAS applications 13-9
  - package files 13-3
  - SQL Links 13-8
  - dereferencing object pointers 41-9
  - deriving classes 41-8
  - descendant classes 3-8, 41-3
    - redefining methods 41-8
  - design tools 2-2
  - designing
    - applications 2-2
  - DESIGNONLY compiler
    - directive 11-10
  - design-time interfaces 41-6
  - design-time packages 11-1, 11-5 to 11-6
  - destination datasets,
    - defined 20-48
  - Destroy method 3-11
  - destructors 3-11, 44-3, 51-6
    - owned objects and 49-5, 49-6
  - developer support 1-3
  - device contexts 8-1, 8-2, 40-7, 45-1
  - device-independent
    - graphics 45-1
  - DeviceType property 8-31 3-7
  - .DFM files 3-7, 10-2, 12-10, 42-11
    - generating 12-13
  - diacritical marks 12-10
  - dialog boxes 52-1 to 52-6
    - common 3-45
    - creating 52-1
    - internationalizing 12-9, 12-10
    - multipage 3-40
    - property editors as 47-9
    - setting initial state 52-1
    - Windows common 52-1
      - creating 52-2
      - executing 52-4
  - Dialogs page (Component palette) 3-30
  - digital audio tapes 8-32
  - DimensionMap property 16-5, 16-7
  - Dimensions property 16-12
  - Direction property
    - parameters 18-45, 18-51
  - directives 10-19
    - \$ELSEIF 10-19
    - \$ENDIF 10-19
    - \$H compiler 4-41, 4-49
    - \$IF 10-19
    - \$IFDEF 10-18
    - \$IFEND 10-19
  - \$IFDEF 10-18
  - \$LIBPREFIX compiler 5-9
  - \$LIBSUFFIX compiler 5-9
  - \$LIBVERSION compiler 5-9
  - \$MESSAGE compiler 10-20
  - \$P compiler 4-49
  - \$V compiler 4-49
  - \$X compiler 4-49
  - conditional
    - compilation 10-18
  - default 42-11, 48-3
  - dynamic 41-9
  - override 41-8, 46-3
  - protected 43-5
  - public 43-5
  - published 42-3, 43-5, 52-3
  - stored 42-12
  - string-related 4-49
  - virtual 41-8
- directories, Linux 10-16
- Directory directive 13-10
- DirtyRead 17-9
- DisableCommit method 39-12
- DisableConstraints
  - method 23-29
- DisableControls method 15-6
- DisabledImages property 6-46
- disconnected model 14-14
- dispatch actions 29-4
- dispatch interfaces 36-12, 36-14
  - calling methods 35-13
  - identifiers 36-14
  - type compatibility 36-15
  - type libraries 34-9
  - Type Library editor 34-16
- Dispatch method 46-3, 46-4
- dispatcher components 29-13
  - adapters 29-13
- dispatcher, Web 28-2, 28-4 to 28-5
- dispatchers
  - action items 29-17
- dispatching requests
  - WebSnap 29-12
- dispIDs 33-15, 36-14
  - binding to 36-14
- dispinterfaces 25-30, 36-12, 36-13, 36-14
  - dynamic binding 34-9
  - type libraries 34-9
- displatcher components 29-4
- DisplayFormat property 15-25, 19-11, 19-15
- DisplayLabel property 15-17, 19-11

- DisplayWidth property 15-16, 19-11
- distributed applications
  - database 5-11
  - MTS and COM+ 5-14
- distributed COM 33-7, 33-8
- distributed data processing 25-2
- DllGetClassObject 39-3
- DllRegisterServer 39-3
- DLLs 10-15
  - Apache 13-10
  - COM servers 33-6
    - threading models 36-7
  - creating 5-9
  - deployment 13-9
  - embedding in HTML 28-14
  - HTTP servers 27-5, 27-6
  - installing 13-5
  - internationalizing 12-12, 12-13
  - MTS 39-2
  - packages 11-1, 11-2
- DML 17-10, 18-42, 18-47, 20-8
- .DMT files 6-38, 6-39
- docking 7-4
- docking site 7-5
- Document Object Model
  - See* DOM
- Document Type Definition file
  - See* DTD file
- documentation
  - ordering 1-3
- DocumentElement
  - property 30-3
- DoExit method 51-12
- DOM 30-2, 30-2 to 30-3
  - implementations 30-2
    - using 30-3
- double-clicks
  - components 47-15
  - responding to 47-17
- Down property 3-35
  - speed buttons 6-44
- .DPK files 11-2, 11-6
- .DPL files 11-2, 11-12
- drag cursors 7-2
- drag object 7-3
- drag-and-dock 3-20, 3-22, 7-4 to 7-6
- drag-and-drop 3-20, 7-1 to 7-4
  - customizing 7-3
  - DLLs 7-4
  - events 49-2
  - getting state information 7-3
  - mouse pointer 7-4
- DragCursor property 3-20
- DragMode property 3-20, 7-1
  - grids 15-19
- draw grids 3-43
- Draw method 8-4, 45-3, 45-7
- drawing modes 8-28
- drawing tools 45-1, 45-7, 49-5
  - assigning as default 6-45
  - changing 8-13, 49-7
  - handling multiple in an application 8-12
  - testing for 8-12, 8-13
- DrawShape 8-15
- drill-down forms 15-14
- dprintf unit 20-52
- drive letters 10-15
- driver names 20-14
- DriverName property 20-14, 22-3
- DropConnections
  - method 20-13, 20-20
- drop-down lists
  - in data grids 15-20
- drop-down menus 6-34 to 6-35
- DropDownCount
  - property 3-38, 15-11
- DropDownMenu property 6-49
- DropDownRows property
  - data grids 15-20, 15-21
  - lookup combo boxes 15-12
- DTD file 30-2
- dual interfaces 36-13 to 36-14
  - Active Server Objects 37-3
  - calling methods 35-12
  - parameters 36-16
  - transactional objects 39-3, 39-16
  - type compatibility 36-15
- durability
  - resource dispensers 39-5
  - transactions 14-4, 39-9
- dynamic binding 25-29
- dynamic columns 15-15
  - properties 15-16
- dynamic directives 41-9
- dynamic fields 19-2 to 19-3
- dynamic memory 3-55
- dynamic methods 41-8

**E**

---

- EAbort 4-16
- early binding 25-29
  - Automation 33-17, 36-12
  - COM 33-16
- EBX register 10-9, 10-21
- Edit control 3-31
- edit controls 3-31 to 3-32, 7-6, 15-2, 15-8
  - multi-line 15-8
  - rich edit formats 15-9
  - selecting text 7-8, 7-9
- Edit method 18-17, 47-9, 47-10
- edit mode 18-17
  - canceling 18-17
- EditFormat property 15-25, 19-11, 19-15
- editing code 2-3, 2-4
- editing script 29-10
- EditKey method 18-27, 18-29
- EditMask property 19-14
  - fields 19-11
- EditRangeEnd method 18-33
- EditRangeStart method 18-33
- Ellipse method 8-4, 8-11, 45-3
- ellipses
  - drawing 8-11, 49-9
- ellipsis (...)
  - buttons in grids 15-21
- \$ELSEIF directive 10-19
- Embed HTML tag
  - (<EMBED>) 28-14
- EmptyDataSet method 18-40, 23-26
- EmptyStr variable 4-46
- EmptyTable method 18-40
- EnableCommit method 39-12
- EnableConstraints
  - method 23-29
- EnableControls method 15-6
- Enabled property
  - action items 28-6
  - data sources 15-4, 15-5
  - data-aware controls 15-7
  - menus 6-41, 7-10
  - speed buttons 6-45
- EnabledChanged
  - property 10-21
- encapsulation 3-5
- encryption
  - TSocketConnection 25-25
- end of file character 10-14
- end user adapter 29-4
- endpoints
  - socket connections 32-5
- EndRead method 9-8
- EndWrite method 9-8
- enumerated types 42-2, 49-3
  - constants vs. 8-13
  - declaring 8-12

- Type Library editor 34-10, 34-17, 34-23
- EOF marker 4-57
- Eof property 18-6, 18-7
- EPasswordInvalid 4-17
- EReadError 4-56
- ERemotableException 31-7
- error messages
  - internationalizing 12-10
- ErrorAddr variable 4-17
- errors
  - sockets 32-8
  - Web adapters 29-8
- Euro conversions 4-62
- European currency
  - conversion 4-64
- event handlers 3-7, 3-25 to 3-28, 10-21, 40-6, 43-2, 51-6
  - associating with events 3-27
  - declarations 43-5, 43-8, 50-12
  - default, overriding 43-9
  - defined 3-25
  - deleting 3-28
  - displaying the Code editor 47-17
  - drawing lines 8-25
  - empty 43-9
  - locating 3-26
  - menus 3-28, 7-11
    - as templates 6-40
  - methods 43-3, 43-5
    - overriding 43-6
  - parameters 43-3, 43-8, 43-9
    - notification events 43-7
  - passing parameters by reference 43-9
  - pointers 43-2, 43-3, 43-8
  - responding to button clicks 8-13
  - Sender parameter 3-27
  - shared 3-27 to 3-28, 8-15
  - types 43-3, 43-7 to 43-8
  - writing 3-9, 3-26
- event notification 6-5
- event notifications 6-5
- event objects 9-9
  - COM+ 39-19
- event sinks 36-12
  - defining 35-13 to 35-14
- EventFilter 6-5, 46-5
- events 3-25 to 3-28, 10-21, 40-6, 43-1 to 43-9
  - accessing 43-5
  - ActiveX controls 38-9 to 38-10
- ADO connections 21-7 to 21-8
- application-level 6-3
- associating with handlers 3-27
- Automation
  - controllers 35-10, 35-13 to 35-15
  - Automation objects 36-5
  - COM 36-10, 36-12
  - COM objects 36-10 to 36-12
    - component wrappers 35-2
  - COM+ 35-14 to 35-15, 39-18 to 39-20
    - firing 39-20
  - cross-platform 10-22
  - data grids 15-25 to 15-26
  - data sources 15-4
  - data-aware controls
    - enabling 15-7
  - default 3-26
  - defining new 43-6 to 43-9
  - field objects 19-15 to 19-16
  - graphical controls 45-7
  - implementing 43-2, 43-4
  - inherited 43-4
  - interfaces 36-11
  - login 17-5
  - message handling 46-3, 46-6
  - mouse 8-23 to 8-25
    - testing for 8-26
  - naming 43-8
  - objects and 3-10
  - providing help 47-4
  - responding to 43-6, 43-7, 43-9, 51-6
  - retrieving 43-3
  - shared 3-27
  - signalling 9-9
  - standard 43-4, 43-4 to 43-6
  - system 3-4
  - timeout 9-10
  - types 3-3
  - user 3-3
  - waiting for 9-9
  - XML brokers 25-38
- EWriteError 4-56
- Exception 4-16
- exception handling 4-4 to 4-17
  - creating a handler 4-10
  - declaring the object 4-16
  - default handlers 4-12
  - executing cleanup code 4-5
  - flow of control 4-6
  - overview 4-4 to 4-17
- protecting blocks of code 4-4
- protecting resource allocations 4-7
- resource protection blocks 4-8
- scope 4-12
  - See also* exceptions statements 4-11
  - TApplication 4-15
- exceptions 4-4 to 4-17, 10-15, 44-2, 46-3, 52-5
  - classes 4-13
  - COM interfaces 34-9
  - component 4-14
  - handling 4-5
  - instances 4-11
  - nested 4-6
  - raising 4-17
  - re-raising 4-13
  - responding to 4-5
  - RTL 4-9
  - See also* exception handling silent 4-15 threads 9-6 user-defined 4-16
- exclusive locks tables 20-6
- Exclusive property 20-6
- ExecProc method 18-53, 22-10
- ExecSQL method 18-47, 22-10
  - update objects 20-45
- executable files 10-15
  - COM servers 33-7
  - threading models 36-7
  - internationalizing 12-12, 12-13
- Execute method
  - ADO commands 21-17 to 21-18, 21-19
  - client datasets 23-27, 24-3
  - connection
    - components 17-10
  - dialogs 3-46, 52-4
  - providers 24-3
  - TBatchMove 20-51
  - threads 9-4
- ExecuteOptions property 21-11
- ExecuteTarget method 6-28
- Expandable property 15-23
- Expanded property
  - columns 15-22, 15-23
  - data grids 15-20
- Expression property 23-11
- ExprText property 19-10
- Extensible Markup Language
  - See* XML

- F**
- 
- factory 29-5
  - FastNet page (Component palette) 3-29
  - features
    - non-portable Windows 10-8
  - Fetch Params command 23-26
  - FetchAll method 10-29, 20-33
  - FetchBlobs method 23-26, 24-3
  - FetchDetails method 23-26, 24-3
  - fetch-on-demand 23-26
  - FetchOnDemand
    - property 23-26
  - FetchParams method 23-26, 24-3
  - field attributes 19-12 to 19-14
    - assigning 19-13
    - in data packets 24-5
    - removing 19-14
  - field datalink class 51-10
  - field definitions 18-38
    - copying 18-38
  - Field Link designer 18-35
  - field objects 19-1 to 19-28
    - accessing values 19-19 to 19-20
    - defining 19-5 to 19-10
    - deleting 19-10
    - display and edit
      - properties 19-11
    - dynamic 19-2 to 19-3
      - vs. persistent 19-2
    - events 19-15 to 19-16
    - persistent 19-3 to 19-16
      - vs. dynamic 19-2
    - properties 19-1, 19-10 to 19-15
      - runtime 19-12
      - sharing 19-12
  - field types
    - converting 19-16, 19-17 to 19-19
  - FieldByName method 18-31, 19-20
  - FieldCount property
    - persistent fields 15-17
  - FieldDefs property 18-38
  - FieldKind property 19-11
  - FieldName property 19-5, 19-11, 25-40
    - data grids 15-20
    - decision grids 16-12
    - persistent fields 15-17
  - fields 19-1 to 19-28
    - abstract data types 19-22 to 19-28
    - activating 19-16
    - adding to forms 8-26 to 8-27
    - assigning values 18-21
    - changing values 15-5
    - databases 51-5, 51-6
    - default formats 19-14
    - default values 19-20
    - displaying values 15-10, 19-17
    - entering data 18-18, 19-14
    - hidden 24-5
    - limiting valid data 19-21 to 19-22
    - listing 17-13
    - message records 46-2, 46-4, 46-6
    - mutually-exclusive
      - options 15-2
    - null values 18-21
    - persistent columns
      - and 15-17
    - properties 19-1
    - read-only 15-5
    - retrieving data 19-17
    - updating values 15-5
    - Web adapters 29-8
  - Fields editor 5-18, 19-3
    - applying field
      - attributes 19-13
    - creating persistent
      - fields 19-4 to 19-5, 19-5 to 19-10
    - defining attribute sets 19-13
    - deleting persistent
      - fields 19-10
    - list of fields 19-4
    - navigation buttons 19-4
    - removing attribute sets 19-14
    - reordering columns 15-19
    - title bar 19-4
  - Fields property 19-19
  - FieldValues property 19-19
  - file I/O
    - types 4-53
  - file lists
    - dragging items 7-2, 7-3
    - dropping items 7-3
  - file permissions 10-15
  - file streams
    - changing the size of 4-57
    - component streaming 4-54
    - creating 4-54
    - end of marker 4-57
    - exceptions 4-56
    - file I/O 4-54 to 4-57
      - getting a handle 4-53
      - opening 4-54
      - portable 4-53
      - TMemoryStream 4-54
  - FileAge function 4-52
  - file-based applications 14-9 to 14-10
    - client datasets 23-31 to 23-34
  - FileExists function 4-51
  - FileGetDate function 4-52
  - FileName property
    - client datasets 14-9, 23-32, 23-33
  - files 4-50 to 4-57
    - See also* file streams
    - copying 4-53
    - copying bytes from 4-57
    - date-time routines 4-52
    - deleting 4-50
    - finding 4-51
    - form 10-2
    - graphics 8-18 to 8-21, 45-4
    - handles 4-53, 4-55
    - incompatible types 4-53
    - manipulating 4-50 to 4-53
    - modes 4-55
    - position 4-57
    - reading from 4-55
    - renaming 4-52
    - resource 6-42
    - routines
      - date-time routines 4-52
      - runtime library 4-50
      - Windows API 4-53
    - seeking 4-56
    - sending over the Web 28-12
    - size 4-57
    - strings 4-56
    - types
      - I/O 4-53
      - text 4-53
      - typed 4-53
      - untyped 4-53
      - working with 4-50 to 4-57
      - writing to 4-55
  - files streams 4-54 to 4-57
  - FileSetDate function 4-52
  - fill patterns 8-8
  - FillRect method 8-4, 45-3
  - Filter property 18-13, 18-13 to 18-14
  - Filtered property 18-12

- FilterGroup property 21-12, 21-13
- FilterOnBookmarks method 21-11
- FilterOptions property 18-15
- filters 18-12 to 18-15
  - blank fields 18-14
  - case sensitivity 18-15
  - client datasets 23-3 to 23-5
    - using parameters 23-28
  - comparing strings 18-15
  - defining 18-13 to 18-15
  - enabling/disabling 18-12
  - operators 18-14
  - options for text fields 18-15
  - queries vs. 18-12
  - ranges vs. 18-30
  - setting at runtime 18-15
  - using bookmarks 21-10 to 21-11
- finally reserved word 45-6, 52-5
- FindClose procedure 4-51
- FindDatabase method 20-20
- FindFirst function 4-51
- FindFirst method 18-16
- FindKey method 18-27, 18-28
  - EditKey vs. 18-29
- FindLast method 18-16
- FindNearest method 18-27, 18-28
- FindNext function 4-51
- FindNext method 18-16
- FindPrior method 18-16
- FindResourceHInstance function 12-13
- FindSession method 20-29
- First Impression 13-5
- First method 18-6
- FixedColor property 3-43
- FixedCols property 3-43
- FixedOrder property 3-36, 6-48
- FixedRows property 3-43
- FixedSize property 3-36
- flags 51-3
- FlipChildren method 12-7
- FloodFill method 8-4, 45-3
- fly-by help 3-42
- fly-over help 15-30
- focus 3-18, 40-4
  - fields 19-16
  - moving 3-33
- FocusControl method 19-16
- FocusControl property 3-41
- Font property 3-19, 3-31, 3-41, 8-4, 45-3
  - column headers 15-20
  - data grids 15-20
  - data-aware memo controls 15-9
- FontChanged property 10-21
- fonts 13-13
  - height of 8-5
- Footer property 28-19
- FOREIGN KEY constraint 24-13
- foreign translations 12-1
- form files 3-16, 10-2, 12-13
- form linking 6-2
- Format property 16-12
- formatting data
  - international applications 12-10
- forms 2-2, 3-23
  - accessing from other forms 3-9
  - adding fields to 8-26 to 8-27
  - adding to projects 3-8, 6-1 to 6-2
  - adding unit references 6-2
  - as components 52-1
  - as new object types 3-5 to 3-7
  - creating at runtime 6-6
  - displaying 6-6
  - drill down 15-14
  - global variable for 6-6
  - instantiating 3-6
  - linking 6-2
  - main 6-1
  - master/detail tables 15-14
  - memory management 6-6
  - modal 6-6
  - modeless 6-6, 6-7
  - navigating among controls 3-19, 3-22
  - passing arguments to 6-8 to 6-9
  - querying properties
    - example 6-9
  - referencing 6-2
  - retrieving data from 6-9 to 6-12
  - sharing event handlers 8-15
  - synchronizing data 15-4
  - using local variables to create 6-7
- Formula One 13-5
- Found property 18-16
- FoxPro tables
  - local transactions 20-31
- FrameRect method 8-4
- frames 6-12, 6-13 to 6-16
  - and component templates 6-14, 6-15
  - graphics 6-15
  - resources 6-15
  - sharing and distributing 6-16
- FReadOnly 51-8
- Free method 3-11, 10-13
- free threading 36-7 to 36-8
- FreeBookmark method 18-9
- freeing resources 52-5
- free-threaded marshaler 36-8
- FromCommon 4-62
- functions 40-6
  - events and 43-3
  - graphics 45-1
  - naming 44-2
  - reading properties 42-6, 47-8, 47-10
  - Windows API 40-3, 45-1

## G

---

- \$G compiler directive 11-10, 11-12
- GDI applications 40-7, 45-1
- Generate event support
  - code 36-11
- geometric shapes
  - drawing 49-9
- GetAliasDriverName method 20-26
- GetAliasNames method 20-26
- GetAliasParams method 20-26
- GetAttributes method 47-10
- GetBookmark method 18-9
- GetConfigParams method 20-26
- GetData method
  - fields 19-16
- GetDatabaseNames method 20-26
- GetDriverNames method 20-26
- GetDriverParams method 20-26
- GetFieldName method 28-9
- GetFieldNames method 17-13, 20-26
- GetFloatValue method 47-8
- GetGroupState method 23-10
- GetHandle 5-24
- GetHelpFile 5-24
- GetHelpStrings 5-25
- GetIDsOfNames method 36-14
- GetIndexNames method 17-14, 18-26
- GetMethodValue method 47-8

- GetNextPacket method 10-29, 20-33, 23-25, 23-26, 24-3
  - GetOptionalParam method 23-15, 24-6
  - GetOrdValue method 47-8
  - GetPalette method 45-5
  - GetParams method 24-3
  - GetPassword method 20-22
  - GetProcedureNames method 17-13
  - GetProcedureParams method 17-14
  - GetProperties method 47-10
  - GetRecords method 24-3, 24-7
  - GetSessionNames method 20-29
  - GetStoredProcNames method 20-26
  - GetStrValue method 47-8
  - GetTableNames method 17-13, 20-26
  - GetValue method 47-8
  - GetVersionEx function 13-14
  - GetViewerName 5-23
  - GetXMLE method 26-10
  - Global Offset Table (GOT) 10-20
  - Glyph property 3-35, 6-44
  - GNU assembler 10-17
  - GNU make utility 10-15
  - GotoBookmark method 18-9
  - GotoCurrent method 18-40
  - GotoKey method 18-27, 18-28
  - GotoNearest method 18-27, 18-28
  - Graph Custom Control 13-5
  - Graphic property 8-17, 8-21, 45-4
  - graphical controls 40-4, 45-3, 49-1 to 49-9
    - bitmaps vs. 49-3
    - creating 40-4, 49-3
    - drawing 49-3 to 49-9
    - events 45-7
    - saving system resources 40-4
  - graphics 45-1 to 45-7
    - adding controls 8-17
    - adding to HTML 28-14
    - associating with strings 3-52
    - changing images 8-20
    - complex 45-6
    - containers 45-4
    - copying 8-21
    - deleting 8-21
    - displaying 3-44
    - drawing lines 8-5, 8-10, 8-27 to 8-28
      - changing pen width 8-6
      - event handlers 8-25
    - drawing tools 45-1, 45-7, 49-5
      - changing 49-7
    - drawing vs. painting 8-4
    - file formats 8-3
    - files 8-18 to 8-21
    - functions, calling 45-1
    - in frames 6-15
    - internationalizing 12-9
    - loading 8-19, 45-4
    - methods 45-3, 45-4, 45-6
      - copying images 45-7
      - palettes 45-5
    - owner-draw controls 7-11
    - pasting 8-22
    - programming overview 8-1 to 8-3
    - redrawing images 45-7
    - replacing 8-20
    - resizing 8-20, 15-9, 45-7
    - rubber banding
      - example 8-23 to 8-28
      - saving 8-19, 45-4
    - standalone 45-3
    - storing 45-4
    - types of objects 8-3
  - graphics boxes 15-2
  - graphics methods
    - palettes 45-5
  - graphics objects
    - threads 9-5
  - GridLineWidth property 3-43
  - grids 3-43, 15-2, 50-1, 50-2, 50-5, 50-11
    - See also* decision grids
    - adding rows 18-18
    - color 8-6
    - customizing 15-16 to 15-21
    - data-aware 15-14, 15-26
    - default state 15-15
      - restoring 15-21
    - displaying data 15-15, 15-16, 15-26
    - drawing 15-25
    - editing data 15-6, 15-25
    - getting values 15-16
    - inserting columns 15-17
    - removing columns 15-16, 15-18
    - reordering columns 15-19
    - runtime options 15-23 to 15-24
  - group boxes 3-39
  - Grouped property
    - tool buttons 6-47
  - GroupIndex property 3-35
    - menus 6-41
    - speed buttons 6-44, 6-45
  - grouping components 3-39 to 3-41
  - grouping levels 23-9
    - maintained aggregates 23-12
  - GroupLayout property 16-10
  - Groups property 16-10
  - GUI applications 3-23
  - GUIDs 4-22, 33-4, 34-8
    - generating 4-22
- ## H
- 
- \$H compiler directive 4-41, 4-49
  - Handle property 4-55, 10-22, 32-6, 40-3, 40-4, 40-5, 45-3
    - device context 8-1, 8-2
  - HandleException 4-15
  - HandleException method 46-3
  - handles
    - resource modules 12-13
    - socket connections 32-6
  - HandleShared property 20-16
  - HandlesTarget method 6-28
  - HasConstraints property 19-11
  - HasFormat method 7-10, 8-22
  - header controls 3-41
  - Header property 28-19
  - headers
    - HTTP requests 27-4
    - owner-draw 7-11
  - Height property 3-19, 3-22, 6-4
    - list boxes 15-10
    - TScreen 13-12
  - Help 47-4
    - context sensitive 3-42
    - hints 3-42
    - tool-tip 3-42
    - type information 34-8
  - Help Hints 15-30
  - Help Manager 5-22, 5-22 to 5-31
  - Help selector 5-30
  - Help selectors 5-27
  - Help system 5-22
    - interfaces 5-22
    - registering objects 5-27
  - Help systems 47-4
    - files 47-4
    - keywords 47-5



- tool buttons 6-49
- Help viewers 5-22
- HelpContext 5-28, 5-29
- HelpContext property 3-42
- HelpFile 5-29
- HelpFile property 3-42
- HelpIntfs.pas 5-22
- HelpKeyword 5-28, 5-29
- HelpSystem 5-28, 5-29
- HelpType 5-28, 5-29
- heterogeneous queries 20-9 to 20-10
  - Local SQL 20-9
- hidden fields 24-5
- Hiding unused items and categories in action bands 6-22
- hierarchy (classes) 41-3
- Hint property 3-42
- hints 3-42
- Hints property 15-30
- home directory 10-15
- horizontal track bars 3-33
- HorzScrollBar 3-33
- host names 32-4
  - IP addresses vs. 32-4
- Host property
  - TSocketConnection 25-25
- HostName property
  - TCorbaConnection 25-27
- hosts 25-25, 32-4
  - addresses 32-4
  - URLs 27-3
- hot keys 3-33
- HotImages property 6-46
- HotKey property 3-33
- HTML commands 28-13
  - database information 28-18
  - generating 28-14
- HTML documents 27-4
  - ASP and 37-1
  - databases and 28-17
  - dataset page
    - producers 28-18
  - datasets 28-19, 28-20
  - embedded ActiveX
    - controls 38-1
  - embedding tables 28-19
  - generated for
    - ActiveForms 38-6
  - HTTP response messages 27-5
- InternetExpress
  - applications 25-34
  - page producers 28-13 to 28-17
  - style sheets 25-40
  - table producers 28-18 to 28-20
  - templates 25-39, 25-41 to 25-42, 28-13 to 28-15
- HTML forms 25-39
- HTML Result view 29-1
- HTML tables 28-14, 28-19
  - captions 28-19
  - creating 28-18 to 28-20
  - setting properties 28-19
- HTML templates 25-41 to 25-42, 28-13 to 28-17
  - default 25-39, 25-41
  - WebSnap page
    - producers 29-9
- HTMLDoc property 25-39, 28-14
- HTMLFile property 28-14
- HTML-transparent tags
  - converting 28-13, 28-14
  - parameters 28-13
  - predefined 25-41 to 25-42, 28-14
  - syntax 28-13
- HTTP 27-3
  - connecting to application server 25-26
  - message headers 27-3
  - multi-tiered
    - applications 25-9 to 25-10
  - overview 27-4 to 27-6
  - request headers 27-4, 28-9, 37-4
  - request messages *See* request messages
  - response headers 28-12, 37-5
  - response messages *See* response messages
  - SOAP 31-1
  - status codes 28-11
- HTTP request messages 29-12
- HTTP responses
  - actions 29-15
- httpd.conf 13-10
- httpsrvr.dll 25-10, 25-13, 25-26
- HyperHelp viewer 5-22
- hypertext links
  - adding to HTML 28-14

**I**

---

- IApplicationObject
  - interface 37-4
- IAppServer interface 23-30, 23-31, 24-3 to 24-4, 25-5
- calling 25-29
  - extending 25-16
  - local providers 24-3
  - remote providers 24-3
  - state information 25-20
  - transactions 25-18
  - XML brokers 25-34
- icon 6-21
- IConnectionPoint
  - interface 36-12
- IConnectionPointContainer
  - interface 36-12
- icons 3-44, 45-4
  - graphics object 8-3
  - toolbars 6-46
  - tree views 3-38
- ICustomHelpViewer 5-22, 5-23, 5-24
  - implementing 5-23
- IDataIntercept interface 25-25
- IDefaultPageFileName 29-7
- identifiers
  - class fields 43-2
  - events 43-8
  - invalid 6-32
  - message-record types 46-6
  - methods 44-2
  - property settings 42-6
- ideographic characters 12-3, 12-4
  - abbreviations and 12-9
- IDispatch interface 33-8, 33-18, 36-12, 36-14
  - Automation 33-12
  - identifiers 36-14
- IDL (Interface Definition Language) 33-16, 33-18, 34-1
  - Type Library editor 34-8
- IDL compiler 33-18
- IDL files
  - exporting from type library 34-26
- IDOMImplementation 30-3
- IETF protocols and standards 27-2
- IExtendedHelpViewer 5-22, 5-26
- \$IFDEF directive 10-18
- \$IFEND directive 10-19
- \$IFNDEF directive 10-18
- IGetDefaultAction 29-7
- IGetProducerComponent 29-7
- IGetScriptObject 29-6
- IGetWebAppComponents 29-7
- IGetWebAppServices 29-7

- IHelpManager 5-23, 5-30
- IHelpSelector 5-23, 5-26, 5-27
- IHelpSystem 5-23, 5-30
- IIDs 33-4
- IInterface interface 4-20, 4-24, 4-25
  - implemented in
    - TInterfacedObject 4-21
- IInvokable 4-27, 31-3
- IIS 37-1
  - version 37-2
- IIteratorObjectSupport 29-6
- Image HTML tag (<IMG>) 28-14
- image requests 29-16
- ImageIndex property 6-46, 6-48
- ImageList 6-19
- ImageMap HTML tag (<MAP>) 28-14
- images 3-44, 15-2, 45-3
  - adding 8-17
  - adding control for 7-13
  - adding to menus 6-35
  - brushes 8-9
  - changing 8-20
  - controls for 8-2, 8-16
  - displaying 3-44
  - drawing 49-8
  - erasing 8-21
  - in frames 6-15
  - internationalizing 12-9
  - redrawing 45-7
  - reducing flicker 45-6
  - regenerating 8-2
  - saving 8-19
  - scrolling 8-17
  - tool buttons 6-46
- Images property
  - tool buttons 6-46
- IMalloc interface 4-18
- IMarshal interface 36-15, 36-16
- IME 12-8
- ImeMode property 12-8
- ImeName property 12-8
- implements keyword 4-22, 4-23
- \$IMPLICITBUILD compiler directive 11-10
- Import ActiveX Control
  - command 35-2, 35-4
- Import Type Library
  - command 35-2, 35-3
- ImportedConstraint
  - property 19-11, 19-21
- \$IMPORTEDDATA compiler directive 11-10
- Increment property 3-33
- incremental fetching 23-25, 25-20
- incremental search 15-10
- Indent property 3-38, 6-45, 6-47, 6-48
- index definitions 18-38
  - copying 18-38
- index files 20-6
- Index Files editor 20-6
- Index property
  - fields 19-11
- index reserved word 50-7
- index-based searches 18-11, 18-12, 18-27 to 18-29
- IndexDefs property 18-38
- indexes 18-25 to 18-36, 42-8
  - batch moves and 20-49, 20-50
  - client datasets 23-7 to 23-10
  - dBASE tables 20-6 to 20-7
  - deleting 23-9
  - grouping data 23-9 to 23-10
  - listing 17-14, 18-26
  - master/detail relationships 18-35
  - ranges 18-30
  - searching on partial keys 18-29
  - sorting records 18-25 to 18-27, 23-7
  - specifying 18-26 to 18-27
- IndexFieldCount
  - property 18-26
- IndexFieldNames
  - property 18-27, 22-7, 23-8
  - IndexName vs. 18-27
- IndexFields property 18-26
- IndexFiles property 20-6
- IndexName property 20-6, 22-7, 23-9
  - IndexFieldNames vs. 18-27
- IndexOf method 3-50, 3-51
- Indy Clients page (Component palette) 3-30
- Indy Misc page (Component palette) 3-30
- Indy Servers page (Component palette) 3-30
- INFINITE constant 9-10
- Informix drivers
  - deploying 13-9
- Inherit (Object Repository) 5-20
- inheritance 3-5, 3-8
- inherited
  - events 43-4
  - methods 43-6
  - properties 49-2, 50-2
    - publishing 42-2
- inheriting from classes 3-8 to 3-12, 41-7
- INI files
  - Win-CGI programs 27-7
- ini files 10-7
- InitWidget property 10-22
- inner objects 33-9
- INotifyWebActivate 29-6
- in-process servers 33-6
  - ActiveX 33-13
  - ASP 37-7
  - MTS 39-2
- input controls 3-32
- input focus 40-4
  - fields 19-16
- Input Mask editor 19-14
- input method editor 12-8
- input parameters 18-50
- input/output parameters 18-50
- INSERT 17-11
- Insert command (Menu designer) 6-37
- Insert From Resource command (Menu designer) 6-37, 6-42
- Insert from Resource dialog box 6-42
- Insert From Template command (Menu designer) 6-37, 6-38
- Insert method 18-18, 18-19
  - Append vs. 18-18
  - menus 6-41
  - strings 3-51
- INSERT statements 20-39, 20-43, 24-9
- Insert Template dialog box 6-39
- InsertObject method 3-52
- InsertRecord method 18-21
- InsertSQL property 20-40
- Install COM+ objects
  - command 39-22
- Install MTS objects
  - command 39-22
- installation programs 13-2
- Installing transactional objects 39-22
- InstallShield Express 2-5, 13-1
  - deploying
    - applications 13-2
    - BDE 13-8
    - packages 13-3
    - SQL Links 13-9
- instances 43-2
- instancing
  - COM objects 36-5 to 36-6

- CORBA data modules 25-16
  - remote data modules 25-14
- IntegralHeight property 3-37, 15-10
- integrated debugger 2-5
- integrity violations 20-51
- InterBase driver
  - deploying 13-9
- InterBase Express
  - deployment 13-7
- InterBase page (Component palette) 3-29, 14-2
- InterBase tables 20-9
- InterBaseExpress 10-24
- interceptors 33-5
- Interface Definition Language (IDL) 34-1
- interface keyword 4-17
- interface pointer 33-5
- interfaces 4-17 to 4-27, 41-4, 41-5, 52-1, 52-3
  - ActiveX 33-19
    - customizing 38-7 to 38-11
  - adding methods 36-10
  - adding properties 36-9 to 36-10
  - aggregation 4-22, 4-23
  - application servers 25-16 to 25-18, 25-29
  - as operator 4-21
  - Automation 36-12 to 36-14
  - CLSIDs 4-26
  - COM 4-26, 5-14, 33-1, 33-3 to 33-5, 34-8 to 34-9, 35-1, 36-3, 36-9 to 36-14
    - declarations 35-5
    - events 36-11
  - COM+ event objects 39-19
  - components 4-25
  - controlling Unknown 4-24, 4-26
  - CORBA 4-26
  - Ctrl+Shift+G 4-22
  - custom 36-14
  - delegation 4-22
  - deriving 4-20
  - design-time 41-6
  - dispatch 36-14
  - distributed applications 4-26
  - DOM 30-2
  - dynamic binding 4-21, 34-9, 36-12
  - Dynamic Invocation Interface 4-27
  - dynamic querying 4-20
  - early binding 25-29
  - example code 4-18, 4-23, 4-24
  - extending single inheritance 4-17, 4-18
  - Help system 5-22
  - IIDs 4-22, 4-26
  - IInterface, implementing 4-20
  - implementing 33-6, 36-3
  - inner objects 4-23
  - internationalizing 12-9, 12-10, 12-13
  - invokable 4-26, 31-2, 31-3 to 31-4
  - language feature 4-18
  - late binding 25-29
  - lifetime management 4-20, 4-24
  - marshaling 4-27
  - memory management 4-21, 4-24
  - nonvisual program elements 40-5
  - object destruction 4-24
  - optimizing code 4-25
  - outer objects 4-23
  - outgoing 36-11, 36-12
  - overview 4-17 to 4-27
  - polymorphism 4-18
  - procedures 4-20
  - properties 42-10
  - properties, declaring 52-3
  - reference counting 4-20, 4-21, 4-24 to 4-26
  - reusing code 4-22
  - runtime 41-6
  - sharing between classes 4-18
  - SOAP 4-26
  - TComponent 4-25
  - type libraries 33-12, 33-17, 35-5, 36-9
  - Type Library editor 34-8 to 34-9, 34-15, 34-20, 36-9
  - using 4-17 to 4-27
  - Web data modules 29-6
  - Web modules 29-7
  - Web page modules 29-7
  - Web Services 31-1
  - XML nodes 30-4
- interfaces and WebSnap 29-6, 29-7
- InternalCalc fields 19-6, 23-10 to 23-11
  - indexes and 23-8
- international applications
  - abbreviations and 12-9
  - converting keyboard input 12-8
  - localizing 12-13
- internationalizing applications 12-1
- Internet Engineering Task Force 27-2
- Internet Information Server (IIS) 37-1
  - version 37-2
- Internet page (Component palette) 3-29
- Internet servers 27-1 to 27-9
- Internet standards and protocols 27-2
- InternetExpress 5-13, 25-33 to 25-42
  - vs. ActiveForms 25-32
- InternetExpress page (Component palette) 3-29
- intranets
  - host names 32-4
- InTransaction property 17-7
- Invalidate method 49-9
- invocation registry 31-4, 31-6
- invokable interfaces 4-26, 31-2, 31-3 to 31-4
  - calling 31-9 to 31-10
  - defining 31-2
  - implementing 31-6 to 31-7
  - namespaces 31-4
  - registering 31-4
  - URI 31-9
- Invoke method 36-14
- IObjectContext interface 33-14, 37-3, 39-4
  - methods to end transactions 39-11
- IObjectContext interface 33-14, 39-2
- IOleClientSite interface 35-15
- IOleDocumentSite interface 35-15
- IP addresses 32-4, 32-6
  - host names 32-4
  - host names vs. 32-4
  - hosts 32-4
- IPageResult 29-7
- IPaint interface 4-19
- IPersist interface 4-18
- IProducerEditorViewSupport 29-7
- IProvideClassInfo 33-16

IProviderSupport interface 24-2  
 IPX/SPX protocols 32-1  
 IRequest interface 37-4  
 IResponse interface 37-5  
 IRotate interface 4-19  
 is reserved word 3-10  
 ISAPI applications 27-6  
   creating 28-1, 29-2  
   debugging 27-8  
   request messages 28-2  
 ISAPI DLLs 13-9  
 IsCallerInRole method 25-6,  
   39-14  
 IScriptingContext interface 37-2  
 ISecurityProperty  
   interface 39-15  
 IServer interface 37-6  
 ISessionObject interface 37-5  
 ISetWebContentOptions 29-7  
 isolation  
   transactions 14-4, 39-9  
 ISpecialWinHelpViewer 5-22  
 IsSecurityEnabled 39-14  
 IsValidChar method 19-16  
 ItemHeight property 3-37  
   combo boxes 15-11  
   list boxes 15-11  
 ItemIndex property 3-37  
   radio groups 3-39  
 Items property  
   list boxes 3-37  
   radio controls 15-13  
   radio groups 3-39  
 ITypeComp 33-17  
 ITypeInfo 33-17  
 ITypeInfo2 33-17  
 ITypeLib 33-17  
 ITypeLib2 33-17  
 IUnknown interface 4-26, 33-3,  
   33-4, 33-18  
   Automation  
     controllers 36-14  
 IVarStreamable 4-35 to 4-36  
 IVarStreamable interface 4-35  
 IWebVariablesContainer 29-6  
 IXMLNode 30-4 to 30-5, 30-6

## J

javascript libraries 25-33, 25-35  
   to 25-36  
   locating 25-35  
 just-in-time activation 25-7, 39-4  
   to 39-5  
   enabling 39-5

## K

K footnotes (Help systems) 47-5  
 KeepConnection property 17-3,  
   17-12, 20-18  
 KeepConnections  
   property 20-13, 20-18  
 key fields 18-32  
   multiple 18-31, 18-32  
 key violations 20-51  
 keyboard events 43-3, 43-9  
   internationalization 12-8  
 keyboard mappings 12-9, 12-10  
 keyboard shortcuts 3-33  
   adding to menus 6-33 to 6-34  
 key-down messages 43-5, 51-8  
 KeyDown method 51-9  
 KeyExclusive property 18-29,  
   18-33  
 KeyField property 15-12  
 KeyFieldCount property 18-29  
 KeyViolTableName  
   property 20-51  
 keyword-based help 5-26  
 KeywordHelp 5-29  
 keywords 47-5  
   protected 43-5  
 Kind property  
   bitmap buttons 3-35  
 Kyxlix 10-1

## L

labels 3-41, 12-9, 15-2, 40-4  
   columns 15-17  
 Last method 18-6  
 late binding 25-29  
   Automation 36-12, 36-14  
 Layout property 3-35  
 -LEpath compiler  
   directive 11-12  
 leap years 50-8  
 Left property 3-19, 3-21, 3-22,  
   6-4  
 LeftCol property 3-43  
 LeftPromotion method 4-32,  
   4-33  
 Length function 4-46  
 libmidas.dcu 13-6  
 \$LIBPREFIX directive 5-9  
 libraries  
   custom controls 40-4  
 LibraryName property 22-3  
 \$LIBSUFFIX directive 5-9  
 \$LIBVERSION directive 5-9  
 .LIC file 38-7

license agreement 13-15  
 license keys 38-6  
 license package file 38-7  
 licensing  
   ActiveX controls 38-5, 38-6  
   to 38-7  
   Internet Explorer 38-7  
 line ending characters 10-14  
 lines  
   drawing 8-5, 8-10, 8-10, 8-27  
   to 8-28  
   changing pen width 8-6  
   event handlers 8-25  
   erasing 8-28  
 Lines property 3-32, 42-8  
 LineSize property 3-33  
 LineTo method 8-4, 8-7, 8-10,  
   45-3  
 Link HTML tag (<A>) 28-14  
 Linux  
   directories 10-16  
   operating  
     environment 10-14  
 list boxes 3-37, 15-2, 15-11, 50-1  
   data-aware 15-10 to 15-12  
   dragging items 7-2, 7-3  
   dropping items 7-3  
   owner-draw 7-11  
     draw-item events 7-15  
     measure-item events 7-14  
   populating 15-10  
   storing properties  
     example 6-9  
 list controls 3-36 to 3-39  
 List property 20-29  
 list views  
   owner draw 7-11  
 listening connections 32-2, 32-3,  
   32-7, 32-9  
   closing 32-7  
   port numbers 32-5  
 ListField property 15-12  
 lists  
   string 3-47 to 3-52  
   using in threads 9-5  
 ListSource property 15-12  
 -LNpath compiler  
   directive 11-12  
 Loaded method 42-13  
 LoadFromFile method  
   ADO datasets 21-14  
   client datasets 14-9, 23-32  
   graphics 8-19, 45-4  
   strings 3-48

- LoadFromStream method
  - client datasets 23-32
- LoadPackage function 11-4
- LoadParamListItems
  - procedure 17-14
- LoadParamsFromIniFile
  - method 22-5
- LoadParamsOnConnect
  - property 22-4
- local databases 14-3
  - accessing 20-5
  - aliases 20-25
  - BDE support 20-5 to 20-7
  - renaming tables 20-7
- Local SQL 20-9, 20-10
  - heterogeneous queries 20-9
- local transactions 20-31 to 20-32
- locale settings 4-44
- locales 12-2
  - data formats and 12-10
  - resource modules 12-10
- LocalHost property
  - client sockets 32-6
- localization 12-13
  - localizing applications 12-2
  - resources 12-10, 12-12, 12-13
- localizing applications 12-13
- LocalPort property
  - client sockets 32-6
- Locate method 18-10
- Lock method 9-7
- locking objects
  - nesting calls 9-7
  - threads 9-7
- LockList method 9-7
- LockType property 21-12, 21-13
- LogChanges property 23-5, 23-33
- logging in
  - SOAP connections 25-27
  - Web connections 25-26
- logical values 15-2, 15-12
- Login dialog box 17-4
- login events 17-5
- login information
  - specifying 17-4
- login scripts 17-4 to 17-5
- LoginPrompt property 17-4
- long strings 4-41
- lookup combo boxes 15-2, 15-11 to 15-12
  - in data grids 15-20
  - lookup fields 15-11
  - populating 15-20
  - secondary data
    - sources 15-12
- lookup fields 15-11, 19-6
  - caching values 19-9
  - defining 19-8 to 19-9
  - in data grids 15-20
  - performance 19-9
  - providing values
    - programmatically 19-9
    - specifying 15-20
- lookup list boxes 15-2, 15-11 to 15-12
  - lookup fields 15-11
  - secondary data
    - sources 15-12
- Lookup method 18-11
- lookup values 15-17
- LookupCache property 19-9
- LookupDataSet property 19-9, 19-11
- LookupKeyFields
  - property 19-9, 19-11
- LookupResultField
  - property 19-11
- lParam parameter 46-2
- .LPK file 38-7
- LPK\_TOOL.EXE 38-7
- LUpackage compiler
  - directive 11-12

## M

---

- main form 6-1
- main VCL thread 9-4
  - OnTerminate event 9-6
- MainMenu component 6-30
- maintained aggregates 14-15, 23-11 to 23-13
  - aggregate fields 19-10
  - specifying 23-11 to 23-12
  - subtotals 23-12
  - summary operators 23-11
  - values 23-13
- MainWndProc method 46-3
- make utility 10-15
- Man pages 5-22
- mappings
  - XML 26-2 to 26-3
  - defining 26-4
- Mappings property 20-50
- Margin property 3-35
- marshaling 33-7
  - COM interfaces 33-8 to 33-9, 36-4, 36-15 to 36-16
  - custom 36-16
- IDispatch interface 33-12, 36-15
  - transactional objects 39-3
  - Web Services 31-5
- mask edit controls 3-31
- masks 19-14
- master/detail forms 15-14
  - example 18-35 to 18-36
- master/detail
  - relationships 15-14, 18-34 to 18-36, 18-46 to 18-47
  - cascaded deletes 24-6
  - cascaded updates 24-6
  - client datasets 23-17
  - indexes 18-35
  - multi-tiered
    - applications 25-19
  - nested tables 18-36, 25-19
  - referential integrity 14-5
  - unidirectional datasets 22-12
- MasterFields property 18-34, 22-12
- MasterSource property 18-34, 22-12
- Max property
  - progress bars 3-42
  - track bars 3-33
- MaxDimensions property 16-19
- MaxLength property 3-31
  - data-aware memo
    - controls 15-8
    - data-aware rich edit
      - controls 15-9
- MaxRecords property 25-37
- MaxRows property 28-19
- MaxStmtsPerConn
  - property 22-3
- MaxSummaries property 16-19
- MaxTitleRows property 15-23
- MaxValue property 19-11
- MBCS 4-43
- MDAC 13-6
- MDI applications 5-1 to 5-2
  - creating 5-2
  - menus
    - merging 6-41 to 6-42
    - specifying active 6-41
- measurement units 4-60
- measurements
  - converting 4-58
- media devices 8-30
- media players 3-23, 8-30 to 8-33
  - example 8-32
- member functions 3-3
- Memo control 3-31

- memo controls 7-6, 42-8
    - properties 3-31
  - memo fields 15-2, 15-8 to 15-9
    - rich edit 15-9
  - memory management
    - COM objects 4-20
    - components 3-11
    - decision components 16-8, 16-19
    - dynamic vs. virtual
      - methods 41-8
      - forms 6-6
      - interfaces 4-25
  - Menu 6-17
  - menu components 6-30
  - Menu designer 3-28, 6-29 to 6-31
    - context menu 6-37
  - menu items 6-32 to 6-34
    - adding 6-32, 6-40
    - defined 6-29
    - deleting 6-33, 6-37
    - editing 6-36
    - grouping 6-33
    - moving 6-35
    - naming 6-32, 6-40
    - nesting 6-34
    - placeholders 6-37
    - separator bars 6-33
    - setting properties 6-36 to 6-37
    - underlining letters 6-33
  - Menu property 6-41
  - menus 6-29 to 6-40
    - accessing commands 6-33
    - adding 6-31 to 6-36
      - drop-down 6-34 to 6-35
      - from other
        - applications 6-42
    - adding images 6-35
    - disabling items 7-10
    - displaying 6-36, 6-37
    - handling events 3-28, 6-40
    - internationalizing 12-9, 12-10
    - moving among 6-37
    - moving items 6-35
    - naming 6-32
    - owner-draw 7-11
    - pop-up 7-10, 7-11
    - reusing 6-37
    - saving as templates 6-38, 6-39 to 6-40
    - templates 6-31, 6-37, 6-38 to 6-40
  - deleting 6-39
    - loading 6-38
  - menus, action lists 6-17
  - merge modules 13-3
  - MergeChangeLog method 23-6, 23-33
  - \$MESSAGE directive 10-20
  - message headers (HTTP) 27-3, 27-4
  - message loop
    - threads 9-4
  - message-based servers
    - See Web server applications
  - messages 46-1 to 46-7, 50-4
    - cracking 46-2
    - defined 46-2
    - handlers 46-1, 46-2, 50-4
      - creating 46-5 to 46-7
      - declarations 46-4, 46-5, 46-7
      - default 46-3
      - methods, redefining 46-7
      - overriding 46-3
    - handling 46-3 to 46-5
    - identifiers 46-6
    - key 51-8
    - mouse 51-8
    - mouse- and key-down 51-8
    - record
      - types, declaring 46-6
    - records 46-2, 46-4
    - trapping 46-4
    - user-defined 46-5, 46-7
    - Windows 6-5
  - messaging 10-21
  - metadata 17-12 to 17-14
    - dbExpress 22-12 to 22-17
    - modifying 22-10 to 22-11
    - obtaining from
      - providers 23-25
    - metafiles 3-44, 8-1, 8-18, 45-4
      - when to use 8-3
  - method pointers 43-2, 43-3, 43-8
  - Method property 28-9
  - methods 3-3, 8-15, 40-6, 44-1, 50-10
    - adding to ActiveX
      - controls 38-8 to 38-9
    - adding to interfaces 36-10
    - calling 43-6, 44-3, 49-4
    - declaring 8-15, 44-4
      - dynamic 41-9
      - public 44-3
      - static 41-7
      - virtual 41-8
  - deleting 3-28
    - dispatching 41-7
    - drawing 49-8, 49-9
  - event handlers 43-3, 43-5
    - overriding 43-6
  - graphics 45-3, 45-4, 45-6, 45-7
    - palettes 45-5
  - inherited 43-6
  - initialization 42-13
  - message-handling 46-1, 46-3, 46-4
  - naming 44-2
  - objects and 3-5, 3-7
  - overriding 41-8, 46-3, 46-4, 50-11
  - properties and 42-5 to 42-6, 44-1, 44-2, 49-4
  - protected 44-3
  - public 44-3
  - redefining 41-8, 46-7
  - virtual 41-8, 44-4
- MethodType property 28-6, 28-10
- Microsoft Server DLLs 27-6
  - creating 28-1, 29-2
  - request messages 28-2
- Microsoft SQL Server
  - deploying driver 13-9
- Microsoft Transaction Server 5-14, 33-14, 39-1
- midas.dll 23-1, 25-3
- midaslib.dcu 25-3
- MIDI files 8-32
- MIDL 33-18
  - See also IDL
- mime 8-21
- MIME messages 27-5
- Min property
  - progress bars 3-42
  - track bars 3-33
- MinSize property 3-34
- MinValue property 19-11
- MM film 8-32
- mobile computing 14-14
- modal forms 6-6
- Mode property 20-49
  - pens 8-5
- modeless forms 6-6, 6-7
- Modified method 51-11
- Modified property 3-32
- Modifiers property 3-33
- ModifyAlias method 20-25
- ModifySQL property 20-40
- modules 40-11

- Type Library editor 34-10 to 34-11, 34-18, 34-24
- Month property 50-5
- MonthCalendar
  - component 3-39
- months, returning current 50-8
- mouse buttons 8-24
  - clicking 8-24, 8-25
  - mouse-move events and 8-25
- mouse events 8-23 to 8-25, 49-2
  - defined 8-23
  - dragging and dropping 7-1 to 7-4
  - parameters 8-24
  - state information 8-24
  - testing for 8-26
- mouse messages 46-2, 51-8
- mouse pointer
  - drag-and-drop 7-4
- mouse-down messages 51-8
- MouseDown method 51-8
- MouseToCell method 3-43
- .MOV files 8-32
- Move method
  - string lists 3-51, 3-52
- MoveBy method 18-7
- MoveCount property 20-51
- MoveFile function 4-52
- MovePt 8-28
- MoveTo method 8-4, 8-7, 45-3
- .MPG files 8-32
- Msg parameter 46-3
- MSI technology 13-3
- MTS 5-14, 25-6, 33-10, 33-14, 39-1
  - See also* transactional objects
  - in-process servers 39-2
  - object references 39-20 to 39-21
  - requirements 39-3
  - runtime environment 39-2
  - transactional objects 33-14 to 33-15
  - transactions 25-18
  - vs. COM+ 39-1
- MTS executive 39-2
- MTS Explorer 39-23
- MTS packages 39-6, 39-22
- multibyte character codes 12-3
- multibyte character set 12-3
- multibyte characters
  - (MBCS) 10-17, 10-22
- multidimensional
  - crosstabs 16-3

- multi-line text controls 15-8, 15-9
- multimedia 8-32
- multipage dialog boxes 3-40
- multiple document interface 5-1 to 5-2
- multiprocessing
  - threads 9-1
- multi-read exclusive-write synchronizer 9-8
  - warning about use 9-8
- MultiSelect property 3-37
- multitasking 10-15
- multi-threaded applications 9-1
  - sessions 20-13, 20-28 to 20-29
- Multitier page (New Items dialog) 25-2
- multi-tiered applications 14-3, 14-12, 25-1 to 25-42
  - advantages 25-2
  - architecture 25-4, 25-5
  - building 25-11 to 25-30
  - callbacks 25-17
  - components 25-2 to 25-3
  - deploying 13-9
  - master/detail relationships 25-19
  - overview 25-3 to 25-4
  - parameters 23-26
  - server licenses 25-3
  - Web applications 25-31 to 25-42
    - building 25-33, 25-34 to 25-42
- multi-tiered architecture 25-4, 25-5
  - Web-based 25-31
- mutually exclusive options 6-45
- MyBase 23-31

## N

---

- Name property
  - fields 19-11
  - menu items 3-28
  - parameters 18-51
- named connections 22-4 to 22-5
  - adding 22-5
  - deleting 22-5
  - loading at runtime 22-4
  - renaming 22-5
- namespaces
  - invokable interfaces 31-4
- naming conventions
  - events 43-8
  - fields 43-2
- message-record types 46-6
- methods 44-2
  - properties 42-6
- navigator 15-2, 15-28 to 15-30, 18-5, 18-6
  - buttons 15-28
  - deleting data 18-20
  - editing 18-17
  - enabling/disabling buttons 15-28, 15-29
  - help hints 15-30
  - sharing among datasets 15-30
- NDX indexes 20-6
- nested details 18-36, 19-26 to 19-27, 25-19
  - fetch on demand 24-5
- nested tables 18-36, 19-26 to 19-27, 25-19
- NetCLX 5-11, 10-6
- NetFileDir property 20-24
- Netscape Server DLLs 27-6
  - creating 28-1, 29-2
  - request messages 28-2
- network control files 20-24
- networks
  - connecting to
    - databases 20-15
- neutral threading 36-9
- New command 40-11
- New Field dialog box 19-5
  - defining fields 19-6, 19-7, 19-8, 19-10
  - Field properties 19-5
  - Field type 19-6
  - Lookup definition 19-6
    - Dataset 19-9
    - Key Fields 19-9
    - Lookup Keys 19-9
    - Result Field 19-9
  - Type 19-6
- New Items dialog 5-19, 5-20, 5-21
- New Thread Object dialog 9-2
- New WebSnap
  - Application 29-2
- newsgroups 1-3
- NewValue property 20-37, 24-11
- Next method 18-6
- NextRecordSet method 18-53, 22-9
- non-blocking connections 32-9 to 32-10
  - blocking vs. 32-9

- no-nonsense license
  - agreement 13-15
- non-production index files 20-6
- nonvisual components 40-5, 40-11, 52-2
- nonvisual objects 3-11
- NOT NULL constraint 24-12
- NOT NULL UNIQUE constraint 24-12
- notebook dividers 3-40
- notification events 43-7
- NotifyID 5-23
- NSAPI applications 27-6
  - creating 28-1, 29-2
  - debugging 27-8
  - request messages 28-2
- null values
  - ranges 18-31
- null-terminated
  - wide strings 4-42
- numbers 42-2
  - internationalizing 12-10
  - property values 42-12
- numeric fields
  - formatting 19-14
- NumericScale property 18-45, 18-51
- NumGlyphs property 3-35

## O

---

- Object Broker 25-27
- object constructors 10-13
- object contexts 39-4
  - ASP 37-3
  - transactions 39-9
- object fields 19-22 to 19-28
  - types 19-22
- Object HTML tag (<OBJECT>) 28-14
- Object Inspector 3-7, 3-24, 42-2, 47-6
  - editing array properties 42-2
  - help with 47-4
  - selecting menus 6-38
- Object Pascal 10-22
  - overview 3-4
- object pooling 39-8
  - disabling 39-8
  - remote data modules 25-8
- Object Repository 5-19 to 5-22, 6-12
  - adding items 5-19
  - specifying shared directory 5-20
  - using items from 5-20 to 5-21

- Object Repository
  - database components 20-16
  - sessions 20-17
- object variables 3-10
- ObjectBroker property 25-25, 25-26, 25-28
- ObjectContext property
  - example 39-13
- ObjectName property
  - TCorbaConnection 25-27
- object-oriented
  - programming 3-4 to 3-12, 41-1 to 41-9
    - declarations 41-3, 41-9
    - classes 41-5, 41-6
    - methods 41-7, 41-8, 41-9
  - defined 3-4
  - inheritance 3-8
- object-oriented programming (OOP) 3-2
- objects 3-1, 3-5 to 3-12, 4-1
  - See also* COM objects
  - accessing 3-8 to 3-9
  - creating 3-11
  - customizing 3-8
  - defined 3-5
  - destroying 3-11
  - dragging and dropping 7-1
  - events and 3-7
  - helper 3-46
  - inheritance 3-8 to 3-12
  - instantiating 3-6, 43-2
  - multiple instances 3-6
  - nonvisual 3-11
  - owned 49-5 to 49-8
    - initializing 49-6
  - properties 3-5
  - scripting 29-11
  - temporary 45-6
  - TObject 3-14
  - type declarations 3-10
- Objects property 3-43
  - string lists 3-52, 7-15
- ObjectView property 15-22, 18-36, 19-23
- .OCX files 13-5
- ODBC drivers
  - using with ADO 21-1, 21-2
  - using with the BDE 20-1, 20-15, 20-16
- ODL (Object Description Language) 33-16, 34-1
- OEM character sets 12-3
- OEMConvert property 3-32
- offscreen bitmaps 45-6 to 45-7

- OldValue property 20-37, 24-11
- OLE
  - containers 3-23
  - merging menus 6-41
- OLE Automation *See* Automation
- OLE DB 21-1, 21-2
- OleObject property 38-13
- OLEView 33-18
- OnAccept event
  - server sockets 32-9
- OnAction event 28-7
- OnAfterPivot event 16-9
- OnBeforePivot event 16-9
- OnBeginTransComplete event 17-6, 21-8
- OnCalcFields event 18-22, 19-7, 23-10
- OnCellClick event 15-26
- OnChange event 19-15, 45-7, 49-7, 50-11, 51-11
- OnClick event 3-34, 43-1, 43-2, 43-4
  - buttons 3-6
  - menus 3-28
- OnClientConnect event 32-7
- OnClientDisconnect event 32-7
- OnColEnter event 15-26
- OnColExit event 15-26
- OnColumnMoved event 15-19, 15-26
- OnCommitTransComplete event 17-8, 21-8
- OnConnect event
  - client sockets 32-8
- OnConnectComplete event 21-7
- OnConnecting event
  - server sockets 32-9
- OnConstrainedResize event 6-4
- OnCreate event 40-13
- OnDataChange event 15-4, 51-6, 51-10
- OnDataRequest event 23-30, 24-3, 24-12
- OnDbClick event 15-26, 43-4
- OnDecisionDrawCell event 16-12
- OnDecisionExamineCell event 16-13
- OnDeleteError event 18-20
- OnDisconnect event 21-8
  - client sockets 32-7
- OnDragDrop event 7-2, 15-26, 43-4



- OnDragOver event 7-2, 15-26, 43-4
- OnDrawCell event 3-43
- OnDrawColumnCell event 15-25, 15-26
- OnDrawDataCell event 15-26
- OnDrawItem event 7-15
- OnEditButtonClick event 15-21, 15-26
- OnEditError event 18-17
- OnEndDrag event 7-3, 15-26, 43-4
- OnEndPage method 37-2
- OnEnter event 15-26, 43-5
- OnError event
  - sockets 32-8
- one-to-many relationships 18-34, 22-12
- OnExecuteComplete event 21-8
- OnExit event 15-26, 51-12
- OnFilterRecord event 18-13, 18-14 to 18-15
- OnGetData event 24-7
- OnGetDataSetProperties event 24-6
- OnGetTableName event 20-11, 23-21, 24-12
- OnGetText event 19-15
- OnGetThread event 32-9
- OnHandleActive event
  - client sockets 32-8
- OnHTMLTag event 25-42, 28-15, 28-16
- OnIdle event handler 9-5
- OnInfoMessage event 21-8
- OnKeyDown event 15-26, 43-5, 51-9
- OnKeyPress event 15-26, 43-5
- OnKeyUp event 15-26, 43-5
- OnLayoutChange event 16-9
- online help 47-4
- OnLogin event 17-5
- OnMeasureItem event 7-14
- OnMouseDown event 8-23, 8-24, 43-4, 51-8
  - parameters passed to 8-23
- OnMouseMove event 8-23, 8-25, 43-4
  - parameters passed to 8-23
- OnMouseUp event 8-14, 8-23, 8-25, 43-4
  - parameters passed to 8-23
- OnNewDimensions event 16-9
- OnNewRecord event 18-18
- OnPaint event 3-45, 8-2
  - providers 24-5 to 24-6
  - TSQClientDataSet 23-16
- Oracle drivers
  - deploying 13-9
- Oracle tables 20-12
- Oracle8
  - limits on creating tables 18-39
- ORDER BY clause 18-25
- Orientation property
  - data grids 15-27
  - track bars 3-33
- Origin property 8-27, 19-11
- outer objects 33-9
- outlines, drawing 8-5
- out-of-process servers 33-7
  - ASP 37-7
- output parameters 18-50, 23-26
- Overload property 20-12
- overloaded stored procedures 20-12
- override directive 41-8, 46-3
- overriding
  - methods 41-8, 46-3, 46-4, 50-11
- owned objects 49-5 to 49-8
  - initializing 49-6
- Owner property 3-11, 40-13
- owner-draw controls 3-52, 7-11
  - declaring 7-12
  - drawing 7-13, 7-15
  - list boxes 3-37, 3-38
  - sizing 7-14
- OwnerDraw property 7-12

## P

---

- \$P compiler directive 4-49
- package
  - dynamically loading 11-4
- Package Collection Editor 11-13
- package collection files 11-13
- package files 13-3
- packages 11-1 to 11-14, 47-19
  - collections 11-13
  - compiler directives 11-10
  - compiling 11-10 to 11-12
    - options 11-10
  - components 47-19
- Contains list 11-6, 11-7, 11-9, 47-19
- creating 5-9, 11-6 to 11-11
- custom 11-4
- default settings 11-7
- deploying applications 11-2, 11-13

- design-only option 11-7
- design-time 11-1, 11-5 to 11-6
- DLLs 11-1, 11-2
- duplicate references 11-9
- editing 11-7
- file-name extensions 11-1
- installing 11-5 to 11-6
- internationalizing 12-12, 12-13
- options 11-7
- referencing 11-3
- Requires list 11-6, 11-7, 11-8, 47-19
- runtime 11-1, 11-2 to 11-4, 11-7
- source files 11-2, 11-12
- using 5-9
- using in applications 11-3 to 11-4
- PacketRecords property 10-29, 20-33, 23-25
- page controls 3-40
  - adding pages 3-41
- page dispatcher 29-4
- page dispatchers
  - dispatchers
  - page 29-18
- page modules 29-3, 29-7
  - Web 29-6
- page name 29-6
- page producers 28-13 to 28-17, 29-6, 29-9
  - chaining 28-16
  - components 29-6
  - Content method 28-14
  - ContentFromStream 28-14
  - ContentFromString 28-14
  - converting templates 28-14
  - data-aware 25-38 to 25-42, 28-18
  - event handling 28-15, 28-16
  - templates 29-6, 29-9
  - types 29-3
- PageSize property 3-33
- paint boxes 3-45
- Paint method 45-6, 49-8, 49-9
- paintboxes 3-23
- palette bitmap files 47-3
- PaletteChanged method 45-5
- PaletteChanged property 10-21
- palettes 45-5
  - default behavior 45-5
  - specifying 45-5
- PanelHeight property 15-27
- panels
  - adding speed buttons 6-44
  - attaching to form tops 6-43
  - beveled 3-45
  - speed buttons 3-35
- Panels property 3-42
- PanelWidth property 15-27
- panes 3-34
  - resizing 3-34
- PAnsiChar 4-42
- PAnsiString 4-47
- Paradox tables 20-3, 20-5
  - accessing data 20-9
  - adding records 18-19
  - batch moves 20-51, 20-52
  - DatabaseName 20-3
  - directories 20-24
  - local transactions 20-31
  - network control files 20-24
  - password protection 20-21 to 20-23
  - renaming 20-7
  - retrieving indexes 18-26
- parallel processes
  - threads 9-1
- ParamBindMode property 20-12
- ParamByName method
  - queries 18-45
  - stored procedures 18-52
- ParamCheck property 18-44, 22-11
- parameter collection
  - editor 18-44, 18-50
- parameterized queries 18-42, 18-43 to 18-46
  - creating
    - at design time 18-44
    - at runtime 18-45
- parameters
  - binding modes 20-12
  - classes as 41-9
  - client datasets 23-26 to 23-28
    - filtering records 23-28
  - dual interfaces 36-16
  - event handlers 43-3, 43-7, 43-8, 43-9
  - from XML brokers 25-37
  - HTML tags 28-13
  - input 18-50
  - input/output 18-50
  - messages 46-2, 46-3, 46-4, 46-6
  - mouse events 8-23, 8-24
  - output 18-50, 23-26
- property settings 42-6
  - array properties 42-8
  - result 18-50
  - TXMLTransformClient 26-9
- Parameters property 21-19
- TADOCCommand 21-19
- TADOQuery 18-44
- TADOStoredProc 18-50
- ParamName property 25-40
- Params property
  - client datasets 23-26, 23-27
  - queries 18-44, 18-45
  - stored procedures 18-50
  - TDatabase 20-15
  - TSQLConnection 22-4
  - XML brokers 25-37
- ParamType property 18-45, 18-51
- ParamValues property 18-45
- parent controls 3-19
- parent properties 3-19
- Parent property 40-13
- ParentColumn property 15-23
- ParentShowHint property 3-42
- partial keys
  - searching 18-29
  - setting ranges 18-32
- passthrough SQL 20-30, 20-30 to 20-31
- passwords
  - dBASE tables 20-21 to 20-23
  - implicit connections
    - and 20-13
    - Paradox tables 20-21 to 20-23
- PasteFromClipboard
  - method 7-9
  - data-aware memo
    - controls 15-9
    - graphics 15-9
- PathInfo property 28-6
- pathnames 10-15
- paths (URLs) 27-3
- patterns 8-9
- .PCE files 11-13
- PChar 4-42
  - string conversions 4-47
- PDOXUSRS.NET 20-24
- Pen property 8-4, 8-5, 45-3
- PenPos property 8-4, 8-7
- pen 8-5, 49-5
  - brushes 8-5
  - changing 49-7
  - colors 8-6
  - default settings 8-6
  - drawing modes 8-28

- getting position of 8-7
- position, setting 8-7, 8-24
- style 8-6
- width 8-6
- PENWIN.DLL 11-11
- persistent columns 15-15, 15-16 to 15-17
  - creating 15-17 to 15-21
  - deleting 15-16, 15-18
  - inserting 15-18
  - reordering 15-19
- persistent fields 15-15, 19-3 to 19-16
  - ADT fields 19-24
  - array fields 19-25
  - creating 19-4 to 19-5, 19-5 to 19-10
  - creating tables 18-38
  - data packets and 24-4
  - data types 19-6
  - dataset fields 18-36
  - defining 19-5 to 19-10
  - deleting 19-10
  - listing 19-4, 19-5
  - naming 19-5
  - ordering 19-5
  - properties 19-10 to 19-15
  - special types 19-5, 19-6
  - switching to dynamic 19-3
- persistent subscriptions 35-15
- per-user subscriptions 35-15
- PickList property 15-20
- picture objects 8-3, 45-4
- Picture property 3-44, 8-17
  - in frames 6-15
- pictures 8-17, 45-3 to 45-5
  - changing 8-20
  - loading 8-19
  - replacing 8-20
  - saving 8-19
- Pie method 8-4
- Pixel property 8-4, 45-3
- pixels
  - reading and setting 8-9
- Pixels property 8-5, 8-9
- pmCopy constant 8-28
- pmNotXor constant 8-28
- pointers
  - class 41-9
  - default property values 42-12
  - method 43-2, 43-3, 43-8
- Polygon method 8-5, 8-11
- polygons 8-11
  - drawing 8-11
- PolyLine method 8-5, 8-10
- polylines 8-10
  - drawing 8-10
- polymorphism 3-2, 3-5
- pop-up menus 7-10 to 7-11
  - displaying 6-36
  - drop-down menus and 6-34
- PopupMenu component 6-30
- PopupMenu property 7-10
- Port property
  - server sockets 32-7
  - TSocketConnection 25-25
- portable code 10-17
- porting applications 10-1 to 10-29
- ports 32-5
  - client sockets 32-6
  - multiple connections 32-5
  - server sockets 32-7
  - services and 32-2
- Position property 3-33, 3-42
- position-independent code (PIC) 10-9, 10-20
- Post method 18-20
  - Edit and 18-17
- Precision property
  - fields 19-11
  - parameters 18-45, 18-51
- preexisting controls 40-4
- Prepared property
  - queries 18-47
  - stored procedures 18-53
  - unidirectional datasets 22-8
- primary indexes
  - batch moves and 20-49, 20-50
- PRIMARY KEY constraint 24-13
- printing 3-54
- Prior method 18-6
- priorities
  - using threads 9-1, 9-2
- Priority property 9-3
- private 3-9
- private properties 42-5
- private section 4-2
- PrivateDir property 20-24
- problem tables 20-51
- ProblemCount property 20-51
- ProblemTableName
  - property 20-51, 20-52
- ProcedureName property 18-49
- procedures 40-6, 43-3
  - naming 44-2
  - property settings 47-11
- processes 10-15
- programming templates 5-3
- progress bars 3-42
- project files
  - changing 2-3
  - distributing 2-5
- Project Manager 6-2
- project options 5-3
  - default 5-3
- Project Options dialog box 5-3
- project templates 5-21
- projects
  - adding forms 6-1 to 6-2
- properties 3-2, 42-1 to 42-13
  - accessing 42-5 to 42-6
  - adding to ActiveX controls 38-8 to 38-9
  - adding to interfaces 36-9 to 36-10
  - array 42-2, 42-8
  - as classes 42-2
  - changing 47-6 to 47-12, 48-2, 48-3
  - COM 33-3, 34-9
    - By Reference Only 34-9
    - Write By Reference 34-9
  - common dialog boxes 52-1
  - declaring 42-3, 42-3 to 42-6, 42-7, 42-12, 43-8, 49-4
    - stored 42-12
    - user-defined types 49-3
  - default values 42-7, 42-11 to 42-12
    - redefining 48-2, 48-3
  - editing
    - as text 47-8
  - events and 43-1, 43-2
  - HTML tables 28-19
  - inherited 42-2, 49-2, 50-2
  - interfaces 42-10
  - internal data storage 42-4, 42-6
  - loading 42-13
  - nodefault 42-7
  - objects and 3-5
  - overview 40-6
  - providing help 47-4
  - published 50-2
  - read and write 42-5
  - reading values 47-8
  - read-only 41-6, 42-6, 51-2
  - redeclaring 42-11, 43-5
  - rich text controls 3-31
  - setting 3-24 to 3-25
  - specifying values 42-11, 47-8
  - storing 42-12

- storing and loading
  - unpublished 42-13 to 42-15
- subcomponents 42-8
- types 42-2, 42-8, 47-8, 49-3
- updating 40-7
- viewing 47-8
- wrapper components 52-3
- write-only 42-6
- writing values 42-6, 47-8
- property editors 3-25, 42-2, 47-6 to 47-12
  - as derived classes 47-7
  - attributes 47-10
  - dialog boxes as 47-9
  - registering 47-11 to 47-12
- property page wizard 38-12
- property pages 38-11 to 38-14
  - ActiveX controls 35-6, 38-3, 38-14
  - adding controls 38-12 to 38-14
  - associating with ActiveX control properties 38-13
  - creating 38-12 to 38-14
  - imported controls 35-4
  - updating 38-13
  - updating ActiveX controls 38-13
- property settings
  - reading 42-8
  - writing 42-8
- Proportional property 8-3
- protected 3-9
  - directive 43-5
  - events 43-5
  - keyword 42-3, 43-5
  - part of classes 41-5
- protected section 4-2
- protocols
  - choosing 25-8 to 25-11
  - connection components 25-8 to 25-11, 25-24
  - Internet 27-2, 32-1
  - network connections 20-15
- Provider property 21-4
- ProviderFlags property 24-5, 24-10
- ProviderName property 14-12, 23-24, 24-3, 25-23, 25-37, 26-9
- providers 24-1 to 24-13
  - applying updates 24-4, 24-8 to 24-11
  - screening updates 24-11
  - associating with datasets 24-2

- associating with XML documents 24-2, 26-8
- client datasets and 23-23 to 23-30
- client-generated events 24-12
- data constraints 24-12
- error handling 24-11
- external 14-11, 23-17, 23-23, 24-1
- internal 23-17, 23-23, 24-1
- local 23-24, 24-3
- remote 23-24, 24-3, 25-6
- supplying data to XML documents 26-9 to 26-10
- using update objects 20-11
- XML 26-8
- providing 24-1, 25-3
- proxy 33-7, 33-8
- transactional objects 39-2
- PString 4-47
- public 3-9
  - directive 43-5
  - keyword 43-5
  - part of classes 41-6
  - properties 42-11
- public section 4-2
- published 3-9, 42-3
  - directive 42-3, 43-5, 52-3
  - keyword 43-5
  - part of classes 41-6
  - properties 42-11, 42-12
  - example 49-2, 50-2
- PVCS Version Manager 2-5
- PWideChar 4-42
- PWideString 4-47

## Q

- QReport page (Component palette) 3-30
- Qt library 10-22
- Qt widget 10-13
- qualifiers 3-8 to 3-9
- queries 18-23, 18-41 to 18-48
  - BDE-based 20-2, 20-8 to 20-11
  - concurrent 20-17
  - live result sets 20-10 to 20-11
- bi-directional cursors 18-48
- executing 18-47 to 18-48
- filtering vs. 18-12
- heterogeneous 20-9 to 20-10
- HTML tables 28-20

- master/detail relationships 18-46 to 18-47
- optimizing 18-47, 18-48
- parameterized 18-42
- parameters 18-43 to 18-46
  - binding 18-44
  - from client datasets 23-27
  - master/detail relationships 18-46 to 18-47
  - named 18-44
  - properties 18-44 to 18-45
  - setting at design time 18-44
  - setting at runtime 18-45
  - unnamed 18-44
- preparing 18-47
- result sets 18-48
- running
  - update objects 20-45
- specifying 18-42 to 18-43, 22-6
- specifying the database 18-41
- unidirectional cursors 18-48
- update objects 20-46 to 20-47
- Web applications 28-20
- Query Builder 18-43
- query part (URLs) 27-3
- Query property
  - update objects 20-46
- QueryInterface method 4-20, 4-24, 4-26, 33-4
  - aggregation 33-9
- query-type datasets
  - See queries

## R

- radio buttons 3-35, 15-2
  - data-aware 15-13 to 15-14
  - grouping 3-39
  - selecting 15-13, 15-14
- radio groups 3-39
- raise reserved word 4-17
- ranges 18-30 to 18-34
  - applying 18-34
  - boundaries 18-32
  - canceling 18-34
  - changing 18-33
  - filters vs. 18-30
  - indexes and 18-30
  - null values 18-31, 18-32
  - specifying 18-30 to 18-33
- raster operations 45-7
- .RC files 6-42

- RDBMS 14-3, 25-1
- RDSCONNECTION property 21-16
- Read method
  - TFileStream 4-55
- read method 42-6
- read reserved word 42-8, 49-4
- ReadBuffer method
  - TFileStream 4-56
- ReadCommitted 17-9
- reading property settings 42-6
- README 13-14, 13-15
- read-only fields 15-5
- read-only properties 41-6, 42-6, 51-2
- ReadOnly property 3-31, 51-3, 51-8, 51-9
  - data grids 15-20, 15-25
  - data-aware controls 15-5
  - data-aware memo controls 15-8
  - data-aware rich edit controls 15-9
  - fields 19-12
  - tables 18-37
- read-only tables 18-37
- read-only-datasets
  - updating 14-10
- realizing palettes 45-5
- ReasonString property 28-11
- rebars 6-42, 6-48
- ReceiveBuf method 32-8
- Receivln method 32-8
- RecNo property
  - client datasets 23-2
- Reconcile method 10-29, 20-33
- RecordCount property
  - TBatchMove 20-51
- records
  - adding 18-18 to 18-19, 18-21
  - appending 18-19
    - batch operations 20-8, 20-49
  - comparison to objects 3-5
  - copying
    - batch operations 20-8, 20-49
  - deleting 18-19 to 18-20, 18-40
    - batch operations 20-8, 20-50
  - displaying 15-26
  - fetching 22-8, 23-25 to 23-26
    - asynchronous 21-11 to 21-12
  - filtering 18-12 to 18-15
  - finding 18-10 to 18-12, 18-27 to 18-29
  - iterating through 18-7
  - marking 18-9 to 18-10
  - moving through 15-28, 18-5 to 18-8, 18-16
  - posting 15-6, 18-20
    - data grids 15-25
    - when closing datasets 18-20
  - reconciling updates 23-22
  - refreshing 15-6, 23-29 to 23-30
  - repeating searches 18-29
  - search criteria 18-10, 18-11
  - sorting 18-25 to 18-27
  - synchronizing current 18-40
  - Type Library editor 34-10, 34-17, 34-24
  - updating 18-21 to 18-22, 24-8 to 24-11, 25-37 to 25-38
    - batch operations 20-8, 20-49
    - client datasets 23-19 to 23-23
    - delta packets 24-8, 24-9
    - from XML documents 26-10
    - identifying tables 24-11
    - multiple 24-6
    - queries and 20-11
    - screening updates 24-11
  - Web adapters 29-8
- RecordSet property 21-19
- Recordset property 21-10
- RecordsetState property 21-10
- RecordStatus property 21-12, 21-13
- Rectangle method 8-5, 8-11, 45-3
- rectangles
  - drawing 8-11, 49-9
- redefining methods 41-8
- redrawing images 45-7
- Reduced XML Data file
  - See XDR file
- reference counting
  - COM objects 4-20, 33-4
  - interfaces 4-24 to 4-26
- reference fields 19-22, 19-27 to 19-28
  - displaying 15-23
- references
  - forms 6-2
  - packages 11-3
- referential integrity 14-5
- Refresh method 15-6, 23-29
- RefreshLookupList
  - property 19-9
- RefreshRecord method 23-29, 24-3
- Register method 8-3
- Register procedure 40-12, 47-2
- RegisterComponents
  - procedure 11-6, 40-12, 47-2
- RegisterConversionType
  - function 4-59, 4-60
- RegisterHelpViewer 5-31
- registering
  - Active Server Objects 37-7 to 37-8
  - ActiveX controls 38-14
  - COM objects 36-16 to 36-17
  - component editors 47-18
  - components 40-12
  - conversion families 4-59
  - property editors 47-11 to 47-12
- registering Help objects 5-27
- RegisterNonActiveX
  - procedure 38-3
- RegisterPooled procedure 25-8
- RegisterPropertyEditor
  - procedure 47-11
- RegisterTypeLib function 33-17
- RegisterViewer function 5-27
- Registry 12-10
- registry 10-15
- REGSERV32.EXE 13-5
- relational databases 14-1
- Release method 4-20, 4-24, 4-25, 33-4
  - TCriticalSection 9-7
- release notes 13-15
- releasing mouse buttons 8-25
- relocatable code 10-20
- remotable class registry 31-5, 31-7
- remotable classes
  - exceptions 31-7
  - registering 31-5
- remote applications
  - TCP/IP 32-1
- remote connections 32-2 to 32-3
  - multiple 32-5
  - opening 32-6, 32-7
  - sending/receiving information 32-9
  - terminating 32-7

- Remote Data Module
  - wizard 25-13 to 25-14
- remote data modules 5-19, 25-3, 25-5, 25-11, 25-13 to 25-16
  - child 25-21
  - instanting 25-14
  - multiple 25-21 to 25-22, 25-30 to 25-31
  - parent 25-21
  - pooling 25-8
  - stateless 25-7, 25-8, 25-19 to 25-21
  - threading models 25-13, 25-14
- Remote Database Management system 14-3
- remote database servers 14-2
- remote servers 20-9, 33-7
  - maintaining
    - connections 20-18
    - unauthorized access 17-4
- RemoteHost property
  - client sockets 32-6
- RemotePort property
  - client sockets 32-6
- RemoteServer property 23-24, 25-23, 25-28, 25-34, 25-37, 26-9
- RemoveAllPasswords
  - method 20-22
- RemovePassword
  - method 20-22
- RenameFile function 4-52, 4-53
- repainting controls 49-7, 49-9, 50-4
- RepeatableRead 17-9
- reports 14-16
- Repository 5-19 to 5-22, 6-12
  - adding items 5-19
  - using items from 5-20 to 5-21
- Repository dialog 5-19
- RepositoryID property 25-24, 25-27
- request
  - actions and HTML 29-15
- Request for Comment (RFC) documents 27-2
- request headers 28-9
- request messages 28-2, 28-3, 37-4
  - action items and 28-5
  - contents 28-10
  - dispatching 28-5
  - header information 28-8 to 28-10
  - HTTP overview 27-4 to 27-6
  - processing 28-5
  - responding to 28-7 to 28-8, 28-12
  - types 28-9
  - XML brokers 25-37
- request objects
  - header information 28-4
- RequestLive property 20-10
- RequestRecords method 25-37
- requests
  - adapters 29-15
  - dispatching 29-12
  - images
    - HTTP requests
    - images 29-16
- Requires list (packages) 11-6, 11-7, 11-8, 47-19
- ResetEvent method 9-9
- resizing controls 3-34, 13-12, 50-4
  - graphics 45-7
- ResolveToDataSet
  - property 24-4
- resolving 24-1, 25-4
- resource dispensers 39-5
  - ADO 39-6
  - BDE 39-5
- Resource DLLs
  - dynamic switching 12-13
  - wizard 12-10
- resource files 6-42
  - loading 6-42
- resource modules 12-10, 12-12
- resource pooling 39-5 to 39-7
- resources 40-7, 45-1
  - caching 45-2
  - freeing 52-5
  - isolating 12-10
  - localizing 12-10, 12-12, 12-13
  - strings 12-10
  - system, optimizing 40-4
- resourcestring reserved word 12-10
- response headers 28-12
- response messages 28-3, 37-5
  - contents 28-12, 28-13 to 28-20
  - creating 28-10 to 28-12, 28-13 to 28-20
  - database information 28-17 to 28-20
  - header information 28-11 to 28-12
  - sending 28-8, 28-12
  - status information 28-11
- response templates 28-13
- responses
  - actions 29-15
  - adapters 29-15
  - images
    - HTTP responses
    - images 29-17
- RestoreDefaults method 15-21
- Result parameter 46-6
- result parameters 18-50
- Resume method 9-10, 9-11
- retaining aborts 21-6
- retaining commits 21-6
- ReturnValue property 9-9
- RevertRecord method 10-29, 20-33, 23-6
- RFC documents 27-2
- rich text controls 3-32, 7-6, 15-9
  - properties 3-31
- rich text edit controls 3-31
  - properties 3-31
- rich-text memo fields 15-2
- RightPromotion method 4-31, 4-33
- role-based security 39-14
- Rollback method 17-8
- RollbackTrans method 17-8
- root directory 10-16
- rounded rectangles 8-11
- RoundRect method 8-5, 8-11
- RowAttributes property 28-19
- RowCount property 15-12, 15-27
- RowHeights property 3-43, 7-14
- RowRequest method 24-3
- rows 3-43
  - decision grids 16-11
  - Web adapters 29-8
- Rows property 3-43
- RowsAffected property 18-48
- RPC 33-8
- RTL 10-6
- RTTI 41-6
  - invokable interfaces 31-2
- rubber banding example 8-23 to 8-28
- \$RUNONLY compiler
  - directive 11-10
- runtime interfaces 41-6
- runtime packages 11-1, 11-2 to 11-4
- runtime type information 41-6

---

**S**

- safe arrays 34-13
- safe references 39-20

- SafeArray 34-13
- safecall calling convention 34-9, 38-9
- SafeRef method 39-20
- Samples page (Component palette) 3-30
- Save as Template command (Menu designer) 6-37, 6-39
- Save Attributes command 19-13
- Save Template dialog box 6-40
- SaveConfigFile method 20-25
- SavePoint property 23-6
- SaveToFile method 8-19
  - ADO datasets 21-14
  - client datasets 14-9, 23-33
  - graphics 45-4
  - strings 3-48
- SaveToStream method
  - client datasets 23-33
- scalability 14-11
- ScaleBy property
  - TCustomForm 13-12
- Scaled property
  - TCustomForm 13-12
- ScanLine property
  - bitmap 8-9
  - bitmap example 8-18
- schema information 22-12 to 22-17
  - fields 22-15
  - indexes 22-16
  - stored procedures 22-14, 22-16 to 22-17
  - tables 22-14
- ScktSrvr.exe 25-9, 25-13, 25-25
- SCM 5-4
- scope (objects) 3-8 to 3-9
- screen
  - refreshing 8-2
  - resolution 13-11
    - programming for 13-11, 13-12
- Screen variable 6-3, 12-9
- script objects 29-11
- ScriptAlias directive 13-10
- scripting 29-9
- scripts
  - active 29-9
  - editing and viewing 29-10
  - generating in
    - WebSnap 29-10
- scripts (URLs) 27-3
- scroll bars 3-32
  - text windows 7-7 to 7-8
- scrollable bitmaps 8-16
- ScrollBars property 3-43, 7-7
  - data-aware memo controls 15-8
- SDI applications 5-1 to 5-2
- search lists (Help systems) 47-5
- search path 10-15
- search path separator 10-16
- Sections property 3-41
- security
  - databases 14-3 to 14-4, 17-4 to 17-5
    - local tables 20-21 to 20-23
  - DCOM 25-36
  - multi-tiered applications 25-2
  - registering socket connections 25-9
  - SOAP connections 25-26
  - transactional data modules 25-6, 25-9
  - transactional objects 39-14 to 39-15
  - Web connections 25-10, 25-26
- Seek method
  - ADO datasets 18-27
- seeking
  - files 4-56
- Select Menu command (Menu designer) 6-37
- Select Menu dialog box 6-37
- SELECT statements 18-42
- SelectAll method 3-32
- SelectCell method 50-12, 51-3
- Selection property 3-43
- SelectKeyword 5-27
- selectors
  - Help 5-27
- SelEnd property 3-33
- Self parameter 40-13
- SelLength property 3-32, 7-8
- SelStart property 3-32, 3-33, 7-8
- SelText property 3-32, 7-8
- SendBuf method 32-8
- Sender parameter 3-27
  - example 8-7
- SendIn method 32-8
- SendStream method 32-8
- separator bars (menus) 6-33
- server applications
  - architecture 25-5
  - COM 33-5 to 33-9, 36-1 to 36-17
  - interfaces 32-2
  - multi-tiered 25-5 to 25-11, 25-11 to 25-18
  - registering 25-11, 25-22
  - services 32-1
    - Web Services 31-2 to 31-8
- server connections 32-2, 32-3
  - port numbers 32-5
- server sockets 32-7
  - accepting client requests 32-7
  - accepting clients 32-9
  - error messages 32-8
  - event handling 32-9
  - specifying 32-6
  - Windows socket objects 32-7
- server types 29-2
- ServerGUID property 25-24
- ServerName property 25-24
- servers
  - Internet 27-1 to 27-9
  - Web application debugger 29-2
- Servers page (Component palette) 3-30
- server-side 29-9
- server-side scripting 29-9
- service applications 5-4 to 5-8
  - example 5-6
  - example code 5-4, 5-6
- Service Control Manager 5-4
- Service Start name 5-8
- service threads 5-6
- services 5-4 to 5-8
  - example 5-6
  - example code 5-4, 5-6
  - implementing 32-1 to 32-2, 32-7
  - installing 5-4
  - name properties 5-8
  - network servers 32-1
  - ports and 32-2
  - requesting 32-6
  - uninstalling 5-4
- Session variable 20-3, 20-16
- SessionName property 20-3, 20-13, 20-28, 28-17
- sessions 20-16 to 20-29
  - activating 20-17 to 20-18
  - associated databases 20-20 to 20-21
  - closing 20-18
  - closing connections 20-19
  - creating 20-27, 20-28
  - current state 20-17
  - databases and 20-13

- datasets and 20-3 to 20-4
- default 20-3, 20-13, 20-16 to 20-17
- default connection
  - properties 20-18
- getting information 20-26 to 20-27
- implicit database
  - connections 20-13
- managing aliases 20-24
- managing connections 20-19 to 20-21
- methods 20-13
- multiple 20-13, 20-27, 20-28 to 20-29
- multi-threaded
  - applications 20-13, 20-28 to 20-29
- naming 20-28, 28-17
- opening connections 20-19
- passwords 20-21
- restarting 20-18
- Web applications 28-17
- Sessions property 20-29
- sessions service 29-4
- Sessions variable 20-17, 20-28
- set types 42-2
- SetAbort method 39-4, 39-7, 39-11
- SetBrushStyle method 8-8
- SetComplete method 25-18, 39-4, 39-7, 39-11
- SetData method 19-16
- SetEvent method 9-9
- SetFields method 18-21
- SetFloatValue method 47-8
- SetKey method 18-27
  - EditKey vs. 18-29
- SetLength procedure 4-46
- SetMethodValue method 47-8
- SetOptionalParam
  - method 23-15
- SetOrdValue method 47-8
- SetPenStyle method 8-7
- SetProvider method 23-24
- SetRange method 18-32
- SetRangeEnd method 18-31
  - SetRange vs. 18-32
- SetRangeStart method 18-30
  - SetRange vs. 18-32
- sets 42-2
- SetSchemaInfo method 22-12
- SetStrValue method 47-8
- SetValue method 47-8
- Shape property 3-44
- shapes 3-44, 8-11 to 8-12, 8-14
  - drawing 8-11, 8-14
  - filling 8-8
  - filling with bitmap
    - property 8-9
  - outlining 8-5
- shared object files 10-9, 10-15
- shared objects 10-22
- shared property groups 39-6
- Shared Property Manager 39-6 to 39-7
  - example 39-6 to 39-7
- sharing forms and dialogs 5-19 to 5-22
- shell scripts 10-14
- Shift states 8-24
- short strings 4-40
- Shortcut property 6-34
- shortcuts
  - adding to menus 6-33 to 6-34
- ShortString 4-41
- Show method 6-7, 6-8
- ShowAccelChar property 3-42
- ShowButtonsProperty 3-38
- ShowColumnHeaders
  - property 3-39
- ShowFocus property 15-27
- ShowHint property 3-42, 15-30
- ShowHintChanged
  - property 10-21
- ShowLines property 3-38
- ShowModal method 6-6
- ShowRoot property 3-38
- ShutDown 5-24
- signalling events 9-9
- signals 10-15
- Simple Object Access Protocol
  - See SOAP
- simple types 42-2
- single document interface 5-1 to 5-2
- single-tiered applications 14-3, 14-9, 14-12
  - file-based 14-9
- Size property
  - fields 19-12
  - parameters 18-45, 18-51
- slow processes
  - using threads 9-1
- SOAP 31-1
  - connecting to application
    - servers 25-26
  - multi-tiered
    - applications 25-10
- SOAP connections 25-10, 25-26
- SOAP Data Module
  - wizard 25-15
- SOAP data modules 25-5
- SOAP fault packets 31-7
- socket components 32-5 to 32-7
- socket connections 25-9, 25-25, 32-2 to 32-3
  - closing 32-7
  - endpoints 32-3, 32-5
  - multiple 32-5
  - opening 32-6, 32-7
  - sending/receiving
    - information 32-9
  - types 32-2
- socket dispatcher
  - application 25-9, 25-13, 25-25
- socket objects
  - clients 32-6
- sockets 32-1 to 32-10
  - accepting client
    - requests 32-3
  - assigning hosts 32-4
  - describing 32-3
  - error handling 32-8
  - event handling 32-8 to 32-9, 32-10
  - implementing services 32-1 to 32-2, 32-7
  - network addresses 32-3, 32-4
  - providing information 32-4
  - reading from 32-10
  - reading/writing 32-9 to 32-10
  - writing to 32-10
- SoftShutDown 5-24
- software license
  - requirements 13-14
- sort order 12-10
  - client datasets 23-7
  - descending 23-8
  - setting 18-27
  - TSQTable 22-7
- Sorted property 3-37, 15-11
- SortFieldNames property 22-7
- source code
  - editing 2-3
  - optimizing 8-15
  - reusing 6-12
  - viewing
    - specific event
      - handlers 3-26
- source datasets, defined 20-48
- source files
  - changing 2-3



- packages 11-2, 11-6, 11-8, 11-12
- source files, sharing 10-13
- SourceXml property 26-6
- SourceXmlDocument
  - property 26-6
- SourceXmlFile property 26-6
- Spacing property 3-35
- SparseCols property 16-9
- SparseRows property 16-9
- speed buttons 3-35
  - adding to toolbars 6-43 to 6-45
  - assigning glyphs 6-44
  - centering 6-44
  - engaging as toggles 6-45
  - event handlers 8-13
  - for drawing tools 8-13
  - grouping 6-45
  - initial state, setting 6-44
  - operational modes 6-43
- spin edit controls 3-33
- splitters 3-34
- SPX/IPX 20-15
- SQL 14-2, 20-8
  - executing commands 17-10 to 17-11
  - local 20-9
  - standards 24-12
    - Decision Query editor and 16-6
- SQL Builder 18-43
- SQL Explorer 20-53, 25-3
  - defining attribute sets 19-13
- SQL Links 13-8, 20-1
  - deploying 13-8, 13-14
  - driver files 13-9
  - drivers 20-9, 20-15, 20-31
- SQL Monitor 20-53
- SQL property 18-42 to 18-43
  - changing 18-47
- SQL queries 18-42 to 18-43
  - copying 18-43
  - executing 18-47 to 18-48
  - loading from files 18-43
  - modifying 18-43
  - optimizing 18-48
  - parameters 18-43 to 18-46, 20-41 to 20-42
    - binding 18-44
    - master/detail
      - relationships 18-46 to 18-47
    - setting at design time 18-44
      - setting at runtime 18-45
      - preparing 18-47
      - result sets 18-48
      - update objects 20-45
- SQL servers
  - logging in 14-4
- SQL statements
  - client-supplied 23-31, 24-6
  - decision datasets 16-4, 16-5
  - executing 22-9 to 22-10
  - generating
    - providers 24-4, 24-9 to 24-10
    - TSQldataSet 22-8
  - parameters 17-11
  - passthrough SQL 20-30
  - provider-generated 24-11
  - update objects and 20-40 to 20-43
- SQLConnection property 22-2, 22-17
- SQLPASSTHRUMODE 20-31
- squares, drawing 49-9
- standard components 3-28 to 3-30
- standard events 43-4, 43-4 to 43-6
  - customizing 43-6
- Standard page (Component palette) 3-29
- StartTransaction method 17-6, 17-7
- state information
  - communicating 24-7, 24-8, 25-19 to 25-21
  - managing 39-5
  - mouse events 8-24
  - shared properties 39-6
  - transactional objects 39-11
  - transactions 39-11
- State property 3-35
  - datasets 18-3, 19-8
  - grid columns 15-15
  - grids 15-15, 15-17
- stateless objects 39-11
- static binding 25-29
  - COM 33-16
- static methods 41-7
- static text 3-42
- static text component 3-41
- status bars 3-42
  - internationalizing 12-9
  - owner draw 7-11
- status information 3-42
- StatusCode property 28-11
- StatusFilter property 10-28, 20-32, 21-12, 23-6, 23-18, 24-8
- StdConvs unit 4-59, 4-60
- Step property 3-42
- StepBy method 3-42
- StepIt method 3-42
- storage media 3-55
- stored directive 42-12
- stored procedures 14-5, 18-23, 18-48 to 18-53
  - BDE-based 20-2, 20-11 to 20-12
    - parameter binding 20-12
- creating 22-11
- dbExpress 22-7
- executing 18-53
- listing 17-13
- overloaded 20-12
- parameters 18-50 to 18-52
  - design time 18-50 to 18-51
  - from client datasets 23-27
  - properties 18-51
  - runtime 18-52
- preparing 18-52 to 18-53
- specifying the
  - database 18-49

- stored procedure-type datasets
- See* stored procedures
- StoredProcName
- property 18-49
- StrByteType 4-44
- streams 3-55
- Stretch property 15-9
- StretchDraw method 8-5, 45-3, 45-7
- string fields
- size 19-6
- string grids 3-43
- String List editor
- displaying 15-10
- string lists 3-47 to 3-52
- adding objects 7-13
- adding to 3-51
- associated objects 3-52
- copying 3-51
- creating 3-48 to 3-50
- deleting strings 3-51
- finding strings 3-50
- iterating through 3-50
- loading from files 3-48
- long-term 3-49
- moving strings 3-51
- owner-draw controls 7-12 to 7-13
- position in 3-50, 3-51

- saving to files 3-48
- short-term 3-48
- sorting 3-51
- substrings 3-50
- string operators 4-50
- string properties 4-41
- string reserved word 4-41
  - default type 4-40
  - property types 4-41
- strings 4-39, 42-2, 42-8
  - 2-byte conversions 12-3
  - associating graphics 7-13
  - compiler directives 4-49
  - declaring and
    - initializing 4-46
  - extended character sets 4-50
  - files 4-56
  - local variables 4-48
  - long 4-41
  - memory corruption 4-49
  - mixing and converting
    - types 4-47
  - PChar conversions 4-47
  - reference counting
    - issues 4-41, 4-47
  - returning 42-8
  - routines
    - case sensitivity 4-43
    - Multi-byte character support 4-44
    - runtime library 4-42
  - size 7-8
  - sorting 12-10
  - starting position 7-8
  - translating 12-2, 12-8, 12-10
  - truncating 12-3
  - types overview 4-40
  - variable parameters 4-49
- Strings property 3-50
- StrNextChar function 10-17
- Structured Query Language
  - See SQL
- stubs
  - COM 33-8
  - transactional objects 39-2
- Style property 3-37, 7-12
  - brushes 3-44, 8-8
  - combo boxes 3-38, 15-11
  - list boxes 3-37
  - pens 8-5
  - tool buttons 6-47
  - web items 25-40
- style sheets 25-40
- StyleChanged property 10-21
- StyleRule property 25-40

- styles 10-6
- Styles property 25-40
- StylesFile property 25-41
- subclassing Windows
  - controls 40-4
- subcomponents
  - properties 42-8
- submenus 6-34
- subscriber objects 35-14 to 35-15
  - persistent
    - subscriptions 35-15
  - per-user subscriptions 35-15
  - transient subscriptions 35-14
- Subtotals property 16-12
- summary values
  - crosstabs 16-2, 16-3
  - decision cubes 16-19
  - decision graphs 16-15
  - maintained aggregates 23-13
- support services 1-3
- SupportCallbacks
  - property 25-18
- Suspend method 9-11
- Sybase driver
  - deploying 13-9
- symbolic links 10-16
- Synchronize method 9-4
- synchronizing data
  - on multiple forms 15-4
- system events 10-21
- system notifications 10-21
- System page (Component palette) 3-29
- system resources,
  - conserving 40-4

## T

- tab controls 3-40
  - owner-draw 7-11
- tab order 3-22
- tab sets 3-40
- Table HTML tag
  - (<TABLE>) 28-14
- table producers 28-18 to 28-20
  - setting properties 28-19
- TableAttributes property 28-19
- TableName property 18-25, 18-38, 22-7
- TableOfContents 5-27
- tables 18-23, 18-24 to 18-41
  - BDE-based 20-2, 20-4 to 20-8
    - access rights 20-6
    - appending records 20-8
    - batch operations 20-8
    - binding 20-5
  - closing 20-5
  - copying records 20-8
  - deleting records 20-8
  - exclusive locks 20-6
  - index-based
    - searches 18-27
  - updating records 20-8
- creating 18-37 to 18-39
  - indexes 18-38
  - persistent fields 18-38
- dbExpress 22-6 to 22-7
- defining 18-37 to 18-38
- deleting 18-40
- displaying in grids 15-16
- emptying 18-40
- field and index
  - definitions 18-38
  - preloading 18-38
- indexes 18-25 to 18-36
- inserting records 18-18 to 18-19, 18-21
- listing 17-13
- master/detail
  - relationships 18-34 to 18-36
- nested 18-36
- non-database grids 3-43
- ranges 18-30 to 18-34
- read-only 18-37
- searching 18-27 to 18-29
- sorting 18-25, 22-7
- specifying the
  - database 18-24
  - synchronizing 18-41
- table-type datasets
  - See tables
- TableType property 18-37, 20-5 to 20-6
- TabOrder property 3-22
- tabs
  - draw-item events 7-15
- Tabs property 3-40
- TabStop property 3-22
- TabStopChanged
  - property 10-21
- tabular display (grids) 3-43
- tabular grids 15-26
- TAction 6-20
- TActionClientItem 6-22
- TActionList 6-18
- TActionMainMenuBar 6-16, 6-17, 6-18, 6-19, 6-21
- TActionManager 6-16, 6-18, 6-19
- TActionToolBar 6-16, 6-17, 6-18, 6-19, 6-21

- TActiveForm 38-3, 38-6
- TAdapterDispatcher 29-13
- TAdapterPageProducer 29-10
- TADOCCommand 21-2, 21-7, 21-9, 21-16 to 21-20
- TADOCConnection 14-8, 17-1, 21-2, 21-2 to 21-8, 21-9
  - connecting to data stores 21-3 to 21-4
- TADODataSet 21-2, 21-9, 21-15 to 21-16
- TADOQuery 21-2, 21-9
  - SQL command 21-17
- TADOSToredProc 21-2, 21-9
- TADOTable 21-2, 21-9
- Tag property 19-12
- TApacheApplication 27-6
- TApacheRequest 27-6
- TApacheResponse 27-6
- TApplication 5-23, 5-29, 10-6
- TApplicationEvents 6-3
- targets, action lists 6-18
- TASM code 10-17
- TASPObject 37-2
- TBatchMove 20-47 to 20-52
  - error handling 20-51 to 20-52
- TBCDField
  - default formatting 19-15
- TBDEClientDataSet 20-2
- TBDEDataSet 18-2
- TBevel 3-45
- TBitmap 45-4
- TBrush 3-44
- tbsCheck constant 6-47
- TCalendar 50-1
- TCanvas
  - using 3-54
- TCGIApplication 27-6, 27-7
- TCGIRequest 27-7
- TCGIResponse 27-7
- TCharProperty type 47-7
- TClassProperty type 47-7
- TClientDataSet 23-17
- TClientDataset 5-18
- TClientSocket 32-6
- TColorProperty type 47-7
- TComObject
  - aggregation 4-24
- TComponent 3-12, 3-15, 40-5
- TComponentProperty type 47-7
- TControl 3-16, 3-18, 40-4, 43-4, 43-5
  - common events 3-20
  - common properties 3-18
- TConvTypeInfo values 4-59
- TConvTypeInfo 4-62
- TCoolBand 3-36
- TCoolBar 6-43
- TCorbaConnection 25-27
- TCorbaDataModule 25-5
- TCP/IP 20-15, 32-1
  - clients 32-6
    - connecting to application server 25-25
  - multi-tiered applications 25-9
  - servers 32-7
- TCurrencyField
  - default formatting 19-15
- TCustomADODataset 18-2
- TCustomClientDataSet 18-2
- TCustomContentProducer 28-13
- TCustomControl 40-4
- TCustomEdit 10-8
- TCustomGrid 50-1, 50-2
- TCustomIniFile 3-52
- TCustomizeDlg 6-22
- TCustomListBox 40-3
- TCustomVariantType 4-27, 4-28 to 4-36
- TDatabase 14-8, 17-1, 20-3, 20-12 to 20-16
  - DatabaseName property 20-3
  - temporary instances 20-20
  - dropping 20-20
- TDataSet 18-1
  - descendants 18-2 to 18-3
- TDataSetProvider 24-1, 24-2
- TDataSetTableProducer 28-20
- TDataSource 15-3 to 15-5
- TDateField
  - default formatting 19-15
- TDateTime type 50-5
- TDateTimeField
  - default formatting 19-15
- TDBChart 14-15
- TDBCheckBox 15-2, 15-12 to 15-13
- TDBComboBox 15-2, 15-10, 15-10 to 15-11
- TDBCtrlGrid 15-2, 15-26 to 15-27
  - properties 15-27
- TDBEdit 15-2, 15-8
- TDBGrid 15-2, 15-15 to 15-26
  - events 15-25
  - properties 15-19
- TDBGridColumn 15-15
- TDBImage 15-2, 15-9 to 15-10
- TDBListBox 15-2, 15-10, 15-10 to 15-11
- TDBLookupComboBox 15-2, 15-10, 15-11 to 15-12
- TDBLookupListBox 15-2, 15-10, 15-11 to 15-12
- TDBMemo 15-2, 15-8 to 15-9
- TDBNavigator 15-2, 15-28 to 15-30, 18-5, 18-6
- TDBRadioGroup 15-2, 15-13 to 15-14
- TDBRichEdit 15-2, 15-9
- TDBText 15-2, 15-8
- TDCOMConnection 25-24
- TDecisionCube 16-1, 16-4, 16-7 to 16-8
  - events 16-7
- TDecisionDrawState 16-12
- TDecisionGraph 16-1, 16-2, 16-13 to 16-18
- TDecisionGrid 16-1, 16-2, 16-10 to 16-13
  - events 16-12
  - properties 16-12
- TDecisionPivot 16-1, 16-2, 16-9 to 16-10
  - properties 16-10
- TDecisionQuery 16-1, 16-4, 16-6
- TDecisionSource 16-1, 16-9
  - events 16-9
  - properties 16-9
- TDefaultEditor 47-15
- TDependency\_object 5-8
- TDragObject 7-3
- TDragObjectEx 7-3
- technical support 1-3
- temperature units 4-61
- templates 5-19, 5-21
  - component 6-12, 6-13
  - decision graphs 16-16
  - HTML 28-13 to 28-17, 29-9
  - menus 6-31, 6-37, 6-38 to 6-40
    - loading 6-38
  - page producers 29-6
  - producer 29-6
  - programming 5-3
  - Web Broker applications 28-2
- temporary objects 45-6
- TEnumProperty type 47-7
- terminal type 10-15
- Terminate method 9-6
- Terminated property 9-6

- test server, Web Application
  - Debugger 29-2
- testing
  - components 40-12, 40-14, 52-6
  - values 42-6
- TEvent 9-9
- text
  - copying, cutting, pasting 7-9
  - deleting 7-9
  - in controls 7-6
  - internationalizing 12-9
  - owner-draw controls 7-11
  - printing 3-32
  - reading right to left 12-6
  - searching for 3-32
  - selecting 7-8, 7-8 to 7-9
  - working with 7-6 to 7-11
- text controls 3-31 to 3-32
- text property 3-31, 3-32, 3-37, 3-42
- TextHeight method 8-5, 45-3
- TextOut method 8-5, 45-3
- TextRect method 8-5, 45-3
- TextWidth method 8-5, 45-3
- TField 18-1, 19-1 to 19-28
  - events 19-15 to 19-16
  - methods 19-16
  - properties 19-1, 19-10 to 19-15
  - runtime 19-12
- TFieldDataLink 51-4
- TFile 4-54
- TFileStream 3-55, 4-54
  - file I/O 4-54 to 4-57
- TFloatField
  - default formatting 19-15
- TFloatProperty type 47-7
- TFMTBcdField
  - default formatting 19-15
- TFontNameProperty type 47-7
- TFontProperty type 47-7
- TForm
  - scroll-bar properties 3-33
- TForm component 3-5
- TFrame 6-13
- TGraphic 45-4
- TGraphicControl 40-4, 49-2
- The 9-6
- THeaderControl 3-41
- thin client applications 25-2, 25-32
- thread function 9-4
- thread objects 9-1
  - defining 9-2
  - initializing 9-2
  - limitations 9-2
- Thread Status box 9-12
- thread variables 9-5
- thread-aware objects 9-4
- ThreadID property 9-12
- threading models 36-6 to 36-9
  - ActiveX controls 38-5
  - Automation objects 36-5
  - COM objects 36-3
  - CORBA data modules 25-16
  - remote data modules 25-13
  - system registry 36-7
  - transactional data modules 25-14
  - transactional objects 39-16 to 39-17
- thread-local variables 9-5
  - OnTerminate event 9-6
- threads 9-1 to 9-12
  - activities 39-18
  - avoiding simultaneous access 9-7
  - BDE and 20-13
  - blocking execution 9-7
  - coordinating 9-4, 9-7 to 9-10
  - creating 9-10
  - critical sections 9-7
  - data access components 9-4
  - exceptions 9-6
  - executing 9-10
  - freeing 9-2, 9-3
  - graphics objects 9-5
  - ids 9-12
  - initializing 9-2
  - ISAPI/NSAPI programs 28-2, 28-17
  - limits on number 9-11
  - locking objects 9-7
  - message loop and 9-4
  - priorities 9-1, 9-2
  - overriding 9-11
  - process space 9-4
  - returning values 9-9
  - service 5-6
  - stopping 9-11
  - terminating 9-5
  - using lists 9-5
  - VCL thread 9-4
  - waiting for 9-9
  - multiple 9-9
  - waiting for events 9-9
- thread-safe objects 9-4
- threadvar 9-5
- three-tiered applications *See* multi-tiered applications
- THTMLTableAttributes 28-18
- THTMLTableColumn 28-19
- THTTPrIo 31-9
- THTTPrSoapDispatcher 31-2, 31-3
- THTTPrSOAPPascalInvoker 31-3
- THTTPrSoapPascalInvoker 31-2
- TIBCustomDataSet 18-2
- TIBDatabase 14-8, 17-1
- TickMarks property 3-33
- TickStyle property 3-33
- TIcon 45-4
- tiers 25-1
- TiledDraw method 45-7
- TImage
  - in frames 6-15
- TImageList 6-46
- time
  - internationalizing 12-10
- time conversion 4-59
- time fields
  - formatting 19-14
- timeout events 9-10
- timers 3-23
- times
  - entering 3-39
- TIniFile 3-52
- TIntegerProperty type 47-7, 47-9
- TInterfacedObject 4-24
  - deriving from 4-21
  - dynamic binding 4-22
  - implementing Interface 4-21
- TInvokableClass 31-6
- TInvokeableVariantType 4-27, 4-37 to 4-38
- TISAPIApplication 27-6
- TISAPIRequest 27-6
- TISAPIResponse 27-6
- Title property
  - data grids 15-20
- TKeyPressEvent type 43-3
- TLabel 3-41, 40-4
- .TLB files 33-16, 34-2, 34-27
- TLIBIMP 33-18, 35-5, 36-14
- tlbimp.exe 35-2
- TListBox 40-3
- TLocalConnection 23-24
- TMainMenu 6-18
- TMemIniFile 3-52, 10-7
- TMemoryStream 3-55
- TMessage 46-4, 46-6

- TMetafile 45-4
- TMethodProperty type 47-7
- TMsg 6-5
- TMTSDataModule 25-5
- TMultiReadExclusiveWriteSync  
hronizer 9-8
- TNestedDataSet 18-36
- TNotifyEvent 43-7
- TObject 3-12, 4-1, 41-3
- ToCommon 4-62
- ToggleButton 10-8
- toggles 6-45, 6-47
- TOleContainer 35-15
  - Active Documents 33-14
- TOleControl 35-5, 35-6
- TOleServer 35-5
- tool buttons 6-46
  - adding images 6-46
  - disabling 6-46
  - engaging as toggles 6-47
  - getting help with 6-49
  - grouping/ungrouping 6-47
  - in multiple rows 6-47
  - initial state, setting 6-46
  - wrapping 6-47
- Toolbar 6-17
- toolbars 3-36, 6-18, 6-42
  - action lists 6-17
  - adding 6-45 to 6-47
  - adding panels as 6-43 to 6-45
  - context menus 6-49
  - default drawing tool 6-45
  - designing 6-42 to 6-50
  - disabling buttons 6-46
  - hiding 6-49
  - inserting buttons 6-43 to 6-45, 6-46
  - owner-draw 7-11
  - setting margins 6-45
  - speed buttons 3-35
  - transparent 6-47, 6-48
- tool-tip help 3-42
- Top property 3-19, 3-21, 3-22, 6-4, 6-44
- TopRow property 3-43
- TOrdinalProperty type 47-7
- TPageControl 3-40
- TPageDispatcher 29-13
- TPageProducer 28-13
- TPaintBox 3-45
- TPanel 3-39, 6-42
- tpHigher constant 9-3
- tpHighest constant 9-3
- TPicture type 45-4
- tpIdle constant 9-3
- tpLower constant 9-3
- tpLowest constant 9-3
- tpNormal constant 9-3
- TPopupMenu 6-49
- TPrinter 3-54
  - using 3-54
- TPropertyAttributes 47-10
- TPropertyEditor class 47-7
- TPropertyPage 38-12
- tpTimeCritical constant 9-3
- TPublishableVariantType 4-27, 4-39
- TQuery 20-2, 20-8 to 20-11
  - decision datasets and 16-5
- TQueryTableProducer 28-20
- track bars 3-33
- transaction attributes 39-9 to 39-11
  - setting 39-10
- transactional data
  - modules 25-15
- transaction isolation level 17-9
  - local transactions 20-31
  - specifying 17-9
- transaction parameters
  - isolation level 17-9
- Transactional Data Module wizard 25-14 to 25-15
- transactional data
  - modules 25-5, 25-6 to 25-7, 25-7
    - database connections 25-6, 25-7
    - pooling 25-6
  - interface 25-18
  - security 25-9
  - threading models 25-14
  - transaction attributes 25-15
- Transactional Object wizard 39-15 to 39-18
- transactional objects 33-10, 33-14 to 33-15, 39-1 to 39-23
  - activities 39-17 to 39-18
  - administering 33-15, 39-23
  - callbacks 39-21
  - characteristics 39-2 to 39-3
  - creating 39-15 to 39-18
  - debugging 39-21 to 39-22
  - dual interfaces 39-3
  - installing 39-22 to 39-23
  - managing resources 39-3 to 39-8
  - marshaling 39-3
  - object contexts 39-4
  - pooling database connections 39-5 to 39-6
  - releasing resources 39-7
  - requirements 39-3
  - security 39-14 to 39-15
  - sharing properties 39-6 to 39-7
  - stateless 39-11
  - transactions 39-5, 39-8 to 39-14
    - attributes 39-9 to 39-11
    - automatic 39-12
    - client-controlled 39-12, 39-12 to 39-13
    - server-controlled 39-12, 39-13
    - timeouts 39-14, 39-21
    - type libraries 39-3
- transactions 14-4 to 14-5, 17-5 to 17-9
  - ADO 21-6 to 21-7, 21-8
    - retaining aborts 21-6
    - retaining commits 21-6
  - applying updates 17-6, 25-18
  - atomicity 14-4, 39-9
  - automatic 39-12
  - BDE 20-30 to 20-32
    - controlling 20-30 to 20-31
    - implicit 20-30
  - cached updates 20-34
  - client-controlled 39-12, 39-12 to 39-13
  - committing 17-8
  - composed of multiple objects 39-9
  - consistency 14-4, 39-9
  - durability 14-4, 39-9
  - ending 17-7 to 17-9, 39-11 to 39-12
  - IAppServer 25-18
  - isolation 14-4, 39-9
    - levels 17-9
  - local 20-31 to 20-32
  - local tables 17-6
  - MTS and COM+ 39-8 to 39-14
  - multi-tiered
    - applications 25-18
  - nested 17-6
    - committing 17-8
  - object contexts 39-9
  - overlapped 17-7
  - rolling back 17-8 to 17-9
  - server-controlled 39-12, 39-13
  - spanning multiple databases 39-8

- starting 17-6 to 17-7
- timeouts 39-14, 39-21
- transaction components 17-7
- transactional data
  - modules 25-7, 25-15, 25-18
- transactional objects 39-5
- using SQL commands 17-6, 20-30
- transfer records 52-2
- transformation files 26-1 to 26-6
  - TXMLTransform 26-7
  - TXMLTransformClient 26-9
  - TXMLTransformProvider 26-8
  - user-defined nodes 26-5, 26-7 to 26-8
- TransformGetData
  - property 26-9
- TransformRead property 26-8
- TransformSetParams
  - property 26-9
- TransformWrite property 26-8
- transient subscriptions 35-14
- TransIsolation property 17-9
  - local transactions 20-31
- translating character
  - strings 12-2, 12-8, 12-10
  - 2-byte conversions 12-3
- translation 12-9
- translation tools 12-1
- Transliterate property 19-12, 20-48
- transparent backgrounds 12-9
- Transparent property 3-42
- transparent toolbars 6-47, 6-48
- TReader 4-54
- tree views 3-38
  - owner-draw 7-11
- TRegIniFile 10-7
- TRegistry 3-52
- TRegistryIniFile 3-52
- TRegSvr 13-5, 33-18
- TRemotable 31-5
- TRemoteDataModule 25-5
- triangles 8-11
- triggers 14-5
- try reserved word 45-6, 52-5
- TScrollBar 3-33, 3-40
- TSearchRec 4-51
- TServerSocket 32-7
- TService\_object 5-8
- TSession 20-16 to 20-29
  - adding 20-27, 20-28
- TSetElementProperty type 47-7
- TSetProperty type 47-7
- TSharedConnection 25-31
- TSoapDataModule 25-5
- TSocketConnection 25-25
- TSpinEdit control 3-33
- TSQLClientDataSet 22-2
- TSQLConnection 14-8, 17-1, 22-2 to 22-5
  - binding 22-3 to 22-5
  - monitoring messages 22-17
- TSQLDataSet 22-2, 22-6, 22-7
- TSQLMonitor 22-17 to 22-18
- TSQLQuery 22-2, 22-6
- TSQLStoredProc 22-2, 22-7
- TSQLTable 22-2, 22-7
- TSQLTimeStampField
  - default formatting 19-15
- TStoredProc 20-2, 20-11 to 20-12
- TStream 3-55
- TStringList 3-47 to 3-52, 5-25
- TStringProperty type 47-7
- TStrings 3-47 to 3-52
- TStringStream 3-55
- TTabControl 3-40
- TTable 20-2, 20-4 to 20-8
  - decision datasets and 16-5
- TThread 9-2
- TThreadList 9-5, 9-7
- TTimeField
  - default formatting 19-15
- TToolBar 6-18, 6-43, 6-45
- TToolButton 6-43
- TTreeView 3-38
- TTypedComObject
  - type library
    - requirement 33-16
- TUpdateSQL 20-39 to 20-47
  - providers and 20-11
- tutorial
  - WebSnap 29-18
- TVarData record 4-27, 4-28
- TWebActionItem 28-3
- TWebAppDataModule 29-5
- TWebApplication 27-6
- TWebAppPageModule 29-5
- TWebConnection 25-26
- TWebContext 29-13
- TWebDataModule 29-5, 29-7
- TWebDispatcher 29-13, 29-17
- TWebPageModule 29-5, 29-7
- TWebRequest 27-6
- TWebResponse 27-6, 28-3
- TWidgetControl 10-6
- TWinCGIRequest 27-7
- TWinCGIResponse 27-7
- TWinControl 3-17, 10-6, 12-9, 40-4, 43-5
  - common events 3-22
  - common properties 3-21
- two-phase commit 25-18
- two-tiered applications 14-3, 14-9, 14-12
- TWriter 4-54
- TWSDLHTMLPublish 31-7
- TXMLDocument 30-3, 30-8
- TXMLTransform 26-6 to 26-8
  - source documents 26-6
- TXMLTransformClient 26-9 to 26-10
  - parameters 26-9
- TXMLTransformProvider 24-1, 24-2, 26-8
- type declarations
  - enumerated types 8-12
  - objects and 3-10
  - properties 49-3
- type definitions
  - Type Library editor 34-10
- type information 33-15, 34-1
  - dispinterfaces 36-13
  - Help 34-8
  - IDispatch interface 36-14
  - importing 35-2 to 35-6
- type libraries 33-10, 33-12, 33-15 to 33-17, 34-1 to 34-27
  - \_TLB unit 33-22, 34-2, 34-20, 35-2, 35-5 to 35-6, 36-14
  - accessing 33-16 to 33-17, 34-19 to 34-20, 35-2 to 35-6
  - Active Server Objects 37-3
  - ActiveX controls 38-3
  - adding
    - interfaces 34-20
    - methods 34-21 to 34-22
    - properties 34-21 to 34-22
  - benefits 33-17
  - browsers 33-17
  - browsing 33-18
  - contents 33-15, 34-1, 35-5 to 35-6
  - creating 33-16, 34-19
  - deploying 34-27
  - exporting as IDL 34-26
  - generated by wizards 34-1
  - IDL and ODL 33-16
  - importing 35-2 to 35-6
  - including as resources 34-27, 38-3
  - interfaces 33-17

- modifying interfaces 34-20
    - to 34-22
  - opening 34-19 to 34-20
  - optimizing
    - performance 34-9
  - registering 33-18, 34-26
  - registering objects 33-17
  - saving 34-25
  - tools 33-18
  - transactional objects 39-3
  - type checking 33-17
  - uninstalling 33-17
  - unregistering 33-18
  - valid types 34-11 to 34-13
  - when to use 33-16
  - Type Library editor 33-16, 34-2
    - to 34-26
    - aliases 34-10, 34-17
      - adding 34-23
    - application servers 25-17
    - binding attributes 38-11
    - CoClasses 34-9, 34-16
      - adding 34-22 to 34-23
    - COM+ page 39-5, 39-8
    - dispatch interfaces 34-16
    - dispinterfaces 34-9
    - elements 34-8 to 34-11
      - common
        - characteristics 34-8
    - enumerated types 34-10, 34-17
      - adding 34-23
    - error messages 34-5, 34-8, 34-25
    - interfaces 34-8 to 34-9, 34-15
      - adding 34-20
      - modifying 34-20 to 34-22
    - methods
      - adding 34-21 to 34-22
    - modules 34-10 to 34-11, 34-18
      - adding 34-24
    - Object list pane 34-5
    - Object Pascal vs. IDL 34-11, 34-13 to 34-19
    - opening libraries 34-19 to 34-20
    - parts 34-3 to 34-8
    - properties
      - adding 34-21 to 34-22
    - records 34-17
    - records and unions 34-10
      - adding 34-24
    - saving and registering type information 34-24 to 34-26
    - selecting elements 34-5
    - status bar 34-5
    - syntax 34-11, 34-13 to 34-19
    - text page 34-8, 34-21
    - toolbar 34-3 to 34-5
    - type definitions 34-10
    - type information pages 34-6 to 34-8
      - for aliases 34-6 to 34-7
      - for CoClasses 34-6
      - for consts 34-7
      - for dispinterfaces 34-6
      - for enumerations 34-6
      - for fields 34-7
      - for interfaces 34-6
      - for methods 34-7
      - for modules 34-7
      - for properties 34-7
      - for type libraries 34-6
      - for unions 34-7
    - unions 34-18
    - updating 34-26
    - type reserved word 8-12
    - types
      - Automation 36-15 to 36-16
      - Char 12-3
      - message-record 46-6
      - properties 42-2, 42-8, 47-8
      - type libraries 34-11 to 34-13
      - user-defined 49-3
      - Web modules 29-5
      - Web Services 31-5 to 31-6
- ## U
- UCS standard 10-22
  - UDP protocol 32-1
  - UnaryOp method 4-34
  - Unassociate Attributes
    - command 19-14
  - undocking controls 7-6
  - UndoLastChange method 23-5
  - Unicode characters 4-39, 12-4
    - strings 4-41, 4-43
  - unidirectional cursors 18-48
  - unidirectional datasets 22-1 to 22-18
    - binding 22-5 to 22-7
    - connecting to servers 22-2
    - editing data 22-10
    - executing commands 22-9 to 22-10
    - fetching data 22-8
    - fetching metadata 22-12 to 22-17
    - limitations 22-1
  - preparing 22-8
  - types 22-2
  - UniDirectional property 18-48
  - unindexed datasets 18-19, 18-21
  - unions
    - Type Library editor 34-10, 34-18, 34-24
  - units
    - accessing from other units 3-9
    - adding components 40-11
    - existing
      - adding a component 40-11
      - including packages 11-3
    - units, in conversion 4-59
  - Unlock method 9-7
  - UnlockList method 9-7
  - UnregisterPooled
    - procedure 25-8
  - UnRegisterTypeLib
    - function 33-17
  - update errors
    - resolving 23-20, 23-22 to 23-23, 24-8, 24-11
    - response messages 25-38
  - update objects 20-39 to 20-47, 23-18
    - executing 20-45 to 20-46
    - parameters 20-41 to 20-42, 20-46, 20-46 to 20-47
    - providers and 20-11
    - queries 20-46 to 20-47
    - SQL statements 20-40 to 20-43
      - using multiple 20-43 to 20-46
  - Update SQL editor 20-40 to 20-41
    - Options page 20-41
    - SQL page 20-41
  - UPDATE statements 20-39, 20-43, 24-9
  - UpdateBatch method 10-28, 21-12, 21-14
  - UpdateCalendar method 51-3
  - UpdateMode property 24-10
    - client datasets 23-21
  - UpdateObject method 38-13
  - UpdateObject property 20-11, 20-32, 20-39, 20-44
  - UpdatePropertyPage
    - method 38-13
  - UpdateRecordTypes
    - property 10-28, 20-32, 23-18

- UpdatesPending
  - property 10-28, 20-32
- UpdateStatus property 10-28, 20-32, 21-12, 23-18, 24-9
- UpdateTarget method 6-28
- updating
  - actions 6-26
- up-down controls 3-33
- URIs
  - URLs vs. 27-3
- URL property 25-26, 28-9, 31-9
- URLs 27-3
  - host names 32-4
  - IP addresses 32-4
  - javascript libraries 25-35
  - SOAP connections 25-26
  - URIs vs. 27-3
  - Web browsers 27-4
  - Web connections 25-26
- Use Unit command 5-18, 6-2
- USEPACKAGE macro 11-7
- user interfaces 3-23, 14-15 to 14-16
  - forms 6-1 to 6-2
  - isolating 14-6
  - layout 6-4
  - multi-record 15-14
  - organizing data 15-7, 15-14
  - single record 15-7
- user list service 29-4
- user-defined messages 46-5, 46-7
- user-defined types 49-3
- uses clause 3-9, 10-6
  - adding data modules 5-18
  - avoiding circular references 6-2
  - including packages 11-3
- UTF-8 character set 10-17

## V

---

- \$V compiler directive 4-49
- validating data entry 19-15
- Value property
  - aggregates 23-13
  - fields 19-17
  - parameters 18-45, 18-51
- ValueChecked property 15-13
- values 42-2
  - Boolean 42-2, 42-12, 51-3
  - default data 15-10
  - default property 42-7, 42-11 to 42-12
  - redefining 48-2, 48-3
  - testing 42-6

- Values property
  - radio groups 15-14
- ValueUnchecked property 15-13
- var reserved word
  - event handlers 43-3
- VarCmplx unit 4-35
- variables
  - declaring
    - example 3-10
  - object 3-10
  - objects and 3-10
- Variant type 4-27
- Variants
  - custom 4-27 to 4-39
- VCL 40-1 to 40-2
  - main thread 9-4
  - objects 3-1
  - overview 3-1 to 3-23
  - TComponent branch 3-15
  - TControl branch 3-16
  - TObject branch 3-14
  - TPersistent branch 3-14
  - TWinControl branch 3-17
- VCL applications
  - porting 10-2 to 10-16
- VCL60 package 11-1, 11-9
  - PENWIN.DLL 11-11
- vc160.bpl 13-6
- VendorLib property 22-3
- version control 2-5
- version information
  - ActiveX controls 38-5
  - type information 34-8
- vertical track bars 3-33
- VertScrollBar 3-33
- video cassettes 8-32
- video clips 8-29, 8-30
- viewing scripts 29-10
- VisualStyle property 3-38
- virtual
  - directive 41-8
  - method tables 41-8
  - methods 41-8, 44-4
    - properties as 42-2
    - property editors 47-8 to 47-9
  - visibility 3-9
- Visible property 3-2
  - cool bars 6-49
  - fields 19-12
  - menus 6-41
  - toolbars 6-49
- VisibleButtons property 15-28, 15-29

- VisibleChanged property 10-21
- VisibleColCount property 3-43
- VisibleRowCount property 3-43
- VisiBroker ORB 25-13
- VisualCLX 10-6
- VisualSpeller Control 13-5
- vtables 33-4
  - COM interface pointer 33-4
  - component wrappers 35-6
  - creator classes and 35-5, 35-12
  - dual interfaces 36-13
  - type libraries and 33-16
  - vs dispinterfaces 34-9

## W

---

- W3C 30-2
- WaitFor method 9-9, 9-10
- WantReturns property 3-32
- WantTabs property 3-32
  - data-aware memo
    - controls 15-8
  - data-aware rich edit
    - controls 15-9
- .WAV files 8-32
- wchar\_t widechar 10-22
- \$WEAKPACKAGEUNIT
  - compiler directive 11-10
- Web adapters
  - actions 29-8
  - errors 29-8
  - fields 29-8
  - records 29-8
- Web application debugger 27-7, 28-2, 29-2
- Web application modules
  - interfaces 29-7
- Web applications
  - ActiveX 33-13, 38-1, 38-15 to 38-16
  - multi-tiered clients 25-32
- adapters 29-8
- ASP 33-12, 37-1
- database 25-31 to 25-42
- deployment 13-9
- object 28-3
- Web Broker 5-11, 27-1 to 27-2
- Web Broker server
  - applications 28-1 to 28-20
    - accessing databases 28-17
    - adding to projects 28-3
    - architecture 28-3
    - creating 28-1 to 28-3
    - creating responses 28-7



- event handling 28-5, 28-7, 28-8
- managing database connections 28-17
- overview 28-1 to 28-4
- posting data to 28-10
- querying tables 28-20
- response templates 28-13
- sending files 28-12
- templates 28-2
- Web dispatcher 28-4
- Web browsers 27-4
  - URLs 27-4
- Web connections 25-9 to 25-10, 25-26
- Web data modules 29-3, 29-5
  - interfaces 29-6
  - structure 29-5
- Web deployment 38-15 to 38-16
  - multi-tiered applications 25-33
- Web Deployment Options dialog box 38-16
- Web dispatcher 28-2, 28-4 to 28-5
  - auto-dispatching objects 25-37, 28-5
  - DLL-based applications and 28-3
  - handling requests 28-3, 28-8
  - selecting action items 28-6, 28-7
- Web dispatchers
  - action items 29-17
- Web items 25-39
  - properties 25-40 to 25-41
- Web modules 28-2, 28-4, 29-5 to 29-7
  - adding database sessions 28-17
  - DLLs and, caution 28-3
  - types 29-5
- Web page editor 25-39 to 25-40
- Web page modules 29-3, 29-6
  - interfaces 29-7
- Web pages 27-4
  - InternetExpress page producer 25-38 to 25-42
- Web scripting 29-9
- Web server applications 5-11, 27-1 to 27-9
  - ASP 37-1
  - creating 29-2
  - debugging 27-7 to 27-9
  - multi-tiered 25-33 to 25-42
  - overview 27-6 to 27-9
  - resource locations 27-3
  - standards 27-2
  - types 27-6
- Web servers 25-33, 27-1 to 27-9, 37-6
  - client requests and 27-5
  - debugging 28-2
  - types
    - types of Web servers 29-2
- Web Service Definition Language *See* WSDL
- Web Services 31-1 to 31-10
  - clients 31-8 to 31-10
  - complex types 31-5 to 31-6
  - exceptions 31-7
  - implementation classes 31-6 to 31-7
    - registering 31-6
  - namespaces 31-4
  - servers 31-2 to 31-8
    - writing 31-2 to 31-3
  - wizard 31-3
- Web Services importer 31-9
- Web site (Delphi support) 1-3
- WebContext 29-13
- WebDispatch property 25-37
- WebPageItems property 25-39
- WebSnap 27-1 to 27-2
- WebSnap applications
  - overview 29-1 to 29-26
- WebSnap tutorial 29-18
- wide characters 12-4
  - runtime library routines 4-43
- WideChar 4-39, 4-41
- widechar 10-22
- WideString 4-41 to 4-42
- widestrings 10-22
- WidgetDestroyed property 10-21
- widgets 3-17, 10-6, 10-22
- Width property 3-19, 3-42, 6-4
  - data grid columns 15-16
  - data grids 15-20
  - pens 8-5, 8-6
  - TScreen 13-12
- Win 3.1 page (Component palette) 3-30
- WIN32 10-18
- Win32 page (Component palette) 3-29
- WIN64 10-18
- Win-CGI programs 27-5, 27-7
  - creating 28-2, 29-2
  - INI files 27-7
- window
  - class 40-4
  - controls 40-3
  - handles 40-3, 40-4, 40-5
  - message handling 50-4
  - procedures 46-2, 46-3
- Windows
  - API functions 40-3, 45-1
  - common dialog boxes 52-1
    - creating 52-2
    - executing 52-4
  - controls, subclassing 40-4
  - device contexts 40-7, 45-1
  - events 43-4
  - Graphics Device Interface (GDI) 8-1
  - messages 46-2
  - pen width support 8-6
- windows
  - resizing 3-34
- Windows messaging 10-17
- Windows NT
  - debugging Web server applications 27-9
- Windows socket objects 32-5
  - client sockets 32-6
  - clients 32-6
  - server sockets 32-7
- wininet.dll 25-26, 25-27
- wizards 5-19
  - Active Server Object 33-20, 37-2 to 37-3
  - ActiveForm 33-20, 38-5 to 38-6
  - ActiveX controls 33-20, 38-4 to 38-5
  - ActiveX library 33-21
  - Automation object 33-20, 36-4 to 36-9
  - COM 33-18 to 33-22, 36-1
  - COM object 33-19, 34-19, 36-2 to 36-4, 36-5 to 36-9
  - COM+ Event object 33-20, 39-19
  - Component 40-9
  - Console Wizard 5-3
  - CORBA Data Module 25-15 to 25-16
  - property page 33-20, 38-12
  - Remote Data Module 25-13 to 25-14
  - Resource DLL 12-10
  - SOAP Data Module 25-15
  - Transactional Data Module 25-14 to 25-15

- transactional object 33-20,  
39-15 to 39-18
- Type Library 33-21, 34-19
- Web Services 31-3
- WebSnap applications 29-1  
to 29-26
- XML Data Binding 30-5 to  
30-8
- WM\_APP constant 46-6
- WM\_KEYDOWN message 51-8
- WM\_LBUTTONDOWN  
message 51-8
- WM\_MBUTTONDOWN  
message 51-8
- WM\_PAINT message 46-4
- WM\_PAINT messages 8-2
- WM\_RBUTTONDOWN  
message 51-8
- WM\_SIZE message 50-4
- WndProc method 46-3, 46-4
- word wrapping 7-7
- WordWrap property 3-32, 7-7,  
48-1
  - data-aware memo  
controls 15-9
- wParam parameter 46-2
- Wrap property 6-47
- Wrapable property 6-47
- wrappers 40-4, 52-2
  - See also* component wrappers
  - initializing 52-3
- Write By Reference  
COM interface  
properties 34-9
- Write method
  - TFileStream 4-55

- write method 42-6
- write reserved word 42-8, 49-4
- WriteBuffer method
  - TFileStream 4-56
- write-only properties 42-6
- WSDL 31-2
  - files 31-8
  - importing 31-8 to 31-9
  - publishing 31-7 to 31-8

## X

---

- \$X compiler directive 4-49
- XDR file 30-2
- Xerox Network System  
(XNS) 32-1
- .xfm files 3-7, 10-2
- XML 26-1, 30-1
  - database applications 26-1 to  
26-10
  - document type
    - declaration 30-1
    - mappings 26-2 to 26-3
    - defining 26-4
  - parsers 30-2
  - processing instructions 30-1
  - SOAP 31-1
- XML brokers 25-34, 25-36 to  
25-38
  - HTTP messages 25-37
- XML Data Binding wizard 30-5  
to 30-8
- XML documents 26-1, 30-1 to  
30-8
  - components 30-3, 30-8

- converting to data  
packets 26-1 to 26-8
- generating interfaces  
for 30-6
- nodes 30-2, 30-4 to 30-5
  - attributes 26-5, 30-5
  - children 30-5
  - mapping to fields 26-2
  - properties 30-6
  - transformation files 26-1
  - values 30-4
- publishing database  
information 26-9
  - root node 30-3, 30-6, 30-8
- XML files 21-14
- XML Schema Data file
  - See* XSD file
- XML schemas 30-2
- XML Tree 29-1
- XMLBroker property 25-40
- XMLDataFile property 24-2,  
26-8
- XMLDataSetField  
property 25-40
- XMLMapper 26-2, 26-4 to 26-6
- XSD file 30-2
- XSL Tree 29-1

## Y

---

- Year property 50-5

## Z

---

- Z compiler directive 11-12





# Object Pascal Language Guide

---



Borland®

## Object Pascal

Borland Software Corporation  
100 Enterprise Way, Scotts Valley, CA 95066-3249  
[www.borland.com](http://www.borland.com)

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 2001 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

ALP0000WW21000 1E0R0501

0102030405-9 8 7 6 5 4 3 2 1

D3

# Contents

<b>Chapter 1</b>	
<b>Introduction</b>	<b>1-1</b>
What's in this manual? . . . . .	1-1
Using Object Pascal . . . . .	1-1
Typographical conventions . . . . .	1-2
Other sources of information . . . . .	1-2
Software registration and technical support . . . . .	1-3
<b>Part I</b>	
<b>Basic language description</b>	
<hr/>	
<b>Chapter 2</b>	
<b>Overview</b>	<b>2-1</b>
Program organization . . . . .	2-1
Pascal source files . . . . .	2-1
Other files used to build applications . . . . .	2-2
Compiler-generated files . . . . .	2-3
Example programs . . . . .	2-3
A simple console application . . . . .	2-3
A more complicated example . . . . .	2-4
A native application . . . . .	2-5
<b>Chapter 3</b>	
<b>Programs and units</b>	<b>3-1</b>
Program structure and syntax . . . . .	3-1
The program heading . . . . .	3-2
The program uses clause . . . . .	3-2
The block . . . . .	3-2
Unit structure and syntax . . . . .	3-3
The unit heading . . . . .	3-3
The interface section . . . . .	3-4
The implementation section . . . . .	3-4
The initialization section . . . . .	3-4
The finalization section . . . . .	3-5
Unit references and the uses clause . . . . .	3-5
The syntax of a uses clause . . . . .	3-6
Multiple and indirect unit references . . . . .	3-6
Circular unit references . . . . .	3-7
<b>Chapter 4</b>	
<b>Syntactic elements</b>	<b>4-1</b>
Fundamental syntactic elements . . . . .	4-1
Special symbols . . . . .	4-2
Identifiers . . . . .	4-2
Qualified identifiers . . . . .	4-2
Reserved words . . . . .	4-3
Directives . . . . .	4-3
Numerals . . . . .	4-4
Labels . . . . .	4-4
Character strings . . . . .	4-4
Comments and compiler directives . . . . .	4-5
Expressions . . . . .	4-5
Operators . . . . .	4-6
Arithmetic operators . . . . .	4-6
Boolean operators . . . . .	4-7
Logical (bitwise) operators . . . . .	4-8
String operators . . . . .	4-9
Pointer operators . . . . .	4-9
Set operators . . . . .	4-10
Relational operators . . . . .	4-10
Class operators . . . . .	4-11
The @ operator . . . . .	4-12
Operator precedence rules . . . . .	4-12
Function calls . . . . .	4-13
Set constructors . . . . .	4-13
Indexes . . . . .	4-14
Typecasts . . . . .	4-14
Value typecasts . . . . .	4-14
Variable typecasts . . . . .	4-15
Declarations and statements . . . . .	4-16
Declarations . . . . .	4-16
Statements . . . . .	4-17
Simple statements . . . . .	4-17
Assignment statements . . . . .	4-17
Procedure and function calls . . . . .	4-18
Goto statements . . . . .	4-18
Structured statements . . . . .	4-19
Compound statements . . . . .	4-20
With statements . . . . .	4-20
If statements . . . . .	4-22
Case statements . . . . .	4-23
Control loops . . . . .	4-25
Repeat statements . . . . .	4-25
While statements . . . . .	4-25
For statements . . . . .	4-26
Blocks and scope . . . . .	4-27
Blocks . . . . .	4-27
Scope . . . . .	4-28
Naming conflicts . . . . .	4-29

<b>Chapter 5</b>	
<b>Data types, variables, and constants</b>	<b>5-1</b>
About types . . . . .	5-1
Simple types . . . . .	5-2
Ordinal types . . . . .	5-2
Integer types . . . . .	5-3
Character types . . . . .	5-5
Boolean types . . . . .	5-5
Enumerated types . . . . .	5-6
Subrange types . . . . .	5-8
Real types . . . . .	5-9
String types . . . . .	5-10
Short strings . . . . .	5-12
Long strings . . . . .	5-12
WideString . . . . .	5-13
About extended character sets . . . . .	5-13
Working with null-terminated strings . . . . .	5-13
Using pointers, arrays, and string constants . . . . .	5-14
Mixing Pascal strings and null-terminated strings . . . . .	5-15
Structured types . . . . .	5-16
Sets . . . . .	5-17
Arrays . . . . .	5-18
Static arrays . . . . .	5-18
Dynamic arrays . . . . .	5-19
Array types and assignments . . . . .	5-21
Records . . . . .	5-21
Variant parts in records . . . . .	5-22
File types . . . . .	5-24
Pointers and pointer types . . . . .	5-25
Overview of pointers . . . . .	5-25
Pointer types . . . . .	5-27
Character pointers . . . . .	5-27
Other standard pointer types . . . . .	5-27
Procedural types . . . . .	5-28
Procedural types in statements and expressions . . . . .	5-29
Variant types . . . . .	5-30
Variant type conversions . . . . .	5-31
Variants in expressions . . . . .	5-33
Variant arrays . . . . .	5-33
OleVariant . . . . .	5-34
Type compatibility and identity . . . . .	5-34
Type identity . . . . .	5-34
Type compatibility . . . . .	5-35
Assignment-compatibility . . . . .	5-35
Declaring types . . . . .	5-36
Variables . . . . .	5-37

Declaring variables . . . . .	5-37
Absolute addresses . . . . .	5-38
Dynamic variables . . . . .	5-38
Thread-local variables . . . . .	5-38
Declared constants . . . . .	5-39
True constants . . . . .	5-39
Constant expressions . . . . .	5-41
Resource strings . . . . .	5-41
Typed constants . . . . .	5-42
Array constants . . . . .	5-43
Record constants . . . . .	5-43
Procedural constants . . . . .	5-44
Pointer constants . . . . .	5-44

## Chapter 6 Procedures and functions **6-1**

Declaring procedures and functions . . . . .	6-1
Procedure declarations . . . . .	6-2
Function declarations . . . . .	6-3
Calling conventions . . . . .	6-4
Forward and interface declarations . . . . .	6-5
External declarations . . . . .	6-6
Linking to object files . . . . .	6-6
Importing functions from libraries . . . . .	6-7
Overloading procedures and functions . . . . .	6-8
Local declarations . . . . .	6-10
Nested routines . . . . .	6-10
Parameters . . . . .	6-11
Parameter semantics . . . . .	6-11
Value and variable parameters . . . . .	6-12
Constant parameters . . . . .	6-13
Out parameters . . . . .	6-13
Untyped parameters . . . . .	6-14
String parameters . . . . .	6-15
Array parameters . . . . .	6-15
Open array parameters . . . . .	6-15
Variant open array parameters . . . . .	6-16
Default parameters . . . . .	6-17
Default parameters and overloaded routines . . . . .	6-18
Default parameters in forward and interface declarations . . . . .	6-19
Calling procedures and functions . . . . .	6-19
Open array constructors . . . . .	6-19

## Chapter 7 Classes and objects **7-1**

Class types . . . . .	7-2
Inheritance and scope . . . . .	7-3

TObject and TClass . . . . .	7-3
Compatibility of class types . . . . .	7-3
Object types . . . . .	7-4
Visibility of class members . . . . .	7-4
Private, protected, and public members . . . . .	7-5
Published members. . . . .	7-5
Automated members. . . . .	7-6
Forward declarations and mutually dependent classes. . . . .	7-6
Fields . . . . .	7-7
Methods. . . . .	7-8
Method declarations and implementations . . . . .	7-8
Inherited . . . . .	7-9
Self . . . . .	7-9
Method binding . . . . .	7-10
Static methods. . . . .	7-10
Virtual and dynamic methods. . . . .	7-10
Abstract methods . . . . .	7-12
Overloading methods . . . . .	7-12
Constructors . . . . .	7-13
Destructors . . . . .	7-14
Message methods . . . . .	7-15
Implementing message methods . . . . .	7-16
Message dispatching . . . . .	7-16
Properties . . . . .	7-17
Property access . . . . .	7-17
Array properties . . . . .	7-19
Index specifiers . . . . .	7-20
Storage specifiers . . . . .	7-21
Property overrides and redeclarations. . . . .	7-22
Class references . . . . .	7-23
Class-reference types. . . . .	7-23
Constructors and class references. . . . .	7-24
Class operators . . . . .	7-24
The is operator . . . . .	7-24
The as operator . . . . .	7-25
Class methods . . . . .	7-25
Exceptions . . . . .	7-26
When to use exceptions . . . . .	7-26
Declaring exception types . . . . .	7-27
Raising and handling exceptions . . . . .	7-27
Try...except statements . . . . .	7-28
Re-raising exceptions. . . . .	7-30
Nested exceptions . . . . .	7-31
Try...finally statements . . . . .	7-31
Standard exception classes and routines. . . . .	7-32

<b>Chapter 8</b>	
<b>Standard routines and I/O</b>	<b>8-1</b>
File input and output. . . . .	8-1
Text files . . . . .	8-3
Untyped files. . . . .	8-4
Text-file device drivers . . . . .	8-4
Device functions. . . . .	8-5
The Open function. . . . .	8-5
The InOut function . . . . .	8-5
The Flush function. . . . .	8-6
The Close function. . . . .	8-6
Handling null-terminated strings . . . . .	8-6
Wide-character strings . . . . .	8-7
Other standard routines . . . . .	8-7

## Part II

### Special topics

---

<b>Chapter 9</b>	
<b>Libraries and packages</b>	<b>9-1</b>
Calling dynamically loadable libraries . . . . .	9-1
Static loading . . . . .	9-1
Dynamic loading. . . . .	9-2
Writing dynamically loadable libraries. . . . .	9-3
The exports clause. . . . .	9-5
Library initialization code . . . . .	9-5
Global variables in a library . . . . .	9-6
Libraries and system variables. . . . .	9-6
Exceptions and runtime errors in libraries . . . . .	9-7
Shared-memory manager (Windows only). . . . .	9-8
Packages . . . . .	9-8
Package declarations and source files. . . . .	9-9
Naming packages . . . . .	9-10
The requires clause . . . . .	9-10
The contains clause . . . . .	9-10
Compiling packages . . . . .	9-11
Generated files . . . . .	9-11
Package-specific compiler directives . . . . .	9-11
Package-specific command-line compiler switches . . . . .	9-12

<b>Chapter 10</b>	
<b>Object interfaces</b>	<b>10-1</b>
Interface types. . . . .	10-1
Interface and inheritance . . . . .	10-2



Interface identification . . . . .	10-3
Calling conventions for interfaces . . . . .	10-3
Interface properties. . . . .	10-4
Forward declarations . . . . .	10-4
Implementing interfaces . . . . .	10-4
Method resolution clauses. . . . .	10-5
Changing inherited implementations . . . . .	10-6
Implementing interfaces by delegation . . . . .	10-6
Delegating to an interface-type property. . . . .	10-7
Delegating to a class-type property. . . . .	10-7
Interface references . . . . .	10-8
Interface assignment-compatibility. . . . .	10-9
Interface typecasts . . . . .	10-10
Interface querying . . . . .	10-10
Automation objects (Windows only) . . . . .	10-10
Dispatch interface types (Windows only) . . . . .	10-10
Dispatch interface methods (Windows only) . . . . .	10-11
Dispatch interface properties . . . . .	10-11
Accessing Automation objects (Windows only) . . . . .	10-11
Automation object method-call syntax . . . . .	10-12
Dual interfaces (Windows only) . . . . .	10-13
<b>Chapter 11</b>	
<b>Memory management</b>	<b>11-1</b>
The memory manager (Windows only) . . . . .	11-1
Variables. . . . .	11-2
Internal data formats . . . . .	11-2
Integer types . . . . .	11-3
Character types . . . . .	11-3
Boolean types . . . . .	11-3
Enumerated types . . . . .	11-3
Real types . . . . .	11-4
The Real48 type . . . . .	11-4
The Single type . . . . .	11-4
The Double type . . . . .	11-5
The Extended type . . . . .	11-5
The Comp type . . . . .	11-5
The Currency type . . . . .	11-5
Pointer types . . . . .	11-5
Short string types. . . . .	11-5
Long string types . . . . .	11-6
Wide string types . . . . .	11-6
Set types . . . . .	11-7
Static array types . . . . .	11-7

Dynamic array types . . . . .	11-7
Record types . . . . .	11-8
File types . . . . .	11-8
Procedural types. . . . .	11-10
Class types . . . . .	11-10
Class reference types . . . . .	11-11
Variant types . . . . .	11-11

**Chapter 12**  
**Program control** **12-1**

Parameters and function results. . . . .	12-1
Parameter passing. . . . .	12-1
Register saving conventions . . . . .	12-3
Function results . . . . .	12-3
Method calls . . . . .	12-3
Constructors and destructors. . . . .	12-4
Exit procedures . . . . .	12-4

**Chapter 13**  
**Inline assembler code** **13-1**

The asm statement . . . . .	13-1
Register use . . . . .	13-2
Assembler statement syntax. . . . .	13-2
Labels . . . . .	13-2
Instruction opcodes . . . . .	13-2
RET instruction sizing. . . . .	13-3
Automatic jump sizing . . . . .	13-3
Assembler directives . . . . .	13-3
Operands. . . . .	13-7
Expressions . . . . .	13-8
Differences between Object Pascal and assembler expressions. . . . .	13-8
Expression elements . . . . .	13-9
Constants . . . . .	13-9
Registers . . . . .	13-10
Symbols. . . . .	13-11
Expression classes . . . . .	13-12
Expression types. . . . .	13-14
Expression operators . . . . .	13-15
Assembler procedures and functions. . . . .	13-17

**Appendix A**  
**Object Pascal grammar** **A-1**

**Index** **I-1**

# Tables

4.1	Reserved words . . . . .	4-3	8.2	Null-terminated string functions . . . . .	8-6
4.2	Directives . . . . .	4-3	8.3	Other standard routines . . . . .	8-7
4.3	Binary arithmetic operators . . . . .	4-6	9.1	Compiled package files . . . . .	9-11
4.4	Unary arithmetic operators . . . . .	4-7	9.2	Package-specific compiler directives . . . . .	9-11
4.5	Boolean operators . . . . .	4-7	9.3	Package-specific command-line compiler switches . . . . .	9-12
4.6	Logical (bitwise) operators . . . . .	4-8	11.1	Long string dynamic memory layout . . . . .	11-6
4.7	String operators . . . . .	4-9	11.2	Wide string dynamic memory layout (Windows only) . . . . .	11-6
4.8	Character-pointer operators . . . . .	4-9	11.3	Dynamic array memory layout . . . . .	11-7
4.9	Set operators . . . . .	4-10	11.4	Type alignment masks . . . . .	11-8
4.10	Relational operators . . . . .	4-11	11.5	Virtual method table layout . . . . .	11-10
4.11	Precedence of operators . . . . .	4-12	13.1	Built-in assembler reserved words . . . . .	13-7
5.1	Generic integer types for 32-bit implementations of Object Pascal . . . . .	5-3	13.2	String examples and their values . . . . .	13-10
5.2	Fundamental integer types . . . . .	5-4	13.3	CPU registers . . . . .	13-10
5.3	Fundamental real types . . . . .	5-9	13.4	Symbols recognized by the built-in assembler assembler . . . . .	13-11
5.4	Generic real types . . . . .	5-10	13.5	Predefined type symbols . . . . .	13-14
5.5	String types . . . . .	5-10	13.6	Precedence of built-in assembler expression operators . . . . .	13-15
5.6	Selected pointer types declared in System and SysUtils . . . . .	5-27	13.7	Definitions of built-in assembler expression operators . . . . .	13-15
5.7	Variant type conversion rules . . . . .	5-32			
5.8	Types for integer constants . . . . .	5-40			
6.1	Calling conventions . . . . .	6-5			
8.1	Input and output procedures/functions . . . . .	8-1			



# Introduction

This manual describes the Object Pascal programming language as it is used in Borland development tools.

## What's in this manual?

---

The first seven chapters describe most of the language elements used in ordinary programming. Chapter 8 summarizes standard routines for file I/O and string manipulation.

The next chapters describe language extensions and restrictions for dynamic-link libraries and packages (Chapter 9), and for object interfaces (Chapter 10). The final three chapters address advanced topics: memory management (Chapter 11), program control (Chapter 12), and assembly-language routines within Object Pascal programs (Chapter 13).

## Using Object Pascal

---

The *Object Pascal Language Guide* is written to describe the Object Pascal language for use on either the Linux or Windows operating systems. Differences in the language relating to platform dependencies are noted where necessary.

Most Delphi/Kylix application developers write and compile their Object Pascal code in the integrated development environment (IDE). Working in the IDE allows the product to handle many details of setting up projects and source files, such as maintenance of dependency information among units. Borland products may enforce certain constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, certain file- and program-naming conventions can be avoided if you write your programs outside of the IDE and compile them from the command prompt.

This manual generally assumes that you are working in the IDE and that you are building applications that use the Visual Component Library (VCL) and/or the Borland Component Library for Cross Platform (CLX). Occasionally, however, Borland-specific rules are distinguished from rules that apply to all Object Pascal programming.

## Typographical conventions

---

Identifiers—that is, names of constants, variables, types, fields, properties, procedures, functions, programs, units, libraries, and packages—appear in *italics* in the text. Object Pascal operators, reserved words, and directives are in **boldface** type. Example code and text that you would type literally (into a file or at the command prompt) are in `monospaced` type.

In displayed program listings, reserved words and directives appear in **boldface**, just as they do in the text:

```
function Calculate(X, Y: Integer): Integer;  
begin  
  :  
end;
```

This is how the Code editor displays reserved words and directives, if you have the Syntax Highlight option turned on.

Some program listings, like the example above, contain ellipsis marks (... or :). The ellipses represent additional code that would be included in an actual file. They are not meant to be copied literally.

In syntax descriptions, *italics* indicate placeholders for which, in real code, you would substitute syntactically valid constructions. For example, the heading of the function declaration above could be represented as

```
function functionName(argumentList): returnType;
```

Syntax descriptions can also contain ellipsis marks (...) and subscripts:

```
function functionName(arg1, ..., argn): ReturnType;
```

## Other sources of information

---

The online Help system for your development tool provides information about the IDE and user interface as well as the most up-to-date reference material for the VCL and/or CLX. Many programming topics, such as database development, are covered in depth in the *Developer's Guide*. For an overview of the documentation set, see the *Quick Start* manual that came with your software package.

## **Software registration and technical support**

---

Borland Software Corporation offers a range of support plans to fit the needs of individual developers, consultants, and corporations. To receive help with this product, return the registration card and select the plan that best suits your needs. For additional information about technical support and other Borland services, contact your local sales representative or visit us online at <http://www.borland.com/>.



## Basic language description

The chapters in Part I present the essential language elements required for most programming tasks. These chapters include:

- Chapter 2, “Overview”
- Chapter 3, “Programs and units”
- Chapter 4, “Syntactic elements”
- Chapter 5, “Data types, variables, and constants”
- Chapter 6, “Procedures and functions”
- Chapter 7, “Classes and objects”
- Chapter 8, “Standard routines and I/O”





## Overview

Object Pascal is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming.

Object Pascal has special features that support Borland's component framework and RAD environment. For the most part, descriptions and examples in this manual assume that you are using Object Pascal to develop applications using Borland development tools such as Delphi or Kylix.

### Program organization

---

Programs are usually divided into source-code modules called *units*. Each program begins with a heading, which specifies a name for the program. The heading is followed by an optional **uses** clause, then a block of declarations and statements. The **uses** clause lists units that are linked into the program; these units, which can be shared by different programs, often have **uses** clauses of their own.

The **uses** clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, Object Pascal programs do not require makefiles, header files, or preprocessor "include" directives. (The Project Manager generates a makefile each time a project is loaded in the IDE, but saves these files only for project groups that include more than one project.)

For further discussion of program structure and dependencies, see Chapter 3, "Programs and units".

### Pascal source files

---

The compiler expects to find Pascal source code in files of three kinds:

- *unit* source files (which end with the .pas extension)

- *project* files (which end with the `.dpr` extension)
- *package* source files (which end with the `.dpk` extension)

Unit source files contain most of the code in an application. Each application has a single project file and several unit files; the project file—which corresponds to the “main” program file in traditional Pascal—organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (`.pas`) files. But if you use the IDE to build your application, you must have a project (`.dpr`) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called *packages*. For more information about packages, see Chapter 9, “Libraries and packages”.

## Other files used to build applications

---

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically and include

- *form* files, which end with the `.dfm` (Delphi) or `.xfm` (Kylix) extension,
- *resource* files, which end with the `.res` extension, and
- *project options* files, which end with the `.dof` (Delphi) or `.kof` (Kylix) extension.

A form file is either a text file or a compiled resource file that can contain bitmaps, strings, and so forth. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland’s visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (`.pas`) file that, by default, has the same name as the form file.

In addition to form files, each project uses a resource (`.res`) file to hold the bitmap for the application’s icon. By default, this file has the same name as the project (`.dpr`) file. To change an application’s icon, use the Project Options dialog.

A project options (`.dof` or `.kof`) file contains compiler and linker settings, search directories, version information, and so forth. Each project has an associated project options file with the same name as the project (`.dpr`) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (`.dsk` or `.desk`) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

## Compiler-generated files

---

The first time you build an application or a standard dynamic-link library, the compiler produces a compiled unit .dcu (Windows) .dcu/.dpu (Linux) file for each new unit used in your project; all the .dcu (Windows) .dcu/.dpu (Linux) files in your project are then linked to create a single executable or shared library file. The first time you build a package, the compiler produces a .dcu (Windows) .dpu (Linux) file for each new unit contained in the package, and then creates both a .dcp and a package file. (For more information about libraries and packages, see Chapter 9.) If you use the `-GD` switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you rebuild a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, or their .dcu (Windows) .dcu/.dpu (Linux) files cannot be found, or you explicitly tell the compiler to reprocess them. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file.

## Example programs

---

The examples that follow illustrate basic features of Object Pascal programming. The examples show simple Object Pascal applications that cannot be compiled from the IDE; but you can compile them from the command line.

### A simple console application

---

The program below is a simple console application that you can compile and run from the command prompt.

```

program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
    MyMessage := 'Hello world!';
    Writeln(MyMessage);
end.

```

The first line declares a program called *Greeting*. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called *MyMessage*, which holds a string. (Object Pascal has genuine string data types.) The program then assigns the string "Hello world!" to the variable *MyMessage*, and sends the contents of *MyMessage* to the standard output using the *Writeln* procedure. (*Writeln* is defined implicitly in the *System* unit, which the compiler automatically includes in every application.)

You can type this program into a file called `Greeting.pas` or `Greeting.dpr` and compile it by entering

On Delphi: `DCC32 Greeting`

On Kylix: `dcc Greeting`

on the command line. The resulting executable prints the message “Hello world!”

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Borland development tools. First, it is a console application. Borland development tools are typically used to write applications with graphical interfaces; hence, you would not ordinarily call *Writeln*. Moreover, the entire example program (save for *Writeln*) is in a single file. In a typical application, the program heading—the first line of the example—would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to *methods* defined in unit files.

## A more complicated example

---

The next example shows a program that is divided into two files: a *project* file and a *unit* file. The project file, which you can save as `Greeting.dpr`, looks like this:

```
program Greeting;  
  
{$APPTYPE CONSOLE}  
  
uses Unit1;  
  
begin  
    PrintMessage('Hello World!');  
end.
```

The first line declares a program called *Greeting*, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that *Greeting* includes a unit called *Unit1*. Finally, the program calls the *PrintMessage* procedure, passing to it the string “Hello World!” Where does the *PrintMessage* procedure come from? It’s defined in *Unit1*. Here’s the source code for *Unit1*, which you can save in a file called `Unit1.pas`:

```
unit Unit1;  
  
interface  
  
procedure PrintMessage(msg: string);  
  
implementation  
  
procedure PrintMessage(msg: string);  
begin  
    Writeln(msg);  
end;  
  
end.
```

*Unit1* defines a procedure called *PrintMessage* that takes a single string as an argument and sends the string to the standard output. (In Pascal, routines that do not return a value are called *procedures*. Routines that return a value are called *functions*.) Notice that *PrintMessage* is declared twice in *Unit1*. The first declaration, under the reserved word **interface**, makes *PrintMessage* available to other modules (such as *Greeting*) that use *Unit1*. The second declaration, under the reserved word **implementation**, actually defines *PrintMessage*.

You can now compile *Greeting* from the command line by entering

```
On Delphi: DCC32 Greeting
```

```
On Kylix: dcc Greeting
```

There's no need to include *Unit1* as a command-line argument. When the compiler processes *Greeting.dpr*, it automatically looks for unit files that the *Greeting* program depends on. The resulting executable does the same thing as our first example: it prints the message "Hello world!"

## A native application

---

Our next example is an application built using VCL or CLX components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of Object Pascal. In addition to multiple units, the program uses classes and objects, which are discussed in Chapter 7, "Classes and objects".

The program includes a project file and two new unit files. First, the project file:

```
program Greeting; { comments are enclosed in braces }

uses
  Forms, {change the unit name to QForms on Linux}
  Unit1 in 'Unit1.pas' { the unit for Form1 },
  Unit2 in 'Unit2.pas' { the unit for Form2 };

{$R *.res} { this directive links the project's resource file }

begin
  { calls to Application }
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Once again, our program is called *Greeting*. It uses three units: *Forms*, which is part of VCL and CLX; *Unit1*, which is associated with the application's main form (*Form1*); and *Unit2*, which is associated with another form (*Form2*).

The program makes a series of calls to an object named *Application*, which is an instance of the *TApplication* class defined in the *Forms* unit. (Every project has an automatically generated *Application* object.) Two of these calls invoke a *TApplication*

method named *CreateForm*. The first call to *CreateForm* creates *Form1*, an instance of the *TForm1* class defined in *Unit1*. The second call to *CreateForm* creates *Form2*, an instance of the *TForm2* class defined in *Unit2*.

*Unit1* looks like this:

```

unit Unit1;

interface

uses { these units are part of the Visual Component Library (VCL) }
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

{
On Linux, the uses clause looks like this:
uses { these units are part of CLX }
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    end;

var
    Form1: TForm1;

implementation

uses Unit2; { this is where Form2 is defined }

{$R *.dfm} { this directive links Unit1's form file }

procedure TForm1.Button1Click(Sender: TObject);
begin
    Form2.ShowModal;
end;

end.

```

*Unit1* creates a class named *TForm1* (derived from *TForm*) and an instance of this class, *Form1*. *TForm1* includes a button—*Button1*, an instance of *TButton*—and a procedure named *TForm1.Button1Click* that is called at runtime whenever the user presses *Button1*. *TForm1.Button1Click* hides *Form1* and it displays *Form2* (the call to *Form2.ShowModal*). *Form2* is defined in *Unit2*:

```

unit Unit2;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;

```

```

{
On Linux, the uses clause looks like this:
uses { these units are part of CLX }
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
    TForm2 = class(TForm)
        Label1: TLabel;
        CancelButton: TButton;
        procedure CancelButtonClick(Sender: TObject);
        procedure FormClose(Sender: TObject; var Action: TCloseAction);
    end;

var
    Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
    Form2.Close;
end;

end.

```

*Unit2* creates a class named *TForm2* and an instance of this class, *Form2*. *TForm2* includes a button (*CancelButton*, an instance of *TButton*) and a label (*Label1*, an instance of *TLabel*). You can't see this from the source code, but *Label1* displays a caption that reads "Hello world!" The caption is defined in *Form2*'s form file, *Unit2.dfm*.

*Unit2* defines one procedure. *TForm2.CancelButtonClick* is called at runtime whenever the user presses *CancelButton*; it closes *Form2*. This procedure (along with *Unit1*'s *TForm1.Button1Click*) is known as an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form (.dfm on Windows .xfrm on Linux) files for *Form1* and *Form2*.

When the *Greeting* program starts, *Form1* is displayed and *Form2* is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's *main form*.) When the user presses the button on *Form1*, *Form1* disappears and is replaced by *Form2*, which displays the "Hello world!" greeting. When the user closes *Form2* (by pressing *CancelButton* or the Close button on the title bar), *Form1* reappears.





## Programs and units

A program is constructed from source-code modules called *units*. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. Units allow you to

- divide large programs into modules that can be edited separately.
- create libraries that you can share among programs.
- distribute libraries to other developers without making the source code available.

In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use a *project* (.dpr) file to store the “main” program, while most other source code resides in *unit* (.pas) files. Each application—or *project*—consists of a single project file and one or more unit files. (Strictly speaking, you needn’t explicitly use any units in a project, but all programs automatically use the *System* unit.) To build a project, the compiler needs either a source file or a compiled unit file for each unit.

### Program structure and syntax

---

A program contains

- a program heading,
- a **uses** clause (optional), and
- a block of declarations and statements.

The program heading specifies a name for the program. The **uses** clause lists units used by the program. The block contains declarations and statements that are executed when the program runs. The IDE expects to find these three elements in a single project (.dpr) file.

The example below shows the project file for a program called Editor.

```

1  program Editor;
2
3  uses
4    Forms, {change to QForms in Linux}
5    REAbout in 'REAbout.pas' {AboutBox},
6    REMain in 'REMain.pas' {MainForm};
7
8  {$R *.res}
9
10 begin
11   Application.Title := 'Text Editor';
12   Application.CreateForm(TMainForm, MainForm);
13   Application.Run;
14 end.
```

Line 1 contains the program heading. The **uses** clause is on lines 3 through 6. Line 8 is a compiler directive that links the project's resource file into the program. Lines 10 through 14 contain the block of statements that are executed when the program runs. Finally, the project file, like all source files, ends with a period.

This is, in fact, a fairly typical project file. Project files are usually short, since most of a program's logic resides in its unit files. Project files are generated and maintained automatically, and it is seldom necessary to edit them manually.

## The program heading

---

The program heading specifies the program's name. It consists of the reserved word **program**, followed by a valid identifier, followed by a semicolon. The identifier must match the project file name. In the example above, since the program is called Editor, the project file should be called EDITOR.dpr.

In standard Pascal, a program heading can include parameters after the program name:

```
program Calc(input, output);
```

Borland's Object Pascal compiler ignores these parameters.

## The program uses clause

---

The **uses** clause lists units that are incorporated into the program. These units may in turn have **uses** clauses of their own. For more information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

## The block

---

The block contains a simple or structured statement that is executed when the program runs. In most programs, the block consists of a compound statement—bracketed between the reserved words **begin** and **end**—whose component

statements are simply method calls to the project's *Application* object. (Every project has an *Application* variable that holds an instance of *TApplication*, *TWebApplication*, or *TServiceApplication*.) The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

## Unit structure and syntax

---

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own unit (.pas) file.

A unit file begins with a unit heading, which is followed by the *interface*, *implementation*, *initialization*, and *finalization* sections. The initialization and finalization sections are optional. A skeleton unit file looks like this:

```
unit Unit1;

interface

uses { List of units goes here }

    { Interface section goes here }

implementation

uses { List of units goes here }

    { Implementation section goes here }

initialization
    { Initialization section goes here }

finalization
    { Finalization section goes here }

end.
```

The unit must conclude with the word **end** followed by a period.

### The unit heading

---

The unit heading specifies the unit's name. It consists of the reserved word **unit**, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the unit file name. Thus, the unit heading

```
unit MainForm;
```

would occur in a source file called MAINFORM.pas, and the file containing the compiled unit would be MAINFORM.dcu.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

## The interface section

---

The interface section of a unit begins with the reserved word **interface** and continues until the beginning of the implementation section. The interface section declares constants, types, variables, procedures, and functions that are available to *clients*—that is, to other units or programs that use the unit where they are declared. These entities are called *public* because a client can access them as if they were declared in the client itself.

The interface declaration of a procedure or function includes only the routine's heading. The block of the procedure or function follows in the implementation section. Thus procedure and function declarations in the interface section work like forward declarations, although the **forward** directive isn't used.

The interface declaration for a class must include declarations for all class members.

The interface section can include its own **uses** clause, which must appear immediately after the word **interface**. For information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

## The implementation section

---

The implementation section of a unit begins with the reserved word **implementation** and continues until the beginning of the initialization section or, if there is no initialization section, until the end of the unit. The implementation section defines procedures and functions that are declared in the interface section. Within the implementation section, these procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the implementation section; but if you include a parameter list, it must match the declaration in the interface section exactly.

In addition to definitions of public procedures and functions, the implementation section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit—that is, inaccessible to clients.

The implementation section can include its own **uses** clause, which must appear immediately after the word **implementation**. For information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

## The initialization section

---

The initialization section is optional. It begins with the reserved word **initialization** and continues until the beginning of the finalization section or, if there is no finalization section, until the end of the unit. The initialization section contains statements that are executed, in the order in which they appear, on program start-up.

So, for example, if you have defined data structures that need to be initialized, you can do this in the initialization section.

The initialization sections of units used by a client are executed in the order in which the units appear in the client's **uses** clause.

## The finalization section

---

The finalization section is optional and can appear only in units that have an initialization section. The finalization section begins with the reserved word **finalization** and continues until the end of the unit. It contains statements that are executed when the main program terminates. Use the finalization section to free resources that are allocated in the initialization section.

Finalization sections are executed in the opposite order from initializations. For example, if your application initializes units *A*, *B*, and *C*, in that order, it will finalize them in the order *C*, *B*, and *A*.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the initialization code might not execute completely.

## Unit references and the uses clause

---

A **uses** clause lists units used by the program, library, or unit in which the clause appears. (For information about libraries, see Chapter 9, "Libraries and packages".) A **uses** clause can occur in

- the project file for a program or library,
- the interface section of a unit, and
- the implementation section of a unit.

Most project files contain a **uses** clause, as do the interface sections of most units. The implementation section of a unit can contain its own **uses** clause as well.

The *System* unit is used automatically by every application and cannot be listed explicitly in the **uses** clause. (*System* implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as *SysUtils*, must be included in the **uses** clause. In most cases, all necessary units are placed in the **uses** clause when your project generates and maintains a source file.

For more information about the placement and content of the **uses** clause, see "Multiple and indirect unit references" on page 3-6 and "Circular unit references" on page 3-7.

## The syntax of a uses clause

---

A **uses** clause consists of the reserved word **uses**, followed by one or more comma-delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;

uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

In the **uses** clause of a program or library, any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;

uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

Include **in ...** after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no reason to do this. Using **in** is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path or the general Library search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the **in ...** construction to determine which units are part of a project. Only units that appear in a project (.dpr) file's **uses** clause followed by **in** and a file name are considered to be part of the project; other units in the **uses** clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the Project Manager and Project Browser.

In the **uses** clause of a unit, you cannot use **in** to tell the compiler where to find a source file. Every unit must be in the compiler's search path, the general Library search path, or the same directory as the unit that uses it. Moreover, unit names must match the names of their source files.

## Multiple and indirect unit references

---

The order in which units appear in the **uses** clause determines the order of their initialization (see "The initialization section" on page 3-4) and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the same name, the compiler uses the one from the unit listed last in the **uses** clause. (To access the identifier from the other unit, you would have to add a qualifier: *UnitName.Identifier*.)

A **uses** clause need include only units used directly by the program or unit in which the clause appears. That is, if unit *A* references constants, types, variables, procedures, or functions that are declared in unit *B*, then *A* must use *B* explicitly. If *B* in turn references identifiers from unit *C*, then *A* is indirectly dependent on *C*; in this case, *C* needn't be included in a **uses** clause in *A*, but the compiler must still be able to find both *B* and *C* in order to process *A*.

The example below illustrates indirect dependency.

```

program Prog;
uses Unit2;
const a = b;
:

unit Unit2;
interface
uses Unit1;
const b = c;
:

unit Unit1;
interface
const c = 1;
:

```

In this example, *Prog* depends directly on *Unit2*, which depends directly on *Unit1*. Hence *Prog* is indirectly dependent on *Unit1*. Because *Unit1* does not appear in *Prog*'s **uses** clause, identifiers declared in *Unit1* are not available to *Prog*.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their .dpu (Windows) or .dpu/.dpu (Linux) files, not their source (.pas) files.

When changes are made in the interface section of a unit, other units that depend on it must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

## Circular unit references

---

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the **uses** clause of one interface section to the **uses** clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the **uses** clause of at least one implementation section.



In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface **uses** clauses. So the following example leads to a compilation error:

```
unit Unit1;  
interface  
uses Unit2;  
:  
  
unit Unit2;  
interface  
uses Unit1;  
:  
:
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;  
interface  
uses Unit2;  
:  
  
unit Unit2;  
interface  
:  
implementation  
uses Unit1;  
:  
:
```

To reduce the chance of circular references, it's a good idea to list units in the implementation **uses** clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface **uses** clause.

## Syntactic elements

Object Pascal uses the ASCII character set, including the letters *A* through *Z* and *a* through *z*, the digits *0* through *9*, and other standard characters. It is *not* case-sensitive. The space character (ASCII 32) and the control characters (ASCII 0 through 31—including ASCII 13, the return or end-of-line character) are called *blanks*.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

### Fundamental syntactic elements

---

On the simplest level, a program is a sequence of tokens delimited by separators. A *token* is the smallest meaningful unit of text in a program. A *separator* is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;  
Price := 10;
```

Tokens are categorized as *special symbols*, *identifiers*, *reserved words*, *directives*, *numerals*, *labels*, and *character strings*. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

## Special symbols

---

Special symbols are nonalphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols.

# \$ & ' ( ) \* + , - . / : ; < = > @ [ ] ^ { }

The following character pairs are also special symbols.

(\* (. \*) .) .. // := <= >= <>

The left bracket—[—is equivalent to the character pair of left parenthesis and period—(.; the right bracket—]—is equivalent to the character pair of period and right parenthesis—.). The left-parenthesis-plus-asterisk and asterisk-plus-right-parenthesis—(\* \*)—are equivalent to the left and right brace—{ }.

Notice that !, " (double quotation marks), %, ?, \, \_ (underscore), | (pipe), and ~ (tilde) are not special characters.

## Identifiers

---

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with a letter or an underscore (\_) and cannot contain spaces; letters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Since Object Pascal is case-insensitive, an identifier like *CalculateValue* could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

On Linux, the only identifiers for which case is important are unit names. Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation.

### Qualified identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to *qualify* the identifier. The syntax for a qualified identifier is

*identifier*<sub>1</sub>.*identifier*<sub>2</sub>

where *identifier*<sub>1</sub> qualifies *identifier*<sub>2</sub>. For example, if two units each declare a variable called *CurrentValue*, you can specify that you want to access the *CurrentValue* in *Unit2* by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the *Click* method in *Button1* of *Form1*.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in "Blocks and scope" on page 4-27.

## Reserved words

---

The following reserved words cannot be redefined or used as identifiers.

**Table 4.1** Reserved words

and	downto	in	or	string
array	else	inherited	out	then
as	end	initialization	packed	threadvar
asm	except	inline	procedure	to
begin	exports	interface	program	try
case	file	is	property	type
class	finalization	label	raise	unit
const	finally	library	record	until
constructor	for	mod	repeat	uses
destructor	function	nil	resourcestring	var
dispinterface	goto	not	set	while
div	if	object	shl	with
do	implementation	of	shr	xor

In addition to the words in Table 4.1, **private**, **protected**, **public**, **published**, and **automated** act as reserved words within object type declarations, but are otherwise treated as directives. The words **at** and **on** also have special meanings.

## Directives

---

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in Object Pascal, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence—although it is inadvisable to do so—you can define an identifier that looks exactly like a directive.

**Table 4.2** Directives

absolute	dynamic	message	private	resident
abstract	export	name	protected	safecall
assembler	external	near	public	stdcall
automated	far	nodefault	published	stored
cdecl	forward	overload	read	varargs
contains	implements	override	readonly	virtual
default	index	package	register	write
deprecated	library	pascal	reintroduce	writeonly
dispid	local	platform	requires	

## Numerals

---

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the + or – operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to +67258) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character *E* or *e* occurs within a real, it means “times ten to the power of”. For example, 7E-2 means  $7 \times 10^{-2}$ , and 12.25e+6 and 12.25e6 both mean  $12.25 \times 10^6$ .

The dollar-sign prefix indicates a hexadecimal numeral—for example, \$8F. For the Integer type (16-bit integer), the sign of a hexadecimal is determined by the leftmost (most significant) bit of its binary representation. For all other types, you must use a prefixed + or - operator to indicate sign.

For more information about real and integer types, see Chapter 5, “Data types, variables, and constants”. For information about the data types of numerals, see “True constants” on page 5-39.

## Labels

---

A label is a sequence of no more than four digits—that is, a numeral between 0 and 9999. Leading zeros are not significant. Identifiers can also function as labels.

Labels are used in **goto** statements. For more information about **goto** statements and labels, see “Goto statements” on page 4-18.

## Character strings

---

A character string, also called a *string literal* or *string constant*, consists of a *quoted string*, a *control string*, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND'      { BORLAND }
'You''ll see'  { You'll see }
''             { ' }
''            { null string }
' '           { a space }
```

A control string is a sequence of one or more *control characters*, each of which consists of the # symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'\#13#10'Line 2'
```

to put a carriage-return–line-feed between “Line 1” and “Line 2”. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the + operator described in “String operators” on page 4-9, or simply combine them into a single quoted string.)

A character string’s *length* is the number of characters in the string. A character string of any length is compatible with any string type and with the *PChar* type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (`{SX+}`), a character string of length  $n \geq 1$  is compatible with zero-based arrays and packed arrays of  $n$  characters. For more information about string types, see Chapter 5, “Data types, variables, and constants”.

## Comments and compiler directives

---

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }
(* Text between a left-parenthesis-plus-asterisk and an
   asterisk-plus-right-parenthesis also constitutes a comment. *)
// Any text between a double-slash and the end of the line constitutes a comment.
```

A comment that contains a dollar sign (\$) immediately after the opening { or (\* is a compiler directive. For example,

```
{$WARNINGS OFF}
```

tells the compiler not to generate warning messages.

## Expressions

---

An *expression* is a construction that returns a value. For example,

X	{ variable }
@X	{ address of a variable }
15	{ integer constant }
InterestRate	{ variable }
Calc(X,Y)	{ function call }
X * Y	{ product of X and Y }
Z / (1 - Z)	{ quotient of Z and (1 - Z) }
X = 1.5	{ Boolean }
C in Range1	{ Boolean }
not Done	{ negation of a Boolean }

```

['a','b','c']      { set }
Char(48)           { value typecast }

```

The simplest expressions are variables and constants (described in Chapter 5, “Data types, variables, and constants”). More complex expressions are built from simpler ones using *operators*, *function calls*, *set constructors*, *indexes*, and *typecasts*.

## Operators

Operators behave like predefined functions that are part of the Object Pascal language. For example, the expression  $(X + Y)$  is built from the variables  $X$  and  $Y$ —called *operands*—with the  $+$  operator; when  $X$  and  $Y$  represent integers or reals,  $(X + Y)$  returns their sum. Operators include **@**, **not**, **^**, **\***, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **-**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in**, and **is**.

The operators **@**, **not**, and **^** are *unary* (taking one operand). All other operators are *binary* (taking two operands), except that  $+$  and  $-$  can function as either unary or binary. A unary operator always precedes its operand (for example,  $-B$ ), except for **^**, which follows its operand (for example,  $P^A$ ). A binary operator is placed between its operands (for example,  $A = 7$ ).

Some operators behave differently depending on the type of data passed to them. For example, **not** performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories.

Except for **^**, **is**, and **in**, all operators can take operands of type *Variant*. For details, see “Variant types” on page 5-30.

The sections that follow assume some familiarity with Object Pascal data types. For information about data types, see Chapter 5, “Data types, variables, and constants”.

For information about operator precedence in complex expressions, see “Operator precedence rules” on page 4-12.

### Arithmetic operators

Arithmetic operators, which take real or integer operands, include  $+$ ,  $-$ ,  $*$ ,  $/$ , **div**, and **mod**.

**Table 4.3** Binary arithmetic operators

Operator	Operation	Operand types	Result type	Example
$+$	addition	integer, real	integer, real	$X + Y$
$-$	subtraction	integer, real	integer, real	Result - 1
$*$	multiplication	integer, real	integer, real	$P * \text{InterestRate}$
$/$	real division	integer, real	real	$X / 2$
<b>div</b>	integer division	integer	integer	Total <b>div</b> UnitSize
<b>mod</b>	remainder	integer	integer	$Y \text{ mod } 6$

**Table 4.4** Unary arithmetic operators

Operator	Operation	Operand type	Result type	Example
+	sign identity	integer, real	integer, real	+7
-	sign negation	integer, real	integer, real	-X

The following rules apply to arithmetic operators.

- The value of  $x/y$  is of type *Extended*, regardless of the types of  $x$  and  $y$ . For other arithmetic operators, the result is of type *Extended* whenever at least one operand is a real; otherwise, the result is of type *Int64* when at least one operand is of type *Int64*; otherwise, the result is of type *Integer*. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of  $x \text{ div } y$  is the value of  $x/y$  rounded in the direction of zero to the nearest integer.
- The **mod** operator returns the remainder obtained by dividing its operands. In other words,  $x \text{ mod } y = x - (x \text{ div } y) * y$ .
- A runtime error occurs when  $y$  is zero in an expression of the form  $x/y$ ,  $x \text{ div } y$ , or  $x \text{ mod } y$ .

## Boolean operators

The Boolean operators **not**, **and**, **or**, and **xor** take operands of any Boolean type and return a value of type *Boolean*.

**Table 4.5** Boolean operators

Operator	Operation	Operand types	Result type	Example
<b>not</b>	negation	Boolean	<i>Boolean</i>	<b>not</b> (C <b>in</b> MySet)
<b>and</b>	conjunction	Boolean	<i>Boolean</i>	Done <b>and</b> (Total > 0)
<b>or</b>	disjunction	Boolean	<i>Boolean</i>	A <b>or</b> B
<b>xor</b>	exclusive disjunction	Boolean	<i>Boolean</i>	A <b>xor</b> B

These operations are governed by standard rules of Boolean logic. For example, an expression of the form  $x \text{ and } y$  is *True* if and only if both  $x$  and  $y$  are *True*.

## Complete versus short-circuit Boolean evaluation

The compiler supports two modes of evaluation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation. *Complete evaluation* means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. *Short-circuit evaluation* means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression  $A \text{ and } B$  is evaluated under short-circuit mode when  $A$  is *False*, the compiler won't evaluate  $B$ ; it knows that the entire expression is *False* as soon as it evaluates  $A$ .



Short-circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string *S*, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  :
  Inc(I);
end;
```

In a case where *S* has no commas, the last iteration increments *I* to a value which is greater than the length of *S*. When the **while** condition is next tested, complete evaluation results in an attempt to read *S[I]*, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the **while** condition—(*S[I] <> ', '*)—is not evaluated after the first part fails.

Use the **\$B** compiler directive to control evaluation mode. The default state is **{\$B-}**, which enables short-circuit evaluation. To enable complete evaluation locally, add the **{\$B+}** directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting Complete Boolean Evaluation in the Compiler Options dialog.

**Note** If either operand involves a variant, the compiler always performs complete evaluation (even in the **{\$B-}** state).

## Logical (bitwise) operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in *X* (in binary) is 001101 and the value stored in *Y* is 100001, the statement

```
Z := X or Y;
```

assigns the value 101101 to *Z*.

**Table 4.6** Logical (bitwise) operators

Operator	Operation	Operand types	Result type	Examples
<b>not</b>	bitwise negation	integer	integer	<b>not</b> X
<b>and</b>	bitwise and	integer	integer	X <b>and</b> Y
<b>or</b>	bitwise or	integer	integer	X <b>or</b> Y
<b>xor</b>	bitwise xor	integer	integer	X <b>xor</b> Y
<b>shl</b>	bitwise shift left	integer	integer	X <b>shl</b> 2
<b>shr</b>	bitwise shift right	integer	integer	Y <b>shr</b> I

The following rules apply to bitwise operators.

- The result of a **not** operation is of the same type as the operand.

- If the operands of an **and**, **or**, or **xor** operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations  $x \text{ shl } y$  and  $x \text{ shr } y$  shift the value of  $x$  to the left or right by  $y$  bits, which is equivalent to multiplying or dividing  $x$  by  $2^y$ ; the result is of the same type as  $x$ . For example, if  $N$  stores the value 01101 (decimal 13), then  $N \text{ shl } 1$  returns 11010 (decimal 26). Note that the value of  $y$  is interpreted modulo the size of the type of  $x$ . Thus for example, if  $x$  is an integer,  $x \text{ shl } 40$  is interpreted as  $x \text{ shl } 8$  because an integer is 32 bits and  $40 \bmod 32$  is 8.

## String operators

The relational operators  $=$ ,  $<>$ ,  $<$ ,  $>$ ,  $<=$ , and  $>=$  all take string operands (see “Relational operators” on page 4-10). The  $+$  operator concatenates two strings.

**Table 4.7** String operators

Operator	Operation	Operand types	Result type	Example
$+$	concatenation	string, packed string, character	string	$S + ' . '$

The following rules apply to string concatenation.

- The operands for  $+$  can be strings, packed strings (packed arrays of type *Char*), or characters. However, if one operand is of type *WideChar*, the other operand must be a long string.
- The result of a  $+$  operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

## Pointer operators

The relational operators  $<$ ,  $>$ ,  $<=$ , and  $>=$  can take operands of type *PChar* (see “Relational operators” on page 4-10). The following operators also take pointers as operands. For more information about pointers, see “Pointers and pointer types” on page 5-25.

**Table 4.8** Character-pointer operators

Operator	Operation	Operand types	Result type	Example
$+$	pointer addition	character pointer, integer	character pointer	$P + I$
$-$	pointer subtraction	character pointer, integer	character pointer, integer	$P - Q$
$\wedge$	pointer dereference	pointer	base type of pointer	$P^\wedge$
$=$	equality	pointer	<i>Boolean</i>	$P = Q$
$<>$	inequality	pointer	<i>Boolean</i>	$P <> Q$

The  $\wedge$  operator dereferences a pointer. Its operand can be a pointer of any type except the generic *Pointer*, which must be typecast before dereferencing.

$P = Q$  is *True* just in case  $P$  and  $Q$  point to the same address; otherwise,  $P <> Q$  is *True*.

You can use the  $+$  and  $-$  operators to increment and decrement the offset of a character pointer. You can also use  $-$  to calculate the difference between the offsets of two character pointers. The following rules apply.

- If  $I$  is an integer and  $P$  is a character pointer, then  $P + I$  adds  $I$  to the address given by  $P$ ; that is, it returns a pointer to the address  $I$  characters after  $P$ . (The expression  $I + P$  is equivalent to  $P + I$ .)  $P - I$  subtracts  $I$  from the address given by  $P$ ; that is, it returns a pointer to the address  $I$  characters before  $P$ .
- If  $P$  and  $Q$  are both character pointers, then  $P - Q$  computes the difference between the address given by  $P$  (the higher address) and the address given by  $Q$  (the lower address); that is, it returns an integer denoting the number of characters between  $P$  and  $Q$ .  $P + Q$  is not defined.

## Set operators

The following operators take sets as operands.

**Table 4.9** Set operators

Operator	Operation	Operand types	Result type	Example
$+$	union	set	set	$\text{Set1} + \text{Set2}$
$-$	difference	set	set	$S - T$
$*$	intersection	set	set	$S * T$
$\leq$	subset	set	<i>Boolean</i>	$Q \leq \text{MySet}$
$\geq$	superset	set	<i>Boolean</i>	$S1 \geq S2$
$=$	equality	set	<i>Boolean</i>	$S2 = \text{MySet}$
$\langle \rangle$	inequality	set	<i>Boolean</i>	$\text{MySet} \langle \rangle S1$
<b>in</b>	membership	ordinal, set	<i>Boolean</i>	$A \text{ in } \text{Set1}$

The following rules apply to  $+$ ,  $-$ , and  $*$ .

- An ordinal  $O$  is in  $X + Y$  if and only if  $O$  is in  $X$  or  $Y$  (or both).  $O$  is in  $X - Y$  if and only if  $O$  is in  $X$  but not in  $Y$ .  $O$  is in  $X * Y$  if and only if  $O$  is in both  $X$  and  $Y$ .
- The result of a  $+$ ,  $-$ , or  $*$  operation is of the type **set of  $A..B$** , where  $A$  is the smallest ordinal value in the result set and  $B$  is the largest.

The following rules apply to  $\leq$ ,  $\geq$ ,  $=$ ,  $\langle \rangle$ , and **in**.

- $X \leq Y$  is *True* just in case every member of  $X$  is a member of  $Y$ ;  $Z \geq W$  is equivalent to  $W \leq Z$ .  $U = V$  is *True* just in case  $U$  and  $V$  contain exactly the same members; otherwise,  $U \langle \rangle V$  is *True*.
- For an ordinal  $O$  and a set  $S$ ,  $O \text{ in } S$  is *True* just in case  $O$  is a member of  $S$ .

## Relational operators

Relational operators are used to compare two operands. The operators  $=$ ,  $\langle \rangle$ ,  $\leq$ , and  $\geq$  also apply to sets (see “Set operators” on page 4-10);  $=$  and  $\langle \rangle$  also apply to pointers (see “Pointer operators” on page 4-9).

**Table 4.10** Relational operators

Operator	Operation	Operand types	Result type	Example
=	equality	simple, class, class reference, interface, string, packed string	<i>Boolean</i>	<code>I = Max</code>
<>	inequality	simple, class, class reference, interface, string, packed string	<i>Boolean</i>	<code>X &lt;&gt; Y</code>
<	less-than	simple, string, packed string, <i>PChar</i>	<i>Boolean</i>	<code>X &lt; Y</code>
>	greater-than	simple, string, packed string, <i>PChar</i>	<i>Boolean</i>	<code>Len &gt; 0</code>
<=	less-than-or-equal-to	simple, string, packed string, <i>PChar</i>	<i>Boolean</i>	<code>Cnt &lt;= I</code>
>=	greater-than-or-equal-to	simple, string, packed string, <i>PChar</i>	<i>Boolean</i>	<code>I &gt;= 1</code>

For most simple types, comparison is straightforward. For example,  $I = J$  is *True* just in case  $I$  and  $J$  have the same value, and  $I <> J$  is *True* otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordering of the extended ASCII character set. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with  $n$  components is compared to a string, the packed string is treated as a string of length  $n$ .
- The operators  $<$ ,  $>$ ,  $<=$ , and  $>=$  apply to *PChar* operands only if the two pointers point within the same character array.
- The operators  $=$  and  $<>$  can take operands of class and class-reference types. With operands of a class type,  $=$  and  $<>$  are evaluated according the rules that apply to pointers:  $C = D$  is *True* just in case  $C$  and  $D$  point to the same instance object, and  $C <> D$  is *True* otherwise. With operands of a class-reference type,  $C = D$  is *True* just in case  $C$  and  $D$  denote the same class, and  $C <> D$  is *True* otherwise. For more information about classes, see Chapter 7, “Classes and objects”.

## Class operators

The operators **as** and **is** take classes and instance objects as operands; **as** operates on interfaces as well. For more information, see Chapter 7, “Classes and objects” and Chapter 10, “Object interfaces”.

The relational operators  $=$  and  $<>$  also operate on classes. See “Relational operators” on page 4-10.

## The @ operator

The @ operator returns the address of a variable, or of a function, procedure, or method; that is, @ constructs a pointer to its operand. For more information about pointers, see “Pointers and pointer types” on page 5-25. The following rules apply to @.

- If  $X$  is a variable, @ $X$  returns the address of  $X$ . (Special rules apply when  $X$  is a procedural variable; see “Procedural types in statements and expressions” on page 5-29.) The type of @ $X$  is *Pointer* if the default **{ $\$T-$ }** compiler directive is in effect. In the **{ $\$T+$ }** state, @ $X$  is of type  $^T$ , where  $T$  is the type of  $X$ .
- If  $F$  is a routine (a function or procedure), @ $F$  returns  $F$ 's entry point. The type of @ $F$  is always *Pointer*.
- When @ is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

```
@TMyClass.DoSomething
```

points to the *DoSomething* method of *TMyClass*. For more information about classes and methods, see Chapter 7, “Classes and objects”.

## Operator precedence rules

In complex expressions, rules of precedence determine the order in which operations are performed.

**Table 4.11** Precedence of operators

Operators	Precedence
@, not	first (highest)
*, /, div, mod, and, shl, shr, as	second
+, -, or, xor	third
=, <>, <, >, <=, >=, in, is	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

$$X + Y * Z$$

multiplies  $Y$  times  $Z$ , then adds  $X$  to the result;  $*$  is performed first, because it has a higher precedence than  $+$ . But

$$X - Y + Z$$

first subtracts  $Y$  from  $X$ , then adds  $Z$  to the result;  $-$  and  $+$  have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

$$(X + Y) * Z$$

multiplies  $Z$  times the sum of  $X$  and  $Y$ .

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

$$X = Y \text{ or } X = Z$$

The intended interpretation of this is obviously

$$(X = Y) \text{ or } (X = Z)$$

Without parentheses, however, the compiler follows operator precedence rules and reads it as

$$(X = (Y \text{ or } X)) = Z$$

—which results in a compilation error unless  $Z$  is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example above could be written as

$$X + (Y * Z)$$

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

## Function calls

---

Because functions return a value, function calls are expressions. For example, if you've defined a function called *Calc* that takes two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If  $I$  and  $J$  are integer variables, then  $I + \text{Calc}(J, 8)$  is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

For more information about functions, see Chapter 6, “Procedures and functions”.

## Set constructors

---

A set constructor denotes a set-type value. For example,

```
[5, 6, 7, 8]
```

denotes the set whose members are 5, 6, 7, and 8. The set constructor

```
[ 5..8 ]
```

could also denote the same set.

The syntax for a set constructor is

```
[ item1, ..., itemn ]
```

where each *item* is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (..) in between. When an *item* has the form *x..y*, it is shorthand for all the ordinals in the range from *x* to *y*, inclusive; but if *x* is greater than *y*, then *x..y* denotes nothing and [*x..y*] is the empty set. The set constructor [ ] denotes the empty set, while [*x*] denotes the set whose only member is the value of *x*.

Examples of set constructors:

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see “Sets” on page 5-17.

## Indexes

---

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if *FileName* is a string variable, the expression *FileName*[3] returns the third character in the string denoted by *FileName*, while *FileName*[*I* + 1] returns the character immediately after the one indexed by *I*. For information about strings, see “String types” on page 5-10. For information about arrays and array properties, see “Arrays” on page 5-18 and “Array properties” on page 7-19.

## Typecasts

---

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, *Integer('A')* casts the character *A* as an integer.

The syntax for a typecast is

```
typeIdentifier (expression)
```

If the expression is a variable, the result is called a *variable typecast*; otherwise, the result is a *value typecast*. While their syntax is the same, different rules apply to the two kinds of typecast.

### Value typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal types or both be pointer types. Examples of value typecasts include

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

## The statement

```
I := Integer('A');
```

assigns the value of `Integer('A')`—that is, 65—to the variable *I*.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

## Variable typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like *Int* and *Trunc*.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
:
Shortint(MyChar) := 122;
```

assigns the character *z* (ASCII 122) to *MyChar*.

You can cast variables to a procedural type. For example, given the declarations

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can make the following assignments.

```
F := Func(P);           { Assign procedural value in P to F }
Func(P) := F;          { Assign procedural value in F to P }
@F := P;               { Assign pointer value in P to F }
P := @F;               { Assign pointer value in F to P }
N := F(N);             { Call function via F }
N := Func(P)(N);      { Call function via P }
```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
  end;
  PByte = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
```



## Declarations and statements

```
P: Pointer;
begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

In this example, *TByteRec* is used to access the low- and high-order bytes of a word, and *TWordRec* to access the low- and high-order words of a long integer. You could call the predefined functions *Lo* and *Hi* for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see “Pointers and pointer types” on page 5-25. For information about casting class and interface types, see “The as operator” on page 7-25 and “Interface typecasts” on page 10-10.

## Declarations and statements

---

Aside from the **uses** clause (and reserved words like **implementation** that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

### Declarations

---

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like 26057 are not identifiers.) Identifiers must be *declared* before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable *Result* when it occurs inside a function block, and the variable *Self* when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called *Size* that holds an *Extended* (real) value, while

```
function DoThis(X, Y: string): Integer;
```

declares a function called *DoThis* that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
  Size: Extended;
  Quantity: Integer;
  Description: string;
```

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or implementation section of a unit (after the **uses** clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the chapters on those topics.

The “hint” directives **platform**, **deprecated**, and **library** may be appended to any declaration, except that units cannot be declared with **deprecated**. In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall; deprecated;
var VersionNumber: Real library;
type AppError = class(Exception)
:
end platform;
```

When source code is compiled in the **{SHINTS ON}** **{WARNINGS ON}** state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use **platform** to mark items that are specific to a particular operating environment (such as Windows or Linux), **deprecated** to indicate that an item is obsolete or supported only for backward compatibility, and **library** to flag dependencies on a particular library or component framework (such as VCL or CLX).

## Statements

---

Statements define algorithmic actions within a program. Simple statements—like assignments and procedure calls—can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

### Simple statements

---

A simple statement doesn’t contain any other statements. Simple statements include assignments, calls to procedures and functions, and **goto** jumps.

### Assignment statements

An assignment statement has the form

*variable* := *expression*

where *variable* is any variable reference—including a variable, variable typecast, dereferenced pointer, or component of a structured variable—and *expression* is any assignment-compatible expression. (Within a function block, *variable* can be replaced with the name of the function being defined. See Chapter 6, “Procedures and functions”.) The := symbol is sometimes called the *assignment operator*.

An assignment statement replaces the current value of *variable* with the value of *expression*. For example,

```
I := 3;
```

assigns the value 3 to the variable *I*. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

```
I := I + 1;
```

increments the value of *I*. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

## Procedure and function calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X, Y);
```

With extended syntax enabled (**(\$X+)**), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see Chapter 6, “Procedures and functions”.

## Goto statements

A **goto** statement, which has the form

```
goto label
```

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

```
label: statement
```

Declare labels like this:

```
label label;
```

You can declare several labels at once:

```
label label1, ..., labeln;
```

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and **goto** statement must belong to the same block. (See “Blocks and scope” on page 4-27.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;
:
StartHere: Beep;
goto StartHere;
```

creates an infinite loop that calls the *Beep* procedure repeatedly.

The **goto** statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

      : {code to execute if no answer is found }
      Exit;

  FoundAnAnswer:
    : { code to execute when an answer is found }
end;
```

Notice that we are using **goto** to jump *out* of a nested loop. Never jump *into* a loop or other structured statement, since this can have unpredictable effects.

## Structured statements

---

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or **with** statement simply executes a sequence of constituent statements.
- A conditional statement—that is, an **if** or **case** statement—executes at most one of its constituents, depending on specified criteria.

- Loop statements—including **repeat**, **while**, and **for** loops—execute a sequence of constituent statements repeatedly.
- A special group of statements—including **raise**, **try...except**, and **try...finally** constructions—create and handle *exceptions*. For information about exception generation and handling, see “Exceptions” on page 7-26.

## Compound statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words **begin** and **end**, and its constituent statements are separated by semicolons. For example:

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

The last semicolon before **end** is optional. So we could have written this as

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

Compound statements are essential in contexts where Object Pascal syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
  I := SomeConstant;
  while I > 0 do
    begin
      :
      I := I - 1;
    end;
end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, **begin** and **end** sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

## With statements

A **with** statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a **with** statement is

```
with obj do statement
```

or

```
with obj1, ..., objn do statement
```

where *obj* is a variable reference denoting an object or record, and *statement* is any simple or structured statement. Within *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone—without qualifiers.

For example, given the declarations

```
type TDate = record
  Day: Integer;
  Month: Integer;
  Year: Integer;
end;

var OrderDate: TDate;
```

you could write the following **with** statement.

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1;
  end
  else
    Month := Month + 1;
```

This is equivalent to

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1;
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before *statement* is executed. This makes **with** statements efficient as well as concise. It also means that assignments to a variable within *statement* cannot affect the interpretation of *obj* during the current execution of the **with** statement.

Each variable reference or method name in a **with** statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the **with** statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year
    :
    :
  end;
```

When multiple objects or records appear after **with**, the entire statement is treated like a series of nested **with** statements. Thus

```
with obj1, obj2, ..., objn do statement
```

is equivalent to

```
with obj1 do
  with obj2 do
    :
  with objn do
    statement
```

In this case, each variable reference or method name in *statement* is interpreted, if possible, as a member of *obj*<sub>*n*</sub>; otherwise it is interpreted, if possible, as a member of *obj*<sub>*n*-1</sub>; and so forth. The same rule applies to interpreting the *objs* themselves, so that, for instance, if *obj*<sub>*n*</sub> is a member of both *obj*<sub>1</sub> and *obj*<sub>2</sub>, it is interpreted as *obj*<sub>2</sub>.*obj*<sub>*n*</sub>.

## If statements

There are two forms of **if** statement: **if...then** and the **if...then...else**. The syntax of an **if...then** statement is

```
if expression then statement
```

where *expression* returns a Boolean value. If *expression* is *True*, then *statement* is executed; otherwise it is not. For example,

```
if J <> 0 then Result := I/J;
```

The syntax of an **if...then...else** statement is

```
if expression then statement1 else statement2
```

where *expression* returns a Boolean value. If *expression* is *True*, then *statement*<sub>1</sub> is executed; otherwise *statement*<sub>2</sub> is executed. For example,

```
if J = 0 then
  Exit
else
  Result := I/J;
```

The **then** and **else** clauses contain one statement each, but it can be a structured statement. For example,

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True
else
  Exit;
```

Notice that there is never a semicolon between the **then** clause and the word **else**. You can place a semicolon after an entire **if** statement to separate it from the next statement in its block, but the **then** and **else** clauses require nothing more than a space or carriage return between them. Placing a semicolon immediately before **else** (in an **if** statement) is a common programming error.

A special difficulty arises in connection with nested **if** statements. The problem arises because some **if** statements have **else** clauses while others do not, but the syntax for

the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer **else** clauses than **if** statements, it may not seem clear which **else** clauses are bound to which **ifs**. Consider a statement of the form

```
if expression1 then if expression2 then statement1 else statement2;
```

There would appear to be two ways to parse this:

```
if expression1 then [ if expression2 then statement1 else statement2 ];
```

```
if expression1 then [ if expression2 then statement1 ] else statement2;
```

The compiler always parses in the first way. That is, in real code, the statement

```
if ... { expression1 } then
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 } ;
```

is equivalent to

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 }
end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each **else** bound to the nearest available **if** on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { statement1 }
end
else
  ... { statement2 } ;
```

## Case statements

The **case** statement provides a readable alternative to complex nested **if** conditionals. A **case** statement has the form

```
case selectorExpression of
  caseList1: statement1;
  ⋮
  caseListn: statementn;
end
```

where *selectorExpression* is any expression of an ordinal type (string types are invalid) and each *caseList* is one of the following:



- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus 7, True, 4 + 5 \* 3, 'A', and Integer('A') can all be used as *caseLists*, but variables and most function calls cannot. (A few built-in functions like *Hi* and *Lo* can occur in a *caseList*. See "Constant expressions" on page 5-41.)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item<sub>1</sub>, ..., item<sub>n</sub>*, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the **case** statement; subranges and lists cannot overlap. A **case** statement can have a final **else** clause:

```

case selectorExpression of
  caseList1: statement1;
  :
  caseListn: statementn;
else
  statements;
end

```

where *statements* is a semicolon-delimited sequence of statements. When a **case** statement is executed, at most one of *statement<sub>1</sub> ... statement<sub>n</sub>* is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the *statement* to be used. If none of the *caseLists* has the same value as *selectorExpression*, then the statements in the **else** clause (if there is one) are executed.

The **case** statement

```

case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end;

```

is equivalent to the nested conditional

```

if I in [1..5] then
  Caption := 'Low'
else if I in [6..10] then
  Caption := 'High'
else if (I = 0) or (I in [10..99]) then
  Caption := 'Out of range'
else
  Caption := '';

```

Other examples of **case** statements:

```

case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X := 3;
  Yellow, Orange, Black: X := 0;

```

```

end;
case Selection of
  Done: Form1.Close;
  Compute: CalculateTotal(UnitCost, Quantity);
else
  Beep;
end;

```

## Control loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Object Pascal has three kinds of control loop: **repeat** statements, **while** statements, and **for** statements.

You can use the standard *Break* and *Continue* procedures to control the flow of a **repeat**, **while**, or **for** statement. *Break* terminates the statement in which it occurs, while *Continue* begins executing the next iteration of the sequence. For more information about these procedures, see the online Help.

## Repeat statements

The syntax of a **repeat** statement is

```
repeat statement1; ...; statementn; until expression
```

where *expression* returns a Boolean value. (The last semicolon before **until** is optional.) The **repeat** statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns *True*, the **repeat** statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of **repeat** statements include

```

repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);

```

## While statements

A **while** statement is similar to a **repeat** statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a **while** statement is

```
while expression do statement
```

where *expression* returns a Boolean value and *statement* can be a compound statement. The **while** statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns *True*, execution continues.

Examples of **while** statements include

```

while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;

```

## For statements

A **for** statement, unlike a **repeat** or **while** statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a **for** statement is

```
for counter := initialValue to finalValue do statement
```

or

```
for counter := initialValue downto finalValue do statement
```

where

- *counter* is a local variable (declared in the block containing the **for** statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with *counter*.
- *statement* is a simple or structured statement that does not change the value of *counter*.

The **for** statement assigns the value of *initialValue* to *counter*, then executes *statement* repeatedly, incrementing or decrementing *counter* after each iteration. (The **for...to** syntax increments *counter*, while the **for...downto** syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the **for** statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a **for...to** statement, or less than *finalValue* in a **for...downto** statement, then *statement* is never executed. After the **for** statement terminates, the value of *counter* is undefined.

For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the **for...to** statement is almost, but not quite, equivalent to this **while** construction:

```

begin
  counter := initialValue;
  while counter <= finalValue do

```

```

begin
  statement;
  counter := Succ(counter);
end;
end

```

The difference between this construction and the **for...to** statement is that the **while** loop re-evaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of **for** statements:

```

for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];
  end if;
end for;

for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
end for;

for I := 1 to 10 do
  for J := 1 to 10 do
    begin
      X := 0;
      for K := 1 to 10 do
        X := X + Mat1[I, K] * Mat2[K, J];
      end for;
      Mat[I, J] := X;
    end;
  end for;
end for;

for C := Red to Blue do Check(C);
end for;

```

## Blocks and scope

---

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

### Blocks

---

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```

declarations
begin
  statements
end

```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program

block, the *declarations* section can also include one or more **exports** clauses (see Chapter 9, “Libraries and packages”).

For example, in a function declaration like

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  :
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. *Ch*, *L*, *Source*, and *Dest* are local variables; their declarations apply only to the *UpperCase* function block and override—in this block only—any declarations of the same identifiers that may occur in the program block or in the interface or implementation section of a unit.

## Scope

---

An identifier, such as a variable or function name, can be used only within the *scope* of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope—especially identifiers declared in functions and procedures—are sometimes called *local*, while identifiers with wider scope are called *global*.

The rules that determine identifier scope are summarized below.

<b>If the identifier is declared in ...</b>	<b>its scope extends ...</b>
the declaration of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Chapter 3, “Programs and units”.)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See “Records” on page 5-21.)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Chapter 7, “Classes and objects”.)

## Naming conflicts

When one block encloses another, the former is called the *outer block* and the latter the *inner block*. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration overrides the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called *MaxValue* in the interface section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of *MaxValue* in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a **uses** clause imposes a new scope that encloses the remaining units used and the program or unit containing the **uses** clause. The first unit in a **uses** clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their interface sections, an unqualified reference to the identifier selects the declaration in the innermost scope—that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the **uses** clause that does declare the identifier.

The *System* unit is used automatically by every program or unit. The declarations in *System*, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and by-pass an inner declaration by using a qualified identifier (see “Qualified identifiers” on page 4-2) or a **with** statement (see “With statements” on page 4-20).



## Data types, variables, and constants

A *type* is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

Object Pascal is a “strongly typed” language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include *typecasting* (see “Typecasts” on page 4-14), *pointers* (see “Pointers and pointer types” on page 5-25), *variants* (see “Variant types” on page 5-30), *variant parts* in records (see “Variant parts in records” on page 5-22), and *absolute addressing* of variables (see “Absolute addresses” on page 5-38).

### About types

---

There are several ways to categorize Object Pascal data types:

- Some types are *predefined* (or *built-in*); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either *fundamental* or *generic*. The range and format of a fundamental type is the same in all implementations of Object Pascal, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It’s a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage



format from one implementation of a generic type to the next could cause compatibility problems—for example, if you are streaming data to a file.

- Types can be classified as *simple*, *string*, *structured*, *pointer*, *procedural*, or *variant*. In addition, type identifiers themselves can be regarded as belonging to a special “type” because they can be passed as parameters to certain functions (such as *High*, *Low*, and *SizeOf*).

The outline below shows the taxonomy of Object Pascal data types.

**simple**

ordinal

integer

character

Boolean

enumerated

subrange

real

**string**

**structured**

set

array

record

file

class

class reference

interface

**pointer**

**procedural**

**variant**

**type identifier**

The standard function *SizeOf* operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) used to store data of the specified type. For example, *SizeOf(Longint)* returns 4, since a *Longint* variable uses four bytes of memory.

Type declarations are illustrated in the sections that follow. For general information about type declarations, see “Declaring types” on page 5-36.

## Simple types

---

Simple types, which include *ordinal* types and *real* types, define ordered sets of values.

### Ordinal types

---

Ordinal types include *integer*, *character*, *Boolean*, *enumerated*, and *subrange* types. An ordinal type defines an ordered set of values in which each value except the first has

a unique *predecessor* and each value except the last has a unique *successor*. Further, each value has an *ordinality* which determines the ordering of the type. In most cases, if a value has ordinality  $n$ , its predecessor has ordinality  $n-1$  and its successor has ordinality  $n+1$ .

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
<i>Ord</i>	ordinal expression	ordinality of expression's value	Does not take <i>Int64</i> arguments.
<i>Pred</i>	ordinal expression	predecessor of expression's value	Do not use on properties that have a <b>write</b> procedure.
<i>Succ</i>	ordinal expression	successor of expression's value	Do not use on properties that have a <b>write</b> procedure.
<i>High</i>	ordinal type identifier or variable of ordinal type	highest value in type	Also operates on short-string types and arrays.
<i>Low</i>	ordinal type identifier or variable of ordinal type	lowest value in type	Also operates on short-string types and arrays.

For example, *High(Byte)* returns 255 because the highest value of type *Byte* is 255, and *Succ(2)* returns 3 because 3 is the successor of 2.

The standard procedures *Inc* and *Dec* increment and decrement the value of an ordinal variable. For example, *Inc(I)* is equivalent to  $I := Succ(I)$  and, if  $I$  is an integer variable, to  $I := I + 1$ .

## Integer types

An integer type represents a subset of the whole numbers. The generic integer types are *Integer* and *Cardinal*; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the current 32-bit Object Pascal compiler.

**Table 5.1** Generic integer types for 32-bit implementations of Object Pascal

Type	Range	Format
<i>Integer</i>	-2147483648..2147483647	signed 32-bit
<i>Cardinal</i>	0..4294967295	unsigned 32-bit

Fundamental integer types include *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word*, and *Longword*.

**Table 5.2** Fundamental integer types

Type	Range	Format
<i>Shortint</i>	-128..127	signed 8-bit
<i>Smallint</i>	-32768..32767	signed 16-bit
<i>Longint</i>	-2147483648..2147483647	signed 32-bit
<i>Int64</i>	$-2^{63}..2^{63}-1$	signed 64-bit
<i>Byte</i>	0..255	unsigned 8-bit
<i>Word</i>	0..65535	unsigned 16-bit
<i>Longword</i>	0..4294967295	unsigned 32-bit

In general, arithmetic operations on integers return a value of type *Integer*—which, in its current implementation, is equivalent to the 32-bit *Longint*. Operations return a value of type *Int64* only when performed on an *Int64* operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
  :
  I := High(Integer);
  J := I + 1;
```

To get an *Int64* return value in this situation, cast *I* as *Int64*:

```
  :
  J := Int64(I) + 1;
```

For more information, see “Arithmetic operators” on page 4-6.

**Note** Most standard routines that take integer arguments truncate *Int64* values to 32 bits. However, the *High*, *Low*, *Succ*, *Pred*, *Inc*, *Dec*, *IntToStr*, and *IntToHex* routines fully support *Int64* arguments. Also, the *Round*, *Trunc*, *StrToInt64*, and *StrToInt64Def* functions return *Int64* values. A few routines—including *Ord*—cannot take *Int64* values at all.

When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the *Shortint* type has the range -128..127; hence, after execution of the code

```
var I: Shortint;
  :
  I := High(Shortint);
  I := I + 1;
```

the value of *I* is -128. If compiler range-checking is enabled, however, this code generates a runtime error.

## Character types

The fundamental character types are *AnsiChar* and *WideChar*. *AnsiChar* values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. *AnsiChar* was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

*WideChar* characters use more than one byte to represent every character. In the current implementations, *WideChar* is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is *Char*, which is equivalent to *AnsiChar*. Because the implementation of *Char* is subject to change, it's a good idea to use the standard function *SizeOf* rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

A string constant of length 1, such as 'A', can denote a character value. The predefined function *Chr* returns the character value for any integer in the range of *AnsiChar* or *WideChar*; for example, *Chr*(65) returns the letter A.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

```
var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
end;
```

*Letter* has the value A (ASCII 65).

For more information about Unicode characters, see “About extended character sets” on page 5-13 and “Working with null-terminated strings” on page 5-13.

## Boolean types

The four predefined Boolean types are *Boolean*, *ByteBool*, *WordBool*, and *LongBool*. *Boolean* is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A *Boolean* variable occupies one byte of memory, a *ByteBool* variable also occupies one byte, a *WordBool* variable occupies two bytes (one word), and a *LongBool* variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants *True* and *False*. The following relationships hold.

Boolean	ByteBool, WordBool, LongBool
<i>False</i> < <i>True</i>	<i>False</i> <> <i>True</i>
<i>Ord(False)</i> = 0	<i>Ord(False)</i> = 0
<i>Ord(True)</i> = 1	<i>Ord(True)</i> <> 0
<i>Succ(False)</i> = <i>True</i>	<i>Succ(False)</i> = <i>True</i>
<i>Pred(True)</i> = <i>False</i>	<i>Pred(False)</i> = <i>True</i>

A value of type *ByteBool*, *LongBool*, or *WordBool* is considered *True* when its ordinality is nonzero. If such a value appears in a context where a *Boolean* is expected, the compiler automatically converts any value of nonzero ordinality to *True*.

The remarks above refer to the ordinality of Boolean values—not to the values themselves. In Object Pascal, Boolean expressions cannot be equated with integers or reals. Hence, if *X* is an integer variable, the statement

```
if X then ...;
```

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```
if X <> 0 then ...;           { use longer expression that returns Boolean value }
var OK: Boolean               { use Boolean variable }
:
:
if X <> 0 then OK := True;
if OK then ...;
```

## Enumerated types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

```
type typeName = (val1, ..., valn)
```

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called *Suit* whose possible values are *Club*, *Diamond*, *Heart*, and *Spade*, where *Ord(Club)* returns 0, *Ord(Diamond)* returns 1, and so forth.

When you declare an enumerated type, you are declaring each *val* to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock);
```

Unfortunately, *Click* is also the name of a method defined for *TControl* and all of the objects in the VCL and/or CLX that descend from it. So if you're writing an application and you create an event handler like

```

procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
  :
  Thing := Click;
  :
end;

```

you'll get a compilation error; the compiler interprets *Click* within the scope of the procedure as a reference to *TForm's Click* method. You can work around this by qualifying the identifier; thus, if *TSound* is declared in *MyUnit*, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:

```

type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe);

```

You can use the (*val*<sub>1</sub>, ..., *val*<sub>n</sub>) construction directly in variable declarations, as if it were a type name:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare *MyCard* this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```

var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);

```

generates a compilation error. But

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```

type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;

```

### Enumerated types with explicitly assigned ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with = *constantExpression*, where *constantExpression* is a constant expression that evaluates to an integer. (See "Constant expressions" on page 5-41) For example,

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

defines a type called *Size* whose possible values include *Small*, *Medium*, and *Large*, where *Ord(Small)* returns 5, *Ord(Medium)* returns 10, and *Ord(Large)* returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration.

In the example above, the *Size* type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.) Only three of these values have names, but the others are accessible through typecasts and through routines such as *Pred*, *Succ*, *Inc*, and *Dec*. In the following example, “anonymous” values in the range of *Size* are assigned to the variable *X*.

```
var X: Size;
X := Small; // Ord(X) = 5
X := Size(6); // Ord(X) = 6
Inc(X); // Ord(X) = 7
```

Any value that isn’t explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn’t assigned an ordinality, its ordinality is 0. Hence, given the declaration

```
type SomeEnum = (e1, e2, e3 = 1);
```

*SomeEnum* has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because *e2* and *e3* have the same ordinality, they represent the same value.

## Subrange types

A subrange type represents a subset of the values in another ordinal type (called the *base type*). Any construction of the form *Low*..*High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here *TMyColors* includes the values *Green*, *Yellow*, *Orange*, *Purple*, and *White*.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```
type
SomeNumbers = -128..127;
Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *Low*..*High* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example above, if *Color* is a variable that holds the value *Green*, `Ord(Color)` returns 2 regardless of whether *Color* is of type *TColors* or *TMyColors*.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character

type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type Percentile = 0..99;
var I: Percentile;
:
I := 100;
```

produces an error,

```
:
I := 99;
Inc(I);
```

assigns the value 100 to *I* (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after = is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;
type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

## Real types

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types.

**Table 5.3** Fundamental real types

Type	Range	Significant digits	Size in bytes
<i>Real48</i>	$2.9 \times 10^{-39}$ .. $1.7 \times 10^{38}$	11–12	6
<i>Single</i>	$1.5 \times 10^{-45}$ .. $3.4 \times 10^{38}$	7–8	4
<i>Double</i>	$5.0 \times 10^{-324}$ .. $1.7 \times 10^{308}$	15–16	8
<i>Extended</i>	$3.6 \times 10^{-4951}$ .. $1.1 \times 10^{4932}$	19–20	10
<i>Comp</i>	$-2^{63}+1$ .. $2^{63}-1$	19–20	8
<i>Currency</i>	-922337203685477.5808.. 922337203685477.5807	19–20	8



The generic type *Real*, in its current implementation, is equivalent to *Double*.

**Table 5.4** Generic real types

Type	Range	Significant digits	Size in bytes
<i>Real</i>	$5.0 \times 10^{-324} .. 1.7 \times 10^{308}$	15–16	8

**Note** The six-byte *Real48* type was called *Real* in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte *Real* type, you may want to change it to *Real48*. You can also use the `{$REALCOMPATIBILITY ON}` compiler directive to turn *Real* back into the six-byte type.

The following remarks apply to fundamental real types.

- *Real48* is maintained for backward compatibility. Since its storage format is not native to the Intel CPU family, it results in slower performance than other floating-point types.
- *Extended* offers greater precision than other real types but is less portable. Be careful using *Extended* if you are creating data files to share across platforms.
- The *Comp* (computational) type is native to the Intel CPU and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a *Comp* value.) *Comp* is maintained for backward compatibility only. Use the *Int64* type for better performance.
- *Currency* is a fixed-point data type that minimizes rounding errors in monetary calculations. It is stored as a scaled 64-bit integer with the four least-significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, *Currency* values are automatically divided or multiplied by 10000.

## String types

A string represents a sequence of characters. Object Pascal supports the following predefined string types.

**Table 5.5** String types

Type	Maximum length	Memory required	Used for
<i>ShortString</i>	255 characters	2 to 256 bytes	backward compatibility
<i>AnsiString</i>	$\sim 2^{31}$ characters	4 bytes to 2GB	8-bit (ANSI) characters
<i>WideString</i>	$\sim 2^{30}$ characters	4 bytes to 2GB	Unicode characters; multiuser servers and multi-language applications

*AnsiString*, sometimes called the *long string*, is the preferred type for most purposes.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a

function or procedure (as **var** and **out** parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type (see “Typecasts” on page 4-14).

The reserved word **string** functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable *S* that holds a string. In the default **{SH+}** state, the compiler interprets **string** (when it appears without a bracketed number after it) as *AnsiString*. Use the **{SH-}** directive to turn **string** into *ShortString*.

The standard function *Length* returns the number of characters in a string. The *SetLength* procedure adjusts the length of a string. See the online Help for details.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value. For example, “AB” is greater than “A”; that is, 'AB' > 'A' returns *True*. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If *S* is a string variable and *i* an integer expression, *S*[*i*] represents the *i*th character—or, strictly speaking, the *i*th byte—in *S*. For a *ShortString* or *AnsiString*, *S*[*i*] is of type *AnsiChar*; for a *WideString*, *S*[*i*] is of type *WideChar*. The statement `MyString[2] := 'A';` assigns the value *A* to the second character of *MyString*. The following code uses the standard *UpCase* function to convert *MyString* to uppercase.

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
    begin
      MyString[I] := UpCase(MyString[I]);
      I := I - 1;
    end;
end;
```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as **var** parameters, because this results in inefficient code.

You can assign the value of a string constant—or any other expression that returns a string—to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';           { space }
MyString := '';           { empty string }
```

For more information, see “Character strings” on page 4-4 and “String operators” on page 4-9.

## Short strings

---

A *ShortString* is 0 to 255 characters long. While the length of a *ShortString* can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If *S* is a *ShortString* variable, `Ord(S[0])`, like `Length(S)`, returns the length of *S*; assigning a value to `S[0]`, like calling `SetLength`, changes the length of *S*. *ShortString* uses 8-bit ANSI characters and is maintained for backward compatibility only.

Object Pascal supports short-string types—in effect, subtypes of *ShortString*—whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word **string**. For example,

```
var MyString: string[100];
```

creates a variable called *MyString* whose maximum length is 100 characters. This is equivalent to the declarations

```
type CString = string[100];
var MyString: CString;
```

Variables declared in this way allocate only as much memory as the type requires—that is, the specified maximum length plus one byte. In our example, *MyString* uses 101 bytes, as compared to 256 bytes for a variable of the predefined *ShortString* type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions *High* and *Low* operate on short-string type identifiers and variables. *High* returns the maximum length of the short-string type, while *Low* returns zero.

## Long strings

---

*AnsiString*, also called a *long string*, represents a dynamically allocated string whose maximum length is limited only by available memory. It uses 8-bit ANSI characters.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty—that is, when it contains a zero-length string—the pointer is **nil** and the string uses no additional storage. When the variable is nonempty, it points to a dynamically allocated block of memory that contains the string value, a 32-bit length indicator, and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called *reference-counting*. When indexing is used to change the value of a single character in a string, a copy of the string is made if—but only if—its reference count is greater than one. This is called *copy-on-write* semantics.

## WideString

---

The *WideString* type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to *AnsiString*.

On Win32, *WideString* is compatible with the COM *BSTR* type. Borland development tools have support features that convert *AnsiString* values to *WideString*, but you may need to explicitly cast or convert your strings to *WideString*.

### About extended character sets

Windows and Linux both support *single-byte* and *multibyte* character sets as well as *Unicode*. With a single-byte character set (SBCS), each byte in a string represents one character. The ANSI character set used by many Western operating systems is a single-byte character set.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the *lead byte*. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages, while the UTF-8 character set used by Linux is a multibyte encoding of Unicode.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called *wide characters* and *wide character strings*. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi/Kylix supports UCS-2 on both platforms.

Object Pascal supports single-byte and multibyte characters and strings through the *Char*, *PChar*, *AnsiChar*, *PAnsiChar*, and *AnsiString* types. Indexing of multibyte strings is not reliable, since  $S[i]$  represents the  $i$ th byte (not necessarily the  $i$ th character) in  $S$ . However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with *Ansi-*. For example, the multibyte version of *StrPos* is *AnsiStrPos*.) Multibyte character support is operating-system dependent and based on the current locale.

Object Pascal supports Unicode characters and strings through the *WideChar*, *PWideChar*, and *WideString* types.

## Working with null-terminated strings

---

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on *null-terminated* strings. A null-terminated string is a zero-based array of characters that ends with NULL (#0); since the array has no length indicator, the first NULL character marks the end of the string. You can use Object Pascal constructions and

special routines in the *SysUtils* unit (see Chapter 8, “Standard routines and I/O”) to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

With extended syntax enabled (**(\$X+)**), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won’t work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to #0. For more information about arrays, see “Arrays” on page 5-18.

## Using pointers, arrays, and string constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See “Pointers and pointer types” on page 5-25.) String constants are assignment-compatible with the *PChar* and *PWideChar* types, which represent pointers to null-terminated arrays of *Char* and *WideChar* values. For example,

```
var P: PChar;
    ⋮
    P := 'Hello world!';
```

points *P* to an area of memory that contains a null-terminated copy of “Hello world!” This is equivalent to

```
const TempString: array[0..12] of Char = 'Hello world!#0';
var P: PChar;
    ⋮
    P := @TempString;
```

You can also pass string constants to any function that takes value or **const** parameters of type *PChar* or *PWideChar*—for example `StrUpper('Hello world!')`. As with assignments to a *PChar*, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize *PChar* or *PWideChar* constants with string literals, alone or in a structured type. Examples:

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Zero-based character arrays are compatible with *PChar* and *PWideChar*. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,

```

var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;

```

This code calls *SomeProcedure* twice with the same value.

A character pointer can be indexed as if it were an array. In the example above, *MyPointer*[0] returns *H*. The index specifies an offset added to the pointer before it is dereferenced. (For *PWideChar* variables, the index is automatically multiplied by two.) Thus, if *P* is a character pointer, *P*[0] is equivalent to *P*<sup>^</sup> and specifies the first character in the array, *P*[1] specifies the second character in the array, and so forth; *P*[-1] specifies the “character” immediately to the left of *P*[0]. *The compiler performs no range checking on these indexes.*

The *StrUpper* function illustrates the use of pointer indexing to iterate through a null-terminated string:

```

function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;

```

## Mixing Pascal strings and null-terminated strings

You can mix long strings (*AnsiString* values) and null-terminated strings (*PChar* values) in expressions and assignments, and you can pass *PChar* values to functions or procedures that take long-string parameters. The assignment *S := P*, where *S* is a string variable and *P* is a *PChar* expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a *PChar*, the *PChar* operand is converted to a long string.

You can cast a *PChar* value as a long string. This is useful when you want to perform a string operation on two *PChar* values. For example,

```

S := string(P1) + string(P2);

```

You can also cast a long string as a null-terminated string. The following rules apply.

- If  $S$  is a long-string expression, `PChar(S)` casts  $S$  as a null-terminated string; it returns a pointer to the first character in  $S$ .

On Windows: For example, if  $Str1$  and  $Str2$  are long strings, you could call the Win32 API `MessageBox` function like this:

```
MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
```

(The declaration of `MessageBox` is in the `Windows` interface unit.)

On Linux: For example, if  $Str$  is a long string, you could call the `opendir` system function like this:

```
opendir(PChar(Str));
```

(The declaration of `opendir` is in the `Libc` interface unit.)

- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if  $S$  is empty, the typecast returns `nil`.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the long string only when all of the following conditions are satisfied.
  - The expression cast is a long-string *variable*.
  - The string is not empty.
  - The string is unique—that is, has a reference count of one. To guarantee that the string is unique, call the `SetLength`, `SetString`, or `UniqueString` procedure.
  - The string has not been modified since the typecast was made.
  - The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing `WideString` values with `PWideChar` values.

## Structured types

---

Instances of a structured type hold more than one value. Structured types include *sets*, *arrays*, *records*, and *files* as well as *class*, *class-reference*, and *interface* types. (For information about class and class-reference types, see Chapter 7, “Classes and objects”. For information about interfaces, see Chapter 10, “Object interfaces”.) Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word **packed** to implement compressed data storage. For example,

```
type TNumbers = packed array[1..100] of Real;
```

Using **packed** slows data access and, in the case of a character array, affects type compatibility. For more information, see Chapter 11, “Memory management”.

## Sets

---

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the *base type*; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

```
set of baseType
```

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

create a set type called *TIntSet* whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

```
type TIntSet = set of 1..250;
```

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
  ⋮
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

You can also use the **set of ...** construction directly in variable declarations:

```
var MySet: set of 'a'..'z';
  ⋮
MySet := ['a', 'b', 'c'];
```

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The **in** operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by []. For more information about sets, see “Set constructors” on page 4-13 and “Set operators” on page 4-10.



## Arrays

---

An array represents an indexed collection of elements of the same type (called the *base type*). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

### Static arrays

Static array types are denoted by constructions of the form

```
array[indexType1, ..., indexTypen] of baseType
```

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexTypes* index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexTypes*. In practice, *indexTypes* are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

```
var MyArray: array[1..100] of Char;
```

declares a variable called *MyArray* that holds an array of 100 character values. Given this declaration, *MyArray*[3] denotes the third character in *MyArray*. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type TMatrix = array[1..10, 1..50] of Real;
```

Whichever way *TMatrix* is declared, it represents an array of 500 real values. A variable *MyMatrix* of type *TMatrix* can be indexed like this: *MyMatrix*[2, 45]; or like this: *MyMatrix*[2][45]. Similarly,

```
packed array[Boolean, 1..10, TShoeSize] of Integer;
```

is equivalent to

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

The standard functions *Low* and *High* operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function *Length* returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of *Char* values is called a *packed string*. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See "Type compatibility and identity" on page 5-34.

An array type of the form *array*[0..*x*] of *Char* is called a *zero-based character array*. Zero-based character arrays are used to store null-terminated strings and are compatible with *PChar* values. See "Working with null-terminated strings" on page 5-13.

## Dynamic arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the *SetLength* procedure. Dynamic-array types are denoted by constructions of the form

array of *baseType*

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for *MyFlexibleArray*. To create the array in memory, call *SetLength*. For example, given the declaration above,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign **nil** to a variable that references the array or pass the variable to *Finalize*; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays of length 0 have the value **nil**. Do not apply the dereference operator (^) to a dynamic-array variable or pass it to the *New* or *Dispose* procedure.

If *X* and *Y* are variables of the same dynamic-array type, *X* := *Y* points *X* to the same array as *Y*. (There is no need to allocate memory for *X* before performing this operation.) Unlike strings and static arrays, dynamic arrays are not automatically copied before they are written to. For example, after this code executes—

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

—the value of *A*[0] is 2. (If *A* and *B* were static arrays, *A*[0] would still be 1.)

Assigning to a dynamic-array index (for example, *MyFlexibleArray*[2] := 7) does not reallocate the array. Out-of-range indexes are not reported at compile time.

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

*A* = *B* returns *False* but *A*[0] = *B*[0] returns *True*.

To truncate a dynamic array, pass it to *SetLength* or *Copy* and assign the result back to the array variable. (The *SetLength* procedure is usually faster.) For example, if *A* is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of *A*.

Once a dynamic array has been allocated, you can pass it to the standard functions *Length*, *High*, and *Low*. *Length* returns the number of elements in the array, *High* returns the array's highest index (that is, *Length*-1), and *Low* returns 0. In the case of a zero-length array, *High* returns -1 (with the anomalous consequence that *High* < *Low*).

**Note** In some function and procedure declarations, array parameters are represented as array of *baseType*, without any index types specified. For example,

```
function CheckStrings(A: array of string): Boolean;
```

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically. See "Open array parameters" on page 6-15.

### Multidimensional dynamic arrays

To declare multidimensional dynamic arrays, use iterated **array of ...** constructions. For example,

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

declares a two-dimensional array of strings. To instantiate this array, call *SetLength* with two integer arguments. For example, if *I* and *J* are integer-valued variables,

```
SetLength(Msgs, I, J);
```

allocates an *I*-by-*J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call *SetLength*, passing it parameters for the first *n* dimensions of the array. For example,

```
var Ints: array of array of Integer;
SetLength(Ints, 10);
```

allocates ten rows for *Ints* but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

```
SetLength(Ints[2], 5);
```

makes the third column of *Ints* five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column—for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the *IntToStr* function declared in the *SysUtils* unit) to create a triangular matrix of strings.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
```

```

begin
  SetLength(A[I], I);
  for J := Low(A[I]) to High(A[I]) do
    A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
  end;
end;

```

## Array types and assignments

Arrays are assignment-compatible only if they are of the same type. Because Pascal uses name-equivalence for types, the following code will not compile.

```

var
  Int1: array[1..10] of Integer;
  Int2: array[1..10] of Integer;
  :
Int1 := Int2;

```

To make the assignment work, declare the variables as

```
var Int1, Int2: array[1..10] of Integer;
```

or

```

type IntArray = array[1..10] of Integer;
var
  Int1: IntArray;
  Int2: IntArray;

```

## Records

---

A record (analogous to a *structure* in some languages) represents a heterogeneous set of elements. Each element is called a *field*; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```

type recordTypeName = record
  fieldList1: type1;
  :
  fieldListn: typen;
end

```

where *recordTypeName* is a valid identifier, each *type* denotes a type, and each *fieldList* is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called *TDateRec*.

```

type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;

```

Each *TDateRec* contains three fields: an integer value called *Year*, a value of an enumerated type called *Month*, and another integer between 1 and 31 called *Day*. The identifiers *Year*, *Month*, and *Day* are the *field designators* for *TDateRec*, and they behave like variables. The *TDateRec* type declaration, however, does not allocate any memory for the *Year*, *Month*, and *Day* fields; memory is allocated when you instantiate the record, like this:

```
var Record1, Record2: TDateRec;
```

This variable declaration creates two instances of *TDateRec*, called *Record1* and *Record2*.

You can access the fields of a record by qualifying the field designators with the record's name:

```
Record1.Year := 1904;  
Record1.Month := Jun;  
Record1.Day := 16;
```

Or use a **with** statement:

```
with Record1 do  
begin  
  Year := 1904;  
  Month := Jun;  
  Day := 16;  
end;
```

You can now copy the values of *Record1*'s fields to *Record2*:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the **record ...** construction directly in variable declarations:

```
var S: record  
  Name: string;  
  Age: Integer;  
end;
```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

### Variant parts in records

A record type can have a *variant* part, which looks like a **case** statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```

type recordTypeName = record
  fieldList1: type1;
  ⋮
  fieldListn: typen;
case tag: ordinalType of
  constantList1: (variant1);
  ⋮
  constantListn: (variantn);
end;

```

The first part of the declaration—up to the reserved word **case**—is the same as that of a standard record type. The remainder of the declaration—from **case** to the optional final semicolon—is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (:) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type *ordinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantLists*.
- Each *variant* is a comma-delimited list of declarations resembling the *fieldList: type* constructions in the main part of the record type. That is, a *variant* has the form

```

fieldList1: type1;
⋮
fieldListn: typen;

```

where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each *type* denotes a type, and the final semicolon is optional. The *types* must not be long strings, dynamic arrays, variants (that is, *Variant* types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several *variants* which share the same space in memory. You can read or write to any field of any *variant* at any time; but if you write to a field in one *variant* and then to a field in another *variant*, you may be overwriting your own data. The *tag*, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```

type
  TEmployee = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Salaried: Boolean of
      True: (AnnualSalary: Currency);
      False: (HourlyWage: Currency);
    end;
end;

```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of *TEmployee*, there is no reason to allocate enough memory for both fields. In this case, the only difference between the *variants* is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```

type
  TPerson = record
    FirstName, LastName: string[40];
    BirthDate: TDate;
    case Citizen: Boolean of
      True: (Birthplace: string[40]);
      False: (Country: string[20];
              EntryPort: string[20];
              EntryDate, ExitDate: TDate);
    end;
type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
      Circle: (Radius: Real);
      Ellipse, Other: ();
    end;

```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest *variant*. The optional *tag* and the *constantLists* (like *Rectangle*, *Triangle*, and so forth in the last example above) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit *Real* as the first field in one *variant* and a 32-bit *Integer* as the first field in another, you can assign a value to the *Real* field and then read back the first 32 bits of it as the value of the *Integer* field (passing it, say, to a function that requires integer parameters).

## File types

---

A file is an ordered set of elements of the same type. Standard I/O routines use the predefined *TextFile* or *Text* type, which represents a file containing characters organized into lines. For more information about file input and output, see Chapter 8, “Standard routines and I/O”.

To declare a file type, use the syntax

```
type fileName = file of type
```

where *fileName* is any valid identifier and *type* is a fixed-size type. Pointer types—whether implicit or explicit—are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```
type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;
  PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the **file of ...** construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word **file** by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see “Untyped files” on page 8-4.

Files are not allowed in arrays or records.

## Pointers and pointer types

---

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it *points* to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure.

Pointers are *typed* to indicate the kind of data stored at the addresses they hold. The general-purpose *Pointer* type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

### Overview of pointers

---

To see how pointers work, look at the following example.

```
1  var
2  X, Y: Integer; // X and Y are Integer variables
3  P: ^Integer; // P points to an Integer
4  begin
5  X := 17; // assign a value to X
6  P := @X; // assign the address of X to P
7  Y := P^; // dereference P; assign the result to Y
8  end;
```

Line 2 declares *X* and *Y* as variables of type *Integer*. Line 3 declares *P* as a pointer to an *Integer* value; this means that *P* can point to the location of *X* or *Y*. Line 5 assigns a value to *X*, and line 6 assigns the address of *X* (denoted by *@X*) to *P*. Finally, line 7



retrieves the value at the location pointed to by  $P$  (denoted by  $^P$ ) and assigns it to  $Y$ . After this code executes,  $X$  and  $Y$  have the same value, namely 17.

The  $@$  operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see “The  $@$  operator” on page 4-12 and “Procedural types in statements and expressions” on page 5-29.

The symbol  $^$  has two purposes, both of which are illustrated in our example. When it appears before a type identifier—

*$^$ typeName*

—it denotes a type that represents pointers to variables of type *typeName*. When it appears after a pointer variable—

*pointer $^$*

—it *dereferences* the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another—something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand Object Pascal, since pointers often operate behind the scenes in code where they don’t appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Object Pascal’s strict data typing. By referencing a variable with an all-purpose *Pointer*, casting the *Pointer* to a more specific type, and then dereferencing it, you can treat the data stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

```

type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  :
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from  $R$  to  $I$ , without converting it.

In addition to assigning the result of an  $@$  operation, you can use several standard routines to give a value to a pointer. The *New* and *GetMem* procedures assign a memory address to an existing pointer, while the *Addr* and *Ptr* functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `P1^.Data^`.

The reserved word `nil` is a special constant that can be assigned to any pointer. When `nil` is assigned to a pointer, the pointer doesn't reference anything.

## Pointer types

You can declare a pointer to any type, using the syntax

```
type pointerTypeName = ^type
```

When you define a record or other data type, it's a common practice also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Standard pointer types exist for many purposes. The most versatile is *Pointer*, which can point to data of any kind. But a *Pointer* variable cannot be dereferenced; placing the `^` symbol after a *Pointer* variable causes a compilation error. To access the data referenced by a *Pointer* variable, first cast it to another pointer type and then dereference it.

## Character pointers

The fundamental types *PAnsiChar* and *PWideChar* represent pointers to *AnsiChar* and *WideChar* values, respectively. The generic *PChar* represents a pointer to a *Char* (that is, in its current implementation, to an *AnsiChar*). These character pointers are used to manipulate null-terminated strings. (See "Working with null-terminated strings" on page 5-13.)

## Other standard pointer types

The *System* and *SysUtils* units declare many standard pointer types that are commonly used.

**Table 5.6** Selected pointer types declared in *System* and *SysUtils*

Pointer type	Points to variables of type
<i>PAnsiString</i> , <i>PString</i>	<i>AnsiString</i>
<i>PByteArray</i>	<i>TByteArray</i> (declared in <i>SysUtils</i> ). Used to typecast dynamically allocated memory for array access.
<i>PCurrency</i> , <i>PDouble</i> , <i>PExtended</i> , <i>PSingle</i>	<i>Currency</i> , <i>Double</i> , <i>Extended</i> , <i>Single</i>
<i>PInteger</i>	<i>Integer</i>
<i>POleVariant</i>	<i>OleVariant</i>
<i>PShortString</i>	<i>ShortString</i> . Useful when porting legacy code that uses the old <i>PString</i> type.
<i>PTextBuf</i>	<i>TTextBuf</i> (declared in <i>SysUtils</i> ). <i>TTextBuf</i> is the internal buffer type in a <i>TTextRec</i> file record.)
<i>PVarRec</i>	<i>TVarRec</i> (declared in <i>System</i> )
<i>PVariant</i>	<i>Variant</i>

**Table 5.6** Selected pointer types declared in System and SysUtils (continued)

Pointer type	Points to variables of type
<i>PWideString</i>	<i>WideString</i>
<i>PWordArray</i>	<i>TWordArray</i> (declared in <i>SysUtils</i> ). Used to typecast dynamically allocated memory for arrays of 2-byte values.

## Procedural types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called *Calc* that takes two integer parameters and returns an integer:

```
function Calc(X,Y: Integer): Integer;
```

You can assign the *Calc* function to the variable *F*:

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

If you take any procedure or function heading and remove the identifier after the word **procedure** or **function**, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the example above) or to declare new types:

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var
  F: TIntegerFunction;           { F is a parameterless function that returns an integer }
  Proc: TProcedure;             { Proc is a parameterless procedure }
  SP: TStrProc;                 { SP is a procedure that takes a string parameter }
  M: TMathFunc;                 { M is a function that takes a Double (real) parameter
                                and returns a Double }
procedure FuncProc(P: TIntegerFunction); { FuncProc is a procedure whose only parameter
                                          is a parameterless integer-valued function }
```

The variables above are all *procedure pointers*—that is, pointers to the address of a procedure or function. If you want to reference a method of an instance object (see Chapter 7, “Classes and objects”), you need to add the words **of object** to the procedural type name. For example

```
type
  TMethod = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

These types represent *method pointers*. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

```

type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    :
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent

```

we could make the following assignment.

```
OnClick := MainForm.ButtonClick;
```

Two procedural types are compatible if they have

- the same calling convention,
- the same return value (or no return value), and
- the same number of parameters, with identically typed parameters in corresponding positions. (Parameter names do not matter.)

Procedure pointer types are always incompatible with method pointer types. The value `nil` can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like *Length* as a procedural value, write a wrapper for it:

```

function FLength(S: string): Integer;
begin
  Result := Length(S);
end;

```

## Procedural types in statements and expressions

---

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```

var
  F: function(X: Integer): Integer;
  I: Integer;
function SomeFunction(X: Integer): Integer;
  :
F := SomeFunction; // assign SomeFunction to F
I := F(4);        // call function; assign result to I

```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```

var
  F, G: function: Integer;
  I: Integer;

```

## Variant types

```
function SomeFunction: Integer;
:
F := SomeFunction; // assign SomeFunction to F
G := F;           // copy F to G
I := G;           // call function; assign result to I
```

The first statement assigns a procedural value to *F*. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to *I*. Because *I* is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of *F* results in a function call; the compiler calls the function pointed to by *F*, then calls the function *MyFunction*, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where *F* references a procedure (which doesn't return a value), or where *F* references a function that requires parameters, the statement above causes a compilation error. To compare the procedural value of *F* with *MyFunction*, use

```
if @F = @MyFunction then ...;
```

@F converts *F* into an untyped pointer variable that contains an address, and @MyFunction returns the address of *MyFunction*.

To get the memory address of a procedural variable (rather than the address stored in it), use @@. For example, @@F returns the address of *F*.

The @ operator can also be used to assign an untyped pointer value to a procedural variable. For example,

```
var StrComp: function(Str1, Str2: PChar): Integer;
:
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

calls the *GetProcAddress* function and points *StrComp* to the result.

Any procedural variable can hold the value **nil**, which means that it points to nothing. But attempting to *call* a **nil**-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function *Assigned*:

```
if Assigned(OnClick) then OnClick(X);
```

## Variant types

---

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type *Variant*, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with

regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See Chapter 10, “Object interfaces”.) They can hold dynamic arrays, and they can hold a special kind of static array called a *variant array*. (See “Variant arrays” on page 5-33.) Variants can mix with other variants and with integer, real, string, and Boolean values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if *V* is a variant that holds a string value, the construction *V*[1] causes a runtime error.

You can define custom Variants that extend the Variant type to hold arbitrary values. For example, you can define a Variant string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the *TCustomVariantType* class.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value *Unassigned*. The special value *Null* indicates unknown or missing data.

The standard function *VarType* returns a variant’s type code. The *varTypeMask* constant is a bit mask used to extract the code from *VarType*’s return value, so that, for example,

```
VarType(V) and varTypeMask = varDouble
```

returns *True* if *V* contains a *Double* or an array of *Double*. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The *TVarData* record type defined in the *System* unit can be used to typecast variants and gain access to their internal representation. See the online Help on *VarType* for a list of codes, and note that new type codes may be added in future implementations of Object Pascal.

## Variant type conversions

---

All integer, real, string, character, and Boolean types are assignment-compatible with *Variant*. Expressions can be explicitly cast as variants, and the *VarAsType* and *VarCast* standard routines can be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1; { integer value }
  V2 := 1234.5678; { real value }
```

## Variant types

```

V3 := 'Hello world!'; { string value }
V4 := '1000'; { string value }
V5 := V1 + V2 + V4; { real value 2235.5678 }
I := V1; { I = 1 (integer value) }
D := V2; { D = 1234.5678 (real value) }
S := V3; { S = 'Hello world!' (string value) }
I := V4; { I = 1000 (integer value) }
S := V5; { S = '2235.5678' (string value) }
end;

```

The compiler performs type conversions according to the following rules.

**Table 5.7** Variant type conversion rules

Target Source	integer	real	string	character	Boolean
<b>integer</b>	converts integer formats	converts to real	converts to string representation	same as string (left)	returns <i>False</i> if 0, <i>True</i> otherwise
<b>real</b>	rounds to nearest integer	converts real formats	converts to string representation using regional settings	same as string (left)	returns <i>False</i> if 0, <i>True</i> otherwise
<b>string</b>	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/character formats	same as string (left)	returns <i>False</i> if string is “false” (non-case-sensitive) or a numeric string that evaluates to 0, <i>True</i> if string is “true” or a nonzero numeric string; raises exception otherwise
<b>character</b>	same as string (above)	same as string (above)	same as string (above)	same as string-to-string	same as string (above)
<b>Boolean</b>	<i>False</i> = 0, <i>True</i> = -1 (255 if <i>Byte</i> )	<i>False</i> = 0, <i>True</i> = -1	<i>False</i> = “0”, <i>True</i> = “-1”	same as string (left)	<i>False</i> = <i>False</i> , <i>True</i> = <i>True</i>
<b>Unassigned</b>	returns 0	returns 0	returns empty string	same as string (left)	returns <i>False</i>
<b>Null</b>	raises exception	raises exception	raises exception	same as string (left)	raises exception

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid assignments or casts raise the *EVariantError* exception.

Special conversion rules apply to the *TDateTime* real type declared in the *System* unit. When a *TDateTime* is converted to any other type, it is treated as a normal *Double*. When an integer, real, or Boolean is converted to a *TDateTime*, it is first converted to a *Double*, then read as a date-time value. When a string is converted to a *TDateTime*, it is interpreted as a date-time value using the regional settings. When an *Unassigned* value is converted to *TDateTime*, it is treated like the real or integer value 0. Converting a *Null* value to *TDateTime* raises an exception.

On Windows, if a variant references a COM interface, any attempt to convert it reads the object's default property and converts that value to the requested type. If the object has no default property, an exception is raised.

## Variants in expressions

---

All operators except `^`, `is`, and `in` take variant operands. Operations on variants return *Variant* values; they return *Null* if one or both operands is *Null*, and raise an exception if one or both operands is *Unassigned*. In a binary operation, if only one operand is a variant, the other is converted to a variant.

The return type of an operation is determined by its operands. In general, the same rules that apply to operands of statically bound types apply to variants. For example, if *V1* and *V2* are variants that hold an integer and a real value, then *V1* + *V2* returns a real-valued variant. (See "Operators" on page 4-6.) With variants, however, you can perform binary operations on combinations of values that would not be allowed using statically typed expressions. When possible, the compiler converts mismatched variants using the rules summarized in Table 5.7. For example, if *V3* and *V4* are variants that hold a numeric string and an integer, the expression *V3* + *V4* returns an integer-valued variant; the numeric string is converted to an integer before the operation is performed.

## Variant arrays

---

You cannot assign an ordinary static array to a variant. Instead, create a *variant array* by calling either of the standard functions *VarArrayCreate* or *VarArrayOf*. For example,

```
V: Variant;
  ⋮
  V := VarArrayCreate([0,9], varInteger);
```

creates a variant array of integers (of length 10) and assigns it to the variant *V*. The array can be indexed using *V*[0], *V*[1], and so forth, but it is not possible to pass a variant array element as a `var` parameter. Variant arrays are always indexed with integers.

The second parameter in the call to *VarArrayCreate* is the type code for the array's base type. For a list of these codes, see the online Help on *VarType*. Never pass the code *varString* to *VarArrayCreate*; to create a variant array of strings, use *varOleStr*.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except *ShortString*



and *AnsiString*, and if the base type of the array is *Variant*, its elements can even be heterogeneous. Use the *VarArrayRedim* function to resize a variant array. Other standard routines that operate on variant arrays include *VarArrayDimCount*, *VarArrayLowBound*, *VarArrayHighBound*, *VarArrayRef*, *VarArrayLock*, and *VarArrayUnlock*.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

## OleVariant

---

The *OleVariant* type exists on both the Windows and Linux platforms. The main difference between *Variant* and *OleVariant* is that *Variant* can contain data types that only the current application knows what to do with. *OleVariant* can only contain the data types defined as compatible with Ole Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Pascal string, or a one of the new custom variant types) to an *OleVariant*, the runtime library tries to convert the *Variant* into one of the *OleVariant* standard data types (such as a Pascal string converts to an Ole BSTR string). For example, if a variant containing an *AnsiString* is assigned to an *OleVariant*, the *AnsiString* becomes a *WideString*. The same is true when passing a *Variant* to an *OleVariant* function parameter.

## Type compatibility and identity

---

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include *type identity*, *type compatibility*, and *assignment-compatibility*.

### Type identity

---

Type identity is almost straightforward. When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

*T1*, *T2*, *T3*, *T4*, and *Integer* all denote the same type. To create distinct types, repeat the word **type** in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called *TMyInteger* which is not identical to *Integer*.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, *TS1* and *TS2*. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```
type MyString = string[10];
var
  S1: MyString;
  S2: MyString;
```

## Type compatibility

---

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of components.
- One is a string type and the other is a string, packed-string, or *Char* type.
- One type is *Variant* and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is *PChar* or *PWideChar* and the other is a zero-based character array of the form `array[0..n] of Char`.
- One type is *Pointer* (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the `{ST+}` compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

## Assignment-compatibility

---

Assignment-compatibility is not a symmetric relation. An expression of type *T2* can be assigned to a variable of type *T1* if the value of the expression falls in the range of *T1* and at least one of the following conditions is satisfied.

- *T1* and *T2* are of the same type, and it is not a file type or structured type that contains a file type at any level.
- *T1* and *T2* are compatible ordinal types.
- *T1* and *T2* are both real types.
- *T1* is a real type and *T2* is an integer type.
- *T1* is *PChar* or any string type and the expression is a string constant.
- *T1* and *T2* are both string types.
- *T1* is a string type and *T2* is a *Char* or packed-string type.
- *T1* is a long string and *T2* is *PChar*.
- *T1* and *T2* are compatible packed-string types.
- *T1* and *T2* are compatible set types.
- *T1* and *T2* are compatible pointer types.
- *T1* and *T2* are both class, class-reference, or interface types and *T2* is a derived from *T1*.
- *T1* is an interface type and *T2* is a class type that implements *T1*.
- *T1* is *PChar* or *PWideChar* and *T2* is a zero-based character array of the form `array[0..n]` of *Char*.
- *T1* and *T2* are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type. See “Procedural types in statements and expressions” on page 5-29.)
- *T1* is *Variant* and *T2* is an integer, real, string, character, Boolean, or interface type.
- *T1* is an integer, real, string, character, or Boolean type and *T2* is *Variant*.
- *T1* is the *IUnknown* or *IDispatch* interface type and *T2* is *Variant*. (The variant’s type code must be *varEmpty*, *varUnknown*, or *varDispatch* if *T1* is *IUnknown*, and *varEmpty* or *varDispatch* if *T1* is *IDispatch*.)

## Declaring types

---

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

```
type newTypeName = type
```

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier’s scope doesn’t include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

$X$  and  $Y$  are of the same type; at runtime, there is no way to distinguish  $TValue$  from  $Real$ . This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type information—for example, to associate a property editor with properties of a particular type—the distinction between “different name” and “different type” becomes important. In this case, use the syntax

```
type newTypeName = type type
```

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called  $TValue$ .

## Variables

---

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

### Declaring variables

---

The basic syntax for a variable declaration is

```
var identifierList: type;
```

where *identifierList* is a comma-delimited list of valid identifiers and *type* is any valid type. For example,

```
var I: Integer;
```

declares a variable  $I$  of type  $Integer$ , while

```
var X, Y: Real;
```

declares two variables— $X$  and  $Y$ —of type  $Real$ .

Consecutive variable declarations do not have to repeat the reserved word **var**:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called *local*, while other variables are called *global*. Global variables can be initialized at the same time they are declared, using the syntax

```
var identifier: type = constantExpression;
```

where *constantExpression* is any constant expression representing a value of type *type*. (For more information about constant expressions, see “Constant expressions” on page 5-41.) Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
:
I := 7;
```

Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Local variables, in contrast, cannot be initialized in their declarations and contain random data until a value is assigned to them.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see Chapter 11, "Memory management".

## Absolute addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive **absolute** after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

specifies that the variable *StrLen* should start at the same address as *Str*. Since the first byte of a short string contains the string's length, the value of *StrLen* is the length of *Str*.

You cannot initialize a variable in an **absolute** declaration or combine **absolute** with any other directives.

## Dynamic variables

You can create dynamic variables by calling the *GetMem* or *New* procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use *FreeMem* to destroy variables created by *GetMem* and *Dispose* to destroy variables created by *New*. Other standard routines that operate on dynamic variables include *ReallocMem*, *Initialize*, *StrAlloc*, and *StrDispose*.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

## Thread-local variables

*Thread-local* (or *thread*) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its

own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with **threadvar** instead of **var**. For example,

```
threadvar X: Integer;
```

Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the **absolute** directive.

Do not create pointer- or procedural-type thread variables, and do not use thread variables in dynamically loadable libraries (other than packages).

Dynamic variables that are ordinarily managed by the compiler—long strings, wide strings, dynamic arrays, variants, and interfaces—can be declared with **threadvar**, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory. For example,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  ⋮
S := ''; // free the memory used by S
```

(You can free a variant by setting it to *Unassigned* and an interface or dynamic array by setting it to **nil**.)

## Declared constants

---

Several different language constructions are referred to as “constants”. There are numeric constants (also called *numerals*) like `17`, and string constants (also called *character strings* or *string literals*) like `'Hello world!'`; for information about numeric and string constants, see Chapter 4, “Syntactic elements”. Every enumerated type defines constants that represent the values of that type. There are predefined constants like *True*, *False*, and **nil**. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

### True constants

---

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called *MaxValue* that returns the integer 237. The syntax for declaring a true constant is

```
const identifier = constantExpression
```

where *identifier* is any valid identifier and *constantExpression* is an expression that the compiler can evaluate without executing your program. (See “Constant expressions” on page 5-41 for more information.)

If *constantExpression* returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called *MyNumber*, of type *Int64*, that returns the integer 17.

Otherwise, the type of the declared constant is the type of the *constantExpression*.

- If *constantExpression* is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.
- If *constantExpression* is a real, its type is *Extended*. If it is an integer, its type is given by the table below.

**Table 5.8** Types for integer constants

Range of constant (hexadecimal)	Range of constant (decimal)	Type
-\$8000000000000000..-\$80000001	-2 <sup>63</sup> ..-2147483649	<i>Int64</i>
-\$80000000..-\$8001	-2147483648..-32769	<i>Integer</i>
-\$8000..-\$81	-32768..-129	<i>Smallint</i>
-\$80..-1	-128..-1	<i>Shortint</i>
0..\$7F	0..127	0..127
\$80..\$FF	128..255	<i>Byte</i>
\$0100..\$7FFF	256..32767	0..32767
\$8000..\$FFFF	32768..65535	<i>Word</i>
\$10000..\$7FFFFFFF	65536..2147483647	0..2147483647
\$80000000..\$FFFFFFFF	2147483648..4294967295	<i>Cardinal</i>
\$100000000..\$FFFFFFFFFFFFFFFF	4294967296..2 <sup>63</sup> -1	<i>Int64</i>

Here are some examples of constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + ' . ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

## Constant expressions

A *constant expression* is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants *True*, *False*, and *nil*; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

<i>Abs</i>	<i>High</i>	<i>Low</i>	<i>Pred</i>	<i>Succ</i>
<i>Chr</i>	<i>Length</i>	<i>Odd</i>	<i>Round</i>	<i>Swap</i>
<i>Hi</i>	<i>Lo</i>	<i>Ord</i>	<i>SizeOf</i>	<i>Trunc</i>

This definition of a *constant expression* is used in several places in Object Pascal's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing **case** statements, and declaring both true and typed constants.

Examples of constant expressions:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

## Resource strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program. For more information, see the online Help topics on localizing applications.

Resource strings are declared like other true constants, except that the word **const** is replaced by **resourcestring**. The expression to the right of the = symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';           { for explanations of format specifiers, }
  OpenError = 'Cannot open file %s';             { see 'Format strings' in the online Help }
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks\000\000';
  SomeResourceString = SomeTrueConstant;
```

The compiler automatically resolves naming conflicts among resource strings in different libraries.



## **Typed constants**

---

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

In the default **{S}-** compiler state, typed constants can not have new values assigned to them; they are, in effect, read-only variables. However, if the **{S}+** compiler directive is in effect, typed constants can have new values assigned to them; they behave essentially like initialized variables.

Declare a typed constant like this:

```
const identifier: type = value
```

where *identifier* is any valid identifier, *type* is any type except files and variants, and *value* is an expression of type *type*. For example,

```
const Max: Integer = 100;
```

In most cases, *value* must be a constant expression; but if *type* is an array, record, procedural, or pointer type, special rules apply.

## Array constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called *Digits* that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the declaration above can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

creates an array called *Maze* where

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

Array constants cannot contain file-type values at any level.

## Record constants

To declare a record constant, specify the value of each field—as *fieldName*: *value*, with the field assignments separated by semicolons—in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag

## Declared constants

field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record constants cannot contain file-type values at any level.

## Procedural constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```
function Calc(X, Y: Integer): Integer;
begin
  :
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Given these declarations, you can use the procedural constant *MyFunction* in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value **nil** to a procedural constant.

## Pointer constants

When you declare a pointer constant, you must initialize it to a value that can be resolved—at least as a relative address—at compile time. There are three ways to do this: with the **@** operator, with **nil**, and (if the constant is of type *PChar*) with a string literal. For example, if *I* is a global variable of type *Integer*, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a *PChar* constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```

Addresses of local (stack-allocated) and dynamic (heap-allocated) variables cannot be assigned to pointer constants.



## Procedures and functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A *function* is a routine that returns a value when it executes. A *procedure* is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls *SomeFunction* and assigns the result to *I*. Function calls cannot appear on the left side of an assignment statement.

Procedure calls—and, when extended syntax is enabled (`{ $\$X+$ }`), function calls—can be used as complete statements. For example,

```
DoSomething;
```

calls the *DoSomething* routine; if *DoSomething* is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

### Declaring procedures and functions

---

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the *prototype*, *heading*, or *header*. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's *body* or *block*.

The standard procedure *Exit* can occur within the body of any procedure or function. *Exit* halts execution of the routine where it occurs and immediately passes program control back to the point from which the routine was called.

## Procedure declarations

---

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;
  localDeclarations;
begin
  statements
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

- For information about the *parameterList*, see “Parameters” on page 6-11.
- For information about *directives*, see “Calling conventions” on page 6-4, “Forward and interface declarations” on page 6-5, “External declarations” on page 6-6, “Overloading procedures and functions” on page 6-8, and “Writing dynamically loadable libraries” on page 9-3. If you include more than one directive, separate them with semicolons.
- For information about *localDeclarations*, which declares local identifiers, see “Local declarations” on page 6-10.

Here is an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Given this declaration, you can call the *NumString* procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value “17” to *MyString* (which must be a **string** variable).

Within a procedure’s statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like *N* and *S* in the example above); the parameter list defines a set of local variables, so don’t try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

## Function declarations

---

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```
function functionName(parameterList): returnType; directives;
localDeclarations;
begin
    statements
end;
```

where *functionName* is any valid identifier, *returnType* is any type, *statements* is a sequence of statements that execute when the function is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

- For information about the *parameterList*, see “Parameters” on page 6-11.
- For information about *directives*, see “Calling conventions” on page 6-4, “Forward and interface declarations” on page 6-5, “External declarations” on page 6-6, “Overloading procedures and functions” on page 6-8, and “Writing dynamically loadable libraries” on page 9-3. If you include more than one directive, separate them with semicolons.
- For information about *localDeclarations*, which declares local identifiers, see “Local declarations” on page 6-10.

The function’s statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function’s return value, as does the predefined variable *Result*.

For example,

```
function WF: Integer;
begin
    WF := 17;
end;
```

defines a constant function called *WF* that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
    Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
    X: Real;
    I: Integer;
begin
    X := A[0];
```



## Declaring procedures and functions

```
    for I := 1 to N - 1 do
        if X < A[I] then X := A[I];
    Max := X;
end;
```

You can assign a value to *Result* or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to *Result* or to the function name becomes the function's return value. For example,

```
function Power(X: Real; Y: Integer): Real;
var
    I: Integer;
begin
    Result := 1.0;
    I := Y;
    while I > 0 do
        begin
            if Odd(I) then Result := Result * X;
            I := I div 2;
            X := Sqr(X);
        end;
    end;
end;
```

*Result* and the function name always represent the same value. Hence

```
function MyFunction: Integer;
begin
    MyFunction := 5;
    Result := Result * 2;
    MyFunction := Result + 1;
end;
```

returns the value 11. But *Result* is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like *Result*) to track the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. *Result*, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

As long as extended syntax is enabled (**(\$X+)**), *Result* is implicitly declared in every function. Do not try to redeclare it.

If execution terminates without an assignment being made to *Result* or the function name, then the function's return value is undefined.

## Calling conventions

---

When you declare a procedure or function, you can specify a *calling convention* using one of the directives **register**, **pascal**, **cdecl**, **stdcall**, and **safecall**. For example,

```
function MyFunction(X, Y: Real): Real; cdecl;
:
```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is **register**.

- The **register** and **pascal** conventions pass parameters from left to right; that is, the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl**, **stdcall**, and **safecall** conventions pass parameters from right to left.
- For all conventions except **cdecl**, the procedure or function removes parameters from the stack upon returning. With the **cdecl** convention, the caller removes parameters from the stack when the call returns.
- The **register** convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The **safecall** convention implements exception “firewalls.” On Windows, this implements interprocess COM error notification.

The table below summarizes calling conventions.

**Table 6.1** Calling conventions

Directive	Parameter order	Clean-up	Passes parameters in registers?
<b>register</b>	Left-to-right	Routine	Yes
<b>pascal</b>	Left-to-right	Routine	No
<b>cdecl</b>	Right-to-left	Caller	No
<b>stdcall</b>	Right-to-left	Routine	No
<b>safecall</b>	Right-to-left	Routine	No

The default **register** convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use **register**.) The **cdecl** convention is useful when you call functions from shared libraries written in C or C++, while **stdcall** and **safecall** are recommended, in general, for calls to external code. On Windows, the operating system APIs are **stdcall** and **safecall**. Other operating systems generally use **cdecl**. (Note that **stdcall** is more efficient than **cdecl**.)

The **safecall** convention must be used for declaring dual-interface methods (see Chapter 10, “Object interfaces”). The **pascal** convention is maintained for backward compatibility. For more information on calling conventions, see Chapter 12, “Program control”.

The directives **near**, **far**, and **export** refer to calling conventions in 16-bit Windows programming. They have no effect in 32-bit applications and are maintained for backward compatibility only.

## Forward and interface declarations

The **forward** directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called *Calculate*. Somewhere after the **forward** declaration, the routine must be redeclared in a *defining declaration* that includes a block. The defining declaration for *Calculate* might look like this:

```
function Calculate;
: { declarations }
begin
: { statement block }
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the **forward** declaration exactly (except that default parameters can be omitted). If the **forward** declaration specifies an overloaded procedure or function (see "Overloading procedures and functions" on page 6-8), then the defining declaration must repeat the parameter list.

Between a **forward** declaration and its defining declaration, you can place nothing except other declarations. The defining declaration can be an **external** or **assembler** declaration, but it cannot be another **forward** declaration.

The purpose of a **forward** declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the **forward**-declared routine before it is actually defined. Besides letting you organize your code more flexibly, **forward** declarations are sometimes necessary for mutual recursions.

The **forward** directive is not allowed in the **interface** section of a unit. Procedure and function headers in the **interface** section, however, behave like **forward** declarations and must have defining declarations in the **implementation** section. A routine declared in the **interface** section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

## External declarations

---

The **external** directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C++ function that takes a variable number of parameters, use the **varargs** directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The **varargs** directive works only with external routines and only with the **cdecl** calling convention.

## Linking to object files

To call routines from a separately compiled object file, first link the object file to your application using the **\$L** (or **\$LINK**) compiler directive. For example,

```
On Windows: {$L BLOCK.OBJ}
On Linux:   {$L block.o}
```

links `BLOCK.OBJ` (Windows) or `block.o` (Linux) into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the *MoveWord* and *FillWord* routines from `BLOCK.OBJ` (Windows) or `block.o` (Linux).

Declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Object Pascal source code; for more information, see Chapter 13, “Inline assembler code”.

## Importing functions from libraries

To import routines from a dynamically loadable library (.so or .DLL), attach a directive of the form

```
external stringConstant;
```

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Windows

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called *SomeFunction* from `strlib.dll`.

On Linux,

```
function SomeFunction(S: string): string; external 'strlib.so';
```

imports a function called *SomeFunction* from `strlib.so`.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the **external** directive:

```
external stringConstant1 name stringConstant2;
```

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine’s original name.

On Windows: For example, the following declaration imports a function from `user32.dll` (part of the Windows API).

```
function MessageBox(HWND: Integer; Text, Caption: PChar; Flags: Integer): Integer;
stdcall; external 'user32.dll' name 'MessageBoxA';
```

The function’s original name is *MessageBoxA*, but it is imported as *MessageBox*.

Instead of a name, you can use a number to identify the routine you want to import:

```
external stringConstant index integerConstant;
```

where *integerConstant* is the routine’s index in the export table.

On Linux: For example, the following declaration imports a standard system function from `libc.so.6`.

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
external 'libc.so.6' name 'open';
```

The function's original name is *open*, but it is imported as *OpenFile*.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

For more information about libraries, see Chapter 9, "Libraries and packages".

## Overloading procedures and functions

---

You can declare more than one routine in the same scope with the same name. This is called *overloading*. Overloaded routines must be declared with the **overload** directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;
begin
    Result := X/Y;
end;

function Divide(X, Y: Integer): Integer; overload;
begin
    Result := X div Y;
end;
```

These declarations create two functions, both called *Divide*, that take parameters of different types. When you call *Divide*, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first *Divide* function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types—for example,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type *Extended*.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```
function Cap(S: string): string; overload;
:
:
procedure Cap(var Str: string); overload;
:
:
```

But the declarations

```
function Func(X: Real; Y: Integer): Real; overload;
:
function Func(X: Integer; Y: Real): Real; overload;
:
```

are legal.

When an overloaded routine is declared in a **forward** or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain `AnsiString/PChar` and `WideString/WideChar` parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is `AnsiString/PChar`.

```
procedure test(const S: String); overload;
procedure test(const W: WideString); overload;

var
  a: string;
  b: widestring;
begin
  a := 'a';
  b := 'b';
  test(a);    // calls String version
  test(b);    // calls WideString version
  test('abc'); // calls String version
  test(WideString('abc')); // calls widestring version
end;
```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);    // integer version
  foo(v);    // variant version
  foo(1.2);  // variant version (float literals -> extended precision)
end;
```

This example calls the variant version of `foo`, not the double version, because the `1.2` constant is implicitly an extended type and extended is not an exact match for

double. Extended is also not an exact match for variant, but variant is considered a more general type (whereas double is a smaller type than extended).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```
const d: double = 1.2;
begin
  foo(d);
end;
```

The above code works correctly, and calls the double version.

```
const s: single = 1.2;
begin
  foo(s);
end;
```

The above code also calls the double version of foo. Single is a better fit to double than to variant.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (Single, Double, Extended) along with the variant version.

If you use default parameters in overloaded routines, be careful of ambiguous parameter signatures. For more information, see “Default parameters and overloaded routines” on page 6-18.

You can limit the potential effects of overloading by qualifying a routine’s name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in *Unit1*; if no routine in *Unit1* matches the name and parameter list in the call, an error results.

For information about distributing overloaded methods in a class hierarchy, see “Overloading methods” on page 7-12. For information about exporting overloaded routines from a shared library, see “The exports clause” on page 9-5.

## Local declarations

---

The body of a function or procedure often begins with declarations of local variables used in the routine’s statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

## Nested routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called *DoSomething* contains a nested procedure.

```
procedure DoSomething(S: string);
var
  X, Y: Integer;
```

```

    procedure NestedProc(S: string);
    begin
        :
    end;

begin
    :
    NestedProc(S);
    :
end;

```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, *NestedProc* can be called only within *DoSomething*.

For real examples of nested routines, look at the *DateTimeToString* procedure, the *ScanDate* function, and other routines in the *SysUtils* unit.

## Parameters

---

Most procedure and function headers include a *parameter list*. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is (X: Real; Y: Integer).

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the = symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by one of the reserved words **var**, **const**, and **out**.

Examples:

```

(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)

```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```

procedure UpdateRecords;
begin
    :
end;

```

Within the procedure or function body, the parameter names (X and Y in the first example above) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

## Parameter semantics

---

Parameters are categorized in several ways:



- Every parameter is classified as *value*, *variable*, *constant*, or *out*. Value parameters are the default; the reserved words **var**, **const**, and **out** indicate variable, constant, and out parameters, respectively.
- Value parameters are always *typed*, while constant, variable, and out parameters can be either *typed* or *untyped*.
- Special rules apply to array parameters. See “Array parameters” on page 6-15.

Files and instances of structured types that contain files can be passed only as variable (**var**) parameters.

## Value and variable parameters

Most parameters are either value parameters (the default) or variable (**var**) parameters. Value parameters are passed *by value*, while variable parameters are passed *by reference*. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;    // X is a value parameter
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X: Integer): Integer; // X is a variable parameter
begin
  X := X * 2;
  Result := X;
end;
```

These functions return the same result, but only the second one—*DoubleByRef*—can change the value of a variable passed to it. Suppose we call the functions like this:

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleByValue(I); // J = 8, I = 4
  W := DoubleByRef(V);   // W = 8, V = 8
end;
```

After this code executes, the variable *I*, which was passed to *DoubleByValue*, has the same value we initially assigned to it. But the variable *V*, which was passed to *DoubleByRef*, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more **var** parameters, no copies are made. This is illustrated in the following example.

```

procedure AddOne(var X, Y: Integer);
begin
    X := X + 1;
    Y := Y + 1;
end;

var I: Integer;
begin
    I := 1;
    AddOne(I, I);
end;

```

After this code executes, the value of *I* is 3.

If a routine's declaration specifies a **var** parameter, you must pass an assignable expression—that is, a variable, typed constant (in the **{\$J+}** state), dereferenced pointer, field, or indexed variable—to the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in **var** parameters—for example, `DoubleByRef(MyArray[I])`—are evaluated once, before execution of the routine.

## Constant parameters

A constant (**const**) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a **var** parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using **const** allows the compiler to optimize code for structured- and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the `CompareStr` function in the `SysUtils` unit:

```

function CompareStr(const S1, S2: string): Integer;

```

Because *S1* and *S2* are not modified in the body of `CompareStr`, they can be declared as constant parameters.

## Out parameters

An **out** parameter, like a variable parameter, is passed by reference. With an **out** parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The **out** parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```

procedure GetInfo(out Info: SomeRecordType);

```

When you call `GetInfo`, you must pass it a variable of type `SomeRecordType`:

```

var MyRecord: SomeRecordType;

```

```

:
  GetInfo(MyRecord);

```

But you're not using *MyRecord* to pass any data to the *GetInfo* procedure; *MyRecord* is just a container where you want *GetInfo* to store the information it generates. The call to *GetInfo* immediately frees the memory used by *MyRecord*, before program control passes to the procedure.

**Out** parameters are frequently used with distributed-object models like COM and CORBA. In addition, you should use **out** parameters when you pass an uninitialized variable to a function or procedure.

## Untyped parameters

You can omit type specifications when declaring **var**, **const**, and **out** parameters. (Value parameters must be typed.) For example,

```

procedure TakeAnything(const C);

```

declares a procedure called *TakeAnything* that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called *Equal* that compares a specified number of bytes of any two variables.

```

function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;

```

Given the declarations

```

type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;

```

you could make the following calls to *Equal*:

```

Equal(Vec1, Vec2, SizeOf(TVector))           // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N)       // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5) // compare first 5 to last 5 elements of Vec1

```

```
Equal(Vec1[1], P, 4) // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

## String parameters

---

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```
procedure Check(S: string[20]); // syntax error
```

causes a compilation error. But

```
type TString20 = string[20];
procedure Check(S: TString20);
```

is valid. The special identifier *OpenString* can be used to declare routines that take short-string parameters of varying length:

```
procedure Check(S: OpenString);
```

When the **{\$H-}** and **{\$P+}** compiler directives are both in effect, the reserved word **string** is equivalent to *OpenString* in parameter declarations.

Short strings, *OpenString*, **\$H**, and **\$P** are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

## Array parameters

---

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer); // syntax error
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. For most purposes, however, *open array parameters* are a better solution.

### Open array parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of type` (rather than `array[X..Y] of type`) in the parameter declaration. For example,

```
function Find(A: array of Char): Integer;
```

declares a function called *Find* that takes a character array of any size and returns an integer.

**Note** The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The example above creates a function that takes any array of *Char* elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;
```

```
function Find(A: TDynamicCharArray): Integer;
```

For information about dynamic arrays, see “Dynamic arrays” on page 5-19.

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard *Low* and *High* functions return 0 and *Length*-1, respectively. The *SizeOf* function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped **var** parameters. They cannot be passed to *SetLength*.
- Instead of an array, you can pass a variable of the open array parameter’s base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine’s stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a *Clear* procedure that assigns zero to each element in an array of reals and a *Sum* function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end;

function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

When you call routines that use open array parameters, you can pass *open array constructors* to them. See “Open array constructors” on page 6-19.

## Variant open array parameters

Variant open array parameters allow you to pass an array of differently-typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify **array of const** as the parameter’s type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called *DoSomething* that can operate on heterogeneous arrays.

The **array of const** construction is equivalent to **array of TVarRec**. *TVarRec*, declared in the *System* unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. *TVarRec*'s *VType* field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as *Pointer* and must be typecast to **string**. See the online Help on *TVarRec* for details.

The following example uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in *SysUtils*.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:   Result := Result + IntToStr(VInteger);
        vtBoolean:   Result := Result + BoolChars[VBoolean];
        vtChar:      Result := Result + VChar;
        vtExtended:  Result := Result + FloatToStr(VExtended^);
        vtString:    Result := Result + VString^;
        vtPChar:     Result := Result + VPChar;
        vtObject:    Result := Result + VObject.ClassName;
        vtClass:     Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:  Result := Result + CurrToStr(VCurrency^);
        vtVariant:   Result := Result + string(VVariant^);
        vtInt64:     Result := Result + IntToStr(VInt64^);
      end;
    end;
  end;
```

We can call this function using an open array constructor (see “Open array constructors” on page 6-19). For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string “test100 T3.14159TForm”.

## Default parameters

---

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed **const** and value parameters. To provide a default value, end the parameter declaration with the = symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real; // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string); // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

the statements

```
F := Resizer;
F(N);
```

result in the values ( $N$ , 1.0) being passed to *Resizer*.

Default parameters are limited to values that can be specified by a constant expression. (See “Constant expressions” on page 5-41.) Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than `nil` as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

For information about calling routines with default parameter values, see “Calling procedures and functions” on page 6-19.

## Default parameters and overloaded routines

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;
:
:
procedure Confused(I: Integer; J: Integer = 0); overload;
:
:
Confused(X); // Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

## Default parameters in forward and interface declarations

If a routine has a **forward** declaration or appears in the interface section of a unit, default parameter values—if there are any—must be specified in the **forward** or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the **forward** or interface declaration exactly.

## Calling procedures and functions

---

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called *actual parameters*, while the parameters in the routine's declaration are called *formal parameters*.

When calling a routine, remember that

- expressions used to pass typed **const** and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass **var** and **out** parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass **var** and **out** parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(, , X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();
DoSomething;
```

## Open array constructors

---

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.



For example, given the declarations

```
var I, J: Integer;  
procedure Add(A: array of Integer);
```

you could call the *Add* procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;  
:  
Temp[0] := 5;  
Temp[1] := 7;  
Temp[2] := I;  
Temp[3] := I + J;  
Add(Temp);
```

Open array constructors can be passed only as value or **const** parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

## Classes and objects

A *class*, or *class type*, defines a structure consisting of *fields*, *methods*, and *properties*. Instances of a class type are called *objects*. The fields, methods, and properties of a class are called its *components* or *members*.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class's fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects—that is, instances of a class. Some methods (called *class methods*) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have *access specifiers*, which determine how their data are read and modified. From other parts of a program—outside of the object itself—a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called *constructors* and *destructors*.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value **nil**. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the *Size* property of the object referenced by *SomeObject*; you would *not* write this as `SomeObject^.Size := 100`.

## Class types

---

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type className = class (ancestorClass)
  memberList
end;
```

where *className* is any valid identifier, (*ancestorClass*) is optional, and *memberList* declares members—that is, fields, methods, and properties—of the class. If you omit (*ancestorClass*), then the new class inherits directly from the predefined *TObject* class. If you include (*ancestorClass*) and *memberList* is empty, you can omit `end`. A class type declaration can also include a list of *interfaces* implemented by the class; see “Implementing interfaces” on page 10-4.

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the *TMemoryStream* class from the *Classes* unit.

```
type
TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
end;
```

*TMemoryStream* descends from *TStream* (in the *Classes* unit), inheriting most of its members. But it defines—or redefines—several methods and properties, including its destructor method, *Destroy*. Its constructor, *Create*, is inherited without change from *TObject*, and so is not redeclared. Each member is declared as *private*, *protected*, or *public* (this class has no *published* members); for explanations of these terms, see “Visibility of class members” on page 7-4.

Given this declaration, you can create an instance of *TMemoryStream* as follows:

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

## Inheritance and scope

---

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called *TSomeControl* that descends from *TControl*. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence *TSomeControl* contains all of the members defined in *TControl* and in each of *TControl*'s ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

## TObject and TClass

The *TObject* class, declared in the *System* unit, is the ultimate ancestor of all other classes. *TObject* defines only a handful of methods, including a basic constructor and destructor. In addition to *TObject*, the *System* unit declares the class-reference type *TClass*:

```
TClass = class of TObject;
```

For more information about *TObject*, see the online help. For more information about class-reference types, see "Class references" on page 7-23.

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from *TObject*. Thus

```
type TMyClass = class
  :
end;
```

is equivalent to

```
type TMyClass = class(TObject)
  :
end;
```

The latter form is recommended for readability.

## Compatibility of class types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TSquare = class(TRectangle);
var
  Fig: TFigure;
```

the variable *Fig* can be assigned values of type *TFigure*, *TRectangle*, and *TSquare*.

## Object types

As an alternative to class types, you can declare *object types* using the syntax

```
type objectTypeName = object (ancestorObjectType)
  memberList
end;
```

where *objectTypeName* is any valid identifier, (*ancestorObjectType*) is optional, and *memberList* declares fields, methods, and properties. If (*ancestorObjectType*) is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from *TObject*, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the *New* procedure and destroy them with the *Dispose* procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended.

## Visibility of class members

---

Every member of a class has an attribute called *visibility*, which is indicated by one of the reserved words **private**, **protected**, **public**, **published**, or **automated**. For example,

```
  published property Color: TColor read GetColor write SetColor;
```

declares a published property called *Color*. Visibility determines where and how a member can be accessed, with private representing the least accessibility, protected representing an intermediate level of accessibility, and public, published, and automated representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default published, provided the class is compiled in the **{*M+*}** state or is derived from a class compiled in the **{*M+*}** state; otherwise, such members are public.

For readability, it is best to organize a class declaration by visibility, placing all the private members together, followed by all the protected members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new "section" of the declaration. So a typical class declaration should like this:

```
type
  TMyClass = class(TControl)
  private
    : { private declarations here}
  protected
    : { protected declarations here }
  public
    : { public declarations here }
  published
    : { published declarations here }
  end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a protected property can be made public in a descendant, but not private. Moreover, published members cannot become public in a descendant class. For more information, see “Property overrides and redeclarations” on page 7-22.

### Private, protected, and public members

A *private* member is invisible outside of the unit or program where its class is declared. In other words, a private method cannot be called from another module, and a private field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another’s private members without making those members more widely accessible.

A *protected* member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. In other words, a protected method can be called, and a protected field or property read or written to, from the definition of any method belonging to a class that descends from the one where the protected member is declared. Members that are intended for use only in the implementation of derived classes are usually protected.

A *public* member is visible wherever its class can be referenced.

### Published members

*Published* members have the same visibility as public members. The difference is that *runtime type information* (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the Object Inspector, and to associate specific methods (called *event handlers*) with specific properties (called *events*).

Published properties are restricted to certain data types. Ordinal, string, class, interface, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except *Real48* can be published. Properties of an array type (as distinct from *array properties*, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties of all publishable types (see “Array properties” on page 7-19), and properties of enumerated types that include anonymous values (see “Enumerated types with explicitly assigned ordinality” on page 5-7). If you publish a property of this kind, the Object Inspector won’t display it correctly, nor will the property’s value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the **{\$M+}** state or descends from a class compiled in the **{\$M+}** state. Most classes with published members derive from *TPersistent*, which is compiled in the **{\$M+}** state, so it is seldom necessary to use the **\$M** directive.

## Automated members

*Automated* members have the same visibility as public members. The difference is that *Automation type information* (required for Automation servers) is generated for automated members. Automated members typically appear only in Windows classes and is not recommended for Linux programming. The **automated** reserved word is maintained for backward compatibility. The *TAutoObject* class in the *ComObj* unit does not use **automated**.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool*, and all interface types.
- Method declarations must use the default **register** calling convention. They can be virtual, but not dynamic.
- Property declarations can include access specifiers (**read** and **write**) but other specifiers (**index**, **stored**, **default**, and **nodefault**) are not allowed. Access specifiers must list a method identifier that uses the default **register** calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a **dispid** directive. Specifying an already used ID in a **dispid** directive causes an error.

On Windows, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Windows only), see “Automation objects (Windows only)” on page 10-10.

## Forward declarations and mutually dependent classes

---

If the declaration of a class type ends with the word **class** and a semicolon—that is, if it has the form

```
type className = class;
```

with no ancestor or class members listed after the word **class**—then it is a *forward declaration*. A forward declaration must be resolved by a *defining declaration* of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```

type
  TFigure = class; // forward declaration
  TDrawing = class
    Figure: TFigure;
    :
  end;

  TFigure = class // defining declaration
    Drawing: TDrawing;
    :
  end;

```

Do not confuse forward declarations with complete declarations of types that derive from *TObject* without declaring any class members.

```

type
  TFirstClass = class; // this is a forward declaration
  TSecondClass = class // this is a complete class declaration
    end; //
  TThirdClass = class(TObject); // this is a complete class declaration

```

## Fields

---

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. All field declarations must occur before any property or method declarations. For example, the following declaration creates a class called *TNumber* whose only member, other than the methods it inherits from *TObject*, is an integer field called *Int*.

```

type TNumber = class
  Int: Integer;
end;

```

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```

type
  TAncestor = class
    Value: Integer;
  end;

  TDescendant = class(TAncestor)
    Value: string; // hides the inherited Value field
  end;

var
  MyObject: TAncestor;

begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!'; // error
  TDescendant(MyObject).Value := 'Hello!'; // works!
end;

```



Although *MyObject* holds an instance of *TDescendant*, it is declared as *TAncestor*. The compiler therefore interprets *MyObject.Value* as referring to the (integer) field declared in *TAncestor*. Both fields, however, exist in the *TDescendant* object; the inherited *Value* is hidden by the new one, and can be accessed through a typecast.

## Methods

---

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example,

```
SomeObject.Free
```

calls the *Free* method in *SomeObject*.

## Method declarations and implementations

---

Within a class declaration, methods appear as procedure and function headings, which work like **forward** declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of *TMyClass* includes a method called *DoSomething*:

```
type
  TMyClass = class(TObject)
    :
    procedure DoSomething;
    :
  end;
```

A defining declaration for *DoSomething* must occur later in the module:

```
procedure TMyClass.DoSomething;
begin
  :
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class's methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does so, the order, type, and names of the parameters must match exactly, and, if the method is a function, so must the return value.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

*reintroduce*; *overload*; *binding*; *calling convention*; **abstract**; *warning*

where *binding* is **virtual**, **dynamic**, or **override**; *calling convention* is **register**, **pascal**, **cdecl**, **stdcall**, or **safecall**; and *warning* is **platform**, **deprecated**, or **library**.

## Inherited

The reserved word **inherited** plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If **inherited** is followed by the name of a member, it represents a normal method call or reference to a property or field—except that the search for the referenced member begins with the immediate ancestor of the enclosing method’s class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited *Create*.

When **inherited** has no identifier after it, it refers to the inherited method with the same name as the enclosing method. In this case, **inherited** takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

## Self

Within the implementation of a method, the identifier *Self* references the object in which the method is called. For example, here is the implementation of *TCollection*’s *Add* method in the *Classes* unit.

```
function TCollection.Add: TCollectionItem;
begin
    Result := FItemClass.Create(Self);
end;
```

The *Add* method calls the *Create* method in the class referenced by the *FItemClass* field, which is always a *TCollectionItem* descendant. *TCollectionItem.Create* takes a single parameter of type *TCollection*, so *Add* passes it the *TCollection* instance object where *Add* is called. This is illustrated in the following code.

```
var MyCollection: TCollection;
    :
MyCollection.Add // MyCollection is passed to the TCollectionItem.Create method
```

*Self* is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class’s methods. In this case, you can access the original member identifier as *Self.Identifier*.

For information about *Self* in class methods, see “Class methods” on page 7-25.

## Method binding

---

Methods can be *static* (the default), *virtual*, or *dynamic*. Virtual and dynamic methods can be *overridden*, and they can be *abstract*. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

### Static methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the *Draw* methods are static.

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to *Figure.Draw*, the *Figure* variable references an object of class *TRectangle*, but the call invokes the implementation of *Draw* in *TFigure*, because the declared type of the *Figure* variable is *TFigure*.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw; // calls TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw; // calls TFigure.Draw
  TRectangle(Figure).Draw; // calls TRectangle.Draw
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw; // calls TRectangle.Draw
  Rectangle.Destroy;
end;
```

### Virtual and dynamic methods

To make a method virtual or dynamic, include the **virtual** or **dynamic** directive in its declaration. Virtual and dynamic methods, unlike static methods, can be *overridden* in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method call—not the declared type of the variable—determines which implementation to activate.

To override a method, redeclare it with the **override** directive. An **override** declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the *Draw* method declared in *TFigure* is overridden in two descendant classes.

```

type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;

```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```

var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw; // calls TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw; // calls TEllipse.Draw
  Figure.Destroy;
end;

```

Only virtual and dynamic methods can be overridden. All methods, however, can be *overloaded*; see “Overloading methods”.

### Virtual versus dynamic

Virtual and dynamic methods are semantically equivalent. They differ only in the implementation of method-call dispatching at runtime. Virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

### Overriding versus hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn’t include **override**, the new declaration merely *hides* the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```

type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act; // Act is redeclared, but not overridden
  end;

```

```

var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act; // calls T1.Act
end;

```

## Reintroduce

The **reintroduce** directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```

procedure DoSomething; reintroduce; // the ancestor class also has a DoSomething method

```

Use **reintroduce** when you want to hide an inherited virtual method with a new one.

## Abstract methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive **abstract** after **virtual** or **dynamic**. For example,

```

procedure DoSomething; virtual; abstract;

```

You can call an abstract method only in a class or instance of a class in which the method has been overridden.

## Overloading methods

---

A method can be redeclared using the **overload** directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the **reintroduce** directive when you redeclare it in descendant classes. For example,

```

type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;
  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
  :
  SomeObject := T2.Create;
  SomeObject.Test('Hello!'); // calls T2.Test
  SomeObject.Test(7);       // calls T1.Test

```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```

type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer // error
    :

```

Methods that serve as property **read** or **write** specifiers cannot be overloaded.

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see “Overloading procedures and functions” on page 6-8.

## Constructors

---

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word **constructor**. Examples:

```

constructor Create;
constructor Create(AOwner: TComponent);

```

Constructors must use the default **register** calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor *Create*.

To create an object, call the constructor method in a class type. For example,

```

MyObject := TMyClass.Create;

```

This allocates storage for the new object on the heap, sets the values of all ordinal fields to zero, assigns **nil** to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the *Destroy* destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor’s implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word **inherited** to execute an inherited constructor.

Here is an example of a class type and its constructor.

```

type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    :
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner); // Initialize inherited parts
  Width := 65; // Change inherited properties
  Height := 65;
  FPen := TPen.Create; // Initialize new fields
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;

```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), **nil** (pointer and class types), empty (string types), or *Unassigned* (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as **virtual** is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects—that is, construction of objects whose types aren't known at compile time. (See "Class references" on page 7-23.)

## Destructors

---

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word **destructor**. Examples:

```

destructor Destroy;
destructor Destroy; override;

```

Destructors must use the default **register** calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited *Destroy* method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, *Destroy* is automatically called to dispose of the unfinished object. This means that *Destroy* must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always **nil**. A destructor should therefore check for **nil** values before operating on class-type or pointer-type fields. Calling the *Free* method (defined in *TObject*), rather than *Destroy*, offers a convenient way of checking for **nil** values before destroying an object.

## Message methods

---

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. The VCL uses message methods to respond to Windows messages. CLX does not use message methods to respond to system events.

A message method is created by including the **message** directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Windows message IDs defined, along with corresponding record types, in the *Messages* unit. A message method must be a procedure that takes a single **var** parameter.

For example, on Windows:

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    :
  end;
```

For example, on Linux or for cross-platform programming, you would handle messages as follows:



```

const
  ID_REFRESH = $0001;

type
  TTextBox = class(TCustomControl)
  private
    procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
    :
  end;

```

A message method does not have to include the **override** directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

## Implementing message methods

The implementation of a message method can call the inherited message method, as in this example (for Windows):

```

procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;

```

On Linux or for cross-platform programming, you would write the same example as follows:

```

procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
  else
    inherited;
end;

```

The **inherited** statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, **inherited** calls the *DefaultHandler* method originally defined in *TObject*.

The implementation of *DefaultHandler* in *TObject* simply returns without performing any actions. By overriding *DefaultHandler*, a class can implement its own default handling of messages. On Windows, the *DefaultHandler* method for VCL controls calls the Windows *DefWindowProc* function.

## Message dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the *Dispatch* method inherited from *TObject*:

```

procedure Dispatch(var Message);

```

The *Message* parameter passed to *Dispatch* must be a record whose first entry is a field of type *Cardinal* containing a message ID.

*Dispatch* searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, *Dispatch* calls *DefaultHandler*.

## Properties

---

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

```
property propertyName[indexes]: type index integerConstant specifiers;
```

where

- *propertyName* is any valid identifier.
- [*indexes*] is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier*<sub>1</sub>, ..., *identifier*<sub>n</sub>: *type*. For more information, see "Array properties" on page 7-19.
- *type* must be a predefined or previously declared data type. That is, property declarations like `property Num: 0..9 ...` are invalid.
- the *index integerConstant* clause is optional. For more information, see "Index specifiers" on page 7-20.
- *specifiers* is a sequence of **read**, **write**, **stored**, **default** (or **nodefault**), and **implements** specifiers. Every property declaration must have at least one **read** or **write** specifier. (For information about **implements**, see "Implementing interfaces by delegation" on page 10-6.)

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as **var** parameters, nor can the @ operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a **read** method that retrieves a value from a database or generates a random value.

### Property access

---

Every property has a **read** specifier, a **write** specifier, or both. These are called *access specifiers* and they have the form

```
read fieldOrMethod
write fieldOrMethod
```

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.
- If *fieldOrMethod* is a method, it cannot be overloaded. Moreover, access methods for a published property must use the default **register** calling convention.
- In a **read** specifier, if *fieldOrMethod* is a method, it must be a parameterless function whose result type is the same as the property's type.
- In a **write** specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or **const** parameter of the same type as the property.

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the *GetColor* method must be declared as

```
function GetColor: TColor;
```

and the *SetColor* method must be declared as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(The name of *SetColor*'s parameter, of course, doesn't have to be *Value*.)

When a property is referenced in an expression, its value is read using the field or method listed in the **read** specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the **write** specifier.

The example below declares a class called *TCompass* with a published property called *Heading*. The value of *Heading* is read through the *FHeading* field and written through the *SetHeading* procedure.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    :
  end;
```

Given this declaration, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the *TCompass* class, no action is associated with reading the *Heading* property; the **read** operation consists of retrieving the value stored in the *FHeading* field. On the other hand, assigning a value to the *Heading* property translates into a call to the *SetHeading* method, which, presumably, stores the new value in the *FHeading* field as well as performing other actions. For example, *SetHeading* might be implemented like this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint; // update user interface to reflect new value
  end;
end;
```

A property whose declaration includes only a **read** specifier is a *read-only* property, and one whose declaration includes only a **write** specifier is a *write-only* property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

## Array properties

---

*Array properties* are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a **read** specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a **write** specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or **const** parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
```

```

function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);

```

An array property is accessed by indexing the property identifier. For example, the statements

```

if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';

```

correspond to

```

if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');

```

On Linux, you would use a path such as `‘/usr/local/bin’` in place of `‘C:\DELPHI\BIN’` in the above example.

The definition of an array property can be followed by the **default** directive, in which case the array property becomes the default property of the class. For example,

```

type
  TStringArray = class
    public
      property Strings[Index: Integer]: string ...; default;
      :
    end;

```

If a class has a default property, you can access that property with the abbreviation *object[index]*, which is equivalent to *object.property[index]*. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always determines an object’s default property statically.

## Index specifiers

---

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive **index** followed by an integer constant between `-2147483647` and `2147483647`. If a property has an index specifier, its **read** and **write** specifiers must list methods rather than fields. For example,

```

type
  TRectangle = class
    private
      FCoordinates: array[0..3] of Longint;
      function GetCoordinate(Index: Integer): Longint;
      procedure SetCoordinate(Index: Integer; Value: Longint);
    public
      property Left: Longint index 0 read GetCoordinate write SetCoordinate;
      property Top: Longint index 1 read GetCoordinate write SetCoordinate;

```

```

property Right: Longint index 2 read GetCoordinate write SetCoordinate;
property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
:
end;

```

An access method for a property with an index specifier must take an extra value parameter of type *Integer*. For a **read** function, it must be the last parameter; for a **write** procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if *Rectangle* is of type *TRectangle*, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

## Storage specifiers

---

The optional **stored**, **default**, and **nodefault** directives are called *storage specifiers*. They have no effect on program behavior, but control the way runtime type information (RTTI) is maintained. Specifically, storage specifiers determine whether or not to save the values of published properties in form files.

The **stored** directive must be followed by *True*, *False*, the name of a *Boolean* field, or the name of a parameterless method that returns a *Boolean* value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no **stored** directive, it is treated as if `stored True` were specified.

The **default** directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited **default** value without specifying a new one, use the **nodefault** directive. The **default** and **nodefault** directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without **default** or **nodefault**, it is treated as if **nodefault** were specified. For reals, pointers, and strings, there is an implicit **default** value of 0, **nil**, and '' (the empty string), respectively.

When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its **default** value (or if there is no **default** value) and the **stored** specifier is *True*, then the property's value is saved. Otherwise, the property's value is not saved.

**Note** Storage specifiers are not supported for array properties. The **default** directive has a different meaning when used in an array property declaration. See "Array properties" on page 7-19.

## Property overrides and redeclarations

---

A property declaration that doesn't specify a type is called a *property override*. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word **property** followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as protected, a derived class can redeclare it in a public or published section of the class. Property overrides can include **read**, **write**, **stored**, **default**, and **nodefault** directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an **implements** directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```

type
  TAncestor = class
  :
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
  :
  end;
type
  TDerived = class(TAncestor)
  :
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
  :
  end;

```

The override of *Size* adds a **write** specifier to allow the property to be modified. The overrides of *Text* and *Color* change the visibility of the properties from protected to published. The property override of *Color* also specifies that the property should be filed if its value isn't *clBlue*.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always *static*. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to *MyObject.Value* invokes *Method1* or *Method2*, even though *MyObject* holds an instance of *TDescendant*. But you

can cast *MyObject* to *TDescendant* to access the descendant class's properties and their access specifiers.

```

type
  TAncestor = class
    :
    property Value: Integer read Method1 write Method2;
  end;

  TDescendant = class(TAncestor)
    :
    property Value: Integer read Method3 write Method4;
  end;

var MyObject: TAncestor;
    :
    MyObject := TDescendant.Create;

```

## Class references

---

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

### Class-reference types

---

A class-reference type, sometimes called a *metaclass*, is denoted by a construction of the form

```
class of type
```

where *type* is any class type. The identifier *type* itself denotes a value whose type is class of *type*. If *type*<sub>1</sub> is an ancestor of *type*<sub>2</sub>, then class of *type*<sub>2</sub> is assignment-compatible with class of *type*<sub>1</sub>. Thus

```

type TClass = class of TObject;
var AnyObj: TClass;

```

declares a variable called *AnyObj* that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value **nil** to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for *TCollection* (in the *Classes* unit):

```

type TCollectionItemClass = class of TCollectionItem;
    :
    constructor Create(ItemClass: TCollectionItemClass);

```

This declaration says that to create a *TCollection* instance object, you must pass to the constructor the name of a class descending from *TCollectionItem*.



Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

## Constructors and class references

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```

type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;

```

The *CreateControl* function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to *CreateControl* can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

## Class operators

---

Every class inherits from *TObject* methods called *ClassType* and *ClassParent* that return, respectively, a reference to the class of an object and of an object's immediate ancestor. Both methods return a value of type *TClass* (where *TClass* = class of *TObject*), which can be cast to a more specific type. Every class also inherits a method called *InheritsFrom* that tests whether the object where it is called descends from a specified class. These methods are used by the **is** and **as** operators, and it is seldom necessary to call them directly.

### The is operator

The **is** operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression

*object is class*

returns *True* if *object* is an instance of the class denoted by *class* or one of its descendants, and *False* otherwise. (If *object* is **nil**, the result is *False*.) If the declared

type of *object* is unrelated to *class*—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to *TEdit* after first verifying that the object it references is an instance of *TEdit* or one of its descendants.

## The as operator

The **as** operator performs checked typecasts. The expression

```
object as class
```

returns a reference to the same object as *object*, but with the type given by *class*. At runtime, *object* must be an instance of the class denoted by *class* or one of its descendants, or be **nil**; otherwise an exception is raised. If the declared type of *object* is unrelated to *class*—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results. For example,

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

The rules of operator precedence often require **as** typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```

## Class methods

---

A class method is a method (other than a constructor) that operates on classes instead of objects. The definition of a class method must begin with the reserved word **class**. For example,

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    :
  end;
```

The defining declaration of a class method must also begin with **class**. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  :
end;
```

In the defining declaration of a class method, the identifier *Self* represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class *C*, then *Self* is of the type `class of C`. Thus

you cannot use *Self* to access fields, properties, and normal (object) methods, but you can use it to call constructors and other class methods.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of *Self*.

## Exceptions

---

An *exception* is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an *exception handler*, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the *SysUtils* unit, all runtime errors are automatically converted into exceptions. Errors that would otherwise terminate an application—such as insufficient memory, division by zero, and general protection faults—can be caught and handled.

### When to use exceptions

---

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The complexity of Object Pascal's exception-handling mechanism, however, makes it inefficient, and it should therefore be used judiciously. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a **try...except** or **try...finally** statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in **if...then** statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
  AssignFile(F, FileName);
  Reset(F); // raises an EInOutError exception if file is not found
except
  on Exception do ...
end;
```

But you could also avoid the overhead of exception handling by using

```
if FileExists(FileName) then // returns False if file is not found; raises no exception
begin
```

```

    AssignFile(F, FileName);
    Reset(F);
end;

```

*Assertions* provide another way of testing a Boolean condition anywhere in your source code. When an *Assert* statement fails, the program either halts or (if it uses the *SysUtils* unit) raises an *EAssertionFailed* exception. Assertions should be used only to test for conditions that you do not expect to occur. For more information, see the online Help for the standard procedure *Assert*.

## Declaring exception types

---

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the *Exception* class defined in *SysUtils*.

You can group exceptions into families using inheritance. For example, the following declarations in *SysUtils* define a family of exception types for math errors.

```

type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);

```

Given these declarations, you can define a single *EMathError* exception handler that also handles *EInvalidOp*, *EZeroDivide*, *EOverflow*, and *EUnderflow*.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```

type EInOutError = class(Exception)
  ErrorCode: Integer;
end;

```

## Raising and handling exceptions

---

To create an exception object, call the exception class's constructor within a **raise** statement. For example,

```

raise EMathError.Create;

```

In general, the form of a **raise** statement is

```

raise object at address

```

where *object* and *at address* are both optional. If *object* is omitted, the statement re-raises the current exception; see "Re-raising exceptions" on page 7-30. When an *address* is specified, it is usually a pointer to a procedure or function; use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is *raised*—that is, referenced in a **raise** statement—it is governed by special exception-handling logic. A **raise** statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose **try...except** block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an *ERangeError* exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S); // StrToInt is declared in SysUtils
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

Notice the *CreateFmt* method called in the **raise** statement. *Exception* and its descendants have special constructors that provide alternative ways to create exception messages and context IDs. See the online Help for details.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

**Note** Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the *SysUtils* unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units—including *SysUtils*—are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program.

## Try...except statements

Exceptions are handled within **try...except** statements. For example,

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide *Y* by *Z*, but calls a routine named *HandleZeroDivide* if an *EZeroDivide* exception is raised.

The syntax of a **try...except** statement is

```
try statements except exceptionBlock end
```

where *statements* is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

```
else statements
```

An exception handler has the form

```
on identifier: type do statement
```

where *identifier*: is optional (if included, *identifier* can be any valid identifier), *type* is a type used to represent exceptions, and *statement* is any statement.

A **try...except** statement executes the statements in the initial *statements* list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial *statements* list, either by a **raise** statement in the *statements* list or by a procedure or function called from the *statements* list, an attempt is made to “handle” the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler “matches” an exception just in case the *type* in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the *statement* in the **else** clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered **try...except** statement that has not yet exited. If no appropriate handler, **else** clause, or statement list is found there, the search propagates to the next-most-recently entered **try...except** statement, and so forth. If the outermost **try...except** statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the **try...except** statement where the handling occurs, and control is transferred to the executed exception handler, **else** clause, or statement list. This process discards all procedure and function calls that occurred after entering the **try...except** statement where the exception is handled. The exception object is then automatically destroyed through a call to its *Destroy* destructor and control is passed to the statement following the **try...except** statement. (If a call to the *Exit*, *Break*, or *Continue* standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. *EMathError* appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
:
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows **on...do**. The scope of the identifier is limited to that statement. For example,

```
try
:
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an **else** clause, the **else** clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
:
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** clause handles any exception that isn't an *EMathError*.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try
:
except
  HandleException;
end;
```

Here, the *HandleException* routine handles any exception that occurs as a result of executing the statements between **try** and **except**.

## Re-raising exceptions

When the reserved word **raise** occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then *re-raise* the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the *GetFileList* function allocates a *TStringList* object and fills it with file names matching a specified search path:

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
```

```

while I = 0 do
begin
  Result.Add(SearchRec.Name);
  I := FindNext(SearchRec);
end;
except
  Result.Free;
  raise;
end;
end;

```

*GetFileList* creates a *TStringList* object, then uses the *FindFirst* and *FindNext* functions (defined in *SysUtils*) to initialize it. If the initialization fails—for example because the search path is invalid, or because there is not enough memory to fill in the string list—*GetFileList* needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a **try...except** statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

## Nested exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the *Tan* function below.

```

type
  ETrigError = class(EMathError);

function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
    end;
  end;
end;

```

If an *EMathError* exception occurs during execution of *Tan*, the exception handler raises an *ETrigError*. Since *Tan* does not provide a handler for *ETrigError*, the exception propagates beyond the original exception handler, causing the *EMathError* exception to be destroyed. To the caller, it appears as if the *Tan* function has raised an *ETrigError* exception.

## Try...finally statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a **try...finally** statement.



The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```

Reset(F);
try
  : // process file F
finally
  CloseFile(F);
end;

```

The syntax of a **try...finally** statement is

```

try statementList1 finally statementList2 end

```

where each *statementList* is a sequence of statements delimited by semicolons. The **try...finally** statement executes the statements in *statementList*<sub>1</sub> (the **try** clause). If *statementList*<sub>1</sub> finishes without raising exceptions, *statementList*<sub>2</sub> (the **finally** clause) is executed. If an exception is raised during execution of *statementList*<sub>1</sub>, control is transferred to *statementList*<sub>2</sub>; once *statementList*<sub>2</sub> finishes executing, the exception is re-raised. If a call to the *Exit*, *Break*, or *Continue* procedure causes control to leave *statementList*<sub>1</sub>, *statementList*<sub>2</sub> is automatically executed. Thus the **finally** clause is always executed, regardless of how the **try** clause terminates.

If an exception is raised but not handled in the **finally** clause, that exception is propagated out of the **try...finally** statement, and any exception already raised in the **try** clause is lost. The **finally** clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

## Standard exception classes and routines

---

The *SysUtils* unit declares several standard routines for handling exceptions, including *ExceptObject*, *ExceptAddr*, and *ShowException*. *SysUtils* and other units also include dozens of exception classes, all of which (aside from *OutlineError*) derive from *Exception*.

The *Exception* class has properties called *Message* and *HelpContext* that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways. See the online Help for details.

## Standard routines and I/O

This chapter discusses text and file I/O and summarizes standard library routines. Many of the procedures and functions listed here are defined in the *System* unit, which is implicitly compiled with every application. Others are built into the compiler but are treated as if they were in the *System* unit.

Some standard routines are in units such as *SysUtils*, which must be listed in a **uses** clause to make them available in programs. You cannot, however, list *System* in a **uses** clause, nor should you modify the *System* unit or try to rebuild it explicitly.

For more information about the routines listed here, see the online Help.

### File input and output

The table below lists input and output routines.

**Table 8.1** Input and output procedures and functions

Procedure or function	Description
<i>Append</i>	Opens an existing text file for appending.
<i>AssignFile</i>	Assigns the name of an external file to a file variable.
<i>BlockRead</i>	Reads one or more records from an untyped file.
<i>BlockWrite</i>	Writes one or more records into an untyped file.
<i>ChDir</i>	Changes the current directory.
<i>CloseFile</i>	Closes an open file.
<i>Eof</i>	Returns the end-of-file status of a file.
<i>Eoln</i>	Returns the end-of-line status of a text file.
<i>Erase</i>	Erases an external file.
<i>FilePos</i>	Returns the current file position of a typed or untyped file.
<i>FileSize</i>	Returns the current size of a file; not used for text files.

**Table 8.1** Input and output procedures and functions (continued)

Procedure or function	Description
<i>Flush</i>	Flushes the buffer of an output text file.
<i>GetDir</i>	Returns the current directory of a specified drive.
<i>IOResult</i>	Returns an integer value that is the status of the last I/O function performed.
<i>MkDir</i>	Creates a subdirectory.
<i>Read</i>	Reads one or more values from a file into one or more variables.
<i>Readln</i>	Does what <i>Read</i> does and then skips to beginning of next line in the text file.
<i>Rename</i>	Renames an external file.
<i>Reset</i>	Opens an existing file.
<i>Rewrite</i>	Creates and opens a new file.
<i>RmDir</i>	Removes an empty subdirectory.
<i>Seek</i>	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
<i>SeekEof</i>	Returns the end-of-file status of a text file.
<i>SeekEoln</i>	Returns the end-of-line status of a text file.
<i>SetTextBuf</i>	Assigns an I/O buffer to a text file.
<i>Truncate</i>	Truncates a typed or untyped file at the current file position.
<i>Write</i>	Writes one or more values to a file.
<i>Writeln</i>	Does the same as <i>Write</i> , and then writes an end-of-line marker to the text file.

A file variable is any variable whose type is a file type. There are three classes of file: *typed*, *text*, and *untyped*. The syntax for declaring file types is given in “File types” on page 5-24.

Before a file variable can be used, it must be associated with an external file through a call to the *AssignFile* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be “opened” to prepare it for input or output. An existing file can be opened via the *Reset* procedure, and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write*, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure *CloseFile*. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{SI+}** and **{SI-}** compiler directives. When I/O checking is off—that is, when a procedure or function call is compiled in the **{SI-}** state—an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function *IOResult* instead.

You must call the *IOResult* function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{SI+}** is the current state, the next I/O function call will fail with the lingering *IOResult* error.

## Text files

---

This section summarizes I/O using file variables of the standard type *Text*.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character). The type *Text* is distinct from the type *file* of *Char*.

For text files, there are special forms of *Read* and *Write* that let you read and write values that are not of type *Char*. Such values are automatically translated to and from their character representation. For example, *Read(F, I)*, where *I* is a type *Integer* variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in *I*.

There are two standard text-file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable *Output* is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, *Input* and *Output* are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

**Note** Text-oriented I/O is available only in console applications—that is, applications compiled with the “Generate console application” option checked on the Linker page of the Project Options dialog box or with the **-cc** command-line compiler option. In a GUI (non-console) application, any attempt to read or write using *Input* or *Output* will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* is assumed by default, depending on whether the procedure or function is

input- or output-oriented. For example, `Read(X)` corresponds to `Read(Input, X)` and `Write(X)` corresponds to `Write(Output, X)`.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using *AssignFile*, and opened using *Reset*, *Rewrite*, or *Append*. An exception is raised if you pass a file that was opened with *Reset* to an output-oriented procedure or function. An exception is also raised if you pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

## Untyped files

---

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word **file** and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the *Reset* and *Rewrite* procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for *Read* and *Write*, all typed-file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

## Text-file device drivers

---

You can define your own text-file device drivers for your programs. A text-file device driver is a set of four functions that completely implement an interface between Object Pascal's file system and some device.

The four functions that define each device driver are *Open*, *InOut*, *Flush*, and *Close*. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where *DeviceFunc* is the name of the function (that is, *Open*, *InOut*, *Flush*, or *Close*). For information about the *TTextRec* type, see the online Help. The return value of a device-interface function becomes the value returned by *IOResult*. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized *Assign* procedure. The *Assign* procedure must assign the addresses of the four device-interface functions to the four function pointers in the text-file variable. In addition, it should store the *fmClosed* "magic" constant in the *Mode* field, store the size of the text-file buffer in *BufSize*, store a pointer to the text-file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device-interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```

procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
    begin
      Mode := fmClosed;
      BufSize := SizeOf(Buffer);
      BufPtr := @Buffer;
      OpenFunc := @DevOpen;
      InOutFunc := @DevInOut;
      FlushFunc := @DevFlush;
      CloseFunc := @DevClose;
      Name[0] := #0;
    end;
end;

```

The device-interface functions can use the *UserData* field in the file record to store private information. This field isn't modified by the product file system at any time.

## Device functions

---

The functions that make up a text-file device driver are described below.

### The Open function

The *Open* function is called by the *Reset*, *Rewrite*, and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput*, *fmOutput*, or *fmInOut* to indicate whether the *Open* function was called from *Reset*, *Rewrite*, or *Append*.

The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutput* before *Open* returns.

*Open* is always called before any of the other device-interface functions. For that reason, *AssignDev* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode*, *Open* can then install pointers to either input- or output-oriented functions. This saves the *InOut*, *Flush* functions and the *CloseFile* procedure from determining the current mode.

### The InOut function

The *InOut* function is called by the *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln*, and *CloseFile* standard routines whenever input or output from the device is required.

When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into *BufPtr*<sup>^</sup>, and returns the number of characters read in *BufEnd*. In addition, it stores zero in *BufPos*. If the *InOut* function returns zero in *BufEnd* as a result of an input request, *Eof* becomes *True* for the file.

When *Mode* is *fmOutput*, the *InOut* function writes *BufPos* characters from *BufPtr*<sup>^</sup>, and returns zero in *BufPos*.

## The Flush function

The *Flush* function is called at the end of each *Read*, *Readln*, *Write*, and *Writeln*. It can optionally flush the text-file buffer.

If *Mode* is *fmInput*, the *Flush* function can store zero in *BufPos* and *BufEnd* to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

## The Close function

The *Close* function is called by the *CloseFile* standard procedure to close a text file associated with a device. (The *Reset*, *Rewrite*, and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutput*, then before calling *Close*, the file system calls the *InOut* function to ensure that all characters have been written to the device.

# Handling null-terminated strings

---

Object Pascal's extended syntax allows the *Read*, *Readln*, *Str*, and *Val* standard procedures to be applied to zero-based character arrays, and allows the *Write*, *Writeln*, *Val*, *AssignFile*, and *Rename* standard procedures to be applied to both zero-based character arrays and character pointers. In addition, the following functions are provided for handling null-terminated strings. For more information about null-terminated strings, see "Working with null-terminated strings" on page 5-13.

**Table 8.2** Null-terminated string functions

Function	Description
<i>StrAlloc</i>	Allocates a character buffer of a given size on the heap.
<i>StrBufSize</i>	Returns the size of a character buffer allocated using <i>StrAlloc</i> or <i>StrNew</i> .
<i>StrCat</i>	Concatenates two strings.
<i>StrComp</i>	Compares two strings.
<i>StrCopy</i>	Copies a string.
<i>StrDispose</i>	Disposes a character buffer allocated using <i>StrAlloc</i> or <i>StrNew</i> .
<i>StrECopy</i>	Copies a string and returns a pointer to the end of the string.
<i>StrEnd</i>	Returns a pointer to the end of a string.
<i>StrFmt</i>	Formats one or more values into a string.
<i>StrIComp</i>	Compares two strings without case sensitivity.
<i>StrLCat</i>	Concatenates two strings with a given maximum length of the resulting string.
<i>StrLComp</i>	Compares two strings for a given maximum length.
<i>StrLCopy</i>	Copies a string up to a given maximum length.
<i>StrLen</i>	Returns the length of a string.
<i>StrLFmt</i>	Formats one or more values into a string with a given maximum length.

**Table 8.2** Null-terminated string functions (continued)

Function	Description
<i>StrLComp</i>	Compares two strings for a given maximum length without case sensitivity.
<i>StrLower</i>	Converts a string to lowercase.
<i>StrMove</i>	Moves a block of characters from one string to another.
<i>StrNew</i>	Allocates a string on the heap.
<i>StrPCopy</i>	Copies a Pascal string to a null-terminated string.
<i>StrPLCopy</i>	Copies a Pascal string to a null-terminated string with a given maximum length.
<i>StrPos</i>	Returns a pointer to the first occurrence of a given substring within a string.
<i>StrRScan</i>	Returns a pointer to the last occurrence of a given character within a string.
<i>StrScan</i>	Returns a pointer to the first occurrence of a given character within a string.
<i>StrUpper</i>	Converts a string to uppercase.

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with *Ansi-*. For example, the multibyte version of *StrPos* is *AnsiStrPos*. Multibyte character support is operating-system dependent and based on the current locale.

## Wide-character strings

The *System* unit provides three functions, *WideCharToString*, *WideCharLenToString*, and *StringToWideChar*, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

For more information about wide-character strings, see “About extended character sets” on page 5-13.

## Other standard routines

The table below lists frequently used procedures and functions found in Borland product libraries. This is not an exhaustive inventory of standard routines. For more information about these and other routines, see the online Help.

**Table 8.3** Other standard routines

Procedure or function	Description
<i>Abort</i>	Ends the process without reporting an error.
<i>Addr</i>	Returns a pointer to a specified object.
<i>AllocMem</i>	Allocates a memory block and initializes each byte to zero.
<i>ArcTan</i>	Calculates the arctangent of the given number.
<i>Assert</i>	Tests whether a boolean expression is <i>True</i> .
<i>Assigned</i>	Tests for a <b>nil</b> (unassigned) pointer or procedural variable.
<i>Beep</i>	Generates a standard beep using the computer speaker.



**Table 8.3** Other standard routines (continued)

Procedure or function	Description
<i>Break</i>	Causes control to exit a <b>for</b> , <b>while</b> , or <b>repeat</b> statement.
<i>ByteToCharIndex</i>	Returns the position of the character containing a specified byte in a string.
<i>Chr</i>	Returns the character for a specified value.
<i>Close</i>	Terminates the association between a file variable and an external file.
<i>CompareMem</i>	Performs a binary comparison of two memory images.
<i>CompareStr</i>	Compares strings case sensitively.
<i>CompareText</i>	Compares strings by ordinal value and is not case sensitive.
<i>Continue</i>	Returns control to the next iteration of <b>for</b> , <b>while</b> , or <b>repeat</b> statements.
<i>Copy</i>	Returns a substring of a string or a segment of a dynamic array.
<i>Cos</i>	Calculates the cosine of an angle.
<i>CurrToStr</i>	Converts a currency variable to a string.
<i>Date</i>	Returns the current date.
<i>DateTimeToStr</i>	Converts a variable of type <i>TDateTime</i> to a string.
<i>DateToStr</i>	Converts a variable of type <i>TDateTime</i> to a string.
<i>Dec</i>	Decrements an ordinal variable.
<i>Dispose</i>	Releases memory allocated for a dynamic variable.
<i>ExceptAddr</i>	Returns the address at which the current exception was raised.
<i>Exit</i>	Exits from the current procedure.
<i>Exp</i>	Calculates the exponential of X.
<i>FillChar</i>	Fills contiguous bytes with a specified value.
<i>Finalize</i>	Uninitializes a dynamically allocated variable.
<i>FloatToStr</i>	Converts a floating point value to a string.
<i>FloatToStrF</i>	Converts a floating point value to a string, using specified format.
<i>FmtLoadStr</i>	Returns formatted output using a resourced format string.
<i>FmtStr</i>	Assembles a formatted string from a series of arrays.
<i>Format</i>	Assembles a string from a format string and a series of arrays.
<i>FormatDateTime</i>	Formats a date-and-time value.
<i>FormatFloat</i>	Formats a floating point value.
<i>FreeMem</i>	Disposes of a dynamic variable.
<i>GetMem</i>	Creates a dynamic variable and a pointer to the address of the block.
<i>GetParentForm</i>	Returns the form or property page that contains a specified control.
<i>Halt</i>	Initiates abnormal termination of a program.
<i>Hi</i>	Returns the high-order byte of an expression as an unsigned value.
<i>High</i>	Returns the highest value in the range of a type, array, or string.
<i>Inc</i>	Increments an ordinal variable.
<i>Initialize</i>	Initializes a dynamically allocated variable.
<i>Insert</i>	Inserts a substring at a specified point in a string.
<i>Int</i>	Returns the integer part of a real number.
<i>IntToStr</i>	Converts an integer to a string.

**Table 8.3** Other standard routines (continued)

<b>Procedure or function</b>	<b>Description</b>
<i>Length</i>	Returns the length of a string or array.
<i>Lo</i>	Returns the low-order byte of an expression as an unsigned value.
<i>Low</i>	Returns the lowest value in the range of a type, array, or string.
<i>LowerCase</i>	Converts an ASCII string to lowercase.
<i>MaxIntValue</i>	Returns the largest signed value in an integer array.
<i>MaxValue</i>	Returns the largest signed value in an array.
<i>MinIntValue</i>	Returns the smallest signed value in an integer array.
<i>MinValue</i>	Returns smallest signed value in an array.
<i>New</i>	Creates a new dynamic variable and references it with a specified pointer.
<i>Now</i>	Returns the current date and time.
<i>Ord</i>	Returns the ordinal value of an ordinal-type expression.
<i>Pos</i>	Returns the index of the first character of a specified substring in a string.
<i>Pred</i>	Returns the predecessor of an ordinal value.
<i>Ptr</i>	Converts a specified address to a pointer.
<i>Random</i>	Generates random numbers within a specified range.
<i>ReallocMem</i>	Reallocates a dynamic variable.
<i>Round</i>	Returns the value of a real rounded to the nearest whole number.
<i>SetLength</i>	Sets the dynamic length of a string variable or array.
<i>SetString</i>	Sets the contents and length of the given string.
<i>ShowException</i>	Displays an exception message with its address.
<i>ShowMessage</i>	Displays a message box with an unformatted string and an OK button.
<i>ShowMessageFmt</i>	Displays a message box with a formatted string and an OK button.
<i>Sin</i>	Returns the sine of an angle in radians.
<i>SizeOf</i>	Returns the number of bytes occupied by a variable or type.
<i>Sqr</i>	Returns the square of a number.
<i>Sqrt</i>	Returns the square root of a number.
<i>Str</i>	Formats a string and returns it to a variable.
<i>StrToCurr</i>	Converts a string to a currency value.
<i>StrToDate</i>	Converts a string to a date format ( <i>TDateTime</i> ).
<i>StrToDateTime</i>	Converts a string to a <i>TDateTime</i> .
<i>StrToFloat</i>	Converts a string to a floating-point value.
<i>StrToInt</i>	Converts a string to an integer.
<i>StrToTime</i>	Converts a string to a time format ( <i>TDateTime</i> ).
<i>StrUpper</i>	Returns a string in upper case.
<i>Succ</i>	Returns the successor of an ordinal value.
<i>Sum</i>	Returns the sum of the elements from an array.
<i>Time</i>	Returns the current time.
<i>TimeToStr</i>	Converts a variable of type <i>TDateTime</i> to a string.
<i>Trunc</i>	Truncates a real number to an integer.

**Table 8.3** Other standard routines (continued)

<b>Procedure or function</b>	<b>Description</b>
<i>UniqueString</i>	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
<i>UpCase</i>	Converts a character to uppercase.
<i>UpperCase</i>	Returns a string in uppercase.
<i>VarArrayCreate</i>	Creates a variant array.
<i>VarArrayDimCount</i>	Returns number of dimensions of a variant array.
<i>VarARrayHighBound</i>	Returns high bound for a dimension in a variant array.
<i>VarArrayLock</i>	Locks a variant array and returns a pointer to the data.
<i>VarArrayLowBound</i>	Returns the low bound of a dimension in a variant array.
<i>VarArrayOf</i>	Creates and fills a one-dimensional variant array.
<i>VarArrayRedim</i>	Resizes a variant array.
<i>VarArrayRef</i>	Returns a reference to the passed variant array.
<i>VarArrayUnlock</i>	Unlocks a variant array.
<i>VarAsType</i>	Converts a variant to specified type.
<i>VarCast</i>	Converts a variant to a specified type, storing the result in a variable.
<i>VarClear</i>	Clears a variant.
<i>VarCopy</i>	Copies a variant.
<i>VarToStr</i>	Converts variant to string.
<i>VarType</i>	Returns type code of specified variant.

For information on format strings, see “Format strings” in the online Help.

## Special topics

The chapters in Part II cover specialized language features and advanced topics. These chapters include:

- Chapter 9, “Libraries and packages”
- Chapter 10, “Object interfaces”
- Chapter 11, “Memory management”
- Chapter 12, “Program control”
- Chapter 13, “Inline assembler code”



# Libraries and packages

A dynamically loadable library is a dynamic-link library (DLL) on Windows or a shared object library file on Linux. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

To distinguish them from standalone executables, on Windows files containing compiled DLLs are named with the .DLL extension. On Linux, files containing shared object files are named with a .so extension. Object Pascal programs can call DLLs or shared objects written in other languages, and applications written in other languages can call DLLs or shared objects written in Object Pascal.

## Calling dynamically loadable libraries

---

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in a shared object, you must *import* them. This can be done in two ways: by declaring an **external** procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

Object Pascal does not support importing of variables from shared libraries.

### Static loading

The simplest way to import a procedure or function is to declare it using the **external** directive. For example,

```
On Windows: procedure DoSomething; external 'MYLIB.DLL';
```

```
On Linux: procedure DoSomething; external 'mylib.so';
```

If you include this declaration in a program, MYLIB.DLL (Windows) or mylib.so (Linux) is loaded once, when the program starts. Throughout execution of the program, the identifier *DoSomething* always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect **external** declarations into a separate “import unit” that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

For more information about **external** declarations, see “External declarations” on page 6-6.

## Dynamic loading

You can access routines in a library through direct calls to OS library functions, including *LoadLibrary*, *FreeLibrary*, and *GetProcAddress*. In Windows, these functions are declared in *Windows.pas*; on Linux, they are implemented for compatibility in *SysUtils.pas*; the actual Linux OS routines are *dlopen*, *dlclose*, and *dlsym* (all declared in Kylix’s *Libc* unit; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example, on Windows or Linux:

```

uses Windows, ...; {On Linux, replace Windows with SysUtils }

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  :
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;

```

When you import routines this way, the library is not loaded until the code containing the call to *LoadLibrary* executes. The library is later unloaded by the call to *FreeLibrary*. This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.

This same example can also be written on Linux as follows:

```

uses Libc, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Pointer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  :
begin
  Handle := dlopen('datetime.so', RTLD_LAZY);
  if Handle <> 0 then
  begin
    @GetTime := dlsym(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
      end;
    end;
  end;
end;

```

In this case, when importing routines, the shared object is not loaded until the code containing the call to *dlopen* executes. The shared object is later unloaded by the call to *dlclose*. This also allows you to conserve memory and to run your program even when some of the shared objects it uses are not present.

## Writing dynamically loadable libraries

---

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word **library** (instead of **program**).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, *Min* and *Max*.



```

library MinMax;

function Min(X, Y: Integer): Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min,
  Max;

begin
end.

```

If you want your library to be available to applications written in other languages, it's safest to specify **stdcall** in the declarations of exported functions. Other languages may not support Object Pascal's default **register** calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a **uses** clause, an **exports** clause, and the initialization code. For example,

```

library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors,
  DoneEditors name Done,
  InsertText name Insert,
  DeleteSelection name Delete,
  FormatSelection,
  PrintSelection name Print,
  :
  SetErrorHandler;

begin
  InitLibrary;
end.

```

You can put **exports** clauses in the **interface** or **implementation** section of a unit. Any library that includes such a unit in its **uses** clause automatically exports the routines listed the unit's **exports** clauses—without the need for an **exports** clause of its own.

The directive **local**, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the **local** directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for standalone procedures and functions, but not for methods. A routine declared with **local**—for example,

```

function Contraband(I: Integer): Integer; local;

```

—does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the **interface** section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the *caller* sets up EBX.

## The exports clause

---

A routine is exported when it is listed in an **exports** clause, which has the form

```
exports entry1, ..., entryn;
```

where each *entry* consists of the name of a procedure, function, or variable (which must be declared prior to the **exports** clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional **name** specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive **resident**, which is maintained for backward compatibility and is ignored by the compiler.)

On Windows only, an **index** specifier consists of the directive **index** followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no **index** specifier, the routine is automatically assigned a number in the export table.

**Note** Use of **index** specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A **name** specifier consists of the directive **name** followed by a string constant. If an entry has no **name** specifier, the routine is exported under its original declared name, with the same spelling and case. Use a **name** clause when you want to export a routine under a different name. For example,

```
exports
  DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the **exports** clause. For example,

```
exports
  Divide(X, Y: Integer) name 'Divide_Ints',
  Divide(X, Y: Real) name 'Divide_Reals';
```

On Windows, do not include **index** specifiers in entries for overloaded routines.

An **exports** clause can appear anywhere and any number of times in the declaration part of a program or library, or in the **interface** or **implementation** section of a unit. Programs seldom contain an **exports** clause.

## Library initialization code

---

The statements in a library's block constitute the library's *initialization code*. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization

code can also install an exit procedure using the *ExitProc* variable, as described in “Exit procedures” on page 12-4; the exit procedure executes when the library is unloaded.

Library initialization code can signal an error by setting the *ExitCode* variable to a nonzero value. *ExitCode* is declared in the *System* unit and defaults to zero, indicating successful initialization. If a library’s initialization code sets *ExitCode* to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an exit procedure.

```
library Test;

var
  SaveExit: Pointer;

procedure LibExit;
begin
  : // library exit code
  ExitProc := SaveExit; // restore exit procedure chain
end;

begin
  : // library initialization code
  SaveExit := ExitProc; // save exit procedure chain
  ExitProc := @LibExit; // install LibExit exit procedure
end.
```

When a library is unloaded, its exit procedures are executed by repeated calls to the address stored in *ExitProc*, until *ExitProc* becomes **nil**. The initialization parts of all units used by a library are executed before the library’s initialization code, and the finalization parts of those units are executed after the library’s exit procedure.

## Global variables in a library

---

Global variables declared in a shared library cannot be imported by an Object Pascal application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries—or multiple instances of a library—to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

## Libraries and system variables

---

Several variables declared in the *System* unit are of special interest to those programming libraries. Use *IsLibrary* to determine whether code is executing in an application or in a library; *IsLibrary* is always *False* in an application and *True* in a library. During a library’s lifetime, *HInstance* contains its instance handle. *CmdLine* is always **nil** in a library.

The *DLLProc* variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. *DLLProc* is available on both Windows and Linux but its use differs on each. On Windows, *DLLProc* is used in multithreading applications; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior. (See “The finalization section” on page 3-5.)

To monitor operating-system calls, create a callback procedure that takes a single integer parameter—for example,

```
procedure DLLHandler(Reason: Integer);
```

—and assign the address of the procedure to the *DLLProc* variable. When the procedure is called, it passes to it one of the following values.

<i>DLL_PROCESS_DETACH</i>	Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to <i>FreeLibrary</i> or ( <i>dlclose</i> on Linux).
<i>DLL_THREAD_ATTACH</i>	Indicates that the current process is creating a new thread (Windows only).
<i>DLL_THREAD_DETACH</i>	Indicates that a thread is exiting cleanly (Windows only).

On Linux, these are defined in the *Libc* unit.

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

## Exceptions and runtime errors in libraries

---

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Object Pascal, the exception can be handled through a normal **try...except** statement.

**Note** Under Linux this is only possible if the library and application have both been built with the same set of (base) runtime packages (which contains the EH code) or if both link to *ShareExcept*.

If the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code `$0EEDFACE`. The first entry in the *ExceptionInformation* array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Object Pascal exception object.

Generally, you should not let exceptions escape from your library. On Windows, Delphi exceptions map to the OS exception model; Linux does not have an exception model.

If a library does not use the *SysUtils* unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates.

Because the library has no way of knowing whether it was called from an Object Pascal program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

### Shared-memory manager (Windows only)

---

On Windows, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the *ShareMem* unit. The same is true if one application or DLL allocates memory with *New* or *GetMem* which is deallocated by a call to *Dispose* or *FreeMem* in another module. *ShareMem* should always be the first unit listed in any program or library **uses** clause where it occurs.

*ShareMem* is the interface unit for the BORLANDMM.DLL memory manager, which allows modules to share dynamically allocated memory. BORLANDMM.DLL must be deployed with applications and DLLs that use *ShareMem*. When an application or DLL uses *ShareMem*, its memory manager is replaced by the memory manager in BORLANDMM.DLL.

Linux uses glibc's *malloc* to manage shared memory.

## Packages

---

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange when code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

*Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their **requires** clauses.

To distinguish them from other libraries, packages are stored in files:

- On Windows, package files end with the .bpl (Borland package library) extension.
- On Linux, packages generally begin with the prefix bpl and have a .so extension.

Ordinarily, packages are loaded statically when an applications starts. But you can use the *LoadPackage* and *UnloadPackage* routines (in the *SysUtils* unit) to load packages dynamically.

**Note** When an application utilizes packages, the name of each packaged unit still must appear in the **uses** clause of any source file that references it. For more information about packages, see the online Help.

## Package declarations and source files

---

Each package is declared in a separate source file, which should be saved with the .dpc extension to avoid confusion with other files containing Object Pascal code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains

- A *name* for the package.
- A list of other packages *required* by the new package. These are packages to which the new package is linked.
- A list of unit files *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

```
package packageName;
  requiresClause;
  containsClause;
end.
```

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the *DATAX* package.

```
package DATAX;
requires
  baseclx,
  visualclx;
contains Db, DBLocal, DBXpress, ... ;
end.
```

The **requires** clause lists other, external packages used by the package being declared. It consists of the directive **requires**, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a **requires** clause.

The **contains** clause identifies the unit files to be compiled and bound into the package. It consists of the directive **contains**, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas'; // Windows
contains MyUnit in '\home\developer\MyProject\MyUnit.pas'; // Linux
```

**Note** Thread-local variables (declared with **threadvar**) in a packaged unit cannot be accessed from clients that use the package.

## Naming packages

A compiled package involves several generated files. For example, the source file for the package called *DATAx* is *DATAx.dpk*, from which the compiler generates an executable and a binary image called

- On Windows: *DATAx.bpl* and *DATAx.dcp*
- On Linux: *bplDATAx.so* and *DATAx.dcp*.

*DATAx* is used to refer to the package in the **requires** clauses of other packages, or when using the package in an application. Package names must be unique within a project.

## The requires clause

The **requires** clause lists other, external packages that are used by the current package. It functions like the **uses** clause in a unit file. An external package listed in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's **requires** clause. If the other packages are omitted from the **requires** clause, the compiler loads the referenced units from their *.dcu* (Windows) or *.dpu* (Linux) files.

## Avoiding circular package references

Packages cannot contain circular references in their **requires** clauses. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package *A* requires package *B*, then package *B* cannot require package *A*; if package *A* requires package *B* and package *B* requires package *C*, then package *C* cannot require package *A*.

## Duplicate package references

The compiler ignores duplicate references in a package's **requires** clause. For programming clarity and readability, however, duplicate references should be removed.

## The contains clause

The **contains** clause identifies the unit files to be bound into the package. Do not include file-name extensions in the **contains** clause.

## Avoiding redundant source code uses

A package cannot be listed in the **contains** clause of another package or the **uses** clause of a unit.

All units included directly in a package's **contains** clause, or indirectly in the **uses** clauses of those units, are bound into the package at compile time. The units

contained (directly or indirectly) in a package cannot be contained in any other packages referenced in **requires** clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

## Compiling packages

---

Packages are ordinarily compiled from the IDE using .dpk files generated by the Package editor. You can also compile .dpk files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

### Generated files

The following table lists the files produced by the successful compilation of a package.

**Table 9.1** Compiled package files

File extension	Contents
dcp	A binary image containing a package header and the concatenation of all dcu (Windows) or dpu (Linux) files in the package. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file.
dcu (Windows) dpu (Linux)	A binary image for a unit file contained in a package. One dcu or dpu file is created, when necessary, for each unit file.
.bpl on Windows bpl<package>.so on Linux	The runtime package. This file is a shared library with special Borland-specific features. The base name for the package is the base name of the dpk source file.

Several compiler directives and command-line switches are available to support package compilation.

### Package-specific compiler directives

The following table lists package-specific compiler directives that can be inserted into source code. See the online Help for details.

**Table 9.2** Package-specific compiler directives

Directive	Purpose
{\$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{\$G-} or {\$IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{\$WEAKPACKAGEUNIT ON}	Packages unit “weakly”, as explained in the online Help.



**Table 9.2** Package-specific compiler directives (continued)

Directive	Purpose
{ <code>\$DENYPACKAGEUNIT ON</code> }	Prevents unit from being placed in a package.
{ <code>\$DESIGNONLY ON</code> }	Compiles the package for installation in the IDE. (Put in .dpc file.)
{ <code>\$RUNONLY ON</code> }	Compiles the package as runtime only. (Put in .dpc file.)

Including {`$DENYPACKAGEUNIT ON`} in source code prevents the unit file from being packaged. Including {`$G-`} or {`$IMPORTEDDATA OFF`} may prevent a package from being used in the same application with other packages.

Other compiler directives may be included, if appropriate, in package source code.

### Package-specific command-line compiler switches

The following package-specific switches are available for the command-line compiler. See the online Help for details.

**Table 9.3** Package-specific command-line compiler switches

Switch	Purpose
<code>-\$G-</code>	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
<code>-LE path</code>	Specifies the directory where the compiled package file will be placed.
<code>-LN path</code>	Specifies the directory where the package dcp file will be placed.
<code>-LUpackageName [;packageName2;...]</code>	Specifies additional runtime packages to use in an application. Used when compiling a project.
<code>-Z</code>	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the `-$G-` switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

## Object interfaces

An *object interface*—or simply *interface*—defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface’s methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models. Custom objects built that support interfaces can interact with objects written in C++, Java, and other languages.

### Interface types

---

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
    ['{GUID}']
    memberList
end;
```

where (*ancestorInterface*) and ['{GUID}'] are optional. In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property **read** and **write** specifiers must be methods.

- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as **default**.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as **virtual**, **dynamic**, **abstract**, or **override**. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-c000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

In some interface declarations, the **interface** reserved word is replaced by **dispinterface**. This construction (along with the **dispid**, **readonly**, and **writeonly** directives) is platform-specific and is not used in Linux programming.

## Interface and inheritance

---

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not *implement* methods. What an interface inherits is the *obligation* to implement methods—an obligation that devolves onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of *IInterface*, which is defined in the *System* unit and is the ultimate ancestor of all other interfaces. *IInterface* declares three methods: *QueryInterface*, *\_AddRef*, and *\_Release*.

**Note** *IInterface* is equivalent to *IUnknown*. You should generally use *IInterface* for platform independent applications and reserve the use of *IUnknown* for specific programs that include Windows dependencies.

*QueryInterface* provides the means to move freely among the different interfaces that an object supports. *\_AddRef* and *\_Release* provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the *System* unit's *TInterfacedObject*. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects (Windows only), however, must be managed through *\_AddRef* and *\_Release*.

## Interface identification

---

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

```
[ '{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}' ]
```

where each *x* is a hexadecimal digit (0 through 9 or A through F). On Windows, the Type Library editor automatically generates GUIDs for new interfaces; you can also generate GUIDs by pressing *Ctrl+Shift+G* in the Code editor (on Linux, you must use *Ctrl+Shift+G*).

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations. (See “Interface querying” on page 10-10.)

The *TGUID* and *PGUID* types, declared in the *System* unit, are used to manipulate GUIDs.

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

When you declare a typed constant of type *TGUID*, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type *TGUID*. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

*Supports* can be called in either of two ways:

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

## Calling conventions for interfaces

---

The default calling convention is **register**, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with **stdcall**. Use **safecall** to implement CORBA interfaces. On Windows, you can use **safecall** to implement methods of dual interfaces (as described in “Dual interfaces (Windows only)” on page 10-13).

For more information about calling conventions, see “Calling conventions” on page 6-4.

## Interface properties

---

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled. For example, on Windows, COM objects do not have properties.

In an interface, property **read** and **write** specifiers must be methods, since fields are not available.

## Forward declarations

---

An interface declaration that ends with the reserved word **interface** and a semicolon, without specifying an ancestor, GUID, or member list, is a *forward declaration*. A forward declaration must be resolved by a *defining declaration* of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    :
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    :
  end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive *IWindow* from *IControl* and also derive *IControl* from *IWindow*.

## Implementing interfaces

---

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor. Such declarations have the form

```
type className = class (ancestorClass, interface1, ..., interfacen)
  memberList
end;
```

For example,

```
type
  TMemoryManager = class (TInterfacedObject, IMalloc, IErrorInfo)
    :
  end;
```

declares a class called *TMemoryManager* that implements the *IMalloc* and *IErrorInfo* interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the declaration of *TInterfacedObject* in the *System* unit.

```

type
TInterfacedObject = class(TObject, IInterface)
protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
end;

```

*TInterfacedObject* implements the *IInterface* interface. Hence *TInterfacedObject* declares and implements each of *IInterface*'s three methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares *TMemoryManager* as a direct descendent of *TInterfacedObject*.) Since every interface inherits from *IInterface*, a class that implements interfaces must implement the *QueryInterface*, *\_AddRef*, and *\_Release* methods. The *System* unit's *TInterfacedObject* implements these methods and is thus a convenient base from which to derive other classes that implement interfaces.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

## Method resolution clauses

---

You can override the default name-based mappings by including *method resolution clauses* in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

```
procedure interface.interfaceMethod = implementingMethod;
```

or

```
function interface.interfaceMethod = implementingMethod;
```

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, TErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    :
  end;
```

maps *IMalloc's Alloc* and *Free* methods onto *TMemoryManager's Allocate* and *Deallocate* methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

## Changing inherited implementations

---

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class's declaration. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    :
  end;

  TWindow = class(TInterfacedObject, IWindow) // TWindow implements IWindow
    procedure Draw;
    :
  end;

  TFrameWindow = class(TWindow, IWindow) // TFrameWindow reimplements IWindow
    procedure Draw;
    :
  end;
```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

## Implementing interfaces by delegation

---

The **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

declares a property called *MyInterface* that implements the interface *IMyInterface*.

The **implements** directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a **read** specifier. If the property uses a **read** method, that method must use the default **register** calling convention and cannot be dynamic (though it can be virtual) or specify the **message** directive.

**Note** The class you use to implement the delegated interface should derive from *TAggregatedObject*.

## Delegating to an interface-type property

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the **implements** directive, and which does so without method resolution clauses. For example,

```

type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;

var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ... // some object whose class implements IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;

```

## Delegating to a class-type property

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```

type
  IMyInterface = interface
    procedure P1;
    procedure P2;

```



```

end;
TMyImplClass = class
  procedure P1;
  procedure P2;
end;
TMyClass = class(TInterfacedObject, IMyInterface)
  FMyImplClass: TMyImplClass;
  property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
  procedure IMyInterface.P1 = MyP1;
  procedure MyP1;
end;
procedure TMyImplClass.P1;
:
:
procedure TMyImplClass.P2;
:
:
procedure TMyClass.MyP1;
:
:
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;      // calls TMyClass.MyP1;
  MyInterface.P2;     // calls TImplClass.P2;
end;

```

## Interface references

---

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. Such variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following limitations.

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```

type
  IAncestor = interface
  end;

  IDescendant = interface(IAncestor)
    procedure P1;
  end;

  TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;

```

```

    procedure P2;
end;
:
var
  D: IDescendant;
  A: IAncestor;
begin
  D := TSomething.Create; // works!
  A := TSomething.Create; // error
  D.P1; // works!
  D.P2; // error
end;

```

In this example,

- *A* is declared as a variable of type *IAncestor*. Because *TSomething* does not list *IAncestor* among the interfaces it implements, a *TSomething* instance cannot be assigned to *A*. But if we changed *TSomething*'s declaration to

```

  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
  :

```

the first error would become a valid assignment.

- *D* is declared as a variable of type *IDescendant*. While *D* references an instance of *TSomething*, we cannot use it to access *TSomething*'s *P2* method, since *P2* is not a method of *IDescendant*. But if we changed *D*'s declaration to

```

  D: TSomething;

```

the second error would become a valid method call.

Interface references are managed through reference-counting, which depends on the `_AddRef` and `_Release` methods inherited from *IInterface*. When an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope.

Global interface-type variables can be initialized only to `nil`.

To determine whether an interface-type expression references an object, pass it to the standard function *Assigned*.

## Interface assignment-compatibility

---

A class type is assignment-compatible with any interface type implemented by the class. An interface type is assignment-compatible with any ancestor interface type. The value `nil` can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type *IDispatch* or a descendant, the variant receives the type code *varDispatch*. Otherwise, the variant receives the type code *varUnknown*.

A variant whose type code is *varEmpty*, *varUnknown*, or *varDispatch* can be assigned to an *IInterface* variable. A variant whose type code is *varEmpty* or *varDispatch* can be assigned to an *IDispatch* variable.

## Interface typecasts

---

Interface types follow the same rules as class types in variable and value typecasts. Class-type expressions can be cast to interface types—for example, `IMyInterface(SomeObject)`—provided the class implements the interface.

An interface-type expression can be cast to *Variant*. If the interface is of type *IDispatch* or a descendant, the resulting variant has the type code *varDispatch*. Otherwise, the resulting variant has the type code *varUnknown*.

A variant whose type code is *varEmpty*, *varUnknown*, or *varDispatch* can be cast to *IInterface*. A variant whose type code is *varEmpty* or *varDispatch* can be cast to *IDispatch*.

## Interface querying

You can use the `as` operator to perform checked interface typecasts. This is known as *interface querying*, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

*object as interface*

where *object* is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and *interface* is any interface declared with a GUID.

An interface query returns `nil` if *object* is `nil`. Otherwise, it passes the GUID of *interface* to the *QueryInterface* method in *object*, raising an exception unless *QueryInterface* returns zero. If *QueryInterface* returns zero (indicating that *object*'s class implements *interface*), the interface query returns an interface reference to *object*.

## Automation objects (Windows only)

---

An object whose class implements the *IDispatch* interface (declared in the *System* unit) is an Automation object. Automation is available on Windows only.

## Dispatch interface types (Windows only)

---

Dispatch interface types define the methods and properties that an Automation object implements through *IDispatch*. Calls to methods of a dispatch interface are routed through *IDispatch*'s *Invoke* method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

where ['{GUID}'] is optional and *memberList* consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
end;
```

### Dispatch interface methods (Windows only)

Methods of a dispatch interface are prototypes for calls to the *Invoke* method of the underlying *IDispatch* implementation. To specify an Automation dispatch ID for a method, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.

A method declared in a dispatch interface cannot contain directives other than **dispid**. Parameter and result types must be automatable—that is, they must be *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool*, or any interface type.

### Dispatch interface properties

Properties of a dispatch interface do not include access specifiers. They can be declared as **readonly** or **writeonly**. To specify a dispatch ID for a property, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as **default**. No other directives are allowed in dispatch-interface property declarations.

## Accessing Automation objects (Windows only)

---

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include *ComObj* in the **uses** clause of one of your units or your program or library.

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The *CreateOleObject* function (defined in *ComObj*) returns an *IDispatch* reference to an Automation object and is assignment-compatible with the variant *Word*.

## Automation objects (Windows only)

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of *varByte* are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the *VarArrayLock* and *VarArrayUnlock* routines.

### Automation object method-call syntax

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both *positional* and *named* parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the `:=` symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type *Byte*, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *AnsiString*, *WordBool*, or *Variant*. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

## Dual interfaces (Windows only)

---

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from *IDispatch*.

All methods of a dual interface (except from those inherited from *IInterface* and *IDispatch*) must use the **safecall** convention, and all method parameter and result types must be automatable. (The automatable types are *Byte*, *Currency*, *Real*, *Double*, *Real48*, *Integer*, *Single*, *Smallint*, *AnsiString*, *ShortString*, *TDateTime*, *Variant*, *OleVariant*, and *WordBool*.)



# Memory management

This chapter explains how programs use memory and describes the internal formats of Object Pascal data types.

## The memory manager (Windows only)

---

**Note** Linux uses glibc functions such as *malloc* for memory management. For information, refer to the *malloc* man page on your Linux system.

On Windows systems, the memory manager manages all dynamic memory allocations and deallocations in an application. The *New*, *Dispose*, *GetMem*, *ReallocMem*, and *FreeMem* standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

On Windows, the memory manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of *GlobalAlloc*, *LocalAlloc*, and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the memory manager interfaces directly with the Win32 virtual memory API (the *VirtualAlloc* and *VirtualFree* functions). The memory manager reserves memory from the operating system in 1-MB sections of address space, and commits memory as required in 16-KB increments. It decommits and releases unused memory in 16-KB and 1-MB sections. For smaller blocks, committed memory is further suballocated.

Memory manager blocks are always rounded upward to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.



The memory manager maintains two status variables, *AllocMemCount* and *AllocMemSize*, which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging.

The *System* unit provides two procedures, *GetMemoryManager* and *SetMemoryManager*, that allow applications to intercept low-level memory manager calls. The *System* unit also provides a function called *GetHeapStatus* that returns a record containing detailed memory-manager status information. For further information about these routines, see the online Help.

## Variables

---

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

**Note** On Linux, stack size is set by the environment only.

On Windows, an application's stack is defined by two values: the *minimum stack size* and the *maximum stack size*. The values are controlled through the `$MINSTACKSIZE` and `$MAXSTACKSIZE` compiler directives, and default to 16,384 (16K) and 1,048,576 (1M) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Windows application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an *EStackOverflow* exception is raised. (Stack overflow checking is completely automatic. The `$S` compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

On Windows or Linux, dynamic variables created with the *GetMem* or *New* procedure are heap-allocated and persist until they are deallocated with *FreeMem* or *Dispose*.

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

## Internal data formats

---

The following sections describe the internal formats of Object Pascal data types.

## Integer types

---

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range  $-128..127$  (*Shortint*), the variable is stored as a signed byte.
- If both bounds are within the range  $0..255$  (*Byte*), the variable is stored as an unsigned byte.
- If both bounds are within the range  $-32768..32767$  (*Smallint*), the variable is stored as a signed word.
- If both bounds are within the range  $0..65535$  (*Word*), the variable is stored as an unsigned word.
- If both bounds are within the range  $-2147483648..2147483647$  (*Longint*), the variable is stored as a signed double word.
- If both bounds are within the range  $0..4294967295$  (*Longword*), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (*Int64*).

## Character types

---

A *Char*, an *AnsiChar*, or a subrange of a *Char* type is stored as an unsigned byte. A *WideChar* is stored as an unsigned word.

## Boolean types

---

A *Boolean* type is stored as a *Byte*, a *ByteBool* is stored as a *Byte*, a *WordBool* type is stored as a *Word*, and a *LongBool* is stored as a *Longint*.

A *Boolean* can assume the values 0 (*False*) and 1 (*True*). *ByteBool*, *WordBool*, and *LongBool* types can assume the values 0 (*False*) or nonzero (*True*).

## Enumerated types

---

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the **{SZ1}** state (the default). If an enumerated type has more than 256 values, or if the type was declared in the **{SZ2}** state, it is stored as an unsigned word. If an enumerated type is declared in the **{SZ4}** state, it is stored as an unsigned double-word.

## Real types

---

The real types store the binary representation of a sign (+ or -), an *exponent*, and a *significand*. A real value has the form

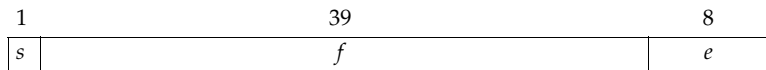
$$+/- \text{significand} * 2^{\text{exponent}}$$

where the *significand* has a single bit to the left of the binary decimal point. (That is,  $0 <= \text{significand} < 2$ .)

In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses. For example, for a *Real48* value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

### The Real48 type

A 6-byte (48-bit) *Real48* number is divided into three fields:



If  $0 < e <= 255$ , the value *v* of the number is given by

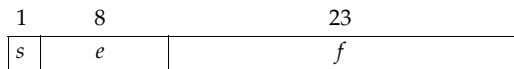
$$v = (-1)^s * 2^{(e-129)} * (1.f)$$

If  $e = 0$ , then  $v = 0$ .

The *Real48* type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a *Real48*, while NaNs and infinities produce an overflow error if an attempt is made to store them in a *Real48*.

### The Single type

A 4-byte (32-bit) *Single* number is divided into three fields:



The value *v* of the number is given by

$$\text{if } 0 < e < 255, \text{ then } v = (-1)^s * 2^{(e-127)} * (1.f)$$

$$\text{if } e = 0 \text{ and } f <> 0, \text{ then } v = (-1)^s * 2^{(-126)} * (0.f)$$

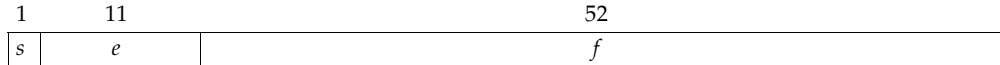
$$\text{if } e = 0 \text{ and } f = 0, \text{ then } v = (-1)^s * 0$$

$$\text{if } e = 255 \text{ and } f = 0, \text{ then } v = (-1)^s * \text{Inf}$$

$$\text{if } e = 255 \text{ and } f <> 0, \text{ then } v \text{ is a NaN}$$

## The Double type

An 8-byte (64-bit) *Double* number is divided into three fields:



The value  $v$  of the number is given by

$$\text{if } 0 < e < 2047, \text{ then } v = (-1)^s * 2^{(e-1023)} * (1.f)$$

$$\text{if } e = 0 \text{ and } f <> 0, \text{ then } v = (-1)^s * 2^{(-1022)} * (0.f)$$

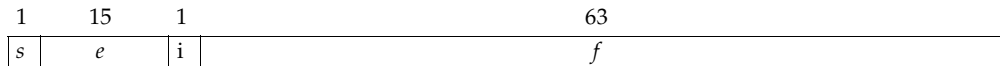
$$\text{if } e = 0 \text{ and } f = 0, \text{ then } v = (-1)^s * 0$$

$$\text{if } e = 2047 \text{ and } f = 0, \text{ then } v = (-1)^s * \text{Inf}$$

$$\text{if } e = 2047 \text{ and } f <> 0, \text{ then } v \text{ is a NaN}$$

## The Extended type

A 10-byte (80-bit) *Extended* number is divided into four fields:



The value  $v$  of the number is given by

$$\text{if } 0 \leq e < 32767, \text{ then } v = (-1)^s * 2^{(e-16383)} * (i.f)$$

$$\text{if } e = 32767 \text{ and } f = 0, \text{ then } v = (-1)^s * \text{Inf}$$

$$\text{if } e = 32767 \text{ and } f <> 0, \text{ then } v \text{ is a NaN}$$

## The Comp type

An 8-byte (64-bit) *Comp* number is stored as a signed 64-bit integer.

## The Currency type

An 8-byte (64-bit) *Currency* number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

## Pointer types

---

A *Pointer* type is stored in 4 bytes as a 32-bit address. The pointer value **nil** is stored as zero.

## Short string types

---

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (**string[255]**).

## Long string types

---

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

**Table 11.1** Long string dynamic memory layout

Offset	Contents
-8	32-bit reference-count
-4	length in bytes
0.. <i>Length</i> - 1	character string
<i>Length</i>	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of -1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of -1.

## Wide string types

---

**Note** On Linux, wide strings are implemented exactly as long strings.

On Windows, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide-string memory block on Windows.

**Table 11.2** Wide string dynamic memory layout (Windows only)

Offset	Contents
-4	32-bit length indicator (in bytes)
0.. <i>Length</i> - 1	character string
<i>Length</i>	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

## Set types

---

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to

$$(Max \text{ div } 8) - (Min \text{ div } 8) + 1$$

where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is

$$(E \text{ div } 8) - (Min \text{ div } 8)$$

and the bit number within that byte is

$$E \text{ mod } 8$$

where *E* denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic *Integer* type or if the program contains code that takes the address of the set.

## Static array types

---

A static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

## Dynamic array types

---

A dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is **nil** and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

**Table 11.3** Dynamic array memory layout

Offset	Contents
-8	32-bit reference-count
-4	32-bit length indicator (number of elements)
0.. <i>Length</i> * (size of element) - 1	array elements

## Record types

---

When a record type is declared in the **{SA+}** state (the default), and when the declaration does not include a **packed** modifier, the type is an *unpacked record type*, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

**Table 11.4** Type alignment masks

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for <i>Real48</i> , 4 for <i>Single</i> , 8 for <i>Double</i> and <i>Extended</i>
Short string types	1
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record
Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	4

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

When a record type is declared in the **{SA-}** state, or when the declaration includes the **packed** modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

## File types

---

File types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```

type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
  case Byte of
    0: (RecSize: Cardinal);
    1: (BufSize: Cardinal;
        BufPos: Cardinal);

```

```

BufEnd: Cardinal;
BufPtr: PChar;
OpenFunc: Pointer;
InOutFunc: Pointer;
FlushFunc: Pointer;
CloseFunc: Pointer;
UserData: array[1..32] of Byte;
Name: array[0..259] of Char; );
end;

```

Text files occupy 460 bytes, which are laid out as follows:

```

type
TTextBuf = array[0..127] of Char;
TTextRec = packed record
  Handle: Integer;
  Mode: word;
  Flags: word;
  BufSize: Cardinal;
  BufPos: Cardinal;
  BufEnd: Cardinal;
  BufPtr: PChar;
  OpenFunc: Pointer;
  InOutFunc: Pointer;
  FlushFunc: Pointer;
  CloseFunc: Pointer;
  UserData: array[1..32] of Byte;
  Name: array[0..259] of Char;
  Buffer: TTextBuf;
end;

```

*Handle* contains the file's handle (when the file is open).

The *Mode* field can assume one of the values

```

const
fmClosed = $D7B0;
fmInput  = $D7B1;
fmOutput = $D7B2;
fmInOut  = $D7B3;

```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

*Name* contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O



routines that control the file; see “Device functions” on page 8-5. Flags determines the line break style as follows:

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

All other Flags bits are reserved for future use. See also *DefaultTextLineBreakStyle* and *SetLineBreakStyle*.

## Procedural types

---

A procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

## Class types

---

A class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object’s fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMTs, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointers—one per user-defined virtual method in the class type—in order of declaration. Each slot contains the address of the corresponding virtual method’s entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Object Pascal’s implementation. Applications should use the methods defined in *TObject* to query this information, since the layout is likely to change in future implementations of Object Pascal.

**Table 11.5** Virtual method table layout

Offset	Type	Description
-76	<i>Pointer</i>	pointer to virtual method table (or <b>nil</b> )
-72	<i>Pointer</i>	pointer to interface table (or <b>nil</b> )
-68	<i>Pointer</i>	pointer to Automation information table (or <b>nil</b> )
-64	<i>Pointer</i>	pointer to instance initialization table (or <b>nil</b> )
-60	<i>Pointer</i>	pointer to type information table (or <b>nil</b> )

**Table 11.5** Virtual method table layout (continued)

Offset	Type	Description
-56	<i>Pointer</i>	pointer to field definition table (or <b>nil</b> )
-52	<i>Pointer</i>	pointer to method definition table (or <b>nil</b> )
-48	<i>Pointer</i>	pointer to dynamic method table (or <b>nil</b> )
-44	<i>Pointer</i>	pointer to short string containing class name
-40	<i>Cardinal</i>	instance size in bytes
-36	<i>Pointer</i>	pointer to a pointer to ancestor class (or <b>nil</b> )
-32	<i>Pointer</i>	pointer to entry point of <i>SafecallException</i> method (or <b>nil</b> )
-28	<i>Pointer</i>	entry point of <i>AfterConstruction</i> method
-24	<i>Pointer</i>	entry point of <i>BeforeDestruction</i> method
-20	<i>Pointer</i>	entry point of <i>Dispatch</i> method
-16	<i>Pointer</i>	entry point of <i>DefaultHandler</i> method
-12	<i>Pointer</i>	entry point of <i>NewInstance</i> method
-8	<i>Pointer</i>	entry point of <i>FreeInstance</i> method
-4	<i>Pointer</i>	entry point of <i>Destroy</i> destructor
0	<i>Pointer</i>	entry point of first user-defined virtual method
4	<i>Pointer</i>	entry point of second user-defined virtual method
⋮	⋮	⋮

## Class reference types

---

A class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

## Variant types

---

A variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The *System* and *Variants* units define constants and types for variants.

The *TVarData* type represents the internal structure of a *Variant* variable (on Windows, this is identical to the *Variant* type used by COM and the Win32 API). The *TVarData* type can be used in typecasts of *Variant* variables to access the internal structure of a variable.

The *VType* field of a *TVarData* record contains the type code of the variant in the lower twelve bits (the bits defined by the *varTypeMask* constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.

The *Reserved1*, *Reserved2*, and *Reserved3* fields of a *TVarData* record are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field. If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.

If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines an array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.

If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and *varArray* bits in the *VType* field.

The *varString* type code is private. Variants containing a *varString* value should never be passed to a non-Delphi function. On Windows, Delphi's Automation support automatically converts *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

On Linux, *VT\_decimal* is not supported.

## Program control

This chapter explains how parameters and function results are stored and transferred. The final section discusses exit procedures.

### Parameters and function results

---

Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

#### Parameter passing

---

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see "Calling conventions" on page 6-4.

Variable (**var**) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (**const**) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A *Single* parameter occupies 4 bytes, and a *Double*, *Comp*, or *Currency* parameter occupies 8 bytes. A *Real48* occupies 8 bytes, with the *Real48* value stored in the lower 6 bytes. An *Extended* occupies 12 bytes, with the *Extended* value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.

- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value `nil` is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the **register** and **pascal** conventions, a variant parameter is passed as a 32-bit pointer to a *Variant* value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32-bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack under the **cdecl**, **stdcall**, and **safecall** conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.
- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

Under the **pascal**, **cdecl**, **stdcall** and **safecall** conventions, all parameters are passed on the stack. Under the **pascal** convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up at the highest address and the last parameter ends up at the lowest address. Under the **cdecl**, **stdcall**, and **safecall** conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

Under the **register** convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the **pascal** convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, *Int64*, and structured types do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to *Test* passes *A* in EAX as a 32-bit integer, *B* in EDX as a pointer to a *Char*, and *D* in ECX as a pointer to a long-string memory block; *C* and *E* are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

## Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a CLD instruction) and must return with the direction flag cleared.

## Function results

---

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type *Currency*, the value in ST(0) is scaled by 10000. For example, the *Currency* value 1.234 is returned in ST(0) as 12340.
- For a string, dynamic array, method pointer, variant, or *Int64* result, the effects are the same as if the function result were declared as an additional **var** parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.
- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional **var** parameter that is passed to the function after the declared parameters.

## Method calls

---

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter *Self*, which is a reference to the instance or class in which the method is called. The *Self* parameter is passed as a 32-bit pointer.

- Under the **register** convention, *Self* behaves as if it were declared *before* all other parameters. It is therefore always passed in the EAX register.
- Under the **pascal** convention, *Self* behaves as if it were declared *after* all other parameters (including the additional **var** parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.
- Under the **cdecl**, **stdcall**, and **safecall** conventions, *Self* behaves as if it were declared *before* all other parameters, but *after* the additional **var** parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional **var** parameter.

## Constructors and destructors

Constructors and destructors use the same calling conventions as other methods, except that an additional *Boolean* flag parameter is passed to indicate the context of the constructor or destructor call.

A value of *False* in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the **inherited** keyword. In this case, the constructor behaves like an ordinary method. A value of *True* in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by *Self*, and returns a reference to the newly created object in EAX.

A value of *False* in the flag parameter of a destructor call indicates that the destructor was invoked using the **inherited** keyword. In this case, the destructor behaves like an ordinary method. A value of *True* in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by *Self* just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the **register** convention, it is passed in the DL register. Under the **pascal** convention, it is pushed before all other parameters. Under the **cdecl**, **stdcall**, and **safecall** conventions, it is pushed just before the *Self* parameter.

Since the DL register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of DL before exiting so that *BeforeDestruction* or *AfterConstruction* can be called properly.

## Exit procedures

---

*Exit procedures* ensure that specific actions—such as updating and closing files—are carried out before a program terminates. The *ExitProc* pointer variable allows you to “install” an exit procedure, so that it is always called as part of the program’s termination—whether the termination is normal, forced by a call to *Halt*, or the result of a runtime error. An exit procedure takes no parameters.

**Note** It is recommended that finalization sections, rather than exit procedures, be used for all exit behavior. (See “The finalization section” on page 3-5.) Exit procedures are available only for executables, shared objects (Linux) or .DLL (Windows) targets; for packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn’t executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents of *ExitProc* before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of *ExitProc*.

The following code shows a skeleton implementation of an exit procedure.

```

var
  ExitSave: Pointer;

procedure MyExit;
begin
  ExitProc := ExitSave; // always restore old vector first
  :
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  :
end.

```

On entry, the code saves the contents of *ExitProc* in *ExitSave*, then installs the *MyExit* procedure. When called as part of the termination process, the first thing *MyExit* does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until *ExitProc* becomes **nil**. To avoid infinite loops, *ExitProc* is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to *ExitProc*. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable. In case of normal termination, *ExitCode* is zero and *ErrorAddr* is **nil**. In case of termination through a call to *Halt*, *ExitCode* contains the value passed to *Halt* and *ErrorAddr* is **nil**. In case of termination due to a runtime error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the *Input* and *Output* files. If *ErrorAddr* is not **nil**, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines *ErrorAddr* and outputs a message if it's not **nil**; before returning, set *ErrorAddr* to **nil** so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in *ExitCode* as a return code.





## Inline assembler code

The built-in assembler allows you to write assembler code within Object Pascal programs. It has the following features:

- Allows for inline assembly
- Supports all instructions found in the Intel Pentium III, SIMD, and the AMD Athlon (including 3D Now!)
- Provides no macro support, but allows for pure assembler function procedures
- Permits the use of Object Pascal identifiers, such as constants, types, and variables in assembler statements

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See “Linking to object files” on page 6-6 for more information.

**Note** If you have external assembler code that you want to use in your applications, you should consider rewriting it in Object Pascal or minimally reimplement it using the inline assembler.

### The `asm` statement

---

The built-in assembler is accessed through `asm` statements, which have the form

```
asm statementList end
```

where *statementList* is a sequence of assembler statements separated by semicolons, end-of-line characters, or Object Pascal comments.

Comments in an `asm` statement must be in Object Pascal style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word `inline` and the directive `assembler` are maintained for backward compatibility only. They have no effect on the compiler.

## Register use

---

In general, the rules of register use in an **asm** statement are the same as those of an **external** procedure or function. An **asm** statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an **asm** statement, BP points to the current stack frame, SP points to the top of the stack, SS contains the segment address of the stack segment, and DS contains the segment address of the data segment. Except for ESP and EBP, an **asm** statement can assume nothing about register contents on entry to the statement.

## Assembler statement syntax

---

This syntax of an assembler statement is

*Label*: *Prefix Opcode Operand*<sub>1</sub>, *Operand*<sub>2</sub>

where *Label* is a label, *Prefix* is an assembler prefix opcode (operation code), *Opcode* is an assembler instruction opcode or directive, and *Operand* is an assembler expression. *Label* and *Prefix* are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembler statements, but not within them. For example,

```
MOV AX,1 {Initial value}    { OK }
MOV CX,100 {Count}         { OK }

MOV {Initial value} AX,1;   { Error! }
MOV CX, {Count} 100        { Error! }
```

## Labels

---

Labels are used in built-in assembler statements as they are in Object Pascal—by writing the label and a colon before a statement. There is no limit to a label's length. As in Object Pascal, labels must be declared in a **label** declaration part in the block containing the **asm** statement. There is one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to **asm** statements, and the scope of a local label extends from the **asm** reserved word to the end of the **asm** statement that contains it. A local label doesn't have to be declared.

## Instruction opcodes

---

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family

- Pentium Pro and Pentium II
- Pentium III
- Pentium IV

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

## RET instruction sizing

The RET instruction opcode always generates a near return.

## Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is  $-128$  to  $127$  bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is  $-128$  to  $127$  bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembler statement

```
JC      Stop
```

where *Stop* isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
JNC     Skip
JMP     Stop
Skip:
```

Jumps to the entry points of procedures and functions are always near.

## Assembler directives

---

The built-in assembler supports three assembler define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between  $-128$  and  $255$ , or a character string of any length.

Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between  $-32,768$  and  $65,535$ , or an address expression. For an address expression, the built-in assembler generates a near pointer—that is, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between  $-2,147,483,648$  and  $4,294,967,295$ , or an address expression. For an address expression, the built-in assembler generates a far pointer—that is, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The DQ directive defines a quadword for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembler statements. To generate uninitialized or initialized data in the data segment, you should use Object Pascal **var** or **const** declarations.

Some examples of DB, DW, and DD directives follow.

```
asm
  DB      0FFH                { One byte }
  DB      0,99                { Two bytes }
  DB      'A'                 { Ord('A') }
  DB      'Hello world...',0DH,0AH { String followed by CR/LF }
  DB      12,"string"        { Object Pascal style string }
  DW      0FFFFH             { One word }
  DW      0,9999             { Two words }
  DW      'A'                { Same as DB 'A',0 }
  DW      'BA'               { Same as DB 'A','B' }
  DW      MyVar              { Offset of MyVar }
  DW      MyProc             { Offset of MyProc }
  DD      0FFFFFFFFFH        { One double-word }
  DD      0,999999999        { Two double-words }
  DD      'A'                { Same as DB 'A',0,0,0 }
  DD      'DCBA'            { Same as DB 'A','B','C','D' }
  DD      MyVar              { Pointer to MyVar }
  DD      MyProc            { Pointer to MyProc }
end;
```

In Turbo Assembler, when an identifier precedes a DB, DW, or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, Turbo Assembler allows the following:

```

ByteVar    DB    ?
WordVar    DW    ?
IntVar     DD    ?
:
           MOV   AL,ByteVar
           MOV   BX,WordVar
           MOV   ECX,IntVar

```

The built-in assembler doesn't support such variable declarations. The only kind of symbol that can be defined in an inline assembler statement is a label. All variables must be declared using Object Pascal syntax; the preceding construction can be replaced by

```

var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
  :
asm
  MOV   AL,ByteVar
  MOV   BX,WordVar
  MOV   ECX,IntVar
end;

```

SMALL and LARGE can be used to determine the width of a displacement:

```
MOV eax, [large $1234]
```

This instruction generates a "normal" move with a 32-bit displacement (\$00001234).

```
MOV eax, [small $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement (\$1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
MOV eax, [SMALL 123]
```

as opposed to

```
mov eax, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual method: VMTOFFSET and DMTINDEX.

VMTOFFSET retrieves the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter, for example, TExample.VirtualMethod.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example, TExample.DynamicMethod. To invoke the dynamic method, call System.@CallDynaInst with the (E)SI register containing the value obtained from DMTINDEX.

**Note** Methods with the "message" directive, are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```
TMyClass = class
  procedure x; message MYMESSAGE;
end;
```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```
program Project2;

type
  TExample = class
    procedure DynamicMethod; dynamic;
    procedure VirtualMethod; virtual;
  end;

procedure TExample.DynamicMethod;
begin
end;

procedure TExample.VirtualMethod;
begin
end;

procedure CallDynamicMethod(e: TExample);
asm
  // Save ESI register
  PUSH  ESI

  // Instance pointer needs to be in EAX
  MOV   EAX, e
  // DMT entry index needs to be in (E)SI
  MOV   ESI, DMTINDEX TExample.DynamicMethod
  // Now call the method
  CALL  System.@CallDynaInst

  // Restore ESI register
  POP  ESI
end;

procedure CallVirtualMethod(e: TExample);
asm
  // Instance pointer needs to be in EAX
  MOV   EAX, e
```

```

// Retrieve VMT table entry
MOV     EDX, [EAX]

// Now call the method at offset VMTOFFSET
CALL   DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]
end;

var
  e: TExample;

begin
  e := TExample.Create;
  try
    CallDynamicMethod(e);
    CallVirtualMethod(e);
  finally
    e.Free;
  end;
end.

```

## Operands

Built-in assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings

**Table 13.1** Built-in assembler reserved words

AH	BYTE	DMTINDEX	EDI	HIGH	QWORD	TBYTE
AL	CH	DS	EDX	LARGE	SHL	TYPE
AND	CL	DWORD	EIP	LOW	SHR	VMTOFFSET
AX	CS	DX	ES	MOD	SI	WORD
BH	CX	EAX	ESI	NOT	SMALL	XOR
BL	DH	EBP	ESP	OFFSET	SP	
BP	DI	EBX	FS	OR	SS	
BX	DL	ECX	GS	PTR	ST	

Reserved words always take precedence over user-defined identifiers. For example,

```

var
  Ch: Char;
  :
asm
  MOV     CH, 1
end;

```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) override operator:

```
MOV     &Ch, 1
```



It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

## Expressions

---

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants.

Expressions are built from *expression elements* and *operators*, and each expression has an associated *expression class* and *expression type*.

### Differences between Object Pascal and assembler expressions

---

The most important difference between Object Pascal expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value—a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid statement.

```
asm
  MOV     Z, X+Y
end;
```

Because both  $X$  and  $Y$  are constants, the expression  $X + Y$  is a convenient way of writing the constant 30, and the resulting instruction simply moves of the value 30 into the variable  $Z$ . But if  $X$  and  $Y$  are variables—

```
var
  X, Y: Integer;
```

—the built-in assembler cannot compute the value of  $X + Y$  at compile time. In this case, to move the sum of  $X$  and  $Y$  into  $Z$  you would use

```
asm
  MOV     EAX, X
  ADD     EAX, Y
  MOV     Z, EAX
end;
```

In an Object Pascal expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Object Pascal the expression  $X + 4$  (where  $X$  is a variable) means the contents of  $X$  plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of  $X$ . So, even though you're allowed to write

```
asm
```

```

MOV     EAX,X+4
end;

```

this code doesn't load the value of  $X$  plus 4 into  $AX$ ; instead, it loads the value of a word stored four bytes beyond  $X$ . The correct way to add 4 to the contents of  $X$  is

```

asm
MOV     EAX,X
ADD     EAX,4
end;

```

## Expression elements

---

The elements of an expression are *constants*, *registers*, and *symbols*.

### Constants

The built-in assembler supports two types of constant: *numeric constants* and *string constants*.

#### Numeric constants

Numeric constants must be integers, and their values must be between  $-2,147,483,648$  and  $4,294,967,295$ .

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a  $B$  after the number, octal notation by writing an  $O$  after the number, and hexadecimal notation by writing an  $H$  after the number or a  $\$$  before the number.

Numeric constants must start with one of the digits 0 through 9 or the  $\$$  character. When you write a hexadecimal constant using the  $H$  suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example,  $0BAD4H$  and  $\$BAD4$  are hexadecimal constants, but  $BAD4H$  is an identifier because it starts with a letter.

#### String constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```

'Z'
'Delphi'
'Linux'
"That's all folks"
'"That's all folks," he said.'
'100'
'''
'''

```

String constants of any length are allowed in  $DB$  directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a

numeric value which can participate in an expression. The numeric value of a string constant is calculated as

$$\text{Ord}(\text{Ch1}) + \text{Ord}(\text{Ch2}) \text{ shl } 8 + \text{Ord}(\text{Ch3}) \text{ shl } 16 + \text{Ord}(\text{Ch4}) \text{ shl } 24$$

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

**Table 13.2** String examples and their values

String	Value
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

## Registers

The following reserved symbols denote CPU registers:

**Table 13.3** CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a *register operand*. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registers—for example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(*X*), where *X* is a constant between 0 and 7 indicating the distance from the top of the register stack.

## Symbols

The built-in assembler allows you to access almost all Object Pascal identifiers in assembler expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol *@Result*, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
    Result := X + Y;
end;
```

could be written in assembler as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
    asm
        MOV     EAX, X
        ADD     EAX, Y
        MOV     @Result, EAX
    end;
end;
```

The following symbols cannot be used in **asm** statements:

- Standard procedures and functions (for example, *WriteLn* and *Chr*).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The *@Result* symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in **asm** statements.

**Table 13.4** Symbols recognized by the built-in assembler

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
<i>@Result</i>	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV    EAX,Count
```

assembles into `MOV EAX,[EBP-4]`.

The built-in assembler treats **var** parameters as a 32-bit pointers, and the size of a **var** parameter is always 4. The syntax for accessing a **var** parameter is different from that for accessing a value parameter. To access the contents of a **var** parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV    EAX,X
    MOV    EAX,[EAX]
    MOV    EDX,Y
    ADD   EAX,[EDX]
    MOV   @Result,EAX
  end;
end;
```

Identifiers can be qualified within **asm** statements. For example, given the declarations

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

the following constructions can be used in an **asm** statement to access fields.

```
MOV    EAX,P.X
MOV    EDX,P.Y
MOV    ECX,R.A.X
MOV    EBX,R.B.Y
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of `[EDX]` into `EAX`.

```
MOV    EAX,(TRect PTR [EDX]).B.X
MOV    EAX,TRect([EDX]).B.X
MOV    EAX,TRect[EDX].B.X
MOV    EAX,[EDX].TRect.B.X
```

## Expression classes

---

The built-in assembler divides expressions into three classes: *registers*, *memory references*, and *immediate values*.

An expression that consists solely of a register name is a *register* expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Object Pascal's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Object Pascal's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```

const
  Start = 10;
var
  Count: Integer;
  :
asm
  MOV     EAX,Start           { MOV EAX,xxxx }
  MOV     EBX,Count          { MOV EBX,[xxxx] }
  MOV     ECX,[Start]        { MOV ECX,[xxxx] }
  MOV     EDX,OFFSET Count   { MOV EDX,xxxx }
end;
```

Because *Start* is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as *Count* is a memory reference. In the third MOV, the brackets convert *Start* into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth MOV, the OFFSET operator converts *Count* into an immediate value (the offset of *Count* in the data segment).

The brackets and OFFSET operator complement each other. The following **asm** statement produces identical machine code to the first two lines of the previous **asm** statement.

```

asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either *relocatable* or *absolute*. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

## Expression types

---

Every built-in assembler expression has a type—or, more correctly, a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an *Integer* variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  :
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

the assembler checks that the size of *QuitFlag* is one (a byte), and that the size of *OutBufPtr* is two (a word). The instruction

```
MOV     DL,OutBufPtr
```

produces an error because *DL* is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

These *MOV* instructions all refer to the first (least significant) byte of the *OutBufPtr* variable.

In some cases, a memory reference is untyped. One example is an immediate value enclosed in square brackets:

```
MOV al, [Buffer]
MOV cx, [Buffer]
MOV edx, [Buffer]
```

The built-in assembler permits both of these instructions, because the expression `[100H]` has no type—it just means “the contents of address 100H in the data segment,” and the type can be determined from the first operand (byte for *AL*, word for *BX*). In cases where the type can’t be determined from another operand, the built-in assembler requires an explicit typecast:

```
INC     BYTE PTR [ECX]
IMUL   WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Object Pascal types.

**Table 13.5** Predefined type symbols

Symbol	Type
BYTE	1
WORD	2

**Table 13.5** Predefined type symbols (continued)

Symbol	Type
DWORD	4
QWORD	8
TBYTE	10

## Expression operators

The built-in assembler provides a variety of operators. Precedence rules are different from Object Pascal; for example, in an **asm** statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

**Table 13.6** Precedence of built-in assembler expression operators

Operators	Remarks	Precedence
&		highest
() , [] , ,, HIGH, LOW		
+, -	unary + and -	
:		
OFFSET, TYPE, PTR, *, /, MOD, SHL, SHR, +, -	binary + and -	
NOT, AND, OR, XOR		lowest

The following table defines the built-in assembler's expression operators.

**Table 13.7** Definitions of built-in assembler expression operators

Operator	Description
&	<b>Identifier override.</b> The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	<b>Subexpression.</b> Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[...]	<b>Memory reference.</b> The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	<b>Structure member selector.</b> The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.



**Table 13.7** Definitions of built-in assembler expression operators (continued)

Operator	Description
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	<b>Unary plus.</b> Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
-	<b>Unary minus.</b> Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	<b>Addition.</b> The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
-	<b>Subtraction.</b> The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	<b>Segment override.</b> Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	<b>Typecast operator.</b> The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	<b>Multiplication.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	<b>Integer division.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	<b>Remainder after integer division.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	<b>Logical shift left.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	<b>Logical shift right.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	<b>Bitwise negation.</b> The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	<b>Bitwise AND.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	<b>Bitwise OR.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	<b>Bitwise exclusive OR.</b> Both expressions must be absolute immediate values, and the result is an absolute immediate value.

# Assembler procedures and functions

---

You can write complete procedures and functions using inline assembler code, without including a **begin...end** statement. For example,

```
function LongMul(X, Y: Integer): Longint;
asm
    MOV     EAX,X
    IMUL   Y
end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were **var** parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a reference to the *@Result* symbol is an error. For strings, variants, and interfaces, the caller always allocates an *@Result* pointer.
- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- The automatically generated entry and exit code for the routine looks like this:

```
PUSH   EBP           ;Present if Locals <> 0 or Params <> 0
MOV    EBP,ESP       ;Present if Locals <> 0 or Params <> 0
SUB    ESP,Locals    ;Present if Locals <> 0
:
MOV    ESP,EBP       ;Present if Locals <> 0
POP    EBP           ;Present if Locals <> 0 or Params <> 0
RET    Params        ;Always present
```

If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.

- *Locals* is the size of the local variables and *Params* is the size of the parameters. If both *Locals* and *Params* are zero, there is no entry code, and the exit code consists simply of a RET instruction.

Assembler functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (*Currency* values are scaled by 10000.)
- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by *@Result*.





# Object Pascal grammar

```
Goal -> (Program | Package | Library | Unit)
Program -> [PROGRAM Ident ['(' IdentList ')'] ';' ]
        ProgramBlock '.'
Unit -> UNIT Ident ';'
      InterfaceSection
      ImplementationSection
      InitSection '.'
Package -> PACKAGE Ident ';'
        [RequiresClause]
        [ContainsClause]
        END '.'
Library -> LIBRARY Ident ';'
        ProgramBlock '.'
ProgramBlock -> [UsesClause]
              Block
UsesClause -> USES IdentList ';'
InterfaceSection -> INTERFACE
                 [UsesClause]
                 [InterfaceDecl]...
InterfaceDecl -> ConstSection
               -> TypeSection
               -> VarSection
               -> ExportedHeading
ExportedHeading -> ProcedureHeading ';' [Directive]
                -> FunctionHeading ';' [Directive]
ImplementationSection -> IMPLEMENTATION
                       [UsesClause]
                       [DeclSection]...
Block -> [DeclSection]
```

```

CompoundStmt

DeclSection -> LabelDeclSection
            -> ConstSection
            -> TypeSection
            -> VarSection
            -> ProcedureDeclSection

LabelDeclSection -> LABEL LabelId

ConstSection -> CONST (ConstantDecl ';')...

ConstantDecl -> Ident '=' ConstExpr
              -> Ident ':' TypeId '=' TypedConstant

TypeSection -> TYPE (TypeDecl ';')...

TypeDecl -> Ident '=' Type
          -> Ident '=' RestrictedType

TypedConstant -> (ConstExpr | ArrayConstant | RecordConstant)

ArrayConstant -> '(' TypedConstant/', '... ')

RecordConstant -> '(' RecordFieldConstant/';'... ')

RecordFieldConstant -> Ident ':' TypedConstant

Type -> TypeId
     -> SimpleType
     -> StructType
     -> PointerType
     -> StringType
     -> ProcedureType
     -> VariantType
     -> ClassRefType

RestrictedType -> ObjectType
               -> ClassType
               -> InterfaceType

ClassRefType -> CLASS OF TypeId

SimpleType -> (OrdinalType | RealType)

RealType -> REAL48
          -> REAL
          -> SINGLE
          -> DOUBLE
          -> EXTENDED
          -> CURRENCY
          -> COMP

OrdinalType -> (SubrangeType | EnumeratedType | OrdIdent)

OrdIdent -> SHORTINT
          -> SMALLINT
          -> INTEGER
          -> BYTE
          -> LONGINT
          -> INT64
          -> WORD
          -> BOOLEAN

```

```

-> CHAR
-> WIDECHAR
-> LONGWORD
-> PCHAR

VariantType -> VARIANT
             -> OLEVARIANT

SubrangeType -> ConstExpr '..' ConstExpr

EnumeratedType -> '(' EnumeratedTypeElement/','...' ')'

EnumeratedTypeElement -> Ident [ '=' ConstExpr ]

StringType -> STRING
            -> ANSISTRING
            -> WIDESTRING
            -> STRING '[' ConstExpr ']'

StructType -> [PACKED] (ArrayType | SetType | FileType | RecType)

ArrayType -> ARRAY ['[' OrdinalType/','...' ']'] OF Type

RecType -> RECORD [FieldList] END

FieldList -> FieldDecl/';'... [VariantSection] [';']

FieldDecl -> IdentList ':' Type

VariantSection -> CASE [Ident ':'] TypeId OF RecVariant/';'...

RecVariant -> ConstExpr/','...' ':' '(' [FieldList] ')'

SetType -> SET OF OrdinalType

FileType -> FILE OF TypeId

PointerType -> '^' TypeId

ProcedureType -> (ProcedureHeading | FunctionHeading) [OF OBJECT]

VarSection -> VAR (VarDecl ';')...

VarDecl -> IdentList ':' Type [(ABSOLUTE (Ident | ConstExpr)) | '=' ConstExpr]

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['+' | '-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator ['(' ExprList ')']
        -> '@' Designator
        -> Number
        -> String
        -> NIL
        -> '(' Expression ')'
        -> NOT Factor
        -> SetConstructor
        -> TypeId '(' Expression ')'

RelOp -> '>'
        -> '<'
        -> '<='
        -> '>='
        -> '<>'

```

```

-> IN
-> IS
-> AS

AddOp -> '+'
-> '-'
-> OR
-> XOR

MulOp -> '*'
-> '/'
-> DIV
-> MOD
-> AND
-> SHL
-> SHR

Designator -> QualId ['. ' Ident | '[' ExprList ']' | '^']...
SetConstructor -> '[' [SetElement/','... ]'
SetElement -> Expression ['. ' Expression]
ExprList -> Expression/','...
Statement -> [LabelId ':' ] [SimpleStatement | StructStmt]
StmtList -> Statement/';'...
SimpleStatement -> Designator ['(' ExprList ')']
-> Designator ':=' Expression
-> INHERITED
-> GOTO LabelId

StructStmt -> CompoundStmt
-> ConditionalStmt
-> LoopStmt
-> WithStmt

CompoundStmt -> BEGIN StmtList END
ConditionalStmt -> IfStmt
-> CaseStmt

IfStmt -> IF Expression THEN Statement [ELSE Statement]
CaseStmt -> CASE Expression OF CaseSelector/';'... [ELSE StmtList] [';'] END
CaseSelector -> CaseLabel/','... ':' Statement
CaseLabel -> ConstExpr ['. ' ConstExpr]

LoopStmt -> RepeatStmt
-> WhileStmt
-> ForStmt

RepeatStmt -> REPEAT Statement UNTIL Expression
WhileStmt -> WHILE Expression DO Statement
ForStmt -> FOR QualId ':=' Expression (TO | DOWNT) Expression DO Statement
WithStmt -> WITH IdentList DO Statement
ProcedureDeclSection -> ProcedureDecl

```

```

        -> FunctionDecl

ProcedureDecl -> ProcedureHeading ';' [Directive]
                Block ';'

FunctionDecl -> FunctionHeading ';' [Directive]
                Block ';'

FunctionHeading -> FUNCTION Ident [FormalParameters] ':' (SimpleType | STRING)

ProcedureHeading -> PROCEDURE Ident [FormalParameters]

FormalParameters -> '(' FormalParm '/' ';' ... ')'

FormalParm -> [VAR | CONST | OUT] Parameter

Parameter -> IdentList [':' ([ARRAY OF] SimpleType | STRING | FILE)]
            -> Ident ':' SimpleType '=' ConstExpr

Directive -> CDECL
            -> REGISTER
            -> DYNAMIC
            -> VIRTUAL
            -> EXPORT
            -> EXTERNAL
            -> FAR
            -> FORWARD
            -> MESSAGE
            -> OVERRIDE
            -> OVERLOAD
            -> PASCAL
            -> REINTRODUCE
            -> SAFECALL
            -> STDCALL

ObjectType -> OBJECT [ObjHeritage] [ObjFieldList] [MethodList] END

ObjHeritage -> '(' QualId ')'

MethodList -> (MethodHeading [';' VIRTUAL]) / ';' ...

MethodHeading -> ProcedureHeading
                -> FunctionHeading
                -> ConstructorHeading
                -> DestructorHeading

ConstructorHeading -> CONSTRUCTOR Ident [FormalParameters]

DestructorHeading -> DESTRUCTOR Ident [FormalParameters]

ObjFieldList -> (IdentList ':' Type) / ';' ...

InitSection -> INITIALIZATION StmtList [FINALIZATION StmtList] END
                -> BEGIN StmtList END
                -> END

ClassType -> CLASS [ClassHeritage]
                [ClassFieldList]
                [ClassMethodList]
                [ClassPropertyList]
                END

ClassHeritage -> '(' IdentList ')'

```



```

ClassVisibility -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]
ClassFieldList -> (ClassVisibility ObjFieldList)/*;'...
ClassMethodList -> (ClassVisibility MethodList)/*;'...
ClassPropertyList -> (ClassVisibility PropertyList ';'...)
PropertyList -> PROPERTY Ident [PropertyInterface] PropertySpecifiers
PropertyInterface -> [PropertyParameterList] ':' Ident
PropertyParameterList -> '[' (IdentList ':' TypeId)/*;'... '['
PropertySpecifiers -> [INDEX ConstExpr]
                        [READ Ident]
                        [WRITE Ident]
                        [STORED (Ident | Constant)]
                        [(DEFAULT ConstExpr) | NODEFAULT]
                        [IMPLEMENTS TypeId]

InterfaceType -> INTERFACE [InterfaceHeritage]
                  [ClassMethodList]
                  [ClassPropertyList]
                  END

InterfaceHeritage -> '(' IdentList ')'
RequiresClause -> REQUIRES IdentList... ';'
ContainsClause -> CONTAINS IdentList... ';'
IdentList -> Ident/','...
QualId -> [UnitId '.'] Ident
TypeId -> [UnitId '.'] <type-identifier>
Ident -> <identifier>
ConstExpr -> <constant-expression>
UnitId -> <unit-identifier>
LabelId -> <label-identifier>
Number -> <number>
String -> <string>

```

# Index

## Symbols

---

- 4-4, 4-6, 4-9, 4-10  
" 13-9  
# 4-4  
\$ 4-4, 4-5  
(\* , \*) 4-5  
( , ) 4-2, 4-12, 4-14, 5-6, 5-43, 6-2, 6-3, 6-11, 7-2, 10-1  
\* 4-2, 4-6, 4-10  
+ 4-4, 4-6, 4-9, 4-10  
, 3-6, 4-24, 5-6, 5-21, 5-23, 6-11, 7-17, 9-5, 9-9, 10-7, 13-2  
. 3-2, 4-2, 4-13, 5-27, 9-9, 10-5  
/ 4-2, 4-6  
// 4-5  
: 4-2, 4-17, 4-23, 5-21, 5-22, 5-37, 5-43, 6-3, 6-11, 7-17, 7-19, 7-29, 13-2  
:= 4-17, 4-26  
    named parameters 10-12  
; 3-2, 3-6, 4-16, 4-17, 4-20, 4-25, 5-21, 5-22, 5-37, 6-2, 6-3, 6-11, 7-2, 7-6, 9-5, 9-9, 10-1, 10-4, 10-10  
    before 'else' 4-22  
< 4-2, 4-9, 4-10  
<= 4-9, 4-10  
<> 4-9, 4-10  
= 4-2, 4-9, 4-10, 4-17, 5-36, 5-37, 5-39, 5-43, 6-11, 6-17, 10-5  
> 4-2, 4-9, 4-10  
>= 4-9, 4-10  
@ 4-6, 4-12, 5-26, 5-30, 5-44, 7-17  
@@ 5-30  
@Result 13-11, 13-17  
[ , ] 4-13, 4-14, 5-11, 5-12, 5-17, 5-31, 6-15, 6-19, 7-17, 7-19, 7-20, 10-1, 10-10  
^ 4-6, 4-9, 5-19, 5-27  
    and variants 5-33  
    pointer overview 5-26  
\_ 4-2  
{ , } 4-5, 10-1, 10-10  
' 4-4, 9-9, 10-1, 10-10

## A

---

\$A directive 11-8  
absolute (directive) 5-38  
absolute addresses 5-38

absolute expressions  
    (assembly) 13-13  
abstract methods 7-12  
access specifiers 7-1, 7-17  
array properties 7-19  
    Automation 7-6  
    calling convention 6-5, 7-18  
    index specifiers and 7-21  
    overloading 7-13, 7-18  
    overriding 7-22  
actual parameters 6-19  
Add method (TCollection) 7-9  
addition 4-6  
    pointers 4-9  
Addr function 5-26  
\_AddRef method 10-2, 10-5, 10-9  
address operator 4-12, 5-26, 5-30, 5-44  
    properties and 7-17  
alignment (data) 5-17, 11-8  
    *See also* internal data formats  
AllocMemCount variable 11-2  
AllocMemSize variable 11-2  
alphanumeric characters 4-1, 4-2  
ampersand *See* Symbols  
ancestors 7-3  
and 4-7, 4-8  
anonymous values (enumerated types) 5-8, 7-5  
ANSI characters 5-5, 5-12, 5-13  
AnsiChar type 5-5, 5-11, 5-13, 5-27, 11-3  
AnsiString type 5-10, 5-12, 5-13, 5-15, 5-27  
    *See also* long strings  
    memory management 11-6  
    variant arrays and 5-33  
Append procedure 8-2, 8-4, 8-5, 8-6  
application partitioning 9-8  
Application variable 2-5, 3-3  
arithmetic operators 4-6, 5-4  
array properties 7-5, 7-19  
    default 7-20  
    in dispatch interfaces 10-11  
    storage specifiers and 7-21  
arrays 5-3, 5-18 to 5-21  
    'array of const' 6-16  
    accessing with  
        PByteArray 5-27  
        accessing with  
            PWordArray 5-28  
    assignments and 5-21  
    character 4-5, 5-13, 5-14, 5-17, 5-18  
    character arrays and string  
        constants 4-5, 5-14, 5-43  
    constant 5-43  
    dynamic 5-19, 5-39, 6-15, 11-7  
    indexes 4-14  
    multidimensional 5-18, 5-20  
    open array  
        constructors 6-17, 6-19  
        parameters 6-12, 6-15  
        static 5-18, 11-7  
        variants and 5-31, 5-33  
as 4-11, 7-24, 7-25, 10-10  
ASCII 4-1, 4-4, 5-13  
asm statements 13-1, 13-17  
assembler (directive) 6-6, 13-1  
assembly language  
    assembler routines 13-17  
    built-in assembler 13-1 to 13-17  
    external routines 6-7  
    Object Pascal and 13-5, 13-7, 13-8, 13-11, 13-13, 13-15  
Assert procedure 7-27  
assertions 7-27  
Assign procedure  
    custom 8-4  
Assigned function 5-30, 10-9  
AssignFile procedure 8-2, 8-4, 8-6  
assignment statements 4-17  
    typecasts 4-15  
assignment-compatibility 5-35, 7-3, 10-9  
asterisk *See* Symbols  
at (reserved word) 7-27  
at-sign *See* @, address operator  
automatable types 7-6, 10-11  
automated class members 7-4, 7-6  
Automation 7-6, 10-10 to 10-13  
    *See also* COM  
    dual interfaces 10-13  
    method calls 10-12  
    variants and 11-12

## B

`$B` directive 4-8  
base types 5-8, 5-17, 5-18, 5-19  
begin (reserved word) 3-2, 4-20, 6-2, 6-3  
binary operators 4-6  
binding  
    fields 7-7  
    methods 7-10  
bitwise operators, not 4-8  
blanks 4-1  
BlockRead procedure 8-4  
blocks 4-27 to 4-28  
    function 3-4, 6-1, 6-3  
    library 9-5  
    outer and inner 4-29  
    procedure 3-4, 6-1, 6-2  
    program 3-1, 3-2  
    scope 4-27 to 4-29  
    try...except 7-28, 7-30  
    try...finally 7-31  
BlockWrite procedure 8-4  
body (routine) 6-1  
boldface 1-2  
Boolean operators 4-7  
    complete vs. partial evaluation 4-7  
Boolean type 5-6, 11-3  
Boolean types 5-5, 11-3  
BORLANDMM.DLL 9-8  
braces *See* Symbols  
brackets *See* Symbols  
Break procedure 4-25  
    exception handlers 7-29  
    in try...finally block 7-32  
BSTR type (COM) 5-13  
built-in assembler 13-1 to 13-17  
built-in types 5-1  
by reference (parameters) 6-12, 6-13, 10-12, 12-1  
by value (parameters) 6-12, 10-12, 12-1  
Byte type 5-4, 11-3  
    assembler 13-14  
ByteBool type 5-6, 11-3

## C

C++ 6-6, 10-1, 11-10  
calling conventions 5-29, 6-4, 12-1  
    access specifiers 6-5, 7-18  
    interfaces 10-3, 10-7  
    methods 12-3  
    shared libraries 9-4

    varargs directive 6-6  
    calling routines 9-1  
Cardinal type 5-3  
caret *See* Symbols  
carriage-return 4-1, 4-5  
case (reserved word) 4-23, 5-22  
case statements 4-23  
case-sensitivity 4-1, 4-2, 6-8  
    unit names and files 4-2  
-cc compiler switch 8-3  
cdecl (calling convention) 6-4, 12-2  
    constructors and destructors 12-4  
    Self 12-3  
    varargs 6-6  
Char type 5-5, 5-13, 5-27, 11-3  
character operators 4-9  
character sets  
    ANSI 5-5, 5-12, 5-13  
    extended 5-13  
    multibyte (MBCS) 5-13  
    Pascal 4-1, 4-2, 4-4  
    single-byte (SBCS) 5-13  
character strings 4-1, 4-4, 5-45  
characters  
    pointers 5-27  
    string literals as 4-5, 5-5  
    types 5-5, 11-3  
    wide 5-13, 11-3  
checked typecasts  
    interfaces 10-10  
    objects 7-25  
Chr function 5-5  
circular references  
    packages 9-10  
    units 3-7 to 3-8  
circumflex *See* Symbols  
classes 7-1 to 7-32  
    class methods 7-1, 7-25  
    class references 7-23  
    class types 7-1, 7-2  
    comparison 4-11  
    compatibility 7-3, 10-9  
    declaring class types 7-2, 7-4, 7-6, 7-7, 7-8, 7-17, 10-5  
    files and 5-24  
    memory 11-10  
    metaclasses 7-23  
    operators 4-11, 7-24  
    scope 4-28  
    variants and 5-31  
Classes unit 7-9, 7-23  
ClassParent method 7-24  
class-reference types 7-23

    comparison 4-11  
    constructors and 7-24  
    memory 11-11  
    variants and 5-31  
ClassType method 7-24  
clients 3-4  
Close function 8-4, 8-6  
CloseFile function 8-5  
CloseFile procedure 8-6  
CLX 1-2  
CmdLine variable 9-6  
colon *See* Symbols  
COM 10-4  
    *See also* Automation  
    interfaces 10-2, 10-10 to 10-13  
    out parameters 6-14  
    variants and 5-31, 5-33, 11-11  
COM error handling 6-5  
comma *See* Symbols  
comments 4-1, 4-5  
ComObj unit 7-6, 10-11  
Comp type 5-9, 5-10, 11-5  
comparison  
    classes 4-11  
    class-reference types 4-11  
    dynamic arrays 5-19  
    integer types 4-11  
    objects 4-11  
    packed strings 4-11  
    PChar type 4-11  
    real types 4-11  
    relational operators 4-10  
    strings 4-11, 5-11  
compiler 2-2, 2-3, 2-5, 3-1  
    command-line 2-3 to 2-5  
    directives 3-2, 4-5  
    packages 9-11  
compile-time binding *See* static methods  
complete evaluation 4-7  
components, of classes *See* members  
compound statements 4-19, 4-20  
concatenation (strings) 4-9  
conditional statements 4-19  
conjunction 4-7  
console applications 2-3, 8-3  
const (reserved word) 5-39, 5-43, 6-11, 6-12, 6-13, 6-16, 12-1  
constant expression  
    array constants 5-43  
    case statements 4-24

- constant declarations 5-40, 5-43
- default paramters 6-18
- defined 5-41
- enumerated types 5-7
- initializing variables 5-37
- subrange types 5-8, 5-9
- type 5-40, 6-8
- constant parameters 6-12, 6-13, 6-19, 12-1
  - open array constructors 6-20
- constants 4-6, 5-39
  - array 5-43
  - assembler 13-9
  - declared 5-39 to 5-45
  - numeric *See* numerals
  - pointer 5-44
  - procedural 5-44
  - record 5-43
  - true 5-39
  - type compatibility 5-40
  - typed 5-42
- constructors 7-1, 7-9, 7-13
  - calling conventions 12-4
  - class references and 7-24
  - exceptions 7-28, 7-32
- contains clause 9-9, 9-10
- context-sensitive Help (error-handling) 7-32
- Continue procedure 4-25
  - exception handlers 7-29
  - in try...finally block 7-32
- control (program) 6-19, 12-1 to 12-5
- control characters 4-1, 4-4
- control loops 4-20, 4-25
- control strings 4-4
- conversion
  - See also* typecasts
  - variants 5-31, 5-31 to 5-33
- Copy function 5-20
- copy-on-write semantics 5-12
- CORBA
  - interfaces 10-3
  - out parameters 6-14
- CPU *See* registers
- Create method 7-13
- Currency type 5-9, 5-10, 5-27, 11-5

**D**

---

- data alignment 5-17, 11-8
- data formats, internal 11-2 to 11-12
- data types *See* types
- .dcp files 9-11, 2-3
- .dcu files 2-3, 3-7, 9-10, 9-11
- Dec procedure 5-3, 5-4
- declarations 4-1, 4-16, 4-27
  - class 7-2, 7-7, 7-8, 7-17, 10-5
  - constant 5-39, 5-43
  - defining 6-6, 7-6, 7-8, 10-4
  - field 7-7
  - forward 3-4, 6-5, 7-6, 10-4
  - function 6-1, 6-3
  - implementation 7-8
  - interface 3-4
  - local 6-10
  - method 7-8
  - package 9-9
  - procedure 6-1, 6-2
  - property 7-17, 7-19
  - type 5-36
  - variable 5-37
- declared types 5-1
- decrementing ordinals 5-3, 5-4, 5-5
- default (directive) 7-20, 10-11
- default parameters 6-11, 6-17 to 6-19
  - Automation objects 10-12
  - forward and interface
    - declarations 6-19
  - overloading and 6-10, 6-18
  - procedural types 6-18
- default properties 7-20
- interfaces 10-2
- default property (COM object) 5-33
- default specifier 7-6, 7-17, 7-21
- DefaultHandler method 7-16, 7-17
- defining declarations 6-6, 7-6, 7-8, 10-4
- DefWindowProc function 7-16
- delegated interface 10-7
- delegation (interface implementation) 10-6
- \$DENYPACKAGEUNIT
  - directive 9-12
- dependency
  - units 3-7 to 3-8
- deprecated (directive) 4-17
- dereference operator 4-9, 5-19
  - pointer overview 5-26
  - variants and 5-33
- descendants 7-3, 7-5
- \$DESIGNONLY directive 9-12
- design-time packages 9-8
- .desk files 2-2
- desktop settings files 2-2
- Destroy method 7-13, 7-15, 7-29
- destructors 7-1, 7-13, 7-14
  - calling conventions 12-4
- device drivers, text-file 8-4
- device functions 8-4, 8-5
- .dfm files 2-2, 2-7, 7-5
- difference (sets) 4-10
- directives 4-1, 4-3
  - See also* reserved words
  - assembler 13-3
  - compiler 3-2, 4-5
  - list 4-3
  - order 7-8
- directory paths
  - in uses clause 3-6
- disjunction 4-7
  - bitwise 4-8
- dispatch interface types 10-10
- Dispatch method 7-16
- dispatching messages 7-16
- dispatching method calls 7-11
- dispid (directive) 7-6, 10-2, 10-11
- dispinterface 10-10
- Dispinterface (reserved word) 10-2
- Dispose procedure 5-19, 5-38, 7-4, 9-8, 11-1, 11-2
- div 4-6
- division 4-6
- dclose 9-2
- .DLL files 6-7, 9-1
- DLL\_PROCESS\_DETACH 9-7
- DLL\_THREAD\_ATTACH 9-7
- DLL\_THREAD\_DETACH 9-7
- DLLProc variable 9-7
- DLLs 9-1 to 9-8
  - calling routines in 6-7
  - dynamic arrays in 9-8
  - dynamic variables in 9-8
  - exceptions 9-7
  - global variables 9-6
  - loading dynamically 9-2
  - loading statically 9-1
  - long strings in 9-8
  - multithreading 9-7
  - variables 9-1
  - writing 9-3
- dlopen 9-2
- dlsym 9-2
- DMTINDEX 13-6
- do (reserved word) 4-20, 4-25, 4-26, 7-29
- .dof files 2-2

- dollar *See* Symbols
- Double type 5-9, 11-5
- downto (reserved word) 4-26
- .dpk files 2-2, 9-11
- .dpr files 2-2, 3-1, 3-6
- .dpu files 2-3, 3-7, 9-10, 9-11
- .drc files 2-3
- .dsk files 2-2
- dual interfaces 10-3, 10-13
  - methods 6-5
- DWORD type (assembler) 13-15
- dynamic arrays 5-19, 11-7
  - assigning to 5-19
  - comparison 5-19
  - files and 5-24
  - freeing 5-39
  - in dynamically loadable libraries 9-8
  - memory management 11-2
  - multidimensional 5-20
  - open array parameters and 6-15
  - records and 5-23
  - truncating 5-20
  - variants and 5-31
- dynamic methods 7-10
- dynamic variables 5-38
  - and pointer constants 5-45
  - in dynamically loadable libraries 9-8
- dynamically loadable libraries 6-7, 9-1 to 9-12
  - dynamic arrays 9-8
  - dynamic variables 9-8
  - exceptions 9-7
  - global variables 9-6
  - loading statically 9-1
  - long strings 9-8
  - variables 9-1
  - writing 9-3
- dynamic-link libraries *See* DLLs

**E**

---

- E (in numerals) 4-4
- EAssertionFailed 7-27
- else (reserved word) 4-22, 4-24, 7-28
- empty set 5-17
- end (reserved word) 3-2, 4-20, 4-23, 5-21, 5-22, 6-2, 6-3, 7-2, 7-28, 7-32, 9-9, 10-1, 10-10, 13-1
- end-of-line character 4-1, 8-3
- enumerated types 5-6 to 5-8, 11-3
  - anonymous values 5-8, 7-5

- publishing 7-5
- Eof function 8-5
- Eoln function 8-5
- equality operator 4-10
- error handling *See* exceptions
- ErrorAddr variable 12-5
- EStackOverflow exception 11-2
- EVariantError exception 5-32
- event handlers 2-7, 7-5
- events 2-7, 7-5
- example programs 2-3 to 2-5
- except (reserved word) 7-28
- ExceptAddr function 7-32
- Exception class 7-27, 7-32
- exception firewalls 6-5
- exception handlers 7-26, 7-29
  - identifiers in 7-30
- ExceptionInformation variable 9-7
- exceptions 4-20, 7-13, 7-15, 7-26 to 7-32
  - constructors 7-28, 7-32
  - declaring 7-27
  - destroying 7-28, 7-29
  - dynamically loadable libraries 9-6, 9-7
  - file I/O 8-3
  - handling 7-28, 7-30, 7-32
  - in initialization section 7-28
  - nested 7-31
  - propagation 7-29, 7-31, 7-32
  - raising 7-27
  - re-raising 7-30
  - standard exceptions 7-32
  - standard routines 7-32
- ExceptObject function 7-32
- executable files 2-3
- Exit procedure 6-1
  - exception handlers 7-29
  - in try...finally block 7-32
- exit procedures 9-6, 12-4 to 12-5
  - packages and 12-4
- ExitCode variable 9-6, 12-5
- ExitProc variable 9-6, 12-4
- export (directive) 6-5
- exports clause 4-28, 9-5
  - overloaded routines 9-5
- expressions 4-1, 4-5
  - assembler 13-8 to 13-16
- extended syntax 4-5, 4-18, 5-14, 6-1, 6-4
- Extended type 4-7, 5-9, 5-10, 5-27, 11-5
- external (directive) 6-6, 9-1, 9-2

**F**

---

- False 5-6, 11-3
- far (directive) 6-5
- fields 5-21 to 5-24, 7-1, 7-7
  - See also* records, classes publishing 7-5
- file (reserved word) 5-24
- file I/O 8-1 to 8-6
  - exceptions 8-3
- file variables 8-2
- FilePos function 8-2
- files
  - as parameters 6-12
  - file types 5-24, 8-2
  - generated 2-2, 2-3, 9-10, 9-11
  - initializing 5-38
  - memory 11-8
  - source code 2-1
  - text 8-2, 8-3
  - typed 5-24, 8-2
  - untyped 5-25, 8-2, 8-4
  - variants and 5-31
- FileSize function 8-2
- finalization section 3-3, 3-5, 12-4
- Finalize procedure 5-19
- finally (reserved word) 7-32
- floating-point types *See* real types
- Flush function 8-4, 8-6
- for statements 4-20, 4-25, 4-26
- form files 2-2, 2-5, 3-1, 7-5, 7-21
- formal parameters 6-19
- forms 2-2
- forward declarations
  - classes 7-6
  - default parameters 6-19
  - interfaces 10-4
  - overloading and 6-9
  - routines 3-4, 6-5
- Free method 7-15
- FreeLibrary function 9-2
- FreeMem procedure 5-38, 9-8, 11-1, 11-2
- functions 3-4, 6-1 to 6-20
  - assembler 13-17
  - calling externally 6-6
  - declaring 6-3, 6-5
  - function calls 4-13, 4-18, 6-1, 6-19 to 6-20
  - nested 5-29, 6-10
  - overloading 6-6, 6-8
  - pointers 4-12, 5-28
  - return type 6-3, 6-4
  - return value 6-3, 6-4

return values in  
  registers 12-3, 13-17  
fundamental types 5-1

## G

-`$G` compiler switch 9-12  
`$G` directive 9-11  
generic types 5-1  
`GetHeapStatus` function 11-2  
`GetMem` procedure 5-26, 5-38,  
  9-8, 11-1, 11-2  
`GetMemoryManager`  
  procedure 11-2  
`GetProcAddress` function 9-2  
getter *See* read specifier  
global identifiers 4-28  
global variables 5-37  
  dynamically loadable  
  libraries 9-6  
  interfaces 10-9  
  memory management 11-2  
`GlobalAlloc` 11-1  
globally unique identifiers *See*  
  GUIDs  
goto statements 4-18  
grammar (formal) A-1 to A-6  
greater-than *See* Symbols  
GUIDs 10-1, 10-3, 10-10  
  generating 10-3

## H

`$H` directive 5-11, 6-15  
`Halt` procedure 12-4, 12-5  
heading  
  program 2-1, 3-1, 3-2  
  routine 6-1  
  unit 3-3  
heap memory 5-38, 11-2  
`Hello world!` 2-3  
`HelpContext` property 7-32  
hexadecimal numerals 4-4  
hiding class members 7-8, 7-11,  
  7-22  
  *See also* overloaded methods  
  reintroduce 7-12  
hiding interface  
  implementations 10-6  
`High` function 5-3, 5-4, 5-12,  
  5-18, 5-20, 6-16  
`HInstance` variable 9-6  
hint directives 4-17  
`$HINTS` directive 4-17

## I

`$I` directive 8-3  
IDE  
  *See also* Delphi  
identifiers 4-1, 4-2, 4-3  
  global and local 4-28  
  in exception handlers 7-30  
  qualified 3-6  
  scope 4-27 to 4-29  
`IDispatch` 10-9, 10-10  
  dual interfaces 10-13  
`if...then` statements 4-22  
  nested 4-22  
`IInterface` 10-2  
immediate values  
  (assembly) 13-12  
implementation section 3-3, 3-4,  
  3-7  
  and forward declarations 6-6  
  methods 7-8  
  scope 4-28  
  uses clause 3-7  
implements (directive) 7-22,  
  10-6  
`$IMPLICITBUILD`  
  directive 9-11  
`$IMPORTEDDATA`  
  directive 9-11  
importing routines from  
  libraries 9-1  
`in` (reserved word) 3-6, 4-10,  
  5-17, 5-33, 9-9  
`Inc` procedure 5-3, 5-4  
incrementing ordinals 5-3, 5-4,  
  5-5  
index (directive) 6-7  
index specifier 7-6, 7-17, 7-20  
index specifier (Windows  
  only) 9-5  
indexes 4-14  
  array 5-18, 5-19, 5-20  
  array properties 7-19  
  in var parameters 5-33, 6-13  
  string 5-11  
  string variants 5-31  
  variant arrays 5-33  
indirect unit references 3-6 to  
  3-7  
inequality operator 4-10  
inheritance 7-2, 7-3, 7-5  
  interfaces 10-2  
inherited (reserved word) 7-9,  
  7-13  
  calling conventions 12-4  
  message handlers 7-16  
`InheritsFrom` method 7-24  
initialization  
  dynamically loadable  
  libraries 9-5  
  files 5-38  
  objects 7-13  
  units 3-4  
  variables 5-37, 5-38  
  variants 5-31, 5-38  
initialization section 3-3, 3-4  
  exceptions 7-28  
`Initialize` procedure 5-38  
inline (reserved word) 13-1  
inline assembler code 13-1 to  
  13-17  
inner block 4-29  
`InOut` function 8-4, 8-5  
input (program parameter) 3-2  
input *See* file I/O  
Input variable 8-3  
`Int64` type 4-7, 5-3, 5-4, 5-10,  
  11-3  
  standard functions and  
  procedures 5-4  
  variants and 5-31  
integer operators 4-6  
Integer type 4-7, 5-3, 5-4  
integer types 5-3  
  comparison 4-11  
  constants 5-40  
  conversion 4-15  
  data formats 11-3  
integrated development  
  environment *See* IDE  
interface declarations 3-4  
  default parameters 6-19  
interface section 3-3, 3-4, 3-7  
  forward declarations and 6-6  
  methods 7-8  
  scope 4-28  
  uses clause 3-7  
interfaces 7-2, 10-1 to 10-13  
  accessing 10-8 to 10-10  
  Automation 10-10  
  calling conventions 10-3  
  compatibility 10-9  
  delegation 10-6  
  dispatch interface  
  types 10-10  
  dual interfaces 10-13  
  freeing 5-39  
  GUIDs 10-1, 10-3, 10-10  
  implementing 10-4 to 10-7

- interface references 10-8 to 10-10
- interface types 10-1 to 10-4
- memory management 11-2
- method resolution
  - clauses 10-5, 10-6
- properties 10-1, 10-4, 10-7
- querying 10-10
- records and 5-23
- typecasts 10-10
- internal data formats 11-2 to 11-12
- intersection (sets) 4-10
- IntToHex function 5-4
- IntToStr function 5-4
- Invoke method 10-10
- IOResult function 8-3, 8-4
- is 4-11, 5-33, 7-24
- IsLibrary variable 9-6
- italics 1-2
- IUnknown 10-2, 10-5, 10-9, 10-13

## J

---

- \$J directive 5-43
- Java 10-1
- jump instructions
  - (assembler) 13-3

## K

---

- .kof files 2-2

## L

---

- labels 4-1, 4-4, 4-18
  - assembler 13-2
- \$LE- compiler switch 9-12
- Length function 5-11, 5-18, 5-20
- less-than *See* Symbols
- libraries *See* DLLs or dynamically loadable libraries
- library (directive) 4-17
- library (reserved word) 9-3
- Library search path 3-6
- line-feed 4-5
- \$LN- compiler switch 9-12
- LoadLibrary function 9-2
- local (directive) 9-4
- local directive (Linux only) 9-4
- local identifiers 4-28
- local variables 5-37, 6-10
  - memory management 11-2
- LocalAlloc 11-1
- locales 5-13

- logical operators 4-8
- long strings 4-9, 5-10, 5-12
  - files and 5-24
  - in dynamically loadable libraries 9-8
  - memory management 11-2, 11-6
  - records and 5-23
- LongBool type 5-6, 11-3
- Longint type 5-4, 11-3
- Longword type 5-4, 11-3
- loop statements 4-20, 4-25
- Low function 5-3, 5-4, 5-12, 5-18, 5-20, 6-16
- \$LU- compiler switch 9-12

## M

---

- \$M directive 7-4, 7-6
- main form 2-5
- \$MAXSTACKSIZE
  - directive 11-2
- members *See* sets
- members, of classes 7-1
  - interfaces 10-2
  - visibility 7-4
- memory 4-1, 5-2, 5-25, 5-26, 5-31, 5-38, 7-14
  - dynamically loadable libraries 9-6
  - heap 5-38
  - management 11-1 to 11-12
  - overlays (in records) 5-23
  - shared memory manager 9-8
- memory references
  - (assembler) 13-12
- message (directive) 7-15
  - interfaces 10-7
- message dispatching 7-16
- message handlers 7-15
  - inherited 7-16
  - overriding 7-16
- Message property 7-32
- Messages unit 7-15
- metaclasses 7-23
- method directives
  - order 7-8
- method pointers 4-12, 5-28
- method resolution clauses 10-5, 10-6
- methods 7-1, 7-2, 7-8 to 7-17
  - abstract 7-12
  - Automation 7-6, 10-12
  - binding 7-10
  - calling conventions 12-3
  - class methods 7-1, 7-25

- constructors 7-13, 12-4
- destructors 7-14, 12-4
- dispatch interface 10-11
- dispatching calls 7-11
- dual-interface 6-5
- dynamic 7-10
- implementation 7-8
- overloading 7-12
- overriding 7-10, 7-11, 10-6
- pointers 4-12, 5-28
- publishing 7-5
- static 7-10
- virtual 7-6, 7-10
- \$MINSTACKSIZE
  - directive 11-2
- minus *See* Symbols
- mod 4-6
- modules *See* units
- multibyte character sets 5-13
  - string-handling routines 8-7
- multidimensional arrays 5-18, 5-20, 5-43
- multiple unit references 3-6 to 3-7
- multiplication 4-6
- multithreaded applications 5-38
  - dynamically loadable libraries 9-7
- mutually dependent classes 7-7
- mutually dependent units 3-7

## N

---

- name (directive) 6-7, 9-5
- named parameters 10-12
- names
  - See also* identifiers
  - exported routines 9-5
  - functions 6-3, 6-4
  - identifiers 4-16
  - packages 9-10
  - programs 3-1, 3-2
  - units 3-3, 3-6
- naming conflicts 3-6, 4-29
- near (directive) 6-5
- negation 4-7
  - bitwise 4-8
- nested conditionals 4-22
- nested exceptions 7-31
- nested routines 5-29, 6-10
- New procedure 5-19, 5-26, 5-38, 7-4, 9-8, 11-1, 11-2
- nil 5-27, 5-30, 5-39, 11-5
- nodefault specifier 7-6, 7-17, 7-21
- not 4-6, 4-7

- Null (variants) 5-31, 5-33
- null character 5-13, 11-6, 11-7, 11-9
- null string 4-4
- null-terminated strings 5-13 to 5-16, 5-27, 11-6, 11-7
  - mixing with Pascal strings 5-15
  - standard routines 8-6, 8-7
- numerals 4-1, 4-4
  - as labels 4-4, 4-19
  - assembler 13-9
  - type 5-40, 6-8

## O

- object files
  - calling routines in 6-6
- Object Inspector 7-5
- Object Inspector (Delphi) 7-5
- object interfaces *See* interfaces, COM, CORBA
- object types 7-4
- objects 4-20, 7-1
  - See also* classes
  - 'of object' 5-28
  - comparison 4-11
  - files and 5-24
  - memory 11-10
- of (reserved word) 4-23, 5-17, 5-19, 5-24, 5-28, 6-15, 6-16, 7-23
- of object (method pointers) 5-28
- Ole Automation 5-34
- OleVariant 5-34
- OleVariant type 5-27, 5-34
- on (reserved word) 7-29
- opcodes (assembler) 13-2
- open array constructors 6-17, 6-19
- open array parameters 6-15, 6-19
  - and dynamic arrays 6-15
- Open function 8-4, 8-5
- OpenString 6-15
- operands 4-6
- operators 4-6 to 4-13
  - assembler 13-15
  - class 7-24
  - precedence 4-12, 7-25
- or 4-7, 4-8
- Ord function 5-3, 5-4
- order of method directives 7-8
- ordinal types 5-2 to 5-9
- ordinality 5-2
  - enumerated types 5-6, 5-7

- out (output) parameters 6-12, 6-13, 6-19
- out (reserved word) 6-11, 6-12, 6-13
- outer block 4-29
- OutlineError 7-32
- output (program parameter) 3-2
- output *See* file I/O
- Output variable 8-3
- overloaded methods 7-12
  - access specifiers 7-13, 7-18
  - publishing 7-5
- overloaded procedures and functions 6-6, 6-8
  - default parameters 6-10, 6-18
  - dynamically loadable libraries 9-5
  - forward declarations 6-9
- overriding interface implementations 10-6
- overriding methods 7-10, 10-6
  - hiding and 7-11
- overriding properties 7-22
  - access specifiers and 7-22
  - Automation 7-6
  - hiding and 7-22

## P

- \$P directive 6-15
- package files 2-2, 2-3, 9-8, 9-9, 9-11
- packages 9-8 to 9-12
  - compiler directives 9-11
  - compiler switches 9-12
  - compiling 9-11
  - declaring 9-9
  - loading dynamically 9-8
  - loading statically 9-8
  - thread variables 9-9
  - uses clause and 9-8
- packed (reserved word) 5-17, 11-8
- packed arrays 4-5, 4-9, 5-18
- packed records 11-8
- packed strings 5-18
  - comparison 4-11
- pairs of symbols 4-2
- PAnsiChar type 5-13, 5-27
- PAnsiString type 5-27
- parameters 5-29, 6-2, 6-3, 6-11 to 6-19
  - See also* overloaded procedures and functions
  - actual 6-19
- array 6-12, 6-15
- array property indexes 7-19
- Automation method calls 10-12
- calling conventions 6-5
- constant 6-13, 12-1
- default 6-17 to 6-19, 10-12
- file 6-12
- formal 6-19
- names 10-12
- open array 6-15
- output (out) 6-13
- overloading and 6-6, 6-8, 6-9
- parameter list 6-11
- passing 12-1
- positional 10-12
- program control 12-1
- properties as 7-17
- registers 6-5, 12-2
- short strings 6-15
- typed 6-12
- untyped 6-14, 6-19
- value 6-12, 12-1
- variable (var) 6-12, 12-1
- variable number 6-6
- variant open array 6-16

- parentheses *See* Symbols
- partial evaluation 4-7
- partitioning
  - application 9-8
- .pas files 2-3, 3-1, 3-3, 3-7
- pascal (calling convention) 6-4, 12-2
  - constructors and destructors 12-4
  - Self 12-3
- PByteArray type 5-27
- PChar type 4-5, 4-9, 5-13, 5-14, 5-15, 5-27, 5-45
  - comparison 4-11
- PCurrency type 5-27
- PDouble type 5-27
- period *See* Symbols
- PExtended type 5-27
- PGUID 10-3
- PInteger type 5-27
- platform (directive) 4-17
- plus *See* Symbols
- Pointer type 5-25, 5-26, 5-27, 11-5
- pointers 5-25 to 5-28
  - arithmetic 4-9
  - character 5-27
  - constants 5-44
  - files and 5-24



- functions 4-12, 5-28
- in var parameters 6-13
- in variant open array
  - parameters 6-17
- long strings 5-16
- memory 11-5
- method pointers 5-28
- nil 5-27, 11-5
- null-terminated strings 5-14, 5-16
- operators 4-9
- overview 5-25
- Pointer type 4-12, 11-5
- pointer types 4-12, 5-26, 5-27 to 5-28, 11-5
- procedural types 4-12, 5-28 to 5-30
- records and 5-23
- standard types 5-27
- variants and 5-31
- POleVariant type 5-27
- polymorphism 7-9, 7-11, 7-14
- positional parameters 10-12
- pound *See* Symbols
- precedence of operators 4-12, 7-25
- Pred function 5-3, 5-4
- predecessor 5-2
- predefined types 5-1
- private class members 7-4, 7-5
- procedural constants 5-44
- procedural types 4-15, 5-28 to 5-30
  - calling dynamically loadable libraries 9-2
  - calling routines with 5-29, 5-30
  - compatibility 5-29
  - default parameters 6-18
  - in assignments 5-29, 5-30
  - memory 11-10
- procedure pointers 4-12, 5-28
- procedures 3-4, 6-1 to 6-20
  - assembler 13-17
  - calling externally 6-6
  - declaring 6-2, 6-5
  - nested 5-29, 6-10
  - overloading 6-6, 6-8
  - pointers 4-12, 5-28
  - procedure calls 4-18, 6-1, 6-2, 6-19 to 6-20
- program (reserved word) 3-2
- program control 6-19, 12-1 to 12-5

- programs 2-1 to 2-5, 3-1 to 3-8
  - examples 2-3 to 2-5
  - syntax 3-1 to 3-3
- project files 2-2, 3-1, 3-2, 3-6
- Project Manager 2-1
- project options files 2-2
- projects 2-5, 3-6
- properties 7-1, 7-17 to 7-23
  - access specifiers 7-17
  - array 7-5, 7-19
  - as parameters 7-17
  - declaring 7-17, 7-19
  - default 7-20, 10-2
  - interfaces 10-4
  - overriding 7-6, 7-22
  - read-only 7-19
  - record 7-5
  - write-only 7-19
- protected class members 7-4, 7-5
- prototypes 6-1
- PShortString type 5-27
- PSingle type 5-27
- PString type 5-27
- PTextBuf type 5-27
- Ptr function 5-26
- public class members 7-4, 7-5
- public identifiers (interface section) 3-4
- published class members 7-4, 7-5
  - \$M directive 7-6
  - restrictions 7-5
- PVariant type 5-28
- PVarRec type 5-27
- PWideChar type 5-13, 5-14, 5-27
- PWideString type 5-28
- PWordArray type 5-28

## Q

- qualified identifiers 3-6, 4-2, 4-29, 5-22
  - in typecasts 4-15
  - pointers 5-27
  - with Self 7-9
- querying (interfaces) 10-10
- QueryInterface method 10-2, 10-5, 10-10
- quotation marks *See* Symbols
- quoted strings 4-4, 5-45
  - assembler 13-9
- QWORD type (assembler) 13-15

## R

- raise (reserved word) 4-20, 7-27, 7-29, 7-30
- range-checking 5-4, 5-5, 5-9
- Read procedure 8-2, 8-3, 8-4, 8-5, 8-6
- read specifier 7-6, 7-17
  - array properties 7-19
  - index specifier and 7-20
  - object interfaces 10-1, 10-4, 10-7
  - overloading 7-13, 7-18
- Readln procedure 8-5, 8-6
- readonly (directive) 10-2, 10-11
- read-only properties 7-19
- real (floating-point)
  - operators 4-6
- Real type 5-10
- real types 5-9, 11-4
  - comparison 4-11
  - conversion 4-15
  - publishing 7-5
- Real48 type 5-9, 5-10, 7-5, 11-4
- \$REALCOMPATIBILITY
  - directive 5-10
- ReallocMem procedure 5-38, 11-1
- records 4-20, 5-21 to 5-24
  - constants 5-43
  - in properties 7-5
  - memory 11-8
  - record types 5-21
  - scope 4-28, 5-22
  - variant parts 5-22 to 5-24
  - variants and 5-31
- recursive procedure and function calls 6-1, 6-4
- reference-counting 5-12, 10-9, 11-6, 11-7
- register (calling convention) 6-4, 7-6, 7-13, 7-14, 12-2
  - constructors and destructors 12-4
  - dynamically loadable libraries 9-4
  - interfaces 10-3, 10-7
  - Self 12-3
- registers 6-5, 12-2, 12-3
  - assembler 13-2, 13-10, 13-12, 13-17
  - storing sets 11-7
- reintroduce (directive) 7-12
- relational operators 4-10

- \_Release method 10-2, 10-5, 10-9
- relocatable expressions (assembler) 13-13
- Rename procedure 8-6
- repeat statements 4-20, 4-25
- requires clause 9-8, 9-9, 9-10
- .RES files 2-2, 3-2
- reserved words 4-1, 4-2, 4-3
  - See also directives
  - assembler 13-7
  - list 4-3
- Reset procedure 8-2, 8-4, 8-5, 8-6
- resident (directive) 9-5
- resource files 2-2, 2-3, 3-2
- resource strings 5-41
- resourcestring (reserved word) 5-41
- Result variable 6-3, 6-4
- RET instruction 13-3
- return type (functions) 6-3, 6-4
- return value (functions) 6-3, 6-4
  - constructors 7-13
- Rewrite procedure 8-2, 8-4, 8-5, 8-6
- Round function 5-4
- routines 6-1 to 6-20
  - See also functions, procedures
  - exporting 9-5
  - standard 8-1 to 8-10
- RTTI 7-5, 7-12, 7-21
- \$RUNONLY directive 9-12
- runtime binding See dynamic methods, virtual methods
- runtime packages 9-8
- runtime type information See RTTI

**S**

---

- \$S directive 11-2
- safecall (calling convention) 6-4, 12-2
  - constructors and destructors 12-4
  - dual interfaces 10-13
  - interfaces 10-3
  - Self 12-3
- scope 4-27 to 4-29
  - classes 7-3
  - records 5-22
  - type identifiers 5-36
- Seek procedure 8-2
- SeekEof function 8-5
- SeekEoln function 8-5

- Self 7-9
  - calling conventions 12-3
  - class methods 7-25
- semicolon See Symbols
- separators 4-1, 4-5
- SetLength procedure 5-11, 5-16, 5-19, 5-20, 6-16
- SetMemoryManager procedure 11-2
- sets
  - empty 5-17
  - memory 11-7
  - operators 4-10
  - publishing 7-5
  - set constructors 4-13
  - set types 5-17
  - variants and 5-31
- SetString procedure 5-16
- setter See write specifier
- shared object files 2-3, 9-1
  - dynamically loadable 9-2
  - exceptions 9-7
  - importing functions 6-7
- ShareMem unit 9-8
- shift-left (bitwise operator) 4-8
- shift-right (bitwise operator) 4-8
- shl 4-8
- short strings 5-3, 5-10, 5-12
- short-circuit evaluation 4-7
- Shortint type 5-4, 11-3
- ShortString type 5-10, 5-12, 5-27, 11-5
  - parameters 6-15
  - variant arrays and 5-33
- ShowException procedure 7-32
- shr 4-8
- sign
  - in typecasts 4-14
  - numerals 4-4
- simple statements 4-17
- simple types 5-2
- Single type 5-9, 11-4
- 16-bit applications (backward compatibility) 6-5
- SizeOf function 5-2, 5-5, 6-16
- slash See Symbols
- Smallint type 5-4, 11-3
- .so files 9-1
- source-code files 2-1
- spaces 4-1
- special symbols 4-1, 4-2
- stack size 11-2
- standard routines 8-1 to 8-10
  - null-terminated strings 8-6, 8-7

- wide-character strings 8-7
- statements 4-1, 4-17 to 4-27, 6-1
- static arrays 5-18, 11-7
  - variants and 5-31
- static methods 7-10
- statically loaded libraries 9-1
- stdcall (calling convention) 6-4, 12-2
  - constructors and destructors 12-4
  - interfaces 10-3
  - Self 12-3
  - shared libraries 9-4
- storage specifiers 7-21
  - array properties and 7-21
- stored specifier 7-6, 7-17, 7-21
- Str procedure 8-6
- StrAlloc function 5-38
- StrDispose procedure 5-38
- streaming (data) 5-2, 7-5
- string
  - See also character sets
  - comparison 4-11, 5-11
  - constants 4-4, 5-45, 13-9
  - handling See also standard routines, null-terminated strings
  - in variant open array
    - parameters 6-17
  - indexes 4-14
  - literals 4-4, 5-45
  - memory management 11-5, 11-6
  - null-terminated 5-13 to 5-16, 5-27
  - operators 4-9, 5-15
  - parameters 6-15
  - types 5-10 to 5-16
  - variant arrays 5-33
  - variants 5-31
  - wide 5-13, 8-7, 11-2
- string (reserved word) 5-11
- StringToWideChar function 8-7
- strong typing 5-1
- StrToInt64 function 5-4
- StrToInt64Def function 5-4
- structured statements 4-19
- structured types 5-16
  - files and 5-24
  - records and 5-23
  - variants and 5-31
- structures 5-21
- StrUpper function 5-15
- subrange types 4-7, 4-24, 5-8
- subset operator 4-10

- subtraction 4-6
  - pointers 4-9
- Succ function 5-3, 5-4
- successor 5-2
- superset operator 4-10
- symbol pairs 4-2
- symbols 4-1, 4-2
  - See also special symbols and Symbols on page I-1*
  - assembler 13-11
- syntax
  - descriptions 1-2
  - formal A-1 to A-6
- System unit 3-1, 3-5, 5-27, 5-31, 5-33, 6-17, 7-3, 7-28, 8-1, 8-7, 10-2, 10-3, 10-5, 10-10, 11-11
  - dynamically loadable libraries 9-6
  - memory management 11-2
  - modifying 8-1
  - scope 4-29
  - uses clause and 8-1
- SysUtils unit 3-5, 5-27, 6-11, 6-17, 7-26, 7-27, 7-28, 7-32
  - dynamically loadable libraries 9-7
  - uses clause and 8-1

## T

---

- \$T directive 4-12
- tag (records) 5-23
- TAggregatedObject 10-7
- TBYTE type (assembler) 13-15
- TByteArray type 5-27
- TClass 7-3, 7-23, 7-24
- TCollection 7-23
  - Add method 7-9
- TCollectionItem 7-23
- TDateTime 5-33
- text files 8-2, 8-3
- Text type 5-24, 8-3
- text-file device drivers 8-4
- TextFile type 5-24
- TGUID 10-3
- then (reserved word) 4-22
- thread variables 5-38, 5-39
  - in packages 9-9
- threadvar 5-38
- TInterfacedObject 10-2, 10-5
- to (reserved word) 4-26
- TObject 7-3, 7-16, 7-24
- tokens 4-1
- TPersistent 7-6
- True 5-6, 11-3
- true constants 5-39

- Trunc function 5-4
- try...except statements 4-20, 7-28
- try...finally statements 4-20, 7-31
- TTextBuf type 5-27
- TTextRec type 5-27
- Turbo Assembler 13-5
- TVarData 5-31, 11-11
- TVarRec 5-27
- TVarRec type 6-17
- TWordArray 5-28
- type identifiers 5-2
- Type Library editor 10-3
- typecasts 4-14 to 4-16, 7-8
  - checked 7-25, 10-10
  - enumerated types 5-8
  - in constant declarations 5-40
  - interface 10-10
  - untyped parameters 6-14
  - variants 5-31
- type-checking (objects) 7-24
- types 5-1 to 5-37
  - array 5-18 to 5-21, 11-7
  - assembler 13-14
  - assignment-compatibility 5-35
  - automatable 7-6, 10-11
  - Boolean 5-5, 11-3
  - built-in 5-1
  - character 5-5, 11-3
  - class 7-1, 7-2, 7-4, 7-6, 7-7, 7-8, 7-17, 11-10
  - classification 5-1
  - class-reference 7-23, 11-11
  - compatibility 5-17, 5-29, 5-35
  - constants 5-40
  - declared 5-1
  - declaring 5-36
  - dispatch interface 10-10
  - enumerated 5-6 to 5-8, 11-3
  - exception 7-27
  - file 5-24, 11-8
  - fundamental 5-1
  - generic 5-1
  - integer 5-3, 11-3
  - interface 10-1 to 10-4
  - internal data formats 11-2 to 11-12
  - object 7-4
  - ordinal 5-2 to 5-9
  - pointer 5-27 to 5-28
  - predefined 5-1
  - procedural 5-28 to 5-30, 11-10

- real 5-9, 11-4
- record 5-21 to 5-24, 11-8
- scope 5-36
- set 5-17, 11-7
- simple 5-2
- string 5-10 to 5-16, 11-5, 11-6
- structured 5-16
- subrange 5-8
- type identity 5-34
- user-defined 5-1
- variant 5-30 to 5-34
- typographical conventions 1-2

## U

---

- UCS-2 5-13
- UCS-4 5-13
- unary operators 4-6
- Unassigned (variants) 5-31, 5-33
- underscores 4-2
- Unicode 5-5, 5-13
- union (sets) 4-10
- UniqueString procedure 5-16
- unit files 3-1, 3-3
  - case-sensitivity 4-2
- units 2-1, 3-1 to 3-8
  - scope 4-29
  - syntax 3-3 to 3-8, 4-17
- until (reserved word) 4-25
- untyped files 5-25, 8-2, 8-4
- untyped parameters 6-14
- UpCase function 5-11
- uses clause 2-1, 3-1, 3-2, 3-4, 3-5 to 3-8
  - interface section 3-7
  - ShareMem 9-8
  - syntax 3-6
  - System unit and 8-1
  - SysUtils unit and 8-1

## V

---

- Val procedure 8-6
- value parameters 6-12, 6-19, 12-1
  - open array constructors 6-20
- value typecasts 4-14
- var (reserved word) 5-37, 6-11, 6-12, 12-1
- varargs (directive) 6-6
- VarArrayCreate function 5-33
- VarArrayDimCount function 5-34
- VarArrayHighBound function 5-34

VarArrayLock function 5-34, 10-12  
 VarArrayLowBound function 5-34  
 VarArrayOf function 5-33  
 VarArrayRedim function 5-34  
 VarArrayRef function 5-34  
 VarArrayUnlock procedure 5-34, 10-12  
 VarAsType function 5-31  
 VarCast procedure 5-31  
 variable (var) parameters 6-12, 6-19, 12-1  
 variable parameters 6-19  
 variable typecasts 4-14, 4-15  
 variables 4-6, 5-37 to 5-39  
   absolute addresses 5-38  
   declaring 5-37  
   dynamic 5-38  
   file 8-2  
   from dynamically loadable libraries 9-1  
   global 5-37, 10-9  
   heap-allocated 5-38  
   initializing 5-37, 5-38  
   local 5-37, 6-10  
   memory management 11-2  
   thread 5-38  
 variant arrays 5-31, 5-33  
 variant open array parameters 6-16, 6-19  
 variant parts (records) 5-22 to 5-24  
 variants 5-30 to 5-34, 11-12  
   and Automation 11-12  
   complete evaluation 4-8  
   conversions 5-31, 5-31 to 5-33  
   files and 5-24  
   freeing 5-39  
   initializing 5-31, 5-38  
   interfaces and 10-9, 10-10  
   memory management 11-2, 11-11  
   OleVariant 5-34  
   operators 4-6, 5-33  
   records and 5-23  
   short-circuit evaluation 4-8  
   typecasts 5-31  
   variant arrays 5-33  
   variant arrays and strings 5-33  
   Variant type 5-28, 5-30  
   variant types 5-30 to 5-34  
 varOleString constant 5-33  
 varString constant 5-33  
 VarType function 5-31  
 varTypeMask constant 5-31  
 VCL 1-2  
 virgule *See* Symbols  
 virtual method table 11-10  
 virtual methods 7-10  
   Automation 7-6  
   constructors 7-14  
   overloading 7-12  
 VirtualAlloc function 11-1  
 VirtualFree function 11-1  
 visibility (class members) 7-4  
   interfaces 10-2  
 Visual Component Library *See* VCL  
 VMT 11-10  
 VMTOFFSET 13-6

**W**

---

\$WARNINGS directive 4-17  
 \$WEAKPACKAGEUNIT directive 9-11  
 while statements 4-20, 4-25  
 wide characters and strings 5-13  
   memory management 11-2  
   standard routines 8-7  
 WideChar type 4-9, 5-5, 5-11, 5-13, 5-27, 11-3  
 WideCharLenToString function 8-7  
 WideCharToString function 8-7  
 WideString type 5-10, 5-13, 5-28  
   memory management 11-6  
 Windows 7-16  
   memory management 11-1, 11-2  
   messages 7-15  
   variants and 11-11  
 Windows unit 9-2  
 with statements 4-19, 4-20, 5-22  
 Word type 5-4, 11-3  
   assembler 13-14  
 WordBool type 5-6, 11-3  
 wrap-around (ordinal types) 5-4, 5-5  
 Write procedure 8-2, 8-3, 8-4, 8-5, 8-6  
 write procedures 5-3  
 write specifier 7-6, 7-17  
   array properties 7-19  
   index specifier and 7-20  
   object interfaces 10-1, 10-4  
   overloading 7-13, 7-18  
 Writeln procedure 2-4, 8-5, 8-6  
 writeonly (directive) 10-2, 10-11  
 write-only properties 7-19

**X**

---

\$X directive 4-5, 4-18, 5-14, 6-1, 6-4  
 .xfm files 2-2, 2-7, 7-5  
 xor 4-7, 4-8

**Z**

---

-\$Z- compiler switch 9-12  
 \$Z directive 11-3



# **Rave Reports**

**Borland Edition 5.0**

## **Developers Guide**

**Tutorial and Reference**

This manual and all material accompanying it is  
Copyright (C) 1995-2002, Nevrona Designs, All Rights Reserved (rev BE50a)



## Table of Contents

<b>Getting Started .....</b>	<b>7</b>
Single User License Agreement .....	7
Limited Warranty .....	7
Technical Support .....	8
<b>Chapter 1 - Introduction .....</b>	<b>9</b>
What's All the RAVE About? .....	9
Report Authoring Visual Environment .....	9
First Glance .....	10
Navigation Area .....	10
The Page (Foundation of Rave) .....	11
Project Tree Panel .....	11
Property Panel .....	12
<b>Chapter 2 - Toolbars and Tool Windows .....</b>	<b>15</b>
Toolbars .....	15
Toolbar Placement .....	15
Toolbar Palette .....	15
Hiding Toolbars .....	16
Component vs. Utility Toolbars .....	16
Tool Windows .....	16
Property Panel .....	16
Project Tree .....	17
<b>Chapter 3 - Components Overview .....</b>	<b>19</b>
What is a Component .....	19
The Component Toolbars .....	20
Standard Components .....	20
Bar Code Components .....	20
Drawing Components .....	21
Report Components .....	21
<b>Chapter 4 - Page Designer .....</b>	<b>23</b>
Overview .....	23
The Basic Component: The Page & it's Panels .....	23
Selecting Components .....	24
Sizing and Moving Components .....	25
Cutting and Pasting .....	25
Exercise: Selecting, Sizing, and Moving Components .....	25
Exercise: Cutting and Pasting .....	26
<b>Chapter 5 - Project Tree Panel .....</b>	<b>27</b>
Overview .....	27
Expanding and Right-Clicking .....	27
Parent-Child Relationship .....	28
Report Library .....	28
Global Page Catalog .....	28



Loading and Unloading Global Pages .....	28
Data View Dictionary .....	29
Ctrl- (Control) Drag and Drop.....	29
Alt-Drag .....	29
Exercise: Navigating the Project Tree.....	29
Exercise: Naming Components.....	30
Exercise: Loading and Unloading Global Pages.....	30
Exercise: Dragging a Component (Ctrl-Drag).....	31
Exercise: Changing the Parent of a Component (Alt-Drag) .....	32
<b>Chapter 6 - Property Panel.....</b>	<b>35</b>
What is the Property Panel.....	35
Types of Properties .....	37
Property Editors.....	37
Right-Click Menu .....	37
Exercise: Navigating the Property Panel.....	37
<b>Chapter 7 - Generating Output.....</b>	<b>39</b>
Overview.....	39
Executing Reports .....	39
Preferences Dialog .....	39
Report Preview .....	40
Executing to the Printer.....	42
NDR & PRN Files .....	42
HTML.....	43
PDF .....	43
RTF .....	43
Exercise: Changing Printing Preferences.....	44
Exercise: Preview and Creating Portable Files.....	45
Exercise: Printing a Report through the Preview .....	47
Exercise: Printing to a File (NDR & PRN) .....	47
<b>Chapter 8 - Utility Toolbars .....</b>	<b>49</b>
Tools are Tools.....	49
Using Tools.....	49
Alignment Toolbar.....	49
Designer Toolbar .....	50
Color Palette .....	51
Font Editor .....	52
Line Editor.....	52
Project Toolbar .....	53
Zoom Toolbar .....	54
Exercise: Aligning Components .....	55
Exercise: Ordering Components .....	57
Exercise: Snapping to the grid .....	59
Exercise: Changing Line Size and Color of a Rectangle.....	60
Exercise: Changing Fonts .....	60

<b>Chapter 9 - Standard Components .....</b>	<b>61</b>
Overview.....	61
Text .....	61
Memo .....	61
Section.....	62
Bitmap and Meta File .....	62
Exercise: Text vs Memo .....	62
Exercise: Section.....	63
Exercise: Placing and Resizing Bitmaps .....	64
<b>Chapter 10 - Drawing Components.....</b>	<b>65</b>
Drawing Component Basics .....	65
Pixels vs Points .....	66
Exercise: Creating Drawing Components .....	66
Exercise: Alignment .....	68
<b>Chapter 11 - Database 101 .....</b>	<b>71</b>
Overview.....	71
What is a database? .....	71
Terms .....	72
Relational Table.....	73
Reporting .....	74
<b>Chapter 12 - Connecting to Data.....</b>	<b>75</b>
Database Connections .....	75
Creating a Database Connection .....	75
Direct DataViews (BE only) .....	76
Driver DataViews .....	76
Status Bar .....	77
<b>Chapter 13 - Wizards.....</b>	<b>79</b>
Wizards.....	79
Exercise: Simple Wizard .....	79
Exercise: Master Detail Wizard .....	80
<b>Chapter 14 - Report Components .....</b>	<b>81</b>
Overview.....	81
Region .....	82
DataBand .....	82
Band .....	82
DataText .....	83
DataMemo .....	84
CalcText .....	85
<b>Chapter 15 - Project Components.....</b>	<b>87</b>
Overview.....	87
Project Manager .....	87
Report.....	88
Page.....	88

Global Page .....	88
Data Objects .....	88
Database Connection.....	88
Security Components.....	89
SQL Data View .....	89
<b>Chapter 16 - Bar Code Components.....</b>	<b>91</b>
Bar Code Component Basics.....	91
Brief Bar Code Descriptions .....	91
<b>Chapter 17 - Advanced Components.....</b>	<b>93</b>
FontMaster .....	93
PageNumInit .....	94
DataCycle .....	94
DataMirror Section .....	95
CalcOp.....	95
CalcController .....	97
CalcTotal.....	97
Exercise: Using Font Master.....	97
Exercise: Setting up PageNumInit for Page Numbering.....	99
<b>Chapter 18 - Adaptable Reports.....</b>	<b>101</b>
Overview.....	101
Anchors .....	101
Waste Fit.....	102
Editor Anchor .....	102
<b>Chapter 19 - Batch and Chain Reporting.....</b>	<b>105</b>
Batch Pages .....	105
Calling Pages .....	105
Chain Pages.....	105
Different First Page Format .....	106
Different Odd/Even Page format .....	106
Batch / Chain Reports .....	106
Exercise: Calling Pages .....	106
Exercise: Chain Pages .....	107\
Exercise: Different First Page .....	108
Exercise: Different Odd/Even Page.....	108
<b>Chapter 20 - Preferences .....</b>	<b>111</b>
Getting to the Preferences Dialog .....	111
Defaults Tab.....	111
Designer Tab.....	111
Environment Tab .....	112
Shortcuts Tab .....	112
<b>Appendix A - Formatting.....</b>	<b>113</b>
AlphaNumeric Items.....	113
Date / Time items .....	114

**Appendix B - Keyboard / Mouse Shortcuts ..... 117**  
Page Designer or Project Tree ..... 117  
Page Designer Only ..... 117  
Project Tree Only ..... 117  
**Appendix C - Property Descriptions..... 119**



## Getting Started

---

# Technical Information

### Single User License Agreement

---

This is a legal Agreement between you, as the end user, and Nevrona Designs. By opening the enclosed sealed disk package, or by using the disk, you are agreeing to be bound by the terms of this Agreement. If you do not agree with the terms of this Agreement, promptly return the unopened disk package and accompanying items, (including written materials), to the place you obtained them for a full refund.

1. **Grant of License** - Nevrona Designs grants to you the right to use one copy of the enclosed Nevrona Designs program, (the Software), on a single terminal connected to a single computer (i.e. CPU). You may make one copy of the Software for back-up purposes for use on your own computer. You must reproduce and include the copyright notice on the back-up copy. You may not network the Software or use it on more than a single computer or computer terminal at any time, unless a copy is purchased for each computer or terminal on the network that will use the Software. You may transfer this Software from one computer to another, provided that the Software is used on only one computer at a time. You may not rent or lease the Software, but you may transfer the Software and accompanying written material and this license to another person on a permanent basis provided you retain no copies and the other person agrees to accept the terms and conditions of this Agreement. THIS SOFTWARE MAY NOT BE DISTRIBUTED, IN MODIFIED OR UNMODIFIED FORM, AS PART OF ANY APPLICATION PROGRAM OR OTHER SOFTWARE THAT IS A LIBRARY-TYPE PRODUCT, DEVELOPMENT TOOL OR OPERATING SYSTEM, OR THAT MAY BE COMPETITIVE WITH, OR USED IN LIEU OF, THE PROGRAM PRODUCT, WITHOUT THE EXPRESS WRITTEN PERMISSION OF NEVRONA DESIGNS. This license does include the right to distribute applications using the enclosed software provided the above requirements are met.

2. **Term** - This Agreement is effective until you terminate it by destroying the Software, together with all copies. It will also terminate if you fail to follow this agreement. You agree upon termination to destroy the Software, together with all copies thereof.

3. **Copyright** - The software is owned by Nevrona Designs and is protected by United States laws and international treaty provisions. Therefore, you must treat the Software like any other copyrighted material (e.g. a book or musical recording) **EXCEPT** that you may either (a) make one copy of the Software solely for back-up or archival purposes, or (b) transfer the Software to a single hard disk provided you keep the original solely for back-up or archival purposes. You may not copy the written materials accompanying the Software.

### Limited Warranty

---

1. **Limited Warranty** - Nevrona Designs warrants that the disks on which the Software is furnished to be free from defects in material and workmanship, under normal use, for a period of 90 days after the date of the original purchase. If, during this 90-day period, a defect in the disk should occur, the disk may be returned with proof of purchase to Nevrona Designs, which will replace the disk without charge. Nevrona Designs warrants that the Software will perform substantially in accordance with the accompanying written materials. Nevrona Designs does not warrant that the functions contained in the Software will meet your requirements, or any operation of the Software will be uninterrupted or error-free. However, Nevrona Designs will, after being notified of significant errors during the 90-day period, correct demonstrable and significant Software or documentation errors within a reasonable period of time, or refund all or a fair portion of the price you have paid for the Software at Nevrona Designs' option.

2. **Disclaimer of Warranties** - Nevrona Designs disclaims all other warranties, either

**expressed or implied, including but not limited to implied warranties of merchantability of fitness from particular purpose, with respect to the Software and accompanying written materials. This limited warranty gives you specific legal rights, you may have others, varying from state to state. Nevrona Designs will have no consequential damages. In no event, shall Nevrona Designs or its suppliers be liable for damages whatsoever, (including without limitation, damages for loss of business profits, business interruption, loss of business information, or any pecuniary loss), arising out of the use or the inability to this Nevrona Designs product, even if Nevrona Designs has been advised of the possibility of such damages. Some states do not allow the exclusion of limitation of liability for consequential or incidental damages, and this limitation may not apply to you.**

**3. Sole Remedy** - Nevrona Designs' entire liability in your inclusive remedy shall be, at Nevrona Designs' option, either: (1) The return of the purchase price paid; or (2) Repair or replacement of the Software that does not meet Nevrona Designs' limited warranty, which is returned to Nevrona Designs with a copy of your receipt.

**4. Governing Law** - This Agreement will be construed and governed in accordance with laws of the State of Arizona.

**5. U.S. Government Restricted Rights** - This Software and documentation are provided with restrictive rights. Use, duplication or disclosure by the Government is subject to restrictions set forth in Section c(1)(ii) of the Rights and Technical Data in Computer Software clause at 52.227-7013.

## **Technical Support**

---

Technical support, product updates, addons and other information relating to Rave Reports can be found at the Nevrona Designs web site. Please visit one of the following web pages for more information:

Technical Support - <http://www.nevrona.com/support>

Addons, Tips and Tricks and other information - <http://www.nevrona.com/rave>

Updates - <http://www.nevrona.com/rave/download.html>

## Chapter 1

# Introduction

### In this Section:

- Rave is introduced and related to reporting needs.
- Navigation of toolbars is introduced.
- Introduction of the main component, the Page.
- Brief overviews of the Toolbars, Project Tree Panel, and Property Panel are given.

## What's All the RAVE About?

Reporting can be one of the most complex, yet most important tasks for anyone dealing with a database. Reports are the primary visual means to express information retrieved from a body of data. To solve the problems associated with presenting a visual report of data in a meaningful and informative manner, traditional visual reporting tools have offered banded layout tools geared towards table-style listings of data. Today, however, much more complex reporting requirements exist and are not easily handled by banded layout tools.

Welcome to the next level in visual reporting! The Rave visual designer offers many unique features that help to make the reporting process simpler, quicker and more efficient. Rave is an intuitive page based visual design environment that can easily handle a wide variety of report formats, much more than a purely banded style tool. Rave also includes mirroring and other technologies to encourage you to reuse the contents of your reports for quicker changes and easier maintenance. In general though, Rave has been designed to offer the most flexibility and functionality in an easy to learn format.

Where do you begin? Since Rave is page-based designer, many of its features should make sense and be easy to use with only a little practice. There are a lot of options and some might not be obvious when just starting. However, remember that many of these options can be ignored in the beginning, but as your needs and knowledge increases these options are readily available. In fact, Wizards generate "standard" reports without having to know behind-the-scene details. However, we do recommend that you take some time and do a quick read of this manual.

Included with the Rave installs is a project called "RaveDemo" that contains several report samples. To see several common designs, start the RaveDemo.exe and open the RaveDemo project to access the different report types. Exploring the RaveDemo project and other samples are excellent ways to learn Rave.

## Report Authoring Visual Environment

A report might be described as data presented in a visual manner, whether it is printed on paper or displayed electronically. Typically, there are sets of database tables that provide data to create a report. For example, suppose there are: a Customer table, a Products table, and an Items Sold table. These tables could be combined to produce form letters, invoices or customer lists.

Let's begin with a quick overview of Rave. The first step to using Rave is to start the program. The first thing seen will be a sheet representing a page on the screen as well as two windows on the side of the page and toolbars across the top of the page. On a first impression, there appears to be many items in the Rave designer, so where do you start?

Let's start with what is shown when Rave is running. First, understand that there are two groups



of toolbars displayed in the Rave designer; they are the components and tools.

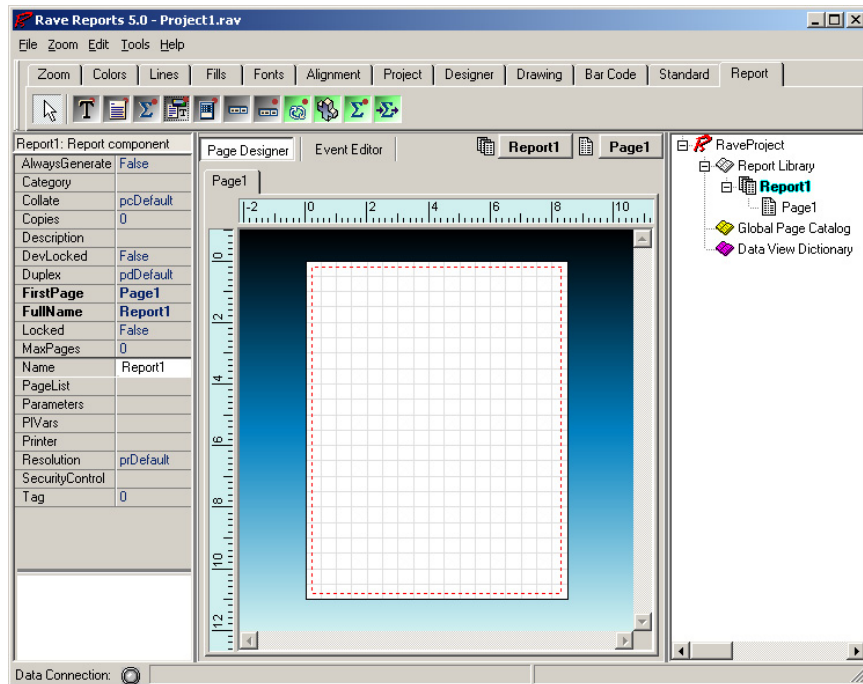
Components are those items that are "dropped" or visible on the page editor. These might be bands, bar codes, lines, shapes, etc. Don't worry about what these things are yet. If the object can be seen on the page layout, it is a component.

Toolbars have the ability to change or modify components, thus distinguishing them as tools. There are several toolbars: alignment, color palette, font editor, etc. If there is a box on the page and it needs to be filled with a background color, first select that item (the box) by clicking on it. Then, use the color tool to change the fill color to any desired color. Toolbars can be hidden and seen by selecting them from the Tools menu. If there is a preference for a more simplified designer look, the toolbars can be hidden until they are needed.

Now that was the very fast overview. There are lots of settings or properties that control the behavior of almost every part of the visual designer. This manual has two main parts, a description of the Rave system including the components and tools, then a reference section that lists the details about each property. Go through the description section first, and then browse the reference section to get a better understanding on just how much control can be had over various design features.

## First Glance

When Rave first starts, it will open a screen that looks like the image below. Since Rave remembers what file was last opened, your screen may look a little different. But, the areas shown are the fundamental pieces. We are going to first explore each part of this screen to give a general tour of the Rave Designer.



## Navigation Area



The top of the visual designer is the Rave Navigation area. This is where toolbars can be docked individually, and can also be tab docked. Individually docking the toolbars will allow the user to see all the buttons on the toolbar and to see more than one toolbar at a time. Tab docking will allow easy access to each toolbar, but only one toolbar can be seen at a time.

The toolbars contain both tools and components. A tool is a feature that will be used to modify an object already on the page, like the Font Editor or the Color Palette. A component is an object that will be placed on the page like a band, line, text, region, or section.

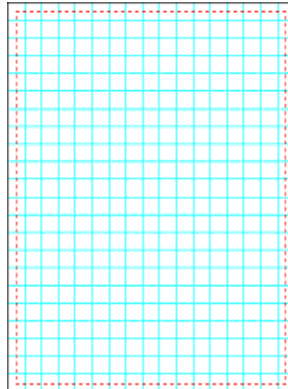
To find out the name of the button control on a toolbar, simply move the mouse pointer over a button, and a popup window will appear with the name of the tool or component. Also, notice the raising of the button when it is about to be selected. It will become 'depressed' when the button is clicked on.

Each toolbar can also be hidden. To hide or view a toolbar, go to the tool menu and select the desired toolbar from Toolbars menu.

## The Page (Foundation of Rave)

---

The starting point with the Rave Visual Designer is the Page. The page is the foundation and is where all the designing action is done. The Page is represented with the grid pattern, which will look something like the image shown to the right. The look and feel of the Page can be changed with the preference settings, which will be covered in Preferences chapter.



One important thing to know and remember is that the Page has properties, such as height and width. To see or change the Page properties, go to the "Project Tree" Panel and expand the Report node by clicking on the "+" sign, then click on the report name (default is "Report1") and this will show a line that will be defaulted to "Page1". Click once on the Page reference and the Page name will be highlighted in the "Project Tree". The highlight means that the Page has been selected and now the Page properties can be seen in the Property Panel.

## Project Tree Panel

---



The Project Tree panel is a very informative part of the Rave designer and it also provides an easy way to navigate the reporting project structure.

For now just the parts of the Project Tree will be examined. More details are covered in the Project Tree Chapter.

There are three main nodes in the Project Tree: Report Library, Global Page Catalog, and Data View Dictionary. Each of these nodes (and any sub nodes) can be expanded or collapsed by clicking on the plus/minus symbol. Sub-nodes can be created and added, by selecting a desired option (New Report, New Global Page, and New Data Object) from the Project Menu.

The Report Library node is where all of the reports within the project are contained. Each report will have one or more pages. Each of those pages will normally have one or more components within them.

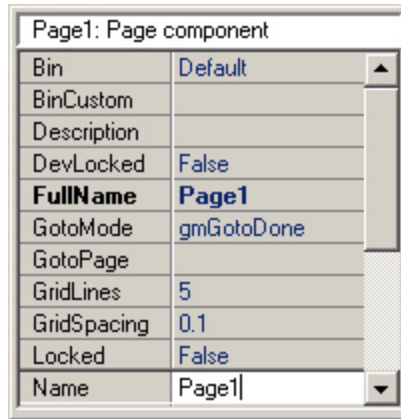
The Global Page Catalog node is where reporting templates are managed. The reporting templates can contain one or more components. These reporting templates can then be reused via Rave's unique mirroring technology. This could include items such as letter headings and footers, pre-printed forms, watermark designs or complete page definitions that could be the foundation for other reports.

The Data View Dictionary node is where all the data connections for reports are defined. A data view retrieves data from the application through data connections installed within that application.

## Property Panel

---

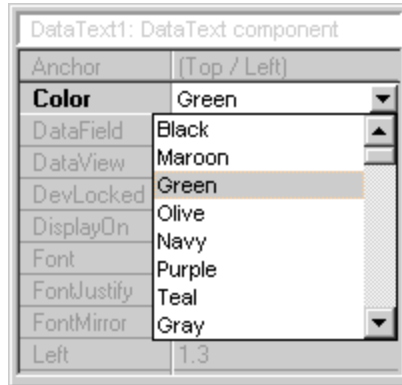
The Property Panel helps to customize the way components appear or behave. When a component is selected on the page, the Property Panel will reflect the selection by displaying the different properties associated with the selected component.



Changing the properties values is easily done by using various associated drop-down menus and edit boxes. If no component is selected, then the property panel will appear blank with no options to choose from.

Another way to change a property is select a value from a list of possible choices. For example, the Color property has a down arrow button. Clicking on the down arrow button will display a list

of colors that can be selected. Any property that has a list of choices can also be double-clicked (instead of clicking on the down arrow button and selecting the option) to advance to the next item in the list.





---

# Toolbars and Tool Windows

In this Section:
<ul style="list-style-type: none"> <li>• General information about the toolbars</li> <li>• Toolbar placement in the Visual Designer environment</li> <li>• How to manipulate Toolbars within the environment</li> <li>• Introduction to Tool Windows and their movement</li> </ul>



## Toolbars

---

A toolbar is a collection of icons that relate in their function. Rave has several toolbars that can be docked to any side of the designer window, tab docked, or hidden to stay out of the way until they are needed.

To access the toolbars, go to the Tools menu, and then from Toolbars select the toolbar that is desired. Selecting a toolbar will place a check mark next to the toolbar.

The toolbars are designed to remember their last placement position. When a toolbar is closed and then opened again, the toolbar will be in the same position on the screen that it was in before it was closed.

## Toolbar Placement

---

When a toolbar is visible it can be in one of two states: tab-docked or single docked. A tab-docked toolbar is docked with tabs displayed. A single docked toolbar has a ribbed section on the left of the toolbar. To change the toolbar state, right-click on one of its icons. This will reveal a small button with either a "dock" or "undock" label. Click on that button to change that toolbar's state.

When a toolbar is undocked, the "toolbar handle" is the ribbed section in docked style. To move a toolbar, place the mouse cursor somewhere on the "toolbar handle", press and hold the left mouse button while dragging the toolbar to a new location in the top portion of the designer window. The best way to get familiar with this is to try docking and undocking toolbars.

Below is an example of docked and tab docked. As previously mentioned, the single docked toolbars can only be placed at the top of the designer window. The user can practice docking and undocking the toolbars to get familiar with this process.

## Toolbar Palette

---

The toolbar palette has a number of special features, which are worth mentioning in more detail. The region on the upper right hand corner of the visual designer and allows one or more toolbars to be docked. When the toolbars are docked, tabs are formed with the name of the toolbar as the name of the tab. As toolbars are placed in the tab docking area, a new tab will be created to reflect the current toolbar.

There is a tab grabber that allows the tabs to be moved to the left or to the right. When the tabs are moved to the far right, double arrows appear to allow movement through the tabs. The entire tabs area can be emptied by moving all the toolbars away from the tab area, or by closing the toolbars from the tool menu. The tab area can also be moved off to the every edge of the screen (moved to the far right) to allow room in the upper area of the visual designer. Use the Tab Grabber to move the Tab area (it can be filled or emptied) off to the far right.

## Hiding Toolbars

---

There are a couple ways to hide toolbars. Hiding Toolbars allow the user to 'clean' up the Rave Designer. This is also helpful to get unused toolbars, or less frequently used toolbars, out of the way or out of the designer environment.

The method to do this is to use the Tools menu to uncheck, and thus hide, the toolbar.

## Component vs. Utility Toolbars

---

There are two types of toolbars: Component toolbars and Utility toolbars.

The component toolbars are used for placing components that work and have function within a report; see the Components Overview Chapter for more details about components. Most of the components appear visually when placed on the page, however there are a few components that are non-visual. Non-visual means that the components only appear in the Project Tree, there will be no visual record of the component on the Page.

Utility toolbars are used to set properties that affect the look of components in a report. For example, the Utility Toolbar Color can change the color of text in a DataText component.

Below is a list of the toolbars classified by type:

### Component Toolbars

Standard  
Report  
Bar Code  
Drawing

### Utility Toolbars

Alignment  
Colors  
Fills  
Fonts  
Lines  
Zoom  
Project  
Designer

## Tool Windows

---

Rave has two very helpful windows, which are generally called the Tool Windows. They are called this because the user will use them as tools to complete and navigate a report. The two Tool Windows are the Project Tree and the Property Panel.

These windows will most likely be displayed when Rave is started. But, if it is not displayed, then go to the Tools menu and select these windows to be opened by placing a check mark in front of them.

## Property Panel

---

The Property Panel gives information about the characteristics of a project and the components within the project. It will be most helpful when designing reports because it gives easy access to component properties, and thus easy visual access to modifying the properties. More detailed information is given in the Property Panel Chapter.

The Property Panel displays and allows modification of values of any component that is placed on the Page, which includes non-visual components. To get property information about a certain component, remember to select that component before looking at the Property Panel (see Selecting a Component in the Page Designer Chapter for more information on how to select an object).

## Project Tree

---

The Project Tree is a very informative part of the Rave designer. It provides an easy way to navigate the reporting project structure.

The Project Tree is composed of three main branches: Report Library, Global Page Catalog and Data View Dictionary. For more information about the functionality of each of these components, please see the Project Tree Panel chapter and the Project Components chapter.

Items within the Project Tree can be selected by clicking on the item. Also, remember that the components seen in the tree are also the components that are on the Page.

**Note:** the non-visual components are in the Project Tree and are usually a green color.

The details of navigating through the Project Tree are saved for a chapter in itself. For now just a general overview of the Project Tree is given.





# Components Overview

## In this Section:

- Introduction to Components in general
- Lists Component Toolbars
- Overview of each Component toolbar is given

## What is a Component

A component is defined as something placed on the page, such as a barcode, line, region, shape, etc. The components available in Rave can be found on any of the component toolbars (e.g. Standard, Drawing, Report and Barcode).

The toolbars are made available by clicking on the Tools menu followed by the Toolbars menu. The available toolbars will then be shown in another submenu and will have checkmarks showing the toolbars that are currently visible. Once a component toolbar is active and selected, a component can be selected and placed on the page. The Page is a special base component, and more details are given in the Page Designer chapter.



Special properties are associated with each component. These component properties can be seen using the Property Panel. Set the properties of each component to the desired setting by either typing the setting in a text dialog, using a drop down menu, or by using the special ellipse (...) button to get to the property dialog box.

There are many properties associated with each component, but don't be intimidated by the number of properties. The properties are there to allow adjustment for a component's behavior and in many cases the default settings are adequate. Also, please note that the number of properties listed with each component may vary depending on the user level that has been set under preferences. To adjust the user's level, please visit the environment tab in the preferences dialog. See the Preferences chapter for more details.

Since there are many properties associated with each component, this chapter will focus mainly on the component toolbars rather than their associated properties. The current chapter provides a good overview of what each component toolbar does without too much detail about the property specifics. Do note that many components share common properties, so once the common properties are learned for one component, they can be applied to other properties.

Components are also defined by their relationship relative to other components. This relationship is defined by a parent-child relationship. When a text component is placed on the page, the parent is the page component and the child is the text component. Another way to look at it is that the page contains the text component, thus the parent component contains the child component.

The parent-child relationship also extends into the positioning of the components. All positions are relative to the upper left corner of the parent, thus the Left property and Top property are used to define the relative position of a component. If the parent is like a Section component, which can contain any number of other components; then as the parent component is moved around, it's children components will move accordingly. If the parent component is deleted, all of its children will also be deleted.

A parent-child relationship is defined when the child component is initially created. If the child component is dropped within the parent component, then it will become the child of that parent. If the component is dropped on the page, then dragged on top of another component (such as a

Band or Section), it will still be considered a child of the page.

The Project Tree, which will be explained in more detail in a later chapter, visually shows the parent-child relationships of the components. In general, the Project component is the master parent of all reports, global pages and data objects. Reports are parents to the report pages. Pages are parents to the components that are placed on them. There are still other components that can be parents to components, such as Regions, Bands and Sections.

## The Component Toolbars

---

There are four standard component toolbars. Other component toolbars may be present if add-on packages have been included. The normal set of component toolbars is: "Standard", "Drawing", "Report", and "Bar Code". There is one particular property that applies to all components, so it will be mentioned here.

The "Name" property is used to assign a new name to that control or to designate the name of the control to give it a unique meaning. By default, Rave assigns sequential names based on the type of the component, such as 'Section1', 'Section2', and so on. Change these to more meaningful names to make the Project Tree more readable and the reporting project easier to maintain. The "Name" property must not contain any spaces or special characters. This is the name that will be used in the Project Tree Panel.

The next several sections will give an overview of the four component toolbars. This will serve as an introduction, and more detail will be covered in later chapters.

## Standard Components

---



The Standard Components toolbar controls the visibility of most frequently used components. Once the Standard components toolbar is active (either docked or undocked in the Rave environment), select and place components such as a bitmap, section, text control, etc. on the page.

The Text component creates a single line of text, which can be changed using the Property Panel.

The Memo component allows several lines of text to be displayed, including multi-line text.

The Section component is a very special component that allows several components to be grouped together.

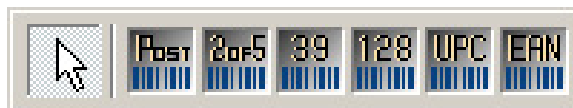
Bitmap and Metafile components allow images to be placed onto the report. The FontMaster component allows the user to define standard fonts for different parts of the report, like the headers, body, and footers.

The final component is called PageNumInit, this component allows the restart of page numbering within the report.

Use these components to create the base of a report, by inserting text and images.

## Bar Code Components

---

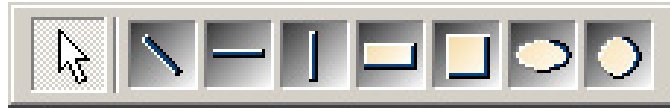


The BarCode Components toolbar contains six bar code components: PostNetBarCode, 2of5BarCode, Code39BarCode, 128BarCode, UPCBarCode, and EANBarCode.

These are used for making all sorts of bar codes for report usage.

## Drawing Components

---



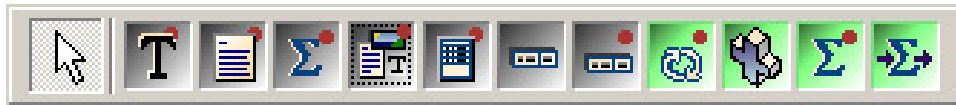
The Drawing toolbar controls all graphical drawing components. This includes drawing lines, boxes, rectangles, circles and elliptical circles.

The components from left to right are: Line, HLine, VLine, Rectangle, Square, Ellipse, and Circle.

These are used for creating graphical images and general report formatting. Use these to separate areas of a report, or to create an informative image.

## Report Components

---



There are several report components: (left to right) DataText, DataMemo, CalcText, DataMirrorSection, Region, Band, DataBand, DataCycle, CalcOp, CalcTotal, CalcController. Use these components to interact with a database and make a functional report.

The components with a red dot in right corner are Data Aware and are capable of displaying information from a database. Each component has a DataView property, which allows the component to interact with a database.

The components with a green color are "non-visual". These can be seen in the Project Tree, but not on the Page layout.

Bands and DataBands must be placed within a Region. Therefore before placing a band on the Page, first place a Region in the area in which a Band or a DataBand will be established. There is no limit on the number of Bands within a region, nor is there a limit to a single region. There can be as many regions as needed. Each Band has its own set of properties to control its behavior. See the Report Components Chapter for more detailed information.



---

# Page Designer

## In this Section:

- Introduce the Page as a basic component
- Discuss how the Project Tree and Property Panel relates to the Page
- Show how to select component(s)
- Learn how to re-size and move components
- Overview of cutting and pasting in Rave

---

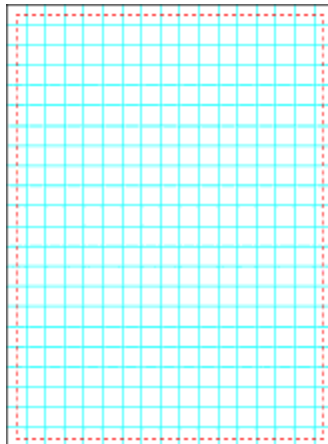
## Overview

Just knowing the components and tools is not enough to work effectively in Rave. One must also learn about the most fundamental component of Rave, which is the Page. In this chapter the Page and its relation to the Project Tree and Property Panel are explained in detail. Although the Project Tree and the Property Panel have their own chapter, as each are important individually, this chapter explains how each is related to the Page. Also in this chapter, some basics on moving objects on the Page are given.

---

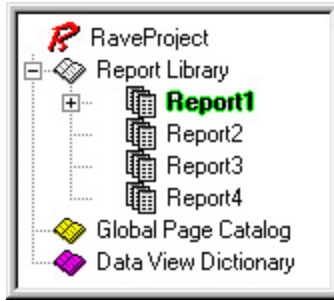
## The Basic Component: The Page & it's Panels

The base component of the Rave Visual Designer is the Page. The Page is not a normal component in that it is not on a Component Toolbar, but it does have properties and it will help to think of it as the base component that provides the container for the report design. All Rave components must be placed on the Page. The Page is represented by the grid pattern on the page panel. The look and feel of the Page is controlled by preference settings that are covered in the Preferences Chapter. Changing the visual look of the Page is discussed in the Utility Toolbars Chapter in the Designer Toolbar section.



Like all components, the Page also has properties such as height, width, description and name. To see or change the Page properties, go to the "Project Tree" panel and expand the Report node by clicking on the "+" sign, then click on the report name ("Report1" is the default report). Clicking on the report name will show "Page1", which is the default name for a Page. Click once on this and notice that "Page1" will be highlighted. This means that the Page has been selected

and now the Page properties for that Page will be seen in the Property Panel.



Page properties allow a report to be more informative. For example, rather than have a report with Page1, Page2, Page3, it might be desirable to have the pages named Invoice, PO, PackingSlip. To do this, use the "Name" property in the Property Panel. The name can be anything the user desires, but note that the name cannot have any spaces or special characters. Remember the Name is for the Rave designer only and is visible only in the Project Tree and on the tab(s) of the Page panel.

In the image of the Project Tree and Property Panel, the page names have been changed to show how useful it is to rename the Page. Now the project tree looks more informative. It is useful and helpful to adjust any properties associated with the Page to make the report more informative and useful for future use and revisions.

## Selecting Components

---

Selecting components is as easy as clicking on them. Moving the mouse pointer to the region where the component is and clicking the left mouse button once will select any component that is placed on the Page. A selected component will appear surrounded with a line that has colored dots, called pips.

The "pips" can be of four different colors: green, gray, red, and yellow. Green pips indicate that the current component is the only one selected. Gray pips show that there is more than one component selected, and red pips indicate that the current component is locked. The yellow pips indicate objects that are that have been mirrored and cannot be modified.

Selecting more than one component at a time is done in a similar way to selecting one component. The difference is that when selecting multiple components; hold down the Shift key while clicking on the objects.

Another way to select either one or more components is to draw a "virtual" box around them. To do this, select an area that is adjacent to the component and drag the mouse to surround the component(s) keeping the left mouse button clicked.

To unselect a component, simply click on the Page Designer tab or move the cursor selection to a different component by clicking on it.

When multiple components are selected, only the common properties will be displayed in the Property Panel. When a property changes, the change will affect all the selected components at that time.

There is a special property for all components called Locked. If the Locked property is True, the component can be selected, but it's properties cannot be changed (except for the Locked property, of course). If selected components are locked, then the pips and the property name for those items will be red (as mentioned previously). If the component is a child component to a parent component, the parent will control the locked access. The Locked property state drop-down menu will show either a True/False indicating the lock status or it will show the parent component name that is controlling its lock state. The remaining discussion is about components

that are NOT locked.

## Sizing and Moving Components

---

To move a component, simply click on it with the left mouse button and drag it to the desired area. When more than one component has been selected, they can all be moved by just moving one of the components. Note that this will NOT affect components that have the Locked property set to True.

The main purpose of the "pips" that appear when selecting a component are for resizing. Resizing only works on ONE component at a time, i.e., a group of selected components cannot be resized at one time. A component can either be resized vertically, horizontally or diagonally. If the resizing is done diagonally, the component will resize both vertically and horizontally. Simply click on one of the pips with the left mouse button and drag the pip until the desired position. Locked components cannot be resized.

## Cutting and Pasting

---

Cutting and pasting is achieved in the same way as in any other standard Windows applications. After selecting a component (or group of components), choose Cut or Copy from the Edit menu; or use the Ctrl-X or Ctrl-C key combinations; or right click on the component and selecting Cut or Copy from the popup menu.

To paste the components, choose from any of the three previous methods (Edit menu, Ctrl-V or popup menu).

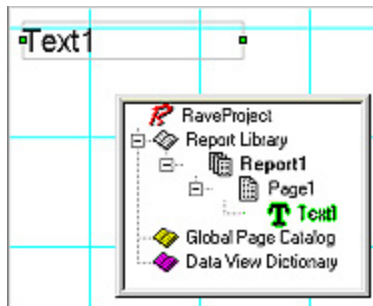
Again it is important to point out that Locked components cannot be cut since this would involve removing the object from its position.

## Exercise: Selecting, Sizing, and Moving Components

---

### To select a component:

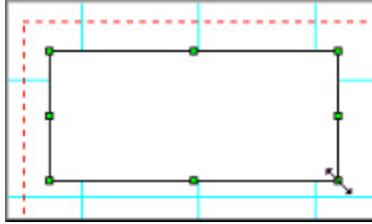
1. Click on the component. The pips and border will appear around it
2. Or, click on the component in the Project Tree, and notice that the selected component will be highlighted.



### To resize a component:

1. Select the component
2. On a pip, place cursor over it until it changes to a double arrow
3. Click and hold the pip, then drag to the desired size





**To move a component:**

1. Select component
2. Move cursor over component (not near a pip, as that would be resizing the component). Click on component and hold
3. Move component by dragging cursor, and thus the component
4. Place in desired location.
5. Release cursor

## Exercise: Cutting and Pasting

---

**Using Edit menu:**

1. Select the component.
2. Select Cut from the Edit menu. This will place the component into the clipboard, and the component will disappear visually from the page.
3. Paste by selecting Paste from the Edit menu.

**Using the short-cut keys:**

1. Select the component.
2. Press Ctrl-key and X-key. This will place the component into the clipboard, and the component will disappear visually from the page.
3. Paste by pressing Ctrl-key and V-key.

**Using the pop-up menu:**

1. Select the component.
2. Right-click on the component, in the pop-menu select Cut. This will place the component into the clipboard, and the component will disappear visually from the page.
3. Paste using one of the two previously mentioned ways.

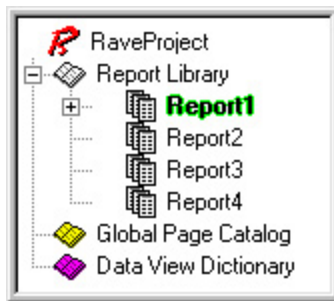
# Project Tree Panel

## In this Section:

- Project Tree structure introduced
- General discussion of Parent-Child relationships
- Introduce the main project categories
- Show How to Load/Unload Global Pages
- Discuss and explain the Drag and Drop features

## Overview

The Project Tree Panel provides an easy way to navigate through the report design structure. This tree style view gives an outline overview of the report pages and the report structure. Thus, this makes the Project Tree a very informative part of the Rave designer. Once you have designed some simple or complex reports, remember to visit this panel and see just how informative it can be.



A project can contain multiple definitions within each main category: Report Library, Global Page Catalog and Data View Dictionary. Each node (and any sub nodes) can be expanded or collapsed by clicking on the plus/minus symbol. Any of these main nodes can be added from the "Project" menu option. There you will see an option for each of the nodes "New Report", "New Global Page" and "New Data View". In order to add a "New Page" to a Report in the Report Library, activate the Report that and use "New Page" from the Project menu.

## Expanding and Right-Clicking

An easy way to complete tasks on items in the project tree is to right-click on the item. There are many different pop-up menus available. Clicking on a Main Node like the Report Library brings up a popup menu with two choices: Expand All and Collapse All. These two choices refer to expanding and collapsing all the nodes, which would provide easy fast access to all items in that tree.

A right-click on a sub-node, like on a Page, would reveal the option of deleting a page. Some of the available functions may be few or many, as in the next image.

Right-clicking on a component can display a pop up menu with many options. These options are the same options available as if a right-click was completed on the component on the designer page. Also, these items reflect the options available to the component through the Utility toolbars.

There are many options that may be available with a right-click of the mouse. To speed up the report making process or to make navigating a report fast and easy try to become familiar with the options that are available through these popup menus.

## Parent-Child Relationship

---

Now, that you have seen how to click through the Project Tree, we'll review some lingo that is associated with a "tree".

The Project Tree is listed in a way that looks like an upside down tree. The base or the root of the tree is at the top of the list. Each subsequent branch is a child of the root. This makes the root also the parent of the child or branch. Each branch can also be its own parent and have its own child or children.

The techniques used in the previous section demonstrated how to move through the tree by expanding and collapsing the nodes associated with each parent. When a branch has children, a node will appear to the left of the branch name.

The Report Library is the root of the Tree, and has one child called Report1, shown with the text highlighted.

Report1 has several children: Page1, Page2, Page3, Page4, Page5, and Page6.

Page1 has the following children: Text1, Text2, and Memo1.

Also there are two sibling groups. Just like in a real family with parents and children, the children can have siblings. The first group of siblings: Page1, Page2, Page3, Page4, Page5, and Page6. The second group of siblings: Text1, Text2, and Memo1. There are other family associations, but the main two that we are concerned with is the Parent-Child relationship.

## Report Library

---

The Report Library node is where the design structure, including all the components used, in each of the report(s) will be displayed. Normally, a component is selected via the Page in the Visual Designer layout panel by simply clicking on the component. However, the Report Library tree can also be used to select components by simply clicking on the component name. When a component name is clicked, that name will be highlighted. To select multiple components, hold the Shift- key down while selecting as many components as you like in the Project Tree.

A project can be simple with one or two report definitions, or it can be complex with many report definitions. Each report that is created would have it's own design structure. To select and switch to a different report, simply double-click on that report name in the Report Library tree structure.

TIP: To make navigation and understanding of a report easier, make use of the Name property. See the Name Exercise at the end of this chapter for help in using this property effectively.

## Global Page Catalog

---



The Global Page Catalog node is where Page definitions, which contain templates to be mirrored, are kept. This includes things like Letterheads, Forms, Watermark designs, and other Page definitions that could be the foundation for several reports.

This can even be a complete report design. An example would be mirroring a section component on a Global Page where the same contents (like an "Invoice") could be printed, but also with a different caption at the bottom of the pages like "Original", "File Copy", and "Shipping".

## Loading and Unloading Global Pages

---

Global Pages are very useful and helpful to the designer, especially when they can be accessed quickly. These pages can hold all sorts of objects and components that can be repeated and mirrored throughout many pages in many different reports. Many of the reports in a Report Library could make use of any of the Global Pages in the Global Page Catalog.

When creating reports, a useful feature of the Rave Designer is the Page Designer Tabs. These are located at the top of the Page Designer window. They are the tabs of pages in use.

Global Pages can be added to this tab for quick referencing, meaning that the designer does not need to scroll through the whole Project Tree to find the right Global Pages frequently needed. Instead the designer just needs to load the page to the Page Designer Tab area and click on the tab when it is needed. Global Pages can be loaded and unloaded to/from the Page Designer Tab area by performing a right-click on the Global Page and using the speed menu to make their choice. A fast way to unload a Global Page is to select it from the tab area, then click on the Page and use the Ctrl-F4 keystroke.

## Data View Dictionary

---



The Data View Dictionary node in the Rave Designer is where all the data connections for the reports would be listed. To add a Data View to this list, select a "New Data View" (from the "Project" menu). This will bring up a "Data Connections" dialog window. This window will show all the Rave data connections that are "active". Then select any of these and this will create a new Data View attached to that data connection.

## Ctrl- (Control) Drag and Drop

---

Ctrl-Drag and Drop is used to create or mirror an item from the Project Tree structure either to another page node in that report structure or to the current page on the designer layout. When dragging from the Project Tree Panel to the page designer, the drop location will be the location where the mouse button is released. However, when dragging from one page node to another (in the Project Tree Panel), the location of the drop will default to the same place as the source.

### NOTE:

When Ctrl-dragging a non-visual component, it must be one that can be mirrored. Copying a component that cannot be mirrored will NOT work. Regions and Bands are components that do not enable mirroring. To mirror a region, make the region part of a section, and then mirror that section. The section could include other components as well.

### CAUTION:

Drag and drop will work from Report Page to Report Page, Global Page to Global Page, Global to Report Page, however, it will NOT work from the Report Page to Global Page, since components on Reports are not visible to other Reports and therefore cannot be used as mirrors on a Global Page.

## Alt-Drag

---

The Alt-Drag is used to change the parent of a component to a new "container" component like a Section. All moves of that component need to be within that same Page design. When a new Section is created, the user may choose to have items from another Section placed in the new Section. In this case the parent must be changed from the old Section to the new Section and then the new Section can be mirrored.

### NOTE:

The target component must be a container type. The Page, Region and Section are all holders or containers. So, if there are two or more sections on a Page, items can be dragged from one Section to the other Section. Also components can be dragged from the Page to one of the Sections.

## Exercise: Navigating the Project Tree

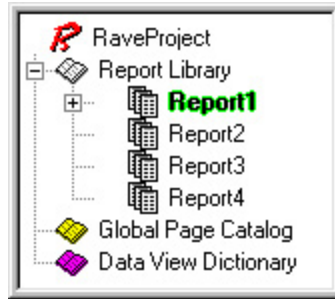
---

Because many report projects are complex, the Project Tree's hierarchy helps to navigate through the reports.

### To navigate through the hierarchy (expand, collapse, and selection):

1. Expand an item by clicking the square icon with a "Plus" symbol. This will change the

- item to a "Minus" symbol and expand the items into the sub-items such as Report1, Page1, etc., depending on which level of expansion is being expanded.
2. Collapse an item by clicking on the "Minus" Symbol. This will change the icon to a "Plus" symbol and collapse the items, so that they will no longer be visible.
  3. Select an item in the Project Tree by clicking on the item. Selection will be indicated by that item being highlighted.



## Exercise: Naming Components

---

Since projects can become very complex, it is highly suggested that the "Name" property be used to make items in the Project Tree easier to understand.

By default, the "Name" property will automatically default to something like "Report1", "Report2", and so on. However, it would be easier to maintain a project if the reports had useful names. This can be done through the "Name" property.

When developing a naming convention, be creative, but remember **no spaces or special characters** are allowed in the "Name" property.

In the example Project Tree Panel, the default report names have been changed. Instead of dealing with reports listed as "Report1" through "Report7", the reports were labeled accordingly: "CustomerList", "CustomerLabels", "CustomerDue", "Invoice", "PO", "ProductsOnHand", and "ProductsOnOrder". It is important to understand that the Project Tree Panel is there to make overseeing a project easier. But, without an informative naming system, the Project Tree Panel will not live up to its full potential.

### To rename a report (or any other component):

1. Select the report (component)
2. Ensure that the report to be renamed has been selected. A selected item is highlighted.
3. Go to the Property Panel and look at the Name Property.
4. In the "Name" Property edit box, type in the desired name.
5. Hit Enter or click somewhere out of the edit box, after a new name has been entered.
6. Look at the Project Tree and notice that the name of the report (component) has been changed from the default name.

## Exercise: Loading and Unloading Global Pages

---

Loading and Unloading Global Pages to the Page Designer Tab area makes referencing these pages fast and helpful. This is especially true when dealing with mirroring sections and objects from Global Pages to the pages in different Reports.

### To load a Global Page:

1. Start with a new report
2. Insert about 4 blank reports, by selecting **New Report** from the **Project** menu or by clicking on **New Report** from the Project toolbar
3. Next, select "Page1" in the first report. Do this by using the Project Tree. Click on "Page1" in the tree or click on the "Page 1" tab in the Page Designer Tab area. Once that Page has been selected, it's name will be highlighted.
4. Add about three Global Pages to the report. Do this by selecting **New Global Page** from the **Project** menu or by clicking on **New Global Page** from the Project toolbar
5. Notice that since the first page in the first report was selected when the Global Pages were added, the new Global Pages were added to the Page Designer Tab area. The pages are also added to the Project Tree under the Global Page Catalog
6. Click on a different Report page. Notice that once a different page is selected, the Page Designer Tab area only has that page in the Page Designer Tab area
7. To add Global Pages to the Page Designer Tab area right-click on a Global Page in the Project Tree and in the menu that pops **up select** Load Page. Try loading all three Global Pages
8. As the Global Pages are loaded they will be seen in the Page Designer Tab area
9. Try loading Global Pages to other pages in the last two reports

**To unload a Global Page:**

1. Complete the steps from the previous exercise
2. Select a page from Report 3 that contains all the loaded Global Pages
3. There are two ways to unload a page. One way is to use the speed menu; the other is to use quick keys
4. The first way, using the speed menu, requires that you first select the page that needs Global Pages unloaded
5. Next, right-click on the Global Page and **select Unload Page** from **the** speed menu. Do this for **all three** Global Pages in Report 3
6. We will use the second way to unload Global Pages next
7. Select the next page in "Report 4". Make sure it contains all the Global Pages in the Page Designer Tab area
8. Once the page in "Report 4" is selected and you can see the three Global Pages in the Page Designer Tab area, select one of the Global Pages from the Page Designer Tab area
9. Now use the **Ctrl-F4** keystroke to unload the Global Page
10. Try this with the rest of the Global Pages in the Global Page Catalog.

**Exercise: Dragging a Component (Ctrl-Drag)**

---

One of the more common items to drag would be to copy a Data Field over to the Page in the Visual Designer. The target could be an existing DataBand or just an open area on the Page. The Data Field will be converted to a DataText or DataMemo on the target area depending on the source's field type. Accomplish this by doing the following steps.

**To drag a component:**

1. Go to the "Data View Dictionary" node, expand it if it is collapsed
2. Select one of the Data Views and expand it if it is collapsed
3. Select (highlight) a field component to copy
4. Press and hold the Ctrl-key, now drag that field over to the Page

5. Release the mouse and Ctrl-key when it is at the desired location
6. Select the newly copied data field and set its properties as required

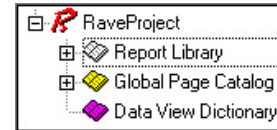
This exercise may have been a little advanced for this point in the manual, but it is put here to give the user the steps necessary to complete a dragging component. So, since a full report has not yet been created, don't worry about getting the report working yet.

## Exercise: Changing the Parent of a Component (Alt-Drag)

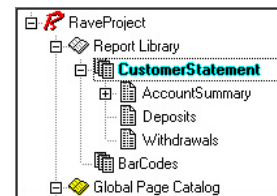
Changing the parent of a component can be very helpful, like when trying to group or ungroup components. For example, say there was a Page that contained components that would be better grouped together into a Section. The current status is that the Page is the parent component of the components that need to be grouped. To group the items together, the parent component would have to be changed to the section component. After this is done, the items will be in a section, which can then be moved around as such. In the reverse sense, maybe a component needs to be taken out of a section and needs to just lay on the Page and not the section, just use the Alt-Drag to complete this task.

### Changing the parent of a component:

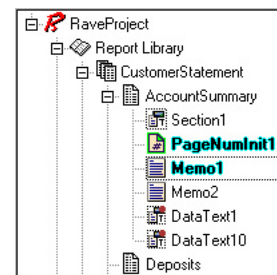
1. Go to the "Report Library" node; expand it if it is collapsed



2. Expand the Report node that contains the Page definition needed

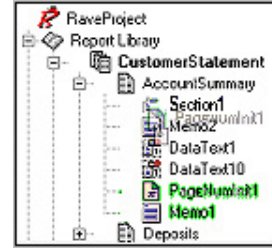


3. Expand the Report node so that both the source and target component can be seen. In Customer Statement, there is a Section component already on the Page. We will change the parent of PageNumInit and Memo1 from the Page to the Section

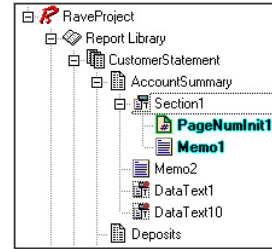


4. Select the components that will have their parent's changed. To select multiple components, hold the **Shift** key while clicking on each component. When done selecting release the **Shift** key. The desired components should be highlighted
5. Press and hold the Alt key, now drag the source components to the new parent component. In this example, the Alt key is held while selecting and dragging either of the two components. While dragging the components to the new parent, notice that cursor changes as it is placed over the target. When the cursor

indicates it is over the target, release the mouse button, then release the **Alt** key



6. Notice that the components are now under the new parent and that the parent has a "minus" sign next to it to indicate that it is the parent of some components



7. When done, the newly moved components can be set as required





---

# Property Panel

In this Section:
<ul style="list-style-type: none"><li>• Defines and describes the Property Panel</li><li>• Explains the different types of properties</li><li>• Describes the property editor</li><li>• Covers the purpose and items in the speed menu</li></ul>



---

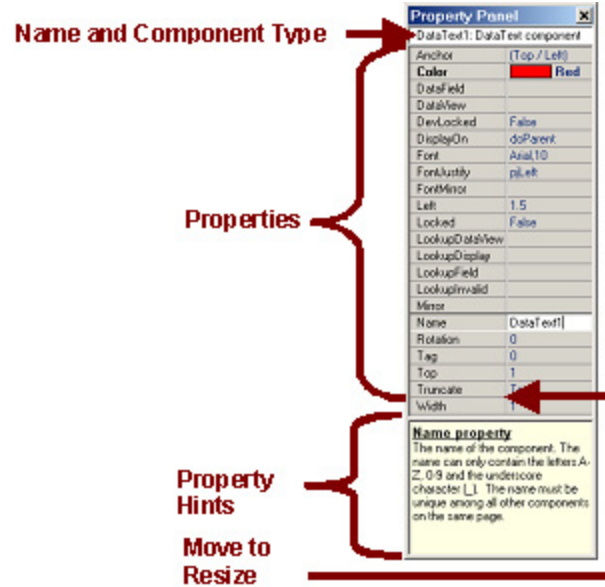
## What is the Property Panel

The Property Panel gives information about the project components. It will be most helpful when designing reports because it gives an easy access to all properties associated with each component in a report.

The Property Panel displays and allows modification of values of any component that is placed on the Page. To get property information about a certain component, remember to select that component before looking at the Property Panel (see *Selecting a Component in the Page Designer Chapter* for more information on how to select an object).

The Property Panel displays the current properties of a component. For example, by clicking on a DataText component, the component will be selected and the Property Panel will display all the properties that are available for the component.

There are three areas to the Property Panel. The first is the very top, which gives the name and type of the component. The second area is the middle, where all the component properties are listed into two columns. The left column displays the property names and the right column displays the current value of those properties. At the bottom of the Property Panel is the third area. A Property Hints box, which gives a brief explanation of the currently selected property, is located in this third area.



In the property listing section (middle), the line that separates the two columns can be moved to resize the columns to allow for better viewing of the names/values. To do this, move the cursor over the line until it changes into the double arrows. Press the left mouse and hold, then drag either left or right to resize the columns.

Remember a component must be selected before the Property Panel will reflect that component's properties. It is also possible to select multiple components at the same time. Once a component is selected, any of the properties in the Panel can be selected by clicking on the desired property name or it's adjacent box selection. With the property selected, it is then possible to change the property value. When any of the fields in the Property Panel are selected, the up/down arrow keys can be used to move up and down between properties.

Mirror	
<b>Name</b>	DataText1
Rotation	0
Tag	0
Top	1

Some of the property names appear in bold. This indicates that the current value differs from the default value. For example, when a DataText component is dropped onto the Page, the default color is Black. By changing the color to something other than Black, like Red, the property name "Color" changes to bold. When the setting is set back to the default value, the bolding will be removed.

Of what use is the bolding? Let's say for a moment that a report that has been designed by a colleague has a problem and it doesn't work. By quickly glancing at the changed properties, this will show what properties have been changed, which might in turn reveal the report functionality problem.

While most components can be selected by clicking directly on them, the Page is a special case. Clicking on the Page itself in the Visual Designer window will clear the Property Panel of all properties, thus not selecting the Page.

To select the properties of the Page, click on the tab at the top (in the area called the Page Designer Tab area) where it shows the page number. Another alternative is to select the Page by clicking on it from the Project Tree.

## Types of Properties

---

Every property has an associated type. These types can be numerous, different, and can be edited in the Property Panel. The property types can be string, or integer values.

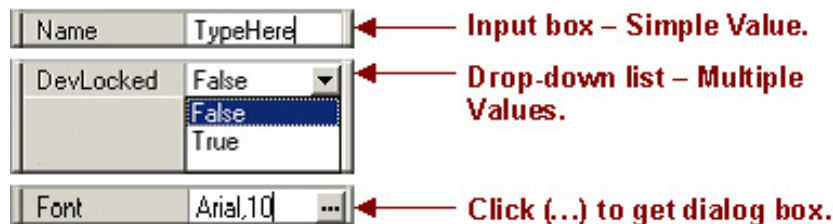
When editing a string property, it is valid to enter any character that can be displayed on the screen. When editing integer values, it is valid to type any numeric value. Decimal numbers are another common property that may be edited.

Often times there are properties that have predefined values that may be selected. These values are displayed in the form of lists containing the valid items that can be selected. There are also properties that have themselves a number of different properties associated with them.

## Property Editors

---

Depending on the type of property selected, there are three types of input boxes available. The first one is a simple input box where a value is typed. The value in the input box can be either numeric, alpha, alphanumeric or one of a set of possible values. A drop down list is the second type of input box a user can encounter. If the values of a property are from a set of previously defined values, a dropdown list will appear by clicking the small down arrow that appears on the right-hand side of the box. Select the desired value by clicking on it. The third option is a dialog box. Some properties require more than one value to set the property (such as fonts). With these properties, a button with three small dots (called an ellipse) appears on the right-hand corner of the property edit box. By clicking on the dots, a dialog box (or wizard) will appear and from this dialog box the multiple properties can be set.



## Right-Click Menu

---

A popup menu will appear when the Property Panel is right-clicked on. This speed menu can be used to set various options of the Property Panel.

Property Hints, the first menu item, hides or shows the Property Hints box which is located at the very bottom of the Property Panel. Highlight Changes, the second menu item, enables or disables the showing of the properties in bold for items that have been changed. Exclude Name, Size and Pos Changes, the third menu item, determines whether Name, Size and position changes (like width and height) are included in the properties that will be bolded when the default value has been changed. The third menu item becomes helpful when many properties have been changed, and there is a need to see changed properties that are not commonly changed.

## Exercise: Navigating the Property Panel

---

### Opening the Property Panel

1. If the Property Panel is not visible, then go to the Tools Menu. From the **Tools** sub-menu, choose **Property Panel**. Make sure that there is a check mark next to the selection.

### Identifying Property Panel Areas

2. For this example, the base component, the Page, will be used. So, to select the Page component, click on the page tab in the Page Designer Tab area. Notice that after selecting the Page, its properties are displayed in the Property Panel.
3. Next make sure that the Property Hints box is visible at the bottom of the Property Panel, if it isn't already. Right-click anywhere in the Property Panel. A pop-up menu will appear. In this menu, check all the options available.
4. Expand the Property Panel out so that you can see all the properties as well as the Property Hints box. On steps 5 to 8, read the Property Hints box to find more information about each property.

**Note:** The image to the right has been modified for the next few examples.

5. Click on the Name property. Notice that it corresponds to the name at the top of the Property Panel "Page1: Page component". The other half of this line is the kind of component that the line is referring to. The component is a Page component and thus it is recognized as that here. Type in any name in the Name property and see the change reflected in the top line. The Name property is an example of an edit box of type String.
6. Click on the Bin property down arrow. Notice that this displays a drop-down menu. Drop-down menus give many predetermined options from which to choose.
7. Click on the Grid Spacing input box. This is an example of a Numeric type property. To see what this does, change the Grid Spacing property value and look at the Page when done.
8. Click on the ellipse (...) on the Parameters property to see a dialog box appear. The dialog box is a way that a property with multiple values and options displays its choices.
9. In step 3, the option Highlight Changes was checked in the speed menu (right-click on the Property Panel). Go to the PaperSize property and choose a different paper size. Notice that not only does the paper size change, but that the property is bolded. This bolding becomes helpful in determining when properties have been changed from their default or initial values. Try changing other values such as the page width and height.
10. Also, checked in step 3, is the Exclude Name, Size, and Pos Changes option. This is a very helpful option. In most cases the name, size, and component position on a Page changes often. So, to avoid highlighting these commonly changed properties, this option excludes them from being bolded. Try unselecting this option and changing these properties.

---

# Generating Output

---

## In this Section:

- Learn how to execute and preview Reports
- Explains the options in the Printing Preferences
- Learn about the toolbars and options in the Report Preview
- Explains the differences in the Report Designer and the Print Preview Output Options dialog
- Learn how to export reports to various formats: NDR, PRN, HTML, and PDF

---

## Overview

After a report is created, displaying and making the report electronically mobile will become the next necessary step. There are several ways to get reports onto paper or into electronic formats. This chapter will cover the different ways to display report output and will also go through simple printing.

---

## Executing Reports

One of the most common features used in Rave is the actual report execution. Executing a report allows the user to see the output of the report design. Execution is very similar to what many developers know as compiling and actually does involve a certain amount of internal "compiling".

There are various ways that a report can be executed. It is done either via the Project menu, using the F9 function key, or clicking on the Execute Report icon on the Project toolbar.

Using the F9 function key becomes very handy since execution becomes a very repetitive and common task performed during all phases of design.

When using any of the ways to execute a report, the user is presented with various options. The Output Options dialog shows the different options available at the time of execution.

In later sections of this chapter, each option will be explained in further detail.

---

## Preferences Dialog

Before going through each execution option, there are some preferences that need to be covered and kept in mind before executing a report. Going to the Edit menu and selecting Preferences will display the Preferences dialog box.

The Printing tab in the Preferences dialog box allows various settings to be changed that effect report execution. Some of the options that can be set here are also reflected in the Output Options dialog box that appears when executing. In this section, each of the parameters in the Printing preferences will be explained.

In Output Options area of Printing Preferences, the Show Setup Dialog indicates whether the Output Options dialog (as shown here) is displayed when executing a report. It is sometimes convenient to disable the Output Options dialog, especially if it is during design and testing time. The drawback to this is that some of the parameters that are contained in the dialog box would have to be set before hiding this dialog. Also, any changes desired would have to be done either by re-activating the dialog box or specifying the changes in the Preferences dialog.

The default report print destination is determined by the settings under Print Destination in the Printing Preferences, and is reflected in the Output Options dialog box. Changing the value in the Print Destination area of the Printing Preferences will directly effect the default selection in the Output Options dialog box. There are two options that can be chosen: Preview and Printer. Selecting either one, by choosing the appropriate radio button, will make that option the default value and when the Output Options dialog appears the default value will be selected.

The Preview Grid Options act in much the same way as the Grid (which are the lines seen on the Page in the Visual Designer) does during design time. These Options allow a Grid to be displayed on the Print Preview screen underlying the Report. The Grid settings allow changes for the Print Preview Page, including Grid changes for color, style, and spacing adjustment. If the spacing is both 0 horizontally and vertically, no grid will be displayed in the preview. But, placing numbers in the horizontal and vertical spacing settings, will space the grid appropriately on the Print Preview Page. The Line Color will change the color of the Grid. The Line Style drop-down menu will give different options of line styles.

The Preview Options, as the name indicates, only affect previewing. There are six parameters that can be set. When using the Preview, the window displayed by default is in normal state. Like any other Window, the Preview can be Minimized or Maximized. To set the window display, use the drop down menu in the Initial Window State setting.

The Zoom Factor indicates the default zoom percentage when the preview is first displayed. The Zoom Increment specifies the Zoom percentage that will occur by each click on the Zoom In/Zoom Out icons. As with the Page Designer, the preview screen can have a horizontal ruler, vertical ruler, or both by specifying this in the Ruler Type (use the drop-down menu to make the selection). The measurements can be indicated in either centimeters or inches.

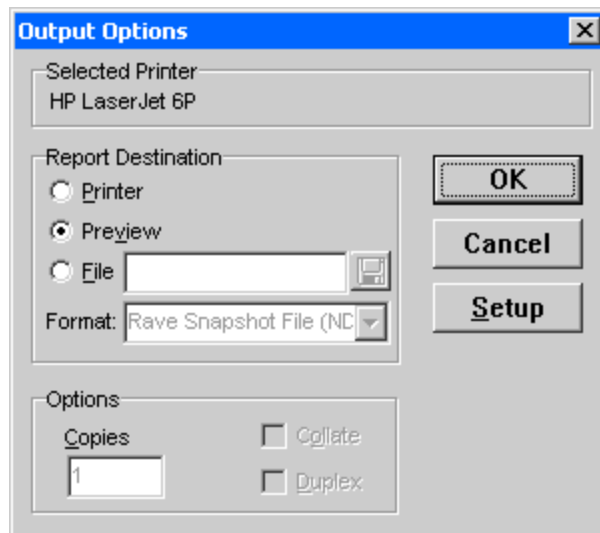
The ShadowDepth is the level of shadow that is displayed on the preview screen and it is only for on screen appearance and aesthetics.

The Monochrome Preview Display check box is there for backward-compatibility with some video cards that would give problems when displaying reports in color. This is not recommended for use, but is there if needed.

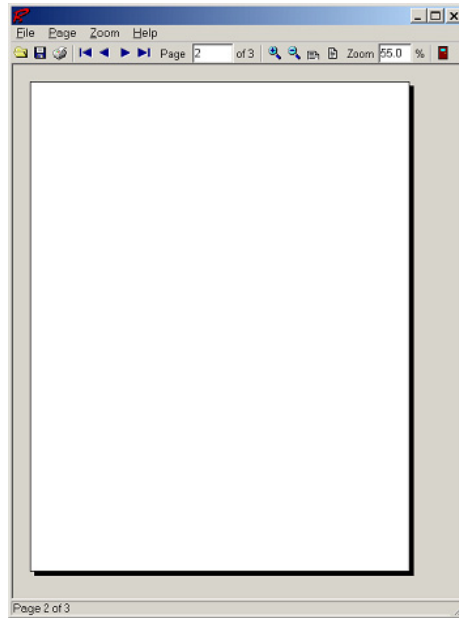
## Report Preview

Although the Preview is primarily used for previewing a report, the Preview option has many more capabilities than what is intuitively understood from its name.

Begin by executing a report. When the Output Options dialog appears, the Preview radio button can be selected. Clicking OK will continue with the Preview process.



The Preview screen is used to see the output of the designed report. The actual appearance on the screen resembles that of the printed report. The screen is based on a page-by-page output. There is a toolbar at the top of the Preview screen that allows navigation as well as other functions, which are explained in the following paragraphs.



Most of the functions permitted in the Preview screen can either be performed using the toolbar or through the Preview menu options. We will go through and explain the menu options in the rest of this section.

The first group of icons (separated by a vertical line on the toolbar), can be found in the File menu. They are used for opening, saving, printing, and exiting the Preview of a report.

Although the Preview screen displays the current report in the designer, it can also be used to load a previously saved report (with an NDR extension). This can be done by clicking on Open in the File menu. A dialog box appears prompting for the desired report file. When the NDR file is opened, the report that was previously in the Preview is discarded. This does not mean that the report can not be regenerated. To get the previous report, simply re-execute the report and choose Preview. Only the report that was Previewed upon execution can be regenerated. Other reports can only be revisited if they were previously saved or if their projects are re-opened and re-executed.

The Save As is used to save the report in the Preview. Click on Save As and the Save File dialog appears. Use this dialog to save the current report to various formats, including PDF and HTML (these two formats are covered later on this chapter). Another format that a report can be saved to is the NDR report type, again this is a Rave snapshot of the report.

Although reports can be printed directly by choosing Execute and then Printer, the Preview screen also provides this option, since many times it is necessary to check the preview of the report before actually printing it.

The remaining functions directly affect the viewing of the report on the Preview screen. The second section on the toolbar and the items in the Page menu are used for navigation between pages. The buttons used to designate the options will look familiar to database users, since they follow the same convention. The first button is used to move to the first page of the report. The second button moves to the previous page. The third button is used to move to the next page,



and the last button can be used to move to the end of the report. On the toolbar, there is a page indicator, which at all times reflects the current page and the number of total pages. This page indicator can also be used to go to a page directly by typing the page in the edit box. Or use the Go to Page in the Page menu. For example, entering 10 in the edit box will display the 10th page of the report (providing there are 10 or more pages available).

Zooming is also available in Preview. The magnifying glass containing a plus sign will zoom in on the page, while the magnifying glass with a minus sign will zoom out. Similar to the page edit box, there is a zoom box that is used both for displaying the actual zoom and allowing the user to enter a specific percentage value. Just type in a number and the Preview will be zoomed in or out to that zoom percentage.

There are also two preset zoom values for fast navigation. The first one is the Fit to Page Width option. By clicking on this, the report can be adjusted so that the width of the page takes up the entire region of the preview screen. The second preset is Fit to Page. It can be used to view the entire page on the screen. The side effect of this is that although the report can be viewed as a whole, the actual contents might be difficult to read. Normally this can be used to get a general overview of the report layout.

The preview window can be exited by either clicking on the Door icon located on the far right of the toolbar or by clicking on the "X" button on the window (like any other window).

## Executing to the Printer

---

Printing can be done, as mentioned before, via the Preview screen or directly when executing the Report and choosing Printer as the Report Destination. One thing to notice is that the Output Options dialogs are a little different when selected from the Rave Designer and from the Print Preview.

The Output Options dialog that appears from the Rave Designer has more options than just printing, as shown in the image.

In the Output Options Dialog there is a Setup option for Printing. The Setup button is underneath the Cancel button and can be used to adjust the printer settings prior to the actual printing. This includes options such as whether the report should be printed in landscape or portrait, the resolution of the printer, the color depth, paper size, etc.

The dialog box also includes several Options such as the number of copies to be printed, whether the copies should be collated, or duplex printed.

In the Output Options Dialog from the Report Preview (see image), there are some different options available. The Print Range has three different options. The All option means to print all pages in the report. The Selection option means to print a selection of pages not adjacent to each other, i.e. "1,3,5,7" or "1,9-5,20". The Pages option prints a range of consecutive pages, i.e. "1-10" or "15-19".

## NDR & PRN Files

---

Printing to a file can be done several ways. We will discuss two that can be done directly from the Output Options dialog. The Output Options dialog is obtained by executing the report (use the F9 key or execute through the Project menu).

The NDR, or the Rave Format, is a Rave snapshot of the report. An NDR file can be seen and opened from the Preview window, but it cannot be changed or edited in the Rave Designer.

The PRN file is the Native Printer Output type file. When this option is selected, the information that is usually sent to the printer to print a report is instead sent to a file. This file is saved based on the file name the user gives.

To create either type of file, simply select desired format and then fill the empty File edit box with a name for the file.

The NDR can also be made using the Save As option in the File menu in the Report Preview window. Using the Save as type drop-down menu and entering a File name will create the NDR file.

There are other ways to print to a file. These are only two formats. We will cover PDF and HTML file in the next sections.

## **HTML**

---

Rave has the capability to save reports as HTML files. This makes it easy to have reports readily available over the Internet or through a company intranet. HTML files can be created in the Visual Designer and in the Report Preview window.

To save a report as HTML from the Visual Designer, first get to the Output Options Dialog. Do this by using the Execute Report from the Project menu, or by using the F9 key.

In the Output Options dialog box, choose the File option from the Report Destination area. Using the Format drop down box to get to the HTML option will set the format to HTML. Clicking on the disk, will allow the user to set the path of where to save the file to and to name the HTML file.

Once a report is saved and properly formatted it may be helpful to first Preview the report. Preview the report by executing the report and choosing Preview from the Output Options dialog. After looking the report over in the Report Preview, choose the Save As option in the File menu. This will bring up the Save File dialog, where a report can be saved into HTML.

To save the HTML format, in the Save as type drop-down menu, select HTML files. Then type in the name of the new HTML file in the File name edit box. When done, click Save.

Each page of the report will be saved as an individual HTML file. For example, a three-page report would produce three pages of HTML. On each page, navigations links to the previous and next pages are provided. Once saved, the HTML files can be viewed with any Internet browser.

## **PDF**

---

A truly useful report is one that is portable over many different platforms. The PDF feature of Rave makes this possible with the help of a free viewer. A report can be saved into PDF format, and through the use of the Free Adobe Acrobat Reader, any person can open, view, and print the report.

This option is available through the Execute Report (also F9) option in the Visual Designer File menu or through Save As in the Report Preview File menu. Using the Report Preview allows the user to see the layout of the report before saving it into PDF format.

After selecting Execute the Output Options dialog will appear from which the format can be chosen. Also, the file name of the PDF can be set.

In the Print Preview, using Save As will bring up the Save File dialog and from there you can save the report as a PDF type, as well as choose the file name.

To save a PDF, in the Save as type drop-down menu, select Adobe PDF. Then, type in the name of the new PDF in the File name edit box. When done, click Save.

## **RTF**

---

Rave also has the capability to save reports as a RTF (Rich Text Format) file. This file format can be loaded into many word processors for a wide variety of reasons. For example, your report(s) could be included in a company document as an attachment. Or a report could be loaded into a word processor and then selected lines could be edited (highlighted, bold) to emphasis a point to a particular audience.

This option is available through the Execute Report (also F9) option in the Visual Designer File menu or through Save As in the Report Preview File menu. Using the Report Preview allows the

user to see the layout of the report before saving it into RTF format.

After selecting Execute the Output Options dialog will appear from which the output format can be chosen. Also, the file name of the output file can be set.

In the Print Preview, using "Save As" will bring up the "Save File" dialog and from there you can save the report as a RTF type, as well as choose the file name.

To save a RTF, in the "Save As" type drop-down menu, select RTF. Then, type in the name of the new RTF in the File name edit box. When done, click Save.

## Exercise: Changing Printing Preferences

---

### Change Print Destination

1. Before beginning this exercise, create a new Project and place a few visual and text components on the Page.
2. First, go to the Printing Preferences tab by either using the Edit Preferences icon on the Designer Toolbar, or by selecting Preferences in the Edit menu.
3. Click on **Printing** in the Preferences option area (left of the screen).
4. Look at the Print Destination area and select **Preview**.
5. Now close the Preferences window by selecting the **OK** button.
6. Execute the report by pressing the **F9** key, or by selecting **Execute** from the **File** menu.
7. When the Output Options dialog appears, notice that the Preview option is the default value selected. Click **Cancel** when done.
8. Now, open the Printing Preferences again, and select **Printer** as the Print Destination. Click **OK** when done.
9. Execute the report again, and in the Output Options dialog, notice that the Printer is now the default value selected.

1. Before beginning this exercise, create a new Project and place a few visual and text components on the Page.
2. First, go to the Printing Preferences by either using the Edit Preferences icon on the Designer Toolbar, or by selecting **Preferences** in the Edit menu.
3. Next, make sure that Preview is selected in Print Destination.
4. Uncheck the **Show Output Options Dialog** in the Output Options area.
5. Select **OK** to close the Preferences dialog.
6. Execute the report by pressing the **F9** key, or by selecting **Execute** from the **File** menu. Notice that the Output Options Dialog did not appear; instead the execution went directly to the Preview. If Printer had been selected in the Print Destination, the report would have been sent to the printer.
7. To get the Output Options dialog back, go to the Printing Preferences and re-check the **Show Output Options Dialog** in the Output Options area.

### Grid Lines and Options in Preview

1. Before beginning this exercise, create a new Project and place a few visual and text components on the Page.
2. First, go to Printing Preferences by either using the Edit Preferences icon on the Designer Toolbar, or by selecting **Preferences** in the **Edit** menu.
3. Next, make sure that **Preview** is selected in Print Destination and uncheck the **Show Output Options Dialog** in the Output Options area. We will be exploring the Preview for this exercise, so using the Print Destination and **Show Output Options Dialog** to

- execute quickly to Preview will be very helpful.
4. When in the Printing Preferences dialog, make sure the initial values of the Preview Grid Options and the Preview Options are exactly as seen in the image.
  5. Click **OK**, when done.
  6. Execute the report and just view the Preview of the window.
  7. Now close the Preview window, and Open up the Printing Preferences.
  8. In the Preview Grid Options, change the **Horizontal Spacing** and **Vertical Spacing** to 1. Then change the line color by clicking on the color wheel.
  9. Click **OK** when done.
  10. Execute the report and notice the lines on the report in the Preview. These lines will not appear on a print out, but are there if needed to help check layout and formatting.
  11. Now, close the Preview and go to the Printing Preferences again. Change the **Line Style** in the Preview Grid Options area to **Dash**. Also, change the **Initial Window State** to **Minimized**.
  12. Then, in the Preview Options, change the **Ruler Type** to **BothIn**. This will display both the horizontal and vertical ruler in inches in the Preview. Also, change the **Shadow Depth** to **10**.
  13. Click **OK** when done.
  14. Execute the report.
  15. First of all noticed that the Preview is minimized. This was set by the Initial Window State. To expand the window to see the Preview, click on the first or second button.
  16. Once the window is restored or maximized. There are several things to notice. First see the dashed gridlines on the Page. These can be useful to check object alignment on the Page. Also, notice the vertical and horizontal ruler. At the very far right edge of the Page, notice the deep shadow behind the Page.
  17. When done, go back into the Printing Preferences dialog. Put all settings back to their original values. The image shown displays our values. These values will be used for the remaining exercises in this chapter.

## Exercise: Preview and Creating Portable Files

---

1. Before beginning, create a New Project. And on four pages, place text and objects.
2. Link the four pages together. Selecting the Page and the Property Panel was covered in the previous chapter. Select the first page and in the Property Panel use the drop-down menu in **GotoPage** to select **Page2**. Do this for every page. Select page 2, and make **GotoPage** equal to **Page3**. Select page 3, and make **GotoPage** equal to **Page4**.
3. Next, go to the Preview window by executing the report. Select Preview from the Output Options dialog.
4. At the top of the Preview window on the toolbar, there should be an edit box that has the number 1. That area should read Page 1 of 4.

### Navigating through Preview

5. Since there is more than one page, use the navigation buttons on the toolbar or in the **Page** menu. Use the **Go to Page #** to jump to any page in the report.
6. Next, use the zoom tools to get a closer and an overall look of the report pages. This also will allow you to become more familiar with moving around in the Preview.

### Creating an NDR File through the Preview

7. To create an NDR (a Rave Snapshot File) from the Preview Window, go to the **Save As** option in the **File** menu.

8. When the Save File dialog appears, use the **Save as type** drop-down menu to select NDR files. And put in the name of the NDR file in the **File name** edit box, in this example we used Text.ndr. Click **Save** when done.
9. Even though the file is already in the Preview, let's try opening the Test.ndr file.
10. Choose **Open** in the **File** menu.
11. The Open File dialog will appear. Look for the Test.ndr file in the directory you placed it. Click **Open** when done.
12. The Preview will most likely look the same, as we did not change the report. But, look at the top of the Preview window to where the name of the file is listed. So, as you peruse through different NDR files, the name will always be listed at the top of the Preview. Nothing in an NDR file can be changed; it is simply a way to view the report.

### Creating a PDF file through the Preview

13. Since we are no longer using the executed file, close the Preview window. Execute the file and select **Preview** again.
14. Now, select **Open** from the **File** menu.
15. In the Save File dialog, select Adobe PDF from the **Save as type** drop-down menu. Then, type in a name in the **File name** edit box, this example uses test.pdf.
16. Click **Save** when done.
17. At this point you can go out to where your file was saved and open the PDF file. If you do not have the PDF reader, go to [www.adobe.com](http://www.adobe.com) and download the free reader.

### Creating HTML Files

18. In the Preview window, select **Open** from the **File** menu.
19. In the Save File dialog, choose **HTML File** in the drop-down menu for **Save as type**. Then type in a name in the **File name** edit box.
20. Click **Save** when done.
21. When done you can go to the HTML file that was saved. We saved our html files in My Documents. You will notice that there are four pages of the test.html, with consecutive numbers after the test title. Each page in the report will have an associated HTML page.
22. Below is the first page of our test.html. This is test1.html, as it is the first page. Note that at the bottom there are navigation links to the get to the next pages.



## **Exercise: Printing a Report through the Preview**

---

1. In the Preview window, select **Print** from the **File** menu.
2. This will bring up the Output Options dialog. Use any of the print ranges to get any of the pages of the report that you want.
3. When done, close the preview window.
4. Then execute the report, and select Printer. This too will send your report to the printer.

## **Exercise: Printing to a File (NDR & PRN)**

---

1. Execute a report and select **File**.
2. In the **Format** drop-down box, there will be two options. **Rave Format** (.NDR) and **Native Printer Output** (.PRN).
3. Make the desired choice.
4. In the edit box next to **File**, type in the name of the file.
5. Click **OK** when done



# Utility Toolbars

## In this Section:

- Alignment Toolbar introduced with Order, Spacing, and Alignment of components
- Designer Toolbar introduced and its relation with Preferences explained
- How to read the Color Palette Toolbar and its designations
- Font Editor briefly introduced and explained
- Use of the Line Editor Toolbar is explained
- Functions of the Project Toolbar are introduced
- Zoom Toolbar functions covered

## Tools are Tools

Even though the toolbars are called toolbars, there are two distinct types, relating to the function performed. There are Utility Toolbars and Component Toolbars.

A tool is typically used to modify an item already on the Page, usually a component. A component is an object placed on the Page, like a Region, Shape, Line, or Barcode (for more information regarding components, see the Components Overview chapter).

This chapter will cover the Utility Toolbars, which contains tools (Color, Alignment, etc.) used to modify components. Utility Toolbars include the Alignment Toolbar, the Color Palette Toolbar, the Fill Toolbar, the Font Editor Toolbar, the Line Editor Toolbar, the Project Toolbar, and the Zoom Toolbar.

## Using Tools

The Utility Toolbars act on components and the Page Designer. Either a component or the Page has to be selected prior to using the toolbar. For example, to change the fill color of a Square, select the Square and then click on the required color.

By right clicking on a component, a popup menu will appear giving access to other options that can change the component. These options are also available from the toolbars; the popup menu just gives faster, easier access.

## Alignment Toolbar



The Alignment Toolbar is used to align, order, and move a group of components on the Page. It is important to note that some of the Alignment options only make sense when more than one component is selected.

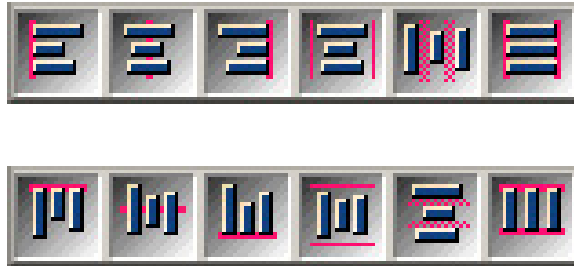
The first step is to select the components that need to be aligned. To select multiple items, hold the shift key down and click the mouse on each object that needs to be aligned.



For most alignment options, the **first** component selected is what determines the alignment position. For example, when choosing Align Left, all selected components will be left aligned to the position of the first selected component. A red line on the icon designates the actual alignment.



The Alignment Toolbar is divided into four sections by separators. The first two sections control horizontal and vertical alignment options. Both allow components to align on the edges, centers, or to be spaced equally.



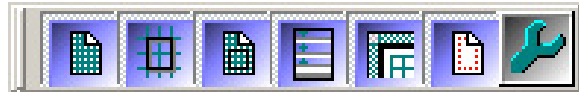
The Order Alignment determines the Z order (depth along the Z axis) between components. Order Alignment does not require more than one component to be selected. To establish the Z order of a component, select the component and right-click the mouse button to display the popup menu. From the Order submenu choose one of the four options: Move Forward, Move Behind, Bring to Front or Send to Back. The figure to the right shows a Rectangle with an Ellipse imposed in front of it. To move the Ellipse to the back of the Rectangle, right-click the Ellipse (by right-clicking on a component previously unselected, it will select it and bring up the popup menu) and choose Send to Back from the Order submenu.

There are an additional four buttons on the Alignment Toolbar, Tap Tools, which allows small incremented movements of a component (or group of components). To move selected component(s), click on the corresponding arrow to move the component(s) towards the desired direction.



## Designer Toolbar

The Designer Toolbar is used to control the Preferences for the Page. All of the Designer Toolbar items toggle. By clicking on a button, the property of the component associated with the button will change status. To change it back to the previous state, click on the button once more. The top toolbar shows what the buttons look like when they are not pressed (toggle status off) and the bottom toolbar is the resulting toolbar when the buttons have been depressed (toggle status on).





"Activate Grid" controls whether the grid lines are visible or not on the Page Layout



"Snap To Grid" controls whether the objects on the Page 'snap'. Snap means that the component will only move and resize according to where the Grid Lines are present



"Grid On Top" determines whether the Grid Lines are always on top of the components or not (i.e. are not hidden when components are placed on the Page Designer)



"Always Show Band Headers" controls the displaying of band headers



"Show Rulers" toggles the appearance of guide rulers in the environment



"Show Waste Area" displays the space between the Page boundary and the end of the Page Setup Boundary



"Edit Preferences" will bring up the Preferences dialog

## Color Palette

---



The Color Palette is the tool used to change a color of a selected object such as a Line, Text component, or the fill area in one of the shapes (Circle, Ellipse, Rectangle, or Square). Color palettes are composed of several sections that show custom, standard and current colors. The left button on the right side displays the currently selected Primary, while the right button shows current Secondary color. All other color squares serve as the Color Palette from which you can select.

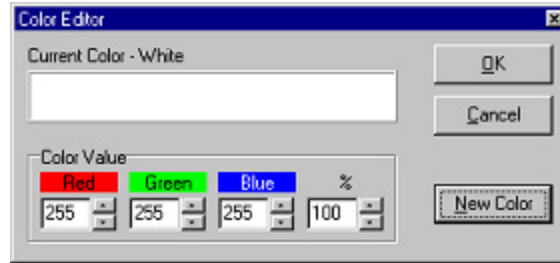
To change the color of an object, select the object and either left or right-click on the Color Palette. A left-click selects the Foreground Color (Primary) while the right-click selects the Background Color (Secondary).

When a Primary Color is selected, a "1" will be placed on the corresponding color indicator on the palette. Similarly, a "2" will be placed indicating the Secondary Color.

To change the color of an object (like a Rectangle), first select it and then click on the desired color with the left mouse button. To change the Fill Color, click on the desired color with the right mouse button.

Double-clicking on the user defined Custom Colors (right side) or on the Foreground and Background Color boxes will open a more detailed color control window called the Color Editor. From the Color Editor, custom colors can be created by adjusting the red, green and blue values,

as well as the saturation of the color by increasing or decreasing the color percentage.



To create and save a new color, use one of the Custom Colors to enter into the Color Editor, then click on the New Color button and choose the desired color from the Color dialog box. By entering the Color Editor from one of the Custom Color boxes, the new color will be saved. If the Color Editor is entered from either the Primary or Secondary Color boxes, the color will only be saved in those boxes until a new color is chosen (either by clicking on the right or left mouse button or by the creation of another color).

It is important to note that the actual exhibited colors are dependent on the display settings of the computer. If there are issues with colored objects when transferring reports from one system to another, check the display color settings. A 256-color setting only uses 256 colors for display, where as a True Color setting uses millions of colors. Thus, for example, if a Report is created on a system set at True Color, and then viewed on another system set at 256 Colors, some objects may appear grainy or to have the wrong color shadings.

## Font Editor

---



The Font Editor can be used to change the type, size, attributes and alignment of any text component in the report. Its aspect and functioning is very similar to Font Toolbars found in the most commonly used Word Processors.

To change the Font settings of a Text component, first select the component. The font type can be adjusted by choosing a font from the drop-down list. Similarly the size can be changed by clicking on one of the available sizes for the font in question. The **Bold**, *Italic* and Underline buttons are toggled depending on the state of the text. To set the font to **Bold**, just click on the Text component and then press the **Bold** button. The button will then remain depressed until it is clicked on again or until focus is moved to a different text object that does not have the **Bold** set. The *Italics* and Underline button work in much the same way. There are three alignment buttons that indicate the alignment of the text within the area it is contained, and the options are left, centered, and right-justified. All text-based components have a property called Font that appears in the Property Panel when the component is selected. Any changes made to the Font, are reflected both in the font toolbar and in the Font property.

**NOTE:** It is important to note that if a font is chosen on the development machine and this font is not available on the deployment or any other machine, Rave will assign the next most similar font to the one chosen.

## Line Editor

---

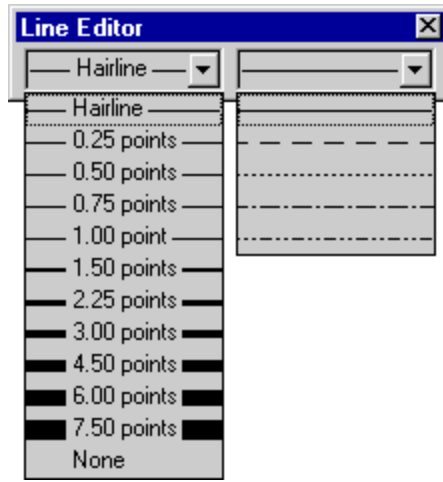
Use the Line Editor to modify the style and thickness of a line or the border of a shape. The Line

Editor Toolbar has several selections to choose from including line styles (solid or combinations of dots and dashes).

To change the style of a line, first select the line or shape, and then choose the point size of the line as Hairline. Finally, choose the style of the line.

To change the thickness of a solid line, first select the line or shape. Then choose the point size desired.

Line style only (dashes, dots, etc) applies if the line thickness is hairline. If a line is set to a certain style (like dashes) and then given a thickness other than hairline, the line style will be changed to a solid line.






## Project Toolbar

---






The Project Toolbar provides the basic functionalities for a Project. It gives a quick easy way to complete the common tasks associated with managing a Project. All options available on the Project Toolbar are accessible through the main Project menu in Rave.



The first three icons deal with managing projects:

- |   |                |                                  |
|---|----------------|----------------------------------|
|  | "New Project"  | creates a new Project            |
|  | "Open Project" | opens a previously saved Project |
|  | "Save Project" | saves the current Project        |

The next three icons deal with creating specific aspects of the current project:

- |   |                   |  |
|---|-------------------|--|
|  | "New Report"      | will create a new Report that will be contained in the same Project file (.rav file). Rave can have more than one Report per Project file            |
|  | "New Global Page" | creates a new Global Page  |
|  | "New Data View"   | allows the creation of DataViews that are associated to the current Project. Please see Connecting to Data chapter more information about DataViews. |

The last two icons deal with the currently selected Report.


- |   |                  |                                       |
|---|------------------|---------------------------------------|
|  | "New Page"       | creates a new Page in current Report  |
|  | "Execute Report" | execute (or print) the current Report |


## Zoom Toolbar


---


The Zoom Toolbar allows the user to Zoom In to view the details of a document better or Zoom Out to view the overall Page uniformity. Within the Zoom Toolbar, there are several ways to zoom into and out from the details of the document.



 To Zoom by Percentage, the Zoom Factor drop menu has many preset percentage factors to Zoom in and Zoom out. Simply choose the desired Zoom Percentage. If one of the presets is not adequate the you can specify a percentage by typing the desired percentage into the edit box next to the zoom factor drop menu.

 Another method to get a customized size is to click on the Zoom Tool (magnifying glass). Drag the tool over the specified area that needs to be magnified.

 For a simple manual Zoom In and Zoom Out, click on the magnifying glasses with the "+" and "-" sign. Nothing will need to be selected to use this tool. Each click is set to a specified incremental amount that is set in the Preference settings. To get to the settings area, go to Preferences in the Edit Menu. In the Edit Menu, select the Designer Tab. In the area called "Zoom Increment", increase or decrease the percentage by using the up/down arrows to increase the percentage or simply type in the percentage.

 To zoom in on selected items, use the magnifying glass with the dashed box. First select the desired item(s), and then click on this tool. The viewable area will frame all item(s) selected.



Use the "Page Width" button to zoom to the full width of the currently selected Page.

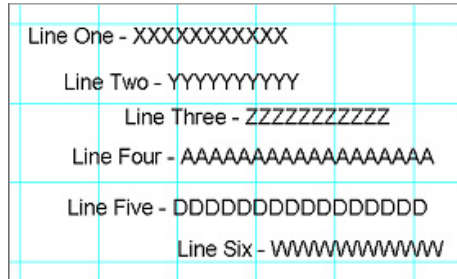
Use the "Whole Page" button to Zoom to the full page size of the current page.

## Exercise: Aligning Components

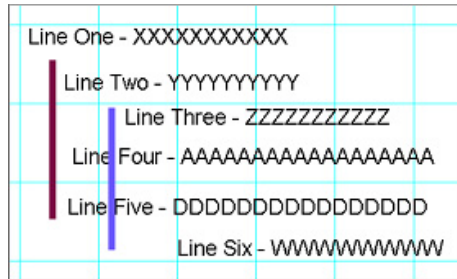
---

This exercise will demonstrate how to align components. The example images have six numbered lines of text.

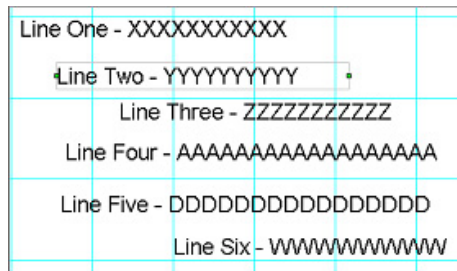
1. To begin, drop several Text components onto a Page. In this example, the components have been number labeled



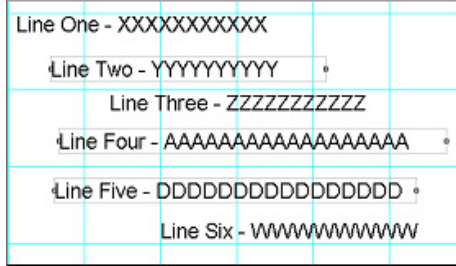
2. Lines two, four, and five will be aligned. Lines three and six will also be aligned



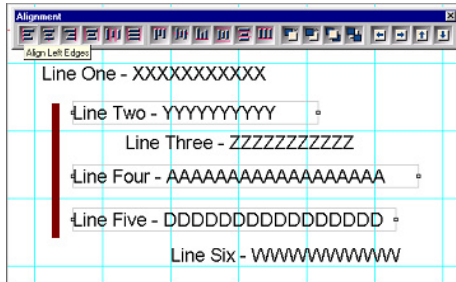
3. Select Line Two. This will be the primary component that the rest of the components alignments will be based on



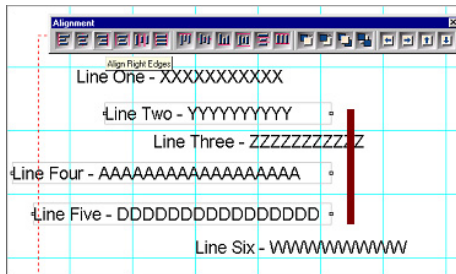
4. While holding the shift key, select lines Four and Five to be aligned with Line Two. Notice the light gray border that appears with pips



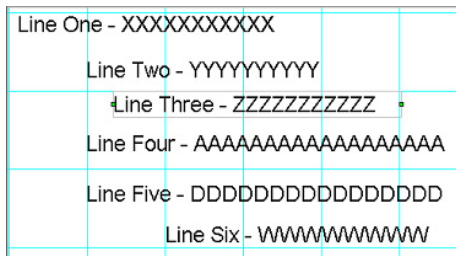
- Once all components have been selected, choose Align Left Edges on the Alignment Toolbar. Notice that all three components left edges have been aligned



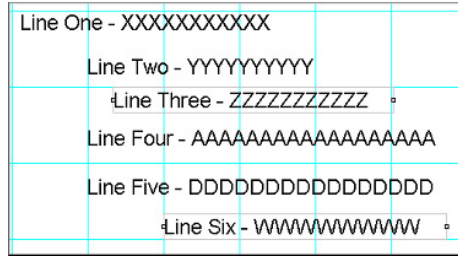
- To further show how the alignment works, click on the Align Right Edges button on the Alignment Toolbar



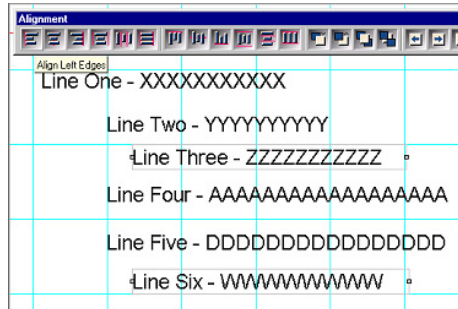
- Go through the rest of the Alignment tools to see how they function
- Let's align the second set of text. First, select the primary component to base the alignment on, in this case Line Three



- Then select the other component(s) that will be aligned, by holding the shift key and clicking on Line Six



10. Then click on the Align Left Edges to align the components.



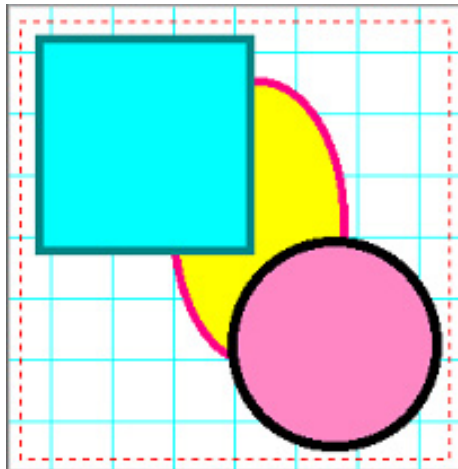
## Exercise: Ordering Components

---

Ordering refers to the ordering of components along the Z-axis. This is the axis that comes out of the Page.

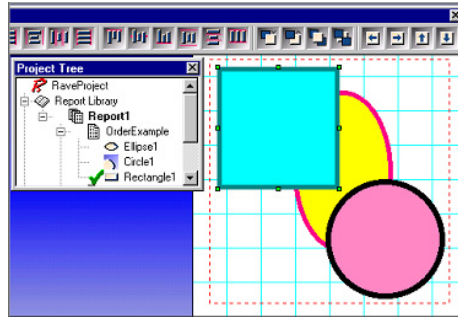
To order components along the Z-Axis, the order buttons on the Alignment Toolbar must be used.

1. To begin, there must first be components on page. These can be any components. For this example we will use a Rectangle, Oval, and Circle

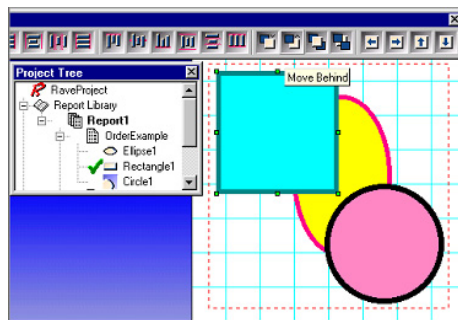


2. First select a component to move. In this example, the rectangle will be moved behind the Oval. Notice in the Project Tree the placement of the Rectangle component

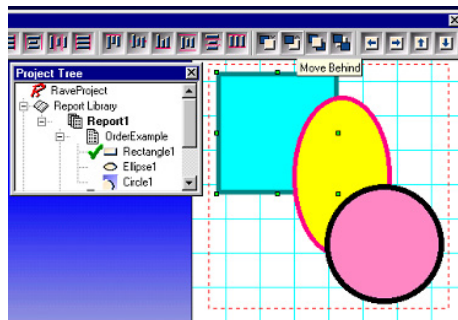




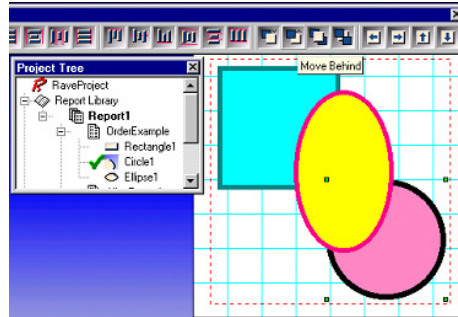
3. Once the rectangle is selected click on the Move Behind button on the Alignment Toolbar. Notice that after clicking the button once the rectangle has not moved behind anything yet, but on the Project Tree the rectangle has moved up the tree



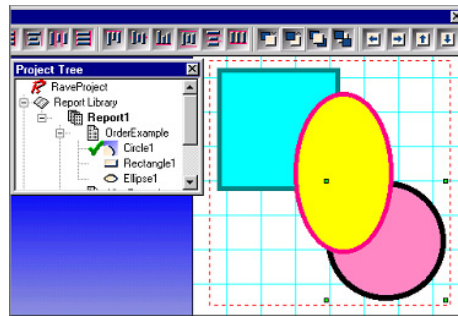
4. Try clicking on the Move Behind button one more time. The rectangle will move up the Project Tree and the Rectangle on the page will move behind the Ellipse



5. For the second component, select the circle and click the Move Behind button. The Circle component will move up the tree and move behind the Ellipse



6. Click on the Move Behind button once more and the Circle will move up the tree one more time and remain behind the Ellipse



## Exercise: Snapping to the grid

Snapping to Grid refers to the moving and resizing of components according to where the grid lines on the page are present.

1. To begin, bring up the Preferences Dialog box by selecting Preferences from the Edit menu
2. Select the Defaults tab and look for the area labeled Grid Spacing
3. In Grid Spacing, change to .02 units. Then look for the area called Draw Grid Every
4. In Draw Grid Every, enter 1 line (type or use the up and down arrow). Draw Grid Every 1 (that's what we chose) line, will show every Grid line made. If this were changed to Draw Grid Every 3 lines, then for every third line the Grid would be marked in the Visual Designer
5. Next go to the Project menu and choose New. This will create a new Project, as well as a new Report Page, that is based on the preferences that were just changed in the previous steps. Remember the Grid lines serve only as page guides for designing reports; they do not show up in any report executions
6. Go to the Designer Toolbar and click on the Snap To Grid
7. Now that the Report Page is created, drop a rectangle on the page
8. Now move the rectangle around and notice that the rectangle only moves according to where the lines are located. Try using the tap buttons. Notice that the component doesn't seem to move until the tap button moves are on one of the grid lines
9. Now to go the Designer Toolbar and click on the Snap To Grid button to turn the feature off
10. Move the rectangle and notice that the rectangle will move to positions between the grid lines



## Exercise: Changing Line Size and Color of a Rectangle

---

1. Create a new Report Page
2. Drop a rectangle component on the Page
3. Look at the Color Toolbar. Notice the color blocks marked 1 and 2, you can review Primary and Secondary Designations in this chapter
4. To make the color change more notable in this example we will increase the boundary lines of the Rectangle. Bring up the Lines Toolbar. Select the Rectangle. Then in the first Lines Toolbar drop-down box, choose a larger border size. In this example, the borderline has been increased from hairline to 2.25 points.
5. Left-click on any color in the Color Toolbar. In this example the color Olive was selected. Notice that the border has changed to Olive. Also notice that the Primary Color has changed. The Primary Color is the color of the border.
6. **Right-click** on any color. In this example the color Blue was selected. Notice that the fill inside the Rectangle has changed to Blue. Also notice that the Secondary Color has changed. The Secondary Color is the color of the inside of the Rectangle

## Exercise: Changing Fonts

---

1.  Create a new Report Page.
2.  Drop four Text components on the Page.
3. Select a component, and then view the Font Toolbar.



4. On drop-down menu displaying the Font Name, click on the down arrow and choose a desired font.
5. Go to the next component and repeat Steps 3 and 4. Do this for all remaining Text components.
6. Also, try using the font-sizing drop-down menu. As well as the **Bold**, *Italic*, and Underline options
7. The last three buttons to the far right deal with placing the font as left-justified, centered, and right-justified. Practice using these options to get the desired formatting effect.

# Standard Components

## In this Section:

- Explains the difference between a Text component and a Memo component
- How to use a Section and its relationship to the Project Tree
- How to insert BMP and META files

## Overview



The Standard Toolbar contains seven components: Text, Memo, Section, Bitmap, Metafile, FontMaster, and PageNumInit. Several of the Standard Components are used most frequently when designing reports. This chapter will provide a detailed explanation of the first five. The remaining two (FontMaster and PageNumInit) are covered in the Advanced Components chapter.

Access to the Standard Toolbar is achieved in the same way as with other toolbars. If it is not present on the screen, it can be displayed by selecting it from the Toolbar submenu in the Tools menu.

## Text



The Text component is useful for displaying a single line of text on the report. It acts basically like a label that can contain simple text (not data). When placed on the report, the Text box is surrounded with a box that indicates its boundaries. This can only be seen while the text component is selected (represented by the pips).

The component can be used for labels, such as for figure or graphical titles, floating text, form titles, and in general just about anything that is denoted by a single line of text.

When the component is selected, the length of the text can be adjusted by resizing it using the pips. Note however that the height of the text is self-adjusting and does not require further intervention by the user.

Like any other text-based component, there is a Font property that can be used to change the font type, size and style. The color can be set using the Color property. The actual text of the component, which is seen on the screen, is denoted by the Text property.

There is an additional property called Rotation, which can be used to rotate the text by a specified number of degrees. The effect can only be seen at runtime.

## Memo



Memo components are similar to Text components. However the most noticeable difference is that Memos can contain multiple lines of text. Memos can be used on forms for areas of explanation, and for titles or comments that are longer than one line.

Similar to the Text component, the border around the Memo can only be seen when the component is selected. When setting the Font of the Memo, all the lines of text in the Memo will have the same font. *It is not possible to set different fonts for certain parts of the text.*

To change the text in the Memo, go to the Property Panel and use the Text property. The Text property will show "(Memo)" and an ellipse button, which are three dots. Click on the ellipse button. This will open up the Memo Editor. Text can be entered into the Memo Editor.

One of the main problems with using multi-line text components is that the text might overlap in height on to other components on the Report. To prevent this, set the ExpandParent property to **True**. This will expand the Parent of the Memo component to properly accommodate the Memo component.

Once the text is entered, the Memo box can be resized. The text within the Memo box will be repositioned accordingly. If there appears to be text missing from the Memo box, after it has been entered into the Memo Editor, resize the box to allow all text to be visible.

## Section

---



The Section component is used to group components. This provides advantages, such as allowing all the components that form part of the Section to be moved together with one mouse-click, as opposed to moving each component individually or trying to select all components before moving.

To use the Section Component, first press the Section Component button and then use the cursor/arrow to make a selection on the Page. When the selection is completed, there will be a box marked by dashes with green pips. Within this area, place any components that need to be grouped together.

The Project Tree becomes very helpful when dealing with the Section component. From an expanded Tee, it is easy to see what components are in each Section. Each Section and component can also be easily selected simply by clicking on the appropriate object in the Tree.

If an object is ever placed beyond the viewable section area, the Project Tree can be used to select the object. Once the object is selected, the object's border pips can be seen and thus its location. The object can then be deleted or moved (hint: to move use the tap buttons in the Alignment toolbar).

As seen from the relationship with the Project Tree and the Section component, there is a need to understand Parent-Child relationships between components. To read more on these types of relationships see the Project Tree Panel chapter, it contains a section on Parent-Child Relationships. The Section component also becomes very important when mirroring. Mirroring will be covered in the Managing Report Projects chapter. For now, just know that once the Section component is mastered, using this with mirroring will become extremely helpful.

## Bitmap and Meta File

---



The Bitmap and Metafile components enable images to be placed in a report. Bitmap supports image files with the extension ".bmp", and MetaFile supports image files with the extensions ".wmf" and ".emf".



To insert an image, first click on either the BMP or the META buttons on the Standard components toolbar, then click any location on the Page. A placeholder box with an X will appear on the Page, and the box may be bigger than what the image will be. This can be fixed after the image is loaded.

Once the BMP or META placeholder is on the Page, go to the Property Panel and to the FileLink property. Click on the ellipse, the button with three dots.

This will bring up the Open Dialog. Find the image file, select the file, and click Open.

Once the image is loaded, resize the placeholder pips on the image border.


## Exercise: Text vs Memo

---


The decision to use either a Text or a Memo component is quite easy. If there is only a single line

of text to be placed on a Page, use the Text component. Use the Memo component when there is more than one line of text to be used.

**To place and change a text component:**

1.  Click on the Text component located on the Standard Toolbar. The component button will change colors to designate activation.
2. Click once on the Page. The Text component will appear on the Page. Also notice that a new component will be added to the Project Tree under the Page that it was placed.
3. Go to the Property Panel and in the Name property change the name of the Text component.
4. In the Text property, change the default text to the desired text to be displayed on the Page.
5. View the Property Panel and the Page. Notice that the Project Tree has the Text component changed to the name entered in the Name property. Also, on the Page the component has the desired text typed into the Text property.


**To place and change a memo component:**

1.  Click on the Memo Component located on the Standard Toolbar. The component button will change colors to designate activation.
2. Click once on the Page. The Memo component will appear on the Page. Also notice a new Memo component will be added to the Project Tree under the Page that it was placed.
3. Go to the Property Panel and in the Name property change the name of the Memo component. This will change the name of the Memo component in the Project Tree under the Page that the component was placed on.
4. In the Property Panel, go to the Text property. Click on the ellipse, the button with three dots (...).
5. The Memo Editor will appear. In the Memo Editor, type in the text that needs to be placed on the Page. The Memo will recognize returns made with the Enter key. Remember though, that the Memo will be subject to the size of the Memo box on the Page.
6. When the text has been entered, click OK.
7. When done, the Memo component can be moved or resized by using the green pips.

## Exercise: Section

---



**To create and use a Section Component:**

1.  Click on the Section component. The component button will change colors to designate activation.
2. Click on the Page, and a dotted box will appear with pips at its boundaries. This is the Section component.
3. In the Project Tree, notice that the Section component was added under the Page it was

- placed.
4. While the Section component is selected, place any other component into the Section. When done, notice that the placed components will now be under the section in the Project Tree.
  5. Once components are placed into a Section, all the components can be moved together. To try this out, place the cursor on the border of the Section, click and hold, then drag the Section around the Page.
  6. The Section can also be resized. Click and hold any of the pips and then move the cursor in the desired direction, the Section box will resize.
  7. When done, click anywhere on the Page to deselect the Section component. The Section is an invisible border displayed by a dashed rectangle, but it will not be printed.

## Exercise: Placing and Resizing Bitmaps

---

1. To complete this exercise, find a Bitmap (BMP) image on your computer. The image will have a ".bmp" extension at the end of the file name. Remember the location of this image.
2.  Create a New Report Page. Do this by selecting New Report Page from the Project menu.
3.  On the Standard Toolbar, click on the BMP button.
4. Then click once on the Page to drop the Bitmap component. The green pips will appear around the edges to indicate that it is selected.
5. While the Bitmap component is still selected, go to the Property Panel. Look for a property called FileLink Property. There will be three dots, called an ellipse button, to the right of it. Click on the ellipse button.
6. Using the Look in drop-down menu and selecting the image in the box below. Once The Open Bitmap dialog will appear. Find your Bitmap image on your computer by the file is selected, click Open.
7. The image will appear on the Page with green pips around the image.
8. Now look at the Property Panel and look at the property MatchSide. This property deals with the proportions of the image when it is resized using the pips.
9. Click on each of the MatchSide choices and move the image around a bit to see the effects of the choices.
10. This is msBoth. The pips on all sides allow the resizing of all pictures in all directions.
11. This is msHeight. This helps in resizing the pips according to height, while still keeping the image proportional. Using the top and bottom pips will resize the image.
12. This is msInside. This allows proportional resizing, but all resizing is done with in the limits of the pip. The image doesn't leave the borders of the pips.
13. This is msWidth. This does proportional resizing according to the left and right pips of the image.



# Drawing Components

## In this Section:

- Explains how to place a Drawing Component.
- Describes all the Drawing Components.
- Discusses Pixels vs. Points.

## Drawing Component Basics

All the Rave graphical components are created using the Drawing Components.



To use any of the Drawing Components, click on the desired component in the Drawing Toolbar then click once on the Page or other container component. This will create the object on the Page or other container component. Once the component object is created it can be resized to the desired dimensions by dragging the pips around the selected object border. Also, to color and stylize the components, use the Fills (where applicable), Lines, and Colors Toolbars. For more information on these toolbars, see the Utility Toolbars chapter.

The Property Panels of each drawing components are almost identical. The Name property may be the only thing that distinguishes one Property Panel from another.



The Line component can be used to draw a line in any direction. Angled lines may be used to draw lines between rotated lines of text. They may also be used as diagonal lines inside a square giving the effect of a box with an "X" in it.

Line styles may be set for the line; however, you should be aware that when setting line styles it is best to stick with a line width of one. Line styles are often ignored when the line width is any greater than this.



The HLine component may be used to draw horizontal lines. One of the common uses for the horizontal line is to drop one into a DataBand and set it so that it is positioned at the very bottom of the Band. It is also set to stretch the whole length of the Band. When the Band is printed, the line creates the effect of having the Band information printed on lined paper.



The VLine component may be used to draw vertical lines. One of the more common uses for the vertical line is to drop one into a DataBand and set it so that it is positioned in between various data fields that are being printed. The height of the line is set to the same height of the Band. When the Band is printed, the vertical line gives the effect of having the information printed in nice neat columns.



The Rectangle component may be used to draw rectangles. One of the more common uses for the Rectangle is to drop one into a DataBand and set it so that it is sized to completely fill the band. It is then moved to the back so that other components dropped on the Band will print over the top of the rectangle. When the Report is printed, this gives the effect of having a nice block around the rows of printed data.





The Square component may be used to draw squares. A common usage of this component is showing checked or unchecked boxes. The checked boxes are created with the use of the Square component and the Line component, or perhaps by simply by placing an "X" in the Square. This is useful when trying to illustrate items that have been selected.



The Ellipse component can be used to draw an ellipse or oval shape, more simply defined as a flattened circle. If you want to change the shape of the Ellipse, you can do this by clicking on the Ellipse with the left mouse button. Then move the mouse over one of the pips that will appear. Press the left mouse button again, and while continuing to hold the mouse button down, move the mouse until the Ellipse is the desired size and shape. You can also resize the Ellipse by setting the properties directly using the Property Panel. Simply enter new values for the Width or Height properties.



The Circle component can be used to draw a circle with a constant radius. If you want to change the size of the Circle, you can do this by clicking on the Circle with the left mouse button. Then move the mouse over one of the pips that will appear. Press the left mouse button again, and while continuing to hold the mouse button down, move the mouse until the Circle is the desired size. You can also resize the Circle by setting the properties directly using the property panel. Simply enter new values for the Width or Height properties.

## Pixels vs Points

---

For every Drawing Component, the Property Panel has two properties that deal with Line Width. The LineWidth property is a numeric value for the width of the line, which depends on the LineWidthType. The LineWidthType property actually lets the user determine which type of value the line will be set. The Types available are Points and Pixels.

When creating drawing objects, it is important to determine in what format the audience will be seeing the Report. This is important when determining if Points or Pixels will be used for the line width type.

Points are a unit of print, not a unit of screen space. Points are the unit of measurement used most commonly with paper design. It has meaning in the paper medium. Unfortunately, points are meaningless on the screen and the web. Due to platform and resolution differences, something like 14pt can mean many things. What it does not mean is a specific unit of screen size. Pixels, on the other hand, are defined unit. A pixel is always equal to a pixel.

Points and Pixels Preferences are dependent upon what the designer wishes the user to experience. To let the user adjust the text size at will or let the user's internet browser preference determine the size, specify points. If the design depends on a rigid text size or if the text size must be consistent across platforms, specify Pixels.

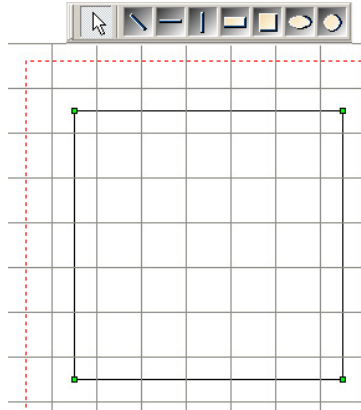
To print the thinnest possible line, use a Line Width of 1 pixel. For thicker lines, values in points should be used for consistency across printers with different resolutions.

## Exercise: Creating Drawing Components

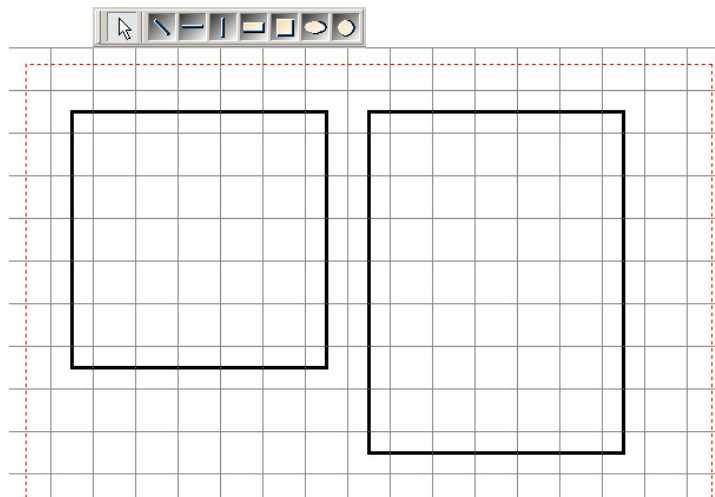
---

1. First create a new Page. Use New Report Page from the File menu, or use the icon on the Project Toolbar.
2. Now, go to the Drawing Toolbar. Pull it down from the Tab area, so that we can concentrate on it here.
3. Click on the Square, notice that it's button changes to a depressed button and changes color. Click your cursor on the Page and hold down the left mouse

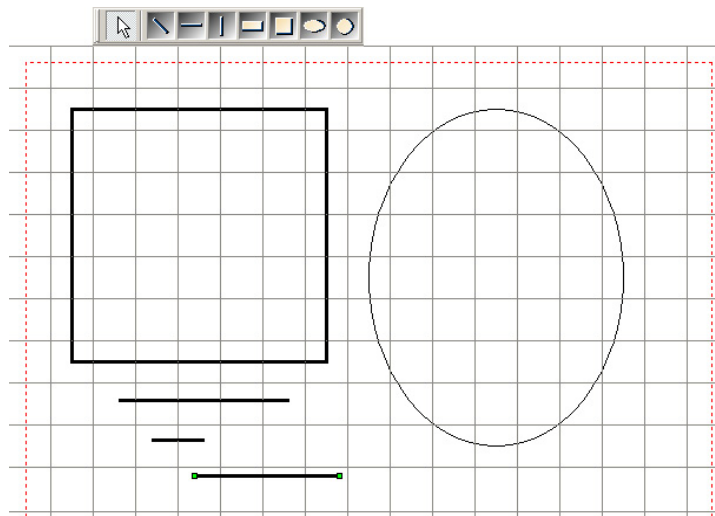
button while dragging out the shape of the square.



4. Next, click on the Ellipse. Click on the Page and hold down the left mouse button while dragging out the shape of the Ellipse. The Ellipse outline will look like a Rectangle, but once the mouse button is let go the ellipse shape will appear.

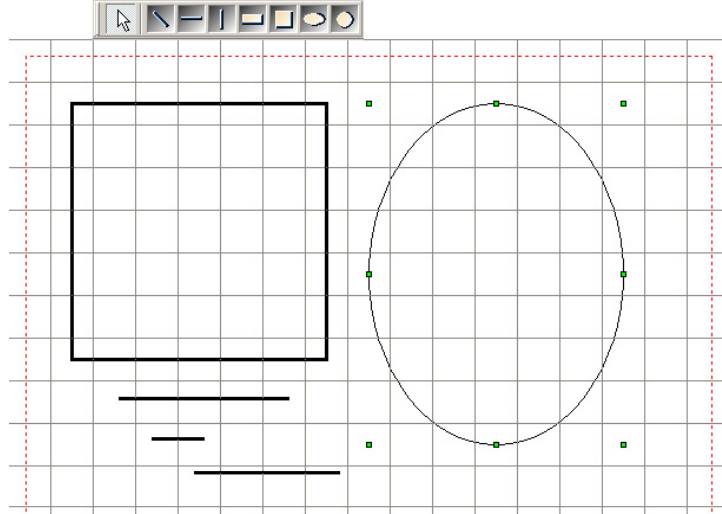


5. Then, click on the Horizontal Line and draw them on the Page. Do this for several lines. Make them different lengths.



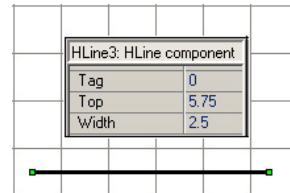
6. One thing to notice for every shape, when the mouse button is released

pips outline the shapes.  
When all shapes are  
selected the pips appear.

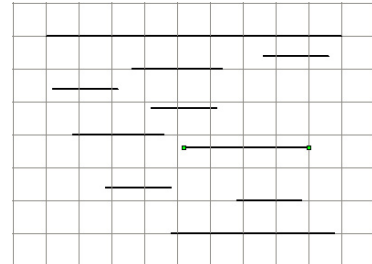


## Exercise: Alignment

1. First create a new Page. Use New Report Page from the File menu, or use the icon on the Project Toolbar.
2. From the Drawing Toolbar, click on the horizontal line button then click on the Page. The line will appear as the default line size on the Page. The clicking on the button and clicking on the Page is what we call dropping an object on the Page.
3. Select the horizontal line by clicking on it.

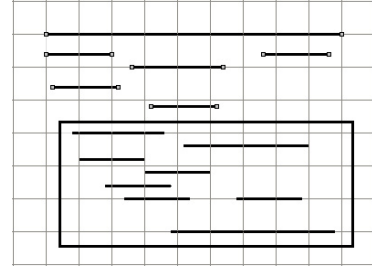


4. Go to the Property Panel and change the default value of the line by typing 2.5 into the Width property edit box.
5. Drop down at least 10 additional horizontal lines about an inch below the first one.

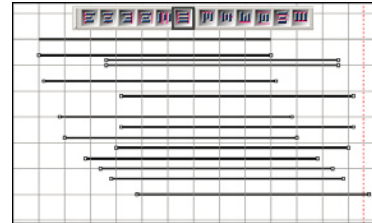


6. Drop down one more horizontal line about 3 inches below the first group of lines.
7. Select the top horizontal line by clicking on it. The green pips will appear at the ends of the line when selected.
8. While the first line is still selected, hold down the Shift

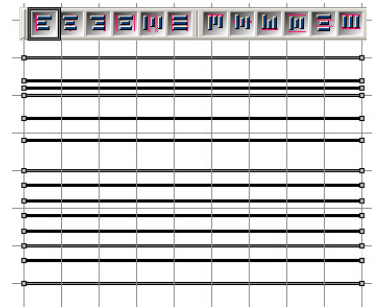
key and start clicking on the other lines. This will add all the other horizontal lines to the selection. To select the remaining lines faster, while the first line is selected and any other additional lines are also selected, keep holding the Shift key and click on the Page and drag the resulting box over the rest of the lines.



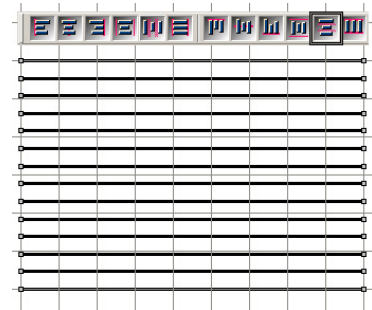
- 9. In the Alignment toolbar, click on Equate Widths alignment button. This will make all the lines the same width as the first selected line.



- 10. On the Alignment Toolbar, click on Align Left Edges button. This will align all the lines to the left edge of the first selected line.



- 11. Next, on the Alignment toolbar, click on Space Equally Vertically. This will space the lines equally away from each other on the vertical axis. For more exercises on Alignment, see the Utility Toolbars. Chapter.





## Chapter 11

# Database 101

### In this Section:

- Gives an overview of databases.
- Explains the needs to make up a database.
- Describes the data table structure and how it relates to reporting.
- Explains database relationships.

## Overview

Information to the left, information to the right, information above and below, we are swimming in a sea of information. The advent of the computer age has not helped. In fact, we now have the ability to store absolutely huge amounts of data. This can easily result in information overload for both individuals and managers. Something is needed to manage this overload potential. The first thing is that the information must be organized and stored in an efficient manner that aids in the quest to extract answer(s) from that wealth of data. One of the methods to retrieve answers is to use a reporting tool (like RAVE) to organize and display the answers in a manner that is easily understood by the reader of the report. But before a reporting tool can extract the information, it must be aware of the structure of the source of information. That container of the information in simple terms is a data table.

This section is intended to assist those that are not familiar with database design goals or might need a quick refresher. This section is not designed to cover this subject in depth, as this is best left to the many thick books already published on this subject and to more formal educational courses.

## What is a database?



Let's begin with some real world examples to describe the concepts of a data table. For example, a library contains lots of information stored in rows and rows of shelves containing books, pamphlets, etc., which is usually arranged according to the Dewey decimal system. To find a book, a card catalog is used to look up the subject or author, noting the Dewey decimal code. Then, the desired book is found and retrieved by going through the stack that contains the book code (if it isn't checked out).

Look up options are far more flexible if the library has a computerized card catalog. Now picture a library with no Dewey decimal system, no order but still the same amount of books. Try and find something in that library. It would be very tough and take a long period of time.

Now, change the words. Change book to information and library to database and we are on the way to explaining what and why of database management system. A library is a collection of knowledge (books) arranged by the Dewey decimal system.

A data table is a collection of information arranged according to some predefined table structure. We will now examine what is needed to design or use a data table.



## Terms

---

To better understand what it takes to create a database, it is necessary to understand the database lingo.

### Database Management System

A Database Management System (DBMS) is the complete system for managing data (information). There are many commercial examples of companies that are known for their DBMS programs. Some of the more well known include Informix, Interbase, Microsoft Access, Microsoft SQL, Oracle, Sybase, and many others. A DBMS can be confused with the physical data file(s). It must be noted that the data table is only one part of a DBMS. However, the real world does require that when extracting information from a DBMS that the structural concepts of a data table be understood. Fortunately, most of the terms are common between the various DBMS. This does not mean a person needs to be an expert in the database field, just knowledgeable.

### Table

A data "Table" is a collection of things like phone numbers, names and addresses. The "report" of this example table would be a telephone book. Of course, a telephone book has some restrictions, like do not include unlisted phone numbers in any print outputs. In some cases, the phone company will have a billing address that is different from the address location for the phone itself.



Now look at a phone book and notice that it is arranged in rows and columns. A row (sometimes called a record) shows the items related to a single member (person) of that table. Each column (sometimes called a field) is giving a specific piece of information about that member, like their name or phone number. While we are here, the order of the phone book (table) is done with an index (sometimes called a key). So, a phone book could be arranged by last name or by phone number.

### Structure

Each specific piece of information about a member of a table has attributes that define it for its DBMS.

Typical attributes for each table (file) include:

- date the table was created
- date the table was last modified
- number of columns defined for table

Typical attributes for each row (record) include:

- Length
- Record number
- Status (locked - deleted)

Typical attributes for each column (field) include:

- Name - a column name that usually "describes" its contents

- Type - character, numeric, date, time, blob etc.
- Length - maximum size of this item (might include number of decimals for numeric types)

A data table is comprised of information arranged according to a pre-defined structure. The structure of each table is important, because the row and column definitions are used by a reporting tool to extract, arrange and show information.

In the phone table example, there are the following fields: name, address, and phone number information. How are those columns (fields) defined? They are defined by the name of the column; meaning name might be called "name", address called "address", and phone number called "phone". But defining the main fields may not end at these simple definitions. For example the "name" field might be broken into three or more sub-fields, like "first", "middle" and "last". These three pieces as well as the first three main fields could then be manipulated as needed by any reporting tools. The three name sub-fields could be combined for a "full name" output or the first letter of the "first" name could be combined with "last" to make a short version of the name. Also, "phone" could be called "number" or "telephone" and the phone number could probably contain only the numbers with no formatting characters. Rave makes the task of understanding the table structure relatively easy as it does most of the work. Once the data connection is made, Rave will extract the table structure and from there the columns (fields) can be chosen as desired.

### **Query**

Rarely is there a need to print all the information in a database. Frequently, there is a need to reduce data by selecting a range of rows (records) in a data table. Data reduction is an important part of information management. One method of accomplishing a reduction is to do a query of the table to only show those rows that meet the range limits that need to be retrieved.

One of the more common query methods is called SQL, which stands for Structured Query Language. Some data tables are SQL compliant. A SQL query allows inquires of a DBMS for information, and the DBMS searches its tables according to the given SQL statement and return a "set" of rows (records) that meet that query.

### **Client-Server**

A method of setting up computer programs used by many people, where the database resides on a central computer accessible to all (the back end), and the user interface resides on the user's computer (the front end). All selection of specific data rows and processing is done back on the host computer minimizing the transmission of data through the network cabling. Significant improvements are achieved to the overall system performance by only transmitting relevant data through the cables. Thus, a well-configured Client-Server system can yield excellent performance gains in applications used by several people.

## **Relational Table**

---

Now the phone book example is good for simple data tables (flat files). However, it is not an efficient design for many real-world needs. We can stay with the phone company, just not the phone book for a more typical need. Every month the phone company needs to send a bill out to each customer. This bill is a good example of a "master-detail" type report. One part of the bill comprises the master (static) information about the customer like: their name, billing address and phone number. Another part of the bill contains the detail (variable) information, like an output line for each phone call made including its length and cost. The variable information could be only one page or for some customers many pages.

There is one important detail in any master-detail work, understanding the concept of a primary key (link key). When a customer gets a phone bill, that customer expects only to receive a listing of their calls; they don't expect calls made by other people to be listed. In order for this customer bill to be created, this means that there needs to be only one master row, called a primary key, for

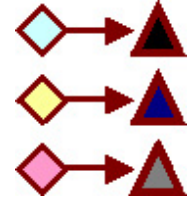


all the detail rows. In the customer bill, the primary (link) key would be the phone number. The primary key must be unique, which in general usually makes it more difficult to change a primary key once it has been assigned. But, there are some DBMS that allow the primary key to be a combination of columns (fields).

That is the quick review of Database operations. To ensure efficient design, with table structure control, there are a few more subjects to be aware of like: "Client-Server", "Normalization", "Relational DBMS", "One to One" "One to Many" and "Many to Many" relations.

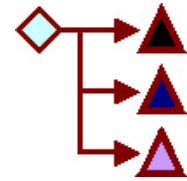
**One-to-One**

One-to-one is two separate tables with the same primary key. Therefore, knowing the primary key allows look up of data for a first row in one table, and for a second row in another table. One to one relationships are rare because it's usually easier to put both tables' columns into a single table. One-to-one relationships probably don't meet the first level of "Normalization" rules.



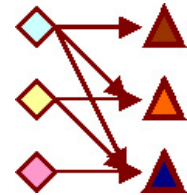
**One-to-Many**

This is accomplished by placing the primary key value of the ONE side of the relationship into an ordinary column of the MANY side of the relationship. For instance, if each employee can belong to only one department, then you could have a "department" column in the employee table, which, for each employee, would be filled with the primary key value for the row of his department in the department table.



**Many-to-Many**

This is difficult to explain, as the real-world examples are complex. If there were several employees that were part of several departments, then this would be a many to many situation



**View**

Some DBMS have an ability for a user to define a form to show contents of information taken from one or more other tables. A view is a definition of a form and has no data of its own. It often includes a query that is executed whenever the view is the subject of a command. Use views to control access to individual columns in a table, or to make two or more tables appear as one.

**Reporting**

---

How does all this relate to designing reports? With the definitions defined, understanding the report design structure and options will be easier. Now you are ready to read the sections on DataView Connection, DataField which will tell how to use Rave to connect to your data table(s) and how to identify the specific column (field) contents to the Rave designer.

# Connecting to Data

## In this Section:

- Explains what a database connection is.
- Show the different methods of retrieving data from a database.
- Explains the SQL Editor and how it is used.

## Database Connections

A reporting tool would be quite pointless if it did not allow methods of displaying information contained in databases. One could only go so far with designing non-data reports. Connecting to data in Rave is one of the more powerful features, not only because of the wide variety of databases and methods supported, but also due to the simplicity of the steps required to make a report data-enabled.

A database connection in Rave is the primary pillar for connecting to a database with SQL DataViews.

## Creating a Database Connection



To create a new Database Connection, click on the DataView icon on the Report toolbar, bringing up the Data Connection Wizard. The first step is to select Database Connection from the list of options available. When done, click Next.

After clicking on the Next button, the second step of the wizard displays the different type of connection options available. The selection depends on the different Rave DIBL Links that are installed on the system. DIBL stands for Database Independent Layer. It is a proprietary system that allows interactions with databases independently of what these database system are (whether it be SQL Server, Interbase, Oracle, etc).

In a folder under the Rave root directory, there should be a folder called DIBLLinks. Inside the folder there are a certain amount of files with extension rvd. The files are loaded when the application first starts. Depending on the number of files, the number of available connection options varies.

It is possible to have many types of connections displayed in the Database Connection Type Dialog screen. It is left up to the user to decide which connection type is the most appropriate. Fewer connection types may be displayed depending on which drivers were installed.

Once one has been selected, the corresponding dialog box will appear in the next step asking for the connection details. This varies from one to another, but generally consists of a path to the database, a server name if the database is not local and optionally a username and password to access the server.



After creating a database connection, it will appear in the Project Tree from where it can be accessed. The properties for a Database Connection can be shown just like any other component by clicking on the connection component in the Project Tree panel.

There are several properties that are specific to the Database component. The most important ones are the AuthDesign and AuthRun properties. Many times, the platform (and server) that one develops the report on does not coincide with that of deployment. Specifically, one characteristic that is most likely to change is the access codes to the server (username and password). Rave has been designed keeping this in mind, and once again making the deployment task easier. For

this, the AuthDesign and AuthRun properties can be used to provide design-time and runtime (deployment) information regarding the database and server.

AuthDesign contains the parameters that were specified when the Database component was made using the wizard and it can be changed by clicking on the ellipse button in the property field. AuthRun is the information for deployment.

LinkType represents the type that was selected when creating the Database connection. Again, similar to other components, there are common properties such as Name, FullName and Description, etc, which are not be explained again to prevent redundancy.

## Direct DataViews (BE only)

---

DirectDataViews provide a link to data connection components located within an application. When a DirectDataView is created, a list of all available data connection components is displayed. The *Name* property value of the data connection components provides the link between it and the DirectDataView (through the *ConnectionName* property in Rave). If connecting to data connections that have event code attached to them, you must have the application actually running and the form containing the data connection components created.

## Driver DataViews

---

DriverDataViews perform the same function as a DirectDataViews except they are self contained and typically obtain their data set from a SQL statement. DriverDataViews also require the use of a valid database connection.



Once the database connection has been established, click on New Data Object to create a new DriverDataView.

The Wizard will take the user through the necessary steps to complete the configuration. After selecting DriverDataView from the dialog box, choose the database connection that it is going to be connected to. If the connection to the database is successful, the wizard will then display the Query Editor dialog box. Before continuing with an explanation of the DriverDataView, let's examine the Query Editor.

### Query Property Editor

The Query Property Editor is used for generating a SQL statement that returns a result set from the database.

There are two ways to use the editor. The first is to take advantage of the graphical interface for constructing the SQL sentence. On the left side of the dialog box, a list of tables available in the current database is displayed. By simply dragging and dropping from the list to the gray area, the corresponding SQL statement will be generated.

Once the table object appears, by clicking on the tick boxes, the list of fields that should appear in the result set can be defined. By default "\*" is marked, which means that all fields will be returned. To select individual fields use the tick fields to make the appropriate selections.

Tables can also be linked (joined). To do this, simply drag another table over and trace a line from one field in the first table to another field in the second time (the figure above shows a join between PRODUCTID of table BUGS and PRODUCTID of table PRODUCTCODES. To view the results from here, click on the Results tab and the data will be displayed in a grid.

Alternatively, the editor can be used to write an SQL sentence directly. To do this, click on the Editor button that appears at the bottom. To enter a user-defined SQL statement, click on the "User Defined SQL" checkbox and type in the desired SQL.

Once the query has been constructed, clicking on the Ok button will return control to the Rave designer.

The SQL Data View component should now appear in the Project Tree. By expanding the + sign

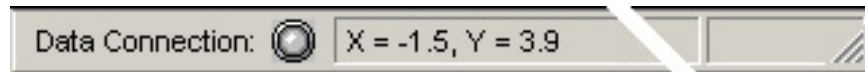
next to it, a list of all the fields that have been selected in the SQL statement will drop down. Properties for each field can then be accessed from the Property Panel prior to selecting the field in the Project Tree.

Although fields can be placed in a report by using the corresponding component from the Report tab, it is much easier to work directly with the Project Tree to accomplish the same task. Pressing down the CTRL button, fields can be dragged and dropped directly from the Project Tree on to the form designer.

Once the DirectDataView has been setup using the Query Editor, none of the other properties are required for accessing the data.





## Status Bar

---



At the bottom of the RAVE screen is the status bar. The status bar will provide some information about the item currently selected. So, watch the bar when designing a report.

The data connection LED will provide status of the Rave data system by the color of the LED.

LED	Color	LED Status
Gray		Connection(s) are inactive
Yellow		Connection active but waiting for response
Green		Connection active and busy
Red		Connection active but has exceeded time-out delay

The X and Y figures are the coordinates of the mouse pointer. Move the mouse around and watch the X, Y amounts change. When a shape is dropped on the page, the size of a shape will appear while it is being created as seen by the dX and dY position. The "d" stands for delta.



## Chapter 13

# Wizards

### In this Section:

- List guides to help in simple report structure.
- Guides will provide detailed steps to complete a reporting task.

## Wizards

Wizards are a new feature to Rave that allow certain types of reports to be created by answering a series of questions. They can be found under the "Tool" menu option. The Wizards can be added and tailored to each user's needs. It is a perfect way to minimize the end-users interface with reporting tasks.

There are two Wizards, a "Simple Table" and a "Master-Detail". Wizards can be designed to ask for the data connection and allow the user to choose the fields that are needed on a report. It is important to note that an active DataView should be available prior to running the wizards, whether it is a Direct Data View or an SQL Data View. Simple Reports are generally used for listings. Common uses include Client reports, Telephone lists, etc. The Master-Detail Wizard is used when more complex reporting is required, such as invoices, product order lists, etc.

Independent of the one that is executed, there are several steps that are common to both. These include the DataView selection, field selection, etc and are covered in greater detail in the exercises included at the end of this chapter.

It is very important that the user understands that these are general purpose reporting wizards and as such, some aspects are not treated with great detail (amount of text that would fit on a page, layout, etc). They can be used as the building blocks for a more complex report or can be adjusted to suite a particular layout required. If the intention is to build very complex reports, the wizards are not recommended for this purpose.

### Exercises

Below are example exercises of both Simple and Master-Detail wizards. Since most of the steps are similar for both wizards, they will only be explained in detail in the first exercise. It is highly recommended to start with the Simple Wizard first to get a good grasp on the concepts presented.

### Exercise: Simple Wizard

1. The first step is to select the desired DataView that will provide the data to the report.
2. Once the DataView has been selected, a list of fields that should appear on the report can be chosen. Fields are selected by clicking on the box on the left. All fields can be selected by clicking on the All button at the top. Note however that because this is a simple wizard, choosing more fields that would fit on the page will make them overlap. Manual intervention is required after the Wizard is complete to correct this problem.
3. If more than one field was selected in the previous step, the Wizard will ask for the ordering of the fields. Moving a field up will place it on the left side of the page. Move it to the lowest position will move the field to the right margin.
4. After the fields are placed in your desired order, the Report layout can be set. Values include the title and the page margins.
5. The last step of the Simple Report Wizard is to choose the fonts that are going to be used in the report. It gives the possibility of changing three fonts: title (of the report),

caption (headers of the field names) and the body (actual field values).

CustomerDVRegion: CustomerDVTitleBand (BGRDr gb 1PC)					
<b>Simple Table Report</b>					
CustomerDVRegion: CustomerDVBand (BGRDr gb 1PC)					
Contact	Company	Addr1	City	State	Zip
CustomerDVRegion: CustomerDVDataBand (Master 1PC)					
[Contact]	[Company]	[Addr1]	[City]	[State]	[Zip]

### Exercise: Master Detail Wizard

The Master-Detail Report Wizard has a few additional steps more than the Simple Wizard Report. This is due to the fact that more than one table is going to be taking part in the report. In particular, there will be a master (for example customer information) and a detail (items ordered).

1. Similar to the Simple Report Wizard, the first step is to choose the DataView. However, in this case the DataView chosen corresponds to the Master table.
2. In step two, the Detail table can be selected. Note how the DataView selected in step 1 is no longer available to avoid errors.

Similar to the Simple Wizard, the next steps are for selecting the fields and ordering of both the master and the detail table.

A new step is determining the key fields. These are field that relate one table to the other.

After this, the remaining steps are identical to that of the Simple Report Wizard.

Once everything is complete, clicking on Generate will produce a simple master-detail report that, again, can later be adjusted and "touched up" to ones particular needs. The figure below shows a sample output of the Wizard.

SQLDataView1Region: SQLDataView1TitleBand (BGRDr gb 1PC)	
<b>Master/Detail Report</b>	
SQLDataView1Region: SQLDataView1Band (BGRDr gb 1PC)	
BUG_ID	DUPLICATE_ID
SQLDataView1Region: SQLDataView1DataBand (Master 1PC)	
[BUG_ID ]	[DUPLICATE_ID ]
SQLDataView1Region: SQLDataView2Band (BGRDr gb 1PC)	
STATUS	
SQLDataView1Region: SQLDataView2DataBand (BGRDr gb 1PC)	
[STATUS ]	

# Report Components

In this Section:

- Gives more detail on using report components.
- Examples are given to aid in comprehension.

## Overview

The Report Toolbar is one of the most used when working with data-aware reports. It is very vital to understand the function of each component, especially since the basics of all database reports are based on the Report Toolbar.



The following components are from the Report Toolbar that are frequently used in designing your reports:

**Band, CalcText, DataBand, DataMemo, DataText, Region**

The following components from the Report Toolbar will be covered in the Advanced Components section of the manual.

*CalcController, CalcOp, CalcTotal, DataCycle, DataMirrorSection*

The following Report components are non-visual components

*CalcController, CalcOp, CalcTotal, DataCycle*

There are some general characteristics of the components that need to be discussed before describing each component individually. First, let's talk about the visual color of the components. For the most part the components look similar in color to all other components, they are gray. But, at the end of the toolbar there are four green colored components. The green indicates that these are non-visual components. When placed on the Page, they cannot be seen on the screen, in preview, or in print. Instead, to 'see' the components the user must use the Project Tree to select, move, and delete the component from the Page.

Most of the Report Components can simply be placed on the Page. The only two exceptions are the Band and the DataBand. These must be placed in a Region component. The behavior of the component is also controlled by the settings made with the "Band Style Editor", which is found through the BandStyle property for both components.

A Report can be designed into a typical report, like one that uses strictly Bands. But, the unique feature of Rave is that the report can also be designed to other specifications. The Bands of the report can be moved and resized to any desired shape.

When designing a Report, the designer window can be changed to the preferences of the Report Designer.

There are several ways to control the "look" of the Bands while in the design mode. The first is whether ALL Band Headers need to be seen or not. This setting can be changed with the Designer Toolbar. Another method is to toggle the visibility of the Band contents. If some of the Band designs are stable and they need to be temporarily hidden, then try setting the



DesignerHide property to True. Only the Band(s) with DesignerHide set to False will show. This might be needed if there are a large number of Bands or a Band that occupies a large space. The setting of the DesignerHide property has NO effect on what will be printed, only what is shown on the designer Page.

## Region

---



Before one can use a Region component properly, it is important to understand what it is. A Region is a container for Bands. In its simplest form the Region could be the whole Page. This would probably be True for reports that are a list type.

Many master-detail reports could be made to fit a single Region design. However, do not be limited by thinking of Regions as the whole Page. The properties for a Region basically deal with its size and location on the Page. Creative use of Regions will give more flexibility when trying to design complex reports. Multiple Regions can be placed on a single Page. They could be side-by-side, one above the other or stagger about the Page. Do NOT confuse a Region with a section. Region components contain Bands and only Bands. A Section component can contain any group of components, including other Region components.

When working with Bands, there is a simple rule that must be followed: Bands must be in a Region. Notice that the number of Regions on a Page is not limited nor is the number of Bands within a Region. As long as the report can be "visualized" mentally, a combination of Regions and Bands can be used to solve any difficulties faced when actually putting the visual thoughts into design. There are two band types: Band and DataBand. The latter is used when iteration is required, for example, in a master-detail report. The first is used for non-iterating needs.

## DataBand

---



The DataBand component is a data-aware Band is used to display iterating information from a data view. In general, a DataBand will contain several DataText components.

A typical use for the DataBand would be on an invoice. An invoice normally consists of a header including information such as the date, invoice number, clients name and address, and one or more lines containing the items that are being invoiced. In this scenario, the customer table would be the master table and the items would be the detail table. Information about the items would be placed in the DataBand and the controller would be the master table.

## Band

---



The Band component is for items that are "fixed" and do not change on the Page. In general, the Band component will contain Text and CalcText components. The primary examples of this would be headers and footers. The Band component can contain data-aware components, so a table field can be in the Band. A group footer might have a '{CustomerDV.CustomerName} Totals' on this Band.

An important property for the Band component is the "ControllerBand". This property determines which DataBand this Band belongs to (or is controlled by). When the controlling Band has been set notice that the graphic symbol on the Band will point in the direction of that controlling Band and that the color of the symbols will match.

There is a preference setting called "Always Show Band Headers". This setting will change the appearance of the Bands while in the design mode. Having this setting "off" will give an appearance closer to the actual output, but will not show the Band Headers with their symbols and codes. When first starting to use Rave it might be beneficial to try it with this setting "on" and take advantage of the visual clues provided by the Band Headers. Once comfortable with the use of Bands, change this setting to fit any necessary needs or preferences. The letter codes shown on the right of the Band are explained later in this chapter under the section called "Editor - Band Style". Basically, they give information about Band behavior. The bold letters are ON or active while the subdued letters are off or inactive.

## Editor - Band Style

Go to a Band Style property on a Band or DataBand component and click on the ellipse button to open the editor for Band Styles.

This provides a simple method to select the features wanted for that Band by using the check boxes to activate or deactivate them. Note that a Band can have several different features active at a time. This means that it is even possible for a Band to be both a Body Header and Body Footer at the same time.

The display area in the Band Style Editor has been designed to represent the flow of a Report in pseudo layout style. DataBand(s) are duplicated 3 times on purpose to show that this is a repeating data area. The current Band that is being edited will be displayed with both **Bold** and Underline formatting.

Both symbols and letters are used on the Band Style Editor display area and one the Bands in the Page Layout Area and are designed be informative about each Band's behavior. The major difference between these two representations is that the Band style editor display will arrange the Bands in a pseudo flow according to the definitions of each Band. The Region display of the Bands will be arranged in the order that they are placed during design. The order of operation is controlled in some cases by this order. Headers (capital letters, BGR) will print first, then the DataBand, then the Footers (lower case letters, bgr) for each level. However, if there is more than one header defined for a particular level, then each header Band will be processed in the order that they are arranged in the Region. So, it is technically possible to put all the Headers at the top, all the DataBands in middle and all the Footers at the bottom of a Region for all levels of a master-detail. Or each level could be "grouped", with the appropriate Headers, Footers and DataBands all together for each level. Rave allows the Region layout maybe be used in a way that makes the most sense to the design flow. Just remember the order of precedence of like Bands at the same level is controlled by their order within the Region.

There are several symbols that are designed to show the Parent - Child/Master - Detail relations of the various Bands. The triangle symbol (up/down arrows) indicates that the Band is controlled by a Master Band with the same color (level) and can be found in the direction of the arrow. The Diamond symbol represents a Master or Controlling Band. These symbols are both color coded and indented to represent the level of Master - Detail flow. Remember that we could have Master - Detail - Detail where both details are both controlled by the same Master or one of the Details could be controlled by the other Detail.

The title bar of each Band contains information about that Band. On the left side of the Band is a name that indicates the Region it is in - "RegionName:BandName". The right side of Band uses several letters to remind you of the Band style settings for that Band. The order of these letters on a master Band is "MASTER 1PC". The order of these letters on a controlled Band is "BGRDrgb1PC". If the letter is subdued (gray) then that setting is inactive (off). If the letter is bold then the setting is active (ON). The following table shows the various letters and what they mean.

## DataText

---



The DataText component is data-aware. This means it can be used to display a field from a dataset anywhere on the Page layout. For example, this could be used to print the customer information inside of a DataBand. The DataText component however is not limited to printing only database data. Through the Data Text Editor (accessible through the DataField property), Project Parameters can be printed, as can Report Variables as well as the DataFields. See the topic Editor-DataText, for more information. The LookupDataView, LookupDisplay and LookupField properties define a lookup definition to be displayed instead of the DataView:DataField properties.

### Editor - DataText

There are two options available for entering data in a DataField property. The first is to select a single field using the drop option. This is fine for normal database reporting needs where only a

single data field for each DataText item may be needed. However, there are many reporting requirements where various fields will need to be combined together. Two common examples are City State and Zip Code or Firstname Lastname combinations. In code this would be accomplished using a statement like the following:

City + '\_' + State + '\_\_\_' + Zip or FirstName + "\_" + LastName

**NOTE:**The underscore character above represents a space for example purposes only.

The DataField property has a DataText Editor which assists in building complex composite fields. To do this click on the ellipsis and open the DataText Editor. This editor will give the power to concatenate fields, parameters, or variables together to build a very complex data aware text field simply by dropping the different list boxes and selecting the item desired.

There are a lot of combinations in this editor, they will be covered quickly here, but try the different combinations in practice and it should help the learning curve.

Note that the dialog box is divided into 4 groups: Data Fields, Report Variables, Project Parameters, Post Initialize Variables and Data Text. Data Text is the result window. Watch this window when inserting different items. The two buttons on the right side of this window are +, "plus", or &, "ampersand". The "plus" sign will add the two items together with no spaces while the "ampersand" will concatenate them with a single space. So, the first step is to decide on doing a + or &, then selecting the text from one of the three groups above the Data Text window.

So as an example, to add the field "OrderNo" to the "CustNo", click once on the "+" sign, go up to the DataField group, drop the DataField list box, and select "OrderNo". Then, click once on the "Insert Field" button and that will be added for the DataText window. The result in the DataText window would be "CustNo + OrderNumber". Even more DataFields can still be added. Notice the "Selected" item in the DataView group. If there is more than one data view active, then select another Data View, and then pick a field from that Data View.

However, do not be restricted to thinking of combining only DataFields. "Report Variables" and "Project Parameters" can be combined as well. Go to the "Report Variables" group and drop the list box for variables and take note of the ones that are available.

Another item available is Project Parameters. This could be a "UserName", "ReportTitle" or "UserOption" parameter initialized by the application. To create the list of "Project Parameters", select the Project node in the Project Tree (very top item). Then in the "Properties" panel there will be a "Parameters" property. Click on the ellipsis button to get a typical strings editor where you can enter the different parameters that will pass to Rave from the application, like "UserName" etc.

**Caution:**

Remember to use a "+" or "&" between each item that you are combining in the Data Text window. You can type in the Data Text window, so you can correct errors by highlighting, deleting or replacing erroneous entries made in the Data Text stream.

## DataMemo

---



The DataMemo component is data-aware. This means it can display a memo field from a DataView just about anywhere on the Page layout. The main difference between the DataMemo component and the DataText component is that the DataMemo component is for printing text that may take more than one line and will thus need to be wrapped. For example, this could be used to print the remarks about a customer invoice at the bottom of each Page of an invoice.

One use of the DataMemo component is to do mail merge functions. The easiest way to accomplish this is to set the DataView and DataField property control to the source of the Memo field. Then launch the mail merge editor by clicking on the ellipsis button of the MailMergeItems property. This will allow the items to be set within that Memo that will be changed.

**Note:** The "Case sensitive" check box is empty (off). If case is important, then be sure to check

this box.

To use the Mail Merge Editor, click on the "Add" button. Now type in the "Search Token" window the item that is in the Memo and will be replaced. After the token is entered, either type the replacement string in the "Replacement" window or click on the "Edit" button and it will start the DataText Editor that will help in selecting different DataViews and Fields.

## CalcText

---



The CalcText component is data-aware. The main difference between the DataText component and the CalcText component is that the CalcText component is specially designed to do some form of calculation and display the results of that calculation. The CalcType property determines the type of calculation being performed and includes Average, Count, Maximum, Minimum, and Sum. For example, this can be used to print the Totals of an invoice at the top of each page of an invoice.

The CountBlanks property determines whether blank field values are included in the Average and Count calculation methods. If RunningTotal is True then the calculation will not be reset to 0 each time it is printed.



# Project Components

**In this Section:**

- Main function of the Project Toolbar discussed.
- Project Manager and special properties covered.
- Report, Page and Global Page components and their special properties covered.
- Data Connections and Objects used to connect to Reporting Server discussed.
- Differences of Security components explained.
- Explanation of SQL and Direct DataView are given.

## Overview

The Project Toolbar provides the basic functionalities for a project. These functions are essential building blocks for all reports. What makes these functions so important is the fact that these functions also provide basic building block components.



The main functions we will be concerned with the first button (Project Manager) and the second section in the toolbar (New Report, New Global Page, New DataView and New Page). These functions are used to add the supporting structure to your report design.

Each will be discussed in the following sections.

## Project Manager



The Project Manager is the component from which all other components are created. Everything is placed under the Project Manager (a.k.a. RaveProject) in the Project Tree. This component is created when any new Rave file is created. There can only be one Project Manager per Rave file, this is unlike most of the other components.

For clarification, Project Manager, RaveProject, and New Project all refer to the same thing. They refer to the main Project file that contains everything.

Like every component in Rave, the Project Manager has properties. To see them, select RaveProject in the Project Tree, and then look in the Property Panel.

To quickly create a new project, simply click on the New Project button in the Project Toolbar. This will create a new file, as well as the new Project Manager.

### Noted Project Manager Properties

The AdminPassword property allows the Rave Server administrator to limit who has access to the data from which the reports are created. This is important to help limit capabilities in Rave.

Parameters property allows items calculated by applications (such as Delphi) to be passed into the report to be used by other components with in the report. When clicking on the Parameters property ellipse button, the DataText Editor will allow a parameter name for each line.

PIVars Parameter is for content that is not typical, and needs to be dynamically modified within the report. This allows for content that needs to be modified in a way that is not typical of table calculation or manipulation.

SecurityControl

## Report

---



The Report component contains the Pages of a Report. There can be multiple Reports in one Project, and each Report can have multiple Pages in that Report.

Each Report has properties. To see the properties, select the Report and view the Property Panel.



The Page component is the base visual component upon which Drawing, Reporting, Standard, and Barcode components are placed. This is where all the designing and layout of a Report are completed.

The Page also has properties which can be viewed by clicking in a blank area somewhere on the page.

## Global Page

---



The Global Pages are located under the Global Page Catalog node in the Project Tree. These Global Page Definitions are used as Master Page definitions. These Pages can also be Mirrored. Global Pages can contain Page layouts for things like letterheads, forms, watermark designs, and other Page layouts that can serve as a foundation for several Reports in the Project. An example would be mirroring a Global Page where you wanted to print the same contents (link an "Invoice"), but you wanted to put a different caption at the bottom of the Pages like "Original", "File Copy", and "Shipping".

Global Pages also have properties. Select the Global Pages from the Project Tree and look at the Property Panel to see all the properties.

## Data Objects

---



Data Objects are the Data Connection components used to connect to data, or components used to control who is allowed to see which Reports from the Reporting Server.

Clicking on the New Data Object button on the Project Toolbar creates each Data Connection. After clicking on the button, the Data Connection dialog will appear. From this point, one of five selections can be made. The Data Object component choices available are DataLookupSecurity Controller, Database Connection, Direct Data View, Simple Security Controller, and SQL Data View.

The details of each Data Connection component will be covered in the next few sections. Once a Data Object is selected, the Data Object component will be placed in the Data View Dictionary in the Project Tree. Selecting the component and looking in the Property Panel will reveal it's associated properties.

## Database Connection

---



The Database Connection component is the Data Object used to connect to data. This component can be added to a Project by clicking on the New Data Object button on the Project Toolbar, then selecting Database Connection from the Data Connections dialog.

Once the Database Connection is chosen from the Data Connection dialog, the Database Connection Type dialog will appear. This is where the type of connection that will be used to

connect to that data will be chosen. In this image there is only one selection, but depending on what types of connections you have, there could be many more selections. After choosing the right data connection type, click Finish.

The Database Connection component, like all other components, does have properties. Select the component from the Project Tree and see the properties in the Property Panel.

## Security Components

---

Using a Security component can control security to individual Reports. TRaveBaseSecurity cannot be used itself, as it is an abstract. However, you can create a descendant or TRaveBaseSecurity to implement your own security scheme or use one of the pre-built TRaveBaseSecurity descendants included with Rave. Rave currently includes two TRaveBaseSecurity descendants: SimpleSecurity and LookupSecurity.

To use a Security component, create one and then set the Report's SecurityControl property to the Security component instance that you want to control access to the Report.

Security components currently only affect Reports when served via the Reporting Server. When an unauthenticated user attempts to access a Secured Report, HTTP authentication will be used to authenticate the user.

### Simple Security Controller



TRaveSimpleSecurity implements the most basic form of security by using a simple list of username and password pairs in the UserList property. UserList contains one username and password pair per line in the format:

Username=password

CaseMatters is a Boolean property that controls whether or not the password is case sensitive. Username is always case insensitive.

### Data Lookup Security Controller



TRaveLookupSecurity allows username and password pairs to be checked against entries in a database table.

DataView specifies the DataView to use to lookup the username and password.

UserField is the field that is used to look up the username. PasswordField is the field that contains the password to verify against.

## SQL Data View

---



SQL Data Views are used for creating self-contained DataViews to SQL databases. A database is specified using the Database property.

Parameters can be specified using the Params property.

The SQL to use is entered into the SQL property. At design time, a property editor for the SQL property invokes a visual query builder.





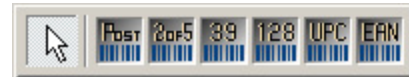
# Bar Code Components

## In this Section:

- Describes how to place Bar Codes
- Describes how to encode Bar Codes.
- Gives brief descriptions of the Bar Code types.

## Bar Code Component Basics

Bar Code components are used to create many different kinds of Bar Codes in a Report. Bar Codes are for users who know exactly what they need, as it requires background knowledge about Bar Codes and how they are used. It is not expected that the beginning user would have the background to use these components.



To place a Bar Code, click on the needed Bar Code component button and click on the Page.

To define the Bar Code value, go to the Property Panel and type the value into the Text property box. For Bar Codes containing check characters, please do not enter the check character, as it will be calculated automatically.

## Brief Bar Code Descriptions

### PostNetBarCode



PostNetBarCode uses the POSTNET (POSTal Numeric Encoding Technique) bar code and is specifically used by the Postal Service. It encodes ZIP Code information on letter mail for rapid and reliable sorting by bar code sorters. The PostNetBarCode can represent a five-digit ZIP code (32 bars), a nine-digit ZIP+4 code (52 bars), or an eleven-digit delivery print code (62 bars). Be aware that for the Post Office to recognize the bar code as being valid strict adherence to the guidelines must be met. Current standards require that the five digit zip plus four be used plus the two digit carrier route, which is most often obtained from the first two digits of the street address. It is recommended that the user check with the Post Office to obtain a copy of their current guidelines.

Example PostNetBarCode: 85210304119

### 2of5Bar Code



2of5Bar Code (interleaved 2 of 5) is a numbers-only bar code; in the Property Panel it is labeled 2of5BarCode. The symbol can be as long as necessary to store the encoded data. The code is a high-density code that can hold up to 18 digits per inch when printed using a 7.5 mil X dimension. "Interleaved" comes from the fact that the digit is encoded in the bars and the next digit is encoded in the spaces. There are five bars, two of which are wide and five spaces, two of which are wide.

Example 2of5BarCode: 2632534

### Code39BarCode



Code39BarCode is an alphanumeric bar code that can encode decimal numbers, the uppercase alphabet, and the following special symbols "\_", ".", "\*", "\$", "/", " ", and "+". The characters are constructed using nine elements, five bars and four spaces. Of these nine elements, two of the bars and one of the spaces are wider than the rest. Wide

elements represent binary ones (1), and narrow elements represent binary zeros (0).

Example Code39: CODE 39

**Code128Bar Code**



Code128Bar Code is a very high-density alphanumeric Bar Code. The symbol will be as long as necessary to store the encoded data. It is designed to encode all 128 ASCII characters and will use the least amount of space for data of 6 characters or more of any 1-D symbology. Each data character is made up of 11 black or white modules. The stop character is made up of 13 modules. Three bars and three spaces are formed out of these 11 modules. Bar and spaces vary between 1 and 4 modules.

Example Code 128: Code 128

**UPCBarCode**



UPCBarCode (Universal Product Code) has a fixed length of 12-digits and can only encode numbers. UPC was designed for coding products. The format allows the symbol to be scanned with any package orientation. The check digit is calculated so there is no need to enter it when typing the value into the Text property.

Example UPCBarCode: 712345678935

**EANBarCode**



EANBarCode (European Article Numbering System) is identical to the UPC, except for the number of digits. EAN has a length of 13 digits - 10 numeric characters, and 2 "flag" characters that are usually country codes, and a check digit. This Bar Code is typically used for Non-U.S coding. The check digit is calculated for you so you do not need to enter it when typing the value into the Text property.

Example EANBarCode: 3847348484584

# Advanced Components

## In this Section:

- Use of the FontMaster and PageNumInit are explained.
- Details of some advanced components are given.
- Calc Component details are given.

## FontMaster



Each Text component in a Report has a Font property. By setting this property, a specific font can be assigned to the component. In many cases it may be somewhat useful and necessary to set the same font properties of more than one object. Although this could be accomplished by selecting more than one component at a time, this method has a drawback. One has to keep track of which fonts have to be of the same typeface, size and style, which is not an easy task when there is more than one report. This is where the FontMaster comes into play. Apart from allowing the user to define a global font for various components, the FontMaster also allows the user to define standard fonts for different parts of the Report, like for instance the headers, body, and footers.

The FontMaster is a non-visual component (designated by the green color of the button). To use one, simply click on the button and then click anywhere on the Page Designer. Note that being a non-visual component, there will be no visual reference of it on the actual Page, and like other non-visual components, it can be accessed using the Project Tree.

As mentioned previously, the main purpose of the FontMaster is to set fonts. It has very few properties. As most other components, it has the DevLocked, Locked, Name and Tag properties. They also contain the most important one of all, which is the Font property.

To set the FontMaster property, use the Font property on the Property panel, click on the ellipse button to get to the Font Dialog box and select the font and size settings. Click OK when done.

Once the FontMaster component is set, linking it to a body of text is simple. On the Report select a Text/Memo component, then use the down arrow button on the FontMirror property in the Property Panel to choose a FontMaster link. Any component that has the FontMirror property set to the FontMaster will be affected by and change to the FontMaster's font property. This allows the user to change fonts on various Text components at once and at the same time keep things consistent across the Report (when required).

It is important to note that by setting the FontMirror property on a component, the Font property will be overridden by the Font property of the FontMaster. This means that if a text object has a Font setting of Font A, by assigning a FontMaster to it with Font B, the Text component will automatically be assigned Font B and it's Font property will be ignored. Another side effect of using the FontMaster is that the Font Toolbar is disabled when the FontMirror property is set for that component.

There can be more than one FontMaster per Page; however, it is good practice to usefully rename a FontMaster. The Project Tree image shown previously has three FontMaster components on the Page. Two of them begin with FM, for FontMaster, and they follow with the name and size of the font that they represent. This is one naming convention that can be used to make the name descriptive and useful to the user.

## PageNumInit

---



PageNumInit is a non-visual component that allows the restart of page numbering within a Report. Using a PageNumInit is similar to other non-visual components. It is used when more advanced formatting is required.

An example of using PageNumInit is in a CustomerStatement Report of a checking account. Account statements that customers receive every month may vary in the number of pages. In the monthly statements, the first page can define the account summary page layout, the second can define the customer's credits/deposits, and the third the withdrawals and debits. The first two pages may only be one page, but if the account activity is high for the customer then a section like withdrawals could be several pages. If the user producing the report wants each section numbered individually, the summary and deposits would have the pages marked "1 of 1" for both. For Withdrawals, an active customer account could have three pages of withdrawals and debits. Since this is a different section in the statement, it needs to be marked "1 of 3", "2 of 3", and "3 of 3". Using this kind of multiple page numbering is where PageNumInit comes in handy.

There are a couple of steps that must be completed to use the component efficiently. First define the Report Pages as usual. Add a DataText component from the Report Toolbar to the Page, placing it where the Page requires a Page number to appear.

Once it is in place correctly, in the Property Panel of the DataText component, click on the ellipse button in the DataField property.

This will open up the DataText Editor. Click on the Report Variables down arrow button and select Relative Page and click on Insert Report Var button to use the variable on the Report.

In the DataText area, "Report.RelativePage" should appear. After the text type in the following: + ' of ' +

From the ReportVariables drop-down list select TotalPages (choose it and press the Insert Report Var button). This will add the remaining text "Report.TotalPages" to the Data Text edit box area. When finished click OK.

The Property Panel now displays the sentence that was entered using the Data Text Editor.

Expanding the DataText component, the text from the editor will also appear.

To initialize the Page to a desired Page number, select PageNumInit from the Project Tree. On the Property Panel, type the desired Page number into the InitValue property box.

For every Report definition created in a Project, the PageNumInit will allow each report definition to be numbered independently of another.

## DataCycle

---



The DataCycle component is an invisible data-aware band that would be used to control iterating information from a DataView. In general, a DataCycle component would be used to "Loop" or "Cycle" through the detail records until a change occurs at the master record level.

For example, imagine a bill being generated for multiple customers and each customer has many different purchases associated with them. For this bill you need to have each person's purchases on their own billing page, which has all and only their own listing of purchases. The table design for this might look like the following tables, where there is a master table listing of all the unique customers, and each customer has their own purchasing table, with all their own purchases listed.

The bill being created will need to have each unique customer ID number printed on one page as well as the items purchased on that page. Thus, the report will have to cycle through the table containing all the purchases made before proceeding to printing the next customer and the associated purchased items.

The main table containing the list of customer ID's would be found in what is called the MasterDataView. In the Property Panel, this master table name would be chosen from the property called MasterDataView. Once that is chosen, we will need to tell Rave what key will link the two tables together. This key is called the DetailKey. The DetailKey, in our example, would be the Customer ID's. Next, the table that will be cycled through, which is the table containing purchases to be printed, will be the DataView table. So, in essence the MasterDataView (driving table) and the DataView (details table) are connected by the Detail Key (the common unique key in each table).

The DataCycle can also be limited, or sorted while the Report is being generated. For example, supposed you just wanted to create reports for the Customers in Arizona. To limit Report creation to just the Arizona customers, the MasterKey will have to be set.

Setting the MasterKey is done by clicking on the ellipse (...) button next to the Property in the Property Panel. The Data Text Editor will appear. In the Data View area, choose the "Selected" radio button and then select the appropriate MasterDataView table, which is the Master Table in our example, by using the drop-down menu. Once you have done that, next select the field that will be used as the filter of the Master Table, or MasterDataView table. Once selected, click on Insert Field that is just underneath the Data Field drop down menu. This will place the appropriate text in the Data Text area at the bottom of the Editor. Now in the Data Text box, type in "=<code>AZ</code>" to limit that field to just process the AZ customers. Click OK when done, and that will filter out non-Arizona customers.

## DataMirror Section

---



The DataMirrorSection component is a section that will "Mirror" other "Sections" based on the contents of a "DataField". Mirroring Sections allow this component to be very flexible. Remember that Sections can contain any other component including graphics, regions, text, etc.

An example using this "DataMirrorSection" component is included in the RaveDemo project file. This example shows how to have a single report produce different envelope formats when sending to International or US addresses. The template for the International customers includes the country line and is centered on the envelope, while the US format does not have the country line and is offset to the right of center and lower on the envelope.

This example shows several techniques on how to use Sections and Mirrors. On page 2 there are three Sections. Section 3 has the common address lines used by both templates. Section 1 and 2 contain Mirrors to Section 3. The International Section has the additional country line added in its template. Note, that "remarks" have been added to the page 2 design and they are not part of any of the sections. The "US Template" and "International Template" are just plain Text remarks that assist in the purpose of each Section shown.

Once understanding how the Master Templates were done, go to Page 1, and select the "DataMirrorSection". Notice the "DataView" and "DataField" properties have been set to the field that will be used to select which Format Template to Mirror. To examine that logic, go to the "DataMirrors" property and click on the ellipsis to open the Data Mirror Editor. Select each "DataMirror" item and note the settings in the bottom portion of the dialog box. This example has two "templates", but feel free to add more to fit more complex reporting needs.

### NOTE:

Normally one of the settings will be defined as the default. If a default is NOT defined and the field value does not match any of the other settings, then the format used will be the normal contents of the DataMirror Section component.

## CalcOp

---



The CalcOp component is a non-visual component that allows an operation (defined by the Operator property) to be performed on two values from different sources

(Src#CalcVar, Src#DataField or Src#Value). The result can then be saved in a project parameter like CalcTotal (through the DestParam and DisplayFormat properties).

For example, there could be two DataText components that need to be calculated together. Like  $A+B=C$ , where A and B represent the two DataText component values and C represents the result. This is where the CalcOp component would be used.

To choose the DataText components in the above example, use the Src1X and Src2X properties, where X is either CalcVar (a calculation variable), DataField, or DataValue. These are the three types of value sources that can be used for calculation in a CalcOp component. Note that the Src (source) can only use ONE of the three available source types, and there can only be two sources, Src1 or Src2.

The three source types have many different values associated with them. For a CalcVar source, the drop down menu will list all the calculation variables available in that Page available for use. This calculation variable is just that, a variable for holding a value. This value can be from another CalcOp component or from some other calculation component. For the DataField source to get available values first designate what DataView the DataField will come from. In other words the DataField is a field in a table, which is the DataView. So, in order to choose a field, the DataView must first be chosen. For a Value source, typing in a numeric value is all that is needed to fill this property. Again, remember only one of these three sources can be assigned to a source.

Once the sources are chosen the operation to be used between them has to be chosen. To choose the desired operation, use the Operator property by using the drop down menu and making the appropriate choice. In the example,  $A+B=C$ , the operator would be "coAdd".

There may be times when a function needs to be performed on a value before it is processed with the second value. This is where the property SrcYFunction, where Y is 1 or 2 for Src1 or Src2, will become handy. With SrcYFunction, the value can be converted (like from hours to minutes), or have a trigonometric function performed on it (like take the sin of the value), or have other functions performed on the value (like take the square root of the value, or take the absolute value).

Once the two values are chosen, it is now time to deal with the result. In the example,  $A+B=C$ , C is the value of the result. The result can either be written out to a project parameter (most likely a DataText component) or just held as an intermediate value. To write out the value to a parameter, use the DestParam property to choose the desired parameter to write the result write to. If the result is going to be in intermediate result, there is nothing else to do to the component. Although, it is highly suggested to rename the component, by using the Name property, to easily recognize the component for future use. This is because, as in intermediate values, you will most likely use the components in another CalcOp component, and a good reference will help to easily create the next step in the calculation process.

After setting the values and setting the result destination, it may be necessary to format the result or to even perform one last function on the result. Using the DisplayType and DisplayFormat properties to format the result into any necessary format. The DisplayType has two options, DateTimeFormat and NumericFormat. DisplayFormat has many options that must be typed into the edit box. To find these options look at the Formatting Appendix in the back of this manual. Like the two values, A and B, in the example,  $A+B=C$ , the result C can also have a function performed on it before being written out to a parameter, or as a holding value. To select a function, use the ResultFunction property drop down menu.

The CalcOp components can also be chained together for more complex expressions using the Src#CalcVar properties, which can be set to other CalcOp or CalcTotal components. For example, to create the complex expression shown to the right, it will be necessary to break up the expression into simple 2 value steps.

To break up the complex expression, four new expressions will need to be created and saved as CalcOp components. So the resulting components, Z, Y, X, and W will be four separate intermediate CalcOp's. But, as you can see from the flow of the expression, Z and Y will have to be completed before X can be created, as it is dependent upon the two previous operations. And finally, W will be completed last, after X, to get the final correct value.

$$\frac{(A+B)*(C*D)}{0.80} = E$$


---


$$A+B=Z$$

$$C*D=Y$$

$$Z*Y=X$$

$$\frac{X}{0.80}=W = E$$

Just as it is important to do the calculations in order, it is important to make sure the components are in order in the Project Tree. When a report is executed, the report executes components down the Project Tree. For CalcOp components or any calculation component, this means that they need to be in the correct order. In the example above, if Z, Y, X, and W were in the Project Tree, Z would have to be first in the tree and W would have to be last in the tree. This would mean that at execution, Z would be processed first, then Y, and so on. It is also important to note that if Z is dependent upon any other components (like other CalcOp components or DataText components), those components must come first in the Tree before the Z component is processed. Making sure the components are in correct order is known as setting the print order. To move the components up and down within the Project Tree, use the Alignment Palette to change the print order of the components.

## CalcController



CalcController is a non-visual component that acts as a controller, along with DataBands, for CalcText and CalcTotal components through their Controller properties. When the controller component is printed, it signals all calculation components that it controls to perform their summation operation. This allows totals to be performed on group bands, detail bands or whole pages depending upon the location of the CalcController component. Another feature of the CalcController component is its ability to initialize a CalcText or CalcTotal component to a specific value (through the InitCalcVar, InitDataField and InitValue properties). A CalcController component will only initialize values if it is used in the Initializer property of CalcText or CalcTotal property.

## CalcTotal



The "CalcTotal" is a non-visual version of the CalcText component. When this component is "printed", its value is typically stored in a project parameter (defined by the DestParam property) and formatted according to the DisplayFormat property. This can be useful when performing totaling calculations that will be used with other calculations before being printed. Leave DestParam blank if the value of CalcTotal will only be used by other calculation components such as CalcOp.

## Exercise: Using Font Master

1. Create a new page in the report. Select New Report Page from the Project window.
2. Go to the Project Tree and select the Page.
3. In the Property Panel, type in "FontMaster" in the Name property, while the Page is still selected.
4. Go to the Standard Toolbar. Find the FontMaster component and click on the button. Then click on the Page.
5. Look at the Project Tree Panel and notice that there is a new component underneath the Page called FontMaster1. But, when looking at the Page, there is no visual component. FontMaster is a non-visual component.



6. Make sure the FontMaster component is selected by clicking on it in the Project Tree. The component will be highlighted in the Project Tree to indicate that it is selected.
7. While the FontMaster component is selected, look at the Property Panel. In the Font Property, click on the ellipse button (the button with three dots).
8. The Font Dialog will appear after clicking the ellipse button. Use the scroll buttons and check mark boxes to make your desired selections. For this first FontMaster component, select Arial Font, Bold Font style, 14 size, and Underline in Effects.
9. Look at the Property Panel, with the component still selected. The Font Property will reflect the selection made in the Font Dialog.
10. Now with the Font Property still selected, scroll to the Name Property and put "FMArial14BldUndrln" in the Name property edit box.
11. Next look at the Project Tree and notice the name change. It becomes very useful and helpful to rename the components in the Project Tree in order to distinguish each component from each other. This is why we demonstrate renaming of the FontMaster component and the Page component.
12. Now, complete steps 4 to 11 three times using the following names and settings. FMArial16ItlcUndrln: Arial Font, size 16, Italic, an Underlined. FMTimesNwRmn12: Times New Roman Font, and size 12. FMCourier12: Courier Font, and size 12. To make understanding the renaming of many components, sometimes it is helpful to use a specific naming convention. In these examples we use the following naming convention for the FontMaster component names:
13. Next, drop down five Text components on the page, as well as one Memo component.
14. We will pretend to write a "letter" using components dropped on the Page.
15. Use one Text component to use for a date holder, and use the other four Text components to make an address label. The Memo component can be used for the body of the message. So, fill your components appropriately. For more information on how to fill the components with your own text, see the Property Descriptions Listing in the appendix, or see the chapter on Standard Components. Feel free to use the alignment tools to get the "letter" components to align correctly. Also, after some of the following steps it may be necessary to resize the text components to display all the text correctly.
16. Next select the Text component that has the date. While selected, change the FontMirror Property to FMCourier12 by using the drop down menu. Notice that the font of the Text component changed to the pre-set font settings of the FMCourier12.
17. Next select the four address Text components. The Property Panel will show the designation that this is a multiple selection of Text components. In the FontMirror property, choose FMTimesNwRmn12.
18. Select the Memo component, which contains the body of the "letter". While it is selected, choose FMArial16ItlcUndrln in the FontMirror drop down menu.
19. This ends mirroring of fonts to components on the Page. One last thing to cover is what to do when you change your mind about the fonts?
20. Select all the address Text components. Change the FontMirror property to FMArial14BldUndrln. Now, this would be a way to change many Text/Memo components from one font setting to another, without changing each Font property in each component.
21. Sometimes there may be many components linked to one FontMirror component, because they all relate to one specific area of the report. In this example, we can assume that all address headers would be linked to the FMArial14Bldunderln Font Mirror property. So, to change this we would have to redo the one FontMirror property.
22. Select the FMArial14BldUnderln FontMirror component in the Project Tree.
23. In the Property Panel, while the component is still selected, click on the Font property ellipse button.
24. Change the Font Dialog properties to the following: Times New Roman, size 10 font, with no other Font Effects.

25. In the Name property of the FontMirror property, put the following name:  
FMTimesNwRmn10.

## **Exercise: Setting up PageNumInit for Page Numbering**

---

1. First create a new Page, or use what is left from the previous example, as we will do here. If this is a new Page, drop some Text components on the Page. These are just filler components in this example.
2. Go to the Report Toolbar, and find the Data Text component. Click on the Data Text component and then click on the Page.
3. Place the DataText component in an area where a Page number might appear. For example, place the DataText component at the bottom of the Page.
4. Make sure the DataText component is placed correctly, and make sure it is still selected. Check for selection by the visual appearance of the green pip surrounding the component, or by the component being highlighted in the Project Tree.
5. While the DataText component is selected, go to the Property Panel and look for the DataField property. Click on the ellipse button.
6. The Data Text Editor will next appear. Look for the *"Report Variables"* section in the Editor.
7. There will be an arrow at the end of the Report Variables selection window. Click on it to see all the selections available. Use the scroll bar to move up and down through the menu.
8. In the Report Variables section, look for *"Relative Page"*. Click on it to select.
9. Next click on the Insert Report Var button to the right of the Report Variables section. This will place the appropriate text into the Data Text area at the bottom of the Data Text Editor.
10. Go to the Data Text area and type in the following: + 'of' + . This will include spaces before and after each symbol.
11. Now return to the Report Variables drop-down list. Scroll to look for TotalPages. Select it when you find it.
12. Click the Insert Report Var button, to get the remaining text into the Data Text area. "Report.TotalPages" will be added to the Data Text area.
13. Click OK when done. Your DataText area will look like the following.
14. After clicking OK, you will be returned to the Designer and to the Property Panel. If you expand out the Property Panel and look at the DataField property, you will see the results of the DataText are from the Data Text Editor.
15. On the Page, the DataText component will probably look like the DataText component labeled A below. If you expand the borders of the DataText component (as seen in B), you can see that the results of the Data Text Editor appear. There is really no need to expand the component; it is just for you to see in this exercise.
16. There is one more thing left to do to complete the page numbering, we need to initialize the Page to the desired page number. So, go to the Project Tree and find the PageNumInit component. Select the component when you find it.
17. While PageNumInit component is still selected, go to the Property Panel and find the InitValue property.
18. In the InitValue property, type in 1. This will mean that our pages will begin with page 1.
19. That is it. That is all that is needed to setup the PageNumInit. Note that in one project, with this feature, each Report can be number independently of each other.



---

# Adaptable Reports

In this Section:
<ul style="list-style-type: none"><li>• Explains the problems related to adaptability.</li><li>• Explains how to use Anchors to "fix" your components.</li><li>• Shows how to work with waste areas on different printers.</li></ul>



---

## Overview

One of the biggest problems that designers face when creating reports is adaptability. What is adaptability exactly? First, when a report is created, the process normally takes place on a specific computer, which has one or more printers connected to it. When that report is executed in the field, however, factors such as paper size, orientation and waste areas all come into play. Secondly, reports should allow for a certain amount of flexibility in the overall design and appearance of the layout of the report.

Rave helps overcome these problems by giving the designer access to particular properties of the report. In this chapter, these properties will be explained in detail and will allow the resulting reports to have a standard look over various platforms. This way, the designer can focus more on creating reports than worrying about deployment difficulties. The reports will also be able to adjust layouts based on external parameters passed in during execution.

---

## Anchors

Most visual components have a common property called Anchors. This powerful feature defines how the object will move when its Parent is resized. There are two Anchor values that are set, one for the horizontal and one for the vertical. The default Anchor is set to the top left corner of the Parent control. What this means is that when the Parent (e.g. a Section component) is resized, the component will stay the same distance from the left and top corner of the Parent. Two other Anchor values, Bottom and Right, are very similar in function (they will Anchor to the bottom right corner of the Parent).

The Anchor setting of Center will Anchor the Child component to the Parent's center. Stretch will actually resize the Child component so that it's sides stay the same distance from the Parent. Stretch is most often used when you want a Child component to always match the width or height of the Parent. Two other special Anchor settings, Resize and Spread are useful for groups of components. Resize will proportionally resize the components and the spaces between them as the Parent is changed. Spread will proportionally resize the spaces only (the components will stay the same size) as the Parent changes. Drop down a Section component and place a few rectangles or Text components inside and change the values of those components to see how the Anchor property can affect things.

Anchors can be used to create adaptable Reports when combined with other Reporting features. Imagine that a Report needs to be defined that can be printed in either landscape or portrait orientation or that the Report may be run on different size papers. Setting the Anchors properly will allow one Report to adjust to these changing conditions. A typical table listing Report will be composed of a Region component; it's Band components and the Text and DataText components inside each band. If the Region component is set to Anchor Stretch on both vertical and horizontal and the Text and DataText components are all set to a horizontal Anchor of Resize, the Report will adjust to any of these changes.

Other changes, such as the printer's individual waste area, can also be solved with Anchors and

will be discussed in the next section.

## Waste Fit

---

Most printers have a Region on the paper that is called the waste area. What this represents is the portion of the Page where no printing can be done because of restrictions of the actual printing hardware (usually because the printer uses that region of the page to "grab" a hold of the paper when it is feeding it through the rollers). These values may range from 0 (common for dot-matrix printers) to 1.5 inches. Ink jet printers typically have large bottom margins (and sometimes a 0 top margin), while most ink jet and laser printers have about a 0.25 inch left and right waste area.

The red-dotted lines that run on the border of the Page represent the waste-area of the currently selected printer (or specific values if the preferences are set that way). Anything in between the red line and the border of the Page is the actual waste-area of the printer and you should avoid placing components within that area. The problem is how to know what the actual waste area of the destination printer will be. If the Report is designed with extremely large margins that will be inside the waste area of all printers, then a lot of unnecessary blank space will be on each Page. The WasteFit property is Rave's answer to this common reporting problem.

The Page has a property called WasteFit, which can have one of two values, either true or false.

By default the value is False. Setting the property to true will make the Page adjust the components it contains so that they fit within the waste-area of the destination printer. The components contained will be resized so that they all fit in and adjust accordingly.

However, it is not sufficient with setting the WasteFit property to True for all this to happen automatically. Child components should have their anchor property set accordingly so that they too can adjust accordingly to the changes of the Page.

When used properly, this property is a very powerful feature that allows the reports developed to automatically adapt to different destination printers.

## Editor Anchor

---

Normally, ReportBands/columns/paragraphs are justified to the Left and Top of a design area. This will be fine for the majority of Reporting needs.

However, have you ever wanted to control the starting or ending position of your report components dynamically? This could be a two-column Section, where the left column is a Memo component which changes in height and the right column is a Text component, which is always a one line item. How do you make both components "float" and align to the bottom of the design block? In Rave this is done with the Anchor property.

To change the Anchor style go to the Anchor property of the desired component and click on the ellipse symbol, it will open the Anchor editor like the one shown above. This provides a method to select the Anchor style wanted for that component by using the appropriate vertical and horizontal radio button. Note that a representation appears below each Anchor selection to give a visual indication of what that setting is designed to accomplish. The last three settings, Stretch, Resize, Spread are a little difficult to explain, but view the sample and there will be a picture showing the differences between the settings.

An important point to note is that the Anchor property settings are relative to that component's Parent. So, if the Parent is a Page, then the settings are relative to the Page sides. If the Parent is a Section, then the settings are relative to the section borders. One way to visually see the parentage is to examine the component(s) on the Report Node in question in the Project Tree. Go up one level and that is the Parent.

The following combinations are not restrictive, but normally the Anchor settings will be paired as follows:

- Left/Top justified
- Right/Bottom justified
- both Center justified
- both Stretch
- both Resize
- both Spread



---

# Batch and Chain Reporting

In this Section:
<ul style="list-style-type: none"> <li>• Discusses how to Batch Reports</li> <li>• Covers calling and Page Chaining</li> </ul>



---

## Batch Pages

Probably the most common linking of Pages would be a batch processing sequence that defines a list of independent Page definitions. The first Page in the list runs to completion, then it calls the second, which runs to completion, and so on until the last Page of the defined sequence has been completed. The main thing to remember with batch Pages is that each Page definition is independent of the others and runs to completion before the other Page starts. Remember that the requirement to do batch processing is to simplify the administrative tasks when running a group of Page definitions on a recurring basis.

Define the sequence of Pages to print through the PageList property, which is available at the Report node level. When this property is selected (click on the ellipsis button), it will open a dialog window that will build the list from the Page definitions within the selected Report node. The advantage of this method is that the PageList is independent of the individual Page definitions. This means that the PageList would run the whole sequence of Pages (Reports), but individual Reports can still be selected individually and run by itself without invoking the batch link. Remember, Report Pages are not visible to other Reports, so if flexibility of calling separate Page definitions is needed, define the reusable Pages as global Pages instead of Report Pages.

### WARNING

Another method to do Batch Pages is to set the first Page definition GotoPage property to the second Page definition name. Now, when the first Page is completed, it will automatically start the second Page definition. The problem with this technique is that anytime the first Page is run it will ALWAYS start the second Page as they are linked together at the Page definition level.

---

## Calling Pages

### (Page Node - GotoMode property - gmCallEach setting)

Another type of linking would be Reports where there must be a particular flow of Pages. The easiest example of this would be a Report where each record ALWAYS produces three Pages of output. This could be a "form Report" where Page 1 has a patient's demographic information, Page 2 has medical information and Page 3 has insurance information. So, when going through the database, each patient's record would produce 3 Pages of output. See the Exercises for more step-by-step instructions on how Call Pages.

---

## Chain Pages

### (Page Node-GotoMode property-gmGotoDone setting)

A chain of Pages is similar to a Batch. The Batch technique discussed earlier used the PageList property and cautioned you about using the GotoPage property. However, if you plan ahead, then the use of the GotoPage property with the Pages from the Global Page Catalog is very powerful. The exercise at the end of this chapter does a Multi-Page definition that includes an invoice, file copy, shipping document and packing slip.



## Different First Page Format

---

### (Page Node-GotoMode property-gmGotoNotDone setting)

Another type of linking would be Reports where the first Page format is different than the remaining Pages. This could be a Report that has a title layout maybe even with a company logo on Page 1, but the remaining Pages are all the same design layout. What you need to do for this is to create the Page 1 definition (first Page) that points to Page 2. Then create the Page 2 definition (all remaining Pages) that points to blank. The Exercise at the end of the chapter goes through the detail steps to complete this process.

## Different Odd/Even Page format

---

### (Page Node-GotoMode property-gmGotoNotDone setting)

Another type of linking would be Reports with a format based on an odd/even Page definition. This could be Reports that are going to be printed on both sides of the paper (duplex style) and have holes punched in one side, so that final Report could be put in a binder. This would mean that the inside margin (say 1 inch) would be larger than the outside margin ( 1/2 inch). To create a loop where the Page 1 definition (odd Page) points to Page 2 and will call it if the Report is not done when it gets to the end of the Page AND the Page 2 definition (even Page) points to Page 1 and will call it if the Report is not done at the end of that Page. This loop will need to continue until Report is completely printed. See the Exercise at the end of this Chapter to get detailed steps to complete this task.

## Batch / Chain Reports

---

Once the Report Design is complete and working, the day-to-day routines get settled. At this point, it would not be unusual to find out there often is a need to print several Reports in some kind of sequence. This could be a group of Reports that are always generated at the end of each month, quarter, etc. Another example might be a series of executive summaries that are needed on a demand basis by upper level management. Of course, it is possible to run each required Report individually for these repetitive cases. Then when they are all done, group them together and give them to the requesting office. This often means that there is a checklist that details what repetitive Reports are required, when and for whom.

This problem of producing a repetitive sequence of Reports is solved with Rave by linking the Reports to each other. Rave has the ability to link the Report Page definitions in a wide variety of ways by setting different combinations of three properties; Page.GotoMode, Page.GotoPage and Report.PageList. It is important to note that the setting of the GotoMode property determines the behavior of the GotoPage property. The PageList property is at Report Node level and is designed to provide a way to initiate several different GotoPage property chains in a defined sequence.

Although we will give examples of how we envision achieving various Batch/Chain sequences of Pages, it will not be possible to include all of the possibilities. The secret is that one should understand and be flexible on how to "mix and match" the various parts of Rave. In particular, pay special attention to the Global Page catalog and the GotoMode, GotoPage, Mirror and PageList properties. Different combinations of these provide a wide range of output options. There is a lot of power here, so the best advice is to start with simple Reports. Then add some of these powerful extras and as they make sense.

## Exercise: Calling Pages

---

The way to accomplish this would be:

- Complete all normal definitions for the Page 1.

- Complete all normal definitions for the Page 2.
- Complete all normal definitions for the Page 3.
- Set the GotoPage property of the first Page to point to the second Page.
- Set the GotoMode property of the first Page to gmCallEach setting.
- Set the GotoPage property of the second Page to point to the third Page.
- Set the GotoMode property of the second Page to gmGotoDone setting.
- Insure that the GotoPage property of the last Page definition is blank.

The gmCallEach setting of the Page 1 GotoMode property will activate the GotoPage property after the first physical Page has printed. So, at the end of physical Page 1, Page 2 definition will be started. Because the Page 2 definition has a gmGotoDone setting, it will go to the Page 3 definition when it has finished the printing the second Page. The Page 3 definition will print that Page's definition and then return control to the Page 1 definition because it's GotoPage property is blank. You can join any number of Pages together in this manner. The key is that a gmCallEach setting will save off the calling Page so that whenever a blank GotoPage property in the Page chain is encountered, it will continue with that calling Page.

## Exercise: Chain Pages

---

For this example, we are going to do a Multi-Page definition that includes an invoice, file copy, shipping document and packing slip.

- Create a section on a Global Page that will be mirrored later.  
Complete all of your normal definitions for the Invoice Page within this section.
- Create a section on a Global Page that will be mirrored later.  
Complete all of your normal definitions for the File Copy Page within this section.
- Create a section on a Global Page that will be mirrored later.  
Complete all of your normal definitions for the Shipping Page within this section.
- Create a section on a Global Page that will be mirrored later.  
Complete all of your normal definitions for the Packing Slip Page within this section.
- Create a "New Report" that will hold the definitions for these Pages.
- Create a new Page definition, "Invoice"  
drop a section on this Page definition and set the Left and Top properties  
mirror the section to the "Global" Invoice Page section  
drop a text component on the bottom of the Page "INVOICE COPY"
- Create a new Page definition, "FileCopy1"  
drop a section on this Page definition and set the Left and Top properties  
mirror the section to the "Global" FileCopy Page section  
drop a text component on the bottom of the Page "FILE COPY 1"
- Create a new Page definition, "FileCopy2"  
drop a section on this Page definition and set the Left and Top properties  
mirror the section to the "Global" FileCopy Page section  
drop a text component on the bottom of the Page "FILE COPY 2"
- Repeat those steps, for the "Shipping Document" and "Packing Slip"

- Now go back to the "Invoice" Page definition (NOT the global definition)  
Set the PageMode setting to gmGotoDone  
Set the PageGoto to point to the "File Copy 1" Page definition.
- Repeat these steps for each of the copies, remember to leave the last one blank

The mirroring of a Global Page offers a lot of "reuse" of a Master Design. This example showed that the "FileCopy" could be mirrored twice and "label" each Page definition differently.

## **Exercise: Different First Page**

---

The way to accomplish this would be:

- Complete all normal definitions for Page 1.
- Create a "New Report Page" for the second Page definition.
- Set the GotoPage property of the first Page to point to the second Page.
- Set the GotoMode property of the first Page to gmGotoNotDone setting.
- Complete all normal definitions for Page 2.
- Insure that the GotoPage property of the second Page is blank.

The gmGotoNotDone setting of the GotoMode property will activate the GotoPage property after the first physical Page has printed but only when the current Page definition has not finished (for example, EOF - End Of File). So, at the end of physical Page 1, Page 2 definition will be started. Because the Page 2 definition does NOT have anything in the GotoPage property, the Page 2 definition will remain in effect for all remaining physical Pages until the Report is done.

## **Exercise: Different Odd/Even Page**

---

The way to accomplish this would be:

- Make a Section on the first Page definition that is set for the "Odd" Page margins.  
For example, set the properties Left = 1.0, Width = 7.0, Top = 0.5 and Height = 10.0.
- Complete all of your normal definitions for Page 1 within this section.
- Create a "New Report Page" for the second Page definition
- Drop a Section on the Page 2 and set the top and left margins for the even Page settings.  
For example, set the Left property = 0.5 and Top property = 0.5.
- Set the Mirror property of Page 2 Section to point to the Section on Page 1.
- Set the GotoPage property of the second Page to point to the first Page.
- Set the GotoPage property of the first Page to point to the second Page.
- Set the GotoMode property of BOTH Pages to gmGotoNotDone setting.

The gmGotoNotDone setting of the GotoMode property will activate the GotoPage property after each physical Page has printed but only when the current Page definition has not finished (for example, EOF - End Of File). So, at the end of physical Page 1, Page 2 definition will be called. At the end of physical Page 2, Page 1 definition will be called. This loop will continue until one of

the Pages is completed. If control would need to be passed to another Page at this point, use the Report components PageList property to select the next Page.



## Chapter 20

# Preferences

In this Section:
<ul style="list-style-type: none"> <li>• Shows how to get to the Preferences Dialog</li> <li>• Explains each tab in the preferences dialog</li> </ul>



---

### Getting to the Preferences Dialog

Before getting started in Rave lets take a short detour and learn about controlling the appearance of the RAVE designer. Under the main menu, select "Edit" then "Preferences". This will open a Preferences dialog where the look and feel of the RAVE system can be customized.

---

### Defaults Tab

"Defaults" tab provides a place to set many of the RAVE Visual Designer Project and Page settings.

Settings in the Default tab section will be used when a new Project or new Page is created. After a Project has been started, the Project properties can be changed at the Project level and then they will be saved for that Project. The same applies to the properties of Pages. For example, a Project could have a default grid of 0.25 inches, but on a special Page the settings could set the Grid Spacing to 0.1 inches.

---

### Designer Tab

The "Designer" section controls changes of the Grid system, Alignment options, units of Measurement and Zooming.

The "Zoom Increment" controls the amount of change that is made with the in/out zoom controls on the Zoom Toolbar.

The "Alignment Options" section, controls the visibility of the Order buttons and Tap buttons. It also controls the 'speed key' assigned to the Tap Tools. The speed key is really a predetermined amount of distance that the component is allowed to move with one keystroke. To use this feature, select the component and click on any of the arrow keys on the toolbar. Or, use the arrow keys on the keyboard while pressing the Ctrl-key.

In the "Grid Settings" area, grid color and associated changes can be applied to the Page of the Report. "Snap To Grid" will cause the mouse cursor to move and size components in increments of the Grid Spacing. The "Draw Grid On Top" option allows the Grid to be seen visually on top of components that have a solid background color. To hide the Grid Lines, make sure that "Show Grid" is unchecked. Several of these preferences are controllable from the Designer Toolbar. For more information about the Designer Toolbar and other toolbars see the Utility Toolbar Chapter.

The "Waste Area" controls the visibility of the amount of excess between the area border (the red dashed line) and the edge of the Page. "Show Waste Area" hides/shows the red dashed border around the Page. Then next two selections control the Waste Area width. The Waste Area can be set to the default printer settings, or it can be set manually by using the Left, Right, Top, and Bottom edit boxes.

In the "Background Gradient" section, the three colors that make up the area between the Page and the borders of the program can be changed. Clicking on the color wheels and selecting the desired colors change the gradient of the background. To make the background a solid color,

simply select the same color for all three color options.

The visibility of the Order Alignment and Tap Tools are controlled by settings in the Alignment Options section of the Designer Tab. To get to this section click Preferences in the Edit menu and choose the Designer Tab.

Also in the Alignment Options, the Tap Tools have a 'speed key' assigned to them. This speed key is really a predetermined amount of distance that the component is allowed to move in one keystroke. The "Tap Distance" is initially set at 0.01 inches. If this distance is increased it will take less time to move the component in the desired direction. To use this feature, select the component and hold the Ctrl key while clicking on any of the arrow keys in the toolbar. Or, use the arrow keys on the keyboard while pressing the ctrl-key.

## Environment Tab

---

"Environment" contains settings for user viewed information, toolbars, and page environment options.

In "User Level", the user's level can be increased or decreased. This setting controls the amount of information to be displayed to the user, mainly dealing with the various properties that will be covered later in this manual. Do not be distracted with all the advanced properties at first. To simplify the early Rave experience, start out at either the Beginner or Intermediate User Level setting. While the Beginner and Intermediate settings are adequate for designing a Rave report, these lower settings will be missing some controls. Thus, if there are controls that are covered and it appears that Rave does not have these controls, check the User Level preference setting.

The "Options" in Environment section, controls the Band Headers, environment settings, and rulers. The "Always Show Band Headers" will have the Band Headers shown at all times. "Save Environment Only Changes When Exiting" will save all the changes made to the environment when the program is shut down. "Display Rulers" will keep the rulers visible around the Page if selected.

"Run in administrator mode," allows the user to be prompted for a password if there is a password in the file. When this selection is not selected, and there is a password in the file, the user will not be prompted for a password, the program will just run in non-administrator mode. An administrator can place a password in different areas to restrict access. Placing a password in the AdminPassword property restricts user access.

"Language" changes the language of the user interface. To have this option, the language file translators needs to be installed. Once installed simply choose a language.

## Shortcuts Tab

---

The "Shortcuts" section allows the user to define keyboard shortcuts for a wide variety of Project actions.

It may save you some time to check the list of "Actions" to see if a shortcut already exists for those tasks that are repeatedly done while you are using the Rave designer. If there is no shortcut, then you may be able to create a shortcut that will be easy for you to remember.

# Appendix A

# Formatting

Below is a list of different format codes and what they will accomplish for each output type.

## AlphaNumeric Items

**Description:** DisplayFormat formats the value given using the format string. The following format specifiers are supported in the format string:

<b>Examples:</b>	<b>Format String</b>	123456.78	-123.0	0.5	0.0
	<b>#,##0.00</b>	123,456.78	-123.00	0.50	0.00
	<b>##</b>	123456.8	-123	.5	0
	<b>\$.0.00</b>	\$123,456.78	-\$123.00	\$0.50	\$0.00
	<b>0.00;(0.00);'</b>	123456.78	(123.00)	0.50	----

Specifier	Represents
0	Digit place holder. If value being formatted has a digit where the '0' appears, then the digit is copied to the output string. Otherwise, a '0' is in the output string.
#	Digit place holder. If value being formatted has a digit where the '#' appears, then the digit is copied to the output string. Otherwise, nothing appears in that position.
.	Decimal point. The first '.' character in the format string determines the location of the decimal separator in the formatted value. The actual character used as the decimal separator in the output string is determined by the Number Format of the International section in the Windows Control Panel.
,	Thousand separator. If the format string contains a ',' characters, the output will have thousand separators inserted between each group of three digits to the left of the decimal point. The actual character used as the thousand separator in the output is determined by the Number Format of the International section in the Windows Control Panel.
E+	Scientific notation. If any of the strings 'E+', 'E-', 'e+', or 'e-' are contained in the format string, the number is formatted using scientific notation. A group of up to four '0' characters can immediately follow the 'E+', 'E-', 'e+', or 'e-' to determine the minimum number of digits in the exponent. The 'E+' and 'e+' formats cause a plus sign to be output for positive exponents and a minus sign to be output for negative exponents. The 'E-' and 'e-' formats output a sign character only for negative exponents.
'xx'/'"xx"	Characters enclosed in single or double quotes are output as-is, and do not affect formatting.
;	Separates sections for positive, negative, and zero numbers in the format string.

The locations of the leftmost '0' before the decimal point in the format string and the rightmost '0' after the decimal point in the format string determine the range of digits that are always present in the output string.

The number being formatted is always rounded to as many decimal places as there are digit placeholders ('0' or '#') to the right of the decimal point. If the format string contains no decimal point, the value being formatted is rounded to the nearest whole number.

If the number being formatted has more digits to the left of the decimal separator than there are



digit placeholders to the left of the '.' character in the format string, the extra digits are output before the first digit placeholder.

To allow different formats for positive, negative, and zero values, the format string can contain between one and three sections separated by semicolons.

One section: The format string applies to all values.

Two sections: The first section applies to positive values and zeros, and the second section applies to negative values.

Three sections: The first section applies to positive values, the second applies to negative values, and the third applies to zeros.

If the section for negative values or the section for zero values is empty, that is if there is nothing between the semicolons that delimit the section, the section for positive values is used instead.

If the section for positive values is empty, or if the entire format string is empty, the value is formatted using general floating-point formatting with 15 significant digits.

## Date / Time items

---

Items that are either a date or time field can use the following format codes. The format specifiers are not case sensitive. If the format parameter is blank then the value is formatted as if a 'c' specifier had been given. The following format specifiers are supported:

Examples: dddd, mmmm d, yyyy => Monday, September 21 1998  
           d mmm yy       => 21 Sep 98

Specifier	Displays
c	Displays date using format given by ShortDateFormat global variable, followed by time using format given by LongTimeFormat global variable. The time is not displayed if fractional part of the DateTime value is zero.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat) using the strings given by the ShortDayNames global variable.
dddd	Displays the day as a full name (Sunday-Saturday) using the strings given by the LongDayNames global variable.
dddddd	Displays the date using the format given by the ShortDateFormat global variable.
dddddd	Displays the date using the format given by the LongDateFormat global variable.
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec) using the strings given by the ShortMonthNames global variable.
mmmm	Displays the month as a full name (January-December) using the strings given by the LongMonthNames global variable.
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).

s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
t	Displays the time using the format given by the ShortTimeFormat global variable.
tt	Displays the time using the format given by the LongTimeFormat global variable.
am/pm	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
a/p	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
ampm	Uses the 12-hour clock for the preceding h or hh specifier, and displays the contents of the TimeAMString global variable for any hour before noon, and the contents of the TimePMString global variable for any hour after noon.
"/"	Displays the date separator character given by the DateSeparator global variable.
:	Displays the time separator character given by the TimeSeparator global variable.
'xx'/'xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.



## Appendix B

# Keyboard / Mouse Shortcuts

Below is a list of different Keyboard / Mouse combinations that can be used as a shortcut. See Preferences - Shortcuts for assigning keyboard keys to your own shortcuts.

### Page Designer or Project Tree

---

Click	on a component selects that component
Right Click	shows context menu for that component
Shift Alt Click	adds all components of the same type as the component clicked on to the selection list of the current page designer
Shift Ctrl Click	adds all children of clicked component to selection list
Shift Click	on a component toggles the selection for that component. This can be used to select multiple components.

### Page Designer Only

---

Click	in blank area of Page Designer removes selection of all components
Ctrl + Arrow Keys	taps (moves) selected components in direction of arrow key
Ctrl C / Ctrl Ins	copies selection to clipboard
Ctrl Click	centers the design window to location clicked
Ctrl F4	unloads current global page
Ctrl V / Shift Ins	paste clipboard to page designer
Ctrl X / Shift Del	cuts selection to clipboards
Delete	deletes currently selected component(s)
Escape	changes selection to parent of current component
F9	executes the current report
F11	toggles between page designer and property panel
Shift + Arrow Keys	changes size of selected components (Up = decrement height, Down = increment height, Left = decrement width, Right = increment width)

### Project Tree Only

---

Alt Drag	DataField component to page designer - creates text component
Alt Drag	selected component to container component in Project Tree - makes the destination component (must be a container component like sections or regions) the parent of all selected components
Ctrl Drag	DataField component to page designer - creates DataText component
Ctrl Drag	component to page designer - creates a mirror of component
Double Click	on Global Page node - loads selected page into page designer
Double Click	on Report node - activates selected report



## Appendix C

# Property Descriptions

Listed below is an alphabetical listing of all properties that make up the RAVE system. Properties are defined by their data type, category, components they are members of, a short description and any relationships they have with other properties. The default values are added where applicable.

## AllowSplit Property

---

### Default

False

### Component/Class

Band, DataBand

### Description

When set to True, the band will be allowed to split across pages.

#### Note:

If the band is too large to fit on a page, it will be split no matter the setting of this property.

## AlwaysGenerate Property

---

### Default

False

### Component/Class

Report

### Description

When set to True, generation of the complete report is forced before sending it to the output device. This is important for insuring that the report variables like TotalPages are known before the first page is printed.

## Anchor Property

---

### Default

Top / Left

### Component/Class

All visible components

### Description

Determines the method that will be used to align a component both vertically / horizontally within its parent's area.

### See also

ExpandParent

## **AutoSize** Property

---

### Default

True

### Component/Class

All BarCode components

### Description

When set to true, BarCode dimensions will automatically resize to display all encoded text.

### See also

Text

## **BandStyle** Property

---

### Default

'None'

### Component/Class

Band, DataBand

### Description

One of the powerful features of RAVE is the ability to set a band's behavior with the BandStyle property. Click on the ellipse in the BandStyle area, this will open an editor dialog window that allows the set the behavior needed for the selected band.

### See also

ControllerBand, Band Style Editor

## **BarCodeJustify** Property

---

### Default

pjLeft

### Component/Class

All BarCode components

### Description

Determines where the bar code is printed relative to its bounding box.

pjLeft	Print the bar code left justified.
pjCenter	Print the bar code centered.
pjRight	Print the bar code right justified.

### See also

Center, Left, Right

## BarCodeRotation Property

---

### Default

Rot0

### Component/Class

All BarCode components

### Description

Allows the bar code to be rotated to 4 different orientations. The pivot point for rotation is the top left corner of the bar code.

Rot0	no rotation
Rot90	rotate 90 degrees relative to page
Rot180	rotate 180 degrees relative to page
Rot270	rotate 270 degrees relative to page

### See also

Left, Top

## BarHeight Property

---

### Default

0.5 ( PostNet component 0.125 )

### Component/Class

All BarCode components

### Description

Sets the height for the tallest bar.

### See also

BarWidth

## BarTop Property

---

### Default

0

### Component/Class

All BarCode components

### Description

Sets the location of the top of the bar code. The location of the readable text is controlled by PrintReadable and PrintTop properties.

### See also

PrintReadable, PrintTop, Top



## **Bin** Property

---

### Default

Default ( Windows system settings )

### Component/Class

Page

### Description

Specifies the paper tray used for the document.

### See also

Collate, Duplex, Orientation, PaperSize, Printer, Resolution

## **BorderColor** Property

---

### Default

Black

### Component/Class

Circle, Ellipse, Rectangle, Square

### Description

Sets the color to be used for the border of the graphic.

### See also

FillColor

## **BorderStyle** Property

---

### Default

psSolid

### Component/Class

Circle, Ellipse, Rectangle, Square

### Description

Sets style of border that appears as a frame around shapes.

psClear	No border is drawn
psDash	Creates a dashed border
psDashDot	Creates an alternating dash and dot border
psDashDotDot	Creates an alternating dash and double dot border pattern
psDot	Creates a dotted border
psInsideFrame	Creates a border that is inside the frame of closed shapes
psSolid	Creates a solid border

### Note:

Only psSolid can have a pen width greater than 1.

### See also

BorderColor, BorderWidth

## Bottom Property

---

### Default

'None'

### Component/Class

All BarCode components

### Description

Sets the position for the bottom of the bar code. The value includes the readable text if it is printed.

### See also

PrintReadable, PrintTop

## CalcType Property

---

### Default

ctSum

### Component/Class

CalcText, CalcTotal

### Description

Sets the type of calculation to be performed by the CalcText over the DataField property contents.

ctAverage	average value for a data field
ctCount	count the number of occurrences
ctMax	maximum value for a data field
ctMin	minimum value for a data field
ctSum	sum of a field

### See also

CountBlanks, RunningTotal

## CalcVar Property

---

### Default

'' ( empty )

### Component/Class

CalcTotal

### Description

Defines the calculation component that will be used in the CalcType operation. If this property is defined, DataField will be ignored.

### See also

CalcType

## Categories Property

---

### Default

'' ( empty )

### Component/Class

ProjectManager

### Description

Defines the available categories to which a report can belong.  
For example, could define categories called 'Accounting', 'General', 'Status' and 'System'.

### See also

Cursor, DevLocked, Parameters

## Category Property

---

### Default

'' ( empty )

### Component/Class

Report

### Description

Sets the category that a report will belong to. For example, you could define categories called 'Accounting', 'General', 'Status' and 'System'.

### See also

Cursor, DevLocked, Parameters

## Center Property

---

### Default

'None'

### Component/Class

All BarCode components

### Description

Sets or returns the position for the horizontal center of the bar code. When a value is assigned to Center the BarCodeJustify property is set to pjCenter as well.

### See also

BarCodeJustify, Left, Right

## **CodePage** Property

---

**Default**

cpCodeA

**Component/Class**

l128BarCode

**Description**

Specifies whether Code A, Code B or Code C is being used.

cpCodeA	sets 128 output to Code A
cpCodeB	sets 128 output to Code B
cpCodeC	sets 128 output to Code C

## **Collate** Property

---

**Default**

False

**Component/Class**

Report

**Description**

Sets the collation style of the print job for the report.

**See also**

Bin, Duplex, Orientation, PaperSize, Printer, Resolution

## **Color** Property

---

**Default**

Black

**Component/Class**

Line and text components

**Description**

Used to set the color of the component's output.

**See also**

BorderColor, FillColor

## Columns Property

---

### Default

1

### Component/Class

DataBand, Region

### Description

Determines the number of columns to be used, this property is component dependent.

DataBand

Defines how many columns the band will print. The width of each column is divided evenly across the width of the band and printing progresses from the left to right column before progressing to the next row.

Region

Sets the number of columns to be use in the region. When the region is printing, it will print bands contents in all columns from left to right in a snaking fashion.

### See also

Columnspacing

## ColumnSpacing Property

---

### Default

0

### Component/Class

DataBand, Region

### Description

Defines the width of a buffer between each column. The spacing will not be applied before the first or after the last column.

### See also

Columns

## ConnectionName Property

---

### Default

'' ( empty )

### Component/Class

DataView

### Description

Sets the name of the data connection from which data will be retrieved for the report.

## **ContainsRTF** Property

---

### Default

False

### Component/Class

DataMemo

### Description

Indicates whether the memo contains RTF or not.

If ContainsRTF is true, then the output will be formatted according to the RTF codes contained in the memo.

## **Controller** Property

---

### Default

' ' ( empty )

### Component/Class

CalcText, CalcTotal

### Description

Defines the controller component that will execute the calculation in the CalcType property when the controller is printed. Normally, the controller is tied to a particular dataview and will signal when new data is available.

### See also

CalcType, ControllerBand, DataField, DataView, Initializer

## **ControllerBand** Property

---

### Default

' ' ( empty )

### Component/Class

Band, DataBand

### Description

Sets the controlling band for the current Band or DataBand.

For example, a header or footer band style would assign the data of the DataBand that it is acting as a header or footer for. Detail bands in a master-detail style report would set their ControllerBand to the master DataBand.

### See also

BandStyle, Controller

## Copies Property

---

**Default**

1

**Component/Class**

Report

**Description**

Sets the number of copies that the report will generate when it is printed.

**See also**

Bin, Collate, Duplex, Orientation, PaperSize, Printer, Resolution

## CountBlanks Property

---

**Default**

True

**Component/Class**

CalcText, CalcTotal

**Description**

This property sets whether the CalcText component will count blank field values for the ctAverage and ctCount calculation types.

**See also**

CalcType

## Cursor Property

---

**Default**

crDefault

**Component/Class**

All drawing

**Description**

Will determine the type of cursor used within that item's selection zone.

crAppStart

crArrow

crCross

crDefault

crDrag

crHandPoint

crHelp

crHourGlass

**See also**

Categories, DevLocked, Parameters

## DataField Property

---

### Default

'' ( empty )

### Component/Class

Most data aware components

### Description

Defines the type of data that the component will display or process. The contents can be made up of data field names, report variables, project parameters or even string constants surrounded by quotes. It is recommended that you use the data text editor to build a DataField value.

### See also

DataView

## DataView Property

---

### Default

'' ( empty )

### Component/Class

Most data aware components

### Description

Defines the data view that will be used if the DataField property is initialized to any field names.

### See also

DataField

## Description Property

---

### Default

'' ( empty )

### Component/Class

DataView, DataField, Page, Report

### Description

Defines a multi-line memo that describes the component. This could be used in conjunction with the FullName property to document how the component is to be used.

### See also

FullName, Name



## DesignerHide Property

---

### Default

False

### Component/Class

Band , DataBand

### Description

This property controls the visibility of the band contents in the designer. Reduce the "clutter" of other bands by setting their DesignerHide property to true. Then only the one(s) set to false will show. This might be needed if there are a large number of bands or a band that is occupying a large space.

#### Note:

This has NO effect on what will be printed, only effects what is shown on the designer page.

### See also

DevLocked

## DestParam Property

---

### Default

' ' ( empty )

### Component/Class

CalcOp, CalcTotal

### Description

value can be blank if the calculation is only going to be used as an intermediate result for other calculation components.

### See also

DataField, Operator

## DestPIVar Property

---

### Default

' ' ( empty )

### Component/Class

Page, ProjectManager, Report

### Description

Initializes the value of a PIVar (Post Initialize Variable). Any PIVars of the same name that were previously printed will show this value. PIVars are normally printed using the Data Text Editor of the DataText component. The main difference between PIVars and parameters is that PIVars will use the value that is set after (in print order) the PIVar while parameters print the value that was set before. A common use for PIVars is to print a total in a header band that would be initialized later in the footer band. This works even across multiple pages. Report.AlwaysGenerate must be true if you are using PIVars in your report.

### See also

AlwaysGenerate, Editor DataText, PIVars

## **DetailKey** Property

---

### Default

'' ( empty )

### Component/Class

DataBand , DataCycle

### Description

Sets the detail key fields that will be matched to the master key fields in a master-detail report. Multiple fields should be separated with +'s.

### See also

MasterKey

## **DevLocked** Property

---

### Default

False

### Component/Class

All components

### Description

Locks out the ability of anyone making changes to the property.

May be used by the developer to prevent accidental changes being made to a stable part of a complex report structure.

### See also

Categories, Cursor, DesignerHide, Parameters

## **DisplayFormat** Property

---

### Default

'' ( empty )

### Component/Class

CalcOp, CalcText, CalcTotal, All Numeric Field Types

### Description

DisplayFormat formats the value given using the given format string. The various format specifiers are listed in appendix A.

### See also

Appendix A

## DisplayOn Property

---

### Default

doParent

### Component/Class

Most components

### Description

Controls whether the component will be used during the print preview display, printer output or both.

doAll	send this item to both preview & printer
doParent	use the parent setting for DisplayOn
doPreviewOnly	item will only be displayed on the preview
doPrinterOnly	component will only be displayed on the printer

## DisplayType Property

---

### Default

dtNumericFormat

### Component/Class

CalcOp, CalcText, CalcTotal

### Description

Defines whether the contents represent numeric or date/time information. The DisplayFormat property will be processed as either numeric or date/time format codes depending upon this setting.

dtDateTimeFormat	sets to Date/Time format
dtNumericFormat	sets to numeric format

## Duplex Property

---

### Default

pdDefault

### Component/Class

Report

### Description

Will set the duplex mode for the current printer.

pdDefault	Use current mode (Duplex mode NOT changed)
pdSimplex	Simplex mode (Duplex mode NOT initialized)
pdHorizontal	Duplex mode initialized - print Head to Toe
pdVertical	Duplex mode initialized - print Head to Head

#### Note:

Not all printers or drivers support duplex printing.

### See also

Bin, Collate, Orientation, PaperSize, Printer, Resolution

## ExpandParent Property

---

### Default

True

### Component/Class

Memo

### Description

When set to true, the height of the parent of the memo will be increased the same amount the height of the memo is increased.

#### Note:

In order to properly print dynamically sized memos; set ExpandParent to true and also set the vertical anchor to stretch.

### See also

Anchor Editor, Anchor Property, DataMemo component

## Extended Property

---

### Default

False

### Component/Class

Code39BarCode

### Description

When set to true, Extended Code 39 format will output instead of standard Code 39 format.

## **FieldName** Property

---

### **Default**

'' ( empty )

### **Component/Class**

All data field components

### **Description**

Defines the field name that this field component will retrieve data from.

## **FileLink** Property

---

### **Default**

'' ( empty )

### **Component/Class**

Bitmap, Metafile

### **Description**

Defines a filename to initialize the bitmap or metafile with a file on the hard disk.

### **See also**

Image

## **FillColor** Property

---

### **Default**

White

### **Component/Class**

Circle, Ellipse, Rectangle, Square

### **Description**

Sets the color that is used to fill the graphical shape

### **See also**

BorderColor, Color, FillStyle

## FillStyle Property

---

### Default

fsSolid

### Component/Class

Circle, Ellipse, Rectangle, Square

### Description

Sets the style used to fill the graphical shape. Use the Fill Toolbar to select the fill style desired or enter it from this properties drop list box.

fsBDiagonal

fsClear

fsCross

fsDiagCross

fsFDiagonal

fsHorizontal

fsNone

fsSolid

fsVertical

### Note:

Setting the FillStyle to fsClear will cause the FillColor to be set to White.

### See also

BorderColor, Color, FillColor

## FinishNewPage Property

---

### Default

False

### Component/Class

Band, DataBand

### Description

Very similar to StartNewPage, but if set to True, it will cause a new page to begin after this band has finished printing.

### Note:

The StartNewPage property would normally be used on header bands while the FinishNewPage property would normally be used on footer bands.

### See also

MaxRows, StartNewPage

## **FirstPage** Property

---

**Default**

'' ( empty )

**Component/Class**

Report

**Description**

Defines the first page that will be printed in a report if the PageList property is empty.

**See also**

GotoMode, GotoPage, PageList

## **Font** Property

---

**Default**

System font

**Component/Class**

All text components

**Description**

Defines the font that will be used to draw the contents of a text component.

**See also**

FontJustify

## **FontJustify** Property

---

**Default**

pjLeft

**Component/Class**

Most text components

**Description**

Sets the horizontal justification of the text data in the box.

pjBlock	block justify the text
pjLeft	left justify the text
pjCenter	center the text
pjRight	right justify the text

## **FontMirror** Property

---

### **Default**

System font

### **Component/Class**

Most text components

### **Description**

This property sets the FontMaster component that is used to define the font for the text. Setting this property to a font master will override the Font property.

### **See also**

Font

## **FullName** Property

---

### **Default**

'' ( empty )

### **Component/Class**

DataField, DataView, Page, Report

### **Description**

Defines the full name or long name of the project item component. The full name is for single line display and may contain special characters and spaces.

### **See also**

Description, Name



## GotoMode Property

---

### Default

gmGotoDone

### Component/Class

Page

### Description

This property determines the behavior of the GotoPage property.

gmGotoDone	Go to the GotoPage property setting when the page definition has completely finished printing. There may be several physical pages if there is a region defined on the page.
gmGotoNotDone	Go to the GotoPage property after a single physical page has printed, only if the current page definition has not finished printing. This option is usually only used with mirrored sections (odd/even layouts, different first page, ...).
gmCallEach	Call GotoPage property after each physical page has printed whether the page definition is finished printing or not. This is useful for inserting other pages after each page of a report prints. Control is returned to the calling page when the page chain ends (i.e., a blank GotoPage property is encountered)

### See also

GotoPage, PageList

## GotoPage Property

---

### Default

'' ( empty )

### Component/Class

Page

### Description

Defines the page that will be printed after the current page according to the GotoMode property rule.

### See also

GotoMode, PageList

## **GridLines** Property

---

**Default**

5

**Component/Class**

Page

**Description**

Controls the grid lines that are visible.

The default setting of 5 means that every fifth grid line will be visible. If GridSpacing is set to 0.1, then it means that the visible grid lines will be every 1/2 inch but snap to grid will occur every 0.1 inches.

**See also**

GridSpacing

## **GridSpacing** Property

---

**Default**

0.1

**Component/Class**

Page

**Description**

Sets the spacing between grid lines.

**See also**

GridLines

## **GroupDataView** Property

---

**Default**

'' ( empty )

**Component/Class**

Band, DataBand

**Description**

Defines the data view that will be used to calculate the GroupKey from.

**See also**

GroupKey

## **GroupKey** Property

---

### Default

'' ( empty )

### Component/Class

Band, DataBand

### Description

Defines the field(s) that will be used to calculate the group key.

When defining a report using grouping headers or footers, the GroupKey property is used to determine when a new group is encountered. Separate fields should be separated with +'s.

### See also

GroupDataView

## **Height** Property

---

### Default

'None'

### Component/Class

All visible components

### Description

Defines the overall height of the component.

For bar codes this is a read only property that contains the height of the entire bar code.

If the bar code PrintReadable property is set to true, then the Height property contains the bar code height plus the line height of the current font.

### See also

BarHeight, PrintReadable

## **HRadius** Property

---

### Default

0

### Component/Class

Rectangle, Square

### Description

Controls the horizontal radius of the rectangle corner. When used in combination with VRadius, these properties round the corners of rectangles or squares.

### See also

VRadius

## Image Property

---

### Default

'' ( empty )

### Component/Class

Bitmap, Metafile

### Description

Defines the image that will be printed with a bitmap or metafile component.

### See also

FileLink

## InitCalcVar Property

---

### Default

'' ( empty )

### Component/Class

CalcController

### Description

When this component is acting as an initializer, it defines the calculation component that will be used as the initializing value.

If this property is defined, InitDataField and InitValue will be ignored.

### See also

InitDataField, InitDataView, InitValue

## InitDataField {CalcController} Property

---

### Default

'' ( empty )

### Component/Class

CalcController

### Description

If the InitCalcVar property is blank, then this property defines the data field or project parameter that will be used as the initial value for any component(s) using this initializer.

If this property is defined, InitValue will be ignored.

### See also

InitCalcVar, InitDataView, InitValue

## InitDataField {PageNumInit} Property

---

### Default

'' ( empty )

### Component/Class

PageNumInit

### Description

Defines the value that the relative page number will start from when this component is printed.

If the text cannot be converted to a valid integer, a default value of 1 will be used.

### See also

InitDataView, InitValue

## InitDataView Property

---

### Default

'' ( empty )

### Component/Class

CalcController, PageNumInit

### Description

Defines the default DataView for the InitDataField property.

### See also

InitDataField

## Initializer Property

---

### Default

'' ( empty )

### Component/Class

CalcText, CalcTotal

### Description

Defines the initializer component. The CalcController is typically the initializer, which will initialize the calculation when the initializer is printed.

Initializers are useful for setting a calculation to a specific value or initializing it at key points in a report

### See also

Controller, CalcController

## **InitToFirst** Property

---

**Default**

True

**Component/Class**

DataBand, DataCycle

**Description**

Moves the DataSet to the Top of File or first record position.

**See also**

ConnectionName

## **InitValue {CalcController}** Property

---

**Default**

0

**Component/Class**

CalcController

**Description**

If InitCalcVar and InitDataField properties are blank, then a constant value is defined that will be used as the initializing value for any components for which this is an initializer.

**See also**

InitCalcVar, InitDataField, InitDataView

## **InitValue {PageNumInit}** Property

---

**Default**

1

**Component/Class**

PageNumInit

**Description**

Defines the value that the relative page number will start from when this component is printed. Only the RelativePage report variable (not CurrentPage) will reflect the new page number.

**See also**

InitDataField, InitDataView

## **KeepBodyTogether** Property

---

### Default

False

### Component/Class

DataBand

### Description

This property, if true, causes the data band to attempt to keep the bands from the body header to the body footer together on the same page.

### See also

KeepRowTogether, OrphanRows, WidowRows

## **KeepRowTogether** Property

---

### Default

False

### Component/Class

DataBand

### Description

This property, if true, causes the data band to attempt to keep the bands from the row header to the row footer together on the same page.

### See also

KeepBodyTogether, OrphanRows

## **Left** Property

---

### Default

'None'

### Component/Class

All visible components

### Description

Sets the position for the left edge of the component.

### See also

Top, Width

## LineStyle Property

---

### Default

psSolid

### Component/Class

All line components

### Description

Sets style of line.

psClear	No line is drawn
psDash	Creates a dashed line
psDashDot	Creates an alternating dash and dot line
psDashDotDot	Creates an alternating dash and double dot line pattern
psDot	Creates a dotted pen
psInsideFrame	Creates a pen that draws a line inside the frame of closed shapes
psSolid	Creates a solid line

### Note:

Only psSolid can have a pen width greater than 1 pixel.

### See also

BorderStyle, BorderWidth, LineWidth

## LineWidth Property

---

### Default

1

### Component/Class

Drawing

### Description

Sets the width of the line. The units for this line thickness is controlled by the LineWidthType property. Line widths greater than 1 pixel can only be used with solid lines.

### See also

LineStyle, LineWidthType



## LineWidthType Property

---

### Default

wtPixels

### Component/Class

Drawing

### Description

Determines whether the LineWidth property of the graphic shape is measure in pixels or points. Use pixels and a LineWidth of 1 (the defaults) if the thinnest line possible is needed. Also use points for all other LineWidth values for consistent thickness across devices.

WtPixels	LineWidth is measured in pixels
WtPoints	LineWidth is measured in points (1 point = 1/72nd of an inch)

### See also

LineWidth

## Locked Property

---

### Default

False

### Component/Class

All components

### Description

Controls the state of the selected component(s) properties and all of its children (if any).

If true, then the properties cannot be selected or changed.

The color of the property names and pips (if selected) will be red when this property is true.

#### Note:

This can be used to a freeze part of a page design, so that the design cannot accidentally be moved while completing a design in another area.

## LookupDataView Property

---

### Default

'' ( empty )

### Component/Class

DataText

### Description

Specifies the data view that lookup will be performed on.

### See also

DataField, LookupDisplay, LookupField, LookupInvalid

## **LookupDisplay** Property

---

### Default

'' ( empty )

### Component/Class

DataText

### Description

Specifies the field in LookupDataView that will actually be displayed in the report after the lookup is performed.

### See also

DataField, LookupDataField, LookupField, LookupInvalid

## **LookupField** Property

---

### Default

'' ( empty )

### Component/Class

DataText

### Description

Specifies the field(s) in LookupDataView that will be matched to the value of the field(s) defined by DataField. Once a match is found, the value of LookupDisplay will be shown in the report.

### See also

DataField, LookupDataField, LookupDisplay, LookupInvalid

## **LookupInvalid** Property

---

### Default

'' ( empty )

### Component/Class

DataText

### Description

Defines the text that will be displayed if no matches are found for the current value of DataField in the LookupDataView.

### See also

DataField, LookupDataField, LookupField, LookupField

## **MailMergeItems** Property

---

### Default

'' ( empty )

### Component/Class

All memo components

### Description

Stores the tokens and replacement data text items that will be used in a search and replace during a mail merge session.

### See also

Mail Merge Editor (page 129)

## **MasterDataView** Property

---

### Default

'' ( empty )

### Component/Class

DataBand, DataCycle

### Description

Defines the DataView that will be used to calculate the MasterKey property from.

### See also

DetailKey, MasterKey

## **MasterKey** Property

---

### Default

'' ( empty )

### Component/Class

DataBand, DataCycle

### Description

Determines the master key field(s) for a master-detail style report. The contents of the MasterKey field(s) will be matched to the DetailKey property to set up the master-detail relationship in the detail databand. Multiple fields should be separated by +'s.

### See also

DetailKey, MasterDataBand

## MatchSide Property

---

### Default

msWidth

### Component/Class

Bitmap, Metafile

### Description

Determines the method that the bitmap or metafile will use to resize itself.

msBoth	The image will size to match both the designed width and height.
msHeight	The image will match the designed height and adjust the width to maintain the correct proportions.
msInside	The image will be drawn proportionally inside the designed area.
msWidth	The image will match the designed width and adjust the height to maintain the correct proportions.

### See also

Height, Width

## MaxRows Property

---

### Default

0

### Component/Class

DataBand, DataCycle

### Description

Defines the maximum number of rows that will be printed for a data band. A value of 0 means print all available rows in the data view.

### See also

StartNewPage

## MinHeightLeft Property

---

### Default

0

### Component/Class

Band

### Description

Defines the minimum height that must remain in the region before this band will print. If the MinHeightLeft value is greater than the remaining height in the region, then band will be printed on the next page.

### See also

Region

## Mirror Property

---

### Default

'' ( empty )

### Component/Class

All components

### Description

Will cause to currently selected component to mirror (duplicate) the properties of the component entered from the list.

### See also

Global Page

## Module Property

---

### Default

None yet

### Component/Class

Page

### Description

This property is not implemented yet.

### See also

Description, FullName

## Name Property

---

### Default

'None'

### Component/Class

All components

### Description

Defines the name of the component as referenced in the application's code. Use the Name property to assign a new name to the control or to find out what the name of the control is. By default, Rave assigns sequential names based on the type of the control, such as 'Rectangle1', 'Rectangle2', and so on. Change these to more meaningful names that make the code more readable. The Name must not contain any spaces or special characters. This is the name that will be used in the Project Tree Panel.

### See also

Description, FullName

## NullText Property

---

### Default

'' ( empty )

### Component/Class

DataField

### Description

Sets the contents that will be printed if the DataField value is blank (empty).

### See also

DataField, TextFalse, TextTrue

## Operator Property

---

### Default

coAdd

### Component/Class

CalcOp

### Description

Defines the type of operation that will be performed on the two source values. The result of this calculation can be placed in a project parameter using the DestParam property or used in other calculations as a CalcVar component.

coAdd	operation set to Source1 + Source2
coAverage	operation set to (Source1 + Source2) / 2.0
coDiv	operation set to Source1 / Source2
coExp	operation set to Source1 raised to the Source2 power
coGreater	operation set to the greater of Source1 and Source2
coLesser	operation set to the lesser of Source1 and Source2
coMod	operation set to Source1 mod Source2
coMul	operation set to Source1 * Source2
coSub	operation set to Source1 - Source2

### See also

DestParam, ResultFunction, Src1Xxxxx, Src2Xxxxx

## Orientation Property

---

### Default

poDefault

### Component/Class

Page

### Description

Sets the orientation for a page to landscape or portrait. The value of poDefault will use the default setting for the printer to determine the orientation.

### See also

Bin, Collate, Duplex, PaperSize, Printer, Resolution

## **OrphanRows** Property

---

**Default**

0

**Component/Class**

DataBand

**Description**

Sets the minimum number of rows that can be by themselves at the bottom of a page. The default value of 0 and allows orphans (i.e. 1 row at the bottom of a page).

**See also**

KeepBodyTogether, KeepRowTogether, WidowRows

## **PageHeight** Property

---

**Default**

11

**Component/Class**

Page

**Description**

Defines the height of the page. Normally this property will be modified via the PaperSize property.

**See also**

PageWidth, PaperSize

## **PageList** Property

---

**Default**

'' ( empty )

**Component/Class**

Report

**Description**

Defines a list of pages to print when then report is executed. If no pages are defined in the PageList then the report will start with what is defined by the FirstPage property.

**See also**

FirstPage , Batch Report example on page 151

## PageWidth Property

---

### Default

'' ( empty )

### Component/Class

Page

### Description

Defines the width of the page. Normally this property will be modified via the PaperSize property.

### See also

PageHeight, PaperSize

## PaperSize Property

---

### Default

"Letter, 8 ½ by 11 inches"

### Component/Class

Page

### Description

Provides a method to select the size of paper being used for a report node. Select one the report nodes, the property panel will show the properties available at the report level. Select the PaperSize and there will be a drop list where paper size can be selected.

#### Note:

All pages within a report must be of the same size. If they are not, the first printed page will determine the size used for all pages within that report.

### See also

Orientation, PageHeight, PageWidth

## Parameters Property

---

### Default

'' ( empty )

### Component/Class

Page, ProjectManager, Report

### Description

Allows parameters to be defined that are available at the project, report and page level. Parameters can be used to print data that is passed from the application (e.g. ReportTitle or UserName) or to store calculation results. Parameters can be printed using the Data Text Editor of the DataText component.

### See also

Editor DataText, DestParam, DestPIVar, PIVars



## PIVars Property

---

### Default

'' ( empty )

### Component/Class

Page, ProjectManager, Report

### Description

Allows PIVars (Post Initialize Variables) to be defined at the project, report and page level. PIVars are initialized using the DestPIVar property of the calculation components. PIVars are normally printed using the Data Text Editor of the DataText component. The main difference between PIVars and parameters is that PIVars will use the value that is set after (in print order) the PIVar while parameters print the value that was set before. A common use for PIVars is to print a total in a header band that would be initialized later in the footer band. This works even across multiple pages. Report.AlwaysGenerate must be true if you are using PIVars in your report.

### See also

AlwaysGenerate, DestPIVar, Parameters, DataText

## PositionMode Property

---

### Default

pmOffset

### Component/Class

Band, DataBand

### Description

Determines how the position of this band will be treated relative to the previously printed band.

pmAbsolute	distances measured from top of region
pmOffset	distances measured from bottom of last band
pmOverlay	does not advance band position

### See also

PositionValue

## PositionValue Property

---

### Default

0

### Component/Class

Band, DataBand

### Description

Sets the starting position of this band based upon the PositionMode property setting. If the setting is pmAbsolute, then this bands starting position is measured from the top of the controlling region for this band. If the setting is pmOffset, then the distance is from the bottom of the last printed band. The pmOverlay setting acts like pmOffset, however, it does not advance the "last" band printed setting.

### See also

PositionMode

## PrintChecksum Property

---

### Default

'None'

### Component/Class

All BarCode components

### Description

Determines if the readable text includes the checksum character.

#### Note:

It is possible that the checksum character may not be a printable character with some of the bar code types.

### See also

BarTop, UseChecksum

## Printer Property

---

### Default

' ' ( empty )

### Component/Class

Report

### Description

Determines the printer that the report will be output to. Normally leave this field blank to print to the default printer.

### See also

Bin, Collate, Duplex, Orientation, PaperSize, Resolution

## **PrintReadable** Property

---

### Default

True

### Component/Class

All bar code components, except UPC

### Description

Set this property to false, to not allow readable text to be printed along with the bar code.

#### **Note:**

For UPC bar codes, text is always printed.

### See also

PrintTop, TextJustify

## **PrintTop** Property

---

### Default

False

### Component/Class

All BarCode components

### Description

Set this property to true to allow the readable text to be printed on top of the bar code. A false value means that the readable text will be printed below the bar code. This property has no effect when printing UPC codes, since the UPC text is always printed at the bottom of the bar code.

### See also

PrintReadable, TextJustify

## **ReprintLocs** Property

---

### Default

(ALL)

### Component/Class

Band, DataBand

### Description

Determines whether the band reprints at the start of a new page. When a band is set to reprint on a new page, the type of band that caused the rollover to the new page will be checked against the band types in the ReprintLocs property. If they band type is found in ReprintLocs, then the band will be reprinted on the new page.

### See also

BandStyle

## **Resolution** Property

---

### **Default**

prDefault

### **Component/Class**

Report

### **Description**

Determines the output resolution of the print job.

prDefault

prDraft

prHigh

prLow

prMedium

### **See also**

Bin, Collate, Duplex, Orientation, PaperSize, Printer

## ResultFunction Property

---

### Default

cfNone

### Component/Class

CalcOp

### Description

Defines the function that will be applied to the result of the calculation after the operation has been performed.

cfAbs	result is the absolute of the value
cfArcTan	result is the ArcTan of the value in radians
cfCos	result is the cosine of the value in radians
cfDec	result is the value - 1
cfFrac	result is the decimal portion of the value
cfHoursToTime	converts the value in hours to DateTime format
cfInc	result is the value + 1
cfMinsToTime	converts the value in minutes to DateTime format
cfNeg	result is the value * -1
cfNone	result is unchanged
cfPercent	result is multiplied by 100
cfRandom	Result is a random value between 0 and the value
cfRound	Result is rounded to the nearest integer
cfRound1	Result is rounded to the 1st decimal place
cfRound2	Result is rounded to the 2nd decimal place
cfRound3	Result is rounded to the 3rd decimal place
cfRound4	Result is rounded to the 4th decimal place
cfRound5	Result is rounded to the 5th decimal place
cfSecsToTime	Converts the value in seconds to datetime format
cfSin	Result is the Sine of the value in radians
cfSqr	Result is the square of the value
cfSqrt	Result is the square root of the value
cfTimeToHours	Converts the value in datetime format to hours
cfTimeToMins	Converts the value in datetime format to minutes
cfTimeToSecs	Converts the value in datetime format to seconds
cfTrunc	Result is the integer portion of the value

### See also

DestParam

## **Right** Property

---

**Default**

'None'

**Component/Class**

All BarCode components

**Description**

Sets or returns the position for the right edge of the bar code. When a value is assigned to Right, the BarCodeJustify property is set to pjRight as well.

**See also**

BarCodeJustify, Center, Left

## **Rotation** Property

---

**Default**

0

**Component/Class**

CalcText, DataText, Text

**Description**

Defines the rotation in degrees of the selected text component. The text is rotated counter clockwise through the values 0 to 359.

**See also**

BarCodeRotation

## **RunningTotal** Property

---

**Default**

False

**Component/Class**

CalcText, CalcTotal

**Description**

Determines if a running total is kept when a NewPage occurs.

False	means NO and the value is reset to 0 each time it is printed
True	means Yes and the value is kept and will continue to total after it is printed

**See also**

CalcType, CountBlanks

## Size Property

---

### Default

'None'

### Component/Class

DataField

### Description

Defines the character width of the data field. This is normally used with design time activities such as dragging and dropping DataText components to determine an approximate width that it will take to print the component.

## SortKey Property

---

### Default

'' ( empty )

### Component/Class

DataBand, DataCycle

### Description

Defines the field(s) that will be passed to the data connection to set up a sort. Separate multiple fields with '+'s.

#### Note:

The underlying database must support the fields being passed as a sort key.

## Src1CalcVar Property

---

### Default

'' ( empty )

### Component/Class

CalcOp

### Description

Defines a calculation component to use as the first source value for the calculation operation. If this property is defined, Src#DataField and Src#Value will be ignored.

### See also

Src#DataField, Src#DataView, Src#Function, Src#Value

## Src1DataField Property

---

### Default

'' ( empty )

### Component/Class

CalcOp

### Description

If Src#CalcVar is blank, defines a data field to use as the first source value for the calculation operation. If this property is defined, Src#Value will be ignored.

### See also

Src#CalcVar, Src#DataView, Src#Function, Src#Value

## Src1DataView Property

---

### Default

'' ( empty )

### Component/Class

CalcOp

### Description

Defines the default dataview that will be used for the Src#DataField property.

### See also

Src#DataField

## Src1Function Property

---

### Default

cfNone

### Component/Class

CalcOp

### Description

Defines the function that will be performed on the source value before the calculation operation is performed.

#### Note:

see ResultFunction (for list of constants).

### See also

Src#CalcVar , Src#DataField , Src#Value

## Src1Value Property

---

### Default

0

### Component/Class

CalcOp

### Description

If Src#CalcVar and Src#DataField are blank, defines a constant value that will be used as the first source value for the calculation operation.

### See also

Src#CalcVar, Src#DataField, Src#Function



## **Src2CalcVar** Property

---

### Default

'' ( empty )

### Component/Class

CalcOp

### Description

Defines a calculation component to use as the first source value for the calculation operation. If this property is defined, Src#DataField and Src#Value will be ignored.

### See also

Src#DataField, Src#DataView, Src#Function, Src#Value

## **Src2DataField** Property

---

### Default

'' ( empty )

### Component/Class

CalcOp

### Description

If Src#CalcVar is blank, defines a data field to use as the first source value for the calculation operation. If this property is defined, Src#Value will be ignored.

### See also

Src#CalcVar, Src#DataView, Src#Function, Src#Value

## **Src2DataView** Property

---

### Default

'' ( empty )

### Component/Class

CalcOp

### Description

Defines the default dataview that will be used for the Src#DataField property.

### See also

Src#DataField

## Src2Function Property

---

**Default**

cfNone

**Component/Class**

CalcOp

**Description**

Defines the function that will be performed on the source value before the calculation operation is performed.

**Note:**

see ResultFunction (for list of constants).

**See also**

Src#CalcVar , Src#DataField , Src#Value

## Src2Value Property

---

**Default**

0

**Component/Class**

CalcOp

**Description**

If Src#CalcVar and Src#DataField are blank, defines a constant value that will be used as the first source value for the calculation operation.

**See also**

Src#CalcVar, Src#DataField, Src#Function

## Tag Property

---

**Default**

nil

**Component/Class**

All components

**Description**

Tag has no predefined meaning to Rave. The Tag property is provided for the convenience of storing additional integer value or pointer information for special needs in an application. For example, use the Tag property when implementing case statements with a component.

## Text Property

---

### Default

'None'

### Component/Class

Text components and all bar code components

### Description

For bar codes, do not include the check character. The check character will be automatically calculated and printed according to the state of the UseChecksum property.

#### Note:

For bar codes, any characters that are invalid for the bar code type will be deleted from the text property upon assignment.

### See also

Font , FontJustify

## TextFalse Property

---

### Default

'' ( empty )

### Component/Class

BooleanField

### Description

Determines what will printed is the field value is False. A blank value for this property will print the text "False".

### See also

TextTrue

## TextJustify Property

---

### Default

pjCenter

### Component/Class

All BarCode components

### Description

Determines how the readable text is justified in relation to the bar code.

pjBock	Block justify the text portion
pjLeft	Left justify the text portion
pjCenter	Center justify the text portion
pjRight	Right justify the text portion

### See also

PrintReadable, PrintTop, Text

## **TextTrue** Property

---

### Default

'' ( empty )

### Component/Class

BooleanField

### Description

Determines what will printed is the field value is True. A blank value for this property will print the text "True".

### See also

TextFalse

## **Top** Property

---

### Default

'None'

### Component/Class

All visible components

### Description

Sets or returns the position for the top edge of the component. For bar codes, the value for this property includes the readable text, if it is printed.

### See also

BarTop, Left, PrintReadable, PrintTop

## **Truncate** Property

---

### Default

False for CalcText and Text, True for DataText

### Component/Class

CalcText, DataText, Text

### Description

When set to true, the text will be truncated to fit the Width property defined in the designer for that component. If the property is false, then all of the text will print without regard to the width property.

### See also

Width

## Units Property

---

### Default

unInch

### Component/Class

Project

### Description

Sets the default units mode for this project. If the setting is unUser then units factor is determined by the value in UnitsFactor.

unCM	Units are in centimeters
unInch	Units are in inches
unMM	Units are in millimeters
unPoint	Units are in points (1/72nd inch)
unUser	Unit are custom defined in UnitsFactor

### See also

UnitsFactor

## UnitsFactor Property

---

### Default

1.0

### Component/Class

Project

### Description

Sets or returns the current conversion factor necessary to convert units to inches. Its value should equal the number of units that equal an inch. (unCM = 2.54 since 2.54 centimeters equal an inch)

### See also

Units

## UseChecksum Property

---

### Default

False (Code128 := true)

### Component/Class

All BarCode components

### Description

Specifies whether a checksum character should be included in the bar code.

### See also

BarHeight, BarWidth, PrintReadable, Text, Width

## **VRadius** Property

---

**Default**

0

**Component/Class**

Rectangle , Square

**Description**

Controls the vertical radius of the rectangle corner. When used in combination with HRadius, these properties round the corners of rectangles or squares.

**See also**

HRadius

## **WasteFit** Property

---

**Default**

True

**Component/Class**

Page, Section

**Description**

If true, components on the page or within the section will dynamically adjust themselves to fit within the printer's waste margins. If a component is not located within the waste area, no adjustment will occur unless it's parent component is located in the waste area.

It is important that components (within a page or section) use the "Anchor" property so that they will adjust as the margins change.

**See also**

Anchor, Anchor Editor, PageHeight, PageWidth, PaperSize

## **WideFactor** Property

---

**Default**

3.0

**Component/Class**

All BarCode components

**Description**

The wide factor is the ratio of the wide bar to the narrow bar width.

**See also**

BarHeight, BarWidth, Width

## WidowRows Property

---

### Default

0

### Component/Class

DataBand

### Description

Sets the minimum number of rows that can be by themselves on the top of the next page. The default setting is 0 and allows widows.

### See also

KeepBodyTogether, KeepRowTogether, OrphanRows

## Width Property

---

### Default

'None'

### Component/Class

All components

### Description

Sets the width of the component. For bar codes, this will return the calculated width of the entire bar code for the current value of Text.

### See also

BarWidth, Height, Left, Text, WideFactor

## WidthType Property

---

### Default

wtPixels

### Component/Class

Drawing components

### Description

Determines how the width of lines is calculated. Generally *wtPixels* and a line width of 1 pixel should be used when the thinnest possible line is desired (typically referred to as hairline). The width type of *wtPoints* should be used on all line widths greater than one pixel to maintain consistency between devices of different DPI resolutions (i.e. Printer and Preview). When using *wtPoints*, line widths are expressed in terms of points (72 points per inch). **See also**

BorderWidth, LineWidth

# INDEX

<b>A</b>	
Adaptable Reports .....	101
Advanced Components .....	93
Alignment Toolbar .....	49
AlphaNumeric Items .....	113
Alt-Drag .....	29
Anchors .....	101
<b>B</b>	
Band .....	82
Bar Code Component Basics.....	91
Bar Code Components.....	20, 91
Batch / Chain Reports .....	106
Batch and Chain Reporting.....	105
Batch Pages .....	105
Bitmap and Meta File .....	62
Brief Bar Code Descriptions .....	91
<b>C</b>	
CalcController .....	97
CalcOp.....	95
CalcText .....	85
CalcTotal.....	97
Calling Pages .....	105
Chain Pages.....	105
Color Palette .....	51
Component vs. Utility Toolbars .....	16
Components Overview.....	19
Connecting to Data .....	75
Creating a Database Connection .....	75
Ctrl- (Control) Drag and Drop.....	29
Cutting and Pasting.....	25
<b>D</b>	
Data Objects .....	88
Data View Dictionary .....	29
DataBand .....	82
Database 101 .....	71
Database Connection.....	88
Database Connections .....	75
DataCycle .....	94
DataMemo .....	84
DataMirror Section .....	95
DataText.....	83
Date / Time items .....	114
Defaults Tab.....	111
Designer Tab.....	111
Designer Toolbar .....	50
Different First Page Format .....	106
Different Odd/Even Page format .....	106
Direct DataViews (BE only) .....	76
Drawing Component Basics .....	65
Drawing Components.....	21, 65
Driver DataViews .....	76
<b>E</b>	
Editor Anchor .....	102
Environment Tab .....	112
Executing Reports .....	39
Executing to the Printer.....	42
Exercise: Aligning Components .....	55
Exercise: Alignment .....	68
Exercise: Calling Pages .....	106
Exercise: Chain Pages .....	107
Exercise: Changing Fonts .....	60
Exercise: Changing Line Size and Color of a Rectangle.....	60
Exercise: Changing Printing Preferences.....	44
Exercise: Changing the Parent of a Component (Alt-Drag) .....	32
Exercise: Creating Drawing Components	66
Exercise: Cutting and Pasting.....	26
Exercise: Different First Page .....	108
Exercise: Different Odd/Even Page.....	108
Exercise: Dragging a Component (Ctrl-Drag) .....	31
Exercise: Loading and Unloading Global Pages .....	30
Exercise: Master Detail Wizard .....	80
Exercise: Naming Components.....	30
Exercise: Navigating the Project Tree.....	29
Exercise: Navigating the Property Panel..	37
Exercise: Ordering Components .....	57
Exercise: Placing and Resizing Bitmaps ..	64
Exercise: Preview and Creating Portable Files.....	45
Exercise: Printing a Report through the Preview .....	47
Exercise: Printing to a File (NDR & PRN)	47
Exercise: Section.....	63
Exercise: Selecting, Sizing, and Moving Components .....	25
Exercise: Setting up PageNumInit for Page Numbering .....	99
Exercise: Simple Wizard .....	79



Exercise: Snapping to the grid .....	59	Project Manager .....	87
Exercise: Text vs Memo .....	62	Project Toolbar .....	53
Exercise: Using Font Master.....	97	Project Tree .....	17
Expanding and Right-Clicking .....	27	Project Tree Only.....	117
<b>F</b>		Project Tree Panel .....	11, 27
First Glance.....	10	Property	
Font Editor .....	52	AllowSplit .....	119
FontMaster .....	93	AlwaysGenerate .....	119
Formatting.....	113	Anchor .....	119
<b>G</b>		AutoSize .....	120
Generating Output .....	39	BandStyle .....	120
Getting Started .....	7	BarCodeJustify .....	120
Getting to the Preferences Dialog .....	111	BarCodeRotation .....	121
Global Page .....	88	BarHeight .....	121
Global Page Catalog .....	28	BarTop.....	121
<b>H</b>		Bin.....	122
Hiding Toolbars .....	16	BorderColor.....	122
HTML.....	43	BorderStyle .....	122
<b>I</b>		Bottom .....	123
Introduction .....	9	CalcType.....	123
<b>K</b>		CalcVar .....	123
Keyboard / Mouse Shortcuts.....	117	Categories .....	124
<b>L</b>		Category .....	124
Limited Warranty .....	7	Center.....	124
Line Editor.....	52	CodePage .....	125
Loading and Unloading Global Pages .....	28	Collate .....	125
<b>M</b>		Color.....	125
Memo .....	61	Columns .....	126
<b>N</b>		ColumnSpacing .....	126
Navigation Area.....	10	ConnectionName .....	126
NDR & PRN Files .....	42	ContainsRTF .....	127
<b>O</b>		Controller .....	127
Overview.....	23, 27, 39, 61, 71, 81, 87, 101	ControllerBand .....	127
<b>P</b>		Copies .....	128
Page .....	88	CountBlanks.....	128
Page Designer.....	23	Cursor.....	128
Page Designer Only.....	117	DataField .....	129
Page Designer or Project Tree.....	117	DataView .....	129
PageNumInit .....	94	Description .....	129
Parent-Child Relationship .....	28	DesignerHide .....	130
PDF .....	43	DestParam .....	130
Pixels vs Points .....	66	DestPIVar .....	130
Preferences .....	111	DetailKey .....	131
Preferences Dialog .....	39	DevLocked .....	131
Project Components .....	87	DisplayFormat .....	131
		DisplayOn .....	132
		DisplayType .....	132

Duplex .....	133	MinHeightLeft .....	149
ExpandParent .....	133	Mirror .....	150
Extended .....	133	Module .....	150
FieldName .....	134	Name .....	150
FileLink .....	134	NullText .....	151
FillColor .....	134	Operator .....	151
FillStyle .....	135	Orientation .....	151
FinishNewPage .....	135	OrphanRows .....	152
FirstPage .....	136	PageHeight .....	152
Font .....	136	PageList .....	152
FontJustify .....	136	PageWidth .....	153
FontMirror .....	137	PaperSize .....	153
FullName .....	137	Parameters .....	153
GotoMode .....	138	PIVars .....	154
GotoPage .....	138	PositionMode .....	154
GridLines .....	139	PositionValue .....	155
GridSpacing .....	139	PrintChecksum .....	155
GroupDataView .....	139	Printer .....	155
GroupKey .....	140	PrintReadable .....	156
Height .....	140	PrintTop .....	156
HRadius .....	140	ReprintLocs .....	156
Image .....	141	Resolution .....	157
InitCalcVar .....	141	ResultFunction .....	158
InitDataField {CalcController} .....	141	Right .....	159
InitDataField {PageNumInit} .....	142	Rotation .....	159
InitDataView .....	142	RunningTotal .....	159
Initializer .....	142	Size .....	160
InitToFirst .....	143	SortKey .....	160
InitValue {CalcController} .....	143	Src1CalcVar .....	160
InitValue {PageNumInit} .....	143	Src1DataField .....	160
KeepBodyTogether .....	144	Src1DataView .....	161
KeepRowTogether .....	144	Src1Function .....	161
Left .....	144	Src1Value .....	161
LineStyle .....	145	Src2CalcVar .....	162
LineWidth .....	145	Src2DataField .....	162
LineWidthType .....	146	Src2DataView .....	162
Locked .....	146	Src2Function .....	163
LookupDataView .....	146	Src2Value .....	163
LookupDisplay .....	147	Tag .....	163
LookupField .....	147	Text .....	164
LookupInvalid .....	147	TextFalse .....	164
MailMergeItems .....	148	TextJustify .....	164
MasterDataView .....	148	TextTrue .....	165
MasterKey .....	148	Top .....	165
MatchSide .....	149	Truncate .....	165
MaxRows .....	149	Units .....	166

UnitsFactor .....	166	<b>U</b>	
UseChecksum .....	166	Using Tools .....	49
VRadius .....	167	Utility Toolbars .....	49
WasteFit .....	167	<b>W</b>	
WideFactor .....	167	Waste Fit.....	102
WidowRows .....	168	What is a Component.....	19
Width .....	168	What is a database? .....	71
WidthType.....	168	What is the Property Panel .....	35
Property Descriptions .....	119	What's All the RAVE About?.....	9
Property Editors .....	37	Wizards.....	79
Property Panel.....	12, 16, 35	<b>Z</b>	
<b>R</b>		Zoom Toolbar .....	54
Region .....	82		
Relational Table.....	73		
Report.....	88		
Report Authoring Visual Environment .....	9		
Report Components.....	21, 81		
Report Library .....	28		
Report Preview .....	40		
Reporting .....	74		
Right-Click Menu .....	37		
RTF .....	43		
<b>S</b>			
Section.....	62		
Security Components.....	89		
Selecting Components .....	24		
Shortcuts Tab .....	112		
Single User License Agreement .....	7		
Sizing and Moving Components.....	25		
SQL Data View .....	89		
Standard Components .....	20, 61		
Status Bar .....	77		
<b>T</b>			
Technical Support .....	8		
Terms .....	72		
Text .....	61		
The Basic Component: The Page & it's Panels .....	23		
The Component Toolbars .....	20		
The Page (Foundation of Rave).....	11		
Tool Windows .....	16		
Toolbar Palette .....	15		
Toolbar Placement .....	15		
Toolbars .....	15		
Toolbars and Tool Windows .....	15		
Tools are Tools.....	49		
Types of Properties .....	37		



# Borland® Delphi™ 7 Studio

## Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL*
<b>New! Interoperability and migration support for Microsoft® .NET</b>				
<b>New!</b> Delphi compiler warnings and hints for .NET compatibility	✓	✓	✓	✓
<b>New!</b> Import any .NET assembly as a COM object	✓	✓	✓	✓
<b>New!</b> Use Delphi COM objects in .NET managed applications	✓	✓	✓	
<b>New! Delphi™ for Microsoft® .NET prerelease preview</b>				
<b>New!</b> .NET preview CIL compiler for the Delphi™ language to create managed applications	✓	✓	✓	✓
<b>New!</b> .NET migration documentation	✓	✓	✓	✓
<b>New! ModelMaker technology</b>				
<b>New!</b> Model-driven design	✓	✓		
<b>New!</b> Visual modeling and refactoring tool based on UML™ technology	✓	✓		
<b>New!</b> Native Delphi integration, reverse engineering, and instant visualization	✓	✓		
<b>New!</b> Apply off-the-shelf design patterns	✓	✓		
<b>New! AToZed Software IntraWeb technology</b>				
<b>New!</b> Deploy your application to the Internet, utilizing drag-and-drop visual Web application development	✓	✓	✓	
<b>New!</b> Add interactive content to your Web sites by rapidly and visually building dynamic server-side HTML Web applications	✓	✓	✓	
<b>New!</b> Transparently manages Web application details such as cookie, session, and user management	✓	✓		
<b>New!</b> Create, debug and maintain Web-based applications quickly with Application Mode	✓	✓		
<b>New!</b> Support for easy-to-use API for implementing custom components	✓	✓		

\* Delphi 7 Personal Edition is not a Studio product, and it is for noncommercial use only.

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
<b>New! Nevrona Rave™ technology</b>				
<b>New!</b> Powerful Rave Reports Borland Edition visual report designer and code-based API	✓	✓	✓	
<b>New!</b> PDF, HTML, RTF, and text rendering formats	✓	✓	✓	
<b>New!</b> Native support for VCL and Borland Component Library for Cross-platform (CLX™) applications	✓	✓	✓	
<b>New!</b> Flexible page-based layout to handle both form and banded style reports	✓	✓	✓	
<b>New! Bold® for Delphi™</b>				
<b>New!</b> Maintain less code with a true Model Driven Architecture™ (MDA™)	✓			
<b>New!</b> UML technology: full support for class diagrams, tagged values, stereotypes, associations (aggregates, compositions), cardinality, visibility, and mixed transient/persistent elements and classes	✓			
<b>New!</b> Integrated support for Rational Rose® and ModelMaker; import/export model information from/to Bold® Model Editor	✓			
<b>New!</b> Automatic database schema generation using SQL	✓			
<b>New! Windows XP™ Themes</b>				
<b>New!</b> Enable applications to take part of Windows XP™ Themes with a look that will blend in with your favorite Windows XP theme	✓	✓	✓	✓
<b>New!</b> Take full advantage of the new XP look for the Windows® common controls library (comctrls.dll version 6.0)	✓	✓	✓	✓
<b>New!</b> ThemeServices wrap around theme APIs to enable an application or control to use theme functions transparently	✓	✓	✓	✓
<b>Borland® BizSnap™—Web Services with XML technologies</b>				
<b>New!</b> UDDI browser to visually locate and import registered Web Services	✓	✓	✓	
<b>New!</b> Automatic UDDI client fail-over support	✓	✓	✓	
<b>New!</b> Global XML Architecture (GXA) for Web Services, header support for Web Services routing, and Web Services Inspection Language (WSIL)	✓	✓	✓	
<b>New!</b> Binary attachments	✓	✓	✓	
<b>New!</b> (for Delphi 7 Studio Professional) Rapidly build server-side W3C-compliant Web Services using SOAP, XML, WSDL, and more	✓	✓	✓	

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Rapidly build client-side applications that consume W3C-compliant Web Services using SOAP, XML, WSDL, and more	✓	✓	✓	
<b>New!</b> (for Delphi 7 Studio Professional) WSDL wizards and components mean you can easily access Internet Web Services and add functionality to your own applications	✓	✓	✓	
<b>New!</b> (for Delphi 7 Studio Professional) Create applications that seamlessly interoperate across the Internet with standards-based Web Services platforms such as Microsoft® .NET, Sun® ONE, and others!	✓	✓	✓	
<b>New!</b> (for Delphi 7 Studio Professional) SOAP communication wizards and components make building Web Services a snap!	✓	✓	✓	
<b>New!</b> (for Delphi 7 Studio Professional Delphi) native SOAP bindings deliver type and syntax checking with CodeInsight™ and more with SOAP Web Services	✓	✓	✓	
XML Transform Components ( XML<-->dataset, XML<-->XML) enable your application to translate XML data between XML and dataset formats easily	✓	✓		
XML Schema transform tools and utilities provide translation between XML data, DTDs, and XDR schema and datasets automatically, making XML Schema analysis, creation, and XML data transformations a snap!	✓	✓		
Native Delphi XML bindings radically simplify XML programming—XML documents become native Delphi objects	✓	✓		
Avoid vendor lock-in and incorporate data from virtually any XML source with multiple XML DOM Level 2 support	✓	✓		
XML doc object in the VCL provides fast programmatic access to new XML formats	✓	✓		
XML Document Objects and native Web Services WSDL bindings leverage CodeInsight and compiler type-checking	✓	✓		

## Borland WebSnap™ —the complete Web application design platform

Deploy your Web applications to Microsoft IIS, Netscape®, and <b>New!</b> Apache™ 2.0! With support for ISAPI, NSAPI, Apache DSO, and CGI	✓	✓		
WebPage Surface Designers to rapidly build and view your WebSnap™ application surfaces in HTML and an instantly updated browser preview of your application	✓	✓		
XML Tree and XSL Tree displays to present any XML and XSL documents with easy-to-read indentations and syntax highlighting	✓	✓		
<b>Enhanced!</b> Server-side scriptable Web page components to easily interface WebSnap/NetCLX™ objects with Web site development teams and scripters	✓	✓		

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Pre-built scriptable Web page components—datasets, login forms, end-user sessions, and other commonly required Web application elements	✓	✓		
Page Producers and Dispatchers for servicing standard and XSL-transformed XML pages, logical pages, and your own custom adapters and services	✓	✓		
Multiple Web modules in a single project to organize your pages and your site	✓	✓		
XSL Page Producers for combining XML source data with XSL transforms to produce full-page outputs to HTML	✓	✓		
<b>Enhanced!</b> Server-side scripting in WebSnap with JavaScript® and VBScript	✓	✓		
Active script debugging support in WebSnap	✓	✓		

### Borland InternetExpress™

XML Web browser tables for high-performance client-cached XML Web database tables	✓	✓		
XML data from DataSnap™ servers to simplify data exchange	✓	✓		
XML broker for quickly providing XML data for Web server applications	✓	✓		
DataSnap PageProducer wizard to migrate your applications to the Internet with XML and HTML 4 for dynamic Web clients	✓	✓		
Web Page Editor for instantly designing HTML 4 Web documents that deliver dynamic XML data	✓	✓		

### Borland NetCLX™ and WebBroker™ —compatible, cross-platform Web application development framework for Windows® and Linux® \*

Develop single-source applications for deployment to Web servers on Windows and Linux® *	✓	✓	✓	
Automated Web Debug Server accelerates your Web application testing cycles	✓	✓	✓	
NetCLX Web application framework for high-speed, high-throughput, cross-platform Web applications compatible with WebBroker™	✓	✓	✓	
NetCLX Web application components for cross-platform development on Apache, IIS, Netscape, and more, using ISAPI, NSAPI, Apache DLL and SO extensions, or CGI	✓	✓	✓	
WebModules™ for centralized information publishing to Internet applications	✓	✓	✓	
Web application wizard gets you up and running quickly	✓	✓	✓	

\* Compiling applications for Linux® requires Kylix.™



## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
WebDispatcher™ component for issuing rapid responses to browser requests and adapting any data module to the Web	✓	✓	✓	
<b>New!</b> Support for Apache 2.0 Web Server in Delphi Web technologies	✓	✓	✓	
<b>Seamless Internet/intranet wizards and components</b>				
WebBridge for an open solution that supports ISAPI, NSAPI, Apache DLL, and CGI	✓	✓	✓	
Page preview with HTML 4 support	✓	✓	✓	
Web deployment wizard for deploying thin-client, zero-configuration applications using the Web	✓	✓	✓	
Web application wizards make getting started on Web development a breeze	✓	✓	✓	
<b>Exclusive!:</b> ActiveForms™ for building rich GUI forms for Win32® client applications and browsers	✓	✓	✓	
Internet socket components to customize your own networked applications	✓	✓	✓	
Borland® Database Engine (BDE) CAB file for easy distribution of database applications over the Web	✓	✓	✓	
JPEG support	✓	✓	✓	
Active Server Object wizard for high-performance ASP server development	✓	✓	✓	
<b>Borland DataSnap™ Web Services-enabled database middleware</b>				
<b>New!</b> Royalty-free distribution with unlimited licenses	✓	✓		
Web client, GUI client, and Web Services access to any supported RDBMS	✓	✓		
SOAP/XML, COM, Web, and TCP/IP access connections available for maximum network connectivity and flexibility	✓	✓		
Easily build XML/SOAP Web Services interfaces to any enterprise-class database—Oracle®, Microsoft SQL Server,™ IBM® DB2,® Borland InterBase,® and more!	✓	✓		
Dataset-based architecture for rapid learning curve—use existing skill sets to scale your applications	✓	✓		
DataSnap support for dbGo,™ BDE, InterBase Express™ (IBX™), and dbExpress data access architectures	✓	✓		
High availability with object broker fail-over safety to guarantee your data is ready when you need it	✓	✓		
Load balancing to promote the highest performance even when under the heaviest loads	✓	✓		

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Distributed data with transaction processing extends the reach of your applications while maintaining data integrity	✓	✓		
Automatic database constraint propagation brings the business rules to the client applications for local processing, conserving server horsepower	✓	✓		
Low-maintenance, thin, and easily-configured client-side applications to reduce deployment costs	✓	✓		
High-speed database connectivity yields higher performance in your applications	✓	✓		
Server object pooling maximizes the conservation of resources on your servers, keeping material costs down	✓	✓		
Supply data to thin-client applications rapidly, efficiently, and securely	✓	✓		
Remote DataBroker to more easily partition applications	✓	✓		
TransactionResolver for transaction conflict resolution	✓	✓		
<b>Exclusive!</b> Advanced support for Master/Detail Provider and Resolver	✓	✓		
BDE Resource Dispenser for Microsoft Transaction Server	✓	✓		
Stateless DataBroker for more control in mobile and low-bandwidth situations	✓	✓		
Server object pooling for complete scalability	✓	✓		
Provider options increase control over how and what information is transmitted	✓	✓		
<b>Support for CORBA®</b>				
<b>New!</b> Borland® Enterprise Server, VisiBroker® Edition 4.5 for Delphi 7 Studio including CORBA® client and server development	✓	✓		
Wizards to simplify development of CORBA clients and servers	✓	✓		
Borland® Enterprise Server, AppServer™ Edition support for SIDL—build new rich applications and Web Services with the Delphi RAD environment that uses EJB™ for AppServer Edition	✓	✓		
Support for simultaneous COM and CORBA objects	✓	✓		
<b>High-performance native Windows® applications</b>				
High-performance 32-bit optimizing native-code compiler	✓	✓	✓	✓
Easily create reusable dynamically linked libraries (.DLL), COM controls (.OCX), and stand-alone executables	✓	✓	✓	✓
Use and create dynamically loaded packages for maximum application efficiency	✓	✓	✓	✓

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Inline assembler with support for full Intel® 32-bit x86 instruction set (including Pentium® Pro, Pentium III, Pentium 4, MMX®, SIMD, and Streaming SIMD Extensions (SSE and SSE2), and AMD® 3DNow!™)	✓	✓	✓	✓
Expression evaluation in compiler directives	✓	✓	✓	✓
Custom variants to support your own data types	✓	✓	✓	✓
Custom variant data handler for passing variants through data sets	✓	✓	✓	✓
Custom variant in safe arrays	✓	✓	✓	✓
Expanded variant support for Int64 and the unsigned types	✓	✓	✓	✓
Complex number support via custom variants	✓	✓	✓	✓
Deprecated Hint Directive to help improve your application development	✓	✓	✓	✓
Platform and Library Hint Directives to assist in cross-platform development	✓	✓	✓	✓

## Borland® Rapid Application Development (RAD) IDE

State-of-the-art IDE with a fully integrated Code Editor and debugger, view history, easy-to-understand error messages, and color syntax highlighting	✓	✓	✓	✓
<b>New!</b> Code completion for DTD described markup language including HTML, WML, XHTML, XSL	✓	✓	✓	
<b>New!</b> Message helper that hooks error messages to documentation	✓	✓	✓	✓
<b>New!</b> Toggle visible symbols in editor that represent use of whitespace as tabs or spaces	✓	✓	✓	✓
<b>New!</b> Syntax highlighting OpenTools API extension with built in support for JavaScript, PHP, and INI files	✓	✓	✓	
<b>Enhanced!</b> Ability to individually customize editor options for particular file types	✓	✓	✓	✓
<b>New!</b> OpenTools API for code completion and symbol table queries	✓	✓	✓	
<b>Enhanced!</b> Component palette layout for quicker navigation	✓	✓	✓	✓
Fully customizable IDE layout with window docking and floating toolbars	✓	✓	✓	✓
Multiple custom Desktop Layout Profiles for total control of your development environment	✓	✓	✓	✓
Object TreeView for quick navigation and manipulation of the objects in the current designer	✓	✓	✓	✓

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Object Inspector™ Subcomponent References to save time locating linked components—directly drill-down into subcomponents	✓	✓	✓	✓
Class Completion accelerates creating new object types	✓	✓	✓	✓
Packages in project manager	✓	✓	✓	✓
RTL package for non-visual development	✓	✓	✓	✓
Context-sensitive component palette displays only components that can be placed on the currently active forms or data modules	✓	✓	✓	✓
Environment Variables in search path to enable automatic location of required files	✓	✓	✓	✓
Advanced multi-target Project Manager	✓	✓	✓	✓
Data modules for centralized data integrity and business rules	✓	✓	✓	✓
Console Application Wizard for quickly writing text-mode applications	✓	✓	✓	✓
RC Files integrated with the Project Manager and compiler	✓	✓	✓	✓
Both VCL and CLX™ Visual Form Inheritance and Form Linking	✓	✓	✓	✓
Editor Key mapping to fine-tune the editor to the way you work	✓	✓	✓	
Project-wide browsing to better understand your code and to navigate the VCL	✓	✓	✓	
DataModule™ designer with Tree View and Data Diagram View to gain a comprehensive understanding of the data in your application	✓	✓	✓	
Control Panel Wizard makes building control applets fast and simple	✓	✓	✓	
To-do lists to keep your development on schedule	✓	✓	✓	
OpenTools API to integrate your favorite tools	✓	✓	✓	
AppBrowser™ code editor with symbol hyperlinks and navigation history	✓	✓	✓	
<b>Maximum reusability with Delphi™ object-oriented architecture</b>				
Object-oriented, fully extensible component and application architecture in both VCL and CLX	✓	✓	✓	✓
Support for Windows common controls	✓	✓	✓	✓
VCL and CLX Visual Form Inheritance and Form Linking to reduce coding and simplify maintenance	✓	✓	✓	✓
Object Repository for storing and reusing forms, DataModules, and experts	✓	✓	✓	✓
Visual Component Library of drag-and-drop reusable components	✓	✓	✓	✓

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Component Library for Cross-platform of drag-and-drop reusable components for Windows and Linux*	✓	✓	✓	
Code Explorer for an instantly updated class reference for your application's unit files	✓	✓	✓	
<b>CodeInsight™ —speed coding and reduce syntax errors</b>				
Code Templates Wizard for simplified code creation	✓	✓	✓	✓
<b>Enhanced!</b> Code Templates with the ability to create, modify, and export templates for any file type	✓	✓	✓	✓
Code Completion Wizard to help ensure proper syntax	✓	✓	✓	✓
Code Parameter Wizard for displaying procedure, method, and event parameter lists	✓	✓	✓	✓
ToolTip Expression Evaluation for easier debugging	✓	✓	✓	
DLL Debugging saves time in creating and debugging DLLs	✓	✓	✓	✓
Class navigation short cuts	✓	✓	✓	✓
Class completion	✓	✓	✓	
ToolTip Symbol Insight	✓	✓	✓	
OpenHelp for easily customizing the contents of help	✓	✓	✓	
<b>Advanced debugging</b>				
<b>New!</b> Multi-tab watch view for logically grouping variables	✓	✓	✓	✓
<b>New!</b> Thread naming for easier debugging of multi-threaded applications to identify threads by user defined name instead of by thread ID	✓	✓	✓	✓
<b>Enhanced!</b> Watch view supporting inplace editing, checkbox enable/disabling, and the watch names separated from the values with columns	✓	✓	✓	✓
Full-featured debugger with color syntax highlighting	✓	✓	✓	✓
ToolTip Expression evaluation	✓	✓	✓	✓
<b>New!</b> Added support for multi-target debugging. Working directly makes DLL debugging easier, providing more complete debugging control of your application extensions	✓	✓	✓	✓
Multiprocess, cross-process, and attach-to-process debugging for easy tracing through complex applications	✓	✓	✓	✓
Temporary Process Options for focused multiprocess debugging	✓	✓	✓	✓
Module view	✓	✓	✓	✓
Streamlined usability and integration among debug windows and the editor including clipboard and drag-and-drop support	✓	✓	✓	✓

\* Compiling applications for Linux® requires Kylix.™

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Run until return	✓	✓	✓	✓
Advanced Breakpoints with DataAware triggers, ToolTips, actions, and groupings for faster and complete debugging control	✓	✓	✓	✓
CPU View for low-level debugging	✓	✓	✓	✓
FPU view with support for MMX for enhanced low-level debugging	✓	✓	✓	
Debug spawned processes	✓	✓	✓	
Attach to and debug running processes	✓	✓	✓	
WinSight® 32 for monitoring Windows messaging	✓	✓	✓	
<b>Exclusive!</b> Debug Inspector™ for monitoring component properties while debugging	✓	✓	✓	
Inspector view with fully qualified names	✓	✓	✓	
Event log for showing process control messages and debug output with enhanced log coloring for different event types	✓	✓	✓	
Local Variables View	✓	✓	✓	
Support for Multiple Evaluators for Borland C++Builder™ interoperability	✓	✓	✓	
Easier debugging with DCUs and debug symbol search paths	✓	✓	✓	
Remote Process debugging for distributed development	✓	✓		

### Delphi™ TeamSource for scaling RAD to your entire development team

Maintain development productivity while protecting your source code	✓	✓		
Easily manage source code change in large distributed development teams	✓	✓		
Automatic reconciliation greatly simplifies the task of synchronizing developer changes with the master archive	✓	✓		
Be in command of changes posted to the master archive with comprehensive history tracking and publishing	✓	✓		
Simplify the management of the milestones in your project with Source Code Bookmarks	✓	✓		
Support for PVCS® and other versioning engines with modular version controllers	✓	✓		

### Simplifying international development

Comprehensive support for bi-directional text (left to right, right to left)	✓	✓	✓	✓
Multibyte-enabled development environment	✓	✓	✓	✓
Multibyte-enabled VCL	✓	✓	✓	✓

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Dynamic Language switching at runtime	✓	✓	✓	✓
Resource DLL Wizard to kick-start the translation of your application	✓	✓		
Delphi Form (DFM) editing without source code	✓	✓		
Translation Repository to facilitate the reuse of translated items	✓	✓		
RC Translator to seamlessly integrate external components into the translation	✓	✓		
DFM Translator to seamlessly integrate Delphi form translation	✓	✓		
Translation and Localization Manager to simplify localization of your applications	✓	✓		

## RADical visual development

### Visual Component Library (VCL)

VCL native components for Rapid Application Development	300+	300+	225+	85+
<b>New!</b> Support for Windows XP Theme to modernize the look of your applications	✓	✓	✓	✓
Windows® 2000-style Shell Control examples to modernize your interface	✓	✓	✓	✓
Component Interface Reference architecture to surface Component Interfaces in the Object Inspector	✓	✓	✓	✓
TdateTime extensions accelerate your date and time calculations	✓	✓	✓	✓
SubComponent classes combine commonly mixed components for greater utility, and they speed the creation of user interfaces	✓	✓	✓	✓
Conversion unit simplifies measurement conversions	✓	✓	✓	✓
Visual Component creation	✓	✓	✓	✓
Package Manager to easily control the contents of custom component packages	✓	✓	✓	✓
Modern UI features: docking, Intellimouse® support, minimum and maximum component sizing constraints, and border anchors in visual components	✓	✓	✓	✓
Action List for managing and directing commonly used code in an application	✓	✓	✓	✓
Windows 98/Me/2000 support for Multi-Monitor, MonthCalendar, PageScroller, and Flat Scrollbars	✓	✓	✓	✓
Owner Draw support in the Property Editors for simplifying display of property choices	✓	✓	✓	✓
Support for Advanced Custom Draw functions for increased control over the Windows API	✓	✓	✓	✓

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
Action MainMenu and Action ToolBar components for sophisticated and centrally controlled User Interface design with dynamic Action Manager-driven UI surfaces	✓	✓	✓	
Action Manager provides central management of your actions and self-configuring dynamic User Interfaces, taking the tedium out of UI design	✓	✓	✓	
ActionBands™ customization manager gives your end-users total control over ActionBands UI configurations at runtime!	✓	✓	✓	
Windows NT® service applications	✓	✓	✓	
Package Collection Editor	✓	✓	✓	
ControlBar for creating floating toolbars in GUI applications	✓	✓	✓	
Frames for building and reusing compound components	✓	✓	✓	
Web browser component for integrating HTML browsing into your application	✓	✓	✓	
Set of Microsoft Office automation components to quickly integrate your applications with Office applications such as Word, Excel, and Outlook®	✓	✓	✓	

### CLX™ (Component Library for Cross-platform) development on Windows®/Linux®\*

More than 160 CLX native components for Rapid Application Development on Windows and Linux*	✓	✓	✓	
BaseCLX™ RTL classes and components	✓	✓	✓	
VisualCLX™ native GUI components and data-aware visual controls	✓	✓	✓	
VisualCLX GUI display components—Form, EditBox, Label, Grids, Image, Bevels, Listbox, ComboBox, ListView, TreeView, ProgressBar, and more	✓	✓	✓	
VisualCLX GUI DataAware components—Navigator, EditBox, Label, Grids, Image, Listboxes, Checkbox, RadioGroup, LookupLists, and more	✓	✓	✓	
VisualCLX GUI input components—Menus, PopupMenus, Buttons, Checkboxes, RadioButtons, SpeedButtons, SystemTimer, Sliders, and more	✓	✓	✓	
VisualCLX GUI control components—Toolbars, Scrollbars, Controlbars, Splitters, GroupBoxes, TabControls, PageControls, StatusBars, and more	✓	✓	✓	
Common CLX GUI dialogs—Open, Save, Font, ColorPicker, Find, and Replace	✓	✓	✓	
DataCLX™ highly scalable data-access components	✓	✓	✓	

\* Compiling applications for Linux® requires Kylix.™



## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
<b>Borland® Kylix™ 3 IDE for the Delphi™ language</b>				
New! Borland Kylix™ 3 Enterprise IDE for the Delphi language	✓	✓		
New! Borland Kylix 3 Professional IDE for the Delphi language			✓	
<b>Borland source code</b>				
VCL source code	✓	✓	✓	
CLX source code	✓	✓	✓	
Property Editor source code	✓	✓	✓	
<b>Database architecture for connecting to corporate data</b>				
<b>Highly scalable DataCLX™ database development architecture</b>				
The Delphi DataSet Abstraction layer makes database programming easy, from local flat files to the most powerful RDBMS	✓	✓	✓	
Produce cross-platform applications with flexible data-aware controls that are single-source between Delphi on Windows and Kylix on Linux!*	✓	✓	✓	
ClientDataSet-maintained aggregates and parameter support	✓	✓	✓	
Advanced application-side transaction caching to optimize RDBMS server connectivity	✓	✓	✓	
DataSetProvider for flexible binary- and XML-distributed data connectivity	✓	✓	✓	
Provider Optional Parameters, calculated fields, and field properties	✓	✓	✓	
Master/Detail Provider and Resolver support	✓	✓	✓	
On-demand Blobs	✓	✓	✓	
MyBase™ XML Data Briefcases for transparent mobile and low-bandwidth connectivity support	✓	✓	✓	
CommandText property for building custom queries at client side	✓	✓	✓	
<b>Broad range of data-access options—get access to data your application needs</b>				
dbExpress for ultra high-performance native SQL RDBMS access	✓	✓	✓	
Borland Database Engine (BDE) 5.1.1	✓	✓	✓	
IBX for Direct-API access to Borland InterBase®	✓	✓	✓	
All Delphi data-access options use common DataSet/DataCLX Programming Model—use any data-access method with one skill set	✓	✓	✓	

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
dbGo for ADO 2.5 direct access to any ADO-compliant dataset, from office applications to RDBMS	✓	✓	✓	
<b>Open database architecture—connect to all your data</b>				
Access, FoxPro®, Paradox®, and dBase® drivers for high-speed access to desktop and LAN database systems	✓	✓	✓	
Complete ODBC connectivity	✓	✓	✓	
BDE API for direct open access to any database engine	✓	✓	✓	
Support for autorefresh	✓	✓	✓	
TCustomConnection class to easily integrate third-party database solutions	✓	✓	✓	
SQL links native drivers with unlimited deployment license for InterBase, Oracle, Sybase®, Informix®, Microsoft SQL Server, and DB2	✓	✓		
<b>Integrated database development tools—simplify client/server development</b>				
<b>New!</b> TSimpleDataSet to simplify use of dbExpress drivers	✓	✓	✓	
<b>New!</b> ClientDataSet actions to simplify synchronization of database actions with user interfaces	✓	✓	✓	
Data-aware components to build powerful database applications	✓	✓	✓	
DB Control Grid	✓	✓	✓	
Cached updates for increased server responsiveness	✓	✓	✓	
Database desktop for creating and managing Paradox and dBase tables	✓	✓	✓	
Database Explorer to visually browse and manage tables, aliases, and indices	✓	✓	✓	
Scalable Data Dictionary to maintain data integrity	✓	✓	✓	
DBGrid support for Nested Tables, ADTs, Array and Reference fields	✓	✓		
SQL Explorer to visually manage server-specific meta-data, including stored procedures and triggers	✓	✓		
SQL Monitor to test, debug, and tune SQL applications for optimal performance	✓	✓		
SQL Builder for writing complex SQL easily	✓	✓		
Data Migration Wizard for rapid upsizing and application scaling	✓	✓		
<b>dbExpress native cross-platform database connectivity—access all your data at extreme speed</b>				
Multiple transactions on a single connection	✓	✓	✓	

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
dbExpress high-performance database driver interface for open access to any database engine	✓	✓	✓	
Single-file distribution of dbExpress applications!	✓	✓	✓	

### High-performance database drivers

<b>New!</b> dbExpress driver for Microsoft SQL 2000	✓	✓		
dbExpress drivers for InterBase, MySQL™	✓	✓	✓	
BDE local drivers for Paradox, dBase, FoxPro, Access	✓	✓	✓	
dbExpress drivers for Oracle, DB2, and Informix	✓	✓		
BDE SQL links for Oracle, Microsoft SQL Server, IBM DB2, Informix, Sybase, and InterBase *	✓	✓		

### Borland MyBase™ personal XML database engine

Database Engine-in-a-component design for lightweight single-file application distribution	✓	✓	✓	
XML file database for complete simplicity and inter-operability	✓	✓	✓	
Fast In-memory table performance	✓	✓	✓	
Zero configuration for instant setup and simple application distribution	✓	✓	✓	
Small footprint—less than 300k linked in executable or 0k with shared packages	✓	✓	✓	
ANSI SQL 92 Expression Syntax for live filtering and aggregation	✓	✓	✓	
Linkable master/detail table relationships	✓	✓	✓	
ClientDataSet based with transparent scalability to dbExpress and DataSnap for unlimited scalability	✓	✓	✓	
Import SQL query results from any dbExpress RDBMS into local MyBase XML tables	✓	✓	✓	
Support for all popular data types including Blobs	✓	✓	✓	

### Advanced Borland InterBase® support

InterBase 6.5 support including IBX native API components	✓	✓	✓	
InterBase 6.5 five-user development license to build and test remote multi-user SQL applications	✓	✓		
InterBase 6.5 two-user development license to build and test local multi-user SQL applications				✓

\* dbExpress, a SQL driver architecture, will replace BDE SQL Links—see <http://community.borland.com/article/0,1410,28688,00.html>

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
<b>Documentation, examples, and online help</b>				
Comprehensive online documentation for quick reference	✓	✓	✓	✓
Printed developer documentation for offline consulting	✓	✓	✓	✓
Sample applications to learn from and get started quickly	✓	✓	✓	✓
<b>COM/ActiveX®/COM+ features</b>				
Automatic COM+ object setup with configuration attributes for simplified COM+ object deployment	✓	✓	✓	
Event object wizards speed linking to COM+ events and VCL support for COM+ event registration	✓	✓	✓	
Transaction objects take advantage of the latest transaction control services in Windows	✓	✓	✓	
Dual MTS/COM+ support for transaction objects	✓	✓	✓	
Transaction datamodule for internal segregation of COM+ object elements	✓	✓		
<b>ActiveInsight™ —an enterprise component foundry for maximum reusability</b>				
Active document support	✓	✓	✓	✓
COM/interfaces for language interoperability	✓	✓	✓	✓
Full Windows 95/98/Me/NT/2000 support for OLE Automation Controllers and Servers	✓	✓	✓	✓
Runtime component packages for easily updated, small application executables	✓	✓	✓	✓
Visual component creation for instantly creating component templates	✓	✓	✓	✓
One-step ActiveForms to deliver applications over the Web	✓	✓	✓	
<b>Microsoft® ActiveX®</b>				
One-step industry-standard ActiveX® components for maximum reusability	✓	✓	✓	
Advanced Type Library Editor with complete support for MS IDL editing	✓	✓	✓	
Support for Automation Object event handling	✓	✓	✓	
COM Object Wizard	✓	✓	✓	
ActiveX® Control Data binding	✓	✓	✓	
Support for sparse vtables to support VB servers	✓	✓	✓	
Import COM servers as components to develop visually	✓	✓	✓	

## Borland® Delphi™ 7 Studio Feature Matrix

	ARCHITECT	ENTERPRISE	PROFESSIONAL	PERSONAL
<b>Third-party component solutions</b>				
Nevrona INDY—Internet Protocol Component Suite including HTTP, FTP, SMTP, POP3, and more	✓	✓	✓	
“Delphi Companion Tools” CD includes numerous third-party components and tools	✓	✓	✓	
TeeCharts	✓	✓	✓	
InstallShield® MSI Lite	✓	✓	✓	
InterBase 6.5	✓	✓	✓	
Commercial development license for professional software sales	✓	✓	✓	

\* Compiling applications for Linux® requires Kylix.™

Made in Borland® Copyright © 2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Microsoft, Windows, and other Microsoft product names are trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries. All other marks are the property of their respective owners. • 13305.1

# Borland<sup>®</sup>

## What's New in Borland<sup>®</sup> Delphi<sup>™</sup> 7 Studio

Improve developer productivity  
and enhance performance

*by Technical Publications  
Borland Software Corp.  
August 2002*

### Contents

Introduction	1
IDE	1
Web technology	3
COM technology	4
Database technology	4
Component library	5
Runtime library	5
Compiler	6
Support for Rave Reports	6
Support for ModelMaker	7
Documentation	7
Upgrade and compatibility issues	7

### Introduction

Borland<sup>®</sup> Delphi<sup>™</sup> 7 Studio includes new features and enhancements in the following areas:

- IDE
- Web technology
- COM technology
- Database technology
- Component library
- Runtime library
- Compiler
- Support for Rave Reports
- Support for ModelMaker
- Documentation

If you are upgrading from a previous version of Delphi, see "Upgrade and compatibility issues" on page 7.

### IDE

The IDE has new features in the following areas:

#### Compiler messages

- The new View | Additional Message Info command displays a Message Hints window from which you can download and view information about compiler messages from the Borland Web site.
- The new Project | Options | Compiler Messages page gives you greater control over which compiler warnings are generated.

# Delphi<sup>™</sup> Studio

# white paper

# white paper

## Component palette

- When you open a Borland CLX™ (Component Library for Cross-platform) application in Delphi, a new CLX-only version of the System page is displayed. It includes several directory and file components. In previous releases, the System page was displayed only for VCL applications and included components for system-level access.
- The new Indy Intercepts and Indy I/O Handlers pages provide open source Internet protocol components (Professional and Enterprise editions).
- The new IW Standard, IW Data, IW Client Side, and IW Control pages provide AtoZed Software IntraWeb components for developing Web-based applications.
- The new Rave page provides components for adding report generation to your applications.
- If a component page can be scrolled horizontally to display additional icons, a new drop-down menu button can also be used to list the additional icons.

## CodeInsight™

- Code completion is now faster and lets you browse to the declaration of items in the code completion list by using Ctrl+click on any identifier in the list.
- New HTML code completion automatically displays valid HTML elements and attributes in the Code editor (Professional and Enterprise editions only).
- You can create customized code completion managers by using the OpenTools API. See “Extending the IDE” in the Delphi online help for details.
- The Tools | Editor Options | Code Insight page lets you set colors for the symbols displayed in the CodeInsight™ tools.

## Debugger

- The Watch List now has:
  - Multiple tabs, allowing you to organize watches into distinct watch groups for easier debugging. To add a watch group, right-click the Watch List and select Add Group.
  - A Watch Name column and a Value column. To show/hide the column headers, right-click the Watch List and select Show Column Headers.
  - A checkbox to enable or disable individual watches.
- The Tools | Debugger Options | Event Log page has the following new options:
  - Use Event Log Colors lets you display different types of event messages in color in the event log.
  - Module messages writes a message to the event log each time a module (exe, dll, ocx, etc.) is loaded or unloaded by the process that you are debugging. Previously, the Process messages option controlled whether these events were logged.
- The Run Parameters dialog box has a new option, Working Directory, that lets you specify the name of the directory to use for the debugging process.

## Miscellaneous IDE improvements

- From the Project Manager, you can partially compile projects within a group by right-clicking on any project and choosing Make All from Here or Build All from Here.
- The Message view has multiple tabs for displaying different types of messages (Build, Search, and so on).
- The View | Component List command lets you multiselect components by pressing the Ctrl key.

- The new Tools | Editor Options | Source Options page lets you:
  - Set different editor options for different source types, such as Pascal, C++, C#, HTML, and XML.
  - Display tab and space characters in the Code editor.
  - Edit code templates.
  - Several of the options on the new page were formerly on the General, Display, and CodeInsight pages of the Editor Properties dialog box.
- The Tools | Editor Options | Color page has two new options, Foreground Color and Background Color, instead of a color grid, for setting colors in the Code Editor.
- Pressing Alt+Page Down and Alt+Page Up cycles through tabbed views such as the Code Editor, Watch Window, and Message view. These keyboard shortcuts are included in the Default, IDE Classic, and BRIEF key mappings.
- Delphi now displays a two-tone main menu.

## Web technology

In Delphi 7 Studio Enterprise and Professional editions:

- Delphi now includes IntraWeb from AToZed Software. You can use IntraWeb to develop Web server applications using standard form tools. You can also use IntraWeb to develop pages for Borland WebBroker™ and WebSnap™ applications. For more information, see “Creating Web server applications using IntraWeb” in the *Developer’s Guide* or online Help. Delphi 7 Studio Enterprise includes the complete IntraWeb product. Delphi 7 Studio Professional includes a subset of the IntraWeb product.
- Delphi now supports Apache™ 2 as a target type for WebBroker, WebSnap, and SOAP.
- Borland has deprecated Win-CGI as a target type for Web server applications and Web Services. Borland recommends using regular CGI, ISAPI/NSAPI, or an Apache target type instead. Existing Win-CGI projects can still be modified and

compiled in the IDE, however, Borland does not guarantee Win-CGI compatibility for the indefinite future.

## Web Services

Web Services includes the following enhancements.

### New UDDI browser

The WSDL Import Wizard has a new Universal Description, Discovery, and Integration (UDDI) browser that lets you search a UDDI registry for a Web Service and import the address of its WSDL document.

### SOAP headers

New classes and interfaces let you read or insert headers into the SOAP envelopes that transmit messages between clients and servers. For more information, see “Defining and using SOAP headers” and “Processing headers in client applications” in the *Developer’s Guide* or online Help.

### Attachments

Web Services applications (both client and server) can now handle attachments. Attachments (TSOAPAttachment descendants) are sent with SOAP-encoded messages as part of a multipart form. When an application receives the attachment, it saves it to a temporary file, which is then available to your application.

### Type support

- You can now customize the conversion between remotable classes and their SOAP representation by overriding two new virtual methods that were added to TRemotable: ObjectToSOAP and SOAPToObject.
- Exception objects for exceptions that occur when responding to a Web Service request (ERemotableException instances) now contain more information from the SOAP fault packet.
- Type definitions are automatically registered with the remotable type registry when you register an invocable interface.



- TXSDecimal has a new AsBcd property for easier conversion between XML and native types. Similarly, TXSHexBinary has a new AsByteArray property. Remotable classes that represent time values now let you work with fractional seconds rather than milliseconds.

## Other enhancements

- New events on THTTPReqResp let you to intercept the HTTP message before it is sent and to monitor progress while sending or receiving long messages.
- THTTPSoapPascalInvoker now publishes events that let you write code to execute before or after the invoker executes a requested method call.
- You now have more control over the mapping between invocable interfaces and WSDL documents. TWSDLHTMLPublish now publishes several events to let you control the generated WSDL. You can also identify the mapping between function return values and parameter names, the use of namespaces, and default SOAP actions. On the client side, literal encodings are now supported as well as RPC-style encoding.
- A new interface, IRIOAccess lets you access the remote interfaced object that implements an invocable interface.
- The IOPConvert interface has a new property: Encoding. This allows you to specify the character set to use for encoded messages that are passed between the client and Web Services provider.
- There are changes to Web Services that affect Borland DataSnap™ applications. For more information, see "Database technology" on page 4.
- The TLinkedRIO constructor now automatically generates separate file names for each method you call, making debugging easier.
- TOPToSoapDomConvert now has two new events that you can use when debugging the deserialization of SOAP packets.
- You can now use overloaded methods on invocable interfaces that you define.

## COM technology

In Delphi 7 Studio Enterprise and Professional editions:

You can now use the Import Type Library dialog box (Project|Import Type Library) to create a CoClass wrapper for Microsoft® .NET assemblies. You can use the resulting wrapper as you would an ordinary COM server, using the interoperability features of .NET.

## Database technology

In Delphi 7 Studio Enterprise and Professional editions:

- The dbExpress drivers have been updated for IBM® DB2® 7.2 and Informix® SE, Oracle9i,™ Borland InterBase® 6.5, and MySQL™ 3.23.49. A new driver is available for Microsoft SQL 2000.
- There are several new and changed database components. See "Component library" on page 5 for details.
- Borland has deprecated SQL Links; no further enhancements will be made to SQL Links, and it will not be included with Delphi after 2002. Borland recommends using dbExpress for SQL server database access in Delphi.

## DataSnap™

In Delphi 7 Studio Enterprise edition only:

- In DataSnap applications, the use of IAppServer has been changed to IAppServerSOAP, which avoids some ambiguities in the IAppServer interface. The UseSOAPAdapter property of TSoapConnection can be used to write clients for servers written with earlier versions of Delphi. TSoapConnection also publishes several new events for you to customize your client applications at various points in the process of executing a Web Services request.
- You can now identify a specific SOAP data module in an application server that has multiple data modules. Use the SOAPServerIID property or add the data module's interface to the end of the URL.

- You can now use the SOAP connection component to call extensions to the application server's interface. Use the SOAPServerIID property and the GetSOAPServer method.
- DataSnap no longer supports CORBA® connections.

## Component library

### Support for Windows XP™ Themes

In Delphi Studio 7 Enterprise and Professional editions, Borland VCL applications now include components that enable support for Windows® common controls version 6. Your application will automatically use the new Windows controls on Windows XP™ systems if it finds a suitable manifest file. For more information, see “Common controls and XP themes” in the *Developer's Guide* or online Help.

### New unit

The new DBClientActns unit contains three new action components for working with client datasets: TClientDataSetApply, TClientDataSetUndo, and TClientDataSetRevert.

### New components

- The dbExpress page of the Component palette includes TSimpleDataSet for use with simple, two-tier database applications (TSimpleDataSet replaces TSQLClientDataSet).
- The Dialogs page of the Component palette includes TPageSetupDialog for providing a Windows-standard page setup dialog box.
- The Additional page of the Component palette includes TXPColorMap, TStandardColorMap, and TTwilightColorMap for coloring menus and toolbars.
- The new CLX version of the System page of the Component palette includes new directory and file components.
- The new Indy Intercepts and Indy I/O Handlers pages on Component palette provide Internet protocols in Professional and Enterprise editions.

### Changed components

- The CLX versions of TOpenDialog and TSaveDialog have been expanded to support additional features such as file previewing.
- The VCL version of TCustomForm has two new properties, ScreenSnap and SnapBuffer, which control whether a form snaps to the edge of the screen when the form is moved.
- TCustomComboBoxEx has a new AutoCompleteOptions property that enables a combo box to respond to user keystrokes.
- CLX dialog objects that descend from TOpenDialog and TQtDialog can now use Windows Common Dialogs in place of Qt Dialogs. This behavior is controlled by the UseNativeDialog property, which defaults to true.

### Deprecated components

Information about deprecated components can be found in the readme.txt file in the Delphi 7 Studio directory.

## Runtime library

### Classes unit

- A new exception class, EFileStreamError, has been added. EFileStreamError and EOpenError descend from this class. This new class may take a FileName parameter. As a result, the exception message text now contains the name of the file the error occurred on.
- The TStrings class has two new properties, ValueFromIndex and NameValueSeparator.
- The TThread.CheckThreadError methods have been promoted from private to protected visibility.

### Math unit

The Math unit has a new default parameter, RaisePending, in the ClearExceptions procedure.

## StdConvs unit

The StdConvs unit now includes stones in the supported weight units.

## StrUtils unit

The StrUtils unit contains the following changes related to multi-byte character set (MBCS) support:

- Previously, LeftStr, RightStr, and MidStr each had an AnsiString parameter type and return type, and did not support MBCS strings. Each of these functions has been replaced by a pair of overloaded functions, one that takes and returns AnsiString, and one that takes and returns WideString. The new functions correctly handle MBCS strings. This change breaks code that uses these functions to store and retrieve byte values in AnsiStrings. Such code should be updated to use the new byte-level functions described below.
- New functions LeftBStr, RightBStr, and MidBStr provide the byte-level manipulation previously provided by LeftStr, RightStr, and MidStr.
- New functions AnsiLeftStr, AnsiRightStr, and AnsiMidStr are the same as the new AnsiStr LeftStr, RightStr, and MidStr functions, except that they are not overloaded with equivalent WideString functions.

The StrUtils unit has a new string-searching function called PosEx.

## SysUtils unit

The SysUtils unit now includes thread-safe overloads of routines that format and parse numbers, date-time values, and currency.

The new routines are thread-safe because they obtain their localization information from a TFormatSettings data structure instead of from global variables. This data structure must be populated before being used; a new function, GetLocaleFormatSettings, is provided to populate the data structure from a specified locale.

## VarCmplx unit

The VarCmplx unit has new functions: VarComplexLog2, VarComplexLog10, VarComplexLogN, VarComplexTimesImaginary, and VarComplexTimesReal.

## Variants unit

- The VarIsError and VarAsError functions have been added.
- The EVariantError exception is now a base class for finer grained exception classes that are thrown from variants code.
- Several new global Variant control variables have been added: NullEqualityRule, NullMagnitudeRule, NullStrictConvert, NullAsStringValue, and PackVarCreation.

## Compiler

The Delphi compiler now supports three additional compiler warnings: Unsafe\_Type, Unsafe\_Code, and Unsafe\_Cast. These warnings are disabled by default, but can be enabled with the compiler directive { \$WARN UNSAFE\_CODE ON }, compiler command line switch `dcc32 -W+UNSAFE_CODE`, and, in the IDE, on the Project | Options | Compiler Messages page.

This feature is intended to help you port your code to the managed execution environment of the .NET platform. In a managed execution environment, "unsafe" means the operation cannot be verified during the static analysis performed by the Just In Time (JIT) compiler. Such code might pose a security risk, since there is not enough information for the JIT compiler to verify its runtime behavior. Examples of unsafe code include pointer operations and memory overwrites.

## Support for Rave Reports

In Delphi 7 Studio Enterprise and Professional editions:

The Delphi environment now includes Rave Reports from Nevrona. By adding Rave Reports components to your application, you can enable your users to generate reports within your application. For more information see "Creating reports with Rave Reports" in the *Developer's Guide* or online Help.

## Support for ModelMaker

In Delphi 7 Studio Enterprise and Professional editions: ModelMaker tools can help simplify the design, construction, and maintenance of classes and interfaces. ModelMaker also includes tools for creating UML™-style diagrams, which can be used to create and modify your projects' source code. For more information, see "Designing classes and components with ModelMaker" in the *Developer's Guide* or online Help.

Delphi 7 Studio Enterprise includes ModelMaker from ModelMaker software. Delphi 7 Studio Professional includes a 30-day trial version of ModelMaker. The ModelMaker functionality is the same in both editions of the Delphi environment.

## Documentation

- All of the documentation files (PDF, HTML, and INT) are now distributed in the **Online+PDF Docs** folder on the Delphi Companion Tools CD instead of the installation CD. You can access the documentation directly from the CD or copy it to the folder of your choice.
- Due to size constraints for the printed *Developer's Guide*, Part V "Creating custom components," has been removed from that book to create the new *Component Writer's Guide*. The new book is available in the online Help and as a PDF file on the Delphi Companion Tools CD.
- The Object Pascal language is now called the Delphi™ language. The online Help and documentation have been updated accordingly.
- The *Object Pascal Language Guide* is now the *Delphi Language Guide*.

- To ensure the continued accuracy of the Delphi tutorials, they have been removed from the *Quick Start* and the *Developer's Guide*. The tutorials are available as PDF files on the Delphi Companion Tools CD.
- Some of the Delphi online help topics include C++ syntax and code examples for Borland Kylix™ and C++Builder™ users. For Delphi development, please disregard these examples.

## Upgrade and compatibility issues

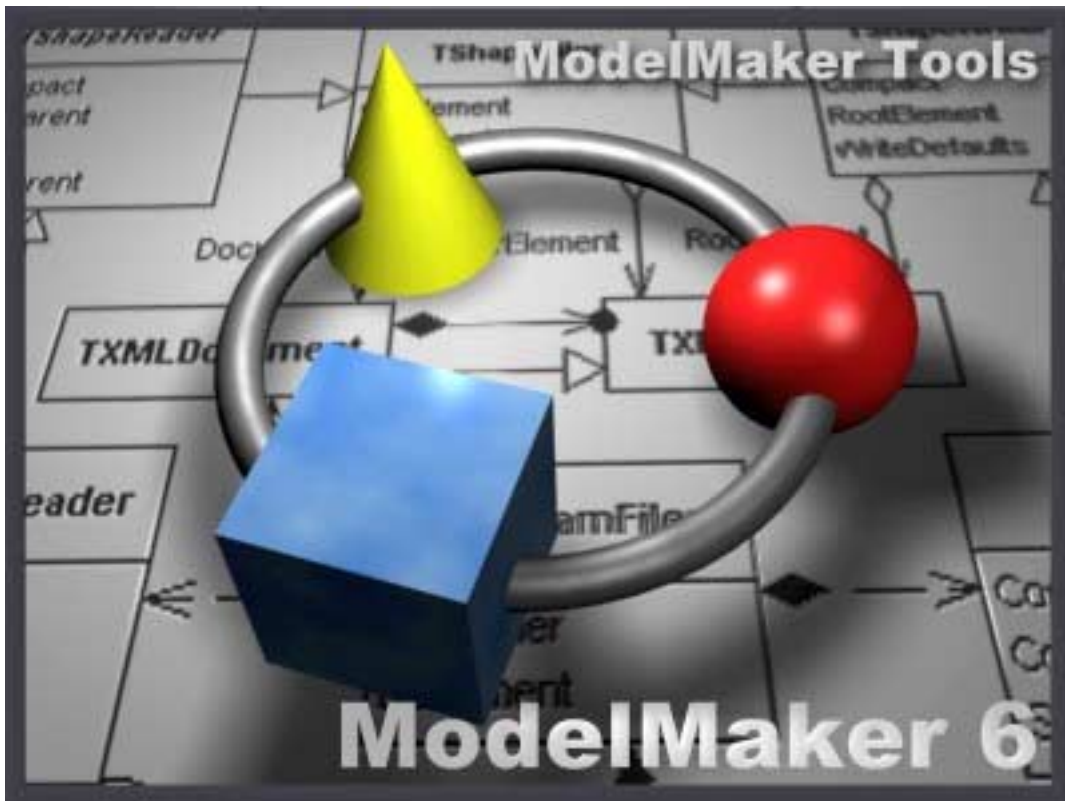
For late-breaking upgrade and compatibility issues, see the readme.txt file in the Delphi 7 installation directory.

- To upgrade a Delphi (formerly Object Pascal) language project from a previous version of Delphi, open it in the new version. The project is automatically updated to the new release.
- Details in Fault messages are now added to the <detail> node rather than as children of the <detail> node. This brings our handling of SOAP faults into accordance with the SOAP specification, but breaks backward compatibility with older code.
- Changes to the StrUtils unit LeftStr, RightStr, and MidStr functions may require you to update code that uses these functions. See "Runtime library" on page 5 for details.
- For Apache 2, the variable "ContentType" has been changed to "handler" in the ApacheApp unit.
- DataSnap no longer supports CORBA® connections.

# Borland®

100 Enterprise Way  
Scotts Valley, CA 95066-3249  
www.borland.com | 831-431-1000

Made in Borland® Copyright © 2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • www.borland.com • Offices in: Australia, Brazil, Canada, China, Czech Republic, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. • 13393



# User Manual

## ModelMaker 6.20

ModelMaker Tools  
Stenenkruis 27 B  
6862 XG Oosterbeek  
The Netherlands

<http://www.modelmakertools.com>  
[info@modelmakertools.com](mailto:info@modelmakertools.com)

<http://www.modelmaker.demon.nl>  
[info@modelmaker.demon.nl](mailto:info@modelmaker.demon.nl)

ModelMaker version 6.20

Copyright © 1997-2002 by:

ModelMaker Tools  
Stenenkruis 27 B  
6862 XG Oosterbeek  
The Netherlands

<http://www.modelmakertools.com>  
[info@modelmakertools.com](mailto:info@modelmakertools.com)

<http://www.modelmaker.demon.nl>  
[info@modelmaker.demon.nl](mailto:info@modelmaker.demon.nl)

All rights reserved.

All brand and product names are trademarks or registered trademarks of their respective holders.

*This user manual focuses on essentials and how things are done in ModelMaker. A GUI reference is available as context sensitive help file. This contains the latest GUI details. The Design Patterns manual focuses on ModelMaker's Design patterns and contains another step by step demo.*

Author: G. Beuze

# Contents

<b>Introduction</b>	<b>8</b>
Installation	9
Contacting ModelMaker Tools	9
<b>Getting started</b>	<b>10</b>
Getting a first impression	10
Loading an example model	10
Visualizing existing code	10
Creating code with ModelMaker, overview	11
The demo component: TIntLabel	11
The ModelMaker Class Creation Wizard	12
Creating a new project	12
Creating new classes	12
Adding properties and methods to a class	13
Implementing methods	15
Creating a source file	18
Creating a Unit	18
Generating the source file	20
Adding the component to the VCL	21
Debugging your component	21
Compiling errors	21
Adding the component to the VCL	21
Improving the component in ModelMaker	22
Keep editing your code in ModelMaker	22
Overriding the constructor Create	22
Implementing Create, non-user sections in code	23
Instant code generation	24
Documenting your component	25
Adding documentation to your component	25
Creating a help file	27
Integrating your help files with Delphi's on line help	28
Documenting the design in a diagram	28
Symbol styles in Diagrams: displaying members	30
Visualizing the unit IntLabel.pas	31
Visualizing the Documentation	33
Summary	34
Where to now?	35

<b>Basic Concepts</b>	<b>36</b>
Overview	36
Code Model contents	38
Diagrams	39
<b>Working with models</b>	<b>41</b>
Model files	41
Model templates	41
Editing a model	42
Ownership in ModelMaker	43
Team development, Model boundaries and Version Control	43
<b>Generation source code</b>	<b>45</b>
Overview	45
Code generation control tags	46
Class related tags	46
Event type declaration tag	47
Editing marker tags	47
Macro expansion control tags	47
Unit documentation tag (obsolete)	48
Obsolete tags	48
Code generation options	49
Maintaining Code Order / Custom member order	49
Adjusting the unit template	50
Unit Time Stamp Checking	51
Source Aliases	52
Version Control support and Aliases	53
Using ModelMaker to generate Instrumentation code	54
<b>Importing source code</b>	<b>56</b>
Background	56
Importing a source file	57
Importing (adding) versus Refreshing	59
Avoiding creep - removing code during import	59



STARTREMOVE and ENDREMOVE tags	59
Comments with remove signature	60
<b>Import restrictions and limitations</b>	<b>60</b>
Class and Interface interfaces	60
Method implementation	61
Comments and white space	62
Unsupported language constructs	62
<b>Conversion errors</b>	<b>64</b>
<b>Auto Refresh Import</b>	<b>65</b>
How it works	65
How it is activated and controlled	65
Warnings	65
<b>Editing Form source files</b>	<b>66</b>
<b>In source documentation</b>	<b>67</b>
Overview	67
Generating in-source documentation	67
Importing in-source documentation	68
<b>Code templates</b>	<b>71</b>
Creating a Code template	71
Applying a Code template	71
Registering a Code template	72
Parameterize a Code template using macros	72
<b>Macros</b>	<b>74</b>
Overview	74
Macros in Code generation	74
Predefined macros	75
Using Macros in code	77
Using macros in the code editors	79
Using macros in your default unit template	79
<b>Diagrams</b>	<b>80</b>
Diagrams, Diagram List view	80
Symbols and visual containment	80
Associations	81

<b>Visual styles</b>	<b>83</b>
Style hierarchy	84
Visual style properties	84
Controlling & assigning styles	85
Style Manager	86
Printing Style	87
<b>Symbol (contents) style</b>	<b>87</b>
Style hierarchy	87
Controlling & assigning styles	88
Class & Interface symbols	89
Package symbols (units)	89
<b>Size and Alignment</b>	<b>90</b>
The Drawing Grid	90
Align & Size Palette	90
<b>Hyperlinks, navigation</b>	<b>90</b>
External documents	91
<b>Coupling Symbols to the Code Model</b>	<b>92</b>
HotLinks	92
Specialized symbols and associations	93
<b>Documentation &amp; OneLiners</b>	<b>94</b>
Floating Documentation view	94
Linked Annotations	94
<b>Diagram Editor</b>	<b>95</b>
Properties	95
Keyboard and Mouse control	95
<b>Drag &amp; Drop and conversions</b>	<b>97</b>
<b>Classes view</b>	<b>97</b>
Internal (tree mode)	97
Internal (list mode)	97
Source	97
Target	97
<b>Members view</b>	<b>98</b>
Internal	98
Source	98
Target	98
<b>Units view</b>	<b>99</b>
Internal (tree mode only)	99
Source	99
Target	99
<b>Method Implementation view</b>	<b>100</b>
Method Local Code Explorer	100
Method Implementation Section list	100
<b>Method Implementation Code Editor</b>	<b>101</b>
Internal	101

Source	101
Target	101
<b>Unit Code view</b>	<b>101</b>
Unit Code Explorer	101
Unit Code Editor	102
<b>Event Library view</b>	<b>102</b>
Internal	102
Source	102
Target	102
<b>Diagrams view</b>	<b>102</b>
Internal (tree mode only)	102
Source	103
Target	103
<b>Diagram Editor</b>	<b>103</b>
Internal	103
Source	103
Target	103
<b>Customizing ModelMaker</b>	<b>104</b>
<b>Integration with the Delphi IDE</b>	<b>105</b>
Integration with Delphi 3 and higher	105
Delphi 4 and higher	106
Delphi 4 and higher syntax highlighting scheme	106
Uninstalling IDE integration experts	107
Integration with Delphi 1 and 2	107
Installing the integration unit in Delphi 1 /2	107
Installing UNITJMP.EXE as a DELPHI 1 /2 IDE tool	108
<b>MMToolsApi primer</b>	<b>109</b>
Interfaces basics	109
Expert DLL basics	109
MMToolsApi version control	110
Interfaces and memory management	110
Adding an expert and menu items	111
Accessing Diagrams through the API	112
Accessing Experts through scripting	113

# Introduction

ModelMaker represents a brand new way to develop classes and component packages for Borland Delphi 1-6. ModelMaker is a two-way class tree oriented productivity, refactoring and UML-style CASE tool specifically designed for generating native Delphi code (in fact it was made using Delphi and ModelMaker). Delphi's Object Pascal language is fully supported by ModelMaker. From the start ModelMaker was designed to be a smart and highly productive tool. It has been used to create classes for both real-time / technical and database type applications. ModelMaker has full reverse engineering capabilities.

ModelMaker supports drawing a set of UML diagrams and from that perspective it looks much like a traditional CASE tool. The key to ModelMaker's magic, speed and power however is the active modeling engine which stores and maintains all relationships between classes and their members. Renaming a class or changing its ancestor will immediately propagate to the automatically generated source code. Tasks like overriding methods, adding events, properties and access methods are reduced to selecting and clicking.

The main difference between ModelMaker and other CASE tools is that design is strictly related to and native expressed in Delphi code. This way there is a seamless transition from design to implementation currently not found in any other CASE tool. This approach makes sure your designs remain down to earth. The main difference between ModelMaker and other Delphi code generators are it's high level overview and restructuring capabilities letting you deal with complex designs.

A unique feature, currently not found in any development environment for Delphi, is the support for design patterns. A number of patterns are implemented as 'ready to use' active agents. A ModelMaker Pattern will not only insert Delphi style code fragments to implement a specific pattern, but it also stays 'alive' to update this code to reflect any changes made to the design

As a result, ModelMaker lets you:

- Speed up development
- Produce designs and code of unequalled quality.
- Think of designing code instead of typing code.
- Design without compromising.
- Refine and experiment with your designs until they are just right.
- Create and maintain magnitudes larger models in magnitudes less time.
- Document you designs in UML style diagrams.
- Document your units in help files by clicking a single button.
  
- In short: *save time and money, making better software.*

## Installation

For installation details, refer to the readme.txt which is part of all ModelMaker distribution archives. We suggest you read this file before installing. The readme.txt also contains the latest information available on precautions related to upgrading.

ModelMaker requires Windows 95/98/ME/2000 or Windows NT 4.0 Both Borland Delphi and ModelMaker use a lot of resources. This might lead to problems under resource limited systems as Win95/98/ME.

ModelMaker is designed to work on a high resolution monitor (800x600 or better).

## Contacting ModelMaker Tools

We find it important to support you in your use of ModelMaker all the ways we can. You can find ModelMaker on the Internet at <http://www.modelmakertools.com> or alternatively <http://www.modelmaker.demon.nl> At this web site:

- We'll update you on the most recent news concerning ModelMaker and it's development.
- We'll have the latest demo versions available.
- You can consult the Tips, FAQ pages.
- You'll find links to the web-based ModelMaker newsgroups.
- You can leave hints or requests for future versions of ModelMaker.
- You can report bugs.

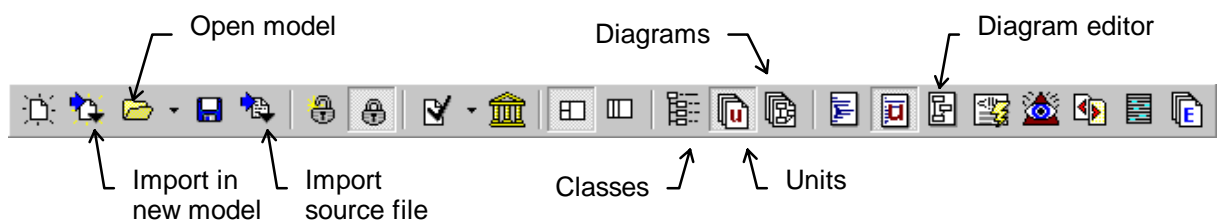
All 'how to do this' are best asked in the ModelMaker newsgroups. For other questions, the address to contact us is: [info@modelmakertools.com](mailto:info@modelmakertools.com) or [info@modelmaker.demon.nl](mailto:info@modelmaker.demon.nl).

# Getting started

## Getting a first impression

Here are some examples to get you started without reading lots of text. The development model will be explained in detail in the next chapters.

### Loading an example model



To get a first impression of what ModelMaker is capable of, load the model `..\ModelMaker\6.0\Demos\MMToolsApi.mpb`. It contains the interfaces making up ModelMaker's open tools API. In diagrams the relations between the interfaces defining this API are visualized. The Classes view shows the inheritance relations of this unit.

Then, to see how ModelMaker treats classes and units, use the 'Import source file in new Model' command from the toolbar. This will create a new model and import a Delphi unit (such as a form unit or a VCL unit). The model is named after the unit: Importing unit1.pas will create model unit1.mpb. Note the use of source aliases in the popup menu associated with the tool button. Source aliases are used to locate the source file and will be explained in chapter Source Aliases, page 52

### Visualizing existing code

Visualizing existing code is a also good way too of getting started with ModelMaker. To visualize code:

1. Import the units containing the classes to visualize. Use the "import source file" tool button in the main toolbar or drag drop source files on the 'unit's view' (View|Units)
2. Create or select a new class diagram in the 'diagrams' view (View|Diagrams)
3. In the Diagram editor (View|Diagram Editor) select the visualization wizard from the Wizard popup-up sub-menu.
4. Use this wizard to select the classes and interfaces to visualize and the kind of relations to visualize (inheritance, uses, supports etc.)
5. Completing the wizard gives you an instant diagram of the code just imported.

You might want to move around classes or interfaces (Drag move) or select different display options for classes or interfaces (Double click on the symbol) or the diagram as a whole

(Double click in empty space). Try to turn on and off member display, select interface style etc.

## Creating code with ModelMaker, overview

A ModelMaker model contains a Code Model and Diagrams. The Code Model contains the classes, class members (properties, methods), units etc. that map to the corresponding entities in Delphi's Object Pascal. Diagrams are used to visualize aspects of the code model or entities that do not exist in the code model at all such as use cases. This 'Getting started' example will focus on the code model and demonstrate creating a new unit containing a new component class.

To create code for a new (component) class (or interface) in ModelMaker you will at a minimum need to,

1. Create a new model in which you want to put related classes, if you don't want to add the new class to the current model.
2. Add a new class to the model defining its class name and ancestor.
3. Add (or override) properties, methods and events to the class's interface.
4. Implement the new methods.
5. Add the new class to a (new) unit.
6. Generate the unit to actually create or update a source file on disk.
7. In Delphi, debug the unit, and if it contains components, add it to the VCL.
8. While debugging, keep editing your code in ModelMaker, switching between Delphi and ModelMaker using ModelMaker's integration experts

An alternative way is to import existing files into a new model to either maintain these units in ModelMaker or to derive new classes from. This will be explained in detail in the Import demo.

The more advanced features of ModelMaker demonstrated in this example include:

9. Creating documentation for your component.
10. Generating a Help file.
11. Creating a class diagram to document your design.

### The demo component: TIntLabel

Let's examine these steps a little closer by creating a new component class `TIntLabel` which is a `TLabel` descendant. `TIntLabel` adds a property `NumValue` of type integer which simply converts the `Caption` property to an Integer. We'll store this class in a new file `INTLABEL.PAS` and register it on page '*MM Demo*' in the VCL. We'll also create the help file `INTLABEL.HLP` and integrate it with Delphi's on line help.

This demo project is also shipped with ModelMaker. You can load the `GETSTART.mpb` in the `[installdir]\DEMOS` folder. The source file `INTLABEL.PAS` is also in this folder.

## The ModelMaker Class Creation Wizard

If you start up ModelMaker the first time you'll see the Class Creation Wizard. This wizard makes it easy to create new classes and add them to a (new) unit. This wizard can be found at the main menu 'Tools|Create Class wizard'.

The wizard is great for adding classes, but for demonstrating the ModelMaker development model it's more instructive to create a new class manually. Therefore, if you started ModelMaker and the wizard is automatically started, abort the wizard by clicking 'Cancel'. You might also want to uncheck the option 'Show at start up' which will stop the wizard from appearing each time you run ModelMaker.

## Creating a new project

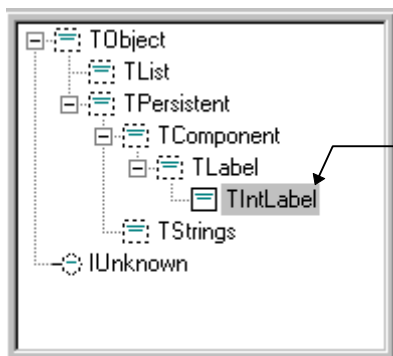
There are three ways to create a new project (or model):

- Select 'File|New', you'll get a clean project just containing the default ancestors `TObject` and `IUnknown`.
- Select 'File|New from default', you'll get a new project loaded from the default template.
- Select 'File|New from template', you'll select a template other than the default to create a new project.

In this case select 'File|New from default' to create a project which at least contains the `TComponent` class, if you didn't modify the default project shipped with ModelMaker `[installdir]\BIN\DEFAULT.mpb`.

## Creating new classes

We'll use the Classes view to create a new class. In this view you add a new class as a



Class tree containing placeholder TLabel and real class TIntLabel

descendant to another class in the model. The Classes view is depicted here.

The ancestor class must *always* be part of the model since ModelMaker needs it to correctly generate the class declaration. In our case this implicates that before adding the `TIntLabel` class, it's ancestor `TLabel` must exist in the model. This raises a problem. If you

started with the same default model, or with a template model that did not contain the class



TLabel, you will have to add the TLabel class first. But, in order to correctly add a class TLabel to your model, you now need to add it's ancestor first, and before that, etc.... help!

Since you do not intend to create code for TLabel, but only use it as an ancestor, there is no need to have the correct ancestor for TLabel. In our example the ancestor for TLabel could be anything, for example TObject. A better fitting ancestor is of course TComponent. Classes like TLabel in our example are called 'placeholder' classes as opposed to 'real' classes such as TIntLabel. Other examples of placeholders are TObject, Delphi's default class ancestor and TComponent.

What we have to do now, is add two classes TLabel ('placeholder') and then add TIntLabel ('real' class).

To do so:

1. Make the Classes view visible by selecting 'View|Classes' (or press F3)
2. Select TComponent by clicking it.
3. Press the "Ins" key or select add 'Add descendant' from the popup menu.
4. Enter TLabel as class name. You might want double click the class and in the class editor dialog check the option 'placeholder' to make it clear that TLabel is just a substitute for the real TLabel (which is in unit StdCtrls).
5. Now add the class TIntLabel using the TLabel as it's ancestor the same way. Of course you don't check 'placeholder' here.

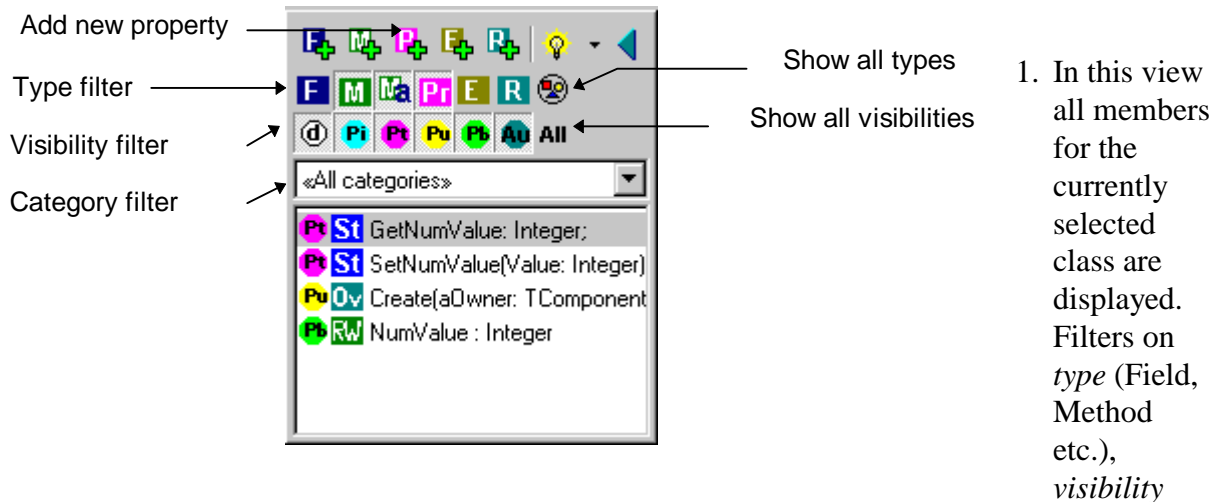
In the Classes view you'll see a tree or list based overview of all classes (and interfaces) in the model. Use the popup menu to toggle between tree and list style.

## Adding properties and methods to a class

In our example we now need to add a new property and a read and write access method to the interface of the class TIntLabel, to get something like:

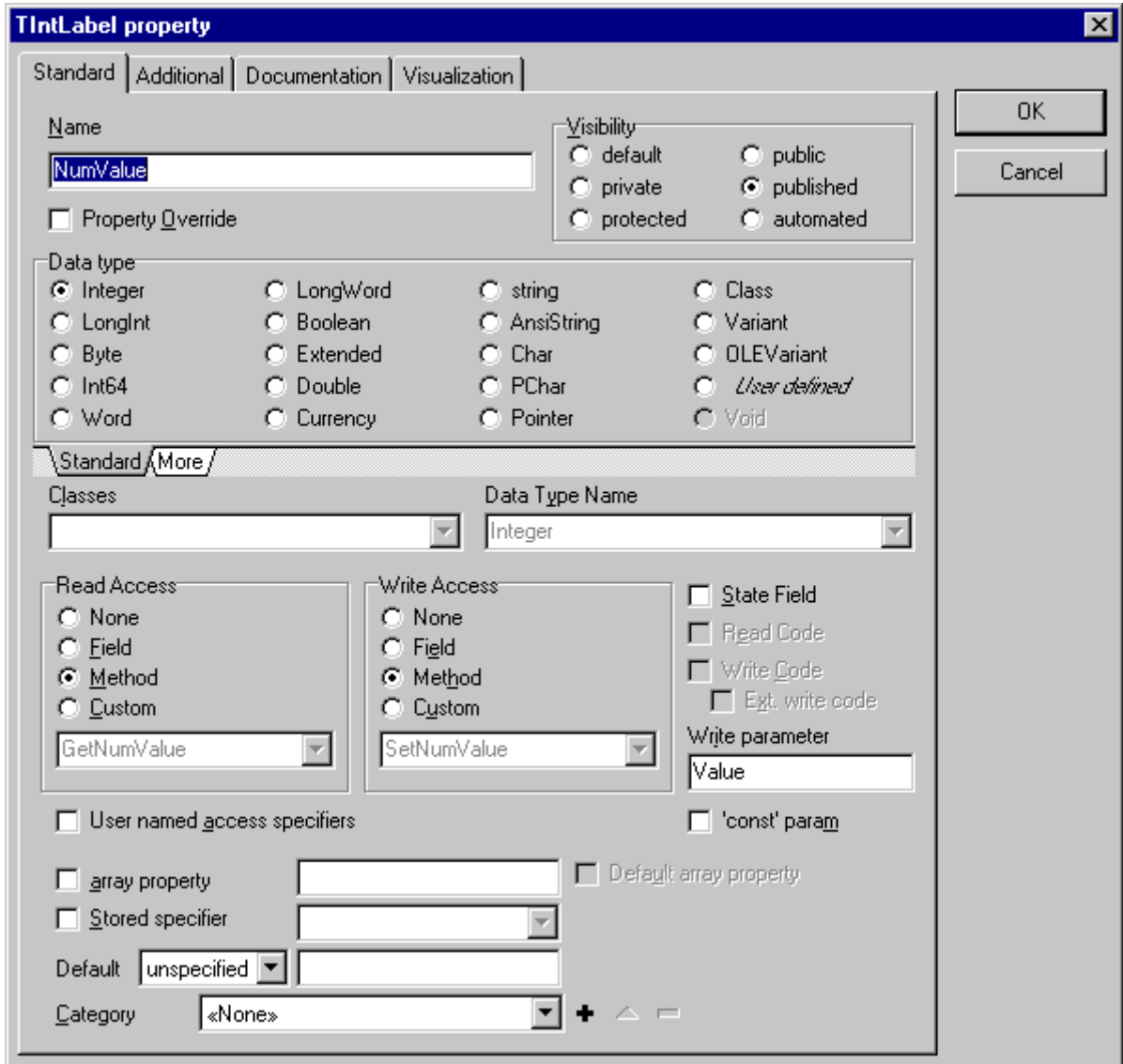
```
type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue: Integer read GetNumValue write SetNumValue;
  end;
```

To do this we'll use the Class Members view - the bottom left window in the main window. Class Members are the fields, methods, properties (and event type properties) that make up a class's interface. The Class Members view is depicted here.



(private, protected etc.) and *category* let you filter which members are displayed. Reset the filters by selecting 'Reset Filter' from the pop-up menu or by clicking the buttons 'Show all types' and 'Show all visibilities'. All filter buttons should be in a 'down' state now, except of course the 'Show all..' buttons which do nothing but (p)reset the filters. Make sure the category filter shows <all categories>. The member list is still empty because we didn't create any new Class Members yet. Note that the filter layout can be toggled using the popup menu 'filter layout or double clicking on the filter area.

2. Click the 'Add property' button.
3. The property editor dialog will appear. See picture below.
4. Enter NumValue as the property's name.
5. Select the visibility 'published'.
6. Make sure the property's data type is 'Integer'.
7. Select for Read Access 'Method'. This defines that the property has read access and you want to use a method to access it, rather than a field.
8. Select for Write Access 'Method' This defines that the property has write access and you want to use a method to access it, rather than a field.
9. Leave the other settings in their default values and click OK. In the property editor's picture below the correct settings are displayed.



Now have a look again in your Class Members view:  
 You'll not only see a property `NumValue`, but also two property access methods `GetNumValue` and `SetNumValue`. This is because properties create and update their access fields and methods automatically. Now that saves time!

The `TIntLabel` class's interface is now defined, but the methods `GetNumValue` and `SetNumValue` still need to be implemented.

## Implementing methods

In our example we will need to add code to the implementation of the methods `GetNumValue` and `SetNumValue`. This code should be something like:

```
function TIntLabel.GetNumValue: Integer;
begin
```

```

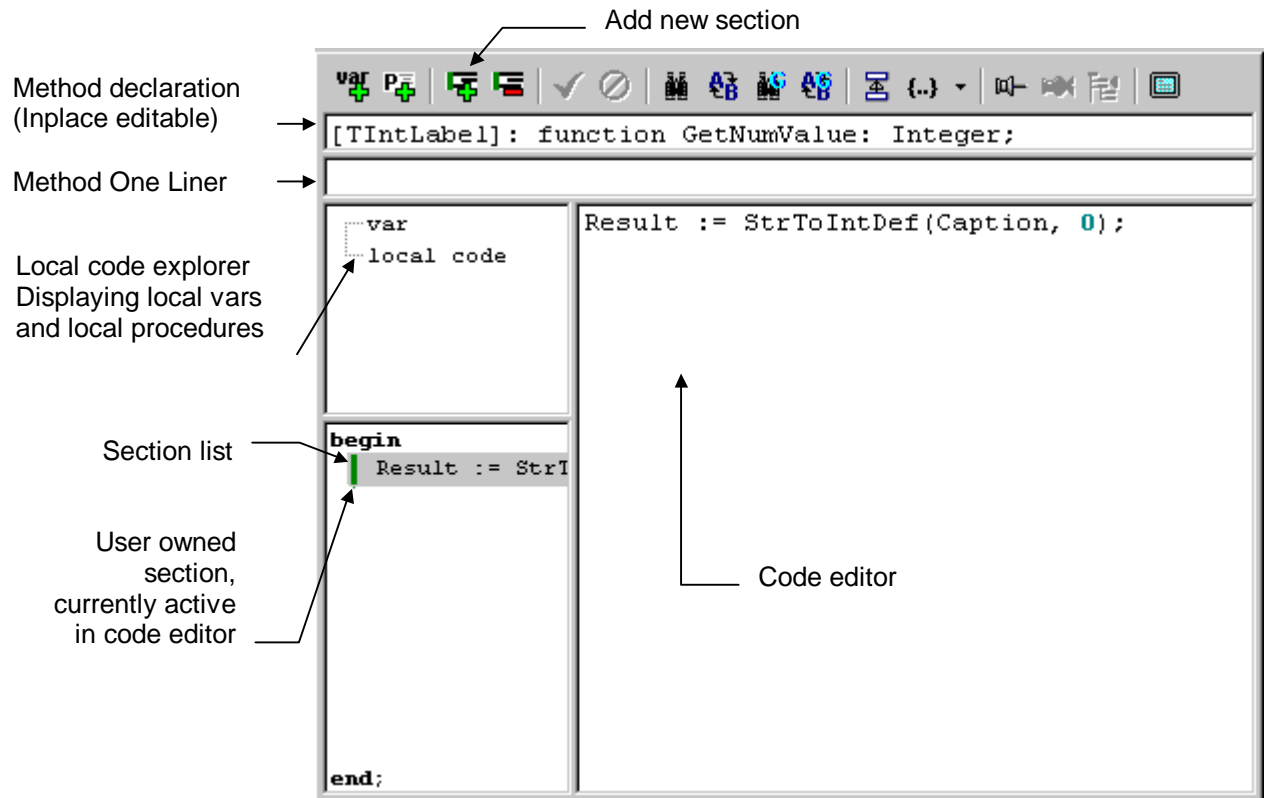
    Result := StrToIntDef(Caption, 0);
end;

procedure TIntLabel.SetNumValue(Value: Integer);
begin
    Caption := IntToStr(Value);
end;

```

To add code to a method's implementation you use the (Method) Implementation view.

1. Select the method you want to implement in the Class Member view, in this case the method `GetNumValue`.
2. To make the Method Implementation view visible, select 'View|Implementation'.



The picture above shows the Method Implementation view. This editor is perhaps the element of ModelMaker that is the most different from other editors. To understand how this editor works you need to know a little more about how ModelMaker generates code for a method's implementation.

Let's have a closer look at the `GetNumValue` method. This is just a simple method, not containing any local variables or local procedures.

ModelMaker will generate the method's header as defined in the interface

This is a *section* of code you add to actually implement the method. ModelMaker will indent this section for you.

ModelMaker will insert the reserved words `begin` and `end`.

```

function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0);
end;

```

The body of a method's implementation consists of a list of local variables, local procedures and *sections* of code which implement the block between `begin` and `end`. A section can take up any number of lines of code. All sections together make up the actual implementation. Using sections, ModelMaker is able to identify certain lines of code within the body. This is for example used to automatically add and update a call to the inherited method, as we'll see later.

On the left side of the method code editor the complete method's code is displayed, although maybe *collapsed* if necessary. On the right we find the actual code editor. It is used to edit the section of code selected in the sections list. The same editor is also used to edit the local procedures code.

As we see in the above picture, the only thing we need to do, is add a section of code containing the statement:

```
Result := StrToIntDef(Caption, 0);
```

To do this, first create a new section. If you didn't change the code options settings in 'Options|Code options' a new section is automatically created if the method does not contain any sections yet.

1. If necessary, add a new section by clicking the 'Add section' button.
2. Enter the statement in the code editor. There's no need to indent the code with spaces since this will be done automatically by ModelMaker.
3. Click the 'save code' button. This is not really necessary, since ModelMaker will automatically save the section as soon as you select a new section or a new method.

Notice that the section is now also displayed in the section list, and is marked with a green line. This green line informs you that *you* created this section and are its *owner*. Red lines indicate that a section is not owned by you, but, for example, is inserted by a pattern which has the only rights to update it. If a section contains more lines than the current 'Fold height' (adjustable in the Environment options tab Editors), the section will be collapsed. Collapsed sections are marked with a second purple line with a mark on the collapsing position. More about this later.

Now you should be able to implement the `SetNumValue` method the same way: select the method in the Class Members view, add a section if necessary and enter your code.

Although we have finished implementing the class `TIntLabel` for now, all code exists only in the ModelMaker model, so the next thing to do is to generate a source file.

## Creating a source file

Units are the gateways to source files on disk. They provide a link between all data in a ModelMaker project such as classes, method implementations etc. and an Object Pascal style unit file which Delphi is able to compile.

### Creating a Unit

In this example we need to create a unit which, after it has been generated, should look something like:

```
unit IntLabel;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue: Integer read GetNumValue write SetNumValue;
  end;

procedure Register;

implementation

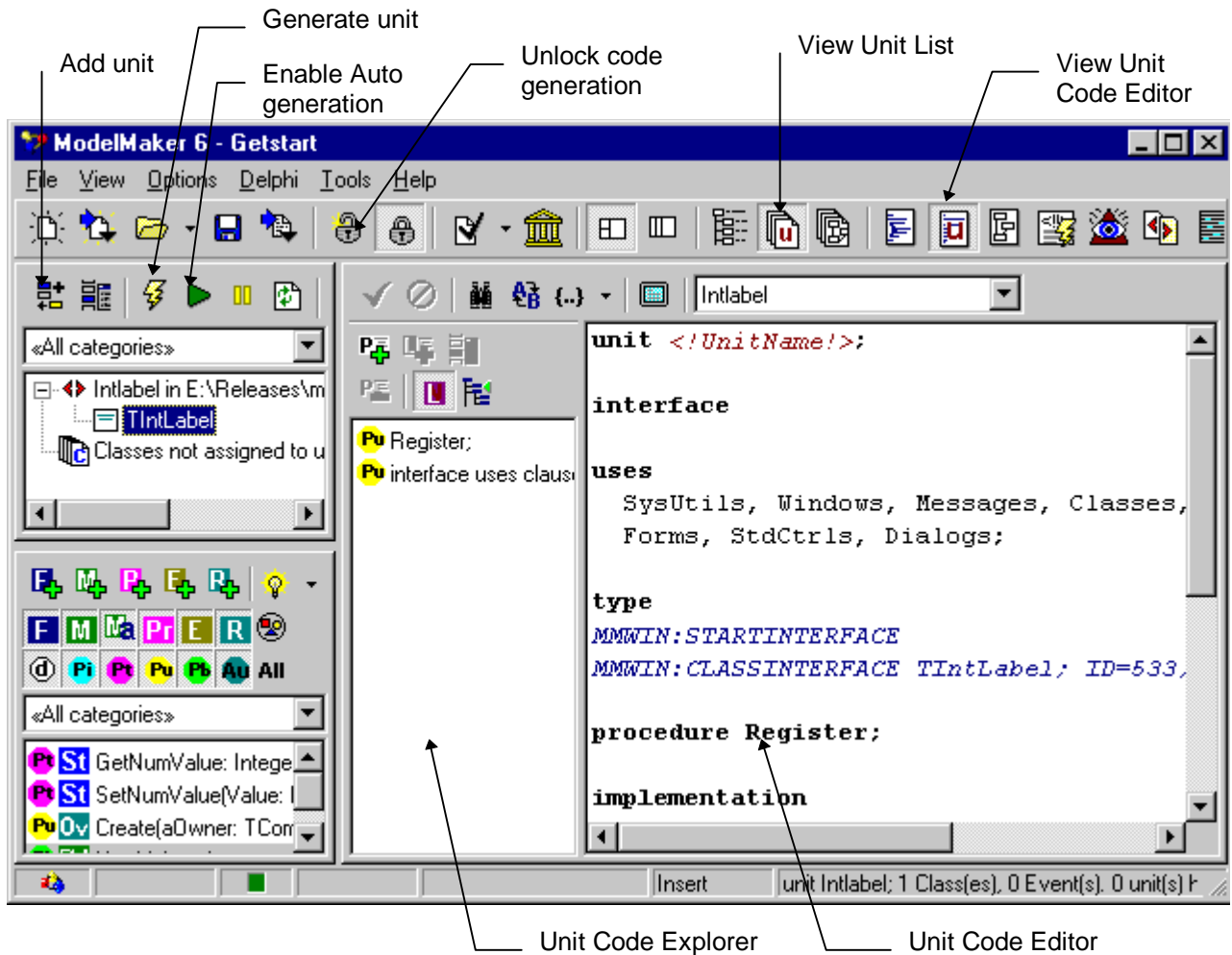
procedure Register;
begin
  RegisterComponents('MM Demo', [TIntLabel]);
end;

function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0);
end;

procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value);
end;

end.
```

To create a new unit we use the Unit List view which is depicted below,



1. To make the Unit list view visible, select 'View|Units'. Repeat this for the Unit Code editor.
2. In the Unit list click the 'Add unit' button, a unit properties dialog will now appear. In the Unit editor dialog you define:
  1. Leave the source path alias <no alias> unchanged.
  2. The source file name (the full path including drive and folders), to define the path you could use an source alias, but for now just click the browse button and locate the \ModelMaker\6.0\TEST folder (or ModelMaker\6.0\Test depending on the base path you installed ModelMaker in) and enter the file name IntLabel.PAS.
  3. On the tab sheet 'Classes' add the class TIntLabel to the list on the right either by dragging or by selecting it and clicking the 'Add selected' (▶) button or by double-clicking.
  4. Change the 'VCL page' from <unregistered> to 'MM Demo' by entering this name in the string grid.
  5. Click OK.

We'll see the newly created unit now listed in the unit list on the left. In this editor unit's code containing a text which should look something like:

```
unit <!UnitName!>;
```

```
interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  MMWIN:STARTINTERFACE
  MMWIN:CLASSINTERFACE TIntLabel; ID=7;

procedure Register;

implementation

procedure Register;
begin
  MMWIN:CLASSREGISTRATION TIntLabel; ID=7; Page='MM Demo';
end;

  MMWIN:STARTIMPLEMENTATION
  MMWIN:CLASSIMPLEMENTATION TIntLabel; ID=7;

end.
```

For now, it is enough to understand that ModelMaker uses tags (like `MMWIN:CLASSINTERFACE`) to insert the interface, VCL registration and implementation of a class in otherwise 'plain' text that you define. There is one problem however: if you look at the **uses** clause in the interface, you'll see that the unit `StdCtrls` which defines the ancestor class `TLabel`, is missing. That is because the default unit template we are using does not contain this unit. For changing this template refer to 'Customizing ModelMaker'. For now, we will have to add `StdCtrls` manually.

To do this,

1. In the unit code editor add `StdCtrls` to the **uses** clause.
2. Click the 'Save code' button in the toolbar above the editor.

## Generating the source file

To generate the source file and create or update a file on disk,

1. Make sure that code generation is not locked. Locking is explained later, for now make sure the button 'unlock code generation' in the ModelMaker toolbar is pressed down.
2. Click the 'Generate current unit' button in the unit list view.
3. Start Delphi (if it was not running already).
4. Either manually switch to Delphi or - much more instructive - click the 'Locate in Delphi' button in the main tool bar - or simply press `Ctrl+F11`. This will open the unit and locate the entity currently selected in ModelMaker.

The generated source file should look pretty much the same as we wanted it to be. (Differences may occur if you modified your file `DEFUNIT.PAS` in ModelMaker's `\BIN` folder.)

We're ready to debug the `TIntLabel` now, and install it in our VCL. Before doing this it's a good idea to save our model. This is very much like in other windows applications, so it won't be explained here. We could use the `[installdir]\TEST` folder to



save this project. Practice shows that it is convenient to name your model after the main source file you create with it, or the main set of components. In this case `INTLABEL.mpb` seems an obvious name.

## Adding the component to the VCL

### Debugging your component

A good practice, is to debug your new component before adding it to the VCL. Do this for example by adding the source file to the current Delphi project (e.g. by using Delphi's project manager) and re-compile the project. This should at least filter out all syntax errors. Either use 'Compile' or 'Syntax Check' since Delphi does not always correctly manage the file's date/time and modified status if you just 'Run' the project.

### Compiling errors

If you didn't make any mistakes, the unit should compile all right. If it doesn't: change the code in the appropriate place in ModelMaker. That is:

- For missing units in the unit's uses clause: the unit code editor.
- For any code not part of the class: the unit editor.
- For errors in the class's name or ancestor name: the Class view.
- For errors in the class's interface declaration: the Class Members view.
- For errors in a method's implementation: the Method Implementation view.

Switch to ModelMaker and fix the code. Use the integration expert's menu 'Jump to ModelMaker' to jump straight from Delphi's code editor to the corresponding position in ModelMaker. Finally, in the Unit list view click the button 'Generate' again and re-compile the Delphi project. If you are having trouble with this: look ahead where we are adding new behavior and editing code is explained.

## Adding the component to the VCL

After debugging your new file, add it to the Delphi's VCL.

In *Delphi 1.0*: select menu 'Options|Install Components', select 'Add' and browse to find the unit `INTLABEL.PAS` in folder `..\ModelMaker\6.0\TEST\`.

In *Delphi 2.0*: Select menu 'Component|Install', select 'Add' and browse to find the unit `INTLABEL.PAS`. Refer to your Delphi User guide for more information about installing components.

In *Delphi 3 and higher*: you must install the new component in a package. Select a new package called `MMtest` in the `..\ModelMaker\6.0\Test` folder. Please refer to your user guide for installing packages.

After recompiling the VCL the new `TIntLabel` component should be on the palette page where you registered it: *MM Demo*. To test it, add a `TIntLabel` component to a (new) form.

You can use the Object Inspector now to set the 'NumValue' property and see the caption change. But the component can be improved!

## Improving the component in ModelMaker

If you watch carefully, you'll see that a `TIntLabel` when dropped on a form, initially has the caption 'IntLabel1' rather than '0'. The `NumValue` however is 0. This is conflicting and not very nice. To improve the component we'll have to override the constructor `Create` like this: (refer to Delphi's on-line help for the `TControl.ControlStyle` property)

```
constructor TIntLabel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Don't let the Object Inspector set a caption }
  ControlStyle := ControlStyle - [csSetCaption];
  { Instead pre-set the Caption ourselves }
  NumValue := 0;
end;
```

To do this we need to return to ModelMaker.

## Keep editing your code in ModelMaker

We could of course change `TIntLabel`'s code in Delphi, but then the ModelMaker model and the modified source file would be out of sync. The next time we would (re-)generate the file from within ModelMaker, the changes made in the Delphi Editor will be lost. Of course, if we do not intend to maintain our code any longer in ModelMaker that's fine, but we won't benefit the advantages ModelMaker offers during maintenance and documentation. And although it may seem a burden at first, after getting used to it, the benefits of keeping the master code in ModelMaker are much higher than the costs. So resist the itch in your fingers and return to ModelMaker now.

## Overriding the constructor Create

To override the constructor `Create`, we could add a method in the Class Members view clicking 'Add method', name it 'Create' adjust it's other attributes such as parameters 'AOwner: TComponent', method kind 'constructor', etc. but overriding methods can be done far more easy. The only thing is: in order to override a method (or property) the method to be overridden must exist in the model. If you used the default project template as was shipped with ModelMaker (which also contains the class `TComponent`), the virtual constructor `TComponent.Create` is in your model with the correct attributes, ready to be overridden.

To do so:

1. In the Members view tool bar click the 'Wizards' button, and

2. Select 'Method override wizard'. Alternatively use the same function from the Wizards popup sub menu.
3. In the 'Override Methods' dialog select the 'Create' method.
4. Make sure the option 'Call inherited method' is checked. This will instruct ModelMaker to add a section of code containing a call to the inherited method.
5. Click OK.

In the Class Members view we'll see that a method called `Create` is added to the list. Just for your information you may check the methods attributes by selecting it in the member list and clicking the 'Edit Member' button, or double clicking it in the member list.

Notice that all relevant attributes are copied from `TComponent.Create`:

- The methods name is `Create`.
- The parameter list is `AOwner: TComponent`.
- The method is 'public'.
- The data type is 'void'.
- The method type is 'constructor'.
- The binding kind is 'override' (since `TComponent.Create` is virtual).
- The option 'Call inherited' is checked because we checked the option 'Call inherited method' running the override wizard.
- The option 'Inheritance restricted' is checked. If this option is checked, the method will automatically be updated to reflect any changes applied to the overridden method in the ancestor class.

Click Cancel to leave the method in it's original state.

## Implementing Create, non-user sections in code

To implement the method `Create` switch to the Method Implementation view again. Notice that in the section list on the left, already one section is added containing the code:

```
inherited Create(AOwner);
```

This section is marked with a red line, indicating that we cannot edit it's contents. The section was added because the method's option 'Call inherited' is checked.

To add the other lines of code,

1. Add a new section by clicking the 'Add section' button.
2. Enter the code in the code editor on the right.
3. Click the 'Save code' button

In the section list on the left we'll see the complete implementation of Create.

Section to call inherited method. automatically added and updated.

```
constructor TIntLabel.Create(AOwner: TComponent);  
begin
```

```
  inherited Create(AOwner);
```

Section in which you enter your code. You must create and update this section yourself.

```
  { Don't let the Object Inspector set a caption }  
  ControlStyle := ControlStyle - [csSetCaption];  
  { Instead preset the Caption ourselves }  
  NumValue := 0;  
end;
```

## Instant code generation

If we have a look in the Delphi editor, we'll see that the source file has not been updated yet. To do this we need to regenerate the unit. Therefore switch to the Unit list view again.

Regenerating the unit could be done by clicking the 'Generate' button again, but it is more instructive to demonstrate ModelMaker's *instant code generation* feature. Rather than having to manually regenerate a source file whenever something has changed, it is possible to 'Enable Auto generation' for a unit. The source file will then be regenerated each time anything changes in the Model that affects the source file. It is a nice feature that ModelMaker not only regenerates the source file, but also instructs Delphi to reload the file if it's opened in Delphi's code editor. Refer to Integration with Delphi.

To watch this:

1. Make sure you have the INTLABEL.PAS file loaded and is on top in the Delphi code editor.
2. Click the 'Enable auto generation' unit button in the Unit view tool bar.
3. See how the Delphi editor now reflects the last changes in your file.

Now return to ModelMaker again, and let's play around:

1. Switch to the Class Members view.
2. Edit the Create Method (Double click or click the Edit button).
3. Now uncheck the 'Call inherited' option and click OK.
4. Watch the code being updated in Delphi.
5. Now check the 'Call inherited' option again.

To have a closer look at the Method Code editor,

1. In the Create method's section list you can drag sections up and down, do this and see how Delphi's code editor follows your changes.

Lets have play around with the units view:

1. Make sure the units view is in 'display as tree mode' (popup menu)
2. Select the class `TIntLabel` by clicking it.
3. Press the Del key once, this will remove the class from the unit. The class is still in the code model. In fact, the class is listed under the 'classes not assigned to units' node. Check the code in the IDE: you'll see that the entire interface and declaration have been removed.
4. Drag the class on the `IntLabel` unit. This will add the class to the unit again. Note that the VCL Component registration page is now reset to 'none'. Edit the unit (double click unit or use toolbar) to change this back to 'MM Demo'.

After you have played around with the instant code generation feature, make sure the `TIntLabel` class still is as you want it to be. In order to actually see the improved behavior,

1. Rebuild your VCL in Delphi.
  - Delphi 1:* menu 'Options|Rebuild library'.
  - Delphi 2:* menu 'Component|Rebuild library'.
  - Delphi 3 and higher:* recompile the package `MMtest.dpk`.
2. Remove any old `TIntLabel` components from forms.
3. Add a new `TIntLabel` to a form and notice how the `Caption` is set to '0' now.

## Documenting your component

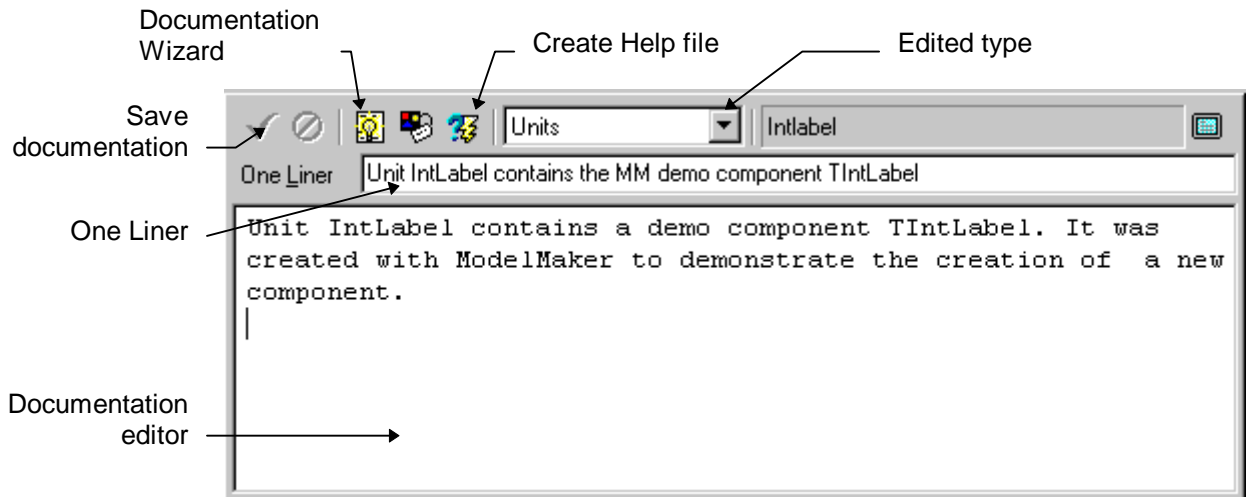
ModelMaker not only supports source code generation for your component, but it has also advanced wizards and generators to document your component. These include

- Documentation wizard which inserts basic standard documentation for all members in a class.
- In source documentation.
- Help file generation.
- Instant visualizing in class diagrams. Although creating diagrams is usually done in the design process, it is also possible to create diagrams from existing code.

To demonstrate these features we will now create a help file and a class diagram for the `TIntLabel` component.

### Adding documentation to your component

Each unit, class, all members of a class, event types and symbols in diagrams can be documented with a short description named One Liner and a longer text named Documentation. For editing One Liner and documentation we'll use the Documentation view. Alternatively we could have used the floating documentation window which is available from the main menu "Views".



To make this view visible: Select menu 'View|Documentation Editor'.

With this Documentation editor you add a One Liner and a more descriptive text to each unit, class and member (method, property etc.). That can be quite a job, so ModelMaker includes a documentation wizard, which does some of the nasty work for you. This wizard will insert pieces of documentation in the currently selected class.

To demonstrate this:

1. Select the class `TIntLabel` in the Classes or Units view.
2. In the Documentation view click the button 'Documentation Wizard'.
3. Click OK to confirm creation of standard documentation.
4. In the drop-down box 'Edited type' select 'class members'.
5. In the Class Members view select the `GetNumValue` method.

What you see now in the documentation editor is that the wizard inserted text like:

*GetNumValue is the read access method for the NumValue property.*

Usually this is sufficient to document `GetNumValue`, since you will be documenting the exact meaning of the property `NumValue` and this avoids redundancy.

Selecting the method `SetNumValue` in the Class Members view makes the documentation for `SetNumValue` visible:

*SetNumValue is the write access method of the NumValue property.*

Again this is usually sufficient to document the `SetNumValue` method.

What remains to be done is documenting the constructor `Create` and the property `NumValue`. But here too, the wizard inserted already some useful text.

Select the documentation for the constructor `Create` and change this to:

*Constructor Create overrides the inherited Create. First inherited Create is called, then the Caption is pre-set to 0, reflecting the initial NumValue state. ControlStyle is modified to exclude csSetCaption.*

Now select the `NumValue` property's documentation and change this to:

*Property NumValue is read/write at run time and design time.*

*It reads and writes the Caption property as an Integer.*

To edit the documentation for the class `TIntLabel`:

1. Make sure class `TIntLabel` is selected in the Classes or Units view,
2. In the Documentation view, select 'classes' in the drop-down box 'Edited type'.

Enter the text:

*TIntLabel is a simple TLabel descendant created with ModelMaker.  
It adds the property NumValue which reads and writes the Caption property  
as an Integer.*

To edit the documentation for unit `IntLabel` which contains the `TIntLabel` class:

1. Make sure the unit `IntLabel` is selected in the Units view.
2. In the Documentation view, select 'units' in the drop-down box 'Edited type'.

Enter the text.

*Unit IntLabel contains a demo component TIntLabel. It was created with  
ModelMaker to demonstrate the creation of a new component.*

The unit `IntLabel` is now completely documented. Documentation is typically used to create a helpfile or for in-source documentation. Third party plug-in experts use the ModelMaker `ToolsApi` to output documentation to other formats.

You add One Liners (short, single line descriptions) the same way. In the Views menu you'll find a 'Floating documentation' view. This view can be used to insert One Liners and documentation too. This view can be docked or stay floating. The edited entity type in this view is automatically updated to reflect the last focused view in ModelMaker.

## Creating a help file

ModelMaker can create a Borland style help file from your documentation. This includes the generation of the Borland `/B` keywords which are necessary for interaction with Delphi's on-line context sensitive Help. Help files are generated from unit's. In our example we'll create a help file for unit `INTLABEL`.

ModelMaker let's you select the visibilities you want to include in your help file. These are:

- 'User' (public, published, automated and default)
- 'Component writer' (user visibilities plus protected)
- 'Developer' (all visibilities)

The default visibility 'User' is the most restricted, since this will include only help for the public, published or automated interface of a class. Use this filter to create a help file you distribute with your components. Selecting the 'Component writer' visibilities will also include help for the protected interface. This is the type of help file you would typically distribute with components if other developers should be able to derive a descendant class from your component. The last selection includes also the private details (typically fields and or property access methods etc.) which you might want to have documented internally.

To create a help file for the `TIntLabel` component,

1. Make sure the unit `IntLabel` is selected in the Units view.
2. In the Documentation view, click the button 'Create help file'.
3. In the 'Create help file' dialog you'll be prompted to enter a file name for the unit's RTF file. A help project file with the same name and extension `.HPJ` will automatically be created. In our example enter `[installdir]\TEST\INTLABEL.RTF`.
4. In the same dialog, select the visibilities you want to include in your help file. In this example we'll go for the 'Component writers' visibility.
5. Leave the reformat paragraphs option checked and Click OK.
6. Open the explorer (file manager) and notice that both `INTLABEL.RTF` and `INTLABEL.HPJ` have been created.
7. Run your Delphi help compiler with the `INTLABEL.HPJ` project. Be aware that you need to use the help compiler that was shipped with the Delphi version you want to create help for. For *Delphi 1.0*: Use `DELPHI\BIN\HC31.EXE` in a DOS box and run it from the folder your HPJ file is in. For example: `DELPHI\BIN\HC31.EXE INTLABEL.HPJ`  
For *Delphi 2, 3 and higher*: Use the `Delphi 2(or 3 / 4/5/6).0\HELP\TOOLS\HCW.EXE` to compile your project: double clicking the `INTLABEL.HPJ` file in the explorer should be enough.

The help files have been created now. You might want to have a look at them, using windows help. Double clicking the newly generated `INTLABEL.HLP` starts help.

The Help File Generator source is available as plug-in expert on request. It can be extended to create help for multiple units at once etc.

## Integrating your help files with Delphi's on line help

To be able to invoke Delphi's help on your `TIntLabel` component you must integrate your `INTLABEL.HLP` help file with the Delphi 1 and 2 on line help. To do this:

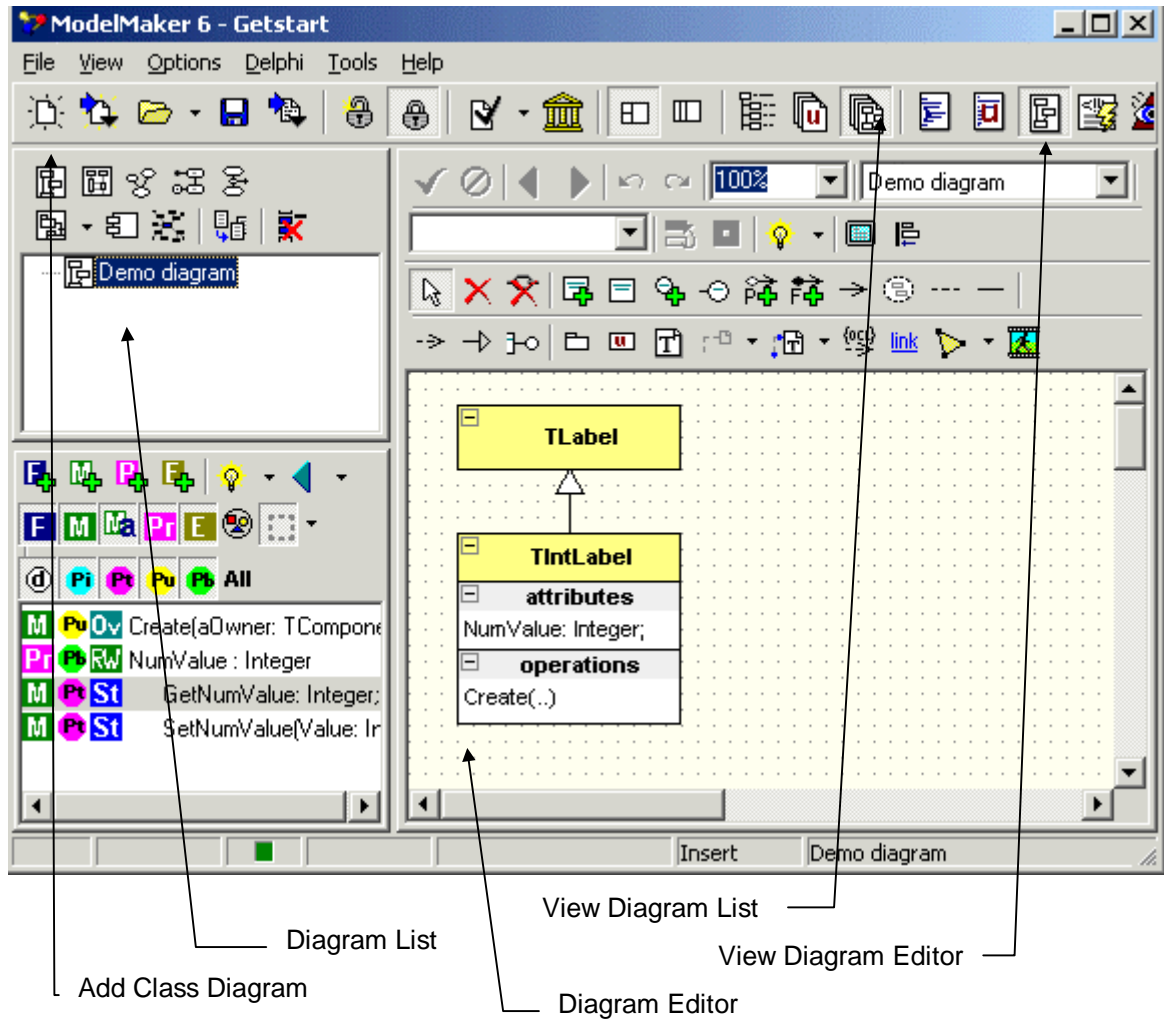
1. Generate key words using `KWGEN.EXE`
2. Install help using `HELPINST.EXE`

Installing help in Delphi 3 and higher is documented in Delphi's Component Writers Guide / User's Guide.

## Documenting the design in a diagram

Although creating diagrams is usually done in the first stages of the design process, it is also possible to create diagrams from existing models using ModelMaker's instant visualization feature. A list of diagrams in the model is edited in the Diagram List view. The actual diagrams are edited in the Diagram Editor.





Make the Diagrams List view and Diagram Editor visible:

1. Select menu 'View|Diagrams'. This will make the diagram list visible in the top left window.
2. Select menu 'View|Diagram Editor'. This will make the diagram editor visible in the editor pane on the right
3. In the Diagram list create a new class diagram by clicking the 'Add Class diagram' button.
4. You can in place edit the diagram's name. Enter 'Demo diagram'. The name is intended only to distinguish different diagrams.

In the newly created diagram we'll demonstrate ModelMaker's instant visualization feature:

1. Make sure the Classes or Units view is visible (View|Classes or Units)
2. Drag the class TLabel1 from the Classes or Units view and drop it on the diagram editor.
3. Do the same with TIntLabel1.
4. Save the Class diagram by clicking the 'Save diagram' button.

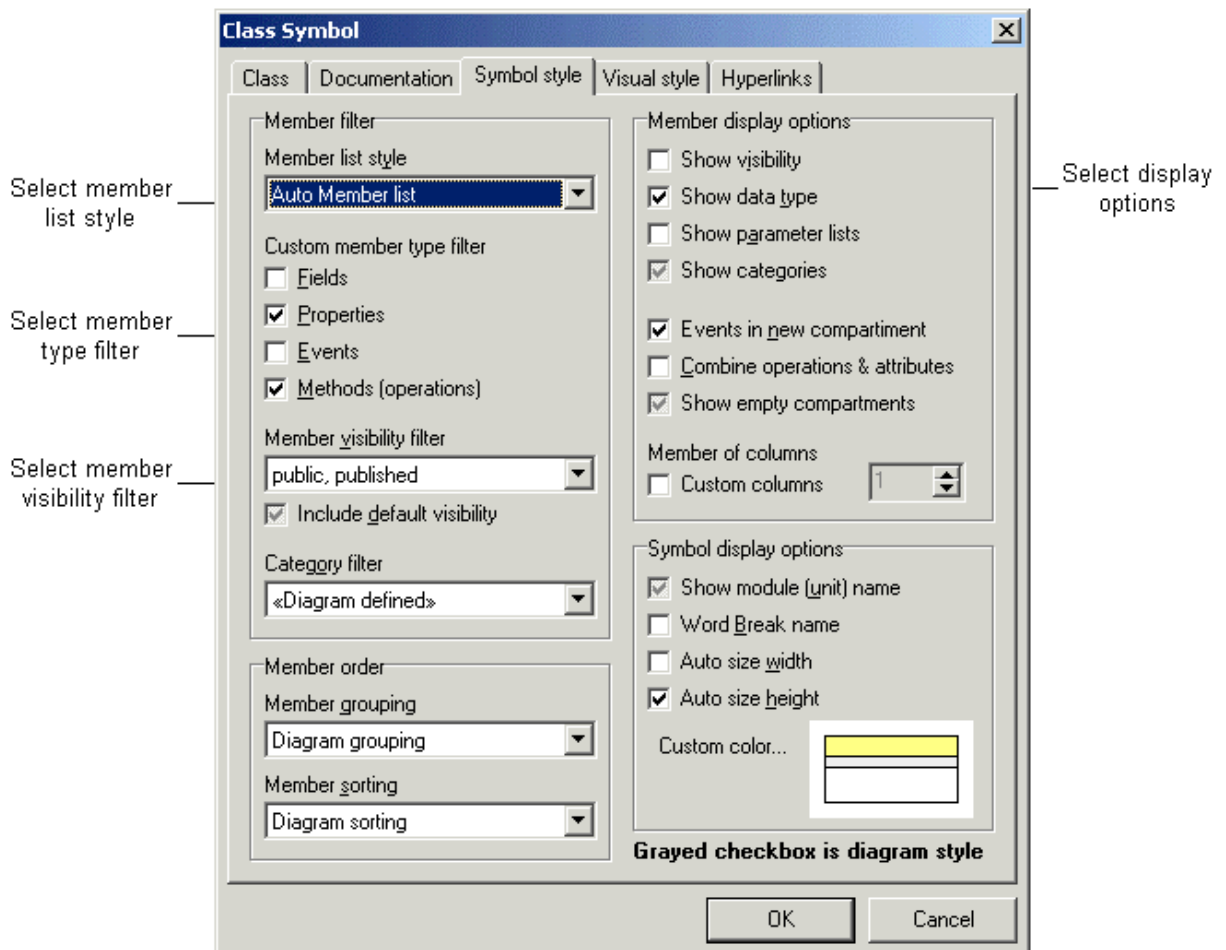
Notice how the inheritance relation `TIntLabel = class (TLabel)` is automatically visualized. If appropriate, ModelMaker will also visualize 'uses' relations if a class is dragged onto a diagram.

You can copy the current selected diagram to the clipboard (in WMF format) and paste it in your word processor to document your design. To do this: press `Shift+Ctrl+C` or use the local menu 'Export as Image'|'Clipboard'. Alternatively you could export the image to a file (bmp, wmf and jpg). To print the diagram, press `Ctrl+P` or use the pop-up menu.

## Symbol styles in Diagrams: displaying members

ModelMaker supports many UML styles of displaying classes and interfaces: show module (unit) names, show members, collapse interfaces etc. To demonstrate a few and give you an idea of what is possible we'll change the symbol style for the `TIntLabel` class symbol. Symbol styles define how a symbol is displayed in a diagram. Note: the visual appearance (colors, fonts etc) is controlled by the visual style which is not part of the symbol style; check this manual for details. Default all symbol styles are "as defined in current diagram". The diagram symbol style defaults to "as defined in the current project". This gives you the possibility to change style on any level you like: just a single class symbol, all class symbols in a diagram or all diagrams in a project.

Here we'll change the symbol for `TIntLabel` only. Double click on the `TIntLabel` class symbol. This will show the class symbol editor.



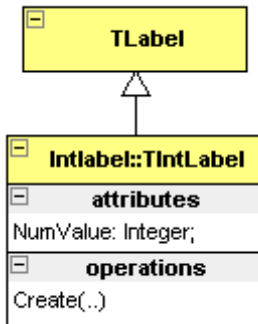
First you define *which* members are displayed. In this dialog change the Member list style from "Diagram Auto Member list" to "Auto Member list". This style will automatically add all members defined by the member filter in this editor. In the Member filter check

“Properties” and “Methods”. Change the visibility filter to public and published members only. This will suppress display of the protected property access methods GetNumValue and SetNumValue.

Then you define *how* members are displayed. Use the Member display options to modify this. Check “Show Data type” and “Events in new Compartment”. Note that a grayed option reverts the option to the parent (diagram) style.

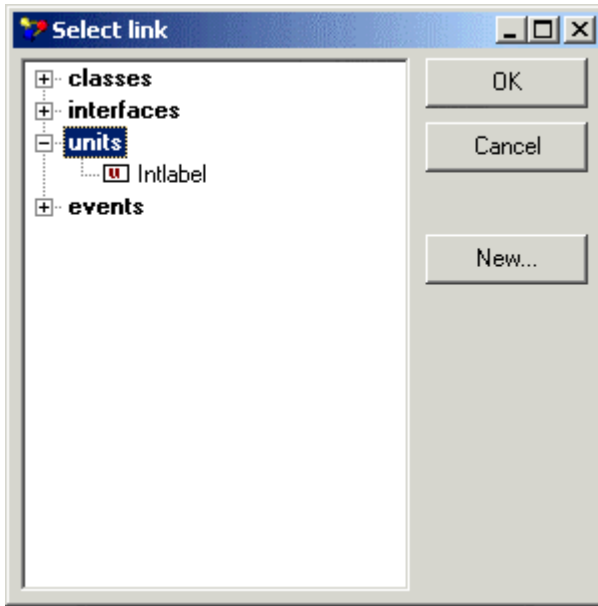
To display the name of the unit that contains this class in the symbol name compartment, check the “Show module name” option.

After clicking OK you should have a diagram that looks something like this (you can stretch a class symbol as needed depending on the Auto Size options – check the class symbol dialog). The property NumValue is displayed in the ‘attributes’ compartment and the method Create is displayed in the ‘operations’ compartment. Likewise events can be displayed in a separate compartment or combined with the attributes. If you wish you could even combine all members into a single compartment.

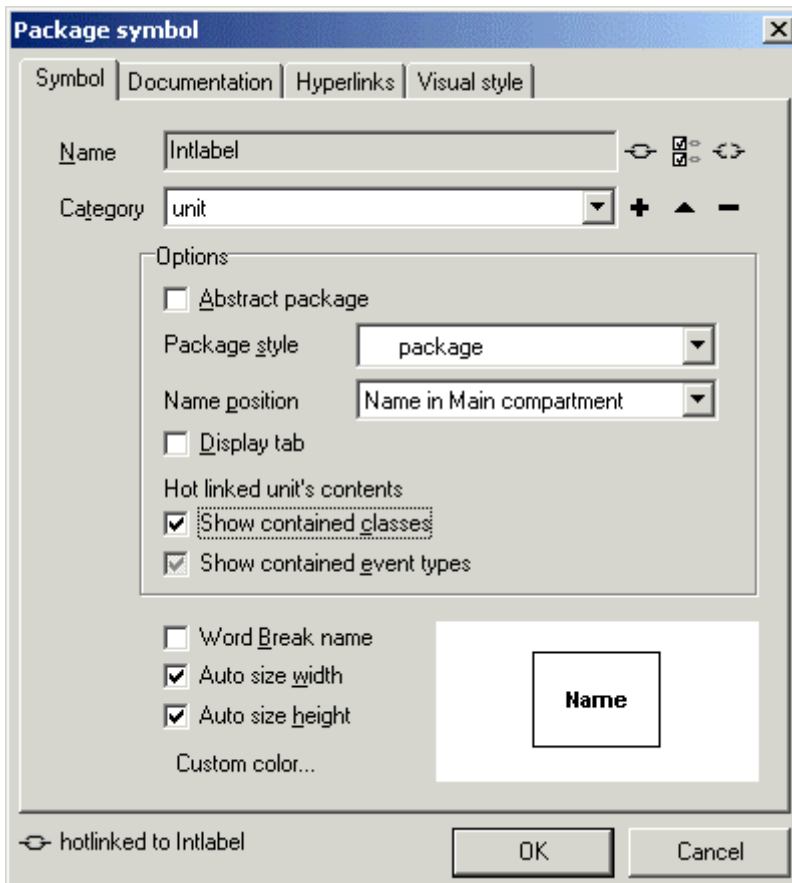


## Visualizing the unit IntLabel.pas

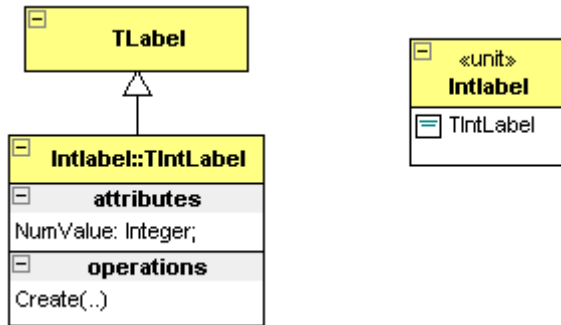
Just like classes can be visualized by class symbols, the unit IntLabel which contains TIntLabel can be visualized. To do this, click the “Add Unit Package” tool on the diagram editor tool bar and then click on the diagram. The following dialog will appear which lets you select the unit to visualize in the package. In this dialog, select “IntLabel” and click OK.



Now the Package Symbol editor will be visible, which is used to edit the visualization of the linked unit.



In this dialog, Make sure the “Show contained classes” option is checked (not grayed) and click OK. Your diagram should now look something like the picture below.

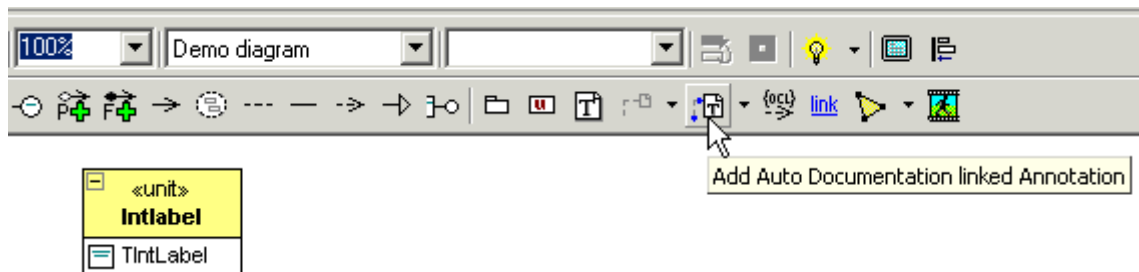


The package symbol displays its contained classes (TIntLabel) and has a stereotype (category) named <<unit>>. Of course, if you add more classes to the unit, the symbol will be updated automatically.

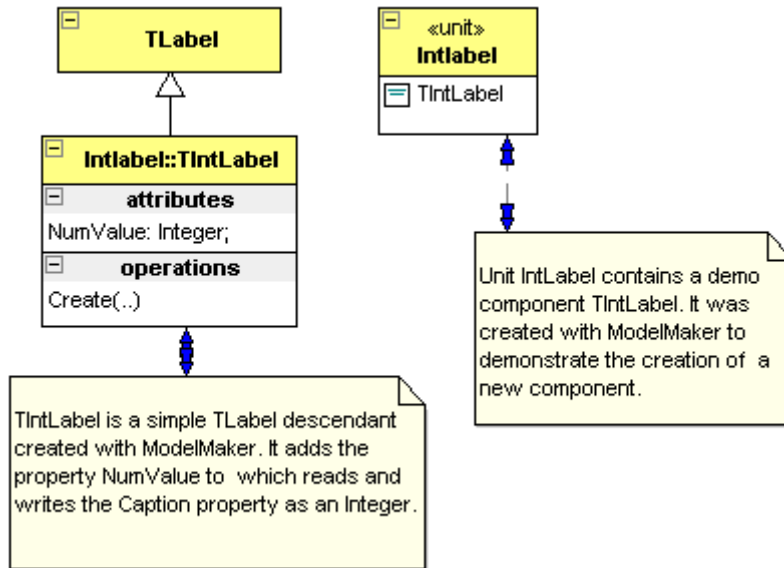
### Visualizing the Documentation

The UML uses Annotation symbols to add notes to diagrams. ModelMaker supports hotlinking annotations to symbol documentation (or OneLiners). This is a two way hot link: the annotation text will automatically show the symbol's documentation and editing the annotation text will update the symbol's documentation. To demonstrate this, we'll add a linked annotation to the class symbol TIntLabel and to the unit package symbol IntLabel.pas.

On the diagram editor toolbar select the "Add Auto Documentation linked Annotation" tool. This tool allows three link styles which can be selected with the drop down button next to it. Link styles are: passive (not linked), documentation and one liner. Make sure the Documentation style is selected.



Then after selecting this tool, click on the class symbol and drag the mouse below it. This will create the link and annotation. Repeat this for the package symbol that is linked to unit IntLabel. Your diagram will now look something like this.



Now try to edit the documentation for **TIntLabel** in the annotation and see it change in the class dialog and the diagram. To do this, click **TIntLabel**'s annotation and press F2. This invokes the annotation's inplace editor. Change the text to your liking and after pressing the Enter key, check the documentation tab in the class editor dialog (classes view). Similar to linking the documentation, a symbol's One Liner can be linked to an annotation.

## Summary

In this chapter you got a first glimpse of some basic features in ModelMaker.

- ModelMaker is all about creating classes.
- It is important to start with the right project template especially if you want to override methods or properties.
- The interface of a class consists of Class Members (fields, methods and properties).
- Properties create and update their access fields and methods.
- Method code consists of sections, which can either be created automatically or manually.
- A Unit provides a link with a source file. Units can have 'auto generation' enabled to instantly reflect any change in the model to the source file.
- ModelMaker's IDE integration experts will take care of reloading units in Delphi's code editor.
- To jump from ModelMaker to the IDE and back, use the **Locate in Delphi** and **Locate in ModelMaker** commands.
- It's easy to document a design using the documentation wizard.
- Help files can be created for a unit, which are ready to integrate with Delphi's on-line help.
- Instant visualization in class diagrams can be used to document your design.
- Class symbols can automatically display contained members, package symbols can automatically display classes contained by a unit (source module).

- Symbols can be hotlinked to annotations, which will then display documentation or One Liners.
- Diagrams support multiple symbol styles that can be defined at different levels.

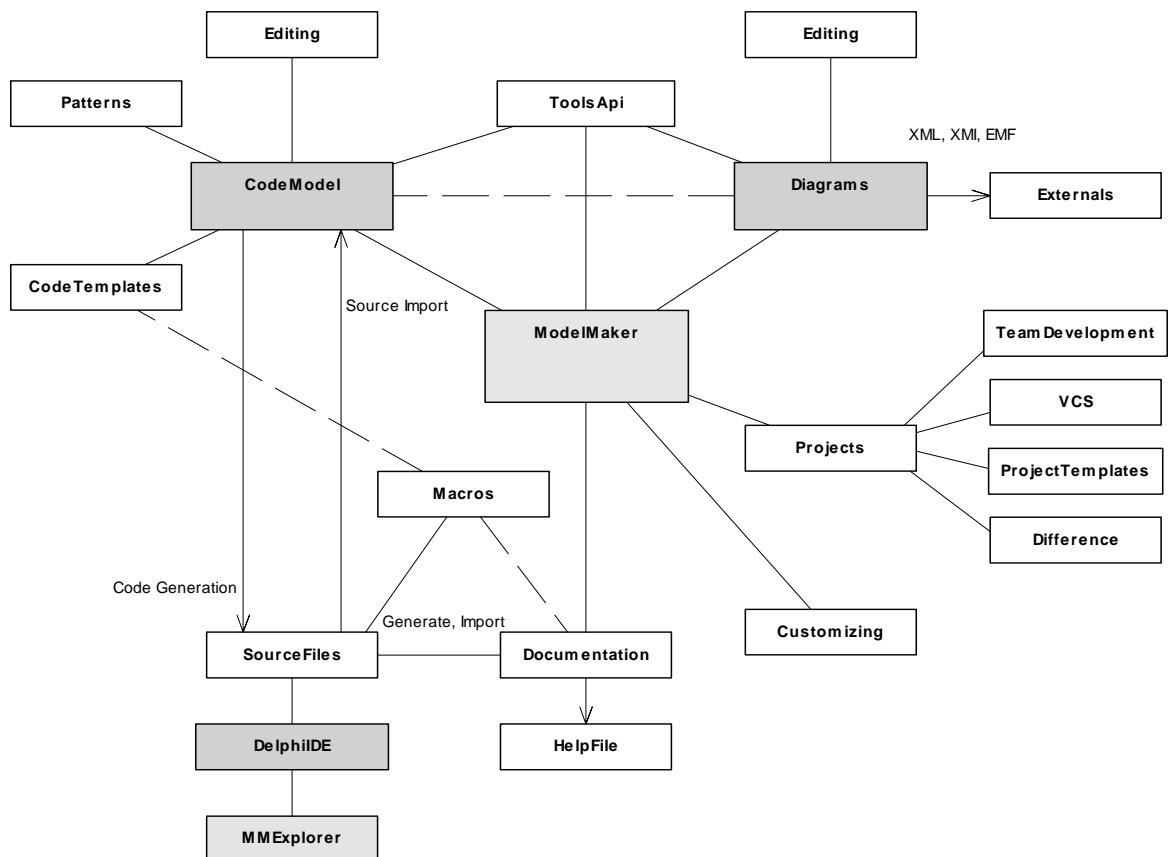
## Where to now?

Now you've seen ModelMaker's basic mechanisms. You can have a look at the following topics:

- There is another step-by-step example in the Design Patterns manual that demonstrates the use of design patterns. You should have enough background now to work through this demo.
- ModelMaker's *Basic Concepts* page 36.
- The other chapters in this manual focus in greater detail on common aspects of ModelMaker such as diagrams, code generation and import.

# Basic Concepts

## Overview



This picture gives an impression of ModelMaker's main parts and how they relate.

The two main entities are the Code Model and the Diagrams. Around them you'll find importing and generating source code, interaction with the Delphi IDE, saving models etc. Going around this picture more or less anti-clockwise we'll see:

The *Code Model* contains the classes, members, units etc. that map to the corresponding concepts in Delphi's Object Pascal. The Classes view, Members view, Method Implementation view and Units (Code) view all deal with visualizing and manipulating the code model directly.

*Code Generation* is the process of creating an Object Pascal source file containing classes, members and unit code. *Units* provide the link between the Code Model and a source file. A unit contains classes and/or event type declarations plus user defined *unit code* such as



module procedures. Refer to Code Generation, page 45, for a more detailed description of source code generation. Code generation is typically, but not only, controlled from the Units view.

**Code Import** is the process of reverse engineering an Object Pascal source file into entities that make up the *Code Model*. This can be initial importing - the unit and / or classes it contains did previously not exist in the Code Model - or **refresh import** - re-importing an existing unit and / or the classes it contains in order to synchronize the Code Model and source file. Code Import is typically controlled from the Units view. It is amongst others also available in the classes view and difference view. Code Import is described in detail in chapter Importing Source Code, page 56

**Design patterns** are proven solutions for a general design problem. It consists of communicating classes and objects that are customized to solve the problem in a particular context. In ModelMaker patterns are active agents that will insert code into the model and stay *alive* to *reflect changes* in the model to the pattern related code. Design patterns currently only relate to the *Code Model*. Patterns are manipulated in the Patterns view. There is more on patterns in the Design Patterns manual.

**Code Templates** are user definable and parameterizable snippets of related code. They are like user definable patterns. Code templates can be created from the Members view and applied from the Patterns view or Members view. There's more in Code Templates page 71

ModelMaker's **Delphi IDE integration** takes care of synchronizing the Delphi IDE editor buffers whenever a file is (re-)generated by ModelMaker. Depending on the Delphi version there are other functions available like: add IDE editor file to model, refresh IDE editor file etc. There's more in chapter Integration with the IDE page 105

The **ModelMaker Code Explorer** is a separate *ModelMaker Tools* product that brings basic ModelMaker Code Model related functionality into the Delphi IDE. With this explorer you can navigate and add, edit, copy properties, methods or even entire classes with the same ease and concepts as in ModelMaker, usually even with the same dialogs.

A **Macro** is a fragment of text that is identified by a macro identifier. While generating source code and in-source documentation, ModelMaker will expand the text, replacing macro identifiers with the macro's text. Macros are also used to customize certain parts of the generation process (custom class separator, method section separator etc.). Macros are maintained in the Macros view. Macros are described in detail in chapter Macros, page 74  
Macros are also used to parameterize *Code Templates* as described in chapter Code templates, page 71

All Code Model entities and Diagram symbols can be documented with **Documentation** and a **One Liner**. In all relevant editors you'll find a Documentation tab. The Documentation view and Floating Documentation view are dedicated to editing One Liners and Documentation. Emitting "in-source documentation" during code generation is controlled by macros. You can redefine these macros to customize the documentation format.

Documentation can be converted to a unit based *Help File*. This is done in the Documentation view. Other documentation output formats can be created with (third party) plug-in experts that use the *MMToolsApi* to access the model.

The *Diagrams* contains multiple types of diagrams. Some diagrams visualize aspects of the Code Model in UML-style. Others visualize entities such as Use Cases that only exist in Diagrams. Most symbols can be ‘HotLinked’ to entities in the code model. Class symbols for example are linked strictly to classes in the Code Model: changing the class in a diagram will also change the class in the code model. Messages in sequence diagrams can be weak linked: they can or cannot be linked to a class member. If they are not linked the message name is just text. Symbols such as Use Case symbols only exist in the Diagram model and have usually no relation with the code model. It is important to realize that symbols linked to the code model (for example Class symbols in class diagrams) only *visualize* an entity (class) in the code model. The same class can be visualized many times in multiple diagrams in different styles depending on the context. Diagrams are created and maintained in the Diagram list view. The actual diagrams are edited in the Diagram Editor view. There’s more in chapter Diagrams, page 39

Diagrams can be *exported* as image, native XML format or in XMI format (third party plug-in expert).

The Environment and Project options are a first means to *Customizing* ModelMaker to your taste or coding style.

In Chapter “Customizing” page 80 there’s more on customizing ModelMaker.

ModelMaker *Projects* or *Models* contain both Code Model and Diagrams. In ModelMaker you work on a single model at a time. Opening a different model will close the model you were working on.

Working with models requires care in the areas *team development* and model (system) *boundaries* as described in chapter Team development. Model boundaries and Version Control, page 43

The *Difference* view is used to compare a model unit to the associated file on disk. Most powerful is the structured difference that does a syntactical comparison rather than a plain file based comparison. Also use the Difference view to compare any disk file or model unit with any other file or model unit or to compare two classes.

## Code Model contents

A *Class* is the most important entity in the Code model, it matches the corresponding concept in Delphi and it is a container of Class Members. Classes always have an ancestor (super) class and sometimes have descendent (sub) classes. The default class ancestor `TObject` is always present in the model.

An **Interface** is similar to a class, as it matches the corresponding concept in Delphi. It is also a container of a restricted set of Class Members. Interfaces always have an ancestor (super) interface and sometimes have descendent (sub) interfaces. From Delphi 6 onwards there are two interface roots; IUnknown and IInterface. These default interface ancestors are always present in the model. Because classes and interfaces are so similar, in the remainder of this manual usually where you read class you can also read interface. Both are

**Class Members** are the fields, methods, properties and events making up a class's interface. They always belong to a class (or interface).

**Fields, Methods** and **Properties** match the corresponding concepts in Delphi. Fields are used to store a class's state and/or data. Methods are used to implement behavior. A method's implementation consists of **sections** of code. This allows ModelMaker to locate specific code within the method's body. Properties let you have controlled access to a class's attributes as though they were fields. **Events** are a special kind of properties. They are used to represent method pointer type properties (delegates). This way ModelMaker makes the same distinction as Delphi's Object Inspector does. It is possible to create a *property* of type TNotifyEvent and ModelMaker will generate the correct code for the property, but ModelMaker will not recognize this property as an event. Therefore: use Events rather than properties to model event types.

**Event type definitions** are used to define the signature of event type properties. ModelMaker relies on these definitions to create and update event handler methods and event dispatch methods. The most used event type TNotifyEvent is automatically inserted in each model. Event types are maintained in the Events view.

**Design patterns** and **Units** as described earlier complete the code model contents.

## Diagrams

ModelMaker supports a set of UML diagrams:

1. Class diagram or static structure diagram
2. Sequence diagram
3. Collaboration diagram
4. Use case diagram
5. Robustness analyses diagram (not defined in the UML)
6. Activity diagram
7. State chart diagram
8. Package diagram or Unit dependency diagram, a static structure diagram, just showing unit package symbols.
9. Implementation diagrams: Deployment diagram and Component Diagram
10. Mind Map diagram (not defined in the UML)

The basic elements of diagrams are **symbols** and **associations**. The meaning and attributes of the symbols and association used are according to the UML specification. This manual will not explain the meaning and details of each symbol. In the ModelMaker on-line help you'll

find a short description for each symbol and association and it's attributes. There are a number of good books available on the UML. Alternatively you could download the latest version of the UML specification as available for free on the Rational web site.

The Chapter "Diagrams" contains a detailed description of organization and editing of diagrams in ModelMaker.

# Working with models

In this chapter, a model is the equivalent of a ModelMaker project that contains both Code Model and Diagrams.

## Model files

Native ModelMaker will save a model into set of files:

1. <model>.mpr; contains the project settings.
2. <model>.mma; contains the project related macros.
3. <model>.mmb; contains the code model data
4. <model>.mmc; contains the documentation for the model.
5. <model>.mmd; contains the diagrams.
6. <model>.mme; contains the event type definitions.
7. <model>.mmf; contains the project messages.

If you manage your source files using a version control system, you should add these model files to version control too.

However, ModelMaker is able to bundle the project files into a single project bundle: \*.mpb. To enable this, check the option 'Bundle project files' in "Options|Environment|General". This option is checked by default. If this option is checked, the file Open and Save dialogs will have the \*.mpb file type as well as the \*.mpr. Using single file bundles makes it easier to work with ModelMaker projects. To convert existing projects to single file project bundles, use File|Save as and manually change the .mpr extension to an .mpb for the project. In very large projects you may find that saving and loading bundles takes some more time than the multi file projects. When using project bundles it is sufficient to add the <model>.mpb file to version control.

ModelMaker cooperates with version control systems by not allowing you to save a project that exists with read-only file attributes. Note that only the file <model>.mpr or .mpb is checked for read only attributes. In an unbundled project file the \*.mma.\*.mme files are not checked.

## Model templates

As you probably have noticed (for example in the Getting Started demo), one of ModelMaker's powerful features is that it's easy to override methods and properties and that changes are automatically propagated down the inheritance tree. But to let this work, the

ancestor class and the methods and properties to override must be part of the model. So it is important with which (new) model you start.

Now you may ask: why didn't I get the complete VCL as default model? This would contain all classes I ever need! The answer is that this would result in very large models, through which it is hard to navigate. We have been working with ModelMaker for quite a few years now, and it shows that it is most practical if models contain classes of a single domain only. Classes contained in a single component package typically reside in single model too. Generally this result in models typically containing 5 to 20 classes or may be up to 50 for really large models.

And that's where you need templates. Templates are just ordinary models containing classes for a certain domain that you use to derive new classes with in a certain domain. Since it's possible to have as many templates as you like, you can create nice compact templates for each relevant sub-domain: like a template for creating simple components, one for simple TCustomPanel descendants etc.

You load a template by selecting 'File|New from template' which will load the template model and then reset the model's name to 'untitled'. Any model can be used as a template, but by design ModelMaker looks in the `[installdir]\TEMPLATE` folder for template models.

ModelMaker has one special template that it uses as default. This is the model `[installdir]\DEFAULT.mpb`. You may open this model and change it to your needs or overwrite it with another template model.

## Editing a model

To edit a model, ModelMaker has multiple views on the model. Most of these views are interlinked: selecting something in one view will show related information in another. Interlinking is based on:

- The current *class*, selected in the Classes view or Diagrams view. For example, the Class Members view displays the members of the current class.
- The current *class member* (if any), selected in the Class Members view.
- The current *method* (if any), equal to the current class member if that is a method. For example, the Method Implementation view displays the implementation of the current method.
- The current *unit*, selected from the Units view.

The other view like the Macros view and the Event Library view are (more or less) independent from these selections.

In the Environment options Navigation tab you'll find options to synchronize and activate views on certain events.

## Ownership in ModelMaker

ModelMaker assigns an *owner* to each entity in the model. Entities can be anything from classes to methods or a section of code in a method. The owner of an entity created the entity and has exclusive rights to update or delete it whenever suitable. Usually you, 'User' will be the owner since you create most classes, class members etc. ModelMaker does not discriminate between users: it does not remember that John created this class and Mary that property.

Deleting an entity will automatically delete all entities it owns too. For example: a property owns its read access method. You cannot delete or edit these methods, other than by deleting or editing the property.

## Team development, Model boundaries and Version Control

In ModelMaker team development support and model boundaries are related issues as they both deal with the question: what should and what should not be in a model. There are a few important reasons to use multiple relative small models according to logical boundaries rather than one big model containing all classes and diagrams you have.

1. It enables team development: while one developer works on one model that is part of a larger project, the other can work on another. There are limited possibilities to merge changes made to the same model. Therefore only one developer can work on a single model at the time.
2. It improved ease of navigation and overview
3. It improves performance.

Usually you'll find logical boundaries to split up models - and usually well-designed modules (as in units or classes) have low coupling and dependencies. Boundaries could be: units in a certain Delphi package, units containing classes that perform a related task etc. In ModelMaker itself for example, we have lots of models, sometimes only containing a single unit. Large models for example contain all classes related to diagrams, the entire code engine or the source importer. And yet other models contain units that are used in many projects: timers, filters etc.

Although we have some happy customers that have models containing 300+ classes in 150+ units, practice shows that a good model size is about 1 to 30 classes in 1..10 units.

The drawback on having multiple smaller models is that ModelMaker maintains active relations only within the same model and not beyond model boundaries. As a result you occasionally might need to re-import a unit after it has been altered in another model.

It's a good idea to use a version control system and put the model files under version control too. Although a model contains everything that is needed to (re-)generate the source files it

contains, you should *always* store the actual source files under version control too. It's from the source files that you build your product, not the model.

Merging models is only partly supported: source code can be generated and imported and diagrams can be exported/imported. However no code model meta-information can be exported/imported. Because ModelMaker stores its data in a native binary format, you cannot use the merging capabilities of a Version Control System without corrupting the model.

If models do get out of sync with the source code or other models you've always got the Difference View with it's powerful structural difference function to help getting the model synchronized with source or other models.

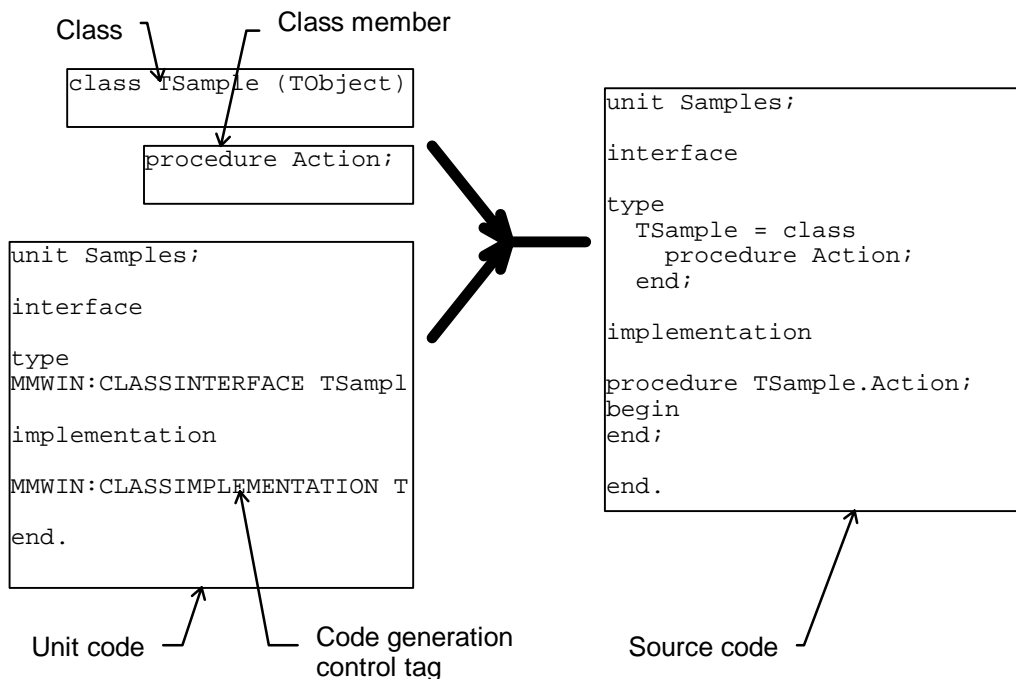
Using source aliases rather than hard coded directories is a must in team development. Check chapter Source Aliases, page 52 on source aliases.



# Generation source code

## Overview

In ModelMaker units are the gateways to source files on disk. During code generation, unit code, classes and class members from the code model are combined into a source file. A unit's unit code contains (can contain) code generation control tags. At the position of a tag ModelMaker will insert the associated entity such as class interface or implementation. Here



is a picture that visualizes this process.

The unit code is read line-by-line and scanned for code generation control tags. If a line contains a code generation control tag, the entity as defined by that tag is inserted instead of the tag. Any lines not containing code tags are just copied to the source file.

During code generation macros in both unit code and method implementation code will be expanded using the predefined macros and the project and environment macros you define yourself. Macro expansion and line formatting are the last stages in the source code generation process for both text generated from code tags and text just copied from a unit's unit code. Macros are explained in detail in chapter Macros, page 74

During generation of the implementation section, class separators, method separators and method section separators can be emitted. This is controlled by code generation settings and macros.

Insertion of in-source documentation can also be part of the generation. This is explained in detail in chapter “In source Documentation”, page 67.

## Code generation control tags

ModelMaker uses code generation control tags to control source file generation. Only code generation control tags placed in the unit code are interpreted. Tags in a method’s implementation are not interpreted. Here are the rules that apply to code generation control tags:

1. Code generation control tags are *case insensitive*.
2. All code generation control tags start with `MMWIN:` at the *first* position of a line.
3. Code generation control tags must reside on a *single line*.
4. Any *semi-colons* or *equal signs* defined in a tag are obligatory.
5. Code generation control tags can contain any white space after the `MMWIN:` definition. For example: `MMWIN: STARTINTERFACE` is the same as tag `MMWIN: START INTERFACE`

## Class related tags

These code generation control tags are used to define the insertion position of class and interface related code:

```
MMWIN:CLASS INTERFACE classname ;ID=###;
MMWIN:CLASS IMPLEMENTATION classname ;ID=###;
MMWIN:CLASS REGISTRATION classname ;ID=###;PAGE=vcl page name
MMWIN:CLASS INITIALIZATION classname;ID=###; // OBSOLETE
```

These class related tags are automatically inserted and maintained by ModelMaker whenever you add a class to a unit or remove it again. Normally you would use the Unit editor dialog or drag and drop in the Units view to insert or delete classes in/from a unit or change the relative position within a unit. However, in special cases you can manually move these tags to any other position in the unit code. This is for example useful if you want the interface of a class to reside in the unit’s implementation.

In these tags ModelMaker ignores the ‘`classname`’ and just uses the ‘`ID=###`’ tag to identify the class. The class name is inserted just for your convenience. Modifying it will have no effect.

Normally, for classes both the `CLASS INTERFACE` and `CLASS IMPLEMENTATION` tags should be put in the unit code. If either one is missing after you’ve edited the unit code manually, you’ll get a warning. In special cases - for example in include or documentation files - you may manually remove either the interface or implementation tag. Interfaces (as opposed to classes) ignore the `IMPLEMENTATION` tag.

The `CLASS REGISTRATION` tag is optional and is used to insert a snippet of code to register the class as component. You may manually remove them from the unit code. You can manually

edit the 'vcl page name=...' text in the registration tag to change the VCL registration page. However, usually you would do this in the unit editor dialog. If you remove the registration tag or the page name, no registration code will be generated for the class.

When importing a unit containing a procedure Register, the registration code is automatically converted to tags.

The tag `CLASS_INITIALIZATION` is obsolete from version MMv6.0 onwards. This tag is maintained for backward compatibility only. The tag is used to insert initialization code for `TStreamable` descendants: `RegisterStreamable(...)`; If you remove the initialization tag, no initialization code will be generated for the class.

## Event type declaration tag

This code generation control tag is used to define the insertion position of an event type declaration.

```
MMWIN: EVENT DEFINITION eventname type declaration; ID=###;
```

This declaration is maintained by ModelMaker and is obligatory for each event definition in a unit. Normally you use the unit editor dialog or drag and drop from the Events and Units view to insert or remove event type definitions in/from a unit, or change their relative positions within a unit. However, you may manually move these tags to any other position. If you remove them, the event type definition is also removed from the unit. The 'eventname' and 'type declaration' texts are ignored, only 'ID=###' is used to identify an event type definition.

## Editing marker tags

These code generation control tags are used to mark the positions at which you want ModelMaker to insert the first class or event type declaration in a unit.

```
MMWIN: STARTINTERFACE  
MMWIN: STARTIMPLEMENTATION
```

These tags are for editing purposes only and they have no role in the code generation process. There's one exception to this. The `MMWIN: STARTINTERFACE` tag is also used to determine the insertion position of class forward declarations - if any. If this `MMWIN: STARTINTERFACE` tag is absent, class forward declarations will be inserted before the first event type declaration or class interface, whichever comes first.

## Macro expansion control tags

These code generation control tags are used to switch on and off macro expansion during code generation:

```
MMWIN: START EXPAND  
MMWIN: END EXPAND
```

By default the expansion is switched ON. You need these tags if your unit or method code contains the text “<!” (“” not included). The macro expander will interpret the sequences

```
"<!" + Identifier + "!>" on a single line  
"<!" + Identifier + "(" param list + ")" + "!>"
```

as a macro. “Identifier” (the macro name) can consist of characters [‘0’..‘9’, ‘a’..‘z’, ‘A’..‘Z’, ‘\_’]. Which is similar to Object Pascal identifiers, although macro names can start with a number. White space surrounding the identifier is ignored.

Because macros can be in any text including comments and strings, this would make it impossible to generate code for units that contain a valid macro sequence <!ident!>.

There are a few workarounds for this problem. The most sensible uses these control tags in the unit code - remember the tags do not work in method code!

```
MMWIN: ENDEXPAND  
const  
  HTMLCommentStart = '<!';  
  HTMLCommentEnd = '!>';  
MMWIN: STARTEXPAND
```

In the rest of the code you can now use the constants and still leave the macro expansion on.

Note that these tags do not affect generation of in-source documentation, which is also based on macros. Documentation related macros are always expanded regardless of the setting of these switches.

## Unit documentation tag (obsolete)

This code generation control tags defines the insertion position of a unit’s documentation. Because inserting unit documentation is much better controlled with the ‘In-source documentation generation’ options, we recommend avoiding the use of this tag although it’s still supported for backward compatibility.

```
MMWIN: INCLUDE UNITDOC; INDENT=##;
```

The tag ‘INCLUDE UNIT DOC’ defines the position at which ModelMaker will insert the unit’s documentation. The unit’s documentation can be edited in the Documentation view. The ‘INDENT=##;’ extension is optional and may be omitted. This defines an indention for the documentation of ## spaces. A typical use would be:

```
{  
MMWIN: INCLUDE UNIT DOC; INDENT=2;  
}  
unit <!UnitName!>
```

## Obsolete tags

These code generation control tags are obsolete from MMv6.0 onwards.

```
MMWIN: STARTREGISTRATION  
MMWIN: STARTINITIALIZATION
```

On loading a model, ModelMaker will remove these tags from the unit code. If you add them manually they will simply be ignored.

## Code generation options

In the Project options|Code Generation tab you will find options that control code generation:

1. Formatting the layout of source code
2. Sorting of class members and method implementations
3. Inserting a (custom) class header to that precedes the class's method implementations
4. Inserting a (custom) method separator to that precedes each method's implementation
5. Inserting a (custom) method section divider in between each section.

The on-line help file explains the meaning of these options.

The Project options|Source Doc generation tab controls the generation of in-source documentation. This is explained in detail in the chapter on "In source Documentation", page 67.

## Maintaining Code Order / Custom member order

ModelMaker supports a user definable custom member interface and method implementation order.

These custom orders can be assigned during import used during code generation. When that is done, the effect is that ModelMaker will maintain the imported code order during generation.

This is how it works:

In the Project options Code Generation tab you define a member sorting scheme in class interface and implementation generation. In these sorting schemes Custom orders can be used as a grouping or additional sorting property.

When **grouped** on custom order, members will be sorted according to the (original) custom code order. Members that have been added later with an unspecified custom order, will be placed after all members with a specified order.

When using the custom order to perform **additional sorting**, the default sorting scheme will be applied and within each 'section' (visibility etc.), the custom order will be applied.

If class members have an unspecified order (which is the default for new members), the effect of enabling Custom Order during Generation or in the Members list is null.

Custom orders can be assigned during Import. In the Project options|Code Import tab you'll find settings to enable / disable assigning the custom order during import. The import dialog allows temporarily overruling these settings.

If the Importer assigns custom orders and Generation uses grouping on Custom Order ModelMaker will effectively maintain the original code order during 'refresh import' and append new members at the end.

Additionally Custom orders can manually be defined - per class - with the "Members custom order" dialog or with the Members view 'Rearrange mode'.

The Rearrange dialog is available from the Members and Classes view 'Wizards' local sub menus and from the Units view 'classes' local sub menu. In this dialog you either you drag and drop to rearrange a class interface and method order or use one of the predefined sorting schemes. The dialog can be also used to clear an interface or implementation custom order. Note that manually defining a custom order will erase an imported code order.

The Members view has a 'Rearrange mode' (members local menu). In this mode an interface custom order can be assigned using drag and drop in the members view itself. For method implementation order you must use the Rearrange dialog. In the in the rearrange mode the Members view filter settings (visibility, type, category) are ignored to make sure all members are displayed. Also Members view grouping is implicitly set to "Custom Order" and sorting is predefined and cannot be changed. As a visual feedback, the background of the members is changed to a silver color in this mode.

## Adjusting the unit template

Whenever you add a new unit to a model, ModelMaker looks for the file `DEFUNIT.PAS` in the folder `ModelMaker\6.0\BIN`. The default unit may also be (re-)defined when adding a new unit. This text file is used as a template for the newly created unit code. You may edit this file to your needs. You can use these code generation control tags to mark the insertion position for the first class ModelMaker will insert in the unit:

```
MMWIN:START INTERFACE
MMWIN:START IMPLEMENTATION
```

You might want to adjust the default unit template in order to:

- Customize the `uses` clauses in the unit `interface` and `implementation`.
- Add a company-defined header.
- Control macro expansion.

As an example: here's a unit template we use for freeware units. In the chapter Macros the same template extensively using macros is shown.

```
{
  File       : <!UnitName!>
  Version    : <!Version!>
  Comment    : <!Comment!>
  Date       : <!Date!>
  Time       : <!Time!>
  Author     : <!Author!>
  Compiler   : <!Compiler!>
```

```
+-----+
|  DISCLAIMER:
|  THIS SOURCE IS FREWARE. YOU ARE ALLOWED TO USE IT IN YOUR OWN PROJECTS
|  WITHOUT ANY RESTRICTIONS. YOU ARE NOT ALLOWED TO SELL THE SOURCE CODE.
|  THERE IS NO WARRANTY AT ALL - YOU USE IT ON YOUR OWN RISC. AUTHOR DOES
|  NOT ASSUME ANY RESPONSIBILITY FOR ANY DAMAGE OR ANY LOSS OF TIME OR MONEY
|  DUE THE USE OF ANY PART OF THIS SOURCE CODE.
+-----+
}

unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
MMWIN:START INTERFACE

procedure Register;

implementation

uses StrUtils, NumUtils;

procedure Register;
begin
end;

MMWIN:START IMPLEMENTATION

initialization
end.
```

Notice how this template contains (from top to bottom):

- A simple standard header and a free ware disclaimer.
- Some statistics macros, like <!Date!> and <!UnitName!>
- The basic unit structure:  
`unit..interface..uses..implementation..uses..initialization..end.`
- A macro <!UnitName!> to define the unit's name.
- The `procedure Register;` definition and implementation for registering components.
- A default uses clause in the unit's implementation to include some often-used units  
StrUtils and NumUtils.

## Unit Time Stamp Checking

The ModelMaker code generator by default uses Time Stamp checking to prevent overwriting source files that may have been changed outside ModelMaker. It checks if the file on disk is newer than the last time a unit was generated. If this is the case you'll be warned that you are about to overwrite that modified file.

You can switch on and off time stamp checking in the Environment options|General tab.

There is a limitation on time stamp checking you must be aware of:

1. If you rename a unit in ModelMaker the time stamp is not reset to 'unknown', so if you have an existing file on disk which is NEWER than the last time the unit was generated (with the old name) you will NOT get a warning. The Unit editor dialog warns you for this (file xxxx already exists, overwrite?), but the in place editor in the Units view does NOT.

The Unit difference View displays the time stamp comparison on activation. The function "Check Time stamps" refreshes this comparison.

## Source Aliases

ModelMaker supports source path aliases in units to avoid hard-coded directories and make your models machine-independent. An alias is associated with an aliased directory, similar to database aliases. In the model a unit's alias is saved rather than the aliased directory. On each machine aliased paths can be defined differently. This allows you to transport models to other machines.

Source aliases are a must in team development.

Example:

Suppose on machine A you have a source directory:

```
C:\DATA\PROJECTS\COMMON
```

An alias defining this directory could be COMMON.

On machine B, COMMON could be defined as

```
\\PROJECTDATA\COMMON
```

You add aliases from the main menu "Options|Source aliases", or from the Units view popup menu.

To avoid that you need to mimic a large directory structure in aliases, unit names can be relative to an alias. That way you only need to define a few aliases for root paths.

Here's an example:

Suppose you have a directory structure which looks like this:

```
C:\Project1\App_a\source  
C:\Project1\App_a\components  
C:\Project1\App_a\utils  
C:\Project1\App_b\source  
C:\Project1\App_b\components  
C:\Project1\App_b\utils  
C:\AllProjects\source  
C:\AllProjects\components  
etc.
```

You could define three aliases

```
App_a = C:\Project1\App_a
```



App\_b = C:\Project1\App\_b  
AllProjects = C:\AllProjects

A unit named C:\Project1\App\_a\source\Samples.pas could then use

Alias = "App\_a" (omit the "")  
Relative unit name = "source\Samples.pas" (omit the "")

Source code aliases are also used to offset the Import source code dialog's initial directory. If you define an alias VCL Source for C:\Program files\Borland\Delphi 3.0\Source\Vcl, the import dialog can be offset to this directory by simply selecting this alias from the drop down menu.

Source aliases also participate in Version Control integration. See next chapter.

## Version Control support and Aliases

By using a plug-in VCS Expert you can add Version Control capabilities to ModelMaker. Check the ModelMaker Tools web site for ready available third party VCS Experts or create your own using the MMTToolsApi VCS interface.

If a VCS expert is installed, in the units view popup menu and main file menu VCS related menu items are available to manually check-in/out a model or unit. Also each time a read-only unit is about to be generated an attempt is made to check the unit out (after your confirmation). VCS Experts can add more VCS related commands to the popup menu such as 'Add to project', 'History' etc.

Usually VCS systems usually need a VCS project name to perform an operation. In ModelMaker Source aliases are used to store the user definable VCS projects.

Each source alias can (but does not need to) store a Version Control project. This string is passed on to an installed VCS integration expert whenever a VCS file related action is performed.

If you install a VCS expert in ModelMaker, the ModelMaker IDE integration experts will use the same expert to integrate VCS in the Delphi IDE.

For details, refer to your VCS system and MM VCS expert provider.

## Using ModelMaker to generate Instrumentation code

ModelMaker supports generating Method instrumentation. This feature makes ModelMaker suited for generating instrumentation code for CodeSite, GpProfile and other tools instrumenting source code on a method base for profiling, tracing etc.

Method instrumentation generation is controlled with the option 'Instrumented' in the Method editor dialog. If this option is checked instrumentation code will be generated for the method. Instrumentation can be (de-) activated for all units at once with "Project options|Code generation|Instrumentation". The actual instrumentation code is defined by two macros you must add manually to the environment or project macro list: "MethodEnterInstrumentation" and "MethodExitInstrumentation" (omit the ""). These macros are expanded before the first and after the last section in a method's main body. For example (note the use of the predefined macros ClassName and MemberName in these macros)

```
macro MethodEnterInstrumentation=
```

```
CodeSite.EnterMethod( '<!ClassName!>.<!MemberName!>' );
try
```

```
macro MethodExitInstrumentation=
```

```
finally
    CodeSite.ExitMethod( '<!ClassName!>.<!MemberName!>' );
end;
```

When this is applied to:

```
procedure TSample.DoSomething;
begin
    ShowMessage('Doing something');
end;
```

This will result in the following code:

```
procedure TSample.DoSomething;
begin
    CodeSite.EnterMethod( 'TSample.DoSomething' );
    try
        ShowMessage('Doing something');
    finally
        CodeSite.ExitMethod( 'TSample.DoSomething' );
    end;
end;
```

To avoid creep when re-importing instrumented methods you can use the code remove tags. Check chapter "START and REMOVE tags, page 59 for details:

```
MMWIN:>>STARTREMOVE
MMWIN:>>ENDREMOVE
```

These tags may be part of a comment. Depending on the setting in the Project options|Code import tab the importer will filter out any code in between a start remove / end remove pair.

The macro MethodEnterInstrumentation using these tags would for example look like:

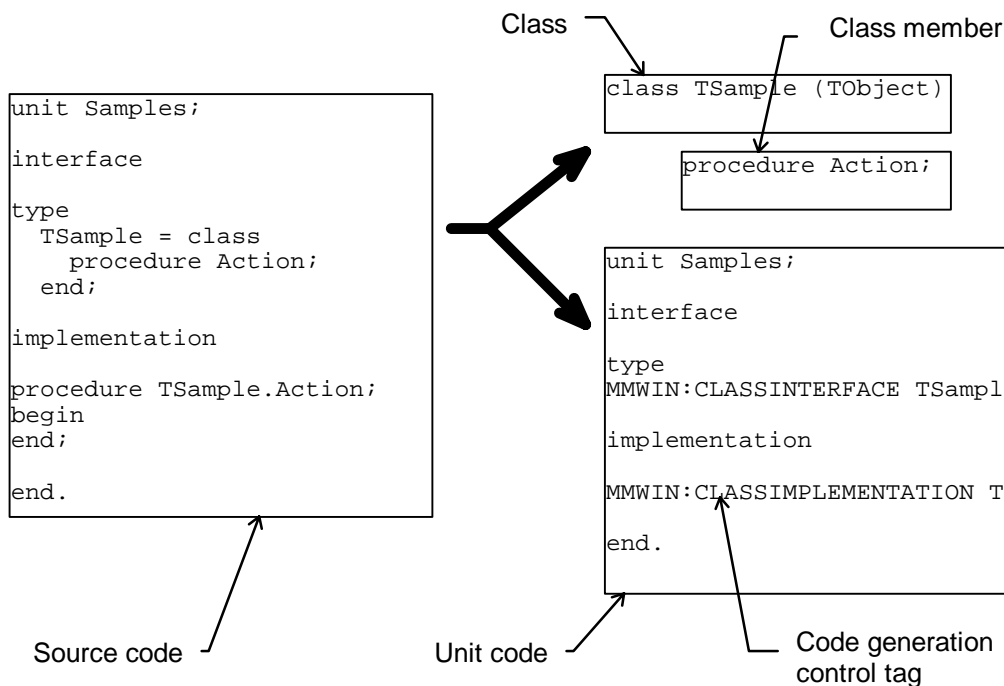
```
//MMWIN:>>STARTREMOVE  
CodeSite.EnterMethod( '<!ClassName!>.<!MemberName!>' );  
try  
//MMWIN:>>ENDREMOVE
```

The Member Manipulator can be used to switch on and off Instrumentation for multiple methods at once. On the ModelMaker Tools web site you'll find a third party plug-in Instrumentation expert. This expert is dedicated to controlling method instrumentation code.

# Importing source code

## Background

ModelMaker imports Delphi Object Pascal source files. This process is basically the inverse from generating a source file from classes, members and unit code. The class related code is converted into Code Model entities such as classes, interfaces, members and method implementations. All non-class or interface related code is moved into a ModelMaker unit's



unit code.

This process is called reverse engineering.

It is important to realize that if an imported class is not currently existing in the model, members and code sections are inserted as ‘User created/owned’ and no attempt is made to extract meta information, such as applied patterns, inherited calls etc. The only exception to this, are the read and write access members of properties, which are restored and linked to the property. For example: all code inserted by patterns is read back, but marked as ‘user’ and the patterns itself is not recreated. The same applies for inherited method calls etc. In general you lose the meta information.

However, if an imported class already occurs in the model it is ‘refreshed’ and all meta-information is restored. Even applied design patterns can be traced back.

Therefore importing source code is great for:

- Importing (an interface of) a class you want to inherit from or use as a client.
- Importing existing code originally not developed with ModelMaker
- Updating an existing class from source code

However it is advised that once imported, you keep editing your code in ModelMaker. The only exceptions to this are form and (other resource module's) source files which by their nature you (partially) need to edit in Delphi.

## Importing a source file

The main toolbar and units view pop-up menu contain buttons and commands to

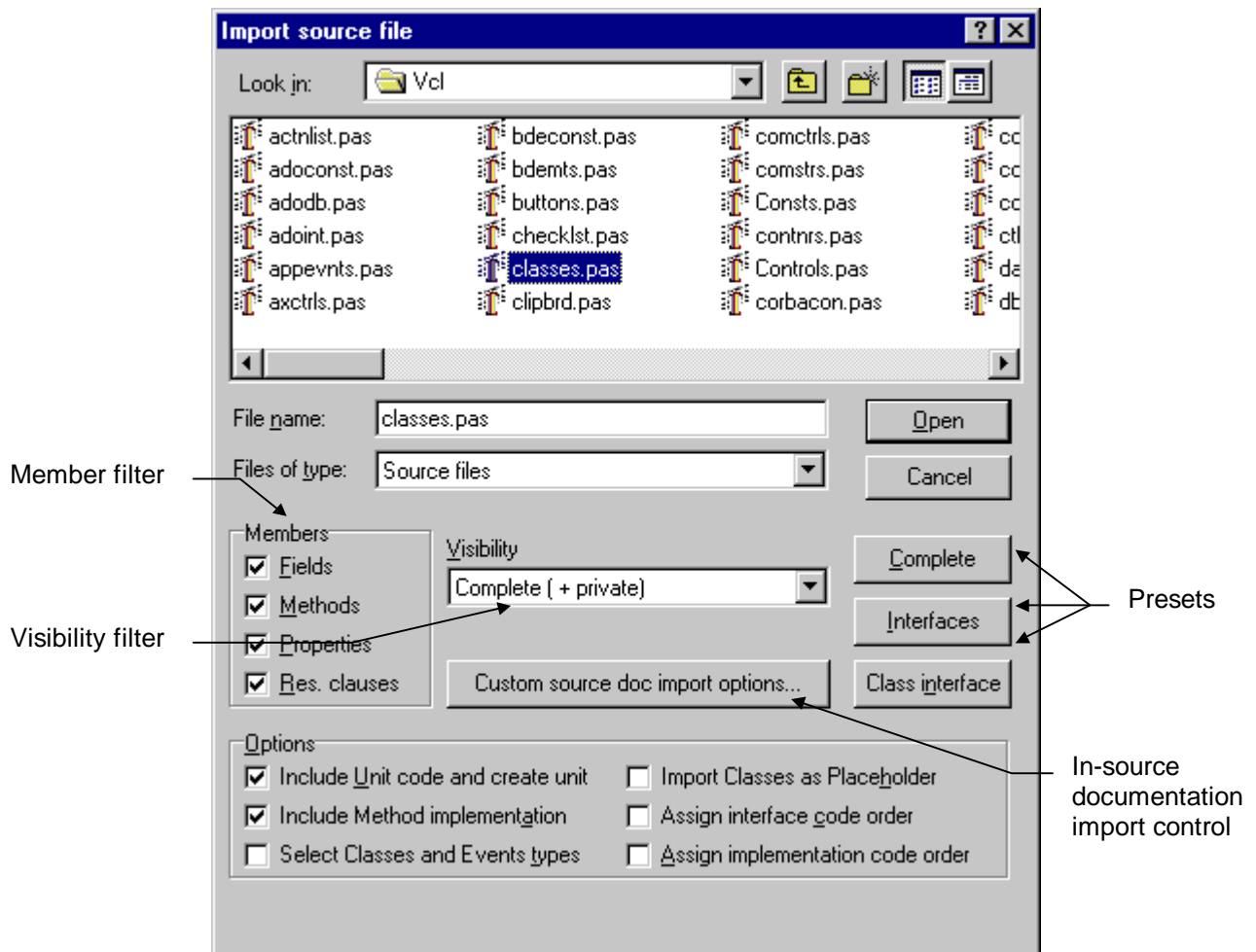
- Import a source file (in the current model or a new model).
- Refresh import source file.

Additionally you'll find import functions in the Classes view (refresh class or associated unit) and the Difference view (refresh unit, class or method).

In the Project options|Code Import tab you'll find the options to control source code import. Check the on-line help file for a detailed description on each option. The project options are used for all (refresh) imports except when you use the Import dialog to interactively import a file. The import dialog allows temporarily overruling some project import control options. The import dialog has some other options that by their nature are not in the project options, typically related to initial first time import. The options in the import dialog are preset (each time!) to the project options.

The import dialog's initial directory is pre-set by selecting a source code alias. You'll find more on (defining) Source code aliases on page 52.

When importing a source file using the import dialog, you enter a source file name and set filters and options to control the import. Most options are rather self-explanatory.



If the option “Include Unit code and create unit” is checked, not only the classes will be imported, but also the non-class related unit code. Check this option if you want a complete import. If you just need a class’s interface this option can be unchecked.

If the option “Select Classes and Events to import” is checked, you may select which of the classes or events found in the source file will actually be imported in the model. Useful if you’re not interested in all classes and events contained by a unit.

If the option “Import Classes as Placeholder” is checked all imported classes will be marked placeholder. If it is unchecked classes will remain their current state or ‘real’ if not found in the current model.

There are three pre-set buttons, which set the filters to a “complete”, “interfaces only” “class interfaces” mode. The default settings (as displayed) are those for a complete import.

The In-source documentation import control settings are explained in detail in chapter “Importing a source file” page 57.

## Importing (adding) versus Refreshing

'Refresh Import' and Import as in 'Add to Model' act similar - the difference is how existing units and classes are treated.

Importing with the 'Add to Model' function (in the IDE or ModelMaker) will add non-existing units or classes contained in the added unit. If either unit or class already exists in the model it will be refreshed.

'Refresh Import' issued from the IDE integration expert will only refresh the unit if already existed in the model and will not add a unit not currently in the model. In ModelMaker you can only refresh an existing unit.

## Avoiding creep - removing code during import

ModelMaker supports removing certain fragments of code during import to avoid creep in a full generation / re-import cycle.

ModelMaker generated default class separators such as

```
{  
***** TSample *****  
}
```

and ModelMaker generated default category markers such as  
{<<Category>>: Model linking}

are automatically removed.

Additionally ModelMaker will remove the following code:

1. All code marked by a MMWIN:>>STARTREMOVE and MMWIN:>>ENDREMOVE pair.
2. All comments with the 'removal signature' as defined in the Code Import options.
3. Matched (and optionally unmatched) comments according to the documentation style comment.

For importing "In source documentation" and removing comments with the documentation signature(s), refer to chapter on "Importing in source Documentation", page 68.

## STARTREMOVE and ENDREMOVE tags

You can use these tags in any code, comment or string to instruct ModelMaker to remove all code between the tags including the tags themselves:

```
MMWIN:>>STARTREMOVE  
MMWIN:>>ENDREMOVE
```

The corresponding setting in the Project options|Code import tab will enable/disable filtering based on these tags.

These tags are commonly used to remove (part of) a macro that was generated by ModelMaker from the input file. Check the chapter on generating Method Instrumentation code for an example.

## Comments with remove signature

If you want to remove certain comments from the source file you can use comments with the Removal Signature. This signature is defined in the Project options|Code import tab which also enables and disables removing comments with this signature.

You will need to use this type of comment remove filter if you

1. Define a custom class separator,
2. Define a custom method separator,
3. Define a custom method section separator
4. Redefine the category expansion macros 'IntfCategory' or 'ImplCategory'

Assuming { - } to be the removal style comment, a custom class header should look like

```
{ -  
*****  
*  
*           <!Classname!>  
*  
*****  
}
```

Similar, a custom category expansion tag could look like

```
{ - Category: <!Category!> }
```

Note that Method End Documentation is automatically removed due to the fact that the importer will remove the method including the line containing the method's final `end;`.

## Import restrictions and limitations

ModelMaker usually imports in about 99.9 % of all cases without problems. If ModelMaker generated the source file, the imported code is usually 100 % correct. ModelMaker uses a combination of syntactical analyses and line based extraction to support importing of code that is not entirely syntactically correct.

ModelMaker's import mechanism imposes the following restrictions on source files in order to be imported correct. ModelMaker's importer uses the same parser as the ModelMaker Code Explorer. The ModelMaker Code Explorer that integrates in the Delphi IDE will display a list of parse errors. That way it acts more or less as tool to check code before importing into ModelMaker.

## Class and Interface interfaces

Restrictions in class and interface declarations.



Any comments, compiler directives and white space in a class's interface are/is ignored except in method parameter lists.

Comma separated Field declarations are converted into separate fields:

```
FA, FB: Integer;
```

is imported as

```
FA: Integer;  
FB: Integer;
```

Procedure or method pointers that are not defined as an type are not imported correct. The work around is to use a type definition.

The following code causes import errors.

```
TSample = class(TObject)  
  FEvent: procedure of object;  
end;
```

Which can be replaced by this code that will import correct:

```
TMyEvent = procedure of object;  
  
TSample = class(TObject)  
  FEvent: TmyEvent;  
end;
```

## Method implementation

Restrictions in method declarations.

Any comments, compiler directives and white space in are/is ignored except in parameter lists.

Local variables immediately following the method declaration will be converted to ModelMaker method variables. If local variables for example are preceded by a **type** or **const** declaration, they will be added to the method's local code section, just like all other local code for that method.

In the following example the local vars. I, J and S will be converted to ModelMaker local vars., the const declaration and procedure CheckIt will be placed in the method's local code section.

```
procedure TSample.Action;  
var  
  I, J: Integer;  
  S: string;  
  const CheckSum = $AAAA; // this will go into local code  
  procedure CheckIt;  
  begin  
  end; // this is the end of the local section  
begin  
  CheckIt;  
end;
```

In the following example the local vars. I, J and S will be placed in the method's local code section together with the const declaration:

```

procedure TSample.Action;
const CheckSum = $AAAA; // this will go into local code including the vars
var
  I, J: Integer;
  S: string;
  procedure CheckIt;
  begin
    end; // end of local code section
begin
  CheckIt;
end;
    
```

During refresh import of a method already existing in the model, the importer will leave the code sections intact wherever possible. If the importer cannot locate a non-user owned section of code, it will simply leave the section in the method and give a warning.

## Comments and white space

The following table shows how ModelMaker treats comments and compiler directives

Comment or compiler directive in:	Import result
Class interface	Ignored
Method header	Ignored except in parameter list
Local vars.	Ignored
Method local code and body	Copied to method
All other code	Copied to unit code

Check the ModelMaker generated default class separators such as

```

{
***** TSample *****
}
    
```

and ModelMaker generated default category markers such as  
{<<Category>>: Model linking }

are automatically removed.

If you define a custom class separator, method section separator, or redefine for example the category expansion macro TODO, you should use the remove style comments to avoid creep during import. Check paragraph “Comments with remove signature”, page 60.

## Unsupported language constructs

Include files are not read during import, so if you find yourself thinking: “where’s my method implementation gone?” you probably need to add the included files to the imported unit using a text editor and re-import the file.

Compiler directives in class interfaces are not supported and are a potential problem source. Using inheritance may sometimes solve this. Worst case you need to create two units.

Compiler directives *around* method implementations are not supported. Placing the directives inside the method can solve this:

```
{ $IFDEF DEMO }  
procedure TSample.Action;  
begin  
end;  
{ $ELSE }  
procedure TSample.Action;  
begin  
    { actually do something useful }  
end;  
{ $ENDIF }
```

Won't be imported correct, but can be replaced by the following code which will be imported fine:

```
procedure TSample.Action;  
begin  
{ $IFDEF DEMO }  
{ $ELSE }  
    { actually do something useful }  
{ $ENDIF }  
end;
```

The importer matches **begin..end try..end, case..end** pairs etc. to locate methods. Because conditional defines are not interpreted, using conditional defines you can create code that will compile correct but will not import correct. In fact The Delphi IDE background compiler uses a similar mechanism and will not be able to function properly either when inserting new methods in code completion or creating a new event handler.

This code for example will confuse the importer's begin end matching. The method will not be imported correct.

```
procedure TSample.Action;  
{ $IFDEF DEMO }  
var  
    S: string;  
begin  
    S := 'Demo';  
    ShowMessage(S);  
{ $ELSE }  
begin  
{ $ENDIF }  
end;
```

You can replace the previous code by the following code that will import correct and as a side effect allows the Delphi IDE to stay on track too:

```
procedure TSample.Action;  
{ $IFDEF DEMO }  
var  
    S: string;  
{ $ENDIF }  
begin  
{ $IFDEF DEMO }
```

```
S := 'Demo';  
  ShowMessage(S);  
{ $ENDIF }  
end;
```

Pure assembler methods are not supported:

```
procedure TSample.Fast; assembler;  
asm  
end;
```

Expressions in an indexed property's index specifier are not supported:

```
property FirstPicture: TBitmap index BM_USER + 0 read GetPicture;  
property SecondPicture: TBitmap index BM_USER + 1 read GetPicture;  
property ThirdPicture: TBitmap index BM_USER + 2 read GetPicture;
```

This can be solved like:

```
property FirstPicture: TBitmap index BM_FIRST read GetPicture;  
property SecondPicture: TBitmap index BM_SECOND read GetPicture;  
property ThirdPicture: TBitmap index BM_THIRD read GetPicture;
```

## Conversion errors

Any import conversion errors or warnings will be displayed in the Message View. The messages may be printed, saved etc.

Not reported conversion errors are:

1. Minor changes in property access method parameters lists.
2. Positioning of code generation tags in the unit code.

The best thing to do after importing a complex unit, is to perform a Delphi syntax check on the re-generated unit. From our experience it shows that if there are any remaining errors, they will evolve here.

Another option is to make a file based difference in the Difference view between the imported unit and the original source file. You should make sure that Code generation sorting scheme matches the scheme used in the original file. You might need to import Custom Code order and use the same order during generation to maintain code order. Check chapter "Maintaining Code Order / Custom member order" on page 49.

Note that to build a *structured* difference the same importer is used that will hide the same type of errors!

## Auto Refresh Import

The Auto Refresh Import feature is only available together with the Delphi 4 and higher. This function will automatically refresh a unit in ModelMaker if you save the unit in the Delphi IDE. This improves synchronization of code developed both in ModelMaker and the Delphi IDE at the same time. However there are some serious warnings.

### How it works

If you change a unit in the IDE editor that is also maintained in a ModelMaker model, the model and source file will be out of sync. Normally you have to 'refresh import' the unit to synchronize the model with the changes on disk. The auto-refresh import feature will do this automatically each time you save a unit in the IDE.

### How it is activated and controlled

In the ModelMaker menu in the Delphi 4 (and higher) IDE check the item 'Enable Auto Refresh'. If this option is set, each time you save a unit (or project) in the IDE, the 'Auto Refresh' command is sent to ModelMaker which checks if:

1. The Environment option 'Auto refresh Import' is checked
2. The unit is in the current model
3. Unit generation is not (user) locked
4. The unit has 'auto code generation' enabled

If all above conditions are met, ModelMaker will do a refresh import and unlike after a manual 'refresh import' leave the unit in 'auto generation enabled' mode and - this is **very important** - regenerate the unit.

Effectively Auto Refresh improves synchronization between ModelMaker and the Delphi IDE: whenever you change something in ModelMaker, the auto-generation enabled unit will regenerate the file and reload it in the IDE. Whenever you change and save a file in the IDE, ModelMaker will resynchronize it in the model.

### Warnings

In the normal, non-auto refresh development model you always have one master and slave: either ModelMaker refreshes the IDE using automatic code generation or you manually refresh the ModelMaker model with the IDE if you want to resynchronize again. With this feature there's no master or slave anymore. This can seriously damage your work as may be clear from the following example: When refreshing the unit, ModelMaker assumes it's reading a 'compilable unit'. If you for example have omitted a single begin or 'end;' or worse, comment out something, have unterminated strings etc. class and method import will be in trouble and not detect your error but simply remove all 'unwanted' methods. Since ModelMaker detected a change the unit is auto-generated and immediately after you saved the unit is reloaded with disastrous results. Experience shows that it's easy to lose lots of work instantly. Auto Refresh must be used with great care. Note that if Auto Save is enabled in the IDE, the Compile / Run command will auto save modified units depending on your IDE environment settings.

If you want more control, rather than just save in the IDE invokes auto refresh, you can use the 'Refresh Import' command from the ModelMaker IDE expert to save a file. This command will not only generate a manual refresh import command but also automatically save your unit in the IDE. Therefore you could use this command with shortcut Ctrl+Shift+H rather than the conventional Ctrl+S to save and refresh. You can add the ModelMaker Refresh command to the IDE tool bar. Check chapter (Integration in) Delphi IDE page 106.

## Editing Form source files

ModelMaker Tools developed the ModelMaker Code Explorer to help editing Form, Data Module and other resource module source files. Due to the nature of these resource module files the IDE editor is more suitable for editing them. The ModelMaker Code Explorer will dock into the IDE editor and bring basic Code Model editing and navigation actions right into the IDE.

If you do not have the ModelMaker Code Explorer installed, you *can* edit form (and other resource module) source files with ModelMaker. This offers many advantages:

1. Use ModelMaker's high level view and filters to navigate through your form code.
2. Automatically *restructure* your source files by regenerating them.
3. Improve your form code quality by adding methods and (array) properties with the same ease as for "normal" non form classes. Especially when you turn your forms into components, you want them to have nice and clean code to improve maintainability.
4. In general speed up form implementation.

For a smooth cooperation between Delphi and ModelMaker, stick to these rules:

In Delphi,

- Create and rename the form and unit.
- Add, delete and rename components.
- Set component properties.
- Create, rename and delete event handler methods.

Delphi adds all its components and event handler methods with the default visibility, so they are easy recognized in ModelMaker (use the members filter to filter out default visibility).

In ModelMaker,

1. Import your form file in an (empty) ModelMaker model.
2. Add, edit and delete all other members
3. Add additional classes to the unit.

To synchronize between Delphi and ModelMaker,

1. Regenerate the ModelMaker unit whenever you changed your code in ModelMaker. The auto-generate feature will help you doing this automatically.
2. Refresh the ModelMaker unit whenever you changed your code in Delphi, the integration experts will help you doing this automatically.

# In source documentation

## Overview

ModelMaker supports generating and importing "in-source" documentation. That is: the documentation and One Liner attributes of Code Model entities can be inserted during code generation and / or read (back) during import. Generation and importing of in-source documentation is controlled by the Project options tab "Source Doc Generation" and "Source Doc Import". The Import source dialog allows temporarily overruling the import settings.

"In-source" documentation must be marked with special (user definable) documentation and one liner signature tags, more or less like in Java Doc. These signatures must be an Object Pascal comment symbol followed by one or more letters, Common used signatures are:

Documentation tags:

```
{ {
{ :
( **
( * :
```

One liner tags:

```
{ 1
//:
//1
```

Generation is internally based on macros. To customize the generated format, you can redefine these macros. To ensure a correct round trip (generation followed by a re-import) the generation and import settings must match. Normally ModelMaker enforces a correct match by using the essential import settings for generation. If you redefine the documentation generation macros, you must ensure correct matching.

## Generating in-source documentation

ModelMaker supports generating and importing 'in-source' documentation in source files. Generation is controlled by the Project options on tab 'Source Doc Generation' and some macros (refer to Macros). You do not need to define these macros, as ModelMaker will on the fly insert the required macros. You can however redefine them either as environment or as project macro to customize the generated format.

*Documentation macros*

Macro name	Description	Example
ModuleDeclDoc	Used to expand Module (unit) documentation. Check Module related macros	<pre>{{ module &lt;!ModuleDoc!&gt; }</pre>
EventDoc	Used to expand Event Type documentation	<pre>{{ event type &lt;!MemberDoc!&gt; }</pre>
ClassIntDoc	Used to expand Class documentation in the class declaration	<pre>{{ class &lt;!ClassName!&gt; &lt;!ClassDoc!&gt; }</pre>
ClassImpDoc	Used to expand Class documentation in the class implementation (emitted just before the first method implementation)	<pre>{{ class &lt;!ClassName!&gt; &lt;!ClassDoc!&gt; }</pre>
MemberIntDoc	Used to expand member documentation in the class interface	<pre>{{ &lt;!ClassName!&gt;.&lt;!MemberName!&gt; &lt;!MemberDoc!&gt; }</pre>
MemberImpDoc	Used to expand method implementation documentation	<pre>{{ &lt;!ClassName!&gt;.&lt;!MemberName!&gt; &lt;!MemberDoc!&gt; }</pre>
MemberEndMacro	used to expand method termination documentation	<pre>{ &lt;!ClassName!&gt;.&lt;!MemberName!&gt;</pre>
OneLinerMacro	Used to expand one liners in classes, members, units and event types	<pre>//: Summary: &lt;!OneLiner!&gt;</pre>

In the examples it is assumed that {{ is the documentation signature and //: is the One Liner signature.

As you see in the examples, you can use any of the predefined macros such as <!ClassName!> inside the documentation macros.

However, ModelMaker is only able to import certain styles of source documentation. This is important if you want to 'Refresh Import' a unit.

## Importing in-source documentation

ModelMaker supports importing "in-source" documentation. Importing source documentation is controlled by settings in the Project options tab "Source Doc Import" and the Import source dialog. As explained: "In-source" documentation and One Liners must be marked with special (user definable) documentation signature tags. More or less like in Java Doc.

An example:



```
//1 SomeMethod does something useful.
{{
procedure TSample.SomeMethod
This method does something useful
It would take pages to tell what.
}
procedure TSample.SomeMethod;
begin
end;
```

In the above example the OneLiner signature is defined as //1 and the documentation signature is defined as {{. The whole comment until the matching comment end symbol is treated as documentation for the first entity defined following or preceding it - depending on the documentation import options. In the example above the comment will be assigned to TSample.SomeMethod. You cannot use multiple line // style comments for documentation.

In the Source documentation options there are three documentation import modes:

1. Import: enables source documentation import and replaces documentation in the model with that in the source file,
2. Clean up: leaves the documentation in the model unaffected, but removes documentation from the unit-code
3. Inactive: which does nothing and leaves the documentation in the unit code.

The option 'Remove unmatched documentation determines if only matched documentation should be removed or just any comment with the documentation signatures.

When assigning the comment to the Documentation attribute, the first and last #n lines are ignored. Defining any other value than 1 (one) is only useful if you redefine the documentation generation macros. The standard macros assume a value of 1.

You could use the fact that the first n lines (user definable) are ignored and place there the additional macros you'd like to generate. For example:

```
{{
<!ClassName!>.<!MemberName!>
(<!Visibility!>)
<!MemberDoc!>
}
```

This macro would require the first three (3) lines and the last one (1) to be skipped. Using this technique the member documentation won't grow each time you (refresh) import a unit. Another option is to use the Import "clean-up" import mode after in-source documentation has been imported once.

Alternatively you could leave the number of lines to be skipped at 1, and use additional removal style comments to customize your in-source documentation format.

For example:

```
macro ClassIntDoc =
{- *****
<!ClassName!>
```

```
*****}  
{  
<!ClassDoc!>  
}
```

Note the removal style comment in the first part of this macro. Check chapter “Comments with Remove signature”, page 60.

Related to this is the use of a custom ClassSeparator or MethodSeparator to emit more than just the documentation preceding the class or a method implementation. Refer to Code Generation options. Using custom separators has the advantage of not needing to redefine the documentation macros and that way ensuring that all expanded documentation looks similar.

For example, defining a MethodSeparator macro like this:

```
{- <!ClassName!>.<!MemberName!>  
  (<!Visibility!>) }
```

Combined with enabling the method implementation in source documentation (without redefining the related macro), this behaves as if the entire macro were:

```
{- <!ClassName!>.<!MemberName!>  
  (<!Visibility!>) }  
{  
<!MemberDoc!>  
}
```

Which would be emitted for example like this:

```
{- TSample.SomeMethod  
  (public) }  
{  
This is the documentation for the method  
}
```

Note the use of removal style comments for the MethodSeparator that avoids creep when re-importing the generated code.

# Code templates

ModelMaker supports the use of code templates. They are like user definable patterns and consist of a (usually consistent) set of members that is put in a code template source file. This template can then be applied whenever needed again. The template file acts like a structured persistent copy / paste buffer. The powerful aspect is that templates can be parameterized using user definable macros. ModelMaker will extract the macros, let you edit them and expand the macros before importing the template file. The template files can be edited in Delphi, but should not be imported in ModelMaker directly.

## Creating a Code template

To create a code template, in the Members view select the members you want the template to contain use the popup menu 'Create code template'. You'll be prompted for a file name. ModelMaker then generates the selected members as part of a stand-in class named TCodeTemplate. Here's an example of a simple template file containing a simple Items property with a standard TList implementation

```
unit SimpleItems;

  TCodeTemplate = class (TObject)
  private
    FItems: TList;
  protected
    function GetItems(Index: Integer): TObject;
  public
    property Items[Index: Integer]: TObject read GetItems;
  end;

function TCodeTemplate.GetItems(Index: Integer): TObject;
begin
  Result := TObject(FItems[Index]);
end;
```

## Applying a Code template

To apply a previously created template, select the popup menu 'Apply template' from the Members view. You'll be prompted for a template file name. ModelMaker will import the members contained by the first class in the template unit and add them to the currently selected class. Other classes and all other code in the unit are/is ignored. There's one exception: event type definitions can also be added to a code template, they will automatically be added to the event library when applying the template.

## Registering a Code template

Code Templates can be registered on the patterns palette (patterns view) and Code Template palette (members view popup menu). These palettes that look like the Delphi component palette make it even easier to apply a Code Template. To (un)register a template use the popup menu functions in the palettes or select the corresponding option when you create a Code Template. Code Templates are shared with the ModelMaker Code Explorer.

## Parameterize a Code template using macros

You can parameterize a code template by adding macros to the template unit. When applying a template, ModelMaker will first extract the macro parameters, let you edit them, expand the code template and finally apply the template. This allows you to create more generic templates. A macro definition should be formatted as

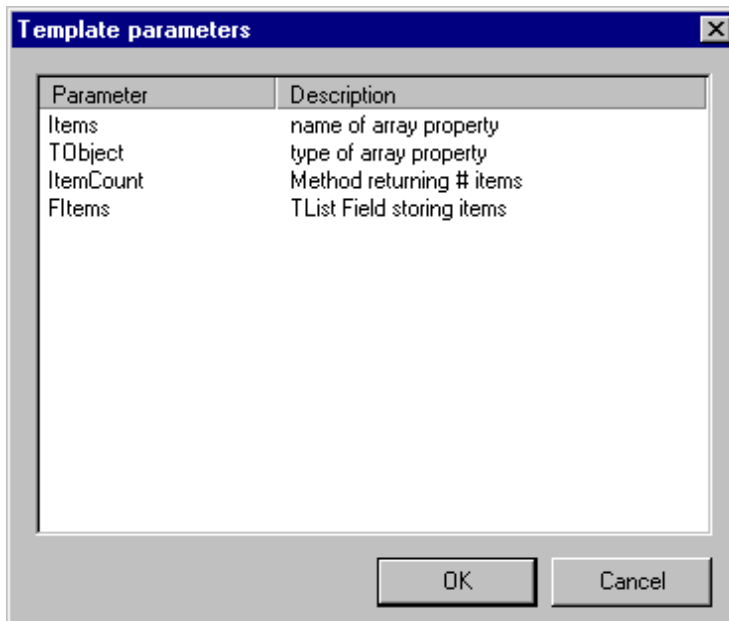
```
//DEFINEMACRO:macroname=macro description
```

The standard ModelMaker macro rules apply. For example: macroname must be an identifier. Parameterizing the above example could for example be done like this.

```
unit SimpleItems;
//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

TCodeTemplate = class (TObject)
  private
    <!FItems!>: TList;
  protected
    function Get<!Items!>(Index: Integer): <!TObject!>;
  public
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
end;

function TCodeTemplate.Get<!Items!>(Index: Integer): <!TObject!>;
begin
  Result := <!TObject!>(<!FItems!>[Index]);
end;
```



If you apply this template, ModelMaker will show a dialog with the list of parameters you defined: Items, TObject, ItemCount and FItems allowing you to change them for example in Members, TMember, MemberCount and FMembers. This way the template can be added multiple times in different contexts. ItemCount is not used in this example, but in the sample code template that is shipped with ModelMaker method ItemCount is part of the template.

ModelMaker predefines one macro: “ClassName” contains the name of the class the macro is applied to. You can redefine ClassName when parameterizing a template. Use ClassName for example to create a singleton implementation macro:

```
function TCodeTemplate.Instance: <!ClassName!>;
begin
    // return the single instance.
end;
```

When applied to a class named TMySample this will expand to:

```
function TMySample.Instance: TMySample;
begin
    // return the single instance.
end;
```

# Macros

## Overview

A macro in ModelMaker is an identifier placed between `<! and !>` tags. They may also include an optional parameter list. When the macro is expanded, the macro identifier and tags are substituted by the text associated to the macro. For example macro `<!UnitName!>` will (in unit Samples) be expanded to the actual unit's name 'Samples'.

Macros are used in

- Code Generation.
- Customizing certain aspects of code generation such as a custom class separator, inserting categories etc.
- Generating in-source documentation.
- Parameterizing Code Templates
- ModelMaker Code editor.

During code generation ModelMaker predefines some model statistics macros at run time. Such as `<!UnitName!>`, `<!Date!>` etc.

You define your own macros in the Macros view. Macros are defined per desktop (environment) and per project. If a macro is both in the project and the environment macros, the project macro overrides (redefines) the environment macro.

For Parameterizing Code Templates using Macros, refer to chapter Parameterize a template using macros , page 72. To expand a Code Template the predefined macros and project and environment macros are not used.

## Macros in Code generation

When generating a source file from a ModelMaker unit (refer to chapter Code Generation page 45), ModelMaker will expand macros in all text that is sent to the output file. Therefore macros can be placed in any code: in unit code, in a section of a method's implementation or even in a local var definition. Macro expansion is switched on and off with the generation control tags `MMWIN:START EXPAND` and `MMWIN:END EXPAND`. By default the expansion is switched ON. Check chapter "Macro expansion control tags", page 47 for an example.

When expanding a macro, first the list with predefined macros is checked, then the Project macros and finally the environment macros. If an identifier is not found, the macro text is

either just removed or generation is aborted depending on the setting of 'Ignore undefined macros' in the Project options Code Generation tab.

## Predefined macros

This table shows the macros ModelMaker predefines when generating a source code file. Some macros may be redefined - especially the documentation expanders, others such as Date and ClassName are fixed.

Macro Name	Allows override	Description	Example
<i>Generic predefined macros</i>			
Date	No	Generation date, for example: 19-02-2003	
Time	No	Generation time, for example: 12:34:56	
LineNr		The 1-based line number in the resulting source file at which the macro is defined.	
<i>Documentation expanders</i>			
ModuleDeclDoc	Yes	Used to expand Module (unit) documentation. Check Module related macros	<pre>{ { &lt;!ModuleDoc!&gt; }</pre>
EventDoc	Yes	Used to expand Event Type documentation	<pre>{ { &lt;!MemberDoc!&gt; }</pre>
ClassIntDoc	Yes	Used to expand Class documentation in the class declaration	<pre>{ { &lt;!ClassDoc!&gt; }</pre>
ClassImpDoc	Yes	Used to expand Class documentation in the class implementation (emitted just before the first method implementation)	<pre>{ { &lt;!ClassDoc!&gt; }</pre>
MemberIntDoc	Yes	Used to expand member documentation in the class interface	<pre>{ { &lt;!MemberDoc!&gt; }</pre>
MemberImpDoc	Yes	Used to expand method implementation documentation	<pre>{ { &lt;!MemberDoc!&gt; }</pre>
<i>Customization Macros</i>			
ClassSeparator	Yes	Custom Class separator, Only active if corresponding option is activated in Project Code	<pre>{-***** ** Class: &lt;!ClassName!&gt; ** Category: &lt;!Category!&gt; *****}</pre>

		Generation options.	assuming {- is the comment remove style
MethodSeparator	Yes	Custom Method implementation separator. Only active if corresponding option is activated in Project Code Generation options.	{- <!MemberName!> -} assuming {- is the comment remove style
SectionSeparator	Yes	Custom Method section separator. Only active if corresponding option is activated in Project Code Generation options.	{----- section -----} assuming {- is the comment remove style
IntfCategory ImplCategory	Yes	Wrappers for Category emission in class or member interface and implementation.	{- Category: <!Category!>}
OneLinerMacro	Yes	Used to expand one liners (class, member, unit event type)	//: Summary: <!OneLiner!>}
MethodEndMacro	Yes	Used to expand method end documentation. Only active if corresponding option is set in Project Source documentation Generation options	{<!ClassName!>.<!MethodName! >}
<i>Entity specific Macros</i>			
OneLiner	No	All entities: for Modules only during Module documentation generation.	
OneLiner Category	No	All entities: for Modules only during Module documentation generation.	
ModuleDoc ModuleName ModulePath Alias UnitName UnitPath	No	Module specific macros (units). Always available (not only in module documentation). The UnitXYZ macros exist for backward compatibility. Use the ModuleXYZ macros in new projects	
ClassDoc ClassName TrimmedClassName Ancestor	No	Class specific macros. Valid in class and members of that class. Valid in unit code after the first class has been generated. TrimmedClassName contains the class name with the first character ('T' / 't') removed.	In unit code use this to insert auto updated global variables or class pointer types:  var <!TrimmedClassName!>: <!ClassName!>; type <!ClassName!>Class = class of <!ClassName!>;
MemberDoc MemberName Visibility DataType	No	Member specific macros. Valid during generation of declaration, documentation and method	



		implementation code.	
Parameters CallParams MethodKind	No	Method specific macros. Valid during generation of declaration, documentation and method implementation code.	
		Event Types use the same macros as methods. Except: Category and Visibility which are undefined	

The Documentation Expander macros are used to generate in-source documentation. To customize the generated format, you must redefine these macros, either in the project or environment macros. Refer to chapter “Generating in source documentation”, page 67.

The Customization macros can be (re)defined to customize the format of the related aspect. ClassSeparator, MethodSeparator and SectionSeparator are only active if the corresponding options are checked in the project Code Generation options. Refer to chapter “Code Generation Options”, page 49.

## Using Macros in code

Some rules that apply to using macros:

- An entire macro including start and end tags must reside on a single line.
- Macro identifiers can contain characters ['0'..'9', 'a'..'z', 'A'..'Z', '\_']. This is similar to Object Pascal identifiers, although macro names can start with a number.
- White space surrounding the macro identifier is ignored.
- A start tag not followed by a valid identifier is not considered to be a macro
- If the macro identifier is not followed by either the end tag or the parameter list, the macro is not considered to be a macro.

Rather than presenting the macro syntax diagram, an example will demonstrate the use and definition of macros.

Assume these (user) macros to be defined, in either the project macro list or in the environment macro list.

Name: Author

Parameters:

Text:

S.M.A.R.T. Programmer

Name: Assert

Parameters: Cond, Msg

Text:

```
{$IFOPT D+}
```

```
if not (<!Cond!>) then
```

```
    raise Exception.Create('Assertion error in line <!LineNr!> of unit <!UnitName!>' +
```

```

                                #13 + Msg);
{$ENDIF}

```

This is how you could use these macros:

```

procedure SomeAction(Index: Integer; C: Char);
begin
  <!Assert(Index >= 0, 'Index out of range')!>
  <!Assert(C in ['a', 'b'], 'Char out of range')!>
  <!Assert(ValidPair(Index, C), 'Additional checks failed')!>
  ShowMessage('<!Author!> created this code');
end;

```

Assuming the procedure was placed on line 100 in unit Demos, this text would expand to:

```

procedure SomeAction(Index: Integer; C: Char);
begin
  {$IFOPT D+}
  if not (Index >= 0) then
    raise Exception.Create('Assertion error in line 102 of unit Demos' +
      #13 + 'Index out of range');
  {$ENDIF}
  {$IFOPT D+}
  if not (C in ['a', 'b']) then
    raise Exception.Create('Assertion error in line 107 of unit Demos' +
      #13 + 'Char out of range');
  {$ENDIF}
  {$IFOPT D+}
  if not (ValidPair(Index, C)) then
    raise Exception.Create('Assertion error in line 112 of unit Demos' +
      #13 + 'Additional checks failed');
  {$ENDIF}
  ShowMessage('S.M.A.R.T. Programmer created this code');
end;

```

The basic rules that apply to the use of macros are:

- When using macros in text, the complete macro including its parameter list must reside on a single line. So this won't work:
 

```

      <!Assert((A > B) and (B > C),
        'This is bad input')!>
      
```
- Arguments in the parameter list are comma delimited, such as in the use of `Assert`.
- Arguments can contain even pairs of `()`, `[]` and `"` characters, such as in sets, arrays and string literals.
- If no parameters are defined, as in `<!Author!>`, you omit the brackets when using the macro.
- Macros can use other macros in their macro text. In fact even parameters can be macros. Nesting is allowed up to 15 levels.
- Circular macro definitions are illegal.
- Macros expand to plain text. See for example the use of the predefined `LineNr` and `UnitName` macros in the `Assert` macro's text. The expanded macro `LineNr` is not an Integer, and the expanded macro `UnitName` is not a string.

## Using macros in the code editors

An entirely different use of macros is to expand a macro in the code editor. To do this, press Ctrl+Space after typing the macro name - or Shift+Space, depending on your Environment Options|Editors settings. This is convenient if you create macros like: **tryf** that could expand to:

```
try
  >< // >< in macro text positions cursor after expansion.
finally
  .Free;
end;
```

## Using macros in your default unit template

To demonstrate the use macros, here's the effect of using macros on the default unit which was described in "Adjusting the unit template" page 50.

```
{
<!UnitHeader!>

<!FreeWareDisclaimer!>
}

unit <!UnitName!>;

interface

// rest of the unit template is unchanged
end.
```

Notice how this template differs from the previous template:

- The company header is now placed in a macro `UnitHeader`. This saves a lot of redundant text.
- The macro `UnitHeader` contains other macros like unit name, model name, etc.
- The disclaimer is now placed in the macro `FreeWareDisclaimer`.

# Diagrams

## Diagrams, Diagram List view

ModelMaker supports a set of diagram types as defined by the UML. Currently the UML v1.3 style is implemented. A model can contain any number of diagrams of any type. The Diagram list view (Main menu View or F5) contains a list of all diagrams in the model. Diagrams contain symbols and associations that can be, but do not need to be linked to entities in the code model.

The Diagrams list View is used to create, rename and delete diagrams. Select a diagram in the Diagrams list view to open it in the diagram editor.

Diagrams are named. Names do not need to be unique.

### Hierarchy

Diagrams can be organized hierarchically in the Diagrams view. This organization is pure visual and has no further meaning in the model. Any diagram can serve as a parent for any other diagram. To rearrange parent child relations, make sure the list's "Order by" is set to Hierarchy (Diagram list Popup menu). Then use drag and drop to rearrange diagrams. If the Ctrl key is pressed while dropping, the hierarchy is edited; else the order within the current parent is changed.

### Styles

Each diagram has a visual style and a symbol style that define the default styles for the symbol inside that diagram. Editing these styles (diagram editor popup menu Diagram attributes), affects the appearance of all symbols in that diagram. Refer to chapter "Visual styles" for more information.

### Default properties

Whenever a new diagram is created (except a Cloned diagram), the format (size) and orientation (portrait or landscape) are set to the defaults as defined in the project options.

## Symbols and visual containment

Symbols can visually contain other symbols. For example package symbols can contain class symbols or other package symbols; and Nodes in a deployment diagram can contain Component symbols. Visual containment implicates visual ownership only. Visual containment is not linked to the code model. Inserting a class symbol into a package that is linked to a unit will not actually move that class to that unit. It is just visualized as being part of that unit.

After a symbol is created, visual containment is fixed and cannot be changed in the diagram editor. The only way to change a symbol's parent is to cut the symbol and paste it on the new parent. Take care: connected associations are only copied if both ends of the association are copied.

## **Symbol names and other visual adornments**

Most symbols have a symbol name text adornment. Depending on the visual style options, the name will display a hotlink status icon, a navigation icon and the hyperlink status. Check chapters Hyperlinks and Hotlinks. The stereotype (category) of a symbol is also displayed in the name adornment.

Other visual adornments are depending on the type of symbol. For example: a state region for a concurrent state contains a state region divider and icons, a sequence diagram role symbol contains a lifeline etc.

## **Associations**

### **Basics**

Associations are used to model relations between symbols. Some associations such as “documentation link” and “constraint” can also be connected to other associations. An association that is not connected to both ends is invalid and will automatically be removed from the diagram.

All associations have a direction: they lead from a source symbol to a target symbol. Usually a navigation arrow is displayed is appropriate. These arrows can be suppressed in the visual style on project, diagram or association level.

Usually the visual path of an association is formed by the two association end points. Shape nodes can be inserted to create more complex paths.

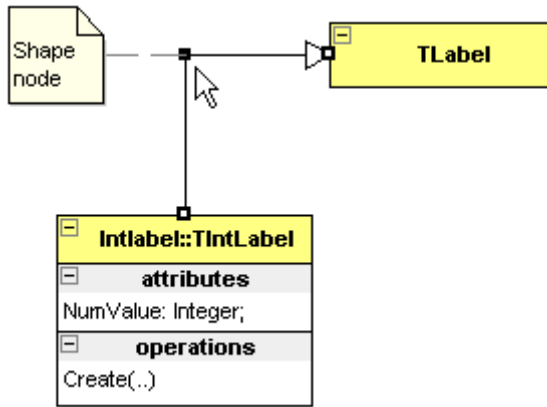
Associations are created by clicking the mouse on the source symbol and while keeping the mouse down, drag to the target symbol where the mouse is released. Once created, associations can be connected to different symbols by moving one of the endpoints to the new source or target symbol.

### **Anchors**

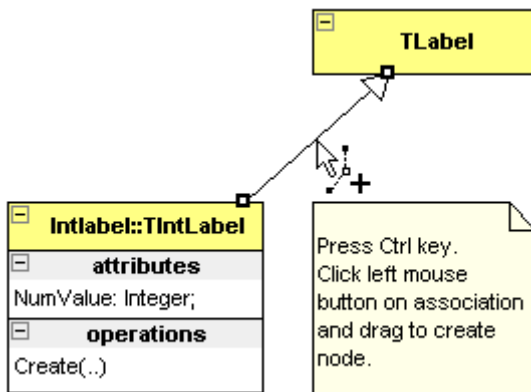
Associations are connected with an anchor point to symbols. Usually the anchor is connected to the center of a symbol. You can however drag-move the connection anchor point within the bounds of the connected symbol. This will change the intersection point of symbol and association. The diagram editor’s popup menu has a function ‘Reset Anchors’ which will reset both anchors to the symbol’s center.

### **Shape Nodes**

Shape nodes can be inserted to change the visual path of an association. Shape nodes allow bending associations visually. The association anchor points (connection points on the connected symbols) and the association’s shape nodes make up the actual visual path of the association. The picture below shows an generalization association from TIntLabel to TLabel with one shape node.



To insert a shape node into an association, press the Ctrl-key and drag on a line segment of an association. After the mouse is released, a new node is inserted. Alternatively, use Insert Shape Node from the diagram editor “Association” popup sub menu. This command is available if you invoke the popup menu on an association.

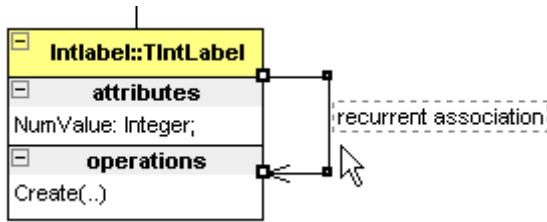


Shape nodes can be moved by selecting the association and then drag them with the mouse.

A shape node can be deleted by aligning it with the two surrounding nodes and is basically the reverse of creating a new shape node. Alternatively, use Delete Shape Node from the diagram editor “Association” popup sub menu. This command is available if you invoke the popup menu on a shape node. The command ‘Clear shape nodes’ in the same popup menu will clear all shape nodes in all selected associations.

### Recurrent associations

Normally associations connect two different symbols. If both ends of an association are connected to the same symbol, the association is recurrent. Most associations can be made recurrent. Some recurrent associations will display a rounded curve rather than a square path. The rounded curve is a bezier that is controlled by two shape nodes. Inserting another shape node will turn the bezier curve into normal straight lines.



To create a new recurrent association, simply make the target symbol the same as the source symbol. Two shape nodes will automatically be inserted that allow control of the visual path.

An existing non-recurrent association can be made recurrent by moving one of the endpoints to the same symbol as the other endpoint. Again, two shape nodes will be inserted automatically.

To convert a recurrent association into a non-recurrent association, move either source or target endpoint to another symbol. The shape nodes will be removed.

### Association Name, Qualifiers, Roles and other adornments

Most associations can be named and have an association name adornment. If the name is currently not visible, select the association and press F2 to invoke the in place editor. Depending on the visual style options, the association name will display a hotlink status icon, a navigation icon and the hyperlink status. Check chapters Hyperlinks and Hotlinks. The stereotype (category) of the association is also displayed in the name text adornment.

Other adornments include

1. Qualifiers in qualified association such as class associations. To reduce visual space, the qualifier's type can automatically be suppressed. This is controlled with the visual style options.
2. Role name, both source and target endpoint can be named in for example class associations and object links.
3. Multiplicity (cardinality), both source and target endpoint can have a multiplicity in for example class associations and actor communications.
4. Conditions, usually only in the source endpoint of sequence diagram messages and state transitions.

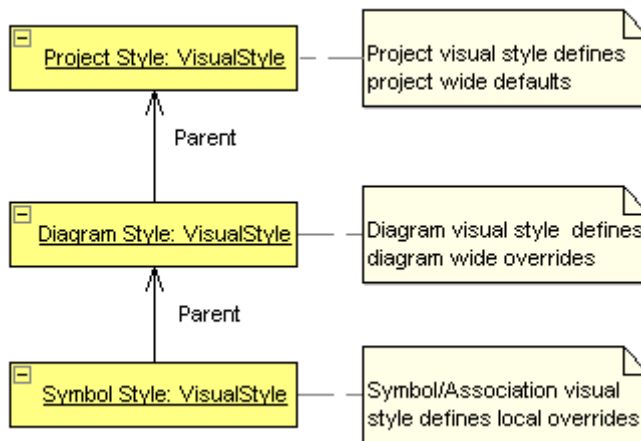
These adornments are all texts and can be moved freely in the diagram. The word break property of text adornments is controlled with the Word break functions on the "Align and Size" palette.

## Visual styles

All symbols, associations and diagrams have a visual style. It is this visual style that defines how a symbol appears. A visual style contains font settings, a color palette and some options that control display of icons. The symbol styles, that are discussed in the next chapter, control *what* is displayed; the visual styles control *how* a symbol is displayed.

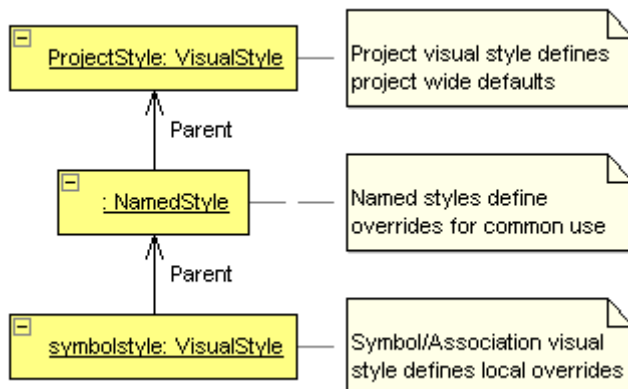
## Style hierarchy

All visual styles are linked in a lookup hierarchy. A style normally looks up an attribute in the parent style but can also override “parent” attributes. By default symbol and association styles are linked to the diagram visual style and have all properties set to “lookup from parent”; that is override or change nothing. As a result, all symbols will appear as defined by the diagram visual style. On their turn, all diagram visual styles are linked to the project visual style and also have all properties set to “lookup from parent”.



This hierarchical structure allows easy adjusting of visual appearance on any level. To change the appearance of an entire project: change the project style. You can even save the project settings as default, and new projects will have the same project style. Change the diagram style to modify the appearance of all symbols a single diagram. To change the appearance of a single symbol, edit the symbol’s style.

To make reuse of visual styles easy, the visual style manager allows creation of “Named styles”. Named styles can be used to define a specific appearance that can be reused. A named style is applied making it the parent of a diagram or symbol style. Named styles can have other named styles or the project visual style as their parent. The diagram editor tool bar contains a parent style combo which is used for this.



## Visual style properties

A visual style consists of a set of properties. Not all symbols / associations use all properties.

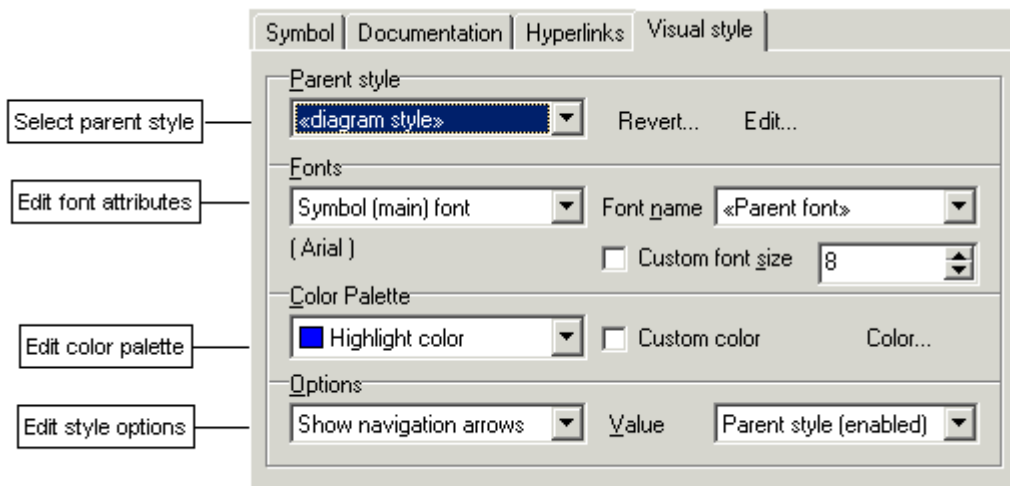


1. A main font name and size. The main font is used for symbol and association names. The font's style, bold, italic and underline, cannot be defined because that is usually has a syntactical meaning in the symbol as defined in the UML specifications: classifier names are bold, instances named are underlined etc.
2. An adornment or text font. Used for all other texts. For example the members in a class symbols and the roles in an association.
3. A color palette defining the colors for basic drawing entities such as main font, symbol compartment, symbol pen, symbol tab, association line etc. Which entries on this palette are used is dependent on the type of symbol.
4. Options that control display of some visual elements, such as: navigation arrows in associations, hotlink icons, navigation (hyperlink) icons etc.

Most symbols properties dialogs and the diagram properties dialog contain a 'visual style' tab that allows editing a style. Refer to next paragraph.

### Controlling & assigning styles

Most symbols properties dialogs and the diagram properties dialog contain a 'visual style' tab that allows editing the symbol's visual style. In this tab you change and or edit the parent style and the style's attributes.



To entirely revert to the parent style, erasing all overridden / redefined attributes, click the 'Revert' button.

The diagram editor's tool bar contains some visual style specific tools.

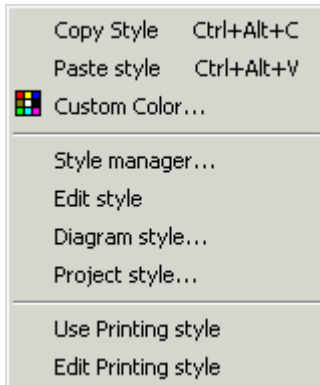


The "parent style" combo displays the parent style for the selected symbols and associations. It is blank if selected symbols have different parents. It allows assigning a new parent for the selected symbols.

The “Revert to parent style” tool will reset the visual styles for all selected symbols and associations. Useful to erase any local redefined style attributes.

The Apply color tool will let you select a color and apply it to the selected symbols and associations. The color is applied as the “main” feature color. Usually this is the symbol color palette entry, but for tabbed symbols like class symbols and package symbols, the symbol tab color is changed.

The Diagram editor contains a ‘Visual style’ sub popup menu that contains some additional visual style related functions.



Most striking functions in this popup menu are:

1. Copy / Paste a visual style. This is useful in case you have redefined the style of a symbol and want to apply the same visual style to a selection of other symbols. Note that similar functions exist for the symbol style that controls the display of members etc.
2. Show the Style Manager and it’s named styles, which is described in the next paragraph.
3. Toggle (and Edit) “Printing Style”. The printing style is described in the next paragraphs.

## Style Manager

The visual Style Manager is used to create and maintain named visual styles. Named styles can be assigned as parent style for symbols and diagrams. They allow creating a predefined set of appearances that can be applied by simply assigning the style.

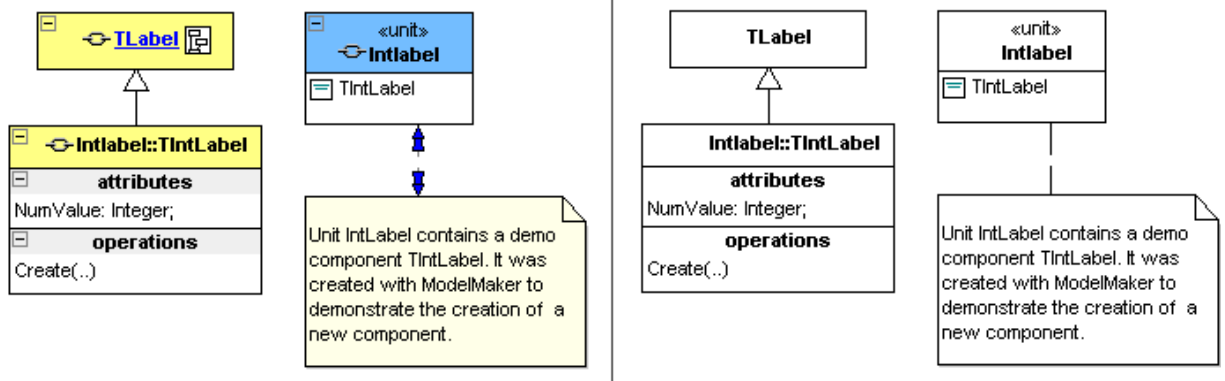
For example, you could create a style named “System components” which specifies a specific blue palette to paint symbols. Another style could be named “GUI components” and define a yellow palette to paint symbols.

In class diagrams, you can then easily change the visual appearance of a class symbol by assigning “System components” as parent style for the class symbol using the diagram editor’s tool bar parent combo.

Named Visual styles can be imported or exported with the Style Manager. This allows synchronizing named styles in projects.

## Printing Style

ModelMaker allows suppressing a specific set of graphical features when printing diagrams. These include: printing in black and white (suppressing colors), no navigation icons, no hotlink icons etc. While these features can help while designing, they may be unwanted in printed output. The following picture shows the same diagram in normal and printing style.



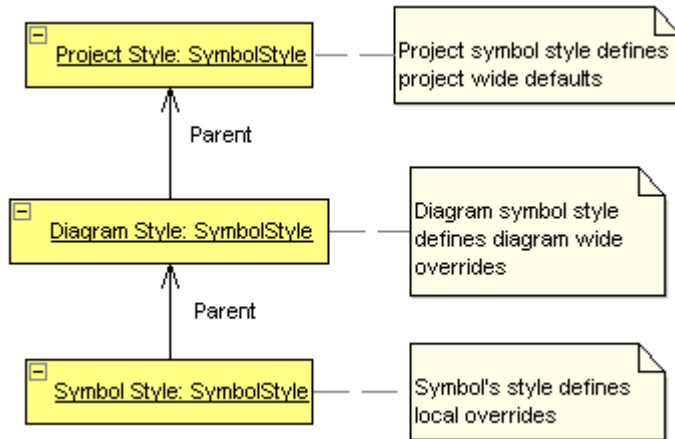
The printing style is defined in the diagram environment options. It is automatically superimposed on all other styles when printing, and can also be manually activated in the diagram editor. The diagram editor popup menu “visual style” menu contains an item that toggles the “Use Printing style” state. If the printing style is active in the diagram editor, it will also be effective when creating visual exports as image file or to the clipboard.

## Symbol (contents) style

Just like visual styles hierarchically control the visual appearance of diagrams and symbols, symbol styles control the contents of displayed symbols. For example: which members are displayed in a class symbol and in which format is controlled by the class symbol “symbol style”. The symbol styles control *what* is displayed; the visual styles control *how* it is displayed. The symbol styles are only applicable for a specific set of symbols such as class, interface and (unit) package symbols. In classes the symbol style controls which members are displayed and how they are displayed. In unit packages they control whether contained classes are automatically displayed.

### Style hierarchy

Just like visual styles, all symbol styles are linked in a lookup hierarchy and can override parent style attributes. Symbol styles are linked to the diagram symbol style and have all properties set to “lookup from parent” that is: override or change nothing. As a result, all symbols will appear as defined by the diagram symbol style. On their turn, all diagram symbol styles are linked to the project symbol style and also have all properties set to “lookup from parent”.



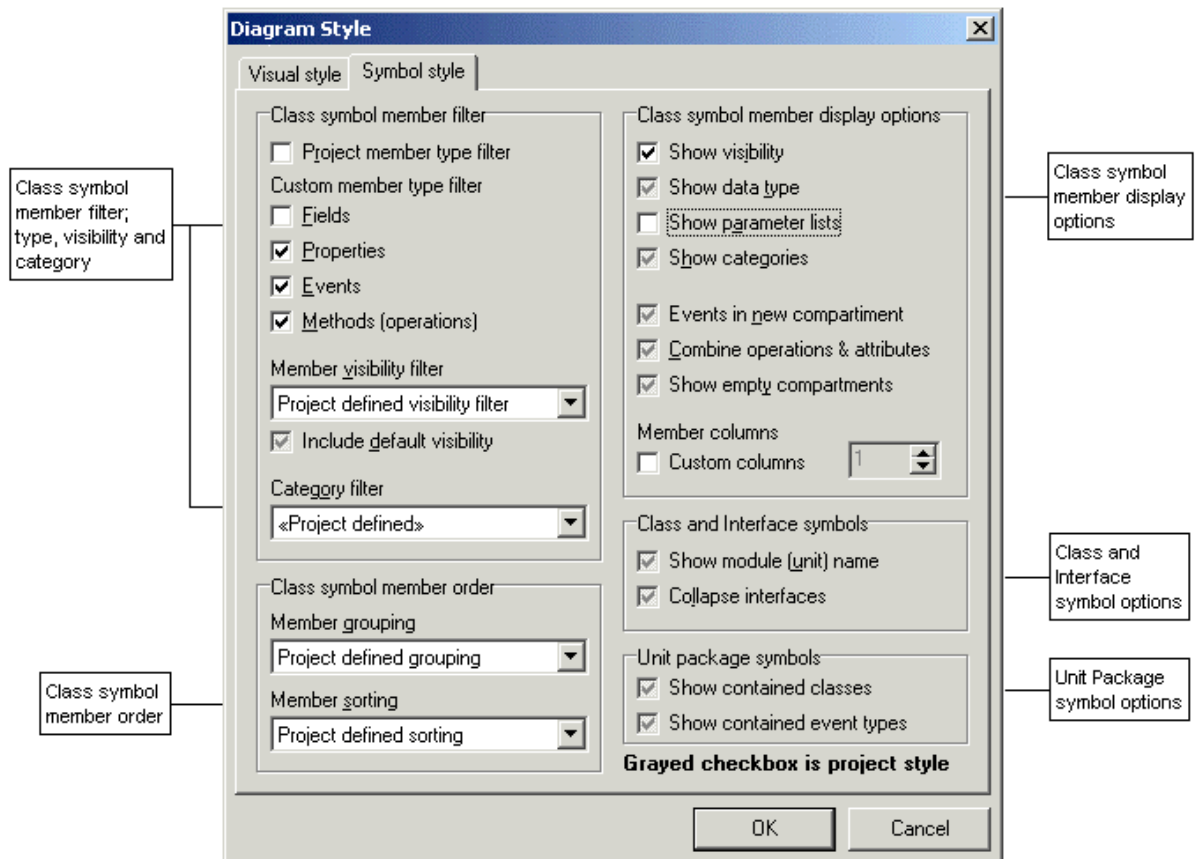
This hierarchical structure allows easy adjusting of what is displayed within symbols. To change the style of an entire project: change the project style. You can even save the project settings as default, and new projects will have the same project style. Change the diagram style to modify all symbols a single diagram. To change the contents of a single symbol, edit the symbol's style.

Unlike the visual style, the symbol style cannot be linked to named styles.

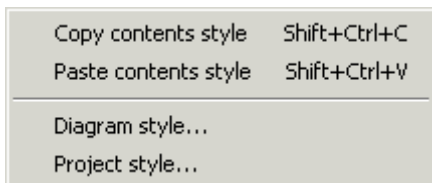
## Controlling & assigning styles

The diagram properties dialog contains a 'symbol style' tab that allows editing the symbol style. The project options "symbol style" tab is similar. In these tabs you control how class and interface symbols display members and unit packages display contained

classes.



The Diagram editor contains a ‘Symbol style’ sub popup menu that contains some symbol style related functions.



Most striking functions in this popup menu are Copy / Paste a symbol style. This is useful in case you have redefined the style of a for example a class symbol and want to apply the same visual style to a selection of other class symbols.

## Class & Interface symbols

The symbol style in class and interface symbols is edited on the “member style” tab of the symbol’s dialog. These are basically the same tabs as the diagram symbol style tab, except that fixed or non-appropriate elements have been removed.

## Package symbols (units)

The symbol style in a package symbol is incorporated in the main symbol tab. Here you control if contained classes and events are displayed. This feature is only available for units and classes that are (imported) in the model.

## Size and Alignment

### The Drawing Grid

The diagram editor's drawing grid is defined in the project options diagram style tab. It helps aligning symbols. All symbols are automatically snapped to the drawing grid. And for most symbols, the extent (bounds) is automatically adjusted to fit on the grid too. The MindMap node symbol allows enabling/disabling this "Bounds on Grid".

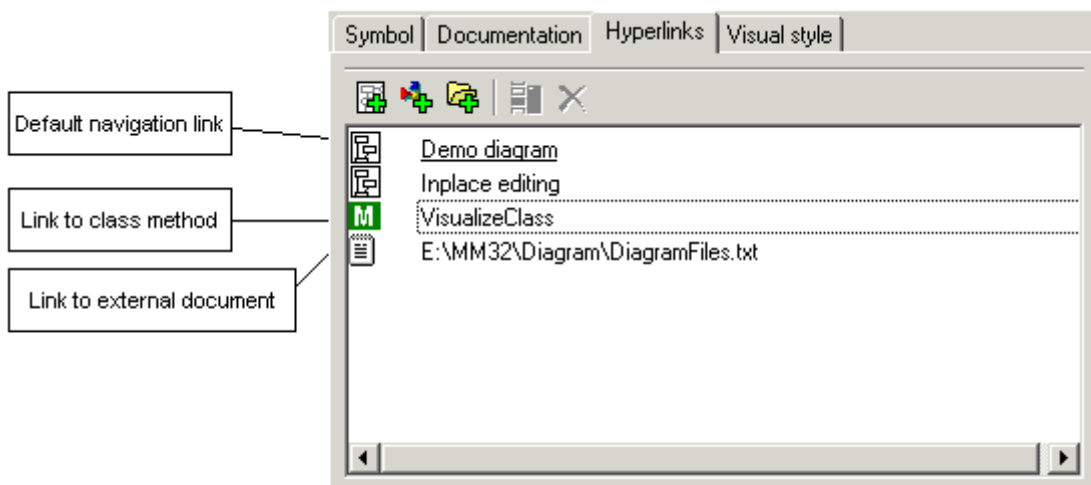
If you change the grid size, all symbols will most likely resize too. Because the grid it affects all diagrams and symbols, it is defined and saved per project.

### Align & Size Palette

The Alignment palette, which is available from the diagram editor tool bar, contains a set of functions to control alignment, auto sizing, text alignment and word break properties. These functions are similar to those in other applications such as Delphi, and are not explained in detail.

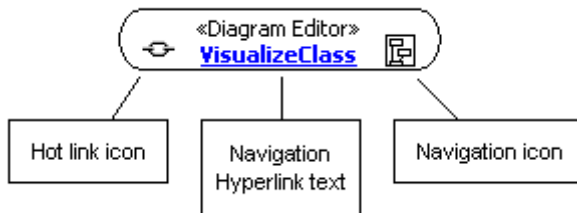
## Hyperlinks, navigation

Virtually all symbols can contain hyperlinks. Hyperlinks can point to other diagrams, code model entities (class, member, event, unit) or to external documents. Hyperlinks are created and maintained in the "hyperlinks" tab of a symbol's dialog. Although there can be many hyperlinks, only one is the default navigation link. This is the first link that supports navigation. The default navigation link is underlined in the hyperlinks list. Only HotLinks to code model entities usually have navigation disabled. Check chapter HotLinks to Code Model.



Most symbols will show a navigation icon next to their name if a navigation hyperlink is available. This is shown in the picture below. These icons can be suppressed by the visual style or the printing style. Also, if a symbol has a default navigation link, the symbol's name

will appear underlined and in the hyperlink color as defined in the visual style's palette. The picture below shows this. The Hotlink icon is explained in Paragraph "Hotlinks to the code model".



If the navigation icon is clicked, ModelMaker will follow the link and navigate to the object. If this is another diagram, the current diagram is saved and the referenced diagram is opened in the diagram editor.

If the link refers to a code model entity, ModelMaker will select the entity (class, member) and make the classes and members views visible. If the entity is a method, the method editor will also be made visible.

If the link refers to an external document, ModelMaker will perform a default "open" command on the filename or URL.

Clicking the hotlink icon (left of the symbol name) will edit the hot linked entity rather than the symbol.

If a symbol contains more than one hyperlink, you can navigate to the non-default hyperlinks by using the diagram editor's popup menu Navigate function. This contains a dynamic submenu with all hyperlinks available in the focused symbol.

## External documents

External documents are defined with the same alias / relative name mechanism as used for source files. Check chapter 'source aliases' in this manual for details. The use of aliases avoids the use of hard coded, machine dependent paths.

The standard shell "open" command is performed to navigate to an external document. This accepts all kinds of external documents such as executables or files that are associated with an application. For example "c:\temp\manual.doc" will be run MS word and open the document.

URLs to web pages or web sites are also valid. For example:

Alias=""

Relative filename = "http://www.modelmakertools.com"

will open a web browser and navigate to the ModelMaker Tools site. This can be used to navigate to html documentation etc.

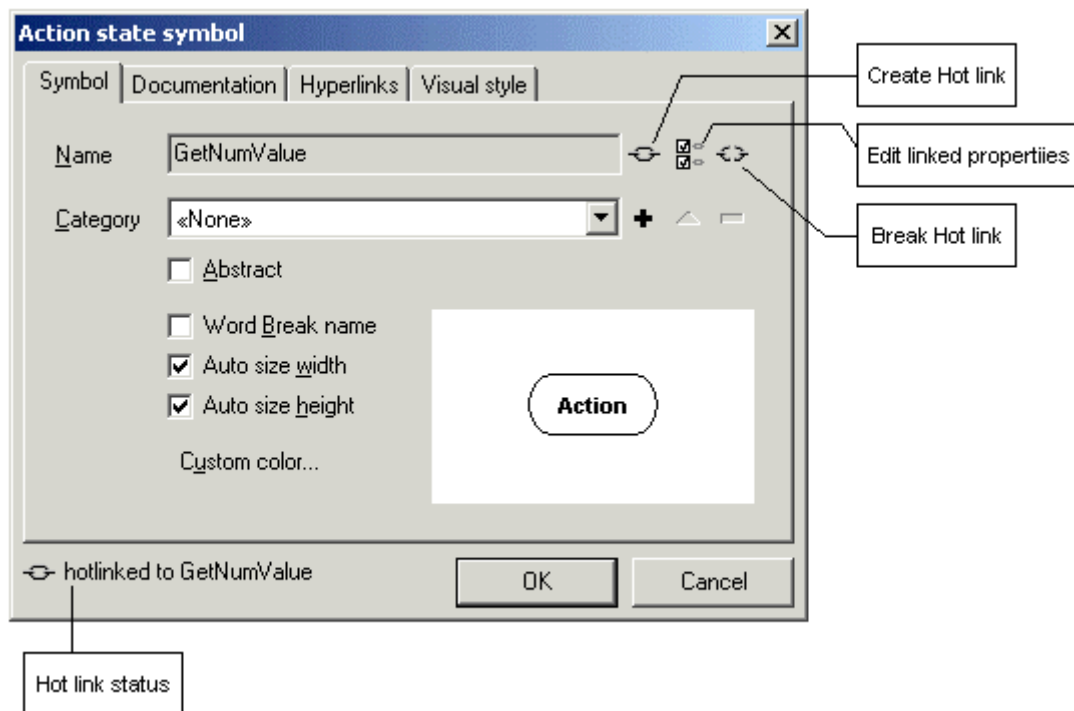
To refer to another ModelMaker model and start another ModelMaker instance, associate the ModelMaker project bundle extension \*.mpb with the ModelMaker executable in the Windows shell.

## Coupling Symbols to the Code Model

Most symbols and associations are linked or can be linked to entities in the code model. This is either hard coded or can be done manually by creating a HotLink. A hotlinked symbol will share name, documentation, one liner, “abstract” state and stereotype (category) with the linked entity. Modifying one of these properties in the symbol reflected to the linked entity and vice versa.

### HotLinks

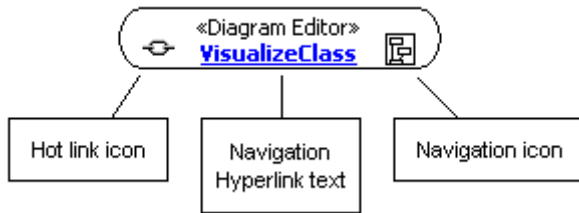
HotLinks are used to link a symbol or association to another entity, usually a code model entity like a class or a method. A hotlink is basically a navigation hyperlink that is (internally) marked as “hot”. Most symbol dialogs contain a set of buttons next to the name that allow creating and editing the hotlink. Here is an example from the Action State editor dialog.



Normally, action states are not coupled to any other entities. If you for example wanted to link an action state symbol to the TIntLabel.GetNumValue method, click the “Create HotLink” button and select the entity to link to. To break the hot link, click the Break hot link button. The icon at the bottom left of the dialog displays the hot link status. Not all properties need to be linked. To edit the linked properties, click “Edit linked properties”. The linked properties dialog lets you select which properties are linked.

If a symbol is hot linked, a hot link icon is displayed at the left of the symbol’s name. The following picture shows this. Hot link icons can be suppressed in the visual style and/or in the printing style.





If you click at the hot link icon in the symbol's name, the linked entity will be edited rather than the symbol.

If a symbol is hot linked to a code model entity, that entity will be selected if you click on the symbol or any of its (text) adornments. Unlike navigation through hyperlinks, this will not ensure that the associated view is made visible.

### Delete HotLinked entity

The Diagram Editor toolbar contains two delete tools: one to delete the symbol from the diagram, one to delete the symbol and the hotlinked entity. The last one will not only delete the symbol from the diagram but also remove the linked entity. You will be asked for confirmation before the linked entity is deleted.

## Specialized symbols and associations

Some symbols are linked to the code model by design. These symbols do not allow any other linking than the built in one.

### Class and Interface symbols

Class and Interface symbols are implicitly linked to a class in the code model. If the class is deleted from the code model, all class symbols linked to that class are removed from all diagrams.

### Property and Field associations

Similar to class symbols, property and field associations are hard linked to properties and fields in a class. The data type of these members must match the association target class.

### Shared Class Association

Shared class associations are associations between class symbols that allow greater flexibility than field and property associations. They do not need to be linked to existing code model members or classes. Because they are shared, they can be auto visualized if both source and target classes are being visualized on a diagram.

### Generalization relation

Generalization (inheritance) relations can be created between all symbols that are generalizable. In most cases they are not coupled the code model. Only if they connect two class or interface symbols, they are implicitly linked to the code model. Therefore, changing a generalization between two use cases does not affect the code model. But creating or changing a generalization between classes will be reflected in the code model by changing the class hierarchy.

### **Realization relation**

Realization relations (such as interface support) can be created between most symbols that allow realization. In most cases they are not coupled the code model. Only if they connect a class and an interface symbol, they are implicitly linked to the code model. Therefore, changing a realization between a package and interface does not affect the code model. But creating or changing a realization between a class and interface will be reflected in the code model by adding or removing interface support to that class.

### **Package (units)**

Units can be visualized as package symbols. Normal hot linking is used to achieve this. However, unit packages can display the contained classes and events. This is controlled by the diagram symbol style and the related options in the package editor dialog. The displayed content is read-only.

## **Documentation & OneLiners**

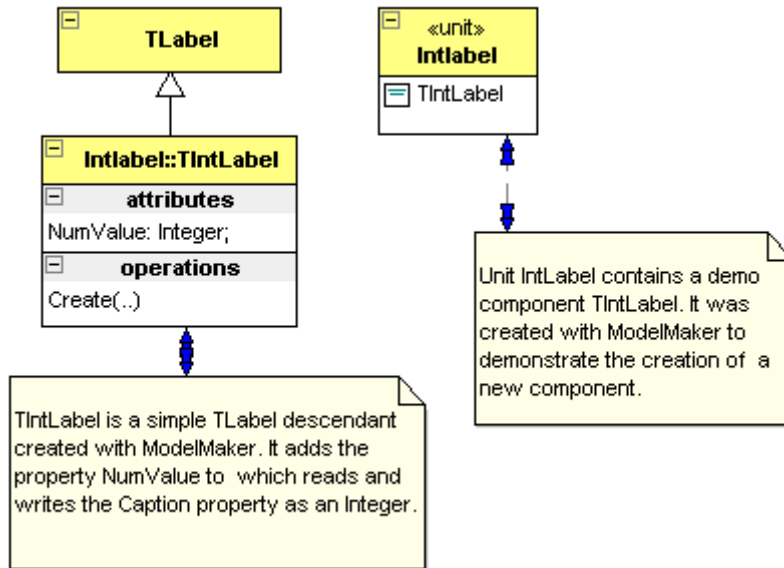
### **Floating Documentation view**

All symbols can be documented with a One Liner and multi line documentation. Most symbol editor dialogs contain a “documentation” tab. The standard documentation view cannot be used to edit the symbol’s documentation because the diagram editor and documentation view cannot be visible at the same time. The floating documentation view however is coupled to the diagram editor’s focused symbol. This view can conveniently be used to edit symbol documentation.

If a symbol is hotlinked to a code model entity, the symbol’s documentation and one liner will be linked to that of the entity. Changing it in the entity will change it in the symbol and vice versa. This is controlled by hot links and the hot linked properties.

### **Linked Annotations**

Annotations can visually be linked to symbols with documentation links. These links can be either passive or automatically link documentation or one liner. Here is an example from the Getting started demo in this manual.



The annotations in this example are coupled in auto documentation style; the blue double arrows on the link paths show this. Changing the annotation text (for example by in place editing) will change the symbol’s documentation. In the example the class symbols are implicitly linked to the code model classes. This means that editing the annotation text will modify both the symbol and class documentation. This also works the other way round: if the code model class’s documentation is changed, the annotation text will be updated.

To change the style of a documentation link, double click on the link path.

## Diagram Editor

### Properties

The environment options “diagrams” tab controls the diagram editor’s properties. These include: printing style, hint feed back, background color, grid style and color etc.

### Keyboard and Mouse control

The diagram editor’s keyboard short cuts are:

<i>Scroll and Move</i>	
Up/Down/Left/Right	Scroll
PageUp/PageDown/Home/End	Scroll one page up/down
Ctrl+PageUp/PageDown	Scroll to top/bottom
Home/End	Scroll one page left/right
Ctrl+Home/End	Scroll to left, right side of page

<i>Zooming</i>	
Numeric + / -	Zoom In/Out by 10%

Ctrl+Shift+I	Zoom in
Ctrl+Shift+U	Zoom out

<i>Editing</i>	
Escape	Cancel operation or select containing (parent) symbol
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo
F2	Rename (invoke inplace editor)
Ctrl+Up/Down/Left/Right	Move selected symbols
Ctrl+C/V/X	Copy/Paste/Cut selection
Ctrl+Alt+C/V	Copy / Paste visual style
Ctrl+Shift+C/V	Copy / Paste symbol style
Del	Delete selection
Ctrl+Del	Delete All (clear diagram)
Ctrl+A	Select All
Ctrl+P	Print Diagram
Ctrl+Alt+P	Print Preview Diagram
F12	Toggle full screen mode

<i>Navigation</i>	
Ctrl+U	Navigate Up; select parent diagram
Ctrl+B	Navigate Backward
Ctrl+F	Navigate Forward

<i>Mouse selection</i>	
Click	Select exclusive
Shift+Click	Toggle selected state, include in selection
Drag	Multiple Select( lasso selection)
Shift+Drag	Extend selection by lasso selection
Ctrl+Drag	Parent selection (only select symbols within parent, excluding the parent). Similar to Delphi IDE from designer

<i>Mouse Wheel control</i>	
Wheel	Scroll up/down
Shift+Wheel	Scroll left/right
Ctrl+Wheel	Zoom in/out

# Drag & Drop and conversions

ModelMaker extensively supports drag and drop between the main model views (classes, members, units, diagrams etc). Each major model view can act as a drag source of entities and as drop target for entities dragged from most other views. The details of each combination are described in the next paragraphs. Since there are always two views cooperating in a drag /drop operation (source and target) the details of conversions are described for the target view only.

## Classes view

### Internal (tree mode)

- Change inheritance.
- Apply interface support, press Control to invoke interface wizard

### Internal (list mode)

- Apply interface support, press Control to invoke interface wizard

## Source

Acts as a source for classes (and interfaces)

- Drag a class or interface to **diagram editor** for instant visualization.
- Drag a class to a **code editor** to insert its name as text.

## Target

Accepts members, local vars, procedures and event types:

- Drop **members** from the **members view** on a class to copy them to the target class. Press Shift (before releasing) to Move rather than copying the members. If a property is copied or moved, it's read and write access members are also copied, even if not included in the dragged members. Restrictions that apply for interfaces are automatically applied: fields are ignored, property write access is restricted and visibility is made default when dropping a class's members on an interface.
- Drop **procedures** from **Method Local Code Explorer** to convert them to new methods.
- Drop **procedures** from **Unit Code Explorer** to convert them to new methods.
- Drop **local vars** from the **Method Local Code Explorer** to convert them to new fields. Press shift on dropping to move rather than copy the var. Since interfaces cannot contain fields, local vars cannot be dropped on interfaces.

- Drop **event type definitions** from **Event library** view and **Units view** (tree mode) to add an event handler or event property using the dragged event as a template. On dropping a popup menu offers the available options.

## Members view

### Internal

- In custom order rearrange mode (Ctrl+R toggles this mode), member custom order can be arranged.

### Source

Acts as a source of members.

- Drag members the **classes view** to copy them to a class or interface.
- Drag members to a **class** in the **units view** (tree mode only). Similar to dragging members to the classes view
- Drag a field or method to the **Method Local Code Explorer** to convert it to a local var.
- Drag a method to the **Unit Code Explorer** to converted it to a local procedure.
- Drag a method to the **Method Implementation view** (toolbars, or tabs) to “pin” the method.
- Drag a method(s) to the **Event Library view** to create new event types using the methods as template.
- Drag a member to a **code editor** to insert its name as text.

### Target

Accepts code sections, local vars, (local) procedures, text, event type definitions and text.

- Drop **code sections** from the **Method Implementation Section list**. The dropped section is copied to the target method. Only if dropped on a method.
- Drop a **Local var** from the **Method Local Code Explorer**. If a var is dropped on a method, the var will be copied to the target method. Press shift to move the var rather than copying it. If a var is dropped on any other member or empty space in the member list, the var is converted to a new field. Drag the vars root node in the method local code explorer instead of a single var to drag all all vars at once.
- Drop a **local procedure** from the **Method Local Code Explorer**. If a local procedure is dropped on a method, it will be copied to that method. Press shift to move rather than copy it. If a local procedure is dropped on any other member or empty space in the member list, the local procedure is converted to a new method.
- Drop a **local procedure** from the **Unit Code Explorer**. Acts similar as local procedures dragged from the Method Local Code Explorer.

- Drop **Event type definitions** from **Event Library View** and **Units view (tree mode)**. A popup menu will let you select between adding an event handler for an event type or creating an event property. Multiple event types may be dragged at once.
- Drop **Text** from the **code editors** on the “add field” “add method” “add property” and “add event” buttons in the toolbar. The corresponding member type will be created and its name will be set to the dragged text. Method parameters are extracted from the dropped text.
- Drop **Text** from **code editors** dropped on the member list acts similar as text dropped on the “add method” button.

## Units view

### Internal (tree mode only)

- Classes and Event types can be copied or moved between units and “Classes not assigned to any units”. Pressing Ctrl will copy rather than Move (default). Within the same unit classes and event types can be rearranged using drag and drop.

### Source

In tree mode acts as source for units, classes and event types. In list mode acts as source for units only.

- Drag a unit to the **diagram editor** to visualize that unit as a package.
- Dragging a class or interface to any other view is similar to dragging a class from the classes view.
- Dragging an **event type** is similar to dragging an event type from the event library view.
- Drag a unit, class or event to a **code editor** to insert its name as text.

### Target

Accepts members, members, (local) procedures, local vars (all in tree mode only) and event types (both modes).

- Drop an **event type** from the **Event Library view**. If dropped on a unit, this will add the event type definition to the unit. If dropped on a class this will add an event handler or event property using the dragged event as a template or alternatively add the event type to the unit. On dropping a popup menu offers the available options. Applies to both tree and list mode.
- Dropping entities on a class is similar to dropping on a class in the classes view (tree mode only).
- Event types contained by units do not accept dropped entities.

## Method Implementation view

### Method Local Code Explorer

#### Internal

- Rearrange local vars
- Rearrange local procedures

#### Source

Acts as a source for **local vars**. Dragging the vars root node will drag all vars at once. Press shift to move rather than copy / convert a local var.

- Drag a Local var (or the root “Vars” node) to a class in the **classes view** to convert it to a field.
- Dragging a Local var to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.
- Drag a local var to the **members view** to copy it to a method or add a new field in the class.
- Drag a local var to a **code editor** to insert its name as text.

Acts as a source for **(local) procedures**. Press shift to move rather than copy/convert a procedure

- Drag a local procedure to a class in the **classes view**. The procedure will be converted to a method.
- Dragging a local procedure to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.
- Drag a procedure to the **members view** to copy it to a method or add a new method.
- Drag a procedure to a **code editor** to insert its name as text.

#### Target

Accepts members, local vars, procedures and event types:

- Drop a **field** or **method** from **members view**. A field will be converted to a local var, a method will be converted to a local procedure. This is usually only relevant for “pinned” methods as selecting a member to drag it will automatically change the “current method”.
- Drop **text** from a code editor containing an “ident + “:” + type” list to convert the text to local vars.

### Method Implementation Section list

#### Internal

- Rearrange sections (drag up/down)
- Indent / unindent sections (drag left/right)



### Source

Acts as a source for code sections.

- Drag a section to a method in the **members view** to copy it to that method. Press shift to move rather than copy the section.

### Target

Does not act as external drop target.

## Method Implementation Code Editor

### Internal

- Rearrange text inside editor copy or move.

### Source

Acts as a source for text.

- Drag a **text** to the members view to add a new member, using the text as name (plus parameter list for methods).
- Drag a **text** containing an "ident + ":" + type" list to the Local Code Explorer to convert the text to vars.

### Target

Accepts all dragged entities and inserts the associated name at the drop point.

## Unit Code view

### Unit Code Explorer

#### Internal

No internal drag and drop support.

#### Source

Acts as a source for (local) procedures. Press shift to move rather than copy/convert a procedure.

- Drag a local procedure to a class in the **classes view** to convert it to a method.
- Dragging a local procedure to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.

- Drag a procedure to the **members view**. This is similar to dragging a local procedure from the Method Local Code Explorer.
- Drag a procedure to a **code editor** to insert its name as text.

### Target

Accepts methods and procedures.

- Drop a **method** from the **members view** to convert it to a module procedure.

### Unit Code Editor

Similar to Method Implementation view Code Editor.

## Event Library view

### Internal

Does not support internal drag and drop.

### Source

Acts as a source for event type definitions.

- Drag an event to the **members view** or on a class in the **classes view**. This will add an event handler or event property.
- Drag an event to the **units view**. If dropped on a unit, this will add the event type definition to the unit. If dropped on a class in the units view (tree mode) this is similar as dropping it on a class in the classes view.

### Target

Accepts methods.

- Drop a **method** from the **members view**. For each dropped method an event type will be created that takes the method's signature (name, parameter list, result type etc) as a template.

## Diagrams view

### Internal (tree mode only)

- In "sort hierarchical" mode, diagrams can be rearranged. Pressing Shift will change diagram hierarchy relations. Other modes: no internal drag drop support.

## Source

Does not act as external drag source.

## Target

Does not act as external drop target.

# Diagram Editor

## Internal

The selected editor tool controls the extensive internal drag drop support.

## Source

Does not act as external drag source.

## Target

Accepts classes and units, depending on the opened diagram.

- A **class** or **interface** dragged from the **classes view** or **units view** will be visualized as class or interface symbol, object flow symbol or role symbol. For class diagrams relations with other classes are automatically visualized. If three or more classes are dropped on a class diagram the “drop visualization wizard” will be invoked. This wizard allows selecting the visualization scheme and relations to visualize.
- A **unit** dragged from the **units view** will be visualized as package symbol. Relations with classes (contained) and other packages (uses and used dependencies) are automatically visualized.

# Customizing ModelMaker

Here are some links to customizing ModelMaker to your wishes. Most of them you'll find in the Environment and Project options dialogs. For details refer to the GUI reference, here are just a few:

- To adjust the appearance of ModelMaker use the Environment options
- To adjust prefixes of property access methods and fields: use Project Options|Coding style.
- To adjust the layout of the generated source code, use Project Options|Code Generation.
- To adjust the way ModelMaker imports source code: use Project options|Code Import tab.
- To adjust the in-source documentation generation and import settings, check the corresponding tabs in the project options.
- To define a basic diagram visual style: Project options|Diagram Style
- To define a basic symbol style (displayed members etc): Project options|Diagram Style
- For defining the code editor's shortcut keys, Use Environment Options|Shortcuts.

Here are some other links to customization:

- Add Version Control capabilities by using a plug in expert. (Check ModelMaker web-site for ready available third party VCS Experts)
- To adjust the unit template used for new units, refer to Adjusting the unit template, page 50.
- Create Parameterizable Code Templates for pattern like groups of members that appear in multiple classes and models.
- For creating model templates, refer to Model templates page 41.
- Define and use your own macros for use in code generation or in the code editor.
- Most views have special display settings that are controlled in their popup menu: In Members view you modify toolbar layout and sorting. In the Classes view you adjust navigation order and history etc.
- Use the MM OpenToolsApi to create you own experts.

# Integration with the Delphi IDE

ModelMaker is a stand-alone application and you don't need Delphi to run it. Integrating ModelMaker with Delphi's IDE will enable some additional features.

- ModelMaker will automatically update Delphi's code editor whenever a source file is (re) generated.
- You'll be able to access Delphi's on-line context sensitive help from within ModelMaker.
- Call Delphi's 'Syntax Check' "Compile" or "Build" commands from within ModelMaker.
- Open source files and locate the member selected in ModelMaker in the Delphi IDE from within ModelMaker

Inside the Delphi IDE integration experts add several features that enable smooth integration:

- Synchronize ModelMaker with the IDE: refresh import a unit and locate the current member.
- Add (multiple) files to a ModelMaker project

Although it is possible to integrate ModelMaker with all versions of Delphi and run multiple instances or versions of Delphi at the same time, it is recommended that only one is running when working with ModelMaker, as integration is based on a one-to-one connection.

For the same reason we suggest that you do not run multiple instances of ModelMaker. In the environment options General tab you'll find an option that will ensure this.

## Integration with Delphi 3 and higher

ModelMaker is integrated with Delphi 3 and higher by use of an integration expert. These experts are automatically installed by the setup program for the IDE versions you have installed on your PC. You can manually (un)install an IDE expert later using the ModelMaker environment options "Delphi IDE" tab.

The experts add a ModelMaker main menu item to the IDE's menu bar. The ModelMaker menu contains:

- Run ModelMaker, (if not already running)
- Jump to ModelMaker (Ctl+F11 in D3, Ctrl+Shift+M in D4 or higher) - this will activate ModelMaker and select the unit, class and member corresponding to the IDE's topmost editor's position. Note that inside MM the inverse command 'Locate in Delphi' - main menu "Delphi" or main tool bar 'Locate in Delphi' - which locates the member selected in ModelMaker in the Delphi editor. ModelMaker's 'Locate in Delphi' command also has shortcut Ctl+F11.
- Add to Model, adds the topmost unit in the Delphi editor to the current MM model.
- Add files to model, lets you select which files to import in a model. In D4 and higher you may select files contained in a project or files opened in the editor.
- Refresh in Model, will re-import the topmost unit just like add to model, but only if the unit is already in the model. If the unit is not in the model, the command is silently ignored.

- Convert to Model, creates a new model and adds the topmost unit to this new model.
- Convert project to Model, which creates a new model and adds all files in the current Delphi project to the new model.

Note that the file in the IDE editor will be **SAVED** prior to performing the actual command. Therefore these commands won't work on read-only (project) files.

## Delphi 4 and higher

The Delphi 4 and higher integration experts have additional commands in the IDE's ModelMaker menu.

- Open Model, opens the model associated with the top most file in the IDE editor. Check the web-site 'Tips' page for details.
- 'Enable Auto refresh' and 'Lock Auto refresh on Run'. These control the Auto refresh feature that is described in detail in chapter Auto Refresh Import, page 65.
- Version Control. This menu item is enabled if you integrated a Version Control system in ModelMaker using a (third party) VCS expert. The available commands depend on this VCS-expert. They usually include at a minimum Check-in and Checkout. Check the ModelMaker Tools web site for available VCS-experts.

Some additional tools and utilities

- Unit Dependency analyzer. This is the same tool as available inside ModelMaker. For a description, check the ModelMaker on-line help by pressing F1 in this tool in ModelMaker.
- Resource string wizard. This will scan a unit for hard coded strings and help in converting them to a section of resource strings or string constants.
- String to Resource string, similar to the Resource string wizard, but only handles the current string token in the editor.
- Shortcut wizard: checks the active form for duplicate keyboard hot keys in control captions like "&Apply this" and "&Surprise me" and suggests alternatives in case of conflicts.

In Delphi 4 and higher you can also add commands to the IDE toolbars. In Delphi's toolbar 'Customize...' dialog, you'll find these commands in the ModelMaker category - right click on Delphi's toolbar, go to 'all commands'.

## Delphi 4 and higher syntax highlighting scheme

In ModelMaker you can specify the syntax-highlighting scheme to mimic your settings in the Delphi IDE. Unlike Delphi 1/2/3, Delphi 4 and higher do not define the default color scheme in the registry unless you manually (re-)define it. If the syntax highlighting scheme in ModelMaker is set to "Delphi 4" or higher, it might display strange settings: anything could happen such as underlined, blue colored normal text.

In order to solve this problem, in Delphi 4+ go to environment options and on the Colors tab define all fore-and background colors you want by explicitly selecting them rather than relying on the 'use default' check boxes. The same applies for the font styles: you must explicitly

select them. After applying these settings and restarting ModelMaker, you should have the highlighting scheme you selected. The following entries should be explicitly defined: "Comment", "Identifier", "Number", "Plain text", "Reserved word", "String", "Symbol", "White space" and "Marked block".

## Uninstalling IDE integration experts

If after uninstalling ModelMaker you still get a message when starting Delphi 3 or higher saying: can't find ..\..\MMEXPT.DLL or similar, you must manually uninstall the ModelMaker integration experts.

To do this:

Either use the ModelMaker environment options 'Delphi IDE' tab to uncheck the IDE version you want uninstall, or

Run RegEdit.exe from the "Start" menu and go to

HKEY\_CURRENT\_USER\Software\Borland\Delphi\3.0\Experts

There should be an entry called ModelMakerExpert, to uninstall the expert you must manually remove that entry. Higher versions of Delphi have a similar registry entry

## Integration with Delphi 1 and 2

Integration with Delphi 1 and 2 does not offer the same functions as the Delphi 3 and higher integration experts. Only basic synchronization functions are supported. And the installer cannot activate the integration - you must install the integration yourself. Integration consists of two aspects:

1. Installing the unit MMINTF.PAS (in directory [installdir]\mmintf in your component library. Installing this unit will enable ModelMaker to:
  - Automatically update Delphi's code editor whenever a source file is (re)generated.
  - Access Delphi's on-line context sensitive help.
  - Call Delphi's 'Syntax Check' command from within ModelMaker.
2. Installing the utility UNITJMP.EXE in your Delphi Tools menu. This will enable you to jump from Delphi's code editor to the corresponding code in ModelMaker and perform some basic file related commands.

### Installing the integration unit in Delphi 1 /2

1. Start Delphi 1 /2
2. Add the unit ModelMaker\6.0\Mmintf\MMIntf.pas to your component library, just like you would do with any other component (refer to your Delphi user manual).  
In Delphi 1: Select menu 'Options|Install Components',  
In Delphi 2: Select menu 'Component|Install',  
select Add and browse to find the unit MMINTF.PAS in folder ModelMaker\6.0\MMintf\.

3. Don't be surprised that you won't see any changes in your VCL component palette after Delphi recompiled the VCL: there is no new component installed, just an integration link, which is not a component.

## Installing UNITJMP.EXE as a DELPHI 1 /2 IDE tool

Installation of UNITJMP is basically the same for Delphi 1 and 2. To install the UNITJMP.EXE utility, (refer to your Delphi user manual)

1. Start Delphi
2. Select menu 'Options|Tools...'. (*Delphi 1.0*)  
Select 'Add' and add a new tool, title it 'Jump &to MM'. (*Delphi 1.0*)  
Select 'Add' and add a new tool, title it 'Jump to &MM'. (*Delphi 2.0*)
3. Select 'Browse' to locate the UNITJMP.EXE file in the ModelMaker\6.0\BIN folder.
4. Select 'Macros' to pass the parameters '\$ROW \$EDNAME', the space is required; the single quotes (") should not be entered.
5. Select OK and Close to add this utility.

Now you'll be able to jump from Delphi's code editor to ModelMaker (which should be running) by pressing Alt+T+T (D1) or Alt+T+M (D2)

UnitJump can be installed more than once as a tool to perform different integration tasks, each time passing different parameters.

1. For automatic refreshing of the top most file in the IDE editor, pass parameters '-1 \$EDNAME', refer to "Refresh Import" for details. You could enter '&Refresh Import' as title.
  2. To add the topmost in the Delphi editor to the current model, use parameters '-3 \$EDNAME'.
  3. To create a new model and add the topmost in the Delphi editor to this model, use parameters '-2 \$EDNAME'. You could enter '&Convert to Model' as title.
- Yes: you have identical "tools" UNITJMP now; the only difference is the parameters passed.



# MMToolsApi primer

This chapter is a introduction on COM interfaces and using the MMToolsApi to create your own experts to ModelMaker's functionality. You should also check the MMExptDemo.dpr that demonstrates most aspects of an expert. If you have not noticed yet: in the ..\ModelMaker\demos directory there's a model MMToolsApi.mpb that contains two diagrams showing relations of the API. Do not use this model to re-create the MMToolsApi.pas, this file will be changed in future versions.

## Interfaces basics

Interfaces are like classes in the way that once you've got a pointer to an interface, you can call methods and read/write properties.

Assume for example you have an interface pointer CodeModel: IMMCodeModel. From the MMToolsApi unit you can see that this interface supports the ClassCount and Classes[idx] properties. Also you can see that Classes[idx] returns IMMClass interface pointers. You could now for example iterate the code model for classes and list their names in a list box

```
for I := 0 to CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

## Expert DLL basics

The big thing is now: how do you get the first interface pointer CodeModel, because once you've got hold of that, everything is easy. In the MMToolsApi there is a central access point called MMToolServices: IMMToolServices that is declared as a global var in unit MMToolsApi. This interface var is initially nil but gives access to all major aspects of the MM engine such as the CodeModel in the above example. ModelMaker Experts are dll's which are loaded dynamically (all experts must be placed in directory ..\installdir\experts. ) After loading the library MM looks for a procedure called MMExpertEntryProc, as defined in MMToolsApi.pas and calls this procedure passing the interface pointer of the actual MMToolServices object. You should store this interface pointer as it provides your central access the tools API. You can access this interface and read the CodeModel: IMMCodeModel interface as shown in this example

```
procedure EntryProc(const Srv: IMMToolServices); stdcall;
begin
  // here you get passed the interface pointer, store it to use later.
  MMToolServices := Srv;
end;
```

The entry procedure should be exported named `SMMExpertEntryProc` (`MMToolsApi.pas`). There's also an exit procedure, which is called upon termination of ModelMaker. In this procedure we use the previously stored `MMToolServices` as central access point:

```
for I := 0 to MMToolServices.CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

## MMToolsApi version control

The unit `MMToolsApi` is continuously under construction and version control is governed by an `ExpertVersionProc`. This is the first procedure ModelMaker attempts to call when loading the expert. In your expert you must export this function and return the `MMToolsApiVersion` constant as defined in unit `MMToolsApi.pas`. ModelMaker will reject any expert not exporting this procedure or experts that do not match the version of the `MMToolsApi` with which ModelMaker was created.

```
function ExpertVersion: LongInt; stdcall;
begin
  Result := MMToolsApiVersion;
end;

exports
  ExpertVersion name SMMExpertVersionProc;
```

## Interfaces and memory management

Memory management for interfaced classes is controlled by reference counting which is automatic done for you by the compiler. Even assigning an interface to a local var will increase the reference count automatically, as will assigning nil to the var or going out of scope (exiting a procedure) will decrease the reference-count again. This means you don't have to worry about freeing classes after creating them. In general all interfaces passed on from ModelMaker to you are objects existing in `MM.exe` space. All interface pointers you pass on to ModelMaker are objects you create (such as an expert) and ModelMaker will only have access through the interface pointer. Since these objects are reference counted the objects will disappear after MM and the expert both drop all references to it.

For example if you retrieve an `IMMClass` interface pointer from ModelMaker and use it in your expert code, the actual interface object will exist as long as you keep a reference to it for example by assigning to a global var. The actual class the `IMMClass` refers to, may be gone in ModelMaker due to user actions (delete the class, open a new model) but the interface object remains live until the expert drops all references. To check whether an interface is actually connected to a real class you could /should check the `Valid` property. If `Valid` is `False`, the actual class object does not exist anymore, just the interface object. The interface object will return default values in all functions (such as `GetName =`”).

## Adding an expert and menu items

After reading the COM interface basics, you might want to create something useful which initiates on user action. The `MMToolsApi` provides the `IMMExpert` mechanism for this. You can create an object implementing `IMMExpert` and register it in ModelMaker, again using the `MMToolServices`. The fun about `IMMExpert` is that it allows you to insert menu-items in the ModelMaker main menu bar Tools menu and some predefined pop-up menus. Each expert should support properties `Verbs`, `VerbCount` and `MenuPositions`. `VerbCount` defines the number of menu-items you want inserted and `Verbs` are the actual menu item Captions. `MenuPositions[...]` defines where to which menu an item should be added. If the user clicks one of these menu-items, the `Execute(Idx)` method of the expert will be called, where `Idx` is `[0..VerbCount -1]`

There are some more methods supported by `IMMExpert`, but these are the basics. In the `MMExptDemo.dpr` you'll find an object implementing `IUnknown` and `IMMExpert`. If you go through the associated code you'll see that it has only one Verb and does only one thing in `Execute`. This demo expert could serve very well as a base for all other experts.

How do you inform ModelMaker that you want such an expert to be installed? Well create it (instantiate the class) and register the interface pointer with ModelMaker using `MMToolServices.AddExpert(...)`. Reference counting will again take care of memory management. On shutdown you should remove your expert again using the index that was passed by `AddExpert`.

```
var
  ExptIndex: Integer = -1;

procedure MMExpertEntryProc(Srv: IMMToolServices); stdcall;
begin
  MMToolServices := Srv;
  ExptIndex := MMToolServices.AddExpert(TMyExpert.Create);
end;

procedure MMExpertExitProc; stdcall;
begin
  if ExptIndex <> - 1 then MMToolServices.RemoveExpert(ExptIndex);
end;
```

Suppose you want to create an expert that does two reports "Simple" and "Extended", you should create an object that returns `VerbCount = 2` and `Verb[0] = 'Simple'`, `Verb[1] = 'Extended'`. To add the items to the Tools menu, return `mpToolsMenu` in `GetMenuPositions`. Then in `Execute(Idx)` you check whether `Idx = 0` or `1` to either create the Simple (0) or extended (1) report. The actual report creation could be something like:

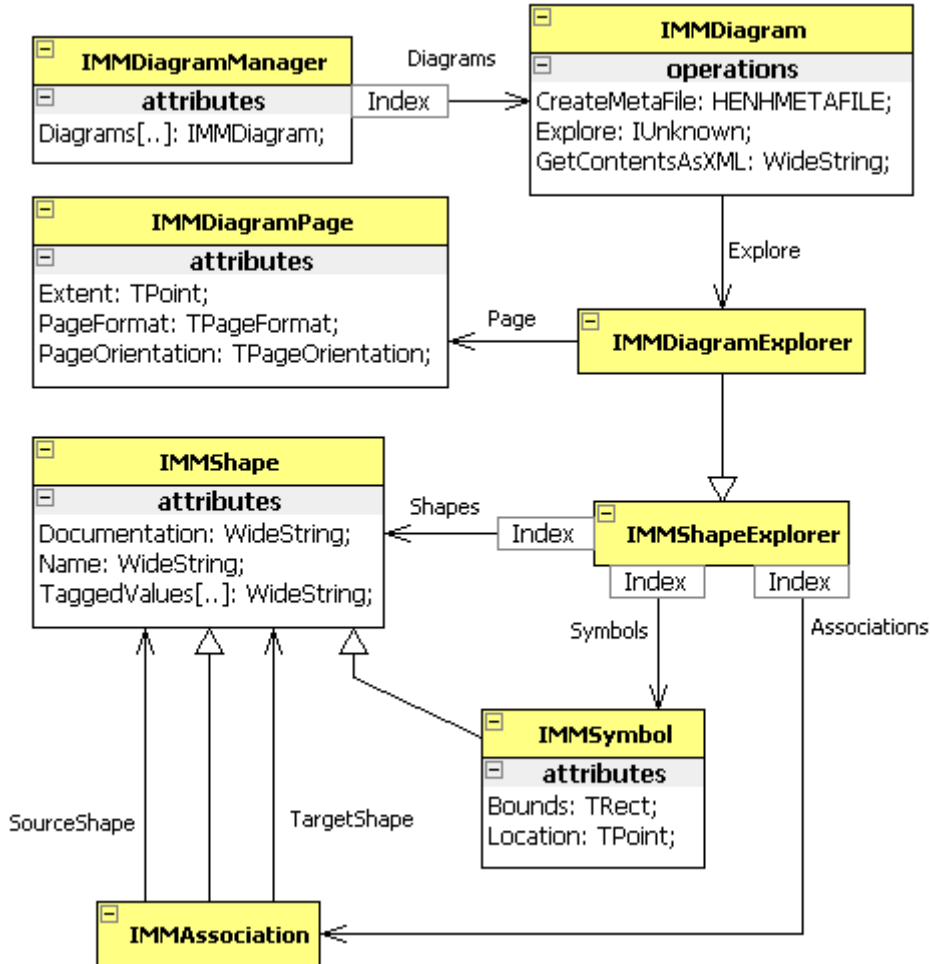
- Create a form containing a list box
- use
 

```
for I := 0 to MMToolServices.CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

 to fill the list box, and
- Call the Form's `ShowModal` method to show the current class list

## Accessing Diagrams through the API

The following picture explains how to access the Diagrams and their contents through the MM ToolsAPI.



The IMMDiagramManager property of the MMToolsServices is the entry point that gives access to all diagrams. The IMMDiagram interfaces are life pointers to the actual diagrams as displayed in the diagram list view. To get at the contents of a diagram, use the IMMDiagram.Explore method. This will create a Diagram Explorer similar to the diagram editor. The interface this method returns can be cast as an IMMDiagramExplorer. The definition of IMMDiagramExplorer and the symbols can be found in MMDiagramAPI.pas.

Note that each time you call Explore, as new explorer is created. And that the contents of two explorers are not linked. If you create two explorers and modify a diagram in explorer\_1, in explorer\_2 you won't see the change made in explorer\_1.

An explorer gives access to a diagram's symbols and associations. The Shapes property simply concatenates the symbols and associations properties. An IMMShape gives access to basic shape behavior: name, documentation and hyperlinks. An IMMymbol gives access to symbol specific properties like Bounds and Location (which associations do not have). IMMAssociation defines the association specific properties like SourceShape and TargetShape.

There are many ways to manipulate a diagram, the ModelMaker Tools demo expert shows a few examples like: create a sequence diagram and create an image containing a single class.

## Accessing Experts through scripting

ModelMaker 6 is a self-registering COM server that allows access to plug-in experts that support IDispatch. The ModelMaker type library can be found in the [installdir]\experts directory. It contains interface IApp that contains a single method: GetExpert.

```
IApp = interface(IDispatch)
  ['{D077CEC1-83F0-11D5-A1D2-00C0DFE529B9}']
  function GetExpert(const ExpertID: WideString): IDispatch; safecall;
end;
```

The parameter ExpertID is used to locate the expert based on the value returned by IMMExpert.ExpertID.

If your expert supports IDispatch and inherits from TAutoObject, you can access it for example in a java script like this (assuming your expert has a method named TestMethod which takes a single WideString parameter).

```
// Java script
var mm = new ActiveXObject("ModelMaker.App");
var api = mm.GetExpert("ModelMakerTools.ScriptingDemoExpert_10");
api.TestMethod("Hello World");
```

The above example will start ModelMaker or locate the active instance. Then locate the test expert and call it's method named "TestMethod".

This mechanism can be used to expose specific interfaces to scripting. For example, assume you have a reporter plug-in that supports this interface:

```
type
  IMyReporter = interface(IDispatch)
    procedure CreateReport(const ModelName, ReportName: WideString); safecall;
  end;

  IMyReporterDisp = dispinterface
    ['{F9BA1301-84EB-11D5-A1D2-00C0DFE529B9}']
    procedure CreateReport(const ModelName, ReportName: WideString); dispid 1;
  end;
```

You could then call this expert from a script to load a model and create a report. To learn more about disp interfaces and IDispatch, please check the Borland Delphi developers guide.



## Getting Started with ModelMaker

### Introduction

by Robert Leahey of [Thoughtsmithy](#)

#### Introduction

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker Training](#)

ModelMaker is an extremely powerful tool; as a result, like all powerful and extensive products, ModelMaker is complex and can be quite daunting to a new user. With that in mind, I've prepared this document as a starting place for those who see the potential in ModelMaker and wish to get up and running faster.

ModelMaker is often billed as a UML diagramming tool or a Delphi CASE tool. However, it is far more than a diagramming tool and the label "CASE tool" sometimes brings to mind some AI attempting to write code for you. A more accurate description of ModelMaker is that of a full-featured, extensible two-way code management tool with support for UML diagramming, design patterns, reverse engineering, refactoring, etc.

At its core, ModelMaker features an "active code model" — all your classes and associated elements (units, diagrams, documentation, event types, etc.) are objects within the model. ModelMaker provides various views into this active model, allowing you to manipulate it from within a class list, member list or a diagram... When you're ready, you can generate source code units from the model to be compiled by Delphi. Since the units are generated afresh each time (rather than MM working in the extant units), you can change various settings (such as code commenting options, code order, method instrumentation, etc.) and regenerate the units for a variety of purposes (instrumented code for debugging, heavily commented code for automated documentation generation, etc.). Note that these tutorials are in no way comprehensive; there is more than one way to do most operations; and there are many, many options in the environment. But the following topics should give you a small taste of what ModelMaker is capable of. For more information or assistance, visit [www.modelmakertools.com](http://www.modelmakertools.com) for downloadable documents. You can also request ModelMaker training services via Thoughtsmithy; see our [Training Services page](#) for details or contact us at [info@thoughtsmithy.com](mailto:info@thoughtsmithy.com).

1. [Importing Existing Code](#)
2. [Visualizing Imported Classes](#)
3. [Creating Classes within Diagrams](#)
4. [Diagramming Overview](#)
5. [Implementing classes/The Member List](#)
6. [Delphi Integration](#)
7. [Commenting Code / Macros](#)
8. [Differencing](#)
9. [The Unit Code Editor](#)
10. [Advanced Stuff](#)

## Getting Started with ModelMaker

### Tutorial 1 — Importing (and Generating) Code

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker Training](#)

**Warning:** *in order to avoid overwriting some of your production code, make a copy of the unit or units you are going to import before continuing.*

The fastest way to see ModelMaker in action is to import some of your existing code.

Open ModelMaker and create a new model by selecting the New button or File|New.

At this point we are going to tell ModelMaker to import a source file — it will reverse engineer the code it finds and add it to the model. Keep in mind that while ModelMaker's parsing engine is good, it can only deal with so much obfuscated code formatting before it chokes. For most people, this is not an issue — if MM encounters problems while importing your code, it will tell you.

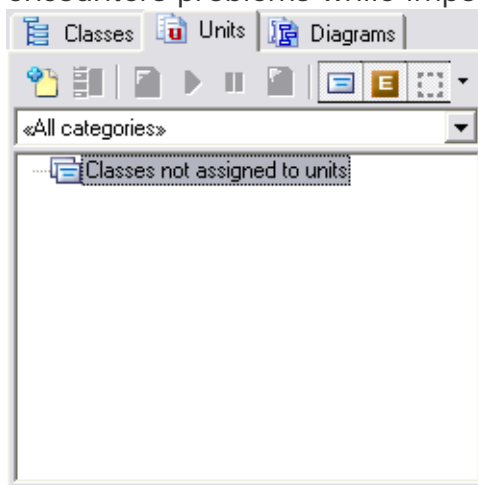


Figure 1 — the empty Units View.

There are multiple ways to import a file, but the quickest way is via drag & drop. In ModelMaker, select the Units View by pressing F4 or by selecting its tab. Find the unit you wish to import in Windows Explorer and then position ModelMaker so that you can drag the file into MM's main window. Drop the unit into the Units View — if you're in a new project that is empty, the Units View should look like figure 1.

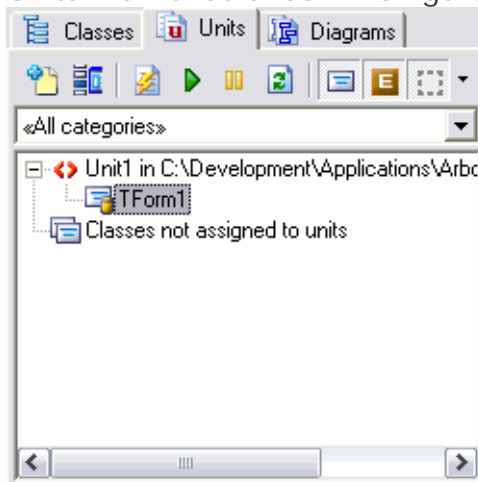


Figure 2 — Units View with imported unit.

Once you have dropped the unit, it will be reverse-engineered. The unit and any classes it contains will be added to the Unit List, which should now look something like Figure 2.

Don't worry if your icons look a little different — I've got some ModelMaker extensions installed that can change the appearance of my screens.

The top-level node in the Unit View treeview represents the unit you've just imported. Any child nodes are the classes, interfaces or event types that reside in that unit. You can double-click these nodes to bring up their editors.

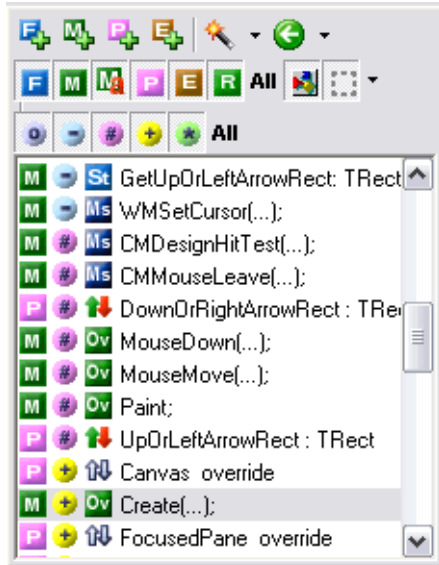


Figure 3 — the Member List.

Note that if you select a class in the Unit View, all of its members (properties, methods, etc.) will be displayed in the Member List, which looks something like figure 3.

We'll cover the Member List in [Tutorial 5](#); for now it's enough to notice that your class' members are enumerated here.

Press F3 to switch to the Classes View. You should notice here a hierarchical representation of the classes you've imported. Figure 4 is an example.

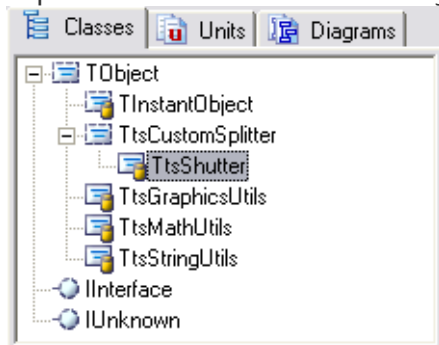


Figure 4 — the Classes View.

Notice that TObject, IInterface and IUnknown are always present. Note also the appearance of my TtsCustomSplitter class in Figure 4 — this is the ancestor of TtsShutter, which I've imported. However because I did not (yet) import the unit containing TtsCustomSplitter, this class must be represented as a "placeholder" (note the dotted lines around the class icon which denote placeholder status.) ModelMaker knows that TtsShutter descends from TtsCustomSplitter, but that's all it knows. If I wish to be able to use ModelMaker's features involving inheritance, I'd have to import TtsCustomSplitter's unit.

ModelMaker is also capable of importing your in-source comments and attaching them to their associated entities, but only if the comments and/or ModelMaker are properly set up to do so. If MM's Unit Code Editor (F7) shows a mass of unassociated code comments, fear not — we'll cover importing comments in [Tutorial 7](#).

This is an extremely abbreviated example of how to import existing code. You can



also perform imports using one of the two import buttons on the main toolbar.



– this button will import your code into a new model.



– this button will import your code into the existing model.

In both cases, the buttons will give you the option of importing from a specific path or from a Source Path Alias (which we have not covered here). See Source Path Aliases in the ModelMaker help file for more information.

### Generating Code

Obviously our ultimate goal is to create code which can be compiled in Delphi. ModelMaker uses its internal code model and applies the various formatting options you've selected to generate the specified units.



– the *Unlock Code Generation* button.



– the *Lock Code Generation* button.



– the *Generate* button.



– the *Enable Auto Generation* button.



– the *Disable Auto Generation* button.

In its default configuration, ModelMaker will not generate source code until you tell it to. To do so now, switch to the Units View by pressing F4 or by selecting the Units tab. Assuming you have one or more units containing classes (if you don't, go back, use your new-found skills and import some units), first, make sure that code generation is enabled by clicking the Unlock Code Generation button in the main toolbar (see sidebar), then select a unit to generate and press the Generate button (found in the Units View toolbar — see sidebar.) You should find that the specified unit has been regenerated by ModelMaker. If the unit was open in Delphi, you'll see that Delphi has already reloaded the new version of the file. This method is handy if you don't want to generate the code until you are ready.

However, if you wish, ModelMaker can be set to automatically regenerate a unit every time you make a change to that unit. This is called Auto Generation. You can enable or disable Auto Generation for each unit in your model by clicking either the Enable or Disable Auto Generation buttons in the Units View toolbar (see sidebar). Now, anytime you make a change to an Auto Generated unit, ModelMaker will regenerate the file. For that reason, care should be taken when enabling Auto Generation.

There are several other options and Delphi integration features which can effect when and how code is generated; we'll take a look at these options in [Tutorial 6](#).

**Return to the [Introduction](#).**

**Go on to the [next tutorial](#).**

## Getting Started with ModelMaker

### Tutorial 2 — Visualizing Imported Classes

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

To get an initial taste of ModelMaker's diagramming capabilities, let's visualize one of the classes we imported in the last [tutorial](#).

#### [Tutorial 1](#)

#### [Tutorial 2](#)

ModelMaker's split screen design allows you to choose from three different possible views on the left of the screen and eight on the right. The views on the left ("master views" I'll call them) are:

#### [Tutorial 3](#)

#### [Tutorial 4](#)

Classes View (F3)

#### [Tutorial 5](#)

Units View (F4)

Diagrams View (F5)

#### [Tutorial 6](#)

The views on the right ("detail views" or editors) are:

#### [Tutorial 7](#)

#### [Tutorial 8](#)

Method Implementation Editor (F6)

Unit Code Editor (F7)

#### [Tutorial 9](#)

Diagram Editor (F8)

#### [Tutorial 10](#)

Macro Editor (Shift + F6)

Design Patterns (Shift + F7)

#### [ModelMaker](#)

Differencing (Shift + F5)

#### [Training](#)

Documentation (Shift + F8)

Event Types (Ctrl + F8)

The various views can, of course, be accessed via their tabs, so you don't have to remember the keyboard shortcuts.

For our demonstration, first we'll want the Diagrams View, so press F5.

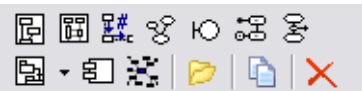


Figure 5 — the Diagram View toolbar.

From the Diagrams View toolbar, (figure 5) click the Add Class Diagram button.

Now that we have a new class diagram open, we'll want to see the Classes View on the left while editing the diagram on the right, so press F3. Depending on how ModelMaker's environment options are set, sometimes changing the master view can change the active editor, so if your diagram goes away, press F8 to bring it back.

There are several ways to add a class to a diagram, but again, the easiest way is to drag it from the Classes View and drop it into the diagram. So select a class from the Classes View, click-and-drag it into the class diagram and drop it there. With the default settings, ModelMaker will display only the class name in the symbol, like in figure 6.

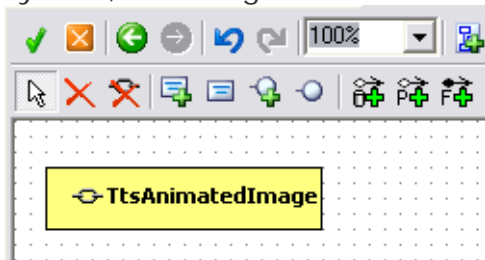


Figure 6 — a visualized class.

To cause the diagram to display some or all of the members of your class, we'll need to change the display style properties. ModelMaker's diagram display styles use an inheritance scheme. You can set properties for a symbol, but by default it will inherit the display properties of the diagram. The diagram will inherit a project's styles unless you override them, and likewise, a project will inherit the environment's styles.



Figure 7 — the class with attributes and operations.

In the interest of time, let's adjust the symbol display properties for the whole diagram. Double-click the diagram anywhere outside your class' symbol. This will bring up the Diagram Style dialog. Select the Symbol Style tab and uncheck the Project Member Type Filter checkbox. We're telling ModelMaker that we want to override the project's display style for this diagram. Now that the checkboxes under Custom Member Type Filter are enabled, check the Properties and Methods checkboxes. We've told the diagram we want it to display any class symbol's properties and methods. Click OK.

Your class symbol should now resemble something like figure 7 — a class with attributes and operations displayed. You should note that the class symbol has some active hot spots — the tiny minus sign in the upper left hand corner of each section will collapse that section (these appear when the class is selected); the little linked chain icon next to the class name will open the class editor if clicked.

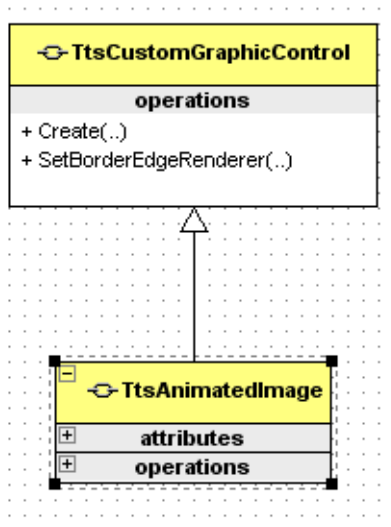


Figure 8 — automatically visualized generalization relation.

If your model contains an ancestor for the class you've just visualized, drag the ancestor to the diagram and drop it there. Notice that ModelMaker automatically visualizes the generalization relation between the two classes as in figure 8. This topic describes how to visualize an existing class from the code model. It is also possible to add a new class to the code model by adding it to the diagram. For more information, read the next [tutorial](#).

Return to the [Introduction](#).

Go back to the [previous tutorial](#).

Go on to the [next tutorial](#).

## Getting Started with ModelMaker

### Tutorial 3 — Creating Classes Within Diagrams

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker Training](#)

Now that we've [visualized existing classes](#), we can examine how to create a new class within a diagram. As we mentioned earlier, the master views in ModelMaker (Classes View, Units View and Diagrams View) are all just different views into the same active model. This means that we can add a model element in any of these views — adding those elements is just a slightly different process in each view. For our purposes, you can either use the class diagram from the previous topic or create a new one. As a point of convention, it's important to note the differences between a few of the tool buttons on the Class Diagram Editor toolbar (see figure 9). There are some buttons here that are common to all diagrams and we'll discuss these in the next [tutorial](#).



Figure 9 — part of the Class Diagram Editor toolbar.

These four buttons are used to add classes and interfaces to the diagram. The first button will create a new class, adding it to the model and the diagram — we'll discuss that in a moment. The second button allows you to choose a class that already exists in the model, and add it to the diagram. The third creates a new interface, adding it to both the model and the diagram while the fourth button will add an existing interface to the diagram. For this topic, we'll focus on the first button.

Click this first button (actually the fourth from the left). The cursor will change to indicate that you're adding a class.

Click on the diagram where you would like to add the class. The Class Symbol editor will be displayed (see figure 10).

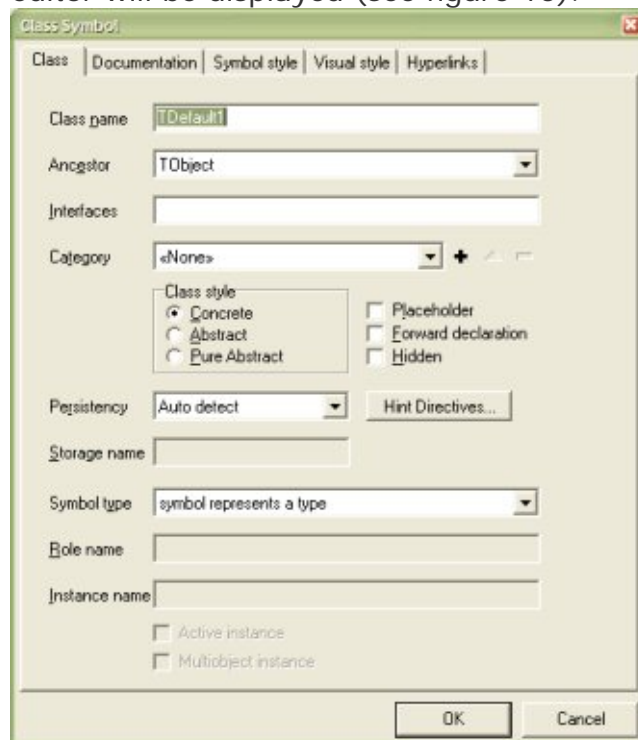


Figure 10 — the Class Symbol editor.

This editor allows you to define most every facet of a class — its role, appearance, documentation, etc. The Class Name field should be self-explanatory — enter your class' name here. Use the Ancestor drop-down to select an ancestor class. The other fields on this tab and the other tabs are optional; for now just set the first two fields and click OK. Your new class will appear in the diagram.

Press F3 (or select the Classes tab) to switch to the Classes View. If your diagram only contains the one class, drag some classes from the Classes View to the diagram to add them. Now notice that, as you select the various classes in your diagram, they become selected in the Classes View as well. In addition, their members are displayed in the Member List.

Your new class is rather empty, so let's add some properties. We need the Member List to add methods, events and simple properties and we'll be discussing that in [Tutorial 5](#), but we can easily add class-type properties and fields in the Diagram Editor, so let's try that now. In the Class Diagram Editor, click the Add Property to Model tool button (8th from the left). Now click-and-hold in the class symbol to which you are adding the property, and drag to the class symbol whose type the property will be. For example, if you have a diagram containing the classes TmyClass and TmyNewPropertyType, to add a property to TmyClass of type TmyNewPropertyType, select Add Property to Model, then click on the TmyClass symbol and drag to the TmyNewPropertyType symbol. The Property Association dialog will appear (see figure 11).

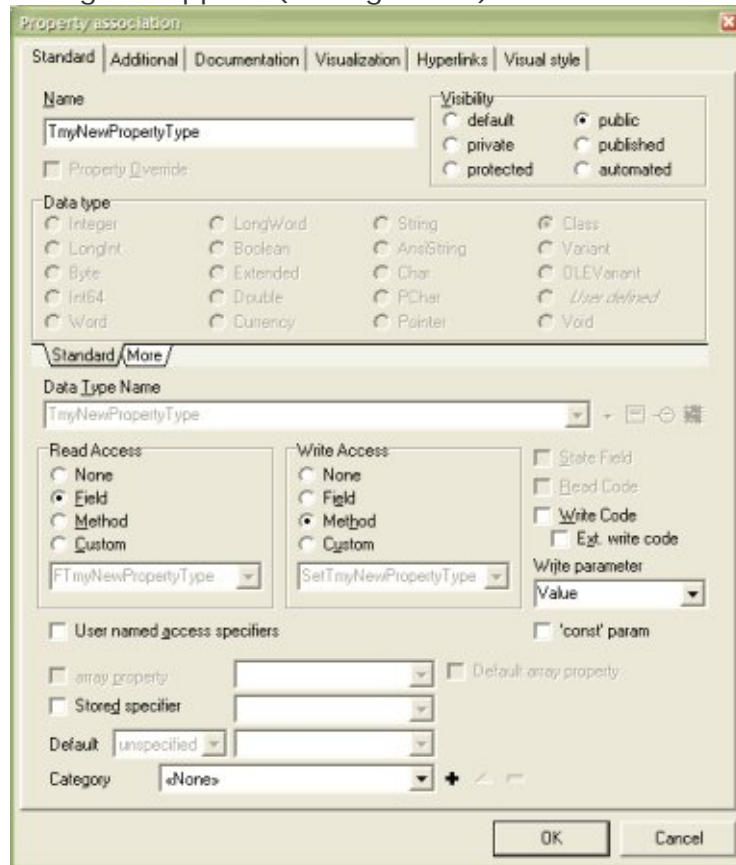


Figure 11 — the Property Association dialog.

This is a complex dialog, but don't be daunted. Most the fields on this tab are the values you're used to entering in Delphi by hand: Name is the name of the new property, this defaults to the name of the properties class type. Use the Visibility group box to set the new property's visibility. The Data Type fields are disabled since, by adding a class-type property in the diagram, we already know the data type. The Read and Write Access group boxes allow you to set the methods of access for this new property. This is a very powerful and timesaving feature of ModelMaker; any accessor methods or state fields needed by this property are added automatically and are always kept up-to-date by ModelMaker. If you change the name of this property, ModelMaker will automatically update its accessors

and/or state field.

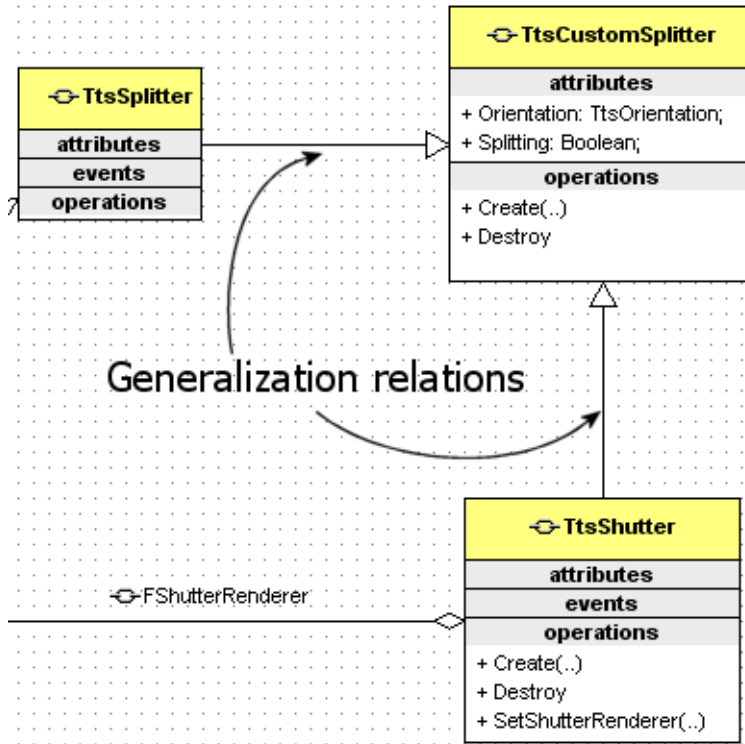


Figure 12 — Automatically visualized Generalization relations.

### Changing Relations in a Diagram

You may have noticed that when visualizing your classes, if a class is added to a diagram and that class' ancestor is also in the diagram, the Generalization relation is automatically visualized (see figure 12).

There are a great many options and wizards for advanced handling and automation of visualizations, but the Generalization relation is pertinent to our current topic, so for our final thought here, notice the following:

You can change the ancestor of a class, not just within the diagram but also for the whole model, from within a class diagram. Make sure you are editing a class diagram that contains 3 classes, one of which is descendant of another in the diagram. Open the Classes View (so you can see the results) by pressing F3. Click the Generalization arrow to select it. A "grabber" should appear at each end of the arrow (see figure 13).

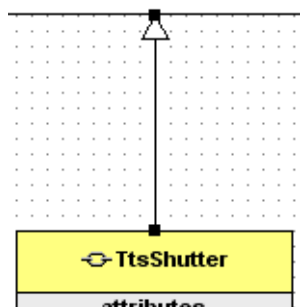


Figure 13 — A Generalization arrow, selected.

To change the class' ancestor, click the grabber at the arrowhead, and drag the arrowhead to a different class. Notice in the Classes View that the ancestor of your class has changed. This is a good example of the interconnectedness of the various editors — they are all looking into the same active code model. Note that this same method can be used to change the class type of a class' property within a diagram; drag the Property relation to a new class to change the property's type. This has been an incomplete view of diagramming and the class creation process. See [Tutorial 4](#) for an overview of diagramming in ModelMaker and [Tutorial 5](#) for an overview of the class implementation process.

Return to the [Introduction](#).  
Go back to the [previous tutorial](#).  
Go on to the [next tutorial](#).



## Getting Started with ModelMaker

### Tutorial 4 — Diagramming Overview

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker Training](#)

Before we begin our discussion of diagrams, remember the following: ModelMaker maintains an internal active code model from which all source code units are generated. The various views (including the Diagram Editor) are simply different ways of looking into that model. The diagrams are not static, stand-alone images; they are graphical representations of the internal model, thus diagram symbols that represent code entities are actually showing us a representation of the entity in the model not just a graphic symbol. It's important to remember that distinction lest you try to delete a class from a diagram and wonder why the class still exists in the model.

ModelMaker currently offers the ability to create and manage 10 different types of diagrams, most of which are UML compliant. While most of the diagrams have toolbars and concepts that are specific to their types, there is some common ground with which I will acquaint you here.

There are two parts to the diagramming interface; the Diagrams View (F5) and the Diagram Editor (F8). The reason that these are mentioned separately is that it is possible to view the Diagram Editor while displaying any of the three main views (Classes, Units or Diagrams). See [tutorial 2](#) for a listing of possible views and keystroke shortcuts. For the moment, display both diagramming views by pressing F5 (and if the Diagram Editor is not subsequently displayed, F8). If you are in a new project, with no diagrams, the Diagrams View should be blank and the Diagram Editor and toolbar should be grayed.



Figure 14 — the Diagrams View toolbar.

The Diagrams View toolbar (figure 14) contains buttons which will add each of the supported diagram types (Class, Sequence, Collaboration, Use Case, Robustness, State, Activity, Unit Dependency, Implementation and Mind Map).

To begin, let's add a new class diagram. Click the Add Class Diagram button (the upper-left button) in the Diagrams Views toolbar. Note that a new diagram is added to the Diagrams View tree and the diagram editor is activated with the new class diagram. You can rename the diagram by selecting its node and pressing F2. It's always possible to simply add a new diagram to the diagram list by clicking one of the "Add..." buttons. Try clicking some of the other "Add..." buttons now to add some other diagrams to the tree. Notice that as the various diagram types are added, the Diagram editors toolbar displays tool buttons specific to that diagram type. You also see some tool buttons that are common to all diagram types and we'll discuss these in a bit.

The Diagrams View is hierarchical; we can add some diagrams and rearrange them to be children of other diagrams. Right-click the Diagrams View and select Advanced Add... from the context menu. The resulting dialog allows you to specify a name for your diagram, select a diagram type and assign a parent diagram all in one step. Select one of your existing diagrams as the parent for your new creation and click OK. Note that the new diagram is added to the tree as the child of the existing diagram. This is a nice organizational feature and comes in handy in projects that contain numerous diagrams. For existing diagrams, it's possible to "re-

parent” them within the Diagrams View by simply holding down the Ctrl key and dragging the diagram to its new parent.

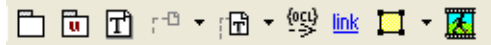


Figure 16.



Figure 15.

Next we'll look at some of the design elements common to all diagrams. Select any of the diagrams you have added and have a look at the toolbar of the Diagram Editor. For any diagram you select, you should see figure 15 at the far left of the toolbar and figure 16 at the far right.

Moving from left to right, these common tools are as follows:



– Select. Use this tool to select diagram elements.



– Delete from Diagram. This tool will delete a symbol from a diagram, but will not delete the entity it represents from the model.



– Delete Symbol and Linked Entity. This tool will delete a symbol from the diagram and will also delete the entity it represents from the model.



– Add Package. Use this tool to create a package symbol in your diagram. This symbol is “container” and can hold other symbols, much like TPanel in Delphi.



– Add Unit Package. Similar to the Add Package tool, this tool adds a symbol representing a physical Delphi unit to your diagram. The symbol automatically displays the entities contained within the unit.



– Add Annotation. This tool will create a standard UML annotation symbol in which you can enter text of any sort.



– Add Documentation Link. This tool and the next, Add Linked Annotation, have some subtle differences, but they can be summed up as follows: the three options in this drop-down button allow you to connect a symbol and an existing annotation symbol. Whereas the next tool, Add Linked Annotation, will create a new annotation symbol and link it to the selected symbol. The three options in the drop-down menu allow you to add your own annotation text, display the linked entity's documentation or its One Liner comment.



– Add Linked Annotation. This tool, like the one above, links documentation to a diagram symbol. Using this tool will add a new annotation symbol and link it to the selected symbol. The three options in the drop-down menu allow you to add your own annotation text, display the linked entity's documentation or its One Liner comment.



– Add Constraint Relation. This is rather self explanatory; use this tool to add a constraint relation between two symbols to the diagram.



– Add Hyperlink. This is a powerful tool. This symbol, when added, will allow you to add a hyperlink to your diagram which can link to another diagram, a code model entity and/or an external document.



– Add shape. Use this tool to add simple geometric shapes to your diagram.



– Add Image. This tool allows you to add an image from an external file to your diagram.

When trying to alter the appearance of your diagrams, keep in mind that double-clicking on a diagram symbol (including associations and relations) will display that symbol's editing dialog where you can change the visual attributes of the diagram symbol. However, before you start trying to format your diagrams one symbol-at-a-time, keep in mind the following: ModelMaker provides access to the display styles of diagrams and their symbols using an inheritance model. It is possible to set styles for diagrams at an environment level via the Diagrams tab in the Environment Options dialog; to override these settings for a particular project, you would do so in the Project Options dialog, available from the Options menu. To override the settings for a diagram, double-click the diagram to display the Diagram Style dialog. Likewise, double-click a symbol to display its Symbol Style dialog.

This should provide enough of a basic overview of diagramming in ModelMaker to get you started. See the online help for information about specific diagram types.

Return to the [Introduction](#).

Go back to the [previous tutorial](#).

Go on to the [next tutorial](#).

## Getting Started with ModelMaker

### Tutorial 5 — Implementing Classes/the Member List

by Robert Leahey of *Thoughtsmithy*

#### [Introduction](#)

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker](#)

#### [Training](#)

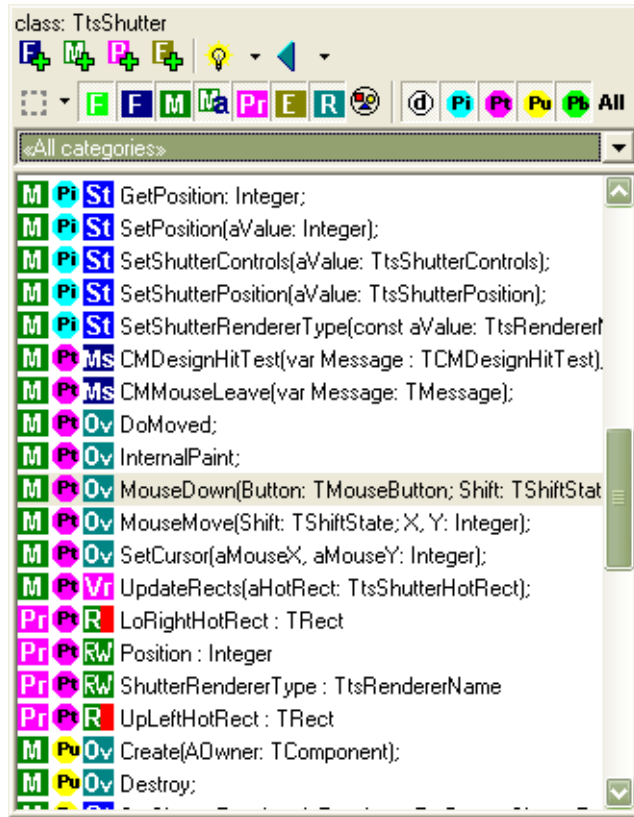


Figure 17 — the Member List.

Given the wide range of options available in adding class members, this step is more informative than tutorial.

Ok, we've looked around enough — it's time to get serious about implementing our classes.

The pane that you'll need to get to know well is the Member List (see figure 17). This window is found below the triumvirate of the Classes View/Units View/Diagrams View.

The Members List allows you to manage the members (fields, methods, properties and events) of your classes. Note that there are many options that control the appearance of the Member List so my screen shots may not look the same as the Member List in your copy of ModelMaker.

The first thing to notice is that the Member List is just that — a list of a class' members. Remember that ModelMaker keeps an active model of your code internally and that all of the various editors are simply different views into the model; the Member List is another example of that. Any changes you make here are simply changing the model, not the physical source code files (unless you have Auto Generation Enabled) until you [generate the source](#) from the model.

On the General tab of the Environment Options dialog (available from the Options menu) is a group box entitled Members Appearance. If you check the first three items, Display Type Bitmaps, Display Visibility Bitmaps and Display Info Bitmaps, you will see all three columns of graphics in your Member List (as shown in figure 17). Select all three options and we'll have a look at the various icons presented in

the Member List.

The first two columns in the Member List represent the member type and visibility:

<b>F</b> — Field	<b>Ⓢ</b> — default visibility
<b>M</b> — Method	<b>Pi</b> — private
<b>Pr</b> — Property	<b>Pt</b> — protected
<b>E</b> — Event	<b>Pu</b> — public
<b>R</b> — Method Resolution Clause	<b>Pb</b> — published

The third column displays a contextual icon depending on the member type. For methods, the icons represent bindings as follows:

<b>St</b> — static	<b>Vr</b> — virtual	<b>Ov</b> — override
<b>Ab</b> — abstract	<b>Dy</b> — dynamic	<b>Ms</b> — message

If the member is a property, the icons are as follows:

<b>RW</b> — Read/write	<b>R</b> — Read-only	<b>No</b> — Property override
------------------------	----------------------	-------------------------------

The Member List allows you to add and edit members as well. Here we will take a general look at that process.



Figure 18 — Add Members toolbar.

Near the top of the Member List, you should see the toolbar in figure 18. The first four buttons on this toolbar are for adding members; they are, from left to right, Add Field, Add Method, Add Property and Add Event. Clicking one of these buttons will display the appropriate editor for the member type you selected. You should notice some similarities in each of the editors: Name, Visibility and Type. Also common to each editor is the Documentation tab where you can add documentation for your new member. The documentation you add is attached to the member within ModelMaker's internal model and can be used in a variety of ways. For example, you can emit the documentation with your code as comments (see [Tutorial 7](#)) or link to it within a diagram as an annotation (see Add Documentation Link and Add Linked Annotation in [Tutorial 4](#)). It is also available via the ModelMaker OpenTools API (see [Tutorial 10](#)).

In addition to the common elements, each editor contains settings specific to its member type. For instance, in the Property Editor, you can control the property's read and write access while the Event Editor allows you to generate a dispatch method.

Here are some quick thoughts on each of the member types which should help you to get more comfortable with ModelMaker:

### Fields and Methods

If you're trying to add a property's state field or an access method, stop. One of the key advantages of ModelMaker is its automation, and the automation surrounding the creation of properties is high. See the next section for more information.

Make sure you visit the ModelMaker online help by pressing F1 in the editor dialogs. Some items of note are the Owned and Initialized options for fields and the Inheritance Restricted option for methods.

### Properties

As mentioned in the previous section, ModelMaker can greatly simplify the process of adding a property through its automation features. This is one of the reasons many people choose to work in ModelMaker as their code editor (as a Delphi IDE replacement); when adding a property by hand in Delphi, you write the declaration code for the property, its access methods, perhaps a state field and the implementation code for the access methods. When adding a property in ModelMaker, you add the property. Period.

Within the Property Editor you will find options for specifying the read/write access for the property (state field, method or none) as well as for generating some rudimentary read/write access code. Once you have set these options and clicked OK, ModelMaker adds not only the property, but the related access code. The generated code is "owned" by the property — if you change the name of the property or delete it, all the generated code is automatically renamed or deleted as well. Try experimenting with these settings and read in the online help for more information.

## Events

Adding an event is somewhat similar to adding a property; most of the overhead is managed for you by ModelMaker. In the Event Editor dialog, you select a name, event type (event types are managed in the Event Type Library, available in the Events tab) and visibility. You can also specify a dispatch method by checking the Dispatch checkbox. The generated method will be owned by the event similar to the way a property owns its access methods.

Note that if you wish your dispatch methods to automatically be prefixed (with "Do", for example) you can specify this and other coding style options in the *Coding Style* tab of the *Project Options* dialog under the *Options* menu.

Return to the [Introduction](#).

Go back to the [previous tutorial](#).

Go on to the [next tutorial](#).



## Getting Started with ModelMaker

### Tutorial 6 — Delphi Integration

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

While some developers use ModelMaker as a complete Delphi IDE replacement, most seem to prefer to use ModelMaker and Delphi simultaneously — switching back and forth as needed. To support this, ModelMaker offers extensive Delphi integration features, which we will examine here.

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker Training](#)

#### Delphi Controls ModelMaker

ModelMaker installs a Delphi integration expert into the Delphi IDE, which allows you to control ModelMaker somewhat from within Delphi.

First, start Delphi if it is not running, and have a look at the ModelMaker menu in the main Delphi menu bar. Notice that the first menu item allows you to launch ModelMaker. Also notice that most of the items in the ModelMaker menu require ModelMaker to be running — if it is not, those items will be disabled.

Launch ModelMaker now, if it is not running, by selecting the first menu item.

**Warning:** *in order to avoid overwriting some of your production code, make a copy of the unit or units you are going to import before continuing.*

The first thing we'll do is to create a new project and import into it some existing classes, all from within Delphi. Open, in Delphi, a project that you'd like to import into ModelMaker. From the ModelMaker menu, select *Convert Project to Model*. The ModelMaker window should come to the front, most likely asking whether to save your current model. After you answer that dialog as appropriate, ModelMaker will create a new project and import the Delphi project's units.

Some of the other menu items that operate similarly are *Add to Model*, which causes the active unit in Delphi to be imported into the current model in ModelMaker; *Add Files to Model*, which allows you to select units to be added to the current model in ModelMaker and *Convert to Model*, which will create a new model in ModelMaker and import the active Delphi unit into it.

Two other noteworthy menu items in the Delphi ModelMaker menu are *Jump to ModelMaker* and *Refresh in Model*. *Jump to ModelMaker* will bring ModelMaker to the front and attempt to find the current code block within the model. *Refresh in Model* will cause ModelMaker to re-import the active unit into the current model.

#### ModelMaker Controls Delphi

If you find that you like editing your code in ModelMaker, and prefer to have ModelMaker control Delphi, you have that option. Make sure that you are familiar with the process of generating code from within ModelMaker, by reading [tutorial 1](#), then examine the Delphi portion of ModelMaker's main toolbar (see figure 19).



Figure 19 — Delphi control buttons.

The first button, *Generate*, you should already be familiar with.

The second button, *Generate and Perform Default Compile Action*, will cause the current unit to be generated, and will request that Delphi perform one of three actions: Syntax Check, Compile or Build. This default compile action can be set on the next button's drop-down menu.

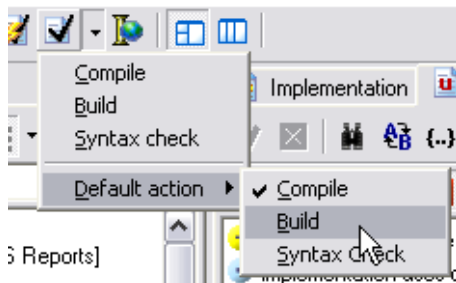


Figure 20 — Default Compile Action drop-down menu.

The third button (its hint will vary depending on the default compile action), will cause Delphi to perform the currently selected default action. This default is set in this button's drop-down menu (see figure 20); after setting a default, that action will be taken any time this button or the previous button is clicked. Note: these Delphi actions will be enacted upon whatever the currently loaded Delphi project may be.

The fourth button, *Locate in Delphi*, will bring Delphi to the front and attempt to locate the current code block in the code editor. It will open the current unit in Delphi if necessary.

### Two-way Integration

The most complete integration can be achieved through ModelMaker's two-way integration, although a warning must be issued here: normally ModelMaker works with a safety net of allowing you to determine when code will be generated and when it will be refreshed into the model. Even if you enable auto-generation (see [tutorial 1](#)), you still have control over when code is re-imported if it has been changed outside of ModelMaker. If you enable the two-way integration, you will lose this control and code will automatically be generated and refreshed without warning. If there are changes to the code you do not wish to overwrite without warning, you should be careful using this option.

Enabling this feature basically causes ModelMaker to generate your units any time they change in ModelMaker and to re-import them any time they change outside of ModelMaker. To use this feature, do the following:

In ModelMaker, unlock code generation. Select one or more units to auto-generate and select the *Enable Auto-Generation* button for each.

Select *Environment Options* from the *Options* menu in ModelMaker. In the General tab, under Code Generation and Import, check the *Auto Refresh from IDE* and *Refresh Keeps Units Enabled* check boxes.

In Delphi, select *Integration Options* from the ModelMaker menu. In that dialog, select *Enable Auto Refresh*.

Open one of your auto-generated units in Delphi by clicking ModelMaker's *Locate in Delphi* button. If you want, arrange your ModelMaker and Delphi code editor windows so that you can see both at the same time. Now go back to ModelMaker and change something in that unit, then save the model. You should see the unit update in Delphi automatically. Now change something in the unit in the Delphi code editor, then save the unit. The unit will be refreshed in ModelMaker. Viola! Two-way code editing integration.

Another warning: if your code comment importing options are not correctly set up, you may see "comment creep" or worse, you might see comments deleted without warning. Be very careful using this two-way feature until you understand how to set up code comment importing options. For more information on that, see the next tutorial.

Return to the [Introduction](#).

Go back to the [previous tutorial](#).

Go on to the [next tutorial](#).





## Getting Started with ModelMaker

### Tutorial 7 — Commenting Code / Macros

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

The text for this tutorial is not yet complete.

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker](#)

#### [Training](#)



## Getting Started with ModelMaker

### Tutorial 8 — Differencing

by Robert Leahey of *Thoughtsmithy*

**Introduction**

The text for this tutorial is not yet complete.

**Tutorial 1**

**Tutorial 2**

**Tutorial 3**

**Tutorial 4**

**Tutorial 5**

**Tutorial 6**

**Tutorial 7**

**Tutorial 8**

**Tutorial 9**

**Tutorial 10**

**ModelMaker**

**Training**



## Getting Started with ModelMaker

### Tutorial 9 — The Unit Code Editor

by Robert Leahey of [Thoughtsmithy](#)

#### [Introduction](#)

The text for this tutorial is not yet complete.

#### [Tutorial 1](#)

#### [Tutorial 2](#)

#### [Tutorial 3](#)

#### [Tutorial 4](#)

#### [Tutorial 5](#)

#### [Tutorial 6](#)

#### [Tutorial 7](#)

#### [Tutorial 8](#)

#### [Tutorial 9](#)

#### [Tutorial 10](#)

#### [ModelMaker](#)

#### [Training](#)



## Getting Started with ModelMaker

### Tutorial 10 — Advanced Stuff

by Robert Leahey of [Thoughtsmithy](#)

[Introduction](#)

The text for this tutorial is not yet complete.

[Tutorial 1](#)

[Tutorial 2](#)

[Tutorial 3](#)

[Tutorial 4](#)

[Tutorial 5](#)

[Tutorial 6](#)

[Tutorial 7](#)

[Tutorial 8](#)

[Tutorial 9](#)

[Tutorial 10](#)

[ModelMaker](#)

[Training](#)

[Appendix A](#) - Extra Delphi Tools by the Author

[Appendix B](#) - Extra Delphi Tools from Other Sources

[Appendix C](#) - Free Companion Books on Delphi

[Index](#)

[List of Figures](#)

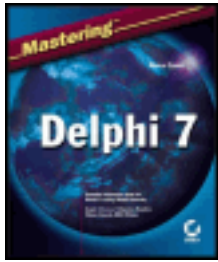
[List of Tables](#)

[List of Listings](#)

[List of Sidebars](#)

**Team LiB**

**NEXT ▶**



## Mastering Delphi 7

by Marco Cantu

Sybex © 2003 (1011 pages)

The best Delphi resource--now updated and expanded.

ISBN:078214201X

[Companion Web Site](#)

### Table of Contents

[Mastering Delphi 7](#)

[Introduction](#)

#### **Part I - Foundations**

[Chapter 1](#) - Delphi 7 and Its IDE

[Chapter 2](#) - The Delphi Programming Language

[Chapter 3](#) - The Run-Time Library

[Chapter 4](#) - Core Library Classes

[Chapter 5](#) - Visual Controls

[Chapter 6](#) - Building the User Interface

[Chapter 7](#) - Working with Forms

#### **Part II - Delphi Object-Oriented Architectures**

[Chapter 8](#) - The Architecture of Delphi Applications

[Chapter 9](#) - Writing Delphi Components

[Chapter 10](#) - Libraries and Packages

[Chapter 11](#) - Modeling and OOP Programming (with ModelMaker)

[Chapter 12](#) - From COM to COM+

#### **Part III - Delphi Database-Oriented Architectures**

[Chapter 13](#) - Delphi's Database Architecture

[Chapter 14](#) - Client/Server with dbExpress

[Chapter 15](#) - Working with ADO

[Chapter 16](#) - Multitier DataSnap Applications

[Chapter 17](#) - Writing Database Components

[Chapter 18](#) - Reporting with Rave

#### **Part IV - Delphi, the Internet, and a .NET Preview**

[Chapter 19](#) - Internet Programming: Sockets and Indy

[Chapter 20](#) - Web Programming with WebBroker and WebSnap

[Chapter 21](#) - Web Programming with IntraWeb

[Chapter 22](#) - Using XML Technologies

[Chapter 23](#) - Web Services and SOAP

[Chapter 24](#) - The Microsoft .NET Architecture from the Delphi Perspective

[Chapter 25](#) - Delphi for .NET Preview: The Language and the RTL

## Back Cover

Whether you're new to Delphi or just making the move from an earlier version, *Mastering Delphi 7* is the one resource you can't do without. Practical, tutorial-based coverage helps you master essential techniques in database, client-server, and Internet programming. And the insights of renowned authority Marco Cantú give you the necessary knowledge to take advantage of what's new to Delphi 7-- particularly its support for .NET.

Coverage includes:

- Creating visual web applications with IntraWeb
- Writing sockets-based applications with Indy
- Creating data-aware controls and custom dataset components
- Creating database applications using ClientDataSet and dbExpress
- Building client-server applications using InterBase
- Interfacing with Microsoft's ADO
- Programming for a multi-tiered application architecture
- Taking advantage of Delphi's support for COM, OLE Automation, and COM+
- Taking advantage of Delphi's XML and SOAP support
- Implementing Internet protocols in your Delphi app
- Creating UML class diagrams using ModelMaker
- Visually preparing reports using RAVE
- Using the Delphi language to create your first .NET programs

### About the Author

Marco Cantú is an internationally known programming author and teacher who specialize in Delphi development and XML-related technologies. Author of the best-selling *Mastering Delphi* series, he teaches advanced Delphi classes, speaks at conferences worldwide, and writes about Delphi programming in print and online magazines.

# Mastering Delphi 7

Marco Cantù

**Associate Publisher:** Joel Fugazzotto

**Acquisitions Editor:** Denise Santoro Lincoln

**Developmental Editor:** Brianne Agatep

**Production Editor:** Kelly Winqvist

**Technical Editor:** Brian Long

**Copyeditor:** Tiffany Taylor

**Compositor:** Rozi Harris, Interactive Composition Corporation

**Proofreaders:** Nancy Riddiough, Emily Hsuan, Leslie Higbee Light, Monique Vandenberg, Laurie O'Connell, Eric Lach

**Indexer:** Ted Lau

**Book Designer:** Maureen Forys, Happenstance Type-O-Rama

**Cover Designer:** Design Site

**Cover Illustrator:** Tania Kac, Design Site

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. The author (s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2002115474

ISBN: 0-7821-4201-X

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.

Mastering is a trademark of SYBEX Inc.

Screen reproductions produced with Collage Complete.

Collage Complete is a trademark of Inner Media Inc.

**TRADEMARKS:** SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

*To the late Andrea Gnesutta,  
a friend the Italian Delphi community prematurely lost.*

## Acknowledgments

This seventh edition of *Mastering Delphi* follows the seventh release of a Delphi development environment by Borland,



a revolution started in the winter of year 1994. As it has for many other programmers, Delphi (and its Linux twin, Kylix) has been my primary interest throughout these years; and writing, consulting, teaching, and speaking at conferences about Delphi have absorbed more and more of my time, leaving other languages and programming tools in the dust of my office. Because my work and my life are quite intertwined, many people have been involved in both, and I wish I had enough space and time to thank them all as they deserve. Instead, I'll just mention a few particular people and say a warm "Thank You" to the entire Delphi community (especially for the Spirit of Delphi 1999 Award I've been happy to share with Bob Swart).

The first official thanks are for the Borland programmers and managers who made Delphi possible and continue to improve it: Chuck Jazdzewski, Danny Thorpe, Eddie Churchill, Allen Bauer, Steve Todd, Mark Edington, Jim Tierney, Ravi Kumar, Jörg Weingarten, Anders Ohlsson, and all the others I have not had a chance to meet. I'd also like to give particular thanks to my friends John Kaster and David Intersimone (at Borland's Developer Relations), and others who have worked at Borland, including Charlie Calvert and Zack Urlocker.

The next thanks are for the Sybex editorial and production crew, many of whom I don't even know. Special thanks go to Brianne Agatep, Denise Santoro Lincoln, Tiffany Taylor, Rozi Harris, and Kelly Winquist; I'd also like to thank Joel Fugazzotto and Monica Baum.

This edition of *Mastering Delphi* has had a very detailed and scrupulous review from Delphi guru Brian Long ([www.blong.com](http://www.blong.com)). His highlights and comments have improved the book in all areas: technical content, accuracy, examples, and even readability and grammar! Thanks a lot. In writing this book I had special contributions (to different extents) to the chapters on add-on tools and in the area of .NET programming from (in alphabetical order) John Bushakra, Jim Gunkel, Chad Hower, and Robert Leahey. A short bio and contact information for each of them is in the chapters they helped me write.

Previous editions also had special contributions: Tim Gooch worked on *Mastering Delphi 4* and Giuseppe Madaffari contributed database material for the Delphi 5 edition. For *Mastering Delphi 6*, Guy Smith-Ferrier rewrote the chapter on ADO and Nando Dessena helped me with the InterBase material. Many improvements to the text and sample programs were suggested by technical reviewers of past editions (Delphi R&D team member Danny Thorpe, Juancarlo Añez, Ralph Friedman, Tim Gooch, and Alain Tadros) and in other reviews over the years by Bob Swart, Giuseppe Madaffari, and Steve Tendon. Uberto Barbini helped me write *Mastering Kylix 2* and some of his ideas ended up also affecting this book.

Special thanks go to my friends Bruce Eckel, Andrea Provaglio, Norm McIntosh, Johanna and Phil of the BUG-UK, Ray Konopka, Mark Miller, Cary Jensen, Chris Frizelle of *The Delphi Magazine*, Mike Orriss, Dan Miser, my co-worker Paolo Rossi, and the entire Italian D&D Team ([www.dedonline.com](http://www.dedonline.com)). Also, a very big "Thank You" to all the attendees of my Delphi programming courses, seminars, and conferences in Italy, the United States, France, the United Kingdom, Singapore, the Netherlands, Germany, Sweden....

My biggest thanks go to my wife Lella who had to endure yet another book-writing session and too many late nights (after spending the evenings with our daughter, Benedetta— I'll thank her with a hug, as Daddy's book looks quite boring to her). Many of our friends (and their kids) provided healthy breaks in the work: Sandro and Monica with Luca, Stefano and Elena, Marco and Laura with Matteo and Filippo, Bianca and Paolo, Luca and Elena with Tommaso, Chiara and Daniele with Leonardo and Matteo, Laura, Vito and Marika with Sofia. Our parents, brothers, sisters, and their families were very supportive, too. It was nice to spend some of our free time with them and our seven nephews— Matteo, Andrea, Giacomo, Stefano, Andrea, Pietro, and Elena.

Finally, I would like to thank all of the people, many of them unknown, who enjoy life and help to build a better world. If I never stop believing in the future and in peace, it is also because of them.

### **Visit Marco's Delphi Developer Website**

This book's author, Marco Cantù, has created a site specifically for Delphi developers, at [www.marcocantu.com](http://www.marcocantu.com). It's a great resource for all of your Delphi programming needs.

The site includes:

- The source code of the book

- Extra examples and tips
- Delphi components, wizards, and tools built by the author
- The online books *Essential Pascal*, *Essential Delphi*, and others
- Papers the author has written about Delphi, C++, and Java
- Extensive links to Delphi-related websites and documents
- Other material related to the author's books, the conferences he speaks at, and his training seminars

The site also hosts a newsgroup, which has a specific section devoted to the author's books, so that readers can discuss the book content with him and among themselves. Other sections of the newsgroup discuss Delphi programming and general topics. The newsgroup can also be accessed from a Web interface.



## Introduction

The first time Zack Urlocker showed me a yet-to-be-released product code-named Delphi, I realized that it would change my work—and the work of many other software developers. I used to struggle with C++ libraries for Windows, and Delphi was and still is the best combination of object-oriented programming and visual programming not only for this operating system but also for Linux and soon for .NET.

Delphi 7 simply builds on this tradition and on the solid foundations of the VCL to deliver another astonishing and all-encompassing software development tool. Looking for database, client/server, multitier, intranet, or Internet solutions? Looking for control and power? Looking for fast productivity? With Delphi and the plethora of techniques and tips presented in this book, you'll be able to accomplish all this.

## Seven Versions and Counting

Some of the original Delphi features that attracted me were its form-based and object-oriented approach, its extremely fast compiler, its great database support, its close integration with Windows programming, and its component technology. But the most important element was the Object Pascal language, which is the foundation of everything else.

Delphi 2 was even better! Among its most important additions were these: the Multi-Record Object and the improved database grid, OLE Automation support and the variant data type, full Windows 95 support and integration, the long string data type, and Visual Form Inheritance. Delphi 3 added to this the code insight technology, DLL debugging support, component templates, the TeeChart, the Decision Cube, the WebBroker technology, component packages, ActiveForms, and an astonishing integration with COM, thanks to interfaces.

Delphi 4 gave us the AppBrowser editor, new Windows 98 features, improved OLE and COM support, extended database components, and many additions to the core VCL classes, including support for docking, constraining, and anchoring controls. Delphi 5 added to the picture many more improvements of the IDE (too many to list here), extended database support (with specific ADO and InterBase datasets), an improved version of MIDAS with Internet support, the TeamSource version-control tool, translation capabilities, the concept of frames, and new components.

Delphi 6 added to all these features support for cross-platform development with the Component Library for Cross-Platform (CLX), an extended run-time library, the dbExpress database engine, Web services and exceptional XML support, a powerful Web development framework, more IDE enhancements, and a plethora of components and classes, still covered in detail in the following pages.

Delphi 7 did make some of these newer technologies more robust with improvement and fixes (SOAP support and DataSnap come to mind) and offers support for newer technologies (like Windows XP themes or UDDI), but it most importantly makes readily available an interesting set of third-party tools: the RAVE reporting engine, the IntraWeb web application development technology, and the ModelMaker design environment. Finally, it opens up a brand new world by providing (even if in a preview version) the first Borland compiler for the Pascal/Delphi language not targeting the Intel CPU, but rather the .NET CIL platform.

Delphi is a great tool, but it is also a complex programming environment that involves many elements. This book will help you master Delphi programming, including the Delphi language, components (both using the existing ones and developing your own), database and client/server support, the key elements of Windows and COM programming, and Internet and Web development.

You do not need in-depth knowledge of any of these topics to read this book, but you do need to know the basics of programming. Having some familiarity with Delphi will help you considerably, particularly after the introductory chapters. The book starts covering its topics in depth immediately; much of the introductory material from previous editions has been removed. Some of this material and an introduction to Pascal is available on my website, as discussed in [Appendix C](#).

## The Structure of the Book

The book is divided into four parts:

- [Part I](#), "Foundations," introduces new features of the Delphi 7 Integrated Development Environment (IDE) in [Chapter 1](#), then moves to the Delphi language and to the run-time library (RTL) and Visual Component Library (VCL). Four chapters in this part provide both foundations and coverage of the most commonly used controls, the development of advanced user interfaces, and the use of forms.
- [Part II](#), "Delphi Object-Oriented Architectures," covers the architecture of Delphi applications, the development of custom components, the use of libraries and packages, modeling with ModelMaker, and COM+.
- [Part III](#), "Delphi Database-Oriented Architectures," covers plain database access, in-depth coverage of the data-aware controls, client/server programming, dbExpress, InterBase, ADO, DataSnap, the development of custom data-aware controls and data sets, and reporting.
- [Part IV](#), "Delphi, the Internet, and a .NET Preview," first discusses TCP/IP sockets, Internet protocols and Indy, then moves on to specific areas like web server-side extensions (with WebBroker, WebSnap, and IntraWeb), and finishes with XML and the development of web services.

As this brief summary suggests, the book covers topics of interest to Delphi users at nearly all levels of programming expertise, from "advanced beginners" to component developers.

In this book, I've tried to skip reference material almost completely and focus instead on techniques for using Delphi effectively. Because Delphi provides extensive online documentation, to include lists of methods and properties of components in the book would not only be superfluous, it would also make it obsolete as soon as the software changes slightly. I suggest that you read this book with the Delphi Help files at hand, to have reference material readily available.

However, I've done my best to allow you to read the book away from a computer if you prefer. Screen images and the key portions of the listings should help in this direction. The book uses just a few conventions to make it more readable. All the source code elements, such as keywords, properties, classes, and functions, appear in `this font`, and code excerpts are formatted as they appear in the Delphi editor, with boldfaced keywords and italic comments.

## Free Source Code on the Web

This book focuses on examples. After the presentation of each concept or Delphi component, you'll find a working program example (sometimes more than one) that demonstrates how the feature can be used. All told, there are over 300 examples presented in the book. These programs are available in a single ZIP file of less than 2MB on both Sybex's website ([www.sybex.com](http://www.sybex.com)) and my website ([www.marcocantu.com](http://www.marcocantu.com)). Most of the examples are quite simple and focus on a single feature. More complex examples are often built step-by-step, with intermediate steps including partial solutions and incremental improvements.

**Note** Some of the database examples also require you to have the Delphi sample database files installed; they are part of the default Delphi installation. Others require the InterBase EMPLOYEE sample database (and also the InterBase server, of course).

On my website there is also an HTML version of the source code, with full syntax highlighting, along with a complete cross-reference of keywords and identifiers (class, function, method, and property names, among others). The cross-reference is an HTML file, so you'll be able to use your browser to easily find all the programs that use a Delphi keyword or identifier you're looking for (not a full search engine, but close enough).

The directory structure of the sample code is quite simple. Basically, each chapter of the book has its own folder, with a subfolder for each example (e.g., 03\FilesList). In the text, the examples are simply referenced by name (e.g., FilesList).

**Note** Be sure to read the source code archive's readme file, which contains important information about using the software legally and effectively.

## How to Reach the Author

If you find any problems in the text or examples in this book, both the publisher and I would be happy to hear from you. Besides reporting errors and problems, please give us your unbiased opinion of the book and tell us which examples you found most useful and which you liked least. There are several ways you can provide this feedback:

- On the Sybex website ([www.sybex.com](http://www.sybex.com)), you'll find updates to the text or code as necessary. To comment on this book, click the Contact Sybex link and then choose Book Content Issues. This link displays a form where you can enter your comments.
- My own website ([www.marcocantu.com](http://www.marcocantu.com)) hosts further information about the book and about Delphi, where you might find answers to your questions. The site has news and tips, technical articles, free online books (outlined in [Appendix C](#)), white papers, Delphi links, and my collection of Delphi components and tools (covered in [Appendix A](#)).
- I have also set up a newsgroup section specifically devoted to my books and to general Delphi Q&A. Refer to my website for a list of the newsgroup areas and for the instructions to subscribe to them. (In fact, these newsgroups are totally free but require a login password.) The newsgroups can also be accessed via a Web interface through a link you can find on my site.
- Finally, you can reach me via e-mail at [marco@marcocantu.com](mailto:marco@marcocantu.com). For technical questions, please try using the newsgroups first, as you might get answers earlier and from multiple people. My mailbox is usually overflowing and, regretfully, I cannot reply promptly to every request. (Please write to me in English or Italian.)

# Part I: Foundations

## Chapter List

[Chapter 1](#): Delphi 7 and Its IDE

[Chapter 2](#): The Delphi Programming Language

[Chapter 3](#): The Run-Time Library

[Chapter 4](#): Core Library Classes

[Chapter 5](#): Visual Controls

[Chapter 6](#): Building the User Interface

[Chapter 7](#): Working with Forms

## Chapter 1: Delphi 7 and Its IDE

### Overview

In a visual programming tool such as Delphi, the role of the integrated development environment (IDE) is at times even more important than the programming language. Delphi 7 provides some interesting new features on top of the rich IDE of Delphi 6. This chapter examines these new features, as well as features added in other recent versions of Delphi. We'll also discuss a few traditional Delphi features that are not well known or obvious to newcomers. This chapter isn't a complete tutorial of the IDE, which would require far too much space; it's primarily a collection of tips and suggestions aimed at the average Delphi user.

If you *are* a beginning programmer, don't be afraid. The Delphi IDE is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi Companion Tools CD) with a tutorial that introduces the development of Delphi applications. You can find a simpler introduction to Delphi and its IDE in my *Essential Delphi* online book (discussed in [Appendix C](#), "Free Companion Books on Delphi"). Throughout this book, I'll assume you already know how to carry out the basic hands-on operations of the IDE; all the chapters after this one focus on programming issues and techniques.



## Editions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

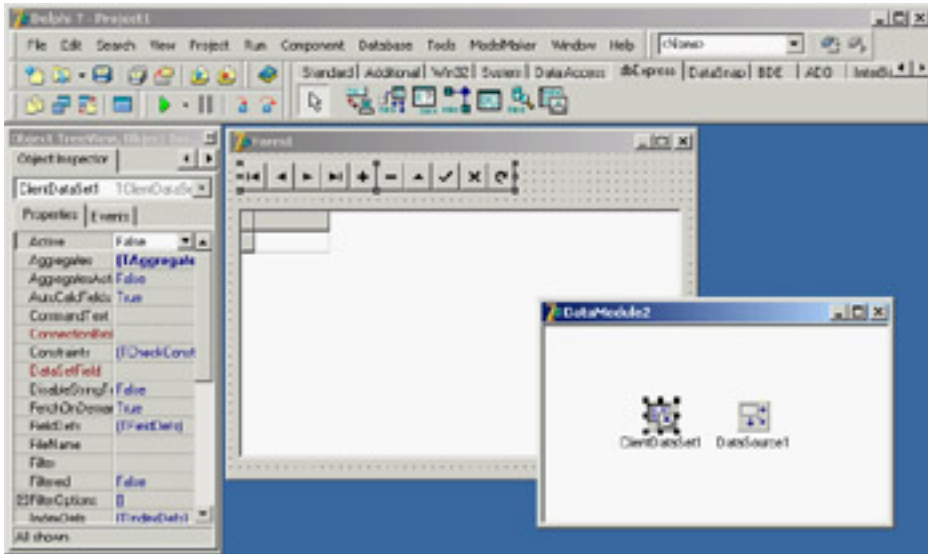
- The "Personal" edition is aimed at Delphi newcomers and casual programmers and has support for neither database programming nor any of the other advanced features of Delphi.
- The "Professional Studio" edition is aimed at professional developers. It includes all the basic features, plus database programming support (including ADO support), basic web server support (WebBroker), and some of the external tools, including ModelMaker and IntraWeb. This book generally assumes you are working with at least the Professional edition.
- The "Enterprise Studio" edition is aimed at developers building enterprise applications. It includes all the XML and advanced web services technologies, CORBA support, internationalization, three-tier architecture, and many other tools. Some chapters of this book cover features included only in Delphi Enterprise; these sections are specifically identified.
- The "Architect Studio" edition adds to the Enterprise edition support for Bold, an environment for building applications that are driven at run time by a UML model and capable of mapping their objects both to a database and to the user interface, thanks to a plethora of advanced components. Bold support is not covered in this book.

Besides the different editions available, there are ways to customize the Delphi environment. In the screen illustrations throughout the book, I've tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which may be reflected in some of the screen shots.

The Professional and higher versions of Delphi 7 include a working copy of Kylix 3, in the Delphi language edition. Other than references to the CLX library and cross-platform features of Delphi, this book doesn't cover Kylix and Linux development. You can refer to *Mastering Kylix 2* (Sybex, 2002) for more information on the topic. (There aren't many differences between Kylix 2 and Kylix 3 in the Delphi language version. The most important new feature of Kylix 3 is its support of the C++ language.)

## An Overview of the IDE

When you work with a visual development environment, your time is spent in two different portions of the application: visual designers and the code editor. Designers let you work with components at the visual level (such as when you place a button on a form) or at a non-visual level (such as when you place a DataSet component on a data module). You can see a form and a data module in action in [Figure 1.1](#). In both cases, designers allow you to choose the components you need and set the initial value of the components' properties.



**Figure 1.1:** A form and a data module in the Delphi 7 IDE

The code editor is where you write code. The most obvious way to write code in a visual environment involves responding to events, beginning with events attached to operations performed by program users, such as clicking on a button or selecting an item of a list box. You can use the same approach to handle internal events, such as events involving database changes or notifications from the operating system.

As programmers become more knowledgeable about Delphi they often begin by writing mainly event-handling code and then move to writing their own classes and components, and often end up spending most of their time in the editor. Because this book covers more than visual programming, and tries to help you master the entire power of Delphi, as the text proceeds you'll see more code and fewer forms.

## An IDE for Two Libraries

An important change appeared for the first time in Delphi 6. The IDE now lets you work on two different visual libraries: VCL (Visual Component Library) and CLX (Component Library for Cross-Platform). When you create a new project, you simply choose which of the two libraries you want to use, starting with the File → New → Application command for a classic VCL-based Windows program and with the File → New → CLX Application command for a new CLX-based portable application.

**Note** CLX is Delphi's Cross Platform library, which allows you to recompile your code with Kylix to run under Linux. You can read more about VCL versus CLX in [Chapter 5](#), "Visual Controls." Using CLX is even more interesting in Delphi 7, because the Delphi language version of Kylix ships with the Windows product.

When you create a new project or open an existing one, the Component Palette is arranged to show only the controls related to the current library (although most of the controls are shared). When you work with a non-visual designer (such as a data module), the tabs of the Component Palette that host only visual components are hidden from view.

## Desktop Settings

Programmers can customize the Delphi IDE in various ways—typically, opening many windows, arranging them, and docking them to each other. However, you'll often need to open one set of windows at design time and a different set at debug time. Similarly, you might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

For this reason, Delphi lets you save a given arrangement of IDE windows (called a *desktop*, or a Global Desktop, to differentiate from a Project Desktop) with a name and restore it easily. You can also make one of these groupings your default debugging setting, so that it will be restored automatically when you start the debugger. All these features are available in the Desktops toolbar. You can also work with desktop settings using the View → Desktops menu.

Desktop setting information is saved in DST files (stored in Delphi's `bin` directory), which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78
```

```
[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1
```

```
[AlignmentPalette]
Create=1
Visible=0
...
```

Desktop settings override project settings, which are saved in a DSK file with a similar structure. Desktop settings help eliminate problems that can occur when you move a project between machines (or between developers) and have to rearrange the windows to your liking. Delphi separates per-user global desktop settings and per-project desktop settings, to better support team development.

**Tip** If you open Delphi and cannot see the form or other windows, I suggest you try checking (or deleting) the desktop settings (from Delphi's `bin` directory). If you open a project received by a different user and cannot see some of the windows or dislike the desktop layout, reload your global desktop settings or delete the project DSK file.

## Environment Options

Quite a few recent updates relate to the commonly used Environment Options dialog box. The pages of this dialog box were rearranged in Delphi 6, moving the Form Designer options from the Preferences page to the new Designer page. In Delphi 6 there were also a few new options and pages:

- The Preferences page of the Environment Options dialog box has a check box that prevents Delphi windows from

automatically docking with each other.

- The Environment Variables page allows you to see system environment variables (such as the standard pathnames and OS settings) and set user-defined variables. The nice point is that you can use both system- and user-defined environment variables in each of the dialog boxes of the IDE—for example, you can avoid hard-coding commonly used pathnames, replacing them with a variable. In other words, the environment variables work similarly to the `$DELPHI` variable, referring to Delphi's base directory, but can be defined by the user.
- In the Internet page you can choose the default file extensions used for HTML and XML files (mainly by the WebSnap framework) and also associate an external editor with each extension.

## About Menus

The main Delphi menu bar (which in Delphi 7 has a more modern look) is an important way to interact with the IDE, although you'll probably accomplish most tasks using shortcut keys and shortcut menus. The menu bar doesn't change much in reaction to your current operations: You need to click the right mouse button for a full list of the operations you can perform on the current window or component.

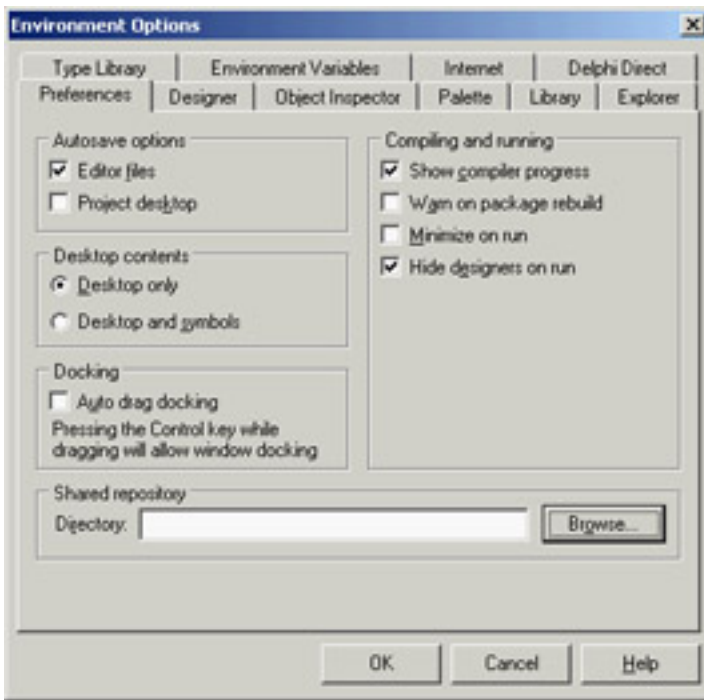
The menu bar can change considerably depending on third-party tools and wizards you've installed. In Delphi 7, ModelMaker has its own menu. You'll see other menus by installing popular add-ons like GExperts or even my own wizards (see [Appendix B](#), "Extra Delphi Tools from other Sources" and [A](#), "Extra Delphi Tools by the Author," respectively, for more details).

A relevant menu added to Delphi in recent editions is the Window menu in the IDE. This menu lists the open windows; previously, you could obtain this list using the Alt+0 key combination or the View → Window List menu item. The Window menu is really handy, because windows often end up behind others and are hard to find. You can control the alphabetic sort order of this menu using a setting in the Windows Registry: Look for the `Main Window` subkey of Delphi (under `HKEY_CURRENT_USER\Software\Borland\Delphi\7.0`). This Registry key uses a string (in place of Boolean values), where '-1' and 'True' indicate true and '0' and 'False' indicate false.

**Tip** In Delphi 7, the Window menu ends with a new command: Next Window. This command is particularly useful in the form of a shortcut, Alt+End. Jumping around the various windows of the IDE has never been so simple (at least, without add-on tools).

## The Environment Options Dialog Box

As I've mentioned, some of the IDE settings require you to edit the Registry directly. I'll discuss a few more of these settings in this chapter. Of course, the most common settings can be easily tuned using the Environment Options dialog box, which is available in the Tools menu along with the Editor Options and the Debugger Options. Most of the settings are quite intuitive and well described in the Delphi Help file. [Figure 1.2](#) shows my standard settings for the Preferences page of this dialog box.



**Figure 1.2:** The Preferences page of the Environment Options dialog box

## The To-Do List

Another feature added in Delphi 5 but still quite underused is the to-do list. This is a list of tasks you still have to do to complete a project—it's a collection of notes for the programmer (or programmers; this tool can be very handy in a team). Although the idea is not new, the key concept of the to-do list in Delphi is that it works as a two-way tool.

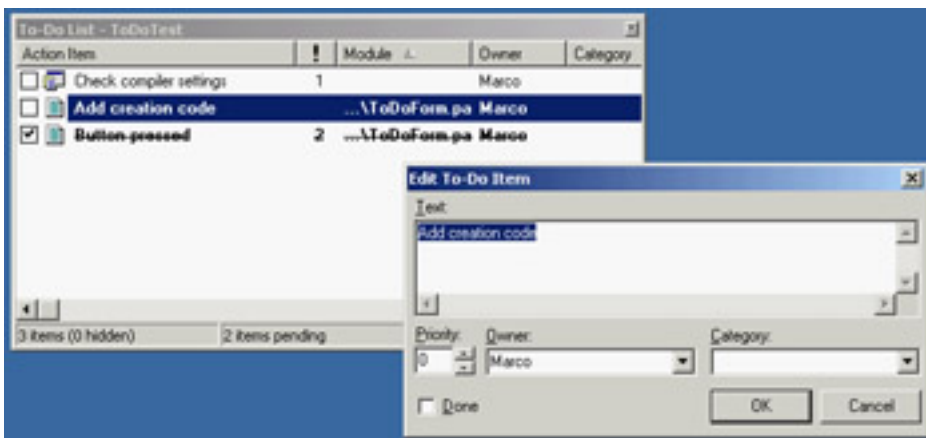
You can add or modify to-do items by adding special `TODO` comments to the source code of any file of a project; you'll then see the corresponding entries in the list. In addition, you can visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look in the source code:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    // TODO -oMarco: Add creation code
end;

```

The same item can be visually edited in the window shown in [Figure 1.3](#), along with the To-Do List window.



**Figure 1.3:** The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code.

The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the `Ctrl+A` key combination in the To-Do List window or right-click in the window and

select Add from the shortcut menu. These items are saved in a special file with the same root name as the project file and a .TODO extension.

You can use multiple options with a TODO comment. You can use `-o` (as in the previous code excerpt) to indicate the owner (the programmer who entered the comment), the `-c` option to indicate a category, or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

Delphi treats everything after the colon—up to the end of the line or the closing brace, depending on the type of comment—as the text of the to-do item.

Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from TODO to DONE. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, which can automatically collect to-do information from the source code files as you type them, sort and filter them, and export them to the Clipboard as plain text or an HTML table. All these options are available on the context menu.

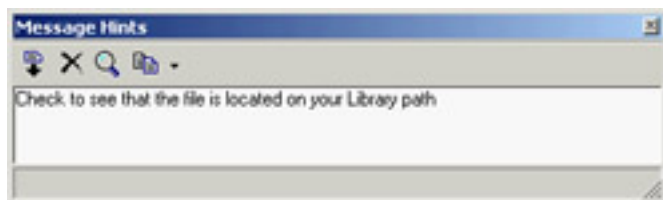
## Extended Compiler Messages and Search Results in Delphi 7

A small Messages window appears by default below the editor; it displays both compiler messages and search results. This window has been considerably modified in Delphi 7. First, search results are displayed in a different tab so they do not interfere with compiler messages as they did in the past. Second, every time you do a different search you can request that Delphi show the results in a different page, so the results of previous search operations remain available:



You can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the tabs of this window. (The same commands work for other tabbed views.)

If compiler errors occur, you can activate another new window with the command View → Additional Message Info. As you compile a program, this Message Hints window will provide extra information for some common error messages, offering suggestions about how to fix them:



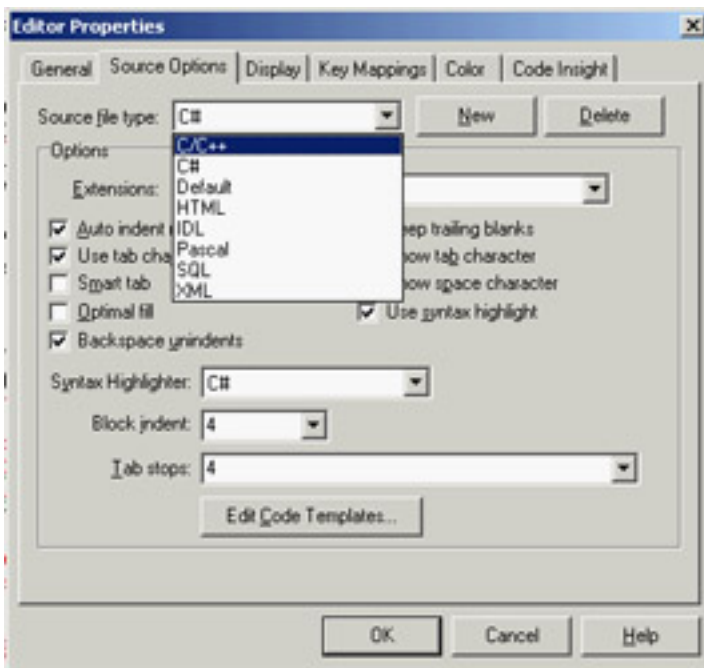
This type of help is intended more for novice programmers, but it might be handy to keep this window around. It's important to realize that this information is thoroughly customizable: A project development leader can put appropriate descriptions of common errors in a form that means something specific to new developers. To do so, follow the comments in the file hosting the settings for this feature, the `msginfo70.ini` file of Delphi's bin folder.



## The Delphi Editor

On the surface, Delphi's editor doesn't appear to have changed much for version 7 of the IDE. However, behind the scenes, it is a totally new tool. Besides using it to work on files in the Object Pascal language (or the Delphi language, as Borland prefers to call it now), you can now use it to work on other files used in Delphi development (such as SQL, XML, HTML, and XSL files), as well as files in other languages (including C++ and C#). XML and HTML editing was already available in Delphi 6, but the changes in this version are significant. For example, while editing an HTML file, you have support for both syntax highlighting and code completion.

The editor settings used on each file (including the behavior of keys like Tab) depend on the extension of the file being opened. You can configure these settings in the new Source Options page of the Editor Properties dialog box, displayed in [Figure 1.4](#). This feature has been extended and made more open, so you can even configure the editor by providing a DTD for XML-based file formats or by writing a custom wizard that provides syntax highlighting for other programming languages. Another feature of the editor, code templates, is now language specific (your predefined Delphi templates will make little sense in HTML or C#).



**Figure 1.4:** The multiple languages supported by the Delphi IDE can be associated with various file extensions in the Source Options page of the Editor Properties dialog box.

**Note** C# is the new language Microsoft introduced with its .NET architecture. Borland is expected to support C# in its own .NET environment, currently code-named Galileo.

Considering only the Delphi language, the editor included in the IDE hasn't changed much in recent versions. However, it has a few features that many Delphi programmers don't know about and use, so I think it's worth a brief examination.

The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Ctrl+Shift+Tab to move in the opposite direction). You can drag-and-drop the tabs with the unit names in the upper portion of the editor to change their order, so that you can use a single Ctrl+Tab to move between the units you are working on any given time. The editor's shortcut menu has also a Pages command, which lists all the available pages in a submenu (a handy feature when many units are loaded).

You can also open multiple editor windows, each hosting multiple tabs. Doing so is the only way to see the source code of two units alongside each other. (Actually, when I need to compare two Delphi units, I invariably use Beyond Compare—[www.scootersoftware.com](http://www.scootersoftware.com)—a superb, low-cost file comparison utility written in Delphi.)

Several options affect the editor, as you can see in the Editor Properties dialog box in [Figure 1.4](#). However, you have to go to the Preferences page of the Environment Options dialog box (see [Figure 1.2](#)) to set the editor's AutoSave feature. This option forces the editor to save all of your source code files each time you run the program, preventing data loss in the (rare) case the program crashes badly in the debugger.

Delphi's editor provides many commands, including some that date back to its WordStar emulation ancestry (of the early Turbo Pascal compilers). I won't discuss the various settings of the editor, because they are quite intuitive and are described in the online help. Notice, though, that the page of the help describing the keyboard shortcuts is accessible as a whole only if you look up the *shortcuts* index item.

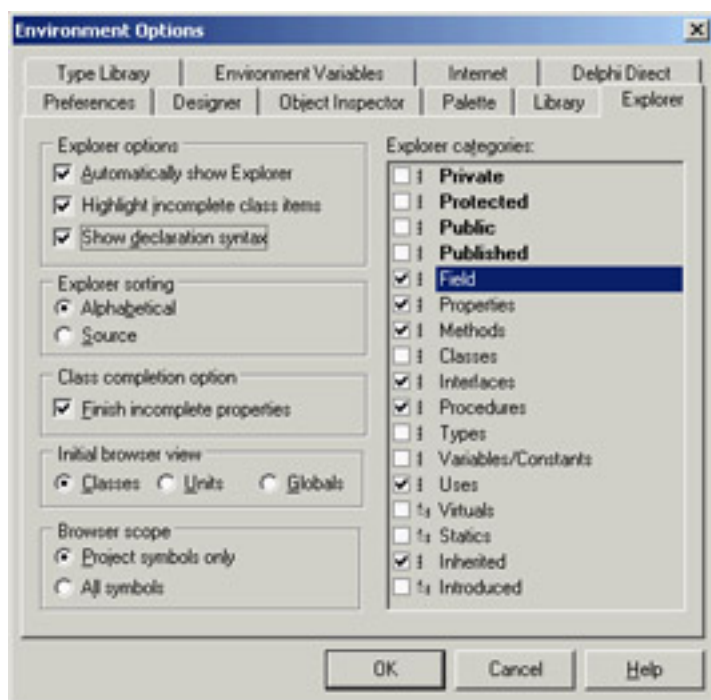
**Tip** A tip to remember is that using the Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. In addition, you can copy text instead of moving it, by pressing the Ctrl key while dragging.

## The Code Explorer

The Code Explorer window, which is generally docked on the side of the editor, lists all the types, variables, and routines defined in a unit, plus other units appearing in `uses` statements. For complex types, such as classes, the Code Explorer can list detailed information, including a list of fields, properties, and methods. All the information is updated as soon as you begin typing in the editor.

You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration. You can also modify variables, properties, and method names directly in the Code Explorer. However, as you'll see, if you want a visual tool to use when you work on your classes, ModelMaker provides many more features.

Although all this functionality is obvious after you've used Delphi for a few minutes, some features of the Code Explorer are not so intuitive. You have full control of the information layout. And, you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer (collapsing the tree can help you make your selections more quickly). You can configure the Code Explorer by using the corresponding page of the Environment Options, as shown in [Figure 1.5](#).



**Figure 1.5:** You can configure the Code Explorer in the Environment Options dialog box.

Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view—it simply adds the node in the tree. For example, if



you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer; on the contrary, the used units are listed as main nodes instead of being kept in the `Uses` folder. I generally disable the Types, Classes, and Variables/Constants selections.

Because each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, because this arrangement requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer provides a handy way of navigating the source code of a large unit—when you double-click a method in the Code Explorer, the focus moves to the definition in the class declaration. You can use Module Navigation (the Ctrl+Shift combination with the Up or Down arrow key) to jump from the definition of a method or procedure to its complete definition (or back again).

**Note** Some of the Explorer Categories shown in [Figure 1.5](#) are used by the Project Browser, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.

## Browsing in the Editor

Another feature of the editor is *Tooltip symbol insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the library.

**Warning** Although it may seem like a good idea at first, you cannot use Tooltip symbol insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, the Tooltip won't appear.

The real bonus of this feature, however, is that you can turn it into a navigational aid called *code browsing*. When you hold down the Ctrl key and move the mouse over the identifier, Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of links in web browsers, and the pointer changes to a hand whenever it's positioned on the link.

For example, you can Ctrl+click the `TLabel` identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again, as in a web browser—using the Browse Back and Browse Forward buttons in the top-right corner of the editor windows or the keystrokes Alt+Left arrow or Alt+Right arrow. You can also click the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

How can you jump directly to the VCL source code if it is not part of your project? The editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment.

## Class Completion

Delphi's editor can also help by generating some source code for you, completing what you've already written. This feature is called *class completion*, and you activate it by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, because Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi simplify life in a similar way for programmers who write a little extra code behind event handlers. This code-generation feature applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration

```
public
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }  
procedure TForm1.Hello(MessageText: string);  
begin  
end;
```

This feature is really handy compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin...end` code for every method copied. Class completion also works the other way around: You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

The most important and useful example of class completion is the automatic generation of code to support properties declared in classes. For example, if you type in a class

```
property Value: Integer;
```

and press Ctrl+Shift+C, Delphi will turn the line into

```
property Value: Integer read fValue write SetValue;
```

Delphi will also add the `SetValue` method to the class declaration and provide a default implementation for it. You'll find more on properties in the [next chapter](#).

## Code Insight

In addition to the Code Explorer, class completion, and the navigational features, the Delphi editor supports the *code insight* technology. Collectively, the code insight techniques are based on a constant background parsing of both the source code you write and the source code of the system units your source code refers to.

Code insight comprises five capabilities: code completion, code templates, code parameters, Tooltip expression evaluation, and Tooltip symbol insight. This last feature was already covered in the section "[Browsing in the Editor](#)"; the other four are discussed in the following subsections. You can enable, disable, and configure each of these features in the Code Insight page of the Editor Properties dialog box.

## Code Completion

Code completion allows you to choose the property or method of an object simply by looking it up on a list or by typing its initial letters. To activate this list, you just type the name of an object, such as `Button1`, then add the dot, and wait. To force the display of the list, press Ctrl+spacebar; to remove it when you don't want it, press Esc. Code completion also lets you look for a proper value in an assignment statement.

As you begin typing, the list filters its content according to the initial portion of the element you've inserted. The code completion list uses colors and shows more details to help you distinguish different items. In Delphi, you can customize these colors in the Code Insight page of the Editor Options dialog box. Another feature is that in the case of functions with parameters, parentheses are included in the generated code, and the parameters list hint is displayed immediately.

As you type `:=` after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click it to change the order of the items, sorting either by scope or by name; you can also resize the window.

Since Delphi 6, code completion also works in the interface section of a unit. If you press Ctrl+spacebar while the cursor is inside the class definition, you'll get a list of virtual methods you can override (including abstract methods), the methods of implemented interfaces, the base class properties, and eventually system messages you can handle. Simply selecting one of them will add the proper method to the class declaration. In this particular case, the code completion list allows multiple selection.

**Tip** When the code you've written is incorrect, code insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific code insight errors in the Message pane (which must already be open—it doesn't open automatically to display compilation errors). To activate this feature, you need to set an undocumented Registry entry, setting the string key `\Delphi\7.0\Compiling\ShowCodeInsightErrors` to the value '1'.

Code completion includes some advanced features that aren't easy to spot. One that I find particularly useful relates to the discovery of symbols in units not used by your project. As you invoke it (with Ctrl+spacebar) over a blank line, the list also includes symbols from common units (such as `Math`, `StrUtils`, and `DateUtils`) not already included in the `uses` statement of the current unit. By selecting one of these *external* symbols, Delphi adds the unit to the `uses` statement for you. This feature (which doesn't work inside expressions) is driven by a customizable list of extra units, stored in the Registry key `\Delphi\7.0\CodeCompletion\ExtraUnits`.

**Tip** Delphi 7 adds the ability to browse to the declaration of items in the code completion list by Ctrl+clicking on any identifier in the list.

## Code Templates

This feature lets you insert one of the predefined code templates, such as a complex statement with an inner `begin...end` block. Code templates must be activated manually, by pressing Ctrl+J to show a list of all of the templates. If you type a few letters (such as a keyword) before pressing Ctrl+J, Delphi will list only the templates starting with those letters.

You can add custom code templates, so that you can build your own shortcuts for commonly used blocks of code. For example, if you use the `MessageDlg` function often, you might want to add a template for it. To modify templates, go to the Source Options page of the Editor Options dialog box, select Pascal from the Source File Type list, and click the Edit Code Templates button. Doing so opens the new Delphi 7 Code Templates dialog box. At this point, click the Add button, type in a new template name (for example, **mess**), type a description, and then add the following text to the template body in the Code memo control:

```
MessageDlg ('/', mtInformation, [mbOK], 0);
```

Now, every time you need to create a message dialog box, you simply type **mess** and then press Ctrl+J, and you get the full text. The vertical line (or pipe) character indicates the position within the source code where the cursor will be in the editor after you expand the template. You should choose the position where you want to begin typing to complete the code generated by the template.

Although code templates might seem at first to correspond to language keywords, they are a more general mechanism. They are saved in the `DELPHI32.DCI` file, a text file in a rather simple format that you can edit directly. Delphi 7 also allows you to export the settings for a language to a file and import them, making it easier for developers to exchange their own customized templates.

## Code Parameters

While you are typing a function or method, code parameters display the data type of the function's or method's parameters in a hint or Tooltip window. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a pop-up hint window. To force the display of code parameters, you can press Ctrl+Shift+spacebar. As a further help, the current parameter appears in bold type.

## Tooltip Expression Evaluation

Tooltip expression evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor. In the case of an expression, you typically need to select it in the editor and then move the mouse over the highlighted text.

## More Editor Shortcut Keys

The editor has many more shortcut keys that depend on the editor style you've selected. Here are a few of the lesser-known shortcuts:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a "gutter" margin on the side of the editor. To jump back to the bookmark, press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the facts that a new bookmark can override an existing one and that bookmarks are not persistent (they are lost when you close the file).
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is set by the Block Indent option in the Editor page of the Editor Options dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R begins recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that might have little meaning once you've moved on to a different source code file. Editor macros are quite useful for performing multistep operations over and over again, such as reformatting source code or arranging data more legibly in source code.
- While holding down the Alt key, you can drag the mouse to select rectangular areas within the editor, not just consecutive lines and words.

## Loadable Views

Another important feature introduced in Delphi 6 is support for multiple views in the editor. For any single file loaded in the IDE, the editor can show multiple views, defined programmatically and added to the system, and then loaded for given files—hence the name *loadable* views.

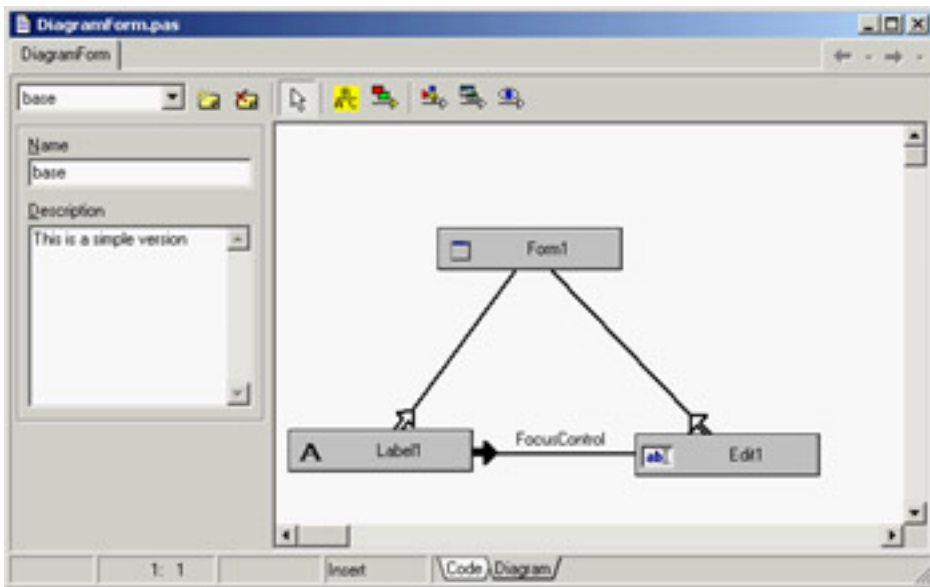
The most frequently used view is the Diagram page, which was available in Delphi 5 data modules, although it was less powerful. Another set of views is available in web applications, including an HTML Script view, an HTML Result preview, and many others discussed in [Chapters 20](#) ("Web Programming with WebBroker and WebSnap") and [22](#) ("Using XML Technologies"). You can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the bottom tabs of this editor; Ctrl+Tab changes the pages (or files) shown in the upper tabs.

## The Diagram View

The Diagram view shows dependencies among components, including parent/child relations, ownership, linked properties, and generic relations. For dataset components, it also supports master/detail relations and lookup connections. You can even add your comments in text blocks linked to specific components.

The diagram is not built automatically. You must drag components from the Tree view to the diagram, which will automatically display the existing relations among the components you drop there. You can select multiple items from the Object TreeView and drag them all at once to the Diagram page.

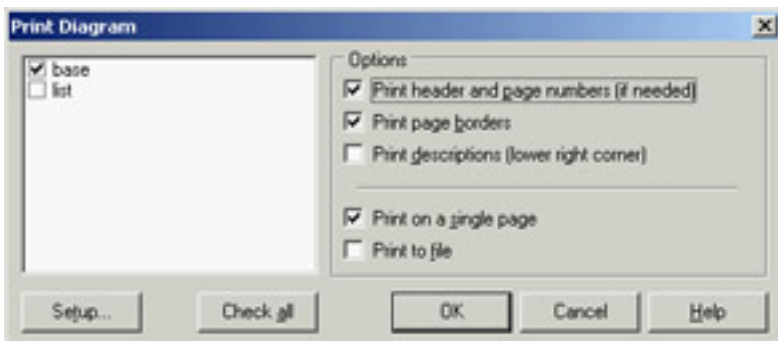
What's nice is that you can set properties by simply drawing arrows between the components. For example, after moving an edit and a label to the diagram, you can select the Property Connector icon, click the label, and drag the mouse cursor over the edit. When you release the mouse button, the Diagram view will set up a property relation based on the `FocusControl` property, which is the only property of the label referring to an edit control. This situation is depicted in [Figure 1.6](#).



**Figure 1.6:** The Diagram view shows relationships among components (and even allows you to set them up).

As you can see, setting properties is *directional*: If you drag the property relation line from the edit to the label, you end up trying to use the label as the value of a property of the edit box. Because this isn't possible, you'll see an error message indicating the problem and offering to connect the components in the opposite way. The Diagram view allows you to create multiple diagrams for each Delphi unit—that is, for each form or data module. You give a name to the diagram and possibly add a description, click the New Diagram button, prepare another diagram, and you'll be able to switch back and forth between diagrams using the combo box available in the toolbar of the Diagram view.

Although you can use the Diagram view to set up relations, its main role is to document your design. For this reason, it is important to be able to print the content of this view. Use the standard File → Print command while the Diagram view is active, and Delphi prompts you for options as shown in [Figure 1.7](#). You can customize the output in many ways.



**Figure 1.7:** The Print Options for the Diagram view

The information in the Diagram view is saved in a separate file, not as part of the DFM file. Delphi 5 used design-time information (DTI) files, which had a structure similar to INI files. Delphi 6 and 7 can still read the older .DTI format, but they use the new Delphi Diagram Portfolio format (.DDP). These files use the DFM binary format (or a similar format), so they are not editable as text. All of these files are obviously useless at run time (it makes no sense to include them in the compilation of the executable file).

**Note** If you want to experiment with the Diagram view, you can start by opening the DiagramDemo project included among the examples for this chapter. The program's form has two associated diagrams: the one in [Figure 1.6](#) and a much more complex one with a pull-down menu and its items.



## The Form Designer

Another Delphi window you'll interact with often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, you can select a component directly with the mouse; you can also use the Object Inspector or the Object TreeView, which is handy when a control is behind another one or is very small. If one control covers another completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click the selected component. Doing so will deselect the current component and select the form by default.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position (when the Snap To Grid option is active), as is holding down Alt while using the mouse to move the control. If you press Ctrl+Shift along with an arrow key, the component will move only at grid intervals.

By pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. Again, you can also do so with the mouse and the Alt key.

To align multiple components or make them the same size, you can select them and set the `Top`, `Left`, `Width`, or `Height` property for all of them at the same time. To select several components, click them with the mouse while holding down the Shift key; or, if all the components fall into a rectangular area, drag the mouse to "draw" a rectangle surrounding them. To select child controls (say, the buttons inside a panel), drag the mouse within the panel while holding down the Ctrl key—otherwise, you move the panel. When you've selected multiple components, you can also set their relative position using the Alignment dialog box (with the Align command of the form's shortcut menu) or the Alignment Palette (accessible through the View → Alignment Palette menu command).

When you've finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component. This command is particularly helpful, because Undo operations on forms are limited (you can only Undelete), but the setting is not persistent.

Among its other features, the Form Designer offers several Tooltip hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. Since version 6, Delphi offers extended hints, with details about the control's position, size, tab order, and more. This is an addition to the Show Component Captions environment setting, which I keep active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the `Left` and `Top` properties).

Finally, you can save DFM (Delphi Form Module) files in the old binary resource format, instead of the plain text format, which is the default. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the Designer page of the Environment Options dialog box. In the same page, you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box).

Having DFM files stored as text lets you operate more effectively with version-control systems. Programmers won't get a real advantage from this feature, because you could already open the binary DFM files in the Delphi editor with a specific command from the designer's shortcut menu. Version-control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file.

In any case, if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFMs are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded

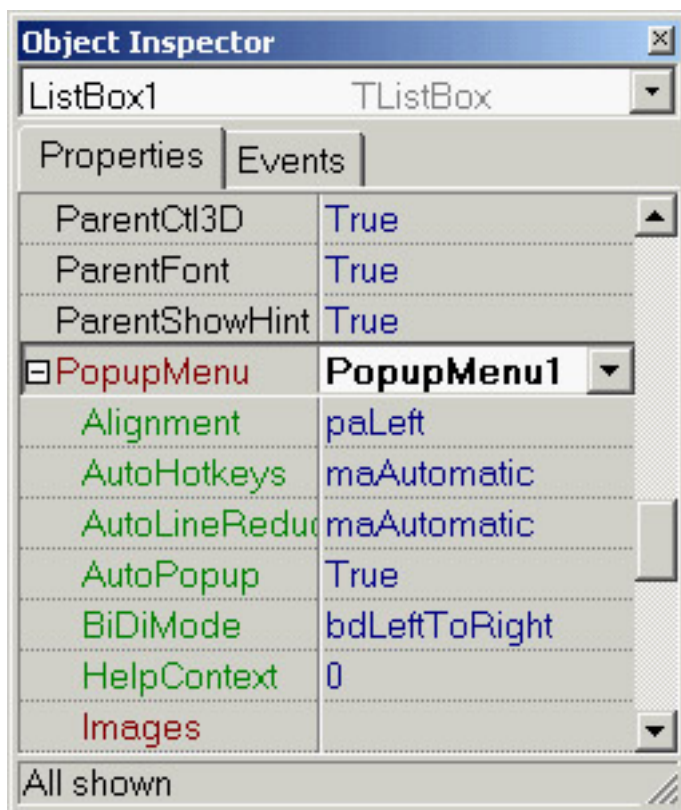
more quickly).

**Note** Text DFM files are more portable between versions of Delphi than their binary version. Although an older version of Delphi might not accept a new property of a control in a DFM created by a newer version of Delphi, the older Delphis will still be able to read the rest of the text DFM file. If the newer version of Delphi adds a new data type, though, older Delphis will be unable to read the newer Delphi's binary DFMs at all. Even if this doesn't sound likely, remember that 64-bit operating systems are just around the corner. When in doubt, save in text DFM format. Also note that all versions of Delphi support text DFMs, using the command-line tool `Convert` in the `bin` directory. Finally, keep in mind that the CLX library uses the XFM extension instead of the DFM extension, both in Delphi and Kylix.

## The Object Inspector

To see and change properties of components placed on a form (or another designer) at design time, you use the Object Inspector. Compared to the early versions of Delphi the Object Inspector has a number of new features. The latest, introduced in Delphi 7, is the use of a bold font to highlight properties that have a value different from the default.

Another important change (introduced in Delphi 6) is the ability of the Object Inspector to expand component references in place. Properties referring to other components are displayed in a different color and can be expanded by selecting the + symbol on the left, as is the case with an internal subcomponent. You can then modify the properties of that other component without having to select it. Here you can see a connected component (a pop-up menu) expanded in the Object Inspector while working on another component (a list box):



This interface-expansion feature also supports subcomponents, as demonstrated by the new LabeledEdit control. A related feature of the Object Inspector lets you select the component referenced by a property. To do this, double-click the property value with the left mouse button while pressing the Ctrl key. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the Menu property of the form and Ctrl+double-clicking the value of this property. Doing so selects the main menu indicated as the value of the property in the Object Inspector.

Here are some other recent changes of the Object Inspector:

- The list at the top of the Object Inspector shows the type of the object and allows you to choose a component. You

might remove this list to save some space, considering that you can select components in the Object TreeView (by default placed on the top of the Object Inspector window).

- The properties that reference an object are now a different color and may be expanded without changing the selection.
- You can optionally view read-only properties in the Object Inspector. Of course, they are grayed out.
- The Object Inspector has a Properties dialog box, which allows you to customize the colors of the various types of properties and the overall behavior of this window.
- Since Delphi 5, the drop-down list for a property can include graphical elements. This feature is used for properties such as `Color` and `Cursor`, and is particularly useful for the `ImageIndex` property of components connected to an `ImageList`.

**Note** Interface properties can now be configured at design time using the Object Inspector. This functionality uses the Interfaced Component Reference model introduced in Kylix/Delphi 6, where components can implement and hold references to interfaces as long as the interfaces are implemented by components. Interfaced Component References work like plain old component references, but interface properties can be bound to any component that implements the necessary interface. Unlike component properties, interface properties are not limited to a specific component type (a class or its derived classes). When you click the drop-down list in the Object Inspector editor for an interface property, all components on the current form (and linked forms) that implement the interface are shown.

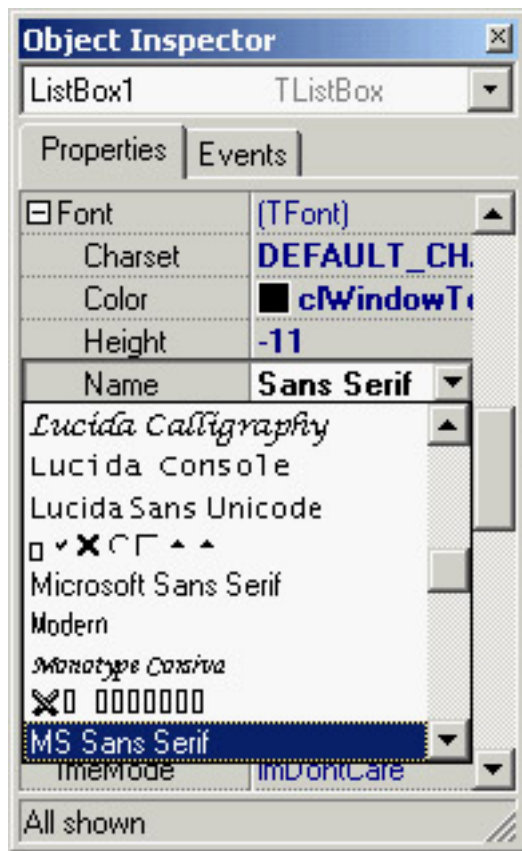
---

### Drop-Down Fonts in the Object Inspector

The Delphi Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting, corresponding to the `Name` subproperty of the `Font` property. This capability is built into Delphi, but it has been disabled because most computers have a large number of fonts installed and rendering them can significantly slow the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplay-FontNames` global variable of the `VCLEditors` unit. I've done this in the `OiFontPk` package, which you can find among the program examples for this chapter.

Once this package is installed, you can move to the `Font` property of any component and use the graphical `Name` drop-down menu, as displayed here:





There is a second, more complex customization of the Object Inspector that I like and use frequently: a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations. See [Appendix A](#) to learn how to obtain this add-on package.

---

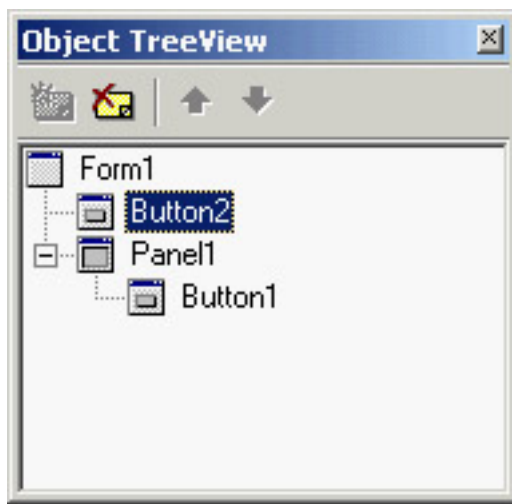
## Property Categories

Delphi includes the idea of property categories, activated by the Arrange option of the Object Inspector's shortcut menu. If you set this option, properties are arranged by group rather than listed alphabetically, with each property possibly appearing in multiple groups. Categories have the benefit of reducing the complexity of the Object Inspector. You can use the View submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories). Although property categories have been available since Delphi 5, programmers rarely use them.

## The Object TreeView

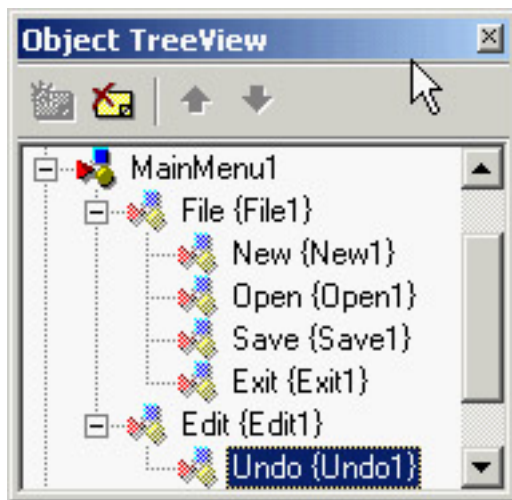
Delphi 5 introduced a TreeView for data modules, in which you could see the relations among nonvisual components, such as datasets, fields, actions, and so on. Delphi 6 extended the idea by providing an Object TreeView for every designer, including plain forms. The Object TreeView is placed by default above the Object Inspector.

The Object TreeView shows all the components and objects on the form in a tree representing their relations. The most obvious is the parent/child relation: If you place a panel on a form, a button inside the panel, and a button outside the panel, the tree will show one button under the form and the other under the panel:



Notice that the TreeView is synchronized with the Object Inspector and Form Designer. So, as you select an item and change the focus in any one of these three tools, the focus changes in the other two tools.

Besides parent/child, the Object TreeView shows other relations, such as owner/owned, component/subobject, and collection/item, plus various specific relations, including dataset/ connection and data source/dataset relations. Here, you can see an example of the structure of a menu in the tree:

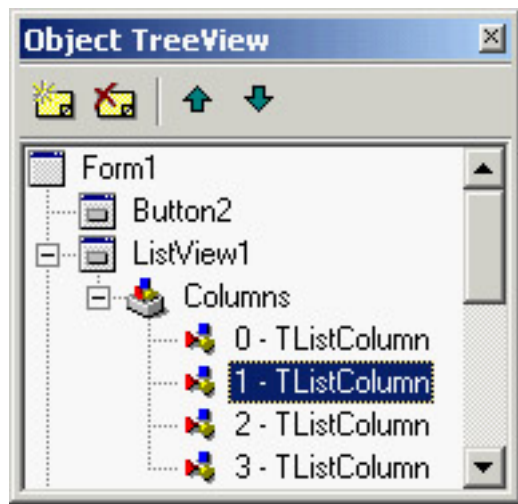


At times, the TreeView also displays "dummy" nodes, which do not correspond to an actual object but do correspond to a predefined object. As an example of this behavior, drop a Table component (from the BDE page); you'll see two grayed icons for the session and the alias. Technically, the Object TreeView uses gray icons for components that do not have design-time persistence. They are real components (at design time and at run time), but because they are default objects that are constructed at run time and have no persistent data that can be edited at design time, the Data Module Designer does not allow you to edit their properties. If you drop a Table on the form, you'll also see items that have next to them a red question mark enclosed in a yellow circle. This symbol indicates partially undefined items.

The Object TreeView supports multiple types of *dragging*:

- You can select a component from the palette (by clicking it, not dragging it), move the mouse over the tree, and click a component to drop it there. This technique allows you to drop a component in the proper container (form, panel, and others) regardless of the fact that its surface might be totally covered by other components—something that prevents you from dropping the component in the designer without first rearranging those components.
- You can drag components within the TreeView—for example, moving a component from one container to another. With the Form Designer, you can do so only with cut-and-paste techniques. Moving instead of cutting provides the advantage that any connections among components are not lost, as happens when you delete the component during the cut operation.
- You can drag components from the TreeView to the Diagram view, as you'll see later.

Right-clicking any element of the TreeView displays a shortcut menu similar to the component menu you get when the component is in a form (and in both cases, the shortcut menu may include items related to the custom component editors). You can even delete items from the tree. The TreeView also doubles as a collection editor, as you can see here for the `Columns` property of a `ListView` control. In this case, you can not only rearrange and delete items, but also add new items to the collection.



**Tip** You can print the contents of the Object TreeView for documentation purposes. Simply select the window and use the File → Print command (there is no Print command on the shortcut menu).

## Secrets of the Component Palette

The Component Palette is used to select components you want to add to the current designer. Move the mouse over a component and you'll see its name. In Delphi 7, the hint displays also the name of the unit that defines the component.

The Component Palette has many tabs—far too many, really. You may want to hide the tabs hosting components you don't plan to use and reorganize the Component Palette to suit your needs. In Delphi 7 you can also drag and drop the tabs to reorder them. Using the Palette page of the Environment Options dialog box, you can completely rearrange the components in the various pages, adding new elements or moving them from page to page.

When you have too many pages in the Component Palette, you'll need to scroll through them to reach a component. You can use a simple trick in this case: Rename the pages with shorter names, so all the pages will fit on the screen. (It's obvious—once you've thought about it.)

Delphi 7 offers another new feature. When there are too many components on a single page, Delphi displays a double down arrow; you click it to display the remaining components without having to scroll within the Palette page.

The Component Palette's shortcut menu has a Tabs submenu that lists all the palette pages in alphabetical order. You can use this submenu to change the active page, particularly when the page you need is not visible on the screen.

**Tip** You can set the order of the entries in the Tabs submenu of the Component Palette shortcut menu to be the same as the order in the palette itself, rather than alphabetical. To do so, go to the *Main Window Registry* section of Delphi (under `\Software\Borland\Delphi\7.0` for the current user) and set the *Sort Palette Tabs Menu* key value to 0 (false).

The significant undocumented feature of the Component Palette is "hot-track" activation. By setting special keys in the Registry, you can simply select a page of the palette by moving the mouse over the tab, without clicking. The same feature can be applied to the component scrollers on both sides of the palette, which appear when a page has too many components. To activate this hidden feature, add an `Extras` key under the `Borland\Delphi\7.0` key of the Registry in the `HKEY_CURRENT_USER\Software` section. Under this key, enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string '1'.

## Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate a component in the form. During this operation, Delphi duplicates all the properties, keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

You can also copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source-code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
```

```
Left = 152
Top = 104
Width = 75
Height = 25
Caption = 'My Button'
TabOrder = 0
Font.Name = 'Arial'
end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

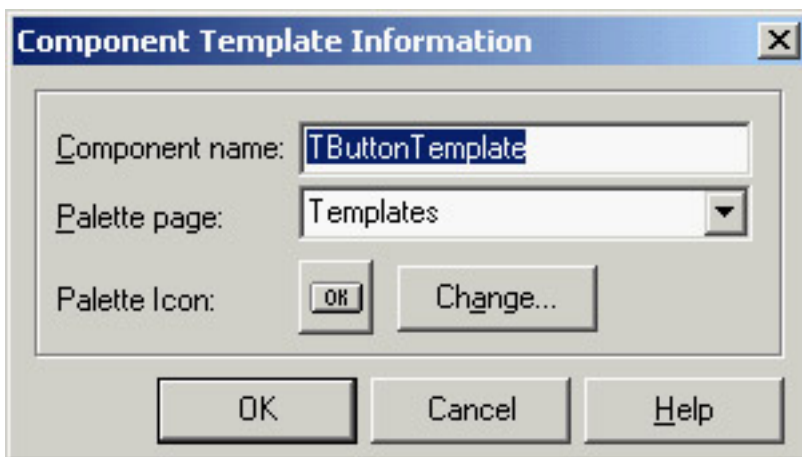
To use this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something that's completely incorrect and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This approach might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

## From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component → Create Component Template menu command. This command opens the Component Template Information dialog box, where you enter the name of the template, the page of the Component Palette where it should appear, and an icon:



By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

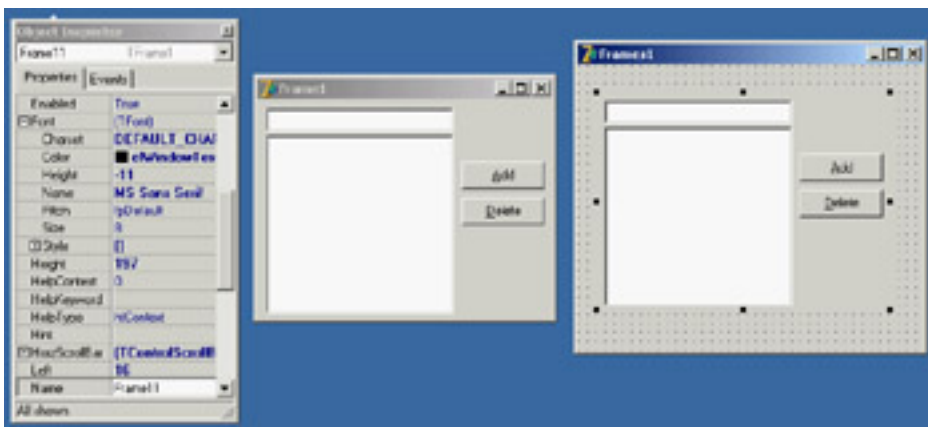
All the information about component templates is stored in a single file, `DELPHI32.DCT`, but there is no documented way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

**Tip** A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry `CCLibDir` under the key `\Software\Borland\Delphi\7.0\Component Templates`.

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the *frames* technology in Delphi does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready, you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each instance of the frame.

You can see a simple example, called `Frames1`, in [Figure 1.8](#). A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames.



**Figure 1.8:** The `Frames1` example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.

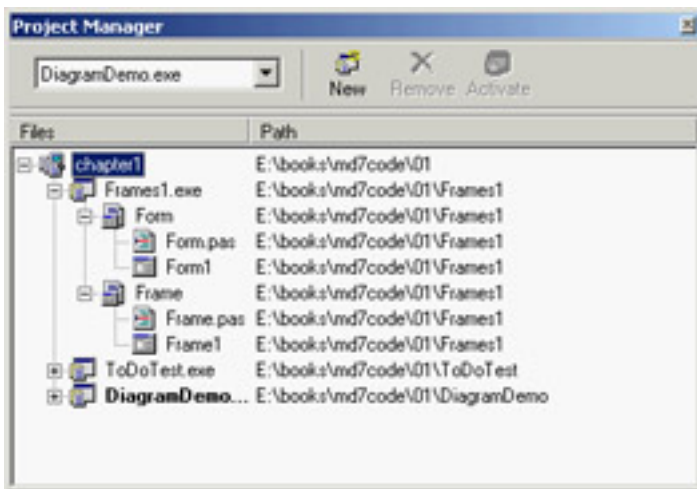
Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than component templates. [Chapter 8](#), "The Architecture of Delphi Applications," provides an in-depth look at VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful technique.



## Managing Projects

Delphi's multitarget Project Manager (View → Project Manager) works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files. All open packages will show up as projects in the Project Manager view, even if they haven't been added to the project group.

In [Figure 1.9](#), you can see the Project Manager with a simple project group, including all the examples of the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on the projects in the group. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, and to open a unit.



**Figure 1.9:** Delphi's multitarget Project Manager

**Tip** Since Delphi 6, the Project Manager also shows the open packages, even if they haven't been added to the project group. A package is a collection of components or other units compiled to a separate executable file, as you'll discover in [Chapter 10](#) ("Libraries and Packages").

Of all the projects in the group, only one is active; this is the project you operate on when you select a command such as Project → Compile. The Project menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.) When you have multiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.

**Tip** In Delphi 7, the Project Manager local menu allows you to compile projects beginning with a given one, using the Make All From Here or Build All From Here command. If you use this command on the first project of the group, you obtain the same effect as a Compile All or Build All command from the shortcut menu.

Among the Project Manager's advanced features, you can drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project (drag-and-drop is also supported to open files in the code editor). You can easily see which project is selected and change it by using the combo box at the top of the Project Manager window, or by using the drop-down arrow next to the Run button on the Delphi toolbar.

Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu item, and choose Resource File (\*.rc) as the file type. This resource file will be bound to the project automatically, even without a corresponding `$R` directive.

Delphi saves the project groups with the .BPG extension, which stands for Borland Project Group. This feature comes

from C++Builder and from past Borland C++ compilers; this history is clearly visible when you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment. Here is a simple example:

```
#-----  
VERSION = BWS.01  
#-----  
!ifndef ROOT  
ROOT = $(MAKEDIR)\..  
!endif  
#-----  
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**  
DCC = $(ROOT)\bin\dcc32.exe $**  
BRCC = $(ROOT)\bin\brcc32.exe $**  
#-----  
PROJECTS = Project1.exe  
#-----  
default: $(PROJECTS)  
#-----  
Project1.exe: Project1.dpr  
    $(DCC)
```

## Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. Instead, you can invoke the Project Options dialog from the Project Manager for each project. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{ $X+ }` and `{ $X- }` directives, where you replace `X` with the option you want to set. This second approach is more flexible, because it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All project options are saved automatically with the project, but in a separate file with a `.DOF` extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a `CFG` file, for command-line compilation. The two files have similar content but a different format: The `dcc` command-line compiler cannot use `.DOF` files, but needs the `.CFG` format.

Another alternative for saving compiler options is to press `Ctrl+O+O` (press the `O` key twice while keeping `Ctrl` pressed). This key combination inserts, at the top of the current unit, compiler directives that correspond to the current project options (including all of the new compiler warning settings), as in the following listing:

```
{ $A8 , B- , C+ , D+ , E- , F- , G+ , H+ , I+ , J- , K- , L+ , M- , N+ , O+ , P+ , Q- , R- , S- , T- , U- , V+ , W- , X+ , Y+ , Z1 }  
{ $MINSTACKSIZE $00004000 }  
{ $MAXSTACKSIZE $00100000 }  
{ $IMAGEBASE $00400000 }  
{ $APPTYPE GUI }  
{ $WARN SYMBOL_DEPRECATED ON }  
{ $WARN SYMBOL_LIBRARY ON }  
{ $WARN SYMBOL_PLATFORM ON }  
{ $WARN UNIT_LIBRARY ON }  
{ $WARN UNIT_PLATFORM ON }  
{ $WARN UNIT_DEPRECATED ON }  
{ $WARN HRESULT_COMPAT ON }  
{ $WARN HIDING_MEMBER ON }  
{ $WARN HIDDEN_VIRTUAL ON }
```



```

{$WARN GARBAGE ON}
{$WARN BOUNDS_ERROR ON}
{$WARN ZERO_NIL_COMPAT ON}
{$WARN STRING_CONST_TRUNCED ON}
{$WARN FOR_LOOP_VAR_VARPAR ON}
{$WARN TYPED_CONST_VARPAR ON}
{$WARN ASG_TO_TYPED_CONST ON}
{$WARN CASE_LABEL_RANGE ON}
{$WARN FOR_VARIABLE ON}
{$WARN CONSTRUCTING_ABSTRACT ON}
{$WARN COMPARISON_FALSE ON}
{$WARN COMPARISON_TRUE ON}
{$WARN COMPARING_SIGNED_UNSIGNED ON}
{$WARN COMBINING_SIGNED_UNSIGNED ON}
{$WARN UNSUPPORTED_CONSTRUCT ON}
{$WARN FILE_OPEN ON}
{$WARN FILE_OPEN_UNITSRC ON}
{$WARN BAD_GLOBAL_SYMBOL ON}
{$WARN DUPLICATE_CTOR_DTOR ON}
{$WARN INVALID_DIRECTIVE ON}
{$WARN PACKAGE_NO_LINK ON}
{$WARN PACKAGED_THREADVAR ON}
{$WARN IMPLICIT_IMPORT ON}
{$WARN HPPEMIT_IGNORED ON}
{$WARN NO_RETVAL ON}
{$WARN USE_BEFORE_DEF ON}
{$WARN FOR_LOOP_VAR_UNDEF ON}
{$WARN UNIT_NAME_MISMATCH ON}
{$WARN NO_CFG_FILE_FOUND ON}
{$WARN MESSAGE_DIRECTIVE ON}
{$WARN IMPLICIT_VARIANTS ON}
{$WARN UNICODE_TO_LOCALE ON}
{$WARN LOCALE_TO_UNICODE ON}
{$WARN IMAGEBASE_MULTIPLE ON}
{$WARN SUSPICIOUS_TYPECAST ON}
{$WARN PRIVATE_PROPACCESSOR ON}
{$WARN UNSAFE_TYPE OFF}
{$WARN UNSAFE_CODE OFF}
{$WARN UNSAFE_CAST OFF}

```

## Compiling and Building Projects

There are several ways to compile a project. If you run the project (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed. If you select Project → Build All instead, every file is compiled, even if it has not changed. You should only need this second command infrequently, because Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options, in which case you have to use the Build All command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi Compiled Unit (DCU). (This step is performed only if the DCU file is not already up to date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option (in the Preferences page of the Environment Options dialog box).

**Warning** Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows a strange error message, the first thing you should try is the Build All command to resynchronize the make feature with the current files on disk.

The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't run unless the proper DLLs (such as `vc170.bpl`, which is quite large) are available on the computer where you want to run the program.

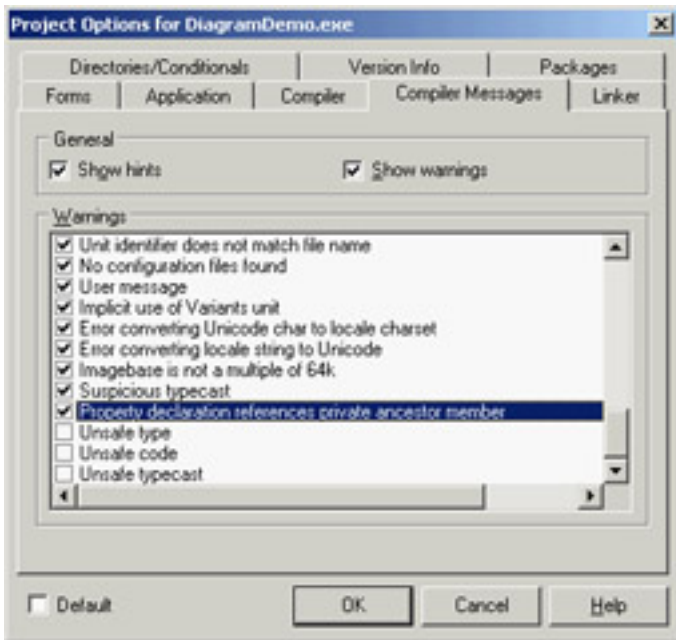
If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course, if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in [Chapter 10](#).

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semicomplied" tools.

## Compiler Message Helpers and Warnings

As I mentioned at the beginning of this chapter (in the section "[Extended Compiler Messages and Search Results in Delphi 7](#)"), in addition to the classic compiler messages, Delphi 7 provides a new window with additional information about some error messages. This window is activated using the View → Additional Message Info menu command. It displays information stored in a local file, which can be updated by downloading a new version from Borland's website.

Another change in Delphi 7 relates to the increased control you have over compiler warnings. The Project Options dialog box now includes a Compiler Messages page where you can choose many individual warnings. This feature was probably introduced due to the fact that Delphi 7 has a new set of warnings related to compatibility with the future Delphi for .NET tool. These warnings are quite extensive, and I've disabled them as shown in [Figure 1.10](#).



**Figure 1.10:** The new Compiler Messages page of the Project Options dialog box

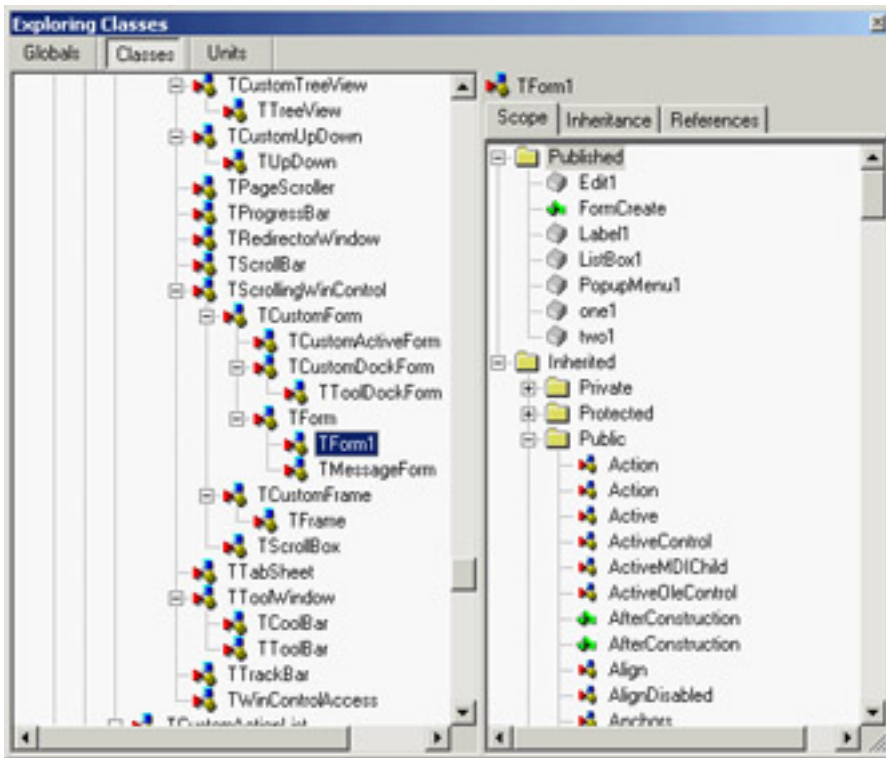
You can also enable or disable some of these warnings using compiler options like these:

```
{ $Warn UNSAFE_CODE OFF }  
{ $Warn UNSAFE_CAST OFF }  
{ $Warn UNSAFE_TYPE OFF }
```

In general, it is better to keep these settings outside the source code of the program—something Delphi 7 finally allows you to do.

## Exploring a Project's Classes

Delphi has always included a tool to browse the symbols of a compiled project, although this tool's name has changed many times (from Object Browser to Project Explorer and now to Project Browser). In Delphi 7, you activate the Project Browser window using the View → Browser menu command, which displays the window shown in [Figure 1.11](#). The browser allows you to see the hierarchical structure of the project's classes and to look for its symbols and the source-code lines where they are referenced.



**Figure 1.11:** The Project Browser

Unlike the Code Explorer, the Project Browser is updated only as you recompile the project. This browser allows you to list classes, units, and globals, and lets you choose whether to look only for symbols defined within your project or for those from both your project and VCL. You can change the settings of the Project Browser and those of the Code Explorer in the Explorer page of the Environment Options or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the categories you see in this window are specific to the Project Browser; others relate to both tools.

## Additional and External Delphi Tools

In addition to the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (`PCE.exe`), and the Image Editor (`ImagEdit.exe`), are available from the Tools menu in the IDE. In addition, the Enterprise edition has a link to the SQL Monitor (`SqlMon.exe`). Other tools that are not directly accessible from the IDE include many command-line utilities you can find in the Delphi `bin` directory. For example, these tools include a command-line Delphi compiler (`DCC32.exe`), a Borland resource compiler (`BRC32.exe` and `BRCC32.exe`), and an executable viewer (`TDump.exe`).

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools, most of which are available in the `\Delphi7\bin` folder and in the Tools menu:

**Web App Debugger** (`WebAppDbg.exe`) The debugging web server introduced in Delphi 6. It is used to keep track of the requests sent to your applications and to debug them. This debugger was rewritten in Delphi 7: It is now a CLX application and its connectivity is based on sockets. I'll discuss this tool in [Chapter 20](#).

**XML Mapper** (`XmlMapper.exe`) A tool for creating XML transformations to be applied to the format produced by the `ClientDataSet` component. You'll find more on this topic in [Chapter 22](#).

**External Translation Manager** (`etm60.exe`) The stand-alone version of the Integrated Translation Manager. This external tool can be given to external translators and was available for the first time in Delphi 6.

**Borland Registry Cleanup Utility** (`D7RegClean.exe`) A tool that helps you remove all the Registry entries that Delphi 7 adds to a computer.

**TeamSource** An advanced version-control system provided with Delphi, starting with version 5. The tool is very similar to its past incarnation and is installed separately from Delphi. Delphi 7 ships with version 1.01 of Team Source, the same version available after applying an available patch to the Delphi 6 version.

**WinSight** (`Ws32.exe`) A Windows "message spy" program available in the `bin` directory.

**Database Explorer** A tool that can be activated from the Delphi IDE or as a stand-alone tool, using the `DBExplorer.exe` program of the `bin` directory. Because it is meant for the BDE, the Database Explorer is not used much nowadays.

**OpenHelp** (`oh.exe`) The tool you can use to manage the structure of Delphi's own Help files, integrating third-party files into the help system.

**Convert** (`Convert.exe`) A command-line tool you can use to convert DFM files into the equivalent textual description and vice versa.

**Turbo Grep** (`Grep.exe`) A command-line search utility, which is much faster than the embedded Find In Files mechanism but not as easy to use.

**Turbo Register Server** (`TRegSvr.exe`) A tool you can use to register ActiveX libraries and COM servers. The source code for this tool is available under `\Demos\ActiveX\TRegSvr`.

**Resource Explorer** A powerful resource viewer (but not a full-blown resource editor) you can find under `\Demos\ResXplor`.

**Resource Workshop** An old 16-bit resource editor that can also manage Win32 resource files. The Delphi installation CD includes a separate installation for Resource Workshop. It was formerly included in Borland C++ and Pascal compilers for Windows and was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long filenames, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files.

Team LiB

◀ PREVIOUS

NEXT ▶

## The Files Produced by the System

Delphi produces various files for each project, and you should know what they are and how they are named. Basically, two elements have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. [Table 1.1](#) lists the extensions of the files you'll find in the directory where a Delphi project resides. The table also shows when or under what circumstances these files are created and their importance for future compilations.

**Table 1.1: Delphi Project File Extensions**

Extension	File Type and Description	Creation Time	Required to Compile?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to endusers.
.CAB	The Microsoft Cabinet compressed-file format used for web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.
.DCP	Delphi Compiled Package: a file with symbol information for the code that was compiled into the package. It doesn't include compiled code, which is stored in DCU files or in the BPL file.	Compilation	Required when you use run-timepackages. You'll distribute it only to other developers along with BPL files. You can compile an application with the units from a package just by having theDCP file and the BPL (and no DCU files).

.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DDP	The Delphi Diagram Portfolio, used by the Diagram view of the editor (was .DTI in Delphi 5)	Development	No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
~DF	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN	Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).	Development (ITE)	Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.
.DLL	Dynamic link library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK and now also .DPKW and .DPKL	Delphi Package: the project source code file of a package (ora specific project file for Windows or Linux).	Development	Yes.
.DPR	Delphi Project file. (This file actually contains Pascal sourcecode.)	Development	Yes.
~DP	Backup of the Delphi Project file( . DPR).	Development	No. This file is generated automatically when you save a new version of a project file.



.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.
.DSM	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.EXE	Executable file: the Windows application you've produced.	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or .HTML, for Hypertext Markup Language: the file format used for Internet web pages.	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
OCX	OLE Control Extension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
.PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.
~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.

.RPS	Translation Repository (part of the Integrated Translation Environment).	Development (ITE)	No. Required to manage the translations.
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
TODO	To-do list file, holding the items related to the entire project.	Development	No. This file hosts notes for the programmers.
.UDL	Microsoft Data Link.	Development	Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see <a href="#">Chapter 12</a> ).

Besides the files generated during the development of a project in Delphi, many others are generated and used by the IDE itself. In [Table 1.2](#), I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

**Table 1.2: Selected Delphi IDE Customization File Extensions**

Extension	File Type
.DCI	Delphi code templates.
.DRO	Delphi's Object Repository. (The repository should be modified with the Tools → Repository command.)
.DMT	Delphi menu templates.
.DBI	Database Explorer information.
.DEM	Delphi edit mask (files with country-specific formats for edit masks).
.DCT	Delphi component templates.
.DST	Desktop settings file (one for each desktop setting you've defined).

## Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time covering their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depends on syntax highlighting, but it isn't saved with the file. It is worth noting that there is a single file for the form's whole code, not just small code fragments.

**Tip** In the listings in this book, I've matched the bold syntax highlighting of the editor for keywords and the italic for strings and comments.

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a .DFM extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is by default a text representation of the form, but it can also be saved in a binary Windows Resource format. You can set the format you want to use for new projects in the Designer page of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command on a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of

a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This command closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can edit the textual description of a form, although you should do so with extreme care. As soon as you save the file, it will be parsed to regenerate the form. If you've made incorrect changes, compilation will stop with an error message; you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have good knowledge of Delphi programming.

**Tip** In the book, I often show you excerpts of DFM files. In most of these excerpts, I show only the relevant components or properties; generally, I have removed the positional properties, the binary values, and other lines providing little useful information.

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application: the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the Project → View Source menu command.

Some of the other, less relevant files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF):

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1
```

```
[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304
```

```
[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

## The Object Repository

Delphi has menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File → New menu and in other pull-down menus. If you simply select File → New → Other, Delphi opens the Object Repository. You can use it to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New Items dialog box (shown in [Figure 1.12](#)) has several pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.



**Figure 1.12:** The first page of the New Items dialog box, generally known as the Object Repository

**Tip** The Object Repository has a shortcut menu you can use to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool—information that is particularly important when you're looking at wizards, projects, or forms that you've added to the Repository.

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and "current project" pages). Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, and it will be ready to use later. Simply use the Project → Add To Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Move to the form that you want to add, and select the Add To Repository command from its shortcut menu. Then, indicate the title, description, author, page, and icon in the resulting dialog box. Keep in mind that as you copy a project or form template to the Repository and then copy it back to another directory, you are simply doing a copy-and-paste operation; this isn't much different than copying the files manually.

---

### The Empty Project Template

When you start a new project, Delphi automatically opens a blank form, too. However, if you want to base a new project on one of the form objects or wizards, you don't need this form. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple:

1. Create a new project as usual.
2. Remove the project's only form.
3. Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you have to remember to use the File → Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools → Repository command. This command opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing elements. You can even add new pages, rename or delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File → New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File → New Application), if no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File → New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the `OBJREPOS` subdirectory of the Delphi main directory. At the same time, if you use a form or any other object directly without copying it, then you end up having some of your project files in this directory. It is important to realize how the Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files without copying data back and forth to the Repository.

---

### Installing New DLL Wizards

Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they are installed the same way you install a component or a package, using the Components → Install Packages menu command and then clicking the Add button. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key `\Software\Borland\Delphi\7.0\Experts`. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.

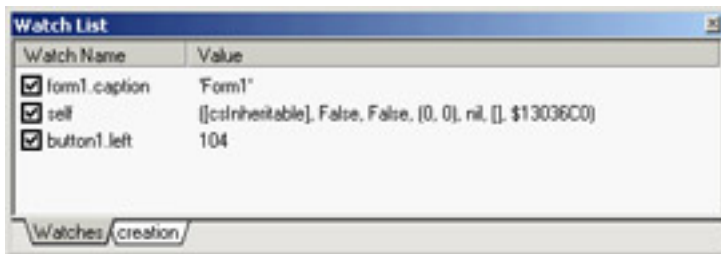
---

## Debugger Updates in Delphi 7

When you run a program in Delphi's IDE, you generally start it in the integrated debugger. You can set breakpoints, execute the code line by line, and explore its inner details, including the assembler code being executed and the use of CPU registries in the CPU view. I don't have space in this book to cover debugging in Delphi; see [Appendix C](#) for information about extra material on this topic. However, I do want to briefly mention a couple of new debugger features.

First, the Run Parameters dialog box in Delphi 7 allows you to set a working directory for the program being debugged. This means the current folder will be the one you indicate, not the one the program has been compiled into.

Another relevant change relates to the Watch List. It now has multiple tabs that let you keep different sets of active watches for different areas of the program you are debugging, without cluttering a single window. You can add a new group to the Watch List by using its shortcut menu, and also alter the visibility of column headers and enable individual watches with the corresponding check boxes.



**Note** This book doesn't cover the Delphi debugger, but this is a very important topic. See the references to online material in [Appendix C](#) for information about how to download a free chapter that discusses debugging in Delphi.

## What's Next?

This chapter has presented an overview of the new and more advanced features of the Delphi 7 programming environment, including tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to begin *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

Now we are ready to spend the [next chapter](#) looking into the Delphi programming language. Then we'll proceed by studying the run-time library (RTL) and the class library included in Delphi.

## Chapter 2: The Delphi Programming Language

### Overview

The Delphi development environment is based on an object-oriented extension of the Pascal programming language known as Object Pascal. Recently, Borland stated its intention to refer to the language as "the Delphi language," probably because the company wants to be able to say that Kylix uses the Delphi language and because Borland will provide the Delphi language on the Microsoft .NET platform. Due to years of habit, I'll use the two names interchangeably.

Most modern programming languages support *object-oriented programming* (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding). Although you can write Delphi code without understanding the core features of its language, you won't be able to master this environment until you fully understand the programming language.

**Note** Due to space constraints and to the fact that the language hasn't changed much in recent years, in this chapter you'll find only a very fast-paced introduction to the language. You can read the more detailed description found in past editions of the book in the material available on my website (see [Appendix C](#), "Free Companion Books on Delphi," for details). This material also includes *Essential Pascal*, a complete introduction to the standard Pascal language.

The following topics are covered in this chapter:

- Classes and objects
- Encapsulation: *private* and *public*
- Using properties
- Constructors
- Objects and memory
- Inheritance
- Virtual methods and polymorphism
- Type-safe down-casting (run-time type information)
- Interfaces
- Working with exceptions
- Class references



## Core Language Features

The Delphi language is an OOP extension of the classic Pascal language, which Borland pushed forward for many years with its Turbo Pascal compilers. The syntax of the Pascal language is known to be quite verbose and more readable than, for example, the C language. Its OOP extension follows the same approach, delivering the same power of the recent breed of OOP languages, from Java to C#.

Even the core language is subject to continuous changes, but few of them will affect your everyday programming needs. In Delphi 6, for example, Borland added support for several features more or less related to the development of Kylix, the Linux version of Delphi:

- A new directive for conditional compilation (`$IF`)
- A set of hint directives (`platform`, `deprecated`, and `library`, of which only the first is used to any extent) and the new `$WARN` directive used to disable them
- A `$MESSAGE` directive to emit custom information among compiler messages

Delphi 7 adds three additional compiler warnings: unsafe type, unsafe code, and unsafe cast. These warnings are emitted in case of operations that you won't be able to use to produce safe "managed" code on the Microsoft .NET platform (more on this in [Chapter 25](#), "Delphi for .NET Preview: The Language and the RTL").

Another change relates to unit names, which can now be formed from multiple words separated by dot, as in the `marco.test` unit, saved in the `marco.test.pas` file. This feature will help support namespaces and more flexible unit references in Delphi for .NET and future versions of the Delphi compiler for Windows, but in Delphi 7 it has limited use.

## Classes and Objects

Delphi is based on OOP concepts, and in particular on the definition of new class types. The use of OOP is partially enforced by the visual development environment, because for every new form defined at design time, Delphi automatically defines a new class. In addition, every component visually placed on a form is an object of a class type available in or added to the system library.

**Note** The terms *class* and *object* are commonly used and often misused, so let's be sure we agree on their definitions. A *class* is a user-defined data type, which has a state (its representation or internal data) and some operations (its behavior or its methods). An *object* is an instance of a class, or a variable of the data type defined by the class. Objects are *actual* entities. When the program runs, objects take up some memory for their internal representation. The relationship between object and class is the same as the one between variable and type.

As in most other modern OOP languages (including Java and C#), in Delphi a class-type variable doesn't provide the storage for the object, but is only a pointer or reference to the object in memory. Before you use the object, you must allocate memory for it by creating a new instance or by assigning an existing instance to the variable:

```
var
    Obj1, Obj2: TMyClass;
begin
    // assign a newly created object
    Obj1 := TMyClass.Create;
    // assign to an existing object
    Obj2 := ExistingObject;
```

The call to `Create` invokes a default constructor available for every class, unless the class redefines it (as described later). To declare a new class data type in Delphi, with some local data fields and some methods, use the following syntax:

```
type
    TDate = class
        Month, Day, Year: Integer;
        procedure SetValue (m, d, y: Integer);
        function LeapYear: Boolean;
    end;
```

**Note** The convention in Delphi is to use the letter *T* as a prefix for the name of every class you write and every other type (*T* stands for *Type*). This is just a convention—to the compiler, *T* is just a letter like any other—but it is so common that following it will make your code easier for other developers to understand.

A method is defined with the *function* or *procedure* keyword, depending on whether it has a return value. Inside the class definition, methods can only be declared; they must be then defined in the *implementation* portion of the same unit. In this case, you prefix each method name with the name of the class it belongs to, using dot notation:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
    Month := m;
    Day := d;
    Year := y;
end;

function TDate.LeapYear: Boolean;
begin
    // call IsLeapYear in SysUtils.pas
    Result := IsLeapYear (Year);
end;
```

**Tip** If you press Ctrl+Shift+C while the cursor is within the class definition, the Class Completion feature of the Delphi editor will generate the skeleton of the definition of the methods declared in a class.

This is how you can use an object of the previously defined class:

```
var
  ADay: TDate;
begin
  // create an object
  ADay := TDate.Create;
  try
    // use the object
    ADay.SetValue (1, 1, 2000);
    if ADay.LeapYear then
      ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
  finally
    // destroy the object
    ADay.Free;
  end;
end;
```

Notice that `ADay.LeapYear` is an expression similar to `ADay.Year`, although the first is a function call and the second a direct data access. You can optionally add parentheses after the call of a function with no parameters. You can find the previous code snippets in the source code of the `Dates1` example; the only difference is that the program creates a date based on the year provided in an edit box.

**Note** The code snippet above uses a *try/finally* block to ensure the destruction of the object even in the case of exceptions in the code. You can find an introduction to the topic of exceptions at the end of this chapter.

## More on Methods

There is a lot more to say about methods. Here are some short notes about the features available in Delphi:

- Delphi supports method *overloading*. This means you can have two methods with the same name, provided that you mark the methods with the `overload` keyword and that the parameter lists of the two methods are sufficiently different. By checking the parameters, the compiler can determine which version you want to call.
- Methods can have one or more parameters with default values. If these parameters are omitted in the method call, they will be assigned the default value.
- Within a method, you can use the `self` keyword to access the current object. When you refer to local data of the object, the reference to `self` is implicit. For example, in the `SetValue` method of the `TDate` class listed earlier, you use `Month` to refer to a field of the current object, and the compiler translates `Month` into `Self.Month`.
- You can define class methods, marked by the `class` keyword. A class method doesn't have an object instance to act upon, because it can be applied to an object of the class or to the class as a whole. Delphi doesn't (currently) have a way to define class data, but you can mimic this functionality by adding global data in the implementation portion of the unit defining the class.
- By default, methods use the `register` calling convention: (simple) parameters and return values are passed from the calling code to the function and back using CPU registers, instead of the stack. This process makes method calls much faster.

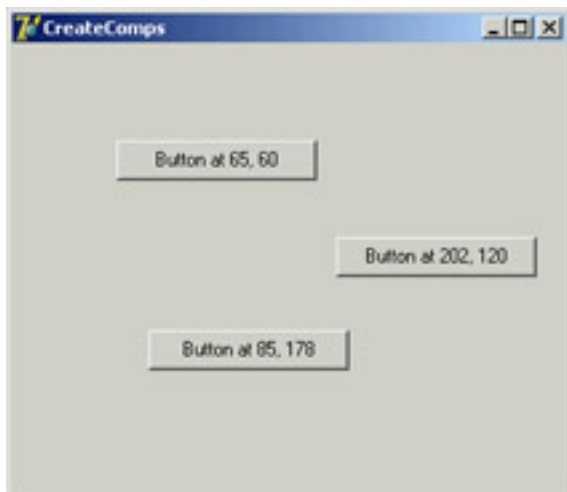
## Creating Components Dynamically

To emphasize the fact that Delphi components aren't much different from other objects (and also to demonstrate the

use of the `Self` keyword), I've written the `CreateComps` example. This program has a form with no components and a handler for its `OnMouseDown` event, which I've chosen because it receives as a parameter the position of the mouse click (unlike the `OnClick` event). I need this information to create a button component in that position. Here is the method's code:

```
procedure TForm1.FormMouseDown (Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
var  
    Btn: TButton;  
begin  
    Btn := TButton.Create (Self);  
    Btn.Parent := Self;  
    Btn.Left := X;  
    Btn.Top := Y;  
    Btn.Width := Btn.Width + 50;  
    Btn.Caption := Format ('Button at %d, %d', [X, Y]);  
end;
```

The effect of this code is to create buttons at mouse-click positions, as you can see in [Figure 2.1](#). In the code, notice in particular the use of the `Self` keyword as both the parameter of the `Create` method (to specify the component's owner) and the value of the `Parent` property. I'll discuss these two elements (ownership and the `Parent` property) in [Chapter 4](#), "Core Library Classes."



**Figure 2.1:** The output of the `CreateComps` example, which creates `Button` components at run time

When writing code like this, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't make any practical difference; but if there are multiple instances of a form, using `Form1` would be an error. In fact, if the `Form1` variable refers to the first form of that type being created, then by clicking in another form of the same type, the new button will always be displayed in the first form. The button's `Owner` and `Parent` will be `Form1`, not the form the user has clicked. In general, referring to a particular instance of a class when the current object is required is bad OOP practice.

## Encapsulation

A class can have any amount of data and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated*, inside the class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, because the methods can check whether the date is valid and refuse to modify the new value if it is not. Encapsulation is important because it allows the class writer to modify the internal representation in a future version.

The concept of encapsulation is often indicated by the idea of a "black box." You don't know about the internals: You only know how to interface with the black box or use it regardless of its internal structure. The "how to use" portion, called the *class interface*, allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Of course, you are supposed to use methods to access the data, which is shielded from unauthorized access. This is the object-oriented approach to a classical programming concept known as *information hiding*. However, in Delphi there is the extra level of hiding, through properties, as we'll see later in this chapter.

Delphi implements this class-based encapsulation, but it still supports the classic module-based encapsulation using the structure of units. Every identifier that you declare in the interface portion of a unit becomes visible to other units of the program, provided there is a `uses` statement referring back to the unit that defines the identifier. On the other hand, identifiers declared in the implementation portion of the unit are local to that unit.

## Private, Protected, and Public

For class-based encapsulation, the Delphi language has three access specifiers: `private`, `protected`, and `public`. A fourth, `published`, controls run-time type information (RTTI) and design-time information (as discussed in more detail in [Chapter 4](#)), but it gives the same programmatic accessibility as `public`. Here are the three *classic* access specifiers:

- The `private` directive denotes fields and methods of a class that are not accessible outside the unit that declares the class.
- The `protected` directive is used to indicate methods and fields with limited visibility. Only the current class and its inherited classes can access `protected` elements. More precisely, only the class, subclasses, and any code in the same unit as the class can access protected members, which opens up to a trick discussed in the sidebar "[Accessing Protected Data of Other Classes \(or, the "Protected Hack"\)](#)" later in this chapter. We'll discuss this keyword again in the section "[Protected Fields and Encapsulation](#)."
- The `public` directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.

Generally, the fields of a class should be private and the methods public. However, this is not always the case. Methods can be private or protected if they are needed only internally to perform some partial computation or to implement properties. Fields might be declared as protected so that you can manipulate them in inherited classes, although this isn't considered a good OOP practice.

**Warning** Access specifiers only restrict code outside your unit from accessing certain members of classes declared in the interface section of your unit. This means that if two classes are in the same unit, there is no protection for their private fields.

As an example, consider this new version of the `TDate` class:

```
type
  TDate = class
  private
```

```

Month, Day, Year: Integer;
public
  procedure SetValue (y, m, d: Integer); overload;
  procedure SetValue (NewDate: TDateTime); overload;
  function LeapYear: Boolean;
  function GetText: string;
  procedure Increase;
end;

```

You might think of adding other functions, such as `GetDay`, `GetMonth`, and `GetYear`, which return the corresponding private data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class you are implementing.

Another new method is the `Increase` procedure, which increases the date by one day. This calculation is far from simple, because you need to consider the different lengths of the various months as well as leap and non-leap years. To make it easier to write the code, I'll change the internal implementation of the class to Delphi's `TDateTime` type for the internal implementation. The class definition will change to the following (the complete code is in the `DateProp` example):

```

type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;

```

Notice that because the only change is in the `private` portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

**Note** The `TDateTime` type is a floating-point number. The integral portion of the number indicates the date since 12/30/1899, the same base date used by OLE Automation and Microsoft Win32 applications. (Use negative values to express previous years.) The decimal portion indicates the time as a fraction. For example, a value of 3.75 stands for the second of January 1900, at 6:00 a.m. (three-quarters of a day). To add or subtract dates, you can add or subtract the number of days, which is much simpler than adding days with a day/month/year representation.

## Encapsulating with Properties

Properties are a very sound OOP mechanism, or a well-thought-out application of the idea of encapsulation. Essentially, you have a name that completely hides its implementation details. This allows you to modify the class extensively without affecting the code using it. A good definition of properties is that of *virtual fields*. From the perspective of the user of the class that defines them, properties look exactly like fields, because you can generally read or write their value. For example, you can read the value of the `Caption` property of a button and assign it to the `Text` property of an edit box with the following code:

```
Edit1.Text := Button1.Caption;
```

It looks like you are reading and writing fields. However, properties can be directly mapped to data, as well as to access methods, for reading and writing the value. When properties are mapped to methods, the data they access can be part of the object or outside of it, and they can produce side effects, such as repainting a control after you change one of its values. Technically, a property is an identifier that is mapped to data or methods using a `read` and a `write` clause. For example, here is the definition of a `Month` property for a date class:

```
property Month: Integer read FMonth write SetMonth;
```

To access the value of the `Month` property, the program reads the value of the private field `FMonth`; to change the property value, it calls the method `SetMonth` (which must be defined inside the class, of course).

Different combinations are possible (for example, you could also use a method to read the value or directly change a field in the `write` directive), but the use of a method to change the value of a property is common. Here are two alternative definitions for the property, mapped to two access methods or mapped directly to data in both directions:

```
property Month: Integer read GetMonth write SetMonth;
```

```
property Month: Integer read FMonth write FMonth;
```

Often, the actual data and access methods are private (or protected), whereas the property is public. For this reason, you must use the property to have access to those methods or data, a technique that provides both an extended and a simplified version of encapsulation. It is an *extended* encapsulation because not only can you change the representation of the data and its access functions, but you can also add or remove access functions without changing the calling code. A user only needs to recompile the program using the property.

**Tip** When you're defining properties, take advantage of the extended class completion feature of Delphi's editor, which you activate with the Ctrl+Shift+C key combination. After you write the property name, type, and semicolon, press Ctrl+Shift+C, and Delphi will provide you with a complete definition and the skeleton of the *setter* method. Write *Get* in front of the name of the identifier after the *read* keyword, and you'll also have a *getter* method with almost no typing.

## Properties for the *TDate* Class

As an example, I've added properties for accessing the year, the month, and the day to an object of the `TDate` class discussed earlier. These properties are not mapped to specific fields, but they all map to the single `fDate` field storing the complete date information. This is why all the properties have both getter and setter methods:

```
type
  TDate = class
  public
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
```

Each of these methods is easily implemented using functions available in the `DateUtils` unit (more details in [Chapter 3](#), "The Run Time Library"). Here is the code for two of them (the others are very similar):

```
function TDate.GetYear: Integer;
begin
  Result := YearOf (fDate);
end;
```

```
procedure TDate.SetYear(const Value: Integer);
begin
  fDate := RecodeYear (fDate, Value);
end;
```

The code for this class is available in the `DateProp` example. The program uses a secondary unit for the definition of the `TDate` class to enforce encapsulation and creates a single-date object that is stored in a form variable and kept in memory for the entire execution of the program. Using a standard approach, the object is created in the form `OnCreate` event handler and destroyed in the form `OnDestroy` event handler. The program form (see [Figure 2.2](#)) has three edit boxes and buttons to copy the values of these edit boxes to and from the properties of the date object.



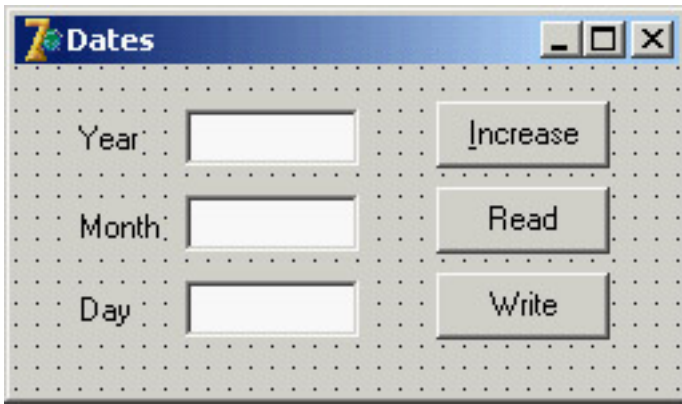


Figure 2.2: The DateProp example's form

**Warning** When writing the values, the program uses the `SetValue` method instead of setting each of the properties. Assigning the month and the day separately can cause you trouble when the month is not valid for the current day. For example, suppose the day is currently January 31, and you want to assign to it February 20. If you assign the month first, this part of the assignment will fail, because February 31 does not exist. If you assign the day first, the problem will arise when doing the reverse assignment. Due to the validity rules for dates, it is better to assign everything at once.

## Advanced Features of Properties

Properties have several advanced features I'll focus on in future chapters. Specifically, in [Chapter 4](#) I'll cover the `TPersistent` class, RTTI, and streaming and I'll discuss writing custom Delphi components in [Chapter 9](#), "Writing Delphi Components." Here is a short summary of these more advanced features:

- The `write` directive of a property can be omitted, making it a *read-only* property. The compiler will issue an error if you try to change the property value. You can also omit the `read` directive and define a *write-only* property, but that approach doesn't make much sense and is used infrequently.
- The Delphi IDE gives special treatment to *design-time* properties, which are declared with the `published` access specifier and generally displayed in the Object Inspector for the selected component. You'll find more on the `published` keyword and its effect in [Chapter 4](#).
- An alternative is to declare properties, often called *run-time only* properties, with the `public` access specifier. These properties can be used in program code.
- You can define *array-based* properties, which use the typical notation with square brackets to access an element of a list. The *string list-based* properties, such as the `Lines` of a list box, are a typical example of this group.
- Properties have special directives, including `stored` and `default`, which control the *component streaming system* (introduced in [Chapter 4](#) and detailed in [Chapter 9](#)).

**Note** You can usually assign a value to a property or read it, and you can even use properties in expressions, but you cannot always pass a property as a parameter to a procedure or method. This is the case because a property is not a memory location, so it cannot be used as a `var` or `out` parameter; it cannot be passed by reference.

## Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a class's field, you only need to change the code of some methods of that class and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.



Let me clarify this idea with an example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of one of the forms:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

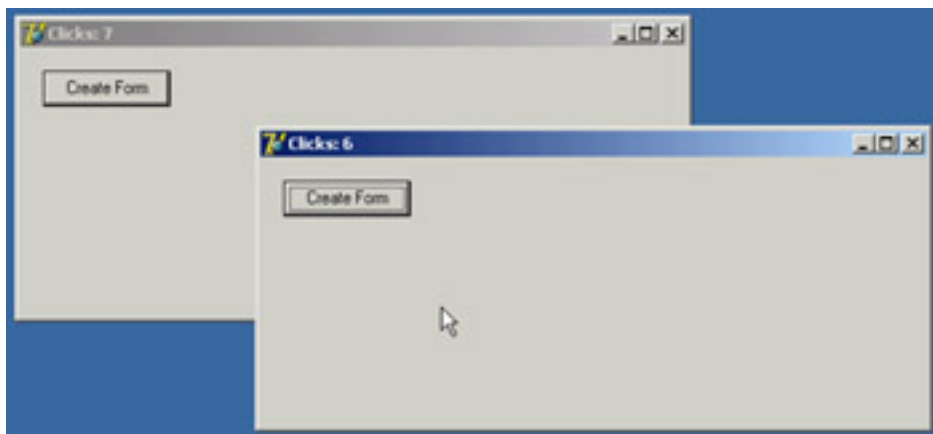
This approach works, but the data is connected to the entire program rather than a specific instance of the form. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

## Adding Properties to Forms

The previous class uses public data, so for the sake of encapsulation, you should instead change it to use private data and data-access functions. An even better solution is to add a property to the form. Every time you want to make some information of a form available to other forms, you should use a property, for all the reasons discussed in the section "[Encapsulating with Properties](#)." To do so, change the field declaration of the form (in the previous code) by adding the keyword `property` in front of it, and then press Ctrl+Shift+C to activate code completion. Delphi will automatically generate all the extra code you need.

The complete code for this form class is available in the FormProp example and illustrated in [Figure 2.3](#). The program can create multi-instances of the form (that is, multiple objects based on the same form class), each with its own click count.



**Figure 2.3:** Two forms of the FormProp example at run time

**Note** Notice that adding a property to a form doesn't add to the list of the form properties in the Object Inspector.

In my opinion, properties should also be used in the form classes to encapsulate the access to the components of a form. For example, if you have a main form with a status bar used to display some information (and with the `SimplePanel` property set to `True`) and you want to modify the text from a secondary form, you might be tempted to write

```
Form1.StatusBar1.SimpleText := 'new text';
```

This is a standard practice in Delphi, but it's not a good one, because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (for example, replacing `StatusBar` with another control or activating multiple panels), you'll have to fix the code in many places. The alternative is to use a method or, even better, a property to hide

the specific control. This property can be defined as

```
property StatusText: string read GetText write SetText;
```

with `GetText` and `SetText` methods that read from and write to the `SimpleText` property of the status bar (or the caption of one of its panels). In the program's other forms, you can refer to the form's `StatusText` property; and if the user interface changes, only the setter and getter methods of the property are affected.

**Note** See [Chapter 4](#) for a detailed discussion of how you can avoid having published form fields for components, which will improve encapsulation. But don't rush there: The description requires a good knowledge of Delphi, and the technique discussed has a few drawbacks.

Team LiB

◀ PREVIOUS

NEXT ▶

## Constructors

So far, to allocate memory for objects, I've called the `Create` method. This is a *constructor*— a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later. All the data of the new instance is set to zero. If you want your instance data to start out with specific values, then you need to write a custom constructor to do that.

Use the `constructor` keyword in front of your constructor. Although you can use any name for a constructor, you should stick to the standard name, `Create`. If you use a name other than `Create`, the `Create` constructor of the base `TObject` class will still be available, but a programmer calling this default constructor might bypass the initialization code you've provided because they don't recognize the name.

By defining a `Create` constructor with some parameters, you replace the default definition with a new one and make its use compulsory. For example, after you define

```
type
  TDate = class
  public
    constructor Create (y, m, d: Integer);
```

you'll only be able to call this constructor and not the standard `Create`:

```
var
  ADay: TDate;
begin
  // Error, does not compile:
  ADay := TDate.Create;
  // OK:
  ADay := TDate.Create (1, 1, 2000);
```

The rules for writing constructors for custom components are different, as you'll see in [Chapter 9](#). The reason is that in this case you have to override a virtual constructor. Overloading is particularly relevant for constructors, because you can add multiple constructors to a class and call them all `Create`; this approach makes the constructors easy to remember and follows a standard path provided by other OOP languages in which constructors must all have the same name. As an example, I've added to the class two separate `Create` constructors: one with no parameters, which hides the default constructor; and one with initialization values. The constructor with no parameter uses as the default value today's date (as you can see in the complete code of the `DataView` example):

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
```

## Destructors and the *Free* Method

In the same way that a class can have a custom constructor, it can have a custom destructor—a method declared with the `destructor` keyword and called `Destroy`. Just as a constructor call allocates memory for the object, a destructor call frees the memory. Destructors are needed only for objects that acquire *external* resources in their constructors or during their lifetime. You can write custom code for a destructor, generally overriding the default `Destroy` destructor, to let an object execute some clean-up code before it is destroyed.

`Destroy` is a virtual destructor of the `TObject` class. You should never define a different destructor, because objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor of the specific class (virtual methods will be discussed later in this chapter).

`Free` is a method of the `TObject` class, inherited by all other classes. The `Free` method basically checks whether the current object (`Self`) is not `nil` before calling the `Destroy` virtual destructor. `Free` doesn't set the object to `nil` automatically; this is something you should do yourself! The object doesn't know which variables may be referring to it, so it has no way to set them all to `nil`.

Delphi 5 introduced a `FreeAndNil` procedure you can use to free an object and set its reference to `nil` at the same time. Call `FreeAndNil(Obj1)` instead of writing the following:

```
Obj1.Free;  
Obj1 := nil;
```

**Note** There's more on this topic in the section "[Destroying Objects Only Once](#)" later in this chapter.

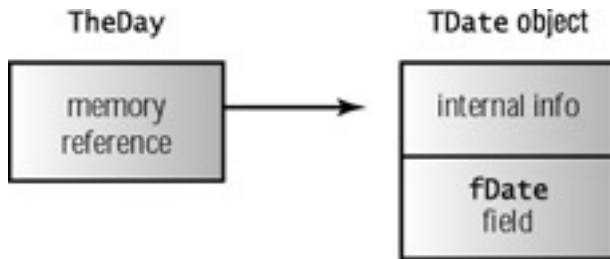
**Team LiB**

◀ PREVIOUS

NEXT ▶

## Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Delphi, instead, is based on an *object reference model*. The idea is that a variable of a class type, such as the `TheDay` variable in the preceding `ViewDate` example, does not hold the value of the object. Rather, it contains a reference, or a *pointer*, to indicate the memory location where the object has been stored. You can see this structure depicted in [Figure 2.4](#).



**Figure 2.4:** A representation of the structure of an object in memory, with a variable referring to it

The only problem with this approach is that when you declare a variable, you don't create an object in memory (which is inconsistent with all other variables, confusing new users of Delphi); you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of the components you place on a form are built automatically by the Delphi library.

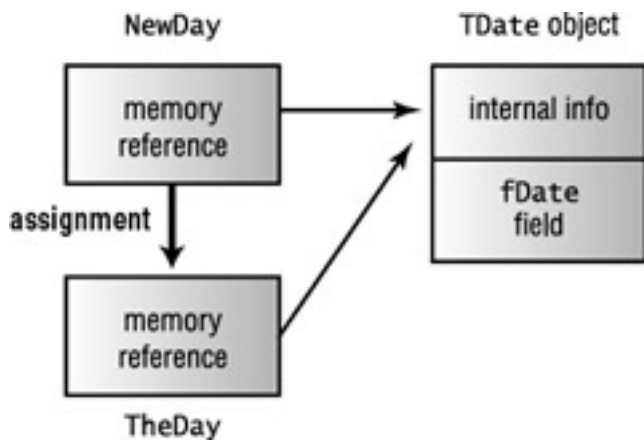
You've seen how to create an instance of an object by applying a constructor to its class. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the `Free` method. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch. The object reference model has many consequences on assigning object and on managing memory, as you'll see in the next two sections.

## Assigning Objects

If a variable holding an object only contains a reference to the object in memory, what happens if you copy the value of that variable? Suppose you write the `BtnTodayClick` method of the `ViewDate` example in the following way:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
    NewDay: TDate;
begin
    NewDay := TDate.Create;
    TheDay := NewDay;
    LabelDate.Caption := TheDay.GetText;
end;
```

This code copies the memory address of the `NewDay` object to the `TheDay` variable (as shown in [Figure 2.5](#)); it doesn't copy the data of one object into the other. In this particular circumstance, this is not a very good approach—you keep allocating memory for a new object every time the button is clicked, but you never release the memory of the object the `TheDay` variable was previously pointing to.



**Figure 2.5:** A representation of the operation of assigning an object reference to another object. This is different from copying the actual content of an object to another.

This specific issue can be solved by freeing the old object, as in the following code (which is also simplified, without the use of an explicit variable for the newly created object):

```

procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    TheDay.Free;
    TheDay := TDate.Create;

```

The important thing to keep in mind is that, when you assign an object to another object, Delphi copies the reference to the object in memory to the new object reference. You should not consider this a negative: In many cases, being able to define a variable referring to an existing object can be a plus. For example, you can store the object returned by accessing a property and use it in subsequent statements, as this code snippet indicates:

```

var
    ADay: TDate;
begin
    ADay := UserInformation.GetBirthDate;
    // use a ADay

```

The same thing happens if you pass an object as a parameter to a function: You don't create a new object, but you refer to the same one in two different places in the code. For example, by writing this procedure and calling it as follows, you'll modify the `Caption` property of the `Button1` object, not of a copy of its data in memory (which would be totally useless):

```

procedure CaptionPlus (Button: TButton);
begin
    Button.Caption := Button.Caption + '+';
end;

// call...
CaptionPlus (Button1)

```

This means that the object is being passed by reference without the use of the `var` keyword and without any other obvious indication of the pass-by-reference semantic, which also confuses newcomers. What if you really want to change the data inside an existing object, so that it matches the data of another object? You have to copy each field of the object, which is possible only if they are all public, or provide a specific method to copy the internal data. Some classes of the VCL have an `Assign` method, which performs this copy operation. To be more precise, most of the VCL classes that inherit from `TPersistent`, but do not inherit from `TComponent`, have the `Assign` method. Other `TComponent`-derived classes have this method but raise an exception when it is called.

In the `DateCopy` example, I've added an `Assign` method to the `TDate` class and called it from the `Today` button, with the following code:

```

procedure TDate.Assign (Source: TDate);

```

```

begin
    fDate := Source.fDate;
end;

procedure TDateForm.BtnTodayClick(Sender: TObject);
var
    NewDay: TDate;
begin
    NewDay := TDate.Create;
    TheDay.Assign(NewDay);
    LabelDate.Caption := TheDay.GetText;
    NewDay.Free;
end;

```

## Objects and Memory

Memory management in Delphi is subject to three rules, at least if you allow the system to work in harmony without Access Violations and without consuming unneeded memory:

- Every object must be created before it can be used.
- Every object must be destroyed after it has been used.
- Every object must be destroyed only once.

Whether you must perform these operations in your code or can let Delphi handle memory management for you depends on the model you choose among the different approaches provided by Delphi.

Delphi supports three types of memory management for dynamic elements:

- Every time you create an object explicitly in your application code, you should also free it (with the only exception of a handful of system objects and of objects that are used through interface references). If you fail to do so, the memory used by that object won't be released for other objects until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form) becomes responsible for destroying all the objects it owns. In other words, when you free a form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it. This is the standard behavior of the components you create at design time by placing them on a form or data module. However, it is mandatory that you choose an owner that you can guarantee will be destroyed; for example, forms are generally owned by the global `Application` objects, which is destroyed by the library when the program ends.
- When Delphi's RTL allocates memory for strings and dynamic arrays, it will automatically free the memory when the reference goes out of scope. You don't need to free a string: When it becomes unreachable, its memory is released.

## Destroying Objects Only Once

If you call the `Free` method (or call the `Destroy` destructor) of an object twice, you get an error. However, if you remember to set the object to `nil`, you can call `Free` twice with no problem.

**Note** You might wonder why you can safely call `Free` if the object reference is `nil`, but you can't call `Destroy`. The reason is that `Free` is a known method at a given memory location, whereas the virtual function `Destroy` is determined at run time by looking at the type of the object—a very dangerous operation if the object no longer exists.

To sum things up, here are a couple of guidelines:

- Always call `Free` to destroy objects, instead of calling the `Destroy` destructor.
- Use `FreeAndNil`, or set object references to `nil` after calling `Free`, unless the reference is going out of scope immediately afterward.

In general, you can also check whether an object is `nil` by using the `Assigned` function. The following two statements are equivalent in most cases:

```
if Assigned (ADate) then ...  
if ADate <> nil then ...
```

Notice that these statements test only whether the pointer is not `nil`; they do not check whether it is a valid pointer. If you write the following code, the test will be satisfied, and you'll get an error on the line with the call to the object method:

```
ToDestroy.Free;  
if ToDestroy <> nil then  
    ToDestroy.DoSomething;
```

It is important to realize that calling `Free` doesn't set the object to `nil`.

**Team LiB**

◀ PREVIOUS

NEXT ▶



## Inheriting from Existing Types

You'll often need to use a slightly different version of an existing class. For example, you might need to add a new method or slightly change an existing one. If you copy and paste the original class and then modify it (certainly a terrible programming practice, unless there is a specific reason to do so), you'll duplicate your code, bugs, and headaches. Instead, in such a circumstance you should use a key feature of OOP: *inheritance*.

To inherit from an existing class in Delphi, you only need to indicate that class at the beginning of the declaration of the new class. For example, this is done each time you create a new form:

```
type
  TForm1 = class(TForm)
  end;
```

This definition indicates that the `TForm1` class inherits all the methods, fields, properties, and events of the `TForm` class. You can call any public method of the `TForm` class for an object of the `TForm1` type. `TForm`, in turn, inherits some of its methods from another class, and so on, up to the `TObject` base class.

As an example of inheritance, you can derive a new class from `TDate` and modify its `GetText` function. You can find this code in the `Dates` unit of the `NewDate` example:

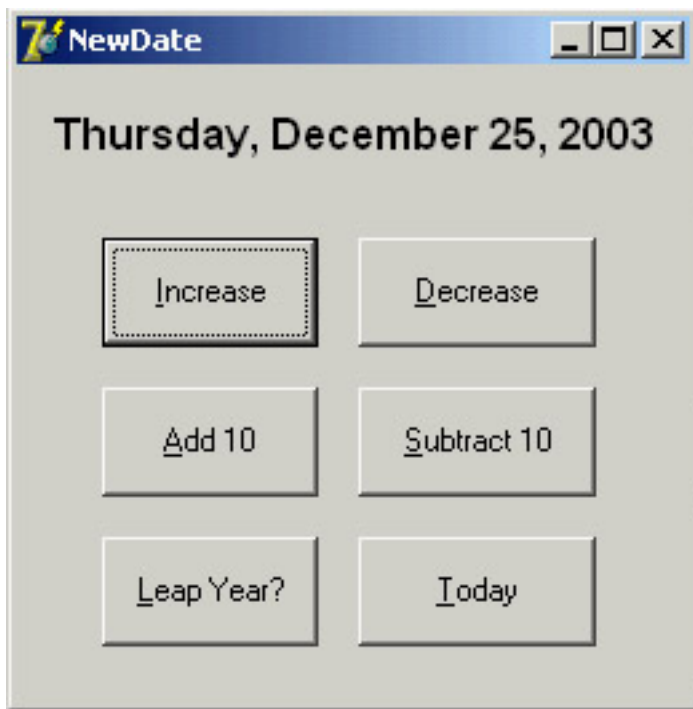
```
type
  TNewDate = class (TDate)
  public
    function GetText: string;
  end;
```

To implement the new version of the `GetText` function, I used the `FormatDateTime` function, which uses (among other features) the predefined month names available in Windows; these names depend on the user's regional and language settings. (Many of these regional settings are copied by Delphi into constants defined in the library, such as `LongMonthNames`, `ShortMonthNames`, and others you can find under the "Currency and date/time formatting variables" topic in the Delphi Help file.) Here is the `GetText` method, where `'dddddd'` stands for the long date format:

```
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

**Tip** Using regional information, the `NewDate` program automatically adapts itself to different Windows user settings. If you run this program on a computer with regional settings referring to a language other than English, it will automatically show month names in that language. To test this behavior, you just need to change the regional settings. Notice that regional-setting changes immediately affect the running programs.

Once you have defined the new class, you need to use this new data type in the code of the form of the `NewDate` example, defining the `TheDay` object of type `TNewDate` and creating an object of this new class in the `FormCreate` method. You don't have to change the code with method calls, because the inherited methods still work exactly in the same way; however, their effect changes, as the new output demonstrates (see [Figure 2.6](#)).



**Figure 2.6:** The output of the NewDate program, with the name of the month and of the day depending on Windows regional settings

## Protected Fields and Encapsulation

The code of the `GetText` method of the `TNewDate` class compiles only if it is written in the same unit as the `TDate` class. In fact, it accesses the `fDate` private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the `fDate` field as protected or add a protected access method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write inherited classes. However, this approach violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If 10 derived classes are accessing this data, changing its definition means potentially modifying the code in each of the 10 classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate solution can be obtained by using a virtual method, a topic I'll discuss in detail in the section "[Late Binding and Polymorphism](#)." If you choose not to use encapsulation in order to obtain faster coding of the inherited classes, then your design might not follow the object-oriented principles.

---

### Accessing Protected Data of Other Classes (or, the "Protected Hack")

You've seen that in Delphi, the private and protected data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this class (part of the Protection example):

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  end;
```

Once you place this class in its own unit, you won't be able to access its protected portion from other units directly. Accordingly, if you write the following code

```
var
```

```
Obj: TTest;  
begin  
  Obj := TTest.Create;  
  Obj.ProtectedData := 20; // won't compile
```

the compiler will issue an error message, "Undeclared identifier: 'ProtectedData.'" At this point, you might think there is no way to access the protected data of a class defined in a different unit. However, there is a way. Consider what happens if you create an apparently useless derived class, such as the following:

```
type  
  TTestHack = class (TTest);
```

Now, if you make a direct cast of the object to the new class and access the protected data through it, this is how the code will look:

```
var  
  Obj: TTest;  
begin  
  Obj := TTest.Create;  
  TTestHack (Obj).ProtectedData := 20; // compiles!
```

This code compiles and works properly, as you can see by running the Protection program. How is it possible for this approach to work? Well, if you think about it, the `TTestHack` class automatically inherits the protected fields of the `TTest` base class, and because the `TTestHack` class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible. As you would expect, if you move the declaration of the `TTestHack` class to a secondary unit, the program will no longer compile.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are times when using this technique is the best solution, as you'll see by looking at the VCL source code and the code of many Delphi components. Two examples that come to mind are accessing the `Text` property of the `TControl` class and the `Row` and `Col` positions of the `DBGrid` control. These two ideas are demonstrated by the `TextProp` and `DBGridCol` examples, respectively. (These examples are quite advanced, so I suggest that only programmers with a good background in Delphi programming read them at this point in the text—other readers might come back later.) Although the first example shows a reasonable example of using the typecast *cracker*, the `DBGrid` example of `Row` and `Col` is a counterexample—it illustrates the risks of accessing bits that the class writer chose not to expose. The `row` and `column` of a `DBGrid` do not mean the same thing as they do in a `DrawGrid` or `StringGrid` (the base classes). First, `DBGrid` does not count the fixed cells as actual cells (it distinguishes data cells from decoration), so your `row` and `column` indexes will have to be adjusted by whatever decorations are currently in effect on the grid (and those can change on the fly). Second, the `DBGrid` is a virtual view of the data. When you scroll up in a `DBGrid`, the data may move underneath it, but the currently selected row might not change.

This technique—declaring a local type only so that you can access protected data members of a class—is often described as a *hack*, and it should be avoided whenever possible. The problem is not accessing protected data of a class in the same unit but declaring a class for the sole purpose of accessing protected data of an existing object of a different class! The danger of this technique is in the hard-coded typecast of an object from a class to a different one.

---

## Inheritance and Type Compatibility

Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a Boolean variable, unless you use an explicit typecast. The rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type refers to a single type definition. To simplify your life, Delphi makes some predefined types assignment compatible: you can assign an `Extended` to a `Double` and vice versa, with automatic promotion or demotion (and potential accuracy loss).

**Warning** If you redefine the same data type in two different units, the types won't be compatible, even if their names are identical. A program using two equally named types from two different units will be a nightmare to compile and debug.

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. You can do so because a dog is an animal! As a general rule, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
var
  MyAnimal: TAnimal;
  MyDog: TDog;
begin
  MyAnimal := MyDog; // This is OK
  MyDog := MyAnimal; // This is an error!!!
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Late Binding and Polymorphism

Pascal functions and procedures are usually based on *static* or *early binding*. This means that a method call is resolved by the compiler and linker, which replace the request with a call to the specific memory location where the function or procedure resides (the routine's address). OOP languages allow the use of another form of binding, known as *dynamic* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

This technique is known as *polymorphism* (a Greek word meaning "many forms"). Polymorphism means you can call a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the class of the object the variable refers to, because of the type-compatibility rule discussed in the [previous section](#). The advantage of polymorphism is being able to write simpler code, treating disparate object types as if they were the same and getting the correct runtime behavior.

For example, suppose that a class and an inherited class (let's say `TAnimal` and `TDog`) both define a method, and this method has late binding. You can apply this method to a generic variable, such as `MyAnimal`, which at run time can refer either to an object of class `TAnimal` or to an object of class `TDog`. The actual method to call is determined at run time, depending on the class of the current object.

The `PolyAnimals` example demonstrates this technique. The `TAnimal` and `TDog` classes have a `Voice` method that outputs the sound made by the selected animal, both as text and as sound (using a call to the `PlaySound` API function defined in the `MMSystem` unit). The `Voice` method is defined as `virtual` in the `TAnimal` class and is later overridden when you define the `TDog` class, by the use of the `virtual` and `override` keywords:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

The effect of the call `MyAnimal.Voice` depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice`, instead. This happens only because the function is `virtual` (you can experiment by removing this keyword and recompiling).

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined in other units—or that haven't been written yet! The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` inherited classes.

**Note** This is the key technical reason why object-oriented programming languages favor reusability. You can write code that uses classes within a hierarchy without any knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition: The ancestor classes of the hierarchy need to be designed very carefully.

In [Figure 2.7](#), you can see an example of the output of the `PolyAnimals` program. By running it, you'll also hear the corresponding sounds produced by the `PlaySound` API call.



Figure 2.7: The output of the PolyAnimals example

## Overriding and Redefining Methods

As you have just seen, to override a late-bound method in a descendant class, you need to use the `override` keyword. Note that this can take place only if the method was defined as `virtual` (or `dynamic`) in the ancestor class. Otherwise, if it is a static method, there is no way to activate late binding, other than to change the code of the ancestor class.

The rules are simple: A method defined as `static` remains `static` in every inherited class, unless you hide it with a new `virtual` method having the same name. A method defined as `virtual` remains late-bound in every inherited class (unless you hide it with a `static` method, which is quite a foolish thing to do). There is no way to change this behavior, because of the way the compiler generates different code for late-bound methods.

To redefine a `static` method, you add a method to an inherited class having the same parameters or different parameters than the original one, without any further specifications. To override a `virtual` method, you must specify the same parameters and use the `override` keyword:

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;

  TMyDerivedClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

Typically, you can override a method two ways: replace the method of the ancestor class with a new version, or add more code to the existing method. This can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write:

```
procedure TMyDerivedClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

When you override an existing `virtual` method of a base class, you must use the same parameters. When you introduce a new version of a method in a descendant class, you can declare it with the parameters you want. In fact, this will be a

new method unrelated to the ancestor method of the same name—they only happen to use the same name. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMyDerivedClass = class (TMyClass)
    procedure One (S: string);
  end;
```

**Note** Using the class definitions just given, when you create an object of the `TMyDerivedClass` class, you can call its `One` method with the string parameter, but not the parameter-less version defined in the base class. If this is what you need, it can be accomplished by marking the redeclared method (the one in the derived class) with the `overload` keyword. If the method has different parameters than the version in the base class, it becomes effectively an overloaded method; otherwise it replaces the base class method. Notice that the method doesn't need to be marked with `overload` in the base class. However, if the method in the base class is virtual, the compiler issues the warning "Method 'One' hides virtual method of base type 'TMyClass.'" To avoid this message and to instruct the compiler more precisely on your intentions, you can use the `reintroduce` directive. If you are interested in this advanced topic, you can find this code in the `Reintr` example and experiment with it further.

## Virtual versus Dynamic Methods

In Delphi, there are two ways to activate late binding. You can declare the method as virtual, as you have seen, or declare it as dynamic. The syntax of the `virtual` and `dynamic` keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

Virtual methods are based on a *virtual method table* (VMT, also known as a *vtable*), which is an array of method addresses. For a call to a virtual method, the compiler generates code to jump to an address stored in the *n*th slot in the object's virtual method table. VMTs allow fast execution of the method calls, but they require an entry for each virtual method for each descendant class, even if the method is not overridden in the inherited class.

Dynamic method calls, on the other hand, are dispatched using a unique number indicating the method, which is stored in a class only if the class defines or overrides it. The search for the corresponding function is generally slower than the one-step table lookup for virtual methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method.

## Message Handlers

A late-bound method can be used to handle a Windows message, too, although the technique is somewhat different. For this purpose Delphi provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter. The `message` directive is followed by the number of the Windows message the method wants to handle.

**Warning** The `message` directive is also available in Kylix and is fully supported by the language and the run-time library (RTL). However, the visual portion of the CLX application framework does not use message methods to dispatch notifications to controls. For this reason, whenever possible, you should use a virtual method provided by the library rather than handle a Windows message directly. Of course, this matters only if you want your code to be more portable.

For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `wm_User` Windows constant:

```
type
  TForm1 = class(TForm)
    ...
```



```
procedure WMUser (var Msg: TMessage);  
    message wm_User;  
end;
```

The name of the procedure and the type of the parameters are up to you, although there are several predefined record types for the various Windows messages. You could later generate this message, invoking the corresponding method, by writing:

```
PostMessage (Form1.Handle, wm_User, 0, 0);
```

This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions. You can also dispatch a message immediately by calling the `SendMessage` API or the `VCL Perform` method.

## Abstract Methods

The `abstract` keyword is used to declare methods that will be defined only in inherited classes of the current class. The `abstract` directive fully defines the method; it is not a forward declaration. If you try to provide a definition for the method, the compiler will complain. In Delphi, you can create instances of classes that have abstract methods. However, when you try to do so, Delphi's 32-bit compiler issues the warning message "Constructing instance of <class name> containing abstract methods." If you happen to call an abstract method at run time, Delphi will raise an exception, as demonstrated by the `AbstractAnimals` example (an extension of the `PolyAnimals` example), which uses the following class:

```
type  
    TAnimal = class  
    public  
        function Voice: string; virtual; abstract;
```

**Note** Most other OOP languages use a stricter approach: you cannot generally create instances of classes containing abstract methods.

You might wonder why you would want to use abstract methods. The reason lies in the use of polymorphism. If class `TAnimal` has the virtual method `Voice`, every inherited class *can* redefine it. If it has the abstract method `Voice`, every inherited class *must* redefine it.

In early versions of Delphi, if a method overriding an abstract method called `inherited`, the result was in an abstract method call. Since Delphi 6, the compiler has been enhanced to notice the presence of the abstract method and skip the `inherited` call. This means you can safely use `inherited` in every overridden method, unless you specifically want to disable executing some code of the base class.



## Type-Safe Down-Casting

The Delphi type-compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned earlier, the reverse is not possible. Now, suppose the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, it should be possible to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, you could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods you are calling actually exist.

To solve the problem, you can use techniques based on *run-time type information* (RTTI, for short). Essentially, because each object "knows" its type and its parent class, you can ask for this information with the `is` operator (or in some peculiar cases using the `InheritsFrom` method of `TObject`). The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then ...
```

The `is` expression evaluates as `True` only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from `TDog`. This means that if you test whether a `TDog` object is of type `TAnimal`, the test will succeed. In other words, this expression evaluates as `True` if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

Now that you know for sure that the animal is a dog, you can make a safe typecast (or type conversion). You can accomplish this direct cast by writing the following code:

```
var
  MyDog: TDog;
begin
  if MyAnimal is TDog then
  begin
    MyDog := TDog (MyAnimal);
    Text := MyDog.Eat;
  end;
```

This same operation can be accomplished directly by the second RTTI operator, `as`, which converts the object only if the requested class is compatible with the actual one. The parameters of the `as` operator are an object and a class type, and the result is an object converted to the new class type. You can write the following snippet:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

If you only want to call the `Eat` function, you might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second approach raises an exception if the object type is incompatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described at the end of this chapter).

To avoid this exception, use the `is` operator and, if it succeeds, make a plain typecast (in fact, there is no reason to use `is` and `as` in sequence, doing the type check twice):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both RTTI operators are very useful in Delphi because you often want to write generic code that can be used with several components of the same type or even of different types. When a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`); so, you often need to cast it back to the original

component type:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Sender is TButton then
        ...
end;
```

This is a common technique in Delphi, and I'll use it in examples throughout the book. The two RTTI operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism first. Only in special cases, where polymorphism alone cannot be applied, should you try using the RTTI operators to complement it. *Do not use RTTI instead of polymorphism.* This is bad programming practice, and it results in slower programs. RTTI has a negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As you have seen, virtual method calls require just a memory lookup, which is much faster.

**Note** Run-time type information (RTTI) involves more than the `is` and `as` operators. You can access detailed class and type information at run time, particularly for published properties, events, and methods. You'll find more on this topic in [Chapter 4](#).

Team LiB

◀ PREVIOUS

NEXT ▶

## Using Interfaces

When you define an abstract class to represent the base class of a hierarchy, you can come to a point at which the abstract class is so abstract that it only lists a series of virtual functions without providing any implementation. This kind of *purely abstract class* can also be defined using a specific technique: an interface. For this reason, we refer to these classes as *interfaces*.

Technically, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element with distinctive features:

- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to the way Delphi manages long strings and makes memory management almost automatic.
- A class can inherit from a single base class, but it can implement multiple interfaces.
- Just as all classes descend from `TObject`, all interfaces descend from `IInterface`, forming a totally separate hierarchy.

**Note** The base interface class was `IUnknown` until Delphi 5, but Delphi 6 introduced a new name for it—`IInterface`—to mark even more clearly the fact that this language feature is separate from Microsoft's COM (which uses `IUnknown` as its base interface). Delphi interfaces are available also in Kylix.

It is important to note that interfaces support a slightly different OOP model than classes. Interfaces provide a less restricted implementation of polymorphism. Object reference polymorphism is based around a specific branch of a hierarchy. Interface polymorphism works across an entire hierarchy. Certainly, interfaces favor encapsulation and provide a looser connection between classes than inheritance. Notice that the most recent OOP languages, from Java to C#, have the notion of interfaces.

Here is the syntax of the declaration of an interface (which, by convention, starts with the letter *I*):

```
type
  ICanFly = interface
    ['{EAD9C4B4-E1C5-4CF4-9FA0-3B812C880A21}']
    function Fly: string;
  end;
```

This interface has a GUID (Globally Unique Identifier)—a numeric ID following its declaration and based on Windows conventions. You can generate these identifiers by pressing Ctrl+Shift+G in the Delphi editor.

**Note** Although you can compile and use an interface without specifying a GUID for it, you'll generally want to generate the GUID because it is required to perform interface querying or dynamic *as* typecasts using that interface type. The whole point of interfaces is (usually) to take advantage of greatly extended type flexibility at run time; so, compared with class types, interfaces without GUIDs are not very useful.

Once you've declared an interface, you can define a class to implement it:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

The RTL already provides a few base classes to implement the basic behavior required by the `IInterface` interface. For internal objects, use the `TInterfacedObject` class I've used in this code.

You can implement interface methods with static methods (as in the previous code) or with virtual methods. You can override virtual methods in inherited classes by using the `override` directive. If you don't use virtual methods, you can still provide a new implementation in an inherited class by redeclaring the interface type in the inherited class,

rebinding the interface methods to new versions of the static methods. At first sight, using virtual methods to implement interfaces seems to allow for smoother coding in inherited classes, but both approaches are equally powerful and flexible. However, the use of virtual methods affects code size and memory.

**Note** The compiler has to generate stub routines to fix up the interface call entry points to the matching method of the implementing class, and adjust the *self* pointer. The interface method stubs for static methods have to adjust *self* and jump to the real method in the class. The interface method stubs for virtual methods are much more complicated, requiring about four times more code (20 to 30 bytes) in each stub than the static case. Also, adding more virtual methods to the implementing class just bloats the virtual method table (VMT) that much more in the implementing class and all its descendents. An interface already has its own VMT, and redeclaring an interface in descendents to rebind the interface to new methods in the descendent is just as polymorphic as using virtual methods, but much smaller in code size.

Now that you have defined an implementation of the interface, you can write some code to use an object of this class, through an interface-type variable:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

As soon as you assign an object to an interface-type variable, Delphi automatically checks to see whether the object implements that interface, using the *as* operator. You can explicitly express this operation as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

**Note** The compiler generates different code for the *as* operator when used with interfaces or with classes. With classes, the compiler introduces run-time checks to verify that the object is effectively "type-compatible" with the given class. With interfaces, the compiler sees at compile time that it can extract the necessary interface from the available class type, so it does. This operation is like a "compile-time *as*," not something that exists at run time.

Whether you use the direct assignment or the *as* statement, Delphi does one extra thing: It calls the `_AddRef` method of the object (defined by `IInterface`). The standard implementation of this method, like the one provided by `TInterfacedObject`, is to increase the object's reference count. At the same time, as soon as the `Flyer1` variable goes out of scope, Delphi calls the `_Release` method (again part of `IInterface`). The `TInterfacedObject`'s implementation of `_Release` decreases the reference count, checks whether the reference count is zero, and, if necessary, destroys the object. For this reason, the previous example doesn't include any code to free the object you've created.

In other words, in Delphi, objects referenced by interface variables are reference-counted, and they are automatically de-allocated when no interface variable refers to them any more.

**Warning** When using interface-based objects, you should generally access them only with object references or only with interface references. Mixing the two approaches breaks the reference counting scheme provided by Delphi and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables. If you want to be able to mix them, instead, disable the reference counting by writing your own base class instead of using `TInterfacedObject`.

## Working with Exceptions

Another key feature of Delphi is its support for *exceptions*. Exceptions make programs more robust by providing a standard way for notifying and handling errors and unexpected conditions. Exceptions make programs easier to write, read, and debug because they allow you to separate the error-handling code from your normal code, instead of intertwining the two. Enforcing a logical split between code and error handling and branching to the error handler automatically makes the actual logic cleaner and clearer. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

At run time, Delphi libraries raise exceptions when something goes wrong (in the run-time code, in a component, or in the operating system). From the point in the code at which it is raised, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, the VCL handles it, by displaying a standard error message and then trying to continue the program by handling the next system message or user request.

The whole mechanism is based on four keywords:

*try* Delimits the beginning of a protected block of code.

*except* Delimits the end of a protected block of code and introduces the exception-handling statements.

*finally* Specifies blocks of code that must always be executed, even when exceptions occur. This block is generally used to perform cleanup operations that should always be executed, such as closing files or database tables, freeing objects, and releasing memory and other resources acquired in the same program block.

*raise* Generates an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also raise exceptions in your own code when it discovers invalid or inconsistent data at run time. The `raise` keyword can also be used inside a handler to *re-raise* an exception; that is, to propagate it to the next handler.

**Tip** Exception handling is no substitute for proper control flow within a program. Keep using *if* statements to test user input and other foreseeable error conditions. You should use exceptions only for abnormal or unexpected situations.

## Program Flow and the *finally* Block

The power of exceptions in Delphi relates to the fact that they are "passed" from a routine or method to the caller, up to a global handler (if the program provides one, as Delphi applications generally do), instead of following the standard execution path of the program. So the real problem you might have is not how to stop an exception but how to execute code even if an exception is raised.

Consider this code, which performs some time-consuming operations and uses the hourglass cursor to show the user that it's doing something:

```
Screen.Cursor := crHourglass;  
// long algorithm...  
Screen.Cursor := crDefault;
```

In case there is an error in the algorithm (as I've included on purpose in the TryFinally example's event handlers), the program will break, but it won't reset the default cursor. This is what a `try/finally` block is for:

```
Screen.Cursor := crHourglass;  
try  
    // long algorithm...  
finally  
    Screen.Cursor := crDefault;
```

```
end;
```

When the program executes this function, it always resets the cursor, regardless of whether an exception (of any sort) occurs.

This code doesn't handle the exception; it merely makes the program robust in case an exception is raised. A `try` block can be followed by either an `except` or a `finally` statement, but not both of them at the same time; so, if you want to also handle the exception, the typical solution is to use two nested `try` blocks. You associate the internal block with a `finally` statement and the external block with an `except` statement, or vice versa as the situation requires. Here is the skeleton of the code for the third button in the TryFinally example:

```
Screen.Cursor := crHourglass;
try try
    // long algorithm...
finally
    Screen.Cursor := crDefault;
end;
except
    on E: EDivByZero do ...
end;
```

Every time you have some finalization code at the end of a method, you should place the code in a `finally` block. You should always, invariably, and continuously (how can I stress this more?) protect your code with `finally` statements, to avoid resource or memory leaks in case an exception is raised.

**Tip** Handling the exception is generally much less important than using `finally` blocks, because Delphi can survive most exceptions. Too many exception-handling blocks in your code probably indicate errors in the program flow and possibly a misunderstanding of the role of exceptions in the language. In the examples in the rest of the book, you'll see many `try/finally` blocks, a few `raise` statements, and almost no `try/except` blocks.

## Exception Classes

In the exception-handling statements shown earlier, you caught the `EDivByZero` exception, which is defined by Delphi's RTL. Other such exceptions refer to run-time problems (such as a wrong dynamic cast), Windows resource problems (such as out-of-memory errors), or component errors (such as a wrong index). Programmers can also define their own exceptions; you can create a new inherited class of the default exception class or one of its inherited classes:

```
type
    EArrayFull = class (Exception);
```

When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
    raise EArrayFull.Create ('Array full');
```

This `Create` constructor (inherited from the `Exception` class) has a string parameter to describe the exception to the user. You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

The code presented in the previous excerpts is part of a sample program called `Exception1`. Some of the routines have been slightly modified, as in the following `DivideTwicePlusOne` function:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
    try
        // error if B equals 0
        Result := A div B;
```

```

// do something else... skip if exception is raised
Result := Result div B;
Result := Result + 1;
except
on EDivByZero do
begin
Result := 0;
MessageDlg ('Divide by zero corrected.', mtError, [mbOK], 0);
end;
on E: Exception do
begin
Result := 0;
MessageDlg (E.Message, mtError, [mbOK], 0);
end;
end; // end except
end;

```

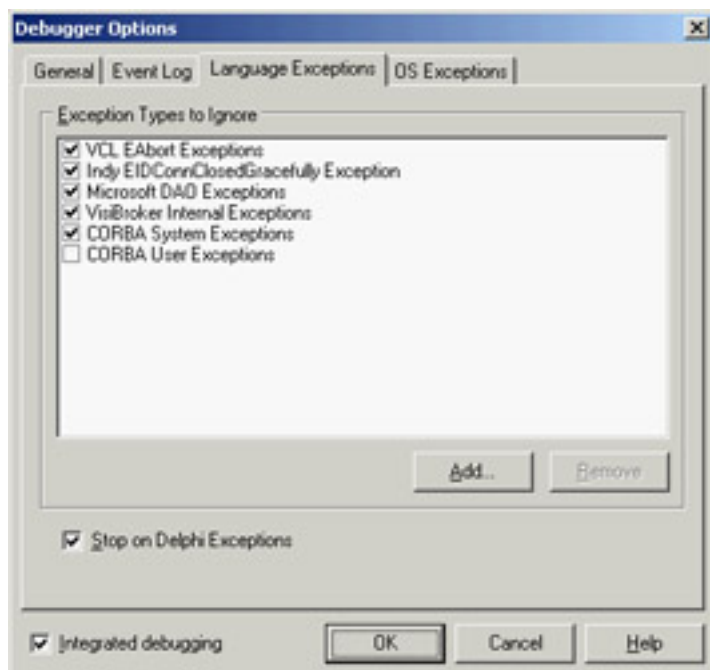
---

## Debugging and Exceptions

When you start a program from the Delphi environment (for example, by pressing the F9 key), you'll generally run it within the debugger. When an exception is encountered, the debugger will suspend the program by default. This result is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step by step. You can also use Delphi's Stack Trace feature to see the sequence of function and method calls that caused the program to raise an exception.

In the case of the Exception1 test program, however, this behavior will confuse a programmer not well aware of how Delphi's debugger works. Even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step by step through the code, you can see how it is handled.

If you just want to let the program run when the exception is properly handled, run the program from Windows Explorer, or temporarily disable the Stop on Delphi Exceptions options in the Language Exceptions page of the Debugger Options dialog box (activated by the Tools → Debugger Options command), shown in the Language Exceptions page of the Debugger Options dialog box shown here (as an alternative you can also disable the debugger):



In the `Exception1` code, there are two different exception handlers after the same `try` block. You can have any number of these handlers, which are evaluated in sequence.

Using a hierarchy of exceptions, a handler is also called for the inherited classes of the type it refers to, as any procedure will do. For this reason, you need to place the broader handlers (the handlers of the ancestor `Exception` classes) at the end. But keep in mind that using a handler for every exception, such as the previous one, is not usually a good choice. It is better to leave unknown exceptions to Delphi. The default exception handler in the VCL displays the error message of the exception class in a message box, and then resumes normal program operation. You can modify the normal exception handler with the `Application.OnException` event or the `OnException` event of the `ApplicationEvents` component, as demonstrated in the `ErrorLog` example in the [next section](#).

Another important element of the previous code is the use of the exception object in the handler (see `on E: Exception do`). The reference `E` of class `Exception` refers to the exception object passed by the `raise` statement. When you work with exceptions, remember this rule: You raise an exception by creating an object and handle it by indicating its type. This has an important benefit, because as you have seen, when you handle a type of exception, you are really handling exceptions of the type you specify as well as any descendant type.

**Team LiB**

◀ PREVIOUS

NEXT ▶



## Logging Errors

Most of the time, you don't know which operation will raise an exception, and you cannot (and should not) wrap each and every piece of code in a `try/except` block. The general approach is to let Delphi handle all the exceptions and eventually pass them to you, by handling the `OnException` event of the global `Application` object. You can do so rather easily with the `ApplicationEvents` component.

In the `ErrorLog` example, I've added to the main form an instance of the `ApplicationEvents` component and written a handler for its `OnException` event:

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);  
var  
    Filename: string;  
    LogFile: TextFile;  
begin  
    // prepares log file  
    Filename := ChangeFileExt (Application.Exename, '.log');  
    AssignFile (LogFile, Filename);  
    if FileExists (FileName) then  
        Append (LogFile) // open existing file  
    else  
        Rewrite (LogFile); // create a new one  
    try  
        // write to the file and show error  
        Writeln (LogFile, DateTimeToStr (Now) + ':' + E.Message);  
        if not CheckBoxSilent.Checked then  
            Application.ShowException (E);  
    finally  
        // close the file  
        CloseFile (LogFile);  
    end;  
end;
```

**Note** The `ErrorLog` example uses the text file support provided by the traditional Turbo Pascal `TextFile` data type. You can assign a text file variable to an actual file and then read or write it. You can find more on `TextFile` operations in [Chapter 12](#) of the e-book *Essential Pascal*, covered in [Appendix C](#).

In the global exceptions handler, you can write to the log, for example, the date and time of the event, and also decide whether to show the exception as Delphi usually does (executing the `ShowException` method of the `TApplication` class). By default, Delphi executes `ShowException` only if no `OnException` handler is installed. In [Figure 2.8](#), you can see the `ErrorLog` program running and a sample exceptions log open in `ConTEXT` (a nice programmer's editor built with Delphi and available at [www.fixedsys.com/context](http://www.fixedsys.com/context)).

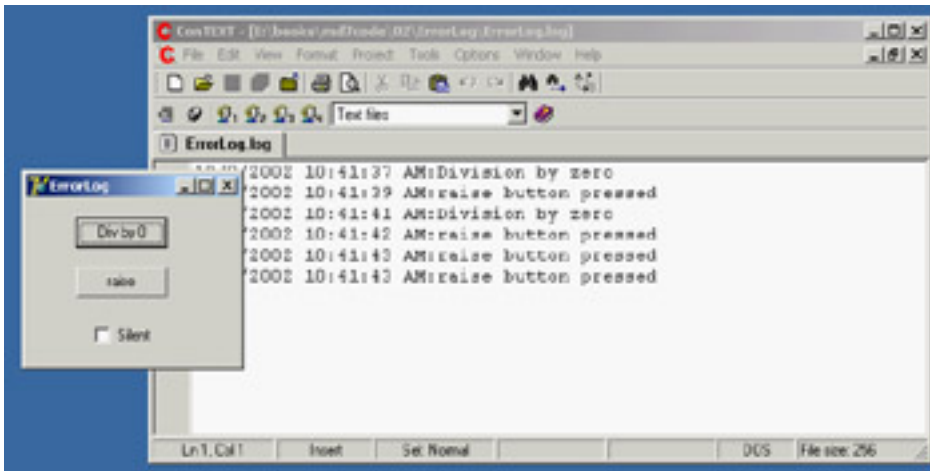


Figure 2.8: The ErrorLog example and the log it produces

## Class References

The last language feature I'll discuss in this chapter is the use of *class references*, which implies the idea of manipulating classes themselves within your code. The first point to keep in mind is that a class reference isn't an object; it is a reference to a class type. A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code will make this concept a little clearer.

Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AnObject: TMyClass;
  AClassRef: TMyClassRef;
begin
  AnObject := TMyClass.Create;
  AClassRef := TMyClass;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. There are not many such expressions, but the few cases are interesting, like the creation of an object. You can rewrite the last line of the previous code as follows:

```
AnObject := AClassRef.Create;
```

This time, you apply the `Create` constructor to the class reference instead of to an actual class; you use a class reference to create an object of that class.

Class reference types wouldn't be as useful if they didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef`, you can then assign to it that specific class and any inherited class. So if `TMyNewClass` is an inherited class of `TMyClass` my class, you can also write

```
AClassRef := TMyNewClass;
```

Delphi declares a lot of class references in the run-time library and the VCL, such as the following:

```
TClass = class of TObject;
TComponentClass = class of TComponent;
TFormClass = class of TForm;
```

In particular, the `TClass` class reference type can be used to store a reference to any class you write in Delphi, because every class is ultimately derived from `TObject`. The `TFormClass` reference is used in the source code of most Delphi projects. The `CreateForm` method of the `Application` object requires as a parameter the class of the form to create:

```
Application.CreateForm(TForm1, Form1);
```

The first parameter is a class reference; the second is a variable that stores a reference to the created object instance.

Finally, when you have a class reference, you can apply to it the class methods of the related class. Considering that each class inherits from `TObject`, you can apply to each class reference some of the methods of `TObject`, as you'll see in [Chapter 3](#).

## Creating Components Using Class References

What is the *practical* use of class references in Delphi? Being able to manipulate a data type at run time is a

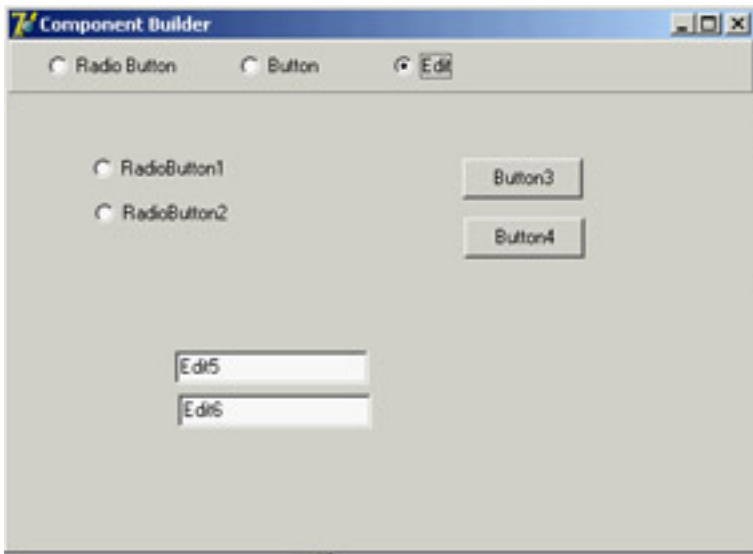
fundamental element of the Delphi environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what Delphi does for you behind the scenes.) In other words, class references give you polymorphism for object construction.

To give you a better idea of how class references work, I've built an example named ClassRef. The form displayed by this example has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, push buttons, and edit boxes.

To make this program run properly, you need to change the names of the three components. The form must also have a class reference field, declared as `ClassRef: TControlClass`. It stores a new data type every time the user clicks one of the three radio buttons, with assignments like `ClassRef := TEdit`. The interesting part of the code is executed when the user clicks the form. Again, I've chosen the `OnMouseDown` event of the form to hold the position of the mouse click:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
var
    NewCtrl: TControl;
    MyName: String;
begin
    // create the control
    NewCtrl := ClassRef.Create (Self);
    // hide it temporarily, to avoid flickering
    NewCtrl.Visible := False;
    // set parent and position
    NewCtrl.Parent := Self;
    NewCtrl.Left := X;
    NewCtrl.Top := Y;
    // compute the unique name (and caption)
    Inc (Counter);
    MyName := ClassRef.ClassName + IntToStr (Counter);
    Delete (MyName, 1, 1);
    NewCtrl.Name := MyName;
    // now show it
    NewCtrl.Visible := True;
end;
```

The first line of the code for this method is the key. It creates a new object of the class data type stored in the `ClassRef` field. You accomplish this by applying the `Create` constructor to the class reference. Now you can set the value of the `Parent` property, set the position of the new component, give the component a name (which is automatically used also as the value of `Caption` or `Text`), and make it visible. You can see an example of the output of this program in [Figure 2.9](#).



**Figure 2.9:** An example of the output of the ClassRef example

**Note** For polymorphic construction to work, the base class type of the class reference must have a virtual constructor. If you use a virtual constructor (as in the example), the constructor call applied to the class reference will call the constructor of the type that the class reference variable *currently refers to*. But without a virtual constructor, your code will call the constructor of *fixed class type* indicated in the class reference declaration. Virtual constructors are required for polymorphic construction in the same way that virtual methods are required for polymorphism.

## What's Next?

In this chapter, we have discussed the foundations of object-oriented programming (OOP) in Delphi. We have considered the definition of classes and the use of methods, encapsulation, and memory management, but also some more advanced concepts such as properties and the dynamic creation of components. Then we moved to inheritance, virtual and abstract methods, polymorphism, safe typecasting, interfaces, exceptions, and class references.

This is certainly a lot of information if you are a newcomer. But if you are fluent in another OOP language or if you've already used past versions of Delphi, you should be able to apply the topics covered in this chapter to your programming.

Understanding the secrets of Delphi's language and library is vital for becoming an expert Delphi programmer. These topics form the foundation of working with the VCL and CLX class libraries; after exploring them in the next two chapters, we'll *finally* go on to explore the development of real applications using all the various components provided by Delphi.

In the meantime, [Chapter 3](#) will give you an overview of the Delphi run-time library (mainly a collection of functions with little OOP involved). The RTL is a collection of assorted routines for performing basic tasks with Delphi. [Chapter 4](#) will give you more information about the language, discussing features related to the structure of the Delphi class library, such as the effect of the `published` keyword and the role of events. That chapter, as a whole, will discuss the overall architecture of the component library.

## Chapter 3: The Run-Time Library

### Overview

The Delphi programming language favors an object-oriented approach, tied with a visual development style. This is where Delphi shines, and we will cover component-based and visual development in this book; however, I want to underline the fact that many of Delphi's ready-to-use features come from its run-time library (RTL). This is a large collection of functions you can use to perform simple tasks, as well as some complex ones, within your Pascal code. (I use "Pascal" here, because the run-time library primarily contains procedures and functions written with the traditional language constructs and not the OOP extensions added to the language by Borland.)

There is a second reason to devote this chapter of the book to the run-time library: Delphi 6 saw a large number of enhancements to this area, and a few more are provided in Delphi 7. New groups of functions are available, functions have been moved to new units, and other elements have changed, creating a few incompatibilities with older code from which you might be porting your projects. So, even if you've used past versions of Delphi and feel confident with the RTL, you should still read at least portions of this chapter.

## The Units of the RTL

In the most recent versions of Delphi, the RTL has a new structure and several new units. Borland added new units because it also added many new functions. In most cases, you'll find the existing functions in the units where they used to be, but the new functions appear in specific units. For example, new functions related to dates are now in the DateUtils unit, but existing date functions have not been moved out of SysUtils in order to avoid incompatibilities with existing code.

The exception to this rule relates to some of the variant support functions, which were moved out of the System unit to avoid unwanted linkage of specific Windows libraries, even in programs that didn't use those features. These variant functions are now part of the Variants unit, described later in the chapter.

**Warning** Some of your Delphi 4 and Delphi 5 code might need to use the Variants unit to recompile. Delphi is smart enough to acknowledge this requirement and auto-include the Variants unit in projects that use the *Variant* type, issuing only a warning.

A little fine-tuning has also been applied to reduce the minimum size of an executable file, which is at times enlarged by the unwanted inclusion of global variables or initialization code.

---

### Executable Size under the Microscope

While touching up the RTL, Borland engineers have been able to trim a little "fat" out of each and every Delphi application. Reducing the minimum program size by a few KB seems odd, given all the bloated applications these days, but it is a good service to developers. In some cases, even a few KB (multiplied by many applications) can reduce size and eventually download time.

As a simple test, I've built the MiniSize program, which is not an attempt to build the smallest possible program, but rather an attempt to build a very small program that does something interesting: It reports the size of its own executable file. All of the example code is as follows:

```
program MiniSize;

uses
  Windows;

{$R *.RES}

var
  nSize: Integer;
  hFile: THandle;
  strSize: String;

begin
  // open the current file and read the size
  hFile := CreateFile (PChar (ParamStr (0)),
    0, FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
  nSize := GetFileSize (hFile, nil);
  CloseHandle (hFile);

  // copy the size to a string and show it
  SetLength (strSize, 20);
  Str (nSize, strSize);
  MessageBox (0, PChar (strSize),
    'Mini Program', MB_OK);
end.
```

The program opens its own executable file, after retrieving its name from the first command-line parameter



(ParamStr (0)), extracts the size, converts it into a string using the simple Str function, and shows the result in a message. The program does not have top-level windows. Moreover, I use the Str function for the integer-to-string conversion to avoid including SysUtils, which defines all of the more complex formatting routines and would impose a little extra overhead.

If you compile this program with Delphi 5, you obtain an executable size of 18,432 bytes. Delphi 6 reduces this size to only 15,360 bytes, trimming about 3 KB. In Delphi 7, the size is only slightly greater, at 15,872 bytes. By replacing the long string with a short string and modifying the code a little, you can trim the program further, to less than 10 KB. (You'll end up removing the string support routines and also the memory allocator, something possible only in programs using exclusively low-level calls.) You can find both versions in the source code of the example file.

Notice that decisions of this type always imply a few trade-offs. In eliminating the overhead of variants from Delphi applications that don't use them, for example, Borland added a little extra burden to applications that do. The real advantage of this operation, though, is in the reduced memory footprint of Delphi applications that do not use variants, as a result of not having to bring in several megabytes of the Ole2 system libraries.

What is really important, in my opinion, is the size of full-blown Delphi applications based on run-time packages. A simple test with a do-nothing program, the MiniPack example, shows an executable size of 17,408 bytes.

---

In the following sections you'll find a list of the RTL units in Delphi, including all the units available (with the complete source code) in the Source\Rtl\Sys subfolder of the Delphi directory and some of those available in the subfolder Source\Rtl\Common. This second directory hosts the source code of units that make up the new RTL package, which comprises both the function-based library and the core classes, discussed at the end of this chapter and in [Chapter 4](#) ("Core Library Classes").

**Note** The original VCL package present up to version 5 of Delphi has been split into the VCL and RTL packages, so that nonvisual applications using run-time packages don't have the overhead of also deploying visual portions of the VCL. This change also helps with Linux compatibility, because the new package is shared between the VCL and CLX libraries. In addition, notice that the package names in Delphi 6 and 7 don't include the version number; when they are compiled, though, the BPL does have the version in its filename, as discussed in more detail in [Chapter 10](#) ("Libraries and Packages").

I'll give a short overview of the role of each unit and an overview of the groups of functions included. I'll also devote more space to the newer units. I won't provide a detailed list of the functions included, because the online help includes similar reference material. However, I've tried to pick a few interesting or little-known functions, and I will discuss them shortly.

## The System and SysInit Units

System is the core unit of the RTL and is automatically included in any compilation (through an automatic and implicit uses statement referring to it). If you try adding the unit to the uses statement of a program, you'll get the following compile-time error:

```
[Error] Identifier redeclared: System
```

The System unit includes, among other things:

- The TObject class, which is the base class of any class defined in the Object Pascal language, including all the classes of the VCL. (This class is discussed later in this chapter.)
- The IInterface, IInvokable, IUnknown, and IDispatch interfaces, as well as the simple implementation class TInterfacedObject. IInterface was added in Delphi 6 to underscore the point that the interface type in the Delphi language definition is in no way dependent on the Windows operating system. IInvokable was added in Delphi 6 to support SOAP-based invocation.

- Some variant support code, including the variant type constants, the `TVarData` record type and the new `TVariantManager` type, a large number of variant conversion routines, and variant record and dynamic array support. This area has seen a lot of changes compared to Delphi 5. The basic information on variants is provided in Chapter 10 of *Essential Pascal* (for more information see [Appendix C](#), "Free Companion Books on Delphi Programming").
- Many base data types, including pointer and array types and the `TDateTime` type described in [Chapter 2](#), "The Delphi Programming Language."
- Memory allocation routines, such as `GetMem` and `FreeMem`, and the actual memory manager, defined by the `TMemoryManager` record and accessed by the `GetMemoryManager` and `SetMemoryManager` functions. For information, the `GetHeapStatus` function returns a `THeapStatus` data structure. Two global variables (`AllocMemCount` and `AllocMemSize`) hold the number and total size of allocated memory blocks. There is more on memory and the use of these functions in [Chapter 8](#), "The Architecture of Delphi Applications" (see in particular the `ObjLeft` example).
- Package and module support code, including the `PackageInfo` pointer type, the `GetPackage-InfoTable` global function, and the `EnumModules` procedure (package internals are discussed in [Chapter 12](#)).
- A rather long list of global variables, including the Windows application instance `MainInstance`; `IsLibrary`, indicating whether the executable file is a library or a stand-alone program; `IsConsole`, indicating console applications; `IsMultiThread`, indicating whether there are secondary threads; and the command-line string `CmdLine`. (The unit also includes `ParamCount` and `ParamStr` for easy access to command-line parameters.) Some of these variables are specific to the Windows platform, some are also available on Linux, and others are specific to Linux.
- Thread-support code, with the `BeginThread` and `EndThread` functions; file support records and file-related routines; wide string and OLE string conversion routines; and many other low-level and system routines (including a number of automatic conversion functions).

The companion unit of System, called `SysInit`, includes the system initialization code, with functions you'll seldom use directly. This is another unit that is always implicitly included, because it is used by the System unit.

## Recent Changes in the System Unit

I've already described some interesting features of the System unit in the previous section's list. Most of the changes in recent Delphi versions relate to making the core RTL more cross-platform portable, replacing Windows-specific features with generic implementations now shared by Delphi and Kylix. Along this line, there are new names for interface types, totally revised support for variants, new pointer types, dynamic array support, and functions to customize the management of exception objects.

**Note** If you read the source code of `System.pas`, you'll notice some heavy use of conditional compilation, with many instances of `{ $IFDEF LINUX }` and `{ $IFDEF MSWINDOWS }` used to discriminate between the two operating systems. Notice that for Windows, Borland uses the `MSWINDOWS` define to indicate the entire platform, because `WINDOWS` was used in 16-bit versions of the OS (and contrasts with the symbol `WIN32`).

For example, an addition for compatibility between Linux and Windows relates to line breaks in text files. The `DefaultTextLineBreakStyle` variable affects the behavior of routines that read and write files, including most text-streaming routines. The possible values for this global variable are `tlbsLF` (the default in Kylix) and `tlbsCRLF` (the default in Delphi). The line-break style can also be set on a file-by-file basis with `SetTextLineBreakStyle` function.

Similarly, the global `sLineBreak` string constant has the value `#13#10` in the Windows version of the IDE and the value `#10` in the Linux version. Another change is that the System unit now includes the `TFileRec` and `TTextRec` structures, which were defined in the `SysUtils` unit in earlier versions of Delphi.

## The SysUtils and SysConst Units

The SysConst unit defines a few constant strings used by the other RTL units for displaying messages. These strings are declared with the `resourcestring` keyword and saved in the program resources. Like other resources, they can be translated by means of the Integrated Translation Manager or the External Translation Manager.

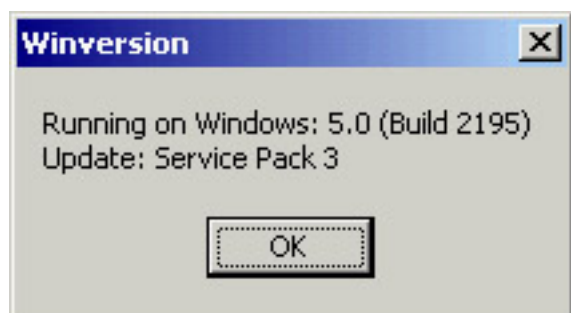
The SysUtils unit is a collection of system utility functions of various types. Unlike other RTL units, it is largely an operating system–dependent unit. The SysUtils unit has no specific focus, but it includes a bit of everything, from string management to locale and multibyte-characters support, from the `Exception` class and several other derived exception classes to a plethora of string-formatting constants and routines. In particular, later in this chapter we'll focus on some of the unit's file management routines.

Some features of SysUtils are used every day by every programmer, such as the `IntToStr` or `Format` string-formatting functions; other features are lesser known, such as the Windows version information global variables. These indicate the Windows platform (Window 9x or NT/2000/XP), the operating system version and build number, and the service pack installed. They can be used as in the following code, extracted from the `WinVersion` example:

```
case Win32Platform of
  VER_PLATFORM_WIN32_WINDOWS:  ShowMessage ('Windows 9x');
  VER_PLATFORM_WIN32_NT:       ShowMessage ('Windows NT');
end;
```

```
ShowMessage ('Running on Windows: ' + IntToStr (Win32MajorVersion) + '.' +
  IntToStr (Win32MinorVersion) + ' (Build ' + IntToStr (Win32BuildNumber) +
  ') ' + #10#13 + 'Update: ' + Win32CSDVersion);
```

The second code fragment produces a message like the one in the following graphic (of course, on the operating-system version you have installed):



Another little-known feature of this unit is the `TMultiReadExclusiveWriteSynchronizer` class—probably the VCL class with the longest name. Borland has defined an alias name for the class, which is much shorter: `TMREWSync` (the two classes are identical). This class supports multithreading: It allows you to work with resources that can be used by multiple threads at the same time for reading (multiread) but must be used by a single thread when writing (exclusive-write). This means writing cannot begin until all the reading threads have terminated.

The implementation of the `TMultiReadExclusiveWriteSynchronizer` class has been updated in Delphi 7, but similar improvements are available in an informal patch released after Delphi 6 update 2. The new version of the class is more optimized and less subject to deadlocks, which are often a problem with synchronization code.

**Note** The multiread synchronizer is unique in that it supports recursive locks and promotion of read locks to write locks. The main purpose of the class is to allow multiple threads easy, fast access to read from a shared resource, but still allow one thread to gain exclusive control of the resource for relatively infrequent updates. Delphi includes other synchronization classes, declared in the `SyncObjs` unit (available under `Source/Rtl/Common`) and closely mapped to operating-system synchronization objects (such as events and critical sections in Windows).

## Recent SysUtils Functions

Over the last couple of versions, Delphi has added some new functions within the SysUtils unit. One of these areas relates to Boolean-to-string conversion. The `BoolToStr` function generally returns '-1' and '0' for true and false values. If the second optional parameter is specified, the function returns the first string in the `TrueBoolStrs` and `FalseBoolStrs` arrays (by default 'TRUE' and 'FALSE'):

```
BoolToStr (True) // returns '-1'  
BoolToStr (False, True) // returns 'FALSE' by default
```

The reverse function is `StrToBool`, which can convert a string containing either one of the values of the two Boolean arrays mentioned or a numeric value. In the latter case, the result will be true unless the numeric value is zero. You can see a simple demo of the use of the Boolean conversion functions in the `StrDemo` example, later in this chapter.

Other functions recently added to SysUtils relate to floating-point conversions to currency and date time types: You can use `FloatToCurr` and `FloatToDateTime` to avoid an explicit typecast. The `TryStrToFloat` and `TryStrToCurr` functions try to convert a string into a floating-point or currency value and will return False in case of error instead of generating an exception (as the classic `StrToFloat` and `StrToCurr` functions do).

The `AnsiDequotedStr` function, which removes quotes from a string, matches the `AnsiQuoteStr` function added in Delphi 5. Speaking of strings, as of Delphi 6 there is much-improved support for wide strings, with a series of routines including `WideUpperCase`, `WideLowerCase`, `WideCompareStr`, `WideSameStr`, `WideCompareText`, `WideSameText`, and `WideFormat`. All of these functions work like their `AnsiString` counterparts.

Three functions (`TryStrToDate`, `TryEncodeDate`, and `TryEncodeTime`) try to convert a string to a date or to encode a date or time, without raising an exception, similar to the `Try` functions previously mentioned. In addition, the `DecodeDateFully` function returns more detailed information, such as the day of the week, and the `CurrentYear` function returns the year of today's date.

A portable, friendly, overloaded version of the `GetEnvironmentVariable` function uses string parameters instead of `PChar` parameters and is definitely easier to use than the original version based on `PChar` pointers:

```
function GetEnvironmentVariable(Name: string): string;
```

Other functions relate to interface support. Two overloaded versions of the little-known `Support` function allow you to check whether an object or a class supports a given interface. The function corresponds to the behavior of the `is` operator for classes and is mapped to the `QueryInterface` method. Here's an example:

```
var  
    W1: IWalker;  
    J1: IJumper;  
begin  
    W1 := TAthlete.Create;  
    // more code...  
    if Supports (w1, IJumper) then  
        begin  
            J1 := W1 as IJumper;  
            Log (J1.Walk);  
        end;
```

SysUtils also includes an `IsEqualGUID` function and two functions for converting strings to GUIDs and vice versa. The function `CreateGUID` has been moved to SysUtils, as well, to make it available on Linux (with a custom implementation, of course).

Finally, more features were added in recent versions to improve cross-platform support. The `AdjustLineBreaks` function can now do different types of *adjustments* to carriage-return and line-feed sequences, and new global variables for text files have been introduced in the System unit, as described earlier. The `FileCreate` function has an overloaded version in which you can specify file-access rights *the Unix way*. The `ExpandFileName` function can locate files (on case-sensitive file systems) even when their cases don't exactly correspond. The functions related to

path delimiters (backslash or slash) have been made more generic than in earlier versions of Delphi and renamed accordingly. (For example, the old `IncludeTrailingBackslash` function is now better known as `IncludingTrailingPathDelimiter`.)

Speaking of files, Delphi 7 adds to the `SysUtils` unit the `GetFileVersion` function, which reads the version number from the version information optionally added to a Windows executable file (which is why this function won't work on Linux).

## Delphi 7 Extended String Formatting Routines

Most of Delphi's string formatting routines (see [Appendix C](#), "Free Companion Books on Delphi," for instructions on how to get an e-book introducing some of them) use global variables to determine decimal and thousand separators, date/time formats, and so on. The values of these variables are first read from the system (Windows regional settings) when a program starts, and you are free to override each of them. However, if the user modifies the Regional Settings in Control Panel while your program is running, the program will respond to the broadcast message by updating the variables, probably losing your hard-coded changes.

If you need different output formats in different places within a program, you can take advantage of the new set of overloaded string formatting routines; they take an extra parameter of type `TFormatSettings`, including all the relevant settings. For example, there are now two versions of `Format`:

```
function Format(const Format: string;
  const Args: array of const): string; overload;
function Format(const Format: string; const Args: array of const;
  const FormatSettings: TFormatSettings): string; overload;
```

Tens of functions have this new extra parameter, which is then used instead of the global settings. However, you can initialize it with the default settings of the computer on which your program is running by calling the new `GetLocaleFormatSettings` function (available only on Windows, not Linux).

## The Math Unit

The Math unit hosts a collection of mathematical functions: about 40 trigonometric functions, logarithmic and exponential functions, rounding functions, polynomial evaluations, almost 30 statistical functions, and a dozen financial functions.

Describing all the functions of this unit would be rather tedious, although some readers are probably very interested in Delphi's mathematical capabilities. For this reason, I've decided to focus on math functions introduced in the latest versions of Delphi (particularly Delphi 6) and then cover one specific topic that often confuses Delphi programmers—rounding.

### New Math Functions

Recent versions add to the Math unit quite a number of features. They include support for infinite constants (`Infinity` and `NegInfinity`) and related comparison functions (`IsInfinite` and `IsNaN`), along with new trigonometric functions for cosecants and cotangents, and new angle-conversion functions.

A handy feature is the availability of an overloaded `IfThen` function, which returns one of two possible values depending on a Boolean expression. (A similar function is available also for strings.) You can use it, for example, to compute the minimum of two values:

```
nMin := IfThen (nA < nB, na, nB);
```

**Note** The `IfThen` function is similar to the `? :` operator of the C/C++ language. I find it handy because you can replace a complete `if/then/else` statement with a much shorter expression, writing less code and often declaring fewer temporary variables.

You can use `RandomRange` and `RandomFrom` instead of the traditional `Random` function to gain more control over the random values produced by the RTL. The first function returns a number within two extremes you specify, and the second selects a random value from an array of possible numbers you pass to it as a parameter.

The `InRange` Boolean function can be used to check whether a number is within two other values. The `EnsureRange` function, on the other hand, forces the value to be within the specified range. The return value is the number itself or the lower limit or upper limit, in the event the number is out of range. Here is an example:

```
// do something only if value is within min and max
if InRange (value, min, max) then
    ...

// make sure the value is between min and max
value := EnsureRange (value, min, max);
...
```

Another set of useful functions relates to comparisons. Floating-point numbers are fundamentally inexact; a floating-point number is an approximation of a theoretical real value. When you do mathematical operations on floating-point numbers, the inexactness of the original values accumulates in the results. Multiplying and dividing by the same number might not return exactly the original number but one that is very close to it. The `SameValue` function allows you to check whether two values are close enough in value to be considered equal. You can specify how close the two numbers should be or let Delphi compute a reasonable error range for the representation you are using. (This is why the function is overloaded.) Similarly, the `IsZero` function compares a number to zero, with the same "fuzzy logic."

The `CompareValue` function uses the same rule for floating-point numbers but is available also for integers; it returns one of the three constants `LessThanValue`, `EqualsValue`, and `GreaterThanValue` (corresponding to  $-1$ ,  $0$ , and  $1$ ). Similarly, the new `Sign` function returns  $-1$ ,  $0$ , or  $1$  to indicate a negative value, zero, or a positive value.

The `DivMod` function is equivalent to both the `div` and `mod` operations, returning the result of the integer division and the remainder (or modulus) at once. The `RoundTo` function allows you to specify the rounding digit—allowing, for example, rounding to the nearest thousand or to two decimals:

```
RoundTo (123827, 3); // result is 124,000
RoundTo (12.3827, -2); // result is 12.38
```

**Warning** Notice that the `RoundTo` function uses a positive number to indicate the power of 10 to round to (for example,  $2$  for hundreds) or a negative number for the number of decimal places. This is exactly the opposite of the `Round` function used by spreadsheets such as Excel.

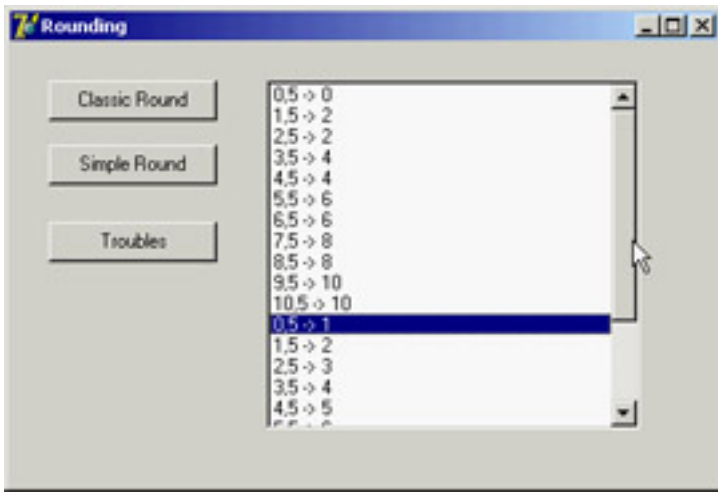
There have also been some changes to the standard rounding operations provided by the `Round` function: You can now control how the FPU (the floating-point unit of the CPU) does the rounding by calling the `SetRoundMode` function. Other functions control the FPU precision mode and its exceptions.

## Rounding Headaches

Delphi's classic `Round` function and the newer `RoundTo` functions are mapped to the CPU/ FPU rounding algorithms. By default, Intel CPUs use *banker's rounding*, which is also the type of rounding typically found in spreadsheet applications.

Banker's rounding is based on the assumption that when you're rounding numbers that lie exactly between two values (the  $.5$  numbers), rounding them all up or all down will statistically increase or reduce the total amount (of money, in general). For this reason, the rule of banker's rounding indicates that  $.5$  numbers should be rounded down or up depending on whether the number (without decimals) is odd or even. This way, the rounding will be balanced, at least statistically. You can see an example of the output of banker's rounding in [Figure 3.1](#), which shows the output of the Rounding example I've built to demonstrate different types of rounding.





**Figure 3.1:** The Rounding example, demon-strated banker's rounding and arithmetic rounding

The program also uses another type of rounding provided by the Math unit in the `SimpleRoundTo` function, which uses *asymmetric arithmetic rounding*. In this case, all .5 numbers are rounded to the upper value. However, as highlighted in the Rounding example, the function doesn't work as expected when rounding to a decimal digit (that is, when you pass a negative second parameter). In this case, due to the representation errors of floating-point numbers, the rounding trims the values; for example, it turns 1.15 into 1.1 instead of the expected 1.2. The solution is to multiply the value by ten before rounding, round it to zero decimal digits, and then divide it, as demonstrated in the sample program:

```
SimpleRoundTo (d * 10, 0) / 10)
```

## The ConvUtils and StdConvs Units

The ConvUtils unit contains the core of the conversion engine introduced in Delphi 6. It uses the conversion constants defined by a second unit, StdConvs. I'll cover these two units later in this chapter and show how to extend them with new measurement units.

**Note** Delphi 7 makes only a single improvement to the conversion unit: It adds support for stones (the British unit of measurement that is equivalent to 14 pounds). In any case, if you have to deal with units of measurements in your code, you'll appreciate the features available in this engine.

## The DateUtils Unit

The DateUtils unit is a collection of date- and time-related functions. It includes functions for picking values from a `TDateTime` variable or counting values from a given interval, such as

```
// pick value
function DayOf(const AValue: TDateTime): Word;
function HourOf(const AValue: TDateTime): Word;
// value in range
function WeekOfYear(const AValue: TDateTime): Integer;
function HourOfWeek(const AValue: TDateTime): Integer;
function SecondOfHour(const AValue: TDateTime): Integer;
```

Some of these functions are quite odd, such as `MilliSecondOfMonth` or `SecondOfWeek`, but Borland developers have decided to provide a complete set of functions, no matter how impractical they sound. (I actually used some of these functions in [Chapter 2](#), to build the `TDate` class.)

There are functions for computing the initial or final value of a given time interval (day, week, month, year) including the current date, and for range checking and querying; for example:

```
function DaysBetween(const ANow, AThen: TDateTime): Integer;
function WithinPastDays(const ANow, AThen: TDateTime;
```

```
const ADays: Integer): Boolean;
```

Other functions cover incrementing and decrementing by each of the possible time intervals, encoding and "recoding" (replacing one element of the `TDateTime` value, such as the day, with a new one), and doing "fuzzy" comparisons (approximate comparisons where a difference of a millisecond will still make two dates equal). Overall, `DateUtils` is quite interesting and not terribly difficult to use.

## The StrUtils Unit

The `StrUtils` unit was introduced in Delphi 6 with some new string-related functions. One of the key features of this unit is the availability of many string comparison functions. There are functions based on a *soundex* algorithm (`AnsiResembleText`), and others that provide lookup in arrays of strings (`AnsiMatchText` and `AnsiIndexText`), substring location, and text replacement (including `AnsiContainsText` and `AnsiReplaceText`).

**Note** *Soundex* is an algorithm that compares names based on how they sound rather than how they are spelled. The algorithm computes a number for each word sound, so that by comparing two such numbers you can determine whether two names sound similar. The system was first applied in 1880 by the U.S. Bureau of the Census; it was patented in 1918 and is now in the public domain. The soundex code is an indexing system that translates a name into a four-character code consisting of one letter and three numbers. More information is available at [www.nara.gov/genealogy/coding.html](http://www.nara.gov/genealogy/coding.html).

Beside comparisons, other functions provide a two-way test (the nice `IfThen` function, similar to the one we've already seen for numbers), duplicate and reverse strings, and replace substrings. Most of these string functions were added as a convenience to Visual Basic programmers migrating to Delphi.

I've used some of these functions in the `StrDemo` example, which uses also some of the Boolean-to-string conversions defined within the `SysUtils` unit. The program is little more than a test for a few of these functions. For example, it uses the soundex comparison between the strings entered in two edit boxes, converting the resulting Boolean into a string and showing it:

```
ShowMessage (BoolToStr (AnsiResemblesText  
    (EditResemble1.Text, EditResemble2.Text), True));
```

The program also showcases the `AnsiMatchText` and `AnsiIndexText` functions, after filling a dynamic array of strings (called `strArray`) with the values of the strings inside a list box. I could have used the simpler `IndexOf` method of the `TStrings` class, but doing so would have defeated the purpose of the example. The two list comparisons are as follows:

```
procedure TForm1.ButtonMatchesClick(Sender: TObject);  
begin  
    ShowMessage (BoolToStr (AnsiMatchText(EditMatch.Text, strArray), True));  
end;
```

```
procedure TForm1.ButtonIndexClick(Sender: TObject);  
var  
    nMatch: Integer;  
begin  
    nMatch := AnsiIndexText(EditMatch.Text, strArray);  
    ShowMessage (IfThen (nMatch >= 0, 'Matches the string number ' +  
        IntToStr (nMatch), 'No match'));  
end;
```

Notice the use of the `IfThen` function in the last few lines of code; it has two alternative output strings, depending on the result of the initial test (`nMatch >= 0`).

Three more buttons do simple calls to three other new functions, with the following lines of code (one for each):

```
// duplicate (3 times) a string
```



```
ShowMessage (DupeString (EditSample.Text, 3));
// reverse the string
ShowMessage (ReverseString (EditSample.Text));
// choose a random string
ShowMessage (RandomFrom (strArray));
```

## From Pos to PosEx

Delphi 7 adds a little to the StrUtils unit. The new `PosEx` function will be handy to many developers and is worth a brief mention. When searching for multiple occurrences of a string within another one, a classic Delphi solution was to use the `Pos` function and repeat the search over the remaining portion of the string. For example, you could count the occurrences of a string inside another string with code like this:

```
function CountSubstr (text, sub: string): Integer;
var
    nPos: Integer;
begin
    Result := 0;
    nPos := Pos (sub, text);
    while nPos > 0 do
        begin
            Inc (Result);
            text := Copy (text, nPos + Length (sub), MaxInt);
            nPos := Pos (sub, text);
        end;
    end;
```

The new `PosEx` function allows you to specify the starting position of the search within a string, so you don't need to alter the original string (wasting quite some time). Thus the previous code can be simplified in the following way:

```
function CountSubstrEx (text, sub: string): Integer;
var
    nPos: Integer;
begin
    Result := 0;
    nPos := PosEx (sub, text, 1); // default
    while nPos > 0 do
        begin
            Inc (Result);
            nPos := PosEx (sub, text, nPos + Length (sub));
        end;
    end;
```

Both code snippets are used in a trivial way in the StrDemo example discussed earlier.

## The Types Unit

The Types unit holds data types common to multiple operating systems. In past versions of Delphi, the same types were defined by the Windows unit; now they've been moved to this common unit, shared by Delphi and Kylix. The types defined here are simple and include, among others, the `TPoint`, `TRect`, and `TSmallPoint` record structures plus their related pointer types.

**Warning** Notice that you will have to update old Delphi programs that refer to `TRect` or `TPoint`, by adding the Types unit in the `uses` statement; otherwise they won't compile.

## The Variants and VarUtils Units

Variants and VarUtils are two more units introduced in Delphi 6 to host the variant-related portion of the library. The Variants unit contains generic code for variants. As mentioned earlier, some of the routines in this unit have been moved here from the System unit. Functions include generic variant support, variant arrays, variant copying, and dynamic array to variant array conversions. In addition, the `TCustomVariantType` class defines customizable variant data types.

The Variants unit is totally platform independent and uses the VarUtils unit, which contains OS-dependent code. In Delphi, this unit uses the system APIs to manipulate variant data; in Kylix, it uses custom code provided by the RTL library.

**Note** In Delphi 7, these units have been extended and some bugs have been patched. The variant implementation has been heavily reworked behind the scenes to improve the speed of this technology and decrease the memory footprint of its code.

A specific area that has seen significant improvement in Delphi 7 is the ability to control the behavior of variant implementations, particularly comparison rules. Delphi 6 saw a change in the variant code so that *null* values cannot be compared with other values. This behavior is correct from a formal point of view, specifically for the fields of a dataset (an area in which variants are heavily used), but this change had the side effect of breaking existing code. Now you can control this behavior using the `NullEqualityRule` and `NullMagnitudeRule` global variables, each of which assumes one of the following values:

**ncrError** Any type of comparison causes an exception to be raised, because you cannot compare an undefined value; this was the (new) default behavior in Delphi 6.

**ncrStrict** Any type of comparison always fails (returning `False`), regardless of the values.

**ncrLoose** Equality tests succeed only among null values (a null is different from any other value). In comparisons null values are considered like empty values or zeros.

Other settings like `NullStrictConvert` and `NullAsStringValue` control how conversion is accomplished in case of null values. I suggest that you carry out your own experiments using the `VariantComp` example available with the code for this chapter. As you can see in [Figure 3.2](#), this program has a form with a `RadioGroup` you can use to change the settings of the `NullEqualityRule` and `NullMagnitudeRule` global variables, and a few buttons to perform various comparisons.

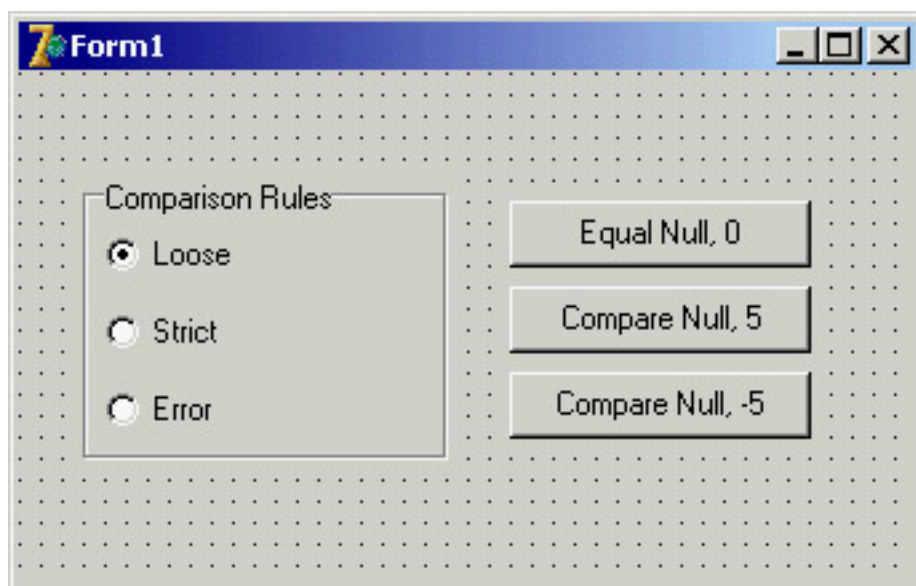


Figure 3.2: The form of the VariantComp example at design time

## Custom Variants and Complex Numbers

Another recent extension to the concept of variants is the possibility of extending the type system with custom variants. This technique allows you to define a new data type that, contrary to a class, overloads standard arithmetic operators.

A variant is a type holding both the type specification and the actual value. One variant can contain a string; another can contain a number. The system defines automatic conversions among variant types, allowing you to mix them inside operations (including custom variants). This flexibility comes at a high cost: Operations on variants are much slower than on native types, and variants use extra memory.

As an example of a custom variant type, Delphi ships with an interesting definition for complex numbers, found in the `VarCmplx` unit (available in source-code format in the `Rt1\Common` folder). You can create complex variants by using one of the overloaded `VarComplexCreate` functions and use them in any expression, as the following code fragment demonstrates:

```
var
  v1, v2: Variant;
begin
  v1 := VarComplexCreate (10, 12);
  v2 := VarComplexCreate (10, 1);
  ShowMessage (v1 + v2 + 5);
```

The complex numbers are defined using classes, but they are surfaced as variants by inheriting a new class from the `TCustomVariantType` class (defined in the `Variants` unit), overriding a few virtual abstract functions, and creating a global object that takes care of the registration within the system.

Besides these internal definitions, the `Variants` unit includes a long list of routines for operating on variants, including mathematical and trigonometric operations. I'll leave them to your study, because not all readers will be interested in complex numbers for their programs.

**Warning** Building a custom variant is certainly not an easy task, and I can hardly find reasons for using them instead of objects and classes. With a custom variant you gain the advantage of using operator overloading on your own data structures, but you lose compile-time checking, make the code much slower, miss several OOP features, and have to write a lot of rather complex code.

## The DelphiMM and ShareMem Units

The `DelphiMM` and `ShareMem` units relate to memory management. The standard Delphi memory manager is declared in the `System` unit.

The `DelphiMM` unit defines an alternative memory manager library to be used when passing strings from an executable to a DLL (a Windows dynamic linking library), both built with Delphi. This memory manager library is compiled by default in the `Borlndmm.dll` library file you'll have to deploy with your program.

The interface to this memory manager is defined in the `ShareMem` unit. You must include this unit (it's required to be the first unit) in the projects of both your executable and library, as described in more detail in [Chapter 10](#), "Libraries and Packages."

**Note** Unlike Delphi, Kylix has no `DelphiMM` and `ShareMem` units, because memory management is provided in the native Linux libraries (in particular, Kylix uses `malloc` from `glibc`) and so is effectively shared among different modules. In Kylix, however, applications with multiple modules must use the `ShareExcept` unit, which allows exceptions raised in a module to be surfaced to another module.

## COM-Related Units

`ComConst`, `ComObj`, and `ComServ` provide low-level COM support. These units are not really part of the RTL, from my point of view, so I won't discuss them here in any detail. You can refer to [Chapter 12](#) for all the related information.

These units have not changed much in recent versions of Delphi.

## Converting Data

As mentioned earlier in this chapter, Delphi includes a new conversion engine, defined in the `Conv Utils` unit. The engine by itself doesn't include any definition of actual measurement units; instead, it has a series of core functions for end users.

The key function is the conversion call, the `Convert` function. You simply provide the amount, the units it is expressed in, and the units you want it converted into. The following converts a temperature of 31 degrees Celsius to Fahrenheit:

```
Convert (31, tuCelsius, tuFahrenheit)
```

An overloaded version of the `Convert` function lets you convert values that have two units, such as speed (which has both a length unit and a time unit). For example, you can convert miles per hour to meters per second with this call:

```
Convert (20, duMiles, tuHours, duMeters, tuSeconds)
```

Other functions in the unit allow you to convert the result of an addition or a difference, check if conversions are applicable, and even list the available conversion families and units.

A predefined set of measurement units is provided in the `StdConvs` unit. This unit has conversion families and an impressive number of values, as shown in the following reduced excerpt:

```
// Distance Conversion Units
// basic unit of measurement is meters
cbDistance: TConvFamily;

duAngstroms: TConvType;
duMicrons: TConvType;
duMillimeters: TConvType;
duMeters: TConvType;
duKilometers: TConvType;
duInches: TConvType;
duMiles: TConvType;
duLightYears: TConvType;
duFurlongs: TConvType;
duHands: TConvType;
duPicas: TConvType;
```

This family and the various units are registered in the conversion engine in the initialization section of the unit, providing the conversion ratios (saved in a series of constants, such as `MetersPerInch` in the following code):

```
cbDistance := RegisterConversionFamily('Distance');
duAngstroms := RegisterConversionType(cbDistance, 'Angstroms', 1E-10);
duMillimeters := RegisterConversionType(cbDistance, 'Millimeters', 0.001);
duInches := RegisterConversionType(cbDistance, 'Inches', MetersPerInch);
```

To test the conversion engine, I built a generic example (`ConvDemo`) that allows you to work with the entire set of available conversions. The program fills a combo box with the available conversion families and a list box with the available units of the active family. This is the code:

```
procedure TForm1.FormCreate(Sender: TObject);
var
    i: Integer;
begin
    GetConvFamilies (aFamilies);
    for i := Low(aFamilies) to High(aFamilies) do
        ComboFamilies.Items.Add (ConvFamilyToDescription (aFamilies[i]));
    // get the first and fire event
    ComboFamilies.ItemIndex := 0;
```

```

ChangeFamily (self);
end;

procedure TForm1.ChangeFamily(Sender: TObject);
var
  aTypes: TConvTypeArray;
  i: Integer;
begin
  ListTypes.Clear;
  CurrFamily := aFamilies [ComboFamilies.ItemIndex];
  GetConvTypes (CurrFamily, aTypes);
  for i := Low(aTypes) to High(aTypes) do
    ListTypes.Items.Add (ConvTypeToDescription (aTypes[i]));
end;

```

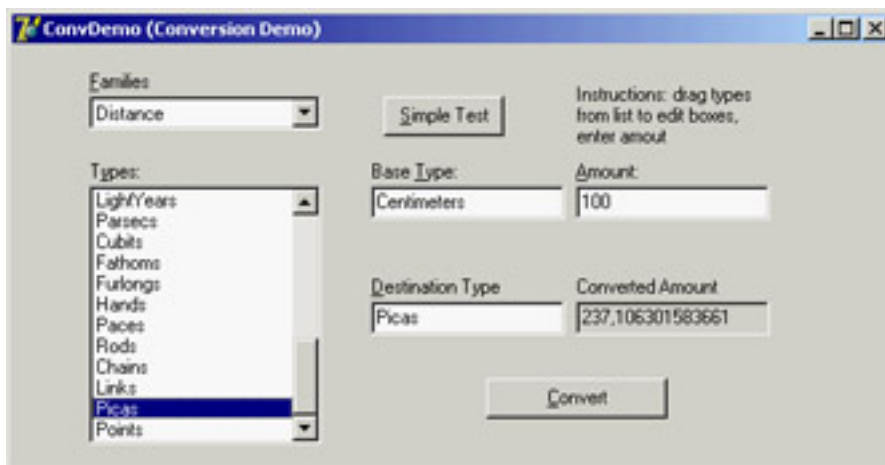
The aFamilies and CurrFamily variables are declared in the private section of the form as follows:

```

aFamilies: TConvFamilyArray;
CurrFamily: TConvFamily;

```

At this point, a user can enter two measurement units and an amount in the corresponding edit boxes on the form, as you can see in [Figure 3.3](#). To make the operation faster, the user can select a value in the list and drag it to one of the two Type edit boxes. The dragging support is described in the following sidebar "[Simple Dragging in Delphi](#)."



**Figure 3.3:** The ConvDemo example at run time

---

### Simple Dragging in Delphi

The ConvDemo example, built to show how to use the conversion engine, uses an interesting technique: dragging. You can move the mouse over the list box, select an item, and then keep the left mouse button pressed and drag the item over one of the edit boxes in the center of the form.

To accomplish this functionality, I had to set the `DragMode` property of the list box (the source component) to `dmAutomatic` and implement the `OnDragOver` and `OnDragDrop` events of the target edit boxes (the two edit boxes are connected to the same event handlers, sharing the same code). In the first method, the program indicates that the edit boxes always accept the dragging operation, regardless of the source. In the second method, the program copies the text selected in the list box (the `Source` control of the dragging operation) to the edit box that fired the event (the `Sender` object). Here is the code for the two methods:

```

procedure TForm1.EditTypeDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;

```

```

end;

procedure TForm1.EditTypeDragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  (Sender as TEdit).Text := (Source as TListBox).Items
    [(Source as TListBox).ItemIndex];
end;

```

---

The units must match those available in the current family. In case of error, the text in the Type edit boxes is shown in red. This is the effect of the first part of the form's `DoConvert` method, which is activated as soon as the value of one of the edit boxes for the units or the amount changes. After checking the types in the edit boxes, the `DoConvert` method performs the conversion, displaying the result in the fourth, grayed edit box. In case of errors, you'll get an appropriate message in the same box. Here is the code:

```

procedure TForm1.DoConvert(Sender: TObject);
var
  BaseType, DestType: TConvType;
begin
  // get and check base type
  if not DescriptionToConvType(CurrFamily, EditType.Text, BaseType) then
    EditType.Font.Color := clRed
  else
    EditType.Font.Color := clBlack;

  // get and check destination type
  if not DescriptionToConvType(CurrFamily, EditDestination.Text,
    DestType) then
    EditDestination.Font.Color := clRed
  else
    EditDestination.Font.Color := clBlack;

  if (DestType = 0) or (BaseType = 0) then
    EditConverted.Text := 'Invalid type'
  else
    EditConverted.Text := FloatToStr (Convert (
      StrToFloat (EditAmount.Text), BaseType, DestType));
end;

```

If all this is not interesting enough for you, consider that the conversion types provided serve only as a demo: You can fully customize the engine by providing the measurement units you are interested in, as described in the [next section](#).

## What About Currency Conversions?

Converting currencies is not exactly the same as converting measurement units, because currency rates change constantly. In theory, you can register a conversion rate with Delphi's conversion engine. From time to time, you check the new rate of exchange, unregister the existing conversion, and register a new one. However, keeping up with the actual rate means changing the conversion so often that the operation might not make a lot of sense. Also, you'll have to triangulate conversions: You must define a base unit (probably the U.S. dollar if you live in America) and convert to and from this currency even if you're converting between two different currencies.

It's more interesting to use the engine to convert member currencies of the euro, for two reasons. First, conversion rates are fixed (until the single euro currency takes over). Second, the conversion among euro currencies is legally done by converting a currency to euros first and then from the euro amount to the other currency—the exact behavior of Delphi's conversion engine. There is one small problem: You should apply a rounding algorithm at every step of the conversion. I'll consider this problem after I've provided the base code for integrating euro currencies with the Delphi conversion engine.

**Note** The ConvertIt demo available among the Delphi examples provides support for euro conversions, using a slightly different rounding approach, still not as precise as demanded by the European currency conversion rules. I've decided to keep this example, because it is instructive in showing how to create a new measurement system.

The example, called EuroConv, teaches how to register any new measurement unit with the engine. Following the template provided by the StdConvs unit, I've created a new unit (called EuroConvConst). In the interface portion, I've declared variables for the family and the specific units, as follows:

### interface

#### var

```
// Euro Currency Conversion Units
cbEuroCurrency: TConvFamily;

cuEUR: TConvType;
cuDEM: TConvType; // Germany
cuESP: TConvType; // Spain
cuFRF: TConvType; // France
// and so on...
```

The implementation portion of the unit defines constants for the various official conversion rates:

### implementation

#### const

```
DEMPerEuros = 1.95583;
ESPPerEuros = 166.386;
FRFPerEuros = 6.55957;
// and so on...
```

Finally, the unit initialization code registers the family and the various currencies, each with its own conversion rate and a readable name:

### initialization

```
// Euro Currency's family type
cbEuroCurrency := RegisterConversionFamily('EuroCurrency');

cuEUR := RegisterConversionType(
    cbEuroCurrency, 'EUR', 1);
cuDEM := RegisterConversionType(
```



```

cbEuroCurrency, 'DEM', 1 / DEMPerEuros);
cuESP := RegisterConversionType(
    cbEuroCurrency, 'ESP', 1 / ESPPerEuros);
cuFRF := RegisterConversionType(
    cbEuroCurrency, 'FRF', 1 / FRFPerEuros);

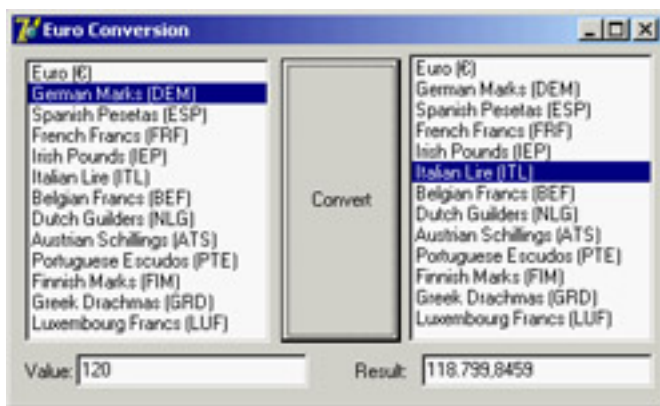
```

**Note** The engine uses as a conversion factor the amount of the base unit necessary to obtain the secondary units, with a constant like *MetersPerInch*, for example. The standard rate of euro currencies is defined the opposite way. For this reason, I've kept the conversion constants with the official values (like *DEMPerEuros*) and passed them to the engine as fractions ( $1/DEMPerEuros$ ).

Having registered this unit, you can now convert 120 German marks to Italian liras as follows:

```
Convert (120, cuDEM, cuITL)
```

The demo program does a little more: It provides two list boxes with the available currencies, extracted as in the previous example, and edit boxes for the input value and final result. You can see the form at run time in [Figure 3.4](#).



**Figure 3.4:** The output of the EuroConv unit, showing the use of Delphi's conversion engine with a custom measurement unit

The program works nicely but is not perfect, because the proper rounding is not applied; you should round not only the final result of the conversion but also the intermediate value. Using the conversion engine to accomplish this rounding directly is not easy. The engine allows you to provide either a custom conversion function or a conversion rate. But writing identical conversion functions for all the currencies seems like a bad idea, so I've decided to take a different path. (You can see examples of custom conversion functions in the StdConvs unit, in the portion related to temperatures.)

In the EuroConv example, I've added to the unit with the conversion rates a custom *EuroConv* function that does the proper conversion. Simply calling this function instead of the standard *Convert* function does the trick (and I see no drawback to this approach, because in such programs you'll rarely mix currencies with distances or temperatures). As an alternative, I could have inherited a new class from *TConvTypeFactor*, providing a new version of the *FromCommon* and *ToCommon* methods; or I could have called the overloaded version of the *RegisterConversionType* that accepts these two functions as parameters. However, neither of these techniques would have allowed me to handle special cases, such as the conversion of a currency to itself.

This is the code of the *EuroConv* function, which uses the internal *EuroRound* function to round to the number of digits specified in the *Decimals* parameter (which must be between 3 and 6, according to the official rules):

```

type
    TEuroDecimals = 3..6;

function EuroConvert (const AValue: Double;
    const AFrom, ATo: TConvType;
    const Decimals: TEuroDecimals = 3): Double;

```



```
function EuroRound (const AValue: Double): Double;  
begin  
    Result := AValue * Power (10, Decimals);  
    Result := Round (Result);  
    Result := Result / Power (10, Decimals);  
end;
```

```
begin  
    // check special case: no conversion  
    if AFrom = ATo then  
        Result := AValue  
    else  
        begin  
            // convert to Euro, then round  
            Result := ConvertFrom (AFrom, AValue);  
            Result := EuroRound (Result);  
            // convert to currency then round again  
            Result := ConvertTo (Result, ATo);  
            Result := EuroRound (Result);  
        end;  
    end;
```

Of course, you might want to extend the example by providing conversion to other non-euro currencies, eventually picking the values automatically from a website. I'll leave this as a rather complex exercise for you.

**Team LiB**

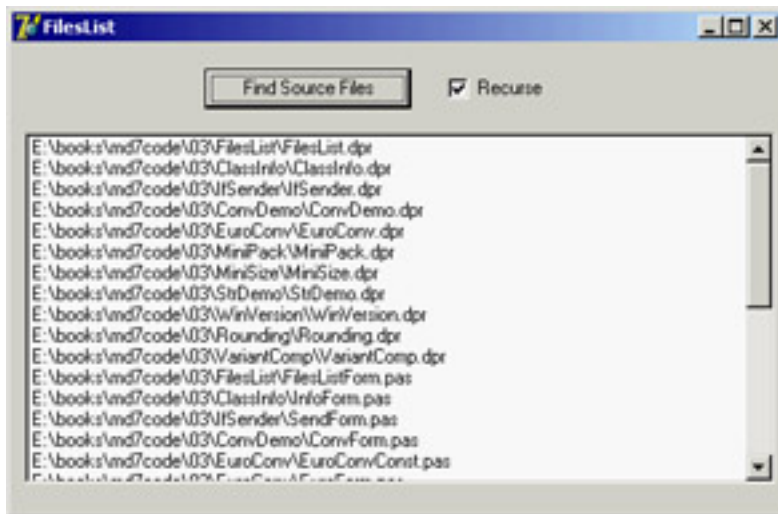
◀ PREVIOUS

NEXT ▶

## Managing Files with SysUtils

To access files and file information, you can generally rely on the standard functions available in the SysUtils unit. Relying on these fairly traditional Pascal libraries makes your code easily portable among quite different operating systems (although you'll have to consider with great care the differences in the file system architectures, particularly case sensitivity on the Linux platform).

For example, the FilesList example uses the `FindFirst`, `FindNext`, and `FindClose` combination to retrieve from within a folder a list of files that match a filter, with the same code you could use on Kylix and Linux (an example of the output appears in [Figure 3.5](#)).



**Figure 3.5:** An example of the output of the FilesList application

The following code adds the filenames to a list box called `lbFiles`:

```
procedure TForm1.AddFilesToList(Filter, Folder: string;
  Recurse: Boolean);
var
  sr: TSearchRec;
begin
  if FindFirst (Folder + Filter, faAnyFile, sr) = 0 then
    repeat
      lbFiles.Items.Add (Folder + sr.Name);
    until FindNext(sr) <> 0;
  FindClose(sr);
```

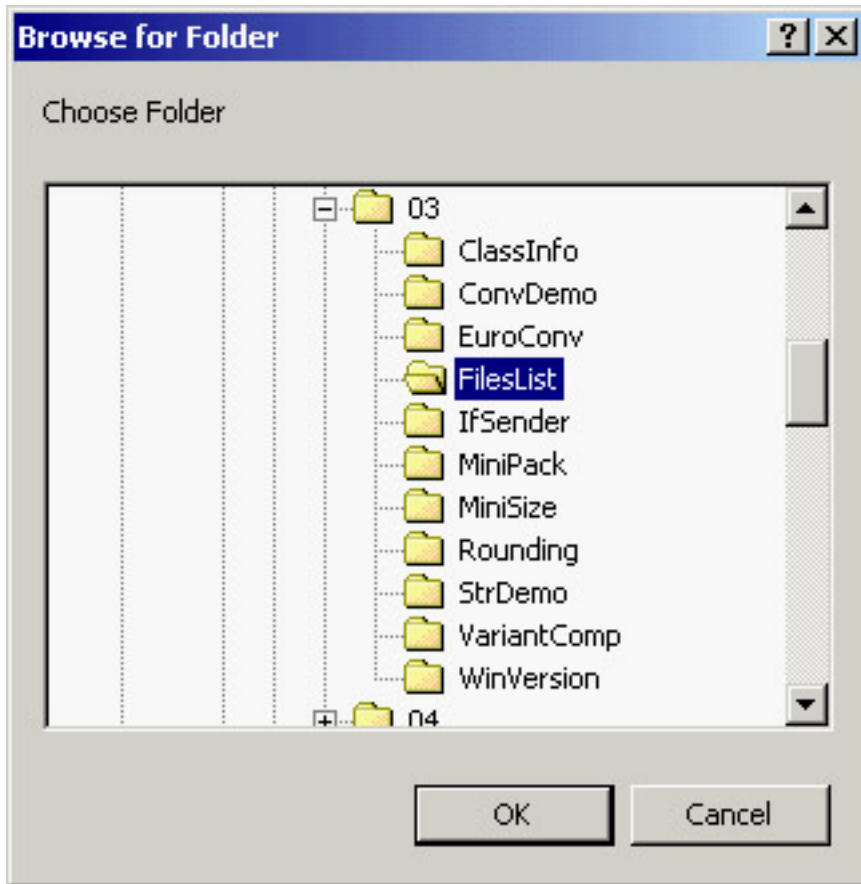
If the `Recurse` parameter is set, the `AddFilesToList` procedure gets a list of subfolders by examining the local files again, and then calls itself for each of the subfolders. The list of folders is placed in a string list object, with the following code:

```
procedure GetSubDirs (Folder: string; sList: TStringList);
var
  sr: TSearchRec;
begin
  if FindFirst (Folder + '.*', faDirectory, sr) = 0 then
    try
      repeat
        if (sr.Attr and faDirectory) = faDirectory then
          sList.Add (sr.Name);
        until FindNext(sr) <> 0;
    finally
```

```
FindClose(sr);  
end;  
end;
```

Finally, the program uses an interesting technique to ask the user to select the initial directory for the file search, by calling the `SelectDirectory` procedure (see [Figure 3.6](#)):

```
if SelectDirectory ('Choose Folder', '', CurrentDir) then ...
```



**Figure 3.6:** The dialog box of the `SelectDirectory` procedure, displayed by the `FilesList` application

## The *TObject* Class

As mentioned earlier, a key element of the System unit is the definition of the `TObject` class, which is the *mother of all Delphi classes*. Every class in the system inherits from the `TObject` class, either directly (if you specify `TObject` as the base class), implicitly (if you indicate no base class), or indirectly (when you specify another class as the ancestor). The entire hierarchy of classes in an Object Pascal program has a single root. So, you can use the `TObject` data type as a replacement for the data type of any class type in the system, according to the type compatibility rules covered in [Chapter 2](#) in the section "[Inheritance and Type Compatibility](#)."

For example, components' event handlers usually have a `Sender` parameter of type `TObject`. This simply means that the `Sender` object can be of any class, because every class is ultimately derived from `TObject`. The typical drawback of such an approach is that to work on the object, you need to know its data type. In fact, when you have a variable or a parameter of the `TObject` type, you can apply to it only the methods and properties defined by the `TObject` class itself. If this variable or parameter happens to refer to an object of the `TButton` type, for example, you cannot directly access its `Caption` property. The solution to this problem lies in the use of the safe down-casting or run-time type information (RTTI) operators (`is` and `as`) discussed in [Chapter 2](#).

You can also use another approach. For any object, you can call the methods defined in the `TObject` class. For example, the `ClassName` method returns a string with the name of the class. Because it is a class method (see [Chapter 2](#) for details), you can apply it both to an object and to a class. Suppose you have defined a `TButton` class and a `Button1` object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;  
Text := TButton.ClassName;
```

On some occasions you need to use the name of a class, but it can also be useful to retrieve a class reference to the class itself or to its base class. The class reference allows you to operate on the class at run time (as you saw in the preceding chapter), whereas the class name is just a string. You can get these class references with the `ClassType` and `ClassParent` methods. The first returns a class reference to the class of the object; the second returns a class reference to the object's base class. Once you have a class reference, you can apply to it any class methods of `TObject`—for example, to call the `ClassName` method.

Another method that might be useful is `InstanceSize`, which returns the run-time size of an object. Although you might think that the `SizeOf` global function provides this information, that function actually returns the size of an object reference—a pointer, which is invariably four bytes—instead of the size of the object itself.

In [Listing 3.1](#), you can find the complete definition of the `TObject` class, extracted from the System unit. In addition to the methods I've already mentioned, notice `InheritsFrom`, which provides a test that's similar to the `is` operator but that can also be applied to classes and class references (the first argument of `is` must be an object).

### Listing 3.1: The definition of the *TObject* class (in the System RTL unit)

---

```
type  
  TObject = class  
    constructor Create;  
    procedure Free;  
    class function InitInstance(Instance: Pointer): TObject;  
    procedure CleanupInstance;  
    function ClassType: TClass;  
    class function ClassName: ShortString;  
    class function ClassNameIs(  
      const Name: string): Boolean;  
    class function ClassParent: TClass;
```

```

class function ClassInfo: Pointer;
class function InstanceSize: Longint;
class function InheritsFrom(AClass: TClass): Boolean;
class function MethodAddress(const Name: ShortString): Pointer;
class function MethodName(Address: Pointer): ShortString;
function FieldAddress(const Name: ShortString): Pointer;
function GetInterface(const IID: TGUID;out Obj): Boolean;
class function GetInterfaceEntry(
    const IID: TGUID): PInterfaceEntry;
class function GetInterfaceTable: PInterfaceTable;
function SafeCallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HRESULT; virtual;
procedure AfterConstruction; virtual;
procedure BeforeDestruction; virtual;
procedure Dispatch(var Message); virtual;
procedure DefaultHandler(var Message); virtual;
class function NewInstance: TObject; virtual;
procedure FreeInstance; virtual;
destructor Destroy; virtual;
end;

```

---

**Note** The *ClassInfo* method returns a pointer to the internal run-time type information (RTTI) of the class, introduced in the [next chapter](#).

These methods of *TObject* are available for objects of every class, because *TObject* is the common ancestor class of every class. Here is how you can use these methods to access class information:

```

procedure TSenderForm.ShowSender(Sender: TObject);
begin
    Mem1.Lines.Add ('Class Name: ' + Sender.ClassName);

    if Sender.ClassParent <> nil then
        Mem1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);

    Mem1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
end;

```

The code checks to see whether the *ClassParent* is *nil*, in case you are using an instance of the *TObject* type, which has no base type.

This *ShowSender* method is part of the *IfSender* example. The method is connected with the *OnClick* event of several controls: three buttons, a check box, and an edit box. When you click each control, the *ShowSender* method is invoked with the corresponding control as sender (more on events in [Chapter 4](#)). One of the buttons is a *Bitmap* button, an object of a *TButton* subclass. You can see an example of the output of this program at run time in [Figure 3.7](#).

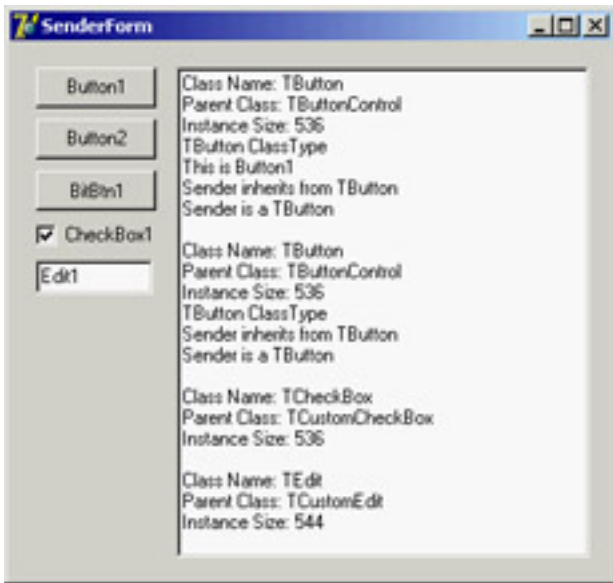


Figure 3.7: The output of the IfSender example

You can use other methods to perform tests. For example, you can check whether the `Sender` object is of a specific type with the following code:

```
if Sender.ClassType = TButton then ...
```

You can also check whether the `Sender` parameter corresponds to a given object, with this test:

```
if Sender = Button1 then...
```

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class *or* one of its subclasses. Doing so lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the `InheritsFrom` method, which is also called when you use the `is` operator. The following two tests are equivalent:

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

## Showing Class Information

I've extended the `IfSender` example to show a complete list of base classes of a given object or class. Once you have a class reference you can add all of its base classes to the `ListParent` list box with the following code:

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
```

You'll notice that I use a class reference at the heart of the `while` loop, which tests for the absence of a parent class (so that the current class is `TObject`). Alternatively, I could have written the `while` statement in either of the following ways:

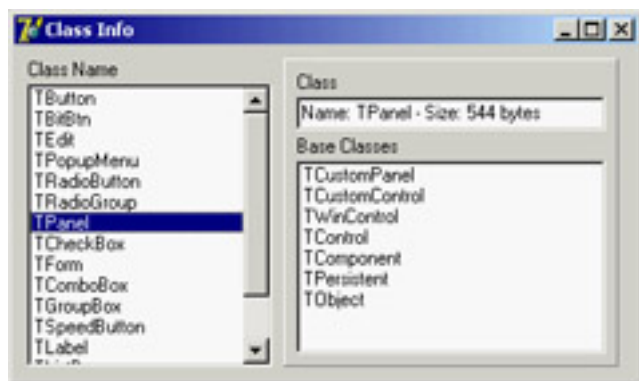
```
while not MyClass.ClassNameIs ('TObject') do...
while MyClass <> TObject do...
```

The code in the `with` statement referring to the `ListParent` list box is part of the `ClassInfo` example, which displays the list of parent classes and some other information about a few components of the VCL (basically those on the Standard page of the Component Palette). These components are manually added to a dynamic array holding classes and declared as

**private**

```
ClassArray: array of TClass;
```

When the program starts, the array is used to show all the class names in a list box. Selecting an item from the list box triggers the visual presentation of its details and its base classes, as you can see in the program output in [Figure 3.8](#).



**Figure 3.8:** The output of the `ClassInfo` example

**Note** As a further extension of this example, you can create a tree with all the base classes of the various components in a hierarchy. To do that, I've created the `VclHierarchy` wizard, discussed in [Appendix A](#) ("Extra Delphi Tools by the Author").

## What's Next?

In this chapter, I've focused my attention on new features of the Delphi function-based run-time library. I have provided only a summary of the RTL, not a complete overview (which would take too much space). You can find more examples of the basic RTL functions of Delphi in the free e-books available on my website, as discussed in [Appendix C](#).

In the [next chapter](#), we'll begin moving from the function-based RTL to the class-based RTL, which is the core of Delphi's class library. I won't debate whether the core classes common to the VCL and CLX, such as `TObject`, actually belong to the RTL or the class library. I've covered everything defined in `System`, `SysUtils`, and other units hosting functions and procedures in this chapter; the [next chapter](#) focuses on the `Classes` unit and other core units that define classes.

Along with the preceding chapter on the Delphi language, [Chapter 4](#) will provide a foundation for discussing visual- and database-oriented classes (or components, if you prefer). Looking at the various library units, you'll find many more global functions, which don't belong to the core RTL but are still quite useful.



## Chapter 4: Core Library Classes

### Overview

We saw in the preceding chapter that Delphi includes a large number of functions and procedures, but the real power of Delphi's visual programming lies in the huge class library that comes with it. Delphi's standard class library contains hundreds of classes, with thousands of methods, and it is so large that I certainly cannot provide a detailed reference in this book. What I'll do, instead, is explore various areas of this library starting with this chapter and continuing through those that follow.

This chapter is devoted to the library's core classes as well as to some standard programming techniques, such as the definition of events. We'll explore some commonly used classes, such as lists, string lists, collections, and streams. We'll devote most of our time to exploring the content of the Classes unit, but we'll also examine other core units of the library.

Delphi classes can be used either entirely in code or within the visual form designer. Some of them are component classes, which show up in the Component Palette, and others are more general-purpose. The terms *class* and *component* can be used almost as synonyms in Delphi. Components are the central elements of Delphi applications. When you write a program, you basically choose a number of components and define their interactions— that's all there is to Delphi visual programming.

Before reading this chapter, you need to have a good understanding of the language, including inheritance, properties, virtual methods, class references, and so on, as discussed in [Chapter 2](#), "The Delphi Programming Language."

## The RTL Package, VCL, and CLX

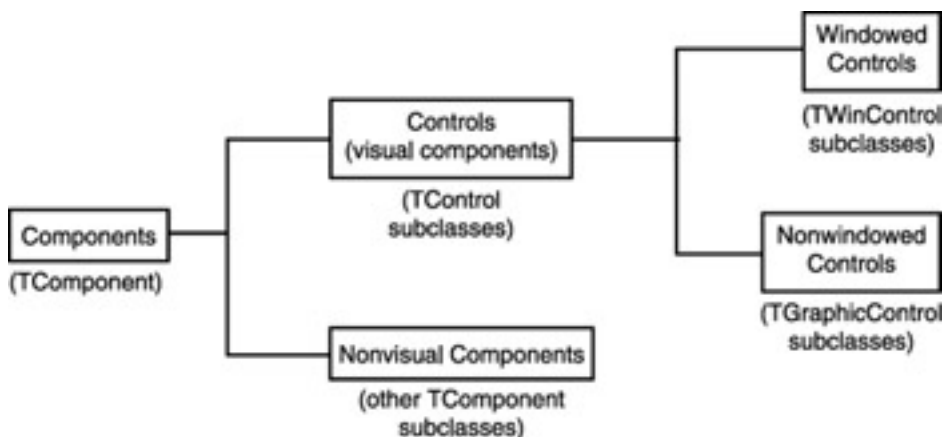
Until version 5, Delphi's class library was known as VCL, which stands for Visual Components Library. This is a component library mapped on top of the Windows API. Kylix, the Delphi version for Linux, introduced a new component library, called CLX (Component Library for X-Platform or Cross Platform; the acronym is pronounced "clicks"). Delphi 6 was the first edition to include both the VCL and CLX libraries. For visual components, the two class libraries are alternative one to the other. However, the core classes and the database and Internet portions of the two libraries are basically shared.

VCL was considered as a single large library, although programmers used to refer to different parts of it (components, controls, nonvisual components, data sets, data-aware controls, Internet components, and so on). CLX introduces a distinction in four parts: BaseCLX, VisualCLX, DataCLX, and NetCLX. Only in VisualCLX does the library use a totally different approach between the Windows and Linux platforms; the rest of the code is inherently portable to Linux. In the following section, I discuss these two libraries; the rest of the chapter focuses on the common core classes.

In recent versions of Delphi, this distinction is underlined by the fact that the core non-visual components and classes of the library are part of the new RTL package, which is used by both VCL and CLX. Moreover, using this package in non-visual applications (for example, web server programs) allows you to reduce considerably the size of the files to deploy and load in memory.

### Traditional Sections of VCL

Delphi programmers used to refer to the sections of the VCL with names Borland originally suggested in its documentation—names that became common afterward for different groups of components. Technically, components are subclasses of the `TComponent` class, which is one of the root classes of the hierarchy, as you can see in [Figure 4.1](#). The `TComponent` class inherits from the `TPersistent` class; the role of these two classes will be explained in the [next section](#).



**Figure 4.1:** A graphical representation of the main groups of VCL components

In addition to components, the library includes classes that inherit directly from `TObject` and from `TPersistent`. These classes are collectively known as *Objects* in portions of the documentation, a rather confusing name. These noncomponent classes are often used for values of properties or as utility classes used in code; not inheriting from `TComponent`, these classes cannot be used directly in visual programming.

**Note** To be more precise, noncomponent classes cannot be made available in the Component Palette and cannot be dropped directly into a form, but they can be visually managed with the Object Inspector as subproperties of other properties or items of collections of various types. So, even noncomponent classes are often easily used when interacting with the Form Designer.

The component classes can be further divided into two main groups: controls and nonvisual components.

**Controls** All the classes that descend from `TControl`. Controls have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different subspecifications—window-based or graphical—that I'll discuss in more detail in [Chapter 5](#), "Visual Controls."

**Nonvisual Components** All the components that are not controls—all the classes that descend from `TComponent` but not from `TControl`. At design time, a nonvisual component appears on the form or data module as an icon, with a caption below it (the caption is optional on forms). At run time, some of these components may be visible (for example, the standard dialog boxes), and others are always invisible (for example, the database table component).

**Tip** You can simply move the mouse cursor over a control or component in the Form Designer to see a Tooltip with its name and class type (and some extended information). You can also use an environment option, Show Component Captions, to see the name of a nonvisual component under its icon.

The traditional subdivision of VCL is very common for Delphi programmers. Even with the introduction of CLX and some new naming schemes, the traditional names will probably survive and merge into Delphi programmers' jargon.

## The Structure of CLX

Borland now refers to different portions of the CLX library using one terminology under Linux and a slightly different (and less clear) naming structure in Delphi. This new subdivision of the cross-platform library represents more logical areas than the structure of the class hierarchy:

**BaseCLX** The core of the class library: the topmost classes (such as `TComponent`) and several general utility classes (including lists, containers, collections, and streams). Compared to the corresponding classes of VCL, BaseCLX is largely unchanged and is highly portable between the Windows and Linux platforms. This chapter is largely devoted to exploring BaseCLX and the common VCL core classes.

**VisualCLX** The collection of visual components, generally called controls. This is the portion of the library that is more tightly related to the operating system: VisualCLX is implemented on top of the Qt library, available both on Windows and on Linux. Using VisualCLX allows for full portability of the visual portion of your application between Delphi on Windows and Kylix on Linux. However, most of the VisualCLX components have corresponding VCL controls, so you can also easily move your code from one library to the other. I'll discuss VisualCLX and the controls of VCL in [Chapter 5](#).

**DataCLX** All the database-related components of the library. DataCLX is the front end of the new dbExpress database engine included in both Delphi and Kylix. Delphi also includes the traditional BDE front end, dbGo, and InterBase Express (IBX). If we consider all these components as part of DataCLX, only the dbExpress front end and IBX are portable between Windows and Linux. In addition, DataCLX includes the ClientDataSet component, now called MyBase, and other related classes. Delphi's data access components are discussed in [Part III](#) of the book.

**NetCLX** The Internet-related components, from the WebBroker framework to the HTML producer components, from Indy (Internet Direct) to Internet Express, from WebSnap to XML support. This part of the library is, again, highly portable between Windows and Linux. Internet support is discussed in [Part IV](#) of the book. (The name is short for Internet CLX, and has nothing to do with the Microsoft .NET technology it predates.)

## VCL-Specific Sections of the Library

The preceding areas of the library are available, with the differences I've mentioned, on both Delphi and Kylix. In Delphi, however, other sections of the VCL are for one reason or another specific to Windows only:

- The Delphi ActiveX (DAX) framework provides support for COM, OLE Automation, ActiveX, and other COM-related technologies. See [Chapter 12](#), "From COM to COM+," for more information on this area of Delphi.
- The Decision Cube components provide Online Analytical Processing (OLAP) support but have ties with the BDE and haven't been updated recently. Decision Cube is not discussed in the book.

Finally, the default Delphi installation includes some third-party components, such as TeeChart for business graphics, RAVE for reporting and printing, and IntraWeb for Internet development. Some of these components will be discussed in the book, but they are not strictly part of the VCL. RAVE and IntraWeb are also available for Kylix.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## The *TPersistent* Class

The first core class of the Delphi library we'll look at is `TPersistent`, which is quite a strange class: It has very little code and almost no direct use, but it provides a foundation for the entire idea of visual programming. You can see the definition of the class in [Listing 4.1](#).

### Listing 4.1: The Definition of the *TPersistent* Class, from the *Classes* Unit

---

```
{ $M+ }
TPersistent = class(TObject)
private
    procedure AssignError(Source: TPersistent);
protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFiler); virtual;
    function GetOwner: TPersistent; dynamic;
public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function GetNamePath: string; dynamic;
end;
```

---

As the name implies, this class handles persistency—that is, saving the value of an object to a file to be used later to re-create the object in the same state and with the same data. Persistency is a key element of visual programming. In fact (as you saw in [Chapter 1](#), "Delphi 7 and Its IDE"), at design time in Delphi you manipulate actual objects, which are saved to DFM files and re-created at run time when the specific component container— form or data module—is created.

**Note** Everything I say about DFM files also applies to XFM files, the file format used by CLX applications. The format is identical. The extension difference is relevant because Delphi uses it to determine whether the form is based on CLX/Qt or on VCL/Windows. In Kylix, every form is a CLX/Qt form, regardless of which extension is used; so, the XFM/ DFM file extension in Kylix really doesn't matter.

Streaming support is not embedded in the `TPersistent` class but is provided by other classes, which target `TPersistent` and its descendants. In other words, you can "persist" with Delphi default streaming-only objects of classes inheriting from `TPersistent`. One of the reasons for this behavior lies in the fact that the class is compiled with a special option turned on, `{ $M+ }`. This flag activates the generation of extended RTTI information for the published portion of the class.

Delphi's streaming system doesn't try to save the in-memory data of an object, which would be complex because of the many pointers to other memory locations and totally meaningless when the object was reloaded. Instead, Delphi saves objects by listing the values of all properties in the `published` section of the class. When a property refers to another object, Delphi saves the name of the object or the entire object (with the same mechanism) depending on its type and relationship with the main object. For a comparison with other approaches, see the sidebar "[Object Streaming versus Code Generation](#)."

The only method of the `TPersistent` class that you'll generally use is the `Assign` procedure, which can be used to copy the actual value of an object. In the library, this method is implemented by many noncomponent classes but by very few components. Most subclasses reimplement the virtual protected `AssignTo` method, called by the default implementation of `Assign`.

Other methods include `DefineProperties`, used for customizing the streaming system and adding extra information (pseudo-properties); and the `GetOwner` and `GetNamePath` methods, used by collections and other special classes to identify themselves to the Object Inspector.

---

## Object Streaming versus Code Generation

The approach used by Delphi (and Kylix) is different from the approach used by other visual development tools and languages. For example, in Java, the effect of the definition of a form inside an IDE is the generation of the Java source code used to create the components and set their properties. Setting properties in an inspector affects the source code. Something similar happens in C#, although properties in this language are closer to the notion of properties in Delphi. You've already seen that in Delphi; you can write code to generate the components instead of relying on streaming, but because there is no specific support in the IDE you'll have to write that code manually.

Each of the two approaches has advantages and disadvantages. When generating source code, you have more control over what goes on and the exact sequence of creation and initialization. Delphi reloads objects and their properties but delays some assignments until a later fix-up phase, to avoid the problems of references to not-yet-initialized objects. This process is more complex, but it is so hidden that it becomes simpler for the programmer.

The Java language allows a tool like JBuilder to recompile a form class and load it in a running program for every change. In a compiled system like Delphi, this approach would be more complex (Delphi uses a fake version, technically called a *proxy*, of your form at design time, not the actual form).

One advantage of the approach used by Delphi is that the DFM files can be translated into different languages without affecting the source code; this is one reason Java is offering XML persistence of forms. Another difference is that Delphi embeds the component's graphic in the DFM file, instead of referring to external files. Doing so simplifies deployment (because everything ends up in the executable file) but can also make the executable much bigger.

---

## The *published* Keyword

Delphi has four directives specifying data access: `public`, `protected`, `private`, and `published`. I've covered the first three in [Chapter 2](#), "The Delphi Programming Language," so now it's time to look at what `published` means. For any `published` field, property, or method, the compiler generates extended RTTI information, so that Delphi's run-time environment or a program can query a class for its published interface. For example, every Delphi component has a `published` interface that is used by the IDE, in particular the Object Inspector. A proper use of `published` items is important when you write components. Usually, the `published` part of a component contains no fields or methods, just properties and events.

When Delphi generates a form or data module, it places the definitions of its components and methods (the event handlers) in the first portion of its definition, before the `public` and `private` keywords. These fields and methods in the initial portion of the class are `published`. The default is `published` when no special keyword is added before an element of a component class.

To be more precise, `published` is the default keyword only if the class was compiled with the `$M+` compiler directive or is descended from a class compiled with `$M+`. This directive is used in the `TPersistent` class, so most classes of the VCL and all the component classes default to `published`. However, noncomponent classes in Delphi (such as `TStream` and `TList`) are compiled with `$M-` and default to `public` visibility.

The methods used to handle events in the IDE (and in DFM files) should be `published`, and the fields corresponding to your components in the form should be `published` to be automatically connected with the objects described in the DFM file and created along with the form. (Later in this chapter I'll discuss the details of this situation and the problems it generates.)

---

## Accessing Published Fields and Methods

As I've mentioned, three different declarations make sense in the `published` section of a class: fields, methods, and properties. In your code, you'll generally refer to `published` items like you refer to `public` ones, that is, by

referring to the corresponding identifiers in the code. In some special cases, though, it is possible to access published items at runtime by name. I'll discuss dynamic access to properties in the section "[Accessing Properties by Name](#);" here I'll introduce possible ways of interacting at runtime with fields and methods. The `TObject` class has three interesting methods for this area: `MethodAddress`, `MethodName`, and `FieldAddress`.

The first function, `MethodAddress`, returns the memory address of the compiled code (a sort of function pointer) of the method passed as parameter in a string. By assigning this method address to the `Code` field of a `TMethod` structure and assigning an object to the `Data` field, you can obtain a complete method pointer. At this point, to call the method you must cast it to the proper method pointer type. Here is a code fragment highlighting the key points of this technique:

```
var
  Method: TMethod;
  Evt: TNotifyEvent;
begin
  Method.Code := MethodAddress ('Button1Click');
  Method.Data := Self;
  Evt := TNotifyEvent(Method);
  Evt (Sender); // call the method
end;
```

Delphi uses similar code to assign an event handler when it loads a DFM file, because these files store the name of the methods used to handle the events, whereas the components store the method pointer. The second method, `MethodName`, does the opposite transformation, returning the name of the method at a given memory address. This method can be used to obtain the name of an event handler, given its value, something Delphi does when streaming a component into a DFM file.

Finally, the `FieldAddress` method of `TObject` returns the memory location of a published field, given its name. Delphi uses this method to connect components created from the DFM files with the fields of their owner (for example, a form) having the same name.

Note that these three methods are seldom used in "normal" programs but play a key role in making Delphi work. They are strictly related to the streaming system. You'll need to use these methods only when writing extremely dynamic programs, special-purpose wizards, or other Delphi extensions.

---

## Accessing Properties by Name

The Object Inspector displays a list of an object's published properties, even for components you've written. To do this, it relies on the RTTI information generated for published properties. Using some advanced techniques, an application can retrieve a list of an object's published properties and use them.

Although this capability is not very well known, in Delphi it is possible to access properties by name simply by using the string with the name of the property and then retrieving its value. Access to the RTTI information of properties is provided through a group of undocumented subroutines, part of the `TTypeInfo` unit.

**Warning** These subroutines have always been undocumented in past versions of Delphi, so that Borland remained free to change them. However, from Delphi 1 to Delphi 7, changes were very limited and related only to supporting new features, with a high level of backward compatibility. In Delphi 5, Borland added many more goodies and a few "helper" routines that are officially promoted (even if still not fully documented in the Help file but explained only with comments provided in the unit).

Rather than explore the entire `TTypeInfo` unit here, we will look at only the minimal code required to access properties by name. Prior to Delphi 5, it was necessary to use the `GetPropInfo` function to retrieve a pointer to some internal property information and then apply one of the access functions, such as `GetStrProp`, to this pointer. You also had to



check for the existence and the type of the property.

Now you can use a new set of `TypeInfo` routines, including the handy `GetPropValue`, which returns a variant with the value of the property and raises an exception if the property doesn't exist. To avoid the exception, you can call the `IsPublishedProp` function first. You simply pass to these functions the object and a string with the property name. A further optional parameter of `GetPropValue` allows you to choose the format for returning values of properties of any set type (either a string or the numeric value for the set). For example, you can call

```
ShowMessage (GetPropValue (Button1, 'Caption'));
```

This call has the same effect as calling `ShowMessage`, passing as parameter `Button1.Caption`. The only real difference is that this version of the code is much slower, because the compiler generally resolves normal access to properties in a more efficient way. The advantage of the run-time access is that you can make it very flexible, as in the following `RunProp` example.

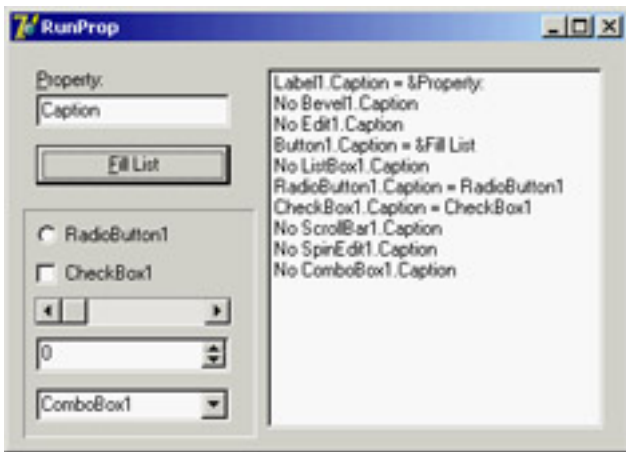
This program displays in a list box the value of a property of any type for each component of a form. The name of the property you are looking for is provided in an edit box. Being able to type the property name in the edit box makes the program very flexible. In addition to the edit box and the list box, the form has a button to generate the output and some other components added only to test their properties. When you click the button, the following code is executed:

```
uses
  TypeInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Value: Variant;
begin
  ListBox1.Clear;
  for I := 0 to ComponentCount -1 do
  begin
    if IsPublishedProp (Components[I], Edit1.Text) then
    begin
      Value := GetPropValue (Components[I], Edit1.Text);
      ListBox1.Items.Add (Components[I].Name + '.' +
        Edit1.Text + ' = ' + string (Value));
    end
    else
      ListBox1.Items.Add ('No ' + Components[I].Name + '.' +
        Edit1.Text);
  end;
end;
```

[Figure 4.2](#) shows the effect of clicking the Fill List button while using the default `Caption` value in the edit box. You can try it with any other property name. Numbers will be converted to strings by the variant conversion. Objects (such as the value of the `Font` property) will be displayed as memory addresses.





**Figure 4.2:** The output of the RunProp example, which accesses properties by name at run time

**Warning** Do not use regularly the TypInfo unit instead of polymorphism and other property-access techniques. Use base-class property access first, or use the safe *as* typecast when required, and reserve RTTI access to properties as a last resort. Using TypInfo techniques makes your code slower, more complex, and more prone to human error; in fact, it skips the compile-time type-checking.

## The TComponent Class

If the `TPersistent` class is more important than it seems at first sight, the key class at the heart of Delphi's component-based class library is `TComponent`, which inherits from `TPersistent` (and from `TObject`). The `TComponent` class defines many core elements of components; however, it is not as complex as you might think, because the base classes and the language already provide most of what's needed.

I won't explore all the details of the `TComponent` class, some of which are more important for component designers than they are for component users. I'll just discuss ownership (which accounts for some public properties of the class) and the two published properties of the class, `Name` and `Tag`.

### Ownership

One of the core features of the `TComponent` class is the definition of ownership. When a component is created, it can be assigned an owner component, which will be responsible for destroying it. So, every component can have an owner and can also be the owner of other components.

Several public methods and properties of the class are devoted to handling the *two sides* of ownership. Here is a list, extracted from the class declaration (in the `Classes` unit of the `VCL`):

```
type
  TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
  public
    constructor Create(AOwner: TComponent); virtual;
    procedure DestroyComponents;
    function FindComponent(const AName: string): TComponent;
    procedure InsertComponent(AComponent: TComponent);
    procedure RemoveComponent(AComponent: TComponent);

    property Components[Index: Integer]: TComponent read GetComponent;
    property ComponentCount: Integer read GetComponentCount;
    property ComponentIndex: Integer
      read GetComponentIndex write SetComponentIndex;
    property Owner: TComponent read FOwner;
```

If you create a component and give it an owner, it will be added to the list of components (`InsertComponent`), which is accessible using the `Components` array property. The specific component has an `Owner` and knows its position in the owner components list, with the `ComponentIndex` property. Finally, the owner's destructor will take care of the destruction of the object it owns by calling `DestroyComponents`. A few more protected methods are involved, but this should give you the overall picture.

It's important to emphasize that component ownership can solve many of your applications' memory management problems, if used properly. When you use the Form Designer or Data Module Designer of the IDE, that form or data module will own any component dropped on it. At the same time, you should generally create components with a form or data module owner, even in your code. In these circumstances you only need to remember to destroy the component containers (form or data module) when they are not needed anymore, and you can forget about the components they contain. For example, you delete a form to destroy all the components it contains at once, which is a major simplification compared to having to remember to free each and every object individually. At a larger scale, forms and data modules are generally owned by the `Application` object, which is destroyed by the `VCL` shutdown code freeing all of the component containers, which free the components they contain.

### The Components Array

The `Components` property can also be used to access one component owned by another—let's say, a form. This property can be very handy (compared to using a specific component directly) for writing generic code, acting on all or

many components at a time. For example, you can use the following code to add to a list box the names of all a form's components (this code is part of the [ChangeOwner](#) example presented in the [next section](#)):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do
    ListBox1.Items.Add (Components [I].Name);
end;
```

This code uses the `ComponentCount` property, which holds the total number of components owned by the current form, and the `Components` property, which is the list of owned components. When you access a value from this list, you get a value of the `TComponent` type. For this reason, you can directly use only the properties common to all components, such as the `Name` property. To use properties specific to particular components, you have to use the proper type-downcast (`as`).

**Note** In Delphi, some components are also component containers: the `GroupBox`, `Panel`, `PageControl`, and, of course, `Form` components. When you use these controls, you can add other components inside them. In this case, the container is the parent of the components (as indicated by the `Parent` property), and the form is their owner (as indicated by the `Owner` property). You can use the `Controls` property of a form or group box to navigate the child controls, and you can use the `Components` property of the form to navigate all the owned components, regardless of their parent.

Using the `Components` property, you can always access each component of a form. If you need access to a specific component, however, instead of comparing each name with the name of the component you are looking for, you can let Delphi do this work by using the form's `FindComponent` method. This method simply scans the `Components` array looking for a name match. More information about the role of the `Name` property for a component is in the section "[The Name Property](#)."

## Changing the Owner

You have seen that almost every component has an owner. When a component is created at design time (or from the resulting DFM file), its owner will invariably be its form. When you create a component at run time, the owner is passed as a parameter to the `Create` constructor.

`Owner` is a read-only property, so you cannot change it. The owner is set at creation time and should generally not change during the lifetime of a component. To understand why you should not change a component's owner at design time nor freely change its name, read the following discussion. Be warned that the topic covered is not simple; if you're just beginning with Delphi, you might want to come back to this section at a later time.

To change the owner of a component, you can call the `InsertComponent` and `RemoveComponent` methods of the owner itself, passing the current component as parameter. However, you cannot apply these methods directly in a form's event handler, as I attempt to do here:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveComponent (Button1);
  Form2.InsertComponent (Button1);
end;
```

This code produces a memory access violation, because when you call `RemoveComponent`, Delphi disconnects the component from the form field (`Button1`), setting it to `nil`. (I talk more about form fields in the section "[Removing Form Fields](#).") The solution is to write a procedure like this:

```
procedure ChangeOwner (Component, NewOwner: TComponent);
begin
```

```

Component.Owner.RemoveComponent (Component);
NewOwner.InsertComponent (Component);
end;

```

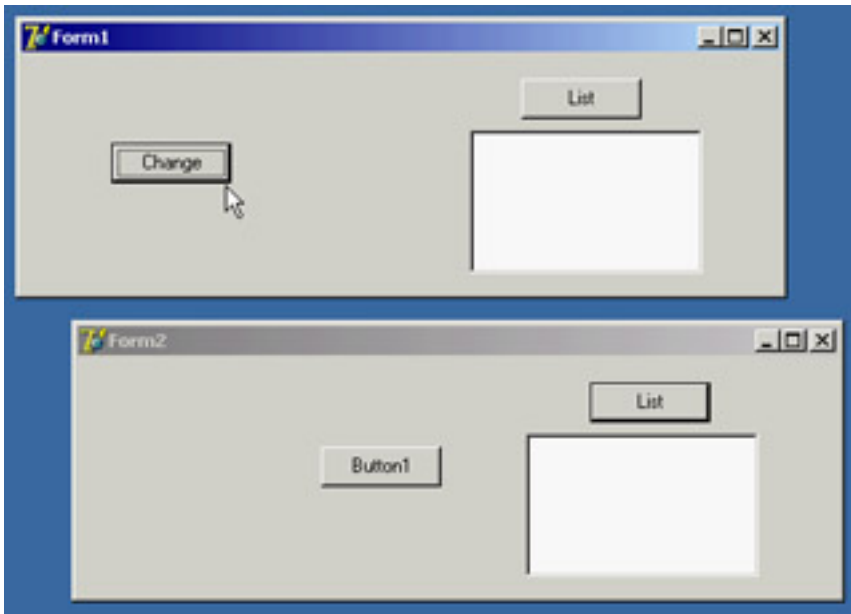
This method (extracted from the `ChangeOwner` example) changes the owner of the component. It is called along with the simpler code used to change the parent component; the two commands combined move the button *completely* to another form, changing its owner:

```

procedure TForm1.ButtonChangeClick(Sender: TObject);
begin
    if Assigned (Button1) then
        begin
            // change parent
            Button1.Parent := Form2;
            // change owner
            ChangeOwner (Button1, Form2);
        end;
end;

```

The method checks whether the `Button1` field still refers to the control, because while moving the component, Delphi will set `Button1` to `nil`. You can see the effect of this code in [Figure 4.3](#).



**Figure 4.3:** In the `ChangeOwner` example, clicking the `Change` button moves the `Button1` component to the second form.

To demonstrate that the owner of the `Button1` component actually changes, I've added another feature to both forms. The `List` button fills the list box with the names of the components each form owns, using the procedure shown in the [previous section](#). Click the two `List` buttons before and after moving the component, and you'll see what happens behind the scenes. As a final feature, the `Button1` component has a simple handler for its `OnClick` event, to display the caption of the owner form:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage ('My owner is ' + ((Sender as TButton).Owner as TForm).Caption);
end;

```

## The Name Property

Every component in Delphi should have a name. The name must be unique within the owner component, which is

generally the form into which you place the component. This means an application can have two different forms, each with a component that has the same name, although you might want to avoid this practice to prevent confusion. It is generally better to keep component names unique throughout an application.

Setting a proper value for the `Name` property is very important: If it's too long, you'll need to type a lot of code to use the object; if it's too short, you may confuse different objects. Usually the name of a component has a prefix with the component type; this makes the code more readable and allows Delphi to group components in the combo box of the Object Inspector, where they are sorted by name.

Three important elements are related to a component's `Name` property:

- At design time, the value of the `Name` property is used to define the name of the form field in the declaration of the form class. This is the name you'll generally use in the code to refer to the object. For this reason, the value of the `Name` property must be a legal Delphi language identifier (it must have no spaces and begin with a letter, not a number).
- If you set a control's `Name` property before changing its `Caption` or `Text` property, the new name is often copied to the caption. That is, if the name and the caption are identical, then changing the name will also change the caption.
- Delphi uses the name of the component to create the default name of the methods related to its events. If you have a `Button1` component, its default `OnClick` event handler will be called `Button1Click` unless you specify a different name. If you later change the name of the component, Delphi will modify the names of the related methods accordingly. For example, if you change the name of the button to `MyButton`, the `Button1Click` method automatically becomes `MyButtonClick`.

As mentioned earlier, if you have a string with the name of a component, you can get its instance by calling the `FindComponent` method of its owner, which returns `nil` if the component is not found. For example, you can write

```
var
  Comp: TComponent;
begin
  Comp := FindComponent ('Button1');
  if Assigned (Comp) then
    with Comp as TButton do
      // some code...
```

**Note** Delphi also includes a `FindGlobalComponent` function, which finds a top-level component (a form or data module) that has a given name. `FindGlobalComponent` calls one or more installed functions, so in theory you can modify the way the function works. However, because `FindGlobalComponent` is used by the streaming system, I strongly recommend against installing your own replacement functions. If you want a customized way to search for components on other containers, simply write a new function with a custom name.

## Removing Form Fields

Every time you add a component to a form, Delphi adds an entry for it, along with some of its properties, to the DFM file. To the Pascal file, Delphi adds the corresponding field in the form class declaration. This field of the form is a reference to the corresponding object, as is any class-type variable in Delphi. When the form is created, Delphi loads the DFM file and uses it to re-create all the components and set their properties back to the design-time values, saved in the DFM file itself. Then it connects the new object with the form field corresponding to its `Name` property. This is why in your code, you can use the form field to operate on the corresponding component.

For this reason, it is possible to have a component without a name. If your application will not manipulate the component or modify it at run time, you can remove the component name from the Object Inspector. Examples include a static label with fixed text, or a menu item, or even more obviously, menu item separators. By blanking out the name, you remove the corresponding element from the form class declaration. Doing so reduces the size of the form object

(by only four bytes, the size of the object reference) and reduces the DFM file by not including a useless string (the component name). Reducing the DFM file size also implies reducing the final executable file size, even if only slightly.

**Warning** If you remove component names, just make sure to leave at least one named component of each class used on the form, so the smart linker and the streaming system will link in the required code for the class and recognize it from the DFM file. For example, if you remove from a form all the fields referring to *TLabel* components, when the system loads the form at run time, it will be unable to create an object of an unknown class and will issue an error indicating that the class is not available. As we'll see in the [next section](#) you can call the *RegisterClass* or *RegisterClasses* routines to avoid such an error.

You can also keep the component name and manually remove the corresponding field of the form class. Even if the component has no corresponding form field, it is created anyway, although using it (through the `FindComponent` method, for example) will be a little more difficult.

## Hiding Form Fields

Many OOP purists complain that Delphi doesn't really follow the encapsulation rules, because all the components of a form are mapped to public fields and can be accessed from other forms and units. Fields for components are listed in the first unnamed section of a class declaration, which has a default visibility of published. However, Delphi does that only as a default to help beginners learn to use the Delphi visual development environment quickly. A programmer can follow a different approach and use properties and methods to operate on forms. The risk, however, is that another programmer on the same team might inadvertently bypass this approach, directly accessing the components if they are left in the published section. The solution, which many programmers don't know about, is to move the components to the private portion of the class declaration.

As an example, I've made a simple form with an edit box, a button, and a list box. When the edit box contains text and the user clicks the button, the text is added to the list box. When the edit box is empty, the button is disabled. This is the code of the HideComp example:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ListBox1.Items.Add (Edit1.Text);  
end;
```

```
procedure TForm1.Edit1Change(Sender: TObject);  
begin  
    Button1.Enabled := Length (Edit1.Text) <> 0;  
end;
```

I've listed these methods only to show you that in a form's code, you usually refer to the available components, defining their interactions. For this reason, it seems impossible to get rid of the fields corresponding to the component. However, you can hide them, moving them from the default published section to the private section of the form class declaration:

```
TForm1 = class(TForm)  
    procedure Button1Click(Sender: TObject);  
    procedure Edit1Change(Sender: TObject);  
    procedure FormCreate(Sender: TObject);  
private  
    Button1: TButton;  
    Edit1: TEdit;  
    ListBox1: TListBox;  
end;
```

Now, if you run the program you'll get in trouble: The form will load, but because the private fields are not initialized, the events will use `nil` object references. Delphi usually initializes the published fields of the form using the components created from the DFM file. What if you do it yourself, with the following code?

```
procedure TForm1.FormCreate(Sender: TObject);
```

**begin**

```
Button1 := FindComponent ('Button1') as TButton;  
Edit1 := FindComponent ('Edit1') as TEdit;  
ListBox1 := FindComponent ('ListBox1') as TListBox;
```

**end;**

It will *almost* work, but it generates a system error, similar to the one discussed in the [previous section](#). This time, the private declarations will cause the linker to link in the implementations of those classes; the problem is, the streaming system needs to know the names of the classes in order to locate the class reference needed to construct the components while loading the DFM file.

The final touch you need is registration code to tell Delphi at run time about the existence of the component classes you want to use. You should do this before the form is created, so I generally place this code in the initialization section of the unit:

**initialization**

```
RegisterClasses ([TButton, TEdit, TListBox]);
```

The question is, is this really worth the effort? You obtain a higher degree of encapsulation, protecting the components of a form from other forms (and other programmers writing them). Replicating these steps for every form can be tedious, so I wrote a wizard to generate the code for me on the fly. The wizard is far from perfect, because it doesn't handle changes automatically, but it is usable. See [Appendix A](#), "Extra Delphi Tools by the Author," for more information about how to get it. For a large project built according to the principles of object-oriented programming, I recommend you consider this or a similar technique.

## The Customizable Tag Property

The `Tag` property is strange, because it has no effect at all. It is merely an extra memory location, present in each component class, where you can store custom values. The kind of information stored and the way it is used are completely up to you.

It is often useful to have an extra memory location to attach information to a component without needing to define it in your component class. Technically, the `Tag` property stores a long integer so that, for example, you can store the entry number of an array or list that corresponds to an object. Using typecasting, you can store in the `Tag` property a pointer, an object reference, or anything else that is four bytes wide. A programmer can associate virtually anything with a component using its tag. You'll see how to use this property in several examples in future chapters.

## Events

Now that I've discussed the `TComponent` class, I need to introduce one more element of Delphi. Delphi components are programmed using PME: properties, methods, and events. Methods and properties should be clear by now, but you haven't yet learned about events. The reason is that events don't imply a new language feature but are simply a standard coding technique. An event is technically a property—the only difference is that it refers to a method (a method pointer type, to be precise) instead of other types of data.

## Events in Delphi

When a user does something with a component, such as click it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding operating system message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than operating system messages, and Delphi provides a number of extra inter-component messages.



From a theoretical point of view, an event is the result of a request sent to a component or control, which can respond to the message. Following this approach, to handle the click event of a button, you would need to subclass the `TButton` class and add the new event handler code inside the new class.

In practice, creating a new class for every component you want to use is too complex to be a reasonable solution. In Delphi, a component's event handler usually is a method of the form that holds the component, not of the component itself. In other words, the component relies on its owner, the form, to handle its events. This technique is called *delegation*, and it is fundamental to the Delphi component-based model. This way, you don't have to modify the `TButton` class, unless you want to define a new type of component, but can simply customize its owner to modify the behavior of the button.

**Note** As you'll see in the [next section](#), events in Delphi are based on pointers to methods. This is quite different from Java, which uses *listener* classes with methods for a family of events. These listener methods call the event handlers. C# and .NET use the similar idea of *delegate* classes, covered in [Chapter 24](#), "The Microsoft .NET Architecture from the Delphi Perspective." Notice that the term *delegate* is the same traditionally used in the Delphi literature to explain the idea of event handlers.

## Method Pointers

Events rely on a specific feature of the Delphi language: *method pointers*. A method pointer type is like a procedural type, but one that refers to a method. Technically, a method pointer type is a procedural type that has an implicit `Self` parameter. In other words, a variable of a procedural type stores the address of a function to call, provided it has a given set of parameters. A method pointer variable stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as `Self` inside the method body when the method code is called using this method pointer.

**Note** This explains the definition of Delphi's generic `TMethod` type, a record with a `Code` field and a `Data` field.

The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords of `object` at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

When you have declared such a method pointer type, you can declare a variable of this type and assign to it a compatible method—a method that has the same signature (parameters, return type, calling convention)—of another object.

When you add an `OnClick` event handler for a button, Delphi does exactly that. The button has a method pointer type property, named `OnClick`, and you can directly or indirectly assign to it a method of another object, such as a form. When a user clicks the button, this method is executed, even if you have defined it inside another class.

What follows is a sketch of the code Delphi uses to define the event handler of a button component and the related method of a form:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;
```

```
MyButton = class
  OnClick: TNotifyEvent;
end;
```

```
TForm1 = class (TForm)
  procedure Button1Click (Sender: TObject);
  Button1: MyButton;
end;
```



```
var
    Form1: TForm1;
```

Now, inside a procedure, you can write

```
MyButton.OnClick := Form1.Button1Click;
```

The only real difference between this code fragment and the VCL code is that `OnClick` is a property name, and the data it refers to is called `FOnClick`. An event that shows up in the Events page of the Object Inspector is nothing more than a property of a method pointer type. This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it.

## Events Are Properties

I've already mentioned that events are properties. To handle an event of a component, you assign a method to the corresponding event property. When you double-click an event value in the Object Inspector, a new method is added to the owner form and assigned to the proper event property of the component.

It is possible for several events to share the same event handler or change an event handler at run time. To use this feature, you don't need much knowledge of the language. In fact, when you select an event in the Object Inspector, you can click the arrow button to the right of the event name to see a drop-down list of compatible methods—methods having the same signature of the method pointer type. Using the Object Inspector, it is easy to select the same method for the same event of different components or for different, compatible events of the same component.

Just as you added some properties to the `TDate` class in [Chapter 2](#), you can add one event. The event will be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the date value has changed. To define an event, you simply define a property corresponding to it and add some data to store the method pointer the event refers to. These are the new definitions added to the class, available in the `DateEvt` example:

```
type
    TDate = class
    private
        FOnChange: TNotifyEvent;
        ...
    protected
        procedure DoChange; dynamic;
        ...
    public
        property OnChange: TNotifyEvent
            read FOnChange write FOnChange;
        ...
    end;
```

The property definition is simple. A user of this class can assign a new value to the property and, hence, to the `FOnChange` private field. The class doesn't assign a value to this `FOnChange` field; the user of the component does the assignment. The `TDate` class simply calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned. The `DoChange` method (declared as a dynamic method as is traditional with event-firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
    if Assigned (FOnChange) then
        FOnChange (Self);
end;
```

The `DoChange` method in turn is called every time one of the values changes, as in the following method:

```
procedure TDate.SetValue (y, m, d: Integer);
begin
```

```
fDate := EncodeDate (y, m, d);  
// fire the event  
DoChange;
```

If you look at the program that uses this class, you can simplify its code considerably. First, add a new custom method to the form class:

```
type  
TDateForm = class(TForm)  
    ...  
    procedure DateChange(Sender: TObject);
```

The method's code simply updates the label with the current value of the TDate object's Text property:

```
procedure TDateForm.DateChange;  
begin  
    LabelDate.Caption := TheDay.Text;  
end;
```

This event handler is then installed in the FormCreate method:

```
procedure TDateForm.FormCreate(Sender: TObject);  
begin  
    TheDay := TDate.Init (2003, 7, 4);  
    LabelDate.Caption := TheDay.Text;  
    // assign the event handler for future changes  
    TheDay.OnChange := DateChange;  
end;
```

This seems like a lot of work. Was I lying when I told you the event handler would save you some coding? No. Now, after you've added some code, you can forget about updating the label when you change some of the object data. For example, here is the handler of the OnClick event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);  
begin  
    TheDay.Increase;  
end;
```

The same simplified code is present in many other event handlers. Once you have installed the event handler, you don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that you had to write some code at the beginning because this is not a component installed in Delphi but simply a class. With a component, you select the event handler in the Object Inspector and write a single line of code to update the label—that's all.

**Note** This is just a short introduction to defining events. A basic understanding of these features is important for every Delphi programmer. If your aim is to write new components with complex events, you'll find a lot more information on all these topics in [Chapter 9](#) ("Writing Delphi Components").

## Lists and Container Classes

It is often important to handle groups of components or objects. In addition to using standard arrays and dynamic arrays, a few VCL classes represent lists of other objects. These classes can be divided into three groups: simple lists, collections, and containers.

### Lists and String Lists

Lists are represented by the generic list of objects, `TList`, and by the two lists of strings, `TStrings` and `TStringList`:

- `TList` defines a list of pointers, which can be used to store objects of any class. A `TList` is more flexible than a dynamic array, because it can be expanded automatically simply by adding new items to it. The advantage of a dynamic array over a `TList` is that the dynamic array allows you to indicate a specific type for contained objects and perform the proper compile-time type checking.
- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves, such as a list box.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with their own storage. You can use this class to define a list of strings in a program.

`TStringList` and `TStrings` objects have both a list of strings and a list of objects associated with the strings. These classes have a number of different uses. For example, you can use them for dictionaries of associated objects or to store bitmaps or other elements to be used in a list box.

The two classes of string lists also have ready-to-use methods to store or load their contents to or from a text file: `SaveToFile` and `LoadFromFile`. To loop through a list, you can use a simple for statement based on its index, as if the list were an array.

All these lists have a number of methods and properties. You can operate on lists using the array notation (`[` and `]`) both to read and to change elements. There is a `Count` property, as well as typical access methods, such as `Add`, `Insert`, `Delete`, `Remove`; search methods (for example, `IndexOf`); and sorting support. The `TList` class has an `Assign` method that, besides copying the source data, can perform set operations on the two lists, including *and*, *or*, and *xor*.

To fill a string list with items and later check whether one is present, you can write code like this:

```
var
  sl: TStringList;
  idx: Integer;
begin
  sl := TStringList.Create;
  try
    sl.Add ('one');
    sl.Add ('two');
    sl.Add ('three');
    // later
    idx := sl.IndexOf ('two');
    if idx >= 0 then
      ShowMessage ('String found');
  finally
    sl.Free;
  end;
```

```
end;
```

## Name-Value Pairs (and Delphi 7 Extensions)

The `TStringList` class has always had another nice feature: support for name-value pairs. If you add to a list a string like 'lastname=john', you can then search for the existence of the pair using the `IndexOfName` function or the `Values` array property. For example, you can retrieve the value 'john' by calling `Values ['lastname']`.

You can use this feature to build much more complex data structures, such as dictionaries, and still benefit from the possibility of attaching an object to the string. This data structure maps directly to initialization files and other common formats.

Delphi 7 further extends the possibilities of name-value pair support by allowing you to customize the separator, beyond the equal sign, using the new `NameValueSeparator` property. In addition, the new `ValueFromIndex` property gives you direct access to the value portion of a string at a given position; you no longer have to extract the name value manually from the complete string using a cumbersome (and extremely slow) expression:

```
str := MyStringList.Values [MyStringList.Names [I]]; // old  
str := MyStringList.ValueFromIndex [I]; // new
```

## Using Lists of Objects

I've written an example focusing on the use of the generic `TList` class. When you need a list of any kind of data, you can generally declare a `TList` object, fill it with the data, and then access the data while casting it to the proper type. The `ListDemo` example demonstrates this approach and also shows its pitfalls. The form has a private variable, holding a list of dates:

```
private  
    ListDate: TList;
```

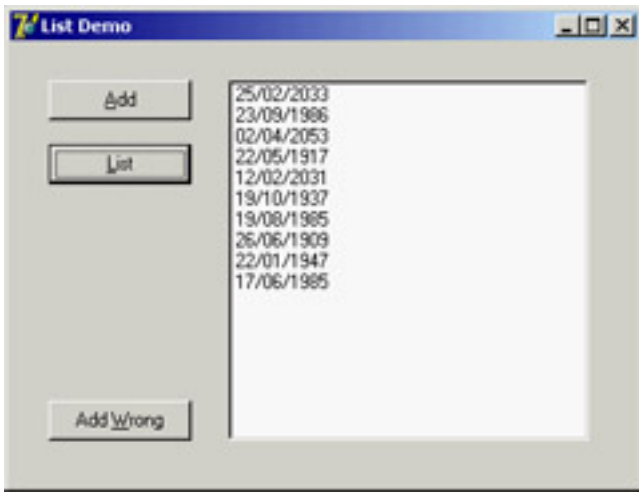
This list object is created when the form itself is created:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Randomize;  
    ListDate := TList.Create;  
end;
```

A button on the form adds a random date to the list (of course, I've included in the project the unit containing the date component built in the [previous chapter](#)):

```
procedure TForm1.ButtonAddClick(Sender: TObject);  
begin  
    ListDate.Add (TDate.Create (1900 + Random (200), 1 + Random (12),  
        1 + Random (30)));  
end;
```

When you extract the items from the list, you have to cast them back to the proper type, as in the following method, which is connected to the `List` button (you can see its effect in [Figure 4.4](#)):



**Figure 4.4:** The list of dates shown by the ListDemo example

```

procedure TForm1.ButtonListDateClick(Sender: TObject);
var
    I: Integer;
begin
    ListBox1.Clear;
    for I := 0 to ListDate.Count - 1 do
        ListBox1.Items.Add ((TObject(ListDate [I]) as TDate).Text);
end;

```

At the end of the code, before you can do an `as` downcast, you first need to hard-cast the pointer returned by the `TList` into a `TObject` reference. This kind of expression can result in an invalid typecast exception, or it can generate a memory error when the pointer is not a reference to an object.

**Tip** If there were no possibility of having anything but date objects in the list, extracting it with a static cast rather than an `as` cast would be more efficient. However, when there's even a remote chance of having a wrong object, I'd suggest using the `as` cast.

To demonstrate that things can indeed go wrong, I've added one more button, which adds a `TButton` object to the list by calling `ListDate.Add(Sender)`. If you click this button and then update one of the lists, you'll get an error. Finally, remember that when you destroy a list of objects, you should destroy all of the list's objects first. The ListDemo program does this in the form's `FormDestroy` method:

```

procedure TForm1.FormDestroy(Sender: TObject);
var
    I: Integer;
begin
    for I := 0 to ListDate.Count - 1 do
        TObject(ListDate [I]).Free;
    ListDate.Free;
end;

```

## Collections

The second group, collections, contains only two VCL classes: `TCollection` and `TCollectionItem`. `TCollection` defines a homogeneous list of objects, which are owned by the collection class. The objects in the collection must be descendants of the `TCollectionItem` class. If you need a collection storing specific objects, you have to create both a subclass of `TCollection` and a matching subclass of `TCollectionItem`.

Collections are used to specify values of component properties; it is very unusual to work with collections to store your own objects. I discuss collections in [Chapter 9](#).

## Container Classes

Recent versions of Delphi include a series of container classes, defined in the `Contnrs` unit. The container classes extend the `TList` classes by adding the idea of ownership and by defining specific extraction rules (mimicking stacks and queues) or sorting capabilities.

The basic difference between `TList` and the new `TObjectList` class, for example, is that the latter is defined as a list of `TObject` objects, not a list of pointers. Even more important, however, is the fact that if the object list has the `OwnsObjects` property set to `True`, it automatically deletes an object when it is replaced by another one and deletes each object when the list itself is destroyed. Here's a list of the container classes:

- The `TObjectList` class (already described) represents a list of objects, eventually owned by the list itself.
- The inherited class `TComponentList` represents a list of components, with full support for destruction notification (an important safety feature when two components are connected using their properties; that is, when a component is the value of a property of another component).
- The `TClassList` class is a list of class references. It inherits from `TList` and requires no specific destruction, as you don't have to destroy class references in Delphi.
- The classes `TStack` and `TObjectStack` represent lists of pointers and objects, from which you can only extract elements starting from the last one you've inserted. A stack follows LIFO order (last in, first out). The typical methods of a stack are `Push` for insertion, `Pop` for extraction, and `Peek` to preview the first item without removing it. You can still use all the methods of the base class, `TList`.
- The classes `TQueue` and `TObjectQueue` represent lists of pointers and objects, from which you always remove the *first* item you've inserted (FIFO: first in, first out). The methods of these classes are the same as those of the stack classes, but they behave differently.

**Warning** Unlike `TObjectList`, the `TObjectStack` and `TObjectQueue` classes do not own the inserted objects and will not destroy those objects left in the data structure when it is destroyed. You can simply `Pop` all the items, destroy them once you're finished using them, and then destroy the container.

To demonstrate the use of these classes, I've modified the earlier `ListDate` example into the `Contain` example. First, I changed the type of the `ListDate` variable to `TObjectList`. In the `FormCreate` method, I've modified the list creation to the following code, which activates the list ownership:

```
ListDate := TObjectList.Create (True);
```

At this point, you can simplify the destruction code, because applying `Free` to the list will automatically free the dates it holds.

I've also added to the program a stack and a queue object, filling each of them with numbers. One of the form's two buttons displays a list of the numbers in each container, and the other removes the last item (displayed in a message box):

```
procedure TForm1.btnQueueClick(Sender: TObject);  
var  
    I: Integer;  
begin  
    ListBox1.Clear;  
    for I := 0 to Stack.Count - 1 do begin  
        ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));  
        Queue.Push(Queue.Pop);  
    end;  
    ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));  
end;
```

By clicking the two buttons, you can see that calling `Pop` for each container returns the last item. The difference is that the `TQueue` class inserts elements at the beginning, and the `TStack` class inserts them at the end.

## Hashed Associative Lists

Since Delphi 6, the set of predefined container classes includes `TBucketList` and `TObjectBucketList`. These two lists are associative, which means they have a key and an actual entry. The key is used to identify the items and search for them. To add an item, you call the `Add` method with two parameters: the key and the data. When you use the `Find` method, you pass the key and retrieve the data. The same effect is achieved by using the `Data` array property, passing the key as parameter.

These lists are based on a hash system. The lists create an internal array of items, called *buckets*, each having a sublist of list elements. As you add an item, its key value is used to compute the *hash* value, which determines the bucket to which the item should be added. When searching the item, the hash is computed again, and the list immediately grabs the sublist containing the item and searches for it there. This process makes for very fast insertion and searches, but only if the hash algorithm distributes the items evenly among the various buckets and if there are enough different entries in the array. When many elements can be in the same bucket, searching is slower.

For this reason, as you create the `TObjectBucketList`, you can specify the number of entries for the list by using the parameter of the constructor and choosing a value between 2 and 256. The value of the bucket is determined by taking the first byte of the pointer (or number) passed as the key and doing an `and` operation with a number corresponding to the entries.

**Note** I don't find this algorithm very convincing for a hash system, but replacing it with your own implies only overriding the `BucketFor` virtual function and eventually changing the number of entries in the array, by setting a different value for the `BucketCount` property.

Another interesting feature, not available for lists, is the `ForEach` method, which allows you to execute a given function on each item contained in the list. You pass the `ForEach` method a pointer to your data and a procedure that receives four parameters: your custom pointer, each key and object of the list, and a Boolean parameter you can set to `False` to stop the execution. In other words, these are the two signatures:

### type

```
TBucketProc = procedure(AInfo, AItem, AData: Pointer;  
    out AContinue: Boolean);
```

```
function TCustomBucketList.ForEach(AProc: TBucketProc;  
    AInfo: Pointer): Boolean;
```

**Note** In addition to these containers, Delphi includes a `THashedStringList` class, which inherits from `TStringList`. This class has no direct relationship with the hashed lists and is defined in a different unit, `IniFiles`. The hashed string list has two associated hash tables (of type `TStringHash`), which are completely refreshed every time the content of the string list changes. So, this class is useful only for reading a large set of fixed strings, not for handling a list of strings changing often over time. On the other hand, the `TStringHash` support class seems to be quite useful in general cases, and has a good algorithm for computing the hash value of a string.

## Type-Safe Containers and Lists

Containers and lists have a problem: They are not type-safe, as I've shown in both examples by adding a button object to a list of dates. To ensure that the data in a list is homogenous, you can check the type of the data you extract before you insert it, but as an extra safety measure you might also want to check the type of the data while extracting it. However, adding run-time type checking slows a program and is risky—a programmer might fail to check the type in some cases.

To solve both problems, you can create specific list classes for given data types and fashion the code from the existing `TList` or `TObjectList` class (or another container class). There are two approaches to accomplish this:



- Derive a new class from the list class and customize the `Add` method and the access methods, which relate to the `Items` property. This is also the approach used by Borland for the container classes, which all derive from `TList`.

**Note** Delphi container classes use static overrides to perform simple type conveniences (parameters and function results of the desired type). Static overrides are not the same as polymorphism; someone using a container class via a `TList` variable will not be calling the container's specialized functions. Static override is a simple and effective technique, but it has one very important restriction: The methods in the descendent should not do anything beyond simple typecasting, because you have no guarantee the descendent methods will be called. The list might be accessed and manipulated using the ancestor methods as much as by the descendent methods, so their operations must be identical. The only difference is the type used in the descendent methods, which allows you to avoid extra typecasting.

- Create a brand-new class that contains a `TList` object, and map the methods of the new class to the internal list using proper type checking. This approach defines a wrapper class, a class that "wraps" around an existing one to provide a different or limited access to its methods (in our case, to perform a type conversion).

I've implemented both solutions in the `DateList` example, which defines lists of `TDate` objects. In the code that follows, you'll find the declaration of the two classes, the inheritance-based `TDateListI` class and the wrapper class `TDateListW`:

```
type
// inheritance-based
TDateListI = class (TObjectList)
protected
    procedure SetObject (Index: Integer; Item: TDate);
    function GetObject (Index: Integer): TDate;
public
    function Add (Obj: TDate): Integer;
    procedure Insert (Index: Integer; Obj: TDate);
    property Objects [Index: Integer]: TDate
        read GetObject write SetObject; default;
end;

// wrapper based
TDateListW = class(TObject)
private
    FList: TObjectList;
    function GetObject (Index: Integer): TDate;
    procedure SetObject (Index: Integer; Obj: TDate);
    function GetCount: Integer;
public
    constructor Create;
    destructor Destroy; override;
    function Add (Obj: TDate): Integer;
    function Remove (Obj: TDate): Integer;
    function IndexOf (Obj: TDate): Integer;
    property Count: Integer read GetCount;
    property Objects [Index: Integer]: TDate
        read GetObject write SetObject; default;
end;
```

Obviously, the first class is simpler to write—it has fewer methods, and they just call the inherited ones. The good thing is that a `TDateListI` object can be passed to parameters expecting a `TList`. The problem is that the code that manipulates an instance of this list via a generic `TList` variable will not be calling the specialized methods, because they are not virtual and might end up adding to the list objects of other data types.



Instead, if you decide not to use inheritance, you end up writing a lot of code; you need to reproduce every one of the original `TList` methods, simply calling the methods of the internal `FList` object. The drawback is that the `TDateListW` class is not type compatible with `TList`, which limits its usefulness. It can't be passed as parameter to methods expecting a `TList`.

Both of these approaches provide good type checking. After you've created an instance of one of these list classes, you can add only objects of the appropriate type, and the objects you extract will naturally be of the correct type. This technique is demonstrated by the `DateList` example. This program has a few buttons, a combo box to let a user choose which of the lists to show, and a list box to show the list values. The program stretches the lists by trying to add a button to the list of `TDate` objects. To add an object of a different type to the `TDateListI` list, you can convert the list to its base class, `TList`. This might happen accidentally if you pass the list as a parameter to a method that expects an ancestor class of the list class. In contrast, for the `TDateListW` list to fail, you must explicitly cast the object to `TDate` before inserting it, something a programmer should never do:

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);  
begin  
    ListW.Add (TDate(TButton.Create (nil)));  
    TList(ListI).Add (TButton.Create (nil));  
    UpdateList;  
end;
```

The `UpdateList` call triggers an exception, displayed directly in the list box, because I've used an `as` typecast in the custom list classes. A wise programmer should never write the previous code. To summarize, writing a custom list for a specific type makes a program much more robust. Writing a wrapper list instead of one that's based on inheritance tends to be a little safer, although it requires more coding.

**Note** Instead of rewriting wrapper-style list classes for different types, you can use my List Template Wizard. See [Appendix A](#) for details.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Streaming

Another core area of the Delphi class library is its support for streaming, which includes file management, memory, sockets, and other sources of information arranged in a sequence. The idea of streaming is that you move through the data while reading it, much like the `Read` and `Write` functions traditionally used by the Pascal language (and discussed in Chapter 12 of *Essential Pascal* (see [Appendix C](#) for availability of this e-book).

### The *TStream* Class

The VCL defines the abstract `TStream` class and several subclasses. The parent class, `TStream`, has just a few properties, and you'll never create an instance of it, but it has an interesting list of methods you'll generally use when working with derived stream classes.

The `TStream` class defines two properties, `Size` and `Position`. All stream objects have a specific size (which generally grows if you write something after the end of the stream), and you must specify a position within the stream where you want to either read or write information.

Reading and writing bytes depends on the actual stream class you are using, but in both cases you don't need to know much more than the size of the stream and your relative position in the stream to read or write data. In fact, that's one of the advantages of using streams. The basic interface remains the same whether you're manipulating a disk file, a binary large object (BLOB) field, or a long sequence of bytes in memory.

In addition to the `Size` and `Position` properties, the `TStream` class also defines several important methods, most of which are virtual and abstract. (In other words, the `TStream` class doesn't define what these methods do; therefore, derived classes are responsible for implementing them.) Some of these methods are important only in the context of reading or writing components within a stream (for instance, `ReadComponent` and `WriteComponent`), but some are useful in other contexts, too. In [Listing 4.2](#), you can find the declaration of the `TStream` class, extracted from the `Classes` unit.

#### Listing 4.2: The Public Portion of the Definition of the *TStream* Class

---

```
TStream = class(TObject)
public
    // read and write a buffer
    function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
    function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
    procedure ReadBuffer(var Buffer; Count: Longint);
    procedure WriteBuffer(const Buffer; Count: Longint);

    // move to a specific position
    function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;
    function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
        overload; virtual;

    // copy the stream
    function CopyFrom(Source: TStream; Count: Int64): Int64;

    // read or write a component
    function ReadComponent(Instance: TComponent): TComponent;
    function ReadComponentRes(Instance: TComponent): TComponent;
    procedure WriteComponent(Instance: TComponent);
    procedure WriteComponentRes(const ResName: string; Instance: TComponent);
    procedure WriteDescendent(Instance, Ancestor: TComponent);
    procedure WriteDescendentRes(
        const ResName: string; Instance, Ancestor: TComponent);
```

```

procedure WriteResourceHeader(const ResName: string; out FixupInfo: Integer);
procedure FixupResourceHeader(FixupInfo: Integer);
procedure ReadResHeader;

// properties
property Position: Int64 read GetPosition write SetPosition;
property Size: Int64 read GetSize write SetSize64;
end;

```

---

The basic use of a stream involves calling the `ReadBuffer` and `WriteBuffer` methods, which are very powerful but not terribly easy to use. The first parameter is an untyped buffer in which you can pass the variable to save from or load to. For example, you can save into a file a number (in binary format) and a string, with this code:

```

var
  stream: TStream;
  n: integer;
  str: string;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test', fmCreate);
  stream.WriteBuffer (n, sizeof(integer));
  stream.WriteBuffer (str[1], Length (str));
  stream.Free;

```

An alternative approach is to let specific components save or load data to and from streams. Many VCL classes define a `LoadFromStream` or a `SaveToStream` method, including `TStrings`, `TStringList`, `TBlobField`, `TMemoField`, `TIcon`, and `TBitmap`.

## Specific Stream Classes

Creating a `TStream` instance makes no sense, because this class is abstract and provides no direct support for saving data. Instead, you can use one of the derived classes to load data from or store it to an actual file, a BLOB field, a socket, or a memory block. Use `TFileStream` when you want to work with a file, passing the filename and some file access options to the `Create` method. Use `TMemoryStream` to manipulate a stream in memory and not an actual file.

Several units define `TStream`-derived classes. The `Classes` unit includes the following classes:

- `THandleStream` defines a stream that manipulates a disk file represented by a file handle.
- `TFileStream` defines a stream that manipulates a disk file (a file that exists on a local or network disk) represented by a filename. It inherits from `THandleStream`.
- `TCustomMemoryStream` is the base class for streams stored in memory but is not used directly.
- `TMemoryStream` defines a stream that manipulates a sequence of bytes in memory. It inherits from `TCustomMemoryStream`.
- `TStringStream` provides a simple way to associate a stream to a string in memory, so that you can access the string with the `TStream` interface and also copy the string to and from another stream.
- `TResourceStream` defines a stream that manipulates a sequence of bytes in memory, and provides read-only access to resource data linked into the executable file of an application (the DFM files are an example of this resource data). It inherits from `TCustomMemoryStream`.

Stream classes defined in other units include the following:

- `TBlobStream` defines a stream that provides simple access to database BLOB fields. There are similar BLOB streams for database access technologies other than the BDE, including `TSQLBlobStream` and `TClientBlobStream`. (Notice that each type of dataset uses a specific stream class for BLOB fields.) All these classes inherit from `TMemoryStream`.
- `TOLEStream` defines a stream for reading and writing information over the interface for streaming provided by an OLE object.
- `TWinSocketStream` provides streaming support for a socket connection.

## Using File Streams

Creating and using a file stream can be as simple as creating a variable of a type that descends from `TStream` and calling components' methods to load content from the file:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
  begin
    S := TFileStream.Create (OpenDialog1.FileName, fmOpenRead);
    try
      Mem1.Lines.LoadFromStream (S);
    finally
      S.Free;
    end;
  end;
end;
```

As you can see in this code, the `Create` method for file streams has two parameters: the name of the file and a flag indicating the requested access mode. In this case, you want to read the file, so you use the `fmOpenRead` flag (other available flags are documented in the Delphi help).

**Note** Of the different modes, the most important are `fmShareDenyWrite`, which you'll use when you're simply reading data from a shared file, and `fmShareExclusive`, which you'll use when you're writing data to a shared file. There is a third parameter in `TFileStream.Create`, called *Rights*. This parameter is used to pass file access permissions to the Linux filesystem when the access mode is `fmCreate` (that is, only when you are creating a new file). This parameter is ignored on Windows.

A big advantage of streams over other file access techniques is that they're very interchangeable, so you can work with memory streams and then save them to a file, or you can perform the opposite operations. This might be a way to improve the speed of a file-intensive program. Here is a snippet of a file-copying function to give you another idea of how you can use streams:

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create (SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create (TargetName, fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom (Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end
  end;
```

```
finally
  Stream1.Free;
end
end;
```

Another important use of streams is to handle database BLOB fields or other large fields directly. You can export such data to a stream or read it from one by calling the `SaveToStream` and `LoadFromStream` methods of the `TBlobField` class.

**Note** Delphi 7 streaming support adds a new exception base class, `EFileStreamError`. Its constructor takes as parameter a filename for error reporting. This class standardizes and largely simplifies the notification of file-related errors in streams.

## The *TReader* and *TWriter* Classes

By themselves, the VCL stream classes don't provide much support for reading or writing data. In fact, stream classes don't implement much beyond simply reading and writing blocks of data. If you want to load or save specific data types in a stream (and don't want to perform a great deal of typecasting), you can use the `TReader` and `TWriter` classes, which derive from the generic `TFileer` class.

Basically, the `TReader` and `TWriter` classes exist to simplify loading and saving stream data according to its type, and not just as a sequence of bytes. To do this, `TWriter` embeds special signatures into the stream that specify the type for each object's data. Conversely, the `TReader` class reads these signatures from the stream, creates the appropriate objects, and then initializes those objects using the subsequent data from the stream.

For example, I could have written out a number and a string to a stream by writing:

```
var
  stream: TStream;
  n: integer;
  str: string;
  w: TWriter;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test.txt', fmCreate);
  w := TWriter.Create (stream, 1024);
  w.WriteInteger (n);
  w.WriteString (str);
  w.Free;
  stream.Free;
```

This time the file will include the extra signature characters, so I can read back this file only by using a `TReader` object. For this reason, using `TReader` and `TWriter` is generally confined to component streaming and is seldom applied in general file management.

## Streams and Persistency

In Delphi, streams play a considerable role in persistency. For this reason, many methods of `TStream` relate to saving and loading a component and its subcomponents. For example, you can store a form in a stream by writing `stream.WriteComponent (Form1);`

If you examine the structure of a Delphi DFM file, you'll discover that it's really just a resource file that contains a custom format resource. Inside this resource, you'll find the component information for the form or data module and for each of the components it contains. As you would expect, the stream classes provide two methods to read and write this custom resource data for components: `WriteComponentRes` to store the data, and `ReadComponentRes` to load

it.

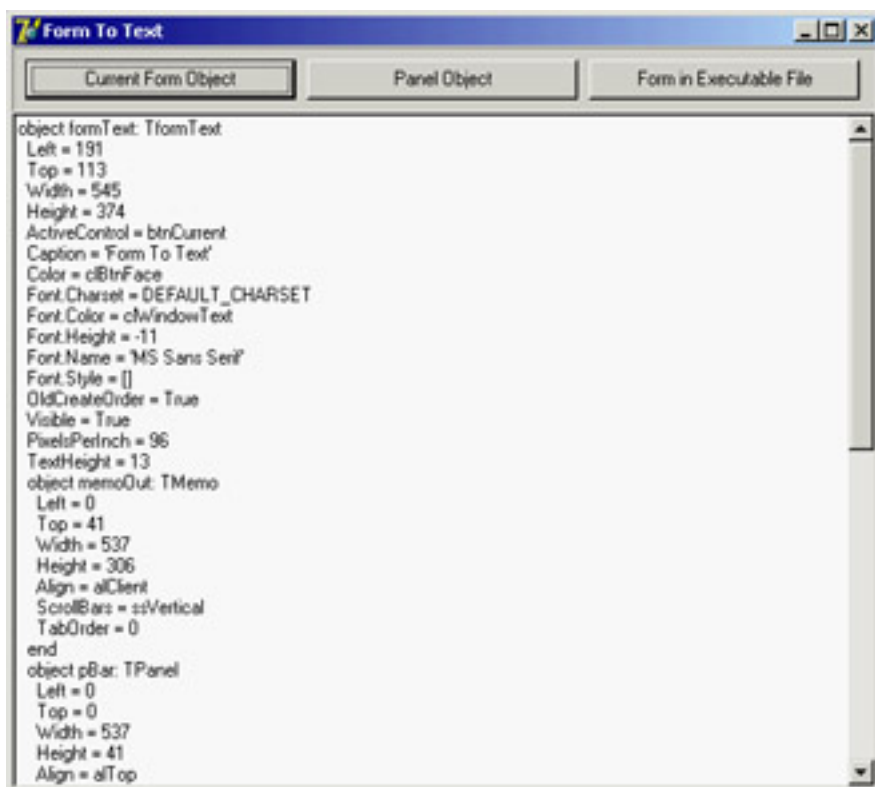
For your experiment in memory (not involving DFM files), though, using `WriteComponent` is generally better suited. After you create a memory stream and save the current form to it, the problem is how to display it. You can do this by transforming the form's binary representation to a textual representation. Even though the Delphi IDE, since version 5, can save DFM files in text format, the representation used internally for the compiled code is invariably a binary format.

The IDE can accomplish the form conversion, generally with the View as Text command of the Form Designer, and in other ways. The Delphi `Bin` directory also contains a command-line utility, `CONVERT.EXE`. Within your own code, the standard way to obtain a conversion is to call the specific VCL methods. There are four functions for converting to and from the internal object format obtained by the `WriteComponent` method:

```
procedure ObjectBinaryToText(Input, Output: TStream); overload;  
procedure ObjectBinaryToText(Input, Output: TStream;  
    var OriginalFormat: TStreamOriginalFormat); overload;  
procedure ObjectTextToBinary(Input, Output: TStream); overload;  
procedure ObjectTextToBinary(Input, Output: TStream;  
    var OriginalFormat: TStreamOriginalFormat); overload;
```

Four different functions, with the same parameters and names containing the name *Resource* instead of *Binary* (as in `ObjectResourceToText`), convert the resource format obtained by `WriteComponentRes`. A final method, `TestStreamFormat`, indicates whether a DFM is storing a binary or textual representation.

In the `FormToText` program, I've used the `ObjectBinaryToText` method to copy the binary definition of a form into another stream, and then I've displayed the resulting stream in a memo, as you can see in [Figure 4.5](#). Here is the code of the two methods involved:



**Figure 4.5:** The textual description of a form component, displayed inside itself by the `FormToText` example

```
procedure TFormText.btnCurrentClick(Sender: TObject);  
var  
    MemStr: TStream;  
begin  
    MemStr := TMemoryStream.Create;
```

```

try
  MemStr.WriteComponent (Self);
  ConvertAndShow (MemStr);
finally
  MemStr.Free
end;
end;

procedure TFormText.ConvertAndShow (aStream: TStream);
var
  ConvStream: TStream;
begin
  aStream.Position := 0;
  ConvStream := TMemoryStream.Create;
  try
    ObjectBinaryToText (aStream, ConvStream);
    ConvStream.Position := 0;
    MemoOut.Lines.LoadFromStream (ConvStream);
  finally
    ConvStream.Free
  end;
end;

```

Notice that by repeatedly clicking the Current Form Object button you'll get more and more text, and the text of the memo is included in the stream. After a few times, the entire operation will become extremely slow, until the program seems to be hung up. In this code, you see some of the flexibility of using streams—you can write a generic procedure that you can use to convert any stream.

**Note** It's important to stress that after you've written data to a stream, you must explicitly seek back to the beginning (or set the *Position* property to 0) before you can use the stream further—unless you want to append data to the stream, of course.

Another button, labeled Panel Object, shows the textual representation of a specific component, the panel, passing the component to the `WriteComponent` method. The third button, Form in Executable File, performs a different operation. Instead of streaming an existing object in memory, it loads in a `TResourceStream` object the design-time representation of the form—that is, its DFM file—from the corresponding resource embedded in the executable file:

```

procedure TFormText.btnResourceClick(Sender: TObject);
var
  ResStr: TResourceStream;
begin
  ResStr := TResourceStream.Create(hInstance, 'TFORMTEXT', RT_RCDATA);
  try
    ConvertAndShow (ResStr);
  finally
    ResStr.Free
  end;
end;

```

By clicking the buttons in sequence (or modifying the form of the program) you can compare the form saved in the DFM file to the current run-time object.

---

### Writing a Custom Stream Class

Besides using the existing stream classes, Delphi programmers can write their own stream classes and use them in place of the existing ones. To accomplish this, you need only specify how a generic block of raw data is saved and loaded, and the VCL will be able to use your new class wherever you call for it. You may not need to create a brand-new stream class to work with a new type of media, but only need to customize an existing stream. In that

case, all you have to do is write the proper read and write methods.

As an example, I created a class to encode and decode a generic file stream. Although this example is limited by its use of a totally dumb encoding mechanism, it fully integrates with the VCL and works properly. The new stream class simply declares the two core reading and writing methods and has a property that stores a key:

**type**

```
TEncodedStream = class (TFileStream)
private
    FKey: Char;
public
    constructor Create(const FileName: string; Mode: Word);
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    property Key: Char read FKey write FKey;
end;
```

The value of the key is added to each of the bytes saved to a file and subtracted when the data is read. Here is the complete code of the write and Read methods, which uses pointers quite heavily:

```
constructor TEncodedStream.Create( const FileName: string; Mode: Word);
```

```
begin
```

```
    inherited Create (FileName, Mode);
```

```
    FKey := 'A'; // default
```

```
end;
```

```
function TEncodedStream.Write(const Buffer; Count: Longint): Longint;
```

```
var
```

```
    pBuf, pEnc: PChar;
```

```
    I, EncVal: Integer;
```

```
begin
```

```
    // allocate memory for the encoded buffer
```

```
    GetMem (pEnc, Count);
```

```
    try
```

```
        // use the buffer as an array of characters
```

```
        pBuf := PChar (@Buffer);
```

```
        // for every character of the buffer
```

```
        for I := 0 to Count - 1 do
```

```
            begin
```

```
                // encode the value and store it
```

```
                EncVal := ( Ord (pBuf[I]) + Ord(Key) ) mod 256;
```

```
                pEnc [I] := Chr (EncVal);
```

```
            end;
```

```
        // write the encoded buffer to the file
```

```
        Result := inherited Write (pEnc^, Count);
```

```
    finally
```

```
        FreeMem (pEnc, Count);
```

```
    end;
```

```
end;
```

```
function TEncodedStream.Read(var Buffer; Count: Longint): Longint;
```

```
var
```

```
    pBuf, pEnc: PChar;
```

```
    I, CountRead, EncVal: Integer;
```

```
begin
```

```
    // allocate memory for the encoded buffer
```

```
    GetMem (pEnc, Count);
```

```
    try
```



```

// read the encoded buffer from the file
CountRead := inherited Read (pEnc^, Count);
// use the output buffer as a string
pBuf := PChar (@Buffer);
// for every character actually read
for I := 0 to CountRead - 1 do
begin
    // decode the value and store it
    EncVal := ( Ord (pEnc[I]) - Ord(Key) ) mod 256;
    pBuf [I] := Chr (EncVal);
end;
finally
    FreeMem (pEnc, Count);
end;
// return the number of characters read
Result := CountRead;
end;

```

The comments in this rather complex code should help you understand the details.

Now I used this encoded stream in a demo program, called EncDemo. The form of this program has two memo components and three buttons, as you can see in the following graphic:



The first button loads a plain text file in the first memo; the second button saves the text of this first memo in an encoded file; and the third button reloads the encoded file into the second memo, decoding it. In this example, after encoding the file, I've reloaded it in the first memo as a plain text file on the left, which of course is unreadable.

Because the encoded stream class is available, the code of this program is very similar to that of any other program using streams. For example, here is the method used to save the encoded file (you can compare its code to that of earlier examples based on streams):

```

procedure TFormEncode.BtnSaveEncodedClick(Sender: TObject);
var
    EncStr: TEncodedStream;
begin
    if SaveDialog1.Execute then
    begin
        EncStr := TEncodedStream.Create(SaveDialog1.Filename, fmCreate);
        try
            Mem1.Lines.SaveToStream (EncStr);
        finally

```

```

    EncStr.Free;
end;
end;
end;

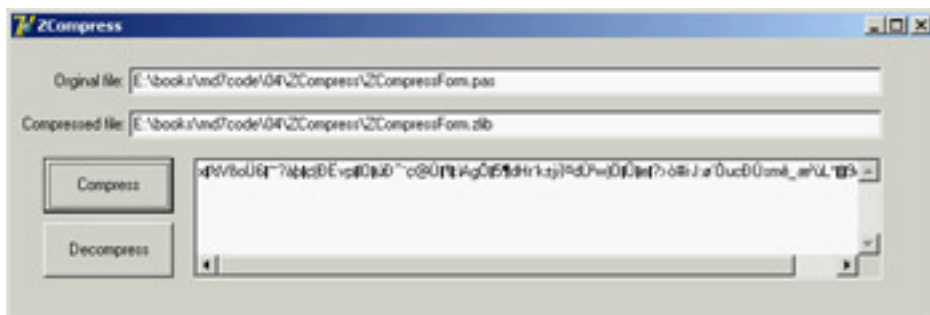
```

---

## Compressing Streams with ZLib

A new feature of Delphi 7 is official support for the ZLib compression library (available and described at [www.gzip.org/zlib](http://www.gzip.org/zlib)). A unit interfacing ZLib has been available for a long time on Delphi's CD, but now it is included in the core distribution and is part of the VCL source (the ZLib and ZLibConst units). In addition to providing an interface to the library (which is a C library you can directly embed in the Delphi program, with no need to distribute a DLL), Delphi 7 defines a couple of helper stream classes: TCompressStream and TDecompressStream.

As an example of using these classes, I've written a small program called ZCompress that compresses and decompresses files. The program has two edit boxes in which you enter the name of the file to compress and the name of the resulting file, which is created if it doesn't already exist. When you click the Compress button, the source file is used to create the destination file; clicking the Decompress button moves the compressed file back to a memory stream. In both cases, the result of the compression or decompression is displayed in a memo. [Figure 4.6](#) shows the result for the compressed file (which happens to be the source code of the form of the current program).



**Figure 4.6:** The ZCompress example can compress a file using the ZLib library.

To make the code of this program more reusable, I've written two functions for compressing or decompressing a stream into another stream. Here is the code:

```

procedure CompressStream (aSource, aTarget: TStream);
var
    comprStream: TCompressionStream;
begin
    comprStream := TCompressionStream.Create(
        clFastest, aTarget);
    try
        comprStream.CopyFrom(aSource, aSource.Size);
        comprStream.CompressionRate;
    finally
        comprStream.Free;
    end;
end;

procedure DecompressStream (aSource, aTarget: TStream) ;
var
    decompStream: TDecompressionStream;
    nRead: Integer;
    Buffer: array [0..1023] of Char;

```

```
begin
  decompStream := TDecompressionStream.Create(aSource);
try
  // aStreamDest.CopyFrom (decompStream, size) doesn't work
  // properly as you don't know the size in advance,
  // so I've used a similar "manual" code
  repeat
    nRead := decompStream.Read(Buffer, 1024);
    aTarget.Write (Buffer, nRead);
  until nRead = 0;
finally
  decompStream.Free;
end;
end;
```

As you can see in the code comment, the decompression operation is slightly more complex because you cannot use the `CopyFrom` method: You don't know the size of the resulting stream in advance. If you pass 0 to the method, it will try to get the size of the source stream, which is a `TDecompressionStream`. However, this operation causes an exception, because the compression and decompression streams can be read only from the beginning to the end and don't allow for seeking the end of the file.

## Summarizing the Core VCL and BaseCLX Units

I've spent most of this chapter discussing the classes of a single unit of the library: `Classes`. This unit, in fact, contains most of the core classes of the library. In this section, I'll provide an overview of what's available in the `Classes` unit and a few other core library units.

### The Classes Unit

The `Classes` unit is at the heart of both VCL and CLX libraries, and although it has many internal changes from the last version of Delphi, little is new for average users. (Most changes are related to modified IDE integration and are meant for expert component writers.)

Here is a list of what you can find in the `Classes` unit, a unit that every Delphi programmer should spend some time with:

- Many enumerated types, the standard method pointer types (including `TNotifyEvent`), and many exception classes.
- Core library classes, including `TPersistent` and `TComponent` as well as many others seldom used directly.
- List classes, including `TList`, `TThreadList` (a thread-safe version of the list), `TInterfaceList` (a list of interfaces, used internally), `TCollection`, `TCollectionItem`, `TOwnedCollection` (which is simply a collection with an owner), `TStrings`, and `TStringList`.
- All the stream classes I discussed in the [previous section](#) but won't list here again. There are also the `TFile`, `TReader`, and `TWriter` classes and a `TParser` class used internally for DFM parsing.
- Utility classes, such as `TBits` for binary manipulation and a few utility routines (for example, point and rectangle constructors, and string list manipulation routines such as `LineStart` and `ExtractStrings`). There are also many registration classes, to notify the system of the existence of components, classes, special utility functions you can replace, and much more.
- The `TDataModule` class, a simple object container alternative to a form. Data modules can contain only nonvisual components and are generally used in database and web applications.

**Note** In early versions of Delphi, the `TDataModule` class was defined in the `Forms` unit; since Delphi 6 it has been moved to the `Classes` unit. This was done to eliminate the code overhead of the GUI classes from non-visual applications (for example, web server modules) and to better separate non-portable Windows code from OS-independent classes, such as `TDataModule`. Other changes relate to the data modules—for example, to allow the creation of web applications with multiple data modules.

- New interface-related classes, such as `TInterfacedPersistent`, aimed at providing further support for interfaces. This particular class allows Delphi code to hold on to a reference to a `TPersistent` object or any descendent implementing interfaces, and is a core element of the new support for interfaced objects in the Object Inspector (see [Chapter 9](#), "Writing Delphi Components," for an example).
- The new `TRecall` class, used to maintain a temporary copy of an object. This class is particularly useful for graphical-based resources.
- The new `TClassFinder` class, which is used to find a registered class instead of the `FindClass` method.
- The `TThread` class, which provides the core operating system-independent support for multithreaded applications.

### New in the Classes Unit

In Delphi 7, the Classes unit has only a few minor additions. Beside the changes I've already mentioned in this chapter, such as the extended support for name-value pairs in the `TStringList` class, there are a couple of new global functions, `AncestorIsValid` and `IsDefaultPropertyValue`.

Both functions were introduced to support the highlighting of non-default properties in the Object Inspector. They serve little other purpose, and I doubt you'll benefit from their use in an application—unless you are interested in saving the status of a component and form, and writing your own custom streaming mechanism.

## Other Core Units

Typical Delphi programmers don't directly use the other units that are part of the RTL package as often as they use Classes. Here is a list of these other units:

- The `TypeInfo` unit includes support for accessing RTTI information for published properties, as discussed in the section "[Accessing Properties by Name](#)."
- The `SyncObjs` unit contains a few generic classes for thread synchronization.
- The `ZLib` unit includes compression and decompression streams, as discussed earlier in the section "[Compressing Streams with ZLib](#)."
- The `ObjAuto` unit contains code to call the published methods of an object by name, passing the parameters in a variant array. This unit is part of the extended support for dynamic method invocation pushed by SOAP and other new Delphi technologies.

Of course, the RTL package also includes the units with functions and procedures discussed in the preceding chapter, such as `Math`, `SysUtils`, `Variants`, `VarUtils`, `StrUtils`, `DateUtils`, and so on.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

As you have seen in this chapter, the Delphi class library has a few root classes that play a considerable role and that you should learn to leverage to the maximum possible extent. Some programmers tend to become expert on the components they use every day, and this is important; but without understanding the core classes (and ideas such as ownership and streaming), you'll have a tough time grasping the full power of Delphi.

Of course, in this book, I also need to discuss visual and database classes. Now that I've introduced all the base elements of Delphi (language, RTL, core classes), I'm ready to discuss the development of real applications with this tool. [Chapter 5](#) covers the structure of the *visual* portion of the component library.

Following chapters are devoted to examples of using the various components to build applications with a modern user interface and use forms effectively. I'll cover the advanced use of traditional controls and menus, discuss the actions architecture, cover the `TForm` class, and then examine toolbars, status bars, dialog boxes, and MDI applications.

## Chapter 5: Visual Controls

### Overview

Now that you've been introduced to the Delphi environment and have seen an overview of the Delphi language and the base elements of component library, we are ready to delve into the use of components and the development of the user interface of applications. This is really what Delphi is about. Visual programming using components is a key feature of this development environment.

Delphi comes with a large number of ready-to-use components. I won't describe every component in detail, examining each of its properties and methods; if you need this information, you can find it in the Help system. The aim of this chapter and the following ones of this book is to show you how to use some of the advanced features offered by the Delphi predefined components to build applications and to discuss specific programming techniques.

I'll start with a comparison of the VCL and VisualCLX libraries and coverage of the core classes (particularly `TControl`). Then I'll examine the various visual components, because choosing the right basic controls will often help you get a project underway more quickly.

## VCL versus VisualCLX

As you saw in [Chapter 4](#), "Core Library Classes," Delphi has two visual class libraries: the cross-platform library (CLX) alongside the traditional Windows library (VCL). There are certainly many differences, even in the use of the RTL and code library classes, between developing programs specifically for Windows or with a cross-platform attitude, but the differences are most striking in the user interface portion.

The visual portion of VCL is a wrapper of the Window API. It includes wrappers of the native Windows controls (like buttons and edit boxes), the common controls (like TreeViews and ListViews), plus a bunch of native Delphi controls bound to the Windows concept of a window. In addition, a `TCanvas` class wraps the basic graphic calls, so you can easily paint on the surface of a window.

VisualCLX, the visual portion of CLX, is a wrapper of the Qt (pronounced "cute") library. It includes wrappers of the native Qt widgets, which range from basic to advanced controls, very similar to Windows' standard and common controls. It also includes painting support using another, similar, `TCanvas` class. Qt is a C++ class library developed by Trolltech ([www.trolltech.com](http://www.trolltech.com)), a Norwegian company with a strong relationship with Borland.

On Linux, Qt is one of the de facto standard user-interface libraries and is the basis of the KDE desktop environment. On Windows, Qt provides an alternative to the use of the native APIs. Unlike VCL, which provides a wrapper to the native controls, Qt provides an alternate implementation to those controls. Even if each of them is based on Windows's window, a QT button isn't a Windows `BUTTON` class control (you can see this by running `WinSight32`). This allows programs to be truly portable, because no hidden differences are created by the operating system (or introduced by the operating system vendor behind the scenes). It also allows you to avoid an extra layer; CLX on top of Qt on top of Windows native controls suggests three layers, but in fact there are two layers in each solution (CLX controls on top of Qt, VCL controls on top of Windows).

**Note** Distributing Qt applications on Windows implies the distribution of the Qt library itself. On the Linux platform, you can generally take the presence of the Qt library for granted, but you still have the interface library to deploy. Also, Linux Borland's CLX is tied to a specific version of Qt (which in Kylix 3 has been specifically patched by Borland), so you'll probably have to distribute it anyway. Distributing the Qt libraries with a professional application (as opposed to an open source project) generally implies paying a license to Trolltech. If you use Delphi or Kylix to build Qt applications, however, Borland has already paid the license to Trolltech for you. You must use at least one CLX class wrapping Qt: If you use the Qt classes exclusively (no CLX at all), you still owe the license to Qt, even when using Delphi or Kylix.

Technically, huge differences exist behind the scenes between a native Windows application built with VCL and a portable Qt program developed with VisualCLX. Suffice to say that at the low level, Windows uses API function calls and messages to communicate with controls, whereas Qt uses class methods and direct method callbacks and has no internal messages. Technically, the Qt classes offer a high-level object-oriented architecture, but the Windows API is still bound to its C legacy and a message-based system dated 1985 (when Windows was released). VCL offers an object-oriented abstraction on top of a low-level API, whereas VisualCLX remaps an already high-level interface into a more *familiar* class library. (For more on the Qt architecture, see the following sidebar "[From Qt to CLX](#).")

**Note** Microsoft has reached the point of starting to abandon the traditional low-level Windows API for a native high-level class library, part of the .NET architecture. You can read more about this topic in [Part IV](#) of this book.

If the underlying architectures of the Windows API on one side and Qt on the other side are relevant, the two class libraries built by Borland (VCL and CLX) flatten out most differences, making the code of Delphi and Kylix applications extremely similar. Using VisualCLX on Linux offers Delphi programmers the advantage of having a familiar class library on top of a totally new platform. From the outside, a button is an object of the `TButton` class for both libraries, and it has more or less the same set of methods, properties, and events. In many cases, you can recompile your existing programs for the new class library in a matter of minutes, if they don't use low-level API calls, platform-dependent features (like ADO or COM), or legacy features (like the BDE).



Qt is a C++ class library that includes a complete set of widgets resembling not only Windows core components (buttons, list boxes, and the like) but also most of the common controls (such as TreeView and ListView controls). Because Qt is a C++ library, it cannot be invoked directly from the Delphi language code. The Qt API is accessible instead through a binding layer, defined in the Qt.pas unit.

This binding layer consists of a long list of wrappers of almost every Qt class suffixed with a final *H*. So, for example, the Qt `QPainter` class becomes the `QPainterH` type in the binding layer. The Qt.pas unit also includes a long list of all the public methods of these classes, transformed into standard functions (not class methods) that have as their first parameter the object to which they apply. The only relevant exception to this approach is the class constructors, which are transformed into functions that return the new instance of the class.

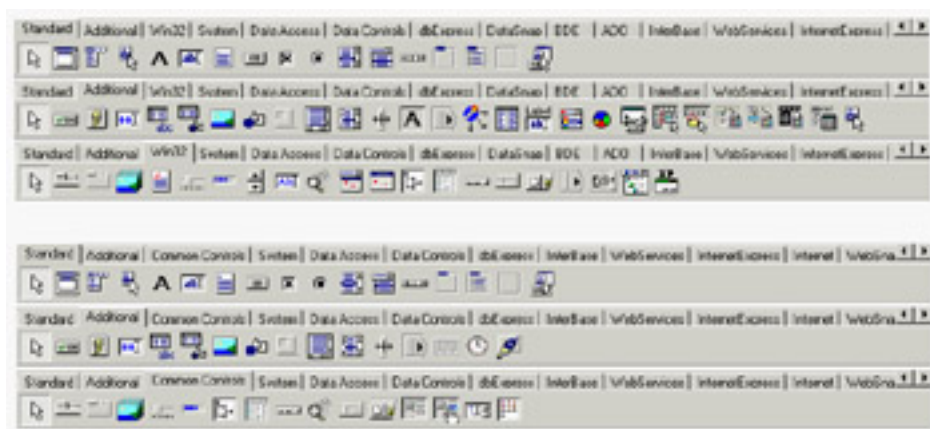
Notice that the use of at least one of the classes of the mapping layer is compulsory for the Qt license that comes with Delphi (and Kylix). Qt is free for non-commercial use under X Window (it is called Qt Free Edition), but you must pay a license fee to Trolltech to develop commercial applications. When you buy Delphi, the Qt license has already been paid for you by Borland, but you must use Qt primarily through CLX (even if low-level calls to Qt within a CLX application are allowed). You cannot use the Qt.pas unit directly and avoid including the QForms unit (which is mandatory). Borland enforces this limitation by omitting from the Qt interface the `QFormH` and `QApplicationH` constructors.

In most of this book's Delphi programs, I'll use only CLX objects and methods. It is important to know that if necessary, you can use some extra Qt features directly; or you may have to do low-level calls to bypass CLX bugs. The Qt documentation is not included in the Delphi help, but you can find it on the Trolltech website ([www.trolltech.com](http://www.trolltech.com)) in HTML and PDF format.

---

## Delphi's Dual Library Support

Delphi has full support for both libraries at design time and at run time. As you begin developing a new application, you can use the File → New Application command to create a new VCL-based program or File → New CLX Application for a new CLX-based program. After you give one of these commands, Delphi's IDE will create a VCL or CLX design-time form and update the Component Palette so that it displays only the visual components compatible with the type of application you've selected (see [Figure 5.1](#) for a comparison). You cannot place a VCL button into a CLX form, and you cannot even mix forms of the libraries within a single executable file. In other words, the user interface of every application must be built using one of the two libraries exclusively, which (aside from the technical implications) makes a lot of sense to me.



**Figure 5.1:** A comparison of the first three pages of the Component Palette for a CLX-based application (above) and a VCL-based application (below)

If you haven't already done so, I suggest you to try experimenting with the creation of a CLX application, looking at the available controls and trying to use them. You'll find few differences in the use of the components, and if you have

been using Delphi for some time, you'll probably be immediately adept with CLX.

## Same Classes, Different Units

One of the cornerstones of the source-code compatibility between CLX and VCL is the fact that similar classes in the two libraries have the same class name. For example, each library has a class called `TButton` representing a push button; the methods and properties are so similar that this code will work with both libraries:

```
with TButton.Create (Self) do
begin
  SetBounds (20, 20, 80, 35);
  Caption := 'New';
  Parent := Self;
end;
```

The two `TButton` classes can have the same name because they are saved in two different units, called `StdCtrls` and `QStdCtrls`. Of course, you cannot have the two components available at design time in the palette, because the Delphi IDE can register only components with unique names. The entire VisualCLX library is defined by units corresponding to the VCL units, but with the letter `Q` as a prefix—so there is a `QForms` unit, a `QDialogs` unit, a `QGraphics` unit, and so on. A few peculiar units, such as `QStyle`, have no corresponding unit in VCL because they map to features of Qt with no correspondence in the Windows API.

Notice that there are no compile settings or other hidden techniques to distinguish between the two libraries; what matters is the set of units referenced in the code. Remember that these references must be consistent—you cannot mix visual controls of the two libraries in a single form or even in a single program.

## DFM and XFM

As you create a form at design time, it is saved to a form definition file. Traditional VCL applications use the DFM extension, which stands for Delphi form module. CLX applications use the XFM extension, which stands for cross-platform (X) form module. A form module is the result of streaming the form and its components: The two libraries share the streaming code, so they produce a similar effect. The format of DFM and XFM files, which can be based on a textual or binary representation, is identical.

So, the reason for having two different extensions doesn't lie in internal compiler tricks or incompatible formats. It is merely an indication to programmers and to the IDE of the type of components you should expect to find within that definition (this indication is *not* included in the file).

If you want to convert a DFM file into an XFM file, you can simply rename the file. However, expect to find some differences in the properties, events, and available components—reopening the form definition for a different library will probably cause quite a few warnings.

**Tip** Delphi's IDE chooses the active library by looking at the extension of the form module, ignoring the references in the `uses` statements. For this reason, you should change the extension if you plan to use CLX. On Kylix, a different extension is useless, because any form is opened in the IDE as a CLX form regardless of the extension. On Linux, there is only the Qt-based CLX library, which is both the cross-platform and the native library.

As an example, I've built two identical applications, `LibComp` and `QLibComp`, with only a few components and a single event handler. [Listing 5.1](#) presents the textual form definitions for two applications, built using the same steps in the Delphi IDE, after choosing a CLX or VCL application. I've marked differences in bold; as you can see, there are very few, most relating to the form and its font. `OldCreateOrder` is a legacy property used for compatibility with Delphi 3 and older code; standard colors have different names; and CLX saves the scroll bars' ranges.

### Listing 5.1: An XFM File (Left) and an Equivalent DFM File (Right)

<pre>object Form1: TForm1   Left = 192   Top = 107</pre>	<pre>object Form1: TForm1   Left = 192   Top = 107</pre>
--	--

```

Width = 350
Height = 210
Caption = 'QLibComp'
Color = clBackground
VertScrollBar.Range = 161
HorzScrollBar.Range = 297

```

```

TextHeight = 13
TextWidth = 6
PixelsPerInch = 96
object Button1: TButton
    Left = 56
    Top = 64
    Width = 75
    Height = 25
    Caption = 'Add'
    TabOrder = 0
    OnClick = Button1Click

```

```

end
object Edit1: TEdit
    Left = 40
    Top = 32
    Width = 105
    Height = 21
    TabOrder = 1
    Text = 'my name'

```

```

end
object ListBox1: TListBox
    Left = 176
    Top = 32
    Width = 121
    Height = 129
    Rows = 3
    Items.Strings = (
        'marco'
        'john'
        'helen')
    TabOrder = 2

```

```

end
end

```

```

Width = 350
Height = 210
Caption = 'LibComp'
Color = clBtnFace
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -11

```

```

Font.Name = 'MS Sans Serif'
Font.Style = []
TextHeight = 13
OldCreateOrder = False
PixelsPerInch = 96
object Button1: TButton
    Left = 56
    Top = 64
    Width = 75
    Height = 25
    Caption = 'Add'
    TabOrder = 0
    OnClick = Button1Click

```

```

end
object Edit1: TEdit
    Left = 40
    Top = 32
    Width = 105
    Height = 21
    TabOrder = 1
    Text = 'my name'

```

```

end
object ListBox1: TListBox
    Left = 176
    Top = 32
    Width = 121
    Height = 129
    ItemHeight = 13
    Items.Strings = (
        'marco'
        'john'
        'helen')
    TabOrder = 2

```

```

end
end

```

## uses Statements

If you look at the source code of a VCL or CLX application, the only relevant difference relates to the `uses` statements. The form of the CLX application has the following initial code:

```

unit QLibCompForm;
interface
uses
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;

```

The form of the VCL program has the traditional `uses` statement:

```

unit LibCompForm;
interface

```

## uses

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls;
```

The code of the class and of the only event handler is absolutely identical. Of course, the compiler directive `{$R *.dfm}` is replaced by `{$R *.xfm}` in the CLX version of a standard program.

## Disabling the Dual Library Help Support

When you press the F1 key in the editor to ask for help on a routine, class, or method of the Delphi library, you'll usually get a choice between the VCL and CLX declarations of the same feature. You'll need to make a choice to proceed to the related help page, which can be quite annoying after a while (especially because the two pages are often identical).

If you don't care about CLX and are planning to use only VCL (or vice versa), you can disable this alternative by choosing the Help → Customize command, removing everything with *CLX* in the name from Contents, Index, and Link, and saving the project. Then restart the Delphi IDE, and the Help engine won't bother asking you about CLX any more. Of course, don't forget to add those help files again if you decide to begin using CLX. Similarly, you can reduce the memory occupation and load time of the Delphi IDE by uninstalling all the CLX-related packages.

## Choosing a Visual Library

Because two different user interface libraries are available in Delphi, you'll have to choose one for each *visual* application. You must evaluate multiple criteria to come to the proper decision, which isn't always easy.

The first criterion is portability. If running your program on Windows and on Linux, with the same user interface, is a major concern to you, then using CLX will make your life simpler and let you keep a single source code file with limited IFDEFs. The same applies if you consider Linux to be (or think it possibly will become) your key platform. On the other hand, if most of your users are on Windows and you just want to extend your offering with a Linux version, you might want to keep a dual VCL/CLX system. Doing so means you'll probably need two different sets of source code files, or you may have too many IFDEFs.

Another criterion is the native look-and-feel. Is you use CLX on Windows, some controls will behave slightly differently than users expect—at least expert users. For a simple user interface (edits, buttons, grids), this probably won't matter much; but if you have many tree view and list view controls, the differences will be clear. On the other hand, with CLX, you'll be able to let your users choose a look-and-feel that's different from the basic Windows look, and use it consistently across platforms. This means that a Motif fan will be able to choose this style even when forced to use the Windows platform. While this flexibility is common on Linux, you'll seldom use a non-native look-and-feel on Windows.

Using native controls also implies that as soon as you get a new version of the Windows operating system, your application will (probably) adapt to it. This is good for the user, but might cause you a lot of headaches in case of incompatibilities. Differences in the Microsoft common controls library over the last few years have been a major source of frustration for Windows programmers in general, including Delphi programmers.

Another criterion is deployment: If you use CLX, you'll have to ship your Windows and Linux program with the Qt libraries.

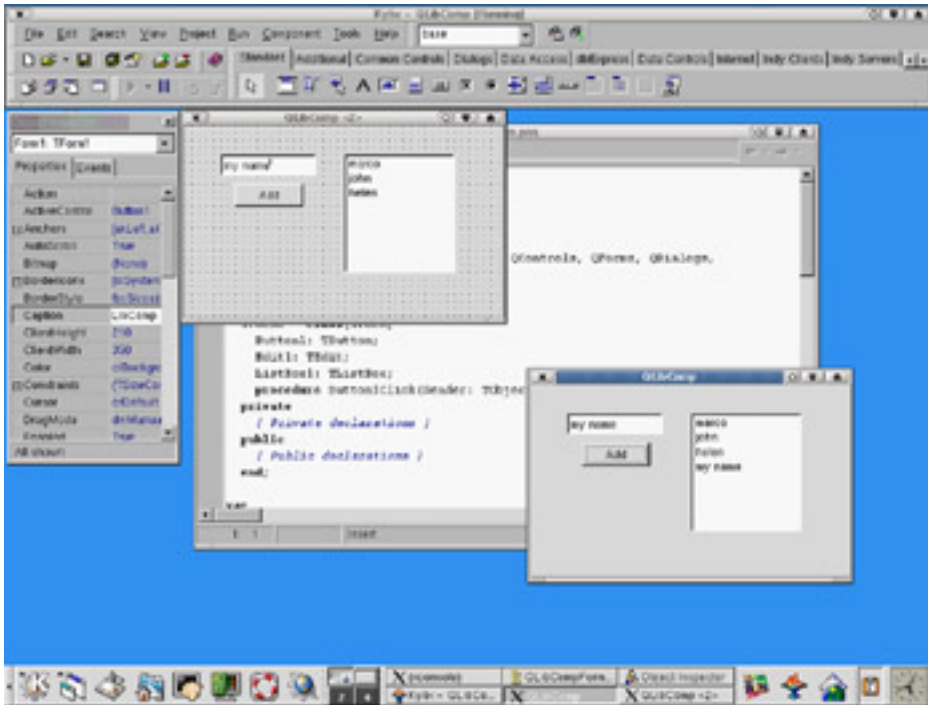
I've done a little testing, and the speed of VCL and CLX applications is similar. I've tried creating 1,000 components and showing them on screen, and the speed differences are few; the VCL-based solution offered a 30 percent advantage (for what this limited benchmark is worth). You can try my tests with the LibSpeed and QLibSpeed examples for this chapter.

Another important criterion for deciding to use CLX instead of VCL is a need for Unicode support. CLX has Unicode support in controls by default (even on Win9x platforms where it isn't supported by Microsoft). VCL, however, has very little Unicode support even on versions of Windows that provide it, making it difficult to build VCL apps for countries where the local character is managed more easily when it is Unicode based.

## Running It on Linux

The real issue of choosing the library resolves to the importance of Linux or Unicode for you and your users. It's important to notice that if you create a CLX application, you'll be able to recompile it unchanged (with the exact source code) with Kylix, producing a native Linux application, unless you've done any Windows API programming at all, in which case conditional compilation will be essential.

As an example, I've recompiled the QLibComp example introduced earlier. [Figure 5.2](#) shows it running; you can also see the Kylix IDE in action on KDE.



**Figure 5.2:** An application written with CLX can be directly recompiled under Linux with Kylix (displayed in the background).

## Conditional Compilation for Libraries

If you want to keep a single source code file but compile with VCL on Windows and CLX on Linux, you can use platform-specific symbols (such as `$IFDEF LINUX`) to distinguish the two situations in case of conditional compilation. But what if you want to be able to compile a portion of code for both libraries on Windows?

You can either define a symbol of your own and use conditional compilation, or (at times) test for the presence of identifiers that exist only in VCL or CLX:

```
{ $IF Declared(QForms) }
  ...CLX-specific code
{ $IFEND }
```

## Converting Existing Applications

Besides starting with new CLX applications, you might want to convert some of your existing VCL applications to the new class library. You must perform a series of operations without any specific help from the Delphi IDE:

- Rename the DFM file using the XFM extension and update all the `{ $R *.DFM }` statements as `{ $R *.XFM }`.
- Update all the `uses` statements in your program (in the units and project files) to refer to the CLX units instead of the VCL units. If you miss even a few, you'll bump into trouble when running your application.

**Tip** To prevent a CLX application from compiling if it contains references to VCL units, you can move the VCL units to a different directory under *lib* and avoid including this folder in your search path. This way, leftover references to VCL units will cause a "Unit not found" error.

[Table 5.1](#) compares the names of the visual VCL and CLX units, excluding the database portion and some rarely referenced units.

**Table 5.1: Names of Equivalent VCL and CLX Units**

VCL	CLX
ActnList	QActnList
Buttons	QButtons
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	QConsts
Controls	QControls
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdCtrls	QStdCtrls

You might also convert references to Windows and Messages into references to the Qt unit. Some Windows data structures are now also available in the Types unit (see [Chapter 3](#), "The Run-Time Library," for details), so you might have to add it to your CLX programs. Notice, however, that the QTypes unit is not the CLX version of VCL's Types unit; these two units are totally unrelated.

**Warning** Watch out for your *uses* statements! If you compile a project that includes a CLX form, but you fail to update the project source code, leaving a reference to the VCL Forms unit there, your program will run but stop immediately. The reason is that no VCL form was created, so the program terminated right away. In other cases, trying to create a CLX form within a VCL application will cause run-time errors. Finally, the Delphi IDE might inappropriately add references to *uses* statements of the wrong library; in this case you end up with a single *uses* statement that refers to the same unit for both libraries, but only the second of the two will be effective. This situation rarely prevents the program from compiling, but you won't be able to run it.

As a helper in converting some of my own programs, I've written a simple unit-replacement tool called VclToClx; it's available in the Tools section of the source code that accompanies the book. You can find more information about this program in [Appendix A](#), "Extra Delphi Tools by the Author."



## TControl and Derived Classes

In [Chapter 4](#), I discussed the base classes of the Delphi library, focusing particularly on the `TComponent` class. One of the most important subclasses of `TComponent` is `TControl`, which corresponds to visual components. This base class is available in both CLX and VCL and defines general concepts, such as the position and the size of the control, the parent control hosting it, and more. For an actual implementation, though, you have to refer to its two subclasses. In VCL these are `TWinControl` and `TGraphicControl`; in CLX they are `TWidgetControl` and `TGraphicControl`. Here are their key features:

**Window-Based Controls (Also Called Windowed Controls)** Visual components based on an operating-system window. A `TWinControl` in VCL has a window handle, which is a number referring to an internal Windows structure. A `TWidgetControl` in CLX has a Qt handle, which is a reference to the internal Qt object. From a user perspective, windowed controls can receive the input focus, and some of them can contain other controls. This is the biggest group of components in the Delphi library. We can further divide windowed controls into two groups: wrappers of native controls of Windows or Qt; and custom controls, which generally inherit from `TCustomControl`.

**Graphical Controls (Also Called Nonwindowed Controls)** Visual components not based on an operating-system window. Therefore, these components have no handle, cannot receive the focus, and cannot contain other controls. They inherit from `TGraphicControl` and are painted by their parent control, which sends them mouse-related and other events. Examples of non-windowed controls are the `Label` and `SpeedButton` components. There are just a few controls in this group, which were critical to minimizing the use of system resources in the early days of Delphi (on 16-bit Windows). Using graphical controls to save Windows resources is still useful on Win9x/Me, which has pushed the system limits higher but hasn't fully gotten rid of them (unlike Windows NT/2000).

## Parent and Controls

The `Parent` property of a control indicates which other control is responsible for displaying it. When you drop a component into a form in the Form Designer, the form will become both parent and owner of the new control. But if you drop the component inside a `Panel`, `ScrollBox`, or any other *container* component, this will become its parent, whereas the form will still be the owner of the control.

When you create the control at run time, you'll need to set the owner (using the `Create` constructor's parameter); but you must also set the `Parent` property, or the control won't be visible.

Like the `Owner` property, the `Parent` property has an inverse. The `Controls` array lists all the controls parented by the current one, numbered from 0 to `ControlCount - 1`. You can scan this property to operate on all the controls hosted by another control, eventually using a recursive method that operates on the controls parented by each subcontrol.

## Properties Related to Control Size and Position

Some of the properties introduced by `TControl` and common to all controls are those related to size and position. The position of a control is determined by its `Left` and `Top` properties, and its size is specified by the `Height` and `Width` properties. Technically, all components have a position, because when you reopen an existing form at design time, you want to be able to see the icons for the nonvisual components in exactly the position where you've placed them. This position is visible in the form file.

**Tip** As you change any of the positional or size properties, you end up calling the single `SetBounds` method. So, any time you need to change two or more of these properties at once, calling `SetBounds` directly will speed up the program. Another method, `BoundsRect`, returns the rectangle bounding the control and corresponds to accessing the properties `Left`, `Top`, `Height`, and `Width`.

An important feature of the position of a component is that, like any other coordinate, it always relates to the client area of its parent component (indicated by its `Parent` property). For a form, the client area is the surface included within its borders and caption (excluding the borders themselves). It would have been messy to work in screen coordinates, although some ready-to-use methods convert the coordinates between the form and the screen and vice versa.

Note, however, that the coordinates of a control are always relative to the parent control, such as a form or another *container* component. If you place a panel in a form and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In this case, the parent component of the button is the panel.

## Activation and Visibility Properties

You can use two basic properties to let the user activate or hide a component. The simpler is the `Enabled` property. When a component is disabled (when `Enabled` is set to `False`), usually some visual hint indicates this state to the user. At design time, the "disabled" property does not always have an effect; but at run time, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding `Hide` method or by setting its `Visible` property to `False`. Be aware, however, that reading the status of the `Visible` property does not tell you whether the control is actually visible. If the container of a control is hidden, even if the control is set to `Visible`, you cannot see it. For this reason, you can read the value of the run-time, read-only `Showing` property to determine whether the control is really visible to the user; that is, if it is visible, its parent control is also visible, the parent control of the parent control is also visible, and so on.

## Fonts

The `Color` and `Font` properties are often used to customize the user interface of a component. Several properties are related to the color. The `Color` property itself usually refers to the background color of the component. There is also a `Color` property for fonts and many other graphical elements. Many components also have `ParentColor` and `ParentFont` properties, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the `Font` property of the form itself.

When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Delphi allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users' computers.

If your program uses a font that your user doesn't have, Windows will select some other font to use in its place. A program's carefully formatted output can be ruined by the font substitution. For this reason, you should rely on standard Windows fonts (such as MS Sans Serif, System, Arial, Times New Roman, and so on).

## Colors

There are various ways to set the value of a color. The type of this property is `TColor`, which isn't a class type but just an integer type. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include `clBlue`, `clSilver`, `clWhite`, `clGreen`, `clRed`, and many others (including Delphi 6's `clMoneyGreen`, `clSkyBlue`, `clCream`, and `clMedGray`). As a better alternative, you can use one of the colors used by the system to denote the status of given elements. These sets of colors are different in VCL and CLX.

VCL includes predefined Windows colors such as the background of a window (`clWindow`), the color of the text of a highlighted menu (`clHighlightText`), the active caption (`clActiveCaption`), and the ubiquitous button face color (`clBtnFace`). CLX includes a different and incompatible set of system colors, including `clBackground`, which is the standard color of a form; `clBase`, used by edit boxes and other visual controls; `clActiveForeground`, the foreground



color for active controls; and `clDisabledBase`, the background color for disabled text controls. All the color constants mentioned here are listed in VCL and CLX Help files under the "TColor type" topic.

Another option is to specify a `TColor` as a number (a 4-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low three bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value `$00FF0000` corresponds to a pure blue color, the value `$0000FF00` to green, the value `$000000FF` to red, the value `$00000000` to black, and the value `$00FFFFFF` to white. By specifying intermediate values, you can obtain any of 16 million possible colors.

Instead of specifying these hexadecimal values directly, you should use the Windows `RGB` function, which has three parameters, all ranging from 0 to 255. The first indicates the amount of red, the second the amount of green, and the last the amount of blue. Using the `RGB` function makes programs generally more readable than using a single hexadecimal constant. `RGB` is *almost* a Windows API function; it is defined by the Windows-related units and not by Delphi units, but a similar function does not exist in the Windows API. C includes a macro that has the same name and effect, which is a welcome addition to the Windows unit. `RGB` is not available on CLX, so I've written my own version:

```
function RGB (red, green, blue: Byte): Cardinal;  
begin  
    Result := blue + green * 256 + red * 256 * 256;  
end;
```

The highest-order byte of the `TColor` type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.)

Regarding palettes and color matching, note that Windows sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, Windows uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors. In 16-color (VGA) adapters and at higher resolutions, you often end up seeing strange patterns of pixels of different colors rather than the color you had in mind.

## The *TWinControl* Class (VCL)

In Windows, most elements of the user interface are windows. From a user standpoint, a window is a portion of the screen surrounded by a border, having a caption and usually a system menu. But technically speaking, a window is an entry in an internal system table, often corresponding to an element visible on the screen that has some associated code. Most of these windows have the role of controls; others are temporarily created by the system (for example, to show a pull-down menu). Still other windows are created by the application but remain hidden from the user and are used only as a way to receive a message (for example, nonblocking sockets use windows to communicate with the system).

The common denominator of all windows is that they are known by the Windows system and refer to a function for their behavior; each time something happens in the system, a notification message is sent to the proper window, which responds by executing some code. Each window of the system has an associated function (generally called its *window procedure*), which handles the various messages the window is interested in.

In Delphi, any `TWinControl` class can override the `WndProc` method or define a new value for the `WindowProc` property. Interesting Windows messages, however, can be better tracked by providing specific message handlers. Even better, VCL converts these lower-level messages into events. In short, Delphi allows you to work at a high level, making application development easier, but still allows you to go low-level when required.

Notice also that creating an instance of a `TWinControl`-based class doesn't automatically create its corresponding Window handle. Delphi uses a lazy initialization technique, so that the low-level control is created only when required—generally as soon as a method accesses the `Handle` property. The `get` method for this property calls `HandleNeeded` the first time, which eventually calls `CreateHandle ...` and so on, eventually reaching `CreateWnd`, `CreateParams`, and `CreateWindowHandle` (the sequence is complex, and you don't need to know it in detail). At the opposite end, you can keep an existing (perhaps invisible) control in memory but destroy its window handle, to save system

resources.

## The *TWidgetControl* Class (CLX)

In CLX, every `TWidgetControl` has an internal Qt object, referenced using the `Handle` property. This property has the same name as the corresponding Windows property, but it is totally different behind the scenes.

A `TWidgetControl` object generally owns the corresponding Qt/C++ object. The CLX class uses delayed construction (the internal object is not created until one of its methods is required) implemented in `InitWidget` and other methods. The CLX class also frees the internal object when it is destroyed. However, it is also possible to create a widget around an existing Qt object: In this case, the CLX object won't own the Qt object and won't destroy it. This behavior is indicated in the `OwnHandle` property.

To be more precise, each `VisualCLX` component has two associated C++ objects: the Qt `Handle` and the Qt `Hook`, which is the object receiving the system events. With the current Qt design, the Qt `Hook` must be a C++ object, which acts as an intermediary to the event handlers of the Delphi language control. The `HookEvents` method associates the hook object to the CLX control.

Unlike Windows, Qt defines two different types of events:

- *Events* are the translation of user input or system events (such as keystroke, mouse move, and paint).
- *Signals* are internal component events (corresponding to VCL internal or abstract operations, such as `OnClick` and `OnChange`).

The events of a CLX component, however, merge events and signals. Generic Delphi CLX control events include `OnMouseDown`, `OnMouseMove`, `OnKeyDown`, `OnChange`, `OnPaint`, and many others, exactly as in the VCL (which fires most events as a response to Windows messages).

**Note** Expert programmers may notice that CLX includes a seldom-used *EventHandler* method, which corresponds more or less to the *WndProc* method of VCL *TWinControl*.

Team LiB

◀ PREVIOUS

NEXT ▶

## Opening the Component Toolbox

So, you want to write a Delphi application. You open a new Delphi project and find yourself faced with a large number of components. The problem is that for every operation, there are multiple alternatives. For example, you can show a list of values using a list box, a combo box, a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which should you use? That's difficult to say. There are many considerations, depending on what you want your application to do. For this reason, I've provided a highly condensed summary of alternative options for a few common tasks.

**Note** For some of the controls described in the following sections, Delphi also includes a data-aware version, usually indicated by the *DB* prefix. As you'll see in [Chapter 13](#), "Delphi's Database Architecture," the *DB* version of a control typically serves a role similar to that of its "standard" equivalent; but the properties and the ways you use it are often quite different. For example, in an Edit control you use the *Text* property, whereas in a DBEdit component you access the *Value* of the related field object.

## The Text Input Components

Although a form or component can handle keyboard input directly using the `OnKeyPress` event, this isn't a common operation. Windows provides ready-to-use controls you can use to get string input and even build a simple text editor. Delphi has several slightly different components in this area.

### The Edit Component

The Edit component allows the user to enter a single line of text. You can also display a single line of text with a Label or a StaticText control, but these components are generally used only for fixed text or program-generated output, not for input. In CLX, there is also a native LCD digit control you can use to display numbers.

The Edit component uses the `Text` property, whereas many other controls use the `Caption` property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the `OnKeyPress` event of the edit box. For example, you can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's not, you change the value of the key to the null character (`#0`), so that it won't be processed by the edit control and will produce a warning beep:

```
procedure TForm1.Edit1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

**Note** A minor difference of CLX is that the Edit control has no built-in Undo mechanism. Another is that the `PasswordChar` property is replaced by the `EchoMode` property. You don't determine the character to display, but whether to echo the entered text or display an asterisk instead.

### The LabeledEdit Control

Delphi 6 added a nice control called LabeledEdit, which is an Edit control with a label attached to it. The Label appears as a property of the compound control, which inherits from `TCustomEdit`.

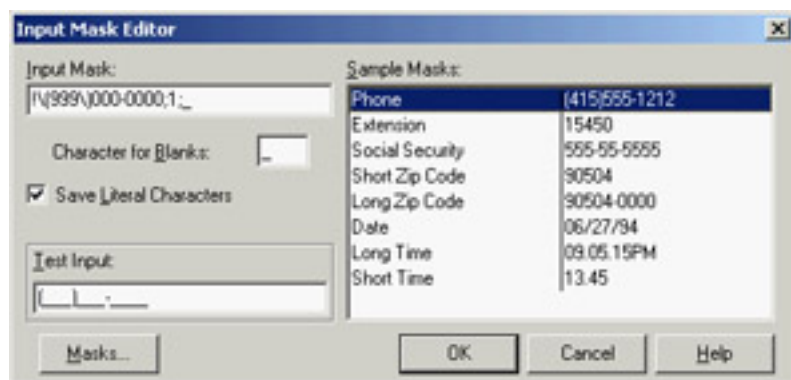
This component is very handy, because it allows you to reduce the number of components on your forms, move them around more easily, and have a more consistent layout for all of the labels of an entire form or application. The

`EditLabel` property is connected with the subcomponent, which has the usual properties and events. Two more properties, `LabelPosition` and `LabelSpacing`, allow you to configure the relative positions of the two controls.

**Note** This component has been added to the `ExtCtrls` unit to demonstrate the use of subcomponents in the Object Inspector. I'll discuss the development of these components in [Chapter 9](#), "Writing Delphi Components." Notice also that this component is not available on CLX.

## The MaskEdit Component

To customize the input of an edit box further, you can use the `MaskEdit` component. It has an `EditMask` property, which is a string indicating for each character whether it should be uppercase, lowercase, or a number, and other similar conditions. You can see the editor for the `EditMask` property here:



The Input Mask Editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to display the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask Editor correspond to the last two fields of the mask (separated by semicolons).

**Tip** Clicking the `Masks` button in the Input Mask Editor lets you choose predefined input masks for different countries.

## The Memo and RichEdit Components

The controls discussed so far allow a single line of input. The `Memo` component, by contrast, can host several lines of text but (on the Win95/98 platforms) still retains the 16-bit Windows text limit (32 KB) and allows only a single font for the entire text. You can work on the text of the memo line by line (using the `Lines` string list) or access the entire text at once (using the `Text` property).

If you want to host a large amount of text or change fonts and paragraph alignments, in VCL you should use the `RichEdit` control, a Win32 common control based on the RTF document format. You can find an example of a complete editor based on the `RichEdit` component among the sample programs that ship with Delphi. (The example is named `RichEdit`, too.)

**Warning** The `RichEdit` control is one of the few commonly used Delphi controls not available in CXL and in Kylix. The latest version of Qt has a similar native control, so these controls may be supported by future versions of CLX.

The `RichEdit` component has a `DefAttributes` property indicating the default styles and a `SelAttributes` property indicating the style of the current selection. These two properties are not of the `TFont` type, but they are compatible with fonts, so you can use the `Assign` method to copy the value, as in the following code fragment:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
```

```

begin
  FontDialog1.Font.Assign (RichEdit1.DefAttributes);
  if FontDialog1.Execute then
    RichEdit1.SelAttributes.Assign (FontDialog1.Font);
end;
end;

```

## The TextViewer CLX Control

CLX and Qt lack a RichEdit control, but on the other hand they provide a full-blown HTML viewer, which is powerful for displaying formatted text but not for typing it. This HTML viewer is embedded in two controls: the single-page TextViewer control and the TextBrowser control with active links.

As a simple demo, I've added a memo and a text viewer to a CLX form and connected them so that everything you type on the memo is immediately displayed in the viewer. I've called the example HtmlEdit not because this is a real HTML editor, but because this is the simplest way I know to build an HTML preview inside a program. The program's form is shown at run time in [Figure 5.3](#).



**Figure 5.3:** The HtmlEdit example at run time: When you add new HTML text to the memo, you get an immediate preview.

**Tip** I originally built this example with Kylix on Linux. To port it to Windows and Delphi, all I had to do was copy the files and recompile.

## Selecting Options

Two standard Windows controls allow the user to choose different options. Two other controls let you group sets of options.

### The CheckBox and RadioButton Components

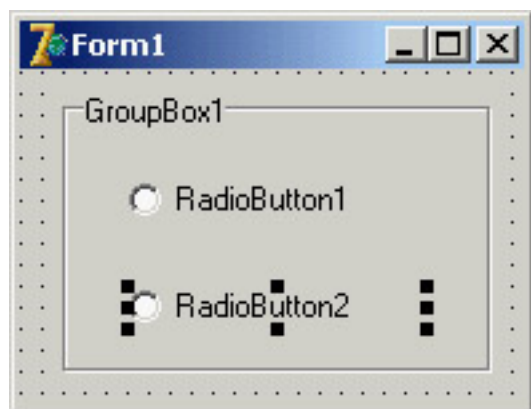
The first standard option-selecting control is the *check box*, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the `AllowGrayed` property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks the check box.

The second type of control is the *radio button*, which corresponds to an exclusive selection. Two radio buttons on the

same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected (as programmer, you are responsible for selecting one of the radio buttons at design time).

## The GroupBox Components

To host several groups of radio buttons, you can use a GroupBox control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the GroupBox component on a form and then add the radio buttons to the group box, as in the following example:



You can handle the radio buttons individually, but it's easier to navigate through the array of controls owned by the group box, as discussed in [Chapter 4](#). Here is a small code excerpt used to get the text of a group's selected radio button:

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := TRadioButton(GroupBox1.Controls[I]).Caption;
```

## The RadioGroup Component

Delphi has a similar component that can be used specifically for radio buttons: the RadioGroup component. A RadioGroup is a group box with some radio buttons inside it. The difference is that these internal radio buttons are managed automatically by the container control. Using a radio group is generally easier than using a group box, because the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Another advantage is that the RadioGroup component can automatically align its radio buttons in one or more columns (as indicated by the `Columns` property), and you can easily add new choices at run time by adding strings to the `Items` string list. By contrast, adding new radio buttons to a group box is quite complex.

## Lists

When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six, to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that display lists of items and allow the user to select one of them.

## The ListBox Component

The selection of an item in a list box uses the `Items` and `ItemIndex` properties as in the earlier code shown for the RadioGroup control. If you need access to the text of selected list box items often, you can write a small wrapper function like this:



```

function SelText (List: TListBox): string;
var
    nItem: Integer;
begin
    nItem := List.ItemIndex;
    if nItem >= 0 then
        Result := List.Items [nItem]
    else
        Result := '';
end;

```

Another important feature is that by using the `ListBox` component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the `MultiSelect` property. There are two kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case, a user selects multiple items simply by clicking them; in the second case, the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items, respectively. The two alternatives are determined by the status of the `ExtendedSelect` property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the `SelCount` property, and it can determine which items are selected by examining the `Selected` array. This array of Boolean values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the `Selected` array as follows:

```

var
    SelItems: string;
    nItem: Integer;
begin
    SelItems := '';
    for nItem := 0 to ListBox1.Items.Count - 1 do
        if ListBox1.Selected [nItem] then
            SelItems := SelItems + ListBox1.Items[nItem] + ' ';

```

Differently from VCL, in CLX you can configure a `ListBox` to use a fixed number of columns and rows, using the `Columns`, `Row`, `ColumnLayout`, and `RowLayout` properties. Of these, the VCL `ListBox` has only the `Columns` property.

## The ComboBox Component

List boxes take up a lot of screen space, and they offer a fixed selection—that is, a user can choose only among the items in the list box and cannot enter any choice the programmer did not specifically foresee.

You can solve both problems by using a `ComboBox` control, which combines an edit box and a drop-down list. The behavior of a `ComboBox` component changes a lot depending on the value of its `Style` property:

- The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request.
- The `csDropDownList` style defines a combo box that does not allow editing (but uses the keystrokes to select an item).
- The `csSimple` style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a `ComboBox` is easier than doing the same operation for a list box, because you can simply use the `Text` property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the Enter key. The following method first tests whether the user has pressed that key, by looking for the character with the numeric (ASCII) value of 13. It then tests to make sure the text of the combo box is not empty and is not already in the list (if its position in the list is less than zero). Here is the code:

```

procedure TForm1.ComboBox1KeyPress(
    Sender: TObject; var Key: Char);
begin
    // if the user presses the Enter key
    if Key = Chr (13) then
        with Sender as TComboBox do
            if (Text <> '') and (Items.IndexOf (Text) < 0) then
                Items.Add (Text);
end;

```

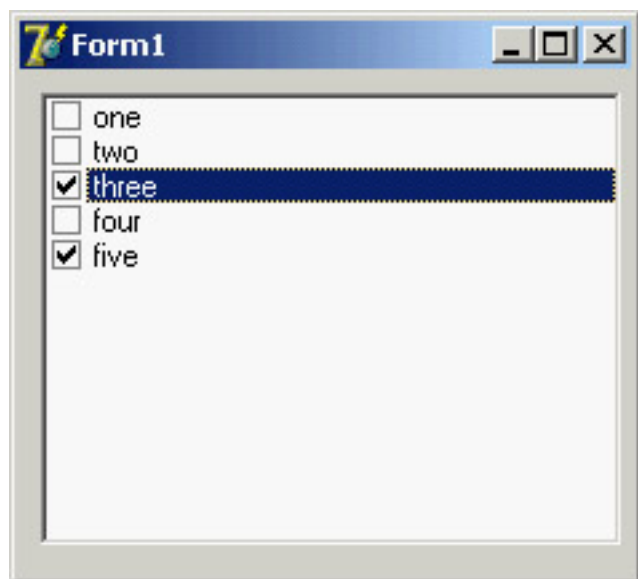
**Tip** In CLX, the combo box can automatically add the text typed into the edit box to the drop-down list when the user presses the Enter key. Also, some events fire at different times than in VCL.

Since version 6, Delphi includes two new events for the combo box. The `OnCloseUp` event corresponds to the closing of the drop-down list and complements the preexisting `OnDropDown` event. The `OnSelect` event fires only when the user selects something in the drop-down list, as opposed to typing in the edit portion.

Another nice feature is the `AutoComplete` property. When it is set, the `ComboBox` component (and the `ListBox`, as well) automatically locates the string nearest to the one the user is entering, suggesting the final part of the text. The core of this feature, also available in CLX, is implemented in the `TCustomListBox.KeyPress` method.

## The CheckListBox Component

Another extension of the list box control is represented by the `CheckListBox` component, a list box with each item preceded by a check box:



A user can select a single item in the list, but can also click the check boxes to toggle their status. This makes the `CheckListBox` a very good component for multiple selections or for highlighting the status of a series of independent items (as in a series of check boxes).

To check the current status of each item, you can use the `Checked` and `State` array properties (use the latter if the check boxes can be grayed). Delphi 5 introduced the `ItemEnabled` array property, which you can use to enable or disable each item of the list. You'll use the `CheckListBox` in the `DragList` example, later in this chapter.

**Tip** Most of the list-based controls share a common and important feature: Each item in the list has an associated 32-bit value, usually indicated by the `TObject` type. This value can be used as a tag for each list item, and it's very useful for storing additional information along with each item. This approach is connected to a specific feature of the native Windows list box control, which offers four bytes of extra storage for each list box item. You'll use this feature in the `ODList` example later in this chapter.



## The Extended Combo Boxes: ComboBoxEx and ColorBox

The ComboBoxEx (where *ex* stands for extended) is the wrapper of a new Win32 common control that extends the traditional combo box by allowing images to appear next to the items in the list. You attach an image list to the combo box, and then select an image index for each item to display. The effect of this change is that the simple `Items` string list is replaced by a more complex collection, the `ItemsEx` property. I'll use the ComboBoxEx control in the RefList2 example in [Chapter 7](#), "Working with Forms."

**Tip** In Delphi 7, the ComboBoxEx component has the new `AutoCompleteOptions` property, enabling the combo box to respond to user keystrokes.

The ColorBox control is a version of the combo box specifically aimed at selecting colors. You can use its `Style` property to choose which groups of colors you want to see in the list (standard color, extended colors, system colors, and so on).

## The ListView and TreeView Components

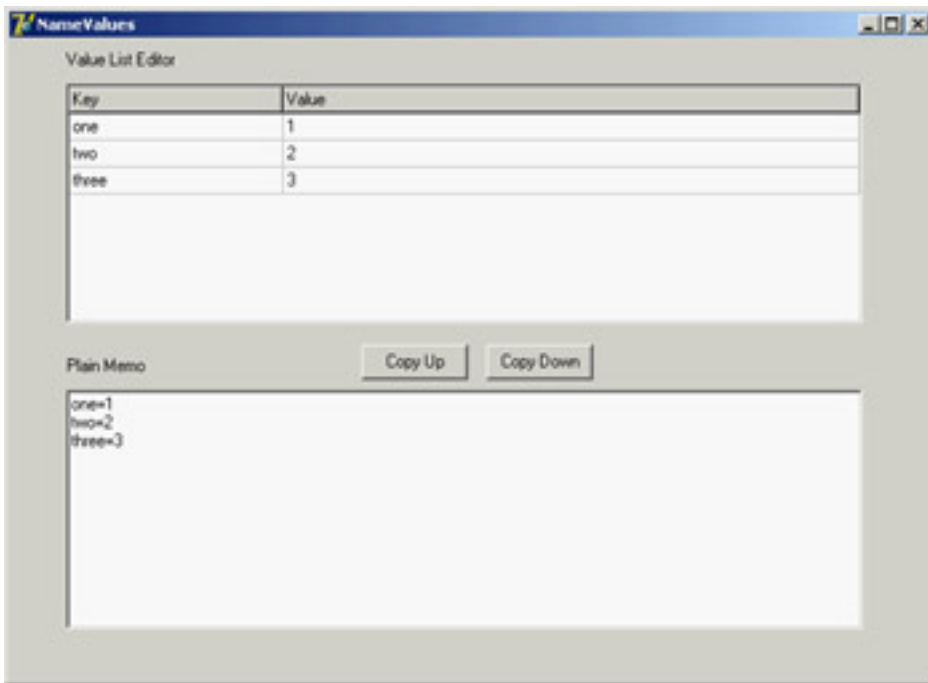
If you want an even more sophisticated list, you can use the ListView common control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described in the section "[ListView and TreeView Controls](#)" later in this chapter. Other alternatives for listing values are the TreeView common control, which shows items in a hierarchical output, and the StringGrid control, which shows multiple elements for each line. For an actual example of the use of this component, refer to the free online chapter "Graphics in Delphi" discussed in [Appendix C](#).

If you use the common controls in your application, users will already know how to interact with them, and they will regard the user interface of your program as up to date. TreeView and ListView are the two key components of Windows Explorer, and you can assume that many users will be familiar with them—even more so than with the traditional Windows controls. CLX adds also an IconView control, which parallels some features of the VCL ListView.

**Warning** The ListView control in CLX doesn't have the small/large icon styles of its Windows counterpart, but a companion control, IconView, provides this capability.

## The ValueListEditor Component

Delphi applications often use the name/value structure natively offered by string lists, which I discussed in [Chapter 4](#). Delphi 6 introduced a version of the StringGrid (technically a `TCustomDrawGrid` descendant class) component specifically geared toward this type of string lists. The ValueList-Editor has two columns in which you can display and let the user edit the contents of a string list with name/value pairs, as you can see in [Figure 5.4](#). This string list is indicated in the `Strings` property of the control.



**Figure 5.4:** The NameValues example uses the ValueListEditor component, which shows the name/value or key/value pairs of a string list, also visible in a plain memo.

The power of this control lies in the fact that you can customize the editing options for each position of the grid or for each key value, using the run-time-only `ItemProps` array property. For each item, you can indicate:

- Whether it is read-only
- The maximum number of characters of the string
- An edit mask (eventually requested in the `OnGetEditMask` event)
- The items in a drop-down pick list (eventually requested in the `OnGetPickList` event), as demonstrated by the first item of the example.
- The display of a button that will show an editing dialog box (in the `OnEditButtonClick` event, which the example handles with a message box)

Needless to say, this behavior resembles what is available generally for string grids and the `DBGrid` control, and also the behavior of the Object Inspector.

The `ItemProps` property must be set up at run time, by creating an object of the `TItemProp` class and assigning it to an index or a key of the string list. To have a default editor for each line, you can assign the same item property object multiple times. In the example, this shared editor sets an edit mask for up to three numbers:

```

procedure TForm1.FormCreate(Sender: TObject);
var
    I: Integer;
begin
    SharedItemProp := TItemProp.Create (ValueListEditor1);
    SharedItemProp.EditMask := '999;0; ';
    SharedItemProp.EditStyle := esEllipsis;

    FirstItemProp := TItemProp.Create (ValueListEditor1);
    for I := 1 to 10 do
        FirstItemProp.PickList.Add(IntToStr (I));

    Mem1.Lines := ValueListEditor1.Strings;

```

```
ValueListEditor1.ItemProps [0] := FirstItemProp;  
for I := 1 to ValueListEditor1.Strings.Count - 1 do  
    ValueListEditor1.ItemProps [I] := SharedItemProp;  
end;
```

You must repeat similar code in case the number of lines changes—for example, by adding new elements in the memo and copying them to the value list.

```
procedure TForm1.ValueListEditor1StringsChange(Sender: TObject);  
var  
    I: Integer;  
begin  
    ValueListEditor1.ItemProps [0] := FirstItemProp;  
    for I := 1 to ValueListEditor1.Strings.Count - 1 do  
        if not Assigned (ValueListEditor1.ItemProps [I]) then  
            ValueListEditor1.ItemProps [I] := SharedItemProp;  
        end;  
    end;
```

**Note** Reassigning the same editor twice causes trouble, so I've assigned the editor only to the lines that don't already have one.

Another property, `KeyOptions`, allows you to let the user edit the keys (the names), add new entries, delete existing entries, and allow for duplicated names in the first portion of the string. Oddly enough, you cannot add new keys unless you also activate the edit options, which makes it hard to let the user add extra entries while preserving the names of the basic entries.

## Ranges

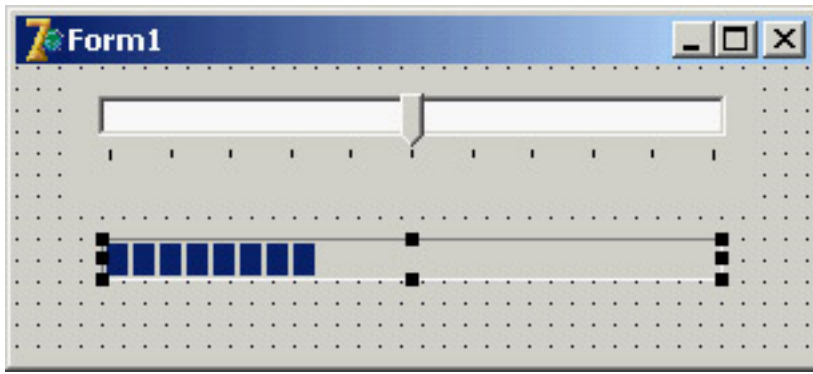
Finally, you can use a few components to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

### The ScrollBar Component

The stand-alone ScrollBar control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border, a feature governed by a specific Windows style of the form window. By *resembling*, I mean that it is not technically a separate window of the ScrollBar component type. These "fake" scroll bars are usually controlled in Delphi using two specific properties of the form and the other components hosting them: `VertScrollBar` and `HorzScrollBar`.

### The TrackBar and ProgressBar Components

Direct use of the ScrollBar component is quite rare, especially with the TrackBar component introduced with Windows 95, which is used to let a user select a value in a range. Among Win32 common controls is the companion ProgressBar control, which allows the program to output a value in a range, showing the progress of a lengthy operation. These two components are visible here:



## The UpDown Component

Another related control is the UpDown component, which is usually connected to an edit box so that the user can either type a number in it or increase and decrease the number using the two small arrow buttons. To connect the two controls, you set the `Associate` property of the UpDown component. Nothing prevents you from using the UpDown component as a stand-alone control, displaying the current value in a label or in some other way.

**Note** CLX has no UpDown control, but it offers a SpinEdit that bundles an Edit with the UpDown in a single control.

## The PageScroller Component

The Win32 PageScroller control is a container allowing you to scroll the internal control. For example, if you place a toolbar in the page scroller and the toolbar is larger than the available area, the PageScroller will display two small arrows on the side. Clicking these arrows will scroll the internal area. This component can be used as a scroll bar, but it also partially replaces the ScrollBox control.

## The ScrollBox Component

The ScrollBox control represents a region of a form that can scroll independently from the rest of the surface. For this reason, the ScrollBox has two scroll bars used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many Windows applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scroll bars of the form, you might allow the user to move the toolbar or status bar out of view (a very odd situation).

## Commands

The final category of components is not as clear-cut as the previous ones, and relates to commands. The basic component of this group is the `TButton` (or *push button*, in Windows jargon). More than stand-alone buttons, Delphi programmers use buttons (or `TToolButton` objects) within toolbars (in the early ages of Delphi, they used speed buttons within panels). Beside buttons and similar controls, the other key technique for issuing commands is the use of menu items, part of the *pull-down* menus attached to forms' main menus or local *pop-up* menus activated with the right mouse button.

Menu- or toolbar-related commands fall into different categories depending on their purpose and the feedback their interface provides to the user:

**Commands** Menu items or buttons used to execute an action.

**State-setters** Menu items or buttons used to toggle an option on and off, to change the state of a particular element. The menu items of these commands usually have a check mark to their left to indicate that they are active (you can automatically obtain this behavior using the `AutoCheck` property). Buttons are generally painted in a *pressed down* state to indicate the same status (the `ToolButton` control has a `Down` property).

**Radio Items** Menu items that display a bullet and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, set the `RadioItem` property to `True` and set the `GroupIndex` property for the alternative menu items to the same value. In a similar fashion, you can have groups of toolbar buttons that are mutually exclusive.

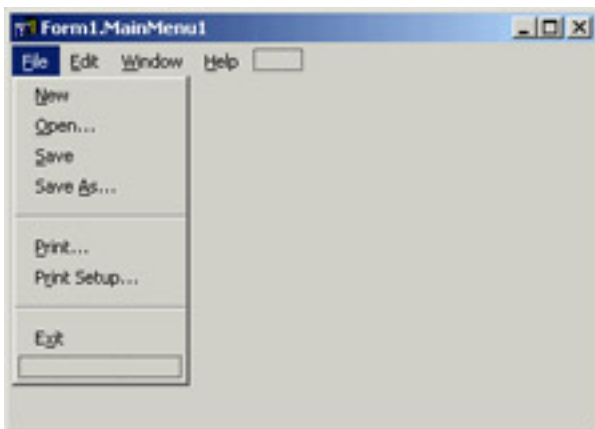
**Dialog Openers** Items that cause a dialog box to appear. They are usually indicated by an ellipsis (...) after the text.

## Commands and Actions

As you'll see in [Chapter 6](#), modern Delphi applications tend to use the `ActionList` component or its `ActionManager` extension to handle menu and toolbar commands. In short, you define a series of action objects and associate each of them to a toolbar button and/or a menu item. You can define the command execution in a single place but also update the user interface simply by targeting the action; the related visual control will automatically reflect the status of the action object.

## The Menu Designer

If you just need to show a simple menu in your application, you can place a `MainMenu` or `PopupMenu` component on a form and double-click on it to fire up the Menu Designer, shown in [Figure 5.5](#). You add new menu items and provide them with a `Caption` property, using a hyphen (-) to separate caption menu items.



**Figure 5.5:** Delphi's Menu Designer in action

Delphi creates new components for each menu item you add. To name each component, Delphi uses the caption you enter and appends a number (so that `Open` becomes `Open1`). After removing spaces and other special characters from the caption, if nothing is left Delphi adds the letter `N` to the name. Finally it appends the number. Thus menu item separators are called `N1`, `N2`, and so on. Knowing what Delphi tends to do by default, you should think of editing the name first, which is necessary if you want to end up with a sensible component naming scheme.

**Warning** Do not use the `Break` property, which is used to lay out a pull-down menu on multiple columns. The `mbMenuBarBreak` value indicates that this item will be displayed on a second or subsequent line; the `mbMenuBreak` value means this item will be added to a second or subsequent column of the pull-down menu.

To obtain a more modern-looking menu, you can add an image list control to the program, hosting a series of bitmaps, and connect the image list to the menu using its `Images` property. You can then set an image for each menu item by setting the proper value of its `ImageIndex` property. The definition of images for menus is quite flexible—you can associate an image list with any specific pull-down menu (and even a specific menu item) using the `SubMenuImages` property. Having a specific smaller image list for each pull-down menu instead of a single large image list for the entire menu allows for more run-time customization of an application.

**Tip** Creating menu items at run time is so common that Delphi provides some ready-to-use functions in the `Menus` unit. The names of these global functions are self-explanatory: `NewMenu`, `NewPopupMenu`, `NewSubMenu`, `NewItem`, and `NewLine`.

## Pop-Up Menus and the `OnContextPopup` Event

The `PopupMenu` component is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property. However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, which is a label in this code fragment:

```
procedure TForm1.Label3MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
    begin
      ScreenPoint := Label3.ClientToScreen (Point (X, Y));
      PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
    end;
end;
```

An alternative approach is to use the `OnContextMenu` event. This event, introduced in Delphi 5, fires when a user right-clicks a component—exactly what I traced previously with the test `if Button = mbRight`. The advantage is that the same event is also fired in response to a `Shift+F10` key combination, as well as the shortcut-menu key of some keyboards. You can use this event to fire a pop-up menu with little code:

```
procedure TFormPopup.Label1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now), 0, False, True, nil, 0, ''));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This result is not particularly useful, but it illustrates that if you need to display a plain pop-up menu, you can easily use the `PopupMenu` property of the control in question or one of its parent controls. Handling the `OnContextMenu` event makes sense only when you want to do some extra processing.

The `Handled` parameter is preinitialized to `False`, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set `Handled` to `True` and the system will stop processing the message. You'll rarely set `Handled` to `True`, because you'll generally handle the `OnContextPopup` to dynamically create or customize the pop-up menu, but then you can let the default handler show

the menu.

The handler of an `OnContextPopup` event isn't limited to displaying a pop-up menu. It can perform any other operation, such as directly display a dialog box. Here is an example of a right-click operation used to change the color of the control:

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;  
    MousePos: TPoint; var Handled: Boolean);  
begin  
    ColorDialog1.Color := Label2.Color;  
    if ColorDialog1.Execute then  
        Label2.Color := ColorDialog1.Color;  
    Handled := True;  
end;
```

All the code snippets from this section are available in the simple `CustPop` example for VCL and `QCustPop` for CLX.

**Team LiB**

◀ PREVIOUS

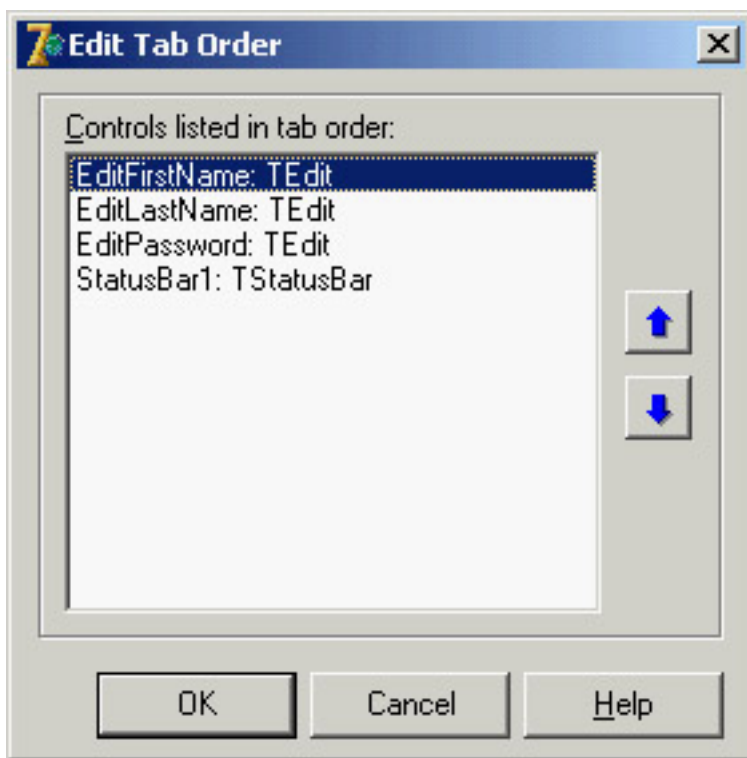
NEXT ▶

## Control-Related Techniques

After this general overview of the most commonly used Delphi controls, I'll devote some space to discussing generic core techniques not related to a specific component. I'll cover the input focus, control anchors, the use of the splitter component, and the display of fly-by hints. Of course, these topics don't include everything you can do with visual controls, but they provide a starting point for exploration to get you up and running with some of the most common techniques.

### Handling the Input Focus

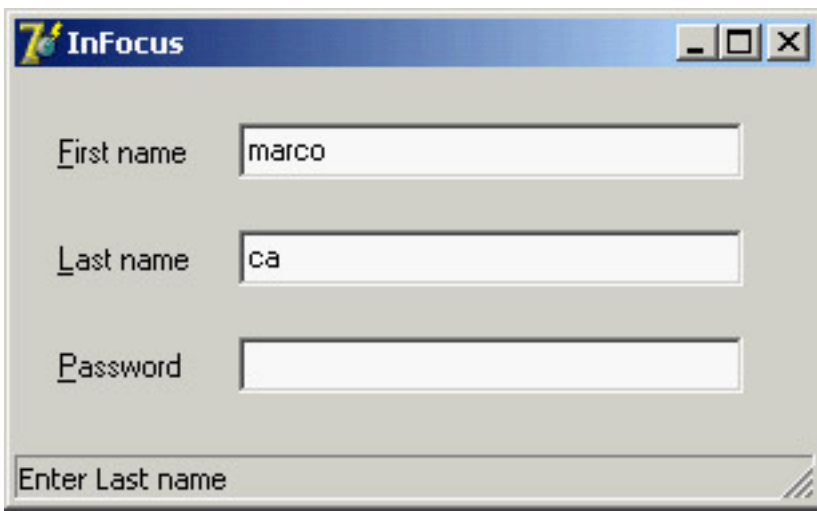
Using the `TabStop` and `TabOrder` properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key. Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, shown in [Figure 5.6](#).



**Figure 5.6:** The Edit Tab Order dialog box

Besides these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the `InFocus` example, which creates a typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in [Figure 5.7](#). At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.





**Figure 5.7:** The InFocus example at run time

For the output of the status information, I've used the `StatusBar` component, with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the `&` character in the labels, indicating a shortcut key, and the connection of these labels with corresponding edit boxes (using the `FocusControl` property):

```

object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
object Label1: TLabel
  Caption = '&First name'
  FocusControl = EditFirstName
end
object EditFirstName: TEdit
  OnEnter = GlobalEnter
  OnExit = EditFirstNameExit
end
object Label2: TLabel
  Caption = '&Last name'
  FocusControl = EditLastName
end
object EditLastName: TEdit
  OnEnter = GlobalEnter
end
object Label3: TLabel
  Caption = '&Password'
  FocusControl = EditPassword
end
object EditPassword: TEdit
  PasswordChar = '*'
  OnEnter = GlobalEnter
end
object StatusBar1: TStatusBar
  SimplePanel = True
end
end

```

The program is simple and performs only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' `OnEnter` event, using a single generic event handler to avoid repetitive code. In the example, instead of storing extra information for each edit box, I've checked each control of the form to determine which label is connected to the current edit box (indicated by the `Sender` parameter):

```

procedure TFocusForm.GlobalEnter(Sender: TObject);
var
    I: Integer;
begin
    for I := 0 to ControlCount - 1 do
        // if the control is a label
        if (Controls [I] is TLabel) and
            // and the label is connected to the current edit box
            (TLabel(Controls[I]).FocusControl = Sender) then
            // copy the text, leaving off the initial & character
            StatusBar1.SimpleText := 'Enter ' +
                Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;

```

The form's second event handler relates to the first edit box's `OnExit` event. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```

procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
    if EditFirstName.Text = '' then
        begin
            // don't let the user get out
            EditFirstName.SetFocus;
            MessageDlg ('First name is required', mtError, [mbOK], 0);
        end
    else if EditFirstName.Text = 'Admin' then
        begin
            // fill the second edit and jump to the third
            EditLastName.Text := 'Admin';
            EditPassword.SetFocus;
        end;
end;

```

**Tip** The CLX version of this example has the same code and is available as the `QInFocus` program.

## Control Anchors

To let you create a nice, flexible user interface, with controls adapting themselves to the current size of the form, Delphi allows you to determine the relative position of a control with the `Anchors` property. Before this feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom, unless it was aligned to the bottom or right side. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to anchor a button to the bottom-right corner of the form, you place the button in the required position and set its `Anchors` property to `[akRight, akBottom]`. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

On the other hand, if you place a large component such as a Memo or a Listbox in the middle of a form, you can set its `Anchors` property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

**Tip** Anchors, like constraints, work both at design time and at run time. You should set them up as early as possible, to benefit from this feature while you're designing the form as well as at run time.

As an example of both approaches, you can try the Anchors application, which has two buttons in the bottom-right corner and a list box in the middle. As shown in [Figure 5.8](#), the controls automatically move and stretch as the form size changes. To make this form work properly, you must also set its `Constraints` property; otherwise, if the form becomes too small, the controls can overlap or disappear.



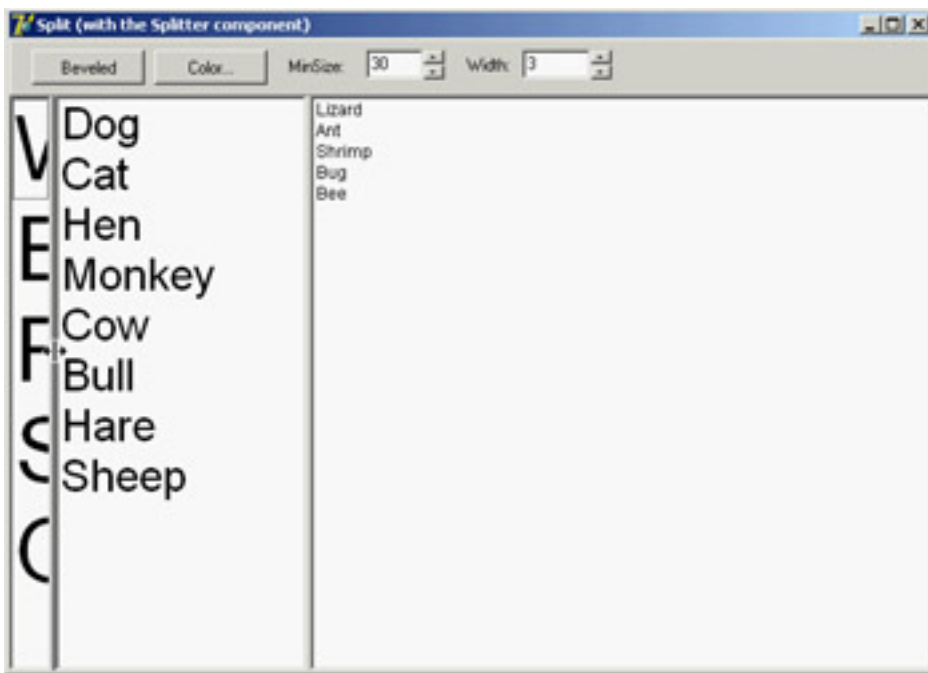
**Figure 5.8:** The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only proper use of the `Anchors` property.

Notice that if you remove all the anchors or two opposite ones (for example, left and right), the resize operations will cause the control to *float* in the form. The control keeps its current size, and the system adds or removes the same number of pixels on each side of it. This anchor can be defined as centered, because if the component is initially in the middle of the form it will keep that position. If you want a centered control you should generally use both opposite anchors, so that if the user makes the form larger, the control size will grow as well. In the case just presented, making the form larger leaves a small control in its center.

## Using the Splitter Component

There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the `Constraints` property of the controls it relates to. As you'll see in the Split1 example, this technique allows you to define maximum and minimum positions for the splitter and the form. To build this example, simply place a Listbox component in a form; then add a Splitter component, a second Listbox, another Splitter, and finally a third Listbox component. The form also has a simple toolbar based on a panel.

By simply placing these two splitter components, you give your form the complete functionality of moving and sizing the controls it hosts at run time. The `Width`, `Beveled`, and `Color` properties of the splitter components determine their appearance, and in the Split1 example you can use the toolbar controls to change them. Another relevant property is `MinSize`, which determines the minimum size of the form's components. During the splitting operation (see [Figure 5.9](#)), a line marks the final position of the splitter, but you cannot drag this line beyond a certain limit. The behavior of the Split1 program is not to let controls become too small. An alternative technique is to set the new `AutoSnap` property of the splitter to `True`. This property will make the splitter hide the control when its size goes below the `MinSize` limit.



**Figure 5.9:** The Split1 example's splitter component determines the minimum size for each control on the form, even those not adjacent to the splitter.

I suggest you try using the Split1 program, so that you'll fully understand how the splitter affects its adjacent controls and the other controls of the form. Even if you set the `MinSize` property, a user can reduce the size of the program's entire form to a minimum, hiding some of the list boxes. If you test the Split2 version of the example, you'll get better behavior. In Split2, I've set some `Constraints` for the `ListBox` controls:

```
object ListBox1: TListBox
  Constraints.MaxHeight = 400
  Constraints.MinHeight = 200
  Constraints.MinWidth = 150
```

The size constraints are applied only as you resize the controls; so, to make this program work satisfactorily, you have to set the `ResizeStyle` property of the two splitters to `rsUpdate`. This value indicates that the controls' position is updated for every movement of the splitter, not only at the end of the operation. If you select the `rsLine` or the new `rsPattern` value, instead, the splitter simply draws a line in the required position, checking the `MinSize` property but not the constraints of the controls.

**Tip** When you set the `Splitter` component's `AutoSnap` property to `True`, the splitter will completely hide the neighboring control when the size of that control is below the minimum set for it in the `Splitter` component.

## Horizontal Splitting

You can also use the `Splitter` component for horizontal splitting, instead of the default vertical splitting. Basically, you place a component on a form, align it to the top, and then place the splitter on the form. By default, the splitter will be left aligned. Choose the `alTop` value for the `Align` property, and you're done. You can see a form with a horizontal splitter in the SplitH example. This program has two memo components into which you can open a file, and it has a splitter dividing them, defined as follows:

```
object Splitter1: TSplitter
  Cursor = crVSplit
  Align = alTop
  OnMoved = Splitter1Moved
end
```

The program features a status bar, which keeps track of the current height of the two memo components. It handles the `OnMoved` event of the splitter (the only event of this component) to update the text of the status bar. The same code is executed whenever the form is resized:

```

procedure TForm1.Splitter1Moved(Sender: TObject);
begin
    StatusBar1.Panels[0].Text := Format ('Upper Memo: %d - Lower Memo: %d',
        [MemoUp.Height, MemoDown.Height]);
end;

```

## Accelerator Keys

Since Delphi 5, you don't need to enter the & character in the `Caption` of a menu item, which provides an automatic accelerator key if you omit one. Delphi's automatic accelerator-key system can also figure out if you have entered conflicting accelerator keys and fix them on the fly. This doesn't mean you should stop adding custom accelerator keys with the & character, because the automatic system simply uses the first available letter and doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This feature is controlled by the `AutoHotkeys` property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to `maAutomatic`; in the pull-downs and menu items, it defaults to `maParent`, so the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of `maAutomatic` or `maManual`.

The engine behind this system is the `RethinkHotkeys` method of the `TMenuItem` class, and the companion `InternalRethinkHotkeys` method. There is also a `RethinkLines` method, which checks whether a pull-down has two consecutive separators or begins or ends with a separator. In all these cases, the separator is automatically removed.

One of the reasons Delphi includes this feature is the support for translations. When you need to translate an application's menu, it is convenient if you don't have to deal with the accelerator keys, or at least if you don't have to worry about whether two items on the same menu conflict. Having a system that can automatically resolve similar problems is definitely an advantage. Another motivation was Delphi's IDE. With all the dynamically loaded packages that install menu items in the IDE main menu or in pop-up menus, and with different packages loaded in different versions of the product, it's next to impossible to get nonconflicting accelerator-key selections in each menu. That is why this mechanism isn't a wizard that does static analysis of your menus at design time; it was created to deal with the real problem of managing menus created dynamically at run time.

**Warning** This feature is certainly handy, but because it is active by default, it can break existing code. I had to modify two of this chapter's program examples, between the Delphi 4 and Delphi 5 edition of the book, just to avoid run-time errors caused by this change. The problem is that I use the caption in the code, and the extra & broke my code. The change was quite simple, though: All I had to do was to set the `AutoHotkeys` property of the main menu component to `maManual`.

## Using the Fly-by Hints

Another common element in toolbars is the *tooltip*, also called *fly-by hint*—text that briefly describes the button currently under the cursor. This text is usually displayed in a yellow box after the mouse cursor has remained steady over a button for a set amount of time. To add hints to a group of buttons or components, simply set the `ShowHints` property of the parent control to `True` and enter some text for the `Hint` property of each element. You might want to enable the hints for all the components on a form, or all the buttons of a toolbar or panel.

If you want to have more control over how hints are displayed, you can use some of the properties and events of the `Application` object. This global object has, among others, the following properties:

Property	Defines
<code>HintColor</code>	The background color of the hint window
<code>HintPause</code>	How long the cursor must remain on a component before hints are displayed

HintHidePause	How long the hint will be displayed
HintShortPause	How long the system should wait to display a hint if another hint has just been displayed

For example, a program might allow a user to customize the hint background color by selecting a specific color with the following code:

```
ColorDialog.Color := Application.HintColor;
if ColorDialog.Execute then
    Application.HintColor := ColorDialog.Color;
```

As an alternative, you can change the hint color by handling the `OnShowHint` property of the `Application` object. This handler can change the hint's color for specific controls. The `OnShowHint` event is used in the `CustHint` example described in the [next section](#).

## Customizing the Hints

Just as you can add hints to an application's toolbar, you can add hints to forms or to the components of a form. For a large control, the hint will show up near the mouse cursor. In some cases, it is important to know that a program can freely customize how hints are displayed. One thing you can do is to change the value of the properties of the `Application` object, as I mentioned at the end of the last section. To obtain more control over hints, you can customize them even further by assigning a method to the application's `OnShowHint` event. You need to either hook up this event manually or—better—add an `ApplicationEvents` component to the form and handle its `OnShowHint` event.

The event handler method has some interesting parameters, such as a string with the hint's text, a Boolean flag for its activation, and a `THintInfo` structure with further information, including the control, the hint position, and its color. The parameters are passed by reference, so you have a chance to change them and also modify the values of the `THintInfo` structure; for example, you can change the position of the hint window before it is displayed.

This is what I've done in the `CustHint` example, which shows the hint for the label at the center of its area.

```
procedure TForm1.ShowHint (var HintStr: string; var CanShow: Boolean;
    var HintInfo: THintInfo);
begin
    with HintInfo do
        // if the control is the label show the hint in the middle
        if HintControl = Label1 then
            HintPos := HintControl.ClientToScreen (Point (
                HintControl.Width div 2, HintControl.Height div 2));
end;
```

The code retrieves the center of the generic control (the `HintInfo.HintControl`) and then converts its coordinates to screen coordinates, applying the `ClientToScreen` method to the control.

You can further update the `CustHint` example in a different way. The form's `ListBox` control has some rather long text items, so you might want to display the entire text in a hint while the mouse moves over the item. Setting a single hint for the list box won't do, of course.

A good solution is to customize the hint system by providing a hint dynamically corresponding to the text of the list box item under the cursor. You also need to indicate to the system which area the hint belongs to, so that by moving over the next line a new hint will be displayed. You accomplish this by setting the `CursorRect` field of the `THintInfo` record, which indicates the area of the component that the cursor can move over without disabling the hint. When the cursor moves outside this area, Delphi hides the hint window. Here is the related code snippet I've added to the `ShowHint` method:

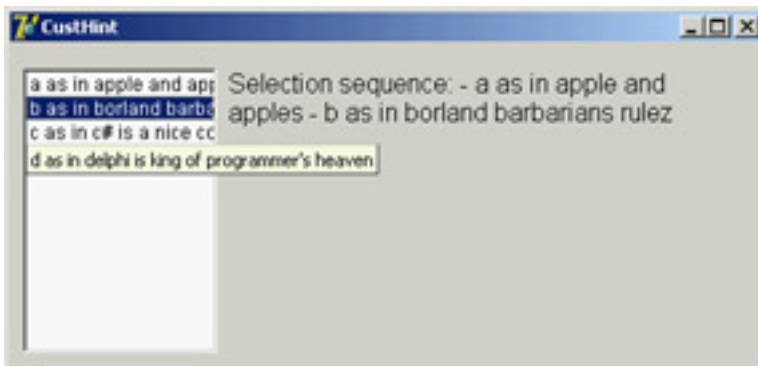
```
else if HintControl = ListBox1 then
```

```

begin
  nItem := ListBox1.ItemAtPos(
    Point (CursorPos.x, CursorPos.Y), True);
  if nItem >= 0 then
    begin
      // set the hint string
      HintStr := ListBox1.Items[nItem];
      // determine area for hint validity
      CursorRect := ListBox1.ItemRect(nItem);
      // display over the item
      HintPos := HintControl.ClientToScreen (Point(
        0, ListBox1.ItemHeight * (nItem - ListBox1.TopIndex)));
    end
  else
    CanShow := False;
end;

```

The resulting effect is that each line of the list box appears to have a specific hint, as shown in [Figure 5.10](#). The hint position is computed so that it covers the current item text, extending beyond the list box border.



**Figure 5.10:** The ListBox control of the CustHint example shows a different hint, depending on which list item the mouse is over.

## Owner-Draw Controls and Styles

In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically, these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In the VCL, the situation is slightly more complex. The components can take care of painting themselves in this case (as in the `TBitBtn` class for bitmap buttons) and possibly activate corresponding events. The system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers. In CLX, some of the controls, such as ListBoxes and ComboBoxes, surface events very similar to Windows owner-draw, but menus lack them. The native approach of Qt is to use styles to determine the graphical behavior of all the controls in the system, of a specific application, or of a given control. I'll introduce styles shortly, later in this section.



**Note** Most of the Win32 common controls have support for the owner-draw technique, generally called *custom drawing*. You can fully customize the appearance of a ListView, TreeView, TabControl, PageControl, HeaderControl, StatusBar, or ToolBar. The ToolBar, ListView, and TreeView controls also support *advanced* custom drawing, a more fine-tuned drawing capability introduced by Microsoft in the latest versions of the Win32 common controls library. The downside to owner-draw is that when the Windows user interface style changes in the future (and it always does), your owner-draw controls that fit in perfectly with the current user interface styles will look outdated and out of place. Because you are creating a custom user interface, you'll need to keep it updated yourself. By contrast, if you use the standard output of the controls, your applications will automatically adapt to a new version of such controls.

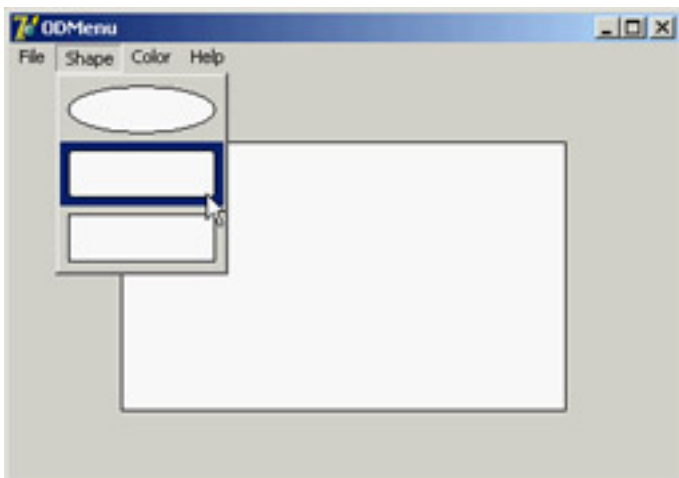
## Owner-Draw Menu Items

VCL makes the development of graphical menu items quite simple compared to the traditional approach of the Windows API: You set the `OwnerDraw` property of a menu item component to `True` and handle its `OnMeasureItem` and `OnDrawItem` events. In the `OnMeasureItem` event, you can determine the size of the menu items. This event handler is activated once for each menu item when the pull-down menu is displayed and has two reference parameters you can set: `Width` and `Height`. In the `OnDrawItem` event, you paint the actual image. This event handler is activated every time the item has to be repainted. This happens when Windows first displays the items and each time the status changes; for example, when the mouse moves over an item, the item should become highlighted.

To paint the menu items, you must consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on. Luckily, the Delphi event passes to the handler the `Canvas` where it should paint, the output rectangle, and the status of the item (selected or not). In the `ODMenu` example, I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've set the `OwnerDraw` property of the menu and written handlers for some of the menu items. To write a single handler for each event of the three color-related menu items, I've set their `Tag` property to the value of the color in the `OnCreate` event handler of the form. This makes the handler of the items' `OnClick` event quite straightforward:

```
procedure TForm1.ColorClick(Sender: TObject);  
begin  
    ShapeDemo.Brush.Color := (Sender as TComponent).Tag;  
end;
```

The handler of the `OnMeasureItem` event doesn't depend on the actual items, but uses fixed values (different from the handler of the other pull-down). The most important portion of the code is in the handlers of the `OnDrawItem` events. For the color, you use the value of the tag to paint a rectangle of the given color, as you can see in [Figure 5.11](#). Before doing this, however, you have to fill the background of the menu items (the rectangular area passed as a parameter) with the standard color for the menu (`clMenu`) or the selected menu items (`clHighlight`):



**Figure 5.11:** The owner-draw menu of the `ODMenu` example

```
procedure TForm1.ColorDrawItem(Sender: TObject; ACanvas: TCanvas;
```



```

ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
ACanvas.FillRect (ARect);
  // show the color
  ACanvas.Brush.Color := (Sender as TComponent).Tag;
  InflateRect (ARect, -5, -5);
  ACanvas.Rectangle (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;

```

The three handlers for this event of the Shape pull-down menu items are all different, although they use similar code:

```

procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
ACanvas.FillRect (ARect);
  // draw the ellipse
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;

```

**Note** To accommodate the increasing number of states in the Windows 2000 user interface style, Delphi includes the *OnAdvancedDrawItem* event for menus.

## A ListBox of Colors

As you have just seen for menus, list boxes have an owner-draw capability, which means a program can paint the items of a list box. The same support is provided for combo boxes and is also available on CLX. To create an owner-draw list box, you set its *Style* property to *lbOwnerDrawFixed* or *lbOwnerDrawVariable*. The first value indicates that you will set the height of the list box items by specifying the *ItemHeight* property and that this will be the height of every item. The second owner-draw style indicates a list box with items of different heights; in this case, the component will trigger the *OnMeasureItem* event for each item, to ask the program for their heights.

In the *ODList* example (and its *QODList* version), I'll stick with the first, simpler, approach. The example stores color information along with the list box items and then draws the items using those colors (instead of using a single color for the whole list).

The DFM or XFM file of every form, including this one, has a *TextHeight* attribute, which indicates the number of pixels required to display text. You should use this value for the list box's *ItemHeight* property. An alternative solution is to compute this value at run time, so that if you later change the font at design time, you don't have to remember to set the height of the items accordingly.

**Note** I've just described *TextHeight* as an attribute of the form, not a property. It isn't a property, but a local value of the form. If it is not a property, you might ask, how does Delphi save it in the *DFM* file? The answer is that Delphi's streaming mechanism is based on properties plus special property clones created by the *DefineProperties* method.

Because `TextHeight` is *not* a property, although it is listed in the form description, you cannot access it directly. Studying the VCL source code, I found that this value is computed by calling a private method of the form: `GetTextHeight`. Because it is private, you cannot call this function. Instead, you can duplicate its code (which is quite simple) in the `FormCreate` method of the form, after selecting the font of the list box:

```
Canvas.Font := ListBox1.Font;  
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

Next you add some items to the list box. Because this is a list box of colors, you want to add color names to the `Items` of the list box and the corresponding color values to the `Objects` data storage related to each list item. Instead of adding the two values separately, I've written a procedure to add new items to the list:

```
procedure TODListForm.AddColors (Colors: array of TColor);  
var  
    I: Integer;  
begin  
    for I := Low (Colors) to High (Colors) do  
        ListBox1.Items.AddObject (ColorToString (Colors[I]), TObject(Colors[I]));  
end;
```

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. For each item passed as a parameter, you add the name of the color to the list, and you add its value to the related data by calling the `AddObject` method. To obtain the string corresponding to the color, you call the Delphi `ColorToString` function. It returns a string containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the `TObject` data type (a four-byte reference), as required by the `AddObject` method.

**Tip** Besides `ColorToString`, which converts a color value into the corresponding string with the identifier or the hexadecimal value, the Delphi `StringToColor` function converts a properly formatted string into a color.

In the `ODList` example, this method is called in the form's `OnCreate` event handler (after the height of the items has been set):

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime, clPurple,  
    clGray, RGB (213, 23, 123), RGB (0, 0, 0), clAqua, clNavy, clOlive, clTeal]);
```

To compile the CLX version of this code, I've added to it the `RGB` function described earlier in the section "[Colors](#)." The code used to draw the items is not particularly complex. You simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl; Index: Integer;  
    Rect: TRect; State: TOwnerDrawState);  
begin  
    with Control as TListBox do  
        begin  
            // erase  
            Canvas.FillRect(Rect);  
            // draw item  
            Canvas.Font.Color := TColor (Items.Objects [Index]);  
            Canvas.TextOut(Rect.Left, Rect.Top, ListBox1.Items[Index]);  
        end;  
end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on your part. Moreover, the program allows you to add new items by double-clicking on the list box:

```
procedure TODListForm.ListBox1DbClick(Sender: TObject);  
begin  
    if ColorDialog1.Execute then
```

```
    AddColors ([ColorDialog1.Color]);  
end;
```

If you try using this capability, you'll notice that some colors you add are turned into color names (one of the Delphi color constants), whereas others are converted into hexadecimal numbers.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## ListView and TreeView Controls

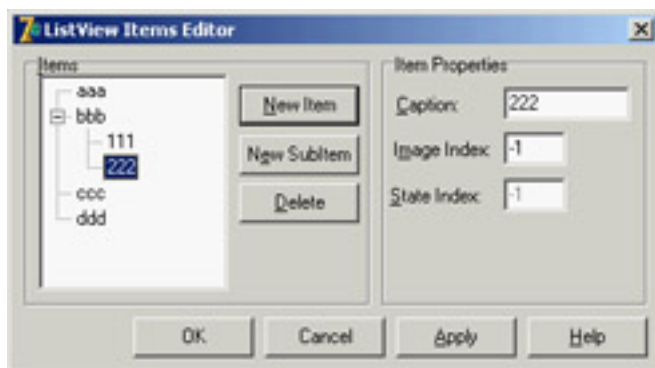
In the earlier section "[Opening the Component Toolbox](#)," I introduced the various visual controls you can use to display lists of values. The standard list box and combo box components are very common, but they are often replaced by the more powerful ListView and TreeView controls. These two controls are part of the Win32 common controls, stored in the `ComCtl32.DLL` library. Similar controls are available in Qt and VisualCLX, both on Windows and Linux.

### A Graphical Reference List

When you use a ListView component, you can provide bitmaps both indicating the status of the element (for example, the selected item) and describing the contents of the item in a graphical way.

To connect the images to a list or tree, you need to refer to the ImageList component you've already used for the menu images. A ListView can have three image lists: one for the large icons (the `LargeImages` property), one for the small icons (the `SmallImages` property), and one for the state of the items (the `StateImages` property). In the `RefList` example, I've set the first two properties using two different ImageList components.

Each item of the ListView has an `ImageIndex`, which refers to its image in the list. For this technique to work properly, the elements in the two image lists should follow the same order. When you have a fixed image list, you can add items to it using Delphi's ListView Item Editor, which is connected to the `Items` property. In this editor, you can define items and subitems. The subitems are displayed only in the detailed view (when you set the `vsReport` value of the `ViewStyle` property) and are connected with the titles set in the `Columns` property:



**Warning** The ListView control in CLX doesn't have the small and large icon views. In Qt, this type of display is available from another component, the `IconView`.

In my `RefList` example (a simple list of references to books, magazines, CD-ROMs, and websites), the items are stored to a file, because users of the program can edit the contents of the list, which are automatically saved as the program exits. This way, edits made by the user become persistent. Saving and loading the contents of a ListView is not trivial, because the `TListItem` type doesn't have an automatic mechanism to save the data. As an alternative approach, I've copied the data to and from a string list, using a custom format. The string list can then be saved to a file and reloaded with a single command.

The file format is simple, as you can see in the following saving code. For each list item, the program saves the caption on one line, the image index on another line (prefixed by the `@` character), and the subitems on the following lines, indented with a tab character:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I, J: Integer;
  List: TStringList;
begin
  // store the items
  List := TStringList.Create;
```

```

try
  for I := 0 to ListView1.Items.Count - 1 do
  begin
    // save the caption
    List.Add (ListView1.Items[I].Caption);
    // save the index
    List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
    // save the subitems (indented)
    for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
      List.Add (#9 + ListView1.Items[I].SubItems [J]);
    end;
    List.SaveToFile (ExtractFilePath (Application.ExeName) + 'Items.txt');
  finally
    List.Free;
  end;
end;

```

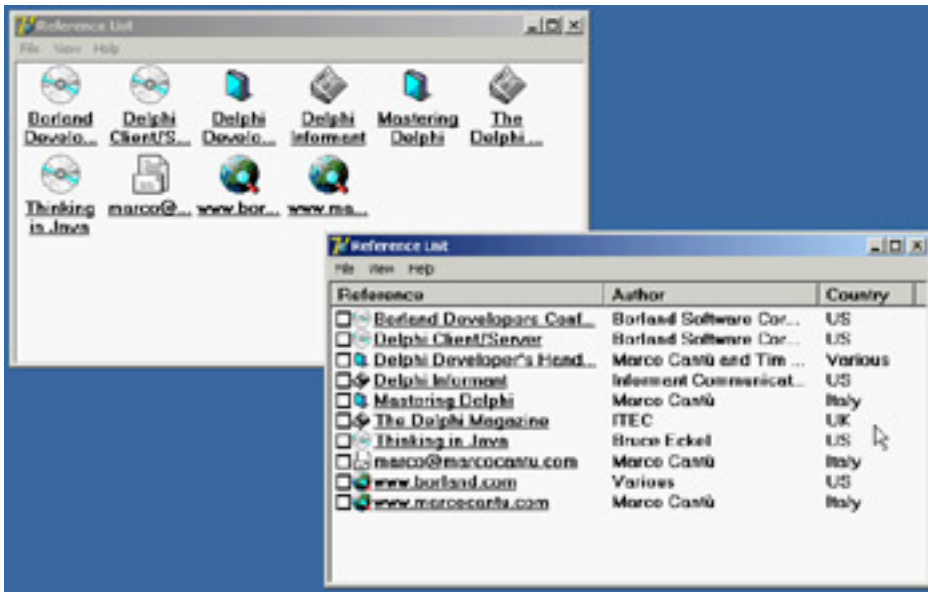
The items are then reloaded in the FormCreate method:

```

procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStringList;
 NewItem: TListItem;
  I: Integer;
begin
  // stops warning message
  NewItem := nil;
  // load the items
  ListView1.Items.Clear;
  List := TStringList.Create;
  try
    List.LoadFromFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
    for I := 0 to List.Count - 1 do
      if List [I][1] = #9 then
        NewItem.SubItems.Add (Trim (List [I]))
      else if List [I][1] = '@' then
        NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
      else
        begin
          // a new item
          NewItem := ListView1.Items.Add;
          NewItem.Caption := List [I];
        end;
    finally
      List.Free;
    end;
  end;
end;

```

The program has a menu you can use to choose one of the different views supported by the ListView control and to add check boxes to the items, as in a CheckListBox control. You can see some combinations of these styles in [Figure 5.12](#).



**Figure 5.12:** Different examples of the output of a ListView component in the RefList program, obtained by changing the `ViewStyle` property and adding the check boxes

Another important feature, which is common in the detailed or report view of the control, lets a user sort the items on one of the columns. In the VCL, this technique requires three operations. First, you set the `SortType` property of the ListView to `stBoth` or `stData`. This way, the ListView will sort based not on the captions, but by calling the `OnCompare` event for each two items it has to sort.

Second, because you want to sort on each of the columns of the detailed view, you also handle the `OnColumnClick` event (which takes place when the user clicks on the column titles in the detailed view, but only if the `ShowColumnHeaders` property is set to `True`). Each time a column is clicked, the program saves the number of that column in the form class's `nSortCol` private field:

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  nSortCol := Column.Index;
  ListView1.AlphaSort;
end;
```

Then, in the third step, the sorting code uses either the caption or one of the subitems according to the current sort column:

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem; Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
      Item2.SubItems [nSortCol - 1]);
end;
```

In the CLX version of the program (called `QRefList`) you don't have to do any of the previous steps. The control is already capable of sorting itself properly when its caption is clicked. You automatically get multiple columns that auto-sort (both ascending and descending).

The final features I've added to the program relate to mouse operations. When the user left-clicks an item, the RefList program shows a description of the selected item. Right-clicking the selected item sets it in edit mode, and a user can change it (keep in mind that the changes will automatically be saved when the program terminates). Here is the code for both operations, in the `OnMouseDown` event handler of the ListView control:

```

procedure TForm1.ListView1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
      begin
        // create and show a description
        strDescr := ListView1.Columns [0].Caption + #9 +
          ListView1.Selected.Caption + #13;
        for I := 1 to ListView1.Selected.SubItems.Count do
          strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
            ListView1.Selected.SubItems [I-1] + #13;
        ShowMessage (strDescr);
      end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
    end;

```

Although it is not feature-complete, this example shows some of the potential of the ListView control. I've also activated the "hot-tracking" feature, which lets the list view highlight and underline the item under the mouse. The relevant properties of the ListView can be seen in its textual description:

```

object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
    item
      Caption = 'Country'
      Width = 80
    end>
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  FullDrag = True
  HideSelection = False
  HotTrack = True
  HotTrackStyles = [htHandPoint, htUnderlineHot]
  SortType = stBoth
  ViewStyle = vsList
  OnColumnClick = ListView1ColumnClick
  OnCompare = ListView1Compare
  OnMouseDown = ListView1MouseDown
end

```

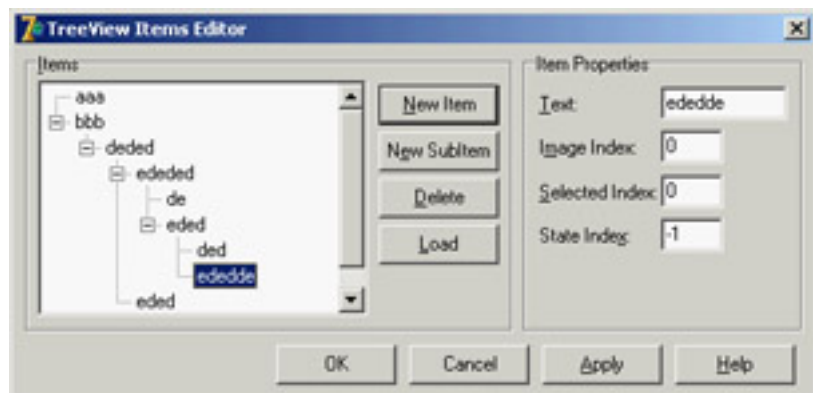
This program is quite interesting, and I'll further extend it in [Chapter 9](#) by adding a dialog box.

To build its CLX version, QRefList, I had to use only one of the image lists and disable the small images and large images menus, because a ListView is limited to the list and report view styles. Large and small icons are available in a different control, called IconView. As previously mentioned, the sorting support was already there, which could have saved most of the code of the example.

## A Tree of Data

Now that you've seen an example based on the ListView, let's examine the TreeView control. The TreeView has a user interface that is flexible and powerful (with support for editing and dragging elements). It is also standard, because it is the Windows Explorer user interface. There are properties and various ways to customize the bitmap of each line or each type of line.

To define the structure of the TreeView nodes at design time, you can use the TreeView Items Editor:



In this case, however, I've decided to load the TreeView data at startup, in a way similar to the last example.

The `Items` property of the TreeView component has many member functions you can use to alter the hierarchy of strings. For example, you can build a two-level tree with the following lines:

```
var
  Node: TTreeNode;
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

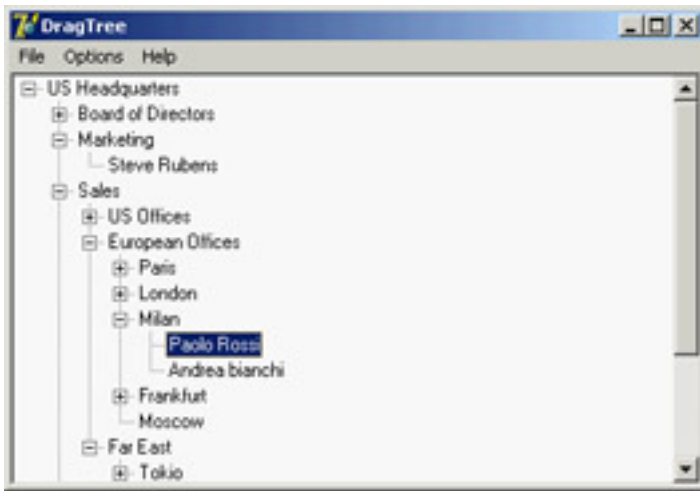
Using the `Add` and `AddChild` methods, you can build a complex structure at run time. To load the information, you can again use a `StringList` at run time, load a text file with the information, and parse it.

However, because the TreeView control has a `LoadFromFile` method, the `DragTree` and `QDragTree` examples use the following simpler code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.LoadFromFile (ExtractFilePath (Application.ExeName) +
    'TreeText.txt');
end;
```

The `LoadFromFile` method loads the data in a string list and checks the level of each item by looking at the number of tab characters. (If you are curious, see the `TTreeStrings.GetBufStart` method, which you can find in the `ComCtrls` unit in the VCL source code included in Delphi.) The data I've prepared for the TreeView is the organizational chart of a multinational company, as you can see in [Figure 5.13](#).





**Figure 5.13:** The DragTree example after loading the data and expanding the branches

Instead of expanding the node items one by one, you can also use the File → Expand All menu of this program, which calls the `FullExpand` method of the `TreeView` control or executes the equivalent code (in this specific case of a tree with a root item):

```
TreeView1.Items [0].Expand(True);
```

Besides loading the data, the program saves the data when it terminates, making the changes persistent. It also has a few menu items to customize the font of the `TreeView` control and change some other simple settings. The specific feature I've implemented in this example is support for dragging items and entire subtrees. I've set the `DragMode` property of the component to `dmAutomatic` and written the event handlers for the `OnDragOver` and `OnDragDrop` events.

In the first of the two handlers, the program makes sure the user is not trying to drag an item over a child item (which would be moved along with the item, leading to an infinite recursion):

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
  begin
    Accept := True;
    // determines source and target
    SourceNode := TreeView1.Selected;
    // look up the target parent chain
    while (TargetNode.Parent <> nil) and (TargetNode <> SourceNode) do
      TargetNode := TargetNode.Parent;
    // if source is found
    if TargetNode = SourceNode then
      // do not allow dragging over a child
      Accept := False;
  end
  else
    Accept := False;
end;
```

The effect of this code is that (except for the particular case you need to disallow) a user can drag a `TreeView` item over another item. Writing the code for moving the items is simple, because the `TreeView` control provides support for this operation through the `TTreeNode` class's `MoveTo` method:

```

procedure TForm1.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
    TargetNode, SourceNode: TTreeNode;
begin
    TargetNode := TreeView1.GetNodeAt (X, Y);
    if TargetNode <> nil then
        begin
            SourceNode := TreeView1.Selected;
            SourceNode.MoveTo (TargetNode, naAddChildFirst);
            TargetNode.Expand (False);
            TreeView1.Selected := TargetNode;
        end;
    end;

```

**Note** Among the demos shipping with Delphi is an interesting one showing a custom-draw TreeView control. The example is in the *CustomDraw* subdirectory.

## The Portable Version of DragTree

Because I use this program in numerous porting demonstrations, I've built a version you can compile as a native VCL application with Delphi and as a CLX application with Kylix. This is different from most other programs in this book, including the previous version of this same example, which can be ported to Delphi by using VisualCLX and also Qt on Windows. Following a different path once in a while can be instructive.

The first thing I had to do was use two different sets of `uses` statements, using conditional compilation. The unit of the PortableDragTree example begins as follows:

```

unit TreeForm;

interface

uses
    SysUtils, Classes,

    {$IFDEF LINUX}
        Qt, Libc, QGraphics, QControls, QForms, QDialogs,
        QStdCtrls, QComCtrls, QMenus, QTypes, QGrids;
    {$ENDIF}

    {$IFDEF MSWINDOWS}
        Windows, Graphics, Controls, Forms, Dialogs,
        StdCtrls, ComCtrls, Menus, Grids;
    {$ENDIF}

```

A similar conditional directive is used in the initial portion of the implementation section, to include the proper resource file for the form (the two resource files are different):

```

    {$IFDEF LINUX}
        {$R *.xfm}
    {$ENDIF}

    {$IFDEF MSWINDOWS}
        {$R *.dfm}
    {$ENDIF}

```

I've omitted some of the Windows-specific features anyway, so the only difference in code is in the `FormCreate` method. The program loads the data file from the user's default folder, not the same folder as the executable. Depending on the operating system, the user's folder is the home directory (and the hidden file has a starting dot) or the specific My Documents area (available with a special API call):

```

procedure TForm1.FormCreate(Sender: TObject);
var
    path: string;
begin
    {$IFDEF LINUX}
        filename := GetEnvironmentVariable('HOME') +
            '\.TreeText.txt';
    {$ELSE}
        SetLength (path, 100);
        ShGetSpecialFolderPath (Handle, PChar(path),
            CSIDL_PERSONAL, False);
        path := PChar (path); // fix string length
        filename := path + '\TreeText.txt';
    {$ENDIF}
    TreeView1.LoadFromFile (filename);
end;

```

## Custom Tree Nodes

Delphi 6 added a few features to the TreeView controls, including multiple selection (see the `MultiSelect` and `MultiSelectStyle` properties and the `Selections` array), improved sorting, and several new events. The key improvement, however, is letting the programmer determine the class of the tree view's node items. Having custom node items implies the ability to attach custom data to the nodes in a simple, object-oriented way. To support this technique, there is a new `AddNode` method for the `TTreeItems` class and a new specific event, `OnCreateNodesClass`. In the handler for this event, you return the class of the object to be created, which must inherit from `TTreeNode`.

This is a very common technique, so I've built an example to discuss it in detail. The `CustomNodes` example doesn't focus on a real-world case, but it shows a rather complex situation in which two different custom tree node classes are derived one from the other. The base class adds an `ExtraCode` property, mapped to virtual methods, and the subclass overrides one of these methods. For the base class, the `GetExtraCode` function simply returns the value; for the derived class, the value is multiplied to the parent node value. Here are the classes and this second method:

```

type
    TMyNode = class (TTreeNode)
    private
        FExtraCode: Integer;
    protected
        procedure SetExtraCode(const Value: Integer); virtual;
        function GetExtraCode: Integer; virtual;
    public
        property ExtraCode: Integer read GetExtraCode write SetExtraCode;
    end;

    TMySubNode = class (TMyNode)
    protected
        function GetExtraCode: Integer; override;
    end;

function TMySubNode.GetExtraCode: Integer;
begin
    Result := fExtraCode * (Parent as TMyNode).ExtraCode;
end;

```

With these custom tree node classes available, the program creates a tree of items, using the first type for the first-level nodes and the second class for the other nodes. Because you have only one `OnCreateNodeClass` event

handler, the program uses the class reference stored in a private field of the form (`CurrentNodeClass` of type `TTreeNodeClass`):

```
procedure TForm1.TreeView1CreateNodeClass(Sender: TCustomTreeView;  
    var NodeClass: TTreeNodeClass);  
begin  
    NodeClass := CurrentNodeClass;  
end;
```

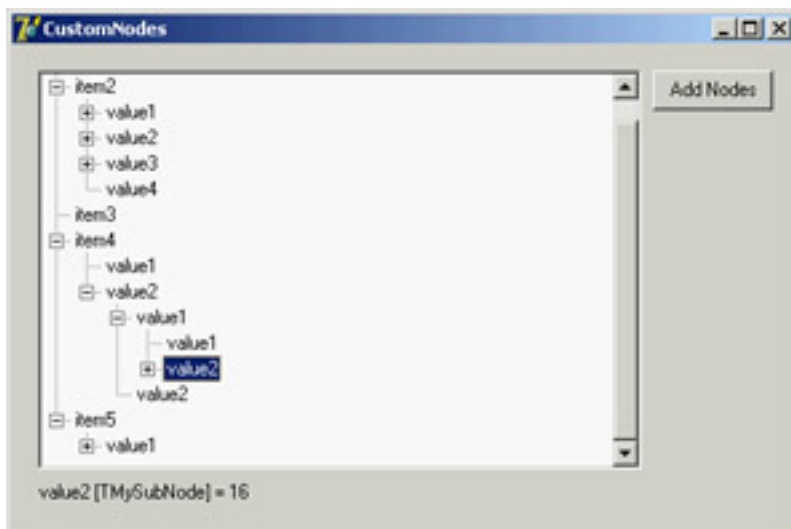
The program sets this class reference before creating nodes of each type—for example, with code like the following:

```
var  
    MyNode: TMyNode;  
begin  
    CurrentNodeClass := TMyNode;  
    MyNode := TreeView1.Items.AddChild (nil, 'item' + IntToStr (nValue))  
        as TMyNode;  
    MyNode.ExtraCode := nValue;
```

Once the entire tree has been created, when the user selects an item, you can cast its type to `TMyNode` and access the extra properties (but also methods and data):

```
procedure TForm1.TreeView1Click(Sender: TObject);  
var  
    MyNode: TMyNode;  
begin  
    MyNode := TreeView1.Selected as TMyNode;  
    Label1.Caption := MyNode.Text + ' [' + MyNode.ClassName + '] = ' +  
        IntToStr (MyNode.ExtraCode);  
end;
```

This is the code used by the `CustomNodes` example to display the description of the selected node in a label, as you can see in [Figure 5.14](#). Note that when you select an item within the tree, its value is multiplied for that of each parent node. There are certainly easier ways to obtain this effect, but having a tree view with item objects created from different classes of a hierarchy provides an object-oriented structure upon which you can base some very complex code.



**Figure 5.14:** The `CustomNodes` example has a tree view with node objects based on different custom classes, thanks to the `OnCreateNodes-Class` event.

## What's Next?

In this chapter, we have explored the foundations of the libraries available in Delphi for building user interfaces: the native-Windows VCL and the Qt-based CLX. I've discussed the `TControl` class, its properties, and its most important derived classes.

We explored some of the basic components available in Delphi, looking at both libraries. These components correspond to the standard Windows controls and some of the common controls, and they are extremely common in applications. You've also seen how to create main menus and pop-up menus and how to add extra graphics to some of these controls.

The next step is to explore in depth the elements of a complete user interface, discussing action lists and the Action Manager, and building some simple but complete examples. This is the topic of [Chapter 6](#); then, [Chapter 7](#) is devoted to forms.

## Chapter 6: Building the User Interface

### Overview

In [Chapter 5](#), I discussed the core concepts of the `TControl` class and its derived classes in the VCL and VisualCLX libraries. Then, I provided a rapid tour of the key controls you can use to build a user interface, including editing components, lists, range selectors, and more. This chapter discusses other controls used to define the overall design of a form, such as the `PageControl` and `TabControl`. After these components, I'll introduce toolbars and status bars, including some slightly advanced features. This will give you the foundation material for the rest of the chapter, which covers actions and the Action Manager architecture.

Modern Windows applications usually have multiple ways of giving a command, including menu items, toolbar buttons, shortcut menus, and so on. To separate the actual commands a user can give from their multiple representations in the user interface, Delphi uses the concept of *actions*. In recent Delphi versions, this architecture has been extended to make the construction of the user interface on top of actions totally visual. You can now also let program users customize this interface easily, as is true in many professional programs. Finally, Delphi 7 adds to the visual controls supporting the Action Manager architecture a better and more modern UI, supporting the XP look and feel. On Windows XP, you can create applications that adapt themselves to the active theme, thanks to a lot of new internal VCL code.

## Multiple-Page Forms

When you need to display a lot of information and controls in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages. You can use two controls to build a multiple-page application in Delphi:

- The *PageControl* component has tabs on one side and multiple pages (similar to panels) covering the rest of its surface. There is one page per tab, so you can simply place components on each page to obtain the proper effect both at design time and at run time.
- The *TabControl* has only the tab portion but offers no pages to hold the information. In this case, you'll want to use one or more components to mimic the *page change* operation, or you can place different forms within the tabs to simulate the pages.

A third related class, the *TabSheet*, represents a single page of the *PageControl*. This is not a stand-alone component and is not available on the Component Palette. You create a *TabSheet* at design time by using the shortcut menu of the *PageControl* or at run time by using methods of the same control.

**Note** Delphi still includes (in the Win 3.1 tab of the Component Palette) the *Notebook*, *TabSet*, and *TabbedNotebook* components introduced in 32-bit versions (that is, since Delphi 2). For any other purpose, the *PageControl* and *TabControl* components, which encapsulate Win32 common controls, provide a more modern user interface. In 32-bit versions of Delphi, the *TabbedNotebook* component was reimplemented using the Win32 *PageControl* internally, to reduce the code size and update the look.

## PageControls and TabSheets

As usual, instead of duplicating the Help system's list of properties and methods for the *PageControl* component, I've built an example that stretches the control's capabilities and allows you to change its behavior at run time. The example, called *Pages*, has a *PageControl* with three pages. The structure of the *PageControl* and the other key components appears in [Listing 6.1](#).

### Listing 6.1: Key Portions of the DFM of the Pages Example

---

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
object PageControl1: TPageControl
  ActivePage = TabSheet1
  Align = alClient
  HotTrack = True
  Images = ImageList1
  MultiLine = True
object TabSheet1: TTabSheet
  Caption = 'Pages'
  object Label3: TLabel
  object ListBox1: TListBox
end
object TabSheet2: TTabSheet
  Caption = 'Tabs Size'
  ImageIndex = 1
  object Label1: TLabel
  // other controls
end
object TabSheet3: TTabSheet
```

```

Caption = 'Tabs Text'
ImageIndex = 2
object Memol: TMemo
  Anchors = [akLeft, akTop, akRight, akBottom]
  OnChange = MemolChange
end
object BitBtnChange: TBitBtn
  Anchors = [akTop, akRight]
  Caption = '&Change'
end
end
end
object BitBtnPrevious: TBitBtn
  Anchors = [akRight, akBottom]
  Caption = '&Previous'
  OnClick = BitBtnPreviousClick
end
object BitBtnNext: TBitBtn
  Anchors = [akRight, akBottom]
  Caption = '&Next'
  OnClick = BitBtnNextClick
end
object ImageList1: TImageList
  Bitmap = {...}
end
end

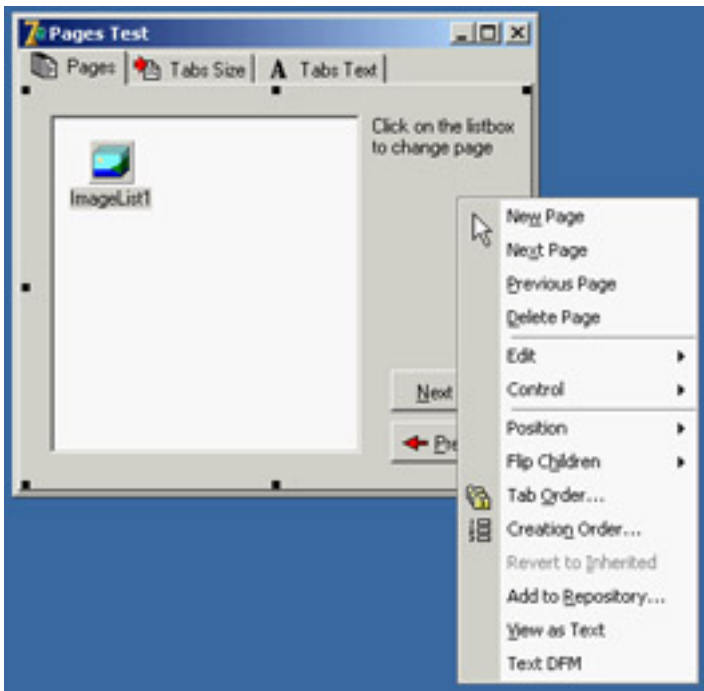
```

---

Notice that the tabs are connected to the bitmaps provided by an ImageList control and that some controls use the `Anchors` property to remain at a fixed distance from the right or bottom borders of the form. Even if the form doesn't support resizing (this would have been far too complex to set up with so many controls), the positions can change when the tabs are displayed on multiple lines (simply increase the length of the captions) or on the left side of the form.

Each TabSheet object has its own `Caption`, which is displayed as the sheet's tab. At design time, you can use the shortcut menu to create new pages and to move between pages. You can see the shortcut menu of the PageControl component in [Figure 6.1](#), together with the first page. This page holds a list box and a small caption, and it shares two buttons with the other pages.





**Figure 6.1:** The first sheet of the PageControl of the Pages example, with its shortcut menu

If you place a component on a page, it is available only in that page. How can you have the same component (in this case, two bitmap buttons) in each page, without duplicating it? Simply place the component on the form, outside the PageControl (or before aligning it to the client area), and then move it in front of the pages, calling the control → Bring To Front command from the form's shortcut menu. The two buttons I've placed in each page can be used to move back and forth between the pages and are an alternative to using the tabs. Here is the code associated with one of them:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
    PageControl1.SelectNextPage (True);
end;
```

The other button calls the same procedure, passing False as its parameter to select the previous page. Notice that there is no need to check whether you are on the first or last page, because the `SelectNextPage` method considers the last page to be the one before the first and will move you directly between those two pages.

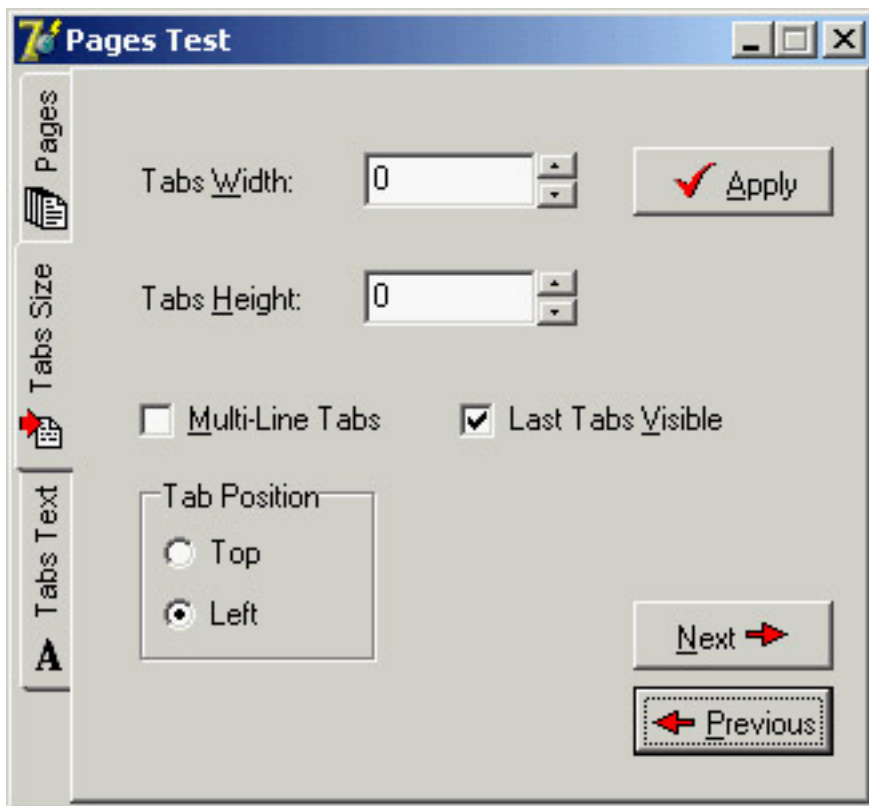
Now let's focus on the first page again. It has a list box, which at run time will hold the names of the tabs. If a user clicks an item in this list box, the current page changes. This is the third method available to change pages (after the tabs and the Next and Previous buttons). The list box is filled in the `FormCreate` method, which is associated with the `OnCreate` event of the form and copies the caption of each page (the `Page` property stores a list of `TabSheet` objects):

```
for I := 0 to PageControl1.PageCount - 1 do
    ListBox1.Items.Add (PageControl1.Pages.Caption);
```

When you click a list item, you can select the corresponding page:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
    PageControl1.ActivePage := PageControl1.Pages [ListBox1.ItemIndex];
end;
```

The second page hosts two edit boxes (connected with two UpDown components), two check boxes, and two radio buttons, as you can see in [Figure 6.2](#). The user can input a number (or choose it by clicking the up or down buttons with the mouse or pressing the Up or Down arrow key while the corresponding edit box has the focus), check the boxes and the radio buttons, and then click the Apply button to make the changes:



**Figure 6.2:** The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control.

```

procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
    // set tab width, height, and lines
    PageControll.TabWidth := StrToInt (EditWidth.Text);
    PageControll.TabHeight := StrToInt (EditHeight.Text);
    PageControll.MultiLine := CheckBoxMultiLine.Checked;
    // show or hide the last tab
    TabSheet3.TabVisible := CheckBoxVisible.Checked;
    // set the tab position
    if RadioButton1.Checked then
        PageControll.TabPosition := tpTop
    else
        PageControll.TabPosition := tpLeft;
end;

```

With this code, you can change the width and height of each tab (remember that 0 means the size is computed automatically from the space taken by each string). You can choose to have either multiple lines of tabs or two small arrows to scroll the tab area, and you can move the tabs to the left side of the window. The control also lets you place tabs on the bottom or on the right, but this program doesn't allow that, because it would make the placement of the other controls quite complex.

You can also hide the last tab on the PageControl, which corresponds to the TabSheet3 component. If you hide one of the tabs by setting its TabVisible property to False, you cannot reach that tab by clicking on the Next and Previous buttons, which are based on the SelectNextPage method. Instead, you should use the FindNextPage function, which will select that page even if the tab won't become visible. A call of FindNextPage method is shown in the following new version of the Next button's OnClick event handler:

```

procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
    PageControll.ActivePage := PageControll.FindNextPage (
        PageControll.ActivePage, True, False);
end;

```

```
end;
```

The last page has a memo component, again with the names of the pages (added in the `FormCreate` method). You can edit the names of the pages and click the Change button to change the text of the tabs, but only if the number of strings matches the number of tabs:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
  if Mem1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please', mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption := Mem1.Lines [I];
    BitBtnChange.Enabled := False;
end;
```

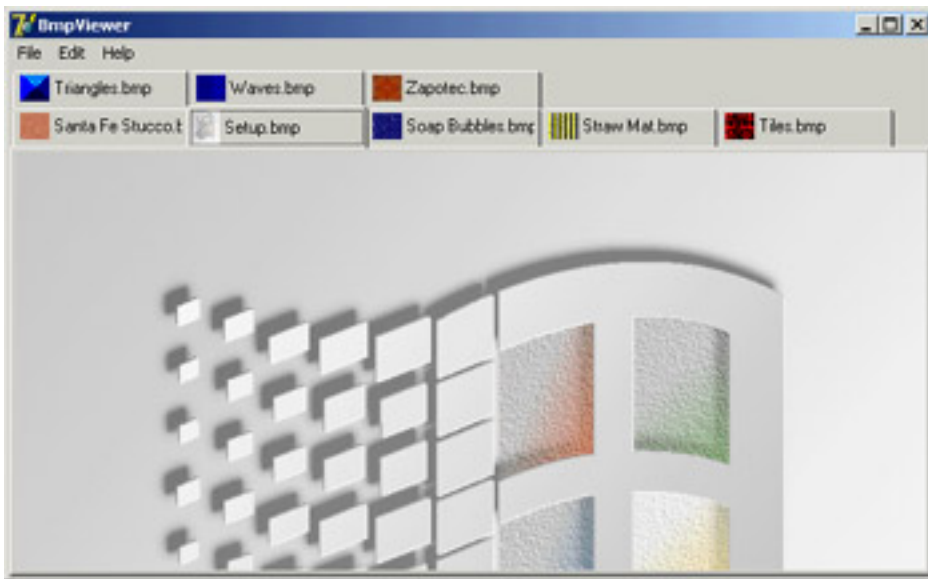
Finally, the last button, Add Page, allows you to add a new tab sheet to the page control, although the program doesn't add any components to it. The (empty) tab sheet object is created using the page control as its owner, but it won't work unless you also set the `PageControl` property. Before doing this, however, you should make the new tab sheet visible. Here is the code:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);
var
  strCaption: string;
  NewTabSheet: TTabSheet;
begin
  strCaption := 'New Tab';
  if InputQuery ('New Tab', 'Tab Caption', strCaption) then
    begin
      // add a new empty page to the control
      NewTabSheet := TTabSheet.Create (PageControl1);
      NewTabSheet.Visible := True;
      NewTabSheet.Caption := strCaption;
      NewTabSheet.PageControl := PageControl1;
      PageControl1.ActivePage := NewTabSheet;
      // add it to both lists
      Mem1.Lines.Add (strCaption);
      ListBox1.Items.Add (strCaption);
    end;
end;
```

**Tip** Whenever you write a form based on a `PageControl`, remember that the first page displayed at run time is the page you were in before the code was compiled. For example, if you are working on the third page and then compile and run the program, it will start with that page. A common way to solve this problem is to add a line of code in the `FormCreate` method to set the `PageControl` or notebook to the first page. This way, the current page at design time doesn't determine the initial page at run time.

## An Image Viewer with Owner-Draw Tabs

You can also use the `TabControl` and a dynamic approach, as described in the last example, in more general (and simpler) cases. Every time you need multiple pages that all have the same type of content, instead of replicating the controls in each page, you can use a `TabControl` and change its contents when a new tab is selected. This is what I've done in the multiple-page bitmap viewer example, called `BmpViewer`. The image that appears in the `TabControl` of this form, aligned to the whole client area, depends on the selection in the tab above it (as you can see in [Figure 6.3](#)).



**Figure 6.3:** The interface of the bitmap viewer in the BmpViewer example. Notice the owner-draw tabs.

At the beginning, the TabControl is empty. After selecting File → Open, the user can choose various files in the File Open dialog box, and the array of strings with the names of the files (the `Files` property of the `OpenDialog1` component) is added to the tabs (the `Tabs` property of `TabControl1`):

```
procedure TFormBmpViewer.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    begin
      TabControl1.Tabs.AddStrings (OpenDialog1.Files);
      TabControl1.TabIndex := 0;
      TabControl1Change (TabControl1);
    end;
end;
```

**Warning** The `Tabs` property of a `TabControl` in CLX is a collection, whereas in the VCL it is simply a string list.

After you display the new tabs, you have to update the image so that it matches the first tab. To accomplish this, the program calls the method connected with the `OnChange` event of the `TabControl`, which loads the file corresponding to the current tab in the image component:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (TabControl1.Tabs [TabControl1.TabIndex]);
end;
```

This example works, unless you select a file that doesn't contain a bitmap. The program will warn the user with a standard exception, ignore the file, and continue its execution.

The program also lets you paste the bitmap on the Clipboard (without immediately getting it, but only adding a tab that will perform the actual paste operation when selected) and copy the current bitmap to it. Clipboard support is available in Delphi via the global `Clipboard` object defined in the `ClipBrd` unit. For copying or pasting bitmaps, you can use the `Assign` method of the `TClipboard` and `TBitmap` classes. When you select the Edit → Paste command in the example, a new tab named `Clipboard` is added to the tab set (unless it is already present). Then the number of the new tab is used to change the active tab:

```
procedure TFormBmpViewer.Paste1Click(Sender: TObject);
var
  TabNum: Integer;
begin
  // try to locate the page
```

```

TabNum := TabControl1.Tabs.IndexOf ('Clipboard');
if TabNum < 0 then
    // create a new page for the Clipboard
    TabNum := TabControl1.Tabs.Add ('Clipboard');
// go to the Clipboard page and force repaint
TabControl1.TabIndex := TabNum;
TabControl1Change (Self);
end;

```

The Edit → Copy operation is as simple as copying the bitmap currently in the image control:

```
Clipboard.Assign (Image1.Picture.Graphic);
```

To account for the possible presence of the Clipboard tab, the code of the TabControl1Change method becomes:

```

procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
var
    TabText: string;
begin
    Image1.Visible := True;
    TabText := TabControl1.Tabs [TabControl1.TabIndex];
    if TabText <> 'Clipboard' then
        // load the file indicated in the tab
        Image1.Picture.LoadFromFile (TabText)
    else
        {if the tab is 'Clipboard' and a bitmap
         is available in the clipboard}
        if Clipboard.HasFormat (cf_Bitmap) then
            Image1.Picture.Assign (Clipboard)
        else
            begin
                // else remove the clipboard tab
                TabControl1.Tabs.Delete (TabControl1.TabIndex);
                if TabControl1.Tabs.Count = 0 then
                    Image1.Visible := False;
            end;
    end;

```

This program pastes the bitmap from the Clipboard each time you change the tab. The program stores only one image at a time, and it has no way to store the Clipboard bitmap. However, if the Clipboard content changes and the bitmap format is no longer available, the Clipboard tab is automatically deleted (as you can see in the previous listing). If no more tabs are left, the Image component is hidden.

An image can also be removed using either of two menu commands: Cut or Delete. Cut removes the tab after making a copy of the bitmap to the Clipboard. In practice, the Cut1Click method does nothing besides call the Copy1Click and Delete1Click methods. The Copy1Click method is responsible for copying the current image to the Clipboard; Delete1Click simply removes the current tab. Here is their code:

```

procedure TFormBmpViewer.Copy1Click(Sender: TObject);
begin
    Clipboard.Assign (Image1.Picture.Graphic);
end;

```

```

procedure TFormBmpViewer.Delete1Click(Sender: TObject);
begin
    with TabControl1 do
        begin
            if TabIndex >= 0 then

```

```

    Tabs.Delete (TabIndex);
    if Tabs.Count = 0 then
        Image1.Visible := False;
    end;
end;

```

One of the special features of the example is that the `TabControl` has the `OwnerDraw` property set to `True`. This means the control won't paint the tabs (which will be empty at design time) but will instead have the application do this, by calling the `OnDrawTab` event. In its code, the program displays the text vertically centered, using the `DrawText` API function. The text displayed is not the entire file path but only the filename. Then, if the text is not `None`, the program reads the bitmap the tab refers to and paints a small version of it in the tab itself. To accomplish this, the program uses the `TabBmp` object, which is of type `TBitmap` and is created and destroyed along with the form. The program also uses the `BmpSide` constant to position the bitmap and the text properly:

```

procedure TFormBmpViewer.TabControl1DrawTab(Control: TCustomTabControl;
    TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
    TabText: string;
    OutRect: TRect;
begin
    TabText := TabControl1.Tabs [TabIndex];
    OutRect := Rect;
    InflateRect (OutRect, -3, -3);
    OutRect.Left := OutRect.Left + BmpSide + 3;
    DrawText (Control.Canvas.Handle, PChar (ExtractFileName (TabText)),
        Length (ExtractFileName (TabText)), OutRect,
        dt_Left or dt_SingleLine or dt_VCenter);
    if TabText = 'Clipboard' then
        if Clipboard.HasFormat (cf_Bitmap) then
            TabBmp.Assign (Clipboard)
        else
            TabBmp.FreeImage
        else
            TabBmp.LoadFromFile (TabText);
    OutRect.Left := OutRect.Left - BmpSide - 3;
    OutRect.Right := OutRect.Left + BmpSide;
    Control.Canvas.StretchDraw (OutRect, TabBmp);
end;

```

The program has also support for printing the current bitmap, after showing a page preview form in which the user can select the proper scaling. This extra portion of the program I built for earlier editions of the book is not discussed in detail, but I've left the code in the program so you can examine it.

## The User Interface of a Wizard

Just as you can use a `TabControl` without pages, you can also take the opposite approach and use a `PageControl` without tabs. Now I'll focus on the development of the user interface of a wizard. In a wizard, you direct the user through a sequence of steps, one screen at a time, and at each step you typically offer the choice of proceeding to the next step or going back to correct input entered in a previous step. Instead of tabs that can be selected in any order, wizards typically offer `Next` and `Back` buttons to navigate. This won't be a complex example; its purpose is just to give you a few guidelines. The example is called `WizardUI`.

The starting point is to create a series of pages in a `PageControl` and set the `TabVisible` property of each `TabSheet` to `False` (while keeping the `Visible` property set to `True`). Since Delphi 5, you can also hide the tabs at design time. In this case, you'll need to use the shortcut menu of the page control, the Object Inspector's combo box, or the Object Tree View to move to another page, instead of the tabs. But why don't you want to see the tabs at design time? You can place controls on the pages and then place extra controls in front of the pages (as I've done in the example),

without seeing their relative positions change at run time. You might also want to remove the useless captions from the tabs; they take up space in memory and in the resources of the application.

In the first page, I've placed an image and a bevel control on one side and some text, a check box, and two buttons on the other side. Actually, the Next button is inside the page, and the Back button is over it (and is shared by all the pages). You can see this first page at design time in [Figure 6.4](#). The following pages look similar, with a label, check boxes, and buttons on the right side and nothing on the left.



**Figure 6.4:** The first page of the WizardUI example at design time

When you click the Next button on the first page, the program looks at the status of the check box and decides which page is the following one. I could have written the code like this:

```
procedure TForm1.btnNext1Click(Sender: TObject);  
begin  
    BtnBack.Enabled := True;  
    if CheckInprise.Checked then  
        PageControll.ActivePage := TabSheet2  
    else  
        PageControll.ActivePage := TabSheet3;  
    // move image and bevel  
    Bevell.Parent := PageControll.ActivePage;  
    Image1.Parent := PageControll.ActivePage;  
end;
```

After enabling the common Back button, the program changes the active page and finally moves the graphical portion to the new page. Because this code has to be repeated for each button, I've placed it in a method after adding a couple of extra features. This is the code:

```
procedure TForm1.btnNext1Click(Sender: TObject);  
begin  
    if CheckInprise.Checked then  
        MoveTo (TabSheet2)  
    else  
        MoveTo (TabSheet3);  
end;
```

```
procedure TForm1.MoveTo(TabSheet: TTabSheet);  
begin  
    // add the last page to the list  
    BackPages.Add (PageControll.ActivePage);  
    BtnBack.Enabled := True;  
    // change page  
    PageControll.ActivePage := TabSheet;  
    // move image and bevel  
    Bevell.Parent := PageControll.ActivePage;
```

```

Image1.Parent := PageControll.ActivePage;
end;

```

Besides the code I've already explained, the `MoveTo` method adds the last page (the one before the page change) to a list of visited pages, which behaves like a stack. The `BackPages` object of the `TList` class is created as the program starts, and the last page is always added to the end. When the user clicks the Back button, which is not dependent on the page, the program extracts the last page from the list, deletes its entry, and moves to that page:

```

procedure TForm1.btnBackClick(Sender: TObject);
var
    LastPage: TTabSheet;
begin
    // get the last page and jump to it
    LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
    PageControll.ActivePage := LastPage;
    // delete the last page from the list
    BackPages.Delete (BackPages.Count - 1);
    // eventually disable the back button
    BtnBack.Enabled := not (BackPages.Count = 0);
    // move image and bevel
    Bevel1.Parent := PageControll.ActivePage;
    Image1.Parent := PageControll.ActivePage;
end;

```

With this code, the user can move back several pages until the list is empty, at which point you disable the Back button. You need to deal with a complication: While moving from a particular page, you know which pages are "next" and "previous," but you don't know which page you we came from, because there can be multiple paths to reach a page. Only by keeping track of the movements with a list can you reliably go back.

The rest of the program code, which simply shows some website addresses, is very simple. The good news is that you can reuse the navigational structure of this example in your own programs and modify only the graphical portion and the content of the pages. Because most of the programs' labels show HTTP addresses, a user can click a label to open the default browser showing that page. You accomplish this by extracting the HTTP address from the label and calling the `ShellExecute` function:

```

procedure TForm1.LabelLinkClick(Sender: TObject);
var
    Caption, StrUrl: string;
begin
    Caption := (Sender as TLabel).Caption;
    StrUrl := Copy (Caption, Pos ('http://', Caption), 1000);
    ShellExecute (Handle, 'open', PChar (StrUrl), '', '', sw_Show);
end;

```

This method is hooked to the `OnClick` event of many labels on the form, which have been turned into *links* by setting the `Cursor` to a hand. This is one of the labels:

```

object Label2: TLabel
    Cursor = crHandPoint
    Caption = 'Main site: http://www.borland.com'
    OnClick = LabelLinkClick
end

```



## The ToolBar Control

To create a toolbar, Delphi includes a specific component that encapsulates the corresponding Win32 common control or the corresponding Qt widget in VisualCLX. This component provides a toolbar with its own buttons, and it has many advanced capabilities. You place the component on a form and then use the component editor (the shortcut menu activated by a right-click) to create buttons and separators.

The toolbar is populated with objects of the `TToolBarButton` class. These objects have a fundamental property, `Style`, which determines their behavior:

- The `tbsButton` style indicates a standard pushbutton.
- The `tbsCheck` style indicates a button with the behavior of a check box, or that of a radio button if the button is grouped with the others in its block (determined by the presence of separators).
- The `tbsDropDown` style indicates a drop-down button (a sort of combo box). The drop-down portion can be easily implemented in Delphi by connecting a `PopupMenu` control to the `DropDownMenu` property of the control.
- The `tbsSeparator` and `tbsDivider` styles indicate separators with no or different vertical lines (depending on the `Flat` property of the toolbar).

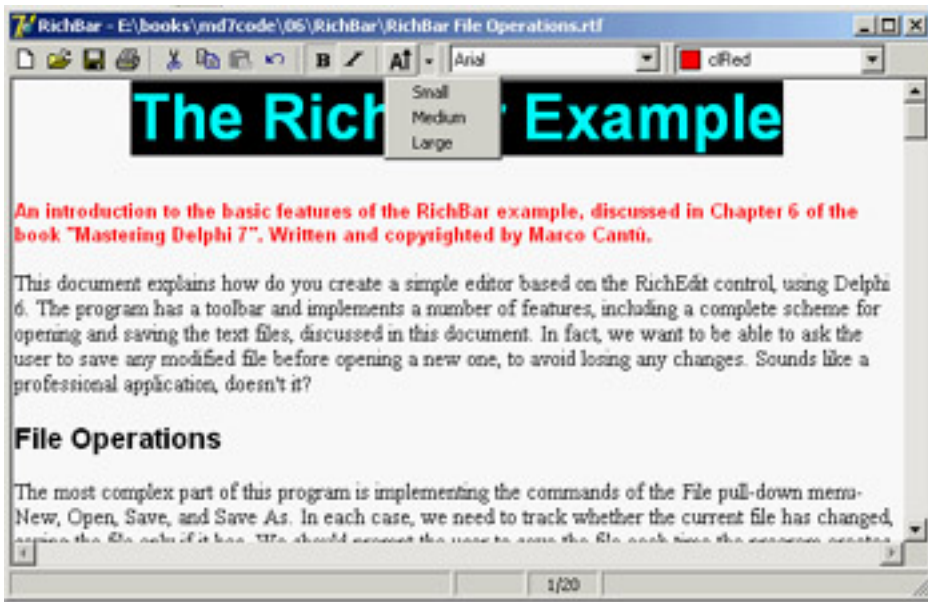
To create a graphic toolbar, you can add an `ImageList` component to the form, load some bitmaps into it, and then connect the `ImageList` with the `Images` property of the toolbar. By default, the images will be assigned to the buttons in the order they appear, but you can change this behavior quite easily by setting the `ImageIndex` property of each toolbar button. You can prepare further `ImageList`s for special button conditions and assign them to the `DisabledImages` and `HotImages` properties of the toolbar. The first group is used for the disabled buttons; the second is used for the button currently under the mouse.

**Note** In a nontrivial application, you will generally create toolbars using an `ActionList` or the recent Action Manager architecture, discussed later in this chapter. In this case, you'll attach little behavior to the toolbar buttons, because their properties and events will be managed by the action components. Moreover, you'll end up using a toolbar of the specific `TActionToolBar` class.

## The RichBar Example

As an example of the use of a toolbar, I've built the RichBar application, which has a RichEdit component you can operate by using the toolbar. The program has buttons for loading and saving files, for copy and paste operations, and to change some of the attributes of the current font.

I won't cover the many details of the RichEdit control's features, which I briefly discussed in a [previous chapter](#), nor discuss the details of this application, which has quite a lot of code. All I'll do is focus on features specific to the ToolBar used by the example and visible in [Figure 6.5](#). This toolbar has buttons, separators, and even a drop-down menu and two combo boxes (discussed in the [next section](#)).



**Figure 6.5:** The RichBar example's toolbar. Notice the drop-down menu.

The various buttons implement features, including opening and saving text files—the program asks the user to save any modified file before opening a new one, to avoid losing any changes. The file-handling portion of the program is quite complex but it is worth exploring, because many file-based applications will use similar code. More details are available in the file `RichBar File Operations.rtf` with the source code for this example, a file you can open with the RichBar program itself.

Besides file operations, the program supports copy and paste operations and font management. The copy and paste operations don't require an interaction with the VCL `Clipboard` object, because the component can handle them with simple commands like these:

```
RichEdit.CutToClipboard;
RichEdit.CopyToClipboard;
RichEdit.PasteFromClipboard;
RichEdit.Undo;
```

It is a little more advanced to know when these operations (and the corresponding buttons) should be enabled. You can enable Copy and Cut buttons when some text is selected, in the `OnSelectionChange` event of the RichEdit control:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
    btnCut.Enabled := RichEdit.SelLength > 0;
    btnCopy.Enabled := btnCut.Enabled;
end;
```

The Copy operation cannot be determined by an action of the user, because it depends on the content of the Clipboard, which is also influenced by other applications. One approach is to use a timer and check the Clipboard content from time to time. A better approach is to use the `OnIdle` event of the `Application` object (or the `ApplicationEvents` component). Because the RichEdit control supports multiple Clipboard formats, the code cannot simply look at those, but should ask the component itself, using a low-level feature not surfaced by the Delphi control:

```
procedure TFormRichNote.ApplicationEventsIdle(Sender: TObject;
    var Done: Boolean);
begin
    // update toolbar buttons
    btnPaste.Enabled := RichEdit.Perform (em_CanPaste, 0, 0) <> 0;
end;
```

Basic font management is given by the Bold and Italic buttons, which have similar code. The Bold button toggles the relative attribute from the selected text (or changes the style at the current edit position):

```

procedure TFormRichNote.BoldExecute(Sender: TObject);
begin
  with RichEdit.SelAttributes do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
end;

```

Again, the current status of the button is determined by the current selection, so you'll need to add the following line to the RichEditSelectionChange method:

```
tbtnBold.Down := fsBold in RichEdit.SelAttributes.Style;
```

## A Menu and a Combo Box in a Toolbar

Besides a series of buttons, the RichBar example has a drop-down menu and a couple of combo boxes, a feature shared by many common applications. The drop-down button allows selection of the font size, and the combo boxes allow rapid selection of the font family and the font color. This second combo is built using a ColorBox control.

The Size button is connected to a PopupMenu component (called SizeMenu) using the DropdownMenu property. A user can click the button, firing its OnClick event as usual, or select the drop-down arrow, open the pop-up menu (see again [Figure 6.5](#)), and choose one of its options. This case has three possible font sizes, per the menu definition:

```

object SizeMenu: TPopupMenu
  object Small1: TMenuItem
    Tag = 10
    Caption = 'Small'
    OnClick = SetFontSize
  end
  object Medium1: TMenuItem
    Tag = 16
    Caption = 'Medium'
    OnClick = SetFontSize
  end
  object Large1: TMenuItem
    Tag = 32
    Caption = 'Large'
    OnClick = SetFontSize
  end
end

```

Each menu item has a tag indicating the actual size of the font, activated by a shared event handler:

```

procedure TFormRichNote.SetFontSize(Sender: TObject);
begin
  RichEdit.SelAttributes.Size := (Sender as TMenuItem).Tag;
end;

```

The Toolbar control is a full-featured control container, so you can take an edit box, a combo box, and other controls and place them directly inside the toolbar. The combo box in the toolbar is initialized in the FormCreate method, which extracts the screen fonts available in the system:

```

ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (RichEdit.Font.Name)

```

The combo box initially displays the name of the default font used in the RichEdit control, which is set at design time. This value is recomputed each time the current selection changes, using the font of the selected text, along with the

current color for the ColorBox:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);  
begin  
    ComboFont.ItemIndex :=  
        ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name);  
    ColorBox1.Selected := RichEdit.SelAttributes.Color;  
end;
```

When a new font is selected from the combo box, the reverse action takes place. The text of the current combo box item is assigned as the name of the font for any text selected in the RichEdit control:

```
RichEdit.SelAttributes.Name := ComboFont.Text;
```

The selection of a color in the ColorBox activates similar code.

## A Simple Status Bar

Building a status bar is even simpler than building a toolbar. Delphi includes a specific StatusBar component, based on the corresponding Windows common control (a similar control is available in VisualCLX). This component can be used almost like a panel when its `SimplePanel` property is set to `True`. In this case, you can use the `SimpleText` property to output some text. The real advantage of this component, however, is that it allows you to define a number of subpanels by activating its `Panels` property editor. (You can also display this property editor by double-clicking the status bar control or perform the same operations using the Object TreeView.) Each subpanel has its own graphical attributes, which you can customize using the Object Inspector. Another feature of the status bar component is the "size grip" area added to the lower-right corner of the bar, which is useful for resizing the form. This is a typical element of the Windows user interface, and you can partially control it with the `SizeGrip` property (it auto-disables when the form is not resizable).

A status bar has various uses. The most common is to display information about the menu item currently selected by the user. In addition, a status bar often displays other information about the status of a program: the position of the cursor in a graphical application, the current line of text in a word processor, the status of the lock keys, the time and date, and so on. To show information on a panel, you use its `Text` property, generally in an expression like this:

```
StatusBar1.Panels[1].Text := 'message';
```

In the RichBar example, I've built a status bar with three panels, for command hints, the status of the Caps Lock key, and the current editing position. The StatusBar component of the example actually has four panels—you need to define the fourth in order to delimit the area of the third panel. The last panel is always large enough to cover the remaining surface of the status bar.

**Tip** Again, for more detail about the RichBar program, see the RTF file in the example's source code. Notice also that because the hints are to be displayed in the first panel of the status bar, I could have simplified the code by using the `AutoHint` property. I preferred showing the more detailed code, so you'll be able to customize it.

The panels are not independent components, so you cannot access them by name, only by position as in the preceding code snippet. A good solution to improve the readability of a program is to define a constant for each panel you want to use, and then use these constants when referring to the panels. This is my sample code:

```
const  
    sbpMessage = 0;  
    sbpCaps = 1;  
    sbpPosition = 2;
```

In the first panel of the status bar, I want to display the toolbar button's hint message. The program obtains this effect by handling the application's `OnHint` event, again using the ApplicationEvents component, and copying the current value of the application's `Hint` property to the status bar:

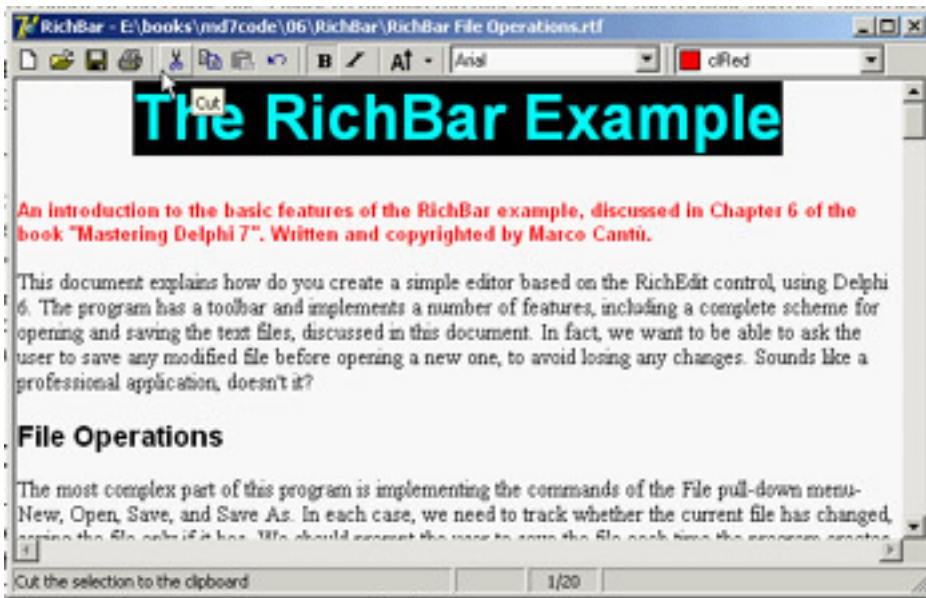
```
procedure TFormRichNote.ApplicationEvents1Hint (Sender: TObject);  
begin
```

```
StatusBar1.Panels[spbMessage].Text := Application.Hint;
end;
```

By default, this code displays in the status bar the same text of the fly-by hints, which aren't generated for menu items. You can use the `Hint` property to specify different strings for the two cases, by writing a string divided into two portions by a separator: the pipe (`|`) character. For example, you might enter the following as the value of the `Hint` property:

```
'New|Create a new document'
```

The first portion of the string, *New*, is used by fly-by hints, and the second portion, *Create a new document*, by the status bar. You can see an example in [Figure 6.6](#).



**Figure 6.6:** The StatusBar of the RichBar example displays a more detailed description than the fly-by hint.

**Tip** When the hint for a control is made up of two strings, you can use the `GetShortHint` and `GetLongHint` methods to extract the first (short) and second (long) substrings from the string you pass as a parameter, which is usually the value of the `Hint` property.

The second panel displays the status of the Caps Lock key, obtained by calling the `GetKeyState` API function, which returns a state number. If the low-order bit of this number is set (that is, if the number is odd), then the key is pressed. I've decided to check this state when the application is idle, so the test is executed every time a key is pressed but also as soon as a message reaches the window (in case the user changes this setting while working with another program). I've added to the `ApplicationEvents1Idle` handler a call to the custom `CheckCapslock` method, implemented as follows:

```
procedure TFormRichNote.CheckCapslock;
begin
  if Odd (GetKeyState (VK_CAPITAL)) then
    StatusBar1.Panels[spbCaps].Text := 'CAPS'
  else
    StatusBar1.Panels[spbCaps].Text := '';
end;
```

Finally, the program uses the third panel to display the current cursor position (measured in lines and characters per line) every time the selection changes. Because the `CaretPos` values are zero-based (that is, the upper-left corner is line 0, character 0), I've added one to each value to make them more reasonable for a casual user:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ...
```

```
// update the position in the status bar
StatusBar.Panels[sbpPosition].Text := Format ('%d/%d',
  [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Themes and Styles

In the past, a GUI-based operating system dictated all the elements of the user interface for programs running on it. In recent years, Linux began to allow users to customize the look-and-feel of both the main windows of applications and user interface controls, like buttons. The same idea (often indicated by the term *skin*) has appeared in numerous programs with such a positive impact that even Microsoft has begun to integrate it (first in programs and then in the entire operating system).

### CLX Styles

As I mentioned, on Linux (on XWindow, to be more precise) the user can generally choose the user interface style of the controls. This approach is fully supported by Qt and by the KDE system built on top of it. Qt offers a few basic styles, such as the Windows look-and-feel, the Motif style, and others. A user can also install new styles in the system and make them available to applications.

**Note** The styles I'm discussing here refer to the user interface of the controls, not of the forms and their borders. This is generally configurable on Linux systems but is technically a separate element of the user interface.

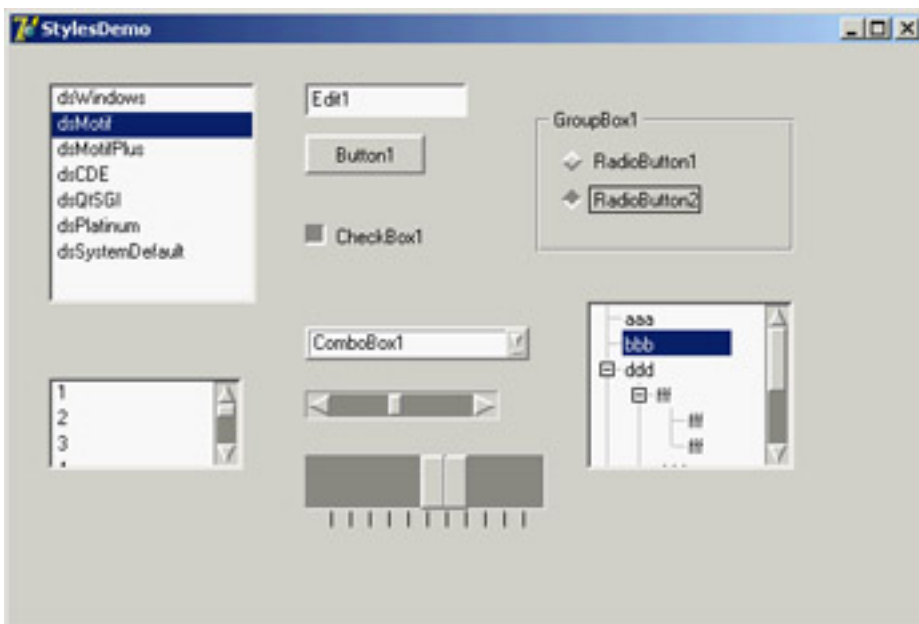
Because this technique is embedded in Qt, it is also available on the Windows version of the library; CLX makes it available to Delphi developers, so that an application can have a Motif look-and-feel on a Microsoft operating system. The CLX `Application` global object has a `Style` property you can use to set a custom style or a default one, indicated by the `DefaultStyle` subproperty. For example, you can select a Motif look-and-feel with this code:

```
Application.Style.DefaultStyle := dsMotif;
```

In the `StylesDemo` program, I've added, among various sample controls, a list box with the names of the default styles, as indicated in the `TDefaultStyle` enumeration, and this code for its `OnDblClick` event:

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  Application.Style.DefaultStyle := TDefaultStyle (ListBox1.ItemIndex);
end;
```

The effect is that, by double-clicking the list box, you can change the current application style and immediately see its effect on screen, as demonstrated in [Figure 6.7](#).



**Figure 6.7:** The `StylesDemo` program, a Windows application that currently has an unusual Motif layout



## Windows XP Themes

With the release of Windows XP, Microsoft has introduced a new, separate version of the common controls library. The old library is still available for compatibility reasons, so that a program running on XP can choose which of the two libraries it wants to use. The new common controls library's main difference is that it doesn't have a fixed rendering engine, but relies on the XP theme engine and delegates the user interface of the controls to the current theme.

In Delphi 7, the VCL fully supports themes, due to a lot of internal code and to the themes management library originally developed by Mike Lischke. Some of these new rendering features are used by the visual controls of the Action Manager architecture, independently of the operating system you are running on. However, full theme support is available only on an operating system that has this feature—at the moment, Windows XP.

Even on XP, Delphi applications use the traditional approach by default. To support XP themes, you must include a manifest file in the program. You can do so multiple ways:

- Place a manifest file in the same folder as the application. This is an XML file indicating the identity and the dependencies of the program. The file has the same name as the executable program with an extra `.manifest` extension at the end (as in `MyProgram.exe.manifest`). You can see a sample of such a file in [Listing 6.2](#).
- Add the same information in a resource file compiled within the application. You have to write a resource file that includes a manifest file. In Delphi 7, the VCL has a `WindowsXP.res` compiled resource file, which is obtained by recompiling the `WindowsXP.rc` file available among the VCL source files. The resource file includes the `sample.manifest` file, again available among the VCL source files.
- Use the `XpManifest` component, which Borland has added to Delphi 7 to further simplify this task. As you drop this do-nothing component in a program's form, Delphi will automatically include its `XPMAN` unit, which imports the VCL resource file mentioned earlier.

**Warning** When you remove the `XpManifest` component from an application, you also have to delete the `XPMAN` unit from the `uses` statement manually—Delphi won't do it for you. If you fail to do so, even without the `XpManifest` component, the program will still bind in the manifest resource file. Using the unit is what really matters (which really makes me wonder why Borland chose to create the component instead of simply providing the unit or the related resource file; by the way the component isn't documented at all).

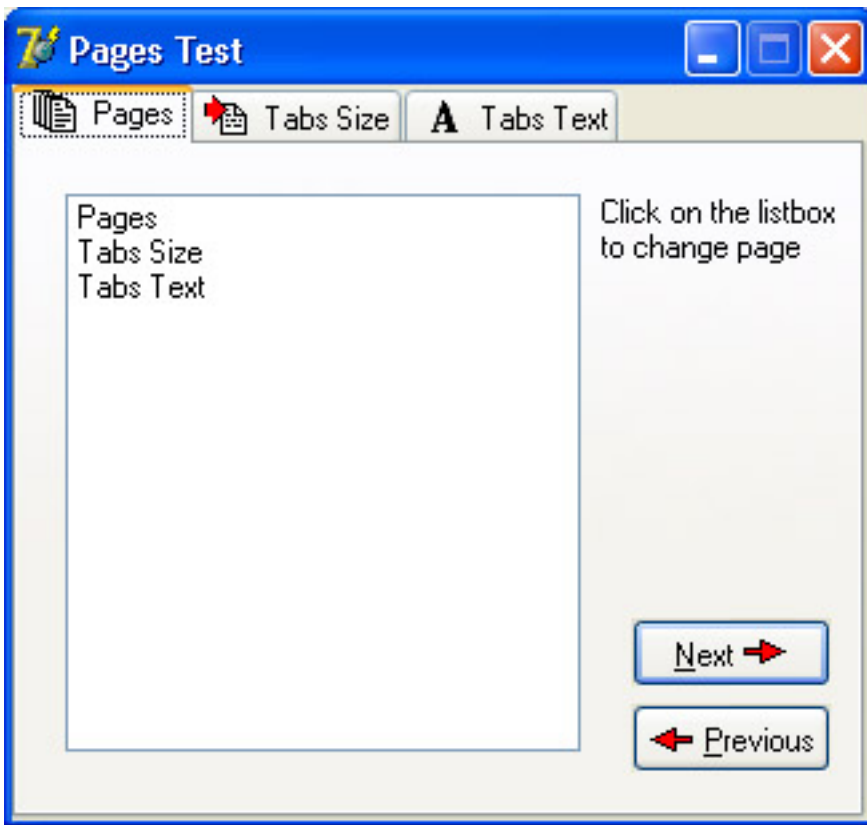
### Listing 6.2: A Sample Manifest File (`pages.exe.manifest`)

---

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="Pages.exe"
    type="win32"
  />
  <description>Mastering Delphi Demo</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        processorArchitecture="X86"
        publicKeyToken="6595b64144ccf1df"
        language="*"
      />
    </dependentAssembly>
  </dependency>
</assembly>
```



As a demo, I've added the manifest file from [Listing 6.2](#) to the folder of the Pages example discussed at the beginning of this chapter. By running it on Windows XP with the standard XP theme, you'll obtain output similar to that shown in [Figure 6.8](#). You can compare this to [Figures 6.1](#) and [6.2](#), which display the same program under Windows 2000.



**Figure 6.8:** The Pages example uses the current Windows XP theme, as it includes a manifest file (compare the figure with [6.1](#))

## The ActionList Component

Delphi's event architecture is very open: You can write a single event handler and connect it to the `OnClick` events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, because the event handler can use the `Sender` parameter to refer to the object that fired the event. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, then every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

To overcome this problem, Delphi includes an event-handling architecture based on actions. An *action* (or command) both indicates the operation to do when a menu item or button is clicked and determines the status of all the elements connected to the action. The connection of the action with the user interface of the linked controls is very important and should not be underestimated, because it is where you can get the real advantages of this architecture.

There are many players in this event-handling architecture. The central role is certainly played by the action objects. An action object has a name, like any other component, and other properties that will be applied to the linked controls (called *action clients*). These properties include the `Caption`, the graphical representation (`ImageIndex`), the status (`Checked`, `Enabled`, and `Visible`), and the user feedback (`Hint` and `HelpContext`). There is also the `Shortcut` and a list of `SecondaryShortcuts`, the `AutoCheck` property for two-state actions, the help support properties, and a `Category` property used to arrange actions in logical groups.

The base class for all action objects is `TBasicAction`, which introduces the abstract core behavior of an action, without any specific binding or correction (not even to menu items or controls). The derived `TContainedAction` class introduces properties and methods that enable actions to appear in an action list or action manager. The further-derived `TCustomAction` class introduces support for the properties and methods of menu items and controls that are linked to action objects. Finally, there is the derived ready-to-use `TAction` class.

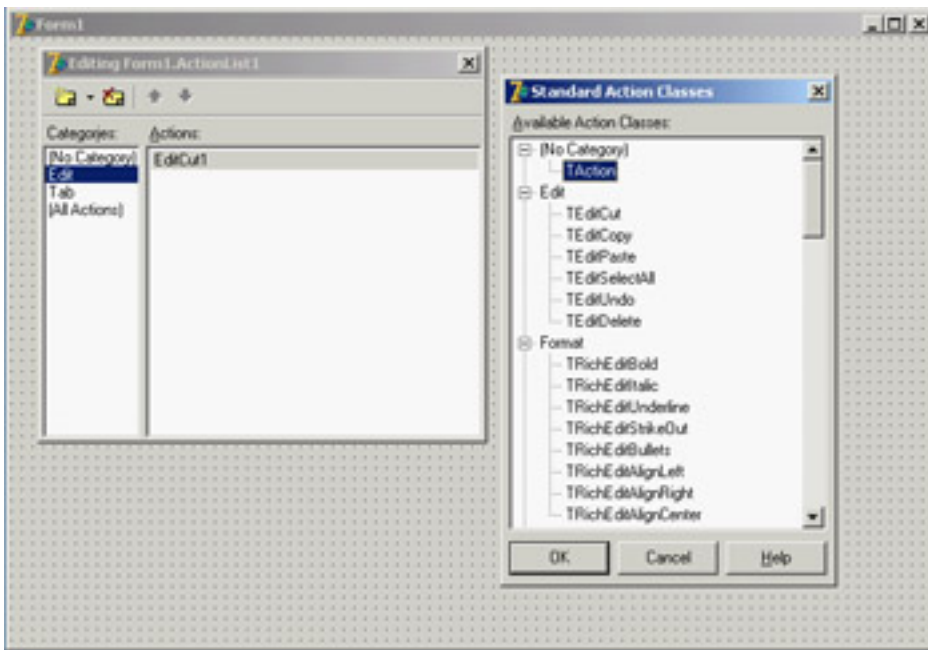
Each action object is connected to one or more client objects through an `ActionLink` object. Multiple controls, possibly of different types, can share the same action object, as indicated by their `Action` property. Technically, the `ActionLink` objects maintain a bidirectional connection between the client object and the action. The `ActionLink` object is required because the connection works in both directions. An operation on the object (such as a click) is forwarded to the action object and results in a call to its `OnExecute` event; an update to the status of the action object is reflected in the connected client controls. In other words, one or more client controls can create an `ActionLink`, which registers itself with the action object.

You should not set the properties of the client controls you connect with an action, because the action will override the property values of the client controls. For this reason, you should generally write the actions first and then create the menu items and buttons you want to connect with them. Note also that when an action has no `OnExecute` handler, the client control is automatically disabled (or grayed), unless the `DisableIfNoHandler` property is set to `False`.

The client controls connected to actions are usually menu items and various types of buttons (pushbuttons, check boxes, radio buttons, speed buttons, toolbar buttons, and the like), but nothing prevents you from creating new components that hook into this architecture. Component writers can even define new actions, as we'll do in [Chapter 9](#) ("Writing Delphi Components"), and new link action objects.

Besides a client control, some actions can also have a target component. Some predefined actions hook to a specific target component. Other actions automatically look for a target component in the form that supports the given action, starting with the active control.

Finally, the action objects are held by an `ActionList` or `ActionManager` component, the only class of the basic architecture that shows up on the Component Palette. The action list receives the execute actions that aren't handled by the specific action objects, firing the `OnExecuteAction`. If even the action list doesn't handle the action, Delphi calls the `OnExecuteAction` event of the `Application` object. The `ActionList` component has a special editor you can use to create several actions, as you can see in [Figure 6.9](#).



**Figure 6.9:** The ActionList component editor, with a list of predefined actions you can use

In the editor, actions are displayed in groups, as indicated by their `Category` property. By simply setting this property to a new value, you instruct the editor to introduce a new category. These categories are basically logical groups, although in some cases a group of actions can work only on a specific type of target component. You might want to define a category for every pull-down menu or group them in some other logical way.

## Predefined Actions in Delphi

With the action list and the ActionManager editor, you can create a new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box, as shown in [Figure 6.9](#). There are many predefined actions, which can be divided into logical groups:

**File Actions** Include open, save as, open with, run, print setup, and exit.

**Edit Actions** Illustrated in the next example. They include cut, copy, paste, select all, undo, and delete.

**RichEdit Actions** Complement the edit actions for RichEdit controls and include bold, italic, underline, strikeout, bullets, and various alignment actions.

**MDI Window Actions** Demonstrated in [Chapter 8](#), "The Architecture of Delphi Applications," as we examine the Multiple Document Interface approach. They include all the most common MDI operations: arrange, cascade, close, tile (horizontally or vertically), and minimize all.

**Dataset Actions** Relate to database tables and queries and will be discussed in [Chapter 13](#). There are many dataset actions, representing all the main operations you can perform on a dataset. Delphi 7 adds to the core dataset actions a group of actions specifically tailored to the ClientDataSet component, including apply, revert, and undo. I'll talk more about these actions in [Chapter 13](#) (where I'll cover database programming in general and the ClientDataSet component in particular) and [Chapter 14](#) (in which I'll discuss updating database data).

**Help Actions** Allow you to activate the contents page or index of the Help file attached to the application.

**Search Actions** Include find, find first, find next, and replace.

**Tab and Page Control Actions** Include previous page and next page navigation.

**Dialog Actions** Activate color, font, open, save, and print dialogs.

**List Actions** Include clear, copy, move, delete, and select all. These actions let you interact with a list control. Another group of actions, including static list, virtual list, and some support classes, allow the definition of lists that can be connected to a user interface. More on this topic is in the section "[Using List Actions](#)" toward the end of this chapter.

**Internet Actions** Include browse URL, download URL, and send mail actions.

**Tools Actions** Include only the dialog to customize the action bars.

In addition to handling the `OnExecute` event of the action and changing the status of the action to affect the client controls' user interface, an action can handle the `OnUpdate` event, which is activated when the application is idle. This gives you the opportunity to check the status of the application or the system and change the user interface of the controls accordingly. For example, the standard `PasteEdit` action enables the client controls only when the Clipboard contains some text.

## Actions in Practice

Now that you understand the main ideas behind this important Delphi feature, let's try an example. The program is called `Actions`, and it demonstrates a number of features of the action architecture. I began building it by placing a new `ActionList` component in its form and adding the three standard edit actions and a few custom ones. The form also has a panel with some speed buttons, a main menu, and a `Memo` control (the automatic target of the edit actions). [Listing 6.3](#) is the list of the actions, extracted from the DFM file.

### Listing 6.3: The Actions of the Actions Example

---

```
object ActionList1: TActionList
  Images = ImageList1
  object ActionCopy: TEditCopy
    Category = 'Edit'
    Caption = '&Copy'
    ShortCut = <Ctrl+C>
  end
  object ActionCut: TEditCut
    Category = 'Edit'
    Caption = 'Cu&t'
    ShortCut = <Ctrl+X>
  end
  object ActionPaste: TEditPaste
    Category = 'Edit'
    Caption = '&Paste'
    ShortCut = <Ctrl+V>
  end
  object ActionNew: TAction
    Category = 'File'
    Caption = '&New'
    ShortCut = <Ctrl+N>
    OnExecute = ActionNewExecute
  end
  object ActionExit: TAction
    Category = 'File'
    Caption = 'E&xit'
```

```

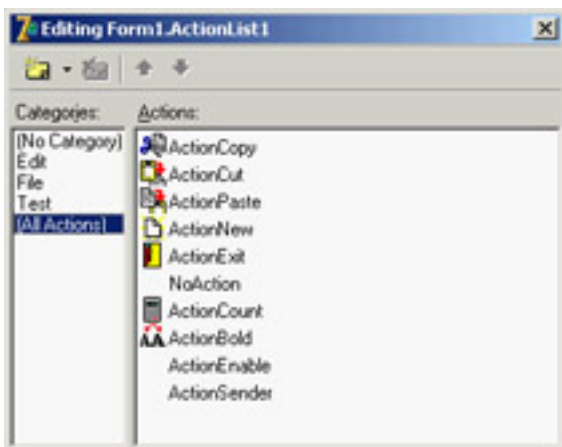
ShortCut = <Alt+F4>
OnExecute = ActionExitExecute
end
object NoAction: TAction
  Category = 'Test'
  Caption = '&No Action'
end
object ActionCount: TAction
  Category = 'Test'
  Caption = '&Count Chars'
  OnExecute = ActionCountExecute
  OnUpdate = ActionCountUpdate
end
object ActionBold: TAction
  Category = 'Edit'
  AutoCheck = True
  Caption = '&Bold'
  ShortCut = <Ctrl+B>
  OnExecute = ActionBoldExecute
end
object ActionEnable: TAction
  Category = 'Test'
  Caption = '&Enable NoAction'
  OnExecute = ActionEnableExecute
end
object ActionSender: TAction
  Category = 'Test'
  Caption = 'Test &Sender'
  OnExecute = ActionSenderExecute
end
end
end

```

---

**Note** The shortcut keys are stored in the DFM files using virtual key numbers, which also include values for the Ctrl and Alt keys. In this and other listings throughout the book, I've replaced the numbers with the literal values, enclosing them in angle brackets.

All these actions are connected to the items of a MainMenu component and some of them also to the buttons of a Toolbar control. Notice that the images selected in the ActionList control affect the actions in the editor only, as you can see in [Figure 6.10](#). In order for the ImageList images to show up in the menu items and in the toolbar buttons, you must also select the image list in the MainMenu and in the Toolbar components.



**Figure 6.10:** The ActionList editor of the Actions example

The three predefined actions for the Edit menu don't have associated handlers, but these special objects have internal code to perform the related action on the active edit or memo control. These actions also enable and disable themselves, depending on the content of the Clipboard and on the existence of selected text in the active edit control. Most other actions have custom code, except the `NoAction` object; because it has no code, the menu item and the button connected with this command are disabled, even if the `Enabled` property of the action is set to `True`.

I've added to the example and to the Test menu another action that enables the menu item connected to the `NoAction` object:

```
procedure TForm1.ActionEnableExecute(Sender: TObject);  
begin  
    NoAction.DisableIfNoHandler := False;  
    NoAction.Enabled := True;  
    ActionEnable.Enabled := False;  
end;
```

Setting `Enabled` to `True` produces the effect for only a short time, unless you set the `Disable-IfNoHandler` property as discussed in the [previous section](#). Once this operation is done, you disable the current action, because there is no need to issue the same command again.

This is different from an action you can toggle, such as the Edit → Bold menu item and the corresponding speed button. Here is the code of the Bold action (which has the `AutoCheck` property set to `True`, so that it doesn't need to change the status of the `Checked` property in code):

```
procedure TForm1.ActionBoldExecute(Sender: TObject);  
begin  
    with Mem01.Font do  
        if fsBold in Style then  
            Style := Style - [fsBold]  
        else  
            Style := Style + [fsBold];  
end;
```

The `ActionCount` object has very simple code, but it demonstrates an `OnUpdate` handler; when the memo control is empty, it is automatically disabled. You could obtain the same effect by handling the `OnChange` event of the memo control itself, but in general it might not always be possible or easy to determine the status of a control simply by handling one of its events. Here is the code for the two handlers of this action:

```
procedure TForm1.ActionCountExecute(Sender: TObject);  
begin  
    ShowMessage ('Characters: ' + IntToStr (Length (Mem01.Text)));  
end;  
  
procedure TForm1.ActionCountUpdate(Sender: TObject);  
begin  
    ActionCount.Enabled := Mem01.Text <> '';  
end;
```

Finally, I've added a special action to test the action event handler's sender object and get some other system information. Besides showing the object class and name, I've added code that accesses the action list object. I've done this mainly to show that you can access this information and how to do it:

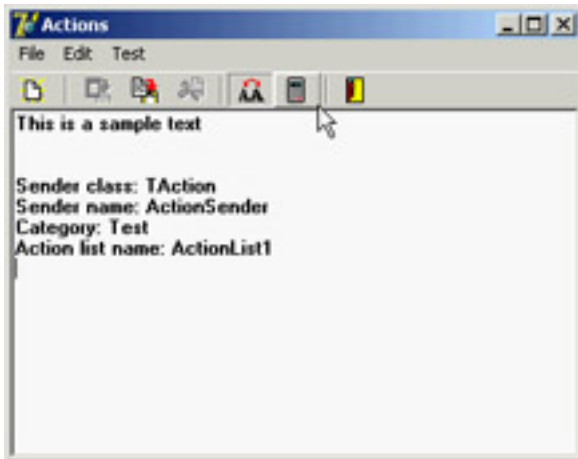
```
procedure TForm1.ActionSenderExecute(Sender: TObject);  
begin  
    Mem01.Lines.Add ('Sender class: ' + Sender.ClassName);  
    Mem01.Lines.Add ('Sender name: ' + (Sender as TComponent).Name);  
end;
```

```

Memol.Lines.Add ('Category: ' + (Sender as TAction).Category);
Memol.Lines.Add (
    'Action list name: ' + (Sender as TAction).ActionList.Name);
end;

```

You can see the output of this code in [Figure 6.11](#), along with the user interface of the example. Notice that the `Sender` is not the menu item you've selected, even if the event handler is connected to it. The `Sender` object, which fires the event, is the action, which intercepts the user operation.



**Figure 6.11:** The Actions example, with a detailed description of the `Sender` of an `Action` object's `OnExecute` event

Finally, keep in mind that you can also write handlers for the events of the `ActionList` object itself, which play the role of global handlers for all the actions in the list, and for the `Application` global object, which fires for all the actions of the application. Before calling the action's `OnExecute` event, Delphi activates the `ActionList`'s `OnExecute` event and the `Application` global object's `OnActionExecute` event. These events can look at the action, eventually execute some shared code, and then stop the execution (using the `Handled` parameter) or let it reach the next level.

If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

**Note** When an action is executed, it searches for a control to play the role of the action target, by looking at the active control, the active form, and other controls on the form. For example, edit actions refer to the currently active control (if they inherit from `TCustomEdit`), and dataset controls look for the dataset connected with the data source of the data-aware control having the input focus. Other actions follow different approaches to find a target component, but the overall idea is shared by most standard actions.

## The Toolbar and `ActionList` of an Editor

In [Chapter 5](#), I built the `RichBar` example to demonstrate the development of an editor with a toolbar and a status bar. Of course, I should have also added a menu bar to the form, but doing so would have created quite a few troubles in synchronizing the status of the toolbar buttons with those of the menu items. A very good solution to this problem is to use actions, as I've done in the `MdEdit1` example discussed in this section.

The application is based on an `ActionList` component, which includes actions for file handling and Clipboard support, with code similar to the `RichBar` version. The font type and color selections are still based on combo boxes, so they don't involve actions—the same is true for the drop-down menu of the `Size` button. The menu, however, has a few extra commands, including one for character counting and one for changing the background color. These commands are based on actions, and the same is true for the three new paragraph justification buttons (and menu commands).

One of the key differences in this new version is that the code never refers to the status of the toolbar buttons, but eventually modifies the status of the actions. In other cases I've used the actions' `OnUpdate` events. For example, the



RichEditSelectionChange method doesn't update the status of the Bold button, which is connected to an action with the following OnUpdate handler:

```
procedure TFormRichNote.acBoldUpdate(Sender: TObject);  
begin  
    acBold.Checked := fsBold in RichEdit.SelAttributes.Style;  
end;
```

Similar OnUpdate event handlers are available for most actions, including the counting operations (available only if there is some text in the RichEdit control), the Save operation (available if the text has been modified), and the Cut and Copy operations (available only if some text is selected):

```
procedure TFormRichNote.acCountcharsUpdate(Sender: TObject);  
begin  
    acCountChars.Enabled := RichEdit.GetTextLen > 0;  
end;
```

```
procedure TFormRichNote.acSaveUpdate(Sender: TObject);  
begin  
    acSave.Enabled := Modified;  
end;
```

```
procedure TFormRichNote.acCutUpdate(Sender: TObject);  
begin  
    acCut.Enabled := RichEdit.SelLength > 0;  
    acCopy.Enabled := acCut.Enabled;  
end;
```

In the older example, the status of the Paste button was updated in the OnIdle event of the Application object. Now that you're using actions you can convert it into yet another OnUpdate handler (see [Chapter 5](#) for details on this code):

```
procedure TFormRichNote.acPasteUpdate(Sender: TObject);  
begin  
    acPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;  
end;
```

The three paragraph-alignment buttons and the related menu items work like radio buttons: they're mutually exclusive, and one of the three options is always selected. For this reason, the actions have the GroupIndex set to 1, the corresponding menu items have the RadioItem property set to True, and the three toolbar buttons have their Grouped property set to True and the AllowAllUp property set to False. (They are also visually enclosed between two separators.)

This arrangement is required so that the program can set the Checked property for the action corresponding to the current style, which avoids unchecking the other two actions directly. This code is part of the OnUpdate event of the action list, because it applies to multiple actions:

```
procedure TFormRichNote.ActionListUpdate(Action: TBasicAction;  
    var Handled: Boolean);  
begin  
    // check the proper paragraph alignment  
    case RichEdit.Paragraph.Alignment of  
        taLeftJustify: acLeftAligned.Checked := True;  
        taRightJustify: acRightAligned.Checked := True;  
        taCenter: acCentered.Checked := True;  
    end;  
    // checks the caps lock status  
    CheckCapslock;  
end;
```



Finally, when one of these buttons is selected, the shared event handler uses the value of the `Tag`, set to the corresponding value of the `TAlignment` enumeration, to determine the proper alignment:

```
procedure TFormRichNote.ChangeAlignment(Sender: TObject);  
begin  
    RichEdit.Paragraph.Alignment := TAlignment ((Sender as TAction).Tag);  
end;
```

[Team LiB](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

## Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement it differently. Delphi has two ready-to-use toolbar containers:

- The CoolBar component is a Win32 common control introduced by Internet Explorer and used by some Microsoft applications.
- The ControlBar component is totally VCL based, with no dependencies on external libraries.

Both components can host toolbar controls as well as some extra elements such as combo boxes and other controls. A toolbar can also replace the menu of an application, as you'll see later. Because the CoolBar component is not frequently used in Delphi applications, it is covered in the sidebar "[A Really Cool Toolbar](#)"; the emphasis in the following sections is on Delphi's ControlBar.

---

### A Really Cool Toolbar

The Win32 CoolBar common control is basically a collection of `TCoolBand` objects that you can activate by using the editor of the Bands property, available also in the component editor menu items or through the Object TreeView. You can customize the CoolBar component in many ways. You can set a bitmap for its background, add some bands to the Bands collection, and then assign an existing component or component container to each band. You can use any window-based control (not graphic controls), but only some of them will show up properly. If you want to have a bitmap on the background of the CoolBar, for example, you need to use partially transparent controls. The typical component used in a CoolBar is the Toolbar, but combo boxes, edit boxes, and animation controls are also quite common.



You can place one band on each line or all of them on the same line. Each would use a part of the available surface, and it would be automatically enlarged when the user clicks on its title. It is easier to use this component than to explain it. Try it yourself or open the CoolDemo example:

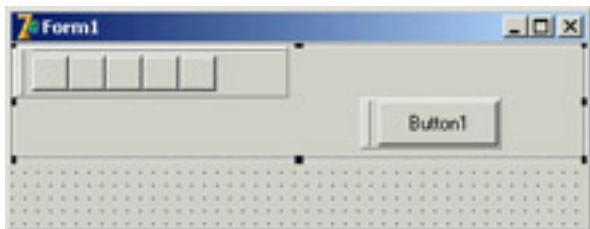
The CoolDemo example's form has a `TCoolBar` component with four bands, two for each of the two lines. The first band includes a subset of the toolbar of the previous example, this time adding an `ImageList` for the highlighted images. The second has an edit box used to set the font of the text; the third has a `ColorGrid` component, used to choose the font color and background color. The last band has a `ComboBox` control with the available fonts.

The user interface of the CoolBar component is nice, and Microsoft uses it in its applications, but alternatives such as the ControlBar component offer a similar UI with no troubles attached. The Windows CoolBar control has had many different and incompatible versions, because Microsoft has released different versions of the common control library with different versions of the Internet Explorer. Some of these versions "broke" existing programs built with Delphi—a very good reason for not using it now even if it is more stable.

---

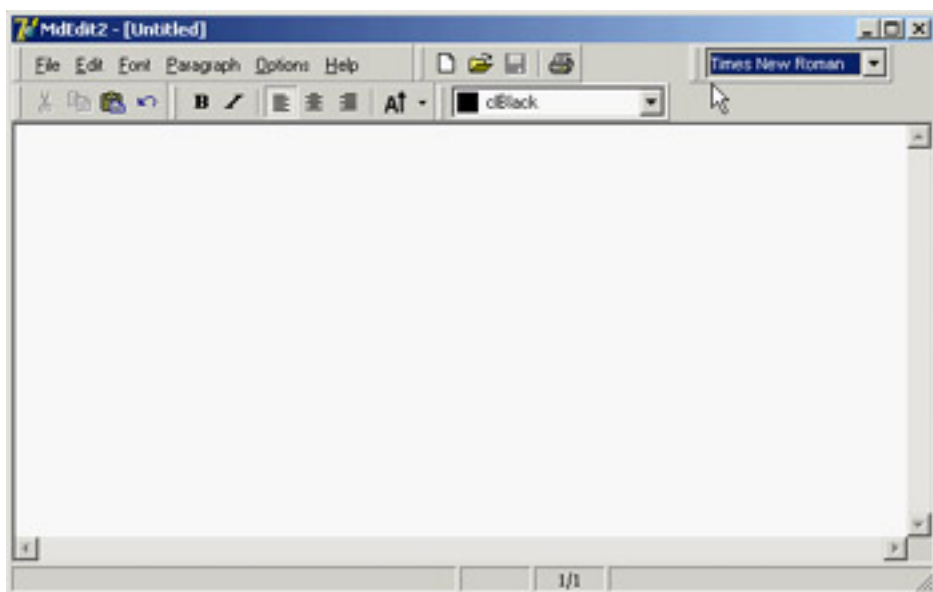
## The ControlBar

The ControlBar is a control container, and you build it by placing other controls inside it, as you do with a panel (there is no list of Bands in it). Every control placed in the bar gets its own dragging area or *grabber* (a small panel with two vertical lines, on the left of the control), even a stand-alone button:



For this reason, you should generally avoid placing specific buttons inside the ControlBar, but instead add containers with buttons inside them. Rather than using a panel, you should use one ToolBar control for every section of the ControlBar.

The MdEdit2 example is another version of the demo I developed to discuss the ActionList component earlier in this chapter. I've grouped the buttons into three toolbars (instead of a single one) and left the two combo boxes as stand-alone controls. All these components are inside a ControlBar so a user can arrange them at runtime, as you can see in [Figure 6.12](#).



**Figure 6.12:** The MdEdit2 example at run time, while a user is rearranging the toolbars in the control bar

The following snippet of the DFM listing of the MdEdit2 example shows how the various toolbars and controls are embedded in the ControlBar component:

```
object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  PopupMenu = BarMenu
object ToolBarFile: TToolBar
  Flat = True
  Images = Images
  Wrapable = False
```

```

object ToolButton1: TToolButton
    Action = acNew
end
    // more buttons...
end
object ToolBarEdit: TToolBar...
object ToolBarFont: TToolBar...
object ToolBarMenu: TToolBar
    AutoSize = True
    Flat = True
    Menu = MainMenu
end
object ComboFont: TComboBox
    Hint = 'Font Family'
    Style = csDropDownList
    OnClick = ComboFontClick
end
object ColorBox1: TColorBox...
end

```

To obtain the standard effect, you have to disable the edges of the toolbar controls and set their style to flat. Sizing all the controls alike, so that you obtain one or two rows of elements of the same height, is not as easy as it might seem at first. Some controls have automatic sizing or various constraints. In particular, to make the combo box the same height as the toolbars, you have to tweak the type and size of its font. Resizing the control itself has no effect.

The ControlBar also has a shortcut menu that allows you to show or hide each of the controls currently inside it. Instead of writing code specific to this example, I've implemented a more generic (and reusable) solution. The shortcut menu, called BarMenu, is empty at design time and is populated when the program starts:

```

procedure TFormRichNote.FormCreate(Sender: TObject);
var
    I: Integer;
    mItem: TMenuItem;
begin
    ...
    // populate the control bar menu
    for I := 0 to ControlBar.ControlCount - 1 do
        begin
            mItem := TMenuItem.Create (Self);
            mItem.Caption := ControlBar.Controls [I].Name;
            mItem.Tag := Integer (ControlBar.Controls [I]);
            mItem.OnClick := BarMenuClick;
            BarMenu.Items.Add (mItem);
        end;
    end;

```

The BarMenuClick procedure is a single event handler used by all the menu items; it uses the Tag property of the Sender menu item to refer to the element of the ControlBar associated with the item in the FormCreate method:

```

procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
    aCtrl: TControl;
begin
    aCtrl := TControl ((Sender as TComponent).Tag);
    aCtrl.Visible := not aCtrl.Visible;
end;

```

Finally, the OnPopup event of the menu is used to refresh the check mark of the menu items:

```

procedure TFormRichNote.BarMenuPopup(Sender: TObject);

```

```

var
  I: Integer;
begin
  // update the menu check marks
  for I := 0 to BarMenu.Items.Count - 1 do
    BarMenu.Items [I].Checked := TControl (BarMenu.Items [I].Tag).Visible;
end;

```

## A Menu in a Control Bar

If you look at the user interface of the MdEdit2 application in [Figure 6.12](#), you'll notice that the form's menu appears inside a toolbar, hosted by the control bar, and below the application caption. All you have to do to accomplish this is set the toolbar's `Menu` property. You must also remove the main menu from the form's `Menu` property (keeping the `MainMenu` component on the form), to avoid having two copies of the menu on screen.

## Delphi's Docking Support

Another feature available in Delphi is support for *dockable* toolbars and controls. In other words, you can create a toolbar and move it to any side of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

Delphi's docking support is connected with container controls, not only with forms. A panel, a `ControlBar`, and other containers (technically, any control derived from `TWinControl`) can be set up as dock targets by enabling their `DockSite` property. You can also set the `AutoSize` property of these containers, so they'll show up only if they hold a control.

To be able to drag a control (an object of any `TControl`-derived class) into the dock site, simply set its `DragKind` property to `dkDock` and its `DragMode` property to `dmAutomatic`. This way, the control can be dragged away from its current position into a new docking container. To undock a component and move it to a special form, you can set its `FloatingDockSiteClass` property to `TCustomDockForm` (to use a predefined stand-alone form with a small caption).

All the docking and undocking operations can be tracked by using special events of the component being dragged (`OnStartDock` and `OnEndDock`) and the component that will receive the docked control (`OnDockOver` and `OnDockDrop`). These docking events are very similar to the dragging events available in earlier versions of Delphi.

There are also commands you can use to accomplish docking operations in code and to explore the status of a docking container. Every control can be moved to a different location using the `Dock`, `ManualDock`, and `ManualFloat` methods. A container has a `DockClientCount` property, indicating the number of docked controls, and a `DockClients` property, which is an array of these controls.

Moreover, if the dock container has the `UseDockManager` property set to `True`, you'll be able to use the `DockManager` property, which implements the `IDockManager` interface. This interface has many features you can use to customize the behavior of a dock container, including support for streaming its status.

As you can see from this brief description, docking support in Delphi is based on a large number of properties, events, and methods—more features than I have room to explore in detail. The next example introduces the main features you'll need.

**Note** Docking support is not currently available in VisualCLX on either platform.

## Docking Toolbars in ControlBars

The MdEdit2 example, already discussed, includes docking support. The program has a second `ControlBar` at the bottom of the form, which accepts dragging one of the toolbars in the `ControlBar` at the top. Because both toolbar

containers have the `AutoSize` property set to `True`, they are automatically removed when the host contains no controls. I've also set the `AutoDrag` and `AutoDock` properties of both `ControlBars` to `True`.

I had to place the bottom `ControlBar` inside a panel, together with the `RichEdit` control. Without this trick, the `ControlBar`, when activated and automatically resized, kept moving below the status bar, which isn't the correct behavior. In the example, the `ControlBar` is the only panel control aligned to the bottom, so there is no possible confusion.

To let users drag the toolbars out of the original container, you once again (as stated previously) set their `DragKind` property to `dkDock` and their `DragMode` property to `dmAutomatic`. The only two exceptions are the menu toolbar, which I decided to keep close to the typical position of a menu bar, and the `ColorBox` control, because unlike the combo box this component doesn't expose the `DragMode` and `DragKind` properties. (In the example's `FormCreate` method, you'll find code you can use to activate docking for the component, based on the "protected hack" discussed in [Chapter 2](#).) The `Fonts` combo box can be dragged, but I don't want to let a user dock it in the lower control bar. To implement this constraint, I've used the control bar's `OnDockOver` event handler, by accepting the docking operation only for toolbars:

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept := Source.Control is TToolbar;
end;
```

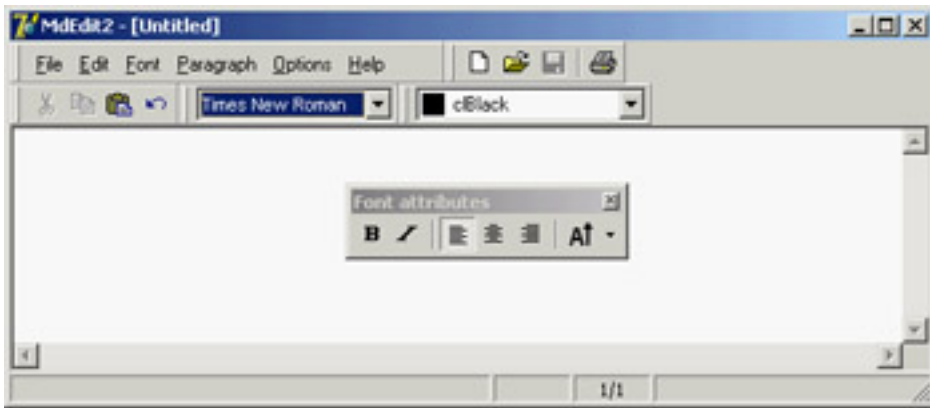
**Warning** Dragging a toolbar directly from the upper control bar to the lower control bar doesn't work. The control bar doesn't resize to host the toolbar during the dragging operation, as it does when you drag the toolbar to a floating form and then to the lower control bar. This is a bug in the VCL, and it is very difficult to circumvent. As you'll see the next example, `MdEdit3` works as expected even if it has the same code: It uses a different component with different VCL support code!

When you move one of the toolbars outside of any container, Delphi automatically creates a floating form; you might be tempted to set it back by closing the floating form. This doesn't work, because the floating form is removed along with the toolbar it contains. However, you can use the shortcut menu of the topmost `ControlBar`, also attached to the other `ControlBar`, to show this hidden toolbar.

The floating form created by Delphi to host undocked controls has a thin caption, the so-called *toolbar caption*, which by default has no text. For this reason, I've added some code to the `OnEndDock` event of each dockable control to set the caption of the newly created form into which the control is docked. To avoid a custom data structure for this information, I've used the text of the `Hint` property for these controls (which is basically not used) to provide a suitable caption:

```
procedure TFormRichNote.EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  if Target is TCustomForm then
    TCustomForm(Target).Caption := GetShortHint((Sender as TControl).Hint);
end;
```

You can see an example of this effect in the `MdEdit2` program in [Figure 6.13](#).

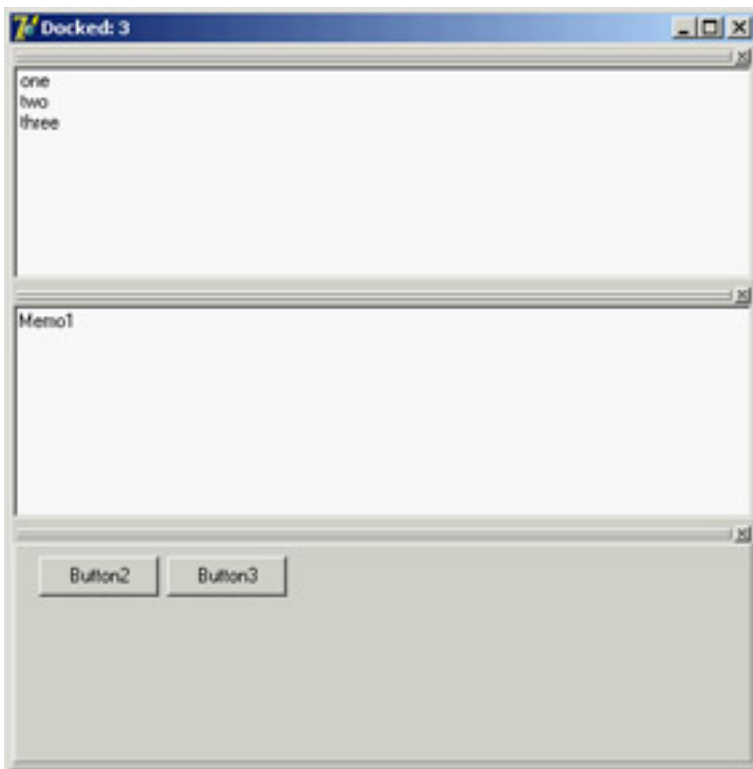


**Figure 6.13:** The MdEdit2 example allows you to dock the toolbars (but not the menu) at the top or bottom of the form or to leave them floating.

Another extension of the example (which I haven't done) might add dock areas on the two sides of the form. The only extra effort this would require would be a routine to turn the toolbars vertically instead of horizontally. Doing so requires switching the `Left` and `Top` properties of each button after disabling the automatic sizing.

## Controlling Docking Operations

Delphi provides many events and methods that give you a lot of control over docking operations, including a dock manager. To explore some of these features, try the DockTest example, a test bed for docking operations shown in [Figure 6.14](#).



**Figure 6.14:** The DockTest example with three controls docked in the main form

The program handles the `OnDockOver` and `OnDockDrop` events of a dock host panel to display messages to the user, such as the number of controls currently docked:

```
procedure TForm1.PanellDockDrop(Sender: TObject; Source: TDragDockObject;
  X, Y: Integer);
begin
  Caption := 'Docked: ' + IntToStr (Panell.DockClientCount);
```

```
end;
```

In the same way, the program handles the main form's docking events. The controls have a shortcut menu you can invoke to perform docking and undocking operations in code, without the usual mouse dragging, with code like this:

```
procedure TForm1.menuFloatPanelClick(Sender: TObject);  
begin  
    Panel2.ManualFloat (Rect (100, 100, 200, 300));  
end;
```

```
procedure TForm1.Floating1Click(Sender: TObject);  
var  
    aCtrl: TControl;  
begin  
    aCtrl := Sender as TControl;  
    // toggle the floating status  
    if aCtrl.Floating then  
        aCtrl.ManualDock (Panel1, nil, alBottom)  
    else  
        aCtrl.ManualFloat (Rect (100, 100, 200, 300));  
end;
```

To make the program perform properly at startup, you must dock the controls to the main panel in the initial code; otherwise you can get a weird effect. Oddly enough, for the program to behave properly, you need to add controls to the dock manager and also dock them to the panel (one operation doesn't trigger the other automatically):

```
// dock memo  
Memo1.Dock(Panell, Rect (0, 0, 100, 100));  
Panell.DockManager.InsertControl(Memo1, alTop, Panell);  
// dock listbox  
ListBox1.Dock(Panell, Rect (0, 100, 100, 100));  
Panell.DockManager.InsertControl(ListBox1, alLeft, Panell);  
// dock panel2  
Panel2.Dock(Panell, Rect (100, 0, 100, 100));  
Panell.DockManager.InsertControl(Panel2, alBottom, Panell);
```

The example's final feature is probably the most interesting and the most difficult to implement properly. Every time the program closes, it saves the current docking status of the panel, using the dock manager support. When the program is reopened, it reapplies the docking information, restoring the window's previous configuration. Here is the code you might think of writing for saving and loading:

```
procedure TForm1.FormDestroy(Sender: TObject);  
var  
    FileStr: TFileStream;  
begin  
    if Panell.DockClientCount > 0 then  
        begin  
            FileStr := TFileStream.Create (DockFileName, fmCreate or fmOpenWrite);  
            try  
                Panell.DockManager.SaveToStream (FileStr);  
            finally  
                FileStr.Free;  
            end;  
        end  
    else  
        // remove the file  
        DeleteFile (DockFileName);  
end;
```



```

procedure TForm1.FormCreate(Sender: TObject);
var
    FileStr: TFileStream;
begin
    // initialization code above...

    // reload the settings
    DockFileName := ExtractFilePath (Application.Exename) + 'dock.dck';
    if FileExists (DockFileName) then
    begin
        FileStr := TFileStream.Create (DockFileName, fmOpenRead);
        try
            Panell1.DockManager.LoadFromStream (FileStr);
        finally
            FileStr.Free;
        end;
    end;
    Panell1.DockManager.ResetBounds (True);
end;

```

This code works fine as long as all controls are initially docked. When you save the program, if one control is floating, you won't see it when you reload the settings. However, because of the initialization code inserted earlier, the control will be docked to the panel anyway, and will appear when you drag away the other controls. Needless to say, this is a messy situation. For this reason, after loading the settings, I added this further code:

```

for i := Panell1.DockClientCount - 1 downto 0 do
begin
    aCtrl := Panell1.DockClients[i];
    Panell1.DockManager.GetControlBounds(aCtrl, aRect);
    if (aRect.Bottom - aRect.Top <= 0) then
    begin
        aCtrl.ManualFloat (aCtrl.ClientRect);
        Panell1.DockManager.RemoveControl(aCtrl);
    end;
end;

```

The complete listing includes more commented code, which I used while developing this program; you might use it to understand what happens (which is often different from what you'd expect!). Briefly, the controls that have no size set in the dock manager (the only way I could figure out they are not docked) are shown in a floating window and are removed from the dock manager list.

If you look at the complete code for the `OnCreate` event handler, you'll see a lot of complex code, just to get a plain behavior. You could add more features to a docking program, but to do so you should remove other features, because some of them might conflict. Adding a custom docking form breaks features of the dock manager. Automatic alignments don't work well with the docking manager's code for restoring the status. I suggest you take this program and explore its behavior, extending it to support the type of user interface you prefer.

**Note** Remember that although docking panels make an application look nice, some users are confused by the fact that their toolbars might disappear or be in a different position than they are used to. Don't overuse the docking features, or some of your inexperienced users may get lost.

## Docking to a PageControl

Another interesting feature of page controls is their specific support for docking. As you dock a new control over a PageControl, a new page is automatically added to host it, as you can easily see in the Delphi environment. To accomplish this, you set the PageControl as a dock host and activate docking for the client controls. This technique works best when you have secondary forms you want to host. Moreover, if you want to be able to move the entire PageControl into a floating window and then dock it back, you'll need a docking panel in the main form.

This is what I've done in the DockPage example, which has a main form with the following settings:

```
object Form1: TForm1
  Caption = 'Docking Pages'
object Panel1: TPanel
  Align = alLeft
  DockSite = True
  OnMouseDown = Panel1MouseDown
object PageControl1: TPageControl
  ActivePage = TabSheet1
  Align = alClient
  DockSite = True
  DragKind = dkDock
object TabSheet1: TTabSheet
  Caption = 'List'
object ListBox1: TListBox
  Align = alClient
end
end
end
object Splitter1: TSplitter
  Cursor = crHSplit
end
object Mem1: TMemo
  Align = alClient
end
end
```

Notice that the Panel has the `UseDockManager` property set to True and that the PageControl invariably hosts a page with a list box, because when you remove all the pages, the code used for automatic sizing of dock containers might cause you trouble.

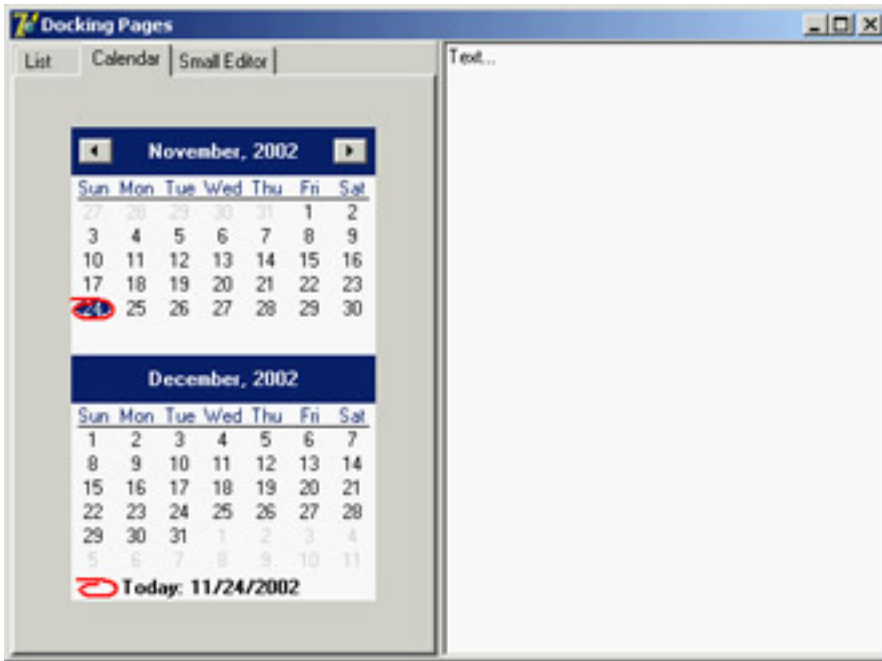
The program has two other forms with similar settings (although they host different controls):

```
object Form2: TForm2
  Caption = 'Small Editor'
  DragKind = dkDock
  DragMode = dmAutomatic
object Mem1: TMemo
  Align = alClient
end
end
```

You can drag these forms onto the page control to add new pages to it, with captions corresponding with the form titles. You can also undock each of these controls and even the entire PageControl. The program doesn't enable automatic dragging, which would make it impossible to switch pages; instead, the feature is activated when the user clicks on the area of the PageControl that has no tabs—that is, on the underlying panel:

```
procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  PageControl1.BeginDrag (False, 10);
end;
```

You can test this behavior by running the DockPage example, and [Figure 6.15](#) tries to depict it. Notice that when you remove the PageControl from the main form, you cannot directly dock the other forms to the panel, as this is prevented by specific code within the program (simply because at times the behavior won't be correct).



**Figure 6.15:** The main form of the DockPage example after a form has been docked to the page control on the left.

## The ActionManager Architecture

You have seen that actions and the ActionManager component can play a central role in the development of Delphi applications, because they allow a much better separation of the user interface from the actual code of the application. The user interface can now easily change without impacting the code too much. The drawback of this approach is that a programmer has more work to do. To create a new menu item, you need to add the corresponding action first, then move to the menu, add the menu item, and connect it to the action.

To solve this issue, and to provide developers and end users with some advanced features, Delphi 6 introduced a new architecture based on the ActionManager component, which largely extends the role of actions. The ActionManager has a collection of actions as well as a collection of toolbars and menus tied to them. The development of these toolbars and menus is completely visual: You drag actions from a special component editor of the ActionManager to the toolbars to access the buttons you need. Moreover, you can let the end user of your programs do the same operation, rearranging their toolbars and menus beginning with the actions you provide them.

In other words, using this architecture allows you to build applications with a modern user interface, customizable by the user. The menu can show only the recently used items (as many Microsoft programs do), allows for animation, and more.

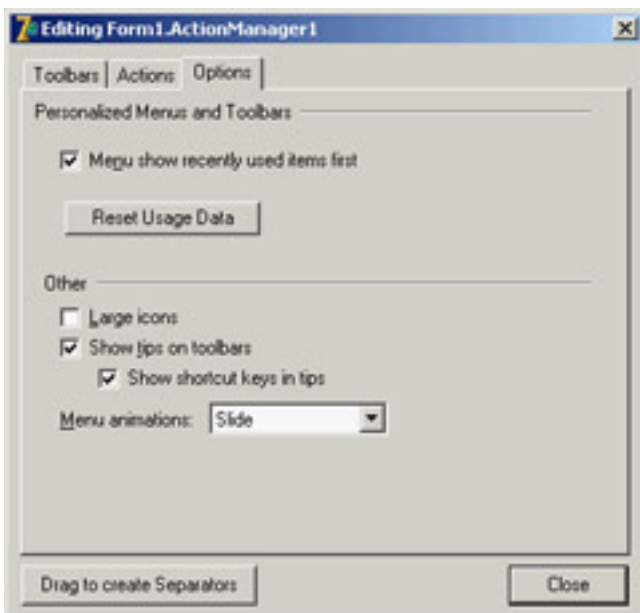
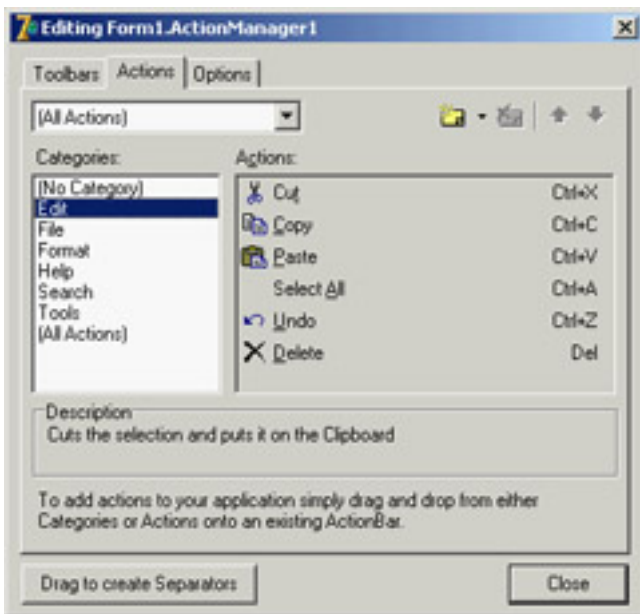
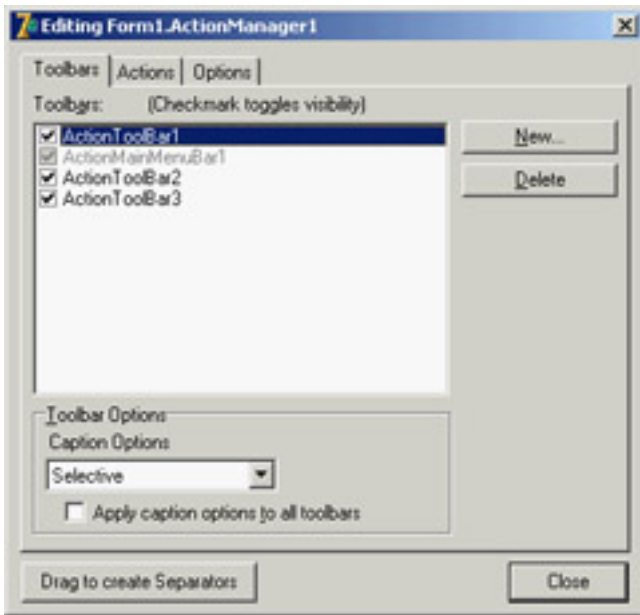
This architecture is centered on the ActionManager component, but it also includes a few other components found at the end of the Additional page of the palette:

- The ActionManager component is a replacement for the ActionList (but can also use one or more existing ActionLists).
- The ActionMainMenuBar control is a toolbar used to display the menu of an application based on the actions of an ActionManager component.
- The ActionToolBar control is a toolbar used to host buttons based on the actions of an ActionManager component.
- The CustomizeDlg component includes the dialog box you can use to let users customize the user interface of an application based on the ActionManager component.
- The PopupActionBarEx component is an extra component you should use to let your pop-up menus follow the same user interface as your main menus. This component doesn't ship with Delphi 7 but is available as a separate download.

**Tip** You can find the PopupActionBarEx (also called ActionPopupMenu) component on Borland's CodeCentral web repository (number 18870). In addition, you'll find more information at the component author's website ([homepages.borland.com/strefethen](http://homepages.borland.com/strefethen)); he is a member of Delphi's R&D Team at Borland; the component is on the site, but is not officially supported.

## Building a Simple Demo

Because this architecture is mostly visual, a demo is worth more than a general discussion (although a printed book is not the best way to discuss a highly visual series of operations). To create a sample program based on this architecture, drop an ActionManager component on a form and double-click it to open its component editor, shown in [Figure 6.16](#). Notice that this editor is not modal, so you can keep it open while doing other operations in Delphi. This same dialog box is also displayed by the CustomizeDlg component, although with some limited features (for example, adding new actions is disabled).

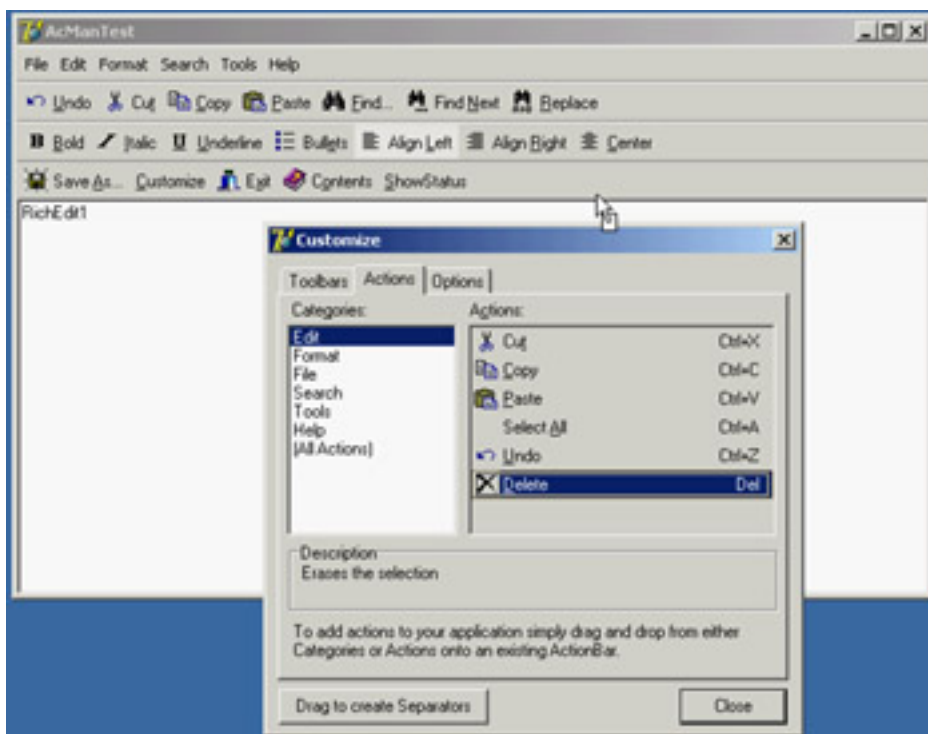


**Figure 6.16:** The three pages of the ActionManager editor dialog box

The editor's three pages are as follows:

- The first page provides a list of visual containers of actions (toolbars or menus). You add new toolbars by clicking the New button. To add new menus, you have to add the corresponding component to the form, then open the `ActionBars` collection of the `ActionManager`, select an action bar or add a new one, and hook the menu to it using the `ActionBar` property. These are the same steps you could follow to connect a new toolbar to this architecture at run time.
- The second page of the `ActionManager` editor is very similar to the `ActionList` editor, providing a way to add new standard or custom actions, arrange them in categories, and change their order. A nice feature of this page, though, is that you can drag a category or a single action from it and drop it onto an action bar control. If you drag a category to a menu, you obtain a pull-down menu with all the category items; if you drag it to a toolbar, each of the category's actions gets a button on the toolbar. If you drag a single action to a toolbar, you get the corresponding button; if you drag it to the menu, you get a direct menu command, which is something you should generally avoid.
- The last page of the `ActionManager` editor allows you (and optionally an end user) to activate the display of recently used menu items and to modify some of the toolbars' visual properties.

The `AcManTest` program is an example that uses some of the standard actions and a `RichEdit` control to showcase the use of this architecture (I haven't written any custom code to make the actions work better, because I wanted to focus only on the action manager for this example). You can experiment with it at design time or run it, click the `Customize` button, and see what an end user can do to customize the application (see [Figure 6.17](#)).



**Figure 6.17:** Using the `CustomizeDlg` component, you can let a user customize the toolbars and the menu of an application by dragging items from the dialog box or moving them around in the action bars.

In the program, you can prevent the user from doing some operations on actions. Any specific element of the user interface (a `TActionClient` object) has a `ChangedAllowed` property that you can use to disable modify, move, and delete operations. Any action client container (the visual bars) has a property to disable hiding itself (`AllowHiding` by default is set to `True`). Each `ActionBar` `Items` collection has a `Customizable` option you can turn off to disable all user changes to the entire bar.

**Tip** When I say ActionBar I don't mean the visual toolbars containing action items, but the items of the *ActionBars* collection of the ActionManager component, which in turn has an *Items* collection. The best way to understand this structure is to look at the subtree displayed by the Object TreeView for an ActionManager component. Each *TActionBar* collection item has a *TCustomActionBar* visual component connected, but not the reverse (so, for example, you cannot reach this *Customizable* property if you start by selecting the visual toolbar). Due to the similarity of the two names, it can take a while to understand what the Delphi help is referring to.

To make user settings persistent, I've connected a file (called *settings*) to the *FileName* property of the ActionManager component. When you assign this property, you should enter the name of the file you want to use; when you start the program, the file will be created for you by the ActionManager. The persistency is accomplished by streaming each *ActionClientItem* connected with the action manager. Because these action client items are based on the user settings and maintain state information, a single file collects both user changes to the interface and usage data.

Because Delphi stores user setting and status information in a file you provide, you can make your application support multiple users on a single computer. Simply use a file of settings for each of them (under the MyDocuments or MySettings virtual folder) and connect it to the action manager as the program starts (using the current user of the computer or after some custom login). Another possibility is to store these settings over the network, so that even when a user moves to a different computer, the current personal settings will move along with them.

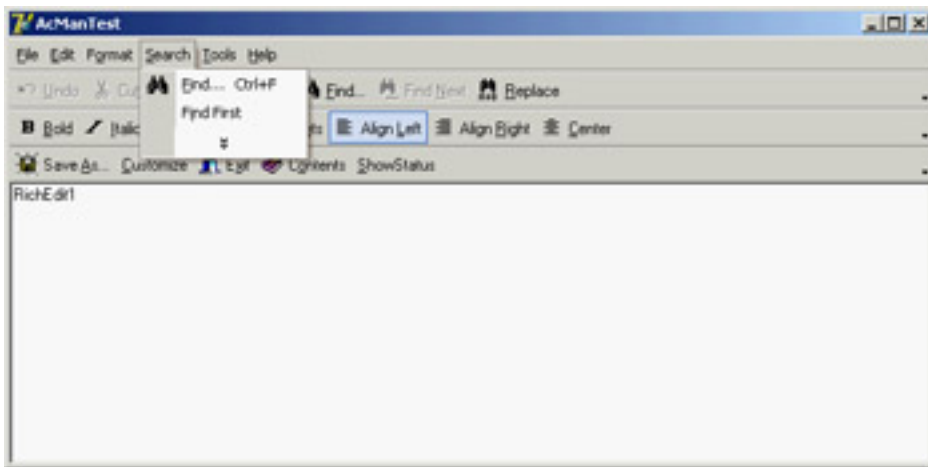
In the program, I've decided to store the settings in a file store in the same folder as the program, assigning the relative path (the filename) to the ActionManager's *FileName* property. The component will fill in the complete filename with the program folder, easily finding the file to load. However, the file includes among its data its own filename, with an absolute path. So, when it is time to save the file, the operation may refer to an older path. This prevents you from copying this program with its settings to a different folder (for example, this is an issue for the AcManTest demo). You can reset the *FileName* property after loading the file. As a further alternative, you could set the filename at runtime, in the form's *OnCreate* event. In this case you also have to force the file to reload, because you are assigning it after the ActionManager component and the ActionBars have already been created and initialized. However, you might want to force the filename after loading it, as just described:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  ActionManager1.FileName :=  
    ExtractFilePath (Application.ExeName) + 'settings';  
  ActionManager1.LoadFromFile(ActionManager1.FileName);  
  // reset the settings file name after loading it (relative path)  
  ActionManager1.FileName :=  
    ExtractFilePath (Application.ExeName) + 'settings';  
end;
```

## Least-Recently Used Menu Items

Once a file for the user settings is available, the ActionManager will save the user preferences into it and also use it to track the user activity. This is essential to let the system remove menu items that haven't been used for some time, making them available in an extended menu using the same user interface adopted by Microsoft (see [Figure 6.18](#) for an example).





**Figure 6.18:** The ActionManager disables least-recently used menu items that you can still see by selecting the menu extension command.

The ActionManager doesn't just show the least-recently used items: It allows you to customize this behavior in a very precise way. Each action bar has a `SessionCount` property that keeps track of the number of times the application has been executed. Each `ActionClientItem` has a `LastSession` property and a `UsageCount` property used to track user operations. Notice, by the way, that a user can reset all this dynamic information by using the Reset Usage Data button in the customization dialog.

The system calculates the number of sessions the action has gone unused by computing the difference between the number of times the application has been executed (`SessionCount`) and the last session in which the action has been used (`LastSession`). The value of `UsageCount` is used to look up in the `PrioritySchedule` how many sessions the items can go unused before it is removed. In other words, the `PrioritySchedule` maps each the usage count with a number of *unused* sessions. By modifying the `PrioritySchedule`, you can determine how quickly the items are removed in case they are not used.

You can also prevent this system from being activated for specific actions or groups of actions. The `Items` property of the ActionManager's `ActionBars` has a `HideUnused` property you can toggle to disable this feature for an entire menu. To make a specific item always visible, regardless of the actual usage, you can also set its `UsageCount` property to `-1`. However, the user settings might override this value.

To help you better understand how this system works, I've added a custom action (`ActionShowStatus`) to the AcManTest example. The action has the following code that saves the current action manager settings to a memory stream, converts the stream to text, and shows it in the memo (refer to [Chapter 4](#) for more information about streaming):

```

procedure TForm1.ActionShowStatusExecute(Sender: TObject);
var
    memStr, memStr2: TMemoryStream;
begin
    memStr := TMemoryStream.Create;
    try
        memStr2 := TMemoryStream.Create;
        try
            ActionManager1.SaveToStream(memStr);
            memStr.Position := 0;
            ObjectBinaryToText(memStr, memStr2);
            memStr2.Position := 0;
            RichEdit1.Lines.LoadFromStream(memStr2);
        finally
            memStr2.Free;
        end;
    finally
        memStr.Free;
    end;

```



```
end;  
end;
```

The output you obtain is the textual version of the `settings` file automatically updated at each execution of the program. Here is a small portion of this file, including the details of one of the pull-down menus and plenty of comments:

```
item // File pulldown of the main menu action bar  
  Items = <  
    item  
      Action = Form1.FileOpen1  
      LastSession = 19 // was used in the last session  
      UsageCount = 4 // was used four times  
    end  
    item  
      Action = Form1.FileSaveAs1 // never used  
    end  
    item  
      Action = Form1.FilePrintSetup1  
      LastSession = 7 // used some time ago  
      UsageCount = 1 // only once  
    end  
    item  
      Action = Form1.FileRun1 // never used  
    end  
    item  
      Action = Form1.FileExit1 // never used  
    end>  
  Caption = '&File'  
  LastSession = 19  
  UsageCount = 5 // the sum of the usage count of the items  
end
```

## Porting an Existing Program

If this architecture is useful, you'll probably need to redo most of your applications to take advantage of it. However, if you're already using actions (with the `ActionList` component), this conversion will be much simpler. The `ActionManager` has its own set of actions but can also use actions from another `ActionManager` or `ActionList`. The `ActionManager`'s `LinkedActionLists` property is a collection of other containers of actions (`ActionLists` or `ActionManagers`), which can be associated with the current `ActionManager`. Associating all the various groups of actions is useful because you can let a user customize the entire user interface with a single dialog box.

If you hook external actions and open the `ActionManager` editor, you'll see in the `Actions` page a combo box listing the current `ActionManager` plus the other action containers linked to it. You can choose one of these containers to see its set of actions and change their properties. The `All Action` option in this combo box allows you to work on all the actions from the various containers at once; however, I've noticed that at startup it is selected but not always *effective*. Reselect it to see all the actions.

As an example of porting an existing application, I've extended the program built throughout this chapter into the `MdEdit3` example. This example uses the same action list as the previous version, hooked to an `ActionManager` that has the extra `customize` property to let users rearrange the user interface. Unlike the earlier `AcManDemo` program, the `MdEdit3` example uses a `ControlBar` as a container for the action bars (a menu, three toolbars, and the usual combo boxes) and has full support for dragging them outside the container as floating bars and dropping them into the lower `ControlBar`.

To accomplish this, I only had to modify the source code slightly to refer to the new classes for the containers (`TCustomActionToolBar` instead of `TToolBar`) in the `ControlBarLowerDockOver` method. I also found that the

ActionToolBar component's OnEndDock event passes as parameter an empty target when the system creates a floating form to host the control, so I couldn't easily give this form a new custom caption (see the form's EndDock method).

## Using List Actions

You'll see more examples of the use of this architecture in the chapters devoted to MDI and database programming ([Chapter 8](#) and [Chapter 13](#), for example). For the moment, I want to add an extra example showing how to use a rather complex group of standard actions: the list actions. List actions comprise two different groups. Some of them (such as Move, Copy, Delete, Clear, and Select All) are normal actions that work on list boxes or other lists. The VirtualListAction and StaticListAction, however, define actions providing a list of items that will be displayed in a toolbar as a combo box.

The ListActions demo highlights both groups of list actions; its ActionManager has five of actions displayed on two separate toolbars. This is a summary of the actions (I've omitted the action bars portion of the component's DFM file):

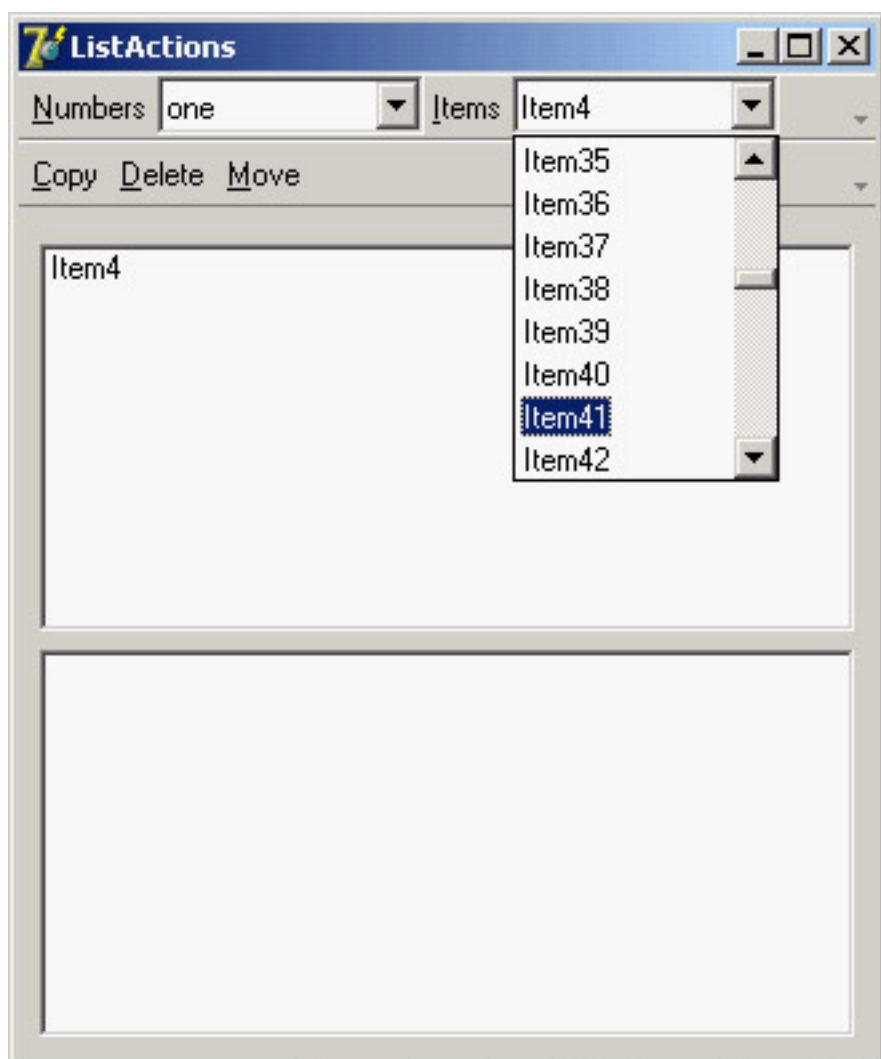
```
object ActionManager1: TActionManager
  ActionBar.SessionCount = 1
  ActionBar = <...>
object StaticListAction1: TStaticListAction
  Caption = 'Numbers'
  Items.CaseSensitive = False
  Items.SortType = stNone
  Items = <
    item
      Caption = 'one'
    end
    item
      Caption = 'two'
    end
  ...>
  OnItemSelected = ListActionItemSelected
end
object VirtualListAction1: TVirtualListAction
  Caption = 'Items'
  OnGetItem = VirtualListAction1GetItem
  OnGetItemCount = VirtualListAction1GetItemCount
  OnItemSelected = ListActionItemSelected
end
object ListControlCopySelection1: TListControlCopySelection
  Caption = 'Copy'
  Destination = ListBox2
  ListControl = ListBox1
end
object ListControlDeleteSelection1: TListControlDeleteSelection
  Caption = 'Delete'
end
object ListControlMoveSelection2: TListControlMoveSelection
  Caption = 'Move'
  Destination = ListBox2
  ListControl = ListBox1
end
end
```

The program has also two list boxes in its form, which are used as action targets. The Copy and Move actions are tied to these two list boxes by their ListControl and Destination properties. The Delete action automatically works

with the list box having the input focus.

The StaticListAction defines a series of alternative items in its `Items` collection. This is not a plain string list, because any item also has an `ImageIndex` that lets you add graphical elements to the control displaying the list. You can, of course, add more items to this list programmatically. However, if the list is highly dynamic, you can also use the `VirtualListAction`. This action doesn't define a list of items but has two events you can use to provide strings and images for the list: `OnGetItemCount` allows you to indicate the number of items to display, and `OnGetItem` is then called for each specific item.

In the ListActions demo, the `VirtualListAction` has the following event handlers for its definition, producing the list you can see in the active combo box in [Figure 6.19](#):



**Figure 6.19:** The ListActions application has a toolbar hosting a static list and a virtual list.

```
procedure TForm1.VirtualListAction1GetItemCount(Sender: TCustomListAction;  
    var Count: Integer);  
begin  
    Count := 100;  
end;  
  
procedure TForm1.VirtualListAction1GetItem(Sender: TCustomListAction;  
    const Index: Integer; var Value: String;  
    var ImageIndex: Integer; var Data: Pointer);  
begin  
    Value := 'Item' + IntToStr (Index);
```

**end;**

**Note** I thought the virtual action items were requested only when needed for display, making this a virtual list. Instead, all the items are created right away. You can prove it by enabling the commented code in the *VirtualListAction1GetItem* method (not included in the previous listing), which adds to each item the time its string is requested.

Both the static list and the virtual list have an *OnItemSelected* event. In the shared event handler, I've written the following code to add the current item to the form's first list box:

```
procedure TForm1.ListActionItemSelected(Sender: TCustomListAction;  
    Control: TControl);  
begin  
    ListBox1.Items.Add ((Control as TCustomActionCombo).SelText);  
end;
```

In this case, the sender is the custom action list, but the *ItemIndex* property of this list is not updated with the selected item. However, by accessing the visual control that displays the list, you can obtain the value of the selected item.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

In this chapter, I've introduced the use of actions, the actions list, and Action Manager architectures. As you've seen, this is an extremely powerful architecture to separate the user interface from your application code, which uses and refers to the actions and not the menu items or toolbar buttons related to them. The recent extension of this architecture allows users of your programs to have a lot of control, and makes your applications resemble high-end programs without much effort on your part. The same architecture is also very handy for designing your program's user interface, regardless of whether you give this ability to users.

I've also covered some user-interface techniques, such as docking toolbars and other controls. You can consider this chapter the first step toward building professional applications. We will take other steps in the following chapters; but you already know enough to make your programs similar to some best-selling Windows applications, which may be very important for your clients.

Now that the elements of your program's main form are properly set up, you can consider adding secondary forms and dialog boxes. This is the topic of [Chapter 7](#), along with a general introduction to forms. [Chapter 8](#) will then discuss the overall structure of a Delphi application.

## Chapter 7: Working with Forms

### Overview

If you've read the previous chapters, you should now be able to use Delphi's visual components to create the user interface for your applications. Now let's turn our attention to another central element of development in Delphi: forms. You have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the `TForm` class are particularly interesting.

This chapter looks at some of the properties and styles of forms and at sizing and positioning them, as well as form scaling and scrolling. I'll also introduce applications with multiple forms, the use of dialog boxes (custom and predefined ones), frames, and visual form inheritance. Finally, I'll devote some time to input on a form, from both the keyboard and the mouse.

## The TForm Class

Forms in Delphi are defined by the `TForm` class, which is included in the Forms unit of VCL. Of course, there is now a second definition of forms in VisualCLX. Although I'll mainly refer to the VCL class in this chapter, I'll also highlight differences with the cross-platform version provided in CLX.

The `TForm` class is part of the windowed-controls hierarchy, which starts with the `TWinControl` (or `TWidgetControl`) class. `TForm` inherits from the *almost complete* `TCustomForm`, which in turn inherits from `TScrollingWinControl` (or `TScrollingWidget`). Having all the features of their many base classes, forms have a long series of methods, properties, and events. For this reason, I won't try to list them here—I'd rather present some interesting techniques related to forms throughout this chapter. I'll begin by presenting a technique for *not* defining the form of a program at design time, using the `TForm` class directly, and then explore a few interesting properties of the form class.

Throughout the chapter, I'll point out a few differences between VCL forms and CLX forms. I've built a CLX version for most of the examples, so you can immediately begin experimenting with forms and dialog boxes in CLX, as well as VCL. As in past chapters, the CLX version of each example is prefixed by the letter Q.

## Using Plain Forms

Delphi developers tend to create forms at design time, which implies deriving a new class from the base class, and build the content of the form visually. This is certainly a reasonable standard practice, but it is not compulsory to create a descendant of the `TForm` class to show a form, particularly if it is a simple one.

Consider this case: Suppose you have to show a rather long message (based on a string) to a user, and you don't want to use the simple predefined message box because it will look too large and won't provide scroll bars. You can create a form with a memo component in it, and display the string inside the memo. Nothing prevents you from creating this form in the standard visual way, but you might consider doing this in code, particularly if you need a large degree of flexibility.

The `DynaForm` and `QDynaForm` examples (available among the book source code), which are somewhat extreme, have no form defined at design time but include a unit with this function:

```
procedure ShowStringForm (str: string);
var
  form: TForm;
begin
  Application.CreateForm (TForm, form);
  form.caption := 'DynaForm';
  form.Position := poScreenCenter;
  with TMemo.Create (form) do
  begin
    Parent := form;
    Align := alClient;
    Scrollbars := ssVertical;
    ReadOnly := True;
    Color := form.Color;
    BorderStyle := bsNone;
    WordWrap := True;
    Text := str;
  end;
  form.Show;
end;
```

I had to create the form by calling the `Application` global object's `CreateForm` method (a feature required by Delphi applications and discussed in [Chapter 8](#), "The Architecture of Delphi Applications"); other than that, this code

does dynamically what you generally do with the Form Designer. Writing this code is undoubtedly more tedious, but it allows also a greater deal of flexibility, because any parameter can depend on external settings.

The previous `ShowStringForm` function is not executed by an event of another form, because there are no traditional forms in this program. Instead, I've modified the project's source code as follows:

```
program DynaForm;

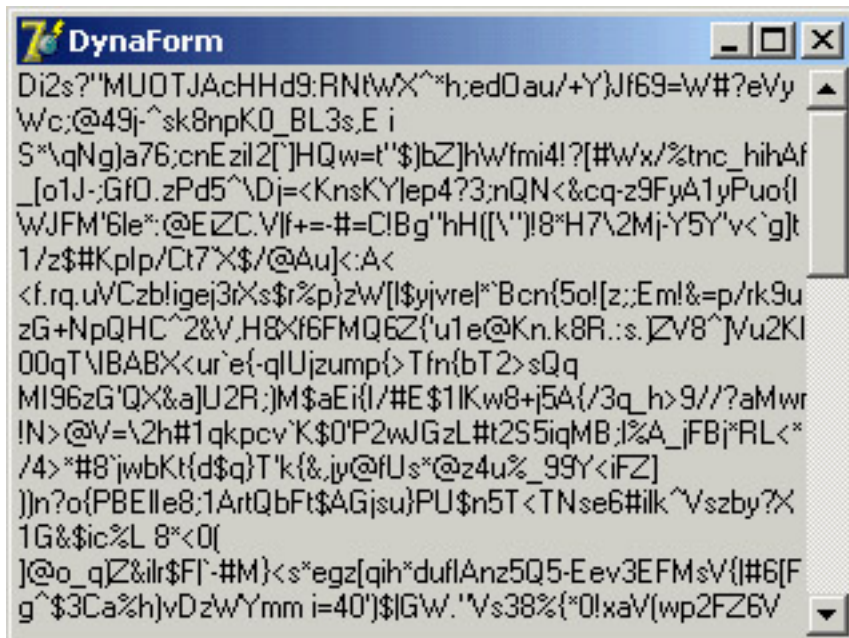
uses
  Forms,
  DynaMemo in 'DynaMemo.pas';

{$R *.RES}

var
  str: string;

begin
  str := '';
  Randomize;
  while Length (str) < 2000 do
    str := str + Char (32 + Random (74));
  ShowStringForm (str);
  Application.Run;
end.
```

The effect of running the DynaForm program is a strange-looking form filled with random characters (as you can see in [Figure 7.1](#))—it isn't terribly useful in itself, but it underscores the idea.



**Figure 7.1:** The dynamic form generated by the DynaForm example is completely created at run time, with no design-time support.



**Tip** An indirect advantage of this approach, compared to the use of DFM files for design-time forms, is that it would be much more difficult for an external programmer to grab information about the structure of the application. In [Chapter 5](#), "Visual Controls," you saw that you can extract the DFM from the current Delphi executable file, but the same can be easily accomplished for any executable file compiled with Delphi for which you don't have the source code. If it is important for you to keep to yourself a specific set of components you are using (maybe those in a specific form), along with the default values of their properties, writing the extra code may be worth the effort.

## The Form Style

The `FormStyle` property allows you to choose between a normal form (`fsNormal`) and the windows that make up a Multiple Document Interface (MDI) application. In this case, you'll use the `fsMDIForm` style for the MDI parent window—that is, the frame window of the MDI application—and the `fsMDIChild` style for the MDI child window. To learn more about the development of an MDI application, look at [Chapter 8](#), "The Architecture of Delphi Applications."

A fourth option is the `fsStayOnTop` style, which determines whether the form must always remain on top of all other windows, except for any that also happen to be "stay-on-top" windows.

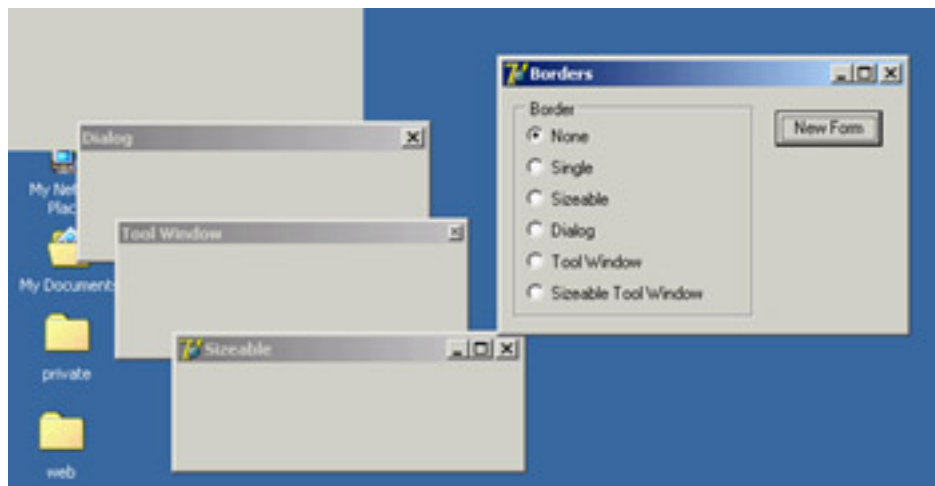
To create a top-most form (a form whose window is always on top), you need only set the `FormStyle` property, as indicated earlier. This property has two different effects, depending on the kind of form you apply it to:

- The main form of an application will remain in front of every other application (unless other applications have the same top-most style). At times, this behavior generates a rather ugly visual effect, so it makes sense only for special-purpose alert programs.
- A secondary form will remain in front of any other form in the application it belongs to. The windows of other applications are not affected. This approach is often used for floating toolbars and other forms that should stay in front of the main window.

**Warning** In the VCL, when this property is applied to a secondary form, the form only remains in front of the other forms in the same application. In CLX, even a secondary form will be kept in front of any other form of the windowing system—something you'd generally rather avoid.

## The Border Style

Another important property of a form is its `BorderStyle`. This property refers to a visual element of the form, but it has a much more profound influence on the *behavior* of the window, as you can see in [Figure 7.2](#).



**Figure 7.2:** Sample forms with the various border styles, created by the Borders example

At design time, the form is always shown using the default value of the `BorderStyle` property, `bsSizeable`. This value

corresponds to a Windows style known as *thick frame*. When a main window has a thick frame around it, a user can resize it by dragging its border. This state is made clear by the special *resize* cursors (with the shape of a double-pointer arrow) displayed when the user moves the mouse onto this thick window border.

A second important choice for this property is `bsDialog`. If you select it, the form uses as its border the typical dialog-box frame—a thick frame that doesn't allow resizing. In addition to this graphical element, note that if you select the `bsDialog` value, the form becomes a dialog box. This involves several changes: For example, the items on its system menu are different, and the form will ignore some of the elements of the `BorderIcons` set property.

**Warning** Setting the `BorderStyle` property at design time produces no visible effect. Several component properties do not take effect at design time, because they would prevent you from working on the component while developing the program. For example, how could you resize the form with the mouse if it were turned into a dialog box? When you run the application, though, the form will have the border you requested.

You can assign four other values to the `BorderStyle` property:

- `bsSingle` creates a main window that's not resizable. Many games and applications based on windows with controls (such as data-entry forms) use this value, simply because resizing these forms makes no sense. Enlarging a form to see an empty area or reducing its size to make some components less visible often doesn't help a program's user (although Delphi's automatic scroll bars partially solve the last problem).
- `bsNone` is used only in very special situations and inside other forms. You'll never see an application with a main window that has no border or caption (except perhaps as an example in a programming book to show you that it makes no sense).
- `bsToolWindow` and `bsSizeToolWin` are related to the specific Win32 extended style `ws_ex_ToolWindow`. This style turns the window into a floating toolbox with a small title font and close button. You should not use this style for the main window of an application.

**Warning** In CLX, the enumeration for the `BorderStyle` property uses slightly different values, prefixed by the letters `fbs` ( form border style): `fbsSingle`, `fbsDialog`, and so on.

To test the effect and behavior of the different values of the `BorderStyle` property, I've written a program called `Borders`, available also as `QBorders` in the CLX version. You've already seen its output in [Figure 7.2](#). However, I suggest you run this example and experiment with it for a while to understand all the differences in the forms. The main form of this program contains only a radio group and a button. The secondary form has no components, and its `Position` property is set to `poDefaultPosOnly`. This value affects the initial position of the secondary form you'll create by clicking the button. (I'll discuss the `Position` property later in this chapter.)

The program code is simple. When you click the button, a new form is dynamically created, depending on the item selected in the radio group:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);  
var  
    NewForm: TForm2;  
begin  
    NewForm := TForm2.Create (Application);  
    NewForm.BorderStyle := TFormBorderStyle (BorderRadioGroup.ItemIndex);  
    NewForm.Caption := BorderRadioGroup.Items[BorderRadioGroup.ItemIndex];  
    NewForm.Show;  
end;
```

This code uses a trick: It casts the number of the selected item into the `TFormBorderStyle` enumeration. This technique works because I've given the radio buttons the same order as the values of the `TFormBorderStyle` enumeration. The `BtnNewFormClick` method then copies the text of the radio button to the caption of the secondary form. This program refers to `TForm2`, the secondary form defined in a secondary unit of the program, which is saved as `Second.pas`. For this reason, to compile the example, you must add the following lines to the implementation

section of the unit of the main form:

```
uses  
    Second;
```

**Tip** Whenever you need to refer to another unit of a program, place the corresponding *uses* statement in the *implementation* portion instead of the *interface* portion if possible. Doing so speeds up the compilation process, results in cleaner code (because the units you include are separate from those included by Delphi), and prevents circular unit compilation errors. To refer to other files within the current project, you can also use the File → Use Unit menu command.

## The Border Icons

Another important element of a form is the presence of icons on its border. By default, a window has a small icon connected to the system menu, a Minimize button, a Maximize button, and a Close button on the far right. You can set different options using the `BorderIcons` property, which has four possible values: `biSystemMenu`, `biMinimize`, `biMaximize`, and `biHelp`.

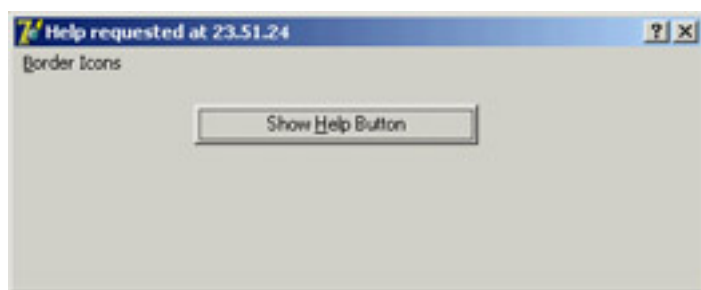
**Note** The `biHelp` border icon enables the "What's this?" Help. When this style is included and the `biMinimize` and `biMaximize` styles are excluded, a question mark appears in the form's title bar. If you click this question mark and then click a component inside the form (but not the form itself!), Delphi activates the help for that object (in a pop-up window in Windows 9x, or in a regular WinHelp window in Windows 2000/XP). This behavior is demonstrated by the `Blcons` example, which has a simple Help file with a page connected to the `HelpContext` property of the button in the middle of the form.

The `Blcons` example demonstrates the behavior of a form with different border icons and shows how to change this property at run time. The example's form is very simple: It has only a menu, with a pull-down containing four menu items, one for each of the possible elements of the set of border icons. I've written a single method, connected with the four commands, that reads the check marks on the menu items to determine the value of the `BorderIcons` property. This code is therefore also a good exercise in working with sets:

```
procedure TForm1.SetIcons(Sender: TObject);  
var  
    BorIco: TBorderIcons;  
begin  
    (Sender as TMenuItem).Checked := not (Sender as TMenuItem).Checked;  
    if SystemMenu1.Checked then  
        BorIco := [biSystemMenu]  
    else  
        BorIco := [];  
    if MaximizeBox1.Checked then  
        Include (BorIco, biMaximize);  
    if MinimizeBox1.Checked then  
        Include (BorIco, biMinimize);  
    if Help1.Checked then  
        Include (BorIco, biHelp);  
    BorderIcons := BorIco;  
end;
```

While running the `Blcons` example, you can easily set and remove the various visual elements of the form's border. You'll immediately see that some of these elements are closely related: If you remove the system menu, all the border icons disappear; if you remove either the Minimize or Maximize button, it becomes grayed; if you remove both of these buttons, they disappear. Notice also that in these last two cases, the corresponding items of the system menu are automatically disabled. This is the standard behavior for any Windows application. When the Maximize and Minimize buttons have been disabled, you can activate the Help button. Actually on Windows 2000 if only one of the Maximize and Minimize buttons has been disabled, the Help button will appear but not work. As a shortcut to obtain this effect, you can click the button inside the form. Also, you can click the button after clicking the Help Menu icon to see a help message, as shown in [Figure 7.3](#). As an extra feature, the program also displays in the caption the time the help was

invoked, by handling the `OnHelp` event of the form. This effect is visible in the figure.



**Figure 7.3:** The `Blcons` example. By selecting the Help border icon and clicking the button, you get the help displayed in the figure.

**Warning** If you look at the `QBIcons` version, built with `CLX`, you will notice that a bug in the library prevents you from changing the border icons at run time. The different design-time settings work fully, so you'll need to modify the program before running it to see any effect at all. At runtime, the program does nothing!

## Setting More Window Styles

The border style and border icons are indicated by two different Delphi properties, which you can use to set the initial value of the corresponding user interface elements. You have seen that besides changing the user interface, these properties affect the behavior of a window. It is important to know that in `VCL` (and obviously not in `CLX`), these border-related properties and the `FormStyle` property primarily correspond to different settings in the *style* and *extended style* of a window. These two terms reflect two parameters of the `CreateWindowEx` API function Delphi uses to create forms.

It is important to acknowledge this fact, because Delphi allows you to modify these two parameters freely by overriding the `CreateParams` virtual method:

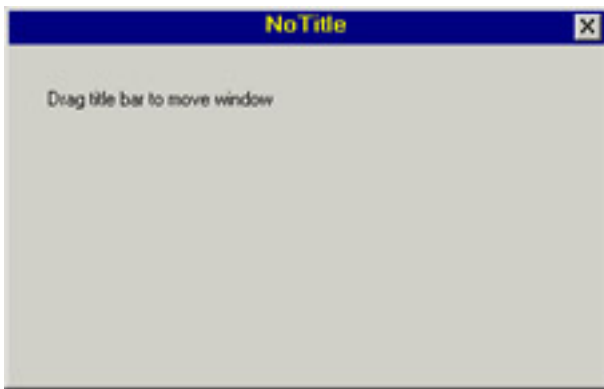
```
public
  procedure CreateParams (var Params: TCreateParams); override;
```

This is the only way to use some of the peculiar window styles that are not directly available through form properties. For a list of window styles and extended styles, see the API help under the topics "CreateWindow" and "CreateWindowEx." You'll notice that the Win32 API has styles for these functions, including those related to tool windows.

To show you how to use this approach, I've written the `NoTitle` example, which lets you create a program with a custom caption. First you must remove the standard caption but keep the resizing frame by setting the corresponding styles:

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := (Params.Style or ws_Popup) and not ws_Caption;
end;
```

To remove the caption, you need to change the overlapped style to a pop-up style; otherwise, the caption will simply stick. To add a custom caption, I've placed a label aligned to the upper border of the form and a small button on the far end. You can see this effect at run time in [Figure 7.4](#).



**Figure 7.4:** The NoTitle example has no real caption but a fake one made with a label.

To make the fake caption work, you have to tell the system that a mouse operation on this area corresponds to a mouse operation on the caption. You can do so by intercepting the `wm_NCHitTest` Windows message, which is frequently sent to Windows to determine where the mouse is. When the hit is in the client area and on the label, you can pretend the mouse is on the caption by setting the proper result:

```
procedure TForm1.WMNCHitTest (var Msg: TWMNCHitTest);  
    // message wm_NCHitTest  
begin  
    inherited;  
    if (Msg.Result = htClient) and  
        (Msg.YPos < Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then  
        Msg.Result := htCaption;  
end;
```

The `GetSystemMetrics` API function used in this listing queries the operating system about the vertical thickness (`cy`) in pixels of the border around a window with a caption but not sizeable. It is important to make this request every time (and not cache the result), because users can customize most of these elements by using the Appearance page of the Desktop options (in Control Panel) and other Windows settings. The small button has a call to the `Close` method in its `OnClick` event handler. The button is kept in its position even when the window is resized by using the `[akTop,akRight]` value for the `Anchors` property. The form also has size constraints, so that a user cannot make it too small, as described in the "[Form Constraints](#)" section later in this chapter.

## Direct Form Input

Having discussed some special capabilities of forms, I'll now move to a very important topic: user input in a form. If you decide to make limited use of components, you might write complex programs as well, receiving input from the mouse and the keyboard. In this chapter, I'll only introduce this topic.

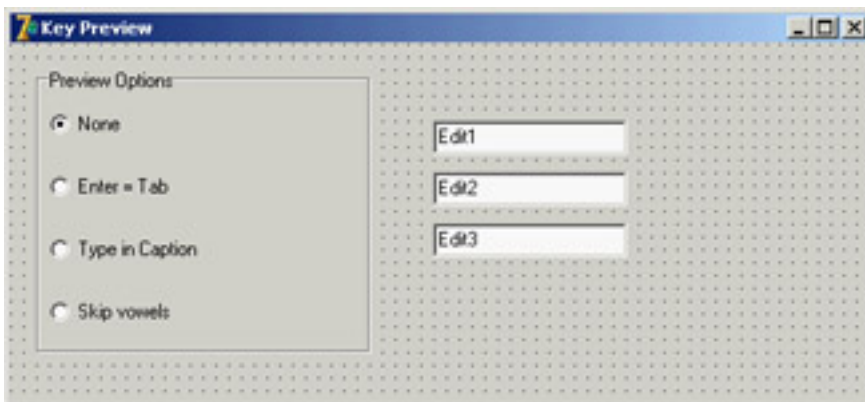
## Supervising Keyboard Input

Generally, forms don't handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. In these cases, you can turn on the form's `KeyPreview` property. Then, even if you have some input controls, the form's `OnKeyPress` event will always be activated for any character-input operation (system and shortcut keys excluded). The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character 0, but the value 0 of the character set, a control character indicated as #0).

The example I've built to demonstrate this approach, `KPreview`, has a form with no special properties (not even `KeyPreview`), a radio group with four options, and some edit boxes, as you can see in [Figure 7.5](#). By default the program does nothing special, except when the various radio buttons are used to enable the key preview:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
    KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```



**Figure 7.5:** The `KPreview` program at design time

Now you'll begin receiving the `OnKeyPress` events, and you can do one of the three actions requested by the three special buttons in the radio group. The action depends on the value of the `ItemIndex` property of the radio group component. This is the reason the event handler is based on a `case` statement:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
    case RadioPreview.ItemIndex of
        ...
```

In the first case, if the value of the `Key` parameter is #13, which corresponds to the Enter key, you disable the operation (setting `Key` to zero) and then mimic the activation of the Tab key. You can do this many ways, but the technique I've chosen is quite particular. I send the `CM_DialogKey` message to the form, passing the code for the Tab key (`VK_TAB`):



```

1: // Enter = Tab
   if Key = #13 then
   begin
       Key := #0;
       Perform (CM_DialogKey, VK_TAB, 0);
   end;

```

**Note** The *CM\_DialogKey* message is an internal, undocumented Delphi message. There are a few of them, and it's quite interesting to build advanced components for them and to use them for special coding, but Borland never described them. For more information on this topic, refer to the section "[Component Messages and Notifications](#)" in [Chapter 9](#). Notice also that this exact message-based coding style is not available under CLX.

To type in the form's caption, the program adds the character to the current `Caption`. There are two special cases. When the Backspace key is pressed, the last character of the string is removed (by copying to the `Caption` all the characters of the current `Caption` but the last one). When the Enter key is pressed, the program stops the operation by resetting the `ItemIndex` property of the radio group control. Here is the code:

```

2: // type in caption
begin
   if Key = #8 then // backspace: remove last char
       Caption := Copy (Caption, 1, Length (Caption) - 1)
   else if Key = #13 then // enter: stop operation
       RadioPreview.ItemIndex := 0
   else // anything else: add character
       Caption := Caption + Key;
   Key := #0;
end;

```

Finally, if the last radio item is selected, the code checks whether the character is a vowel (by testing for its inclusion in a constant "vowel set"). In this case, the character is skipped altogether:

```

3: // skip vowels
if UpCase(Key) in ['A', 'E', 'I', 'O', 'U'] then
   Key := #0;

```

## Getting Mouse Input

When a user clicks one of the mouse buttons over a form (or over a component), Windows sends the application messages. Delphi defines events you can use to write code that responds to these messages. The two basic events are `OnMouseDown`, received when a mouse button is clicked, and `OnMouseUp`, received when the button is released. Another fundamental system message is related to mouse movement: `OnMouseMove`. Although it should be easy to understand the meaning of the three messages—down, up, and move—you may wonder how they relate to the `OnClick` event you have often used up to now.

You have used the `OnClick` event for components, but it is also available for the form. Its general meaning is that the left mouse button has been clicked and released on the same window or component. However, between these two actions, the cursor might have been moved outside the area of the window or component while the left mouse button was held down.

Another difference between the `OnMouseXX` and `OnClick` events is that the latter relates only to the *left* mouse button. Most of the mouse types connected to a Windows PC have two mouse buttons, and some even have three. Usually you refer to these buttons as the left mouse button (generally used for selection), the right mouse button (for accessing shortcut menus), and the middle mouse button (seldom used).

Nowadays most new mouse devices have a button wheel instead of the middle button; users typically use the wheel for scrolling (causing an `OnMouseWheel` event), but they can also press it (generating the `OnMouseWheelDown` and

OnMouseWheelUp events). Mouse wheel events are automatically converted into scrolling events.

---

## Using Windows without a Mouse

A user should always be able to use any Windows application without the mouse. This is not an option; it is a Windows programming rule. Of course, an application might be easier to use with a mouse, but that should never be mandatory. Some users may not have a mouse connected, such as travelers with a small laptop and no space, workers in industrial environments, and bank clerks with other peripherals around.

There is another reason to support the keyboard: Using the mouse is nice, but it tends to be slower. If you are a skilled touch typist, you won't use the mouse to drag a word of text; you'll use shortcut keys to copy and paste it without moving your hands from the keyboard.

For these reasons, you should always set up a proper tab order for a form's components. Remember to add keys for buttons and menu items for keyboard selection, use shortcut keys on menu commands, and so on.

---

## The Parameters of the Mouse Events

All the lower-level mouse events have the same parameters: the usual `Sender` parameter, a `Button` parameter indicating which of the three mouse buttons has been clicked (`mbRight`, `mbLeft`, or `mbCenter`), the `Shift` parameter indicating which of the *mouse-related virtual keys* (the *shift-state modifiers* `Alt`, `Ctrl`, and `Shift`, plus the three mouse buttons) was pressed when the event occurred; and the `x` and `y` coordinates of the position of the mouse in *client area* coordinates of the current window.

Using this information, it is simple to draw a small circle in the position of a left mouse button-down event:

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

**Note** To draw on the form, you use a special property: *Canvas*. A *TCanvas* object has two distinctive features: It holds a collection of drawing tools (such as a pen, a brush, and a font) and it has some drawing methods, which use the current tools. The kind of direct drawing code in this example is not correct, because the on-screen image is not persistent; moving another window over the current one will clear its output. The next example demonstrates the Windows "store-and-draw" approach.

## Dragging and Drawing with the Mouse

To demonstrate a few of the mouse techniques discussed so far, I've built an example based on a form without any components. The program is called `MouseOne` in the VCL version and `QMouseOne` in the CLX version. It displays the current position of the mouse in the form's `Caption`:

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
end;
```

You can use this feature of the program to better understand how the mouse works. Make this test: Run the program (this simple version or the complete one) and resize the windows on the desktop so that the form of the `MouseOne` or `QMouseOne` program is behind another window and inactive but with the title visible. Now move the mouse over the



form, and you'll see that the coordinates change. This behavior means the `OnMouseMove` event is sent to the application even if its window is not active, and it proves what I have mentioned: Mouse messages are always directed to the window under the mouse. The only exception is the mouse capture operation I'll discuss in this same example.

Besides showing the position in the title of the window, the `MouseOne/QMouseOne` example can track mouse movements by painting small pixels on the form if the user keeps the Shift key pressed (again, this direct painting code produces non-persistent output):

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if ssShift in Shift then
    // mark points in yellow
    Canvas.Pixels [X, Y] := clYellow;
end;
```

**Tip** The `TCanvas` class of the CLX library for Kylix 1 and Delphi 6 didn't include the `Pixels` array. Instead, you could call the `DrawPoint` method after setting a proper color for the pen, as I've done in the `QMouseOne` example. Kylix 2 and Delphi 7 re-introduce the `Pixels` array property.

The most interesting feature of this example is its direct mouse-dragging support. Contrary to what you might think, Windows has no system support for dragging, which is implemented in VCL by means of lower-level mouse events and operations. (I discussed an example of dragging from one control to another in [Chapter 6](#).) In VCL, forms cannot originate dragging operations, so in this case you are obliged to use the low-level approach. The aim of this example is to draw a rectangle from the initial position of the dragging operation to the final one, giving users visual clues about the operation they are doing.

The idea behind dragging is quite simple. The program receives a sequence of button-down, mouse-move, and button-up messages. When the button is pressed, dragging begins, although the real actions take place only when the user moves the mouse (without releasing the mouse button) and when dragging terminates (when the button-up message arrives). The problem with this basic approach is that it is not reliable. A window usually receives mouse events only when the mouse is over its client area; so if the user presses the mouse button, moves the mouse onto another window, and then releases the button, the second window will receive the button-up message.

There are two solutions to this problem. One (seldom used) is mouse clipping. Using a Windows API function (`ClipCursor`), you can force the mouse not to leave a certain area of the screen. When you try to move it outside the specified area, it stumbles against an invisible barrier. The second and more common solution is to capture the mouse. When a window captures the mouse, all the subsequent mouse input is sent to that window. This is the approach I've used for the `MouseOne/QMouseOne` example.

The example's code is built around three methods: `FormMouseDown`, `FormMouseMove`, and `FormMouseUp`. Clicking the left mouse button over the form starts the process, setting the `fDragging` Boolean field of the form (which indicates that dragging is in action in the other two methods). The method also uses a `TRect` variable that keeps track of the initial and current position of the dragging. Here is the code:

```
procedure TMouseForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    begin
      fDragging := True;
      Mouse.Capture := Handle;
      fRect.Left := X;
      fRect.Top := Y;
      fRect.BottomRight := fRect.TopLeft;
      dragStart := fRect.TopLeft;
    end;
```

```

Canvas.DrawFocusRect (fRect);
end;
end;

```

An important action of this method is the call to the `SetCapture` API function, obtained by setting the `Capture` property of the global object `Mouse`. Now, even if a user moves the mouse outside the client area, the form still receives all mouse-related messages. You can see that behavior by moving the mouse toward the upper-left corner of the screen; the program shows negative coordinates in the caption.

**Tip** The global `Mouse` object allows you to get global information about the mouse, such as its presence, type, and current position, as well as set some of its global features. This global object hides a few API functions, making your code simpler and more portable. In the VCL the `Capture` property has a `Handle` type, whereas in CLX it has a `TControl` type (the object of the component that captures the mouse). So, the code included in this section will become `Mouse.Capture := self`, as you can see in the `QMouseOne` example.

When dragging is active and the user moves the mouse, the program draws a dotted rectangle corresponding to the mouse's position. The program calls the `DrawFocusRect` method twice. The first time this method is called, it deletes the current image, thanks to the fact that two consecutive calls to `DrawFocusRect` reset the original situation. After updating the position of the rectangle, the program calls the method a second time:

```

procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
    begin
      // remove and redraw the dragging rectangle
      Canvas.DrawFocusRect (fRect);
      if X > dragStart.X then
        fRect.Right := X
      else
        fRect.Left := X;
      if Y > dragStart.Y then
        fRect.Bottom := Y
      else
        fRect.Top := Y;
      Canvas.DrawFocusRect (fRect);
    end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
  end;

```

On Windows 2000 (and other versions) the `DrawFocusRect` function doesn't draw rectangles with a negative size, so the code of the program has been fixed (as you can see above) by comparing the current position with the initial position of the dragging, saved in the `dragStart` point. When the mouse button is released, the program terminates the dragging operation by resetting the `Capture` property of the `Mouse` object (which internally calls the `ReleaseCapture` API function) and by setting the value of the `fDragging` field to `False`:

```

procedure TMouseForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
    begin
      Mouse.Capture := 0; // calls ReleaseCapture
    end
  end;

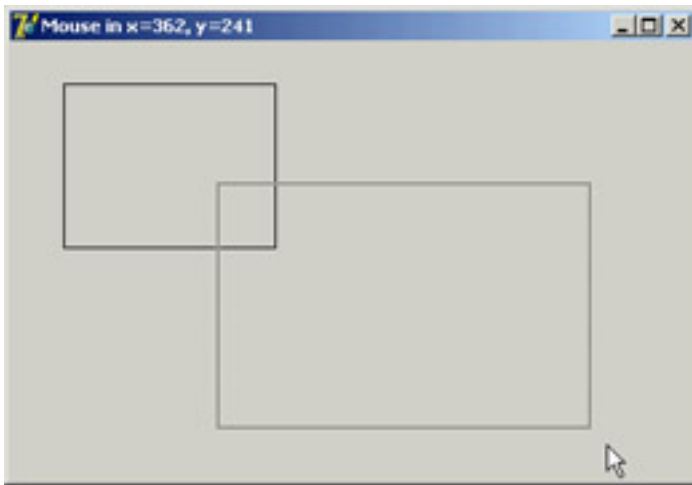
```

```
fDragging := False;  
Invalidate;  
end;  
end;
```

The final call, `Invalidate`, triggers a painting operation and executes the following `OnPaint` event handler:

```
procedure TMouseForm.FormPaint(Sender: TObject);  
begin  
  Canvas.Rectangle (fRect.Left, fRect.Top, fRect.Right, fRect.Bottom);  
end;
```

This makes the output of the form persistent, even if you hide it behind another form. [Figure 7.6](#) shows a previous version of the rectangle and a dragging operation in action.



**Figure 7.6:** During a dragging operation, the `MouseOne` example uses a dotted line to indicate the final area of a rectangle.

## Painting on Forms

Why do you need to handle the `OnPaint` event to produce proper output, and why can't you paint directly over the form canvas? It depends on Windows' default behavior. As you draw on a window, Windows does *not* store the resulting image. When the window is covered, its contents are usually lost.

The reason for this behavior is simple: to save memory. Windows assumes it's "cheaper" in the long run to redraw the screen using code than to dedicate system memory to preserving the display state of a window. It's a classic memory-versus-CPU-cycles trade-off. A color bitmap for a 600×800 image at 256 colors requires about 480 KB. By increasing the color count or the number of pixels, you can easily reach 4 MB of memory for a 1280×1024 resolution at 16 million colors.

In the event that you want to have consistent output for your applications, you can use two techniques. The general solution is to store enough data about the output to be able to reproduce it when the system sends a *painting* requested. An alternative approach is to save the output of the form in a bitmap while you produce it, by placing an `Image` component over the form and drawing on the canvas of this image component.

The first technique, painting, is the common approach to handling output in most windowing systems, aside from particular graphics-oriented programs that store the form's whole image in a bitmap. The approach used to implement painting has a very descriptive name: *store and paint*. When the user clicks a mouse button or performs any other operation, you need to store the position and other elements; then, in the painting method, you use this information to paint the corresponding image.

This approach lets the application repaint its whole surface under any of the possible conditions. If you provide a method to redraw the contents of the form, and if this method is automatically called when a portion of the form has been hidden and needs repainting, you will be able to re-create the output properly.

Because this approach takes two steps, you must be able to execute these two operations in a row, asking the system to repaint the window—without waiting for the system to ask for a repaint operation. You can use several methods to invoke repainting: `Invalidate`, `Update`, `Repaint`, and `Refresh`. The first two correspond to the Windows API functions, and the latter two have been introduced by Delphi:

- The `Invalidate` method informs Windows that the entire surface of the form should be repainted. The most important point is that `Invalidate` does *not* enforce a painting operation immediately. Windows stores the request and responds to it only after the current procedure has been completely executed (unless you call `Application.ProcessMessages` or `Update`) and as soon as no other events are pending in the system. Windows deliberately delays the painting operation because it is one of the most time-consuming operations. At times, with this delay, it is possible to paint the form only after multiple changes have taken place, avoiding multiple consecutive calls to the (slow) paint method.
- The `Update` method asks Windows to update the contents of the form, repainting it immediately. However, this operation will take place only if there is an *invalid area*. This happens if the `Invalidate` method has just been called or as the result of an operation by the user. If there is no invalid area, a call to `Update` has no effect. For this reason, it is common to see a call to `Update` just after a call to `Invalidate`, as is done by the two Delphi methods `Repaint` and `Refresh`.
- The `Repaint` method calls `Invalidate` and `Update` in sequence. As a result, it activates the `OnPaint` event immediately. A slightly different version of this method, called `Refresh`, by default calls `Repaint`. The fact that there are two methods for the same operation dates back to Delphi 1 days, when the two were subtly different.

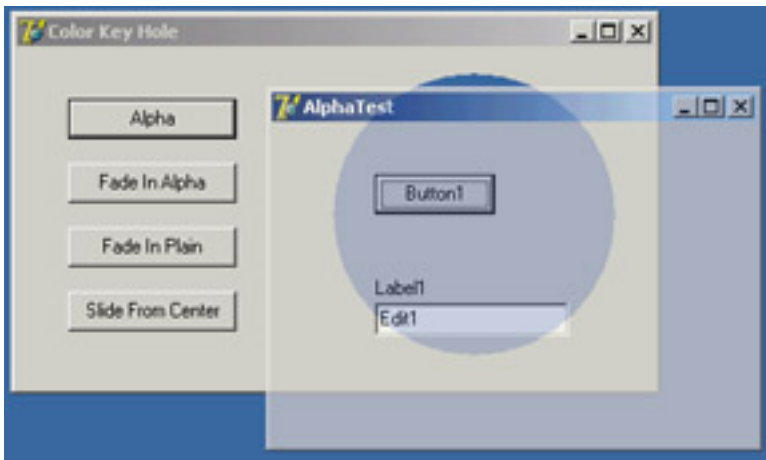
When you need to ask the form for a repaint operation, you should generally call `Invalidate`, following the standard Windows approach. Doing so is particularly important when you need to request this operation frequently, because if Windows takes too much time to update the screen, the requests for repainting can be accumulated into a simple repaint action. The `wm_Paint` message in Windows is a low-priority message; if a request for repainting is pending but other messages are waiting, the other messages are handled before the system performs the paint action.

On the other hand, if you call `Repaint` several times, the screen must be repainted each time before Windows can process other messages; because paint operations are computationally intensive, this behavior can make your application less responsive. Sometimes, however, you want the application to repaint a surface as quickly as possible. In these less-frequent cases, calling `Repaint` is the way to go.

**Note** Another important consideration is that during a paint operation Windows redraws only the so-called *update region*, to speed up the operation. For this reason, if you invalidate a portion of a window, only that area will be repainted. To accomplish this, you can use the *InvalidateRect* and *InvalidateRegion* functions. This feature is a double-edged sword: It is a powerful technique that can improve speed and reduce the flickering caused by frequent repaint operations, but, it can also produce incorrect output. A typical problem occurs when only some of the areas affected by the user operations are modified, while others remain as they were even if the system executes the source code that is supposed to update them. If a painting operation falls outside the update region, the system ignores it, as if it were outside the visible area of a window.

## Unusual Techniques: Alpha Blending, Color Key, and the Animate API

One of the recent Delphi features related to forms is support for new Windows APIs that affect the way forms are displayed (in Windows 2000/XP, but not available under Qt/CLX). *Alpha blending* allows you to merge the content of a form with what's behind it on the screen—functionality you'll rarely need, at least in a business application. The technique is more interesting when applied to bitmap (with the new `AlphaBlend` and `AlphaDIBBlend` API functions) than to a form. In any case, by setting the `AlphaBlend` property of a form to `True` and giving to the `AlphaBlendValue` property a value lower than 255, you'll be able to see in transparency what's behind the form. The lower the `AlphaBlendValue`, the more the form will *fade*. You can see an example of alpha blending in [Figure 7.7](#), taken from the `ColorKeyHole` example.



**Figure 7.7:** The output of the `ColorKeyHole`, showing the effect of the new `TransparentColor` and `AlphaBlend` properties and the `Animate-Window` API

Another unusual Delphi feature is the `TransparentColor` boolean property, which allows you to indicate a transparent color that will be replaced by the background, creating a sort of hole in a form. The actual transparent color is indicated by the `TransparentColorValue` property. Again, you can see an example of this effect in [Figure 7.7](#).

Finally, you can use a native Windows technique, *animated display*, which is not directly supported by Delphi (beyond the display of hints). For example, instead of calling the `Show` method of a form, you can write

```
Form3.Hide;
AnimateWindow (Form3.Handle, 2000, AW_BLEND);
Form3.Show;
```

Notice you have to call the `Show` method at the end for the form to behave properly. You can also obtain a similar animation effect by changing the `AlphaBlendValue` property in a loop. The `AnimateWindow` API can also be used to control how the form is brought into view, starting from the center (with the `AW_CENTER` flag) or from one of its sides (`AW_HOR_POSITIVE`, `AW_HOR_NEGATIVE`, `AW_VER_POSITIVE`, or `AW_VER_NEGATIVE`), as is common for slide shows.

You can apply this same function to windowed controls, obtaining a fade-in effect instead of the usual direct appearance. I have serious doubts about the waste of CPU cycles these animations cause, but I must say that if they are applied properly and in the right program, they can improve the user interface.

## Position, Size, Scrolling, and Scaling

Once you have designed a form in Delphi, you run the program, and you expect the form to show up exactly as you prepared it. However, a user of your application might have a different screen resolution or might want to resize the form (if this is possible, depending on the border style), eventually affecting the user interface. I've already discussed (mainly in [Chapter 7](#)) some techniques related to controls, such as alignment and anchors. Here I'll specifically address elements related to the form as a whole.

Besides differences in the user system, there are many reasons to change Delphi defaults in this area. For example, you might want to run two copies of the program and avoid having all the forms show up in exactly the same place. I've collected many other related elements, including form scrolling, in this portion of the chapter.

### The Form Position

You can use a few properties to set the position of a form. The `Position` property indicates how Delphi determines the initial position of the form. The default `poDesigned` value indicates that the form will appear where you designed it and where you use the positional (`Left` and `Top`) and size (`Width` and `Height`) properties of the form.

Some of the other choices (`poDefault`, `poDefaultPosOnly`, and `poDefaultSizeOnly`) depend on an operating system feature: Using a specific flag, Windows can position and/ or size new windows using a cascade layout. In this way, the positional and size properties you set at design time will be ignored, but if the user runs the application twice the windows won't overlap. The default positions are ignored when the form has a dialog border style. The `poScreenCenter` value displays the form in the center of the screen, with the size you set at design time. This is a common setting for dialog boxes and other secondary forms.

Another property that affects the initial size and position of a window is its *state*. You can use the `WindowState` property at design time to display a maximized or minimized window at startup. This property has only three possible values: `wsNormal`, `wsMinimized`, and `wsMaximized`. If you set a minimized window state, at startup the form will be displayed in the Windows Taskbar. For the main form of an application, this property can be automatically set by specifying the corresponding attributes in a shortcut referring to the application.

Of course, you can maximize or minimize a window at run time, too: Changing the value of the `WindowState` property to `wsMaximized` or `wsNormal` produces the expected effect. Setting the property to `wsMinimized`, however, creates a minimized window that is placed over the Taskbar, not within it. This is not the expected action for a main form, but for a secondary form! The simple solution to this problem is to call the `Minimize` method of the `Application` object. There is also a `Restore` method in the `TApplication` class that you can use when you need to restore a form, although most often the user will do this operation using the system menu's `Restore` command.

### Snapping to the Screen (in Delphi 7)

Forms in Delphi 7 have two new properties:

- The Boolean `ScreenSnap` determines whether the form should be *snapped* to the display area of the screen when it is close to one of its borders.
- The integer `SnapBuffer` determines the distance from the borders considered *close*. Although not a particularly astonishing feature, it's handy to let users snap forms to a side of the screen and take advantage of the entire screen surface; it's particularly handy for applications with multiple forms visible at the same time. Do not set too high a value for the `SnapBuffer` property (something as large as your screen), or the system will become confused!

### The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the `Width` and `Height` properties



or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the `OnResize` event, where you can perform custom actions to adapt the user interface to the new size of the form).

However, if you look at a form's properties in source code or in the online help, you can see that two properties refer to its width and two refer to its height. `Height` and `Width` refer to the size of the form, including the borders; `ClientHeight` and `ClientWidth` refer to the size of the internal area of the form, excluding the borders, caption, scroll bars (if any), and menu bar. The client area of the form is the surface you can use to place components on the form, to create output, and to receive user input. Notice that in CLX, even `Height` and `Width` refer to the size of the internal area of the form.

Because you may be interested in having a certain available area for your components, it often makes more sense to set the client size of a form instead of its global size. Doing so is straightforward, because as you set one of the two client properties, the corresponding form property changes accordingly.

**Tip** In Windows, you can also create output and receive input from the nonclient area of the form—that is, its border. Painting on the border and getting input when you click it are complex issues. If you are interested, look in the Help file at the description of such Windows messages as `wm_NCPaint`, `wm_NCCalcSize`, and `wm_NCHitTest`, and the series of nonclient messages related to the mouse input, such as `wm_NCLButtonDown`. The difficulty of this approach is in combining your code with the default Windows behavior.

## Form Constraints

When you choose a resizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the `wm_Size` message, which generates the `OnResize` event. `OnResize` takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

Delphi provides a specific property for forms and also for all controls: the `Constraints` property. Setting the subproperties of the `Constraints` property to the proper maximum and minimum values creates a form that cannot be resized beyond those limits. Here is an example:

```
object Form1: TForm1
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

Notice that as you set up the `Constraints` property, it has an immediate effect even at design time, changing the size of the form if it is outside the permitted area.

Delphi also uses the maximum constraints for maximized windows, producing an awkward effect. For this reason, you should generally disable the Maximize button of a window that has a maximum size. In some cases maximized windows with a limited size make sense—this is the behavior of Delphi's main window. If you need to change constraints at run time, you can also consider using two specific events, `OnCanResize` and `OnConstrainedResize`. The first of the two can also be used to disable resizing a form or control in given circumstances.

## Scrolling a Form

When you build a simple application, a single form might hold all the components you need. As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms. If you reduce the space occupied by the components, you might add the capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen (or at least bigger than its visible portion on the screen).



Adding a scroll bar to a form is simple. In fact, you don't need to do anything—if you place several components in a big form and then reduce its size, a scroll bar will be added to the form automatically, as long as you haven't changed the value of the `AutoScroll` property from its default of `True`.

Along with `AutoScroll`, forms have two properties, `HorzScrollBar` and `VertScrollBar`, which you can use to set several properties of the two `TFormScrollBar` objects associated with the form. The `Visible` property indicates whether the scroll bar is present, the `Position` property determines the initial status of the scroll thumb, and the `Increment` property determines the effect of clicking one of the arrows at the ends of the scroll bar. The most important property, however, is `Range`.

The `Range` property of a scroll bar determines the virtual size of the form, not the range of values of the scroll bar. Suppose you need a form that will host several components and will therefore need to be 1000 pixels wide. You can use this value to set the "virtual range" of the form, changing the `Range` of the horizontal scroll bar.

The `Position` property of the scroll bar will range from 0 to 1000 minus the current size of the client area. For example, if the client area of the form is 300 pixels wide, you can scroll 700 pixels to see the far end of the form (the thousandth pixel).

## A Scroll Testing Example

To demonstrate the specific case I've just discussed, I've built the `Scroll1` example, which has a virtual form 1000 pixels wide. I've set the range of the horizontal scroll bar to 1000:

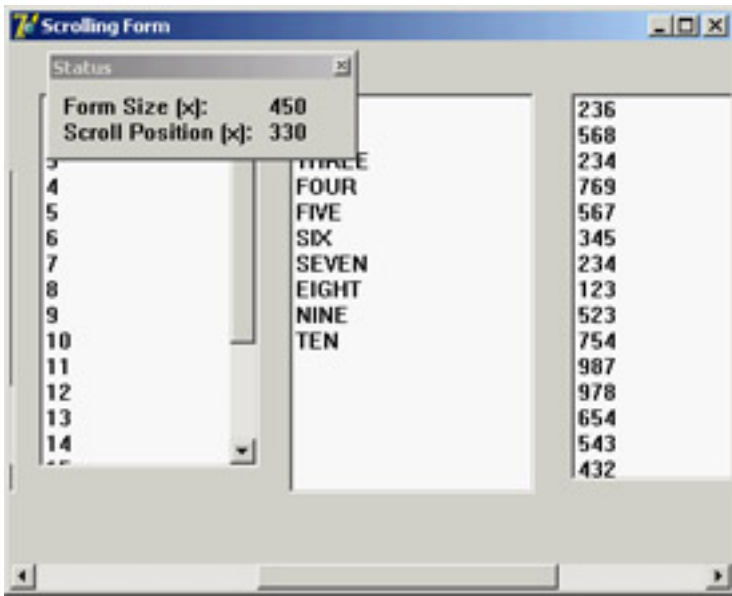
```
object Form1: TForm1
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  OnResize = FormResize
  ...
```

The example's form is filled with meaningless list boxes, and I could have obtained the same scroll-bar range by placing the right-most list box so that its position (`Left`) plus its size (`Width`) equaled 1000.

The interesting part of the example is the presence of a toolbox window displaying the status of the form and of its horizontal scroll bar. This second form has four labels: two with fixed text and two with the output. In addition, the secondary form (called `Status`) has a `bsToolWindow` border style and is a top-most window. You should also set its `Visible` property to `True`, so its window is automatically displayed at startup:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  FormStyle = fsStayOnTop
  Visible = True
object Label1: TLabel...
  ...
```

There isn't much code in this program. Its aim is to update the values in the toolbox each time the form is resized or scrolled (as you can see in [Figure 7.8](#)). The first part is extremely simple. You can handle the `OnResize` event of the form and copy a couple of values to the two labels. The labels are part of another form, so you need to prefix them with the name of the form instance, `Status`:



**Figure 7.8:** The output of the Scroll1 example

```

procedure TForm1.FormResize(Sender: TObject);
begin
    Status.Label3.Caption := IntToStr(ClientWidth);
    Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;

```

If you wanted to change the output each time the user scrolls the contents of the form, you could not use a Delphi event handler, because forms don't have an `OnScroll` event (although stand-alone `ScrollBar` components have one). Omitting this event makes sense, because Delphi forms handle scroll bars automatically in a powerful way. In Windows, by contrast, scroll bars are extremely low-level elements, requiring a lot of coding. Handling the scroll event makes sense only in special cases, such as when you want to keep track precisely of the scrolling operations made by a user.

Here is the code you need to write. First, add a method declaration to the class and associate it with the Windows horizontal scroll message (`wm_HScroll`); then write the code for this procedure, which is almost the same as the code of the `FormResize` method you've seen before:

```

public
    procedure WMHScroll (var ScrollData: TWMScroll); message wm_HScroll;

procedure TForm1.WMHScroll (var ScrollData: TWMScroll);
begin
    inherited;
    Status.Label3.Caption := IntToStr(ClientWidth);
    Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;

```

It's important to add the call to `inherited`, which activates the method related to the same message in the base class form. The `inherited` keyword in Windows message handlers calls the method of the base class you are overriding, which is associated with the corresponding Windows message (even if the procedure name is different). Without this call, the form won't have its default scrolling behavior; that is, it won't scroll at all.

**Note** Because in CLX you cannot handle the low-level scroll messages, there seems to be no easy way to create a program similar to `Scroll1`. This isn't terribly important in real-world applications, because the scrolling system is automatic, and you can probably hook in the CLX library at a lower level.

## Automatic Scrolling

The scroll bar's `Range` property can seem strange until you begin to use it consistently. When you think about it, you'll

start to understand the advantages of the "virtual range" approach. The scroll bar is automatically removed from the form when the client area of the form is big enough to accommodate the virtual size; and when you reduce the size of the form, the scroll bar is added again.

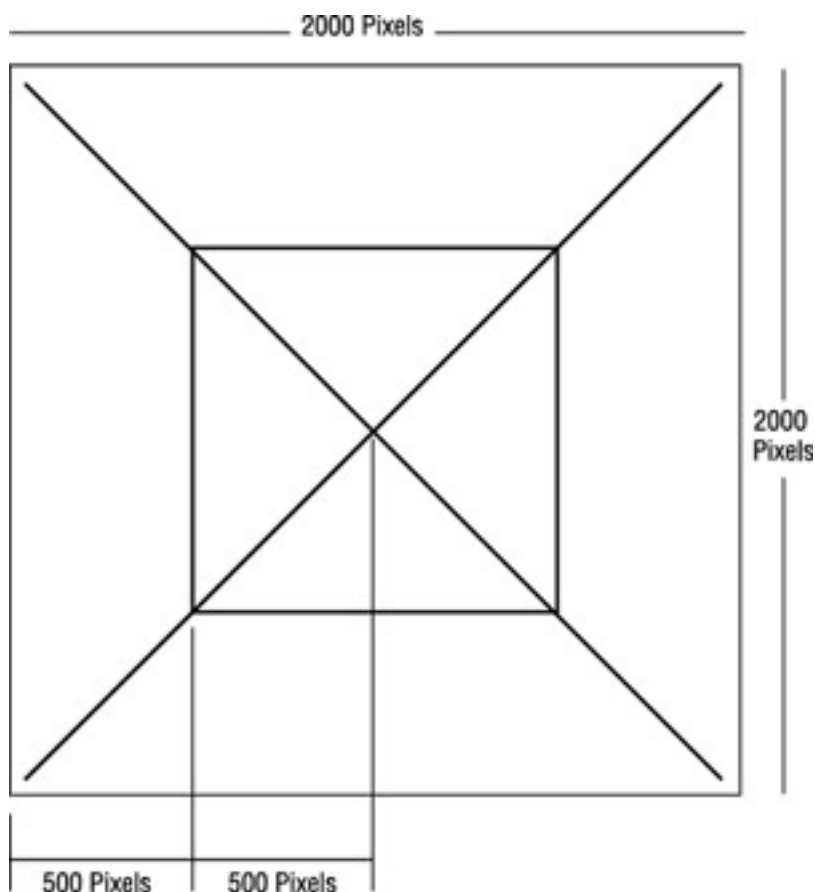
This feature becomes particularly interesting when the `AutoScroll` property of the form is set to `True`. In this case, the extreme positions of the rightmost and lower controls are automatically copied into the `Range` properties of the form's two scroll bars. Automatic scrolling works well in Delphi. In the previous example, the virtual size of the form would be set to the right border of the last list box. This was defined with the following attributes:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Therefore, the horizontal virtual size of the form would be 977 (the sum of the two preceding values). This number is automatically copied into the `Range` field of the `HorzScrollBar` property of the form, unless you change it manually to have a bigger form (as I've done for the `Scroll1` example, setting it to 1000 to leave some space between the last list box and the border of the form). You can see this value in the Object Inspector, or make the following test: Run the program, size the form as you like, and move the scroll thumb to the rightmost position. When you add the size of the form and the position of the thumb, you'll always get 1000, the virtual coordinate of the right-most pixel of the form, whatever the size.

## Scrolling and Form Coordinates

You have just seen that forms can automatically scroll their components. But what happens if you paint directly on the surface of the form? Some problems arise, but their solution is at hand. Suppose you want to draw lines on the virtual surface of a form, as shown in [Figure 7.9](#). Because you probably do not own a monitor capable of displaying 2000 pixels on each axis, you can create a smaller form, add two scroll bars, and set their `Range` property, as I've done in the `Scroll2` example.



**Figure 7.9:** The lines to draw on the virtual surface of the form

If you draw the lines using the virtual coordinates of the form, the image won't display properly. In the `OnPaint` response method, you need to compute the virtual coordinates yourself. Fortunately, doing so is easy, because you know that the virtual `X1` and `Y1` coordinates of the upper-left corner of the client area correspond to the current positions of the two scroll bars:

```
procedure TForm1.FormPaint(Sender: TObject);  
var  
    X1, Y1: Integer;  
begin  
    X1 := HorzScrollBar.Position;  
    Y1 := VertScrollBar.Position;  
  
    // draw a yellow line  
    Canvas.Pen.Width := 30;  
    Canvas.Pen.Color := clYellow;  
    Canvas.MoveTo (30-X1, 30-Y1);  
    Canvas.LineTo (1970-X1, 1970-Y1);  
  
    // and so on ...
```

As a better alternative, instead of computing the proper coordinate for each output operation, you can call the `SetWindowOrgEx` API to move the origin of the coordinates of the `Canvas`. This way, your drawing code will directly refer to virtual coordinates but will be displayed properly:

```
procedure TForm2.FormPaint(Sender: TObject);  
begin  
    SetWindowOrgEx (Canvas.Handle, HorzScrollbar.Position,  
        VertScrollbar.Position, nil);  
  
    // draw a yellow line  
    Canvas.Pen.Width := 30;  
    Canvas.Pen.Color := clYellow;  
    Canvas.MoveTo (30, 30);  
    Canvas.LineTo (1970, 1970);  
  
    // and so on ...
```

This is the version of the program you'll find in the source code of the book. Try using the program and commenting out the `SetWindowOrgEx` call to see what happens if you don't use virtual coordinates: You'll find that the output of the program is not correct—it won't scroll, and the same image will always remain in the same position, regardless of scrolling operations. Notice also that the Qt/CLX version of the program, called `QScroll2`, doesn't use virtual coordinates but simply subtracts the scroll positions from each of the hard-coded coordinates.

## Scaling Forms

When you create a form with multiple components, you can select a fixed-size border or let the user resize the form and automatically add scroll bars to reach the components falling outside the visible portion of the form, as you've just seen. This might also happen because a user of your application has a display driver with a much smaller number of pixels than yours.

Instead of reducing the form size and scrolling the content, you might want to reduce the size of each of the components at the same time. This automatically happens if the user has a system font with a different pixel-per-inch ratio than the one you used for development. To address these problems, Delphi has some nice scaling features, but they aren't fully intuitive.

The form's `ScaleBy` method allows you to scale the form and each of its components. The `PixelsPerInch` and `Scaled` properties let Delphi resize an application automatically when the application is run with a different system font

size, often because of a different screen resolution. In both cases, to make the form scale its window, be sure to also set the `AutoScroll` property to `False`. Otherwise, the contents of the form will be scaled, but the form border itself will not. These two approaches are discussed in the next two sections.

**Note** Form scaling is calculated based on the difference between the font height at run time and the font height at design time. Scaling ensures that edit and other controls are large enough to display their text using the user's font preferences without clipping the text. The form scales as well, as you will see later, but the main point is to make edit and other controls readable.

## Manual Form Scaling

Any time you want to scale a form, including its components, you can use the `ScaleBy` method, which has two integer parameters, a multiplier and a divisor—it's a fraction. For example, this statement reduces the size of the current form to three-quarters of its original size:

```
ScaleBy (3, 4);
```

The same effect can be obtained by using

```
ScaleBy (75, 100);
```

When you scale a form, all the proportions are maintained, but if you go below or above certain limits, the text strings can alter their proportions slightly. The problem is that in Windows, components can be placed and sized only in whole pixels, whereas scaling almost always involves multiplying by fractional numbers. So, any fractional portion of a component's origin or size will be truncated.

I've built a simple example, `Scale` (or `QScale`), to show how you can scale a form manually, responding to a request by the user. The application form has two buttons, a label, an edit box, and an `UpDown` control connected to it (via its `Associate` property). With this setting, a user can type numbers in the edit box or click the two small arrows to increase or decrease the value (by the amount indicated by the `Increment` property). To extract the input value, you can use the `Text` property of the edit box or the `Position` of the `UpDown` control. When you click the `Do Scale` button, the current input value is used to determine the scaling percentage of the form:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);  
begin  
    AmountScaled := UpDown1.Position;  
    ScaleBy (AmountScaled, 100);  
    UpDown1.Height := Edit1.Height;  
    ScaleButton.Enabled := False;  
    RestoreButton.Enabled := True;  
end;
```

This method stores the current input value in the form's `AmountScaled` private field and enables the `Restore` button, disabling the button that was clicked. Later, when the user clicks the `Restore` button, the opposite scaling takes place. By having to restore the form before another scaling operation takes place, I avoid an accumulation of round-off errors. I've also added a line to set the `Height` of the `UpDown` component to the same `Height` as the edit box it is attached to. This prevents small differences between the two, due to scaling problems of the `UpDown` control.

**Note** If you want to scale the text of the form properly, including the captions of components, the items in list boxes, and so on, you should use `TrueType` fonts exclusively. The system font (MS Sans Serif) doesn't scale well. The font issue is important because the size of many components depends on the text height of their captions, and if the caption does not scale well, the component might not work properly. For this reason, in the `Scale` example I've used an `Arial` font.

The same scaling technique also works in `CLX`, as you can see by running the `QScale` example. The only real difference is that I replaced the `UpDown` component (and the related edit box) with a `SpinEdit` control, because the former is not available in `Qt`.

## Automatic Form Scaling

Instead of playing with the `ScaleBy` method, you can have Delphi do the work for you. When Delphi starts, it asks the system for the display configuration and stores the value in the `PixelsPerInch` property of the `Screen` object, a special global object of VCL that's available in any application.

`PixelsPerInch` sounds like it has something to do with the pixel resolution of the screen (actually available in `Screen.Height` and `Screen.Width`), but unfortunately, it doesn't. If you change your screen resolution from 640×480 to 800×600 to 1024×768 or even 1600×1280, you will find that Windows reports the same `PixelsPerInch` value in all cases, unless you change the system font. `PixelsPerInch` really refers to the screen pixel resolution for which the currently installed system font was designed. When a user changes the system font scale, usually to make menus and other text easier to read, the user will expect all applications to honor those settings. An application that does not reflect user desktop preferences will look out of place and, in extreme cases, may be unusable to visually impaired users who rely on very large fonts and high-contrast color schemes.

The most common `PixelsPerInch` values are 96 (small fonts) and 120 (large fonts), but other values are possible. Newer versions of Windows let the user set the system font size to an arbitrary scale. At design time, the `PixelsPerInch` value of the screen, which is a read-only property, is copied to every form of the application. Delphi then uses the value of `PixelsPerInch`, if the `Scaled` property is set to `True`, to resize the form when the application starts.

As I've mentioned, both automatic scaling and the scaling performed by the `ScaleBy` method operate on components by changing the size of the font. The size of each control depends on the font it uses. With automatic scaling, the value of the form's `PixelsPerInch` property (the design-time value) is compared to the current system value (indicated by the corresponding property of the `Screen` object), and the result is used to change the font of the components on the form. To improve the accuracy of this code, the final height of the text is compared to the design-time height of the text, and its size is adjusted if the heights do not match.

Thanks to Delphi automatic support, the same application running on a system with a different system font size automatically scales itself, without any specific code. The application's edit controls will be the correct size to display their text in the user's preferred font size, and the form will be the correct size to contain those controls. Although automatic scaling has problems in some special cases, if you comply with the following rules, you should get good results:

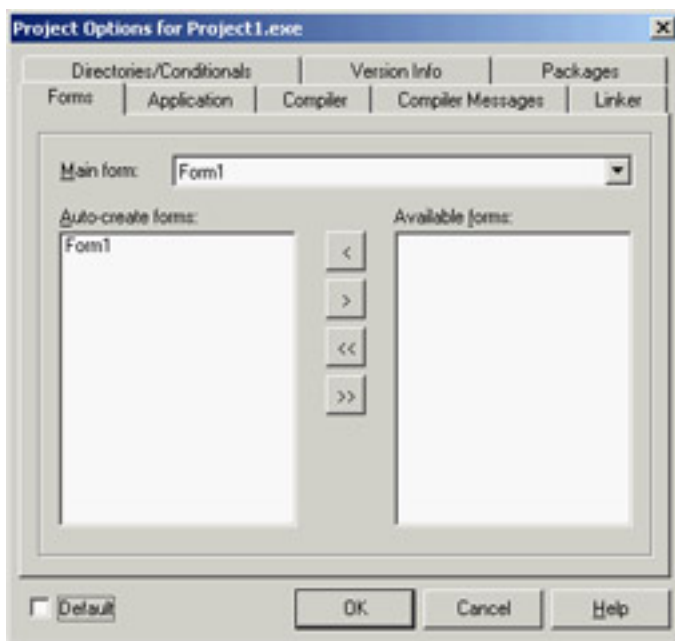
- Set the `Scaled` property of forms to `True` (the default value).
- Use only TrueType fonts.
- Use Windows small fonts (96 dpi) on the computer you use to develop the forms.
- Set the `AutoScroll` property to `False` if you want to scale the form and not just the controls inside it. (`AutoScroll` defaults to `True`, so don't forget this step.)
- Set the form position either near the upper-left corner or in the center of the screen (with the `poScreenCenter` value) to avoid having an out-of-screen form.

## Creating and Closing Forms

Up to now I have ignored the issue of form creation. You know that when the form is created, you receive the `OnCreate` event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in the project's source file:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box (see [Figure 7.10](#)). In this dialog box, you can decide whether the form should be automatically created. If you disable automatic creation, the project's initialization code becomes the following:



**Figure 7.10:** The Forms page of the Delphi Project Options dialog box

```
begin
  Applications.Initialize;
  Application.Run;
end.
```

If you now run this program, nothing happens. It terminates immediately because no main window is created. The call to the application's `CreateForm` method creates a new instance of the form class passed as the first parameter and assigns it to the variable passed as the second parameter.

Something else happens behind the scenes. When `CreateForm` is called, if there is currently no main form, the current form is assigned to the application's `MainForm` property. For this reason, the form indicated as Main Form in the dialog box shown in [Figure 7.10](#) corresponds to the first call to the application's `CreateForm` method (that is, when several forms are created at startup).

The same holds for closing the application. Closing the main form terminates the application, regardless of the other forms. If you want to perform this operation from the program's code, call the `Close` method of the main form, as you've done several times in past examples.



## Form Creation Events

Regardless of the manual or automatic creation of forms, when a form is created, you can intercept many events. Form-creation events are fired in the following order:

1. `OnCreate` indicates that the form is being created.
2. `OnShow` indicates that the form is being displayed. Besides main forms, this event happens after you set the `Visible` property of the form to `True` or call the `Show` or `ShowModal` method. This event is fired again if the form is hidden and then displayed again.
3. `OnActivate` indicates that the form becomes the active form within the application. This event is fired every time you move from another form of the application to the current one.
4. Other events, including `OnResize` and `OnPaint`, indicate operations always done at startup but then repeated many times.

**Note** In Qt, the `OnResize` event won't fire as it does in Windows when the form is created. To make the code more portable from Delphi to Kylix, CLX simulates this event, although it would make more sense to tweak the VCL to avoid this odd behavior (a comment in the CLX source code describes this situation).

As you can see in the previous list, every event has a specific role apart from form initialization, except `OnCreate`, which is guaranteed to be called only once as the form is created.

However, there is an alternative approach to adding initialization code to a form: overriding the constructor. This is usually done as follows:

```
constructor TForm1.Create(AOwner: TComponent);  
begin  
    inherited Create (AOwner);  
    // extra initialization code  
end;
```

Before the call to the `Create` method of the base class, the properties of the form are still not loaded and the internal components are not available. For this reason the standard approach is to call the base class constructor first and then do the custom operations.

**Note** Up to version 3, Delphi used a different creation order, which has led to the `OldCreateOrder` compatibility property of the VCL. When this property is set to the default value of `False`, all the code in a form constructor is executed before the code in the `OnCreate` event handler (which is fired by the special `AfterConstruction` method). If you enable the old creation order, instead, the constructor's `inherited` call leads to the call of the `OnCreate` event handler. You can examine the behavior of the `CreateOrd` example using the two values of the `OldCreateOrder` property.

## Closing a Form

When you close the form using the `Close` method or by the usual means (Alt+F4, the system menu, or the Close button), the `OnCloseQuery` event is called. In this event, you can ask the user to confirm the action, particularly if there is unsaved data in the form. Here is an example of the code you can write:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);  
begin  
    if MessageDlg ('Are you sure you want to exit?', mtConfirmation,  
        [mbYes, mbNo], 0) = mrNo then  
        CanClose := False;  
end;
```

If `OnCloseQuery` indicates that the form should still be closed, the `OnClose` event is called. The third step is to call



the `OnDestroy` event, which is the opposite of the `OnCreate` event and is generally used to de-allocate objects related to the form and free the corresponding memory.

**Note** To be more precise, the `BeforeDestruction` method generates an `OnDestroy` event before the `Destroy` destructor is called. That is, unless you have set the `OldCreateOrder` property to `True`, in which case Delphi uses a different closing sequence.

So, what is the use of the intermediate `OnClose` event? In this method, you have another chance to avoid closing the application, or you can specify alternative "close actions." The method has an `Action` parameter passed by reference. You can assign the following values to this parameter:

**caNone** The form is not allowed to close. This corresponds to setting the `CanClose` parameter of the `OnCloseQuery` method to `False`.

**caHide** The form is not closed, just hidden. This makes sense if there are other forms in the application; otherwise, the program terminates. This is the default for secondary forms, and it's the reason I had to handle the `OnClose` event in the previous example to close the secondary forms.

**caFree** The form is closed, freeing its memory, and the application eventually terminates if this was the main form. This is the default action for the main form and the action you should use when you create multiple forms dynamically (if you want to remove the windows and destroy the corresponding Delphi object as the form closes).

**caMinimize** The form is not closed but only minimized. This is the default action for MDI child forms.

**Note** When a user shuts down Windows, the `OnCloseQuery` event is activated, and a program can use it to stop the shutdown process. In this case, the `OnClose` event is not called even if `OnCloseQuery` sets the `CanClose` parameter to `True`.

## Dialog Boxes and Other Secondary Forms

When you write a program, there is no significant difference between a dialog box and another secondary form, aside from the border, the border icons, and similar user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have nonmodal—or *modeless*—dialog boxes.

So, if you think dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. You must consider two different elements: The form's border and its user interface determine whether it looks like a dialog box; the use of two different methods (`Show` and `ShowModal`) to display the secondary form determines its behavior (modeless or modal).

### Adding a Second Form to a Program

To add a second form to an application, you click the New Form button on the Delphi toolbar or use the File → New → Form menu command. As an alternative, you can select File → New → Other, move to the Forms or Dialogs page, and choose one of the available form templates or form wizards.

If you have two forms in a project, you can use the View Form or View Unit button on the Delphi toolbar to navigate through them at design time. You can also choose which form is the main one and which forms should be automatically created at startup using the Forms page of the Project Options dialog box. This information is reflected in the source code of the project file.

**Tip** Secondary forms are automatically created in the project source-code file depending on the status of the Auto Create Forms check box on the Designer page of the Environment Options dialog box. Although automatic creation is the simplest and most reliable approach for novice developers and quick-and-dirty projects, I suggest that you disable this check box for any serious development. When your application contains hundreds of forms, they shouldn't all be created at application startup. Create instances of secondary forms when and where you need them, and free them when you're done.

Once you have prepared the secondary form, you can set its `Visible` property to `True`, and both forms will show up as the program starts. In general, the secondary forms of an application are left "invisible" and are then displayed by calling the `Show` method (or setting the `Visible` property at run time). If you use the `Show` function, the second form will be displayed as modeless, so you can move back to the first form while the second is still visible. To close the second form, you might use its system menu or click a button or menu item that calls the `Close` method. As you've just seen, the default close action (see the `OnClose` event) for a secondary form is simply to hide it, so the secondary form is not destroyed when it is closed. It is kept in memory (again, not always the best approach) and is available if you want to show it again.

### Creating Secondary Forms at Run Time

Unless you create all the forms when the program starts, you'll need to check whether a form exists and create it if necessary. The simplest case occurs when you want to create multiple copies of the same form at run time. In the MultiWin/QMultiWin example, I've done this by writing the following code:

```
with TForm3.Create (Application) do  
  Show;
```

Every time you click the button, a new copy of the form is created. Notice that I don't use the `Form3` global variable, because it doesn't make much sense to assign this variable a new value every time you create a new form object. The important thing, however, is not to refer to the global `Form3` object in the code of the form or in other portions of the application. The `Form3` variable will invariably be a pointer to `nil`. My suggestion, in such a case, is to remove it from the unit to avoid any confusion.

**Tip** In the code of a form that can have multiple instances, you should never explicitly refer to the form by using the global variable Delphi sets up for it. For example, suppose that in the code for *TForm3* you refer to *Form3.Caption*. If you create a second object of the same type (the class *TForm3*), the expression *Form3.Caption* will refer to the caption of the form object referenced by the *Form3* variable, which might not be the current object executing the code. To avoid this problem, refer to the *Caption* property in the form's method to indicate the caption of the current form object, and use the *Self* keyword when you need a specific reference to the object of the current form. To avoid any problem when creating multiple copies of a form, I suggest removing the global form object from the interface portion of the unit declaring the form. This global variable is required only for the automatic form creation.

When you create multiple copies of a form dynamically, remember to destroy each form object as is it closed, by handling the corresponding event:

```
procedure TForm3.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := caFree;  
end;
```

Failing to do so will result in a lot of memory consumption, because all the forms you create (both the windows and the Delphi objects) will be kept in memory and hidden from view.

## Creating Single-Instance Secondary Forms

Now let's focus on the dynamic creation of a form, in a program that accounts for only one copy of the form at a time. Creating a modal form is quite simple, because the dialog box can be destroyed when it is closed, with code like this:

```
var  
    Modal: TForm4;  
begin  
    Modal := TForm4.Create (Application);  
    try  
        Modal.ShowModal;  
    finally  
        Modal.Free;  
    end;
```

Because the *ShowModal* call can raise an exception, you should write it in a *try* block followed by a *finally* block to make sure the object will be de-allocated. Usually this block also includes code that initializes the dialog box before displaying it and code that extracts the values set by the user before destroying the form. The final values are read-only if the result of the *ShowModal* function is *mrOK*, as you'll see in the next example.

The situation is a little more complex when you want to display only one copy of a modeless form. You have to create the form, if it is not already available, and then show it:

```
if not Assigned (Form2) then  
    Form2 := TForm2.Create (Application);  
Form2.Show;
```

With this code, the form is created the first time it is required and then is kept in memory, visible on the screen or hidden from view. To avoid using up memory and system resources unnecessarily, you'll want to destroy the secondary form when it is closed. You can do that by writing a handler for the *OnClose* event:

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := caFree;  
    // important: set pointer to nil!  
    Form2 := nil;  
end;
```

Notice that after you destroy the form, the global *Form2* variable is set to *nil*, which contradicts the rule set earlier for

forms with multiple instances, but as this is a single-instance we are in the exact opposite case. Without this code, closing the form would destroy its object, but the `Form2` variable would still refer to the original memory location. At this point, if you try to show the form once more with the `btnSingleClick` method shown earlier, the `if not Assigned ( )` test will succeed, because it checks whether the `Form2` variable is `nil`. The code fails to create a new object, and the `Show` method (invoked on a nonexistent object) will result in a system memory error.

As an experiment, you can generate this error by removing the last line of the previous listing. As you have seen, the solution is to set the `Form2` object to `nil` when the object is destroyed, so that properly written code will "see" that a new form must be created before using it. Again, experimenting with the `MultiWin/QMultiWin` example can prove useful to test various conditions. (I haven't shown any screens from this example because the forms it displays are totally empty, except for the main form, which has three buttons.)

**Note** Setting the form variable to `nil` makes sense—and works—if there is to be only one instance of the form present at any given instant. If you want to create multiple copies of a form, you'll have to use other techniques to keep track of them. Also keep in mind that in this case you cannot use the `FreeAndNil` procedure, because you cannot call `Free` on `Form2`—you cannot destroy the form before its event handlers have finished executing.

Team LiB

◀ PREVIOUS

NEXT ▶

## Creating a Dialog Box

I stated earlier in this chapter that a dialog box is not very different from other forms. To build a dialog box instead of a form, you just select the `bsDialog` value for the `BorderStyle` property. With this simple change, the interface of the form becomes like that of a dialog box, with no system icon and no Minimize and Maximize boxes. Of course, such a form has the typical thick dialog box border, which is non-resizable.

Once you have built a dialog box form, you can display it as a modal or modeless window using the two usual show methods (`Show` and `ShowModal`). Modal dialog boxes, however, are more common than modeless ones. This is the reverse of forms; modal forms should generally be avoided, because a user won't expect them.

## The Dialog Box of the RefList Example

In [Chapter 5](#), "Visual Controls," we explored the `RefList/QRefList` program, which used a `ListView` control to display references to books, magazines, websites, and more. In the `RefList2` version (and its `QRefList2 CLX` counterpart), I added to the basic version a dialog box that's used in two different circumstances: adding new items to the list and editing existing items.

**Warning** The CLX `ListView` component has a problem. In case you activate the check boxes and then disable them, the images will disappear. This is the behavior of the `QRefList` example of [Chapter 5](#). In the `QRefList2` version I've added code to reassign the `ImageIndex` property of each item as a workaround to this bug.

The only particularly interesting feature of this form in the VCL example is the use of the `ComboBoxEx` component, which is attached to the same `ImageList` used by the `ListView` control of the main form. The drop-down items of the list, used to select a type of reference, include both a textual description and the corresponding image.

As I mentioned, this dialog box is used in two different cases. The first takes place as the user selects `File → Add Items` from the menu:

```
procedure TForm1.AddItem1Click(Sender: TObject);
var
 NewItem: TListItem;
begin
  FormItem.Caption := 'New Item';
  FormItem.Clear;
  if FormItem.ShowModal = mrOK then
  begin
    NewItem := ListView1.Items.Add;
    NewItem.Caption := FormItem.EditReference.Text;
    NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
    NewItem.SubItems.Add (FormItem.EditAuthor.Text);
    NewItem.SubItems.Add (FormItem.EditCountry.Text);
  end;
end;
```

Besides setting the proper caption for the form, this procedure initializes the dialog box, because you are entering a new value. If the user clicks OK, however, the program adds a new item to the list view and sets all its values. To empty the dialog's edit boxes, the program calls the custom `Clear` method, which resets the text of each edit box control:

```
procedure TFormItem.Clear;
var
  I: Integer;
begin
  // clear each edit box
  for I := 0 to ControlCount - 1 do
```

```

if Controls [I] is TEdit then
    TEdit (Controls[I]).Text := '';
end;

```

Editing an existing item requires a slightly different approach. First, the current values are moved to the dialog box before it is displayed. Second, if the user clicks OK, the program modifies the current list item instead of creating a new one. Here is the code:

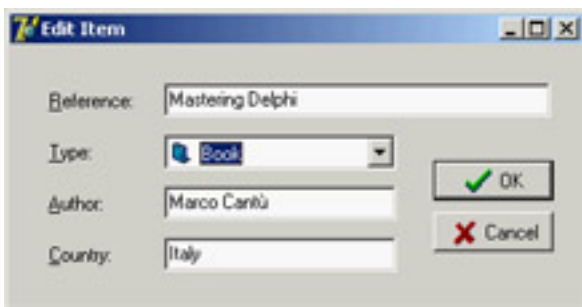
```

procedure TForm1.ListView1DbClick(Sender: TObject);
begin
    if ListView1.Selected <> nil then
    begin
        // dialog initialization
        FormItem.Caption := 'Edit Item';
        FormItem.EditReference.Text := ListView1.Selected.Caption;
        FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
        FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
        FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

        // show it
        if FormItem.ShowModal = mrOK then
        begin
            // read the new values
            ListView1.Selected.Caption := FormItem.EditReference.Text;
            ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
            ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
            ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
        end;
    end;
end;

```

You can see the effect of this code in [Figure 7.11](#). Notice that the code used to read the value of a new item or modified item is similar. In general, you should try to avoid this type of duplicated code and perhaps place the shared code statements in a method added to the dialog box. In this case, the method could receive as parameter a `TListItem` object and copy the proper values into it.

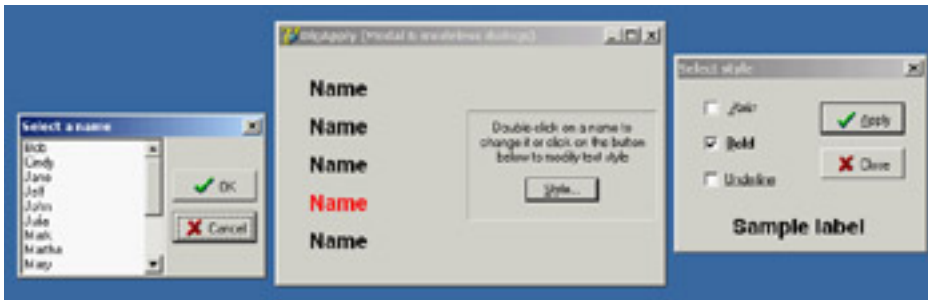


**Figure 7.11:** The dialog box of the RefList2 example used in edit mode. Notice the ComboBoxEx graphical component in use.

**Note** What happens internally when the user clicks the OK or Cancel button in the dialog box? A modal dialog box is closed by setting its `ModalResult` property, and it returns the value of this property. You can indicate the return value by setting the `ModalResult` property of the button. When the user clicks the button, its `ModalResult` value is copied to the form, which closes the form and returns the value as the result of the `ShowModal` function.

## A Modeless Dialog Box

The second example of dialog boxes shows a more complex modal dialog box that uses the standard approach as well as a modeless dialog box. The main form of the DlgApply example (and of the identical CLX-based QDlgApply demo) has five labels with names, as you can see in [Figure 7.12](#) and by viewing the source code of the example.



**Figure 7.12:** The three forms (a main form and two dialog boxes) of the DlgApply example at run time

If the user clicks a name, its color changes to red; if the user double-clicks it, the program displays a modal dialog box with a list of names to choose from. If the user clicks the Style button, a modeless dialog box appears, allowing the user to change the font style of the main form's labels. The five labels on the main form are connected to two methods, one for the `OnClick` event and the second for the `OnDoubleClick` event. The first method turns the last label a user clicked red, resetting all the others to black (they have the `Tag` property set to 1, as a sort of group index). Notice that the same method is associated with all the labels:

```
procedure TForm1.LabelClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount - 1 do
    if (Components[I] is TLabel) and (Components[I].Tag = 1) then
      TLabel (Components[I]).Font.Color := clBlack;
  // set the color of the clicked label to red
  (Sender as TLabel).Font.Color := clRed;
end;
```

The second method common to all the labels is the `OnDoubleClick` event handler. The `LabelDoubleClick` method selects the `Caption` of the current label (indicated by the `Sender` parameter) in the list box of the dialog and then shows the modal dialog box. If the user closes the dialog box by clicking OK and a list item is selected, the selection is copied back to the label's caption:

```
procedure TForm1.LabelDoubleClick(Sender: TObject);
begin
  with ListDial.ListBox1 do
    begin
      // select the current name in the list box
      ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
      // show the modal dialog box, checking the return value
      if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
        // copy the selected item to the label
        (Sender as TLabel).Caption := Items [ItemIndex];
    end;
end;
```

**Tip** Notice that all the code used to customize the modal dialog box is in the `LabelDoubleClick` method of the main form. The form of this dialog box has no added code.

The modeless dialog box, by contrast, has a lot of coding behind it. The main form displays the dialog box when the Style button is clicked (notice that the button caption ends with three dots to indicate that it leads to a dialog box), by calling its `Show` method. You can see the dialog box running in [Figure 7.12](#).



Two buttons, Apply and Close, replace the OK and Cancel buttons in a modeless dialog box. (The fastest way to obtain these buttons is to select the `bkOK` or `bkCancel` value for the `Kind` property and then edit the `Caption`.) At times, you may see a Cancel button that works as a Close button, but the OK button in a modeless dialog box usually has no meaning. Instead, one or more buttons might perform specific actions on the main window, such as Apply, Change Style, Replace, Delete, and so on.

If the user clicks one of the check boxes in this modeless dialog box, the style of the sample label's text at the bottom changes accordingly. You accomplish this by adding or removing the specific flag that indicates the style, as in the following `OnClick` event handler:

```
procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);  
begin  
    if ItalicCheckBox.Checked then  
        LabelSample.Font.Style := LabelSample.Font.Style + [fsItalic]  
    else  
        LabelSample.Font.Style := LabelSample.Font.Style - [fsItalic];  
end;
```

When the user clicks the Apply button, the program copies the style of the sample label to each of the form's labels, rather than consider the values of the check boxes:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);  
begin  
    Form1.Label1.Font.Style := LabelSample.Font.Style;  
    Form1.Label2.Font.Style := LabelSample.Font.Style;  
    ...  
end;
```

As an alternative, instead of referring to each label directly, you can look for it by calling the `FindComponent` method of the form, passing the label name as a parameter, and then casting the result to the `TLabel` type. The advantage of this approach is that you can create the names of the various labels with a `for` loop:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);  
var  
    I: Integer;  
begin  
    for I := 1 to 5 do  
        (Form1.FindComponent ('Label' + IntToStr (I)) as TLabel).Font.Style :=  
            LabelSample.Font.Style;  
end;
```

**Tip** The `ApplyBitBtnClick` method could also be written by scanning the `Controls` array in a loop, as I've done in other examples. I decided to use the `FindComponent` method here to demonstrate a different technique.

This second version of the code is certainly slower, because it has more operations to do, but you won't notice the difference because it is still very fast. Of course, this approach is also more flexible; if you add a new label, you only need to fix the higher limit of the `for` loop, provided all the labels have consecutive numbers.

Notice that when the user clicks the Apply button, the dialog box does not close—only the Close button has this effect. Consider also that this dialog box needs no initialization code because the form is not destroyed, and its components maintain their status each time the dialog box is displayed. Notice, however, that in the CLX version of the program, `QDlgApply`, the dialog is modal, even if it is called with the `Show` method.



## Predefined Dialog Boxes

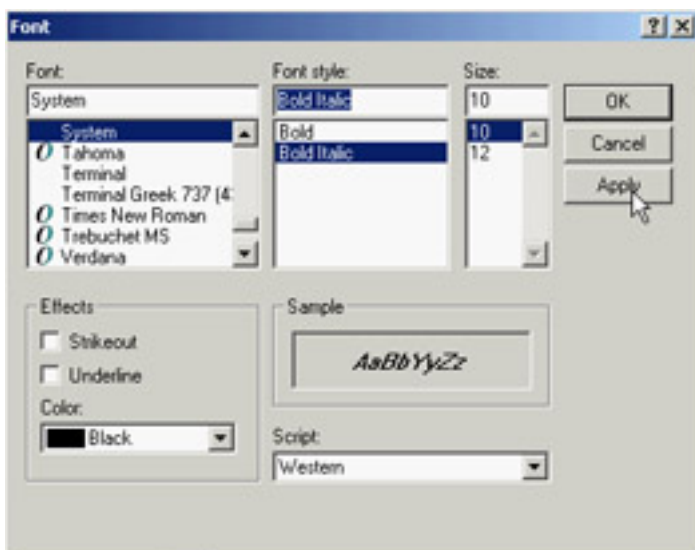
Besides building your own dialog boxes, Delphi allows you to use some default dialog boxes of various kinds. Some are predefined by Windows; others are simple dialog boxes (such as message boxes) displayed by a Delphi routine. The Delphi Component Palette contains a page of dialog box components. Each of these dialog boxes—known as *Windows common dialogs*—is defined in the system library `ComDlg32.DLL`.

## Windows Common Dialogs

I have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the `Execute` method, returning a Boolean value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, I've built the `CommDlgTest` program.

I'll highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The `OpenDialog` Component can be customized by setting different file extension filters using the `Filter` property, which has a handy editor and can be assigned a value directly with a string like `Text File (*.txt)|*.txt`. Another useful feature lets the dialog check whether the extension of the selected file matches the default extension, by checking the `ofExtensionDifferent` flag of the `Options` property after executing the dialog. Finally, this dialog allows multiple selections by setting its `ofAllowMultiSelect` option. In this case you can get the list of selected files by looking at the `Files` string list property.
- The `SaveDialog` component is used in similar ways and has similar properties, although of course you cannot select multiple files.
- The `OpenPictureDialog` and `SavePictureDialog` components provide similar features but have a customized form that shows a preview of an image. It only makes sense to use these components for opening or saving graphical files.
- The `FontDialog` component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (WYSIWYG), or only TrueType fonts. You can show or hide the portion related to the special effects, and obtain other different versions by setting its `Options` property. You can also activate an `Apply` button by providing an event handler for its `OnApply` event and using the `fdApplyButton` option. A `Font` dialog box with an `Apply` button (see [Figure 7.13](#)) behaves almost like a modeless dialog box (but isn't one).



**Figure 7.13:** The Font selection dialog box with an Apply button

- The `ColorDialog` component is used with different options to show the dialog fully open at first or to prevent it from opening fully. These settings are the `cdFullOpen` or `cdPreventFullOpen` values of the `Options` property.
- The Find and Replace dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the `CommDlgTest` example. The custom code is connected to the buttons of the two dialog boxes by providing the `OnFind` and `OnReplace` events.

**Note** Qt offers a similar set of predefined dialog boxes, but the set of options is often more limited. I've created the `QCommDlg` version of the example you can use to experiment with these settings. The CLX program has fewer menu items, because some of the options are not available; there are other minimal changes in the source code.

## A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. You can use many Delphi procedures and functions to display simple dialog boxes:

- The `MessageDlg` function shows a customizable message box with one or more buttons and usually a bitmap. The `MessageDlgPos` function is similar to the `MessageDlg` function, but the message box is displayed in a given position, not in the center of the screen (unless you use the `-1, -1` position to make it appear in the screen center).
- The `ShowMessage` procedure displays a simpler message box with the application name as the caption and an OK button. The `ShowMessagePos` procedure does the same, but you also indicate the position of the message box. The `ShowMessageFmt` procedure is a variation of `ShowMessage`, which has the same parameters as the `Format` function. It corresponds to calling `Format` inside a call to `ShowMessage`.
- The `MessageBox` method of the `Application` object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the `MessageBox` function of the Windows API, which passes as a main window parameter the handle of the `Application` object. This handle is required to make the message box behave like a modal window.
- The `InputBox` function asks the user to input a string. You provide the caption, the query, and a default string. The `InputQuery` function asks the user to input a string, too. The only difference between this and the `InputBox` function is in the syntax. The `InputQuery` function has a Boolean return value that indicates whether the user has clicked OK or Cancel.

To demonstrate some of the message boxes available in Delphi, I've written another sample program, with a similar approach to the preceding `CommDlgTest` example. In the `MBParade` example, you have a high number of choices (radio buttons, check boxes, edit boxes, and spin edit controls) to set before you click one of the buttons that displays a message box. The similar `QMbParade` example is missing only the Help button, which is not available in the CLX message boxes.

## About Boxes and Splash Screens

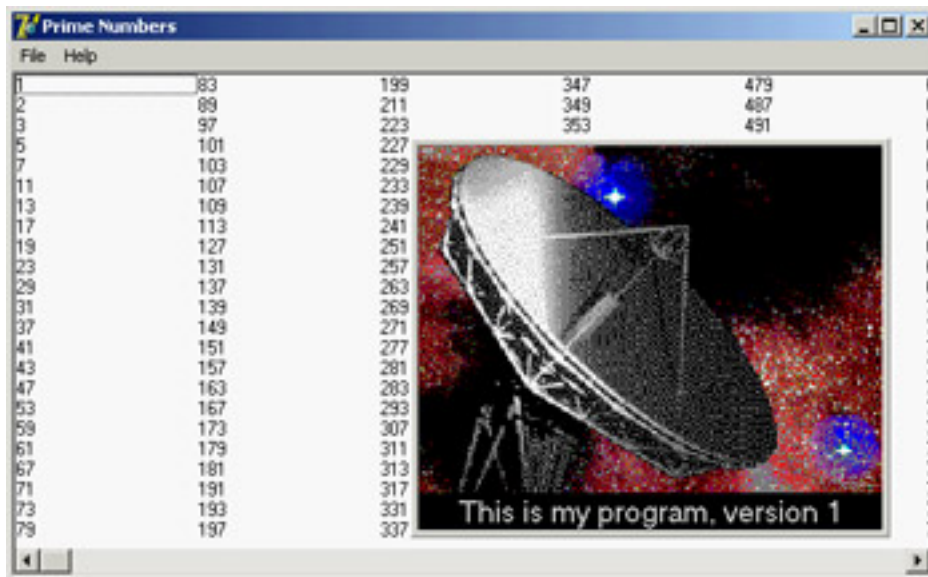
Applications usually have an About box in which you can display information such as the version of the product, a copyright notice, and so on. The simplest way to build an About box is to use the `MessageDlg` function. With this method, you can show only a limited amount of text and no special graphics.

Therefore, the usual method for creating an About box is to use a dialog box, such as the one generated with one of the Delphi default templates. In this About box, you might want to add some code to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

### Building a Splash Screen

Another typical technique displays an initial screen before the application's main form is shown. Doing so makes the application seem more responsive, because you show something to the user while the program is loading, and it also makes a nice visual effect. Sometimes this same window is displayed as the application's About box. For an example in which a splash screen is particularly useful, I've built a program displaying a list box filled with prime numbers.

The prime numbers are computed on program startup, with a `for` loop running from 1 to 30,000; the numbers are displayed as soon as the form becomes visible. Because I've used (on purpose) a slow function to compute prime numbers, this initialization code takes quite some time. The numbers are added to a list box that covers the full client area of the form and allows multiple columns to be displayed, as you can see in [Figure 7.14](#).



**Figure 7.14:** The main form of the Splash example, with the splash screen (this is the Splash2 version)

There are three versions of the Splash program (plus the three corresponding CLX versions). As you can see by running the Splash0 example, the problem with this program is that the initial operation, which takes place in the `FormCreate` method, takes a lot of time. When you start the program, it takes several seconds to display the main form. If your computer is very fast or very slow, you can change the upper limit of the `for` loop in the `FormCreate` method to make the program faster or slower.

This program has a simple dialog box with an image component, a caption, and a bitmap button, all placed inside a panel taking up the whole surface of the About box. This form is displayed when you select the Help → About menu item. But you really want to display this About box while the program starts. You can see this effect by running the Splash1 and Splash2 examples, which show a splash screen using two different techniques.

First, I've added a method to the `TAboutBox` class. This method, called `MakeSplash`, changes some properties of the form to make it suitable for a splash form. Basically, it removes the border and caption, hides the OK button, makes the

border of the panel thick (to replace the border of the form), and then shows the form, repainting it immediately:

```
procedure TAboutBox.MakeSplash;  
begin  
  BorderStyle := bsNone;  
  BitBtn1.Visible := False;  
  Panell1.BorderWidth := 3;  
  Show;  
  Update;  
end;
```

This method is called after creating the form in the project file of the Splash1 example. This code is executed before creating the other forms (in this case only the main form), and the splash screen is then removed before running the application. These operations take place within a `try/finally` block. Here is the source code of the main block of the project file for the Splash2 example:

```
var  
  SplashAbout: TAboutBox;  
  
begin  
  Application.Initialize;  
  
  // create and show the splash form  
  SplashAbout := TAboutBox.Create (Application);  
  try  
    SplashAbout.MakeSplash;  
    // standard code...  
    Application.CreateForm(TForm1, Form1);  
    // get rid of the splash form  
    SplashAbout.Close;  
  finally  
    SplashAbout.Free;  
  end;  
  
  Application.Run;  
end.
```

This approach makes sense only if your application's main form takes a while to create, to execute its startup code (as in this case), or to open database tables. Notice that the splash screen is the first form created, but because the program doesn't use the `Application` object's `CreateForm` method, it doesn't become the main form of the application. In this case, closing the splash screen would terminate the program!

An alternative approach is to keep the splash form on the screen a little longer and use a timer to get rid of it. I've implemented this technique in the Splash2 example. This example also uses a different approach for creating the splash form: Instead of creating the splash form in the project source code, it creates the form at the very beginning of the `FormCreate` method of the main form.

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
  I: Integer;  
  SplashAbout: TAboutBox;  
begin  
  // create and show the splash form  
  SplashAbout := TAboutBox.Create (Application);  
  SplashAbout.MakeSplash;  
  // slow code (omitted)...  
  // get rid of the splash form, after a while  
  SplashAbout.Timer1.Enabled := True;  
end;
```

The timer is enabled just before terminating the method. After its interval has elapsed (in the example, 3 seconds) the `OnTimer` event is activated, and the splash form handles it by closing and destroying itself, calling `Close` and then `Release`.

**Note** The *Release* method of a form is similar to the *Free* method of objects, but the destruction of the form is delayed until all event handlers have completed execution. Using *Free* inside a form might cause an access violation, because the internal code that fired the event handler might refer again to the form object.

There is one more thing to fix. The main form will be displayed later and in front of the splash form, unless you make it a top-most form. For this reason, I've added one line to the `MakeSplash` method of the About box in the `Splash2` example:

```
FormStyle := fsStayOnTop;
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

In this chapter, we've explored some important form properties. Now you know how to handle the size and position of a form, how to resize it, and how to get mouse input and paint over it. You know more about dialog boxes, modal forms, predefined dialogs, splash screens, and many other techniques, including the funny effect of alpha blending. Understanding the details of working with forms is critical to proper use of Delphi, particularly for building complex applications (unless, of course, you're building services or web applications with no user interface).

In [Chapter 8](#), we'll continue by exploring the overall structure of a Delphi application, with coverage of the role of two global objects: `Application` and `Screen`. I'll also discuss MDI development as you learn about more advanced features of forms, such as visual form inheritance. In addition, I'll discuss frames, which are visual component containers similar to forms.

In this chapter, I've also provided a short introduction to direct painting and to the use of the `TCanvas` class. More about graphics in Delphi forms can also be found in the bonus chapter "Graphics in Delphi", discussed in [Appendix C](#).

## Part II: Delphi Object-Oriented Architectures

### Chapter List

[Chapter 8](#): The Architecture of Delphi Applications

[Chapter 9](#): Writing Delphi Components

[Chapter 10](#): Libraries and Packages

[Chapter 11](#): Modeling and OOP Programming (with ModelMaker)

[Chapter 12](#): From COM to COM+

## Chapter 8: The Architecture of Delphi Applications

### Overview

Although together we've built Delphi applications since the beginning of the book, we haven't focused on the structure and the architecture of an application built with Delphi's class libraries. For example, I haven't included much coverage about the global `Application` object, techniques for keeping tracks of forms you've created, the flow of messages in the system, and other such elements.

In [Chapter 7](#), "Working with Forms," you saw how to create applications with multiple forms and dialog boxes. However, I haven't discussed how these forms can be related one to the other, how you can share similar features of forms, and how you can operate on multiple similar forms in a consistent way.

My ambitious goal in this chapter is to discuss all these topics. The chapter covers both basic and advanced techniques, including visual form inheritance, the use of frames, and MDI development, as well as the use of interfaces for building complex hierarchies of form classes.



## The Application Object

I've mentioned the `Application` global object on multiple occasions, but because this chapter focuses on the structure of Delphi applications, it is time to delve into the details of this global object and its corresponding class. `Application` is a global object of the `TApplication` class, defined in the Forms unit and created in the Controls unit. The `TApplication` class is a component, but you cannot use it at design time. Some of its properties can be directly set in the Application page of the Project Options dialog box; others must be assigned in code.

To handle its events, Delphi includes a handy `ApplicationEvents` component. Besides allowing you to assign handlers at design time, the advantage of this component is that it allows for multiple handlers. If you simply place an instance of the `ApplicationEvents` component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple `ApplicationEvents` components can chain the handlers.

Some of these application-wide events, including `OnActivate`, `OnDeactivate`, `OnMinimize`, and `OnRestore`, allow you to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as in `OnActionExecute`, `OnActionUpdate`, `OnHelp`, `OnHint`, `OnShortCut`, and `OnShowHint`. Finally, there is the `OnException` global exception handler we used in [Chapter 2](#) ("The Delphi Programming Language"), the `OnIdle` event used for background computing, and the `OnMessage` event, which fires when a message is posted to any of the windows or windowed controls of the application.

Although its class inherits directly from `TComponent`, the `Application` object has a window associated with it. The application window is hidden from sight but appears on the Taskbar. This is why Delphi names the window `Form1` and the corresponding Taskbar icon `Project1`.

The window related to the `Application` object—the application window—serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this invisible owner window, for example, is fundamental when the application is activated. When your program's windows are behind other programs' windows, clicking one window in your application will bring all of that application's windows to the front. In other words, the unseen application window is used to connect the application's various forms. (The application window is not *hidden*, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.)

**Tip** In Windows, the Minimize and Maximize operations are associated by default with system sounds and a visual animated effect. Applications built with Delphi produce the sound and display the visual effect by default.

When you create a new, blank application, Delphi generates code for the project file that includes the following:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

As you can see in this standard code, the `Application` object can create forms, setting the first one as the `MainForm` (one of the `Application` properties) and closing the entire application when this main form is destroyed. The program execution is enclosed in the `Run` method, which embeds the system loop to process system messages. This loop continues until the application's main window (the first window you created) is closed.

**Tip** As we saw in the splash screen example in [Chapter 7](#), the main form is not necessarily the first form you create, but the first one you create by calling `Application.CreateForm`.

The Windows message loop embedded in the `Run` method delivers the system messages to the proper application windows. A message loop is required by any Windows application, but you don't need to write one in Delphi because the `Application` object provides a default loop.

In addition to performing this main role, the `Application` object manages a few other interesting areas:

- Hints (discussed at the end of [Chapter 5](#), "Visual Controls")
- The help system, which includes the ability to define the type of help viewer (a topic not covered in detail in this book)
- Application activation, minimization, and restoration
- A global exceptions handler, as discussed in the ErrorLog example of [Chapter 2](#).
- General application information, including the `MainForm`, executable filename and path (`ExeName`), `Icon`, and `Title` displayed in the Windows Taskbar and when you scan the running applications with the Alt+Tab keys

**Tip** To avoid a discrepancy between the two titles, you can change the application's title at design time. In case the caption of the main form changes at runtime, you can copy it to the title of the application with this code:

```
Application.Title := Form1.Caption.
```

In most applications, you don't care about the application window, apart from setting its `Title` and icon and handling some of its events. However, you can perform some other simple operations. Setting the `ShowMainForm` property to `False` in the project source code indicates that the main form should not be displayed at startup. Inside a program, you can use the `Application` object's `MainForm` property to access the main form.

## Displaying the Application Window

There is no better proof that a window exists for the `Application` object than to display it, as in the `ShowApp` example. You don't need to show it—you just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. You can perform these operations using Windows API functions on the window indicated by the `Application` object's `Handle` property:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    OldStyle: Integer;
begin
    // add border and caption to the app window
    OldStyle := GetWindowLong (Application.Handle, gwl_Style);
    SetWindowLong (Application.Handle, gwl_Style,
        OldStyle or ws_ThickFrame or ws_Caption);
    // set the size of the app window
    SetWindowPos (Application.Handle, 0, 0, 0, 200, 100,
        swp_NoMove or swp_NoZOrder);
end;
```

The `GetWindowLong` and `SetWindowLong` API functions access the system information related to the window. In this case, you use the `gwl_Style` parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. This code gets the current styles and adds (using an `or` statement) a standard border and a caption to the form.

Of course, you generally won't need to implement something like this in your programs. But knowing the application object has a window connected to it is an important aspect of understanding the default structure of Delphi applications and being able to modify it when needed.

## Activating Applications and Forms

To show how the activation of forms and applications works, I've written a self-explanatory example called `ActivApp`. This example has two forms. Each form has a `Label` component (`LabelForm`) used to display the form's status. The program uses text and color to indicate this status information, as the handlers of the first form's `OnActivate` and `OnDeactivate` events demonstrate:

```

procedure TForm1.FormActivate(Sender: TObject);
begin
    LabelForm.Caption := 'Form2 Active';
    LabelForm.Color := clRed;
end;

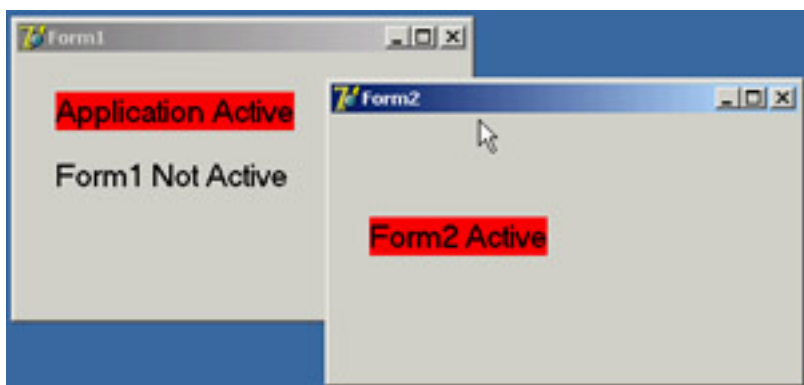
procedure TForm1.FormDeactivate(Sender: TObject);
begin
    LabelForm.Caption := 'Form2 Not Active';
    LabelForm.Color := clBtnFace;
end;

```

The second form has a similar label and similar code.

The main form also displays the status of the entire application. It uses an `ApplicationEvents` component to handle the `Application` object's `OnActivate` and `OnDeactivate` events. These two event handlers are similar to the two listed previously; the only difference is that they modify the text and color of a second label on the form and that one of them makes a beep.

If you run this program, you'll see whether this application is active and, if so, which of its forms is active. By looking at the output (see [Figure 8.1](#)) and listening for the beep, you can understand how Delphi triggers each of the activation events. Run the program and play with it for a while to understand how it works. Later, we'll get back to other events related to the activation of forms.



**Figure 8.1:** The `ActivApp` example shows whether the application is active and which of the application's forms is active.

## Tracking Forms with the `Screen` Object

We have already explored some of the properties and events of the `Application` object. Other interesting global information about an application is available through the `Screen` object, whose base class is `TScreen`. This object holds information about the system display (the screen size and the screen fonts) and also about the current set of forms in a running application. For example, you can display the screen size and the list of fonts by writing:

```

Label1.Caption := IntToStr (Screen.Width) + 'x' + IntToStr (Screen.Height);
ListBox1.Items := Screen.Fonts;

```

`TScreen` also reports the number and resolution of monitors in a multimonitor system. Right now, however, I will focus on the list of forms held by the `Screen` object's `Forms` property, the top-most form indicated by the `ActiveForm` property, and the related `OnActiveFormChange` event. Note that the forms the `Screen` object references are the forms of the application and not those of the system.

These features are demonstrated by the `Screen` example, which maintains a list of the current forms in a list box. This list must be updated each time a new form is created, an existing form is destroyed, or the program's active form changes. To see how this process works, you can create secondary forms by clicking the button labeled `New`:

```

procedure TMainForm.NewButtonClick(Sender: TObject);
var
    NewForm: TSecondForm;
begin
    // create a new form, set its caption, and run it
    NewForm := TSecondForm.Create (Self);
    Inc (nForms);
    NewForm.Caption := 'Second ' + IntToStr (nForms);
    NewForm.Show;
end;

```

Note that you need to disable the automatic creation of the secondary form by using the Forms page of the Project Options dialog box. One of the key portions of the program is the form's `OnCreate` event handler, which fills the list the first time and then connects a handler to the `OnActiveFormChange` event:

```

procedure TMainForm.FormCreate(Sender: TObject);
begin
    FillFormsList (Self);
    // set the secondary form's counter to 0
    nForms := 0;
    // set an event handler on the screen object
    Screen.OnActiveFormChange := FillFormsList;
end;

```

The code used to fill the Forms list box is inside a second procedure, `FillFormsList`, which is also installed as an event handler for the `Screen` object's `OnActiveFormChange` event:

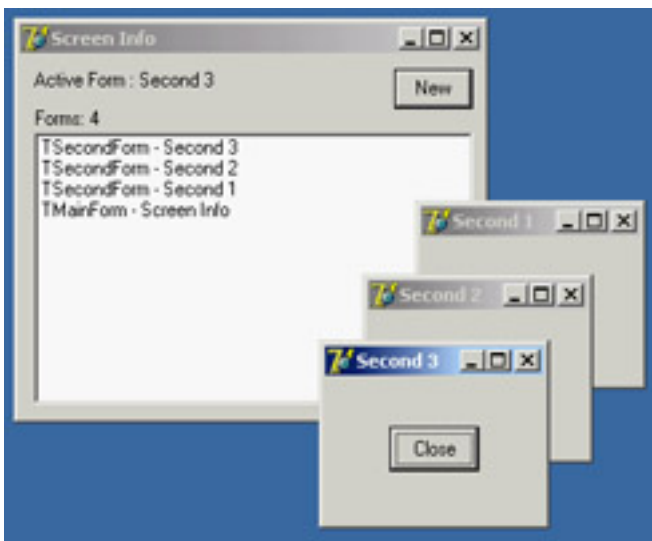
```

procedure TMainForm.FillFormsList (Sender: TObject);
var
    I: Integer;
begin
    // skip code in destruction phase
    if Assigned (FormsListBox) then
        begin
            FormsLabel.Caption := 'Forms: ' + IntToStr (Screen.FormCount);
            FormsListBox.Clear;
            // write class name and form title to the list box
            for I := 0 to Screen.FormCount - 1 do
                FormsListBox.Items.Add (Screen.Forms[I].ClassName + ' - ' +
                    Screen.Forms[I].Caption);
            ActiveLabel.Caption := 'Active Form : ' + Screen.ActiveForm.Caption;
        end;
    end;

```

**Warning** It is very important not to execute this code while the main form is being destroyed. As an alternative to testing whether the list box is set to `nil`, you could test the form's `ComponentState` for the `csDestroying` flag. Another approach would be to remove the `OnActiveFormChange` event handler before exiting the application; that is, handle the main form's `OnClose` event and assign `nil` to `Screen.OnActiveFormChange`.

The `FillFormsList` method fills the list box and sets a value for the two labels above it to show the number of forms and the name of the active form. When you click the New button, the program creates an instance of the secondary form, gives it a new title, and displays it. The Forms list box is updated automatically because of the handler installed for the `OnActiveFormChange` event. [Figure 8.2](#) shows the output of this program when some secondary windows have been created.



**Figure 8.2:** The output of the Screen example with some secondary forms

Each secondary form has a Close button you can click to remove it. The program handles the `OnClose` event, setting the `Action` parameter to `caFree`, so that the form is destroyed when it is closed. This code closes the form, but it doesn't update the list of the windows properly. The system moves the focus to another window first, firing the event that updates the list, and destroys the old form only after this operation.

The first idea I had to update the windows list properly was to introduce a delay, posting a user-defined Windows message. Because the posted message is queued and not handled immediately, if you send it at the last possible moment of the secondary form's life, the main form will receive it when the other form is destroyed. The trick is to post the message in the secondary form's `OnDestroy` event handler. To accomplish this, you need to refer to the `MainForm` object by adding a `uses` statement in the implementation portion of this unit. I've posted a `wm_User` message, which is handled by a specific `message` method of the main form, as shown here:

```
public
  procedure ChildClosed (var Message: TMessage); message wm_User;

procedure TMainForm.ChildClosed (var Message: TMessage);
begin
  FillFormsList (Self);
end;
```

The problem is that if you close the main window before closing the secondary forms, the main form exists, but its code can no longer be executed. To avoid another system error (an Access Violation Fault), you need to post the message only if the main form is not closing. But how do you determine whether the form is closing? One way is to add a flag to the `TMainForm` class and change its value when the main form is closing, so that you can test the flag from the code of the secondary window.

This is a good solution—so good that the VCL already provides similar functionality with the `ComponentState` property and its `csDestroying` flag, as mentioned earlier. Therefore, you can write the following code:

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

With this code, the list box always lists all the forms in the application.

After giving this approach some thought, however, I found an alternative and much more Delphi-oriented solution. The trick is to consider that every time a component is destroyed, it tells its owner about the event by calling the `Notification` method defined in the `TComponent` class. Because the secondary forms are owned by the main form, as specified in the `NewButtonClick` method's code, you can override this method and simplify the code (see the

Screen2 folder for this version's code):

```
procedure TMainForm.Notification(AComponent: TComponent;  
    Operation: TOperation);  
begin  
    inherited Notification(AComponent, Operation);  
    if (Operation = opRemove) and Showing and (AComponent is TForm) then  
        FillFormsList;  
end;
```

**Note** If the secondary forms were not owned by the main form, you could have used the *FreeNotification* method to get the secondary forms to notify the main form when they are destroyed. *FreeNotification* receives as parameter the component to notify when the current component is destroyed. The effect is a call to the *Notification* method that comes from a component other than the owned components. *FreeNotification* is generally used by component writers to safely connect components on different forms or data modules.

The last feature I've added to both versions of the program is simple: When you click an item in the list box, the corresponding form is activated using the `BringToFront` method. Nice—well, almost nice. If you click the list box when the main form is not active, the main form is activated first, and the list box is rearranged; so, you might end up selecting a different form than you were expecting. If you experiment with the program, you'll soon realize what I mean. This minor glitch in the program is an example of the risks you face when you dynamically update information and let the user work on it at the same time.

Team LiB

◀ PREVIOUS

NEXT ▶

## From Events to Threads

To understand how Windows applications work internally, let's spend a minute discussing how multitasking is supported in this environment. You also need to understand the role of timers (and the `Timer` component) and of background (or *idle*) computing, as well as the `ProcessMessages` method of the `Application` global object.

In short, we need to delve deeper into the event-driven structure of Windows and its multitasking support. Because this is a book about *Delphi* programming, I won't discuss this topic in detail, but I will provide an overview for readers who have limited experience with Windows API programming.

### Event-Driven Programming

The basic idea behind event-driven programming is that specific events determine the control flow of the application. A program spends most of its time waiting for these events and provides code to respond to them. For example, when a user clicks one of the mouse buttons, an event occurs. A message describing this event is sent to the window currently under the mouse cursor. The program code that responds to events for that window receives the event, processes it, and responds accordingly. When the program has finished responding to the event, it returns to a waiting or "idle" state.

As this explanation shows, events are serialized; each event is handled only after the previous one is completed. When an application is executing event-handling code (that is, when it is not waiting for an event), other events for that application have to wait in a message queue reserved for that application (unless the application uses multiple threads). When an application has responded to a message and returned to a waiting state, it becomes the last in the list of programs waiting to handle additional messages. In every version of Win32 (9x, NT, Me, and 2000), after a fixed amount of time has elapsed, the system interrupts the current application and immediately gives control to the next program in the list. The first program is resumed only after each application has had a turn. This process is called *preemptive multitasking*.

So, an application performing a time-consuming operation in an event handler doesn't prevent the system from working properly (because other processes have their time-slice of the CPU), but the application generally is unable even to repaint its own windows properly—with a very nasty effect. If you've never experienced this problem, try it for yourself: Write a time-consuming loop that executes when a button is clicked, and try to move the form or move another window on top of it. The effect is really annoying. Now try adding the call `Application.ProcessMessages` within the loop; you'll see that the operation becomes much slower, but the form will be refreshed immediately.

As an example of the use of `Application.ProcessMessages` within a time-consuming loop (and the lack of this call), you can refer to the `BackTask` example. Here is the code using this approach (ignore the naïve technique for computing the sum of a given set of prime numbers):

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    I, Tot: Integer;  
begin  
    Tot := 0;  
    for I := 1 to Max do  
        begin  
            if IsPrime (I) then  
                Tot := Tot + I;  
            ProgressBar1.Position := I * 100 div Max;  
            Application.ProcessMessages;  
        end;  
    ShowMessage (IntToStr (Tot));  
end;
```



**Tip** There is a second alternative to calling *ProcessMessages*: the *HandleMessage* function. There are two differences: *HandleMessage* processes at most one message each time it is called, whereas *ProcessMessages* keeps processing messages in the queue; and *HandleMessage* also activates idle time processing, such as action update calls.

If an application has responded to its events and is waiting for its turn to process messages, it has no chance to regain control until it receives another message (unless it uses multithreading). This is a reason to use a *timer*: a system component that will send a message to your application whenever a specified time interval elapses. Using a timer is the only way to make an application perform operations automatically from time to time, even when the user is absent or not using the program (so that it is not processing any events).

One final note—when you think about events, remember that input events (generated using the mouse or the keyboard) account for only a small percentage of the total message flow in a Windows application. Most of the messages are the system's internal messages or messages exchanged between different controls and windows. Even a familiar input operation such as clicking a mouse button can result in a huge number of messages, most of which are internal Windows messages. You can test this yourself by using the WinSight utility included in Delphi. In WinSight, choose to view the Message Trace, and select the messages for all the windows. Click Start, and then perform some normal operations with the mouse. You'll see hundreds of messages in a few seconds.

## Windows Message Delivery

Before looking at some real examples, let's consider another key element of message handling. Windows has two different ways to send a message to a window:

**PostMessage API Function** Places a message in the application's message queue. The message will be handled only when the application has a chance to access its message queue (that is, when it receives control from the system), and only after earlier messages have been processed. This is an asynchronous call, because you do not know when the message will be received.

**SendMessage API function** Executes message-handler code immediately. *SendMessage* bypasses the application's message queue and sends the message directly to a target window or control. This is a synchronous call. This function even has a return value, which is passed back by the message-handling code. Calling *SendMessage* is no different than directly calling another method or function of the program.

The difference between these two ways of sending messages is similar to that between mailing a letter, which will reach its destination sooner or later, and sending a fax, which goes immediately to the recipient. Although you will rarely need to use these low-level functions in Delphi, this description should help you determine which one to use if you do need to write this type of code.

## Background Processing and Multitasking

Suppose you need to implement a time-consuming algorithm. If you write the algorithm as a response to an event, your application will be stopped completely during the time it takes to process that algorithm. To let the user know that something is being processed, you can display the hourglass cursor or show a progress bar, but this is not a user-friendly solution. Win32 allows other programs to continue their execution, but the program in question will appear to be frozen; it won't even update its own user interface if a repaint is requested. While the algorithm is executing, the application won't be able to receive and process any other messages, including paint messages.

The simplest solution to this problem is to call the *ProcessMessages* and *HandleMessage* methods, discussed earlier. The problem with this approach, however, is that the user might click the button again or re-press the keystrokes that started the algorithm. To fix this possibility, you can disable the buttons and commands you don't want the user to select, and you can display the hourglass cursor (which technically doesn't prevent a mouse-click event, but does suggest that the user should wait before doing any other operation).

For some low-priority background processing, you can also split the algorithm into smaller pieces and execute each of



them in turn, letting the application fully respond to pending messages in between processing the pieces. You can use a timer to let the system notify you once a time interval has elapsed. Although you can use timers to implement some form of background computing, this is far from a good solution. A better technique would be to execute each step of the program when the `Application` object receives the `OnIdle` event.

The difference between calling `ProcessMessages` and using the `OnIdle` event is that calling `ProcessMessages` gives your code more processing time. Calling `ProcessMessages` lets the program perform other operations while a long operation is being executed; using the `OnIdle` event lets your application perform background tasks when it doesn't have pending requests from the user.

## Delphi Multithreading

When you need to perform background operations or any processing not strictly related to the user interface, you can follow the technically most correct approach: spawn a separate thread of execution within the process. Multithreading programming might seem like an obscure topic, but it really isn't that complex, even if you must consider it with care. It is worth knowing at least the basics of multithreading, because in the world of sockets and Internet programming, there is little you can do without threads.

Delphi's RTL library provides a `TThread` class that will let you create and control threads. You will never use the `TThread` class directly, because it is an abstract class—a class with a virtual abstract method. To use threads, you always subclass `TThread` and use the features of this base class.

The `TThread` class has a constructor with a single parameter (`CreateSuspended`) that lets you choose whether to start the thread immediately or suspend it until later. If the thread object starts automatically, or when it is resumed, it will run its `Execute` method until it is done. The class provides a protected interface, which includes the two key methods for your thread subclasses:

```
procedure Execute; virtual; abstract;  
procedure Synchronize(Method: TThreadMethod);
```

The `Execute` method, declared as a virtual abstract procedure, must be redefined by each thread class. It contains the thread's main code—the code you would typically place in a *thread function* when using the system functions.

The `Synchronize` method is used to avoid concurrent access to VCL components. The VCL code runs inside the program's main thread, and you need to synchronize access to VCL to avoid re-entry problems (errors from re-entering a function before a previous call is completed) and concurrent access to shared resources. The only parameter of `Synchronize` is a method that accepts no parameters, typically a method of the same thread class. Because you cannot pass parameters to this method, it is common to save some values within the data of the thread object in the `Execute` method and use those values in the *synchronized* methods.

**Note** Delphi 7 includes two new versions of `Synchronize` that allow you to synchronize a method with the main thread without calling it from the thread object. Both the new overloaded `Synchronize` and `StaticSynchronize` are class methods of `TThread` and require a thread as parameter.

Another way to avoid conflicts is to use the synchronization techniques offered by the operating system. The `SyncObjs` unit defines a few VCL classes for some of these low-level synchronization objects, such as events (with the `TEvent` class and the `TSingleEvent` class) and critical sections (with the `TCriticalSection` class). (Synchronization events should not be confused with Delphi events, as the two concepts are unrelated.)

## An Example of Threading

For an example of a thread, you can refer again to the `BackTask` example. This example spawns a secondary thread for computing the sum of the prime numbers. The thread class has the typical `Execute` method, an initial value passed in a public property (`Max`), and two internal values (`FTotal` and `FPosition`) used to synchronize the output in the `ShowTotal` and `UpdateProgress` methods. The following is the complete class declaration for the custom thread object:

```

type
  TPrimeAdder = class(TThread)
  private
    FMax, FTotal, FPosition: Integer;
  protected
    procedure Execute; override;
    procedure ShowTotal;
    procedure UpdateProgress;
  public
    property Max: Integer read FMax write FMax;
end;

```

The `Execute` method is very similar to the code used for the buttons in the `BackTask` example listed earlier. The only difference is in the final call to `Synchronize`, as you can see in the following two fragments:

```

procedure TPrimeAdder.Execute;
var
  I, Tot: Integer;
begin
  Tot := 0;
  for I := 1 to FMax do
  begin
    if IsPrime (I) then
      Tot := Tot + I;
    if I mod (fMax div 100) = 0 then
      begin
        FPosition := I * 100 div fMax;
        Synchronize(UpdateProgress);
      end;
    FTotal := Tot;
    Synchronize(ShowTotal);
  end;

procedure TPrimeAdder.ShowTotal;
begin
  ShowMessage ('Thread: ' + IntToStr (FTotal));
end;

procedure TPrimeAdder.UpdateProgress;
begin
  Form1.ProgressBar1.Position := fPosition;
end;

```

The thread object is created when a button is clicked and is automatically destroyed as soon as its `Execute` method is completed:

```

procedure TForm1.Button3Click(Sender: TObject);
var
  AdderThread: TPrimeAdder;
begin
  AdderThread := TPrimeAdder.Create (True);
  AdderThread.Max := Max;
  AdderThread.FreeOnTerminate := True;
  AdderThread.Resume;
end;

```

Instead of setting the maximum number using a property, it would have been better to pass this value as an extra parameter of a custom constructor; I've avoided doing so only to remain focused on the example of using a thread.

You'll see more examples of threads in other chapters—particularly [Chapter 19](#), "Internet Programming: Sockets and Indy," which discusses the use of sockets.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Checking for a Previous Instance of an Application

One form of multitasking is the execution of two or more instances of the same application. Any application can generally be executed by a user in more than one instance, and it needs to be able to check for a previous instance already running, in order to disable this default behavior and allow for one instance at most. This section demonstrates several ways of implementing such a check, allowing me to discuss some interesting Windows programming techniques.

### Looking for a Copy of the Main Window

To find a copy of the main window of a previous instance, use the `FindWindow` API function and pass it the name of the window class (the name used to register the form's window type, or `WNDCLASS`, in the system) and the caption of the window for which you are looking. In a Delphi application, the name of the `WNDCLASS` window class is the same as the Object Pascal name for the form's class (for example, `TForm1`). The result of the `FindWindow` function is either a handle to the window or zero (if no matching window was found).

The main code of your Delphi application should be written so that it will execute only if the `FindWindow` result is zero:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', nil);
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
  else
    SetForegroundWindow (Hwnd)
end.
```

To activate the window of the application's previous instance, you can use the `SetForegroundWindow` function, which works for windows owned by other processes. This call produces its effect only if the window passed as parameter hasn't been minimized. When the main form of a Delphi application is minimized, it is hidden, and for this reason the activation code has no effect.

Unfortunately, if you run a program that uses the `FindWindow` call just shown from within the Delphi IDE, a window with that caption and class may already exist: the design-time form. Thus, the program won't start even once. However, it will run if you close the form and its corresponding source code file (closing only the form simply hides the window), or if you close the project and run the program from the Windows Explorer. Consider also that having a form called *Form1* is quite likely to not work as expected, as many Delphi applications might have a form with the same name. This will be fixed in the following versions of the code.

### Using a Mutex

A completely different approach is to use a *mutex*, or mutual exclusion object. This is a typical Win32 approach, commonly used for synchronizing threads. Here you will use a mutex to synchronize two different applications or, to be more precise, two instances of the same application.

Once an application has created a mutex with a given name, it can test whether this object is already owned by another application by calling the `WaitForSingleObject` Windows API function. If the mutex has no owner, the application calling this function becomes the owner. If the mutex is already owned, the application waits until the time-out (the function's second parameter) elapses. It then returns an error code.

To implement this technique, you can use the following project source code:

```
var
  hMutex: THandle;
begin
  HMutex := CreateMutex (nil, False, 'OneCopyMutex');
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

If you run this example twice, you'll see that it creates a new, temporary copy of the application (the icon appears in the Taskbar) and then destroys it when the time-out elapses. This approach is certainly more robust than the previous one, but it lacks a feature: How do you enable the existing instance of the application? You still need to find its form, but you can use a better technique.

## Searching the Window List

When you want to search for a specific main window in the system, you can use the `EnumWindows` API functions. Enumeration functions are peculiar in Windows, because they usually require another function as a parameter. These enumeration functions require a pointer to a function (often described as a *callback* function) as parameter. This function is applied to each element of the list (in this case, the list of main windows), until the list ends or the function returns `False`. Here is the enumeration function from the `OneCopy` example:

```
function EnumWndProc (hwnd: THandle; Param: Cardinal): Bool; stdcall;
var
  ClassName, WinModuleName: string;
  WinInstance: THandle;
begin
  Result := True;
  SetLength (ClassName, 100);
  GetClassName (hwnd, PChar (ClassName), Length (ClassName));
  ClassName := PChar (ClassName);
  if ClassName = TForm1.ClassName then
  begin
    // get the module name of the target window
    SetLength (WinModuleName, 200);
    WinInstance := GetWindowLong (hwnd, GWL_HINSTANCE);
    GetModuleFileName (WinInstance,
      PChar (WinModuleName), Length (WinModuleName));
    WinModuleName := PChar(WinModuleName); // adjust length
    // compare module names
    if WinModuleName = ModuleName then
    begin
      FoundWnd := Hwnd;
      Result := False; // stop enumeration
    end;
  end;
end;
```

This function, which is called for each nonchild window of the system, checks the name of each window's class, looking for the name of the `TForm1` class. When it finds a window with this string in its class name, it uses `GetModuleFilename` to extract the name of the executable file of the application that owns the matching form. If the module name matches that of the current program (which was extracted previously with similar code), you can be sure

that you have found a previous instance of the same program. Here is how you can call the enumerated function:

```
var
  FoundWnd: THandle;
  ModuleName: string;
begin
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
    ...
  else
    begin
      // get the current module name
      SetLength (ModuleName, 200);
      GetModuleFileName (HInstance, PChar (ModuleName), Length (ModuleName));
      ModuleName := PChar (ModuleName); // adjust length
      // find window of previous instance
      EnumWindows (@EnumWndProc, 0);
```

## Handling User-Defined Window Messages

I've mentioned that the `SetForegroundWindow` call doesn't work if the main form of the program has been minimized. Now you can solve this problem. You can ask the form of another application—the previous instance of the same program, in this case—to restore its main form by sending it a user-defined window message. You can then test whether the form is minimized and post a new user-defined message to the old window. Here is the code; in the `OneCopy` program, it follows the last fragment shown in the preceding section:

```
if FoundWnd <> 0 then
begin
  // show the window, eventually
  if not IsWindowVisible (FoundWnd) then
    PostMessage (FoundWnd, wm_App, 0, 0);
  SetForegroundWindow (FoundWnd);
end;
```

The `PostMessage` API function sends a message to the message queue of the application that owns the destination window. In the form's code, you can add a special function to handle this message:

```
public
  procedure WMAApp (var msg: TMessage); message wm_App;

procedure TForm1.WMAApp (var msg: TMessage);
begin
  Application.Restore;
end;
```

**Note** The program uses the `wm_App` message rather than the `wm_User` message; some system windows use `wm_User`, so there is no guarantee that other applications or the system won't send this message. That's why Microsoft introduced `wm_App` for messages that are restricted to the application's interpretation.

## Creating MDI Applications

MDI (Multiple Document Interface) is a common approach for an application's structure. An MDI application is made up of several forms that appear inside a single main form. If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application. But with your favorite word processor, you can probably open several different documents, each in its own child window, because the word processor is an MDI application. All these document windows are usually held by a *frame*, or *application*, window.

**Note** Increasingly, Microsoft is departing from the MDI model stressed in Windows 3 days. Even recent versions of Office tend to use a specific main window for every document: the classic SDI (Single Document Interface) approach. However, MDI isn't dead and can sometimes be a useful structure, as demonstrated by browsers like Opera and Mozilla.

### MDI in Windows: A Technical Overview

The MDI structure gives programmers several benefits automatically. For example, Windows handles a list of the child windows in one of an MDI application's pull-down menus, and specific Delphi methods activate the corresponding MDI functionality to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows:

- The main window of the application acts as a frame or a container.
- A special window, known as the *MDI client*, covers the whole client area of the frame window. This MDI client is one of the Windows predefined controls, just like an edit box or a list box. The MDI client window lacks any specific user-interface element, but it is visible. You can change the standard system color of the MDI work area (called the Application Background) in the Appearance page of the Display Properties dialog box in Windows.
- There are multiple child windows, of the same kind or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the frame window.

## Frame and Child Windows in Delphi

Delphi makes it easy to develop MDI applications, even without using the MDI Application template available in Delphi (see the Applications page of the File → New → Other dialog box). You only need to build at least two forms, one with the `FormStyle` property set to `fsMDIForm` and the other with the same property set to `fsMDIChild`. Set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at startup, and you need to provide a way to create one or more child windows. You can do so by adding a menu with a New menu item and writing the following code:

```
var
  ChildForm: TChildForm;
begin
  ChildForm := TChildForm.Create (Application);
  ChildForm.Show;
```

Another important feature to add is a Window pull-down menu, which you use as the value of the form's `WindowMenu` property. This pull-down menu will automatically list all the available child windows. (Of course, you can choose any other name for the pull-down menu, but Window is the standard.)

To make this program work properly, you can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  WindowMenu := Window1;
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption := ChildForm.Caption + ' ' + IntToStr (Counter);
  ChildForm.Show;
end;
```

You can also open child windows, minimize or maximize each of them, close them, and use the Window pull-down menu to navigate among them. Now suppose you want to close some of these child windows, to unclutter the client area of your program. Click the Close boxes in some of the child windows, and they are minimized! What is happening? Remember that when you close a window, you generally hide it from view. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, hiding them won't work, because the MDI Window menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi minimizes the MDI child windows when you try to close them. To solve this problem, you need to delete the child windows when they are closed by setting the `Action` reference parameter of the `OnClose` event to `caFree`.

## Building a Complete Window Menu

Your first task is to define a better menu structure for the example. Typically the Window pull-down menu has at least three items: Cascade, Tile, and Arrange Icons. To handle the menu commands, you can use some of the predefined methods of `TForm` that can be used only for MDI frames:

**Cascade Method** Cascades the open MDI child windows. The windows overlap each other. Iconized child windows are also arranged (see `ArrangeIcons`).

**Tile Method** Tiles the open MDI child windows; the child forms are arranged so that they do not overlap. The default behavior is horizontal tiling, although if you have several child windows, they will be arranged in several columns. This default can be changed by using the `TileMode` property (set the value to either `tbHorizontal` or `tbVertical`).



**ArrangeIcons Procedure** Arranges all the iconized child windows. Open forms are not moved.

As a better alternative to calling these methods, you can place an `ActionList` in the form and add to it a series of predefined MDI actions. The related classes are `TWindowArrange`, `TWindowCascade`, `TWindowClose`, `TWindowTileHorizontal`, `TWindowTileVertical`, and `TWindowMinimizeAll`. The connected menu items will perform the corresponding actions and will be disabled if no child window is available. The `MdiDemo` example, which we'll look at next, demonstrates the use of the MDI actions, among other things.

There are some other interesting methods and properties related strictly to MDI in Delphi:

**ActiveMDIChild Property** A run-time, read-only property of the MDI frame form that holds the active child window. The user can change this value by selecting a new child window, or the program can change it using the `Next` and `Previous` procedures, which activate the child window following or preceding the currently active one.

**ClientHandle Property** Holds the Windows handle of the MDI client window, which covers the client area of the main form.

**MDIChildren Property** An array of child windows you can use together with the `MDIChildCount` property to cycle among all the child windows. This property can be useful for finding a particular child window or to operate on each of them.

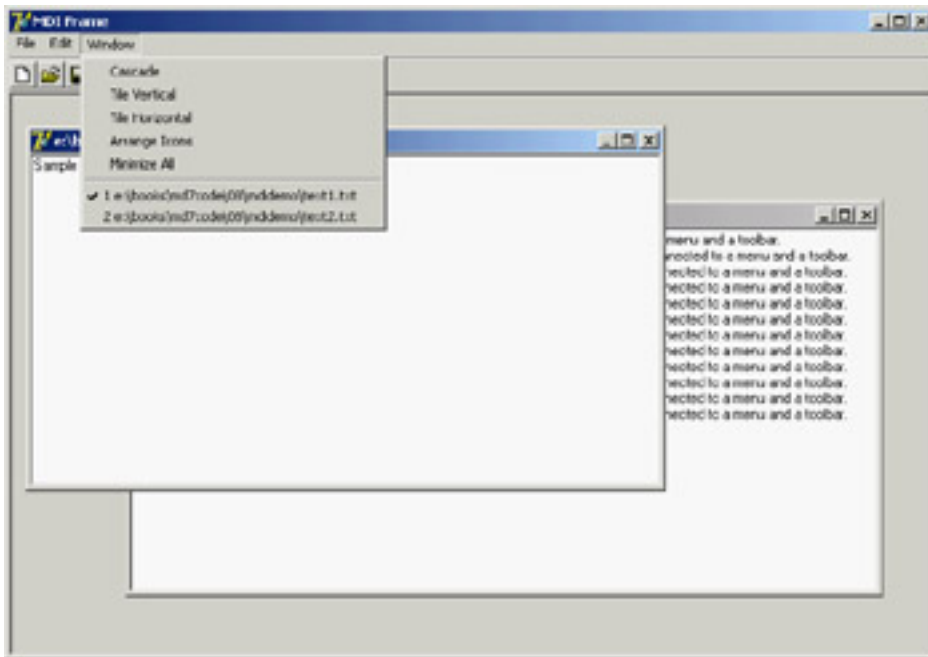
Note that the internal order of the child windows is the reverse order of activation. This means the last child window selected is the active window (the first in the internal list), the second-to-last child window selected is the second, and the first child window selected is the last. This order determines how the windows are arranged on the screen. The first window in the list is above all the others, whereas the last window is below all the others, and probably hidden. You can imagine an axis (the z-axis) coming out of the screen toward you. The active window has a higher value for the z coordinate and, thus, covers other windows. For this reason, the Windows ordering schema is known as the *z-order*.

**Note** The Window menu can be used along with the `ActionManager` and the `ActionMainMenuBar` control hosting the menu, starting with Delphi 7. This control has a specific property, `WindowMenu`, that you have to set to specify the menu that is going to list the MDI child windows.

## The MdiDemo Example

I've built an example to demonstrate most of the features of a simple MDI application. `MdiDemo` is a full-blown MDI text editor, because each child window hosts a `Memo` component and can open and save text files. The child form has a `Modified` property that indicates whether the text of the memo has changed (it is set to `True` in the handler of the memo's `OnChange` event). `Modified` is set to `False` in the `Save` and `Load` custom methods and is checked when the form is closed (prompting the user to save the file).

As I've mentioned, the example's main form is based on an `ActionList` component. The actions are available through some menu items and a toolbar, as shown in [Figure 8.3](#). You can see the details of the `ActionList` in the example's source code; I'll focus on the code of the custom actions.



**Figure 8.3:** The MdiDemo program uses a series of predefined Delphi actions connected to a menu and a toolbar.

One of the simplest actions is the `ActionFont` object, which has both an `OnExecute` handler (which uses a `FontDialog` component) and an `OnUpdate` handler (which disables the action—and hence the associated menu item and toolbar button—when there are no child forms):

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
    if FontDialog1.Execute then
        (ActiveMDIChild as TChildForm).Memo1.Font := FontDialog1.Font;
end;
```

```
procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
    ActionFont.Enabled := MDIChildCount > 0;
end;
```

The action named `New` creates the child form and sets a default filename. The `Open` action calls the `ActionNewExecute` method prior to loading the file:

```
procedure TMainForm.ActionNewExecute(Sender: TObject);
var
    ChildForm: TChildForm;
begin
    Inc (Counter);
    ChildForm := TChildForm.Create (Self);
    ChildForm.Caption :=
        LowerCase (ExtractFilePath (Application.Exename)) + 'text' +
        IntToStr (Counter) + '.txt';
    ChildForm.Show;
end;
```

```
procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
    if OpenDialog1.Execute then
        begin
            ActionNewExecute (Self);
            (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
        end;
end;
```

```
end;
```

The file loading is performed by the form's `Load` method. Likewise, the child form's `Save` method is used by the `Save` and `Save As` actions. Notice the `Save` action's `OnUpdate` handler, which enables the action only if the user has changed the memo's text:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
```

```
begin
```

```
    // suggest the current file name
```

```
    SaveDialog1.FileName := ActiveMDIChild.Caption;
```

```
    if SaveDialog1.Execute then
```

```
    begin
```

```
        // modify the file name and save
```

```
        ActiveMDIChild.Caption := SaveDialog1.FileName;
```

```
        (ActiveMDIChild as TChildForm).Save;
```

```
    end;
```

```
end;
```

```
procedure TMainForm.ActionSaveUpdate(Sender: TObject);
```

```
begin
```

```
    ActionSave.Enabled := (MDIChildCount > 0) and
```

```
    (ActiveMDIChild as TChildForm).Modified;
```

```
end;
```

```
procedure TMainForm.ActionSaveExecute(Sender: TObject);
```

```
begin
```

```
    (ActiveMDIChild as TChildForm).Save;
```

```
end;
```

Team LiB

◀ PREVIOUS

NEXT ▶

## MDI Applications with Different Child Windows

In complex MDI applications, it's common to include child windows of different kinds (that is, based on different child forms). I built an example called `MdiMulti` to highlight some problems you may encounter with this approach. This example has two different types of child forms: the first type hosts a circle drawn in the position of the last mouse click, and the second contains a bouncing square. The main form also has a custom background, obtained by painting a tiled image in it.

### Child Forms and Merging Menus

The first type of child form displays a circle in the position where the user clicked one of the mouse buttons. [Figure 8.4](#) shows an example of the output of the `MdiMulti` program. The program includes a `Circle` menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. It's interesting that to program the child form, you do not need to consider the existence of other forms or of the frame window. You simply write the code for the form, and that's all. The only special care required is for the menus of the two forms.



**Figure 8.4:** The output of the `MdiMulti` example, with a child window that displays circles

If you prepare a main menu for the child form, it will replace the main menu of the frame window when the child form is activated: An MDI child window cannot have a menu of its own. But the fact that a child window can't have any menus should not bother you, because this is the standard behavior of MDI applications. You can use the frame window's menu bar to display the child window's menus. Even better, you can merge the frame window's menu bar with that of the child form. For example, in this program, the child form's menu can be placed between the frame window's `File` and `Window` pull-down menus. You can accomplish this using the following `GroupIndex` values:

- `File` pull-down menu, main form: 1
- `Circle` pull-down menu, child form: 2
- `Window` pull-down menu, main form: 3

Using these settings for the menu group indexes, the menu bar of the frame window will have either two or three pull-down menus. At startup, the menu bar has two menus. As soon as you create a child window, there are three menus; and when the last child window is closed (destroyed), the `Circle` pull-down menu disappears. You should spend some time testing this behavior by running the program.

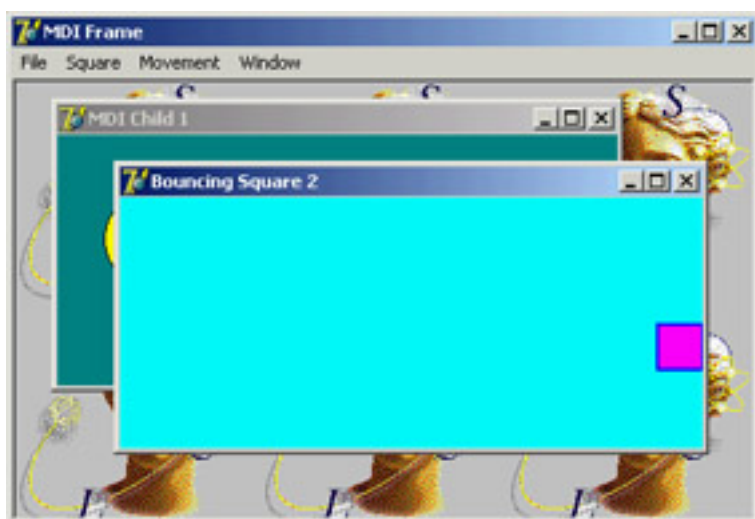
The second type of child form shows a moving image. The square (a `Shape` component) moves around the client area of the form at fixed time intervals, using a `Timer` component, and bounces against the edges of the form, changing its direction. This turning process is determined by a fairly complex algorithm, which I don't have space to examine; the main point of the example is to show you how menu merging behaves when you have an MDI frame with child forms of

different types. (You can study the source code to see how it works.)

## The Main Form

Now let's integrate the two child forms into an MDI application. The File pull-down menu has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. The Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the main form's original menu bar is reset. By using the proper menu group indexes, you let Delphi accomplish everything automatically, as you can see in [Figure 8.5](#).



**Figure 8.5:** The menu bar of the MdiMulti application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of [Figure 8.4](#).

I've added a few other menu items in the main form to close every child window and show some statistics about them. The method related to the Count command scans the MDIChildren array property to count the number of child windows of each kind (using the RTTI operator is):

```
for I := 0 to MDIChildCount - 1 do
  if MDIChildren is TBounceChildForm then
    Inc (NBounce)
  else
    Inc (NCircle);
```

## Subclassing the MDI Client Window

The example program also includes support for a background-tiled image. The bitmap is taken from an Image component and should be painted on the form in the `wm_EraseBkgnd` Windows message's handler. The problem is that you cannot simply connect the code to the main form, because a separate window (the MDI Client) covers its surface.

You have no corresponding Delphi form for this window, so how can you handle its messages? You have to resort to a low-level Windows programming technique known as *subclassing*. (In spite of the name, it has little to do with OOP inheritance.) The basic idea is that you can replace the window procedure that receives all the window messages with a new procedure you provide. You can do so by calling the `SetWindowLong` API function and providing the memory address of the procedure (the function pointer).

**Note** A window procedure is a function that receives all the messages for a window. Every window must have a window procedure and can have only one. Even Delphi forms have a window procedure; although it is hidden in the system, it calls the *WndProc* virtual function, which you can use. However, the VCL has a predefined handler for the messages, which are then forwarded to the form's message-handling methods after some preprocessing. With all this support, you need to handle window procedures explicitly only when working with non-Delphi windows, as in this case.

Unless you have a reason to change the default behavior of this system window, you can simply store the original procedure and call it to obtain default processing. The two function pointers referring to the two procedures (old and new) are stored in two local fields on the form:

```
private
  OldWinProc, NewWinProc: Pointer;
  procedure NewWinProcedure (var Msg: TMessage);
```

The form also has a method you'll use as a new window procedure; the code will be used to paint on the background of the window. Because this is a method and not a plain window procedure, the program has to call the *MakeObjectInstance* method to add a prefix to the method and let the system use it as if it were a function. All this description is summarized by just two complex statements:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  NewWinProc := MakeObjectInstance (NewWinProcedure);
  OldWinProc := Pointer (SetWindowLong (ClientHandle, gwl_WndProc, Cardinal
    (NewWinProc)));
  OutCanvas := TCanvas.Create;
end;
```

The window procedure you install calls the default procedure. Then, if the message is *wm\_EraseBkgnd* and the image is not empty, you draw it on the screen many times using the *Draw* method of a temporary canvas. This canvas object is created when the program starts (see the previous code) and connected to the handle passed as *wParam* parameter by the message. With this approach, you don't have to create a new *TCanvas* object for every background painting operation requested, thus saving a little time in the frequent operation. Here is the code, which produces the output already seen in [Figure 8.5](#):

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
  BmpWidth, BmpHeight: Integer;
  I, J: Integer;
begin
  // default processing first
  Msg.Result := CallWindowProc (OldWinProc, ClientHandle, Msg.Msg, Msg.wParam,
    Msg.lParam);

  // handle background repaint
  if Msg.Msg = wm_EraseBkgnd then
  begin
    BmpWidth := MainForm.Imagel.Width;
    BmpHeight := MainForm.Imagel.Height;
    if (BmpWidth <> 0) and (BmpHeight <> 0) then
    begin
      OutCanvas.Handle := Msg.wParam;
      for I := 0 to MainForm.ClientWidth div BmpWidth do
        for J := 0 to MainForm.ClientHeight div BmpHeight do
          OutCanvas.Draw (I * BmpWidth, J * BmpHeight,
            MainForm.Imagel.Picture.Graphic);
    end;
  end;
end;
```



## Visual Form Inheritance

When you need to build two or more similar forms, possibly with different event handlers, you can use dynamic techniques, hide or create new components at run time, change event handlers, and use `if` or `case` statements. Or, you can apply the object-oriented techniques, thanks to visual form inheritance. In short, instead of creating a form based on `TForm`, you can inherit a form from an existing form, adding new components or altering the properties of the existing components. But what is the advantage of visual form inheritance?

It mostly depends on the kind of application you are building. If the program has multiple forms, some of which are very similar or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the application's forms.

You can also use visual form inheritance to customize an application for different clients without duplicating any source code or form definition code—you inherit the specific versions for a client from the standard forms. Remember, the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method in a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

**Note** Delphi includes another feature that resembles visual form inheritance: frames. In both cases, you can work at design time on two versions of a form/frame. However, in visual form inheritance, you define two different classes (parent and derived), whereas with frames, you work on a class and an instance. Frames are discussed in detail later in this chapter.

## Inheriting from a Base Form

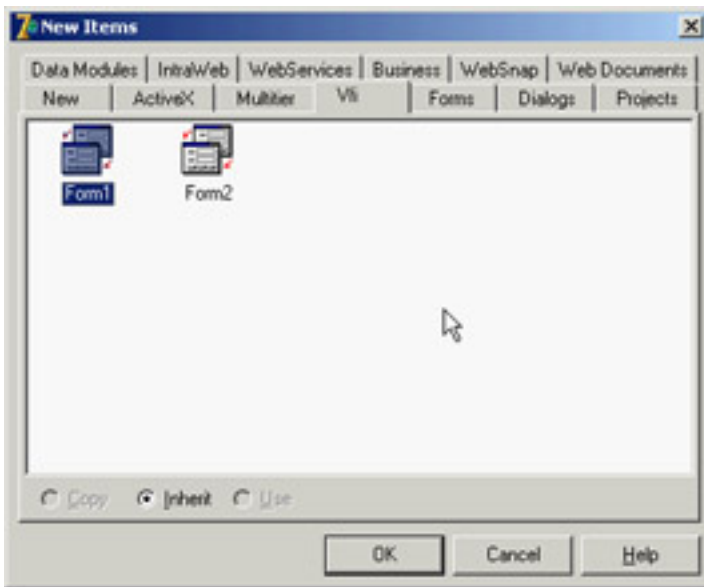
The rules governing visual form inheritance are simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command in the Object Inspector. You can do the same thing by setting the two properties to the same value and recompiling the code. After modifying multiple properties, you can resynchronize them all to the base version by applying the Revert to Inherited command from the component's local menu.

Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To describe how visual form inheritance works, I've built a simple example called VFI. To build it, first start a new project and add four buttons to its main form. Then select File → New → Other and choose the page with the name of the project in the New Items dialog box (see [Figure 8.6](#)).





**Figure 8.6:** The New Items dialog box allows you to create an inherited form.

In the New Items dialog, you can choose the form from which you want to inherit. The new form has the same four buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

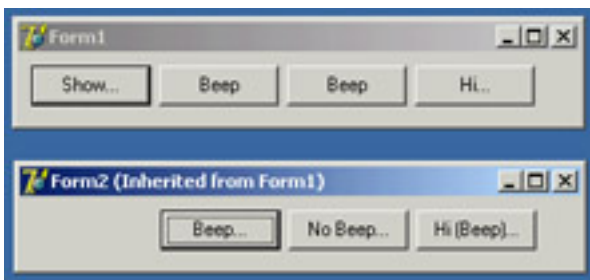
And here is its initial class declaration, where you can see that the base class is not the usual TForm but the base class form:

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
end;
```

Notice the presence of the `inherited` keyword in the textual description; also notice that the form has some components, although they are defined in the base class form. If you move the form and add the caption of one of the buttons, the textual description changes accordingly:

```
inherited Form2: TForm2
  Left = 313
  Top = 202
  Caption = 'Form2'
  inherited Button2: TButton
    Caption = 'Beep...'
  end
end
```

Only the properties with a different value are listed (and by removing these properties from the textual description of the inherited form, you can reset them to the value of the base form, as I mentioned earlier). I've changed the captions of most of the buttons, as you can see in [Figure 8.7](#).



**Figure 8.7:** The two forms of the VFI example at run time

Each of the first form's buttons has an `OnClick` handler with simple code. The first button shows the inherited form by calling its `Show` method; the second and third buttons call the `Beep` procedure; and the last button displays a message.

In the inherited form you should first remove the `Show` button, because the secondary form is already visible. However, you cannot delete a component from an inherited form. An alternative solution is to set the component's `Visible` property to `False`—the button will still be there, but it won't be visible (as you can guess from [Figure 8.7](#)). The other three buttons will be visible but with different handlers. If you select the `OnClick` event of a button in the inherited form (by double-clicking it), you'll get an empty method that's slightly different from the default one, because it includes the `inherited` keyword. This keyword stands for a call to the corresponding event handler of the base form. Notice, though, that this keyword is always added by Delphi, even if the handler is not defined in the parent class (and this is reasonable, because it might be defined later) or if the component is not present in the parent class (which doesn't seem like a great idea to me). It is simple to execute the base form's code and perform some other operations:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
    inherited;
    ShowMessage ('Hi');
end;
```

This is not the only choice. Alternatively, you can write a new event handler and not execute the base class's code, as I've done for the VFI example's third button: To accomplish this, simply remove the `inherited` keyword.

Still another choice includes calling a base-class method after some custom code has been executed, calling it when a condition is met, or calling the handler of a different event of the base class, as I've done for the fourth button:

```
procedure TForm2.Button4Click(Sender: TObject);
begin
    inherited Button3Click (Sender);
    inherited;
end;
```

You probably won't inherit from a different handler often, but you must be aware that you can. Of course, you can consider each method of the base form as a method of your form, and call it freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at real-world examples more complex than this book has room to explore. Next I want to show you *visual form polymorphism*.

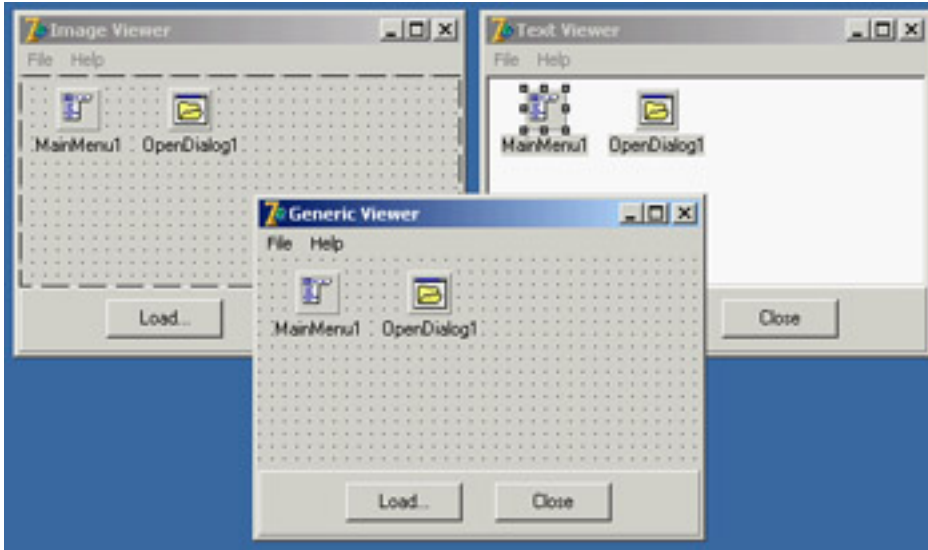
**Note** Visual form inheritance doesn't work nicely with collections: You cannot extend a collection property of a component in an inherited form. This limitation prevents the practical use of a series of components like Toolbars or ListViews with details. Of course, you can use those components in the parent or inherited form, but you cannot extend the elements they contain, because they are stored in a collection. A solution to this problem is to avoid assigning these collections at design time, and instead use a run-time technique. You'll still use form inheritance, but lose the visual portion of it. If you try to use the Action Manager component, you'll find you cannot even inherit from a form hosting it. Borland disabled this feature, because it would cause you too much trouble.

## Polymorphic Forms

If you add an event handler to a form and then change it in an inherited form, there is no way to refer to the two

methods using a common variable of the base class, because the event handlers use static binding by default.

Confusing? Here is an example, which is intended for experienced Delphi programmers. Suppose you want to build a bitmap viewer form and a text viewer form in the same program. The two forms have similar elements, a similar toolbar, a similar menu, an `OpenDialog` component, and different components for viewing the data. So, you decide to build a base-class form containing the common elements and inherit the two forms from it. You can see the three forms at design time in [Figure 8.8](#).



**Figure 8.8:** The base-class form and the two inherited forms of the PoliForm example at design time

The main form contains a toolbar panel with a few buttons (real toolbars have a few problems with visual form inheritance), a menu, and an open dialog component. The two inherited forms have only minor differences, but they feature a new component: either an image viewer (`TImage`) or a text viewer (`TMemo`). They also modify the settings of the `OpenDialog` component, to refer to different types of files.

The main form includes some common code. The `Close` button and the `File → Close` command call the `Close` method of the form. The `Help → About` command shows a simple message box. The base form's `Load` button has only a `ShowMessage` call displaying an error message. The `File → Load` command calls another method:

```
procedure TViewerForm.Load1Click(Sender: TObject);  
begin  
    LoadFile;  
end;
```

This method is defined in the `TViewerForm` class as a virtual abstract method (so that the class of the base form is an abstract class). Because this is an abstract method, you must redefine it (and override it) in the inherited forms. The code for this `LoadFile` method uses the `OpenDialog1` component to ask the user to select an input file and loads it into the image component:

```
procedure TImageViewerForm.LoadFile;  
begin  
    if OpenDialog1.Execute then  
        Image1.Picture.LoadFromFile (OpenDialog1.FileName);  
end;
```

The other inherited class has similar code, which loads the text into the memo component. The project has one more form, a main form with two buttons, that reloads the files in each of the viewer forms. The main form is the only form created by the project when it starts. The generic viewer form is never created: It is only a generic base class, containing common code and components of the two subclasses. The forms of the two subclasses are created in the main form's `OnCreate` event handler:

```

procedure TMainForm.FormCreate(Sender: TObject);
var
    I: Integer;
begin
    FormList [1] := TTextViewerForm.Create (Application);
    FormList [2] := TImageViewerForm.Create (Application);
    for I := 1 to 2 do
        FormList[I].Show;
end;

```

FormList is a *polymorphic* array of generic TViewerForm objects, declared in the TMainForm class. Note that to make this declaration in the class, you need to add the Viewer unit (but not the specific forms) in the `uses` clause of the interface portion of the main form. The array of forms is used to load a new file in each viewer form when one of the two buttons is clicked. The handlers of the two buttons' `OnClick` events use different approaches:

```

// ReloadButton1Click
for I := 1 to 2 do
    FormList [I].ButtonLoadClick (Self);

// ReloadButton2Click
for I := 1 to 2 do
    FormList [I].LoadFile;

```

The second button calls a virtual method, and it works without any problem. The first button calls an event handler and always reaches the generic TFormView class (displaying the error message of its `ButtonLoadClick` method). This happens because the method is static, not virtual.

To make this approach work, you can declare the `ButtonLoadClick` method of the TFormView class as virtual and declare it as overridden in each of the inherited form classes, as you do for any other virtual method:

```

type
    TViewerForm = class(TForm)
        procedure ButtonLoadClick(Sender: TObject); virtual;
    public
        procedure LoadFile; virtual; abstract;
    end;

type
    TImageViewerForm = class(TViewerForm)
        procedure ButtonLoadClick(Sender: TObject); override;
    public
        procedure LoadFile; override;
    end;

```

This trick really works, although it is never mentioned in the Delphi documentation. This ability to use virtual event handlers is what I mean by visual form polymorphism. In other (more technical) words, you can assign a virtual method to an event property, which will take the address of the method according to the instance available at run time.

## Understanding Frames

[Chapter 1](#), "Delphi 7 And Its IDE," briefly discussed frames. You've seen that you can create a new frame, place components in it, write event handlers for the components, and then add the frame to a form. In other words, a frame is similar to a form, but it defines only a portion of a window, not a complete window. The interesting element of frames is that you can create multiple instances of a frame at design time, and you can modify the class and the instance at the same time. Thus frames are an effective tool for creating customizable composite controls at design time—something close to a visual component-building tool.

In visual form inheritance, you can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one (unless doing so overrides a property or event). With frames, you work on a class (as usual in Delphi), but you can also customize one or more instances of the class at design time. When you work on a form, you cannot change a property of the `TForm1` class for a specific instance of this form, and not the others, at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, you can customize each instance at design time. You could already do this with component templates, but component templates were based on the concept of copying and pasting components and their code. You could not change the original definition of the template and see the effect every place it was used. With frames (and, in a different way, with visual form inheritance), changes to the original version (the class) are reflected in the copies (the instances).

Let's discuss a few more elements of frames with an example called `Frames2`. This program has a frame with a list box, an edit box, and three buttons with code operating on the components. The frame also has a bevel aligned to its client area, because frames have no border. Of course, the frame has also a corresponding class, which looks like a form class:

```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

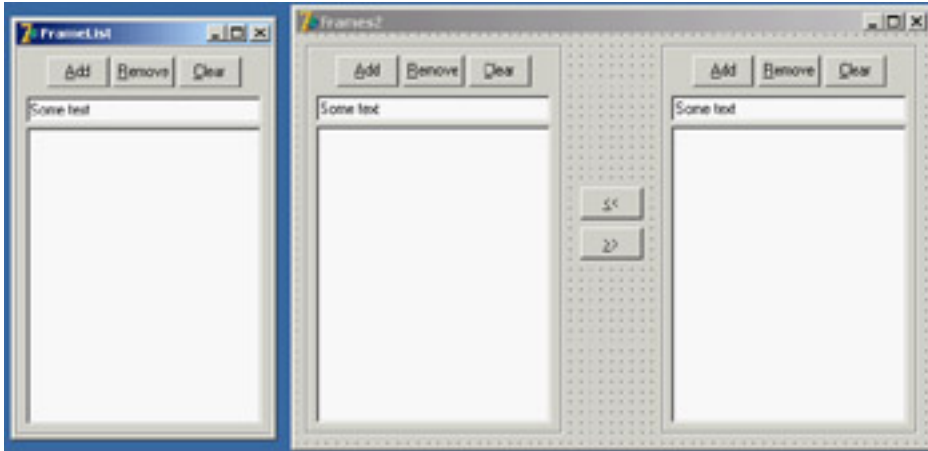
What is different from a form is that you can add the frame to a form. I've used two instances of the frame in the example (as you can see in [Figure 8.9](#)) and modified the behavior slightly. The first instance of the frame has the list box items sorted. When you change a property of a component of a frame, the DFM file of the hosting form will list the differences, as it does with visual form inheritance:

```
object FormFrames: TFormFrames
  Caption = 'Frames2'
  inline FrameList1: TFrameList
    Left = 8
    Top = 8
    inherited ListBox: TListBox
      Sorted = True
```

```

end
end
inline FrameList2: TFrameList
  Left = 232
  Top = 8
  inherited btnClear: TButton
    OnClick = FrameList2btnClearClick
  end
end
end
end

```



**Figure 8.9:** A frame and two instances of it at design time, in the Frames2 example

As you can see from the listing, the DFM file for a form that has frames uses a specific DFM keyword, `inline`. The references to the modified components of the frame, however, use the `inherited` keyword, although this term is used with an extended meaning: In this case, `inherited` doesn't refer to a base class you are inheriting from, but to the class from which you are instantiating (or inheriting) an object. It was a good idea, though, to use an existing feature of visual form inheritance and apply it to the new context. This approach lets you use the Revert to Inherited command of the Object Inspector or of the form to cancel the changes and get back to the default value of properties.

Notice also that unmodified components of the frame class are not listed in the DFM file of the form using the frame; and, the form has two frames with different names, but the components on the two frames have the same name. These components are not owned by the form, but are owned by the frame. This implies that the form has to reference those components through the frame, as you can see in the code for the buttons that copy items from one list box to the other:

```

procedure TFormFrames.btnLeftClick(Sender: TObject);
begin
  FrameList1.ListBox.Items.AddStrings (FrameList2.ListBox.Items);
end;

```

Finally, in addition to modifying properties of any instance of a frame, you can change the code of any of its event handlers. If you double-click one of the frame's buttons while working on the form (not on the stand-alone frame), Delphi will generate this code for you:

```

procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  FrameList2.btnClearClick(Sender);
end;

```

The line of code automatically added by Delphi corresponds to a call to the inherited event handler of the base class in visual form inheritance. This time, however, to get the default behavior of the frame, you need to call an event handler and apply it to a specific instance—the frame object itself. The current form doesn't include this event handler and knows nothing about it. Whether you leave this call in place or remove it depends on the effect you are looking for.

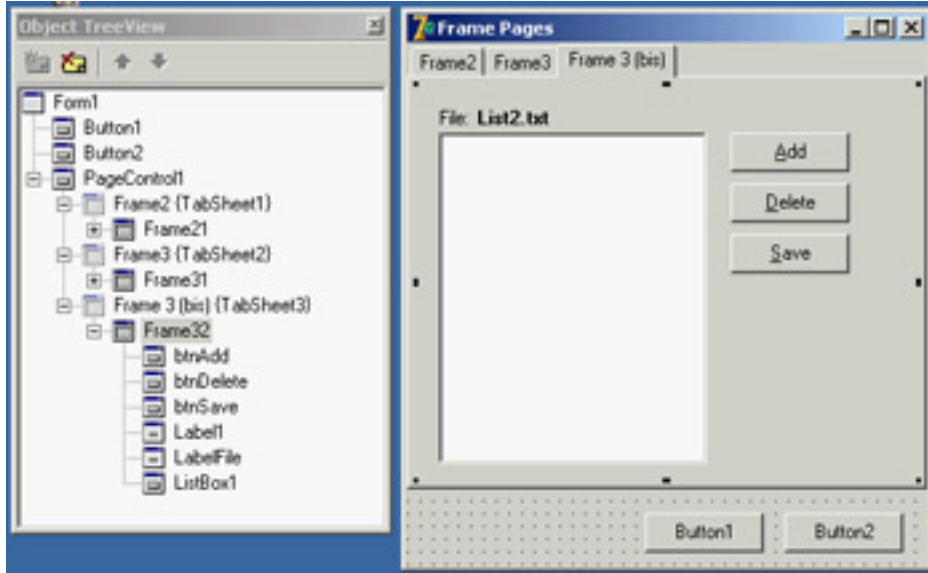
**Tip** Note that because the event handler has some code, leaving it as Delphi generated it and saving the form won't remove it as usual: It isn't empty! Instead, if you want to omit the default code for an event, you need to add at least a comment to it to avoid the system removing it automatically.

## Frames and Pages

When a dialog box has many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared in a single form. In addition, creating all these components (and initializing them) might delay the display of the dialog box. Frames don't reduce the construction and initialization time of equivalently loaded forms; quite the contrary, because loading frames is more complicated for the streaming system than loading simple components. However, using frames, you can load only the visible pages of a multipage dialog box, reducing the *initial* load time, which is what the user perceives.

Frames can solve both of these issues. You can easily divide the code of a single complex form into one frame per page. The form will host all the frames in a PageControl. This approach yields simpler, more focused units and makes it easier to reuse a specific page in a different dialog box or application. Reusing a single page of a PageControl without using a frame or an embedded form is far from simple. (For an alternative approach, see the sidebar "[Forms in Pages.](#)")

As an example of this approach, I've built the FramePag example. It has some frames placed inside the three pages of a PageControl, as you can see in [Figure 8.10](#) by looking at the Object TreeView on the side of the design-time form. All the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them. Two of the pages have the same frame, but the two instances of the frame have some differences at design time. The `Frame3` frame in the example has a list box populated with a text file at startup, and it has buttons to modify the items in the list and save them to a file. The filename is placed in a label so you can easily select a file for the frame at design time by changing the `Caption` of the label.



**Figure 8.10:** Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks.

Being able to use multiple instances of a frame is one of the reasons this technique was introduced, and customizing the frame at design time is even more important. Because adding properties to a frame and making them available at design time requires some customized and complex code, it is nice to use a component to host these custom values. You have the option of hiding these components (such as the label in this example) if they don't pertain to the user interface.

In the example, you need to load the file when the frame instance is created. Because frames have no `OnCreate` event, your best choice is probably to override the `CreateWnd` method. Writing a custom constructor doesn't work, because it is executed too early—before the specific label text is available. In the `CreateWnd` method, you load the list



box content from a file.

**Note** When questioned about the issue of the missing *OnCreate* event handler for frames, Borland R&D members have stated that they could not fire it in correspondence with the *wm\_Create* message, because it happens with forms. The creation of the frame window (as is true for most controls) is delayed for performance reasons. More trouble happens in the case of inheritance among forms holding frames, so to avoid problems, this features has been disabled—programmers can write the code they deem reasonable.

---

## Forms in Pages

Although you can use frames to define the pages of a *PageControl* at design time, I generally use other forms at run time. This approach leaves me with the flexibility of having the pages defined in separate units (and DFM files) but at the same time allows me to also use those forms as stand-alone windows. In addition, I avoid having to live with the subtly different behaviors of frames.

Once you have a main form with a page control and one or more secondary forms to display in it, all you have to do is write the following code to create the secondary forms and place them in the pages:

```
var
  Form: TForm;
  Sheet: TTabSheet;
begin
  // create a tabsheet within the page control
  Sheet := TTabSheet.Create(PageControll);
  Sheet.PageControl := PageControll;
  // create the form and place it in the tabsheet
  Form := TForm2.Create (Application);
  Form.BorderStyle := bsNone;
  Form.Align := alClient;
  Form.Parent := Sheet;
  Form.Visible := True;
  // activate and set title
  PageControll.ActivePage := Sheet;
  Sheet.Caption := Form.Caption;
end;
```

You can find this code in the *FormPage* example, but this is all the program does. For an application, see the *RWBlocks* demo in [Chapter 14](#), "Client/Server with dbExpress."

---

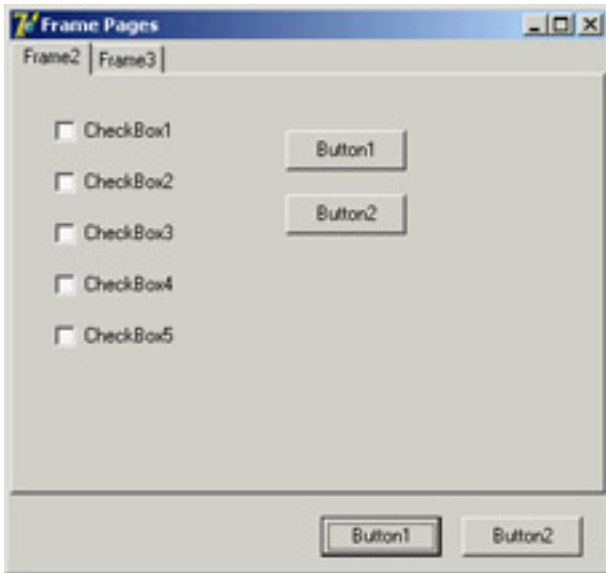
## Multiple Frames with No Pages

Another approach avoids creating all the pages along with the form hosting them, by leaving the *PageControl* empty and creating the frames only when a page is displayed. When you have frames on multiple pages of a *PageControl*, the windows for the frames are created only when they are first displayed, as you can find out by placing a breakpoint in the creation code of the previous example.

As an even more radical approach, you can get rid of the page controls and use a *TabControl*. Used this way, the tab has no connected tab sheets (or pages) and can display only one set of information at a time. For this reason, you must create the current frame and either destroy the previous one or hide it by setting its *Visible* property to *False* or calling the new frame's *BringToFront*. Although this sounds like a lot of work, in a large application this technique can be worth it because of the reduced resource and memory usage you can obtain.

To demonstrate this approach, I've built the *FrameTab* example, which is similar to the previous one but it is based on a *TabControl* and dynamically created frames. The main form, visible at run time in [Figure 8.11](#), has a *TabControl* with one page for each frame:





**Figure 8.11:** The first page of the FrameTab example at run time. The frame inside the tab is created at run time.

```

object Form1: TForm1
  Caption = 'Frame Pages'
  OnCreate = FormCreate
  object Button1: TButton...
  object Button2: TButton...
  object Tab: TTabControl
    Anchors = [akLeft, akTop, akRight, akBottom]
    Tabs.Strings = ( 'Frame2' 'Frame3' )
    OnChange = TabChange
  end
end

```

I've given each tab a caption corresponding to the name of the frame, because I'll use this information to create the new pages. When the form is created, and whenever the user changes the active tab, the program gets the tab's current caption and passes it to the custom `ShowFrame` method. The method's code checks whether the requested frame already exists (frame names in this example follow the Delphi standard of having a number appended to the class name) and then brings it to the front. If the frame doesn't exist, the method uses the frame name to find the related frame class, creates an object of that class, and assigns a few properties to it. The code makes extensive use of class references and dynamic creation techniques:

```

type
  TFrameClass = class of TFrame;

procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + '1') as TFrame;
  if not Assigned (Frame) then
  begin
    FrameClass := TFrameClass (FindClass ('T' + FrameName));
    Frame := FrameClass.Create (Self);
    Frame.Parent := Tab;
    Frame.Visible := True;
    Frame.Name := FrameName + '1';
  end;

```

```
Frame.BringToFront;  
end;
```

To make this code work, remember to add a call to `RegisterClass` in the initialization section of each unit defining a frame.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Base Forms and Interfaces

You have seen that when you need two similar forms in an application, you can use visual form inheritance to inherit one from the other or both of them from a common ancestor. The advantage of visual form inheritance is that you can use it to inherit the visual definition: the DFM. However, this is not always required.

At times, you might want several forms to exhibit a common behavior or respond to the same commands without having any shared component or user interface elements. Using visual form inheritance with a base form that has no extra components makes little sense to me. I rather prefer to define my own custom form class, inherited from `TForm`, and then manually edit the form class declarations to inherit from this custom base form class instead of the standard one. If you only need to define shared methods or override `TForm` virtual methods in a consistent way, defining custom form classes can be a good idea.

### Using a Base Form Class

A simple demonstration of this technique is available in the `FormIntf` demo; it also showcases the use of interfaces for forms. In a new unit called `SaveStatusForm`, I've defined the following form class (with no related DFM file—instead of using the `New Form` command, create a new unit and type the code in it):

```
type
  TSaveStatusForm = class (TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

The two overridden methods are called at the same time as the event handler so you can attach extra code (allowing the event handler to be defined as usual). Inside the two methods, you load or save the form position in an INI file of the application, in a section marked with the form caption. Here is the code for the two methods:

```
procedure TSaveStatusForm.DoCreate;
var
  Ini: TIniFile;
begin
  inherited;
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Left := Ini.ReadInteger(Caption, 'Left', Left);
  Top := Ini.ReadInteger(Caption, 'Top', Top);
  Width := Ini.ReadInteger(Caption, 'Width', Width);
  Height := Ini.ReadInteger(Caption, 'Height', Height);
  Ini.Free;
end;
```

```
procedure TSaveStatusForm.DoDestroy;
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Ini.WriteInteger(Caption, 'Left', Left);
  Ini.WriteInteger(Caption, 'Top', Top);
  Ini.WriteInteger(Caption, 'Width', Width);
  Ini.WriteInteger(Caption, 'Height', Height);
  Ini.Free;
  inherited;
end;
```

Again, this is a simple common behavior for your forms, but you can define a complex class here. To use this as a base class for the forms you create, let Delphi create the forms as usual (with no inheritance) and then update the form declaration to something like the following:

```
type
  TFormBitmap = class(TSaveStatusForm)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    ...
```

Simple as it seems, this technique is very powerful, because all you need to do is change the definition of your application's forms to refer to this base class. If even this step is too tedious, because you might want to change this base class in your program at some point, you can use an extra trick: "interposer" classes.

---

## INI Files and the Registry in Delphi

To save information about the status of an application in order to restore it the next time the program is executed, you can use the explicit support Windows provides for storing this kind of information. INI files, the old Windows standard, are once again the preferred way to save application data. The alternative is the Registry, which is still quite popular. Delphi provides ready-to-use classes to manipulate both.

### The *TIniFile* Class

For INI files, Delphi has a `TIniFile` class. Once you have created an object of this class and connected it to a file, you can read and write information to it. To create the object, you need to call the constructor, passing a filename to it, as in the following code:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create ('myprogram.ini');
```

There are two choices for the location of the INI file. The code just listed will store the file in the Windows directory or a user folder for settings in Windows 2000. To store data locally to the application (as opposed to local to the current user), you should provide a full path to the constructor.

INI files are divided into sections, each indicated by a name enclosed in square brackets. Each section can contain multiple items of three possible kinds: strings, integers, or Booleans. The `TIniFile` class has three Read methods, one for each kind of data: `ReadBool`, `ReadInteger`, and `ReadString`. There are also three corresponding methods to write the data: `WriteBool`, `WriteInteger`, and `WriteString`. Other methods allow you to read or erase a whole section. In the Read methods, you can also specify a default value to be used if the corresponding entry doesn't exist in the INI file.

Notice that Delphi uses INI files quite often, but they are disguised with different names. For example, the desktop (`.dsk`) and options (`.dof`) files are structured as INI files.

### The *TRegistry* and *TRegIniFile* classes

The Registry is a hierarchical database of information about the computer, software configuration, and user preferences. Windows has a set of API functions to interact with the Registry; you basically open a key (or folder) and then work with subkeys (or subfolders) and values (or items), but you must be aware of the structure and the details of the Registry.

Delphi provides two approaches to using the Registry. The `TRegistry` class provides a generic encapsulation of the Registry API, whereas the `TRegIniFile` class provides the interface of the `TIniFile` class but saves the data in the Registry. This class is the natural choice for portability between INI-based and Registry-based versions of the same program. When you create a `TRegIniFile` object, your data ends up in the current user information, so you'll generally use a constructor like this:

```
IniFile := TRegIniFile.Create ('Software\MyCompany\MyProgram');
```

By using the `TIniFile` and the `TRegIniFile` classes offered by the VCL, you can move from one model of local and per-user storage to the other. Not that I think you should use the Registry much, because having a centralized repository for the settings of each application was an architectural error—even Microsoft acknowledges this fact (without really admitting the error) by suggesting, in the Windows 2000 Compatibility Requirements, that you no longer use the Registry for applications settings, but instead go back to using INI files within the Documents and Settings folder of the current user (something not many programmers know of).

---

## An Extra Trick: Interposer Classes

In contrast with Delphi VCL components, which must have unique names, Delphi classes in general must be unique only within their unit. Thus you can have two different units defining a class with the same name. This technique looks weird at first sight, but can be useful. For example, Borland uses this approach to provide compatibility between VCL and VisualCLX classes. Both have a `TForm` class, one defined in the Forms unit and the other in the QForms unit.

**Note** This technique is much older than CLX/VCL. For example, the service and control panel applet units define their own `TApplication` object, which is not related to the `TApplication` used by VCL visual GUI applications and defined in the Forms unit.

I've seen a technique called "interposer classes" mentioned in an old issue of *The Delphi Magazine*. It suggested replacing standard Delphi class names with your own versions that have the same class name. This way, you can use Delphi's form designer and refer to Delphi standard components at design time, but use your own classes at run time.

The idea is simple. In the `SaveStatusForm` unit, you can define the new form class as follows:

```
type
  TForm = class (Forms.TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

This class is called `TForm`, and it inherits from `TForm` of the Forms unit (this last reference is compulsory to avoid a kind of recursive definition). In the rest of the program, you don't need to change the class definition for your form, but simply add the unit defining the interposer class (the `SaveStatusForm` unit in this case) in the `uses` statement *after* the unit defining the Delphi class. The order of the unit in the `uses` statement is important, and is the reason some people criticize this technique, because it is difficult to know what is going on. I have to agree: I find interposer classes handy at times (more for components than for forms), but their use makes programs less readable and at times harder to debug.

## Using Interfaces

Another technique, which is slightly more complex but even more powerful than the definition of a common base form class, is to create forms that implement specific interfaces. You can have forms that implement one or more of these interfaces, query each form for the interfaces it implements, and call the supported methods.

As an example (available in the same `FormIntf` program I began discussing in the last section), I've defined a simple interface for loading and storing:

```
type
  IFormOperations = interface
    ['{DACFDB76-0703-4A40-A951-10D140B4A2A0}']
    procedure Load;
    procedure Save;
```

```
end;
```

Each form can optionally implement this interface, as in the following `TFormBitmap` class:

```
type  
TFormBitmap = class(TForm, IFormOperations)  
  Image1: TImage;  
  OpenPictureDialog1: TOpenPictureDialog;  
  SavePictureDialog1: TSavePictureDialog;  
public  
  procedure Load;  
  procedure Save;  
end;
```

The example code includes the `Load` and `Save` methods, which use the standard dialog boxes to load or save the image. (In the example's code, the form also inherits from the `TSaveStatusForm` class.)

When an application has one or more forms implementing interfaces, you can apply a given interface method to all the forms supporting it, with code like this (extracted from the main form of the `FormIntf` example):

```
procedure TFormMain.btnLoadClick(Sender: TObject);  
var  
  i: Integer;  
  iFormOp: IFormOperations;  
begin  
  for i := 0 to Screen.FormCount - 1 do  
    if Supports (Screen.Forms [i], IFormOperations, iFormOp) then  
      iFormOp.Load;  
end;
```

Consider a business application in which you can synchronize all the forms to the data of a specific company or a specific business event. Also consider that, unlike inheritance, you can have several forms that implement multiple interfaces, with unlimited combinations. This is why using such an architecture can improve a complex Delphi application a great deal, making it much more flexible and easier to adapt to implementation changes.

## Delphi's Memory Manager

I'll end this chapter devoted to the structure of Delphi applications with a section devoted to memory management. This topic is very complex, and probably worth an entire chapter of its own; here I can only scratch it and provide a few indications for further experiments. For more detailed memory analysis you can refer to the many Delphi add-on tools addressing memory verification and control, including MemCheck, MemProof, MemorySleuth, Code Watch, and AQTime.

Delphi has a memory manager, accessible using the `GetMemoryManager` and `SetMemoryManager` functions of the `System` unit. These functions allow you to retrieve the current memory manager record or modify it with your custom memory manager. A *memory manager record* is a set of three functions used to allocate, deallocate, and reallocate memory:

**type**

```
TMemoryManager = record
  GetMem: function(Size: Integer): Pointer;
  FreeMem: function(P: Pointer): Integer;
  ReallocMem: function(P: Pointer; Size: Integer): Pointer;
end;
```

It's important to know how these functions are called when you create an object, because you can hook in two different steps. As you call a constructor, Delphi invokes the `NewInstance` virtual class function, defined in `TObject`. Because this is a virtual function, you can modify the memory manager for a specific class by overriding it. To perform the memory allocation, however, `NewInstance` typically ends up calling the `GetMem` function of the active memory manager, which provides you with your second chance to customize the standard behavior.

Unless you have very special needs, you won't generally need to hook into the memory manager to modify how memory allocation works. However, I find it quite useful to hook into a memory manager to determine whether memory allocation is working properly—that is, to be sure the program has no memory leaks. For example, you can override a class's `NewInstance` and `FreeInstance` methods to keep a count of the number of objects of the class being created and destroyed and check if the total is zero.

An even simpler technique is to perform the same test over the number of objects allocated by the entire memory manager. In the early versions of Delphi doing so required extra code, but the memory manager exposes two global variables (`AllocMemCount` and `AllocMemSize`) that can help you determine what is going on in the system.

**Note** For more detailed information about how the memory manager is working internally, you can use the `GetHeapStatus` function. It is available only on Windows, because it provides information about the status of the memory allocator. On Linux, the RTL uses the system allocator, not a custom memory allocator.

The simplest way to determine whether your program is handling memory properly is to test whether `AllocMemCount` goes back to zero. The problem is deciding when to perform such a test. A program begins by executing the initialization section of its units, which usually allocate memory freed by the respective finalization sections. To guarantee that your code is executed at the very end, you must write it in the finalization section of a unit and place it at the very beginning of the units list in the project source code file. You can see such a unit in [Listing 8.1](#). This is the `SimpleMemTest` unit of the `ObjsLeft` example, which has a sample form with a button to show the current allocations count and a button to create a memory leak (which is then caught when the program terminates).

### Listing 8.1: A Simple Unit for Testing Memory Leaks, from the `ObjsLeft` Example

---

```
unit SimpleMemTest;
```

```
interface
```

```
implementation
```

```
uses
```

Windows;

```
var
  msg: string;

initialization

finalization
  if AllocMemCount > 0 then
  begin
    Str (AllocMemCount, msg);
    msg := msg + ' heap blocks left';
    MessageBox (0, PChar(msg), 'Memory Leak', MB_OK);
  end;
end.
```

---

**Tip** When writing code that involves checking, implementing, or extending memory manager code, you have to avoid using any high-level functions, as they might affect the memory manager. For example, in the SimpleMemTest unit, I couldn't include the SysUtils unit because it allocates memory. I had to resort to the traditional Turbo Pascal *Str* function instead of Delphi's standard *IntToStr* conversion.

This program is handy, but doesn't really help you understand what went wrong. For this purpose, powerful third-party tools are available (some of which have free trial versions), or you can refer to my custom memory manager, which tracks memory allocations (described in [Appendix A](#) of this book).

Team LiB

◀ PREVIOUS

NEXT ▶



## What's Next?

After the detailed description of forms and secondary forms in the previous chapters, I have focused on the architecture of applications, discussing both how Delphi's `Application` object works and how you can structure applications with multiple forms.

In particular, I've discussed MDI, visual form inheritance, and frames. Toward the end of the chapter I also discussed custom architectures, with form inheritance and interfaces. Now we can move forward to another key element of non-trivial Delphi applications: building custom components to use in your programs. I could write a book about this topic, so the description won't be exhaustive; but I'll offer a comprehensive overview.

Another element associated with the architecture of Delphi applications is the use of packages, which I'll introduce as a technology related to components but which really goes further. You can structure the code of a large application in multiple packages containing forms and other units. The development of programs based on multiple executable files, libraries, and packages, is discussed in [Chapter 10](#).

After this step, I will begin delving into Delphi database programming, another key element of the Borland development environment and for many developers, the prime focus.

## Chapter 9: Writing Delphi Components

### Overview

Most Delphi programmers are probably familiar with using existing components, but at times it can also be useful to write your own components or to customize existing ones. One of the most interesting aspects of Delphi is that creating components is not much more difficult than writing programs. For this reason, even though this book is intended for Delphi application programmers and not Delphi tool writers, this chapter will discuss creating components and introduce Delphi add-ins, such as component and property editors.

This chapter gives you an overview of writing Delphi components and presents some examples. There is not enough space to present very complex components, but the ideas I've included cover all the basics and will get you started.

**Note** You'll find more information about writing components in [Chapter 17](#), "Writing Database Components," including how to build data-aware components.

## Extending the Delphi Library

Delphi components are classes, and the Visual Components Library (VCL) is the collection of all the classes defining Delphi components. You extend the VCL by writing new component classes in a package and installing it in Delphi. These new classes will be derived from one of the existing component-related classes or the generic `TComponent` class, adding new capabilities to those they inherit.

You can derive a new component from an existing component or from an *abstract component class*—one that does not correspond to a usable component. The VCL hierarchy includes many of these intermediate classes (often indicated with the `TCustom` prefix in their name) to let you choose a default behavior for your new component and change its properties.

## Component Packages

Components are added to component packages. Each component package is basically a DLL (a dynamic link library) with a BPL extension (which stands for Borland Package Library).

Packages come in two flavors: design-time packages used by the Delphi IDE and run-time packages optionally used by applications. The design-only or run-only package option determines the package's type. When you attempt to install a package, the IDE checks whether it has the design-only or run-only flag, and decides whether to let the user install the package and whether it should be added to the list of run-time packages. Because there are two nonexclusive options, each with two possible states, there are four different kinds of component packages—two main variations and two special cases:

- Design-only component packages can be installed in the Delphi environment. These packages usually contain the design-time parts of a component, such as its property editors and the registration code. Often they also contain the components themselves, although this is not the most professional approach. The code of a design-only package's components is usually statically linked into the executable file, using the code of the corresponding Delphi Compiled Unit (DCU) files. Keep in mind, however, that it is also technically possible to use a design-only package as a run-time package.
- Run-only component packages are used by Delphi applications at run time. They cannot be installed in the Delphi environment, but they are automatically added to the list of run-time packages when they are required by a design-only package you install. Run-only packages usually contain the code of the component classes, but no design-time support (this is done to minimize the size of the component libraries you ship with your executable file). Run-only packages are important because they can be freely distributed with applications, but other Delphi programmers won't be able to install them in the environment to build new programs.
- Plain component packages (having neither the design-only nor the run-only option set) cannot be installed and will not be added to the list of run-time packages automatically. This option might make sense for utility packages used by other packages, but such packages are certainly rare.
- Packages with both flags set can be installed and are automatically added to the list of run-time packages. Usually these packages contain components requiring little or no design-time support (apart from the limited component registration code).

**Tip** The filenames of Delphi's own design-only packages begin with the letters *DCL* (for example, *DCLSTD60.BPL*); filenames of run-only packages begin with the letters *VCL* (for example, *VCL60.BPL*). You can follow the same approach for your own packages, if you wish.

In [Chapter 1](#), "Delphi 7 and Its IDE," we discussed the effect of packages on the size of a program's executable file. Now we'll focus on building packages, because this is a required step in creating or installing components in Delphi.

When you compile a run-time package, you produce both a DLL with the compiled code (the BPL file) and a file with only symbol information (a DCP file), including no compiled machine code. The Delphi compiler uses the latter file to gather symbol information about the units that are part of the package without having access to the unit (DCU) files,

which contain both the symbol information and the compiled machine code. This process reduces compilation time and allows you to distribute just the packages without the precompiled unit files. The precompiled units are still required to statically link the components into an application. Distribution of precompiled DCU files (or source code) may make sense depending on the kind of components you develop. You'll see how to create a package after we've discussed some general guidelines and built a component.

**Note** DLLs are executable files containing collections of functions and classes, which can be used by an application or another DLL at run time. The typical advantage is that if many applications use the same DLL, only one copy needs to be on the disk or loaded in memory, and the size of each executable file will be much smaller. This is what happens with Delphi packages, as well. [Chapter 10](#), "Libraries and Packages," looks at DLLs and packages in more detail.

## Rules for Writing Components

Some general rules govern the writing of components. You can find a detailed description of most of them in the *Delphi Component Writer's Guide* Help file, which is required reading for Delphi component writers.

Here is my own summary of the rules for component writers:

- Study the Delphi language with care. Particularly important concepts are inheritance, method overriding and overloading, the difference between public and published sections of a class, and the definition of properties and events. If you don't feel confident with the Delphi language or basic VCL concepts, you can refer to the overall description of the language and library presented in [Part I](#) of the book, particularly [Chapters 2](#) ("The Delphi Programming Language") and [4](#) ("Core Library Classes").
- Study the structure of the VCL class hierarchy and keep a graph of the classes at hand (such as the one included with Delphi).
- Follow the standard Delphi naming conventions. There are several for components, as you will see, and following these rules makes it easier for other programmers to interact with your components and further extend them.
- Keep components simple, mimic other components, and avoid dependencies. These three rules basically mean that a programmer using your components should be able to use them as easily as preinstalled Delphi components. Use similar property, method, and event names whenever possible. If users don't need to learn complex rules about the use of your component (that is, if the dependencies between methods or properties are limited) and can access properties with meaningful names, they'll be happy.
- Use exceptions. When something goes wrong, the component should raise an exception. When you are allocating resources of any kind, you must protect them with `try/finally` blocks or destructor calls as appropriate.
- To complete a component, add a bitmap to it, which will be used by Delphi's Component Palette. If you intend for your component to be used by more than a few people, consider adding a Help file as well.
- Be ready to write *real* code and forget about the visual aspects of Delphi. Writing components generally means writing code without visual support (although Class Completion can speed up the coding of plain classes). The exception to this rule is that you can use frames to write components visually.

**Note** You can also use a third-party component-writing tool to build your component or to speed up its development. The most powerful third-party tool I know of for creating Delphi components is the Class Developer Kit (CDK) from Eagle Software ([www.eagle-software.com](http://www.eagle-software.com)), but many others are available.

## The Base Component Classes

To build a new component, you generally begin with an existing one or with one of the VCL base classes. In either case, your component is in one of three broad component categories (introduced in [Chapter 4](#)), set by the three basic classes of the component hierarchy:

- `TWinControl` is the parent class of any component based on a window. Components that descend from this class can receive the input focus and get Windows messages from the system. You can also use their window handle when calling API functions. When creating a brand-new window control, you'll generally inherit from the derived class `TCustomControl`, which has a few extra useful features (particularly some support for painting the control).
- `TGraphicControl` is the parent class of visible components that have no Windows handle (which saves some Windows resources). These components cannot receive the input focus or respond to Windows messages directly. When creating a brand-new graphical control, you'll inherit directly from this class (which has a set of features very similar to `TCustomControl`).
- `TComponent` is the parent class of all components (including the controls) and can be used as a direct parent class for nonvisual components.

In the rest of the chapter, you will build some components using various parent classes, and we'll look at the differences among them. Let's begin with components inheriting from existing components or classes at a low level of the hierarchy, and then look at examples of classes inheriting directly from the ancestor classes just mentioned.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Building Your First Component

Building components is an important activity for Delphi programmers. Basically, any time you need the same behavior in two different places in an application, or in two different applications, you can place the shared code in a class—or, even better, in a component.

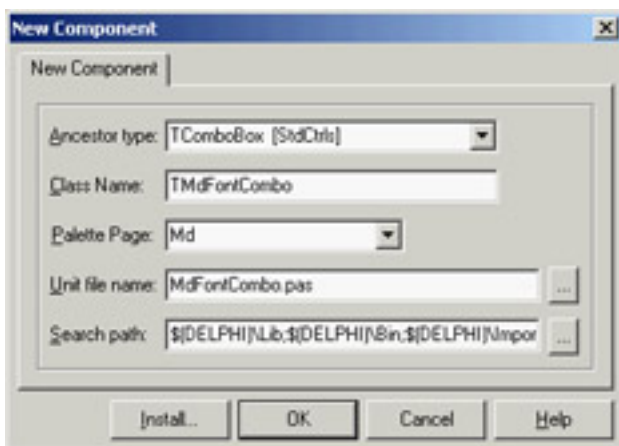
In this section, I'll introduce a couple of components to give you an idea of the steps required to build one. I'll also show you different things you can do to customize an existing component with a limited amount of code.

### The Fonts Combo Box

Many applications have a toolbar with a combo box you can use to select a font. If you often use such a customized combo box, why not turn it into a component? Doing so will probably take less than a minute.

To begin, close any active projects in the Delphi environment and start the Component Wizard, either by choosing Component → New Component, or by selecting File → New → Other to open the Object Repository and then choosing the component in the New page. As you can see, the Component Wizard requires the following information:

- The name of the ancestor type: the component class you want to inherit from. In this case, use `TComboBox`.
- The name of the class of the new component you are building; use `TMdFontCombo`.
- The Component Palette page where you want to display the new component, which can be a new or an existing page. Create a new page, called *Md*.
- The filename of the unit where Delphi will place the source code of the new component; type `MdFontCombo`.
- The current search path (which should be set up automatically).



Click the OK button, and the Component Wizard will generate the source file shown in [Listing 9.1](#) with the structure of your component. The Install button can be used to install the component in a package immediately. Let's look at the code first and then discuss the installation.

#### Listing 9.1: Code of the `TMdFontCombo` Class, Generated by the Component Wizard

```
unit MdFontCombo;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
  StdCtrls;
```

```

type
  TMdFontCombo = class (TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md', [TMdFontCombo]);
end;

end.

```

---

One of the key elements of this listing is the class definition, which begins by indicating the parent class. The only other relevant portion is the `Register` procedure. As you can see, the Component Wizard does very little work.

**Warning** The *Register* procedure *must* be written with an uppercase *R*. This requirement is imposed for C++Builder compatibility (identifiers in C++ are case-sensitive).

**Tip** Use a naming convention when building components. All the components installed in Delphi should have different class names. For this reason, most Delphi component developers have chosen to add a two- or three-letter signature prefix to the names of their components. I've done the same, using *Md* ( for *Mastering Delphi*) to identify components built in this book. The advantage of this approach is that you can install my *TMdFontCombo* component even if you've already installed a component named *TFontCombo*. Notice that the unit names must also be unique for all the components installed in the system, so I've applied the same prefix to the unit names.

That's all it takes to build a component. Of course, this example doesn't include much code. You need only copy all the system fonts to the `Items` property of the combo box at startup. To do so, you might try to override the `Create` method in the class declaration, adding the statement `Items := Screen.Fonts`. However, this is not the correct approach. The problem is, you cannot access the combo box's `Items` property before the window handle of the component is available; the component cannot have a window handle until its `Parent` property is set; and that property isn't set in the constructor, but later.

For this reason, instead of assigning the new strings in the `Create` constructor, you must perform this operation in the `CreateWnd` procedure, which is called to create the window control after the component is constructed, its `Parent` property is set, and its window handle is available. Again, you execute the default behavior, and then you can write your custom code. I could have skipped the `Create` constructor and written all the code in `CreateWnd`, but I decided to use both startup methods to demonstrate the difference between them. Here is the declaration of the component class:

```

type
  TMdFontCombo = class (TComboBox)
  private
    FChangeFormFont: Boolean;

```

```

procedure SetChangeFormFont(const Value: Boolean);
public
  constructor Create (AOwner: TComponent); override;
  procedure CreateWnd; override;
  procedure Change; override;
published
  property Style default csDropDownList;
  property Items stored False;
  property ChangeFormFont: Boolean
    read FChangeFormFont write SetChangeFormFont default True;
end;

```

And here is the source code of its two methods executed at startup:

```

constructor TMdFontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
  FChangeFormFont := True;
end;

procedure TMdFontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items.Assign (Screen.Fonts);

  // grab the default font of the owner form
  if FChangeFormFont and Assigned (Owner) and (Owner is TForm)
  then
    ItemIndex := Items.IndexOf ((Owner as TForm).Font.Name);
end;

```

Notice that besides giving a new value to the component's `Style` property in the `Create` method, you redefine this property by setting a value with the `default` keyword. You have to do both operations because adding the `default` keyword to a property declaration has no direct effect on the property's initial value. Why specify a property's default value then? Because properties that have a value equal to the default are not streamed with the form definition (and they don't appear in the textual description of the form, the DFM file). The `default` keyword tells the streaming code that the component initialization code will set the value of that property.

**Tip** It is important to specify a default value for a published property to reduce the size of the DFM files and, ultimately, the size of the executable files (which include the DFM files).

The other redefined property, `Items`, is set as a property that should not be saved to the DFM file at all, regardless of the actual value. This behavior is obtained with the `stored` directive followed by the value `False`. The component and its window will be created again when the program starts, so it doesn't make sense to save in the DFM file information that will be discarded later (to be replaced with the new list of fonts).

**Note** You can write the code of the `CreateWnd` method to copy the fonts to the combo box items only at run time, using statements such as `if not (csDesigning in ComponentState)`. But for this first component you are building, the less efficient but more straightforward method I've described offers a clearer illustration of the basic procedure.

The third property, `ChangeFormFont`, is not inherited but introduced by the component. It is used to determine whether the current font selection in the combo box should specify the font of the form hosting the component. Again, this property is declared with a default value, which is set in the constructor. The `ChangeFormFont` property is used in the code of the `CreateWnd` method, shown earlier, to set up the initial selection of the combo depending on the font of the form hosting the component. This is generally the component's `Owner`, although I could also have walked the



Parent tree looking for a form component. This code isn't perfect, but the `Assigned` and `is` tests provide some extra safety.

The `ChangeFormFont` property and the same `if` test play a key role in the `Changed` method, which in the base class triggers the `OnChange` event. By overriding this method, you provide a default behavior (which can be disabled by toggling the value of the property) but also allow the execution of the `OnChange` event, so that users of this class can fully customize its behavior. The final method, `SetChangeFormFont`, has been modified to refresh the form's font in case the property is being turned on. The complete code is as follows:

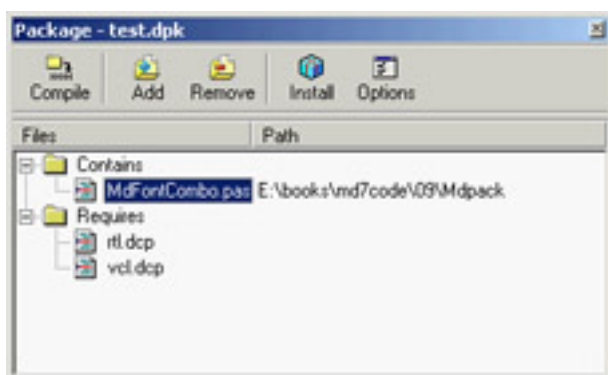
```
procedure TMdFontCombo.Change;  
begin  
    // assign the font to the owner form  
    if FChangeFormFont and Assigned (Owner)  
        and (Owner is TForm)  
    then  
        TForm (Owner).Font.Name := Text;  
    inherited;  
end;  
  
procedure TMdFontCombo.SetChangeFormFont(const Value: Boolean);  
begin  
    FChangeFormFont := Value;  
    // refresh font  
    if FChangeFormFont then  
        Change;  
end;
```

## Creating a Package

Now you have to install the component in the environment, using a package. For this example, you can either create a new package or use an existing one, such as the default user's package.

In either case, choose the `Component → Install Component` menu command. The resulting dialog box has a page that lets you install the component into an existing package, and a page where you can create a new package. In the latter case, type in a filename and a description for the package. Clicking `OK` opens the Package Editor (see [Figure 9.1](#)), which has two parts:

- The `Contains` list indicates the components included in the package (or, to be more precise, the units defining those components).
- The `Requires` list indicates the packages required by this package. Your package will generally require the `rtl` and `vcl` packages (the main run-time library package and core VCL package), but it might also need the `vcldb` package (which includes most of the database-related classes) if the components of the new package do any database-related operations.



## Figure 9.1: The Package Editor

**Note** Since Delphi 6, package names aren't version specific, even if the compiled packages have a version number in the filename. See the section "[Changing Project and Library Names](#)" in [Chapter 10](#) for more details about how this is achieved technically.

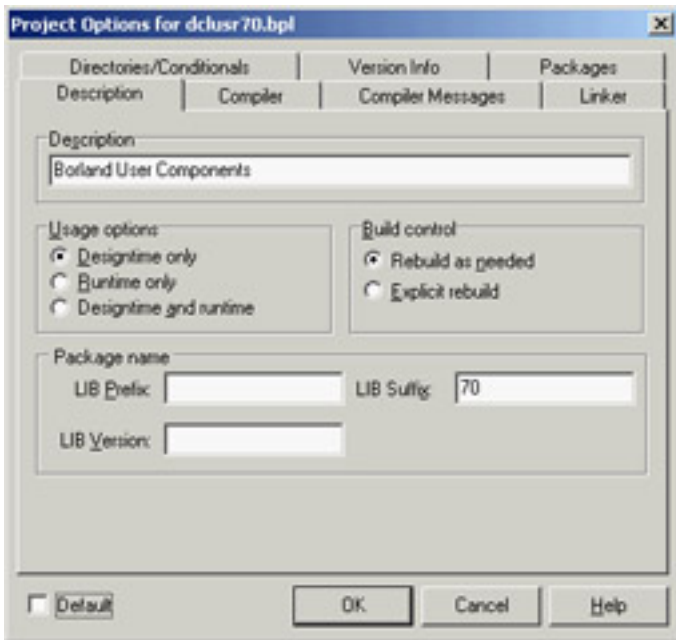
Add the component to the new package you've just defined, and then compile the package and install it (using the two corresponding toolbar buttons in the Package Editor); the new component will immediately appear in the Md page of the Component Palette. The component unit file's `Register` procedure told Delphi where to install the new component. By default, the bitmap used will be the same as that of the parent class, because you haven't provided a custom bitmap (you will do this in later examples). Notice also that if you move the mouse over the new component, Delphi displays as a hint the name of the class without the initial letter *T*.

## What's Behind a Package?

The Package Editor generates the source code for the package project: a special kind of DLL built in Delphi. The package project is saved in a file with the DPK (for Delphi PacKage) extension, displayed if you press the F12 key in the package editor. A typical package project looks like this:

```
package MdPack;  
  
{ $R *.RES }  
{ $ALIGN ON }  
{ $BOOLEVAL OFF }  
{ $DEBUGINFO ON }  
...  
{ $DESCRIPTION 'Mastering Delphi Package' }  
{ $IMPLICITBUILD ON }  
  
requires  
    vcl;  
  
contains  
    MdFontBox in 'MdFontBox.pas';  
  
end.
```

As you can see, Delphi uses specific language keywords for packages. The first is the `package` keyword (similar to the `library` keyword I'll discuss in the [next chapter](#)), which introduces a new package project. Then comes a list of all the compiler options, some of which I've omitted from the listing. Usually the options for a Delphi project are stored in a separate file; packages, by contrast, include all the compiler options directly in their source code. Among the compiler options is a `DESCRIPTION` compiler directive, used to make the package description available to the Delphi environment. After you've installed a new package, its description will appear in the Packages page of the Project Options dialog box, a page you can also activate by selecting the Component → Install Packages menu item. This dialog box is shown in [Figure 9.2](#).



**Figure 9.2:** The Project Options for packages

In addition to common directives like `DESCRIPTION`, there are other compiler directives specific to packages. The most common of these options are easily accessible through the Package Editor's Options button. After this list of options come the `requires` and `contains` keywords, which list the items displayed visually in the two pages of the Package Editor. Again, `requires` lists the packages required by the current package, and `contains` lists the units installed by this package.

Let's consider the technical effect of building a package. Besides the DPK file with the source code, Delphi generates a BPL file with the dynamic link version of the package and a DCP file with the symbol information. In practice, this DCP file is the sum of the symbol information for the DCU files of the units contained in the package.

At design time, Delphi requires both the BPL and DCP files, because the BPL file has the code of the components created on the design form and the symbol information required by the code insight technology. If you link the package dynamically (using it as a run-time package), the DCP file will also be used by the linker, and the BPL file should be shipped along with the application's main executable file. If you instead link the package statically, the linker refers to the DCU files, and you'll need to distribute only the final executable file.

For this reason, as a component designer, you should generally distribute at least the BPL file, the DCP file, and the DCU files of the units contained in the package and any corresponding DFM files, plus a Help file. As an option, of course, you can also make available the source code files of the package units (the PAS files) and of the package itself (the DPK file).

**Warning** By default, Delphi places all the compiled package files (BPL and DCP) not in the package source code's folder but under the `\Projects\BPL` folder. It does this so the IDE can easily locate the files, and the location creates no particular problem. However, when you have to compile a project using components declared in those packages, Delphi may complain that it cannot find the corresponding DCU files, which are stored in the package source code folder. You can solve this problem by indicating the package source code folder in the library path (specified in the Environment Options, which affect all projects) or by indicating it in the search path for the current project (in the Project Options). If you choose the first approach, placing different components and packages in a single folder may result in a real time savings.

---

### Installing the Components Created in This Chapter

Having built your first package, you can now begin using the component you've added to it. Before you do so, however, I should mention that I've extended the MdPack package to include all the components you will build in this chapter, including different versions of the same component. I suggest you install this package. The best approach is to copy it into a directory of your path, so that it will be available both to the Delphi environment and to the programs you build with it. I've collected all the component source code files and the package definition in a

single subdirectory, called MdPack. This allows the Delphi environment (or a specific project) to refer to only one directory when looking for the package's DCU files. As suggested in the earlier warning, I could have collected all the components presented in the book in a single folder on the website; however, I decided that keeping the chapter-based organization was more understandable for readers.

---

Remember, if you compile an application using the packages as run-time libraries, you'll need to install these new libraries on your clients' computers. If you instead compile the programs by statically linking to the units contained in the package, the package library will be required only by the development environment and not by the users of your applications.

## Using the Font Combo Box

Let's create a new Delphi program to test the Font combo box. Move to the Component Palette, select the new component, and add it to a new form. A traditional-looking combo box will appear. However, if you open the Items property editor, you'll see a list of the fonts installed on your computer. To build a simple example, I added a Memo component to the form with some text in it. By leaving the `ChangeFormFont` property on, you don't need to write any other, as you'll see in the example. As an alternative, I could have turned off the property and handled the component's `OnChange` event with code like this:

```
Memo1.Font.Name := MdFontCombo1.Text;
```

The aim of this program is only to test the behavior of the new component you have built. The component is still not very useful—you could have added a couple of lines of code to a form to obtain the same effect—but looking at a couple of components should help you get an idea of what is involved in component building.

## The Component Palette Bitmaps

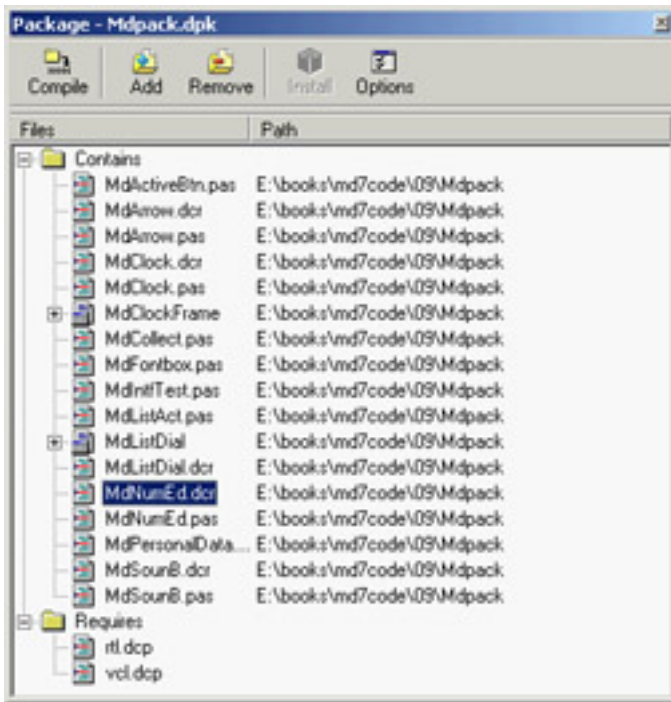
Before installing a component, you can take one further step: define a bitmap for the Component Palette. If you fail to do so, the Palette uses the parent class's bitmap, or a default object's bitmap if the parent class is not an installed component. Defining a new bitmap for the component is easy, once you know the rules. You can create one with the Image Editor included in Delphi by starting a new project and selecting the Delphi Component Resource (DCR) project type.

**Tip** DCR files are standard RES files with a different extension. If you prefer, you can create them with any resource editor—including the Borland Resource Workshop, which is certainly a more powerful tool than the Delphi Image Editor. When you finish creating the resource file, rename the RES file to use a DCR extension.

The resource file should have one (or more) bitmap, each 24×24 pixels. The only important rules refer to naming. In this case, the naming rules are not just a convention; they are required so that the IDE can find the image for a given component class:

- The name of the bitmap resource must match the name of the component, including the initial *T*. In this case, the name of the bitmap resource should be `TMDFONTCOMBO`. The name of the bitmap resource must be uppercase—this is mandatory.
- If you want the Package Editor to recognize and include the resource file, the name of the DCR file must match the name of the compiled unit that defines the component. In this case, the filename should be `MdFontBox.dcr`. If you manually include the resource file via a `$R` directive, you can give it any name you like, as well as use an RES extension and add multiple bitmaps in it.

When the bitmap for the component is ready, you can install the component in Delphi by using the Package Editor's Install Package toolbar button. After this operation, the Contains section of the editor will list both the component's PAS file and the corresponding DCR file. In [Figure 9.3](#), you can see all the files (including the DCR files) of the final version of the MdPack package. If the DCR installation doesn't work properly, you can manually add the `{ $R unitname.dcr }` statement in the package source code.



**Figure 9.3:** The Contains section of the Package Editor shows both the units that are included in the package and the component resource files.

## Creating Compound Components

Components don't exist in isolation. Programmers often use components in conjunction with other components, coding the relationship in one or more event handlers. An alternative approach is to write compound components, which can encapsulate this relationship and make it easy to handle. There are two different types of compound components:

**Internal Components** Created and managed by the main component, which may surface some of their properties and events.

**External Components** Connected using properties. Such a compound component automates the interaction of two separate components, which can be on the same or a different form or designer.

In both cases, development follows some standard rules.

A third, less-explored alternative, involves the development of component containers, which interact with the child controls. This is a more advanced topic, and I won't explore it here.

### Internal Components

The next component I will focus on is a digital clock. This example has some interesting features. First, it embeds a component (a `Timer`) in another component; second, it shows the live-data approach: you'll be able to see a dynamic behavior (the clock time being updated) even at design time, as it happens, for example, with data-aware components.

**Note** The first feature has become more relevant since Delphi 6, because the Object Inspector now allows you to expose properties of subcomponents directly. As a result, the example presented in this section has been modified (and simplified) compared to the Delphi 5 edition of the book. I'll mention the differences, when relevant.

The digital clock will provide some text output, so I considered inheriting from the `TLabel` class. However, doing so would allow a user to change the label's caption—that is, the text of the clock. To avoid this problem, I used the `TCustomLabel` component as the parent class. A `TCustomLabel` object has the same capabilities as a `TLabel` object, but few published properties. In other words, a class that inherits from `TCustomLabel` can decide which properties should be available and which should remain hidden.

**Note** Most of the Delphi components, particularly the Windows-based ones, have a `TCustomXxx` base class, which implements the entire functionality but exposes only a limited set of properties. Inheriting from these base classes is the standard way to expose only some of the properties of a component in a customized version. You cannot hide public or published properties of a base class, unless you hide them by defining a new property with the same name in the descendant class.

With past versions of Delphi, the component had to define a new property, `Active`, wrapping the `Enabled` property of the `Timer`. A *wrapper* property means that the get and set methods of this property read and write the value of the *wrapped* property, which belongs to an internal component (a wrapper property generally has no local data). In this specific case, the code looks like this:

```
function TMdClock.GetActive: Boolean;
begin
    Result := FTimer.Enabled;
end;

procedure TMdClock.SetActive (Value: Boolean);
begin
    FTimer.Enabled := Value;
end;
```

## Publishing Subcomponents

Beginning with Delphi 6, you can expose the entire subcomponent (the timer) in a property of its own that will be regularly expanded by the Object Inspector, allowing a user to set each of its subproperties and even to handle its events.

Here is the full type declaration for the `TMdClock` component, with the subcomponent declared in the private data and exposed as a published property (in the last line):

```
type
  TMdClock = class (TCustomLabel)
  private
    FTimer: TTimer;
  protected
    procedure UpdateClock (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
  published
    property Align;
    property Alignment;
    property Color;
    property Font;
    property ParentColor;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property Transparent;
    property Visible;
    property Timer: TTimer read FTimer;
  end;
```

The `Timer` property is read-only, because I don't want users to select another value for this component in the Object Inspector (or detach the component by clearing the value of this property). Developing sets of subcomponents that can be used alternately is certainly possible, but adding write support for this property in a safe way is far from trivial (considering that the users of your component might not be expert Delphi programmers). So, I suggest you stick with read-only properties for subcomponents.

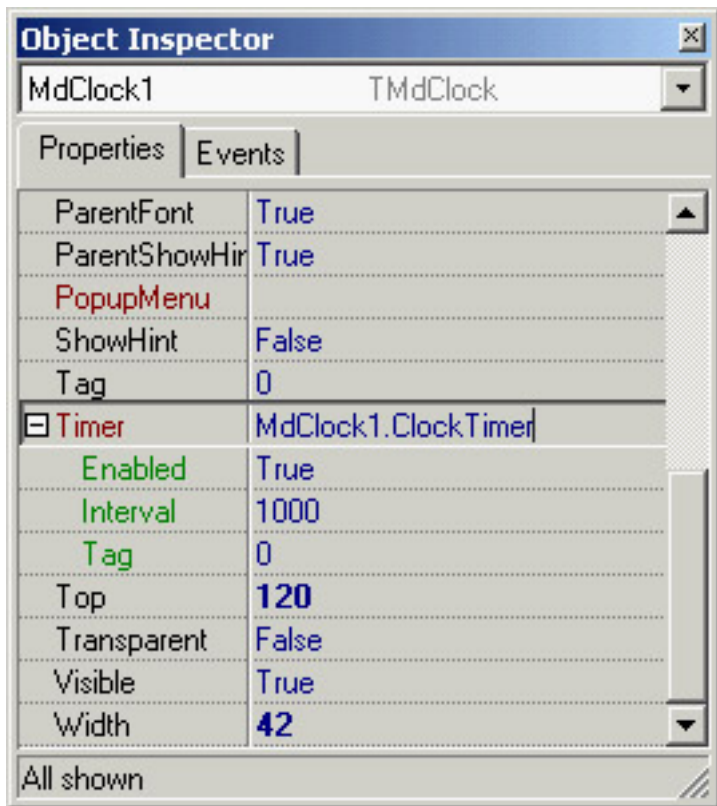
To create the `Timer`, you must override the clock component's constructor. The `Create` method calls the corresponding method of the base class and creates the `Timer` object, installing a handler for its `OnTimer` event:

```
constructor TMdClock.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  // create the internal timer object
  FTimer := TTimer.Create (Self);

  FTimer.Name := 'ClockTimer';
  FTimer.OnTimer := UpdateClock;
  FTimer.Enabled := True;
  FTimer.SetSubComponent (True);
end;
```

The code gives the component a name for display in the Object Inspector (see [Figure 9.4](#)) and calls the specific `SetSubComponent` method. You don't need a destructor; the `FTimer` object has the `TMdClock` component as owner (as indicated by the parameter of its `Create` constructor), so it will be destroyed automatically when the clock component is destroyed.





**Figure 9.4:** The Object Inspector can automatically expand sub-components, showing their properties, as in the case of the `Timer` property of the `TMdClock` component.

**Note** In the previous code, the call to the `SetSubComponent` method sets an internal flag that's saved in the `ComponentStyle` property. The flag (`csSubComponent`) affects the streaming system, allowing the subcomponent and its properties to be saved in the DFM file. The streaming system by default ignores components that are not owned by the form.

The key piece of the component's code is the `UpdateClock` procedure, which is just one statement:

```
procedure TMdLabelClock.UpdateClock (Sender: TObject);
begin
    // set the current time as caption
    Caption := TimeToStr (Time);
end;
```

This method uses `Caption`, which is an unpublished property, so that a user of the component cannot modify it in the Object Inspector. This statement displays the current time continuously, because the method is connected to the Timer's `OnTimer` event.

**Note** Events of subcomponents can be edited in the Object Inspector, so that a user can handle them. If you handle the event internally, as I did in `TMdLabelClock`, a user can override the behavior by handling the event, in this case `OnTimer`. In general, the solution is to define a derived class for the internal component, overriding its virtual methods, such as the `TTimer` class's `Timer` method. In this case, though, this technique won't work, because Delphi activates the timer only if an event handler is attached to it. If you override the virtual method and do not provide the event handler (as would be correct in a subcomponent), the timer won't work.

## External Components

When a component refers to an *external* component, it doesn't create this component itself (which is why this is called *external*). It is the programmer using the components that creates both of them separately (for example dragging them to a form from the Components Palette) and connects the two components using one of their properties. So we can



say that a property of a component refers to an externally linked component. This property must be of a class type that inherits from TComponent.

To demonstrate, I've built a nonvisual component that can display data about a person on a label and refresh the data automatically. The component has these published properties:

```
type
  TMdPersonalData = class(TComponent)
    ...
  published
    property FirstName: string read FFirstName write SetFirstName;
    property LastName: string read FLastName write SetLastName;
    property Age: Integer read FAge write SetAge;
    property Description: string read GetDescription;
    property OutLabel: TLabel read FLabel write SetLabel;
  end;
```

There is some basic data plus a read-only Description property that returns all the information at once. The OutLabel property is connected with a local private field called FLabel. In the component's code, I've used this external label by means of the internal FLabel reference, as in the following:

```
procedure TMdPersonalData.UpdateLabel;
begin
  if Assigned (FLabel) then
    FLabel.Caption := Description;
end;
```

This UpdateLabel method is triggered every time one of the other properties changes (as you can see at design time in [Figure 9.5](#)), as shown here:

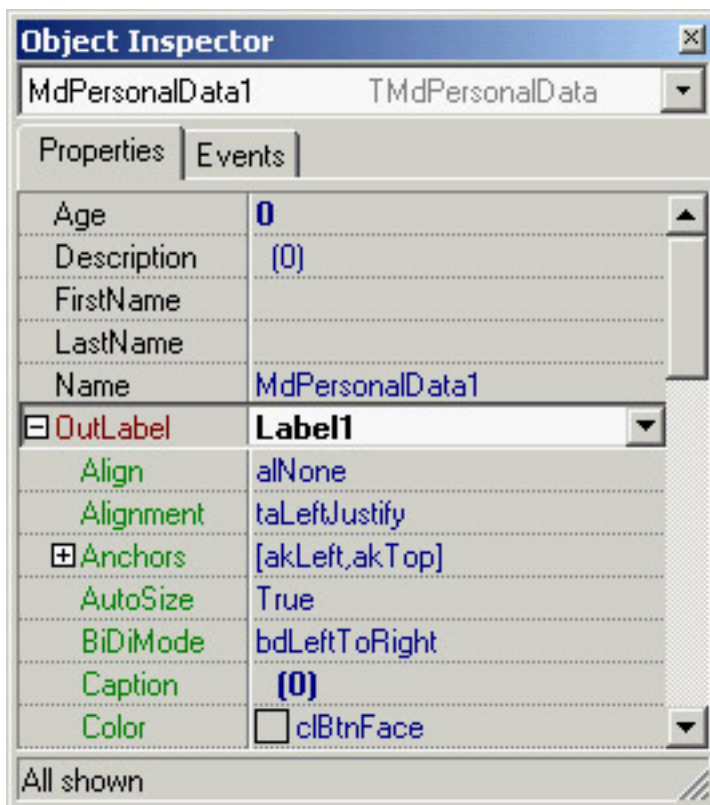


Figure 9.5: A component referencing an external label at design time

```
procedure TMdPersonalData.SetFirstName(const Value: string);
begin
```

```

if FFirstName <> Value then
begin
    FFirstName := Value;
    UpdateLabel;
end;
end;

```

Of course, you cannot use the label if it is not assigned; hence the need for the initial test. However, this test doesn't guarantee the label won't be used after it is destroyed (either at run time or at design time). When you write a component with a reference to an external component you need to override the `Notification` method in the component you are developing (the one with the external reference). This method is fired when a sibling component (one having the same owner) is created or destroyed. Consider the case of the `TMdPersonalData` class that receives the notification of the destruction (`opRemove`) of the `Label` component:

```

procedure TMdPersonalData.Notification(
    AComponent: TComponent; Operation: TOperation);
begin
    inherited;
    if (AComponent = FLabel) and (Operation = opRemove) then
        FLabel := nil;
end;

```

This code is enough to avoid problems with components in the same form or designer (such as a data module), because when a component is destroyed, its owner notifies all the other components it owns (the siblings of the one being destroyed). To account for components connected across forms or data modules, however, you need to perform an extra step. Every component has an internal notification list of one or more components it must notify about its destruction. Your component can add itself to the notification list of components hooked to it (in this case, the label) by calling its `FreeNotification` method. So, even if the externally referenced label is on a different form, it will tell the component it is being destroyed by firing the `Notification` method (which is already handled and doesn't need to be updated):

```

procedure TMdPersonalData.SetLabel(const Value: TLabel);
begin
    if FLabel <> Value then
        begin
            FLabel := Value;
            if FLabel <> nil then
                begin
                    UpdateLabel;
                    FLabel.FreeNotification (Self);
                end;
            end;
        end;
end;

```

**Tip** You can also use the opposite notification (`opInsert`) to hook up components automatically as they are added to the same form or designer. I'm not sure why this technique is so rarely used, because it's helpful in many common situations. It is true that it makes more sense to build specific property and component editors to support design-time operations, rather than embed special code within the components.

## Referring to Components with Interfaces

When referring to external components, we've traditionally been limited to a subhierarchy. For example, the component built in the [previous section](#) can refer only to objects of class `TLabel` or classes inheriting from `TLabel`, although it would make sense to also be able to output the data to other components. Delphi 6 added support for an interesting feature that has the potential to revolutionize some areas of the VCL: interface-type component references.

**Note** This feature is used sparingly in Delphi 7. As it is probably too late to update Delphi's data-aware components architecture using interfaces, all we can hope is that this will be used to express other future complex relationships within the library.

If you have components supporting a given interface (even if they are not part of the same subhierarchy), you can declare a property with an interface type and assign any of those components to it. For example, suppose you have a nonvisual component attached to a control for its output, similar to what I did in the [previous section](#). I used a traditional approach and hooked the component to a label, but you can now define an interface as follows:

```
type
  IMdViewer = interface
    ['{9766860D-8E4A-4254-9843-59B98FEE6C54}']
    procedure View (const str: string);
end;
```

A component can use this *viewer* interface to provide output to another control (of any type). [Listing 9.2](#) shows how to declare a component that uses this interface to refer to an external component.

### **Listing 9.2: A Component that Refers to an External Component Using an Interface**

---

```
type
  TMDIntfTest = class(TComponent)
  private
    FViewer: IViewer;
    FText: string;
    procedure SetViewer(const Value: IViewer);
    procedure SetText(const Value: string);
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  published
    property Viewer: IViewer read FViewer write SetViewer;
    property Text: string read FText write SetText;
end;

{ TMDIntfTest }

procedure TMDIntfTest.Notification(AComponent: TComponent;
  Operation: TOperation);
var
  intf: IMdViewer;
begin
  inherited;
  if (Operation = opRemove) and
    (Supports (AComponent, IMdViewer, intf)) and (intf = FViewer) then
  begin
    FViewer := nil;
  end;
end;

procedure TMDIntfTest.SetText(const Value: string);
begin
  FText := Value;
  if Assigned (FViewer) then
    FViewer.View(FText);
end;

procedure TMDIntfTest.SetViewer(const Value: IMdViewer);
```

```

var
  iComp: IInterfaceComponentReference;
begin
  if FViewer <> Value then
  begin
    FViewer := Value;
    FViewer.View(FText);
    if Supports (FViewer, IInterfaceComponentReference, iComp) then
      iComp.GetComponent.FreeNotification(Self);
  end;
end;

```

---

The use of an interface implies two relevant differences, compared to the traditional use of a class type to refer to an external component. First, in the `Notification` method, you must extract the interface from the component passed as a parameter and compare it to the interface you already hold. Second, to call the `FreeNotification` method, you must see whether the object passed as the parameter supports the `IInterfaceComponentReference` interface. This is declared in the `TComponent` class and provides a way to refer back to the component (`GetComponent`) and call its methods. Without this help you would have to add a similar method to your custom interface, because when you extract an interface from an object, there is no automatic way to refer back to the object.

Now that you have a component with an interface property, you can assign to it any component (from any portion of the VCL hierarchy) by adding the `IViewer` interface to it and implementing the `View` method. Here is an example:

```

type
  TViewerLabel = class (TLabel, IViewer)
  public
    procedure View(str: String);
  end;

procedure TViewerLabel.View(const str: String);
begin
  Caption := str;
end;

```

---

## Building Compound Components with Frames

Instead of building the compound component in code and hooking up the timer event manually, you can obtain a similar effect by using a frame. Frames make the development of compound components with custom event handlers a visual operation, and thus simpler. You can share this frame by adding it to the Repository or by creating a template using the Add to Palette command on the frame's shortcut menu.

As an alternative, you might want to share the frame by placing it in a package and registering it as a component. Technically, this technique is not difficult: You add a `Register` procedure to the frame's unit, add the unit to a package, and build it. The new component/frame appears in the Component Palette like any other component. When you place this component/frame on a form, you'll see its subcomponents. You cannot select these subcomponents with a mouse click in the Form Designer, but you can do so in the Object TreeView. However, any change you make to these components at design time will be lost when you run the program or save and reload the form, because the changes to those subcomponents aren't streamed, unlike what happens with standard frames you place inside a form).

If this is not what you expect, I've found a *reasonable* way to use frames in packages, demonstrated by the `MdFramedClock` component (part of the examples for this chapter on the Sybex website). The components owned by the frame are turned into subcomponents by calling the `SetSubComponent` method. I also exposed the internal components with properties, even though this step isn't compulsory (they can be selected in the Object TreeView). Here is the component's declaration and the code for its methods:

```

type

```

```

TMdFramedClock = class(TFrame)
  Label1: TLabel;
  Timer1: TTimer;
  Bevel1: TBevel;
  procedure Timer1Timer(Sender: TObject);
public
  constructor Create(AOwner: TComponent); override;
published
  property SubLabel: TLabel read Label1;
  property SubTimer: TTimer read Timer1;
end;constructor TMdFramedClock.Create(AOwner: TComponent);
begin
  inherited;
  Timer1.SetSubComponent (True);
  Label1.SetSubComponent (True);
end;procedure TMdFramedClock.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr (Time);
end;

```

In contrast to the clock component built earlier, there is no need to set up the properties of the timer, or to connect the timer event to its handler function manually, because this is done visually and saved in the DFM file of the frame. Notice also that I haven't exposed the Bevel component (I haven't called `SetSubComponent` on it). I did this on purpose so you can experiment with this fault behavior: try editing it at design time and see that all the changes are lost, as I mentioned earlier.

After you install this frame/component, you can use it in any application. In this case, as soon as you drop the frame on the form, the timer will begin to update the label with the current time. However, you can still handle its `OnTimer` event, and the Delphi IDE (recognizing that the component is in a frame) will create a method with this predefined code:

```

procedure TForm1.MdFramedClock1Timer1Timer(Sender: TObject);
begin
  MdFramedClock1.Timer1Timer(Sender);
end;

```

As soon as this timer is connected (even at design time) the live clock will stop, because its original event handler is disconnected. After you compile and run the program, however, the original behavior will be restored (at least, if you don't delete the previous line); your extra custom code will be executed as well. This behavior is exactly what you expect from frames. You can find a complete demo of the use of this frame/ component in the `FrameClock` example.

This approach is still far from linear. It is much better than in past versions of Delphi, where frames inside packages were unusable, but it isn't worth the effort. If you work alone or with a small team, it's better to use plain frames stored in the Repository. In larger organizations or to distribute your frames to a larger audience, most people will prefer to build their components the traditional way, without trying to use frames. I hope that Borland will address more complete support for the visual development of packaged components based on frames.

---

## A Complex Graphical Component

In this section, I'll demonstrate how to build a graphical Arrow component. You can use such a component to indicate a flow of information or an action. This component is quite complex, so I'll show you the various steps instead of looking directly at the complete source code. The component I've added to the MdPack package is the final version of this process, which demonstrates several important concepts:

- The definition of new enumerated properties, based on custom enumerated data types.
- The implementation of the component's `Paint` method, which provides its user interface and should be generic enough to accommodate all the possible values of the various properties, including its `Width` and `Height`. The `Paint` method plays a substantial role in this graphical component.
- The use of properties of `TPersistent`-derived classes, such as `TPen` and `TBrush`, as well as the issues related to their creation and destruction and to handling their `OnChange` events internally in your component.
- The definition of a custom event handler for the component, which responds to user input (in this case, a double-click on the point of the arrow). This requires direct handling of Windows messages and the use of the Windows API for graphic regions.
- The registration of properties in Object Inspector categories and the definition of a custom category.

### Defining an Enumerated Property

After generating the new component with the Component Wizard and choosing `TGraphicControl` as the parent class, you can begin to customize the component. The arrow can point in any of four directions: up, down, left, or right. An enumerated type expresses these choices:

```
type
  TMdArrowDir = (adUp, adRight, adDown, adLeft);
```

This enumerated type defines a private data member of the component, a parameter of the procedure used to change it, and the type of the corresponding property.

The `ArrowHeight` property determines the size of the arrowhead, and the `Filled` property specifies whether to fill the arrowhead with color:

```
type
  TMdArrow = class (TGraphicControl)
  private
    FDirection: TMdArrowDir;
    FArrowHeight: Integer;
    FFilled: Boolean;
    procedure SetDirection (Value: TMd4ArrowDir);
    procedure SetArrowHeight (Value: Integer);
    procedure SetFilled (Value: Boolean);
  published
    property Width default 50;
    property Height default 20;
    property Direction: TMd4ArrowDir
      read FDirection write SetDirection default adRight;
    property ArrowHeight: Integer
      read FArrowHeight write SetArrowHeight default 10;
    property Filled: Boolean read FFilled write SetFilled default False;
```

**Note** A graphic control has no default size, so when you place it in a form, its size will be a single pixel. For this reason, it is important to add a default value for the *Width* and *Height* properties and set the class fields to the default property values in the class constructor.

The three custom properties are read directly from the corresponding field and are written using three *Set* methods, all having the same standard structure:

```
procedure TMDArrow.SetDirection (Value: TMDArrowDir);  
begin  
  if FDirection <> Value then  
    begin  
      FDirection := Value;  
      ComputePoints;  
      Invalidate;  
    end;  
end;
```

Notice that you ask the system to repaint the component (by calling *Invalidate*) only if the property is really changing its value and after calling the *ComputePoints* method, which computes the triangle delimiting the arrowhead. Otherwise, the code is skipped and the method ends immediately. This code structure is common, and you will use it for most of the *Set* procedures of properties.

You must also remember to set the properties' default values in the component's constructor:

```
constructor TMDArrow.Create (AOwner: TComponent);  
begin  
  // call the parent constructor  
  inherited Create (AOwner);  
  // set the default values  
  FDirection := adRight;  
  Width := 50;  
  Height := 20;  
  FArrowHeight := 10;  
  FFilled := False;
```

As mentioned before, the default value specified in the property declaration is used only to determine whether to save the property's value to disk. The *Create* constructor is defined in the public section of the new component's type definition, and the constructor is marked by the *override* keyword, as it replaces the virtual *Create* constructor of *TComponent*. It is fundamental to remember the *override* specifier; otherwise, when Delphi creates a new component of this class, it will call the base class's constructor, rather than the one you've written for your derived class.

---

## Property-Naming Conventions

In the definition of the *Arrow* component, notice the use of several naming conventions for properties, access methods, and fields. Here is a summary:

- A property should have a meaningful and readable name.
- When a private data field is used to hold the value of a property, the field should be named with an uppercase *F* (field) at the beginning, followed by the name of the corresponding property.
- When a procedure is used to change the value of the property, the function should have the word *Set* at the beginning, followed by the name of the corresponding property.
- A corresponding function used to read the property should have the word *Get* at the beginning, again followed by the property name.

These are just guidelines to make your programs more readable. The compiler doesn't enforce them. These



conventions are described in the *Delphi Component Writers' Guide* and are followed by Delphi's class completion mechanism.

---

## Writing the *Paint* Method

Drawing the arrow in the various directions and with the various styles requires a fair amount of code. To perform custom painting, you override the `Paint` method and use the protected `Canvas` property.

Instead of computing the position of the arrowhead points in drawing code that will be executed often, I've written a separate function to compute the arrowhead area and store it in an array of points defined among the private fields of the component:

```
FArrowPoints: array [0..3] of TPoint;
```

These points are determined by the `ComputePoints` private method, which is called every time a component property changes. Here is an excerpt of its code:

```
procedure TMDArrow.ComputePoints;
var
  XCenter, YCenter: Integer;
begin
  // compute the points of the arrowhead
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  case FDirection of
    adUp: begin
      FArrowPoints [0] := Point (0, FArrowHeight);
      FArrowPoints [1] := Point (XCenter, 0);
      FArrowPoints [2] := Point (Width-1, FArrowHeight);
    end;
  // and so on for the other directions
```

The code computes the center of the component area (dividing the `Height` and `Width` properties by two) and then uses the center to determine the position of the arrowhead. In addition to changing the direction or other properties, you need to refresh the position of the arrowhead when the size of the component changes. You can override the `SetBounds` method of the component, which is called by VCL every time the `Left`, `Top`, `Width`, and `Height` properties of a component change:

```
procedure TMDArrow.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited SetBounds (ALeft, ATop, AWidth, AHeight);
  ComputePoints;
end;
```

Once the component knows the position of the arrowhead, its painting code becomes simpler. Here is an excerpt of the `Paint` method:

```
procedure TMDArrow.Paint;
var
  XCenter, YCenter: Integer;
begin
  // compute the center
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;

  // draw the arrow line
  case FDirection of
```



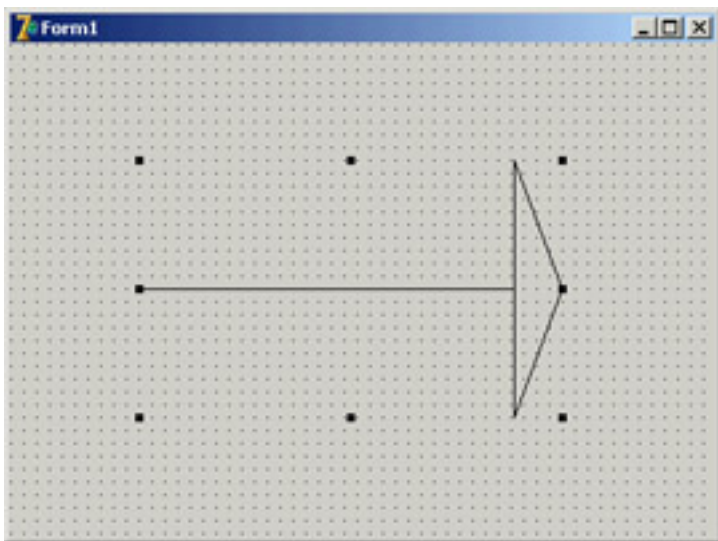
```

adUp: begin
    Canvas.MoveTo (XCenter, Height-1);
    Canvas.LineTo (XCenter, FArrowHeight);
end;
// and so on for the other directions
end;

// draw the arrow point, eventually filling it
if FFilled then
    Canvas.Polygon (FArrowPoints)
else
    Canvas.PolyLine (FArrowPoints);
end;

```

You can see an example of the output of this component in [Figure 9.6](#).



**Figure 9.6:** The output of the Arrow component

## Adding *TPersistent* Properties

To make the output of the component more flexible, I've added to it two new properties, `Pen` and `Brush`, defined with a class type (a `TPersistent` data type, which defines objects that Delphi can automatically stream). These properties are a little more complex to handle, because the component now has to create and destroy these internal objects. This time, however, you also export the internal objects using properties, so that users can directly change these internal objects from the Object Inspector. To update the component when these subobjects change, you'll also need to handle their internal `OnChange` property. Here is the definition of the `Pen` property and the other changes to the definition of the component class (the code for the `Brush` property is similar):

```

type
    TMdArrow = class (TGraphicControl)
    private
        FPen: TPen;
        ...
        procedure SetPen (Value: TPen);
        procedure RepaintRequest (Sender: TObject);
    published
        property Pen: TPen read FPen write SetPen;
    end;

```

You first create the object in the constructor and set its `OnChange` event handler:

```

constructor TMdArrow.Create (AOwner: TComponent);
begin
    ...
    // create the pen and the brush
    FPen := TPen.Create;
    // set a handler for the OnChange event
    FPen.OnChange := RepaintRequest;
end;

```

These `OnChange` events are fired when one of the properties of the pen changes; all you have to do is to ask the system to repaint your component:

```

procedure TMdArrow.RepaintRequest (Sender: TObject);
begin
    Invalidate;
end;

```

You must also add a destructor to the component, to remove the graphical object from memory (and free its system resources). All the destructor has to do is call the Pen object's `Free` method.

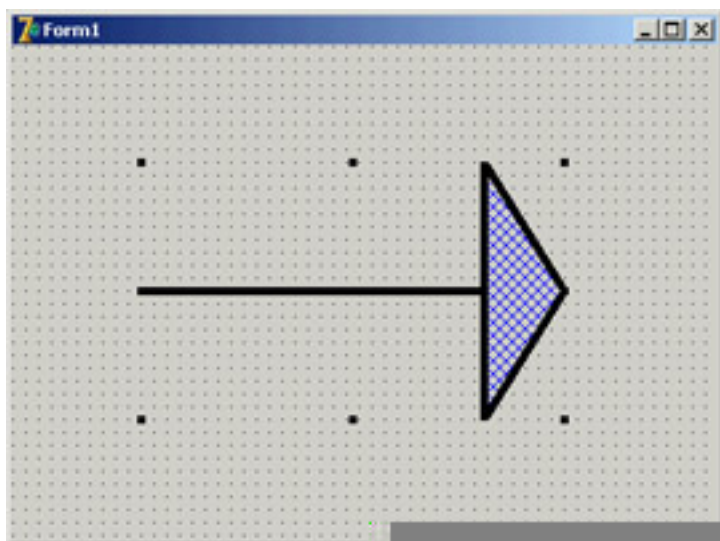
A property related to persistent objects requires special handling: Instead of copying the pointer to the object, you have to copy the internal data of the object passed as a parameter. The standard `:=` operation copies the pointer, so in this case you have to use the `Assign` method:

```

procedure TMdArrow.SetPen (Value: TPen);
begin
    FPen.Assign(Value);
    Invalidate;
end;

```

Many `TPersistent` classes have an `Assign` method you should use when you need to update the data of these objects. Now, to use the pen for the drawing, you must modify the `Paint` method, setting the corresponding property of the component `Canvas` to the value of the internal object before drawing a line (see the example of the component's new output in [Figure 9.7](#)):



**Figure 9.7:** The output of the Arrow component with a thick pen and a special hatch brush

```

procedure TMdArrow.Paint;
begin
    // use the current pen
    Canvas.Pen := FPen;

```

As the `Canvas` uses a setter routine to `Assign` the pen object, you're not simply storing a reference to the pen in a field of the `Canvas`, but you are copying all of its data. This means that you can freely destroy the local `Pen` object (`FPen`) and that modifying `FPen` won't affect the canvas until `Paint` is called and the code above is executed again.

## Defining a Custom Event

To complete the development of the `Arrow` component, let's add a custom event. Most of the time, new components use the events of their parent classes. For example, in this component, I've made some standard events available by redeclaring them in the published section of the class:

```
type
  TMDArrow = class (TGraphicControl)
  published
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
```

Thanks to this declaration, the events (originally declared in a parent class) will be available in the Object Inspector when the component is installed.

Sometimes, however, a component requires a custom event. To define a new event, you first need to ensure that there is already a method pointer type suitable for use by the event; if not, you need to define a new event type. This type is a method pointer type (see [Chapter 5](#), "Visual Controls," for details). In both cases, you need to add to the class a field of the event's type: here is the definition I've added in the private section of the `TMDArrow` class:

```
FArrowDbClick: TNotifyEvent;
```

I've used the `TNotifyEvent` type, which has only a `Sender` parameter and is used by Delphi for many events, including `OnClick` and `OnDbClick` events. Using this field I've defined a published property, with direct access to the field:

```
property OnArrowDbClick: TNotifyEvent
  read FArrowDbClick write FArrowDbClick;
```

(Notice again the standard naming convention, with event names starting with *On*.) The `fArrowDbClick` method pointer is activated (executing the corresponding function) inside the specific `ArrowDbClick` dynamic method. This happens only if an event handler has been specified in the program that uses the component:

```
procedure TMDArrow.ArrowDbClick;
begin
  if Assigned (FArrowDbClick) then
    FArrowDbClick (Self);
end;
```

**Tip** The use of *Self* as parameter of the invocation of the event handler method ensures that when the method is called its *Sender* parameter would actually refer to the object that fired the event, which you generally expect as a component user.

## Using Low-Level Windows API Calls

The `fArrowDbClick` method is defined in the protected section of the type definition to allow future descendant classes to both call and change it. Basically, this method is called by the handler of the `WM_LBUTTONDOWN` Windows message, but only if the double-click took place inside the arrow's point. To test this condition, you can use some of the Windows API's region functions.

**Note** A *region* is a screen area enclosed by any shape. For example, you can build a polygonal region using the three vertices of the arrow-point triangle. The only problem is that to fill the surface properly, you must define an array of *TPoints* in a clockwise direction (see the description of the *CreatePolygonalRgn* in the Windows API Help for the details of this strange approach). That's what I did in the *ComputePoints* method.

Once you have defined a region, you can use the *PtInRegion* API call to test whether the point where the double-click occurred is inside the region. The complete source code for this procedure is as follows:

```
procedure TMDArrow.WMLButtonDblClk (  
    var Msg: TWMLButtonDblClk); // message wm_LButtonDblClk;  
var  
    HRegion: HRgn;  
begin  
    // perform default handling  
    inherited;  
  
    // compute the arrowhead region  
    HRegion := CreatePolygonRgn (FArrowPoints, 3, WINDING);  
    try // check whether the click took place in the region  
        if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then  
            ArrowDblClick;  
    finally  
        DeleteObject (HRegion);  
    end;  
end;
```

## The CLX Version: Calling Qt Native Functions

The previous code won't be portable to Linux and makes no sense for the CLX/Qt version of the component. If you want to build a similar component for the CLX class library, you can replace the Win32 API calls with direct (low-level) calls to the Qt layer, creating an object of the *QRegion* class, as in the following listing:

```
procedure TMDArrow.DblClick;  
var  
    HRegion: QRegionH;  
    MousePoint: TPoint;  
begin  
    // perform default handling  
    inherited;  
  
    // compute the arrow head region  
    HRegion := QRegion_create (PPointArray(FArrowPoints), True);  
    try  
        // get the current mouse position  
        GetCursorPos (MousePoint);  
        MousePoint := ScreenToClient(MousePoint);  
  
        // check whether the click took place in the region  
        if QRegion_contains(HRegion, PPoint(@MousePoint)) then  
            ArrowDblClick;  
    finally  
        QRegion_destroy(HRegion);  
    end;  
end;
```

## Registering Property Categories

You've added to this component some custom properties and a new event. If you arrange the properties in the Object Inspector by category, all the new elements will appear in the generic Miscellaneous category. Of course, this is far

from ideal, but you can easily register the new properties in one of the available categories.

You can register a property (or an event) in a category by calling one of the four overloaded versions of the `RegisterPropertyInCategory` function, defined in the `DesignIntf` unit. When calling this function, you indicate the name of the category, and you can specify the property name, its type, or the property name and the component it belongs to. For example, you can add the following lines to the `Register` procedure of the unit to register the `OnArrowDbClick` event in the `Input` category and the `Filled` property in the `Visual` category:

```
uses
  DesignIntf;

procedure Register;
begin
  RegisterPropertyInCategory ('Input', TMDArrow, 'OnArrowDbClick');
  RegisterPropertyInCategory ('Visual', TMDArrow, 'Filled');
end;
```

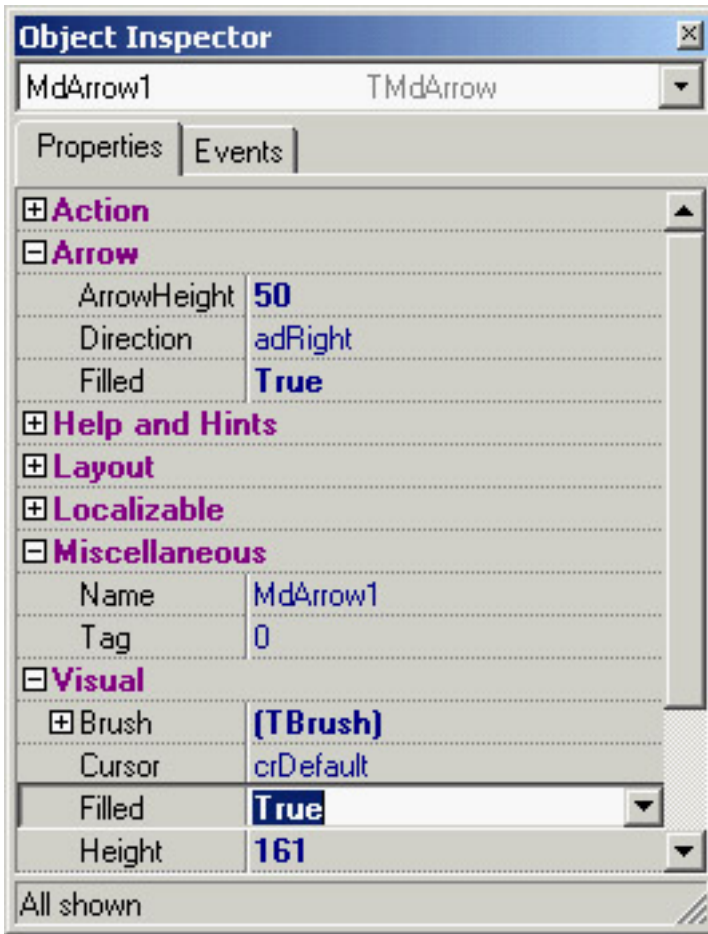
The first parameter is a string indicating the category name—a much simpler solution than the original Delphi 5 approach of using category classes. You can define a new category in a straightforward manner by passing its name as the first parameter of the `RegisterPropertyInCategory` function:

```
RegisterPropertyInCategory ('Arrow', TMDArrow, 'Direction');
RegisterPropertyInCategory ('Arrow', TMDArrow, 'ArrowHeight');
```

Creating a new category for the specific properties of your component can make it much simpler for a user to locate its specific features. Notice, though, that because you rely on the `DesignIntf` unit, you should compile the unit containing these registrations in a design-time package, not a run-time package (the required `DesignIde` unit cannot be distributed). For this reason, I've written this code in a separate unit from the one defining the component and added the new unit (`MdArrReg`) to the package `MdDesPk`, including all the design-time-only units; this approach is discussed later, in the section "[Installing the Property Editor](#)."

**Warning** It's debatable whether using a category for the specific properties of a component is a good idea. On one hand, a user of the component can easily spot specific properties. At the same time, some of the new properties may not pertain to any of the existing categories. On the other hand, categories can be overused. If every component introduces new categories, users may be confused. You also face the risk of having as many categories as there are properties.

Notice that my code registers the `Filled` property in two different categories. This is not a problem, because the same property can show up multiple times in the Object Inspector under different groups, as you can see in [Figure 9.8](#). To test the arrow component, I've written the `ArrowDemo` program, which allows you to modify most of its properties at run time. This type of test is important after you have written a component or while you are writing it.



**Figure 9.8:** The Arrow component defines a custom property category, Arrow, as you can see in the Object Inspector. Notice that properties can be visible in multiple section, such as the Filled property in this case.

**Note** The *Localizable* property category has a special role related to the use of the ITE (Integrated Translation Environment). When a property is part of this category, its value is listed in the Translation Environment as a property that can be translated into another language.

## Customizing Windows Controls

One of the most common ways of customizing existing components is to add predefined behavior to their event handlers. Every time you need to attach the same event handler to components of different forms, you should consider adding the event code to a descendant class of the component. An obvious example is edit boxes that accept only numeric input. Instead of attaching a common `OnChar` event handler to each edit box, you can define a new component.

However, this component won't handle the event; events are for component users only. Instead, the component can either handle the Windows message directly or override a method, often called a *second-level* message handler. The former technique was commonly used in the past, but it makes a component specific to the Windows platform. To create a component that's portable to CLX and Linux—and, in the future, to the .NET architecture—you should avoid low-level Windows messages and instead override virtual methods of the base component and control classes.

**Note** When most VCL components handle a Windows message, they call a second-level message handler (usually a *dynamic* method), instead of executing code directly in the message-response method. This approach makes it easier for you to customize the component in a derived class. Typically, a second-level handler will do its own work and then call any event handler the component user has assigned. So, you should always call *inherited* to let the component fire the event as expected.

In addition to portability, there are other reasons why overriding existing second-level handlers is generally a better approach than handling straight Windows messages. First, this technique is more sound from an object-oriented perspective. Instead of duplicating the message-response code from the base class and then customizing it, you're overriding a virtual method call that the VCL designers planned for you to override. Second, if someone needs to derive another class from one of your component classes, you should make it as easy for them to customize as possible, and overriding second-level handlers is less likely to induce errors (if only because you're writing less code). For example, I could have written the following numeric edit box control by handling the `wm_Char` system message:

```
type
  TMdNumEdit = class (TCustomEdit)
  public
    procedure WmChar (var Msg: TWmChar); message wm_Char;
```

However, the code is more portable if I override the `KeyPress` method, as I've done in the code of the next component. (In a later example I'll have to handle custom Windows messages, because there is no corresponding method to override.)

## The Numeric Edit Box

To customize an edit box component to restrict the input it will accept, all you need to do is override its `KeyPress` method, which is called when the component receives the `wm_Char` Windows message. Here is the code for the `TMdNumEdit` class:

```
type
  TMdNumEdit = class (TCustomEdit)
  private
    FInputError: TNotifyEvent;
  protected
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    procedure KeyPress(var Key: Char); override;
  public
    constructor Create (Owner: TComponent); override;
  published
    property OnInputError: TNotifyEvent read FInputError write FInputError;
    property Value: Integer read GetValue write SetValue default 0;
    property AutoSelect;
```



```
property AutoSize;  
// and so on...
```

This component inherits from `TCustomEdit` instead of `TEdit` so that it can hide the `Text` property and surface the `Integer Value` property instead. Notice that you don't create a new field to store this value, because you can use the existing (but now unpublished) `Text` property. To do so, you convert the numeric value to and from a text string. The `TCustomEdit` class (actually, the Windows control it wraps) automatically paints the information from the `Text` property on the surface of the component:

```
function TmNumEdit.GetValue: Integer;  
begin  
    // set to 0 in case of error  
    Result := StrToIntDef (Text, 0);  
end;  
  
procedure TmNumEdit.SetValue (Value: Integer);  
begin  
    Text := IntToStr (Value);  
end;
```

The most important method is the redefined `KeyPress` method, which filters out all the nonnumeric characters and fires a specific event in case of an error:

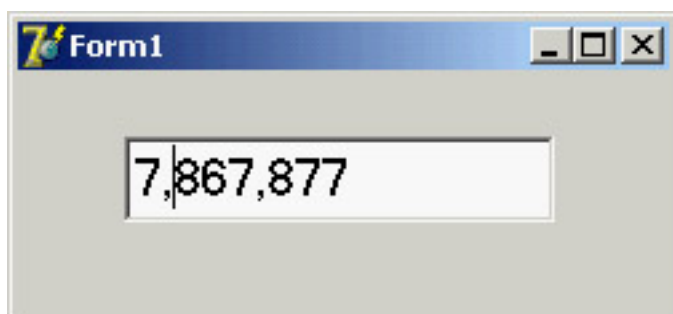
```
procedure TmNumEdit.KeyPress (var Msg: TWmChar);  
begin  
    if not (Key in ['0'..'9']) and not (Key = #8) then  
    begin  
        Key := #0; // pretend that nothing was pressed  
        if Assigned (FInputError) then  
            FInputError (Self);  
    end  
    else  
        inherited;  
end;
```

This method checks each character as the user enters it, testing for numerals and the Backspace key (which has an ASCII value of 8). The user should be able to use Backspace in addition to the system keys (the arrow keys and Del), so you need to check for that value.

Now, place this component on a form, type something in the edit box, and see how it behaves. You might also want to attach a method to the `OnInputError` event to provide feedback to the user when an incorrect key is pressed.

## A Numeric Edit with Thousands Separators

As a further extension of the example, when the user types large numbers (stored internally as floating point numbers, which compared to integers can be larger and have decimal digits) it would be nice for the thousands separators to automatically appear and update themselves as required by the input:





You can do this by overriding the internal `Change` method and formatting the number properly. There are only a couple of small problems to consider. The first is that to format the number you need to have a string containing a number, but the text in the edit box is not a numeric string Delphi recognizes, as it has thousands of separators and cannot be converted to a number directly. I've written a modified version of the `StringToFloat` function, called `StringToFloatSkipping`, to accomplish this conversion.

The second small problem is that if you modify the text in the edit box, the current position of the cursor will be lost. So, you need to save the original cursor position, reformat the number, and then reapply the cursor position—considering that if a separator has been added or removed, the cursor position should change accordingly.

All these considerations are summarized by the following complete code for the `TMdThousandEdit` class:

```
type
  TMdThousandEdit = class (TMdNumEdit)
  public
    procedure Change; override;
  end;

function StringToFloatSkipping (s: string): Extended;
var
  s1: string;
  I: Integer;
begin
  // remove non-numbers
  s1 := '';
  for i := 1 to length (s) do
    if s[i] in ['0'..'9'] then
      s1 := s1 + s[i];
  Result := StrToFloat (s1);
end;

procedure TMdThousandEdit.Change;
var
  CursorPos, // original position of the cursor
  LengthDiff: Integer; // number of new separators (+ or -)
begin
  if Assigned (Parent) then
  begin
    CursorPos := SelStart;
    LengthDiff := Length (Text);
    Text := FormatFloat ('#,###',
      StringToFloatSkipping (Text));
    LengthDiff := Length (Text) - LengthDiff;
    // move the cursor to the proper position
    SelStart := CursorPos + LengthDiff;
  end;
  inherited;
end;
```

## The Sound Button

The next component, `TMdSoundButton`, plays one sound when you press the button and another sound when you release it. The user specifies each sound by modifying two string properties that name the appropriate WAV files for the respective sounds. Once again, you need to intercept some system messages (`wm_LButtonDown` and `wm_LButtonUp`) or override the appropriate *second-level* handler.

Here is the code for the `TMdSoundButton` class, with the two protected methods and the two string properties that identify the sound files, mapped to private fields because you don't need to do anything special when the user changes those properties:

```
type
  TMdSoundButton = class(TButton)
  private
    FSoundUp, FSoundDown: string;
  protected
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  published
    property SoundUp: string read FSoundUp write FSoundUp;
    property SoundDown: string read FSoundDown write FSoundDown;
end;
```

Here is the code for one of the two methods:

```
uses
  MMSystem;

procedure TMdSoundButton.MouseDown(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
  PlaySound (PChar (FSoundDown), 0, snd_Async);
end;
```

Notice that you call the inherited version of the methods *before* you do anything else. For most second-level handlers, this is a good practice, because it ensures that you execute the standard behavior before you execute any custom behavior. Next, notice that you call the `PlaySound` Win32 API function to play the sound. You can use this function (defined in the `MmSystem` unit) to play either WAV files or system sounds, as the `SoundB` example demonstrates. Here is a textual description of the form of this sample program (from the `DFM` file):

```
object MdSoundButton1: TMdSoundButton
  Caption = 'Press'
  SoundUp = 'RestoreUp'
  SoundDown = 'RestoreDown'
end
```

**Note** Selecting a proper value for these sound properties is far from simple. Later in this chapter, I'll show you how to add a property editor to the component to simplify the operation.

## Handling Internal Messages: The Active Button

The Windows interface is evolving toward a new standard, including components that become highlighted as the mouse cursor moves over them. Delphi provides similar support in many of its built-in components. Mimicking this behavior for a button might seem a complex task to accomplish, but it is not. The development of a component can become much simpler once you know which virtual function to override or which message to hook onto.

The next component, the `TMdActiveButton` class, demonstrates this technique by handling some internal Delphi messages to accomplish its task in a simple way. (For information about where these internal Delphi messages come from, see the [next section](#), "Component Messages and Notifications.") The `ActiveButton` component handles the `cm_MouseEnter` and `cm_MouseExit` internal Delphi messages, which are received when the mouse cursor enters or leaves the area corresponding to the component:

```
type
```

```

TMdActiveButton = class (TButton)
protected
  procedure MouseEnter (var Msg: TMessage); message cm_mouseEnter;
  procedure MouseLeave (var Msg: TMessage); message cm_mouseLeave;
end;

```

The code you write for these two methods can do whatever you want. For this example, I've decided to toggle the bold style of the button's font. You can see the effect of moving the mouse over one of these components in [Figure 9.9](#).



**Figure 9.9:** An example of the use of the ActiveButton component

```

procedure TMdActiveButton.MouseEnter (var Msg: TMessage);
begin
  Font.Style := Font.Style + [fsBold];
end;

```

```

procedure TMdActiveButton.MouseLeave (var Msg: TMessage);
begin
  Font.Style := Font.Style - [fsBold];
end;

```

You can add other effects, including enlarging the font, making the button the default, or increasing the button's size a little. The best effects usually involve colors, but you must inherit from the `TBitBtn` class to have this support (`TButton` controls have a fixed color).

## Component Messages and Notifications

To build the `ActiveButton` component, I used two internal Delphi component messages, as indicated by their *cm* prefix. These messages can be quite interesting, as the example highlights, but they are almost completely undocumented by

Borland. There is also a second group of internal Delphi messages, indicated as component notifications and distinguished by their *cn* prefix. I don't have enough space here to discuss each of them or provide a detailed analysis; browse the VCL source code if you want to learn more.

**Warning** This is a rather advanced topic, so feel free to skip this section if you are new to writing Delphi components. Component messages are not documented in the Delphi help file, so I felt it was important to at least list them here.

## Component Messages

A Delphi component passes *component messages* to other components to indicate any change in its state that might affect those components. Most of these messages begin as Windows messages, but some of them are more complex, higher-level translations and not simple remappings. In addition, components send their own messages as well as forwarding those received from Windows. For example, changing a property value or some other characteristic of the component may necessitate telling one or more other components about the change.

You can group these messages into categories:

- Activation and input focus messages are sent to the component being activated or deactivated, receiving or losing the input focus:

cm_Activate	Corresponds to the OnActivate event of forms and of the application
cm_Deactivate	Corresponds to OnDeactivate
cm_Enter	Corresponds to OnEnter
cm_Exit	Corresponds to OnExit
cm_FocusChanged	Sent whenever the focus changes between components of the same form (later, you'll see an example using this message)
cm_GotFocus	Declared but not used
cm_LostFocus	Declared but not used

- Messages sent to child components when a property changes:

cm_BiDiModeChanged	cm_IconChanged
cm_BorderChanged	cm_ShowHintChanged
cm_ColorChanged	cm_ShowingChanged
cm_Ctl3DChanged	cm_SysFontChanged
cm_CursorChanged	cm_TabStopChanged
cm_EnabledChanged	cm_TextChanged
cm_FontChanged	cm_VisibleChanged

Monitoring these messages can help track changes in a property. You might need to respond to these messages in a new component, but it's not likely.

- Messages related to *ParentXxx* properties: cm\_ParentFontChanged, cm\_ParentColorChanged, cm\_ParentCtl3DChanged, cm\_ParentBiDiModeChanged, and cm\_ParentShowHintChanged. These are similar to the messages in the previous group.
- Notifications of changes in the Windows system: cm\_SysColorChange, cm\_WinIniChange, cm\_TimeChange, and cm\_FontChange. Handling these messages is useful only in special components that

need to keep track of system colors or fonts.

- **Mouse messages:** `cm_Drag` is sent many times during dragging operations. `cm_MouseEnter` and `cm_MouseLeave` are sent to the control when the cursor enters or leaves its surface, but they are sent by the `Application` object as low-priority messages. `cm_MouseWheel` corresponds to wheel-based operations.

Application messages:

<code>cm_AppKeyDown</code>	Sent to the <code>Application</code> object to let it determine whether a key corresponds to a menu shortcut
<code>cm_AppSysCommand</code>	Corresponds to the <code>wm_SysCommand</code> message
<code>cm_DialogHandle</code>	Sent in a DLL to retrieve the value of the <code>DialogHandle</code> property (used by some dialog boxes not built with Delphi)
<code>cm_InvokeHelp</code>	Sent by code in a DLL to call the <code>InvokeHelp</code> method
<code>cm_WindowHook</code>	Sent in a DLL to call the <code>HookMainWindow</code> and <code>UnhookMainWindow</code> methods

You'll rarely need to use these messages. There is also a `cm_HintShowPause` message, which is never handled in VCL.

- **Delphi internal messages:**

<code>cm_CancelMode</code>	Terminates special operations, such as showing the pull-down list of a combo box
<code>cm_ControlChange</code>	Sent to each control before adding or removing a child control (handled by some common controls)
<code>cm_ControlListChange</code>	Sent to each control before adding or removing a child control (handled by the <code>DBCtrlGrid</code> component)
<code>cm_DesignHitTest</code>	Determines whether a mouse operation should go to the component or to the form designer
<code>cm_HintShow</code>	Sent to a control just before displaying its hint (only if the <code>ShowHint</code> property is <code>True</code> )
<code>cm_HitTest</code>	Sent to a control when a parent control is trying to locate a child control at a given mouse position (if any)
<code>cm_MenuChanged</code>	Sent after MDI or OLE menu-merging operations

- **Messages related to special keys:**

<code>cm_ChildKey</code>	Sent to the parent control to handle some special keys (in Delphi, this message is handled only by <code>DBCtrlGrid</code> components)
<code>cm_DialogChar</code>	Sent to a control to determine whether a given input key is its accelerator character
<code>cm_DialogKey</code>	Handled by modal forms and controls that need to perform special actions
<code>Cm_IsShortCut</code>	Is currently not used (as most code simply calls <code>IsShortCut</code> ), but it is intended to be used to identify if a shortcut is known to be supported by a form, through either the <code>OnShortCut</code> event, a menu item, or an action.
<code>cm_WantSpecialKey</code>	Handled by controls that interpret special keys in an unusual way (for example, using the <code>Tab</code> key for navigation, as some <code>Grid</code> components do)

- Messages for specific components:

<code>cm_GetDataLink</code>	Used by DBCtrlGrid controls (and discussed in <a href="#">Chapter 17</a> , "Writing Database Components")
<code>cm_TabFontChanged</code>	Used by TabbedNotebook components
<code>cm_ButtonPressed</code>	Used by SpeedButtons to notify other sibling SpeedButton components (to enforce radio-button behavior)
<code>cm_DeferLayout</code>	Used by DBGrid components

- OLE container messages: `cm_DocWindowActivate`, `cm_IsToolControl`, `cm_Release`, `cm_UIActivate`, and `cm_UIDeactivate`.
- Dock-related messages, including `cm_DockClient`, `cm_DockNotification`, `cmFloat`, and `cm_UndockClient`.
- Method-implementation messages, such as `cm_RecreateWnd`, called inside the `RecreateWnd` method of `TControl`; `cm_Invalidated`, called inside `TControl.Invalidated`; `cm_Changed`, called inside `TControl.Changed`; and `cm_AllChildrenFlipped`, called in the `DoFlipChildren` methods of `TWinControl` and `TScrollingWinControl`. In the similar group fall two action list-related messages, `cm_ActionUpdate` and `cm_ActionExecute`.

## Component Notifications

Component notification messages are those sent from a parent form or component to its children. These notifications correspond to messages sent by Windows to the parent control's window, but logically intended for the control. For example, interaction with controls such as buttons, edit boxes, or list boxes causes Windows to send a `wm_Command` message to the parent of the control. When a Delphi program receives these messages, it forwards the message to the control itself, as a notification. The Delphi control can handle the message and eventually fire an event. Similar dispatching operations take place for many other messages.

The connection between Windows messages and component notification ones is so tight that you'll often recognize the name of the Windows message from the name of the notification message, replacing the initial *cn* with *wm*. There are several distinct groups of component notification messages:

- General keyboard messages: `cn_Char`, `cn_KeyUp`, `cn_KeyDown`, `cn_SysChar`, and `cn_SysKeyDown`
- Special keyboard messages used only by list boxes with the `lbs_WantKeyboardInput` style: `cn_CharToItem` and `cn_VKeyToItem`
- Messages related to the owner-draw technique: `cn_CompareItem`, `cn_DeleteItem`, `cn_DrawItem`, and `cn_MeasureItem`
- Messages for scrolling, used only by scroll bar and track bar controls: `cn_HScroll` and `cn_VScroll`
- General notification messages, used by most controls: `cn_Command`, `cn_Notify`, and `cn_ParentNotify`
- Control color messages: `cn_CtlColorBtn`, `cn_CtlColorDlg`, `cn_CtlColorEdit`, `cn_CtlColorListbox`, `cn_CtlColorMsgbox`, `cn_CtlColorScrollbar`, and `cn_CtlColorStatic`

Other control notifications are defined for common controls support (in the `ComCtrls` unit).

## An Example of Component Messages

As an example of the use of some component messages, I've written the CMNTest program. It has a form with three edit boxes and associated labels. The first message it handles, `cm_ DialogKey`, allows it to treat the Enter key as if it were a Tab key. The code of this method checks for the Enter key's code and sends the same message, but passes the `vk_Tab` key code. To halt further processing of the Enter key, you set the result of the message to 1:

```
procedure TForm1.CMDialogKey(var Message: TCMDialogKey);  
begin  
  if (Message.CharCode = VK_RETURN) then  
    begin  
      Perform (CM_DialogKey, VK_TAB, 0);  
      Message.Result := 1;  
    end  
  else  
    inherited;  
end;
```

The second message, `cm_ DialogChar`, monitors accelerator keys. This technique can be useful to provide custom shortcuts without defining an extra menu for them. Notice that while this code is correct for a component, in a normal application this can be achieved more easily by handling the form's `OnShortCut` event. In this case, you log the special keys in a label:

```
procedure TForm1.CMDialogChar(var Msg: TCMDialogChar);  
begin  
  Label1.Caption := Label1.Caption + Char (Msg.CharCode);  
  inherited;  
end;
```

Finally, the form handles the `cm_ FocusChanged` message, to respond to focus changes without having to handle the `OnEnter` event of each of its components. Again, the action displays a description of the focused component:

```
procedure TForm1.CmFocusChanged(var Msg: TCmFocusChanged);  
begin  
  Label5.Caption := 'Focus on ' + Msg.Sender.Name;  
end;
```

The advantage of this approach is that it works independently of the type and number of components you add to the form, and it does so without any special action on your part. Again, this is a trivial example for such an advanced topic, but if you add to this the code of the `ActiveButton` component, you have at least a few reasons to look into these special, undocumented messages. At times, writing the same code without their support can become extremely complex.

## A Dialog Box in a Component

The next component we'll examine is completely different from those you have seen up to now. After building window-based controls and graphic components, I'll now show you how to build a nonvisual component.

The basic idea is that forms are components. When you have built a form that might be particularly useful in multiple projects, you can add it to the Object Repository or make a component out of it. The second approach is more complex than the first, but it makes using the new form easier and allows you to distribute the form without its source code. As an example, I'll build a component based on a custom dialog box, trying to mimic as much as possible the behavior of standard Delphi dialog box components.

The first step in building a dialog box in a component is to write the code for the dialog box itself, using the standard Delphi approach. Just define a new form and work on it as usual. When a component is based on a form, you can almost visually design the component. Of course, once the dialog box has been built, you have to define a component around it in a nonvisual way.

The standard dialog box in this example is based on a list box, because it is common to let a user choose a value from a list of strings. I've customized this common behavior in a dialog box and then used it to build a component. The `ListBoxForm` form has a list box and the typical OK and Cancel buttons, as shown in its textual description:

```
object MdListBoxForm: TMdListBoxForm
  BorderStyle = bsDialog
  Caption = 'ListBoxForm'
object ListBox1: TListBox
  OnDblClick = ListBox1DblClick
end
object BitBtn1: TBitBtn
  Kind = bkOK
end
object BitBtn2: TBitBtn
  Kind = bkCancel
end
end
```

The only method of this dialog box form relates to the list box's double-click event, which closes the dialog box as though the user clicked the OK button, by setting the `ModalResult` property of the form to `mrOk`. Once the form works, you can begin changing its source code, adding the definition of a component and removing the declaration of the global variable for the form.

**Note** For components based on a form, you can use two Pascal source code files: one for the form and the other for the component encapsulating it. It is also possible to place both the component and the form in a single unit, as I've done for this example. In theory, it would be better to declare the form class in the implementation portion of this unit, hiding it from the component's users. But in practice, this is not a good idea. To manipulate the form visually in the Form Designer, the form class declaration must appear in the interface section of the unit. The rationale behind this behavior of the Delphi IDE is that, among other things, this constraint minimizes the amount of code the module manager has to scan to find the form declaration—an operation that must be performed often to maintain the synchronization of the visual form with the form class definition.

The most important operation is the definition of the `TMdListBoxDialog` component. This component is defined as nonvisual because its immediate ancestor class is `TComponent`. The component has one public property and these three published properties:

- `Lines` is a `TStrings` object, which is accessed via two methods, `GetLines` and `SetLines`. This second method uses the `Assign` procedure to copy the new values to the private field corresponding to this property. This internal object is initialized in the `Create` constructor and destroyed in the `Destroy` method.



- Selected is an integer that directly accesses the corresponding private field. It stores the selected element of the list of strings.
- Title is a string used to change the title of the dialog box.

The public property is `SelItem`, a read-only property that automatically retrieves the selected element of the list of strings. Notice that this property has no storage and no data; it accesses other properties, providing a virtual representation of data:

```

type
  TMDListBoxDialog = class (TComponent)
  private
    FLines: TStrings;
    FSelected: Integer;
    FTitle: string;
    function GetSelItem: string;
    procedure SetLines (Value: TStrings);
    function GetLines: TStrings;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Execute: Boolean;
    property SelItem: string read GetSelItem;
  published
    property Lines: TStrings read GetLines write SetLines;
    property Selected: Integer read FSelected write FSelected;
    property Title: string read FTitle write FTitle;
  end;

```

Most of this example's code is in the `Execute` method, a function that returns `True` or `False` depending on the modal result of the dialog box. This is consistent with the `Execute` method of most standard Delphi dialog box components. The `Execute` function creates the form dynamically, sets some of its values using the component's properties, shows the dialog box, and, if the result is correct, updates the current selection:

```

function TMDListBoxDialog.Execute: Boolean;
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create ('No items in the list');
  ListBoxForm := TListBoxForm.Create (Self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
      begin
        Result := True;
        Selected := ListBoxForm.ListBox1.ItemIndex;
      end
    else
      Result := False;
    finally
      ListBoxForm.Free;
    end;
  end;

```

Notice that the code is contained within a `try/finally` block, so if a run-time error occurs when the dialog box is

displayed, the form will be destroyed anyway. I've also used exceptions to raise an error if the list is empty when a user runs it. This error is by design, and using an exception is a good technique to enforce it. The component's other methods are straightforward. The constructor creates the `FLines` string list, which is deleted by the destructor; the `GetLines` and `SetLines` methods operate on the string list as a whole; and the `GetSelItem` function (which follows) returns the text of the selected item:

```
function TMdListBoxDialog.GetSelItem: string;
begin
  if (Selected >= 0) and (Selected < FLines.Count) then
    Result := FLines [Selected]
  else
    Result := '';
end;
```

Of course, because you are manually writing the component code and adding it to the original form's source code, you have to remember to write the `Register` procedure.

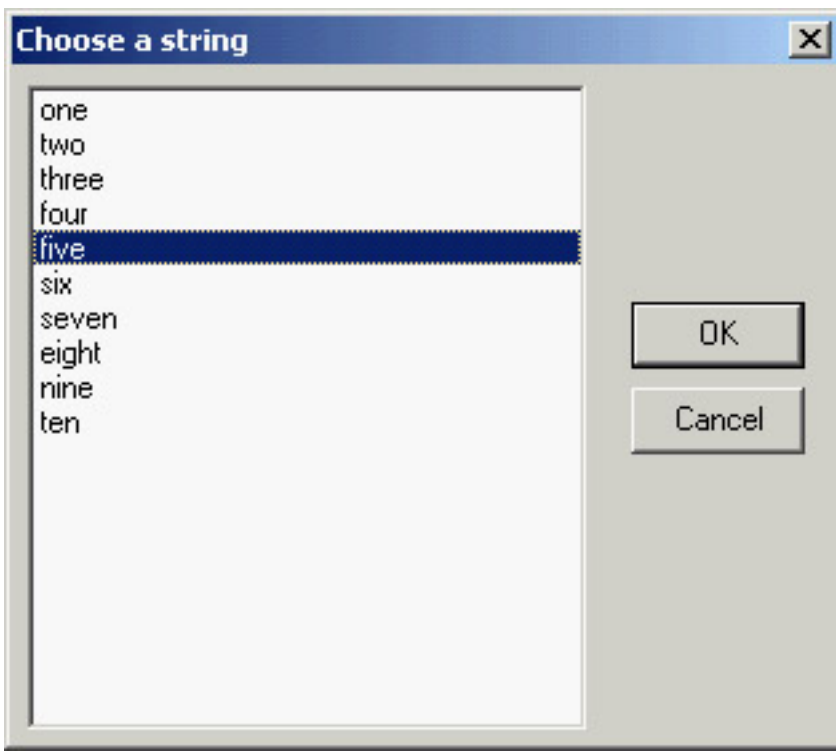
Once you've written the `Register` procedure and the component is ready, you must provide a bitmap. For nonvisual components, bitmaps are very important because they are used not only for the Component Palette, but also when you place the component on a form.

## Using the Nonvisual Component

I've written a project to test the component once the bitmap has been prepared and the component has been installed. The form of this test program has a button, an edit box, and the `MdListDialog` component. In the program, I've added only a few lines of code, corresponding to the button's `OnClick` event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // select the text of the edit, if corresponding to one of the strings
  MdListDialog1.Selected := MdListDialog1.Lines.IndexOf (Edit1.Text);
  // run the dialog and get the result
  if MdListDialog1.Execute then
    Edit1.Text := MdListDialog1.SelItem;
end;
```

That's all the code you need to run the dialog box placed in the component, as you can see in [Figure 9.10](#). As you've seen, this is an interesting approach to the development of some common dialog boxes.



**Figure 9.10:** The ListDialDemo example shows the dialog box encapsulated in the ListDial component.

## Collection Properties

At times you need a property holding a list of values, not a single value. Sometimes you can use a `TStringList`-based property, but it accounts only for textual data (even though an object can be attached to each string). When you need a property hosting an array of objects, the most VCL-sound solution is to use a collection. The role of collections, by design, is to build properties that contain a list of values. Examples of Delphi collection properties include the `DBGrid` component's `Columns` property, and the `TStatusBar` component's `Panels` property.

A collection is basically a container of objects of a given type. For this reason, to define a collection, you need to inherit a new class from the `TCollection` class and also inherit a new class from the `TCollectionItem` class. This second class defines the objects held by the collection; the collection is created by passing to it the class of the objects it will hold.

Not only does the collection class manipulate the items of the collection, but it is also responsible for creating new objects when its `Add` method is called. You cannot create an object and then add it to an existing collection. [Listing 9.3](#) shows two classes for the items and a collection, with their most relevant code.

### Listing 9.3: The Classes for a Collection and Its Items

---

```

type
  TMyItem = class (TCollectionItem)
  private
    FCode: Integer;
    FText: string;
    procedure SetCode(const Value: Integer);
    procedure SetText(const Value: string);
  published
    property Text: string read FText write SetText;
    property Code: Integer read FCode write SetCode;
  end;

  TMyCollection = class (TCollection)
  private
    FComp: TComponent;
    FCollString: string;
  public
    constructor Create (CollOwner: TComponent);
    function GetOwner: TPersistent; override;
    procedure Update(Item: TCollectionItem); override;
  end;

{ TMyCollection }

constructor TMyCollection.Create (CollOwner: TComponent);
begin
  inherited Create (TMyItem);
  FComp := CollOwner;
end;

function TMyCollection.GetOwner: TPersistent;
begin
  Result := FComp;
end;

procedure TMyCollection.Update(Item: TCollectionItem);
var
```

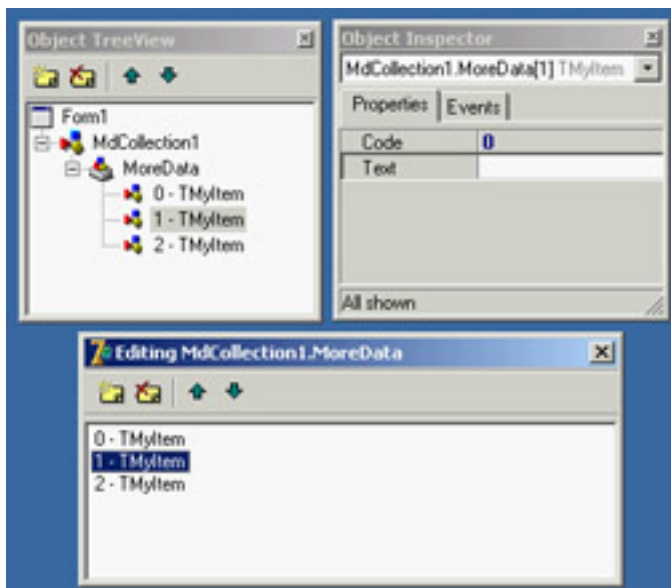
```

str: string;
i: Integer;
begin
  inherited;
  // update everything in any case...
  str := '';
  for i := 0 to Count - 1 do
  begin
    str := str + (Items [i] as TMyItem).Text;
    if i < Count - 1 then
      str := str + '-';
  end;
  FCollString := str;
end;

```

---

The collection must define the `GetOwner` method to be displayed properly in the collection property editor provided by the Delphi IDE. For this reason, it needs a link to the component hosting it, the collection owner (stored in the `FComp` field in the code). You can see this sample component's collection in [Figure 9.11](#).



**Figure 9.11:** The collection editor, with the Object TreeView and the Object Inspector for the collection item

Every time data changes in a collection item, its code calls the `Changed` method (passing `True` or `False` to indicate whether the change is local to the item or refers to the entire set of items in the collection). As a result of this call, the `TCollection` class calls the virtual method `Update`, which receives as a parameter the single item requesting the update, or `nil` if all items changed (and when the `Changed` method is called with `True` as a parameter). You can override this method to update the values of other elements of the collection, of the collection itself, or of the target component.

In this example you update a string with a summary of the collection data that you've added to the collection and that the host component will surface as a property. Using the collection within a component is simple. You declare a collection, create it in the constructor and free it at the end, and expose it through a property:

```

type
  TCanTest = class(TComponent)
  private
    FColl: TMyCollection;
    function GetCollString: string;
  public

```

```

constructor Create (aOwner: TComponent); override;
destructor Destroy; override;
published
  property MoreData: TMyCollection read FColl write SetMoreData;
  property CollString: string read GetCollString;
end;

constructor TCanTest.Create(aOwner: TComponent);
begin
  inherited;
  FColl := TMyCollection.Create (Self);
end;

destructor TCanTest.Destroy;
begin
  FColl.Free;
  inherited;
end;

procedure TCanTest.SetMoreData(const Value: TMyCollection);
begin
  FColl.Assign (Value);
end;

function TCanTest.GetCollString: string;
begin
  Result := FColl.FCollString;
end;

```

Notice that the collection items are streamed in DFM files along with the component hosting them, using the special `item` markers and angle brackets, as in the following example:

```

object MdCollection1: TMdCollection
  MoreData = <
    item
      Text = 'one'
      Code = 1
    end
    item
      Text = 'two'
      Code = 2
    end
    item
      Text = 'three'
      Code = 3
    end>
end

```

## Defining Custom Actions

In addition to defining custom components, you can define and register new standard actions, which will be made available in the Action List component's Action Editor. Creating new actions is not complex. You have to inherit from the `TAction` class and override some of the methods of the base class.

You must override three methods:

- The `HandlesTarget` function returns whether the action object wants to handle the operation for the current target, which is by default the control with the focus.
- The `UpdateTarget` procedure can set the user interface of the controls connected with the action, eventually disabling the action if the operation is currently not available.
- You can implement the `ExecuteTarget` method to determine the code to execute, so that the user can select the action and doesn't have to implement it.

To show you this approach in practice, I've implemented the three cut, copy, and paste actions for a list box, in a way similar to what VCL does for an edit box (although I've simplified the code a little). I've written a base class, which inherits from the generic `TListControlAction` class of the `ExtActns` unit. This base class, `TMdCustomListAction`, adds some common code shared by all the specific actions and publishes a few action properties. The three derived classes have their own `ExecuteTarget` code, plus little more. Here are the four classes:

```

type
  TMdCustomListAction = class (TListControlAction)
  protected
    function TargetList (Target: TObject): TCustomListBox;
    function GetControl (Target: TObject): TCustomListControl;
  public
    procedure UpdateTarget (Target: TObject); override;
  published
    property Caption;
    property Enabled;
    property HelpContext;
    property Hint;
    property ImageIndex;
    property ListControl;
    property ShortCut;
    property SecondaryShortCuts;
    property Visible;
    property OnHint;
  end;

  TMdListCutAction = class (TMdCustomListAction)
  public
    procedure ExecuteTarget (Target: TObject); override;
  end;

  TMdListCopyAction = class (TMdCustomListAction)
  public
    procedure ExecuteTarget (Target: TObject); override;
  end;

  TMdListPasteAction = class (TMdCustomListAction)
  public

```

```

procedure UpdateTarget (Target: TObject); override;
procedure ExecuteTarget (Target: TObject); override;
end;

```

The `HandlesTarget` method, one of the three key methods of action classes, is provided by the `TListControlAction` class with this code:

```

function TListControlAction.HandlesTarget(Target: TObject): Boolean;
begin
  Result := ((ListControl <> nil) or
    (ListControl = nil) and (Target is TCustomListControl)) and
    TCustomListControl(Target).Focused;
end;

```

The `UpdateTarget` method has two different implementations. The default implementation is provided by the base class and used by the copy and cut actions. These actions are enabled only if the target list box has at least one item and an item is currently selected. The status of the paste action depends on the Clipboard status:

```

procedure TMdCustomListAction.UpdateTarget (Target: TObject);
begin
  Enabled := (TargetList (Target).Items.Count > 0)
    and (TargetList (Target).ItemIndex >= 0);
end;

```

```

function TMdCustomListAction.TargetList (Target: TObject): TCustomListBox;
begin
  Result := GetControl (Target) as TCustomListBox;
end;

```

```

function TMdCustomListAction.GetControl(Target: TObject): TCustomListControl;
begin
  Result := Target as TCustomListControl;
end;

```

```

procedure TMdListPasteAction.UpdateTarget (Target: TObject);
begin
  Enabled := Clipboard.HasFormat (CF_TEXT);
end;

```

The `TargetList` function uses the `TListControlAction` class's `GetControl` function, which returns either the list box connected to the action at design time or the target control (the list box control with the input focus).

Finally, the three `ExecuteTarget` methods perform the corresponding actions on the target list box:

```

procedure TMdListCopyAction.ExecuteTarget (Target: TObject);
begin
  with TargetList (Target) do
    Clipboard.AsText := Items [ItemIndex];
end;

```

```

procedure TMdListCutAction.ExecuteTarget(Target: TObject);
begin
  with TargetList (Target) do
    begin
      Clipboard.AsText := Items [ItemIndex];
      Items.Delete (ItemIndex);
    end;
end;

```



```
procedure TMDListPasteAction.ExecuteTarget(Target: TObject);  
begin  
    (TargetList (Target)).Items.Add (Clipboard.AsText);  
end;
```

Once you've written this code in a unit and added it to a package (in this case, the MdPack package), the final step is to register the new custom actions in a given category. This category is indicated as the first parameter of the `RegisterActions` procedure; the second parameter is the list of action classes to register:

```
procedure Register;  
begin  
    RegisterActions ('List',  
        [TMDListCutAction, TMDListCopyAction, TMDListPasteAction], nil);  
end;
```

To test the use of these three custom actions, I've written the ListTest example (included with the source code for this chapter). This program has two list boxes plus a toolbar that contains three buttons connected to the three custom actions and an edit box for entering new values. The program allows a user to cut, copy, and paste list box items. Nothing special, you might think—but the strange fact is that the program has no code!

**Warning** To set up an image for an action (and to define default property values in general) you need to use the third parameter of the `RegisterActions` procedure, which is a data module hosting the image list and an action list with the predefined values. As you have to register the actions before you can set up such a data module, you'll need a double registration while developing these actions. This issue is quite complex so I won't cover it here, but a detailed description can be found on <http://www.blong.com/Conferences/BorCon2002/Actions/2110.htm> in the sections "Registering Standard Actions" and "Standard Actions And Data Modules."

## Writing Property Editors

Writing components is an effective way to customize Delphi, helping developers to build applications more quickly without requiring a detailed knowledge of low-level techniques. The Delphi environment is also open to extensions. In particular, you can extend the Object Inspector by writing custom property editors and the Form Designer by adding component editors.

Along with these techniques, Delphi offers internal interfaces to add-on tool developers. Using these interfaces, known as the OpenTools API, requires an advanced understanding of how the Delphi environment works and a fairly good knowledge of many advanced techniques that are not discussed in this book. For references to technical information and some examples of these techniques, see [Appendix A](#), "Extra Delphi Tools by the Author."

**Note** The OpenTools API in Delphi has changed considerably over time. For example, the `DsgnIntf` unit from Delphi 5 has been split into `DesignIntf`, `DesignEditors`, and other specific units. Borland has also introduced interfaces to define the sets of methods for each kind of editor. However, most of the simpler examples, such as those presented in this book, compile almost unchanged from earlier versions of Delphi. For more information, you can study the extensive source code in Delphi's `\Source\ToolsApi` directory. Notice also that with Delphi 6 Update Pack 2 Borland has for the first time shipped a Help file with the documentation of the OpenTools API.

Every property editor must inherit from the abstract `TPropertyEditor` class, which is defined in the `DesignEditors` unit and provides a standard implementation for the `IProperty` interface. Delphi already defines some specific property editors for strings (the `TStringProperty` class), integers (the `TIntegerProperty` class), characters (the `TCharProperty` class), enumerations (the `TEnumProperty` class), and sets (the `TSetProperty` class), so you can inherit your property editor from the one for the property type you are working with.

In any custom property editor, you must redefine the `GetAttributes` function so it returns a set of values indicating the capabilities of the editor. The most important attributes are `paValueList` and `paDialog`. The `paValueList` attribute indicates that the Object Inspector will show a combo box with a list of values (eventually sorted if the `paSortList` attribute is set) provided by overriding the `GetValues` method. The `paDialog` attribute style activates an ellipsis button in the Object Inspector, which executes the editor's `Edit` method.

### An Editor for the Sound Properties

The sound button you built earlier has two sound-related properties: `SoundUp` and `SoundDown`. These are strings, so you can display them in the Object Inspector using a default property editor. However, requiring the user to type the name of a system sound or an external file is not friendly, and it's a bit error-prone.

We could write a generic editor to handle filenames, but you want to be able to choose the name of a system sound as well. (System sounds are predefined names of sounds connected with user operations, associated with actual sound files in the Windows Control Panel's Sounds applet.) For this reason, I built a more complex property editor. My editor for sound strings allows a user to either choose a value from a drop-down list or display a dialog box from which to load and test a sound (from a sound file or a system sound). The property editor provides both `Edit` and `GetValues` methods:

```
type
  TSoundProperty = class (TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure Edit; override;
  end;
```

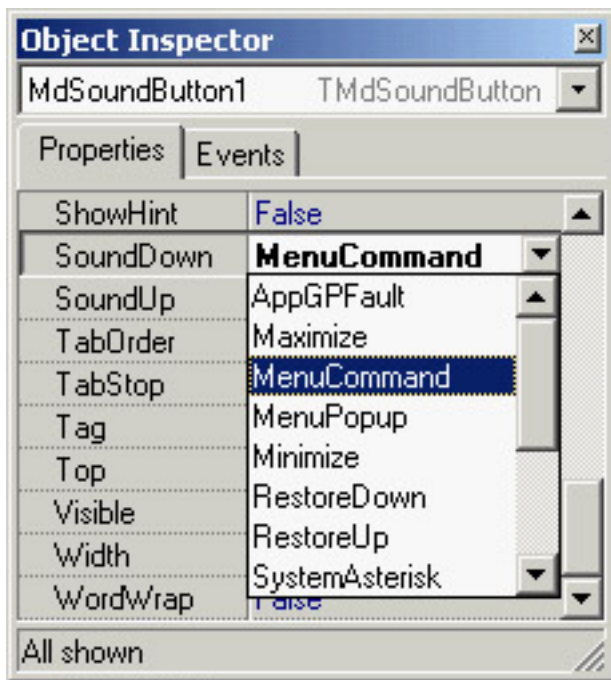
**Tip** The default Delphi convention is to name a property editor class with a name ending with *Property* and all component editors with a name ending with *Editor*.

The `GetAttributes` function combines the `paValueList` (for the drop-down list) and `paDialog` (for the custom edit box) attributes, and also sorts the lists and allows the selection of the property for multiple components:

```
function TSoundProperty.GetAttributes: TPropertyAttributes;  
begin  
    // editor, sorted list, multiple selection  
    Result := [paDialog, paMultiSelect, paValueList, paSortList];  
end;
```

The `GetValues` method calls the procedure it receives as a parameter many times, once for each string it wants to add to the drop-down list (as you can see in [Figure 9.12](#)):

```
procedure TSoundProperty.GetValues(Proc: TGetStrProc);  
begin  
    // provide a list of system sounds  
    Proc ('Maximize');  
    Proc ('Minimize');  
    Proc ('MenuCommand');  
    Proc ('MenuPopup');  
    Proc ('RestoreDown');  
    ...  
end;
```



**Figure 9.12:** The list of sounds provides a hint for the user, who can also type in the property value or double-click to activate the editor (shown later, in [Figure 9.13](#)).

A better approach would be to extract these values from the Windows Registry, where all these names are listed. The `Edit` method is straightforward: It creates and displays a dialog box. I could have displayed the Open dialog box directly, but I decided to add an intermediate step to allow the user to test the sound. This is similar to what Delphi does with graphic properties: You open the preview first, and load the file only after you've confirmed that it's correct. The most important step is to load the file and test it before you apply it to the property. Here is the code for the `Edit` method:

```
procedure TSoundProperty.Edit;  
begin  
    SoundForm := TSoundForm.Create (Application);  
    try
```

```

SoundForm.ComboBox1.Text := GetValue;
// show the dialog box
if SoundForm.ShowModal = mrOK then
    SetValue (SoundForm.ComboBox1.Text);
finally
    SoundForm.Free;
end;
end;

```

The `GetValue` and `SetValue` methods are defined by the base class, the string property editor. They read and write the value of the current component's property that you are editing.

As an alternative, you can access the component you're editing by using the `GetComponent` method (which requires a parameter indicating which of the selected components you are working on—0 indicates the first component). When you access the component directly, you also need to call the `Designer` object's `Modified` method (a property of the base class property editor). You don't need this `Modified` call in the example, because the base class `SetValue` method does this automatically for you.

The previous `Edit` method displays a dialog box—a standard Delphi form that is built visually, as always, and added to the package hosting the design-time components. The form is quite simple; a `ComboBox` displays the values returned by the `GetValues` method, and four buttons allow you to open a file, test the sound, and terminate the dialog box by accepting the values or canceling. You can see an example of the dialog box in [Figure 9.13](#). Providing a drop-down list of values *and* a dialog box for editing a property causes the Object Inspector to display only the arrow button that indicates a drop-down list and to omit the ellipsis button to indicate that a dialog box editor is available. In this case, as it happened for the default `Color` property editor, the dialog box is obtained by double-clicking the current value or pressing `Ctrl+Enter`.



**Figure 9.13:** The Sound Property Editor's form displays a list of available sounds and lets you load a file and hear the selected sound.

The form's first two buttons have a method assigned to their `OnClick` event:

```

procedure TSoundForm.btnLoadClick(Sender: TObject);
begin
    if OpenDialog1.Execute then
        ComboBox1.Text := OpenDialog1.FileName;
end;

procedure TSoundForm.btnPlayClick(Sender: TObject);
begin
    PlaySound (PChar (ComboBox1.Text), 0, snd_Async);
end;

```

Notice that it is far from simple to determine whether a sound is properly defined and is available. (You can check the file, but the system sounds create a few issues.) The `PlaySound` function returns an error code when played synchronously, but only if it can't find the default system sound it attempts to play if it can't find the sound you ask for. If the requested sound is not available, it plays the default system sound and doesn't return the error code. `PlaySound` looks for the sound in the Registry first and, if it doesn't find the sound there, checks to see whether the specified sound file exists.

**Tip** If you want to further extend this example, you might add graphics to the drop-down list displayed in the Object Inspector—if you can decide which graphics to attach to particular sounds.

## Installing the Property Editor

After you've written this code, you can install the component and its property editor in Delphi. To accomplish this, you have to add the following statement to the unit's `Register` procedure:

```
procedure Register;  
begin  
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundUp',  
    TSoundProperty);  
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundDown',  
    TSoundProperty);  
end;
```

This call registers the editor specified in the last parameter for use with properties of type string (the first parameter), but only for a specific component and for a property with a specific name. These last two values can be omitted to provide more general editors. Registering this editor allows the Object Inspector to show a list of values and the dialog box called by the `Edit` method.

To install this component, you can add its source code file to an existing or new package. Instead of adding this unit and the others in this chapter to the `MdPack` package, I created a second package containing all the add-ins built in this chapter. The package is named `MdDesPk` (*Mastering Delphi design package*). What's new about this package is that I compiled it using the `{ $DESIGNONLY }` compiler directive. This directive is used to mark packages that interact with the Delphi environment, installing components and editors, but are not required at run time by applications you've built.

**Note** The source code for all the add-on tools is in the `MdDesPk` subdirectory, along with the code for the package used to install them. There are no examples demonstrating how to use these design-time tools, because all you have to do is select the corresponding components in the Delphi environment and see how they behave.

The property editor's unit uses the `SoundB` unit, which defines the `TMdSoundButton` component. For this reason, the new package should refer to the existing package. Here is its initial code (I'll add other units to it later in this chapter):

```
package MdDesPk;  
  
{ $R *.RES }  
{ $ALIGN ON }  
...  
{ $DESCRIPTION 'Mastering Delphi DesignTime Package' }  
{ $DESIGNONLY }  
  
requires  
  vcl,  
  MdPack,  
  designide;  
  
contains  
  PeSound in 'PeSound.pas',  
  PeFSound in 'PeFSound.pas' {SoundForm};
```

## Writing a Component Editor

Using property editors allows the developer to make a component more user-friendly. The Object Inspector represents one of the key pieces of the user interface of the Delphi environment, and Delphi developers use it quite often. However, you can adopt a second approach to customize how a component interacts with Delphi: write a custom component editor.

Just as property editors extend the Object Inspector, component editors extend the Form Designer. When you right-click within a form at design time, you see some default menu items, plus the items added by the component editor of the selected component. Examples of these menu items are those used to activate the Menu Designer, the Fields Editor, the Visual Query Builder, and other editors in the environment. At times, displaying these special editors becomes the default action of a component when it is double-clicked.

Common uses of component editors include adding an About box with information about the developer of the component, adding the component name, and providing specific wizards to set up component properties. In particular, the original intent was to allow a wizard, or some direct code, to set multiple properties in one shot, rather than setting them all individually.

### Deriving from the *TComponentEditor* Class

A component editor should generally inherit from the `TComponentEditor` class, which provides the base implementation of the `IComponentEditor` interface. The most important methods of this interface are as follows:

- `GetVerbCount` returns the number of menu items to add to the Form Designer's shortcut menu when the component is selected.
- `GetVerb` is called once for each new menu item and should return the text that will go in the shortcut menu for each.
- `ExecuteVerb` is called when one of the new menu items is selected. The number of the item is passed as the method's parameter.
- `Edit` is called when the user double-clicks the component in the Form Designer to activate the default action.

Once you get used to the idea that a "verb" is nothing but a new menu item with a corresponding action to execute, the names of the methods of this interface become quite intuitive. This interface is much simpler than those of property editors you've seen before.

**Note** Like property editors, component editors were modified from Delphi 5 to Delphi 6, and are now defined in the `DesignEditors` and `DesignIntf` units.

### A Component Editor for the `ListDialog`

Now that I've introduced the key ideas about writing component editors, let's look at an example—an editor for the `ListDialog` component built earlier. In my component editor, I want to be able to show an About box, add a copyright notice to the menu (an improper but common use of component editors), and allow users to perform a special action—previewing the dialog box connected with the dialog component. I also want to change the default action to show the About box after a beep (which is not particularly useful but demonstrates the technique).

To implement this component editor, the program must override the four methods listed in the [previous section](#):

```
uses
    DesignIntf;

type
    TMDListCompEditor = class (TComponentEditor)
```

```

function GetVerbCount: Integer; override;
function GetVerb(Index: Integer): string; override;
procedure ExecuteVerb(Index: Integer); override;
procedure Edit; override;
end;

```

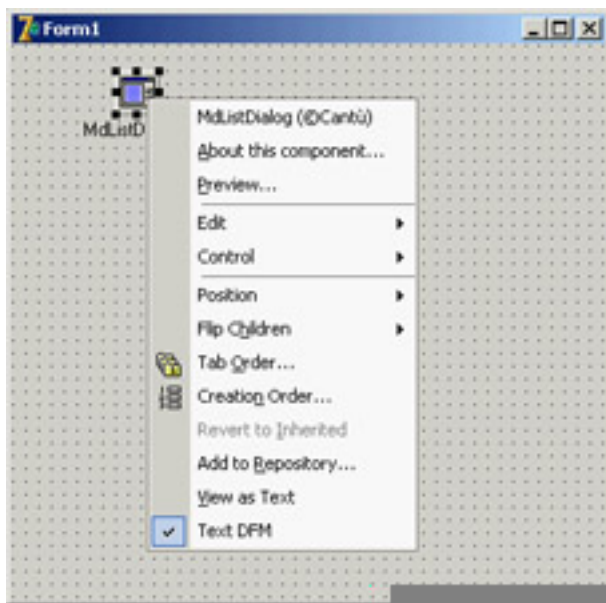
The first method returns the number of menu items to add to the shortcut menu, in this case 3. This method is called only once: before displaying the menu. The second method is called once for each menu item, so in this case it is called three times:

```

function TMDListCompEditor.GetVerb (Index: Integer): string;
begin
  case Index of
    0: Result := ' MdListDialog (@Cantù)';
    1: Result := '&About this component...';
    2: Result := '&Preview...';
  end;
end;

```

This code adds the menu items to the form's shortcut menu, as you can see in [Figure 9.14](#). Selecting any of these menu items activates the `ExecuteVerb` method of the component editor:



**Figure 9.14:** The custom menu items added by the component editor of the ListDialog component

```

procedure TMDListCompEditor.ExecuteVerb (Index: Integer);
begin
  case Index of
    0: ; // nothing to do
    1: MessageDlg ('This is a simple component editor'#13 +
      'built by Marco Cantù'#13 +
      'for the book "Mastering Delphi"', mtInformation, [mbOK], 0);
    2: with Component as TMDListDialog do
      Execute;
  end;
end;

```

I decided to handle the first two items in a single branch of the `case` statement, although I could have skipped the code for the copyright notice item. The other command changes calls the `Execute` method of the component you are



editing, determined using the `TComponentEditor` class's `Component` property. Knowing the type of the component, you can easily access its methods after a dynamic typecast.

The last method refers to the component's default action and is activated by double-clicking the component in the Form Designer:

```
procedure TMDListCompEditor.Edit;  
begin  
    // produce a beep and show the about box  
    Beep;  
  
    ExecuteVerb (0);  
end;
```

## Registering the Component Editor

To make this editor available to the Delphi environment, you need to register it. Once more, you can add to its unit a `Register` procedure and call a specific registration procedure for component editors:

```
procedure Register;  
begin  
    RegisterComponentEditor (TMDListDialog, TMDListCompEditor);  
end;
```

I've added this unit to the `MdDesPk` package, which includes all the design-time extensions from this chapter. After installing and activating this package, you can create a new project, place a list dialog component in it, and experiment with it.

**Team LiB**

◀ PREVIOUS

NEXT ▶



## What's Next?

In this chapter, you have seen how to define various types of properties, how to add events, and how to define and override component methods. You have seen various examples of components, including simple changes to existing components, new graphical components, and, in the final section, a dialog box inside a component. While building these components, you have faced some new Windows programming challenges. In general, programmers often need to use the Windows API directly when writing new Delphi components.

Writing components is a handy technique for reusing software, but to make your components easier to use, you should integrate them as much as possible within the Delphi environment by writing property editors and component editors. You can also write many more extensions of the Delphi IDE, including custom wizards. I've built many Delphi extensions, some of which are discussed in [Appendix A](#).

[Chapter 10](#) focuses on Delphi DLLs. You have used DLLs in previous chapters, and it is time for a detailed discussion of their role and how to build them. I'll also further discuss the use of Delphi packages, which are a special type of DLL. To learn more about component development, refer to [Chapter 17](#), which focuses specifically on data-aware controls and custom dataset components.

## Chapter 10: Libraries and Packages

### Overview

Windows executable files come in two flavors: *programs* (EXEs) and *dynamic link libraries* (DLLs). When you write a Delphi application, you typically generate a program file. However, Delphi applications often use calls to functions stored in dynamic libraries. Each time you call a Windows API function directly, you actually access a dynamic library. It is easy to generate a DLL in the Delphi environment. However, some problems can arise due to the nature of DLLs. Writing a dynamic library in Windows is not always as simple as it seems, because the dynamic library and the calling program need to agree on calling conventions, parameter types, and other details. This chapter covers the basics of DLL programming from the Delphi point of view.

The second part of the chapter will focus on a specific type of dynamic link library: the Delphi *package*. Packages in Delphi provide a good alternative to plain DLLs, although not so many Delphi programmers take advantage of them outside the realm of component writing. Here I'll share with you some tips and techniques for using packages to partition a large application.

## The Role of DLLs in Windows

Before delving into the development of DLLs in Delphi and other programming languages, I'll give you a short technical overview of DLLs in Windows, highlighting the key elements. We will start by looking at dynamic linking, then see how Windows uses DLLs, and end with some general rules to follow when writing DLLs.

### What Is Dynamic Linking?

First, it is important to fully understand the difference between static and dynamic linking of functions or procedures. When a subroutine is not directly available in a source file, the compiler adds the subroutine to an internal symbol table. Of course, the Delphi compiler must have seen the declaration of the subroutine and know about its parameters and type, or it will issue an error.

After compilation of a normal—*static*—subroutine, the linker fetches the subroutine's compiled code from a Delphi compiled unit (or static library) and adds it to the program's code. The resulting executable file includes all the code of the program and of the units involved. The Delphi linker is smart enough to include only the minimum amount of code from the program's units and to link only the functions and methods that are actually used. This is why it is called "smart linker."

**Note** A notable exception to this rule is the inclusion of virtual methods. The compiler cannot determine in advance which virtual methods the program will call, so it has to include them all. For this reason, programs and libraries with too many virtual functions tend to generate larger executable files. While developing the VCL, the Borland developers had to balance the flexibility obtained with virtual functions against the reduced size of the executable files achieved by limiting the virtual functions.

In the case of dynamic linking, which occurs when your code calls a DLL-based function, the linker uses the information in the `external` declaration of the subroutine to set up an import table in the executable file. When Windows loads the executable file in memory, first it loads all the required DLLs, and then the program starts. During this loading process, Windows fills the program's import table with the addresses of the DLL functions in memory. If for some reason the DLL is not found or a referenced routine is not present in a DLL that is found, the program won't even start.

Each time the program calls an external function, it uses this import table to forward the call to the DLL code (which is now located in the program's address space). Note that this scheme does not involve two different applications. The DLL becomes part of the running program and is loaded in the same address space. All the parameter passing takes place on the application's stack (because the DLL doesn't have a separate stack) or in CPU registers. Because a DLL is loaded into the application's address space, any memory allocations of the DLL or any global data it creates reside in the address space of the main process. Thus, data and memory pointers can be passed directly from the DLL to the program and vice versa. This can also be extended to passing object references, which can be quite troublesome as the EXE and the DLL might have a different compiled class (and you can use packages exactly for this purpose, as we'll see later in this chapter).

There is another approach to using DLLs that is even more dynamic than the one I just discussed: At run time, you can load a DLL in memory, search for a function (provided you know its name), and call the function by name. This approach requires more complex code and takes some extra time to locate the function. The execution of the function, however, occurs with the same speed as calling an implicitly loaded DLL. On the positive side, you don't need to have the DLL available to start the program. We will use this approach in the DynaCall example later in the chapter.

### What are DLLs For?

Now that you have a general idea of how DLLs work, we can focus on the reasons for using them. The first advantage is that if different programs use the same DLL, the DLL is loaded in memory only once, thus saving system memory. DLLs are mapped into the private address space of each process (each running application), but their code is loaded in memory only once.

**Note** To be more precise, the operating system will try to load the DLL at the same address in each application's address space (using the preferred base address specified by the DLL). If that address is not available in a particular application's virtual address space, the DLL code image for that process will have to be relocated—an operation that is expensive in terms of both performance and memory use, because the relocation happens on a per-process basis, not system-wide.

Another interesting feature is that you can provide a different version of a DLL, replacing the current one, without having to recompile the application using it. This approach will work, of course, only if the functions in the DLL have the same parameters as the previous version. If the DLL has new functions, it doesn't matter. Problems may arise only if a function in the older version of the DLL is missing in the new one or if a function takes an object reference and the classes, base classes, or even compiler versions don't match.

This second advantage is particularly applicable to complex applications. If you have a very big program that requires frequent updates and bug fixes, dividing it into several executables and dynamic libraries allows you to distribute only the changed portions instead of a single large executable. Doing so makes sense for Windows system libraries in particular: You generally don't need to recompile your code if Microsoft provides an updated version of Windows system libraries—for example, in a new version of the operating system or a service pack.

Another common technique is to use dynamic libraries to store nothing but resources. You can build different versions of a DLL containing strings for different languages and then change the language at run time, or you can prepare a library of icons and bitmaps and then use them in different applications. The development of language-specific versions of a program is particularly important, and Delphi includes support for it through its Integrated Translation Environment (ITE).

Another key advantage is that DLLs are independent of the programming language. Most Windows programming environments, including most macro languages in end-user applications, allow a programmer to call a function stored in a DLL. This flexibility applies only to the use of functions, though. To share objects in a DLL across programming languages, you should move to the COM infrastructure or the .NET architecture.

## Rules for Delphi DLL Writers

Delphi DLL programmers need to follow several rules. A DLL function or procedure to be called by external programs must follow these guidelines:

- It must be listed in the DLL's `exports` clause. This makes the routine visible to the outside world.
- Exported functions should also be declared as `stdcall`, to use the standard Win32 parameter-passing technique instead of the optimized `register` parameter-passing technique (which is the default in Delphi). The exception to this rule is if you want to use these libraries only from other Delphi applications. Of course you can also use another calling convention, provided the other compiler understands it (like `cdecl`, which is the default on C compilers).
- The types of a DLL's parameters should be the default Windows types (mostly C-compatible data types), at least if you want to be able to use the DLL within other development environments. There are further rules for exporting strings, as you'll see in the `FirstDLL` example.
- A DLL can use global data that won't be shared by calling applications. Each time an application loads a DLL, it stores the DLL's global data in its own address space, as you will see in the `DllMem` example.
- Delphi libraries should trap all internal exceptions, unless you plan to use the library only from other Delphi programs.

## Using Existing DLLs

You have already used existing DLLs in examples in this book, when calling Windows API functions. As you might remember, all the API functions are declared in the system Windows unit. Functions are declared in the interface portion of the unit, as shown here:

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL; stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints; p4: Integer):
    BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL; stdcall;
```

Then, in the implementation portion, instead of providing each function's code, the unit refers to the external definition in a DLL:

```
const
    gdi32 = 'gdi32.dll';

function PlayMetaFile; external gdi32 name 'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name 'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

**Note** *Windows.PAS* makes heavy use of the `{$EXTERNALSYM identifier}` directive. This directive has little to do with Delphi itself; it applies to C++Builder. The symbol prevents the corresponding Delphi symbol from appearing in the C++ translated header file. This action helps keep the Delphi and C++ identifiers in synch, so that code can be shared between the two languages.

The external definition of these functions refers to the name of the DLL they use. The name of the DLL must include the .DLL extension, or the program will not work under Windows NT/2000/XP (although it will work under Windows 9x). The other element is the name of the DLL function. The `name` directive is not necessary if the Delphi function (or procedure) name matches the DLL function name (which is case sensitive).

To call a function that resides in a DLL, you can provide its declaration in the interface section of a unit and external definition in the implementation section, as shown earlier, or you can merge the two in a single declaration in the implementation section of a unit. Once the function is properly defined, you can call it in your Delphi application code just like any other function.

**Tip** Delphi includes the Delphi language translation of a large number of Windows APIs, as you can see in the many files available in Delphi's `Source\Rtl\Win` folder. More Delphi units referring to other APIs are available as part of the Delphi Jedi project at [www.delphi-jedi.org](http://www.delphi-jedi.org).

## Using a C++ DLL

As an example, I've written a DLL in C++ with some trivial functions, just to show you how to call DLLs from a Delphi application. I won't explain the C++ code in detail (it's basically C code) but will focus instead on the calls between the Delphi application and the C++ DLL. In Delphi programming, it is common to use DLLs written in C or C++.

Suppose you are given a DLL built in C or C++. You'll generally have in your hands a .DLL file (the compiled library), an .H file (the declaration of the functions inside the library), and a .LIB file (another version of the list of exported functions for the C/C++ linker). This LIB file is useless in Delphi; the DLL file is used as-is, and the H file must be translated into a Delphi unit with the corresponding declarations.

In the following listing, you can see the declaration of the C++ functions I've used to build the CppDll library example. The complete source code and the compiled version of the C++ DLL and the source code of the Delphi application using it are in the `CppDll` directory. You should be able to compile this code with any C++ compiler; I've tested it only

with Borland C++Builder. Here are the C++ declarations of the functions:

```
extern "C" __declspec(dllexport)
int WINAPI Double (int n);
extern "C" __declspec(dllexport)
int WINAPI Triple (int n);
__declspec(dllexport)
int WINAPI Add (int a, int b);
```

The three functions perform some basic calculations on the parameters and return the result. Notice that all the functions are defined with the `WINAPI` modifier, which sets the proper parameter-calling convention; they are preceded by the `__declspec(dllexport)` declaration, which makes the functions available to the outside world.

Two of these C++ functions also use the C naming convention (indicated by the `extern "C"` statement), but the third one, `Add`, doesn't. This difference affects the way you call these functions in Delphi. The internal names of the first two functions correspond to their names in the C++ source code file. But because I didn't use the `extern "C"` clause for the `Add` function, the C++ compiler uses *name mangling*. This technique is used to include information about the number and type of parameters in the function name, which the C++ language requires in order to implement function overloading. The result when using the Borland C++ compiler is a funny function name: `@Add$qqsii`. You must use this name in the Delphi code to call the `Add` DLL function (which explains why you'll generally avoid C++ name mangling in exported functions and declare them all as `extern "C"`). The following are the declarations of the three functions in the Delphi CallCpp example:

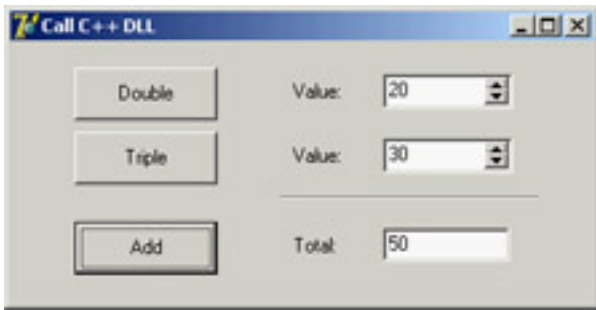
```
function Add (A, B: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name '@Add$qqsii';
function Double (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name 'Double';
function Triple (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL';
```

As you can see, you can either provide or omit an alias for an external function. I've provided one for the first function (there was no alternative, because the exported DLL function name `@Add$qqsii` is not a valid Delphi identifier) and for the second, although in the second case it was unnecessary. If the two names match, you can omit the `name` directive, as I did for the third function. If you are not sure of the actual names of the functions exported by the DLL, you can use Borland's `TDump` command-line program, available in the Delphi `BIN` folder, using the `-ee` command-line switch.

Remember to add the `stdcall` directive to each definition, so that the caller module (the application) and the module being called (the DLL) use the same parameter-passing convention. If you fail to do so, you will get unpredictable values passed as parameters, a bug that is very hard to trace.

**Note** When you have to convert a large C/C++ header file to the corresponding Delphi declarations, instead of doing a manual conversion you can use a tool to partially automate the process. One of these tools is `HeadConv`, written by Bob Swart. You'll find a copy on his website, [www.drbob42.com](http://www.drbob42.com). The tool is being extended by Project Jedi, under the name of DARTH project ([www.delphi-jedi.org/team\\_darth\\_home](http://www.delphi-jedi.org/team_darth_home)). Notice, though, that automatic header translation from C/C++ to Delphi is not possible; the Delphi language is more strongly typed than C/C++, so you have to use types more precisely.

To use this C++ DLL, I've built a Delphi example named `CallCpp`. Its form has only the buttons used to call the functions of the DLL and some visual components for input and output parameters (see [Figure 10.1](#)). Notice that to run this application, you should have the DLL in the same directory as the project, in one of the directories on the path, or in the Windows main folder (`\Windows, \WinNT...`) or the Windows' system folder (`\Windows\System, \WinNT\System32...`). If you move the executable file to a new directory and try to run it, you'll get a run-time error indicating that the DLL is missing:



**Figure 10.1:** The output of the CallCpp example when you have clicked each of the buttons

## Creating a DLL in Delphi

Besides using DLLs written in other environments, you can use Delphi to build DLLs that can be used by Delphi programs or with any other development tool that supports DLLs. Building DLLs in Delphi is so easy that you might overuse this feature. In general, I suggest you try to build packages instead of plain DLLs. As I'll discuss later in this chapter, packages often contain components, but they can also include plain noncomponent classes, allowing you to write object-oriented code and to reuse it effectively. Of course, packages can contain also simple routines, constants, variables, etc.

As I've already mentioned, building a DLL is useful when a portion of a program's code is subject to frequent changes. In this case, you can often replace the DLL, keeping the rest of the program unchanged. Similarly, when you need to write a program that provides different features to different groups of users, you can distribute different versions of a DLL to those users.

### Your First Delphi DLL

As a starting point in exploring the development of DLLs in Delphi, I'll show you a library built in Delphi. The primary focus of this example will be to show the syntax you use to define a DLL in Delphi, but it will also illustrate a few considerations involved in passing string parameters. To start, select the File → New → Other command and choose the DLL option in the New page of the Object Repository. Doing so creates a very simple source file that begins with the following definition:

```
library Project1;
```

The `library` statement indicates that you want to build a DLL instead of an executable file. Now you can add routines to the library and list them in an `exports` statement:

```
function Triple (N: Integer): Integer; stdcall;
begin
  try
    Result := N * 3;
  except
    Result := -1;
  end;
end;

function Double (N: Integer): Integer; stdcall;
begin
  try
    Result := N * 2;
  except
    Result := -1;
  end;
end;

exports
  Triple, Double;
```

In this basic version of the DLL, you don't need a `uses` statement; but in general, the main project file includes only the `uses` and `exports` statements, whereas the function declarations are placed in a separate unit. In the final source code of the FirstDll example, I've changed the code slightly from the version listed here, to show a message each time a function is called. You can accomplish this two ways; the simplest is to use the Dialogs unit and call the `ShowMessage` function.

The code requires Delphi to link a lot of VCL code into the application. If you statically link the VCL into this DLL, the resulting size will be a few hundred KB. The reason is that the `ShowMessage` function displays a VCL form that



contains VCL controls and uses VCL graphics classes; those indirectly refer to things like the VCL streaming system and the VCL application and screen objects. In this case, a better alternative is to show the messages using direct API calls, using the Windows unit and calling the `MessageBox` function, so that the VCL code is not required. This code change brings the size of the application down to less than 50 KB.

**Note** This huge difference in size underlines the fact that you should not overuse DLLs in Delphi, to avoid compiling the VCL code in multiple executable files. Of course, you can reduce the size of a Delphi DLL by using run-time packages, as detailed later in this chapter.

If you run a test program like the `CallFirst` example (described later) using the API-based version of the DLL, its behavior won't be correct. In fact, you can click the buttons that call the DLL functions several times without first closing the message boxes displayed by the DLL. This happens because the first parameter of the `MessageBox` API call is zero. Its value should instead be the handle of the program's main form or the application form—information you don't have at hand in the DLL.

## Overloaded Functions in Delphi DLLs

When you create a DLL in C++, overloaded functions use name mangling to generate a different name for each function. The type of the parameters is included right in the name, as you saw in the `CppDll` example.

When you create a DLL in Delphi and use overloaded functions (that is, multiple functions using the same name and marked with the `overload` directive), Delphi allows you to export only one of the overloaded functions with the original name, indicating its parameters list in the `exports` clause. If you want to export multiple overloaded functions, you should specify different names in the `exports` clause to distinguish the overloads. This technique is demonstrated by this portion of the `FirstDLL` code:

```
function Triple (C: Char): Integer; stdcall; overload;  
function Triple (N: Integer): Integer; stdcall; overload;
```

```
exports  
  Triple (N: Integer),  
  Triple (C: Char) name 'TripleChar';
```

**Note** The reverse is possible as well: You can import a series of similar functions from a DLL and define them all as overloaded functions in the Delphi declaration. Delphi's `OpenGL.PAS` unit contains a series of examples of this technique.

## Exporting Strings from a DLL

In general, functions in a DLL can use any type of parameter and return any type of value. There are two exceptions to this rule:

- If you plan to call the DLL from other programming languages, you should try using Windows native data types instead of Delphi-specific types. For example, to express color values, you should use integers or the Windows `ColorRef` type instead of the Delphi native `TColor` type, doing the appropriate conversions (as in the `FormDLL` example, described in the [next section](#)). For compatibility, you should avoid using some other Delphi types, including objects (which cannot be used by other languages) and Delphi strings (which can be replaced by `PChar` strings). In other words, every Windows development environment must support the basic types of the API, and if you stick to them, your DLL will be usable with other development environments. Also, Delphi file variables (text files and binary file of record) should not be passed out of DLLs, but you can use Win32 file handles.
- Even if you plan to use the DLL only from a Delphi application, you cannot pass Delphi strings (and dynamic arrays) across the DLL boundary without taking some precautions. This is the case because of the way Delphi manages strings in memory—allocating, reallocating, and freeing them automatically. The solution to the problem is to include the `ShareMem` system unit both in the DLL and in the program using it. This unit must be included as the first unit of each of the projects. Moreover, you have to deploy the `BorlndMM.DLL` file (the name stands for Borland Memory Manager) along with the program and the specific library.

In the FirstDLL example, I've included both approaches: One function receives and returns a Delphi string, and another receives as parameter a PChar pointer, which is then filled by the function. The first function is written as usual in Delphi:

```
function DoubleString (S: string; Separator: Char): string; stdcall;
begin
    try
        Result := S + Separator + S;
    except
        Result := '[error]';
    end;
end;
```

The second function is quite complex because PChar strings don't have a simple + operator, and they are not directly compatible with characters; the separator must be turned into a string before being adding. Here is the complete code; it uses input and output PChar buffers, which are compatible with any Windows development environment:

```
function DoublePChar (BufferIn, BufferOut: PChar;
    BufferOutLen: Cardinal; Separator: Char): LongBool; stdcall;
var
    SepStr: array [0..1] of Char;
begin
    try
        // if the buffer is large enough
        if BufferOutLen > StrLen (BufferIn) * 2 + 2 then
            begin
                // copy the input buffer in the output buffer
                StrCopy (BufferOut, BufferIn);
                // build the separator string (value plus null terminator)
                SepStr [0] := Separator;
                SepStr [1] := #0;
                // append the separator
                StrCat (BufferOut, SepStr);
                // append the input buffer once more
                StrCat (BufferOut, BufferIn);
                Result := True;
            end
        else
            // not enough space
            Result := False;
        except
            Result := False;
        end;
    end;
```

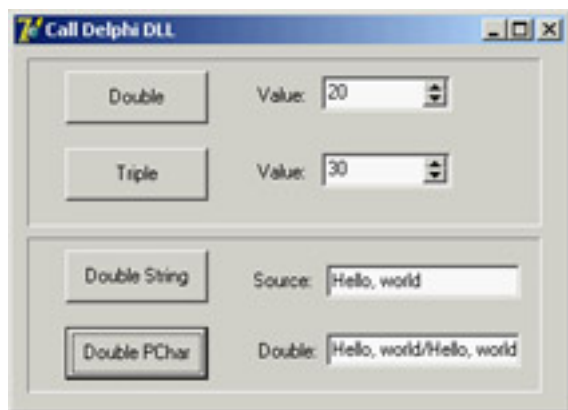
This second version of the code is certainly more complex, but the first can be used only from Delphi. Moreover, the first version requires you to include the ShareMem unit and to deploy the file BorlndMM.DLL, as discussed earlier.

## Calling the Delphi DLL

How can you use the library you've just built? You can call it from within another Delphi project or from other environments. As an example, I've built the CallFrst project (stored in the FirstDLL directory). To access the DLL functions, you must declare them as external, as with the C++ DLL. This time, however, you can copy and paste the definition of the functions from the source code of the Delphi DLL, adding the external clause, as follows:

```
function Double (N: Integer): Integer;
    stdcall; external 'FIRSTDLL.DLL';
```

This declaration is similar to those used to call the C++ DLL. This time, however, you have no problems with function names. Once they are redeclared as external, the functions of the DLL can be used as if they were local functions. Here are two examples, with calls to the string-related functions (an example of the output is visible in [Figure 10.2](#)):



**Figure 10.2:** The output of the CallFirst example, which calls the DLL you've built in Delphi

```
procedure TForm1.BtnDoubleStringClick(Sender: TObject);
begin
    // call the DLL function directly
    EditDouble.Text := DoubleString (EditSource.Text, ';');
end;

procedure TForm1.BtnDoublePCharClick(Sender: TObject);
var
    Buffer: string;
begin
    // make the buffer large enough
    SetLength (Buffer, 1000);
    // call the DLL function
    if DoublePChar (PChar (EditSource.Text), PChar (Buffer), 1000, '/') then
        EditDouble.Text := Buffer;
end;
```

## Advanced Features of Delphi DLLs

Beside this introductory example, you can do a few extra things with dynamic libraries in Delphi. You can use some new compiler directives to affect the name of the library, you can call a DLL at run time, and you can place entire Delphi forms inside a dynamic library. These are the topics of the following sections.

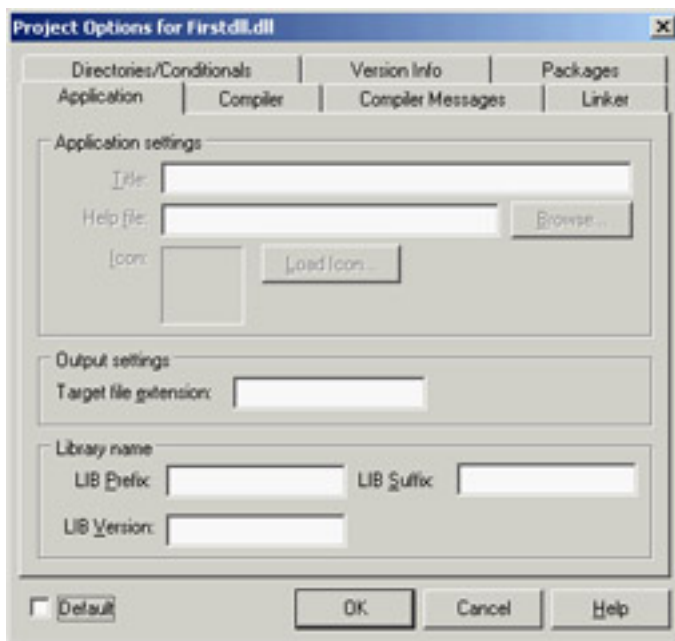
### Changing Project and Library Names

For a library, as for a standard application, you end up with a library name matching a Delphi project filename. Following a technique similar to that introduced in Kylix for compatibility with standard Linux naming conventions for shared object libraries (the Linux equivalent of Windows DLLs), Delphi 6 introduced special compiler directives you can use in libraries to determine their executable filename. Some of these directives make more sense in the Linux world than on Windows, but they've all been added anyway:

- `$LIBPREFIX` is used to add something in front of the library name. Paralleling the Linux technique of adding *lib* in front of library names, this directive is used by Kylix to add *bpl* at the beginning of package names. It is necessary because Linux uses a single extension (`.SO`) for libraries, whereas in Windows you can have different library extensions, something Borland uses for packages (`.BPL`).
- `$LIBSUFFIX` is used to add text after the library name and before the extension. This text can be used to specify versioning information or other variations on the library name and can be quite useful on Windows.
- `$LIBVERSION` is used to add a version number after the extension—something very common in Linux, but that you should generally avoid on Windows.

These directives can be set in the IDE from the Application page of the Project Options dialog box, as you can see in [Figure 10.3](#). As an example, consider the following directives, which generate a library called `MarcoNameTest60.dll`:

```
library NameTest ;  
{ $LIBPREFIX 'Marco' }  
{ $LIBSUFFIX '60' }
```



**Figure 10.3:** The Application page of the Project Options dialog box now has a Library Name section.

**Note** Delphi 6 packages introduced the extensive use of the `$LIBSUFFIX` directive. For this reason, the VCL package now generates the `VCL.DCP` file and the `VCL70.BPL` file. The advantage of this approach is that you won't need to change the *requires* portions of your packages for every new version of Delphi. Of course, this is helpful when you move projects from Delphi 6 to Delphi 7, because past versions of Delphi didn't provide this feature. When you reopen Delphi 5 packages you still have to upgrade their source code, an operation the Delphi IDE does automatically for you.

## Calling a DLL Function at Run Time

Up to now, you've referenced in your code the functions exported by the libraries, so the DLLs were loaded along with the program. I mentioned earlier that you can also delay the loading of a DLL until the moment it is needed, so you can use the rest of the program in case the DLL is not available.

Dynamic loading of a DLL in Windows is accomplished by calling the `LoadLibrary` API function, which searches for the DLL in the program folder, in the folders on the path, and in some system folders. If the DLL is not found, Windows will show an error message, something you can skip by calling Delphi's `SafeLoadLibrary` function. This function has the same effect as the API it encapsulates, but it suppresses the standard Windows error message and should be the preferred way to load libraries dynamically in Delphi.

If the library is found and loaded (something you know by checking the return value of `LoadLibrary` or `SafeLoadLibrary`), a program can call the `GetProcAddress` API function, which searches the DLL's exports table, looking for the name of the function passed as a parameter. If `GetProcAddress` finds a match, it returns a pointer to the requested procedure. Now you can cast this function pointer to the proper data type and call it.

Whichever loading functions you've used, don't forget to call `FreeLibrary` at the end, so that the DLL can be properly released from memory. In fact, the system uses a reference-counting technique for libraries, releasing them when each loading request has been followed by a freeing request.

The example I've built to show dynamic DLL loading is named `DynaCall`. It uses the `FirstDLL` library built earlier in this chapter (to make the program work, you have to copy the DLL from its source folder into the folder as the `DynaCall` example). Instead of declaring the `Double` and `Triple` functions and using them directly, this example obtains the same effect with somewhat more complex code. The advantage, however, is that the program will run even without the DLL. Also, if new *compatible* functions are added to the DLL, you won't have to revise the program's source code and recompile it to access those new functions. Here is the core code of the program:

```
type
  TIntFunction = function (I: Integer): Integer; stdcall;

const
  DllName = 'Firstdll.dll';

procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFunct: TIntFunction;
begin
  HInst := SafeLoadLibrary (DllName);
  if HInst > 0 then
    try
      FPointer := GetProcAddress (HInst,
        PChar (Edit1.Text));
      if FPointer <> nil then
        begin
          MyFunct := TIntFunction (FPointer);
          SpinEdit1.Value := MyFunct (SpinEdit1.Value);
        end
    end
```

```

else
    ShowMessage (Edit1.Text + ' DLL function not found');
finally
    FreeLibrary (HInst);
end
else
    ShowMessage (DllName + ' library not found');
end;

```

**Warning** As the library uses the Borland memory manager, the program dynamically loading it must do the same. So you need to add the `ShareMem` unit in the project of the `DynaCall` example. Oddly enough, this was not so with past versions of Delphi, in case the library didn't effectively use strings. Be warned that if you omit this inclusion, you'll get a harsh system error, which can even stall the debugger on the `FreeLibrary` call.

How do you call a procedure in Delphi, once you have a pointer to it? One solution is to convert the pointer to a procedural type and then call the procedure using the procedural-type variable, as in the previous listing. Notice that the procedural type you define must be compatible with the definition of the procedure in the DLL. This is the Achilles' heel of this method—there is no actual check of the parameter types.

What is the advantage of this approach? In theory, you can use it to access any function of any DLL at any time. In practice, it is useful when you have different DLLs with compatible functions or a single DLL with several compatible functions, as in this case. You can call the `Double` and `Triple` methods by entering their names in the edit box. Now, if someone gives you a DLL with a new function receiving an integer as a parameter and returning an integer, you can call it by entering its name in the edit box. You don't even need to recompile the application.

With this code, the compiler and the linker ignore the existence of the DLL. When the program is loaded, the DLL is not loaded immediately. You might make the program even more flexible and let the user enter the name of the DLL to use. In some cases, this is a great advantage. A program may switch DLLs at run time, something the direct approach does not allow. Note that this approach to loading DLL functions is common in macro languages and is used by many visual programming environments.

Only a system based on a compiler and a linker, such as Delphi, can use the direct approach, which is generally more reliable and also a little faster. In my opinion, the indirect loading approach of the `DynaCall` example is useful only in special cases, but it can be extremely powerful. On the other hand, I see a lot of value in using dynamic loading for packages including forms, as you'll see toward the end of this chapter.

## Placing Delphi Forms in a Library

Besides writing a library with functions and procedures, you can place a complete form built with Delphi into a dynamic library. This can be a dialog box or any other kind of form, and it can be used not only by other Delphi programs, but also by other development environments or macro languages with the ability to use dynamic link libraries. Once you've created a new library project, all you need to do is add one or more forms to the project and then write exported functions that will create and use those forms.

For example, a function activating a modal dialog box to select a color could be written like this:

```

function GetColor (Col: LongInt): LongInt; cdecl;
var
    FormScroll: TFormScroll;
begin
    // default value
    Result := Col;
    try
        FormScroll := TFormScroll.Create (Application);
        try
            // initialize the data
            FormScroll.SelectedColor := Col;

```

```

    // show the form
    if FormScroll.ShowModal = mrOK then
        Result := FormScroll.SelectedColor;
    finally
        FormScroll.Free;
    end;
except
    on E: Exception do
        MessageDlg ('Error in library: ' + E.Message, mtError, [mbOK], 0);
    end;
end;

```

What makes this different from the code you generally write in a program is the use of exception handling:

- A `try/except` block protects the whole function. Any exception generated by the function will be trapped, and an appropriate message will be displayed. You handle every possible exception because the calling application might be written in any language—in particular, one that doesn't know how to handle exceptions. Even when the caller is a Delphi program, it is sometimes helpful to use the same protective approach.
- A `try/finally` block protects the operations on the form, ensuring that the form object will be properly destroyed even when an exception is raised.

By checking the return value of the `ShowModal` method, the program determines the result of the function. I've set the default value before entering the `try` block to ensure that it will always be executed (and also to avoid the compiler warning indicating that the result of the function might be undefined).

You can find this code snippet in the `FormDLL` and `UseCol` projects, available in the `FormDLL` folder. (There's also a `WORDCALL.TXT` file showing how to call the routine from a Word macro.). The example also shows that you can add a modeless form to the DLL, but doing so causes far too much trouble. The modeless form and the main form are not synchronized, because the DLL has its own global `Application` object in its own copy of the VCL. This situation can be partially fixed by copying the `Handle` of the application's `Application` object to the `Handle` of the library's `Application` object. Not all of the problems are solved with the code that you can find in the example. A better solution might be to compile the program and the library to use Delphi packages, so that the VCL code and data won't be duplicated. But this approach still causes a few troubles: it's generally advised that you don't use Delphi DLLs and packages together. So what is the best suggestion I can give you? For making the forms of a library available to other Delphi programs, use packages instead of plain DLLs!



## Libraries in Memory: Code and Data

Before I discuss packages, I want to focus on a technical element of dynamic libraries: how they use memory. Let's start with the code portion of the library, then we'll focus on its global data. When Windows loads the code of a library, like any other code module, it has to do a *fixup* operation. This *fixup* consists of patching addresses of jumps and internal function calls with the actual memory address where they've been loaded. The effect of this operation is that the code-loaded memory depends on where it has been loaded.

This is not an issue for executable files, but might cause a significant problem for libraries. If two executables load the same library at the same base address, there will be only one physical copy of the DLL code in the RAM (the physical memory) of the machine, thus saving memory space. If the second time the library is loaded the memory address is already in use, it needs to be *relocated*, that is, moved with a different *fixup* applied. So you'll end up with a second physical copy of the DLL code in RAM.

You can use the dynamic loading technique, based on the `GetProcAddress` API function, to test which memory address of the current process a function has been mapped to. The code is as follows:

```
procedure TForm1.Button3Click(Sender: TObject);
var
    HDLLInst: THandle;
begin
    HDLLInst := SafeLoadLibrary ('dllmem');
    Label1.Caption := Format ('Address: %p', [
        GetProcAddress (HDLLInst, 'SetData')]);
    FreeLibrary (HDLLInst);
end;
```

This code displays, in a label, the memory address of the function, within the address space of the calling application. If you run two programs using this code, they'll generally both show the same address. This technique demonstrates that the code is loaded only once at a common memory address.

Another technique to get more information about what's going on is to use Delphi's Modules window, which shows the base address of each library referenced by the module and the address of each function within the library, as shown here:

Name	Base Address	Path	Entry Point	Address
VERSION.dll	\$77820000	C:\WINDOWS\system32\w...	FreeTerminateProc	\$00807808
LZ32.dll	\$759B0000	C:\WINDOWS\system32\L...	InitDriveSpacePt	\$00807858
COMCTL32.dll	\$71780000	C:\WINDOWS\system32\c...	FreeAndNil	\$008078B4
Dllmem.dll	\$00800000	E:\books\lmd7\code\10\DI...	Finalization	\$00807B94
			SysUtils	\$00807D14
			SetData	\$00807D94
			GetData	\$00807DA4
			SetShareData	\$00807DAC
			GetShareData	\$00807DC0
			Finalization	\$00807DC8
			DIMemU	\$00807E10
			Finalization	\$00807E48
			dllmem	\$00807F18

It's important to know that the base address of a DLL is something you can request by setting the base address option. In Delphi this address is determined by the Image Base value in the linker page of the Project Options dialog box. In the `DllMem` library, for example, I've set it to `$00800000`. You need to have a different value for each of your libraries, verifying that it doesn't clash with any system library or other library (package, ActiveX, and so on) used by the executable. Again, this is something you can figure out using the Module window of the debugger.

Although this doesn't guarantee a unique placement, setting a base address for the library is always better than not setting one; in this case a relocation always takes place, but the chance that two different executables will relocate the same library at the same address are not high.



**Note** You can also use Process Explorer from <http://www.sysinternals.com> to examine any process on any machine. This tool even has an option to highlight relocated DLLs. Check the effect of running the same program with its libraries on different operating systems (Windows 2000, Windows XP, and Windows ME) and settle on an unused area.

This is the case for the DLL code, but what about the global data? Basically, each copy of the DLL has its own copy of the data, in the address space of the calling application. However, it is possible to share global data between applications using a DLL. The most common technique for sharing data is to use memory-mapped files. I'll use this technique for a DLL, but it can also be used to share data directly among applications.

This example is called DllMem for the library and UseMem for the demo application. The DLL code has a project file that exports four subroutines:

```
library dllmem;

uses
  SysUtils,
  DllMemU in 'DllMemU.pas';

exports
  SetData, GetData,
  GetShareData, SetShareData;
end.
```

The actual code is in the secondary unit (DllMemU.PAS), which contains the code for the four routines that read or write two global memory locations. These memory locations hold an integer and a pointer to an integer. Here are the variable declarations and the two Set routines:

```
var
  PlainData: Integer = 0; // not shared
  ShareData: ^Integer; // shared

procedure SetData (I: Integer); stdcall;
begin
  PlainData := I;
end;

procedure SetShareData (I: Integer); stdcall;
begin
  ShareData^ := I;
end;
```

## Sharing Data with Memory-Mapped Files

For the data that isn't shared, there isn't anything else to do. To access the shared data, however, the DLL has to create a memory-mapped file and then get a pointer to this memory area. These operations require two Windows API calls:

- `CreateFileMapping` requires as parameters the filename (or `$FFFFFFFF` to use a virtual file in memory), some security and protection attributes, the size of the data, and an internal name (which must be the same to share the mapped file from multiple calling applications).
- `MapViewOfFile` requires as parameters the handle of the memory-mapped file, some attributes and offsets, and the size of the data (again).

Here is the source code of the `initialization` section, which is executed every time the DLL is loaded into a new process space (that is, once for each application that uses the DLL):

```

var
  hMapFile: THandle;

const
  VirtualFileName = 'ShareDllData';
  DataSize = sizeof (Integer);

initialization
  // create memory mapped file
  hMapFile := CreateFileMapping ($FFFFFFFF, nil,
    Page_ReadWrite, 0, DataSize, VirtualFileName);
  if hMapFile = 0 then
    raise Exception.Create ('Error creating memory-mapped file');

  // get the pointer to the actual data
  ShareData := MapViewOfFile (
    hMapFile, File_Map_Write, 0, 0, DataSize);

```

When the application terminates and the DLL is released, it has to free the pointer to the mapped file and the file mapping:

```

finalization
  UnmapViewOfFile (ShareData);
  CloseHandle (hMapFile);

```

The UseMem demo program's form has four edit boxes (two with an UpDown control connected), five buttons, and a label. The first button saves the value of the first edit box in the DLL data, getting the value from the connected UpDown control:

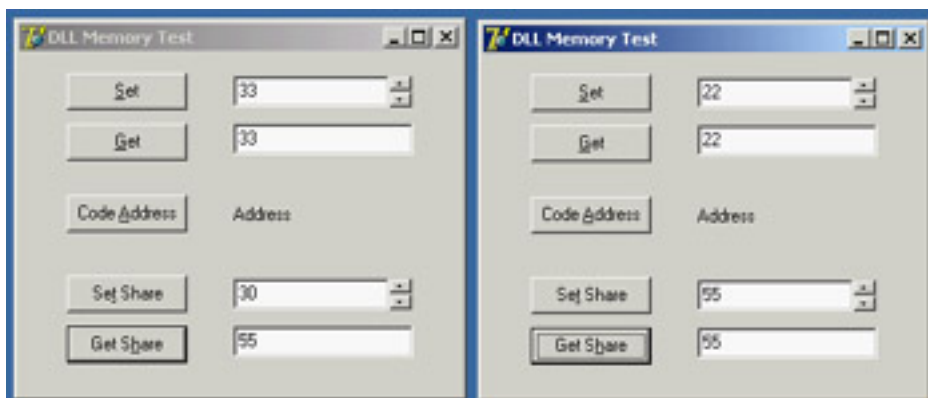
```
SetData (UpDown1.Position);
```

If you click the second button, the program copies the DLL data to the second edit box:

```
Edit2.Text := IntToStr(GetData);
```

The third button is used to display the memory address of a function, with the source code shown at the beginning of this section. The last two buttons have basically the same code as the first two, but they call the `SetShareData` procedure and the `GetShareData` function.

If you run two copies of this program, you can see that each copy has its own value for the plain global data of the DLL, whereas the value of the shared data is common. Set different values in the two programs and then get them in both, and you'll see what I mean. This situation is illustrated in [Figure 10.4](#).



**Figure 10.4:** If you run two copies of the UseMem program, you'll see that the global data in its DLL is not shared.

**Warning**

Memory-mapped files reserve a minimum of a 64 KB range of virtual addresses and consume physical memory in 4 KB pages. The example's use of 4-byte Integer data in shared memory is rather expensive, especially if you use the same approach for sharing multiple values. If you need to share several variables, you should place them all in a single shared memory area (accessing the different variables using pointers or building a record structure for all of them).

[Team LiB](#)[◀ PREVIOUS](#)[NEXT ▶](#)

## Using Delphi Packages

In Delphi, component packages are an important type of DLL. Packages allow you to bundle a group of components and then link the components either statically (adding their compiled code to the executable file of your application) or dynamically (keeping the component code in a DLL, the run-time package that you'll distribute along with your program, along with all other packages you will need). In [Chapter 9](#), "Writing Delphi Components," you saw how to build a package. Now I want to underline some advantages and disadvantages of the two forms of linking for a package. You need to keep many elements in mind:

- Using a package as a DLL makes the executable files much smaller.
- Linking the package units into the program allows you to distribute only part of the package code. The size of the executable file of an application plus the size of the required package DLLs is always much bigger than the size of the statically linked program. The linker includes only the code used by the program, whereas a package must link in all the functions and classes declared in the `interface` sections of all the units contained in the package.
- If you distribute several Delphi applications based on the same packages, you might end up distributing less code, because the run-time packages are shared. In other words, once the users of your application have the standard Delphi run-time packages, you can ship them very small programs.
- If you run several Delphi applications based on the same packages, you can save some memory space at run time; the code of the run-time packages is loaded in memory only once among the multiple Delphi applications.
- Don't worry too much about distributing a large executable file. Keep in mind that when you make minor changes to a program, you can use any of various tools to create a *patch file*, so that you distribute only a file containing the differences, not a complete copy of the files.
- If you place a few of your program's forms in a run-time package, you can share them among programs. When you modify these forms, however, you'll generally need to recompile the main program as well, and distribute both of them again to your users. The [next section](#) discusses this complex topic in detail.
- A package is a collection of compiled units (including classes, types, variables, routines), which don't differ at all from the units inside a program. The only difference is in the build process. The code of the package units and that of the units of the main program using them remains identical. This is arguably one of the key advantages of packages over DLLs.

## Package Versioning

A very important and often misunderstood element is the distribution of updated packages. When you update a DLL, you can ship the new version, and the executable programs requiring this DLL will still work (unless you've removed existing exported functions or changed some of their parameters).

When you distribute a Delphi package, however, if you update the package and modify the `interface` portion of any unit of the package, you may need to recompile all the applications that use the package. This step is required if you add methods or properties to a class, but not if you add new global symbols (or modify anything not used by client applications). There is no problem if you make changes affecting only the `implementation` section of the package's units.

A DCU file in Delphi has a version tag based on its timestamp and a checksum computed from the interface portion of the unit. When you change the `interface` portion of a unit, every other unit based on it should be recompiled. The compiler compares the timestamp and checksum of the unit from previous compilations with the new timestamp and checksum, and decides whether the dependent unit must be recompiled. For this reason, you must recompile each unit when you get a new version of Delphi that has modified system units.

In Delphi 3 (when packages were first introduced), the compiler added an extra entry function to the package library named with a checksum of the package, obtained from the checksum of the units it contained and the checksum of the

packages it required. This checksum function was then called by programs using the package so that an older executable would fail at startup.

Delphi 4 and following versions up to Delphi 7 have relaxed the run-time constraints of the package. (The design-time constraints on DCU files remain identical, though.) The checksum of the packages is no longer checked, so you can directly modify the units that are part of a package and deploy a new version of the package to be used with the existing executable file. Because methods are referenced by name, you cannot remove any existing method. You cannot even change its parameters, because of name-mangling techniques that protect a package's method against changes in parameters.

Removing a method referenced from the calling program will stop the program during the loading process. If you make other changes, however, the program might fail unexpectedly during its execution. For example, if you replace a component placed on a form compiled in a package with a similar component, the calling program might still be able to access the component in that memory location, although it is now different!

If you decide to follow this treacherous road of changing the interface of units in a package without recompiling all the programs that use it, you should at least limit your changes. When you add new properties or nonvirtual methods to the form, you should be able to maintain full compatibility with existing programs already using the package. Also, adding fields and virtual methods might affect the internal structure of the class, leading to problems with existing programs that expect a different class data and virtual method table (VMT) layout.

**Warning** Here I'm referring to the distribution of compiled programs divided between EXEs and packages, not to the distribution of components to other Delphi developers. In this latter case the versioning rules are more stringent, and you must take extra care in package versioning.

Having said this, I recommend never changing the interface of any unit exported by your packages. To accomplish this, you can add to your package a unit with form-creation functions (as in the DLL with forms presented earlier) and use it to access another unit, which defines the form. Although there is no way to *hide* a unit that is linked into a package, if you never directly use the class defined in a unit, but use it only through other routines, you'll have more flexibility in modifying it. You can also use form inheritance to modify a form within a package without affecting the original version.

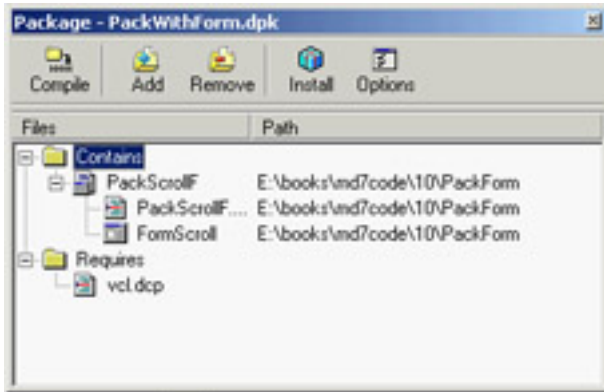
The most stringent rule for packages is the following one used by component writers: For long-term deployment and maintenance of code in packages, plan on having a major release with minor maintenance releases. A major release of your package will require all client programs to be recompiled from source; the package file should be renamed with a new version number, and the interface sections of units can be modified. Maintenance releases of that package should be restricted to implementation changes to preserve full compatibility with existing executables and units, as is generally done by Borland with its Update Packs.

## Forms Inside Packages

In [Chapter 9](#), I discussed the use of component packages in Delphi applications. Now I'm discussing the use of packages and DLLs for partitioning an application, so I'll begin talking about the development of packages holding forms. I've mentioned earlier in this chapter that you can use forms inside DLLs, but doing so causes quite a few problems. If you are building both the library and the executable file in Delphi, using packages results in a much better and cleaner solution.

At first sight, you might believe that Delphi packages are solely a way to distribute components to be installed in the environment. However, you can also use packages as a way to structure your code but, unlike DLLs, retain the full power of Delphi's OOP. Consider this: A package is a collection of compiled units, and your program uses several units. The units the program refers to will be compiled inside the executable file, unless you ask Delphi to place them inside a package. As discussed earlier, this is one of the main reasons for using packages.

To set up an application so that its code is split among one or more packages and a main executable file, you only need to compile some of the units in a package and then set up the options of the main program to dynamically link this package. For example, I made a copy of the "usual" color selection form and renamed its unit `PackScrollF`; then I created a new package and added the unit to it, as you can see in [Figure 10.5](#).



**Figure 10.5:** The structure of the package hosting a form in Delphi's Package Editor

Before compiling this package, you should change its default output directories to refer to the current folder, not the standard `/Projects/Bpl` subfolder of Delphi. To do this, go to the `Directories/Conditional` page of the package Project Options, and set the current directory (a single dot, for short) for the Output directory (for the BPL) and DCP output directory. Then compile the package and do not install it in Delphi—there's no need to.

At this point, you can create a normal application and write the standard code you'll use in a program to show a secondary form, as in the following listing:

```
uses
  PackScrollF;

procedure TForm1.BtnChangeClick(Sender: TObject);
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  try
    // initialize the data
    FormScroll.SelectedColor := Color;
    // show the form
    if FormScroll.ShowModal = mrOK then
      Color := FormScroll.SelectedColor;
  finally
```

```

    FormScroll.Free;
end;
end;

procedure TForm1.BtnSelectClick(Sender: TObject);
var
    FormScroll: TFormScroll;
begin
    FormScroll := TFormScroll.Create (Application);
    // initialize the data and UI
    FormScroll.SelectedColor := Color;
    FormScroll.BitBtn1.Caption := 'Apply';
    FormScroll.BitBtn1.OnClick := FormScroll.ApplyClick;
    FormScroll.BitBtn2.Kind := bkClose;
    // show the form
    FormScroll.Show;
end;

```

One of the advantages of this approach is that you can refer to a form compiled into a package with the same code you'd use for a form compiled in the program. If you compile this program, the unit of the form will be bound to the program. To keep the form's unit in the package, you'll have to use run-time packages for the application and manually add the `PackWithForm` package to the list of run-time packages (this is not suggested by the Delphi IDE, because you have not installed the package in the development environment).

Once you've performed this step, compile the program; it will behave as usual. But now the form is in a DLL package, and you can modify the form in the package, recompile it, and run the application to see the effects. Notice, though, that for most changes affecting the `interface` portion of the package's units (for example, adding a component or a method to the form), you should also recompile the executable program calling the package.

**Note** You can find the package and the program testing it in the `PackForm` folder of the source code related to the current chapter. The code for the next example is in the same folder. The package and projects are all referenced by the project group (BPG) file within the folder.

## Loading Packages at Run Time

In the previous example, I indicated that the `PackWithForm` package is a run-time package to be used by the application. This means the package is required to run the application and is loaded when the program starts, just as with the typical use of DLLs. You can avoid both aspects by loading the package dynamically, as you've done with DLLs. The resulting program will be more flexible, start more quickly, and use less memory.

An important element to keep in mind is that you'll need to call the `LoadPackage` and `UnloadPackage` Delphi functions rather than the `LoadLibrary/SafeLoadLibrary` and `FreeLibrary` Windows API functions. The functions provided by Delphi load the packages, but they also call their proper initialization and finalization code.

Besides this important element—which is easy to accomplish once you know about it—the program will require some extra code, because you cannot refer from the main program to the unit hosting the form. You cannot use the form class directly, nor access its properties or components—at least, not with the standard Delphi code. Both issues, however, can be solved using class references, class registration, and RTTI (run-time type information).

Let me begin with the first approach. In the form unit, in the package, I've added this initialization code:

```

initialization
    RegisterClass (TFormScroll);

```

As the package is loaded, the main program can use Delphi's `GetClass` function to get the class reference of the registered class and then call the `Create` constructor for this class reference.

To solve the second problem, I've made the `SelectedColor` property of the form in the package a published property, so that it is accessible via RTTI. Then I've replaced the code accessing this property (`FormScroll.Color`) with the following:

```
SetPropValue (FormScroll, 'SelectedColor', Color);
```

Summing up all these changes, here is the code used by the main program (the `DynaPackForm` application) to show the modal form from the dynamically loaded package:

```
procedure TForm1.BtnChangeClick(Sender: TObject);  
var  
    FormScroll: TForm;  
    FormClass: TFormClass;  
    HandlePack: HModule;  
begin  
    // try to load the package  
    HandlePack := LoadPackage ('PackWithForm.bpl');  
    if HandlePack > 0 then  
        begin  
            FormClass := TFormClass(GetClass ('TFormScroll'));  
            if Assigned (FormClass) then  
                begin  
                    FormScroll := FormClass.Create (Application);  
                    try  
                        // initialize the data  
                        SetPropValue (FormScroll, 'SelectedColor', Color);  
                        // show the form  
                        if FormScroll.ShowModal = mrOK then  
                            Color := GetPropValue (FormScroll, 'SelectedColor');  
                    finally  
                        FormScroll.Free;  
                    end;  
                end  
            else  
                ShowMessage ('Form class not found');  
                UnloadPackage (HandlePack);  
            end  
        else  
            ShowMessage ('Package not found');  
        end;
```

Notice that the program unloads the package as soon as it is done with it. This step is not compulsory. I could have moved the `UnloadPackage` call in the `OnDestroy` handler of the form, and avoided reloading the package after the first time.

Now you can try running this program without the package available. You'll see that it starts properly, only to complain that it cannot find the package as you click the Change button. In this program, you don't need to use run-time packages to keep the unit outside your executable file, because you are not referring to the unit in your code. Also, the `PackWithForm` package doesn't need to be listed in the run-time packages. However, you must use run-time packages for it to work at all, or else your program will include VCL global variables (such as the `Application` object) and the dynamically loaded package will include another version, because it will refer to the VCL packages anyway.



**Warning** When a program that loads a package dynamically is closed, you may experience access violations. Frequently, they occur because an object whose class is defined in the package is kept in memory even after the package is unloaded. When the program shuts down, it may try to free that object by calling the *Destroy* method of a non-existing VMT, and thus cause the error. Having said this, I know by experience that these types of errors are very difficult to track and fix. I suggest that you make sure to destroy all the objects before unloading the package.

## Using Interfaces in Packages

Accessing forms' classes by means of methods and properties is much simpler than using RTTI all over the place. To build a larger application, I definitely try to use interfaces and to have multiple forms, each implementing a few standard interfaces defined by the program. An example cannot really do justice to this type of architecture, which becomes relevant for a large program, but I've tried to build a program to show how this idea can be applied in practice.

**Note** If you don't know much about interfaces, I suggest you refer to the related portion of [Chapter 2](#), "The Delphi Programming Language," before reading this section.

To build the IntfPack project, I've used three packages plus a demo application. Two of the three packages (IntfFormPack and IntfFormPack2) define alternative forms used to select a color. The third package (IntfPack) hosts a shared unit, used by both other packages. This unit includes the definition of the interface. I couldn't add it to both other packages because you cannot load two packages that have the same name with a unit (even by run-time loading).

The IntfPack package's only file is the IntfColSel unit, displayed in [Listing 10.1](#). This unit defines the common interface (you'll probably have a number of them in real-world applications) plus a list of registered classes; it mimics Delphi's *RegisterClass* approach, but makes available the complete list so that you can easily scan it.

### Listing 10.1: The IntfColSel Unit of the IntfPack Package

---

```
unit IntfColSel;

interface

uses
  Graphics, Contnrs;

type
  IColorSelect = interface
    ['{3F961395-71F6-4822-BD02-3B475FF516D4}']
    function Display (Modal: Boolean = True): Boolean;
    procedure SetSelColor (Col: TColor);
    function GetSelColor: TColor;
    property SelColor: TColor
      read GetSelColor write SetSelColor;
  end;

procedure RegisterColorSelect (AClass: TClass);

var
  ClassesColorSelect: TClassList;

implementation

procedure RegisterColorSelect (AClass: TClass);
begin
  if ClassesColorSelect.IndexOf (AClass) < 0 then
    ClassesColorSelect.Add (AClass);
end;
```

### initialization

```
ClassesColorSelect := TClassList.Create;
```

### finalization

```
ClassesColorSelect.Free;
```

```
end.
```

---

Once you have this interface available, you can define forms that implement it, as in the following example taken from IntfFormPack:

### type

```
TFormSimpleColor = class(TForm, IColorSelect)
    ...
private
    procedure SetSelColor (Col: TColor);
    function GetSelColor: TColor;
public
    function Display (Modal: Boolean = True): Boolean;
```

The two access methods read and write the value of the color from some components of the form (a ColorGrid control in this case), whereas the Display method internally calls either Show or ShowModal, depending on the parameter:

```
function TFormSimpleColor.Display(Modal: Boolean): Boolean;
```

### begin

```
Result := True; // default
if Modal then
    Result := (ShowModal = mrOK)
else
begin
    BitBtn1.Caption := 'Apply';
    BitBtn1.OnClick := ApplyClick;
    BitBtn2.Kind := bkClose;
    Show;
end;
end;
```

As you can see from this code, when the form is modeless the OK button is turned into an Apply button. Finally, the unit has the registration code in the initialization section, so that it is executed when the package is dynamically loaded:

```
RegisterColorSelect (TFormSimpleColor);
```

The second package, IntfFormPack2, has a similar architecture but a different form. You can look it up in the source code (I've not discussed the second form here as its code doesn't add much to the structure of the example).

With this architecture in place, you can build a rather elegant and flexible main program, which is based on a single form. When the form is created, it defines a list of packages (HandlesPackages) and loads them all. I've hard-coded the package in the code of the example, but of course you can search for the packages of the current folder or use a configuration file to make the application structure more flexible. After loading the packages, the program shows the registered classes in a list box. This is the code of the LoadDynaPackage and FormCreate methods:

```
procedure TFormUseIntf.FormCreate(Sender: TObject);
```

### var

```
I: Integer;
```

### begin

```
// loads all runtime packages
```

```

HandlesPackages := TList.Create;
LoadDynaPackage ('IntfFormPack.bpl');
LoadDynaPackage ('IntfFormPack2.bpl');

// add class names and select the first
for I := 0 to ClassesColorSelect.Count - 1 do
  lbClasses.Items.Add (ClassesColorSelect [I].ClassName);
lbClasses.ItemIndex := 0;
end;

procedure TFormUseIntf.LoadDynaPackage(PackageName: string);
var
  Handle: HModule;
begin
  // try to load the package
  Handle := LoadPackage (PackageName);
  if Handle > 0 then
    // add to the list for later removal
    HandlesPackages.Add (Pointer(Handle))
  else
    ShowMessage ('Package ' + PackageName + ' not found');
end;

```

The main reason for keeping the list of package handles is to be able to unload them all when the program ends. You don't need these handles to access the forms defined in those packages; the run-time code used to create and show a form uses the corresponding component classes. This is a snippet of code used to display a modeless form (an option controlled by a check box):

```

var
  AComponent: TComponent;
  ColorSelect: IColorSelect;
begin
  AComponent := TComponentClass
    (ClassesColorSelect[LbClasses.ItemIndex]).Create (Application);
  ColorSelect := AComponent as IColorSelect;
  ColorSelect.SelColor := Color;
  ColorSelect.Display (False);

```

The program uses the `Supports` function to check that the form really does support the interface before using it, and also accounts for the modal version of the form; but its essence is properly depicted in the preceding four statements.

By the way, notice that the code doesn't require a form. A nice exercise would be to add to the architecture a package with a component encapsulating the color selection dialog box or inheriting from it.

**Warning** The main program refers to the unit hosting the interface definition but should not link this file in. Rather, it should use the run-time package containing this unit, as the dynamically loaded packages do. Otherwise the main program will use a different copy of the same code, including a different list of global classes. It is this list of global classes that should not be duplicated in memory.

## The Structure of a Package

You may wonder whether it is possible to know if a unit has been linked in the executable file or if it's part of a run-time package. Not only is this possible in Delphi, but you can also explore the overall structure of an application. A component can use the undocumented `ModuleIsPackage` global variable, declared in the `SysInit` unit. You should never need this variable, but it is technically possible for a component to have different code depending on whether it is packaged. The following code extracts the name of the run-time package hosting the component, if any:

```
var
  fPackageName: string;
begin
  // get package name
  SetLength (fPackageName, 100);
  if ModuleIsPackage then
  begin
    GetModuleFileName (HInstance, PChar (fPackageName), Length (fPackageName));
    fPackageName := PChar (fPackageName) // string length fixup
  end
  else
    fPackageName := 'Not packaged';
```

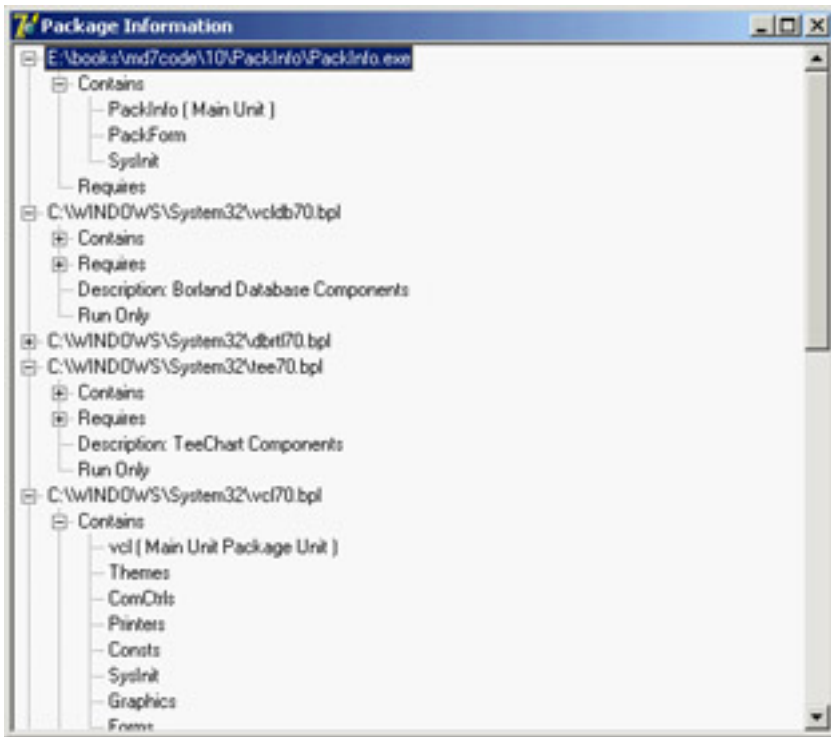
Besides accessing package information from within a component (as in the previous code), you can also do so from a special entry point of the package libraries, the `GetPackageInfoTable` function. This function returns some specific package information that Delphi stores as resources and includes in the package DLL. Fortunately, you don't need to use low-level techniques to access this information, because Delphi provides some high-level functions to manipulate it.

You can use two functions to access package information:

- `GetPackageDescription` returns a string that contains a description of the package. To call this function, you must supply the name of the module (the package library) as the only parameter.
- `GetPackageInfo` doesn't directly return information about the package. Instead, you pass it a function that it calls for every entry in the package's internal data structure. In practice, `GetPackageInfo` will call your function for every one of the package's contained units and required packages. In addition, `GetPackageInfo` sets several flags in an Integer variable.

These two function calls allow you to access internal information about a package, but how do you know which packages your application is using? You could determine this information by looking at an executable file using low-level functions, but Delphi helps you again by supplying a simpler approach. The `EnumModules` function doesn't directly return information about an application's modules; but it lets you pass it a function, which it calls for each module of the application, for the main executable file, and for each of the packages the application relies on.

To demonstrate this approach, I've built a program that displays the module and package information in a `TreeView` component. Each first-level node corresponds to a module; within each module I've built a subtree that displays the contained and required packages for that module, as well as the package description and compiler flags (`RunOnly` and `DesignOnly`). You can see the output of this example in [Figure 10.6](#).



**Figure 10.6:** The output of the PackInfo example, with details of the packages it uses

In addition to the TreeView component, I've added several other components to the main form but hidden them from view: a DBEdit, a Chart, and a FilterComboBox. I added these components simply to include more run-time packages in the application, beyond the ubiquitous Vcl and Rtl packages. The only method of the form class is `FormCreate`, which calls the module enumeration function:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    EnumModules(ForEachModule, nil);
end;
```

The `EnumModules` function accepts two parameters: the callback function (in this case, `ForEachModule`) and a pointer to a data structure that the callback function will use (in this case, `nil`, because you don't need this). The callback function must accept two parameters—an `HInstance` value and an untyped pointer—and must return a Boolean value. The `EnumModules` function will, in turn, call your callback function for each module, passing the instance handle of each module as the first parameter and the data structure pointer (`nil` in this example) as the second:

```
function ForEachModule (HInstance: Longint;
    Data: Pointer): Boolean;
var
    Flags: Integer;
    ModuleName, ModuleDesc: string;
    ModuleNode: TTreeNode;
begin
    with Form1.TreeView1.Items do
        begin
            SetLength (ModuleName, 200);
            GetModuleFileName (HInstance,
                PChar (ModuleName), Length (ModuleName));
            ModuleName := PChar (ModuleName); // fixup
            ModuleNode := Add (nil, ModuleName);

            // get description and add fixed nodes
            ModuleDesc := GetPackageDescription (PChar (ModuleName));
            ContNode := AddChild (ModuleNode, 'Contains');
```

```

ReqNode := AddChild (ModuleNode, 'Requires');

// add information if the module is a package
GetPackageInfo (HInstance, nil, Flags, ShowInfoProc);
if ModuleDesc <> '' then
begin
  AddChild (ModuleNode, 'Description: ' + ModuleDesc);
  if Flags and pfDesignOnly = pfDesignOnly then
    AddChild (ModuleNode, 'Design Only');
  if Flags and pfRunOnly = pfRunOnly then
    AddChild (ModuleNode, 'Run Only');
  end;
end;
Result := True;
end;

```

As you can see in the preceding code, the `ForEachModule` function begins by adding the module name as the main node of the tree (by calling the `Add` method of the `TreeView1.Items` object and passing `nil` as the first parameter). It then adds two fixed child nodes, which are stored in the `ContNode` and `ReqNode` variables declared in the implementation section of this unit.

Next, the program calls the `GetPackageInfo` function and passes it another callback function, `ShowInfoProc`, which I'll discuss shortly, to provide a list of the application's or package's units. At the end of the `ForEachModule` function, if the module is a package the program adds more information, such as its description and compiler flags (the program knows it's a package if its description isn't an empty string).

Earlier, I mentioned passing another callback function (the `ShowInfoProc` procedure) to the `GetPackageInfo` function, which in turn calls the callback function for each contained or required package of a module. This procedure creates a string that describes the package and its main flags (added within parentheses), and then inserts that string under one of the two nodes (`ContNode` and `ReqNode`), depending on the type of the module. You can determine the module type by examining the `NameType` parameter. Here is the complete code for the second callback function:

```

procedure ShowInfoProc (const Name: string; NameType: TNameType; Flags: Byte;
  Param: Pointer);
var
  FlagStr: string;
begin
  FlagStr := ' ';
  if Flags and ufMainUnit <> 0 then
    FlagStr := FlagStr + 'Main Unit ';
  if Flags and ufPackageUnit <> 0 then
    FlagStr := FlagStr + 'Package Unit ';
  if Flags and ufWeakUnit <> 0 then
    FlagStr := FlagStr + 'Weak Unit ';
  if FlagStr <> ' ' then
    FlagStr := ' (' + FlagStr + ')';
  with Form1.TreeView1.Items do
    case NameType of
      ntContainsUnit: AddChild (ContNode, Name + FlagStr);
      ntRequiresPackage: AddChild (ReqNode, Name);
    end;
end;

```

## What's Next?

In this chapter, you have seen how you can call functions that reside in DLLs and how to create DLLs using Delphi. After discussing dynamic libraries in general, I focused on Delphi packages, covering in particular how to place forms and other classes in a package. This is a handy technique for dividing a Delphi application into multiple executable files. While discussing packages, I explained how advanced techniques including RTTI and interfaces can be used to obtain dynamic and flexible application architectures.

I'll return to the topic of libraries that expose objects and classes when I discuss COM and OLE in [Chapter 12](#), "From COM to COM+." For the moment, let's move to another topic related to the architecture of Delphi applications: the use of modeling tools and more examples of OOP-related techniques.

# Chapter 11: Modeling and OOP Programming (with ModelMaker)

## Overview

When Borland decided to provide a UML design solution for the Enterprise and Architect editions of Delphi 7, it chose to bundle ModelMaker, by ModelMaker Tools of Holland ([www.modelmakertools.com](http://www.modelmakertools.com)). ModelMaker is a high-quality UML design tool with integration into the Delphi IDE. But as you become acquainted with ModelMaker and over the course of this chapter, you'll see that ModelMaker is far more than a UML diagramming tool. I will, of course, cover the UML diagramming capabilities of ModelMaker, but I will also spend some time on the other features of the tool as well as a conceptual overview of the product that should allow you to begin getting the most out of it.

ModelMaker has been around since the early days of Delphi, and over time it has accumulated options to support almost the entire Delphi language as well as a vast number of conveniences for programmers. The result is a huge feature set that can be daunting at first glance. The ModelMaker user interface comprises more than 100 forms, and without the proper grounding the uninitiated may become frustrated. Stick with me, and you will soon be navigating ModelMaker fearlessly.

Although ModelMaker is often referred to as a UML diagramming tool, I prefer to describe it as a Delphi-specific, customizable, extensible full-cycle UML diagramming and CASE tool. It's Delphi-specific because it is designed to handle Delphi code. When working with a property, for instance, the dialog boxes in ModelMaker present options that are specific to Delphi language keywords and concepts. ModelMaker is customizable because, as you'll see, hundreds of options control how Delphi code is generated from your object model. ModelMaker is extensible because it includes a robust OpenTools API that allows the creation of plug-in experts to extend the functionality of the product. It's a full-cycle tool because it offers features that apply to all phases of the standard development cycle. Finally, ModelMaker can be described as a CASE tool because it will automatically generate some of the redundant, obvious code required for Delphi classes, leaving it to you to provide the operational code for your classes.

**Note** This chapter has been co-written with Robert Leahey and benefits from his in-depth knowledge of and extensive experience with ModelMaker. In the world of software, Robert is an architect, programmer, author, and speaker. As a musician, he has played professionally for over 20 years and is currently a graduate student at the University of North Texas in the area of music theory. Via his company, Thoughtsmithy ([www.thoughtsmithy.com](http://www.thoughtsmithy.com)), Robert offers consulting and training services, commercial software, audio production, and large-scale LEGO brick sculptures. He lives in north Texas with his wife and daughters.



## Understanding ModelMaker's Internal Model

Before proceeding with a discussion of ModelMaker's UML support and other features, it is vital that you understand conceptually how ModelMaker manages your code model. Unlike Delphi and other editors, ModelMaker does not continually parse a source code file to visually represent the contents. Consider Delphi: Any IDE convenience you use to alter your code will directly change the contents of the source code file (which you can then save in order to persist the changes). In contrast, ModelMaker maintains an internal model representing your classes, code, documentation, and so on, from which your source code files are generated. When you edit your model through the various editors in ModelMaker, the changes are applied to the internal model—not the external source code files; at least, not until you tell ModelMaker to regenerate the external files. Understanding this distinction should save you some frustration.

Another concept to understand is that ModelMaker is capable of representing a single internal code model with multiple views in its user interface. The model can be viewed and edited, for example, as a class hierarchy, or as a list of units with contained classes. Class members can be sorted, filtered, grouped, and edited in a variety of ways. Any number of views can be seen in the various plug-ins available for ModelMaker. But most important for this discussion, the UML diagram editor itself is another view into the model. When you visualize elements of the model (such as classes and units) in your diagrams, you are creating visual representations of the code model elements; if you delete a symbol from a diagram, you are not necessarily deleting the element from the model—you are simply removing the representation from your diagram.

One last consideration about diagramming in ModelMaker: Although ModelMaker offers several wizards and automation features in the area of visualization, the product will not read your code and magically produce attractive UML diagrams with no effort on your part. Upon importing your source code and adding your classes to diagrams, you will need to arrange the symbols in order to create usable UML diagrams.

## Modeling and UML

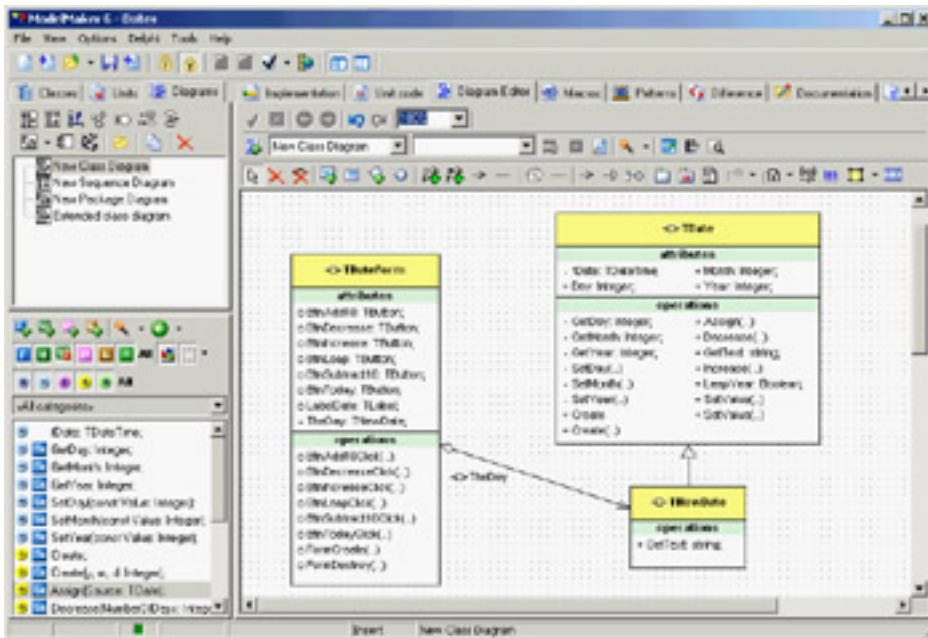
UML (Unified Modeling Language) is a graphical notation used to express the analysis and design of a software project and communicate it to others. UML is language independent, but it's intended to describe object-oriented projects. As the creators of UML stress, it is not itself a methodology; it can be used as a descriptive tool no matter what your preferred design process.

My goal is to look at UML diagrams from the perspective of a Delphi programmer using ModelMaker. An in-depth discussion of UML is well beyond the scope of this chapter.

**Note** The best introduction to UML I've seen is Martin Fowler's compact *UML Distilled* (Addison-Wesley, 1999).

## Class Diagrams

One of the most common UML diagrams supported by ModelMaker is the class diagram. Class diagrams can display a wide variety of class relationships, but at its simplest this type of diagram depicts a set of classes or objects and the static relationships between them. For example, [Figure 11.1](#) is a class diagram containing the classes from the *NewDate* program presented in [Chapter 2](#), "The Delphi Programming Language." If the results are different when you import these classes into ModelMaker and create your own class diagram, keep in mind the numerous options I discussed earlier. Many settings control how your visualized classes will appear. You can open the ModelMaker file (MPB file) used for [Figure 11.1](#) from the corresponding source code folder of the current chapter.



**Figure 11.1:** A class diagram in ModelMaker

Earlier, I mentioned that the ModelMaker diagram editor is just another view into the internal model. Some of the symbols in the ModelMaker diagrams map directly to code model elements, and others do not. With low-level diagrams like class diagrams, most symbols represent actual code model elements. Manipulating these symbols can change the code generated by ModelMaker. At the opposite end of the spectrum, in use case diagrams most (if not all) symbols have no representation within the code model. In your class diagram, you can add new classes, interfaces, fields, properties, and even documentation to the model. Likewise, you can change the inheritance of a class in the model from within the diagram. At the same time, you can add several symbols to a class diagram that have no logical representation within the code model.

Class diagrams in ModelMaker also allow you to code to interfaces, thus working at a higher abstraction level. [Figure 11.2](#) shows the relationships of the classes and interfaces in a complex example of interface use, *IntfDemo*. This example is covered in an online book discussed in [Appendix C](#), "Free Companion Books on Delphi," and is available

among the chapter's source code examples.

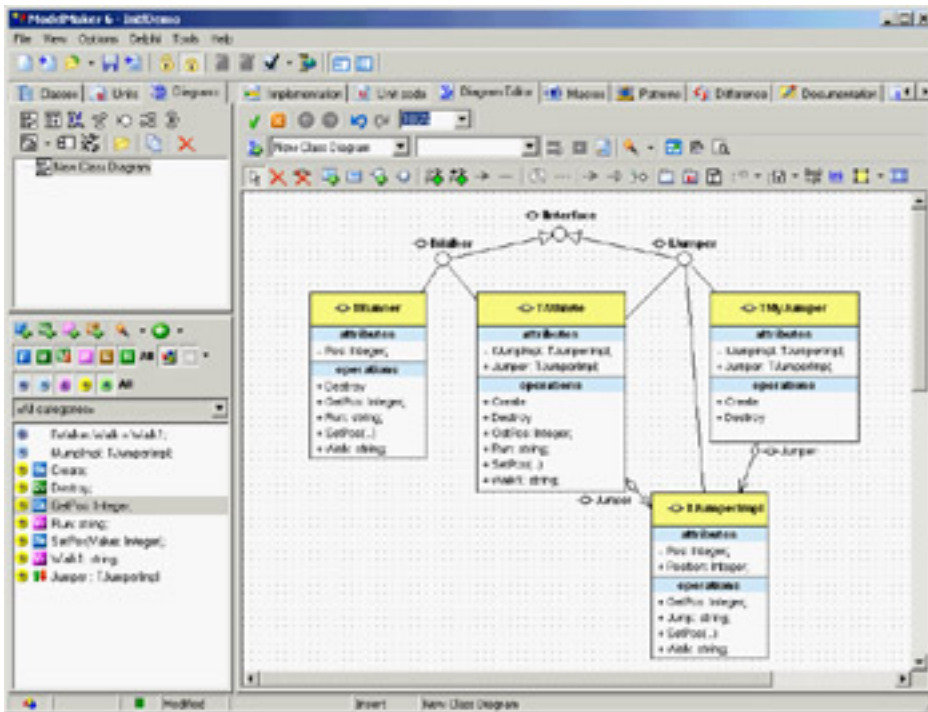


Figure 11.2: A class diagram with interfaces, classes, and interface delegation

When you're using interfaces in class diagrams, you can specify interface implementation relationships between classes and interfaces, and those implementations will be added to the code model. Adding an interface implementation within a diagram results in the appearance of one of ModelMaker's niftier features: the Interface Wizard (see [Figure 11.3](#)).

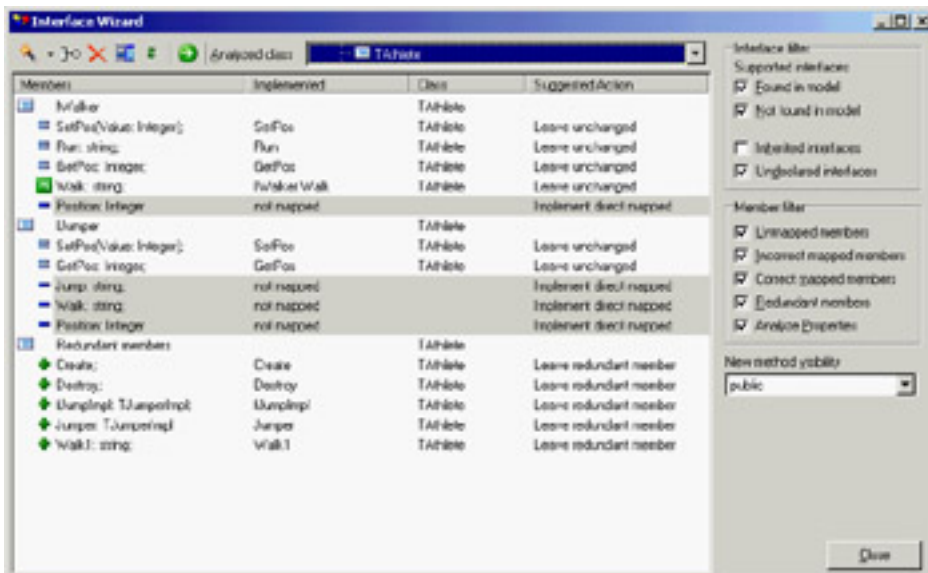
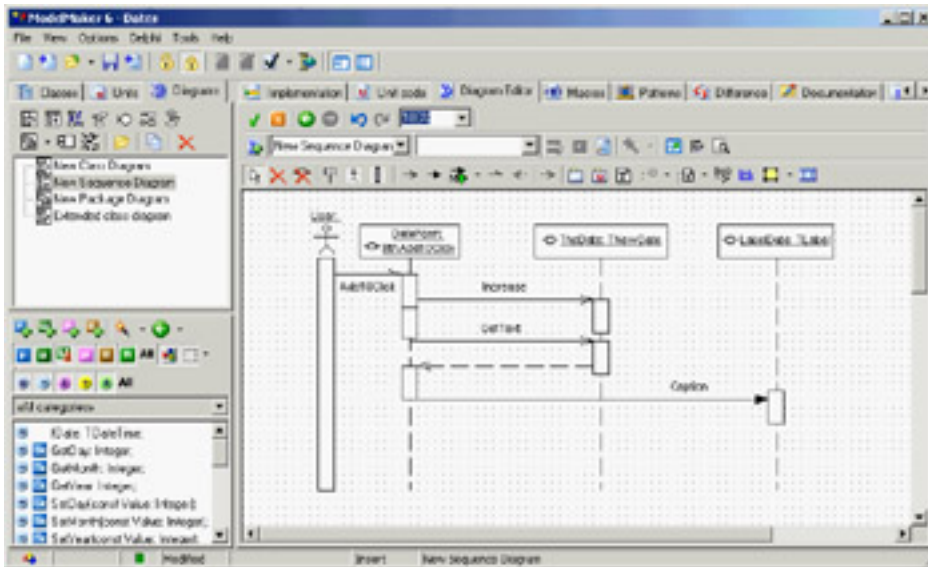


Figure 11.3: ModelMaker's Interface Wizard

Activating the Interface Wizard for a class greatly simplifies the task of implementing an interface. The wizard enumerates the methods and properties a class needs in order to implement a given interface (or interfaces); if told to, the wizard will add those needed members to the implementing class. Note that it is up to you to provide meaningful code for any methods added to the class. In [Figure 11.3](#), the wizard is evaluating `TAthlete` for its implementation of `IWalker` and `IJumpor` and suggesting the changes that are necessary for correct implementation of these interfaces.

## Sequence Diagrams

Sequence diagrams model object interaction by rendering objects and the messages that pass between them over time. In a typical sequence diagram, the objects interacting within a system are arrayed along the x-axis, and time is represented as passing from top to bottom along the y-axis. Messages are represented as arrows between objects. You can see an example of a rather trivial sequence diagram in [Figure 11.4](#). Sequence diagrams can be created at various levels of abstraction, allowing you to represent high-level system interaction involving just a few messages or low-level interaction with many messages.



**Figure 11.4:** A sequence diagram for an event handler of the NewDate example

Along with class diagrams, sequence diagrams are among the UML diagrams supported by ModelMaker that are most closely related to your code model. You can create several diagrams in ModelMaker in which the diagram symbols have no direct relation to your code model, but in sequence diagramming, you can directly affect the code as you model classes and their methods. For example, as you create a symbol for a message between objects, you can choose from a list of methods belonging to the recipient object; or, you can choose to add a new method to the object, and that new method will be added to the code model.

Note that, as mentioned earlier, ModelMaker will not automatically create sequence diagrams from your imported code; you will need to create them yourself.

## Use Cases and Other Diagrams

I've discussed two of the lowest-level UML diagrams first, but ModelMaker supports several other higher-level UML diagrams designed to provide a path from the highest-level user interaction modeling of use case diagrams to low-level class and sequence diagrams. Use case diagrams are among the most-used diagrams, in spite of the fact that their symbols bear no relation to code model elements. These diagrams are intended to model what the software is supposed to do, and they are self-explanatory enough to use in analysis sessions with non-developers.

A simple use case diagram consists of *actors* (users or application subsystems) and *use cases* (things the actors do). One of the most frequent questions regarding use cases is how to handle use case texts in ModelMaker. *Texts* for use cases are a typical next step when doing preliminary analysis. For example, a use case is a short description of an action an actor might take ("Preview Sales Report" or "Resize Window"); a use case text is a longer, more detailed description of the text. ModelMaker does not specifically support the longer use case texts; you can either use an annotation symbol within the diagram attached to the use case symbol, or you can link the use case symbol to an external file containing the use case text. You'll learn more about these techniques in the "[Common Diagram Elements](#)" section of this chapter.

The other UML diagrams supported by ModelMaker are as follows:

**Collaboration Diagrams** Interaction diagrams, much like sequence diagrams. They differ, however, in that the order of messages is specified by numbering rather than by time-scale. This results in a different diagram layout where the relationships between objects can sometimes be seen more clearly.

**State Diagrams** Diagrams that describe the behavior of a system by identifying all the states an object can assume as a result of messages it receives. A state diagram should list all the state transitions an object is subject to, indicating the starting and resulting state of each transition.

**Activity Diagrams** Diagrams that depict the workflow of a system and are particularly well suited for visualizing parallel processing.

**Component and Deployment Diagrams** Also known as implementation diagrams. Diagrams that allow you to model the relationships between components (modules, actually—executables, COM objects, DLLs, and so on) or, in the case of deployment diagrams, physical resources (referred to as *nodes*).

## Non-UML Diagrams

ModelMaker supports three diagrams that are not UML-standard, but are quite useful:

**Mind-Map Diagrams** Originated by Tony Buzan in the 1960s. An excellent method for brainstorming, exploring branching topics, or quickly recording related thoughts. I've often used mind-map diagrams for generic data display during presentations.

**Unit Dependency Diagrams** Often used to display the results of ModelMaker's powerful Unit Dependency Analyzer. These diagrams can show the branching relations of units within a Delphi project.

**Robustness Diagrams** Questionably left out of the UML specification. These diagrams help to bridge the gap between interface-only use case modeling and implementation-specific sequence diagrams. Robustness diagrams can help an analysis team verify their use case and begin looking toward implementation details.

## Common Diagram Elements

Each type of diagram supported by ModelMaker contains symbols specific to that diagram type, but there are elements within the Diagram Editor that are common to all diagram types. You can add images and shapes to your diagrams, as well as package symbols (containers to which you can add other symbols).

The Hyperlink symbol lets you add to a diagram a label linked to some other entity. In fact, the vast majority of diagram symbols support this hyperlinking feature. You can link to another diagram (clicking the link will open the linked diagram in the editor), you can link to an element within the code model (be it class, unit, method, interface, and so on—clicking this link will open the appropriate editor for the linked element), or you can link to an external document (this link will open the linked document with the appropriate application).

Three different types of annotation tools are available to each diagram type. Several documents more fully explain these tools, so suffice it to say here that you can add a stand-alone annotation symbol; you can add an annotation symbol that automatically displays the linked object's internal documentation; or you can add an annotation symbol, type in your text, and link this symbol to an object. When you do this, the object's internal documentation is updated to match the annotation symbol's text.

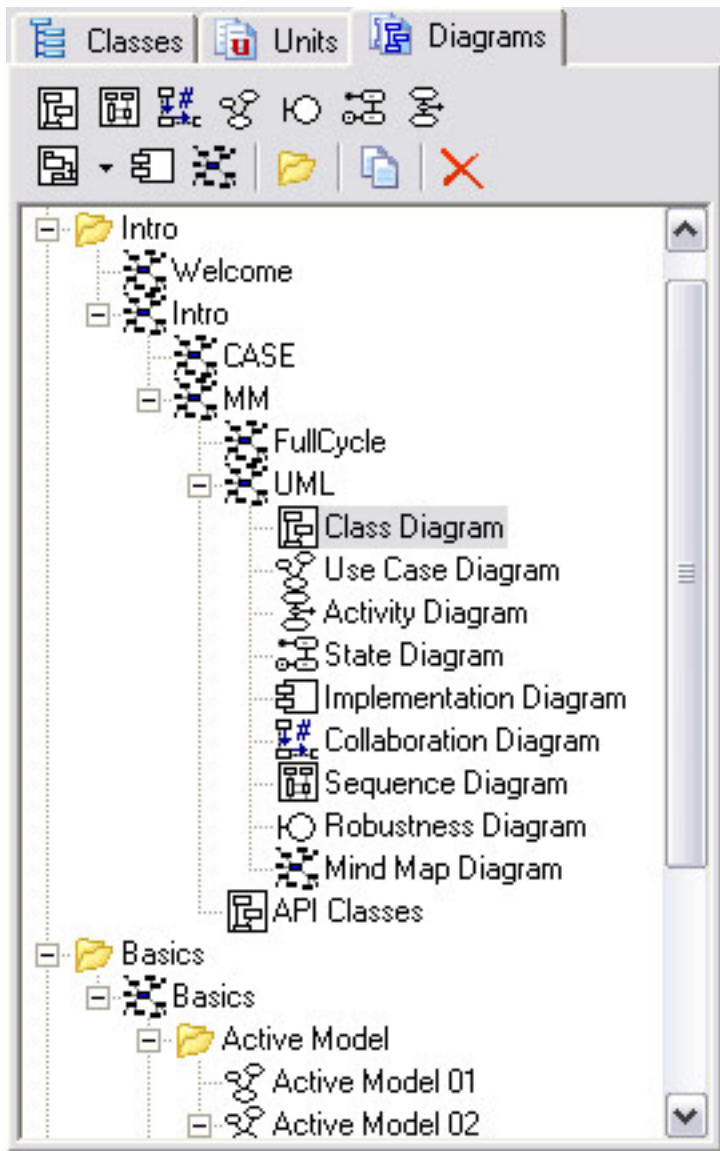
Relationship or association symbols default to straight lines. However, you can turn them into orthogonal lines by selecting the symbol and pressing Ctrl+O. You can also add nodes to these symbols by pressing Ctrl and clicking the line. ModelMaker attempts to keep these symbols orthogonal when at all possible.

ModelMaker also offers a robust set of visual symbol styles. You can define font and color styles in a hierarchical



manner and apply them by name to your diagram symbols. See the entry "style manager" in the online help for more information.

One final common feature to note is the ability to hierarchically order the Diagrams list (see [Figure 11.5](#)). You can add folders for organizational purposes and to "re-parent" diagrams; to do so, Ctrl+drag a diagram to its new parent.



**Figure 11.5:** The organizational possibilities of the Diagrams view

ModelMaker's diagramming capabilities encompass a vast set of possibilities; once you've put some time into learning the feature set, you may find that it transforms your development process. For the Delphi developer, the two-way active nature of the Diagram Editor provides a far more dynamic diagramming experience than most UML editors that simply generate static "pretty pictures."

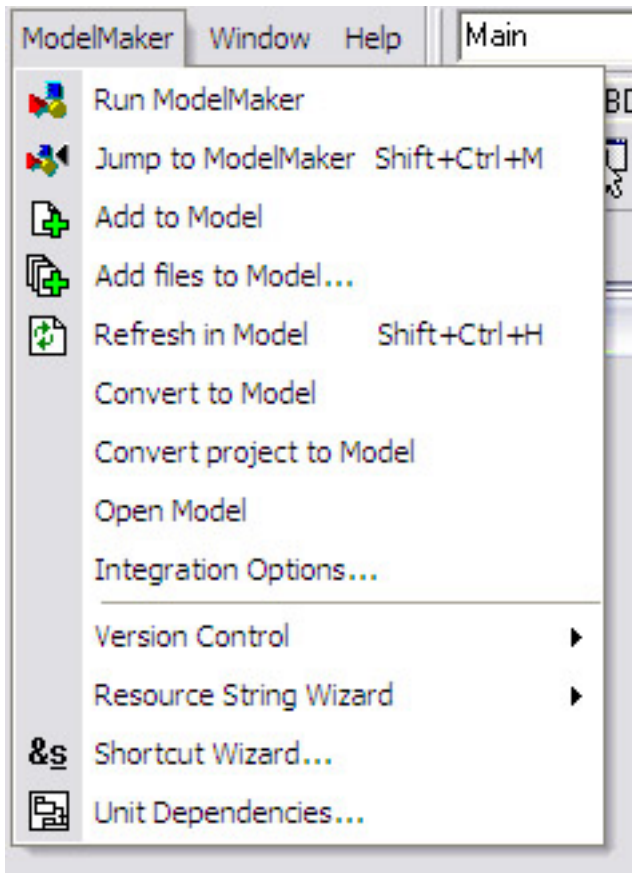
## Coding Features of ModelMaker

As you've seen, ModelMaker is a powerful UML diagramming tool, and you can do a great deal of analysis and design work exercising only those capabilities; but, as I'll explain in the following sections, ModelMaker offers a great deal more than just diagramming. Many developers use ModelMaker as their primary development environment, supplanting Delphi in that regard. This is due in part to the visual way in which ModelMaker represents Delphi programming tasks, automating many of the repetitive parts of coding Delphi classes. But it is also because Delphi, for all its strengths, tends to facilitate the development of code where the line between the presentation domain and problem domain is blurred. In other words, it is easy to write application implementation code—the code that actually does stuff—and put it right in the event handlers of your forms. This is typically not considered good object-oriented design. On the other hand, ModelMaker facilitates the creation and refactoring of problem domain objects, and the decoupling of those objects from the user interface.

Before we get to these topics, though, I'll first discuss how Delphi and ModelMaker work together.

## Delphi/ModelMaker Integration

When properly installed, ModelMaker adds a menu to the Delphi IDE, appropriately labeled ModelMaker:



If you don't see this menu, you need to install ModelMaker's DLL-based wizard in the Delphi registry, as covered in the sidebar "[Installing New DLL Wizards](#)" at the end of [Chapter 1](#). From this menu, you can control ModelMaker somewhat and quickly import your code into ModelMaker projects.

Most of the menu options are available only if ModelMaker is running. Once you start ModelMaker (either from the Run ModelMaker menu item or in the normal fashion), the other items will become available. The integration menu contains a number of ways to add your code to a model. The Add to Model, Add Files to Model, Convert to Model, and Convert Project to Model items cause ModelMaker to import the specified units: The *Add* items import units into the currently loaded model in ModelMaker, and the *Convert* items create a new model and import the units into it.

Convert Project to Model is a great place to begin—make sure you back up your code, and then select this menu item

while one of your projects is open in Delphi. The entire project will be imported into a new model in ModelMaker.

---

## Where's the VCL? Inheritance and Importing Code in ModelMaker

Upon examining your newly imported code in ModelMaker, you may notice that only the units that are part of the project were imported. ModelMaker won't automatically import units that are "used" by your project—doing so would create inordinately large models with many unnecessarily imported classes. However, many of ModelMaker's nicer inheritance-related features require that ancestor classes exist in the code model. (For instance, when properly set up, changes to ancestor classes will automatically propagate down to overriding descendants.)

Fortunately, you have many options when importing code. Although you can import via the ModelMaker integration menu in Delphi (or by dragging a unit from Windows Explorer into ModelMaker), the more flexible way is to use one of the Import buttons on ModelMaker's main toolbar. Importing units this way opens the Import Source File dialog, where you can set options for how to import the code. Taking advantage of these options lets you import part of the VCL as placeholder classes so that you can leverage ModelMaker's inheritance tools without bloating your model.

---

Also in the integration menu is Refresh in Model, which forces ModelMaker to re-import the current unit. This is a good time to discuss one of the consequences of ModelMaker's internal code model that I mentioned earlier. Because ModelMaker operates on its internal model and not on your external source code files (until you regenerate the files), it is common to find that both your model and your source code files have been edited—the result being that your model is now out of synch with the source files. When the source files have changed but not the model, you can re-synch the model by re-importing the source units. But if both the model and source files have been changed, the situation is more complicated. Fortunately, ModelMaker offers a robust set of tools to handle synchronization problems. See the section "[The Difference View](#)" for more information.

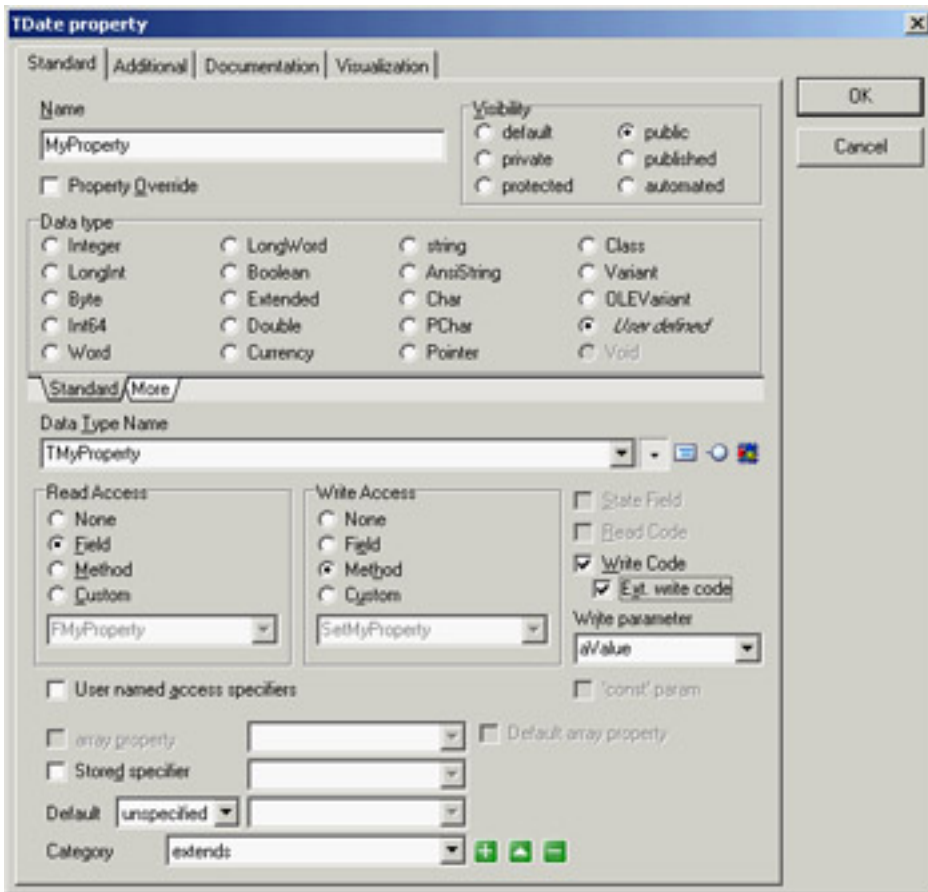
Another item of note in the integration menu is Jump to ModelMaker. When you select this item, ModelMaker attempts to find the current code position within its loaded model, bringing ModelMaker to the front in the process.

Although ModelMaker can be controlled from Delphi, the integration *is* two-way. Similar to the ModelMaker menu in the Delphi IDE, a Delphi menu appears in ModelMaker. In that menu are commands that let you jump from the currently selected model element to its corresponding position in the source code file in Delphi, as well as commands that cause Delphi to perform a syntax check, a compile, or a build. Thus you can edit your code, generate it, and compile it, all from within ModelMaker.

## Managing the Code Model

It's time to discuss the nuts and bolts of coding within ModelMaker. Due to the objectified nature of ModelMaker's internal code model, editing code model elements is typically a more visual process than it is in Delphi. Editing a class property, for instance, is done via the Property Editor dialog, as you can see in [Figure 11.6](#). Here is one of the best examples of ModelMaker's automation. When you add a new property, you don't have to worry about all the overhead of also adding a private state field, any read or write methods, or even the property's declaration. All you do is choose the appropriate settings in the editor, and ModelMaker creates the necessary supporting class members. This is more extended than the similar benefits offered in the Delphi IDE by Class Completion.





**Figure 11.6:** The Property Editor dialog

Notice that the attributes of a property that you might normally type in by hand are represented by various controls in the dialog. Visibility, Type, Read and Write specifications, and so on are all managed in the editor. The benefits are in the area of refactoring (not to mention the elimination of certain repetitive typing tasks). For instance, because ModelMaker manages a property as an object in its code model, changing something about the property—its type, for instance—will cause ModelMaker to apply that change to any references of which it is aware. If later you want to change the read access from a field to a method, you can make that change in the editor, and ModelMaker will take care of adding the get method and changing the property's declaration. Best of all, if you decide to rename the property or move it to another class, the property owns its supporting class members: They will be automatically renamed or moved as appropriate.

The same approach is used for each of the class member types; similar editors exist for methods, events, fields, and even method resolution clauses.

There's a developer-level sense of abstraction to developing in ModelMaker. It decouples you from the need to think about implementation details when editing class members; you need merely think in terms of interface, while ModelMaker handles most of the repetitive parts of implementing the member. (Don't confuse my metaphor with writing the code of a method implementation—you'll still need to do that.)

## The Unit Code Editor

ModelMaker includes two code editors: the editor for implementing class methods, which I'll discuss next, and the Unit Code Editor, which requires some explanation. ModelMaker really is a class/object oriented tool—its conveniences are mostly built around managing class-level code. When it comes to code that is not part of a class (non-class type declarations, metaclass declarations, unit methods and variables, and so on), ModelMaker takes a more no-frills approach. When ModelMaker imports a unit, anything that can be stored within the code model is handled accordingly, and what's left over appears in the Unit Code Editor. (Often, for new users, this includes any documentation not residing within method implementations—but ModelMaker can reverse-engineer your documentation as well; more on that later.)

The following is an example of what you might see in the Unit Code Editor:

```

unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Dates, StdCtrls;

type
MMWIN:CLASSINTERFACE TDateForm; ID=37;
var
  DateForm: TDateForm;

implementation

{$R *.DFM}

MMWIN:CLASSIMPLEMENTATION TDateForm; ID=37;
end.

```

Although this code looks vaguely familiar to a Delphi programmer, it obviously will not compile. You're looking at the shell that ModelMaker's code generation engine will use when expanding or generating a unit of code. When ModelMaker generates a unit, it starts at the top of this code and begins emitting lines of text while looking for one of three things: plain text, macros, or code-generation tags.

Plain text, in this example, can be found in the very first line: `unit`. ModelMaker will emit this text exactly as is. The next token on that line is a macro, `<!UnitName!>`. I'll discuss macros in depth later; for now, understand that ModelMaker will expand the macro in-place. In this case, the macro represents the name of the unit, and that text will be emitted.

Finally, an example of a code-generation tag appears directly under the `type` keyword:

```
MMWIN:CLASSINTERFACE TDateForm; ID=37;
```

In this case, the tag tells ModelMaker to expand the class interface for `TDateForm` at this point in the unit code.

Thus, when editing code in the Unit Code Editor, you are looking at a hybrid of code managed by you and code managed by ModelMaker. Take care when editing this code not to disturb the ModelMaker-managed code unless you know what you're doing. It's analogous to editing code in a Delphi-managed DPR file—you can get in trouble fast if you're not careful. Nevertheless, this is where you would add a non-class-type declaration (an enumerated type, for instance). You would handle it just as you would in Delphi, adding the type declaration into the `type` section of the unit:

```

unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Dates, StdCtrls;

type
  TmyWeekday = (wdSunday, wdMonday, wdTuesday, wdWednesday,
    wdThursday, wdFriday, wdSaturday);

MMWIN:CLASSINTERFACE TDateForm; ID=37;
var
  DateForm: TDateForm;

implementation

```

```
{ $R *.DFM }
```

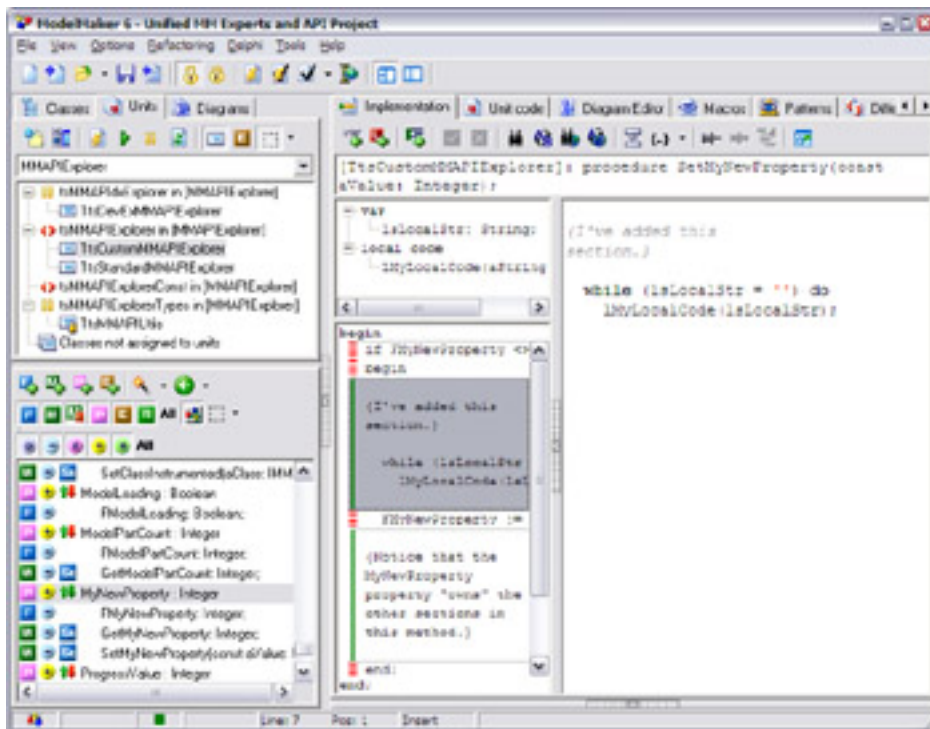
```
MMWIN:CLASSIMPLEMENTATION TDateForm; ID=37;  
end.
```

In this example, ModelMaker will emit the type declaration just as you've entered it—as plain text—and then begin expanding the `TDateForm` class declaration.

ModelMaker offers tools within the Unit Code Editor for managing unit-level methods, and they are a significant convenience when you have large routine library-type units. However, now that you're using ModelMaker, you can leverage its strong refactoring features to objectify some of those routines.

## The Method Implementation Code Editor

ModelMaker's Method Implementation Code Editor (see [Figure 11.7](#)) is quite different from the Unit Code Editor. The editor takes up the right two thirds of the screen. In this example, I've added a fictional property named `MyNewProperty` and allowed ModelMaker to generate the state field and the read and write access methods. The write access method is active in the editor.



**Figure 11.7:** ModelMaker with the Implementation tab active

Next to the code editor on the right you can see two interesting windows. The tree view on top is the local code explorer: Here you can manage local variables and local procedures. Below that is the Section List; ModelMaker allows you to break up code within a method implementation into sections. In part, this is an organizational convenience; but more importantly, it allows ModelMaker to control specific sections of code. Just as ModelMaker can own parts of the model (like a property access method automatically generated for a property), it can also own sections of code within a method. Most often this occurs when you have chosen to have ModelMaker generate the read or write code within a property access method. Notice that in this example, the first, third, and fifth sections have a red and white dashed left margin, indicating that they are owned by ModelMaker. The sections with the green margin are user-owned. When ModelMaker generates this method, the code will be emitted in the order shown, one section after another.

**Warning** A big drawback of writing code within ModelMaker Implementation windows is that it lacks any of the forms of Code Completion offered by the Delphi IDE.

## The Difference View

As I mentioned earlier, it's easy to get into a situation where your model is out of synch with your source files. If both your model and your source files have been edited, you can't simply regenerate the files from the model, lest you overwrite the source file changes. Likewise, you can't re-import the units, for fear of eliminating the changes to the model. Fortunately, ModelMaker offers one of the most robust differencing tools I've seen. When you find your model out of synch, it's time to visit the Difference tab (see [Figure 11.8](#)).

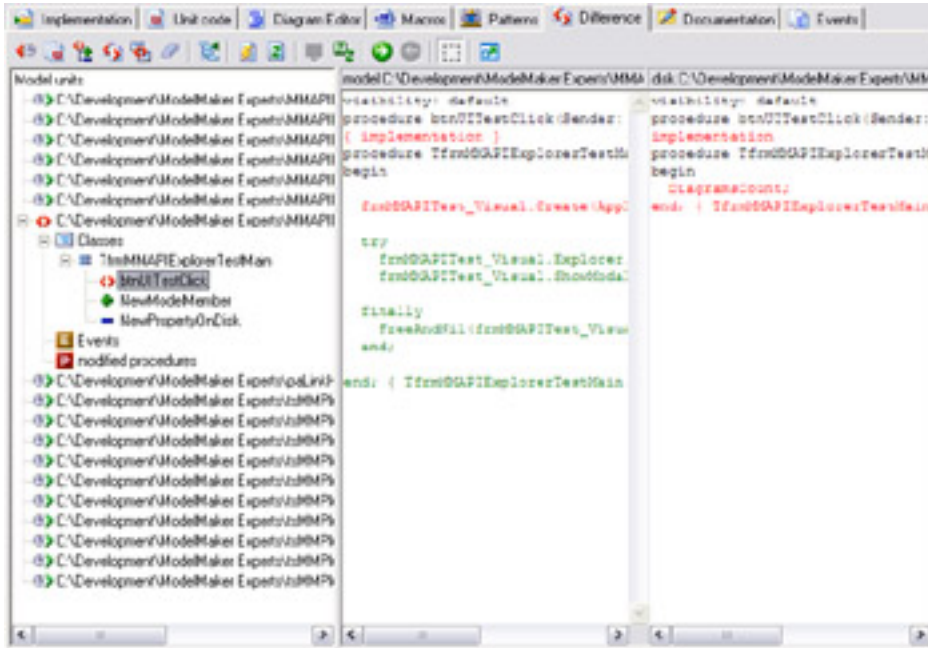
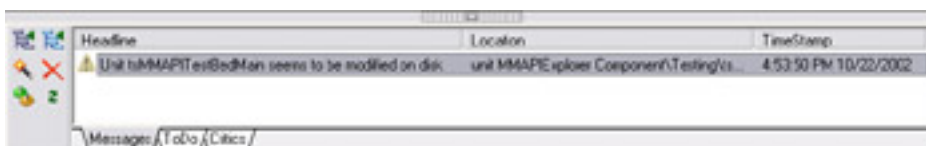


Figure 11.8: ModelMaker's Difference view

ModelMaker offers a variety of ways to view differences. You can view a standard text file comparison, time stamp differences, or even a comparison of two selected classes within the model. My favorite is on display in [Figure 11.8](#)—a structured difference. ModelMaker temporarily imports the source file on disk (thereby objectifying it just like the internal code model) and compares the imported file with the same unit and classes in the model on an object and attribute level rather than as text. The result is a much faster, much more accurate comparison.

Notice in the tree view in [Figure 11.8](#) the icons that denote differences. The red <> indicates that both the model and the source file contain the indicated class member (`btnUITestClick` in the example) but the two instances differ. The differing code is displayed in the memo controls to the right. The green + in the tree view indicates that the indicated class member exists only in the model, not on the disk. The blue - indicates that the class member exists only on disk, not in the model. With this information, you can choose how to proceed in re-synchronizing your model. One nifty feature is the ability to re-import a selected method (rather than the whole unit) from within the Difference view.

This approach implies that it is very important to know when your model is out of synch, so you don't override your changes on disk when regenerating the source files. Fortunately, ModelMaker offers several safeguards that can prevent this situation. One of these is a Design Critic (see the "[Little-Known Tidbits](#)" section later in the chapter). If your Design Critics are enabled, ModelMaker's Message View will warn you when a file on disk has changed:



## The Event Types View

ModelMaker enables the management of event types on the Events tab; there you can edit event type declarations. But keep in mind that although a new event type may exist in ModelMaker's internal code model, it does not exist in a source file until you add the event type's declaration to a unit. The easiest way to manage that process is to drag the event type declaration from the Events view list to the Unit list and drop the item into a unit.

**Team LiB**

◀ PREVIOUS

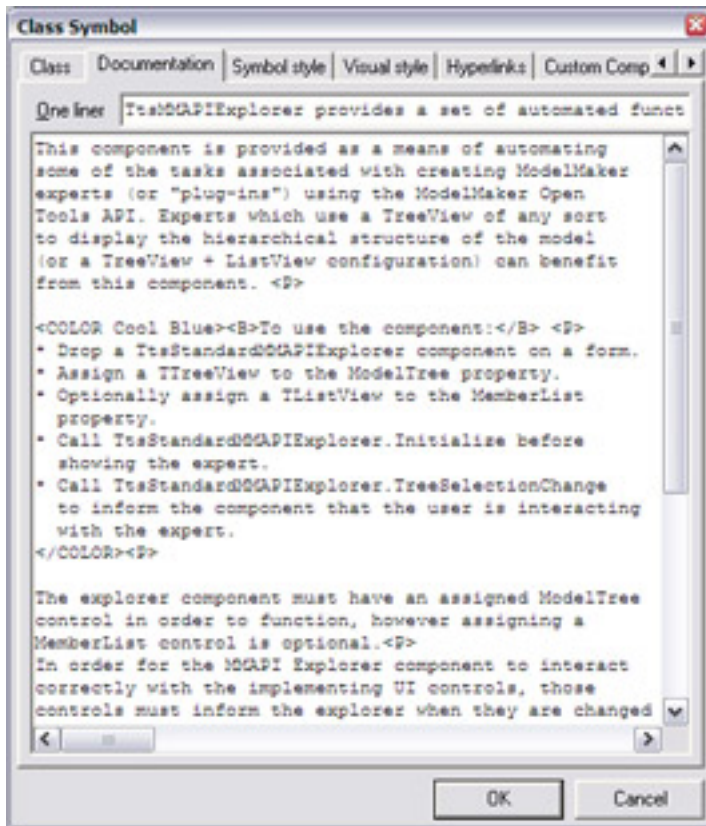
NEXT ▶

## Documentation and Macros

ModelMaker can come in very handy in supporting software documentation efforts. You need to master an important concept prior to proceeding (fortunately, it is not complex): Within ModelMaker, documentation does not equate to comments. Fear not; you can do complicated things with source code commenting, but you must take some steps to cause ModelMaker to emit (or import) those comments. Virtually every model element (classes, units, members, diagram symbols, and so on) can have documentation, but entering documentation for an element will not automatically cause that documentation to appear in your source code. That text is attached to the element within the model, but you must cause ModelMaker to generate a source code comment that contains your documentation.

### Documentation versus Comments

Every element in a ModelMaker code model can own two types of documentation: a standard, large text block of documentation, and a short one liner (see [Figure 11.9](#)). These texts can serve multiple purposes within the context of ModelMaker and, as noted earlier, do not directly equate to source code comments, although such comments can be generated from them.



**Figure 11.9:** The Documentation tab of a Class Symbol

In this example, a class has both types of documentation. This documentation might appear in a diagram as an attached annotation (either the one liner or the standard documentation can be used), or you can specify that either or both be used as part of the source code file's comments. To do this, you'll need to use ModelMaker's powerful macros (discussed in the [next section](#)) and change some of your project options. For now, let's not worry about the macros (accepting the defaults) and look only at the project options.

Access the Project Options dialog by selecting Project Options from the Options menu and select the Source Doc Generation tab. Here you will find many options regarding the generation of source code comments from ModelMaker documentation. (For more information, see the online ModelMaker help.) To see source code commenting in action, select Before Declaration from the Method Implementation section of the In Source Documentation Generation group. Now any methods that contain documentation will use the default macro to generate source code comments.



ModelMaker can also import comments from a source unit and associate those comments with the appropriate code model elements. To do this, you must signify your comments with a Documentation Import Signature (see the Source Doc Import tab of the Project Options dialog) and tell ModelMaker what lines to import into the model. Thus if your method implementation has comments like the following, you can tell ModelMaker to ignore the first five lines and import only the actual comment text:

```
{*****  
TmyClass.DoSomething  
Returns:   String  
Visibility: Public  
Comment:  
    This is the actual comment that we want ModelMaker to import. The  
    first 5 lines of this comment block should not be imported into  
    the model.}
```

When you're configuring ModelMaker for source commenting, it's important to watch out for *comment creep*. This can occur when your comment import and export settings do not quite match. For instance, if the macro controlling your source comment output adds six lines to the comment before adding the documentation text, but your import settings eliminate only five lines, then each import/generation cycle will add a redundant line of text to your comment.

## Working with Macros

Macros represent one of the key features of ModelMaker: They are easy to learn, but difficult to master. A macro is an identifier that represents a block of text. When ModelMaker encounters a macro, it attempts to replace the macro name with the text the macro represents.

You've seen this process in action in the Unit Code Editor: `<!UnitName!>` is replaced at code generation time with the name of the unit being generated. This is an example of an entity-specific macro that is always different depending on what unit is being generated. The macro, `UnitName`, is predefined, but the result will differ by context.

ModelMaker includes many predefined macros (the complete list is on page 75 of the User Manual, the `usermanual620.pdf` file you can find on Delphi's Companion CD). You can create your own macros of varying complexity (even nested macros) in the Macros tab. You can also *override* certain predefined documentation expander macros. For instance, if you enable method implementation documentation but supply no macro, ModelMaker will use its built-in macro to generate the comments. However, if you declare and define a macro named `MemberImpDoc`, ModelMaker will use this macro when generating method comments. (See ModelMaker's online help for a list of override-able macros used for source comment generation, looking for the topic "Documentation Macros.")

Macros are not used only at code-generation time. You can also expand macros while typing within a code editor. In this case, you can parameterize a macro so that when ModelMaker attempts to expand it, you will be prompted for values. These values can be inserted into the text being expanded.

## Refactoring Your Code

*Refactoring* is one of those trendy terms in computer programming that is constantly bandied about, but that means different things to different people. Refactoring is basically the process of improving your existing code in place without altering its external behavior. There is no single refactoring process to which you must adhere—it's simply the task of trying to improve your code in place without breaking too much around it.

Many texts are dedicated to the concept of refactoring, so I will simply look at some of the specific ways ModelMaker can assist you in refactoring your code. Again, ModelMaker's internal code model plays a big role—remember that developing in ModelMaker is not just development of object-oriented code; it's also a development process that is assisted by object orientation. Because all these code model elements are stored internally as objects—objects that have references to each other—and because source code units are completely regenerated from this model every time you choose to generate your code, any changes to the code elements are propagated throughout your classes instantly.

The perfect example is again a class property. If you have a property named `MyNewProperty` with attending read/write methods (maintained by ModelMaker and named `GetMyNewProperty` and `SetMyNewProperty`), and you would like to rename the property `MyProperty`, doing so requires only one step: rename the property. ModelMaker takes care of the rest—the access methods are automatically renamed `GetMyProperty` and `SetMyProperty`. If the property appears in a diagram, the diagram is automatically updated to represent the change. (One caveat: ModelMaker will not automatically search your code for instances of `MyNewProperty`—you'll have to do this with a global search and replace within ModelMaker.) This is a simple example, but it illustrates how ModelMaker simplifies the task of refactoring; as you move and rename code elements, ModelMaker will handle the majority of the details for you. Now let's look at some specifics:

**Simple Renaming** This task is quite simple and we've already touched on it, but its usefulness cannot be overstressed. Code model element name changes are propagated by ModelMaker through the code model to all instances of that element of which it is aware.

**Reparenting Classes** This absurdly simple process can be done a number of different ways. Most commonly, you can simply drag a class in the Classes view from one parent node to another (you can also do this in a class diagram by dragging the generalization arrow from the old parent to the new parent)—now your class has a new parent. If inheritance is restricted, ModelMaker will automatically update the child class's inherited methods to match the new parent's declarations. The next time you generate your code, these changes will automatically appear.

**Moving Classes between Units** It is also simple to move a class to a new unit. In the Units view, drag the class from its current place to the new unit. All relevant code (declarations, implementations, and comments) will be regenerated in the new unit.

**Moving Members between Classes** In refactoring, this process is known as "moving features (or responsibilities) between objects." The idea is simple: as development progresses, you may discover that certain responsibilities (implemented as class members) are more appropriately moved to another class. You can do so using drag and drop. Select the desired class members from the member list and drag them to the new class in the Classes (hold down the Shift key to move rather than copy).

**Converting Members** This is one of ModelMaker's niftier refactoring features. Right-clicking on a member in the Member List will display the context menu that contains the Convert To menu item and subitems. Selecting one of these subitems lets you convert an existing class member from one member type to another. For instance, if you have a private field named `FMyInteger`, and you opt to convert it to a property, ModelMaker automatically creates a public property named `MyInteger`, which reads from and writes to `FMyInteger`. Likewise, you can convert this field into a method—it will be a private function named `MyInteger` that returns an integer.

**Restricted Inheritance** In the method editor dialog is an Inheritance Restricted check box. When this



box is checked, ModelMaker does not allow you to change most of the method's attributes, because those attributes are being set based on the ancestor class's implementation of the overridden method. If you change the declaration of a method in an ancestor class, those changes will automatically be applied in any descendant classes where the overriding method is set to restricted inheritance.

If you have experience in refactoring (or you are familiar with the latest versions of JBuilder), this may not seem like a particularly impressive set of refactoring tools. However, when compared to what is possible in Delphi alone, this is a wonderful collection of possibilities. In addition, ModelMaker's OpenTools API gives you access to most of the code model. If you're discontented with what ModelMaker offers out of the box, you can extend its refactoring capabilities on your own.

**Note** Additionally, I can clandestinely say that I've seen beta versions (at the time of this writing) of a future release of ModelMaker that contain sets of new refactoring tools. Most of them are pulled straight from Martin Fowler's book on refactoring, and they are impressive.

## Applying Design Patterns

ModelMaker puts it all together impressively in its support of design patterns. (A full discussion of design patterns is beyond the scope of this chapter; however, if you're unfamiliar with patterns, see the sidebar "[Design Patterns 101](#)" for a short introduction.) ModelMaker provides the convenience of applying a pattern implementation with a single mouse click. Depending on the pattern you select, a variety of actions take place—some patterns display a wizard before adding code, and others simply add their members to the selected class. As I've discussed earlier, these new members are owned by ModelMaker and are easily updated as a result. In addition, should you choose to un-apply the pattern, ModelMaker removes any class members it added for that pattern.

As an example, let's look at the Singleton pattern. Suppose you have a class and you want only one instance of it at most. Here is the sample class:

```
type
  TOneTimeData = class (TObject)
  private
    FGlobalCount: Integer;
    procedure SetGlobalCount(const Value: Integer);
  public
    property GlobalCount: Integer read FGlobalCount write SetGlobalCount;
  end;
```

The Singleton pattern mandates the use of a single entry point (the class function `Instance` in the ModelMaker implementation of this pattern, as seen next) to gain access to your single instance of the class. If the instance does not yet exist, it will be created and returned; otherwise the existing instance will be returned. Because `Instance` is the entry point, you'll disallow the use of `Create` for this class. Once you apply the Singleton pattern to your class in ModelMaker, it appears thus:

```
type
  TOneTimeData = class (TObject)
  private
    FGlobalCount: Integer;
    procedure SetGlobalCount(const Value: Integer);
  protected
    constructor CreateInstance;
    class function AccessInstance(Request: Integer): TOneTimeData;
  public
    constructor Create;
    destructor Destroy; override;
    class function Instance: TOneTimeData;
    class procedure ReleaseInstance;
    property GlobalCount: Integer read FGlobalCount write SetGlobalCount;
```

end;

I won't list the method implementations here; you can look them up either by applying the pattern yourself, or by looking up the source code of the PatternDemo example.

**Warning** The code used by ModelMaker to implement the Singleton patterns is based on the interesting use of a constant within a method to mimic per-class data. However, this code fails to compile unless you enable the Assignable Typed Constants Delphi compiler option, which is disabled by default.

---

## Design Patterns 101

Whereas programmers concentrate on implementing specific classes, designers are more focused on making different classes/objects work together. Although it is difficult to give a precise definition of software design, essentially it is the organization of the overall structure of a program.

Looking at different people's design solutions to different problems, you can notice similarities and common elements. A *pattern* is the acknowledgement of such a common design, expressed in a standard way, and abstracted enough to be applicable in a number of different situations. Design patterns relate to design reuse more than code reuse. Although the coded solution of a pattern can be an inspiration to the programmer, the actual focus is in the design: Even if you might have to rewrite the code, starting with a clear and proven design will save you considerable time. Design patterns do not cover primitive building blocks (like a hash table or a linked list) or domain-specific problems (like analysis patterns do).

The recognized originator of the pattern movement was not a software designer but an architect, who noticed the use of patterns in buildings. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice." Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides wrote *the* book that started the pattern movement in the software world: *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995). The authors are often indicated as "Gamma et al.," but are more frequently called the Gang of Four or simply GoF. The book is often colloquially referenced as the "GoF book."

The book describes the notion of software patterns, indicates a precise way of describing them, and provides a catalog of 23 patterns divided into three groups: creational, structural, and behavioral. Most of the GoF patterns are implemented in C++, and some in Smalltalk, although they generally abstract the language and are equally applicable in Java or Delphi.

The core structure of a pattern is the following:

- The *pattern name* is important, so you can refer to the pattern when talking to other programmers and designers.
- The *problem* describes when to apply a pattern, eventually including context and conditions.
- The *solution* describes the elements part of the design and their relationships. It isn't an implementation, only an abstract description of class responsibilities and collaborations.
- The *consequences* are the results and trade-off of applying a pattern, including space and time constraints.

Currently, no book is devoted to patterns from the Delphi perspective. However, many articles have appeared in the Delphi magazines (including *Delphi Informant* and *The Delphi Magazine*). The classic GoF patterns have been a source of inspiration for many articles, along with the detailed discussion of patterns available with the ModelMaker documentation (see the website to download it).

I don't always agree with the Delphi implementation of some standard patterns. In fact, I tend to focus on the underlying design and how it can be preserved while moving the GoF implementation (often in C++ or Java) to Delphi and leveraging its specific language features. Other authors tend to port the code, which is *only* a way to implement the design. Learning about patterns is important, because you'll develop a common jargon with other

programmers and learn better ways to apply OOP techniques (particularly encapsulation and low coupling).

As a final hint, consider that most patterns are better implemented in Delphi using interfaces rather than classes (as ModelMaker tends to do, following the classic approach).

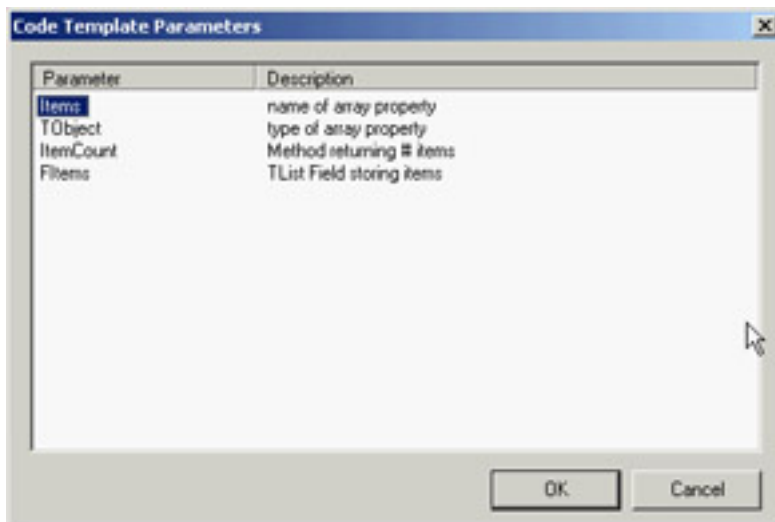
---

ModelMaker offers implementations of several other patterns, including Visitor, Observer, Wrapper, Mediator, and Decorator. They are hard-coded within ModelMaker to be applied a specific way, and some of the implementations are better than others. This has been a point of contention among some developers, and for that reason (among others) ModelMaker supports another means of applying patterns: code templates (discussed in the [next section](#)). This approach allows creation and customization on the part of the developer. However, don't overlook ModelMaker's extant support for patterns; they're quite good and offer a fixed, solid, working, Delphi-specific implementation of these common problems.

## Code Templates

Yet another powerful feature in ModelMaker (which is seemingly lost, tucked in among the myriad other conveniences) is code templates, a technique you can use to create your own implementations of design patterns. Code templates are like a snapshot of part of a class that can be applied to another class. In other words, it's a collection of class members, saved in a template that can be added to another class, en masse. Better still, these templates can be parameterized (much like macros) so that when you apply one to a class, a dialog will pop up to ask you to fill in certain values, which are then applied as part of the template.

One example is an array property. Declaring an array property is simple in ModelMaker, but completely implementing one requires several steps: You must have not only the array property itself, but also a `TList` or descendant to contain the array elements, and a means of supplying a count of the stored elements. Even for this simple example, it takes a bit of work to get your array property up and running. Enter the array property template. Open a model in ModelMaker (or create a new model and add a `TObject` descendant) and select a class to which you'd like to add your new array property. Right-click in the Member List and select Code Templates. There should now be a floating toolbar named Code Templates (note that this same toolbar is available in the Patterns tab). Click the Apply Array Property Template button to open the Code Template Parameters dialog. It contains a list of items you can specify for the template that is about to be applied, as you can see in [Figure 11.10](#). You can highlight any item in the left column and press F2 to edit the value for that parameter. Accept the defaults and click OK.



**Figure 11.10:** ModelMaker's Code Template Parameters dialog box

Your class should now contain the following members:

```
private  
    FItems: TList;
```

```

protected
    function GetItemCount: Integer;
    function GetItems(Index: Integer): TObject;
public
    property ItemCount: Integer read GetItemCount;
    property Items[Index: Integer]: TObject read GetItems;

```

You can see how flexible this technique can be. Other common tasks (like strongly typed lists) and your own implementations of design patterns are easily implemented; let's see how.

To create your own code template, begin with an existing class that already has the members you wish to turn into a template. Select that class, and then, in the Member List, select the members you wish to use (these can be any type of member). Right-click in the Member List and select Create Code Template; the Save Code Template dialog will appear. It is much like a standard Save As dialog (and you *do* specify where to save the template), but you can also detail how you'd like the template to appear. Specify a name for the template and on which page of the template palette you wish it to appear. Take note of the resulting confirmation message; you can change the palette bitmap if you wish.

Your new template is now available in the template palette; you can add this template to any class. To parameterize your template, you must alter the PAS file that was created when you saved the template. For example, here is part of the `ArrayProp_List.pas` file used for the Array Property template:

```

unit ArrayProp_List;

//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

TCodeTemplate = class (TObject)
private
    <!FItems!>: TList;
protected
    function Get<!ItemCount!>: Integer;
    function Get<!Items!>(Index: Integer): <!TObject!>;
public
    property <!ItemCount!>: Integer read Get<!ItemCount!>;
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
end;

```

Notice the lines that begin with `//DEFINEMACRO`. This is where you declare your parameters; they will appear in the Code Template Parameters dialog you saw earlier. Each line is a Name/ Value pair: the element on the left of the = is the editable value, and the element on the right is a description you can provide to explain the parameter.

After you supply a list of parameters, they can be used as macros in your template code. Note in the example lines like this:

```

property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;

```

When this property is added to a class as part of the template, the macros (things like `<!Items!>`) will be replaced with the value of the appropriate parameter. In this way, you can use parameters to deeply customize your code templates.

## Little-Known Tidbits

As a parting thought, here's a list of interesting features you might want to examine more closely:

**Rethink Orthogonal** You can change the default straight, diagonal lines in the Diagram Editor to orthogonal lines by pressing Ctrl+O. You can also force ModelMaker to try to find the shortest possible orthogonal path for a line by pressing Shift+Ctrl+O.

**Visual Styles Manager** This manager (available in the shortcut menu of the diagram view, under Visual style → Style manager) deserves an entire section. Take some time to check it out. You can define a wide variety of hierarchically related visual styles for your diagram symbols and apply them on the fly. Also, don't forget to click the Use Printing Style button in the Diagram Editor to strip out the non-printing elements and see what the diagram will look like on paper.

**Design Critics** Design critics are an impressive QA feature in ModelMaker. They are little proofreaders running the background, checking out your code. To get at them, make sure Show Messages is enabled (Shift+Ctrl+M), right-click the Message view, and select Show Critics Manager. I advise against turning off the time-stamp-checking design critic, because it will warn you if the source file on the disk has changed outside of ModelMaker. You can also create your own design critics via the ModelMaker OpenTools API.

**Creational Wizard** This is yet another nifty bit of automation for the busy Delphi programmer. The Creational Wizard (available from the Wizards button in the Member List) checks the model for class members that need to be instantiated or freed and adds them to the appropriate constructor or destructor. It will do a few other things too, and there are some caveats; press F1 while you're in the wizard to access the online help.

**Open Tools API** Much like in Delphi's Tools API, this ModelMaker feature allows the creation of plug-in experts for ModelMaker. The API is robust and includes access to diagrams as well as the entire code model. The possibilities for extending ModelMaker in this way are extreme.

## What's Next?

This chapter has provided a limited overview of the capabilities of ModelMaker, covering apparently unrelated topics like UML diagrams, patterns, refactoring, and developer's documentation. There was a reason to touch on all these aspects, of course: ModelMaker can help you with all of them, whereas the Delphi IDE by itself provides little or no support for these features.

Refer to the websites and other documentation mentioned to gain a better understanding of the various techniques. As mentioned, ModelMaker provides a lot of documentation, including material about diagrams and patterns. Visit the product website [www.modelmakertools.com](http://www.modelmakertools.com) to download more than you'll find in the default installation.

If you want more information about the techniques I've discussed, or just how to get started, refer to the ModelMaker online help and the User's Manual (available on Delphi 7's Companion CD). I also recommend the Thoughtsmithy website, where Robert Leahey has authored a set of "Getting Started with ModelMaker" tutorials. You can find them at: [www.thoughtsmithy.com/mmjump/MMGettingStarted\\_Intro.html](http://www.thoughtsmithy.com/mmjump/MMGettingStarted_Intro.html)

Next up is another chapter devoted to Delphi applications architecture. [Chapter 12](#) features an in-depth analysis of COM-related technologies available in the Windows operating system and fully supported by Delphi. After that, we'll delve into the database-programming portion of the book.

## Chapter 12: From COM to COM+

### Overview

For about 10 years, starting soon after the release of Windows 3.0, Microsoft has been promising that its operating system and its API would be based on a real object model instead of functions. According to the speculations, Windows 95 (and later Windows 2000) should have been based on this revolutionary approach. Nothing like this happened, but Microsoft kept pushing COM (Component Object Model), built the Windows 95 shell on top of it, pushed applications integration with COM and derivative technologies (such as Automation), and reached the peak by introducing COM+ with Windows 2000.

Now, soon after the release of the complete foundation required for high-level COM programming, Microsoft has decided to switch to a new core technology, part of the .NET initiative. My impression is that COM wasn't really suited for the integration of fine-grained objects, although it succeeded in providing an architecture for integrating applications or large objects.

In this chapter, you'll build your first COM object; I'll stick to the basic elements to let you understand the role of this technology without delving heavily into the details. We'll continue by discussing Automation and the role of type libraries, and you'll see how to work with Delphi data types in Automation servers and clients.

In the final part of the chapter, we'll explore the use of embedded objects, with the OleContainer component, and the development of ActiveX controls. I'll also introduce stateless COM (MTS and COM+) technologies and a few other advanced ideas including the .NET integration support offered by Delphi 7.

## A Short History of OLE and COM

Part of the confusion related to COM technology comes from the fact that Microsoft used different names for it in the early years for marketing reasons. Everything began with Object Linking and Embedding (OLE, for short), which was an extension of the DDE (Dynamic Data Exchange) model. Using the Clipboard allows you to copy raw data, and using DDE allows you to connect parts of two documents. OLE lets you copy data from a server application to a client application, along with information regarding the server or a reference to information stored in the Windows Registry. The raw data might be copied along with the link (object embedding) or kept in the original file (object linking). OLE documents are now called *active documents*.

Microsoft updated OLE to OLE 2 re-implementing it not to be just an extension to DDE and began adding new features, such as OLE Automation and OLE Controls. The next step was to build the Windows 95 shell using OLE technology and interfaces and then to rename the OLE Controls (previously known also as OCX) as ActiveX controls, changing the specification to allow for lightweight controls suitable for distribution over the Internet. For a while, Microsoft promoted ActiveX controls as suitable for the Internet, but the idea was never fully accepted by the development community—certainly not as "suitable" for Internet development.

As this technology was extended and became increasingly important to the Windows platform, Microsoft changed the name back to OLE, and then to COM, and finally to COM+ for Windows 2000. These changes in naming were only partially related to technological changes and were driven to a large extent by marketing purposes.

What, then, is COM? Basically, the Component Object Model is a technology that defines a standard way for a client module and a server module to communicate through a specific interface. Here, *module* indicates an application or a library (a DLL); the two modules may execute on the same computer or on different machines connected via a network. Many interfaces are possible, depending on the role of the client and server, and you can add new interfaces for specific purposes. These interfaces are implemented by server objects. A server object usually implements more than one interface, and all the server objects have a few common capabilities, because they must all implement the `IUnknown` interface (which corresponds to the Delphi-specific `IInterface` I introduced in [Chapter 2](#), "The Delphi Programming Language").

The good news is that Delphi is fully compliant with COM. When Delphi 3 came out its COM implementation was much easier and integrated in the language than than C++ or other languages were at that time, up to the point that even programmers on the Windows R&D team commented "we should have done COM the way Delphi does COM." This simplicity mainly derives from the incorporation of interface types into the Delphi language. (Interfaces are also used in a similar way to integrate Java with COM on the Windows platform.)

As I've mentioned, the purpose of COM interfaces is to communicate between software modules, which can be executable files or DLLs. Implementing COM objects in DLLs is generally simpler, because in Win32, a program and the DLL it uses reside in the same memory address space. This means that if the program passes a memory address to the DLL, the address remains valid. When you use two executable files, COM has a lot of work to do behind the scenes to let the two applications communicate. This mechanism is called *marshaling* (which, to be accurate, is required also by DLLs if the client is multi-threaded). Note that a DLL implementing COM objects is described as an *in-process* server, whereas when the server is a separate executable, it is called an *out-of-process* server. However, when DLLs are executing on another machine (DCOM) or inside a host environment (MTS), they are also out-of-process.



## Implementing *IUnknown*

Before we begin looking at an example of COM development, I'll introduce a few COM basics. Every COM object must implement the `IUnknown` interface, also dubbed `IInterface` in Delphi for non-COM usage of interfaces (as you saw in [Chapter 2](#)). This is the base interface from which every Delphi interface inherits, and Delphi provides a couple of different classes with ready-to-use implementations of `IUnknown/IInterface`, including `TInterfacedObject` and `TComObject`. The first can be used to create an internal object unrelated to COM, and the second is used to create objects that can be exported by servers. As you'll see later in this chapter, several other classes inherit from `TComObject` and provide support for more interfaces, which are required by Automation servers or ActiveX controls.

As mentioned in [Chapter 2](#), the `IUnknown` interface has three methods: `_AddRef`, `_Release`, and `QueryInterface`. Here is the definition of the `IUnknown` interface (extracted from the System unit):

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

The `_AddRef` and `_Release` methods are used to implement reference counting. The `QueryInterface` method handles the type information and type compatibility of the objects.

**Note** In the previous code, you can see an example of an *out* parameter, a parameter passed back from the method to the calling program but without an initial value passed by the calling program to the method. The *out* parameters have been added to the Delphi language to support COM, but they can as well be used in a normal application, as in certain circumstances this makes parameters passing more efficient (significant cases are those of interfaces, strings, and dynamic arrays). It's also important to note that although Delphi's language definition for the interface type is designed for compatibility with COM, Delphi interfaces do not require COM. This was already highlighted in [Chapter 2](#), where I built an interface-based example with no COM support.

You don't usually need to implement these methods, because you can inherit from one of the Delphi classes already supporting them. The most important class is `TComObject`, defined in the `ComObj` unit. When you build a COM server, you'll generally inherit from this class.

`TComObject` implements the `IUnknown` interface (mapping its methods to `ObjAddRef`, `ObjQueryInterface`, and `ObjRelease`) and the `ISupportErrorInfo` interface (through the `InterfaceSupportsErrorInfo` method). Notice that the implementation of reference counting for the `TComObject` class is thread-safe, because it uses the `InterlockedIncrement` and `InterlockedDecrement` API functions instead of the plain `Inc` and `Dec` procedures.

As you would expect if you remember the discussion of reference counting in [Chapter 2](#), the `_Release` method of `TInterfacedObject` destroys the object when there are no more references to it. The `TComObject` class does the same. Also keep also in mind that Delphi automatically adds the reference-counting calls to the compiled code when you use interface-based variables, including COM variables.

Finally, notice that the role of the `QueryInterface` method is twofold:

- `QueryInterface` is used for type checking. The program can ask an object the following questions: Are you of the type I'm interested in? Do you implement the interface and the specific methods I want to call? If the answers are no, the program can look for another object, maybe asking another server.
- If the answers are yes, `QueryInterface` usually returns a pointer to the object, using its reference output

parameter (Obj).

To understand the role of the `QueryInterface` method, it is important to keep in mind that a COM object can implement multiple interfaces, as the `TComObject` class does. When you call `QueryInterface`, you ask for one of the possible interfaces of the object, using the `TGUID` parameter.

In addition to the `TComObject` class, Delphi includes several other predefined COM classes. Here is a list of the most important COM classes of the Delphi VCL, which you'll use in the following sections:

- `TTypedComObject`, defined in the `ComObj` unit, inherits from `TComObject` and implements the `IProvideClassInfo` interface (in addition to the `IUnknown` and `ISupportErrorInfo` interfaces already implemented by the base class, `TComObject`).
- `TAutoObject`, defined in the `ComObj` unit, inherits from `TTypedComObject` and implements also the `IDispatch` interface.
- `TActiveXControl`, defined in the `AxCtrls` unit, inherits from `TAutoObject` and implements several interfaces (`IPersistStreamInit`, `IPersistStorage`, `IOleObject`, and `IOleControl`, to name just a few).

## Globally Unique Identifiers

The `QueryInterface` method has a parameter of the `TGUID` type. This type represents a unique ID used to identify COM object classes (in which case the GUID is called CLSID), interfaces (in which case you'll see the term IID), and other COM and system entities. When you want to know whether an object supports a specific interface, you ask the object whether it implements the interface that has a given IID (which for the default COM interfaces is determined by Microsoft). Another ID is used to indicate a specific class or CLSID. The Windows Registry stores this CLSID with indications of the related DLL or executable file. The developers of a COM server define the class identifier.

Both of these IDs are known as GUIDs, or *globally unique identifiers*. If each developer uses a number to indicate its COM server, how can we be sure these values are not duplicated? The short answer is that we cannot. The real answer is that a GUID is such a long number (16 bytes, or 128 bits—a number with 38 digits!) that it is almost impossible to come up with two random numbers having the same value. Moreover, programmers should use the specific API call `CoCreateGuid` (directly or through their development environment) to come up with a valid GUID that reflects some system information.

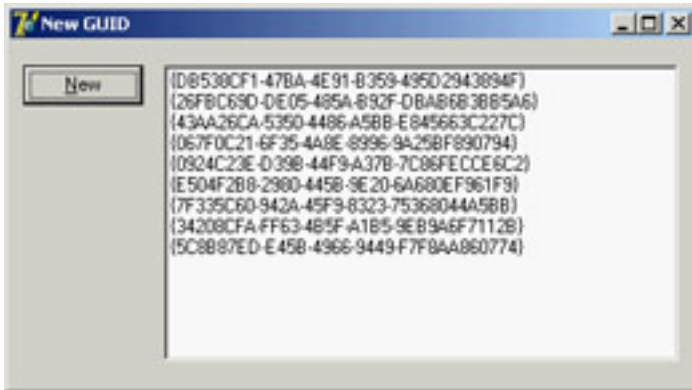
GUIDs created on machines with network cards are guaranteed to be unique, because network cards contain unique serial numbers that form a base for the GUID creation. GUIDs created on machines with CPU IDs (such as the Pentium III) should also be guaranteed unique, even without a network card. With no unique hardware identifier, GUIDs are unlikely to ever be duplicated.

**Warning** Be careful not to copy the GUID from someone else's program (which can result in two different COM objects using the same GUID). You should also not make up your own ID by entering a casual sequence of numbers. To avoid any problem, press Ctrl+Shift+G in the Delphi editor, and you will get a new, properly defined, truly unique GUID.

In Delphi, the `TGUID` type (defined in the `System` unit) is a record structure, which is quite odd but required by Windows. Thanks to some Delphi compiler magic, typically set up to help make more straightforward some tedious or time consuming task, you can assign a value to a GUID using the standard hexadecimal notation stored in a string, as in this code fragment:

```
const
  Class_ActiveForm1: TGUID = '{1AFA6D61-7B89-11D0-98D0-444553540000}';
```

You can also pass an interface identified by an IID where a GUID is required, and again Delphi will magically extract the referenced IID. If you need to generate a GUID manually and not in the Delphi environment, you can call the `CoCreateGuid` Windows API function, as demonstrated by the `NewGuid` example (see [Figure 12.1](#)). This example is so simple that I've decided not to list its code.



**Figure 12.1:** An example of the GUIDs generated by the NewGUID example. Values depend on my computer and the time I run this program.

To handle GUIDs, Delphi provides the `GUIDToString` function and the opposite `StringToGUID` function. You can also use the corresponding Windows API functions, such as `StringFromGuid2`, but in this case, you must use the `WideString` type instead of the `string` type. Any time COM is involved, you have to use the `WideString` type, unless you use Delphi functions that automatically do the required conversion for you. When you need to bypass Delphi functions that can call COM API functions directly, you can use the `PWideChar` type (pointer to null-terminated arrays of wide characters) or casting a `WideString` to `PWideChar` (exactly as you cast a `string` to the `PChar` type when calling a low-level Windows API.) does the trick.

## The Role of Class Factories

When have registered the GUID of a COM object in the Registry, you can use a specific API function to create the object, such as the `CreateComObject` API:

```
function CreateComObject (const ClassID: TGUID): IUnknown;
```

This API function will look into the Registry, find the server registering the object with the given GUID, load it, and, if the server is a DLL, call the `DLLGetClassObject` method of the DLL. This is a function every in-process server must provide and export:

```
function DllGetClassObject (const CLSID, IID: TGUID;
  var Obj): HRESULT; stdcall;
```

This API function receives as parameters the requested class and interface, and it returns an object in its reference parameter. The object returned by this function is a *class factory*.

As the name suggests, a class factory is an object capable of creating other objects. Each server can have multiple objects. The server exposes a class factory for each of the COM objects it can create. One of the many advantages of the Delphi simplified approach to COM development is that the system can provide a class factory for you. For this reason, I didn't add a custom class factory to my example.

The call to the `CreateComObject` API doesn't stop at the creation of the class factory, however. After retrieving the class factory, `CreateComObject` calls the `CreateInstance` method of the `IClassFactory` interface. This method creates the requested object and returns it. If no error occurs, this object becomes the return value of the `CreateComObject` API.

By setting up this mechanism (including the class factory and the `DLLGetClassObject` call), Delphi makes it simple to create COM objects. At the same time, Windows' `CreateComObject` is just a simple function call with complex behavior behind the scenes. What's great in Delphi is that many complex COM mechanisms are handled for you by the RTL. Let's begin looking in detail at how Delphi makes COM easy to master.

For each of the core VCL COM classes, Delphi also defines a class factory. The class factory classes form a hierarchy

and include `TComObjectFactory`, `TTypedComObjectFactory`, `TAutoObjectFactory`, and `TActiveXObjectFactory`. Class factories are important, and every COM server requires them. Usually Delphi programs use class factories by creating an object in the initialization section of the unit defining the corresponding server object class.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## A First COM Server

There is no better way to understand COM than to build a simple COM server hosted by a COM server DLL. A library hosting a COM object is indicated in Delphi as an ActiveX library. For this reason, you can begin the development of this project by selecting File → New → Other, moving to the ActiveX page, and selecting the ActiveX Library option. Doing so generates a project file I saved as FirstCom among the book demos. Here is its complete source code:

```
library FirstCom;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

The four functions exported by the DLL are required for COM compliance and are used by the system as follows:

- To access the class factory (DllGetClassObject)
- To check whether the server has destroyed all its objects and can be unloaded from memory (DllCanUnloadNow)
- To add or remove information about the server in the Windows Registry (DllRegisterServer and DllUnregisterServer)

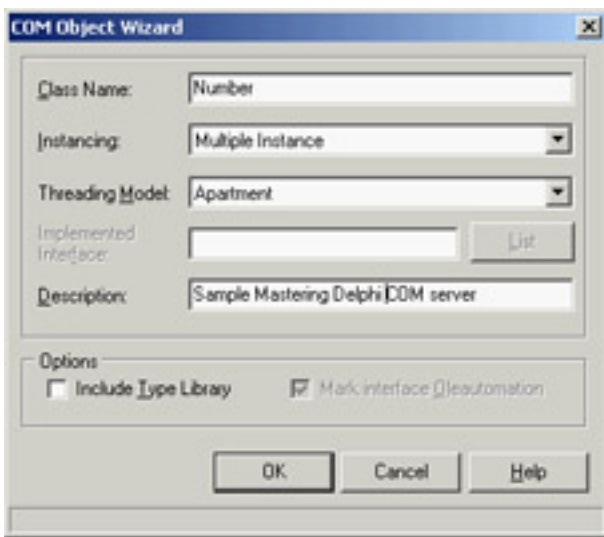
You generally don't have to implement these functions, because Delphi provides a default implementation in the ComServ unit. For this reason, in the server code, you only need to export them.

## COM Interfaces and Objects

Now that the structure of your COM server is in place, you can begin developing it. The first step is to write the code of the interface you want to implement in the server. Here is the code of a simple interface, which you should add to a separate unit (called NumIntf in the example):

```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

After declaring the custom interface, you can add the object to the server. To accomplish this, you can use the COM Object Wizard (available in the ActiveX page of the File → New → Other dialog box). You can see this wizard's dialog box in [Figure 12.2](#). Enter the name of the server's class and a description. I've disabled the generation of the type library (in which case the wizard disables the interface field in Delphi 7, different from what happened in Delphi 6) to avoid introducing too many topics at once. You should also choose an instancing and a threading model, as described in the related sidebar.



**Figure 12.2:** The COM Object Wizard

The code generated by the COM Object Wizard is quite simple. The interface contains the definition of the class to fill with methods and data:

#### type

```
TNumber = class(TComObject, INumber)
protected
    {Declare INumber methods here}
end;
```

Beside the GUID for the server (saved in the `Class_Number` constant), there is also code in the initialization section of the unit, which uses most of the options you've set up in the wizard's dialog box:

#### initialization

```
TComObjectFactory.Create(ComServer, TNumber, Class_Number, 'Number',
    'Number Server', ciMultiInstance, tmApartment);
```

This code creates an object of the `TComObjectFactory` class, passing as parameters the global `ComServer` object, a class reference to the class you've just defined, the GUID for the class, the server name, the server description, and the instancing and threading models you want to use.

The global `ComServer` object, defined in the `ComServ` unit, is a manager of the class factories available in the server library. It uses its own `ForEachFactory` method to look for the class supporting a given COM object request, and it keeps track of the number of allocated objects. As you've already seen, the `ComServ` unit implements the functions required by the DLL to be a COM library.

Having examined the source code generated by the wizard, you can now complete it by adding to the `TNumber` class the methods required for implementing the `INumber` interface and write their code, and you'll have a working COM object in your server.

---

## COM Instancing and Threading Models

When you create a COM server, you should choose a proper instancing and threading model, which can significantly affect the behavior of the COM server.

*Instancing* affects primarily out-of-process servers (any COM server in a separate executable file, rather than a DLL) and can assume three values:

**Multiple** Indicates that when several client applications require the COM object, the system starts multiple instances of the server

**Single** Indicates that, even when several client applications require the COM object, there is only one instance of the server application; it creates multiple internal objects to service the requests

**Internal** Indicates that the object can only be created inside the server; client applications cannot ask for one (this specific setting affects also in-process servers).

The second decision relates to the COM object's thread support, which is valid for in-process servers only (DLLs). The threading model is a joint decision of the client and the server application: If both sides agree on one model, it is used for the connection. If no agreement is found, COM can still set up a connection using marshaling, which can slow down the operations. Also keep in mind that a server must not only publish its threading model in the Registry (as a result of setting the option in the wizard); it must also follow the rules for that threading model in the code. Here are the key highlights of the various threading models:

**Single Model** No real support for threads. The requests reaching the COM server are serialized so the client can perform one operation at a time.

**Apartment Model, or "Single-Threaded Apartment"** Only the thread that created the object can call its methods. This means the requests for each server object are serialized, but other objects of the same server can receive requests at the same time. For this reason, the server object must take extra care to access only global data of the server (using critical sections, mutexes, or some other synchronization techniques). This is the threading model generally used for ActiveX controls inside Internet Explorer.

**Free Model, or "Multithreaded Apartment"** The client has no restrictions, which means multiple threads can use the same object at the same time. For this reason, every method of every object must protect itself and the nonlocal data it uses against multiple simultaneous calls. This threading model is more complex for a server to support than the Single and Apartment models, because even access to the object's own instance data must be handled with thread-safe care.

**Both** The server object supports both the Apartment model and the Free model.

**Neutral** Introduced in Windows 2000 and available only under COM+. This model indicates that multiple clients can call the object on different threads at the same time, but COM guarantees you that the same method is not invoked twice at the same time. Guarding for concurrent access to the object's data is required. Under COM, it is mapped to the Apartment model.

---

## Initializing the COM Object

If you look at the definition of the `TComObject` class, you will notice it has a nonvirtual constructor. Actually it has multiple constructors, each calling the virtual `Initialize` method. For this reason, to set up a COM object properly, you should not define a new constructor (which will never be called), but instead override its `Initialize` method, as I've done in the `TNumber` class. Here is the final version of this class:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
    procedure Initialize; override;
    destructor Destroy; override;
```



```
end;
```

As you can see, I've also overridden the destructor of the class, because I wanted to test the automatic destruction of the COM objects provided by Delphi.

## Testing the COM Server

Now that you've finished writing the COM server object, you can register and use it. Compile its code and then use the Run → Register ActiveX Server menu command in Delphi. You do this to register the server on your own machine, updating the local Registry.

When you distribute this server, you should install it on the client computers. To accomplish this, you could write a REG file to install the server in the Registry. However, this is not the best approach, because the server already includes a function you can activate to register the server. This function can be activated by the Delphi environment, as you've seen, or in a few other ways:

- You can pass the COM server DLL as a command-line parameter to Microsoft's `RegSvr32.exe` program, found in the `\Windows\System` directory.
- You can use the similar `TRegSvr.exe` demo program that ships with Delphi. (The compiled version is in the `\Bin` directory, and its source code is in the `\Demos\ActiveX` directory.)
- You can let an installation builder program call the server's registration function.

Having registered the server, you can now turn to the client side of the example. This time, the example is called `TestCom`, and is stored in a separate directory. The program loads the server DLL through the COM mechanism, thanks to the server information present in the Registry, so it's not necessary for the client to know which directory the server resides in.

The form displayed by this program is similar to the one you used to test some of the DLLs in [Chapter 10](#), "Libraries and Packages." In the client program, you must include the source code file with the interface and redeclare the COM server GUID. The program starts with all the buttons disabled (at design time), and it enables them only after an object has been created. This way, if an exception is raised while creating one of the objects, the buttons related to the object won't be enabled:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    // create first object  
    Num1 := CreateComObject (Class_Number) as INumber;  
    Num1.SetValue (SpinEdit1.Value);  
    Label1.Caption := 'Num1: ' + IntToStr (Num1.GetValue);  
    Button1.Enabled := True;  
    Button2.Enabled := True;  
  
    // create second object  
    Num2 := CreateComObject (Class_Number) as INumber;  
    Label2.Caption := 'Num2: ' + IntToStr (Num2.GetValue);  
    Button3.Enabled := True;  
    Button4.Enabled := True;  
end;
```

Notice in particular the call to `CreateComObject` and the following `as` cast. The API call starts the COM object-construction mechanism I've already described in detail. This call also dynamically loads the server DLL. The return value is an `IUnknown` object. This object must be converted to the proper interface type before it is assigned to the `Num1` and `Num2` fields, which now have the interface type `INumber` as their data type.



**Warning** To downcast an interface to the type, always use the `as` cast, which for interfaces performs a *QueryInterface* call behind the scenes. Alternatively, you can do a direct *QueryInterface* or *Supports* call. In the case of interfaces, the `as` cast (or a specific function call) is the only way to extract an interface from another interface. Casting an interface pointer to another interface pointer directly is an error—never do it.

The program also has a button (toward the bottom of the form) with an event handler that creates a new COM object used to get the value of the number following 100. To see why I added this method to the example, click the button in the message showing the result. You'll see a second message indicating that the object has been destroyed. This demonstrates that letting an interface variable go out of scope invokes the object's `_Release` method, decreases the object's reference count, and destroys the object if its reference count reaches zero.

The same thing happens to the other two objects as soon as the program terminates. Even if the program doesn't explicitly do so, the two objects are indeed destroyed, as the message shown by their `Destroy` destructor clearly demonstrates. This happens because they were declared to be of an interface type, and Delphi will use reference counting for them. By the way, in case you want to destroy a COM object reference with an interface, you cannot call a `Free` method (interfaces don't have `Free`) but can assign `nil` to the interface variable; this causes the removal of the reference and possibly the destruction of the object.

## Using Interface Properties

As a further small step, you can extend the example by adding a property to the `INumber` interface. When you add a property to an interface, you indicate the data type and then the `read` and `write` directives. You can have read-only or write-only properties, but the `read` and `write` clauses must always refer to a method because interfaces don't hold anything but methods.

Here is the updated interface, which is part of the `PropCom` example:

```
type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer read GetValue write SetValue;
    procedure Increase; stdcall;
  end;
```

I've given this interface a new name and, even more important, a new interface ID. I could have inherited the new interface type from the previous one, but doing so would have provided no real advantage. COM by itself doesn't support inheritance, and from the perspective of COM, all interfaces are different because they have different interface IDs. Needless to say, in Delphi you can use inheritance to improve the structure of the code of the interfaces and of the server objects implementing them.

In the `PropCom` example, I've updated the server class declaration by referring to the new interface and providing a new server object ID. The client program (called `TestProp`) can now use the `Value` property instead of the `SetValue` and `GetValue` methods. Here is a small excerpt from the `FormCreate` method:

```
Num1 := CreateComObject (Class_NumPropServer) as INumberProp;
Num1.Value := SpinEdit1.Value;
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

The difference between using methods and properties for an interface is only syntactical, because interface properties cannot access private data as Delphi class properties can. By using properties, you can make the code a little more readable.

## Calling Virtual Methods

You've built a couple of examples based on COM, but you might still feel uncomfortable with the idea of a program calling methods of objects that are created within a DLL. How is this possible if those methods are not exported by the DLL? The COM server (the DLL) creates an object and returns it to the calling application. By doing this, the DLL creates an object with a *virtual method table* (VMT). To be more precise, the object has a VMT for its class plus virtual method tables for each of the interfaces it implements.

The main program receives back an interface variable with the virtual method table of the requested interface. This VMT can be used to invoke methods, but also can be used to query for other interfaces supported by the COM object (since the `QueryInterface` method is available as part of the `IUnknown` interface VMT).

The main program doesn't need to know the memory address of those methods, because the objects know it, exactly as they do with a polymorphic call. But COM is even more powerful than this: You don't have to know which programming language was used to create the object, provided its VMT follows the standard dictated by COM.

**Tip** The COM-compatible VMT implies a strange effect. The method names are not important, provided their address is in the proper position in the VMT. This is why you can map a method of an interface to a function implementing it.

To sum things up, COM provides a language-independent binary standard for objects. The objects you share among modules are compiled, and their VMT has a particular structure determined by COM and not by the development environment you've used.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Automation

Up to now, you have seen that you can use COM to let an executable file and a library share objects. Most of the time, however, users want applications that can talk to each other. One of the approaches you can use for this goal is Automation (previously called *OLE Automation*). After presenting a couple of examples that use custom interfaces based on type libraries, I'll cover the development of Word and Excel Automation controllers, showing how to transfer database information to those applications.

**Note** The current Microsoft documentation uses the term *Automation* instead of *OLE Automation*, and uses the terms *active document* and *compound document* instead of *OLE document*. This book tends to use the new terminology, although the older "OLE" terminology is still indicated as it was probably more clear.

In Windows, applications don't exist in separate worlds; users often want them to interact. The Clipboard and DDE offer a simple way for applications to interact, as users can copy and paste data between applications. However, more and more programs offer an Automation interface to let other programs drive them. Beyond the obvious advantage of programmed automation compared to manual user operations, these interfaces are completely language-neutral, so you can use Delphi, C++, Visual Basic, or a macro language to drive an Automation server regardless of the programming language used to write it. Automation is straightforward to implement in Delphi, thanks to the extensive work by the compiler and VCL to shield developers from its intricacies. To support Automation, Delphi provides a wizard and a powerful type-library editor, and it supports dual interfaces. When you use an in-process DLL, the client application can use the server and call its methods directly, because they are in the same address space. When you use Automation, the situation is more complex. The client (called the *controller*) and the server are generally two separate applications running in different address spaces. For this reason, the system must dispatch the method calls using a complex parameter passing mechanism called *marshaling* (something I won't cover in detail).

Technically, supporting Automation in COM implies implementing the `IDispatch` interface, declared in Delphi in the System unit as:

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): HRESULT; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer): HRESULT; stdcall;
  end;
```

The first two methods return type information; the last two are used to invoke an actual method of the Automation server. Actually the invocation is performed only by the last method, `Invoke`, while `GetIDsOfNames` is used to determine the dispatch id (required by `Invoke`) from the method name. When you create an Automation server in Delphi all you have to do is define a type library and implement its interface. Delphi provides everything else through compiler magic and VCL code (actually a portion of the VCL originally called DAX framework).

The role of `IDispatch` becomes more obvious when you consider that there are three ways a controller can call the methods exposed by an Automation server:

- It can ask for the execution of a method, passing its name in a string, in a way similar to the dynamic call to a DLL. This is what Delphi does when you use a variant (see the following note) to call the Automation server. This technique is easy to use, but it is rather slow and provides little compiler type-checking. It implies a call to `GetIDsOfNames` followed by one to `Invoke`.
- It can import the definition of a Delphi dispatch interface (`dispinterface`) for the object on the server and call its methods in a more direct way (dispatching a number, that is calling `Invoke` directly as the `DispId` of each

method is known at compile time). This technique is based on interfaces and allows the compiler to check the types of the parameters and produces faster code, but it requires a little more effort from the programmer (namely the use of a type library). Also, you end up binding your controller application to a specific version of the server.

- It can call the interface directly, through the interface *vtable*, i.e. treating it as a normal COM object. This works in most cases as most Automation servers interfaces provide dual interfaces (that is support both *IDispatch* and a plain COM interface).

In the following examples, you'll use these techniques and compare them a little further.

**Note** You can use a variant to store a reference to an Automation object. In the Delphi language a variant is a *type-variant* data type, that is a variable that can assume different data types as its value. Variant data types include the basic ones (such as Integers, strings, characters, and Boolean values) but also the *IDispatch* interface type. Variants are type-checked at run time; this is why the compiler can compile the code even if it doesn't know about the methods of the Automation server, as we'll see later on.

## Dispatching an Automation Call

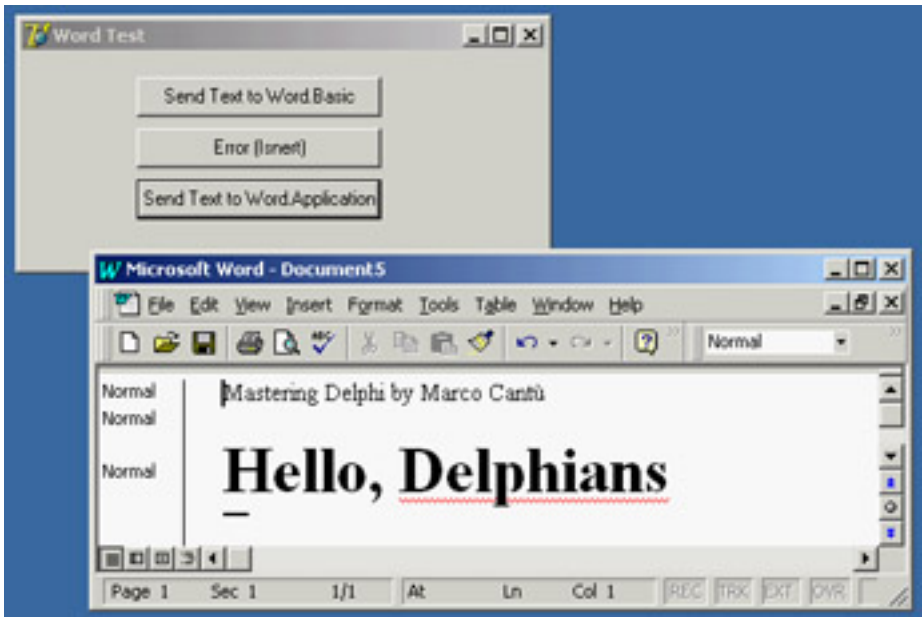
The most important difference between the two approaches is that the second generally requires a *type library*, one of the foundations of COM. A type library is basically a collection of type information, which is often found also in a COM object (with not dispatch support). This collection generally describes all the elements (objects, interfaces, and other type information) made available by a generic COM server or an Automation server. The key difference between a type library and other descriptions of these elements (such as C or Pascal code) is that a type library is language-independent. The type elements are defined by COM as a subset of the standard elements of programming languages, and any development tool can use them. Why do you need this information?

I've mentioned earlier that if you invoke a method of an Automation object using a variant, the Delphi compiler doesn't need to know about this method at compile time. A small code fragment using Word's old Automation interface, registered as `Word.Basic`, illustrates how simple it is for a programmer:

```
var
  VarW: Variant;
begin
  VarW := CreateOleObject ('Word.Basic');
  VarW.FileNew;
  VarW.Insert ('Mastering Delphi by Marco Cantù');
```

**Note** As you'll see later, recent versions of Word still register the `Word.Basic` interface, which corresponds to the internal WordBasic macro language, but they also register the new interface `Word.Application`, which corresponds to the VBA macro language. Delphi provides components that simplify the connection with Microsoft Office applications, introduced later on in this chapter.

These three lines of code start Word (unless it was already running), create a new document, and add a few words to it. You can see the effect of this application in [Figure 12.3](#).



**Figure 12.3:** The Word document is being created and composed by the WordTest Delphi application.

Unfortunately, the Delphi compiler has no way to check whether the methods exist. Doing all the type checks at run time is risky, because if you make even a minor spelling error in a function name, you get no warning about your error until you run the program and reach that line of code. For example, if you type `varW.Insert`, the compiler will not complain about the misspelling, but at run time, you'll get an error. Because it doesn't recognize the name, Word assumes the method does not exist.

Although the `IDispatch` interface supports the approach you've just seen, it is also possible—and safer—for a server to export the description of its interfaces and objects using a type library. This type library can then be converted by a specific tool (such as Delphi) into definitions written in the language you want to use to write your client or controller program (such as the Delphi language). This makes it possible for a compiler to check whether the code is correct and for you to use Code Completion and Code Parameters in the Delphi editor.

Once the compiler has done its checks, it can use either of two different techniques to send the request to the server. It can use a plain `VTable` (that is, an entry in an interface type declaration), or it can use a `dispinterface` (dispatch interface). You used an interface type declaration earlier in this chapter, so it should be familiar. A `dispinterface` is basically a way to map each entry in an interface to a number. Calls to the server can then be dispatched by number calling `IDispatch.Invoke` only, without the extra step of calling `IDispatch.GetIDsOfNames`. You can consider this an intermediate technique, in between dispatching by function name and using a direct call in the `VTable`.

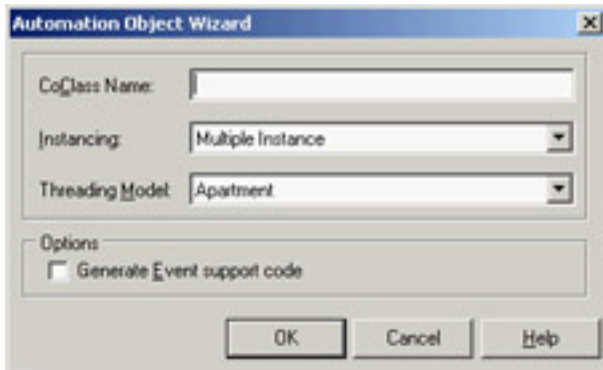
**Note** The term `dispinterface` is a keyword. A `dispinterface` is automatically generated by the type-library editor for every interface. Along with `dispinterface`, Delphi uses other related keywords: `dispid` indicates the number to associate with each element; `readonly` and `writeonly` are optional specifiers for properties.

The term used to describe this ability to connect to a server in two different ways, using a more dynamic or a more static approach, is *dual interfaces*. When writing a COM controller, you can choose to access the methods of a server two ways: you can use late binding and the mechanism provided by the `dispinterface`, or you can use early binding and the mechanism based on the `VTables`, the interface types.

It is important to keep in mind that (along with other considerations) different techniques result in faster or slower execution. Looking up a function by name (and doing the type checking at run time) is the slowest approach, using a `dispinterface` is much faster, and using the direct `VTable` call is the fastest approach. You'll do this kind of test in the `TLibCli` example, later in this chapter.

## Writing an Automation Server

Let's begin by writing an Automation server. To create an Automation object, you can use Delphi's Automation Object Wizard. Begin with a new application, open the Object Repository by selecting File → New → Other, move to the ActiveX page, and choose Automation Object. You'll see the Automation Object Wizard:

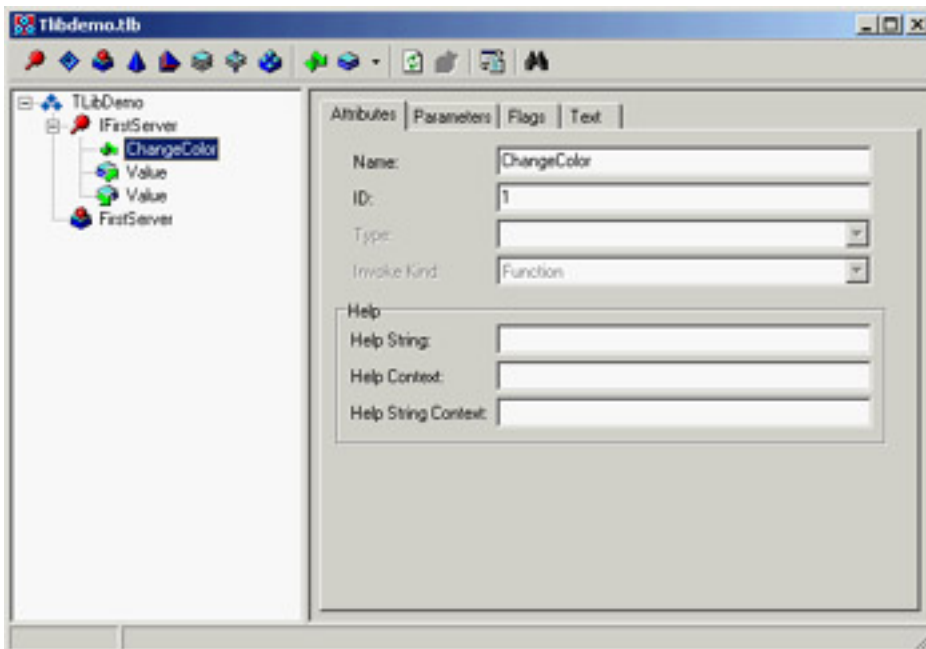


In this wizard, enter the name of the class (without the initial *T*, because it will be added automatically for you to the Delphi implementing class) and click OK. Delphi will now open the type-library editor.

**Tip** Delphi can generate Automation servers that also export events. Select the corresponding check box in the Wizard, and Delphi will add the proper entries in the type library and in the source code it generates.

## The Type-Library Editor

You can use the type-library editor to define a type library in Delphi. [Figure 12.4](#) shows its window after I've added some elements to it. The type-library editor allows you to add methods and properties to the Automation server object you've just created or to a COM object that was created using the COM Object wizard. Once you do, it can generate both the type library (TLB) file and the corresponding Delphi language source code stored in a unit called a type library import unit.



**Figure 12.4:** The type-library editor, showing the details of an interface

I have two relevant suggestions to let you work better with Delphi's type-library editor. The first and most simple is that if you right-click on the toolbar and turn on the Text Labels option you'll see in each toolbar button a caption with its



effect, making the editor easier to use. The second more relevant suggestion is to go to the Type Library page of Delphi's Environment Options dialog box and choose the Pascal language radio button over the IDL language radio button. This setting determines the notation used by the type-library editor to display methods and parameters, and even to edit the types of the parameters of a method or the type of a property. Unless you are used to writing COM code in C or C++, you'd probably prefer thinking in terms of Delphi rather than in terms of IDL.

**Warning** In this portion of the book I'll describe how to interact with the type-library editor when you have this setting, since also providing a description in terms of IDL would be both needlessly confusing and rather complex.

To build a first example, you can add a property and a method to the server by using the editor's corresponding toolbar buttons and typing their names either in the Tree View control on the left side of the window or in the Name edit box on the right side. You add these two elements to an interface, which I've called `IFirstServer`.

For the procedure you'll be able to define the parameters in the Parameters page, for a function you'd also be able to set a return type in the same page. In this specific case the `ChangeColor` method has no parameters and its Delphi definition would be:

```
procedure ChangeColor; safecall;
```

**Note** The methods contained in Automation interfaces in Delphi generally use the *safecall* calling convention. It wraps a *try/except* block around each method and provides a default return value indicating error or success. It also sets up a COM rich error object containing the exception message, so interested clients (such as Delphi clients) can re-create the server exception on the client side.

Now you can add a property to the interface by clicking the Property button on the type-library editor's toolbar. Again, you can type a name for it, such as `Value`, and select a data type in the Type combo box. Besides selecting one of the many types already listed, you can also enter other types directly, particularly interfaces of other objects.

The definition of the `Value` property of the example corresponds to the following elements of the Delphi interface:

```
function Get_Value: Integer; safecall;  
procedure Set_Value(Value: Integer); safecall;  
property Value: Integer read Get_Value write Set_Value;
```

Clicking the Refresh button on the type-library editor toolbar generates (or updates) the Delphi unit with the interface.

## The Server Code

Now you can close the type-library editor and save the changes. This operation adds three items to the project: the type library file, a corresponding Delphi definition, and the declaration of the server object. The type library is connected to the project using a resource-inclusion statement, added to the source code of the project file:

```
{ $R *.TLB }
```

You can always reopen the type-library editor by using the View → Type Library command or by selecting the proper TLB file in Delphi's normal File Open dialog box.

As mentioned earlier, the type library is also converted into an interface definition and added to a new Delphi unit. This unit is quite long, so I've listed in the book only its key elements. The most important part is the new interface declaration:

```
type  
  IFirstServer = interface(IDispatch)  
    ['{89855B42-8EFE-11D0-98D0-444553540000}']  
    procedure ChangeColor; safecall;  
    function Get_Value: Integer; safecall;  
    procedure Set_Value(Value: Integer); safecall;  
    property Value: Integer read Get_Value write Set_Value;
```

```
end;
```

Then comes the dispinterface, which associates a number with each element of the IFirstServer interface:

```
type
  IFirstServerDisp = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
end;
```

The last portion of the file includes a creator class, which is used to create an object on the server (and for this reason used on the client side of the application, not on the server side):

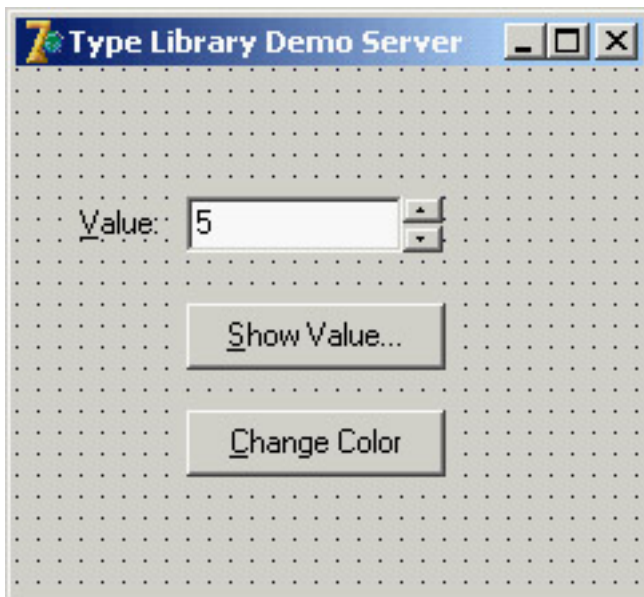
```
type
  CoFirstServer = class
    class function Create: IFirstServer;
    class function CreateRemote(const MachineName: string): IFirstServer;
end;
```

All the declarations in this file (I've skipped some others) can be considered internal, hidden implementation support. You don't need to understand them fully in order to write most Automation applications.

Finally, Delphi generates a file containing the implementation of your Automation object. This unit is added to the application and is the one you'll work on to finish the program. This unit declares the class of the server object, which must implement the interface you've just defined:

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
end;
```

Delphi already provides the skeleton code for the methods, so you only need to complete the lines in between. In this case, the three methods refer to a property and two methods I've added to the form. In general, you should not add code related to the user interface inside the class of the server object. I've done it because I wanted to be able to change the Value property and have a visible side effect (displaying the value in an edit box). Here you can see this form at design time:





## Registering the Automation Server

The unit containing the server object has one more statement, added by Delphi to the `initialization` section:

### `initialization`

```
TAutoObjectFactory.Create(ComServer, TFirstServer, Class_FirstServer,  
    ciMultiInstance);
```

`end.`

**Note** In this case, I've selected multiple instancing. For the various instancing styles possible in COM, see the sidebar "[COM Instancing and Threading Models](#)" earlier in this chapter.

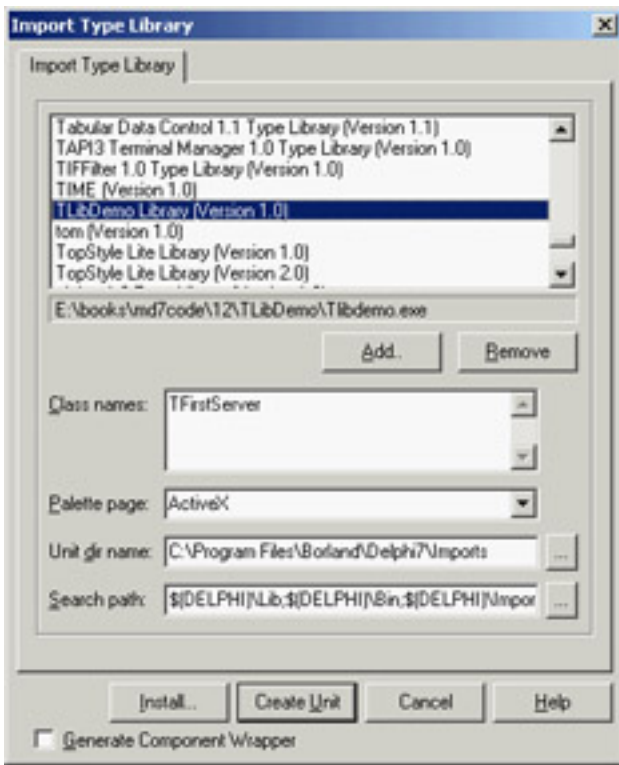
This is not very different from the creation of class factories you saw at the beginning of this chapter. The `ComServer` unit hooks the `InitProc` system function to register all COM objects as part of the COM server application startup. The execution of this code is triggered by the `Application.Initialize` call, which Delphi adds by default to the project source code of any program.

You can add the server information to the Windows Registry by running this application on the target machine (the computer where you want to install the Automation server), or by running it and passing to it the `/regserver` parameter on the command line. You can do this by selecting Start → Run, by creating a shortcut in Explorer, or by running the program within Delphi after you've entered a command-line parameter (using the Run → Parameters command). Another command-line parameter, `/unregserver`, is used to remove this server from the Registry.

## Writing a Client for the Server

Now that you have built a server, you can prepare a client program to test it. This client can connect to the server either by using variants or by using the new type library. This second approach can be implemented manually or by using Delphi techniques for wrapping components around Automation servers. You'll try all these approaches.

Create a new application—I've called it `TLibCli`—and import the server's type library, using the Project → Import type library menu command of the Delphi IDE. This command shows the Import Type Library dialog box, visible in [Figure 12.5](#). This dialog lists registered COM servers having a type library in the upper portion. You can add other projects to this list pressing the Add button and browsing for the proper file module. The lower portion of the Import Type Library dialog box shows some details of the selected library (such as the list of server objects) and about the type library import unit this dialog box is going to produce as you press the Create Unit button (or the Install button).



**Figure 12.5:** Delphi's Type Library Import dialog box.

**Warning** Do not add the type library to the client application, because you are writing the Automation controller, not a server. A controller's Delphi project should not include the type library of the server it connects to.

The type library import unit is named by Delphi after the type library, with an `_TLB` at the end. In this case the unit name is `TlibdemoLib_TLB`. I've already mentioned that one of the elements of this unit, generated also by the type library editor, is the *creation* class. I've shown you the interface of this class, but here is the implementation of the first of the two functions:

```
class function CoFirstServer.Create: IFirstServer;
begin
  Result := CreateComObject(Class_FirstServer) as IFirstServer;
end;
```

You can use it to create a server object (and possibly start the server application) on the same computer. As you can see in the code, the function is a shortcut for the `CreateComObject` call, which allows you to create an instance of a COM object if you know its GUID. As an alternative, you can use the `CreateOleObject` function, which requires as a parameter a `ProgID`, which is the registered name of the server. There is another difference between these two creation functions: `CreateComObject` returns an object of the `IUnknown` type, whereas `CreateOleObject` returns an object of the `IDispatch` type.

In this example, let's use the `CoFirstServer.Create` shorthand. When you create the server object, you get as a return value an `IFirstServer` interface. You can use it directly or store it in a variant variable. Here is an example of the first approach:

```
var
  MyServer: Variant;
begin
  MyServer := CoFirstServer.Create;
  MyServer.ChangeColor;
```

This code, based on variants, is not very different from that of the first controller you built in this chapter (the one that used Microsoft Word). Here is the alternate code, which has the same effect:

```
var
  IMyServer: IFirstServer;
```

```
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

You've already seen how you can use the interface and the variant. What about the dispatch interface? You can declare a variable of the dispatch interface type, in this case:

```
var
  DMyServer: IFirstServerDisp;
```

Then you can use it to call the methods as usual, after you've assigned an object to it by casting the object returned by the creator class:

```
DMyServer := CoFirstServer.Create as IFirstServerDisp;
```

---

## Interfaces, Variants, and Dispatch Interfaces: Testing the Speed Difference

As I mentioned in the section introducing type libraries, one of the differences between these approaches is speed. It is complicated to assess the exact performance of each technique because many factors are involved. I've added a simple test to the TLibCli example among the demos for this chapter, to give you an idea. The code for the test is a loop that accesses the `Value` of the server 100 times. The output of the program shows the timing, which is determined by calling the `GetTickCount` API function before and after executing the loop. (Two alternatives are to use Delphi's own time functions, which are slightly less precise, or to use the very precise timing functions of the multimedia support unit, `MMSystem`.)

With this program, you can roughly compare the output obtained by calling this method based on an interface, the corresponding version based on a variant, and a third version based on a dispatch interface. Looking at the timing of the example, you should see that interfaces are quicker and variants are slower, with dispatch interfaces falling in between but closer to interfaces.

---

## The Scope of Automation Objects

Another important element to keep in mind is the *scope* of the Automation objects. Variants and interface objects use reference-counting techniques, so if a variable that is related to an interface object is declared locally in a method, then at the end of the method the object will be destroyed and the server may terminate (if all the objects created by the server have been destroyed). For example, writing a method with this code produces minimal effect:

```
procedure TClientForm.ChangeColor;
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
end;
```

Unless the server is already active, a copy of the program is created and the color is changed, but then the server is immediately closed as the interface-typed object goes out of scope. The alternative approach I've used in the TLibCli example declares the object as a field of the form and creates the COM objects at startup, as in this procedure:

```
procedure TClientForm.FormCreate(Sender: TObject);
begin
  IMyServer := CoFirstServer.Create;
end;
```

With this code, as the client program starts, the server program is immediately activated. At program termination, the form field is destroyed and the server closes. A further alternative is to declare the object in the form, but then create it only when it is used, as in these two code fragments:

```
// MyServerBis: Variant;
if varType (MyServerBis) = varEmpty then
    MyServerBis := CoFirstServer.Create;
MyServerBis.ChangeColor;

// IMyServerBis: IFirstServer;
if not Assigned (IMyServerBis) then
    IMyServerBis := CoFirstServer.Create;
IMyServerBis.ChangeColor;
```

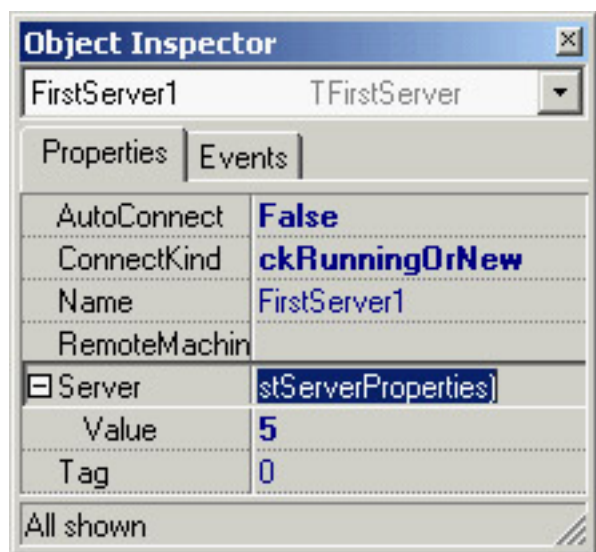
**Note** A variant is initialized to the *varEmpty* type when it is created. If you instead assign the value null to the variant, its type becomes *varNull*. Both *varEmpty* and *varNull* represent variants with no value assigned, but they behave differently in expression evaluation. The *varNull* value always propagates through an expression (making it a null expression), whereas the *varEmpty* value quietly disappears.

## The Server in a Component

When creating a client program for your server or any other Automation server, you can use a better approach: wrapping a Delphi component around the COM server. If you look at the final portion of the `TLibdemoLib_TLB` file, you can find the declaration of a `TFirstServer` class inheriting from `TOleServer`. This is a component generated when importing the library, which the system registers in the unit's `Register` procedure.

If you add this unit to a package, the new server component will become available on the Delphi Component Palette (in the ActiveX page, by default). The generation of the code of this component is controlled by a check box at the bottom of the Import Type Library dialog box, already shown in [Figure 12.5](#).

I've created a new package, `PackAuto`, available in a directory having the same name. In this package, I added the directive `LIVE_SERVER_AT_DESIGN_TIME` in the Directories/Conditionals page of the package's Project Options dialog box. This directive enables an extra feature that you don't get by default: At design time, the server component will have an extra property that lists as subitems all the properties of the Automation server:



**Warning** The `LIVE_SERVER_AT_DESIGN_TIME` directive should be used with care with the most complex Automation servers (including programs such as Word, Excel, PowerPoint, and Visio). Certain servers must be in a particular mode before you can use some properties of their automation interfaces. Because this feature is problematic at design time for many servers, it is not active by default.

As you can see in the Object Inspector, the component has few properties. `AutoConnect` indicates when to activate the COM server. When the value is `True` the server object is loaded as soon as the wrapper component is created (both at runtime and design time). When the `AutoConnect` property is set to `False`, the Automation server is loaded only the first time one of its methods is called. Another property, `ConnectKind`, indicates how to establish the

connection with the server. It can always start a new instance (`ckNewInstance`), use the running instance (`ckRunningInstance`, which shows an error message if the server is not already running), or select the current instance or start a new one if none is available (`ckRunningOrNew`). Finally, you can ask for a remote server with `ckRemote` and directly attach a server in the code after a manual connection with `ckAttachToInterface`.

**Note** To connect to an existing object, this needs to be registered in the Running Object Table (ROT). The registration must be performed by the server calling the `RegisterActiveObject` API function. Of course, only one instance for each COM server can be registered at a given time.

## COM Data Types

COM dispatching doesn't support all the data types available in Delphi. This is particularly important for Automation, because the client and the server are often executed in different address spaces, and the system must move (or *marshal*) the data from one side to the other. Also keep in mind that COM interfaces should be accessible by programs written in any language.

COM data types include basic data types such as Integer, SmallInt, Byte, Single, Double, WideString, Variant, and WordBool (but not Boolean).

In addition to the basic data types, you can use COM types for complex elements such as fonts, string lists, and bitmaps, using the `IFontDisp`, `IStrings`, and `IPictureDisp` interfaces. The following sections describe the details of a server that provides a list of strings and a font to a client.

## Exposing Strings Lists and Fonts

The `ListServ` example is a practical demonstration of how you can expose two complex types, such as a list of strings and a font, from an Automation server written in Delphi. I've chosen these two specific types because they are both supported by Delphi.

The `IFontDisp` interface is provided by Windows and is available in the ActiveX unit. The `AxCtrls` Delphi unit extends this support by providing conversion methods like `GetOleFont` and `SetOleFont`. Delphi supports the `IStrings` interface in the `StdVCL` unit, and the `AxCtrls` unit provides conversion functions for this type (along with a third type I won't use here, `TPicture`).

**Warning** To run this and similar applications, you must install and register the `StdVCL` library on the client computer. On your computer, it is registered during Delphi's installation.

The `Set` and `Get` methods of the complex types' properties copy information from the COM interfaces to the local data and from there to the form, and vice versa. The strings' two methods, for example, do this by calling the `GetOleStrings` and `SetOleStrings` Delphi functions. The client application used to demonstrate this feature is called `ListCli`. The two programs are complex; but rather than list their details here I've decided to leave the source code for your study, because Delphi programmers seldom use this advanced technique.

## Using Office Programs

So far, you've built both the client and the server side of the Automation connection. If your aim is just to let two applications you've built cooperate, this is a useful technique, although it is not the only one. You've seen the use of memory-mapped files in [Chapter 10](#). (Another technique not covered in this edition of the book is the use of the `wm_CopyData` message.) The real value of Automation is that it is a standard, so you can use it to integrate your Delphi programs with other applications your users own. A typical example is the integration of a program with Office applications, such as Microsoft Word and Microsoft Excel, or even with stand-alone applications, such as AutoCAD.

Integration with these applications provides a two-fold advantage:

- You can let your users work in an environment they know—for example, generating reports and memos from database data in a format they can easily manipulate.

- You can avoid implementing complex functionality from scratch, such as writing your own word-processing code inside a program. Instead of just reusing components, you can reuse complex applications.

This approach also has some drawbacks, which are worth mentioning:

- The user must own the application you plan to integrate with, and they may also need a recent version of it to support all the features you are using in your program.
- You have to learn a new programming architecture, often with limited documentation at hand. It is true that you are still using the Delphi language, but the code you write depends on the data types, the types introduced by the server, and in particular a collection of interrelated classes that are often difficult to understand.
- You might end up with a program that works only with a specific version of the server application, especially if you try to optimize the calls by using interfaces instead of variants. In particular, Microsoft does not attempt to maintain script compatibility between major releases of Word or other Office applications.

Delphi simplifies the use of Microsoft Office applications by preinstalling some ready-to-use components that wrap the Automation interface of these servers. These components, available in the Servers page of the Palette, are installed using the same technique I demonstrated in the last section. The real plus lies in the technique of creating components to wrap existing Automation servers, rather than in the availability of predefined server components. Notice also that these Office components exist in different versions depending on your version of the Microsoft suite: All components are installed, but only one set is registered at design time, according to your choice in the Delphi install program. You can change this setting later by removing the related component package and adding a new one.

You won't see an actual example in this section because it is very difficult to write a program that works with all of the different versions of Microsoft Office. You'll find some sample code and tips in Essential Delphi (see [Appendix C](#) for instructions on how to download this free ebook).



## Using Compound Documents

*Compound documents* is Microsoft's name for the technology that allows in-place editing of a document within another document (for example, a picture in a Word document). This technology originated the term *OLE*, but its role is now definitely more limited than Microsoft envisioned when it was introduced in the early 1990s. Compound documents have two capabilities: *object linking* and *embedding* (hence the term *OLE*):

- *Embedding* an object in a compound document corresponds to a smart version of the copy and paste operations you perform with the Clipboard. The key difference is that when you copy an OLE object from a server application and paste it into a container application, you copy both the data and some information about the server (its GUID). This allows you to activate the server application from within the container to edit the data.
- *Linking* an object to a compound document copies only a reference to the data and the information about the server. You generally activate object linking by using the Clipboard and performing a Paste Link operation. When editing the data in the container application, you modify the original data, which is stored in a separate file.

Because the server program refers to an entire file (only part of which may be linked in the client document), the server will be activated in a stand-alone window, and it will act upon the entire original file, not just the data you've copied. When you have an embedded object, however, the container may support visual (or *in-place*) editing, which means you can modify the object in context inside the container's main window. The server and container application windows, their menus, and their toolbars are merged automatically, allowing the user to work in a single window on several different object types—and therefore with several different OLE servers—without leaving the window of the container application.

Another key difference between embedding and linking is that an embedded object's data is stored and managed by the container application. The container saves the embedded object in its own files. By contrast, a linked object physically resides in a separate file, which is handled by the server exclusively, even if the link refers only to a small portion of the file. In both cases, the container application doesn't have to know how to handle the object and its data—not even how to display it—without the help of the server. Considering the relative slowness of OLE and the amount of work necessary to develop COM servers, you can understand why this approach never took off.

Compound document containers can support COM in varying degrees. You can place an object in a container by inserting a new object, by pasting or *paste-linking* one from the Clipboard, by dragging one from another application, and so on. Once the object is placed in the container, you can then perform operations on it, using the server's available *verbs*, or actions. Usually the *Edit verb* is the default action—the action performed when you double-click on the object. For other objects, such as video or sound clips, *Play* is defined as the default action. You can typically see the list of actions supported by the current contained object by right-clicking it. The same information is available in many programs via the Edit → Object menu item, which displays a submenu that lists the available verbs for the current object.

## The Container Component

To create a COM container application in Delphi, place an `OleContainer` component in a form. Then select the component and right-click to activate its shortcut menu, which will include an Insert Object command. When you select this command, Delphi displays the standard OLE Insert Object dialog box. This dialog box allows you to choose from one of the server applications registered on the computer.

Once the COM object is inserted in the container, the control container component's shortcut menu will include several more custom menu items. The new menu items include commands to change the properties of the COM object, insert another object, copy the existing object, or remove the existing object. The list also includes the verbs (actions) of the object (such as Edit, Open, or Play). Once you have inserted a COM object in the container, the corresponding server will launch to let you edit the new object. As soon as you close the server application, Delphi updates the object in the container and displays it at design time in the form of the Delphi application you are developing.

If you look at the textual description of a form containing a component with an object inside, you'll notice a `Data` property, which contains the COM object's data. Although the client program stores the object's data, it doesn't know

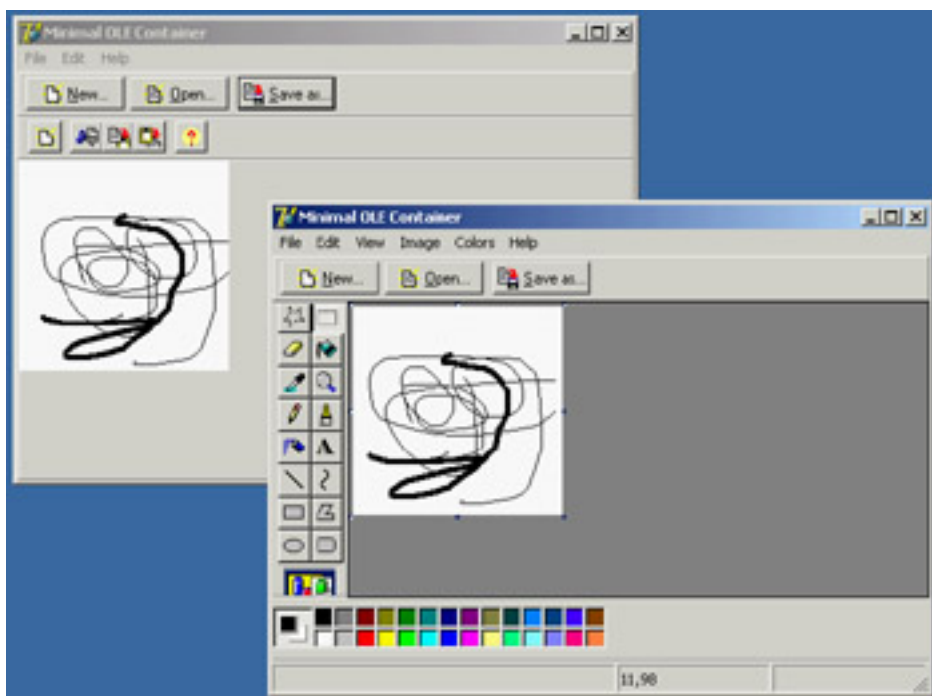
how to handle and show that data without the help of the proper server (which must be available on the computer where you run the program). This means the COM object is *embedded*.

To fully support compound documents, a program should provide a menu and a toolbar or panel. These extra components are important because in-place editing implies a merging of the client's user interface and the server program's user interface. When the COM object is activated in place, some of the pull-down menus on the server application's menu bar are added to the container application's menu bar.

Menu merging is handled almost automatically by Delphi. You only need to set the proper indexes for the menu items of the container, using the `GroupIndex` property. Any menu item with an odd index number is replaced by the corresponding element of the active OLE object. Specifically, the File (0) and Window (4) pull-down menus belong to the container application. The Edit (1), View (3), and Help (5) pull-down menus (or the groups of pull-down menus with those indexes) are taken by the COM server. A sixth group, Object (2), can be used by the container to display another pull-down menu between the Edit and View groups, when the COM object is active. The `OleCont` demo program I've written to demonstrate these features allows a user to create a new object by calling the `InsertObjectDialog` method of the `TOleContainer` class.

Once a new object has been created, you can execute its primary verb using the `DoVerb` method. The program also displays a small toolbar with some bitmap buttons. I placed some `TWinControl` components in the form to let the user select them and thus disable the `OleContainer`. To keep this toolbar/panel visible while in-place editing is occurring, you should set its `Locked` property to `True`. This setting forces the panel to remain present in the application and not be replaced by a toolbar of the server.

To show what happens when you don't use this approach, I've added to the program a second panel with more buttons. Because I haven't set its `Locked` property, this new toolbar will be replaced with that of the active server. When in-place editing launches a server application that displays a toolbar, that server's toolbar replaces the container's toolbar, as you can see in the lower part of [Figure 12.6](#).



**Figure 12.6:** The second toolbar of the `OleCont` example (above) is replaced by the server's toolbar (below).

**Tip** To make all the automatic resizing operations work smoothly, you should place the OLE container component in a panel component and align both of them to the client area of the form.

Alternatively, you can create a COM object using the `PasteSpecialDialog` method, called in the example's `PasteSpecialClick` event handler. Another standard COM dialog box, wrapped in a Delphi function, shows the properties of the object; this dialog box is activated with the Object Properties item in the Edit pull-down menu by



calling the `ObjectPropertiesDialog` method of the `OleContainer` component.

The last feature of the `OleCont` program is support for files. This is one of the simplest additions you can make, because the OLE container component already provides file support.

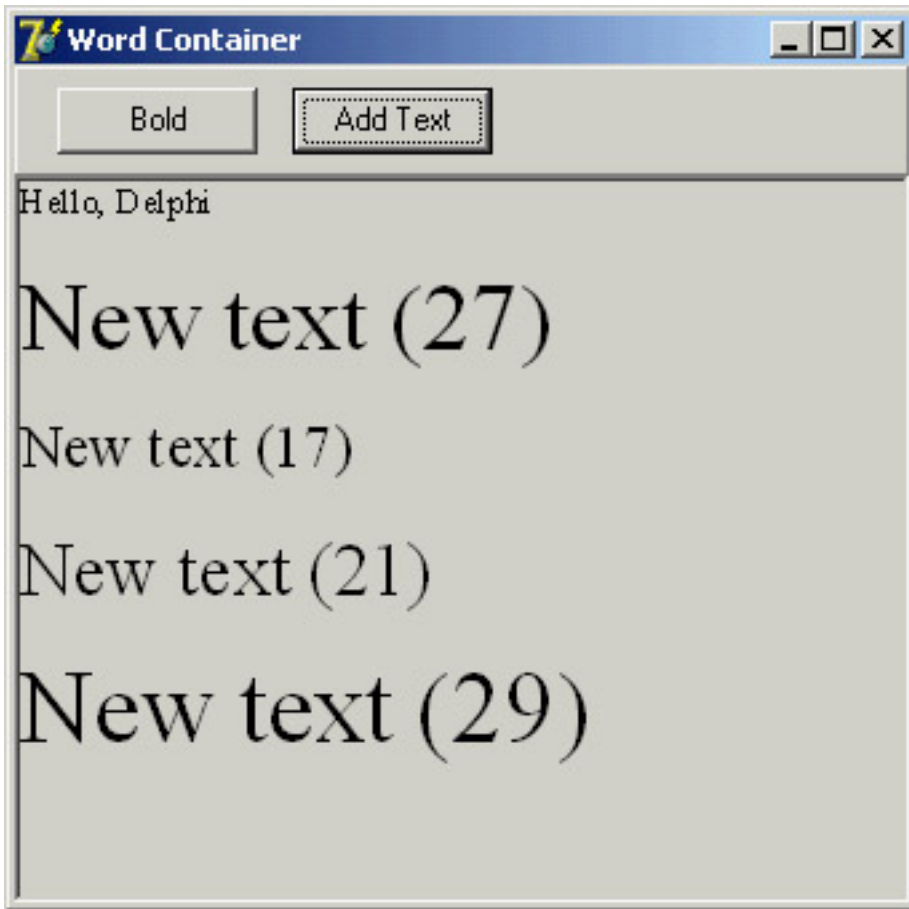
## Using the Internal Object

In the preceding program, the user determined the type of the internal object created by the program. In this case, there is little you can do to interact with the internal objects. Suppose, instead, that you want to embed a Word document in a Delphi application and then modify it by code. You can do this by using Automation with the embedded object, as demonstrated by the `WordCont` example (the name stands for *Word container*).

**Warning** Because the `WordCont` example includes an object of a specific type (a Microsoft Word document), it won't run if you don't have that application installed. Having a different version of the server might also create problems (I've tested the examples in this chapter only with Office 97). For other versions, you might have to rebuild the program following the same steps I used.

In the example's form, I added an `OleContainer` component, set its `AutoActivate` property to `aaManual` (so that the only possible interaction is with my code), and added a toolbar with a couple of buttons. The code is straightforward, once you know that the embedded object corresponds to a Word document. Here is an example (you can see the effect of this code in [Figure 12.7](#)):

```
procedure TForm1.Button3Click(Sender: TObject);
var
    Document, Paragraph: Variant;
begin
    // activate if not running
    if not (OleContainer1.State = osRunning) then
        OleContainer1.Run;
    // get the document
    Document := OleContainer1.OleObject;
    // add paragraphs, getting the last one
    Document.Paragraphs.Add;
    Paragraph := Document.Paragraphs.Add;
    // add text to the paragraph, using random font size
    Paragraph.Range.Font.Size := 10 + Random (20);
    Paragraph.Range.Text := 'New text (' +
        IntToStr (Paragraph.Range.Font.Size) + ')'#13;
end;
```



**Figure 12.7:** The WordCont example shows how to use Automation with an embedded object.

## Introducing ActiveX Controls

Microsoft's Visual Basic was the first program development environment to introduce the idea of supplying software components to the mass market, even if the concept of reusable software components is older than Visual Basic—it's well rooted in the theories of object-oriented programming (OOP). The first technical standard promoted by Visual Basic was *VBX*, a 16-bit specification that was fully available in Delphi 1. In moving to the 32-bit platforms, Microsoft replaced the VBX standard with the more powerful and more open *ActiveX* controls.

**Note** ActiveX controls used to be called OLE controls (or OCX). The name change reflects a new marketing strategy from Microsoft rather than a technical innovation. Not surprisingly, then, ActiveX controls are usually saved in files with the `.ocx` extension.

From a general perspective, an ActiveX control is not very different from a Windows control. The key difference is in the control's *interface*—the interaction between the control and the rest of the application. Typical Windows controls use a message-based interface; Automation objects and ActiveX controls use properties, methods, and events (like Delphi's own components).

Using COM jargon, an ActiveX control is a "compound document object which is implemented as an in-process server DLL and supports Automation, visual editing, and inside-out activation." Perfectly clear, right? Let's see what this definition means. COM servers can be implemented three ways:

- As stand-alone applications (for example, Microsoft Excel)
- As out-of-process servers—that is, executables files that cannot be run by themselves and can only be invoked by a server (for example, Microsoft Graph and similar applications)
- As in-process servers, such as DLLs loaded into the same memory space as the program using them

ActiveX controls can only be implemented using the last technique, which is also the fastest: as in-process servers. Furthermore, ActiveX controls are Automation servers. This means you can access properties of these objects and call their methods. You can see an ActiveX control in the application that is using it and interact with it directly in the container application window. This is the meaning of the term *visual editing*, or *in-place activation*. A single click activates the control rather than the double-click used by OLE documents, and the control is active whenever it is visible (which is what the term *inside-out activation* means) without your having to double-click it.

In an ActiveX control, properties can identify states, but they can also activate methods. Properties can refer to aggregate values, arrays, subobjects, and so on. Properties can also be dynamic (or read-only, to use the Delphi term). The properties of an ActiveX control are divided into groups: stock properties that most controls need to implement; ambient properties that offer information about the container (similar to the `ParentColor` and `ParentFont` properties in Delphi); extended properties managed by the container, such as the position of the object; and custom properties, which can be anything.

Events and methods are, well, events and methods. *Events* relate to a mouse click, a key press, the activation of a component, and other specific user actions. *Methods* are functions and procedures related to the control. There is no major difference between the ActiveX and Delphi concepts of events and methods.

## ActiveX Controls Versus Delphi Components

Before I show you how to use and write ActiveX controls in Delphi, let's go over some of the differences between the two kinds of controls. ActiveX controls are DLL-based: When you use them, you need to distribute their code (the OCX file) along with the application using them. In Delphi, the components' code can be statically linked to the executable file or dynamically linked to it using a run-time package, so you can always choose whether to deploy a single large file or many smaller modules.

Having a separate file allows you to share code among different applications, as DLLs usually do. If two applications use the same control (or run-time package), you need only one copy of it on the hard disk and a single copy in

memory. The drawback, however, is that if the two programs have to use two different versions (or builds) of the ActiveX control, some compatibility problems may arise. An advantage of having a self-contained executable file is that you will also have fewer installation problems.

The drawback of using Delphi components is not that there are fewer Delphi components than ActiveX controls, but that if you buy a Delphi component, you'll only be able to use it in Delphi and Borland C++Builder. If you buy an ActiveX control, on the other hand, you'll be able to use it in multiple development environments from multiple vendors. Even so, if you develop mainly in Delphi and find two similar components based on the two technologies, I suggest you buy the Delphi component—it will be more integrated with your environment, and therefore easier for you to use. Also, the native Delphi component will probably be better documented (from the Delphi perspective), and it will take advantage of Delphi and its language features not available in the general ActiveX interface, which is traditionally based on C and C++.

**Note** In the .NET world, this situation will change completely. Not only will you be able to use any system component in a more seamless way, but you'll also be able to make Delphi components available to other .NET programming languages and tools.

## Using ActiveX Controls in Delphi

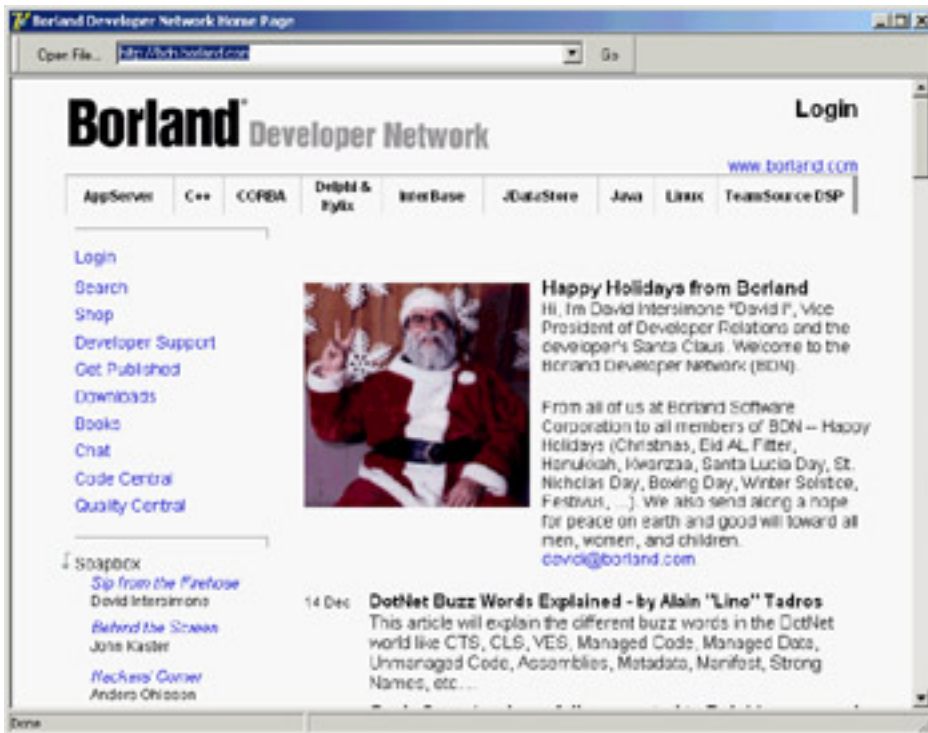
Delphi comes with some preinstalled ActiveX controls, and you can buy and install more third-party ActiveX controls. After this description of how ActiveX controls work in general, I'll demonstrate one in an example.

The Delphi installation process is simple:

1. Select Component → Import ActiveX Control in the Delphi menu to open the Import ActiveX dialog box, where you can see the list of ActiveX control libraries registered in Windows.
2. Choose one, and Delphi will read its type library, list its controls, and suggest a filename for its unit.
3. If the information is correct, click the Create Unit button to view the Delphi language source code file created by the IDE as a *wrapper* for the ActiveX control.
4. Click the Install button to add this new unit to a Delphi package and to the Component Palette.

## Using the WebBrowser Control

To build an example, I've used a preinstalled ActiveX control available in Delphi. Unlike the third-party controls, it is not available in the ActiveX page of the palette, but in the Internet page. The control, called WebBrowser, is a wrapper around Microsoft's Internet Explorer engine. The WebDemo example is a very limited web browser; it has a `TWebBrowser` ActiveX control covering its client area, a control bar at the top, and a status bar at the bottom. To move to a given web page, a user can type a URL in the toolbar's combo box, select one of the visited URLs (saved in the combo box), or click the Open File button to select a local file. [Figure 12.8](#) shows an example of this program.



**Figure 12.8:** The WebDemo program after choosing a page that's well known by Delphi developers

The implementation of the code used to select a web or local HTML file is in the `GotoPage` method:

```
procedure TForm1.GotoPage(ReqUrl: string);
begin
    WebBrowser1.Navigate (ReqUrl, EmptyParam, EmptyParam, EmptyParam,
        EmptyParam);
end;
```

`EmptyParam` is a predefined `OleVariant` you can use to pass a default value as a reference parameter. This is a handy shortcut you can use to avoid creating an empty `OleVariant` variable each time you need a similar parameter. The program calls the `GotoPage` method when the user presses the Open File button, or when the user presses the Enter key while in the combo box or clicks the Go button, as you can see in the source code. The program also handles four events of the `WebBrowser` control. When the download operations ends, the program updates the text of the status bar and also the combo box's drop-down list:

```
procedure TForm1.WebBrowser1DownloadComplete(Sender: TObject);
var
    NewUrl: string;
begin
    StatusBar1.Panels[0].Text := 'Done';
    // add URL to combo box
    NewUrl := WebBrowser1.LocationURL;
    if (NewUrl <> '') and (ComboURL.Items.IndexOf (NewUrl) < 0) then
        ComboURL.Items.Add (NewUrl);
end;
```

Other useful events are the `OnTitleChange`, used to update the caption with the title of the HTML document, and the `OnStatusTextChange` event, used to update the second part of the status bar. This code basically duplicates the information displayed in the first part of the status bar by the previous two event handlers.

## Writing ActiveX Controls

Besides using existing ActiveX controls in Delphi, you can easily develop new ones, using one of two techniques:

- You can use the ActiveX Control Wizard to turn a VCL control into an ActiveX control. You begin from an existing VCL component, which must be a `TWinControl` descendant (and must not have inappropriate properties, in which case it is removed from the combo box of the Wizard), and Delphi wraps an ActiveX around it. During this step, Delphi adds a type library to the control. (Wrapping an ActiveX control around a Delphi component is the opposite of what you did to use an ActiveX control in Delphi.)
- You can create an ActiveForm, place several controls in it, and use the entire form (without borders) as an ActiveX control. This second technique was introduced to build Internet applications, but it is also a very good alternative for constructing an ActiveX control based on multiple Delphi controls or on Delphi components that do not descend from `TWinControl`.

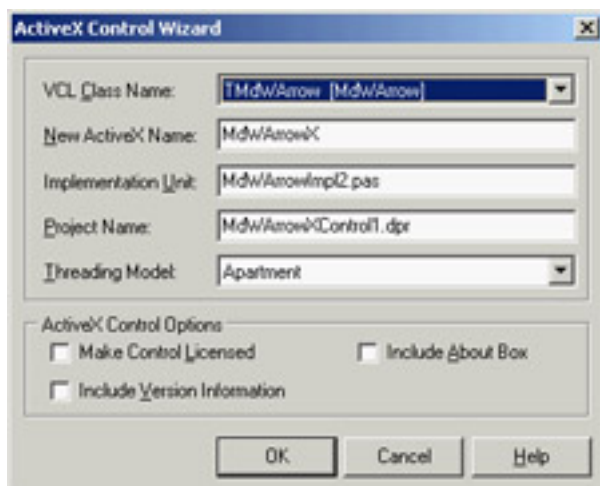
In either case, you can optionally prepare a property page for the control, to use as a sort of property editor for setting the initial value of the control's properties in any development environment—an alternative to the Object Inspector in Delphi. Because most development environments allow only limited editing, it is more important to write a property page than it is to write a component or property editor for a Delphi control.

### Building an ActiveX Arrow

As an example of the development of an ActiveX control, I've decided to take the Arrow component from [Chapter 9](#), "Writing Delphi Components," and turn it into an ActiveX control. You cannot use that component directly, because it is a graphical control (a subclass of `TGraphicControl`). However, turning a graphical control into a window-based control is usually a straightforward operation.

In this case, you change the base class name to `TCustomControl` (and change the name of the control's class to `TMdWArrow`, as well, to avoid a name clash), as you can see in the source code files in the `XArrow` folder. After installing this component in Delphi, you are ready to begin developing the new example. To create a new ActiveX library, select File → New → Other, move to the ActiveX page, and choose ActiveX library. Delphi creates the bare skeleton of a DLL, as you saw at the beginning of this chapter. I've saved this library as `XArrow`, in a directory with the same name, as usual.

Now it is time to use the ActiveX Control Wizard, available in the ActiveX page of the Object Repository—Delphi's New dialog box:



In this wizard, you select the VCL class you are interested in, customize the names shown in the edit boxes, and click OK; Delphi then builds the complete source code of an ActiveX control for you.



The use of the three check boxes at the bottom of the ActiveX Control Wizard window may not be obvious. If you make the control licensed, Delphi will include a license key in the code and provide this same GUID in a separate .LIC file. This license file is necessary to use the control in a design environment without the proper *license key* for the control or use it within a web page. The second check box allows you to include version information for the ActiveX control in the OCX file. If the third check box is selected, the ActiveX Control Wizard automatically adds an About box to the control.

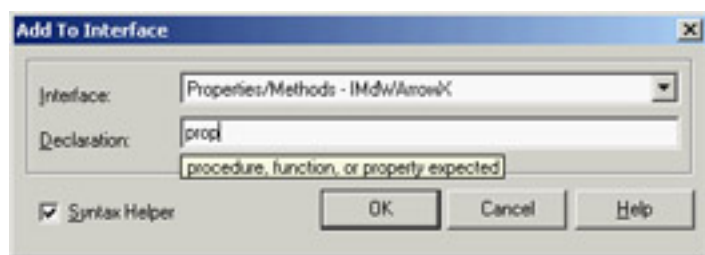
Look at the code the ActiveX Control Wizard generates. This wizard's key element is the generation of a type library and, of course, a corresponding type library import unit with the definition of an interface (*dispinterface*) and other types and constants. In this example, the import file is named `XArrow_TLB.PAS`: I suggest you study it to understand how Delphi defines an ActiveX control. The unit includes a GUID for the control, constants for the definition of values corresponding to the COM enumerated types used by properties of the Delphi control (like `TxMdWArrowDir`), and the declaration of the `IMdWArrowX` interface. The final part of the import unit includes the declaration of the `TMdWArrowX` class. This is a `TOLEControl`-derived class you can use to install the control in Delphi, as you saw in the first part of this chapter. You don't need this class to build the ActiveX control; you need it to install the ActiveX control in Delphi.

The rest of the code, and the code you'll customize, is in the main unit, which in the `XArrow` example is called `MdWArrowImpl1`. This unit has the declaration of the ActiveX server object, `TMdWArrowX`, which inherits from `TActiveXControl` and implements the specific `IMdWArrowX` interface.

Before you customize this control, let's see how it works. Compile the ActiveX library and then register it using Delphi's `Run → Register ActiveX Server` menu command. Now you can install the ActiveX control as you've done in the past, except you have to specify a different name for the new class to avoid a name clash. If you use this control, it doesn't look much different from the original VCL control, but the same component can now be installed in other development environments.

## Adding New Properties

Once you've created an ActiveX control, adding new properties, events, or methods to it—surprisingly—simpler than doing the same operation for a VCL component. Delphi, provides specific visual support for adding properties, methods, or events to an ActiveX control, but not for a VCL control. You can open the Delphi unit with the implementation of the ActiveX control, and choose `Edit → Add To Interface`. As an alternative, you can use the same command from the editor's shortcut menu. Delphi opens the Add To Interface dialog box:



In the combo box, you can choose between a new property, method, or event. In the edit box, you can then type the declaration of this new interface element. If the Syntax Helper check box is activated, you'll get hints describing what you should type next and highlighting any errors. When you define a new ActiveX interface element, keep in mind that you are restricted to COM data types.

In the `XArrow` example, I've added two properties to the ActiveX control. Because the `Pen` and `Brush` properties of the original Delphi components are not accessible, I've made their color available. These are examples of what you can write in the Add To Interface dialog box's edit box (executing it twice):

```
property FillColor: Integer;  
property PenColor: Integer;
```

**Note** Because a `TColor` is a specific Delphi definition, it is not legal to use it. `TColor` is an integer subrange that defaults to integer size, so I've used the standard `Integer` type directly.

The declarations you enter in the Add To Interface dialog box are automatically added to the control's type library (TLB)

file, to its import library unit, and to its implementation unit. All you have to do to finish the ActiveX control is fill in the `Get` and `Set` methods of the implementation. If you now install this ActiveX control in Delphi once more, the two new properties will appear. The only problem with this property is that Delphi uses a plain integer editor, making it difficult to enter the value of a new color by hand. A program, by contrast, can easily use the `RGB` function to create the proper color value.

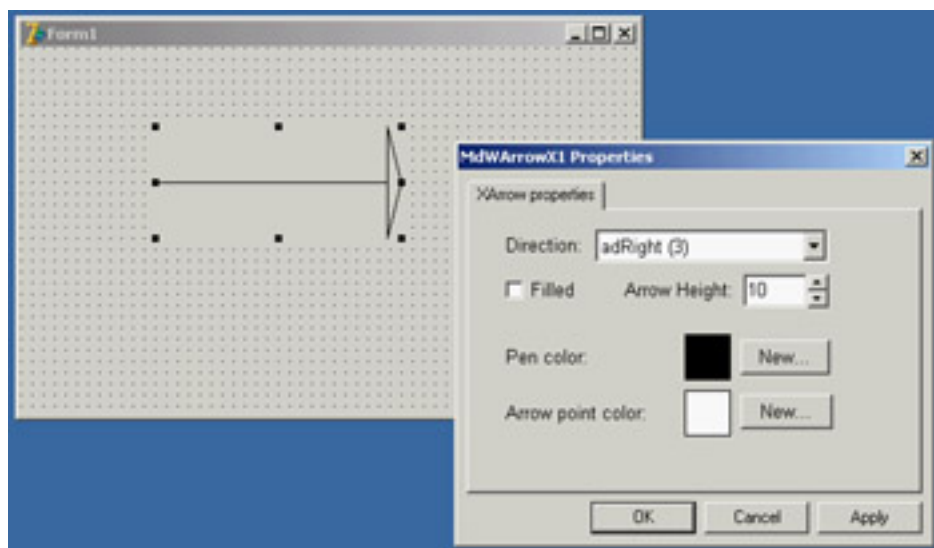
## Adding a Property Page

As it stands, other development environments can do very little with your component, because you've prepared no property page—no property editor. A property page is fundamental so that programmers using the control can edit its attributes. However, adding a property page is not as simple as adding a form with a few controls. The property page, will integrate with the host development environment. The property page for your control will show up in a property page dialog of the host environment, which will provide the OK, Cancel, and Apply buttons, and the tabs for showing multiple property pages (some of which may be provided by the host environment).

The nice thing is that support for property pages is built into Delphi, so adding one takes little time. You open an ActiveX project, then open the usual New Items dialog box, move to the ActiveX page, and choose Property Page. What you get is not very different from a form—the `TPropertyPage1` class (created by default) inherits from the `TPropertyPage` class of VCL, which in turn inherits from `TCustomForm`.

**Tip** Delphi provides four built-in property pages for colors, fonts, pictures, and strings. The GUIDs of these classes are indicated by the constants `Class_DColorPropPage`, `Class_DFontPropPage`, `Class_DPicturePropPage`, and `Class_DStringPropPage` in the `AxCtrls` unit.

In the property page, you can add controls as in a normal Delphi form, and you can write code to let the controls interact. In the `XArrow` example, I've added to the property page a combo box with the possible values of the `Direction` property, a check box for the `Filled` property, an edit box with an `UpDown` control to set the `ArrowHeight` property, and two shapes with corresponding buttons for the colors. You can see this form in the Delphi IDE while working on the ActiveX control in [Figure 12.9](#).



**Figure 12.9:** The `XArrow` ActiveX control and its property page, hosted by the Delphi environment

The only code added to the form relates to the two buttons used to change the color of the two shape components, which offer a preview of the ActiveX control's color. The button's `OnClick` event uses a `ColorDialog` component, as usual:

```
procedure TPropertyPage1.ButtonPenClick(Sender: TObject);  
begin  
    with ColorDialog1 do  
        begin
```



```

Color := ShapePen.Brush.Color;
if Execute then
begin
    ShapePen.Brush.Color := Color;
    Modified; // enable Apply button!
end;
end;
end;

```

It's important to notice in this code the call to the `Modified` method of the `TPropertyPage` class. This call is required to let the property page dialog box know you've modified one of the values and to enable the Apply button. When a user interacts with one of the other controls on the form, this `Modified` call is made automatically to the `TPropertyPage` class method that handles the internal `cm_Changed` message. As a user, you don't *change* the buttons for these controls, however, you need to add this line yourself.

**Tip** Another tip relates to the *Caption* of the property page form. It will be used in the host environment's property dialog box as the caption of the tab corresponding to the property page.

The next step is to associate the property page's controls with the ActiveX control's properties. The property page class automatically has two methods for this functionality: `UpdateOleObject` and `UpdatePropertyPage`. As their names suggest, these methods copy data from the property page to the ActiveX control and vice versa, as you can see in the example code.

The final step is to connect the property page to the ActiveX control. When the control was created, the Delphi ActiveX Control Wizard automatically added a declaration for the `Define-PropertyPages` method to the implementation unit. In this method, you call the `DefinePropertyPage` method (this time the method name is singular) for each property page you want to add to the control. The parameter of this method is the GUID of the property page, which you can find in the corresponding unit:

```

procedure TMDWArrowX.DefinePropertyPages (
    DefinePropertyPage: TDefinePropertyPage);
begin
    DefinePropertyPage(Class_PropertyPage1);
end;

```

You've finished developing the property page. After recompiling and reregistering the ActiveX library, you can install the ActiveX control in a host development environment (including Delphi) and see how it looks, as I already did in [Figure 12.9](#).

## ActiveForms

As I've mentioned, Delphi provides an alternative to the use of the ActiveX Control Wizard to generate an ActiveX control. You can use an ActiveForm, which is an ActiveX control that is based on a form and can host one or more Delphi components. This technique is used in Visual Basic to build new controls, and it makes sense when you want to create a compound component.

In the XClock example, I've placed on an ActiveForm a label (a graphic control that cannot be used as a starting point for an ActiveX control) and a timer, and connected the two with a little code. The form/control becomes a container of other controls, which makes it easy to build compound components (easier than for a VCL compound component).

To build such a control, select the ActiveForm icon in the ActiveX page of the File → New dialog box. Delphi will ask you for some information in the ActiveForm Wizard dialog box, which is similar to the ActiveX Control Wizard dialog box.

## ActiveForm Internals

Before you continue with the example, let's look at the code generated by the ActiveForm Wizard. The key difference

from a plain Delphi form is in the declaration of the new form class, which inherits from the `TActiveForm` class and implements a specific `ActiveForm` interface. The code generated for the active form class implements quite a few `Set` and `Get` methods, which change or return the corresponding properties of the Delphi form; this code also implements the events, which again are the events of the form.

The `TForm` events are set to the internal methods when the form is created. For example:

```
procedure TAXForm1.Initialize;
begin
  OnActivate := ActivateEvent;
  ...
end;
```

Each event then maps itself to the external `ActiveX` event, as in the following method:

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;
```

Because of this mapping, you should not handle the form's events directly. Instead, you can either add code to these default handlers or override the `TForm` methods that end up calling the events. This mapping problem relates only to the events of the form itself, not to the events of the form's components. You can continue to handle the components' events as usual.

**Note** These problems (and possible solutions) are demonstrated by the `XForm1` example. I won't discuss it in detail, but leave it for you as a self-study example.

## The XClock ActiveX Control

Now that I've covered some foundations, let's return to the development of the `XClock` example:

1. Place on the form a timer and a label with a large font and centered text, aligned to the client area.
2. Write an event handler for the timer's `OnTimer` event, so that the control updates the output of the label with the current time every second:

```
procedure TXClock.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr (Time);
end;
```

3. Compile this library, register it, and install it in a package to test it in the Delphi environment.

Notice the effect of the sunken border. This is controlled by the active form's `AxBorderStyle` property, one of the few properties of active forms that is not available for a plain form.

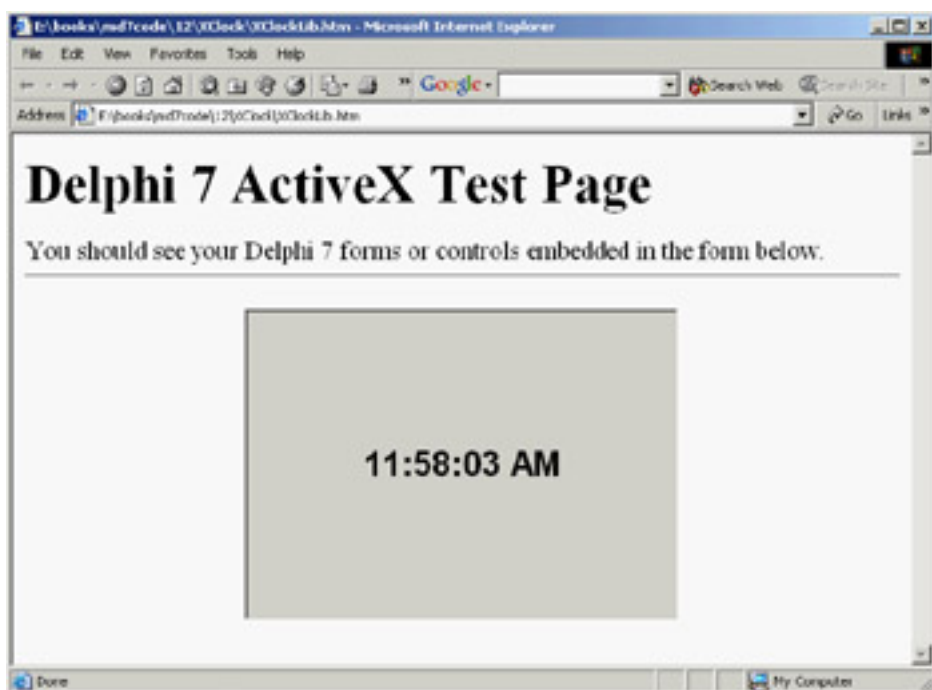
## ActiveX in Web Pages

In the previous example, you used Delphi's `ActiveForm` technology to create a new `ActiveX` control. An `ActiveForm` is an `ActiveX` control based on a form. Borland documentation often implies that `ActiveForms` should be used in HTML pages, but you can use any `ActiveX` control on a web page. Basically, each time you create an `ActiveX` library, Delphi should enable the `Project → Web Deployment Options` and `Project → Web Deploy` menu items.

**Warning** Due to what I consider a bug, in Delphi 7 these commands are activated only for an `ActiveForm`. If they are disabled, you can use a trick: Add an `ActiveForm` to your current `ActiveX` library, which will enable the menu commands; then immediately remove the `ActiveForm`, and the menu items will still be available. The trouble is, you'll have to repeat this operation every time you reopen the project—that is, until Borland fixes the bug.

The first command allows you to specify how and where to deliver the proper files. In this dialog box you can set the server directory for deploying the ActiveX component, the URL of this directory, and the server directory for deploying the HTML file (which will have a reference to the ActiveX library using the URL you provide).

You can also specify the use of a compressed CAB file, which can store the OCX file and other auxiliary files, such as packages, making it easier and faster to deliver the application to the user. A compressed file, means a faster download. I've generated the HTML file and CAB file for the XClock project in the same directory. Opening this HTML file in Internet Explorer produces the output shown in [Figure 12.10](#). If all you get is a red X marker indicating a failure to download the control, there are various possible explanations for this problem: Internet Explorer doesn't allow the download of controls, it doesn't match the security level for the unsigned control, there is a mismatch in the control version number, and so on.



**Figure 12.10:** The XClock control in the sample HTML page

Notice that in the portion of the HTML file referring to the control, you can use the special `param` tag to customize the control's properties. For example, in the XArrow control's HTML file, I've modified the automatically generated HTML file (in the XArrowCust.htm file) with these three `param` tags:

```
<object classid="clsid:482B2145-4133-11D3-B9F1-00000100A27B"
  codebase="./XArrow.cab" width="350" height="250" align="center"
  hspace="0" vspace="0">
  <param name="ArrowHeight" value="100">
  <param name="Filled" value="-1">
  <param name="FillColor" value="111829">
</object>
```

Although this might seem to be a useful technique, it is important to consider the (limited) role of an ActiveX form placed in a web page. It corresponds to letting a user download and execute a custom Windows application, which raises many concerns about security. An ActiveX control can access the computer's system information, such as the user name, directory structure, and so on. I could continue, but my point is clear.

## Introducing COM+

In addition to plain COM servers, Delphi also allows you to create enhanced COM objects, including stateless objects and transaction support. Microsoft first introduced this type of COM object with the MTS (Microsoft Transaction Server) acronym in Windows NT and 98, and later renamed it COM+ in Windows 2000/XP (I'll call it COM+, but I'm referring to both MTS and COM+).

Delphi supports building both standard stateless objects and DataSnap remote data modules based on stateless objects. In both cases, you begin development by using one of the available Delphi wizards, using the New Items dialog box and selecting the Transactional Object icon on the ActiveX page or the Transactional Data Module icon on the Multitier page. You must add these objects to an ActiveX library project, not to a plain application. Another icon, COM+ Event Object, is used to support COM+ events.

COM+ provides a run-time environment supporting database transaction services, security, resource pooling, and an overall improvement in robustness for DCOM applications. The run-time environment manages objects called *COM+ components*. These are COM objects stored in an in-process server (that is, a DLL). Whereas other COM objects run directly in the client application, COM+ objects are handled by this run-time environment, in which you install the COM+ libraries. COM+ objects must support specific COM interfaces, starting with `IObjectControl`, which is the base interface (like `IUnknown` for a COM object).

Before getting into too many technical and low-level details, let's consider COM+ from a different perspective: the benefits of this approach. COM+ provides a few interesting features, including:

**Role-Based Security** The role assigned to a client determines whether it has the right to access the interface of a data module.

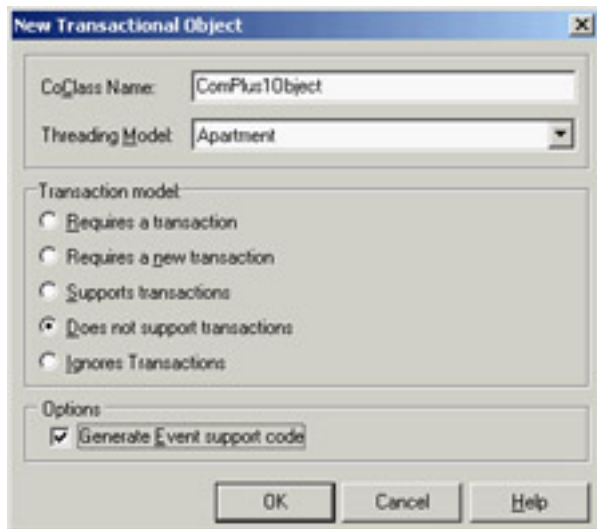
**Reduced Database Resources** You can reduce the number of database connections, because the middle tier logs on to the server and uses the same connections for multiple clients (although you cannot have more clients connected at once than you have licenses for the server).

**Database Transactions** COM+ transaction support includes operations on multiple databases, although few SQL servers other than Microsoft's support COM+ transactions.

## Creating a COM+ Component

The starting point for creating a COM+ component is the creation of an ActiveX library project. Then, follow these steps:

1. Select a new Transactional Object in the ActiveX page of the New Items dialog box.
2. In the resulting dialog box (see [Figure 12.11](#)), enter the name of the new component (*ComPlus1Object* in my ComPlus1 example).



**Figure 12.11:** The New Trans-actional Object dialog box, used to create a COM+ object

The New Transactional Object dialog box allows you to enter a name for the class of the COM+ object, the threading model (because COM+ serializes all the requests, Single or Apartment will generally do), and a transactional model:

**Requires a Transaction** Indicates that each call from the client to the server is considered a transaction (unless the caller supplies an existing transaction context).

**Requires a New Transaction** Indicates that each call is considered a new transaction.

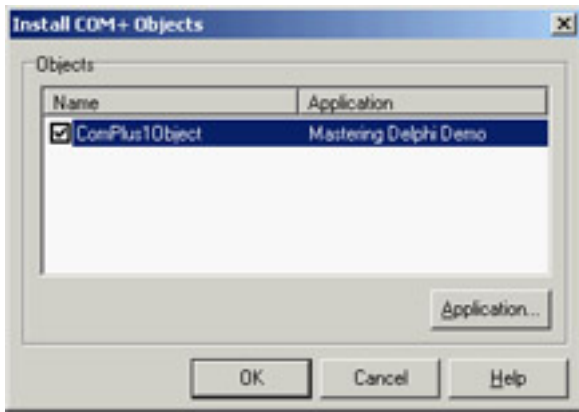
**Supports Transactions** Indicates that the client must explicitly provide a transaction context.

**Does Not Support Transaction** (The default choice, and the one I've used.) Indicates that the remote data module won't be involved in any transaction. This option prevents the object from being activated if the client calling it has a transaction.

**Ignores Transactions** Indicates that objects do not participate in transactions, but can be used regardless of whether the client has a transaction.

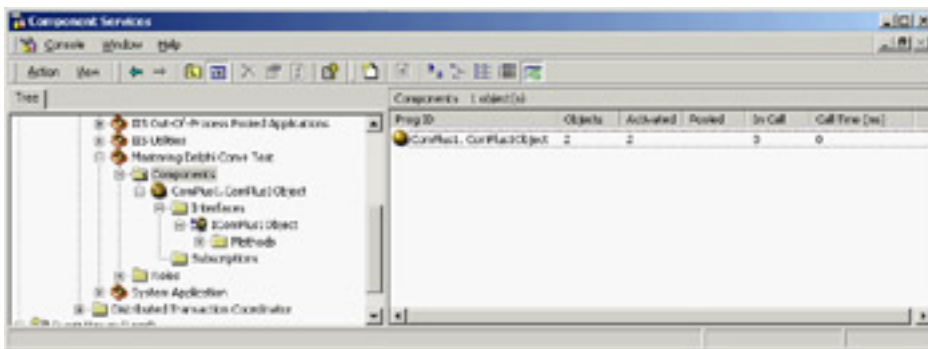
3. As you close this dialog, Delphi adds a type library and an implementation unit to the project and opens the type-library editor, where you can define the interface of your new COM object. For this example, add a `Value` integer property, an `Increase` method having as its parameter an amount, and an `AsText` method returning a `WideString` with the formatted value.
4. As you accept the edits in the type-library editor (by clicking the Refresh button or closing the window), Delphi shows the Implementation File Update Wizard, but only if you set the *Display updates before refreshing* option of the Type Library page of the Environment Options dialog box. This wizard asks for your confirmation before adding four methods to the class, including the get and set methods of the property. You can now write some code for the COM object, which in my example is quite trivial.

Once you've compiled an ActiveX library, or COM library, which hosts a COM+ component, you can use the Component Services administrative tool (shown in the Microsoft Management Console, or MMC) to install and configure the COM+ component. Even better, you can use the Delphi IDE to install the COM+ component using the Run → Install COM+ Object menu command. In the subsequent dialog box, you can select the component to install (a library can host multiple components) and choose the COM+ application where you want to install the component:



A COM+ application is nothing more than a way to group COM+ components; it is not a program or anything like one (why they call it an application is not clear to me). So, in the Install COM+ Object dialog, you can select an existing application/group, choose the Install Into New Application page, and enter a name and description.

I've called the COM+ application *Mastering Delphi Com+ Test*, as you can see in [Figure 12.12](#) in Microsoft's Component Services administration console. This is the front end you can use to fine-tune the behavior of your COM+ components, setting their activation model (just-in-time activation, object pooling, and so on), their transaction support, and the security and concurrency models you want to use. You can also use this console to monitor the objects and method calls (in case they take a long time to execute). In [Figure 12.12](#), you can see that there are currently two active objects.



**Figure 12.12:** The newly installed COM+ component in a custom COM+ application (as shown by Microsoft's Component Services tool)

**Warning** Because you've created one or more objects, the COM library remains loaded in the COM+ environment and some of the objects may be kept in cache, even if no clients are connected to them. For this reason, you generally cannot recompile the COM library after using it, unless you use the MMC to shut it down or set a Transaction Timeout of 0 seconds in MMC.

I've created a client program for the COM+ object, but it is like any other Delphi COM client. After importing the type library, which is automatically registered while installing the component, I created an interface-type variable referring to it and called its methods as usual.

## Transactional Data Modules

The same types of features are available when you create a *transactional data module*—a remote data module within a COM+ component. Once you've created a transactional data module, you can build a Delphi DataSnap application (as you'll see in [Chapter 16](#), "Multitier DataSnap Applications"). You can add one or more dataset components, add one or more providers, and export the provider(s). You can also add custom methods to the data module type library by editing the type library or using the Add To Interface command.

Within a COM+ component or transactional data module, you can also use specific methods that support transactions. These methods are technically provided, at a lower level, in the `IObjectContext` interface returned by the



`GetObjectContext` method:

- `SetComplete` tells the COM+ environment the object has finished working and can be deactivated, so that the transaction can be committed.
- `EnableCommit` indicates that the object hasn't finished but the transaction should be committed.
- `DisableCommit` stops the commit operation, even if the method is done, disabling the object deactivation between method calls.
- `SetAbort` says the object has finished and can be activated but the transaction cannot be committed.
- `IsInTransaction` checks whether the object is part of a transaction.

Other methods of the `IContextObject` interface include `CreateInstance`, which creates another COM+ object in the same context and within the current transaction; `IsCallerInRole`, which checks if the object's caller is in a particular "security" role; and `IsSecurityEnabled` (whose name is self-explanatory).

Once you've built a transactional data module within a server library, you can install it as I showed earlier for a plain COM+ object. After the transactional data module has been installed, it will be directly available to other applications and visible in the management console.

An important feature of COM+ is that it becomes much easier to configure DCOM support using this environment. A client computer's COM+ environment can grab information from a server computer's COM+ environment, including registration information for the COM+ object you want to be able to call over a network. The same network configuration is much more complex if done with plain DCOM, without MTS or COM+.

**Tip** Even though COM+ configuration is much better than DCOM configuration, you are limited to computers with a recent version of the Windows operating system. Considering that even Microsoft is moving away from DCOM technology, before you build a large system based on it you should evaluate the alternative provided by SOAP (discussed in [Chapter 22](#), "Using XML Technologies").

## COM+ Events

Client applications that use traditional COM objects and Automation servers can call methods of those servers, but this is not an efficient way to check whether the server has updated data for the client. For this reason, a client can define a COM object that implements a *callback* interface, pass this object to the server, and let the server call it. Traditional COM events (which use the `IConnectionPoint` interface) are simplified by Delphi for Automation objects, but are still complex to handle.

COM+ introduces a simplified event model, in which the events are COM+ components and the COM+ environment manages the connections. In traditional COM callbacks, the server object has to keep track of the multiple clients it has to notify, something Delphi doesn't provide us automatically (the default Delphi event code is limited to a single client). To support COM callbacks for multiple clients you need to add the code to hold references to each of the clients. In COM+, the server calls into a single event interface, and the COM+ environment forwards the event to all clients that have expressed interest in it. This way, the client and the server are less coupled, making it possible for a client to receive notification from different servers without any change in its code.

**Note** Some critics say that Microsoft introduced this model only because it was difficult for Visual Basic developers to handle COM events in the traditional way. Windows 2000 provided a few operating-system features specifically intended for VB developers.

To create a COM+ event, you should create a COM library (or ActiveX library) and use the COM+ Event Object wizard. The resulting project will contain a type library with the definition of the interface used to fire the events, plus some *fake* implementation code. The server that receives the notification of the events will provide the interface implementation. The fake code is there only to support Delphi's COM registration system.

While building the MdComEvents library, I added to the type library a single method with two parameters, resulting in the following code (in the interface definition file):

```
type
  IMdInform = interface(IDispatch)
    [ '{202D2CC8-8E6C-4E96-9C14-1FAAE3920ECC}' ]
    procedure Informs(Code: Integer; const Message: WideString); safecall;
end;
```

The main unit includes the fake COM object (notice that the method is abstract, so it has no implementation) and its class factory, to let the server register itself. At this point, you can compile the library and install it in the COM+ environment, following these steps:

1. In Microsoft's Component Services console, select a COM+ application, move to the Components folder, and use the shortcut menu to add a new component to it.
2. In the COM Component Install Wizard, click the Install New Event Class button and select the library you've just compiled. Your COM+ event definition will be automatically installed.

To test whether it works, you'll have to build an implementation of this event interface and a client invoking it. The implementation can be added to another ActiveX library, hosting a plain COM object. Within Delphi's COM Object Wizard, you can select the interface to implement from the list that appears when you click the List button.

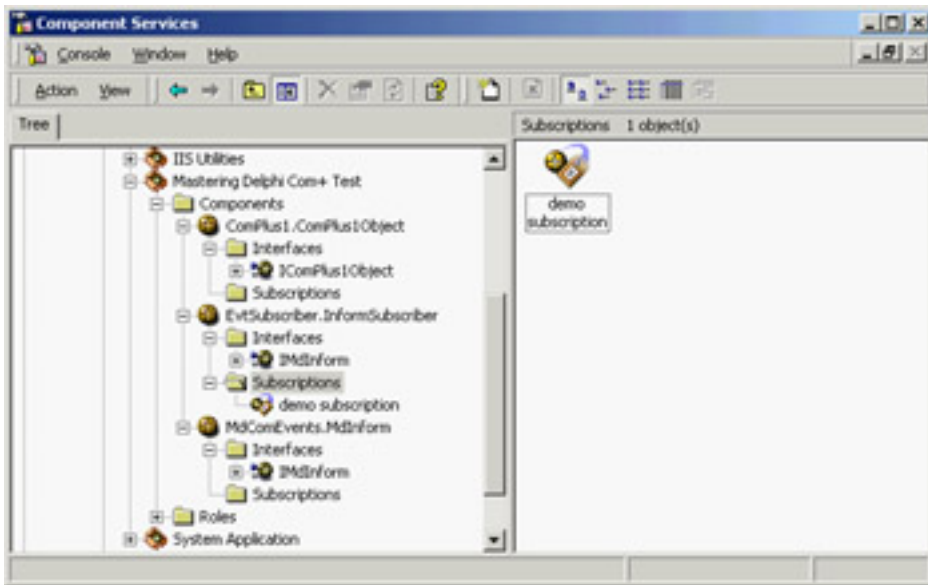
The resulting library, which in my example is called EvtSubscriber, exposes an Automation object: a COM object implementing the IDispatch interface (which is mandatory for COM+ events). The object has the following definition and code:

```
type
  TInformSubscriber = class(TAutoObject, IMdInform)
  protected
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;
```

```
procedure TInformSubscriber.Informs(Code: Integer; const Message: WideString);
begin
  ShowMessage ('Message <' + IntToStr (Code) + '>: ' + Message);
end;
```

After compiling this library, you can first install it into the COM+ environment, and then bind it to the event. This second step is accomplished in the Component Services management console by selecting the Subscriptions folder under the event object registration, and using the New → Subscription shortcut menu. In the resulting wizard, choose the interface to implement (there is probably only one interface in your COM+ event library); you'll see a list of COM+ components that implement this interface. Selecting one or more of them sets up the subscription binding, which is listed under the Subscriptions folder. You can see an example of my configuration while building this example in [Figure 12.13](#).





**Figure 12.13:** A COM+ event with two subscriptions in the Component Services management console

Finally, you can focus on the application that fires the event, which I've called Publisher (because it publishes the information other COM objects are interested in). This is the simplest step of the process, because it is a plain COM client that uses the event server. After importing the COM+ event type library, you can add to the publisher code like this:

```
var
  Inform: IMdInform;
begin
  Inform := CoMdInform.Create;
  Inform.Informs (20, Edit1.Text);
```

My example creates the COM object in the `FormCreate` method to keep the reference around, but the effect is the same. Now the client program thinks it is calling the COM+ event object, but this object (provided by the COM+ environment) calls the method for each of the active subscribers. In this case you'll end up seeing a message box:



To make things more interesting, you can subscribe the same server twice to the event interface. The net effect is that without touching your client code, you'll get two message boxes, one for each of the subscribed servers. Obviously this effect becomes interesting when you have multiple different COM components that can handle the event, because you can easily enable and disable them in the management console, changing the COM+ environment without modifying any program code.

## COM and .NET in Delphi 7

While putting out the new .NET infrastructure, Microsoft has tried to help companies that are continuing their existing programs. One of these migration paths is represented by the compatibility of .NET objects with COM objects. You can use an existing COM object in a .NET application, although not in the realm of managed and safe code. You can also use .NET assemblies from Windows applications as if they were native COM objects. This functionality takes place thanks to wrappers provided by Microsoft.

Borland's claim to support COM/.NET interoperability in Delphi 7 is mainly a reference to the fact that COM objects compiled with Delphi won't create trouble for the .NET importer. In addition, Delphi's type library importer can work seamlessly with .NET assemblies as with standard COM libraries.

Having said this, unless you have an existing large investment in COM, I'd discourage you from following this path. If you want to bet on Microsoft technologies, the future lies in native .NET solutions. If you don't like Microsoft technologies or want a cross-platform solution, COM will still be a worse choice than .NET (in the future, we may have a .NET framework for other operating systems).

**Tip** The steps suggested here should work also in Delphi 6. Delphi 7 adds an automatic import system that at times has troubles with some of the code generated by the compiler of the Delphi for .NET Preview.

To demonstrate .NET importing features, I've created a .NET library with an interface and a class implementing it. The interface and class resemble those of the FirstCom example discussed at the beginning of this chapter. Here is the code of the library, which must be compiled with the Delphi for .NET preview compiler. You must create an object, or the linker will remove almost everything from your compiled library (*assembly*, in .NET jargon):

```
library NetLibrary;  
  
uses  
    NetNumberClass in 'NetNumberClass.pas';  
  
{ $R *.res }  
  
begin  
    // create an object to link all of the code  
    TNumber.Create;  
end.
```

The code is in the NetNumberClass unit, which defines an interface and a class implementing it:

```
type  
    INumber = interface  
        function GetValue: Integer;  
        procedure SetValue (New: Integer);  
        procedure Increase;  
    end;  
  
    TNumber = class(TObject, INumber)  
    private  
        fValue: Integer;  
    public  
        constructor Create;  
        function GetValue: Integer;  
        procedure SetValue (New: Integer);  
        procedure Increase;  
    end;
```

Notice that, unlike a COM server, the interface doesn't require a GUID, following .NET rules (although it can have one

using an attribute of the `GuidAttribute` class). The system will generate one for you. After compiling this code (available in the `NetImport` folder of this chapter's code) with Delphi for .NET Preview (not with Delphi 7!), you need to perform two steps: First, run Microsoft's .NET Framework Assembly Registration Utility, *regasm*; second, run Borland's Type Library Importer, *tlibimp*. (In theory, you should be able to skip this step and directly use the Import Type Library dialog box, but with some libraries the use of the *tlibimp* program is required.)

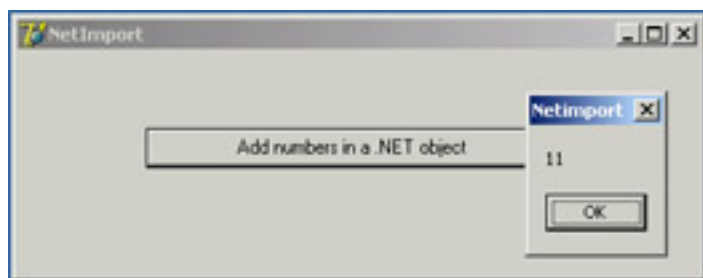
In practice, go to the folder where you've compiled the library and type from the command line the two commands in bold (you should see the rest of the text I've captured here):

```
C:\md7code\NetImport>regasm netlibrary.dll
Microsoft (R) .NET Framework Assembly Registration Utility 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Types registered successfully
```

```
C:\md7code\NetImport>tlibimp netlibrary.dll
Borland TLIBIMP Version 7.0
Copyright (c) 1997, 2002 Borland Software Corporation
Type library loaded ....
Created E:\books\md7code\12\NetImport\mscorlib_TLB.dcr
Created E:\books\md7code\12\NetImport\mscorlib_TLB.pas
Created E:\books\md7code\12\NetImport\NetLibrary_TLB.dcr
Created E:\books\md7code\12\NetImport\NetLibrary_TLB.pas
```

The effect is to create a unit for the project's type library and a unit for the imported Microsoft .NET Core Library (`mscorlib.dll`). Now you can create a new Delphi 7 application (a standard Win32 program) and use the .NET objects as if they were COM objects. Here is the code from the `NetImport` example, shown in [Figure 12.14](#):



**Figure 12.14:** The `NetImport` program uses a .NET object to sum numbers.

```
uses
  NetLibrary_TLB;

procedure TForm1.btnAddClick(Sender: TObject);
var
  num: INumber;
begin
  num := CoTNumber.Create as INumber;
  num.Increase;
  ShowMessage (IntToStr (num.GetValue));
end;
```

## What's Next?

In this chapter, I have discussed applications of Microsoft's COM technology, covering automation, documents, controls, and more. You've seen how Delphi makes the development of Automation servers and clients and ActiveX controls, reasonably simple. Delphi even enables you to wrap components around Automation servers, such as Word and Excel. I've also introduced elements of COM+ and discussed briefly the use of ActiveForms inside a browser. I've stated this is not a good approach to Internet web programming—a topic discussed later in the book.

As I mentioned earlier, if COM has a key role in Windows 2000/XP, future versions of Microsoft's operating systems will downplay its role to push the .NET infrastructure (including SOAP and XML). But you'll have to wait until [Chapter 23](#) for a complete discussion of Delphi SOAP support.

## Part III: Delphi Database-Oriented Architectures

### Chapter List

[Chapter 13](#): Delphi's Database Architecture

[Chapter 14](#): Client/Server with dbExpress

[Chapter 15](#): Working with ADO

[Chapter 16](#): Multitier DataSnap Applications

[Chapter 17](#): Writing Database Components

[Chapter 18](#): Reporting with Rave

## Chapter 13: Delphi's Database Architecture

### Overview

Delphi's support for database applications is one of the programming environment's key features. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming.

What you *won't* find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports database access.

I'll begin with an explanation of the alternatives Delphi offers in terms of data access, and then I'll provide an overview of the database components available in Delphi. This chapter focuses on the use of the `TClientDataSet` component for accessing local data, deferring all the client/server database access to [Chapter 14](#), "Client/Server with dbExpress." I'll include an overview of the `TDataSet` class, an in-depth analysis of the `TField` components, and the use of data-aware controls. [Chapter 14](#) follows up by providing information about more advanced database programming topics, particularly client/server programming with the use of the dbExpress library (and the InterBase Express components).

Finally, notice that almost everything discussed in this chapter is cross-platform. In particular, the examples can be ported to CLX and to Linux by recompiling and referring to CDS files in the proper folders.

## Accessing a Database: dbExpress, Local Data, and Other Alternatives

The early incarnations of Delphi—immediately adopted as a tool for building database-oriented applications—could access a database only by means of the Borland Database Engine (BDE). Starting with Delphi 3, the portion of VCL related to database access was restructured to open it up to multiple database access solutions, which currently include ADO, native InterBase components, the dbExpress library, and the BDE. Many third-party vendors are now able to offer alternative database access mechanisms to a variety of data formats (some not accessible through Borland components) and still provide a solution integrated with Delphi's VCL.

**Tip** In Kylix, the overall picture is slightly different. Borland decided not to port the old BDE technology to Linux and focused instead on the new thin database access layer, dbExpress.

As a further solution, for simple applications you can use Delphi's ClientDataSet component, which has the ability to save tables to local files—something Borland touts with the name MyBase. Notice that the typical simple Delphi application based on Paradox tables does not port to Kylix, due to the lack of the BDE.

### The dbExpress Library

One of the most relevant new features of Delphi in the recent years has been the introduction of the dbExpress database library (DBX), available both for the Linux and the Windows platforms. I say *library* and not *database engine* because, unlike other solutions, dbExpress uses a lightweight approach and requires basically no configuration on end-user machines.

Being light and portable are the two key characteristics of dbExpress; Borland introduced it for those reasons, along with the development of the Kylix project. Compared to other powerhouses, dbExpress is very limited in its capabilities. It can access only SQL servers (no local files); it has no caching capabilities and provides only unidirectional access to the data; and it can natively work only with SQL queries and is unable to generate the corresponding SQL update statements.

At first, you might think these limitations make the library useless. On the contrary: These are *features* that make it interesting. Unidirectional datasets with no direct update are the norm if you need to produce reports, including generating HTML pages showing the content of a database. If you want to build a user interface to edit the data, however, Delphi includes specific components (the ClientDataSet and Provider, in particular) that provide caching and query resolution. These components allow your dbExpress-based application much more control than a separate (monolithic) database engine, which does extra things for you but often does them the way it wants to, not the way you would like.

dbExpress allows you to write an application that, aside from problems with different SQL dialects, can access many different database engines without much code modification. Supported SQL servers include Borland's own InterBase database, the Oracle database server, the MySQL database (which is popular particularly on Linux), the Informix database, IBM's DB2, and Microsoft SQL Server in Delphi 7. A more detailed description of dbExpress, the related VCL components, and many examples of its use will be provided in [Chapter 14](#); the current chapter focuses on database architecture foundations.

**Tip** The availability of a dbExpress driver for Microsoft SQL Server in Delphi 7 fills a significant gap. This database is frequently used on the Windows platform, and developers who needed a solution portable among different database servers often had to include support for Microsoft SQL Server. Now there is one less reason to stick with the BDE. Borland has released an update of the SQL Server dbExpress driver shipping with Delphi 7 to fix a couple of bugs.

### The Borland Database Engine

Delphi still ships with the BDE, which allows you to access local database formats (like Paradox and dBase) and SQL servers as well as anything accessible through ODBC drivers. This was the standard database technology in early

versions of Delphi, but Borland now considers it *obsolete*. This is particularly true for the use of the BDE to access SQL servers through the SQL Links drivers. Using the BDE to access local tables is still officially supported, simply because Borland doesn't provide a direct migration path for this type of application.

In some cases, a local table can be replaced with the ClientDataSet component (MyBase) specifically for temporary and small lookup tables. However, this approach won't work for larger local tables, because MyBase requires the entire table to be loaded in memory to access even a single record. The suggestion is to move larger tables to a SQL server installed on the client computer. InterBase, with its small footprint, is ideal in this particular situation. This type of migration will also open to you the doors of Linux, where the BDE is not available.

Of course, if you have existing applications that use the BDE, you can continue using them. The BDE page of Delphi's Component Palette still has the Table, Query, StoreProc, and other BDE-specific components. I'd discourage you from developing new programs with this old technology, which is almost discontinued by its producer. Eventually, you should look to third-party engines to replace the BDE when your programs require a similar architecture (or you need compatibility with older database file formats).

**Note** This is the reason I've removed any coverage of the BDE in the current edition of this book. This chapter used to be based on the Table and Query components; it has been rewritten to describe the architecture of Delphi database applications using the ClientDataSet component.

## InterBase Express

Borland has made available another set of database access components for Delphi: InterBase Express (IBX). These components are specifically tailored to Borland's own InterBase server. Unlike dbExpress, this is not a server-independent database engine, but a set of components for accessing a specific database server. If you plan to use only InterBase as your back-end database, using a specific set of components can give you more control over the server, provide the best performance, and allow you to configure and maintain the server from within a custom client application.

**Note** The use of InterBase Express highlights the case of database-specific custom datasets, which are available from third-party vendors for many servers. (There are other dataset components for InterBase, just as there are for Oracle, local or shared dBase files, and many others.)

You can consider using IBX (or other comparable sets of components) if you are sure you won't change your database and you want to achieve the best performance and control at the expense of flexibility and portability. The down side is that the extra performance and control you gain may be limited. You'll also have to learn to use another set of components with a specific behavior, rather than learn to use a generic engine and apply your knowledge to different situations.

## MyBase and the ClientDataSet Component

The ClientDataSet is a dataset accessing data kept in memory. The in-memory data can be temporary (created by the program and lost as you exit it), loaded from a local file and then saved back to it, or imported by another dataset using a Provider component.

Borland indicates that you should use the ClientDataSet component mapped to a file with the name MyBase, to indicate that it can be considered a local database solution. I have trouble with the way Borland marketing has promoted this technology, but it has a place, as I'll discuss in the section "[MyBase: Stand-alone ClientDataSet.](#)"

Accessing data from a provider is a common approach both for client/server architectures (as you'll see in [Chapter 14](#)) and for multitier architectures (discussed in [Chapter 16](#), "Multitier DataSnap Applications"). The ClientDataSet component becomes particularly useful if the data-access components you are using provide limited or no caching, which is the case with the dbExpress engine.

## dbGo for ADO



ADO (ActiveX Data Objects) is Microsoft's high-level interface for database access. ADO is implemented on Microsoft's data-access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. ADO is an engine with features comparable to those of the BDE: database server independence supporting local and SQL servers alike, a heavyweight engine, and a simplified configuration (because it is not centralized). Installation should (in theory) not be an issue, because the engine is part of recent versions of Windows. However, the limited compatibility among versions of ADO will force you to upgrade your users' computers to the same version you've used for developing the program. The sheer size of the MDAC (Microsoft Data Access Components) installation, which updates large portions of the operating system, makes this operation far from simple.

ADO offers definite advantages if you plan to use Access or SQL Server, because Microsoft's drivers for its own databases are of better quality than the average OLE DB providers. For Access databases, specifically, using Delphi's ADO components is a good solution. If you plan to use other SQL servers, first check the availability of a good-quality driver, or you might have some surprises. ADO is very powerful, but you have to learn to live with it— it stands between your program and the database, providing services but occasionally also issuing different commands than you may expect. On the negative side, do not even think of using ADO if you plan future cross-platform development; this Microsoft-specific technology is not available on Linux or other operating systems.

In short, use ADO if you plan to work only on Windows, want to use Access or other Microsoft databases, or find a good OLE DB provider for each of the database servers you plan to work with (at the moment, this factor excludes InterBase and many other SQL servers).

ADO components (part of a package Borland calls dbGo) are grouped in the ADO page of the Component Palette. The three core components are ADOConnection (for database connections), ADOCommand (for executing SQL commands), and ADODataset (for executing requests that return a result set). There are also three compatibility components—ADOTable, ADOQuery, and ADOStoredProc—that you can use to port BDE-based applications to ADO. Finally, the RDSConnection component lets you access data in remote multitier applications.

**Note** [Chapter 15](#), "Working with ADO," covers ADO and related technologies in detail. Notice that Microsoft is replacing ADO with its .NET version, which is based on the same core concepts. So, using ADO might provide you with a good path toward native .NET applications (although Borland plans to move dbExpress to that platform, too).

## Custom Dataset Components

As a further alternative, you can write your own custom dataset components, or choose one of the many offerings available. Developing custom dataset components is one of the most complex issues of Delphi programming; it's covered in depth in [Chapter 17](#), "Writing Database Components." Reading that material, you'll also learn about the internal workings of the `TDataSet` class.

## MyBase: Stand-alone ClientDataSet

If you want to write a single-user database application in Delphi, the simplest approach is to use the ClientDataSet component and map it to a local file. This local file mapping is different from the traditional data mapping to a local file. The traditional approach is to read from the file a record at a time, and possibly have a second file that stores indexes. The ClientDataSet maps an entire table (and possibly a master/detail structure) to the file in its entirety: When a program starts, the entire file is loaded in memory, and then everything is saved at once.

**Warning** This explains why you cannot use this approach in a multiuser or multiapplication situation. If two programs or two instances of the same program load the same ClientDataSet file in memory and modify the data, the last table saved will overwrite changes made by other programs.

This support for persisting the content of a ClientDataSet was created a few years ago as a way to implement the so-called briefcase model. A user could (and still can) download data from its database server to the client, save some of the data, work disconnected (while traveling with a laptop computer, for example), and finally reconnect to commit the changes.

### Connecting to an Existing Local Table

To map a ClientDataSet to a local file, you set its `FileName` property. To build a minimal program (called MyBase1 in the example), all you need is a ClientDataSet component hooked to a CDS file (there are a few in the `Data` folder available under `\Program Files\Common Files\Borland Shared`), a DataSource (more on this later), and a DBGrid control. Hook the ClientDataSet to the DataSource via the DataSource's `DataSet` property and the DataSource to the DBGrid via the grid's `DataSource` property, as in [Listing 13.1](#). At this point turn on the `Active` property of the ClientDataSet and you'll have a program showing database data even at design time; see [Figure 13.1](#).

#### Listing 13.1: The DFM File of the MyBase1 Sample Program

---

```
object Form1: TForm1
  ActiveControl = DBGrid1
  Caption = 'MyBase1'
  OnCreate = FormCreate
  object DBGrid1: TDBGrid
    DataSource = DataSource1
  end
  object DataSource1: TDataSource
    DataSet = cds
  end
  object cds: TClientDataSet
    FileName = 'C:\Program Files\Common Files\Borland
      Shared\Data\customer.cds'
  end
end
```

---

CustNo	Company	Addr1	Addr2
	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
	Unisco	PO Box 2-547	
	DataSpaceLight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
9984	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	232 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Frey.	
1563	Blue Sports	203 12th Ave. Box 746	
1624	Makai SCUBA Club	PO Box 8534	
1645	Action Club	PO Box 5451-F	
1651	Jamaica SCUBA Centre	PO Box 68	
1680	Island Finders	6133 1/3 Stone Avenue	
1984	Adventure Undersea	PO Box 744	
2118	Blue Sports Club	63365 Nez Perce Street	
2135	Frank's Divers Supply	1455 North 44th St.	

**Figure 13.1:** A sample local table active at design time in the Delphi IDE

As you make changes and close the application, the data will be automatically saved to the file. (You might want to disable the change log, as discussed later, to reduce the size of this data.) The dataset also has a `SaveToFile` method and a `LoadFromFile` method you can use in your code.

I also made another change: I disabled the `ClientDataSet` at design time to avoid including all of its data in the program's DFM file and in the compiled executable file; I want to keep the data in a separate file. To do this, close the dataset at design time, after testing, and add a line to the form's `OnCreate` event to open it:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  cds.Open;
end;
```

## From the Midas DLL to the MidasLib Unit

To run any application using the `ClientDataSet` component, you need to also deploy the `midas.dll` dynamic library referenced by the `DSIntf.pas` unit. The `ClientDataSet` component's core code is not directly part of the VCL and is not available in source code format. This is unfortunate, because many developers are accustomed to debugging the VCL source code and using it as the ultimate reference.

**Warning** The `midas.dll` library has no version number in its name. So, if a computer has an older version, your program will apparently run on it but may not behave properly.

The Midas library is a C-language library, but since Delphi 6 it can be bound directly into an executable by including the specific `MidasLib` unit (a special DCU produced by a C compiler). In this case you won't need to distribute the library in the DLL format.

## XML and CDS Formats

The `ClientDataSet` component supports two different streaming formats: the native format and an XML-based format. The `Borland\Shared\Demo` folder mentioned earlier holds versions of a number of tables in each of the two formats. By default, MyBase saves the datasets in XML format. The `SaveToFile` method has a parameter allowing you to specify the format, and the `LoadFromFile` method works automatically for both formats.

Using the XML format has the advantage of making the persistent data also accessible with an editor and with other programs not based on the `ClientDataSet` component. However, this approach implies converting the data back and forth, because the CDS format is close to the internal memory representation invariably used by the component,

regardless of the streaming format. Also, the XML format generates large files, because they are text based. On average, a MyBase XML file is twice the size of the corresponding CDS file.

**Tip** While you have a ClientDataSet in memory, you can extract its XML representation by using the *XMLData* property without streaming out the data. The next example puts this technique into practice.

## Defining a New Local Table

Besides letting you hook to an existing database table stored in a local file, the ClientDataSet component allows you to create new tables easily. All you have to do is use its *FieldDefs* property to define the structure of the table. After doing this, you can physically create the file for the table with the Create DataSet command on the ClientDataSet component's shortcut menu in the Delphi IDE or by calling its *CreateDataSet* method at run time.

This is an extract from the MyBase2 example's DFM file, which defines a new local database table:

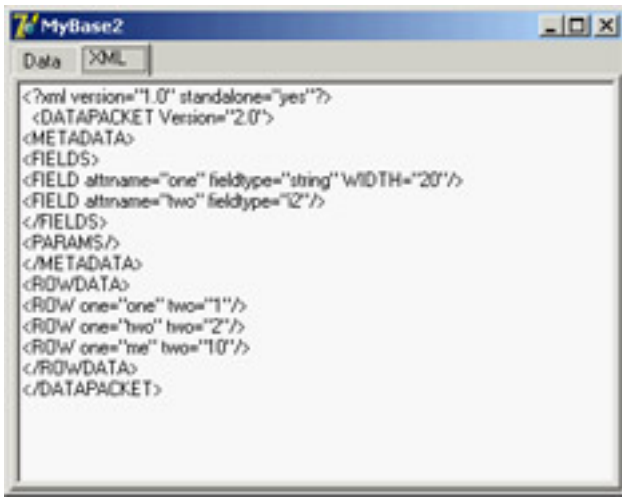
```
object ClientDataSet1: TClientDataSet
  FileName = 'mybase2.cds'
  FieldDefs = <
    item
      Name = 'one'
      DataType = ftString
      Size = 20
    end
    item
      Name = 'two'
      DataType = ftSmallint
    end>
  StoreDefs = True
end
```

Notice the *StoreDefs* property, which is automatically set to True when you edit the collection of field definitions. By default, a dataset in Delphi loads its metadata before opening. Only if a local definition is stored in the DFM file is this local metadata used (saving field definitions in the DFM file is also helpful to cache this metadata in a client/server architecture).

To account for the optional dataset creation, the disabling of the log (described later), and the display of the XML version of the initial data in a Memo control, the program's form class has the following *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not FileExists (cds.FileName) then
    cds.CreateDataSet;
  cds.Open;
  cds.MergeChangeLog;
  cds.LogChanges := False;
  Mem1.Lines.Text := StringReplace (
    Cds.XMLData, '>', '>' + sLineBreak, [rfReplaceAll]);
end;
```

The last statement includes a call to *StringReplace* to provide a poor man's XML formatting: The code adds a new line at the end of each XML tag by adding a new line after the close angle bracket. You can see the table's XML display with a few records in [Figure 13.2](#). You'll learn a lot more about XML in Delphi in [Chapter 22](#), "Using XML Technologies."



**Figure 13.2:** The XML display of a CDS file in the MyBase2 example. The table structure is defined by the program, which creates a file for the dataset on its first execution.

## Indexing

Once you have a `ClientDataSet` in memory, you can perform many operations on it. The simplest are indexing, filtering, and searching for records; more complex operations include grouping, defining aggregate values, and managing the change log. Here I'll cover only the simplest techniques; more complex material appears at the end of the chapter.

Filtering a `ClientDataSet` is a matter of setting the `IndexFieldNames` property. This is often accomplished when the user clicks the field title in a `DBGrid` component (firing the `OnTitleClick` event), as in the MyBase2 example:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);  
begin  
    cds.IndexFieldNames := Column.Field.FieldName;  
end;
```

Unlike other local databases, a `ClientDataSet` can have this type of dynamic index without any database configuration because indexes are computed in memory.

**Tip** The component also supports indexes based on a calculated field, specifically an *internally calculated* field, available only for this dataset (as I'll describe later in this chapter). Unlike ordinary calculated fields, which are computed every time the record is used, values of internally calculated fields are calculated once and kept in memory. For this reason, indexes consider them plain fields.

In addition to assigning a new value to the `IndexFieldNames` property, you can define an index using the `IndexDefs` property. Doing so allows you to define several indexes and keep them in memory, switching even faster from one to the other.

**Tip** Defining a separate index is the only way to have a descending index, rather than an ascending index.

## Filtering

As with any other dataset, you can use the `Filter` property to specify the inclusion in the dataset of portions of the data the component is bound to. The filtering operation takes place in memory after loading all the records, so this is a way to present less data to the user, not to limit the memory footprint of a large local dataset.

When you're retrieving large amounts of data from a server (in a client/server architecture) you should try to use a proper query so you don't retrieve a large dataset from a SQL server. Filtering up front in the server should generally be your first choice. With local data, you might consider splitting a large number of records into a set of different files, so you can load only those you need and not all of them.

Local filtering in the `ClientDataSet` can be useful, particularly because the filter expressions you can use with this

component are much more extensive than those you can use with other datasets. In particular, you can use the following:

- The standard comparison and logical operators: for example, `Population > 1000` and `Area < 1000`
- Arithmetic operators: for example, `Population / Area < 10`
- String functions: for example, `Substring(Last_Name, 1, 2) = 'Ca'`
- Date and time functions: for example, `Year (Invoice_Date) = 2002`
- Others, including a `Like` function, wildcards, and an `In` operator

These filtering capabilities are fully documented in the VCL Help file. You should look for the page "Limiting what records appear" linked from the description of the `Filter` property of the `TClientDataSet` class, or reach it from the Help Contents page following this path down the tree: Developing Database Applications, Using client datasets, Limiting what records appear.

## Locating Records

Filtering allows you to limit the records displayed to the program's user, but many times you want to display all the records and only move to a specific one. The `Locate` method does this. If you've never used `Locate`, at first sight the Help file won't be terribly clear. The idea is that you must provide a list of fields you want to search and a list of values, one for each field. If you want to match only one field, the value is passed directly, as in this case (where the search string is in the `EditName` component):

```
procedure TForm1.btnLocateClick(Sender: TObject);
begin
    if not cds.Locate ('LastName', EditName.Text, []) then
        MessageDlg ('"' + EditName.Text + '" not found', mtError, [mbOk], 0);
end;
```

If you search for multiple fields, you have to pass a variant array with the list of values you want to match. The variant array can be created from a constant array with the `VarArrayOf` function or from scratch using the `VarArrayCreate` call. This is a code snippet:

```
cds.Locate ('LastName;FirstName', VarArrayOf (['Cook', 'Kevin']), [])
```

Finally, you can use the same method to look for a record even if you know only the initial portion of the field you are looking for. All you have to do is to add the `loPartialKey` flag to the `Options` parameter (the third) of the `Locate` call.

**Note** Using `Locate` makes sense when you're working with a local table, but it doesn't port well to client/server applications. On a SQL server, similar client-side techniques imply moving all the data to the client application first (which is generally a bad idea) and then searching for a specific record. You should locate the data with restricted SQL statements. You can still call `Locate` after you retrieve a limited dataset. For example, you can search for a customer by name after you select all the customers of a given town or area, obtaining a result set of a limited size. There's more about this topic in [Chapter 14](#), which is devoted to client/server development.

## Undo and SavePoint

As a user modifies the data in a `ClientDataSet` component, the updates are stored in a memory area called `Delta`. The reason for keeping track of user changes instead of holding the resulting table is due to the way the updates are handled in a client/server architecture. In this case, the program doesn't have to send the entire table back to the server, but only a list of the user's changes (by means of specific SQL statements, as you'll see in [Chapter 14](#)).

Because the `ClientDataSet` component keeps track of changes, you can reject those changes, removing entries from the delta. The component has an `UndoLastChange` method to accomplish this. The method's `FollowChange` parameter allows you to *follow* the undo operation—the client dataset will move to the record that has been restored by the undo operation. Here is the code you can use to connect to an Undo button:

```
procedure TForm1.ButtonUndoClick(Sender: TObject);  
begin  
    cds.UndoLastChange (True);  
end;
```

An extension of the undo support is the possibility of saving a sort of bookmark of the change log position (the current status) and to restore it later by undoing all successive changes. You can use the `SavePoint` property either to save the number of changes in the log or to reset the log to a past situation. However, you can only remove records from the change log, not reinsert changes. In other words, the `SavePoint` property refers to a position in a log, so it can only go back to a position where there were fewer records! This log position is a number of changes, so if you save the current position, undo some changes, and then do more edits, you won't be able to get back to the position you bookmarked.

**Tip** Delphi 7 has a new standard action mapped to the `ClientDataSet`'s Undo operation. Other new actions include Revert and Apply, which you'll need when the component is connected to a dataset accessing a database.

## Enabling and Disabling Logging

Keeping track of changes makes sense if you need to send the updated data back to a server database. In local applications with data stored to a MyBase file, keeping this log around can become useless and consumes memory. For this reason, you can disable logging with the `LogChanges` property. This will also stop the undo operations, though.

You can also call the `MergeChangesLog` method to remove all current editing from the change log, and confirm the edits performed so far. Doing so makes sense if you want to keep the undo log around within a single session and then save the final dataset without the keeping the change log.

**Note** The MyBase2 example disables the change log as discussed here: You can remove that code and re-enable it to see the difference in the size of the CDS file and in the XML text after editing the data.



## Using Data-Aware Controls

Once you set up the proper data-access components, you can build a user interface to let a user view the data and eventually edit it. Delphi provides many components that resemble the usual controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all these components in the Data Controls page of the Delphi Component Palette.

All these components are connected to a data source using the corresponding property, `DataSource`. Some of them relate to the entire dataset, such as the DBGrid and DBNavigator components, and the others refer to a specific field of the data source, as indicated by the `DataField` property. Once you select the `DataSource` property, the `DataField` property editor will contain a list of available values.

Notice that all the data-aware components are unrelated to the data-access technology, provided the data-access component inherits from `TDataSet`. Thus your investment in the user interface is preserved when you change the data-access technology. However, some of the lookup components and extended use of the DBGrid (displaying a lot of data) make sense only when you're working with local data and should generally be avoided in a client/server situation, as you'll see in [Chapter 14](#).

### Data in a Grid

The DBGrid is a grid capable of displaying a whole table at once. It allows scrolling and navigation, and you can edit the grid's contents. It is an extension of the other Delphi grid controls.

You can customize the DBGrid by setting its `Options` property's various flags and modifying its `Columns` collection. The grid allows a user to navigate the data using the scrollbars and perform all the major actions. A user can edit the data directly, insert a new record in a given position by pressing the Insert key, append a new record at the end by going to the last record and pressing the Down arrow key, and delete the current record by pressing Ctrl+Del.

The `Columns` property is a collection from which you can choose the table fields you want to see in the grid and set column and title properties (color, font, width, alignment, caption, and so on) for each field. Some of the more advanced properties, such as `ButtonStyle` and `DropDownRows`, can be used to provide custom editors for a grid's cells or a drop-down list of values (indicated in the column's `PickList` property).

### DBNavigator and Dataset Actions

DBNavigator is a collection of buttons used to navigate and perform actions on the database. You can disable some of the DBNavigator control's buttons by removing some of the elements of the `VisibleButtons` set property.

The buttons perform basic actions on the connected dataset, so you can easily replace them with your own toolbar, particularly if you use an `ActionList` component with the predefined database actions provided by Delphi. In this case, you get all the standard behaviors, but you'll also see the various buttons enabled only when their action is legitimate. The advantages of using the actions is that you can display the buttons in the layout you prefer, intermix them with other buttons of the application, and use multiple client controls, including main and popup menus.

**Tip** If you use the standard actions, you can avoid connecting them to a specific `DataSource` component, and the actions will be applied to the dataset connected to the visual control that currently has the input focus. This way, a single toolbar can be used for multiple datasets displayed by a form, which can be very confusing to the user if not considered carefully.

### Text-Based Data-Aware Controls

There are multiple text-oriented components:

**DBText** Displays the contents of a field that cannot be modified by the user. It is a data-aware Label



graphical control. It can be very useful, but users might confuse this control with the plain labels that indicate the content of each field-based control.

**DBEdit** Lets the user edit a field (change the current value) using an Edit control. At times, you might want to disable editing and use a DBEdit as if it were a DBText, but highlight the fact that this is data coming from the database.

**DBMemo** Lets the user see and modify a large text field, eventually stored in a memo or BLOB (binary large object) field. It resembles the Memo component and has full editing capabilities, but all the text is rendered in a single font.

## List-Based Data-Aware Controls

To let a user choose a value in a predefined list (which reduces input errors), you can use many different components. DBListBox, DBComboBox, and DBRadioGroup are similar, providing a list of strings in the `Items` property, but they do have some differences:

**DBListBox** Allows selection of predefined items (*closed selection*), but not text input, and can be used to list many elements. Generally it's best to show only about six or seven items to avoid using up too much space on the screen.

**DBComboBox** Can be used both for closed selection and for user input. The `csDropDown` style of the DBComboBox allows a user to enter a new value, in addition to selecting one of the available values. The component also uses a smaller area on the form because the drop-down list is usually displayed only on request.

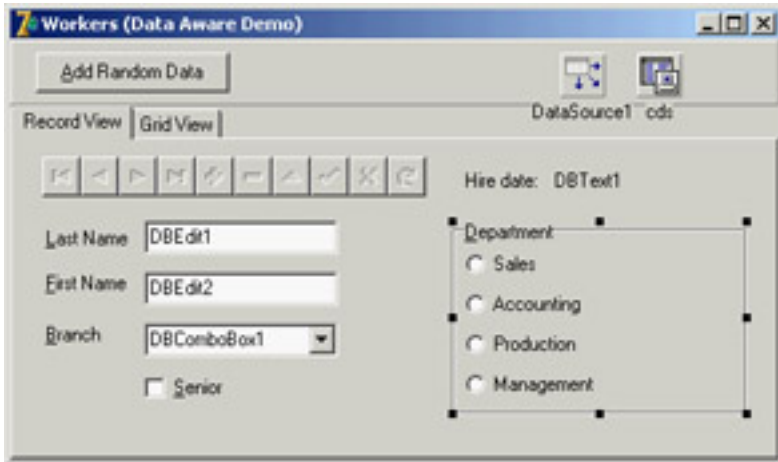
**DBRadioGroup** Presents radio buttons (which permit only one selection), allows only closed selection, and should be used only for a limited number of alternatives. A nice feature of this component is that the values displayed can be those you want to insert in the database, but you can also choose to provide mapping. The values of the user interface (descriptive strings stored in the `Items` property) will map to corresponding values stored in the database (numeric or character-based codes listed in the `Values` property). For example, you can map numeric codes indicating departments to a few descriptive strings:

```
object DBRadioGroup1: TDBRadioGroup
  Caption = 'Department'
  DataField = 'Department'
  DataSource = DataSource1
  Items.Strings = (
    'Sales'
    'Accounting'
    'Production'
    'Management')
  Values.Strings = (
    '1'
    '2'
    '3'
    '4')
end
```

The DBCheckBox component is slightly different; it is used to show and toggle an option, corresponding to a Boolean field. It is a limited list because it has only two possible values plus the undetermined state for fields with null values. You can determine which are the values to send back to the database by setting the `ValueChecked` and `ValueUnchecked` properties of this component.

## The DbAware Example

The DbAware example highlights the usage of a DBRadioGroup control with the settings discussed in the [previous section](#) and a DBCheckBox control. This example is not much more complex than earlier ones, but it has a form with field-oriented data-aware controls, instead of a grid encompassing them all. You can see the example's form at design time in [Figure 13.3](#).



**Figure 13.3:** The data-aware controls of the DbAware example at design time in Delphi

As in the MyBase2 program, the application defines its own table structure, using the `FieldDefs` collection property of the `ClientDataSet`. [Table 13.1](#) provides a short summary of the fields defined.

**Table 13.1: The Dataset Fields in the DbAware Example**

Name	Data Type	Size
LastName	ftString	20
FirstName	ftString	20
Department	FtSmallint	
Branch	ftString	20
Senior	ftBoolean	
HireDate	ftDate	

The program has some code to fill in the table with random values. This code is tedious and not too complex, so I won't discuss the details here, but you can look at the DbAware source code if you are interested.

## Using Lookup Controls

If the list of values is extracted from another dataset, then instead of the `DBListBox` and `DBComboBox` controls you should use the specific `DBLookupListBox` or `DBLookupComboBox` component. These components are used every time you want to select for a field a value that corresponds to a record of another dataset (and not to choose a different record to display!).

For example, if you build a standard form for taking orders, the orders dataset will generally have a field hosting a number indicating the customer who made the order. Working directly with the customer number is not the most natural way; most users will prefer to work with customer names. However, in the database, the customers' names are stored in a different table, to avoid duplicating the customer data for each order by the same customer. To get around such a situation, with local databases or small lookup tables, you can use a `DBLookupComboBox` control. (This technique doesn't port well to client/server architecture with large lookup tables, as discussed in the [next chapter](#).)

The `DBLookupComboBox` component can be connected to two data sources at the same time: one source containing

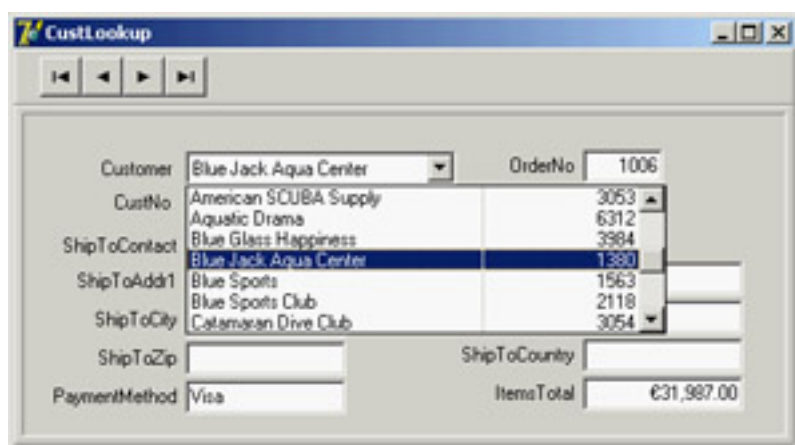
the data and a second containing the display data. I built a standard form using the `orders.cds` file from the Delphi sample data folder; the form includes several DBEdit controls.

You should remove the standard DBEdit component connected to the customer number and replace it with a DBLookupComboBox component (and a DBText component, to fully understand what is going on). The lookup component (and the DBText) is connected to the DataSource for the order and to the CustNo field. To let the lookup component show the information extracted from another file (`customer.cds`) you need to add another ClientDataSet component referring to the file, along with a new data source.

For the program to work, you need to set several properties of the DBLookupComboBox1 component. Here is a list of the relevant values:

```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSourceOrders
  KeyField = 'CustNo'
  ListField = 'Company;CustNo'
  ListSource = DataSourceCustomer
  DropDownWidth = 300
end
```

The first two properties determine the main connection, as usual. The next four properties determine the field used for the join (`KeyField`), the information to display (`ListField`), and the secondary source (`ListSource`). In addition to entering the name of a single field, you can provide multiple fields, as I did in the example. Only the first field is displayed as combo box text, but if you set a large value for the `DropDownWidth` property, the combo box's drop-down list will include multiple columns of data. You can see this output in [Figure 13.4](#).



**Figure 13.4:** The output of the CustLookup example, with the DBLookupCombo-Box showing multiple fields in its drop-down list

**Tip** If you set the `IndexFieldNames` property of the ClientDataSet containing the orders data to the `Company` field, the drop-down list will show the companies in alphabetical order instead of customer-number order. I did this in the example.

## Graphical Data-Aware Controls

Delphi includes a graphical data-aware control: DBImage. It is an extension of an Image component that shows a picture stored in a BLOB field, provided the database uses a graphic format that the Image component supports, such as BMP or JPEG (if you add the JPEG unit to your uses clause).

Once you have a table that includes a BLOB storing an image with a compatible graphic format, hooking it to the component is trivial. If, instead, the graphic format requires a custom transformation in order to be displayed, it might be easier to use a standard non-data-aware Image component and write code so the image is updated each time the

current record changes. Before I can discuss this subject, however, you need to know more about the `TDataSet` class and the dataset field classes.

**Team LiB**

[◀ PREVIOUS](#)

[NEXT ▶](#)

## The DataSet Component

Instead of proceeding with the discussion of the capabilities of a specific dataset at this point, I prefer to devote some space to a generic introduction of the features of the `TDataSet` class, which are shared by all inherited data-access classes. The `DataSet` component is very complex, so I won't list all its capabilities—I will only discuss its core elements.

This component provides access to a series of records that are read from some source of data, kept in internal buffers (for performance reasons), and eventually modified by a user, with the possibility of writing back changes to the persistent storage. This approach is generic enough to be applied to different types of data (even non-database data), but it has a few rules:

- There can be only one active record at a time, so if you need to access data in multiple records, you must move to each of them, read the data, then move again, and so on. You'll find an example of this and related techniques in the section "[Navigating a Dataset.](#)"
- You can edit only the active record: You cannot modify a set of records at the same time, as you can in a relational database.
- You can modify data in the active buffer only after you explicitly declare you want to do so, by giving the `Edit` command to the dataset. You can also use the `Insert` command to create a new blank record and close both operations (insert or edit) by giving a `Post` command.

Other interesting elements of a dataset that I'll explore in the following sections are its status (and the status change events), navigation and record positions, and the role of field objects. As a summary of the capabilities of the `DataSet` component, I included the public methods of its class in [Listing 13.2](#) (the code has been edited and commented for clarity). Not all of these methods are directly used everyday, but I kept them all in the listing.

### Listing 13.2: The Public Interface of the `TDataSet` Class (Excerpted)

```
TDataSet = class(TComponent, IProviderSupport)
...
public
    // create and destroy, open and close
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Open;
    procedure Close;
    property BeforeOpen: TDataSetNotifyEvent read FBeforeOpen write FBeforeOpen;
    property AfterOpen: TDataSetNotifyEvent read FAfterOpen write FAfterOpen;
    property BeforeClose: TDataSetNotifyEvent
        read FBeforeClose write FBeforeClose;
    property AfterClose: TDataSetNotifyEvent read FAfterClose write FAfterClose;

    // status information
    function IsEmpty: Boolean;
    property Active: Boolean read GetActive write SetActive default False;
    property State: TDataSetState read FState;
    function ActiveBuffer: PChar;
    property IsUniDirectional: Boolean
        read FIsUniDirectional write FIsUniDirectional default False;
    function UpdateStatus: TUpdateStatus; virtual;
    property RecordSize: Word read GetRecordSize;
    property ObjectView: Boolean read FObjectView write SetObjectView;
    property RecordCount: Integer read GetRecordCount;
    function IsSequenced: Boolean; virtual;
    function IsLinkedTo(DataSource: TDataSource): Boolean;
```

```

// datasource
property DataSource: TDataSource read GetDataSource;
procedure DisableControls;
procedure EnableControls;
function ControlsDisabled: Boolean;

// fields, including blobs, details, calculated, and more
function FieldByName(const FieldName: string): TField;
function FindField(const FieldName: string): TField;
procedure GetFieldList(List: TList; const FieldNames: string);
procedure GetFieldNames(List: TStrings); virtual; // virtual since Delphi 7
property FieldCount: Integer read GetFieldCount;
property FieldDefs: TFieldDefs read FFieldDefs write SetFieldDefs;
property FieldDefList: TFieldDefList read FFieldDefList;
property Fields: TFields read FFields;
property FieldList: TFieldList read FFieldList;
property FieldValues[const FieldName: string]: Variant
    read GetFieldValue write SetFieldValue; default;
property AggFields: TFields read FAggFields;
property DataSetField: TDataSetField
    read FDataSetField write SetDataSetField;
property DefaultFields: Boolean read FDefaultFields;
procedure ClearFields;
function GetBlobFieldData(FieldNo: Integer;
    var Buffer: TBlobByteData): Integer; virtual;
function CreateBlobStream(Field: TField;
    Mode: TBlobStreamMode): TStream; virtual;
function GetFieldData(Field: TField;
    Buffer: Pointer): Boolean; overload; virtual;
procedure GetDetailDataSets(List: TList); virtual;
procedure GetDetailLinkFields(MasterFields, DetailFields: TList); virtual;
function GetFieldData(FieldNo: Integer;
    Buffer: Pointer): Boolean; overload; virtual;
function GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean):
    Boolean; overload; virtual;
property AutoCalcFields: Boolean
    read FAutoCalcFields write FAutoCalcFields default True;
property OnCalcFields: TDataSetNotifyEvent
    read FOnCalcFields write FOnCalcFields;

// position, movement
procedure CheckBrowseMode;
procedure First;
procedure Last;
procedure Next;
procedure Prior;
function MoveBy(Distance: Integer): Integer;
property RecNo: Integer read GetRecNo write SetRecNo;
property Bof: Boolean read FBOF;
property Eof: Boolean read FEOF;
procedure CursorPosChanged;
property BeforeScroll: TDataSetNotifyEvent
    read FBeforeScroll write FBeforeScroll;
property AfterScroll: TDataSetNotifyEvent
    read FAfterScroll write FAfterScroll;

```

```

// bookmarks
procedure FreeBookmark(Bookmark: TBookmark); virtual;
function GetBookmark: TBookmark; virtual;
function BookmarkValid(Bookmark: TBookmark): Boolean; virtual;
procedure GotoBookmark(Bookmark: TBookmark);
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; virtual;
property Bookmark: TBookmarkStr read GetBookmarkStr write SetBookmarkStr;

// find, locate
function FindFirst: Boolean;
function FindLast: Boolean;
function FindNext: Boolean;
function FindPrior: Boolean;
property Found: Boolean read GetFound;
function Locate(const KeyFields: string; const KeyValues: Variant;
    Options: TLocateOptions): Boolean; virtual;
function Lookup(const KeyFields: string; const KeyValues: Variant;
    const ResultFields: string): Variant; virtual;

// filtering
property Filter: string read FFilterText write SetFilterText;
property Filtered: Boolean read FFiltered write SetFiltered default False;
property FilterOptions: TFilterOptions
    read FFilterOptions write SetFilterOptions default [];
property OnFilterRecord: TFilterRecordEvent
    read FOnFilterRecord write SetOnFilterRecord;

// refreshing, updating
procedure Refresh;
property BeforeRefresh: TDataSetNotifyEvent
    read FBeforeRefresh write FBeforeRefresh;
property AfterRefresh: TDataSetNotifyEvent
    read FAfterRefresh write FAfterRefresh;
procedure UpdateCursorPos;
procedure UpdateRecord;
function GetCurrentRecord(Buffer: PChar): Boolean; virtual;
procedure Resync(Mode: TResyncMode); virtual;

// editing, inserting, posting, and deleting
property CanModify: Boolean read GetCanModify;
property Modified: Boolean read FModified;
procedure Append;
procedure Edit;
procedure Insert;
procedure Cancel; virtual;
procedure Delete;
procedure Post; virtual;
    procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
procedure SetFields(const Values: array of const);

// events related to editing, inserting, posting, and deleting
property BeforeInsert: TDataSetNotifyEvent
    read FBeforeInsert write FBeforeInsert;
property AfterInsert: TDataSetNotifyEvent
    read FAfterInsert write FAfterInsert;

```

```

property BeforeEdit: TDataSetNotifyEvent read FBeforeEdit write FBeforeEdit;
property AfterEdit: TDataSetNotifyEvent read FAfterEdit write FAfterEdit;
property BeforePost: TDataSetNotifyEvent read FBeforePost write FBeforePost;
property AfterPost: TDataSetNotifyEvent read FAfterPost write FAfterPost;
property BeforeCancel: TDataSetNotifyEvent
    read FBeforeCancel write FBeforeCancel;
property AfterCancel: TDataSetNotifyEvent
    read FAfterCancel write FAfterCancel;
property BeforeDelete: TDataSetNotifyEvent
    read FBeforeDelete write FBeforeDelete;
property AfterDelete: TDataSetNotifyEvent
    read FAfterDelete write FAfterDelete;
property OnDeleteError: TDataSetErrorEvent
    read FOnDeleteError write FOnDeleteError;
property OnEditError: TDataSetErrorEvent
    read FOnEditError write FOnEditError;
property OnNewRecord: TDataSetNotifyEvent
    read FOnNewRecord write FOnNewRecord;
property OnPostError: TDataSetErrorEvent
    read FOnPostError write FOnPostError;

    // support utilities
function Translate(Src, Dest: PChar;
    ToOem: Boolean): Integer; virtual;
property Designer: TDataSetDesigner read FDesigner;
property BlockReadSize: Integer read FBlockReadSize write SetBlockReadSize;
property SparseArrays: Boolean read FSparseArrays write SetSparseArrays;
end;

```

---

## The Status of a Dataset

When you operate on a dataset in Delphi, you can work in different states. These states are indicated by a specific `State` property, which can assume several different values:

**dsBrowse** Indicates that the dataset is in normal browse mode; used to look at the data and scan the records.

**dsEdit** Indicates that the dataset is in edit mode. A dataset enters this state when the program calls the `Edit` method or the `DataSource` has the `AutoEdit` property set to `True`, and the user begins editing a data-aware control, such as a `DBGrid` or `DBEdit`. When the changed record is posted, the dataset exits the `dsEdit` state.

**dsInsert** Indicates that a new record is being added to the dataset. This might happen when calling the `Insert` or `Append` methods, moving to the last line of a `DBGrid`, or using the corresponding command of the `DBNavigator` component.

**dsInactive** Indicates a closed dataset.

**dsCalcFields** Indicates that a field calculation is taking place (during a call to an `OnCalcFields` event handler).

**dsNewValue, dsOldValue, and dsCurValue** Indicate that an update of the cache is in progress.

**dsFilter** Indicates that a dataset is setting a filter (during a call to an `OnFilterRecord` event handler).



In simple examples, the transitions between these states are handled automatically, but it is important to understand them because many events refer to the state transitions. For example, every dataset fires events before and after any state change. When a program requests an `Edit` operation, the component fires the `BeforeEdit` event just before entering edit mode (an operation you can stop by raising an exception). Immediately after entering edit mode, the dataset receives the `AfterEdit` event. After the user has finished editing and requests to store the data by executing the `Post` command, the dataset fires a `BeforePost` event (which can be used to check the input before sending the data to the database); it fires an `AfterPost` event after the operation has been successfully completed.

Another more general state-change tracking technique involves handling the `DataSource` component's `OnStateChange` event. As an example, you can show the current status with code like this:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);  
var  
    strStatus: string;  
begin  
    case cds.State of  
        dsBrowse: strStatus := 'Browse';  
        dsEdit: strStatus := 'Edit';  
        dsInsert: strStatus := 'Insert';  
    else  
        strStatus := 'Other state';  
    end;  
    StatusBar.Panels[0].Text := strStatus;  
end;
```

The code considers only the three most common states of a dataset component, ignoring the inactive state and other special cases.

## The Fields of a Dataset

I mentioned earlier that a dataset has only one record that is current, or active. The record is stored in a buffer, and you can operate on it with some generic methods, but to access the data of the record you need to use the dataset's field objects. This explains why field components (technically, instances of a class derived from the `TField` class) play a fundamental role in every Delphi database application. Data-aware controls are directly connected to these field objects, which correspond to database fields.

By default, Delphi automatically creates the `TField` components at run time, each time the program opens a dataset component. This is done after reading the metadata associated with the table or the query the dataset refers to. These field components are stored in the dataset's `Fields` array property. You can access these values by number (accessing the array directly) or by name (using the `FieldByName` method). Each field can be used to read or modify the current record's data by using its `Value` property or type-specific properties such as `AsDate`, `AsString`, `AsInteger`, and so on:

```
var
  strName: string;
begin
  strName := Cds.Fields[0].AsString
  strName := Cds.FieldByName('LastName').AsString
```

`Value` is a variant type property, so using the type-specific access properties is a little more efficient. The dataset component has also a shortcut property for accessing the variant-type value of a field: the default `FieldValues` property. A *default property* means you can omit it from the code by applying the square brackets directly to the dataset:

```
strName := Cds.FieldValues ['LastName'];
strName := Cds ['LastName'];
```

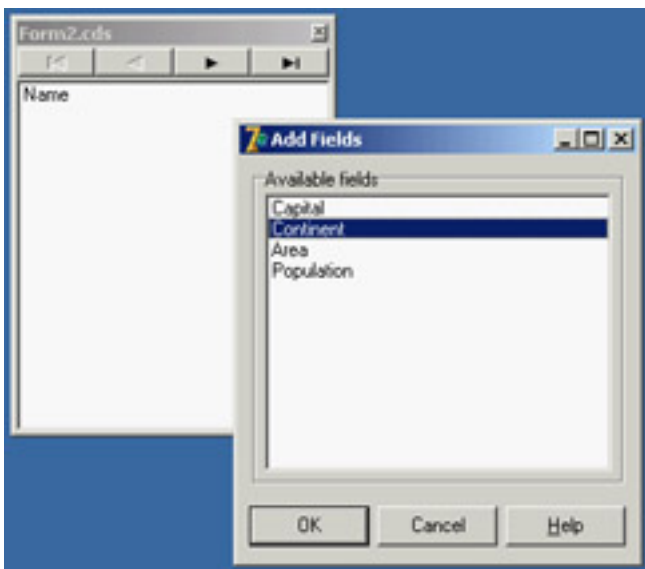
Creating the field components each time a dataset is opened is only a default behavior. As an alternative, you can create the field components at design time, using the Fields Editor (double-click a dataset to see the Fields Editor in action, or activate the dataset's shortcut menu or that of the Object TreeView and choose the Fields Editor command). After creating a field for the `LastName` column of a table, for example, you can refer to its value by applying one of the `AsXxx` methods to the proper field object:

```
strName := CdsLastName.AsString;
```

In addition to being used to access the value of a field, each field object has properties for controlling visualization and editing of its value, including range of values, edit masks, display format, constraints, and many others. These properties, of course, depend on the type of the field—that is, on the specific class of the field object. If you create persistent fields, you can set some properties at design time instead of writing code at run time (perhaps in the dataset's `AfterOpen` event).

**Note** Although the Fields Editor is similar to the editors of the collections used by Delphi, fields are not part of a collection. They are components created at design time, listed in the published section of the form class and available in the drop-down combo box at the top of the Object Inspector.

As you open the Fields Editor for a dataset, it appears empty. You have to activate the shortcut menu of this editor or of the Fields pseudonode in the Object TreeView to access its capabilities. The simplest operation you can do is to select the Add command, which allows you to add any other fields in the dataset to the list of fields. [Figure 13.5](#) shows the Add Fields dialog box, which lists all the fields available in a table. These are the database table fields that are not already present in the list of fields in the editor.



**Figure 13.5:** The Fields Editor with the Add Fields dialog box

The Fields Editor's Define command lets you define a new calculated field, lookup field, or field with a modified type. In this dialog box, you can enter a descriptive field name, which might include blank spaces. Delphi generates an internal name—the name of the field component—which you can further customize. Next, select a data type for the field. If it is a calculated field or a lookup field, and not just a copy of a field redefined to use a new data type, select the proper radio button. You'll see how to define a calculated field and a lookup field in the sections, "[Adding a Calculated Field](#)" and "[Lookup Fields](#)."

**Note** A *TField* component has both a *Name* property and a *FieldName* property. The *Name* property is the usual component name. The *FieldName* property is either the name of the column in the database table or the name you define for the calculated field. It can be more descriptive than the *Name*, and it allows blank spaces. The *FieldName* property of the *TField* component is copied to the *DisplayLabel* property by default. You can change the field name to any suitable text. It is used, among other things, to search a field in the *TDataSet* class's *FieldByName* method and when using the array notation.

All the fields you add or define are included in the Fields Editor and can be used by data-aware controls or displayed in a database grid. If a field of the physical dataset is not in this list, it won't be accessible. When you use the Fields Editor, Delphi adds the declaration of the available fields to the form's class as new components (much as the Menu Designer adds *TMenuItem* components to the form). The components of the *TField* class (more specifically, its subclasses) are fields of the form, and you can refer to these components directly in your program code to change their properties at run time or to get or set their value.

In the Fields Editor, you can also drag the fields to change their order. Proper field ordering is particularly important when you define a grid, which arranges its columns using this order.

**Tip** You can also drag the fields from the editor to the form to let the IDE create visual components for you. This is a handy feature that can save you a lot of time when you're creating database-related forms.

## Using Field Objects

Before we look at an example, let's go over the use of the *TField* class. Don't underestimate the importance of this component: Although it is often used behind the scenes, its role in database applications is fundamental. As I already mentioned, even if you do not define specific objects of this kind, you can always access the fields of a table or a query using their *Fields* array property, the *FieldValues* indexed property, or the *FieldByName* method. Both the *Fields* property and the *FieldByName* function return an object of type *TField*, so you sometimes have to use the *as* operator to downcast their result to its type (like *TFloatField* or *TDateField*) before accessing specific properties of these subclasses.

The *FieldAcc* example has a form with three speed buttons in the toolbar panel, which access various field properties

at run time. The first button changes the formatting of the grid's population column. To do this, you have to access the `DisplayFormat` property, which is a specific property of the `TFloatField` class:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);  
begin  
    (cds.FieldByName ('Population') as  
        TFloatField).DisplayFormat := '###,###,###';  
end;
```

When you set field properties related to data input or output, the changes apply to every record in the table. When you set properties related to the value of the field, however, you always refer to the current record only. For example, you can output the population of the current country in a message box by writing the following:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);  
begin  
    ShowMessage (string (cds ['Name']) + ': ' + string (cds ['Population']));  
end;
```

When you access the value of a field, you can use a series of `As` properties to handle the current field value using a specific data type (if this data type is available; otherwise, an exception is raised):

```
AsBoolean: Boolean;  
AsDateTime: TDateTime;  
AsFloat: Double;  
AsInteger: LongInt;  
AsString: string;  
AsVariant: Variant;
```

These properties can be used to read or change the value of the field. Changing the value of a field is possible only if the dataset is in edit mode. As an alternative to the `As` properties, you can access the value of a field by using its `Value` property, which is defined as a variant.

Most of the other properties of the `TField` component, such as `Alignment`, `DisplayLabel`, `DisplayWidth`, and `Visible`, reflect elements of the field's user interface and are used by the various data-aware controls, particularly `DBGrid`. In the `FieldAcc` example, clicking the third speed button changes the `Alignment` of every field:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);  
var  
    I: Integer;  
begin  
    for I := 0 to cds.FieldCount - 1 do  
        cds.Fields[I].Alignment := taCenter;  
end;
```

This change affects the output of the `DBGrid` and of the `DBEdit` control I added to the toolbar, which shows the name of the country. You can see this effect, along with the new display format, in [Figure 13.6](#).

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32,300,003
Bolivia	La Paz	South America	1096575	7,300,000
Brazil	Brasilia	South America	8511196	150,400,000
Canada	Ottawa	North America	9976147	26,500,000
Chile	Santiago	South America	756943	13,200,000
Colombia	Bagota	South America	1138907	33,000,000
Cuba	Havana	North America	114524	10,600,000
Ecuador	Quito	South America	455502	10,600,000
El Salvador	San Salvador	North America	20865	5,300,000
Guyana	Georgetown	South America	214969	800,000
Jamaica	Kingston	North America	11424	2,500,000
Mexico	Mexico City	North America	1967180	88,600,000
Nicaragua	Managua	North America	139000	3,900,000
Paraguay	Azuncion	South America	406576	4,660,000
Peru	Lima	South America	1285215	21,600,000
United States of America	Washington	North America	9363130	249,200,000

**Figure 13.6:** The output of the FieldAcc example after the Center and Format buttons have been clicked

## A Hierarchy of Field Classes

The VCL includes a number of field class types. Delphi automatically uses one of them depending on the data definition in the database, when you open a table at run time or when you use the Fields Editor at design time. [Table 13.2](#) shows the complete list of subclasses of the TField class.

**Table 13.2: The Subclasses of TField**

Subclass	Base Class	Definition
TADTField	TObjectField	An ADT (Abstract Data Type) field, corresponding to an object field in an object relational database.
TAggregateField	TField	Represents a maintained aggregate. It is used in the ClientDataSet component and is discussed in <a href="#">Chapter 14</a> .
TArrayField	TObjectField	An array of objects in an object relational database.
TAutoIncField	TIntegerField	A whole positive number connected with a Paradox table's auto-increment field (a special field automatically assigned a different value for each record). Note that Paradox AutoInc fields do not always work perfectly, as discussed in <a href="#">Chapter 14</a> .
TBCDField	TNumericField	Real numbers with a fixed number of digits after the decimal point.
TBinaryField	TField	Generally not used directly. This is the base class of the next two classes.
TBlobField	TField	Binary data with no size limit (BLOB stands for binary large object). The theoretical maximum limit is 2 GB.
TBooleanField	TField	A Boolean value.
TBytesField	TBinaryField	Arbitrary data with a large (up to 64 KB characters) but fixed size.

TCurrencyField	TFloatField	Currency values with the same range as the Real data type.
TDataSetField	TObjectField	An object corresponding to a separate table in an object relational database.
TDateField	TDateTimeField	A date value.
TDateTimeField	TField	A date and time value.
TFloatField	TNumericField	Floating-point numbers (8 byte).
TFMTBCDField	TNumericField	(New field type in Delphi 6.) A true binary-coded decimal (BCD), as opposed to the existing TBCDField type, which converted BCD values to the Currency type. This field type is used automatically only by dbExpress datasets.
TGraphicField	TBlobField	A graphic of arbitrary length.
TGuidField	TStringField	A field representing a COM Globally Unique Identifier (part of the ADO support).
TIDispatchField	TInterfaceField	A field representing pointers to IDispatch COM interfaces (part of the ADO support).
TIntegerField	TNumericField	Whole numbers in the range of long integers (32 bits).
TInterfacedField	TField	Generally not used directly. This is the base class of fields that contain pointers to interfaces (IUnknown) as data.
TLargeIntField	TIntegerField	Very large integers (64 bit).
TMemoField	TBlobField	Text of arbitrary length.
TNumericField	TField	Generally not used directly. This is the base class of all the numeric field classes.
TObjectField	TField	Generally not used directly. This is the base class for fields providing support for object relational databases.
TReferenceField	TObjectField	A pointer to an object in an object relational database.
TSmallIntField	TIntegerField	Whole numbers in the range of integers (16 bits).
TSQLTimeStampField	TField	(New field type in Delphi 6.) Supports the date/time representation used in dbExpress drivers.
TStringField	TField	Text data of a fixed length (up to 8192 bytes).
TTimeField	TDateTimeField	A time value.
TVarBytesField	TBytesField	Arbitrary data; up to 64 KB of characters. Very similar to the TBytesField base class.
TVariantField	TField	A field representing a variant data type (part of the ADO support).
TWideStringField	TStringField	A field representing a Unicode (16 bits per character) string.

TWordField

TIntegerField

Whole positive numbers in the range of words or unsigned integers (16 bits).

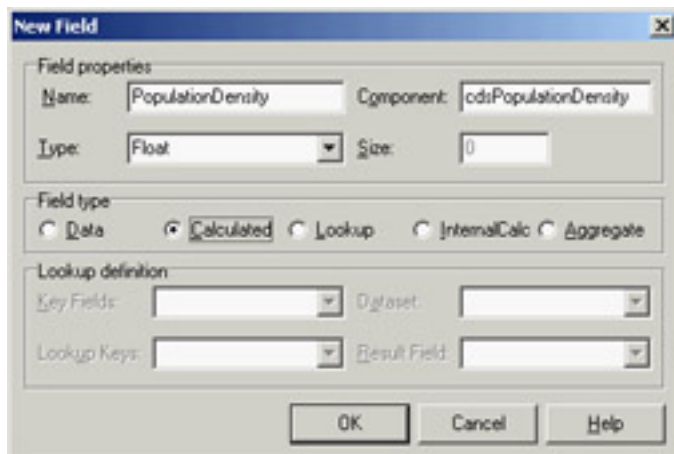
The availability of any particular field type, and the correspondence with the data definition, depends on the database in use. This is particularly true for the field types that provide support for object relational databases.

## Adding a Calculated Field

Now that you've been introduced to `TField` objects and have seen an example of their run time use, I will build an example based on the declaration of field objects at design time using the Fields Editor and then add a calculated field. In the `country.cds` sample dataset, both the population and the area of each country are available; you can use this data to compute the population density.

To build the new example, named Calc, follow these steps:

1. Add a `ClientDataSet` component to a form.
2. Open the Fields Editor. In this editor, right-click, choose the Add Field command, and select some of the fields. (I included them all.)
3. Select the New Field command and enter a proper name and data type (Float, for a `TFloatField`) for the new calculated field, as you can see in [Figure 13.7](#).



**Figure 13.7:** The definition of a calculated field in the Calc example

**Warning** Obviously, because you create some field components at design time using the Fields Editor, the fields you skip won't get a corresponding object. As I already mentioned, the fields you skip will not be available even at run time, with `Fields` or `FieldByName`. When a program opens a table at run time, if there are no design-time field components, Delphi creates field objects corresponding to the table definition. If there are some design-time fields, however, Delphi uses those fields without adding any extra field objects.

Of course, you also need to provide a way to calculate the new field. This is accomplished in the `OnCalcFields` event of the `ClientDataSet` component, which has the following code (at least in a first version):

```
procedure TForm2.cdsCalcFields(DataSet: TDataSet);
begin
    cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value;
end;
```



**Note** In general, calculated fields are computed for each record and recalculated each time the record is loaded in an internal buffer, invoking the *OnCalcFields* event over and over. For this reason, a handler of this event should be extremely fast to execute and cannot alter the status of the dataset by accessing different records. A more time-efficient (but less memory-efficient) version of a calculated field is provided by the ClientDataSet component with internally calculated fields: These fields are evaluated only once—when they are loaded—and the result is stored in memory for future requests.

Everything fine? Not at all! If you enter a new record and do not set the value of the population and area, or if you accidentally set the area to zero, the division will raise an exception, making it problematic to continue using the program. As an alternative, you could have handled every exception of the division expression and set the resulting value to zero:

```
try
  cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value;
except
  on Exception do
    cdsPopulationDensity.Value := 0;
end;
```

However, you can do even better: You can check whether the value of the area is defined—if it is not null—and whether it is not zero. It is better to avoid using exceptions when you can anticipate possible error conditions:

```
if not cdsArea.IsNull and (cdsArea.Value <> 0) then
  cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value
else
  cdsPopulationDensity.Value := 0;
```

The code for the *cdsCalcFields* method (in each of the three versions) accesses some fields directly. It can do so because you used the Fields Editor, and it automatically created the corresponding field declarations, as you can see in this excerpt of the form's interface declaration:

```
type
  TCalcForm = class(TForm)
    cds: TClientDataSet;
    cdsPopulationDensity: TFloatField;
    cdsArea: TFloatField;
    cdsPopulation: TFloatField;
    cdsName: TStringField;
    cdsCapital: TStringField;
    cdsContinent: TStringField;
    procedure cdsCalcFields(DataSet: TDataset);
    ...
```

Each time you add or remove fields in the Fields Editor, you can see the effect of your action immediately in the grid present in the form (unless the grid has its own column objects defined, in which case you often don't see any change). Of course, you won't see the values of a calculated field at design time; they are available only at run time, because they result from the execution of compiled Delphi language code.

Because you have defined components for the fields, you can use them to customize some of the grid's visual elements. For example, to set a display format that adds a comma to separate thousands, you can use the Object Inspector to change the *DisplayFormat* property of some field components to *###,###,###*. This change has an immediate effect on the grid at design time.

**Note** The display format I just mentioned (and used in the previous example) uses the Windows International Settings to format the output. When Delphi translates the numeric value of this field to text, the comma in the format string is replaced by the proper *ThousandSeparator* character. For this reason, the output of the program will automatically adapt itself to different international settings. On computers that have the Italian configuration, for example, the comma is replaced by a period.

After working on the table components and the fields, I customized the DBGrid using its *Columns* property editor. I set



the Population Density column to read-only and set its `ButtonStyle` property to `cbsEllipsis` to provide a custom editor. When you set this value, a small button with an ellipsis is displayed when the user tries to edit the grid cell. Clicking the button invokes the DBGrid's `OnEditButtonClick` event:

```
procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
    MessageDlg (Format (
        'The population density (%.2n)'#13 +
        'is the Population (%.0n)'#13 +
        'divided by the Area (%.0n).'end;
```

I haven't provided a real editor but rather a message describing the situation, as you can see in [Figure 13.8](#), which shows the values of the calculated fields. To create an editor, you might build a secondary form to handle special data entries.

Name	Capital	Continent	Population	Area	Population Density
Argentina	Buenos Aires	South America	32,300,003	2,777,815	11.63
Bolivia	La Paz	South America	7,300,000	1,098,575	6.64
Brazil	Brasilia	South America	150,400,000	8,511,196	17.67
Canada	Ottawa	North America	26,500,000	9,976,147	2.66
Chile	Santiago	South America	13,200,000	756,943	17.44
Colombia	Bogota	South America	33,000,000	1,138,907	28.98
Cuba	Havana	North America	10,600,000	114,524	92.56
Ecuador	Quito	South America	10,600,000	455,502	23.27
El Salvador	San Salvador	North America	5,300,000	20,865	254.01

**Figure 13.8:** The output of the Calc example. Notice the Population Density calculated column and the ellipsis button displayed when you edit it.

## Lookup Fields

As an alternative to placing a `DBLookupComboBox` component in a form (discussed earlier in this chapter in the section "[Using Lookup Controls](#)"), you can also define a lookup field, which can be displayed with a drop-down lookup list inside a `DBGrid` component. You've seen that to add a fixed selection to a `DBGrid`, you can edit the `PickList` subproperty of the `Columns` property. To customize the grid with a live lookup, however, you have to define a lookup field using the Fields Editor.

As an example, I built the `FieldLookup` program, which has a grid that displays orders; it includes a lookup field to display the name of the employee who took the order, instead of the employee's code number. To accomplish this functionality, I added to the data module a `ClientDataSet` component referring to the `employee.cds` dataset. Then I opened the Fields Editor for the orders dataset and added all the fields. I selected the `EmpNo` field and set its `Visible` property to `False` to remove it from the grid (you cannot remove it altogether, because it is used to build the cross-reference with the corresponding field of the employee dataset).

Now it is time to define the lookup field. If you followed the preceding steps, you can use the Fields Editor of the orders dataset and select the `New Field` command to open the `New Field` dialog box. The values you specify here will affect the properties of a new `TField` added to the table, as demonstrated by the DFM description of the field:

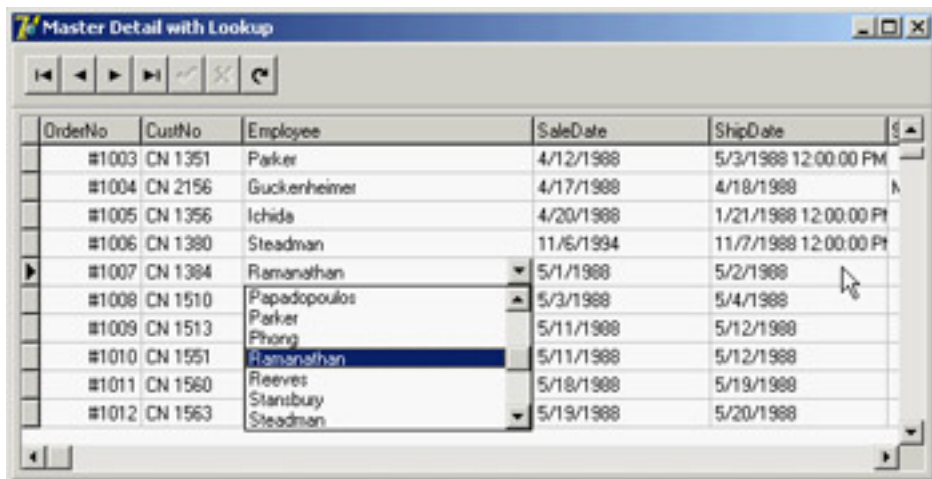
```
object cds2Employee: TStringField
    FieldKind = fkLookup
    FieldName = 'Employee'
```

```

LookupDataSet = cds2
LookupKeyFields = 'EmpNo'
LookupResultField = 'LastName'
KeyFields = 'EmpNo'
Size = 30
Lookup = True
end

```

This is all that is needed to make the drop-down list work (see [Figure 13.9](#)) and to also view the value of the cross-reference field at design time. Notice that you don't need to customize the `Columns` property of the grid because the drop-down button and the value of seven rows are used by default. However, this doesn't mean you cannot use this property to further customize these and other visual elements of the grid.



**Figure 13.9:** The output of the FieldLookup example, with the drop-down list inside the grid displaying values taken from another database table

This program has another specific feature. The two `ClientDataSet` components and the two `DataSource` components have not been placed on a form but rather on a special container for nonvisual components called a *data module* (see the sidebar "[A Data Module for Data-Access Components](#)"). You can obtain a data module from Delphi's File → New menu. After adding components to it, you can link them from controls on other forms with the File → Use Unit command.

---

### A Data Module for Data-Access Components

To build a Delphi database application, you can place data-access components and the data-aware controls in a form. This approach is handy for a simple program, but having the user interface and the data access and data model in a single (often large) unit is far from a good idea. For this reason, Delphi implements the idea of a *data module*: a container of non-visual components.

At design time, a data module is similar to a form, but at run time it exists only in memory. The `TDataModule` class derives directly from `TComponent`, so it is unrelated to the Windows concept of a window (and is fully portable among different operating systems). Unlike a form, a data module has just a few properties and events. So, it's useful to think of data modules as components and method containers.

Like a form or a frame, a data module has a designer. Delphi creates a specific unit for the definition of the data module's class and a form definition file that lists its components and their properties.

There are several reasons to use data modules. The simplest is that they let you share data-access components among multiple forms, as I'll demonstrate at the beginning of [Chapter 14](#). This technique works in conjunction with visual form linking—the ability to access components of another form or data module at design time (with the File → Use Unit command). The second reason is that data modules separate the data from the user interface,

improving the structure of an application. Data modules in Delphi even exist in versions specific for multitier applications (remote data modules) and server-side HTTP applications (web data modules).

## Handling Null Values with Field Events

In addition to a few interesting properties, field objects have a few key events. The `OnValidate` event can be used to provide extended validation of a field's value and should be used whenever you need a complex rule that the ranges and constraints provided by the field cannot express. This event is triggered before the data is written to the record buffer, whereas the `OnChange` event is fired soon after the data has been written.

Two other events—`OnGetText` and `OnSetText`—can be used to customize a field's output. These two events are extremely powerful: They allow you to use data-aware controls even when the representation of a field you want to display is different from the one Delphi will provide by default.

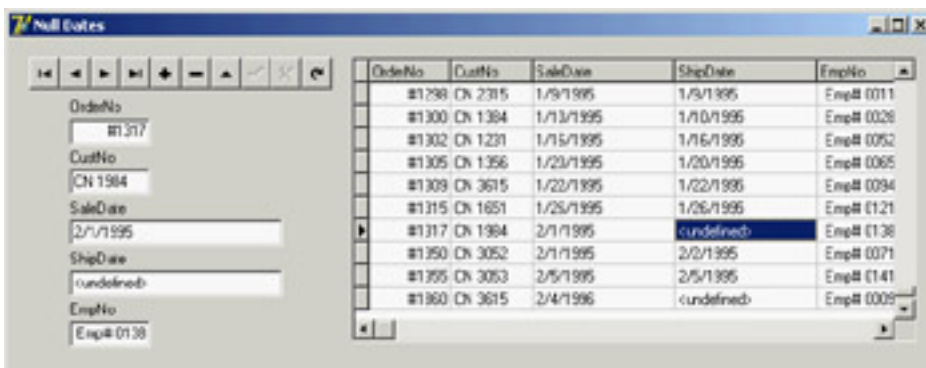
Handling null values provides an example of the use of these events. On SQL servers, storing an empty value for a field is a separate operation from storing a null value for a field. The latter tends to be more correct, but Delphi by default uses empty values and displays the same output for an empty or a null field. Although this behavior can be useful in general for strings and numbers, it becomes extremely important for dates, where it is hard to set a reasonable default value and where if the deletes the contents of the field, you might have invalid input.

The `NullDates` program displays specific text for dates that have a null value and clears the field (setting it to the null value) when the user uses an empty string in input. Here is the relevant code of the field's two event handlers:

```
procedure TForm1.cdsShipDateGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if Sender.IsNull then
    Text := '<undefined>'
  else
    Text := Sender.AsString;
end;

procedure TForm1.cdsShipDateSetText(Sender: TField; const Text: String);
begin
  if Text = '' then
    Sender.Clear
  else
    Sender.AsString := Text;
end;
```

[Figure 13.10](#) shows an example of the program's output, with undefined (or null) values for some shipping dates.



**Figure 13.10:** By handling the `OnGetText` and `On-SetText` events of a date field, the `NullDates` example displays

specific output for null values.

**Warning**

The handling of null values in Delphi 6 and 7 can be affected by changes in the way null variants work. As discussed in [Chapter 3](#), "The Run Time Library," in the section "The Variants and VarUtils Units," comparing a field with a null value with another field will have a different effect in the latest versions of Delphi than in the past. As discussed in that section, in Delphi 7 you can use global variables to fine-tune the effect of comparisons involving variants.

Team LiB

◀ PREVIOUS

NEXT ▶

## Navigating a Dataset

You've seen that a dataset has only one active record; this active record changes frequently in response to user actions or because of internal commands given to the dataset. To move around the dataset and change the active record, you can use methods of the `TDataSet` class as you saw in [Listing 13.2](#), (particularly in the section commented `position, movement`). You can move to the next or previous record, jump back and forth by a given number of records (with `MoveBy`), or go directly to the first or last record of the dataset. These dataset operations are available in the `DBNavigator` component and the standard dataset actions, and they are not difficult to understand.

What is not obvious, though, is how a dataset handles extreme positions. If you open any dataset with a navigator attached, you can see that as you move record by record, the `Next` button remains enabled even when you reach the last record. Only when you try to move forward after the last record does the button become disabled (and the current record doesn't change). This happens because the `EOF` (end of file) test succeeds only when the cursor has been moved to a special position after the last record. If you jump with the `Last` button instead, you'll immediately be at the end. You'll encounter the same behavior for the first record (and the `BOF` test). As you'll soon see, this approach is handy because you can scan a dataset testing for `EOF` to be `True`, and know at that point you've already processed the last record of the dataset.

In addition to moving record by record or by a given number of records, programs might need to jump to specific records or positions. Some datasets support the `RecordCount` property and allow movement to a record at a given position in the dataset using the `RecNo` property. You can use these properties only for datasets that support positions natively, which basically excludes all client/server architectures, unless you grab all the records in a local cache (something you'll generally want to avoid) and then navigate on the cache. As you'll see in [Chapter 14](#), when you open a query on a SQL server, you fetch only the records you are using, so Delphi doesn't know the record count (at least, not in advance).

You can use two alternatives to refer to a record in a dataset, regardless of its type:

- You can save a reference to the current record and then jump back to it after moving around. You do so by using bookmarks, either in the `TBookmark` or the more modern `TBookmarkStr` form. This approach is discussed in the section "[Using Bookmarks](#)."
- You can locate a dataset record that matches given criteria, using the `Locate` method. This approach, which is presented in the [next section](#), works even after you close and reopen the dataset, because you're working at a logical (not physical) level.

## The Total of a Table Column

So far in our examples, the user can view the current contents of a database table and manually edit the data or insert new records. Now you will see how you can change data in the table through the program code. The employee dataset you have already used has a `Salary` field, so a manager of the company can browse through the table and change the salary of a single employee. But what is the company's total salary expense? And what if the manager wants to give everyone a 10 percent salary increase (or decrease)?

The program, which also demonstrates the use of an action list for the standard dataset actions, has buttons to calculate the sum of the current salaries and change them. The total action lets you calculate the sum of the salaries of all the employees. Basically, you need to scan the table, reading the value of the `cdsSalary` field for each record:

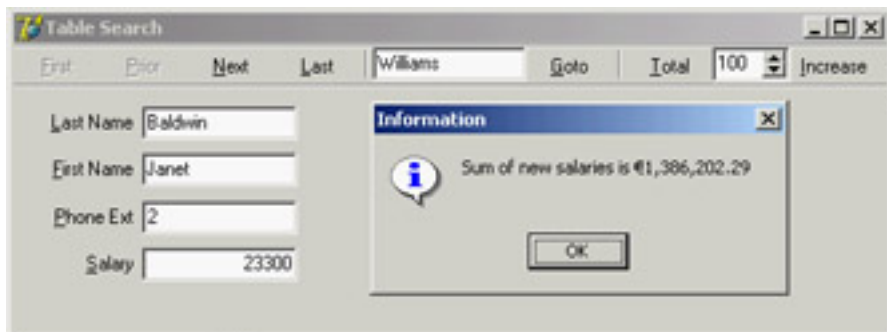
```
var
    Total: Double;
begin
    Total := 0;
    cds.First;
    while not cds.EOF do
        begin
```

```

Total := Total + cdsSalary.Value;
cds.Next;
end;
MessageDlg ('Sum of new salaries is ' +
Format ('%m', [Total]), mtInformation, [mbOk], 0);
end

```

This code works, as you can see from the output in [Figure 13.11](#), but it has some problems. One problem is that the record pointer is moved to the last record, so the previous position in the table is lost. Another is that the user interface is refreshed many times during the operation.



**Figure 13.11:** The output of the Total program, showing the total salaries of the employees

## Using Bookmarks

To avoid the two problems I just mentioned, you need to disable updates and to store the current position of the record pointer in the table and restore it at the end. You can do so using a *table bookmark*: a special variable that stores the position of a record in a database table. Delphi's traditional approach is to declare a variable of the `TBookmark` data type and initialize it while getting the current position from the table:

```

var
  Bookmark: TBookmark;
begin
  Bookmark := cds.GetBookmark;

```

At the end of the `ActionTotalExecute` method, you can restore the position and delete the bookmark with the following two statements (inside a `finally` block to ensure the pointer's memory is definitely freed):

```

cds.GotoBookmark (Bookmark);
cds.FreeBookmark (Bookmark);

```

As a better (and more up-to-date) alternative, you can use the `TDataset` class's `Bookmark` property, which refers to a bookmark that is disposed of automatically. (The property is technically implemented as an *opaque string*—a structure subject to string lifetime management—but it is not a string, so you're not supposed to look at what's inside it.) This is how you can modify the previous code:

```

var
  Bookmark: TBookmarkStr;
begin
  Bookmark := cds.Bookmark;
  ...
  cds.Bookmark := Bookmark;

```

To avoid the other side effect of the program (you see the records scrolling while the routine browses through the data), you can temporarily disable the visual controls connected with the table. The dataset has a `DisableControls` method you can call before the `while` loop starts and an `EnableControls` method you can call at the end, after the record pointer is restored.

**Tip** Disabling the data-aware controls connected with a dataset during long operations not only improves the user interface (because the output is not changing constantly), but also speeds up the program considerably. The time spent updating the user interface is much greater than the time spent performing the calculations. To test this fact, try commenting out the *DisableControls* and *EnableControls* methods in the Total example and see the speed difference.

You face some dangers from errors in reading the table data, particularly if the program is reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. To avoid this situation, you should use a `try/finally` block; to make the program 100 percent error-proof, use two nested `try/finally` blocks. Including this change and the two just discussed, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);  
var  
    Bookmark: TBookmarkStr;  
    Total: Double;  
begin  
    Bookmark := Cds.Bookmark;  
    try  
        cds.DisableControls;  
        Total := 0;  
        try  
            cds.First;  
            while not cds.EOF do  
                begin  
                    Total := Total + cdsSalary.Value;  
                    cds.Next;  
                end;  
            finally  
                cds.EnableControls;  
            end  
        finally  
            cds.Bookmark := Bookmark;  
        end;  
    MessageDlg ('Sum of new salaries is ' +  
        Format('%m', [Total]), mtInformation, [mbOK], 0);  
end;
```

**Note** I wrote this code to show you an example of a loop to browse the contents of a dataset, but there is an alternative approach based on the use of a SQL query that returns the sum of the values of a field. When you use a SQL server, the speed advantage of a SQL call to compute the total can be significant, because you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result. There is also a better alternative when you're using a `ClientDataSet`, because totaling a column is one of the features provided by aggregates (discussed toward the end of this chapter). Here I discussed a generic solution, which should work for any dataset.

## Editing a Table Column

The code of the increase action is similar to the action you just saw. The `ActionIncreaseExecute` method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference: When you increase the salary, you change the data in the table. The two key statements are within the `while` loop:

```
while not cds.EOF do  
begin  
    cds.Edit;  
    cdsSalary.Value := Round (cdsSalary.Value * SpinEdit1.Value) / 100;  
    Total := Total + cdsSalary.Value;  
end;
```



```
    cds.Next ;  
end;
```

The first statement brings the dataset into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary, multiplying the old salary by the value of the SpinEdit component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount with the click of a button.

**Warning** Notice that the dataset enters edit mode every time the *while* loop is executed. This happens because in a dataset, edit operations can take place only one record at a time. You must finish the edit operation by calling *Post* or by moving to a different record, as in the previous code. Then, to change another record, you have to re-enter edit mode.

Team LiB

◀ PREVIOUS

NEXT ▶



## Customizing a Database Grid

Unlike most other data-aware controls, which have few properties to tune, the DBGrid control has many options and is more powerful than you might think. The following sections explore some of the advanced operations you can do using a DBGrid control. The first example shows how to draw in a grid, and the second shows how to use the grid's multiple-selection feature.

### Painting a DBGrid

There are many reasons you might want to customize the output of a grid. A good example is to highlight specific fields or records. Another is to provide output for fields that usually don't show up in the grid, such as BLOB, graphic, and memo fields.

To thoroughly customize the drawing of a DBGrid control, you must set its `DefaultDrawing` property to `False` and handle its `OnDrawColumnCell` event. If you leave the value of `DefaultDrawing` set to `True`, the grid will display the default output before the method is called. In that case, all you can do is add something to the default output of the grid (unless you decide to draw over it, which will take extra time and cause flickering).

The alternative approach is to call the grid's `DefaultDrawColumnCell` method, perhaps after changing the current font or restricting the output rectangle. In this last case, you can provide an extra drawing in a cell and let the grid fill the remaining area with the standard output. This is what I did in the `DrawData` program.

The DBGrid control in this example, which is connected to the Borland's classic `Biolife` table, has the following properties:

```
object DBGrid1: TDBGrid
  Align = alClient
  DataSource = DataSource1
  DefaultDrawing = False
  OnDrawColumnCell = DBGrid1DrawColumnCell
end
```

The `OnDrawColumnCell` event handler is called once for each grid cell and has several parameters, including the rectangle corresponding to the cell, the index of the column you have to draw, the column itself (with the field, its alignment, and other subproperties), and the status of the cell. To set the color of specific cells to red, you change it in the special cases:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  // red font color if length > 100
  if (Column.Field = cdsLengthcm) and (cdsLengthcm.AsInteger > 100) then
    DBGrid1.Canvas.Font.Color := clRed;

  // default drawing
  DBGrid1.DefaultDrawDataCell (Rect, Column.Field, State);
end;
```

The next step is to draw the memo and the graphics fields. For the memo, you can implement the memo field's `OnGetText` and `OnSetText` events. The grid will even allow editing on a memo field if its `OnSetText` event is not `nil`. Here is the code for the two event handlers. I used `Trim` to remove trailing nonprinting characters that make the text appear to be empty when editing:

```
procedure TForm1.cdsNotesGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
```

```
Text := Trim (Sender.AsString);  
end;
```

```
procedure TForm1.cdsNotesSetText(Sender: TField; const Text: String);  
begin  
    Sender.AsString := Text;  
end;
```

For the image, the simplest approach is to create a temporary `TBitmap` object, assign the graphics field to it, and paint the bitmap to the grid's `Canvas`. As an alternative, I removed the graphics field from the grid by setting its `Visible` property to `False` and added the image to the fish name, with the following extra code in the `OnDrawColumnCell` event handler:

```
var  
    Picture: TPicture;  
    OutRect: TRect;  
    PictWidth: Integer;  
begin  
    // default output rectangle  
    OutRect := Rect;  
  
    if Column.Field = cdsCommon_Name then  
    begin  
        // draw the image  
        Picture := TPicture.Create;  
        try  
            Picture.Assign(cdsGraphic);  
            PictWidth := (Rect.Bottom - Rect.Top) * 2;  
            OutRect.Right := Rect.Left + PictWidth;  
            DBGrid1.Canvas.StretchDraw (OutRect, Picture.Graphic);  
        finally  
            Picture.Free;  
        end;  
        // reset output rectangle, leaving space for the graphic  
        OutRect := Rect;  
        OutRect.Left := OutRect.Left + PictWidth;  
    end;  
  
    // red font color if length > 100 (omitted – see above)  
  
    // default drawing  
    DBGrid1.DefaultDrawDataCell (OutRect, Column.Field, State);
```

As you can see in this code, the program shows the image in a small rectangle on the left of the grid cell and then changes the output rectangle to the remaining area before activating the default drawing. You can see the effect in [Figure 13.12](#).

Species No	Category	Common Name	Species Name	Length (cm)	Length (in)
90020	Triggerfish	Clown Triggerfish	<i>Fallicaudes conspicillum</i>	50	19.685
90030	Surfscia	Red Emperor	<i>Lutjanus seabe</i>	60	23.622
90050	Wrasse	Giant Moorii Wrasse	<i>Cheilodactylus</i>	229	90.157
90070	Angelfish	Blue Angelfish	<i>Pomacentrus saurochus</i>	30	11.811
90080	Cod	Lanternfish	<i>Varicorhinus</i>	80	31.49
90090	Scorpaenid	Firefish	<i>Pterois volitans</i>	38	14.960
90100	Butterflyfish	Ornate Butterflyfish	<i>Chaetodon ornatissimus</i>	19	7.4803
90110	Shark	Shovel Shark	<i>Cephaloscyllium ventriosum</i>	102	40.157
90120	Ray	Bat Ray	<i>Myliobatis californica</i>	56	22.047
90130	Eel	California Moray	<i>Gymnothorax mordax</i>	150	59.055
90140	Cod	Langcod	<i>Ophiodes elongatus</i>	150	59.055

**Figure 13.12:** The DrawData program displays a grid that includes the text of a memo field and the ubiquitous Borland fish.

## A Grid Allowing Multiple Selection

The second example of customizing the DBGrid control relates to multiple selection. You can set up the DBGrid so that a user can select multiple rows (that is, multiple records). Doing so is easy, because all you have to do is toggle the `dgMultiSelect` element of the grid's `Options` property. Once you select this option, a user can keep the `Ctrl` key pressed and click with the mouse to select multiple grid rows, with the effect shown in [Figure 13.13](#).

Name	Capital	Continent
Argentina	Buenos Aires	South Ame
Bolivia	La Paz	South Ame
Brazil	Brasilia	South Ame
Canada	Ottawa	North Amer
Chile	Santiago	South Ame
Colombia	Bagota	South Ame
Cuba	Havana	North Amer
Ecuador	Quito	South Ame
El Salvador	San Salvador	North Amer
Guyana	Georgetown	South Ame
Jamaica	Kingston	North Amer
Mexico	Mexico City	North Amer
Nicaragua	Managua	North Amer
Paraguay	Asuncion	South Ame

Get Selected

Canada  
Cuba  
El Salvador

**Figure 13.13:** The MltGrid example has a DBGrid control that allows the selection of multiple rows.

Because the database table can have only one active record, the grid keeps a list of bookmarks to the selected records. This list is available in the `SelectedRows` property, which is of type `TBookmarkList`. Besides accessing the number of objects in the list with the `Count` property, you can get to each bookmark with the `Items` property, which is the default array property. Each list item is of a `TBookmarkStr` type, which represents a bookmark pointer you can assign to the table's `Bookmark` property.

**Note** *TBookmarkStr* is a string type for convenience, but its data should be considered opaque and volatile. You shouldn't rely on any particular structure to the data you find if you peek at a bookmark's value, and you shouldn't hold on to the data too long or store it in a separate file. Bookmark data will vary with the database driver and index configuration, and it may be rendered unusable when rows are added to or deleted from the dataset (by you or by other users of the database).

To summarize the steps, here is the code for the MltGrid example, which is activated by clicking the button to move the Name field of the selected records to the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  BookmarkList: TBookmarkList;
```

```

Bookmark: TBookmarkStr;
begin
  // store the current position
  Bookmark := cds.Bookmark;
  try
    // empty the list box
    ListBox1.Items.Clear;
    // get the selected rows of the grid
    BookmarkList := DbGrid1.SelectedRows;
    for I := 0 to BookmarkList.Count - 1 do
      begin
        // for each, move the table to that record
        cds.Bookmark := BookmarkList[I];
        // add the name field to the listbox
        ListBox1.Items.Add (cds.FieldByName ('Name').AsString);
      end;
    finally
      // go back to the initial record
      cds.Bookmark := Bookmark;
    end;
end;

```

## Dragging to a Grid

Another interesting technique is to use dragging with grids. Dragging *from* a grid is not difficult, because you know which current record and column the user has selected. Dragging *to* a grid, however, is tricky to program. Recall that in [Chapter 3](#) I mentioned the "protected hack"; I'll use this technique to implement dragging to a grid.

The example, called DragToGrid, has a grid connected to the country dataset, an edit box in which you can type the new value for a field, and a label you can drag over a grid cell to modify the related field. The problem is how to determine this field. The code is only a few lines, as you can see here, but it is cryptic and requires some explanation:

```

type
  TDBGHack = class (TDbGrid)
  end;

procedure TFormDrag.DBGrid1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  gc: TGridCoord;
begin
  gc := TDBGHack (DbGrid1).MouseCoord (x, y);
  if (gc.y > 0) and (gc.x > 0) then
    begin
      DbGrid1.DataSource.DataSet.MoveBy (gc.y - TDBGHack(DbGrid1).Row);
      DbGrid1.DataSource.DataSet.Edit;
      DBGrid1.Columns.Items [gc.X - 1].Field.AsString := EditDrag.Text;
    end;
  DBGrid1.SetFocus;
end;

```

The first operation determines the cell over which the mouse was released. Starting with the *x* and *y* mouse coordinates, you can call the protected `MouseCoord` method to access the row and column of the cell. Unless the drag target is the first row (usually hosting the titles) or the first column (usually hosting the indicator), the program moves the current record by the difference between the requested row (`gc.y`) and the current active row (the grid's protected `Row` property). The next step puts the dataset into edit mode, grabs the field of the target column (`Columns.Items [gc.X - 1].Field`), and changes its text.



## Database Applications with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this approach is not required. When you need precise control over the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. My view is that doing so is necessary only in specific cases, because you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you better understand Delphi's default behavior.

The development of an application not based on data-aware controls can follow two different approaches: You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a more customized approach. I'll demonstrate the first technique in the NonAware example and the latter in the SendToDb example.

### Mimicking Delphi Data-Aware Controls

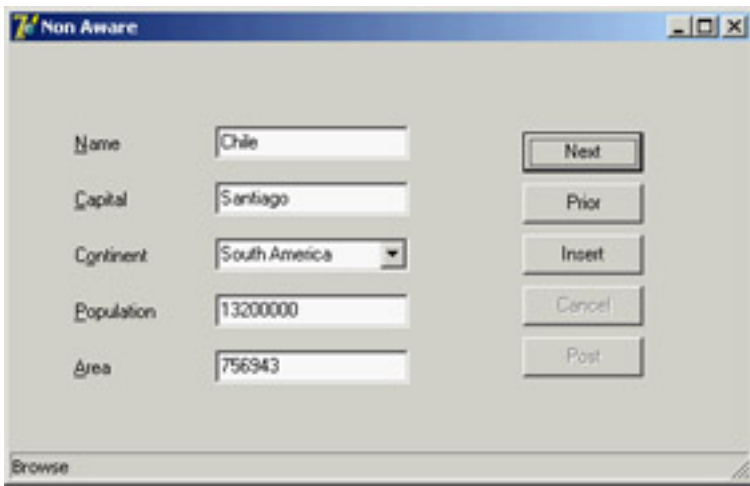
To build an application that doesn't use data-aware controls but behaves like a standard Delphi application, you can write event handlers for the operations that would be performed automatically by data-aware controls. Basically, you need to place the dataset in edit mode as the user changes the content of the visual controls and update the field objects of the dataset as the user exits from the controls, moving the focus to another element.

**Tip** This approach can be handy for integrating a control that's not data aware into a standard application.

The other element of the NonAware example is a list of buttons corresponding to some of the buttons in the DBNavigator control; these buttons are connected to five custom actions. I could not use the standard dataset actions for this example because they automatically hook to the data source associated with the control having the focus—a mechanism that fails with the example's non-data-aware edit boxes. In general, you could also hook a data Source with each of the actions' `DataSource` property, but in this specific case we don't have a data source in the example.

The program has several event handlers I haven't used for past applications using data-aware controls. First, you have to show the current record's data in the visual controls (as in [Figure 13.14](#)) by handling the `OnAfterScroll` event of the dataset component:

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);  
begin  
    EditName.Text := cdsName.AsString;  
    EditCapital.Text := cdsCapital.AsString;  
    ComboContinent.Text := cdsContinent.AsString;  
    EditArea.Text := cdsArea.AsString;  
    EditPopulation.Text := cdsPopulation.AsString;  
end;
```



**Figure 13.14:** The output of the NonAware example in Browse mode. The program manually fetches the data every time the current record changes.

The control's `OnStateChange` event handler displays the table's status in a status bar control. As the user begins typing in one of the edit boxes or drops down the combo box list, the program sets the table to edit mode:

```

procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
    if not (cds.State in [dsEdit, dsInsert]) then
        cds.Edit;
end;

```

This method is connected to the `OnKeyPress` event of the five components and is similar to the `OnDropDown` event handler of the combo box. As the user leaves one of the visual controls, the `OnExit` event handler copies the data to the corresponding field, as in this case:

```

procedure TForm1.EditCapitalExit(Sender: TObject);
begin
    if (cds.State in [dsEdit, dsInsert]) then
        cdsCapital.AsString := EditCapital.Text;
end;

```

The operation takes place only if the table is in edit mode—that is, only if the user has typed in this or another control. This behavior is not ideal, because extra operations are done even if the edit box's text didn't change; however, the extra steps happen fast enough that they aren't a concern. For the first edit box, you check the text before copying it, raising an exception if the edit box is empty:

```

procedure TForm1.EditNameExit(Sender: TObject);
begin
    if (cds.State in [dsEdit, dsInsert]) then
        if EditName.Text <> '' then
            cdsName.AsString := EditName.Text
        else
            begin
                EditName.SetFocus;
                raise Exception.Create ('Undefined Country');
            end;
end;

```

An alternative approach for testing the value of a field is to handle the dataset's `BeforePost` event. Keep in mind that in this example, the posting operation is not handled by a specific button but takes place as soon as a user moves to a new record or inserts a new one:

```

procedure TForm1.cdsBeforePost(DataSet: TDataSet);
begin

```

```

if cdsArea.Value < 100 then
    raise Exception.Create ('Area too small');
end;

```

In each case, an alternative to raising an exception is to set a default value. However, if a field has a default value, it is better to set it up front, so a user can see which value will be sent to the database. To accomplish this, you can handle the dataset's `AfterInsert` event, which is fired immediately after a new record has been created (I could have used the `OnNewRecord` event, as well):

```

procedure TForm1.cdsAfterInsert(DataSet: TDataSet);
begin
    cdsContinent.Value := 'Asia';
end;

```

## Sending Requests to the Database

You can further customize your application's user interface if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This approach allows you complete freedom, although it might cause some side effects (such as limited ability to handle concurrency, which I'll discuss in [Chapter 14](#)).

For this new example, I replaced the first edit box with another combo box and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom buttons that get the data from the database and send an update to it. Again, this example has no `DataSource` component.

The `GetData` method, connected to the corresponding button, gets the fields corresponding to the record indicated in the first combo box:

```

procedure TForm1.GetData;
begin
    cds.Locate ('Name', ComboName.Text, [loCaseInsensitive]);
    ComboName.Text := cdsName.AsString;
    EditCapital.Text := cdsCapital.AsString;
    ComboContinent.Text := cdsContinent.AsString;
    EditArea.Text := cdsArea.AsString;
    EditPopulation.Text := cdsPopulation.AsString;
end;

```

This method is called whenever the user clicks the button, selects an item in the combo box, or presses the Enter key while in the combo box:

```

procedure TForm1.ComboNameClick(Sender: TObject);
begin
    GetData;
end;

procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);
begin
    if Key = #13 then
        GetData;
end;

```

To make this example work smoothly, at startup the combo box is filled with the names of all the countries in the table:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    // fill the list of names
    cds.Open;
    while not cds.Eof do
        begin

```

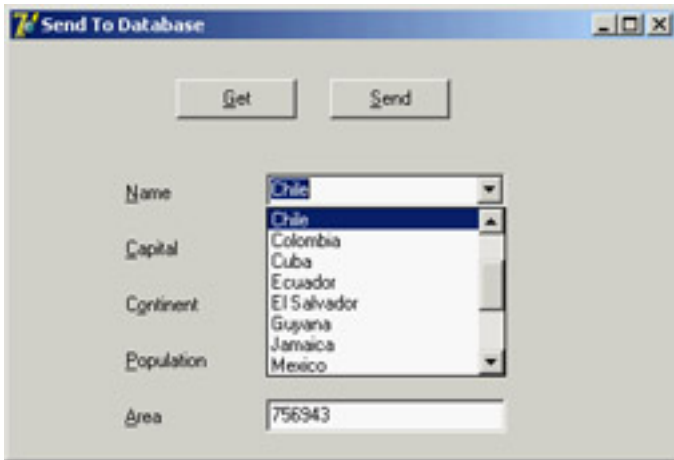


```

ComboName.Items.Add (cdsName.AsString);
cds.Next;
end;
end;

```

With this approach, the combo box becomes a sort of selector for the record, as you can see in [Figure 13.15](#). Thanks to this selection, the program doesn't need navigational buttons.



**Figure 13.15:** In the SendToDb example, you can use a combo box to select the record you want to see.

The user can also change the values of the controls and click the Send button. The code to be executed depends on whether the operation is an update or an insert. You can determine this by looking at the name (although with this code, a wrong name can no longer be modified):

```

procedure TForm1.SendData;
begin
  // raise an exception if there is no name
  if ComboName.Text = '' then
    raise Exception.Create ('Insert the name');

  // check if the record is already in the table
  if cds.Locate ('Name', ComboName.Text, [loCaseInsensitive]) then
    begin
      // modify found record
      cds.Edit;
      cdsCapital.AsString := EditCapital.Text;
      cdsContinent.AsString := ComboContinent.Text;
      cdsArea.AsString := EditArea.Text;
      cdsPopulation.AsString := EditPopulation.Text;
      cds.Post;
    end
  else
    begin
      // insert new record
      cds.InsertRecord ([ComboName.Text, EditCapital.Text,
        ComboContinent.Text, EditArea.Text, EditPopulation.Text]);
      // add to list
      ComboName.Items.Add (ComboName.Text)
    end;

```

Before sending the data to the table, you can do any sort of validation test on the values. In this case, it doesn't make sense to handle the events of the database components, because you have full control over when the update or insert operation is performed.



## Grouping and Aggregates

You've already seen that a `ClientDataSet` can have an index different from the order in which the data is stored in the file. Once you define an index, you can group the data by that index. In practice, a *group* is defined as a list of consecutive records (according to the index) for which the value of the indexed field doesn't change. For example, if you have an index by state, all the addresses within that state will fall in the group.

### Grouping

The `CdsCalcs` example has a `ClientDataSet` component that extracts its data from the familiar `Country` dataset. The group is obtained, along with the definition of an index, by specifying a grouping level for the index:

```
object ClientDataSet1: TClientDataSet
  IndexDefs = <
    item
      Name = 'ClientDataSet1Index1'
      Fields = 'Continent'
      GroupingLevel = 1
    end>
  IndexName = 'ClientDataSet1Index1'
```

When a group is active, you can make it obvious to the user by displaying the grouping structure in the `DBGrid`, as shown in [Figure 13.16](#). All you have to do is handle the `OnGetText` event for the grouped field (the `Continent` field in the example) and show the text only if the record is the first in the group:

```
procedure TForm1.ClientDataSet1ContinentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if gbFirst in ClientDataSet1.GetGroupState (1) then
    Text := Sender.AsString
  else
    Text := '';
end;
```

Continent	Name	Capital	Area	Population
North America	Mexico	Mexico City	1,967,160	88,600,000
	Nicaragua	Managua	139,000	3,900,000
	El Salvador	San Salvador	20,865	5,300,000
	Cuba	Havana	114,524	10,600,000
	Jamaica	Kingston	11,424	2,500,000
	United States of America	Washington	9,363,130	249,200,000
	Canada	Ottawa	9,976,147	26,500,000
South America	Paraguay	Azuncion	406,576	4,660,000
	Uruguay	Montevideo	176,140	3,002,000
	Venezuela	Caracas	912,047	19,700,000
	Peru	Lima	1,285,215	21,600,000
	Argentina	Buenos Aires	2,777,815	32,300,003
	Guyana	Georgetown	214,969	800,000
	Ecuador	Quito	455,502	10,600,000
	Colombia	Bogota	1,138,907	33,000,000
	Chile	Santiago	756,943	13,200,000
	Brazil	Brasilia	8,511,196	150,400,000

TotalArea aggregate: 17,733,885  
 Get Aggregates  
 Area: 17,733,885  
 Population: 683,162,003  
 Number: 11

**Figure 13.16:** The `CdsCalcs` example demonstrates that by writing a little code, you can have the `DBGrid` control visually show the grouping defined in the `ClientDataSet`.

## Defining Aggregates

Another feature of the ClientDataSet component is support for aggregates. An *aggregate* is a calculated value based on multiple records, such as the sum or average value of a field for the entire table or a group of records (defined with the grouping logic I just discussed). Aggregates are *maintained*; that is, they are recalculated immediately if one of the records changes. For example, the total of an invoice can be maintained automatically while the user types in the invoice items.

**Note** Aggregates are maintained incrementally, not by recalculating all the values every time one value changes. Aggregate updates take advantage of the deltas tracked by the ClientDataSet. For example, to update a sum when a field is changed, the ClientDataSet subtracts the old value from the aggregate and adds the new value. Only two calculations are needed, even if there are thousands of rows in that aggregate group. For this reason, aggregate updates are instantaneous.

There are two ways to define aggregates. You can use the `Aggregates` property of the ClientDataSet, which is a collection; or you can define aggregate fields using the Fields Editor. In both cases, you define the aggregate expression, give it a name, and connect it to an index and a grouping level (unless you want to apply it to the entire table). Here is the `Aggregates` collection of the `CdsCalcs` example:

```
object ClientDataSet1: TClientDataSet
  Aggregates = <
    item
      Active = True
      AggregateName = 'Count'
      Expression = 'COUNT (NAME)'
      GroupingLevel = 1
      IndexName = 'ClientDataSet1Index1'
      Visible = False
    end
    item
      Active = True
      AggregateName = 'TotalPopulation'
      Expression = 'SUM (POPULATION)'
      Visible = False
    end
  end>
  AggregatesActive = True
```

Notice in the last line of the previous code snippet that you must activate the support for aggregates, in addition to activating each specific aggregate you want to use. Disabling aggregates is important, because having too many of them can slow down a program.

The alternative approach, as I mentioned, is to use the Fields Editor, select the New Field command from its shortcut menu, and choose the Aggregate option (available, along with the InternalCalc option, only in a ClientDataSet). This is the definition of an aggregate field:

```
object ClientDataSet1: TClientDataSet
  object ClientDataSet1TotalArea: TAggregateField
    FieldName = 'TotalArea'
    ReadOnly = True
    Visible = True
    Active = True
    DisplayFormat = '###,###,###'
    Expression = 'SUM(AREA)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
  end
```

The aggregate fields are displayed in the Fields Editor in a separate group, as you can see in [Figure 13.17](#). The advantage of using an aggregate field, compared to a plain aggregate, is that you can define the display format and hook the field directly to a data-aware control, such as a DBEdit in the CdsCalcs example. Because the aggregate is connected to a group, as soon as you select a record from a different group, the output is automatically updated. Also, if you change the data, the total immediately shows the new value.



**Figure 13.17:** The bottom portion of a ClientDataSet's Fields Editor displays aggregate fields.

To use plain aggregates, you have to write a little code, as in the following example (notice that the `Value` of the aggregate is a variant):

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Label1.Caption :=  
        'Area: ' + ClientDataSet1.TotalArea.DisplayText + #13'Population : '  
        + FormatFloat ('###,###,###', ClientDataSet1.Aggregates [1].Value) +  
        #13'Number : ' + IntToStr (ClientDataSet1.Aggregates [0].Value);  
end;
```

## Master/Detail Structures

Often, you need to relate tables that have a one-to-many relationship. This means that for a single record in the master table, there are many detailed records in a secondary table. A classic example is an invoice and the items of the invoice; another is a list of customers and the orders each customer has placed.

These are common situations in database programming, and Delphi provides explicit support with the master/detail structure. The `TDataSet` class has a `DataSource` property for setting up a master data source. This property is used in a detail dataset to hook to the current record of the master dataset, in combination with the `MasterFields` property.

### Master/Detail with ClientDataSets

The `MastDet` example uses the customer and orders sample datasets. I added a data source component for each dataset, and for the secondary dataset I assigned the `DataSource` property to the data source connected to the first dataset. Finally, I related the secondary table to a field of the main table, using the `MasterFields` property's special editor. I did all this using a data module, as discussed in the earlier sidebar "[A Data Module for Data-Access Components](#)."

The following is the complete listing (but without the irrelevant positional properties) of the data module used by the `MastDet` program:

```
object DataModule1: TDataModule1
  OnCreate = DataModule1Create
  object dsCust: TDataSource
    DataSet = cdsCustomers
  end
  object dsOrd: TDataSource
    DataSet = cdsOrders
  end
  object cdsOrders: TClientDataSet
    FileName = 'orders.cds'
    IndexFieldNames = 'CustNo'
    MasterFields = 'CustNo'
    MasterSource = dsCust
  end
  object cdsCustomers: TClientDataSet
    FileName = 'customer.cds'
  end
end
```

In [Figure 13.18](#), you can see an example of the `MastDet` program's main form at run time. I placed data-aware controls related to the master table in the upper portion of the form, and I placed a grid connected with the detail table in the lower portion. This way, for every master record, you immediately see the list of connected detail records—in this case, all orders placed by the current client. Each time you select a new customer, the grid below the master record displays only the orders pertaining to that customer.

**Master Detail**

Navigation: [Home] [Back] [Forward] [Next] [Add] [Delete] [Refresh] [Print] [Cancel] [OK]

Company: Tom Sawyer Diving Centre    Add1: 632-1 Third Fydenhoj    Add2:    City: Christiansted

State: St. Croix    Zip: 00820    Country: US Virgin Islands    Phone: 504-798-3022    FAX: 504-798-7772

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
1072	1356	4/11/1989	4/12/1989	29	
1005	1356	4/20/1988	1/21/1988 12:00:00 PM	110	
1280	1356	12/26/1994	12/26/1994	118	
1059	1356	2/24/1989	2/25/1989	109	
1080	1356	5/5/1989	5/6/1989	45	
1305	1356	1/20/1995	1/20/1995	65	

**Figure 13.18:** The MastDet example at run time

## Handling Database Errors

Another important element of database programming is handling database errors in custom ways. Of course, you can let Delphi show an exception message each time a database error occurs, but you might want to try to correct the errors or show more details. You can use three approaches to handle database-related errors:

- Wrap a `try/except` block around risky database operations. This is not possible when the operation is generated by interaction with a data-aware control.
- Install a handler for the `OnException` event of the global `Application` object.
- Handle specific dataset events related to errors, such as `OnPostError`, `OnEditError`, `OnDeleteError`, and `OnUpdateError`.

Although most of the exception classes in Delphi deliver an error message, database exceptions often include error codes, native SQL server error codes and messages, and the like. The `ClientDataSet` adds only an error code to its exception class, `EDBClient`. Showing how to handle it, as I'll do next, will provide you with a guideline for other cases.

As an example, I built a database program that shows the details of the errors in a memo component (errors are automatically generated when the user clicks the program buttons). To handle all the errors, the `DBError` example installs a handler for the `OnException` event of the `Application` global object. The event handler logs some information in a memo showing the details of the database error if it is an `EDBClient`:

```
procedure TForm1.ApplicationError (Sender: TObject; E: Exception);
begin
  if E is EDBClient then
  begin
    Mem1.Lines.Add('Error: ' + (E.Message));
    Mem1.Lines.Add('  Error Code: ' +
      IntToStr(EDBClient (E).ErrorCode));
  end
  else
    Mem1.Lines.Add('Generic Error: ' + (E.Message));
end;
```



## What's Next?

In this chapter, you saw examples of database access from Delphi programs. I covered the basic data-aware components, as well as the development of database applications based on standard controls. I explored the internal architecture of the `TDataSet` class and of field objects and discussed many events and properties shared by all datasets and used by all database applications. Even though most of the examples used the `ClientDataSet` component accessing local data, the component is often also the gateway for data in a SQL server (in a client/server architecture) or in a remote application (in a three-tier architecture). I discussed calculated fields, lookup fields, customizations of the `DBGrid` control, and some advanced techniques.

I haven't delved into the database and data-access side of the picture, which depends on the type of database engine and server you're using. [Chapter 14](#) will focus on this topic, with an in-depth overview of client/server development using the `dbExpress` library provided by Borland. I'll also cover `InterBase` and the `IBX` component, giving some elements of a real-world application.

Following chapters will continue to explore the database side of Delphi programming, discussing ADO connectivity and components, Borland's three-tier architecture (`DataSnap`, formerly `MIDAS`), the development of data-aware controls and custom dataset components, and reporting technologies.

## Chapter 14: Client/Server with dbExpress

### Overview

In the [last chapter](#), we examined Delphi's support for database programming, using local files (particularly using the ClientDataSet component, or MyBase) in most of the examples but not focusing on any specific database technology. This chapter moves on to the use of SQL server databases, focusing on client/server development with the BDE and the new dbExpress technology. A single chapter cannot cover this complex topic in detail, so I'll introduce it from the perspective of the Delphi developer and add some tips and hints.

For the examples I'll use InterBase, because this Borland RDBMS (relational database management system), or SQL server, is included in the Professional and higher editions of Delphi; in addition, it is a free and open-source server (although not in all of its versions, as discussed later). I'll discuss InterBase from the Delphi perspective, without delving into its internal architecture. A lot of the information presented also applies to other SQL servers, so even if you've decided not to use InterBase, you may still find it valuable.

## The Client/Server Architecture

The database applications presented in previous chapters used native components to access data stored in files on the local machine and loaded the entire file in memory. This is an extreme approach. More traditionally, the file is read record by record so that multiple applications can access it at the same time, provided write synchronization mechanisms are used.

When the data is on a remote server, copying an entire table in memory for processing it is time- and bandwidth-consuming, and often also useless. As an example, consider taking a table like EMPLOYEE (part of the InterBase sample database, which ships with Delphi), adding thousands of records to it, and placing it on a networked computer working as a file server. If you want to know the highest salary paid by the company, you can open a dbExpress table component (`EmpTable`) or a query selecting all the records, and run this code:

```
EmpTable.Open;
EmpTable.First;
MaxSalary := 0;
while not EmpTable.Eof do
begin
  if EmpTable.FieldName ('Salary').AsCurrency > MaxSalary then
    MaxSalary := EmpTable.FieldName ('Salary').AsCurrency;
  EmpTable.Next;
end;
```

The effect of this approach is to move all the data of the table from the networked computer to the local machine—an operation that might take minutes. In this case, the proper approach is to let the SQL server compute the result directly, fetching only this single piece of information. You can do so using a SQL statement like this:

```
select Max(Salary) from Employee
```

**Note** The previous two code excerpts are part of the `GetMax` example, which includes code to time the two approaches. Using the `Table` component on the small `Employee` table takes about 10 times longer than using the query, even if the InterBase server is installed on the computer running the program.

To store a large amount of data on a central computer and avoid moving the data to client computers for processing, the only solution is to let the central computer manipulate the data and send back to the client only a limited amount of information. This is the foundation of client/server programming.

In general, you'll use an existing program on the server (an RDBMS) and write a custom client application that connects to it. Sometimes, however, you may want to write both a custom client and a custom server, as in three-tier applications. Delphi support for this type of program—which has been called the Middle-tier Distributed Application Services (MIDAS) architecture and is now dubbed `DataSnap`—is covered in [Chapter 16](#), "Multitier `DataSnap` Applications."

The *upsizing* of an application—that is, the transfer of data from local files to a SQL server database engine—is generally done for performance reasons and to allow for larger amounts of data. Going back to the previous example, in a client/server environment, the query used to select the maximum salary would be computed by the RDBMS, which would send back to the client computer only the final result—a single number. With a powerful server computer (such as a multiprocessor Sun SparcStation), the total time required to compute the result might be minimal.

However, there are other reasons to choose a client/server architecture. Such an architecture:

- Helps you manage a larger amount of data, because you don't want to store hundreds of megabytes in a local file.
- Supports the need for concurrent access to the data by multiple users at the same time. SQL server databases generally use *optimistic locking*, an approach that allows multiple users to work on the same data and delays the concurrency control until users send back updates.
- Provides data integrity, transaction control, security, access control, backup support, and the like.

- Supports programmability—the possibility of running part of the code (stored procedures, triggers, table views, and other techniques) on the server, thereby reducing the network traffic and the workload of the client computers.

Having said this, we can begin focusing on particular techniques useful for client/server programming. The general goal is to distribute the workload properly between the client and the server and reduce the network bandwidth required to move information back and forth.

The foundation of this approach is good database design, which involves both table structure and appropriate data validation and constraints, or business rules. Enforcing the validation of the data on the server is important, because the integrity of the database is one of the key aims of any program. However, the client side should include data validation as well, to improve the user interface and make the input and the processing of the data more user-friendly. It makes little sense to let the user enter invalid data and then receive an error message from the server, when you can prevent the wrong input in the first place.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Elements of Database Design

Although this is a book about Delphi programming, not databases, I feel it's important to discuss a few elements of good (and modern) database design. The reason is simple: If your database design is incorrect or convoluted, you'll either have to write terribly complex SQL statements and server-side code, or write a lot of Delphi code to access your data, possibly even fighting against the design of the `TDataSet` class.

### Entities and Relations

The classic relational database design approach, based on the entity-relation (E-R) model, involves having one table for every entity you need to represent in your database, with one field for each data element you need plus one field for every one-to-one or one-to-many relation to another entity (or table). For many-to-many relations, you need a separate table.

As an example of a one-to-one relation, consider a table representing a university course. It will have a field for each relevant data element (name and description, room where the course is held, and so on) plus a single field indicating the teacher. The teacher data really should not be stored within the course data, but in a separate table, because it may be referenced from elsewhere.

The schedule for each course can include an undefined number of hours on different days, so they cannot be added in the same table describing the course. Instead, this information must be placed in a separate table that includes all the schedules, with a field referring to the class each schedule is for. In a one-to-many relation like this, *many* records of the schedule table point to the same *one* record in the course table.

A more complex situation is required to store information about which student is taking which class. Students cannot be listed directly in the course table, because their number is not fixed, and the classes cannot be stored in the student's data for the same reason. In a similar many-to-many relation, the only approach is to create an extra table representing the relation—it lists references to students and courses.

### Normalization Rules

The classic design principles include a series of so-called *normalization* rules. The goal of these rules is to avoid duplicating data in your database (not only to save space, but mainly to avoid ending up with incongruous data). For example, you don't repeat all the customer details in each order, but refer to a separate customer entity. This way you save memory, and when a customer's details change (for example, because of a change of address), all of the customer's orders reflect the new data. Other tables that relate to the same customer will be automatically updated as well.

Normalization rules imply using codes for commonly repeated values. For example, suppose you have a few different shipment options. Rather than include a string-based description for these options within the orders table, you can use a short numeric code that's mapped to a description in a separate lookup table.

The previous rule, which should not be taken to the extreme, helps you avoid having to join a large number of tables for every query. You can either account for some de-normalization (leaving a short shipment description within the orders table) or use the client program to provide the description, again ending up with a formally incorrect database design. This last option is practical only when you use a single development environment (let's say, Delphi) to access this database.

### From Primary Keys to OIDs

In a relational database, records are identified not by a physical position (as in Paradox and other local databases) but by the data within the record. Typically, you don't need the data from every field to identify a record, but only a subset of the data, forming the *primary key*. If the fields that are part of the primary key must identify an individual record, their value must be different for each possible record of the table.

**Note** Many database servers add internal record identifiers to tables, but they do so only for internal optimization; this process has little to do with the logical design of a relational database. These internal identifiers work differently in different SQL servers and may change among versions, so you shouldn't rely on them.

Early incarnations of relational theory dictated the use of *logical keys*, which means selecting one or more fields that indicate an entity without risk of confusion. This is often easier to say than to accomplish. For example, company names are not generally unique, and even the company name and its location don't provide a complete guarantee of uniqueness. Moreover, if a company changes its name (not an unlikely event, as Borland can teach us) or its location, and you have references to the company in other tables, you must change all those references as well and risk ending up with dangling references.

For this reason, and also for efficiency (using strings for references implies using a lot of space in secondary tables, where references often occur), logical keys have been phased out in favor of physical or surrogate keys:

**Physical Key** A single field that identifies an element in a unique way. For example, each person in the U.S. has a Social Security Number (SSN), but almost every country has a tax ID or other government-assigned number that identifies each person. The same is typically true for companies. Although these ID numbers are guaranteed to be unique, they can change depending on the country (creating troubles for the database of a company that sells goods abroad) or within a single country (to account for new tax laws). They are also often inefficient, because they can be quite large (Italy, for example, uses a 16-character code—letters and numbers—to identify people).

**Surrogate Key** A number identifying a record, in the form of a client code, order number, and so on. Surrogate keys are commonly used in database design. However, in many cases, they end up being *logical identifiers*, with client codes showing up all over the place (not a great idea).

**Warning** The situation becomes particularly troublesome when surrogate keys also have a meaning and must follow specific rules. For example, companies must number invoices with unique and consecutive numbers, without leaving holes in the numbering sequence. This situation is extremely complex to handle programmatically, if you consider that only the database can determine these unique consecutive numbers when you send it new data. At the same time, you need to identify the record before you send it to the database—otherwise you won't be able to fetch it again. Practical examples of how to solve this situation are discussed in [Chapter 15](#), "Working with ADO."

---

### OIDs to the Extreme

An extension to the use of surrogate keys is the use of a unique Object Identifier (OID). An OID is either a number or a string with a sequence of numbers and digits; it's added to each record of each table representing an entity (and sometimes to records of tables representing relations). Unlike client codes, invoice numbers, SSNs, or purchase order numbers, OIDs are random: They have no sequencing rule and are never visible to the end user. This means you can use surrogate keys (if your company is used to them) along with OIDs, but all the external references to the table will be based on OIDs.

Another common rule suggested by the promoters of this approach (which is part of the theories supporting object-relational mapping) is the use of system-wide unique identifiers. If you have a table of client companies and a table of employees, you may wonder why you should use a unique ID for such diverse data. The reason is that you'll be able to sell goods to an employee without having to duplicate the employee information in the customer table—you can refer to the employee in your order and invoice. An order is placed by someone identified by an OID, and this OID can refer to many different tables.

Using OIDs and object-relational mapping is an advanced element of the design of Delphi database applications. I suggest that you investigate this topic before embracing medium or large Delphi projects because the benefit can be relevant (after some investment in studying this approach and building some basic support code).

---

## External Keys and Referential Integrity

The keys identifying a record (whatever their type) can be used as external keys in other tables—for example, to represent the various types of relations discussed earlier. All SQL servers can verify these external references, so you cannot refer to a nonexistent record in another table. These referential integrity constraints are expressed when you create a table.

Besides not being allowed to add references to nonexistent records, you're generally prevented from deleting a record if external references to it exist. Some SQL servers go one step further: As you delete a record, instead of denying the operation, they can automatically delete all records that refer to it from other tables.

## More Constraints

In addition to the uniqueness of primary keys and the referential constraints, you can generally use the database to impose more validity rules on the data. You can ask for specific columns (such as those referring to a tax ID or a purchase order number) to include only unique values. You can impose uniqueness on the values of multiple columns—for example, to indicate that you cannot hold two classes in the same room at the same time.

In general, simple rules can be expressed to impose constraints on a table, whereas more complex rules generally imply the execution of stored procedures activated by triggers (every time the data changes, for instance, or there is new data).

Again, there is much more to proper database design, but the elements discussed in this section can provide you with a starting point or a good refresher.

**Note** For more information about SQL's Data Definition Language and Data Manipulation Language, see the chapter "[Essential SQL](#)" in the electronic book described in [Appendix C](#), "Free Companion Books on Delphi."

## Unidirectional Cursors

In local databases, tables are sequential files whose order either is the physical order or is defined by an index. By contrast, SQL servers work on logical sets of data that aren't related to a physical order. A *relational* database server handles data according to the relational model: a mathematical model based on set theory.

For this discussion, it's important for you to know that in a relational database, the records (sometimes called *tuples*) of a table are identified not by position but exclusively through a primary key, based on one or more fields. Once you've obtained a set of records, the server adds to each of them a reference to the following record; thus you can move quickly from a record to the following one, but moving back to the previous record is extremely slow. For this reason, it is common to say that an RDBMS uses a *unidirectional* cursor. Connecting such a table or query to a DBGrid control is practically impossible, because doing so would make browsing the grid backward terribly slow.

Some database engines keep the data already retrieved in a cache, to support full *bidirectional* navigation on it. In the Delphi architecture, this role can be played by the ClientDataSet component or another caching dataset. You'll see this process in more detail later, when we focus on dbExpress and the SQLDataset component.

**Note** The case of a DBGrid used to browse an entire table is common in local programs but should generally be avoided in a client/server environment. It's better to filter out only part of the records and only the fields you are interested in. If you need to see a list of names, return all those starting with the letter *A*, then those with *B*, and so on, or ask the user for the initial letter of the name.

If proceeding backward might result in problems, keep in mind that jumping to the last record of a table is even worse; usually this operation implies fetching all the records! A similar situation applies to the `RecordCount` property of datasets. Computing the number of records often implies moving them all to the client computer. For this reason, the thumb of the DBGrid's vertical scrollbar works for a local table but not for a remote table. If you need to know the number of records, run a separate query to let the server (and not the client) compute it. For example, you can see how many records will be selected from the EMPLOYEE table if you are interested in those records having a salary field higher than 50,000:

```
select count(*)  
from Employee  
where Salary > 50000
```

**Tip** Using the SQL instruction `count(*)` is a handy way to compute the number of records returned by a query. Instead of the `*` wildcard, you could use the name of a specific field, as in `count(First_Name)`, possibly combined with either `distinct` or `all`, to count only records with different values for the field or all the records having a non-*null* value.

Team LiB

◀ PREVIOUS

NEXT ▶



## Introducing InterBase

Although it has a limited market share, InterBase is a powerful RDBMS. In this section, I'll introduce the key technical features of InterBase without getting into too much detail (because this is a book about Delphi programming). Unfortunately, little is currently published about InterBase. Most of the available material is either in the documentation that accompanies the product or on a few websites devoted to it (your starting points for a search can be [www.borland.com/interbase](http://www.borland.com/interbase) and [www.ibphoenix.com](http://www.ibphoenix.com)).

InterBase was built from the beginning with a modern and robust architecture. Its original author, Jim Starkey, invented an architecture for handling concurrency and transactions without imposing physical locks on portions of the tables, something other well-known database servers can barely do even today. The InterBase architecture is called Multi-Generational Architecture (MGA); it handles concurrent access to the same data by multiple users, who can modify records without affecting what other concurrent users see in the database.

This approach naturally maps to the *Repeatable Read* transaction isolation mode, in which a user within a transaction keeps seeing the same data regardless of changes made and committed by other users. Technically, the server handles this situation by maintaining a different version of each accessed record for each open transaction. Even though this approach (also called *versioning*) can lead to larger memory consumption, it avoids most physical locks on the tables and makes the system much more robust in case of a crash. MGA also pushes toward a clear programming model—Repeatable Read—which other well-known SQL servers don't support without losing most of their performance.

In addition to the MGA at the heart of InterBase, the server has many other technical advantages:

- A limited footprint, which makes InterBase the ideal candidate for running directly on client computers, including portables. The disk space required by InterBase for a minimal installation is well below 10 MB, and its memory requirements are also incredibly limited.
- Good performance on large amounts of data.
- Availability on many different platforms (including 32-bit Windows, Solaris, and Linux), with totally compatible versions. Thus the server is scalable from very small to huge systems without notable differences.
- A very good track record, because InterBase has been in use for 15 years with few problems.
- A language complaint with the ANSI SQL standard.
- Advanced programming capabilities, including positional triggers, selectable stored procedures, updateable views, exceptions, events, generators, and more.
- Simple installation and management, with limited administrative headaches.

---

### A Short History of InterBase

Jim Starkey wrote InterBase for his Groton Database Systems company (hence the `.gds` extension still used for InterBase files). The company was later bought by Ashton-Tate, which was then acquired by Borland. Borland handled InterBase directly for a while and then created an InterBase subsidiary, which was later re-absorbed into the parent company.

Beginning with Delphi 1, an evaluation copy of InterBase has been distributed with the development tool, spreading the database server among developers. Although it doesn't have a large piece of the RDBMS market, which is dominated by a handful of players, InterBase has been chosen by a few relevant organizations, from Ericsson to the U.S. Department of Defense, from stock exchanges to home banking systems.

More recent events include the announcement of InterBase 6 as an open-source database (December 1999), the effective release of source code to the community (July 2000), and the release of the officially certified version of

InterBase 6 by Borland (March 2001). Between these events came announcements of the spin-off of a separate company to run the consulting and support business in addition to the open-source database. A group of former InterBase developers and managers (who had left Borland) formed IBPhoenix ([www.ibphoenix.com](http://www.ibphoenix.com)) with the plan of supporting InterBase users.

At the same time, independent groups of InterBase experts started the Firebird open-source project to further extend InterBase. The project is hosted on SourceForge at the address [sourceforge.net/projects/firebird/](http://sourceforge.net/projects/firebird/). For some time, SourceForge also hosted a Borland open-source project, but later the company announced it would continue to support only its proprietary version, dropping its open-source effort. So, the picture is now clearer. If you want a version with a traditional license (costing a fraction of most competing professional SQL servers), stick with Borland; but if you prefer an open-source, totally free model, go with the Firebird project (and eventually buy professional support from IBPhoenix).

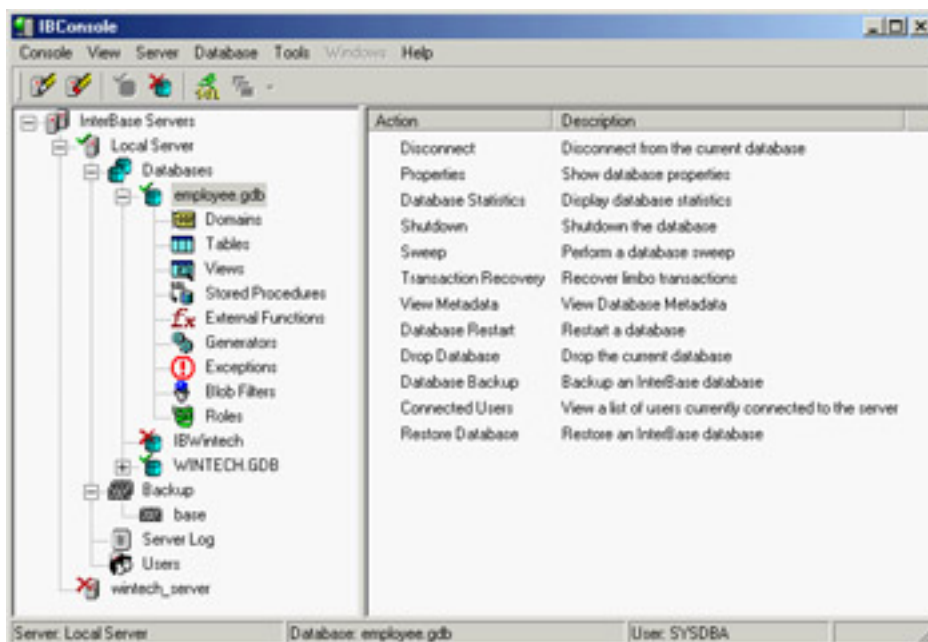
---

## Using IBConsole

In past versions of InterBase, you could use two primary tools to interact directly with the program: the Server Manager application, which could be used to administer both a local and a remote server; and Windows Interactive SQL (WISQL). Version 6 includes a much more powerful front-end application, called IBConsole. This full-fledged Windows program (built with Delphi) allows you to administer, configure, test, and query an InterBase server, whether local or remote.

IBConsole is a simple and complete system for managing InterBase servers and their databases. You can use it to look into the details of the database structure, modify it, query the data (which can be useful to develop the queries you want to embed in your program), back up and restore the database, and perform any other administrative tasks.

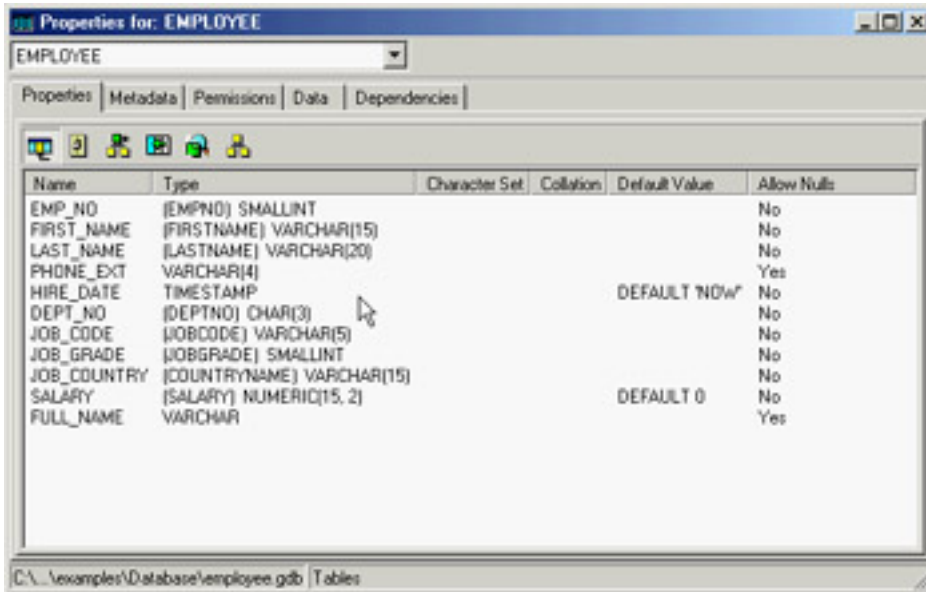
As you can see in [Figure 14.1](#), IBConsole allows you to manage multiple servers and databases, all listed in a single configuration tree. You can ask for general information about the database and list its entities (tables, domains, stored procedures, triggers, and everything else), accessing the details of each. You can also create new databases and configure them, back up the files, update the definitions, check what's going on and who is currently connected, and so on.



**Figure 14.1:** IBConsole lets you manage, from a single computer, InterBase databases hosted by multiple servers.

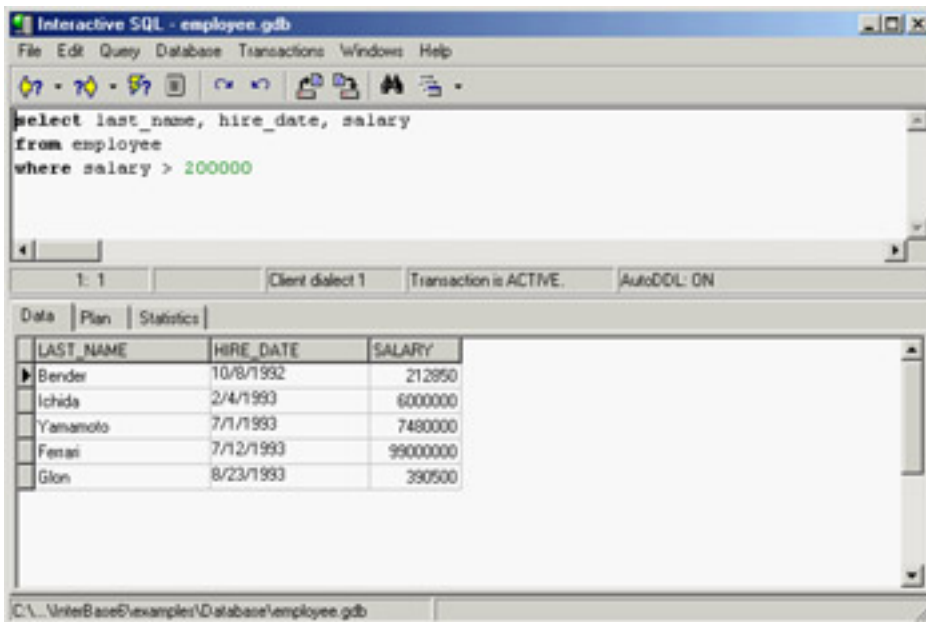
The IBConsole application allows you to open multiple windows to look at detailed information, such as the tables

window shown in [Figure 14.2](#). In this window, you can see lists of the key properties of each table (columns, triggers, constraints, and indexes), see the raw metadata (the SQL definition of the table), access permissions, look at the data, modify the data, and study the table's dependencies. Similar windows are available for each of the other entities you can define in a database.



**Figure 14.2:** IBConsole can open separate windows to show you the details of each entity—in this case, a table.

IBConsole embeds an improved version of the original Windows Interactive SQL application (see [Figure 14.3](#)). You can type a SQL statement in the upper portion of the window (without any help from the tool, unfortunately) and then execute the SQL query. As a result, you'll see the data, but also the access plan used by the database (which an expert can use to determine the efficiency of the query) and statistics about the operation performed by the server.



**Figure 14.3:** IBConsole's Interactive SQL window lets you try in advance the queries you plan to include in your Delphi programs.

This has been a minimal description of IBConsole, which is a powerful tool (and the only one Borland includes with the server other than command-line tools). IBConsole is not the most complete tool in its category, though. Quite a few third-party InterBase management applications are more powerful, although they are not all stable or user-friendly. Some InterBase tools are shareware programs, and others are free. Two examples out of many are InterBase Workbench ([www.upscene.com](http://www.upscene.com)) and IB\_WISQL (done with and part of InterBase Objects, [www.ibobjects.com](http://www.ibobjects.com)).

## InterBase Server-Side Programming

At the beginning of this chapter, I underlined the fact that one of the objectives of client/ server programming—and one of its problems—is the division of the workload between the computers involved. When you activate SQL statements from the client, the burden falls on the server to do most of the work. However, you should try to use `select` statements that return a large result set, to avoid jamming the network.

In addition to accepting DDL (Data Definition Language) and DML (Data Manipulation Language), most RDBMS servers allow you to create routines directly on the server using the standard SQL commands plus their own server-specific extensions (which generally are not portable). These routines typically come in two forms: stored procedures and triggers.

### Stored Procedures

Stored procedures are like the global functions of a Delphi unit and must be explicitly called by the client side. Stored procedures are generally used to define routines for data maintenance, to group sequences of operations you need in different circumstances, or to hold complex `select` statements.

Like Delphi procedures, stored procedures can have one or more typed parameters. Unlike Delphi procedures, they can have more than one return value. As an alternative to returning a value, a stored procedure can also return a result set—the result of an internal `select` statement or a custom fabricated one.

The following is a stored procedure written for InterBase; it receives a date as input and computes the highest salary among the employees hired on that date:

```
create procedure MaxSalOfTheDay (ofday date)
returns (maxsal decimal(8,2)) as
begin
  select max(salary)
  from employee
  where hiredate = :ofday
  into :maxsal;
end
```

Notice the use of the `into` clause, which tells the server to store the result of the `select` statement in the `maxsal` return value. To modify or delete a stored procedure, you can later use the `alter procedure` and `drop procedure` commands.

Looking at this stored procedure, you might wonder what its advantage is compared to the execution of a similar query activated from the client. The difference between the two approaches is not in the result you obtain but in its speed. A stored procedure is compiled on the server in an intermediate and faster notation when it is created, and the server determines at that time the strategy it will use to access the data. By contrast, a query is compiled every time the request is sent to the server. For this reason, a stored procedure can replace a very complex query, provided it doesn't change too often.

From Delphi, you can activate a stored procedure with the following SQL code:

```
select *
from MaxSalOfTheDay ('01/01/2003')
```

### Triggers (and Generators)

Triggers behave more or less like Delphi events and are automatically activated when a given event occurs. Triggers can have specific code or call stored procedures; in both cases, the execution is done completely on the server. Triggers are used to keep data consistent, checking new data in more complex ways than a check constraint allows, and to automate the side effects of some input operations (such as creating a log of previous salary changes when the current salary is modified).

Triggers can be fired by the three basic data update operations: `insert`, `update`, and `delete`. When you create a trigger, you indicate whether it should fire before or after one of these three actions.

As an example of a trigger, you can use a generator to create a unique index in a table. Many tables use a unique index as a primary key. InterBase doesn't have an `AutoInc` field. Because multiple clients cannot generate unique identifiers, you can rely on the server to do this. Almost all SQL servers offer a counter you can call to ask for a new ID, which you should later use for the table. InterBase calls these automatic counters *generators*, and Oracle calls them *sequences*. Here is the sample InterBase code:

```
create generator cust_no_gen;
...
gen_id (cust_no_gen, 1);
```

The `gen_id` function extracts the new unique value of the generator passed as the first parameter; the second parameter indicates how much to increase (in this case, by one).

At this point you can add a trigger to a table (an automatic handler for one of the table's events). A trigger is similar to the event handler of the Table component, but you write it in SQL and execute it on the server, not on the client. Here is an example:

```
create trigger set_cust_no for customers
before insert position 0 as
begin
  new.cust_no = gen_id (cust_no_gen, 1);
end
```

This trigger is defined for the `customers` table and is activated each time a new record is inserted. The `new` symbol indicates the new record you are inserting. The `position` option indicates the order of execution of multiple triggers connected to the same event. (Triggers with the lowest values are executed first.)

Inside a trigger, you can write DML statements that also update other tables, but watch out for updates that end up reactivating the trigger and create endless recursion. You can later modify or disable a trigger by calling the `alter trigger` or `drop trigger` statement.

Triggers fire automatically for specified events. If you have to make many changes in the database using batch operations, the presence of a trigger can slow the process. If the input data has already been checked for consistency, you can temporarily deactivate the trigger. These batch operations are often coded in stored procedures, but stored procedures generally cannot issue DDL statements like those required for deactivating and reactivating the trigger. In this situation, you can define a view based on a `select * from table` command, thus creating an alias for the table. Then you can let the stored procedure do the batch processing on the table and apply the trigger to the view (which should also be used by the client program).

## The dbExpress Library

Nowadays, the mainstream access to a SQL server database in Delphi is provided by the dbExpress library. As mentioned in [Chapter 13](#), "Delphi's Database Architecture," this is not the only possibility but is certainly the mainstream approach. The dbExpress library, first introduced in Kylix and Delphi 6, allows you to access different servers (InterBase, Oracle, DB2, MySQL, Informix, and now Microsoft SQL Server). I provided a general overview of dbExpress compared with other solutions in [Chapter 13](#), so here I'll skip the introductory material and focus on technical elements.

**Note** The inclusion of a driver for Microsoft SQL Server is the most important update to dbExpress provided by Delphi 7. It is not implemented by interfacing the vendor library natively, like other dbExpress drivers, but by interfacing Microsoft's OLEDB provider for SQL Server. (I'll talk more about OLEDB providers in [Chapter 15](#).)

### Working with Unidirectional Cursors

The motto of dbExpress could be "fetch but don't cache." The key difference between this library and BDE or ADO is that dbExpress can only execute SQL queries and fetch the results with a *unidirectional cursor*. As you've just seen, in unidirectional database access, you can move from one record to the next, but you cannot get back to a previous record of the dataset (unless by reopening the query and fetching all the records again minus one, an incredibly slow operation that dbExpress blocks). This is because the library doesn't store the data it has retrieved in a local cache, but only passes it from the database server to the calling application.

Using a unidirectional cursor might sound like a limitation, and it is—in addition to having problems with navigation, you cannot connect a database grid to a dataset. However, a unidirectional dataset is good for the following:

- You can use a unidirectional dataset for reporting purposes. In a printed report, but also an HTML page or an XML transformation, you move from record to record, produce the output, and that's it—no need to return to past records and, in general, no user interaction with the data. Unidirectional datasets are probably the best option for web and multitier architectures.
- You can use a unidirectional dataset to feed a local cache, such as the one provided by a ClientDataSet component. At this point, you can connect visual components to the in-memory dataset and operate on it with all the standard techniques, including the use of visual grids. You can freely navigate and edit the data in the in-memory cache, but also control it far better than with the BDE or ADO.

It's important to notice that in these circumstances, avoiding the caching of the database engine saves time and memory. The library doesn't have to use extra memory for the cache and doesn't need to waste time storing data (and duplicating information). Over the last couple of years, many programmers moved from BDE-based cached updates to the ClientDataSet component, which provides more flexibility in managing the content of the data and updating information they keep in memory. However, using a ClientDataSet on top of the BDE (or ADO) exposes you to the risk of having two separate caches, which wastes a lot of memory.

Another advantage of using the ClientDataSet component is that its cache supports editing operations, and the updates stored in this cache can be applied to the original database server by the DataSetProvider component. This component can generate the proper SQL `update` statements, and can do so in a more flexible way than the BDE (although ADO is powerful in this respect). In general, the provider can also use a dataset for the updates, but this isn't directly possible with the dbExpress dataset components.

### Platforms and Databases

A key element of the dbExpress library is its availability for both Windows and Linux, in contrast to the other database engines available for Delphi (BDE and ADO), which are available only for Windows. However, some of the database-specific components, such as InterBase Express, are also available on multiple platforms.

When you use dbExpress, you are provided with a common framework, which is independent of the SQL database



server you are planning to use. dbExpress comes with drivers for MySQL, InterBase, Oracle, Informix, Microsoft SQL Server, and IBM DB2.

**Note** It is possible to write custom drivers for the dbExpress architecture. This is documented in detail in the paper "dbExpress Draft Specification," published on the Borland Community website. At the time of this writing, this document is at <http://community.borland.com/article/0,1410,22495,00.html>. You'll probably be able to find third-party drivers. For example, there is a free driver that bridges dbExpress and ODBC. A complete list is hosted in the article at <http://community.borland.com/article/0,1410,28371,00.html>.

## Driver Versioning Troubles and Embedded Units

Technically, the dbExpress drivers are available as separate DLLs you have to deploy along with your program. This was the case with Delphi 6 and is still the case with Delphi 7. The problem is, these DLLs' names haven't changed. So, if you install a Delphi 7 compiled application on a machine that has the dbExpress drivers found in Delphi 6, the application will apparently work, open a connection to the server, and then fail when retrieving data. At that point you'll see the error "SQL Error: Error mapping failed." This is not a good hint that there is a version mismatch in the dbExpress driver!

To verify this problem, look at whether the DLL has any version information—it was missing from the Delphi 6 drivers. To make your applications more robust, you can provide a similar check within your code, accessing the version information using the related Windows APIs:

```
function GetDriverVersion (strDriverName: string): Integer;
var
    nInfoSize, nDetSize: DWord;
    pVInfo, pDetail: Pointer;
begin
    // the default, in case there is no version information
    Result := 6;

    // read version information
    nInfoSize := GetFileVersionInfoSize (pChar(strDriverName), nDetSize);
    if nInfoSize > 0 then
        begin
            GetMem (pVInfo, nInfoSize);
            try
                GetFileVersionInfo (pChar(strDriverName), 0, nInfoSize, pVInfo);
                VerQueryValue (pVInfo, '\', pDetail, nDetSize);
                Result := HiWord (TVSFixedFileInfo(pDetail^).dwFileVersionMS);
            finally
                FreeMem (pVInfo);
            end;
        end;
    end;
end;
```

This code snippet is taken from the DbxMulti example discussed later. The program uses it to raise an exception if the version is incompatible:

```
if GetDriverVersion ('dbexpint.dll') <> 7 then
    raise Exception.Create (
        'Incompatible version of the dbExpress driver "dbexpint.dll" found');
```

If you try to put the driver found in Delphi 6's bin folder in the application folder, you'll see the error. You'll have to modify this extra safety check to account for updated versions of the drivers or libraries, but this step should help you avoid the installation troubles dbExpress meant to solve in the first place.

You also have an alternative: You can statically link the dbExpress drivers' code into your application. To do so, include

a given unit (like `dbexpint.dcu` or `dbexpora.dcu`) in your program, listing it in one of the `uses` statements.

**Warning** Along with one of these units you need to include the `MidasLib` unit and link the code of the `MIDAS.DLL` into your program. If you fail to do so, the linker of Delphi 7 will stop showing an internal error, which is rather pointless information. Notice also that the embedded `dbExpress` drivers don't work properly with the international character set.

Team LiB

◀ PREVIOUS

NEXT ▶



## The dbExpress Components

The VCL components used to interface the dbExpress library encompass a group of dataset components plus a few ancillary ones. To differentiate these components from other database-access families, the components are prefixed with the letters *SQL*, underlining the fact that they are used for accessing RDBMS servers.

These components include a database connection component, a few dataset components (a generic one; three specific versions for tables, queries, and stored procedures; and one encapsulating a ClientDataSet component), and a monitor utility.

## The SQLConnection Component

The `TSQLConnection` class inherits from the `TCustomConnection` component. It handles database connections, the same as its sibling classes (the `Database`, `ADOConnection`, and `IBConnection` components).

**Tip** Unlike other component families, in dbExpress the connection is compulsory. In the dataset components, you cannot specify directly which database to use, but can only refer to a `SQLConnection`.

The connection component uses the information available in the `drivers.ini` and `connections.ini` files, which are dbExpress's only two configuration files (these files are saved by default under `Common Files\Borland Shared\DBExpress`). The `drivers.ini` file lists the available dbExpress drivers, one for each supported database. For each driver there is a set of default connection parameters. For example, the `InterBase` section reads as follows:

```
[Interbase]
GetDriverFunc=getSQLDriverINTERBASE
LibraryName=dbexpint.dll
VendorLib=GDS32.DLL
BlobSize=-1
CommitRetain=False
Database=database.gdb
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
SQLDialect=1
Interbase TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

The parameters indicate the dbExpress driver DLL (the `LibraryName` value), the entry function to use (`GetDriverFunc`), the vendor client library, and other specific parameters that depend on the database. If you read the entire `drivers.ini` file, you'll see that the parameters are really database-specific. Some of these parameters don't make a lot of sense at the driver level (such as the database to connect to), but the list includes all the available parameters, regardless of their usage.

The `connections.ini` file provides the database-specific description. This list associates settings with a name, and you can enter multiple connection details for every database driver. The connection describes the physical database you want to connect to. As an example, this is the portion for the default `IBLocal` definition:

```
[IBLocal]
BlobSize=-1
CommitRetain=False
Database=C:\Program Files\Common Files\Borland Shared\Data\employee.gdb
DriverName=Interbase
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
```

```
SQLDialect=1
Interbase TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

As you can see by comparing the two listings, this is a subset of the driver's parameters. When you create a new connection, the system will copy the default parameters from the driver; you can then edit them for the specific connection—for example, providing a proper database name. Each connection relates to the driver for its key attributes, as indicated by the `DriverName` property. Notice also that the database referenced here is the result of my editing, corresponding to the settings I'll use in most examples.

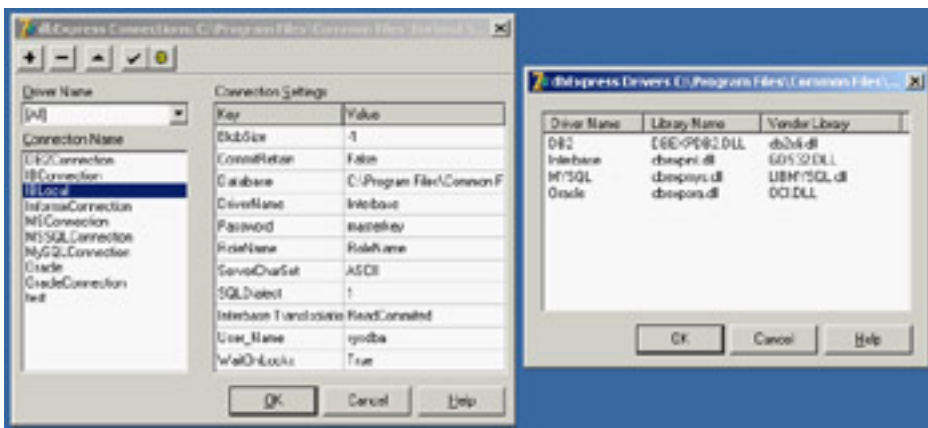
It's important to remember that these initialization files are used only at design time. When you select a driver or a connection at design time, the values of these files are copied to corresponding properties of the `SQLConnection` component, as in this example:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  DriverName = 'Interbase'
  GetDriverFunc = 'getSQLDriverINTERBASE'
  LibraryName = 'dbexpint.dll'
  LoginPrompt = False
  Params.Strings = (
    'BlobSize=-1'
    'CommitRetain=False'
    'Database=C:\Program Files\Common Files\Borland Shared\Data\employee.gdb'
    'DriverName=Interbase'
    'Password=masterkey'
    'RoleName=RoleName'
    'ServerCharSet=ASCII'
    'SQLDialect=1'
    'Interbase TransIsolation=ReadCommitted'
    'User_Name=sysdba'
    'WaitOnLocks=True' )
  VendorLib = 'GDS32.DLL'
end
```

At run time, your program will rely on the properties to have all the required information, so you don't need to deploy the two configuration files along with your programs. In theory, the files will be required if you want to change the `DriverName` or `ConnectionName` properties at run time. However, if you want to connect your program to a new database, you can set the relevant properties directly.

When you add a new `SQLConnection` component to an application, you can proceed in different ways. You can set up a driver using the list of values available for the `DriverName` property and then select a predefined connection by selecting one of the values available in the `ConnectionName` property. This second list is filtered according to the driver you've already selected. As an alternative, you can begin by selecting the `ConnectionName` property directly; in this case it includes the entire list.

Instead of hooking up an existing connection, you can define a new one (or see the details of the existing connections) by double-clicking the `SQLConnection` component and launching the dbExpress Connection Editor (see [Figure 14.4](#)). This editor lists on the left all the predefined connections (for a specific driver or all of them) and allows you to edit the connection properties using the grid on the right. You can use the toolbar buttons to add, delete, rename, and test connections, and to open the read-only dbExpress Drivers Settings window (also shown in [Figure 14.4](#)).



**Figure 14.4:** The dbExpress Connection Editor with the dbExpress Drivers Settings dialog box

In addition to letting you edit the predefined connection settings, the dbExpress Connection Editor allows you to select a connection for the SQLConnection component by clicking the OK button. Note that if you change any settings, the data is immediately written to the configuration files—clicking the Cancel button doesn't undo your editing!

To define access to a database, editing the connection properties is certainly the suggested approach. This way, when you need to access the same database from another application or another connection within the same application, all you need to do is select the connection. However, because this operation copies the connection data, updating the connection doesn't automatically refresh the values within other SQLConnection components referring to the same named connection: You must reselect the connection to which these other components refer.

What really matters for the SQLConnection component is the value of its properties. Driver and vendor libraries are listed in properties you can freely change at design time (although you'll rarely want to do this), whereas the database and other database-specific connection settings are specified in the `Params` properties. This is a string list including information such as the database name, the username and password, and so on. In practice, you could set up a SQLConnection component by setting up the driver and then assigning the database name directly in the `Params` property, forgetting about the predefined connection. I'm not suggesting this as the best option, but it is certainly a possibility; the predefined connections are handy, but when the data changes, you still have to manually refresh every SQLConnection component.

To be complete, I have to mention that there is an alternative. You can set the `LoadParamsOnConnect` property to indicate that you want to refresh the component parameters from the initialization files every time you open the connection. In this case, a change in the predefined connections will be reloaded when you open the connection, at either design time or run time. At design time, this technique is handy (it has the same effect as reselecting the connection); but using it at run time means you'll also have to deploy the `connections.ini` file, which can be a good idea or inconvenient, depending on your deployment environment.

The only property of the SQLConnection component that is not related to the driver and database settings is `LoginPrompt`. Setting it to `False` allows you to provide a password among the component settings and skip the login request dialog box, both at design time and at run time. Although this is handy for development, it can reduce the security of your system. Of course, you should also use this option for unattended connections, such as on a web server.

## The dbExpress Dataset Components

The dbExpress component's family provides four different dataset components: a generic dataset, a table, a query, and a stored procedure. The latter three components are provided for compatibility with the equivalent BDE components and have similarly named properties. If you don't have to port existing code, you should generally use the general `SQLDataSet` component, which lets you execute a query and also access a table or a stored procedure.

The first important thing to notice is that all these datasets inherit from a new special base class, `TCustomSQLDataSet`. This class and its derived classes represent unidirectional datasets, with the key features I've already described. In practice, this means that the browse operations are limited to calling `First` and `Next; Prior`,

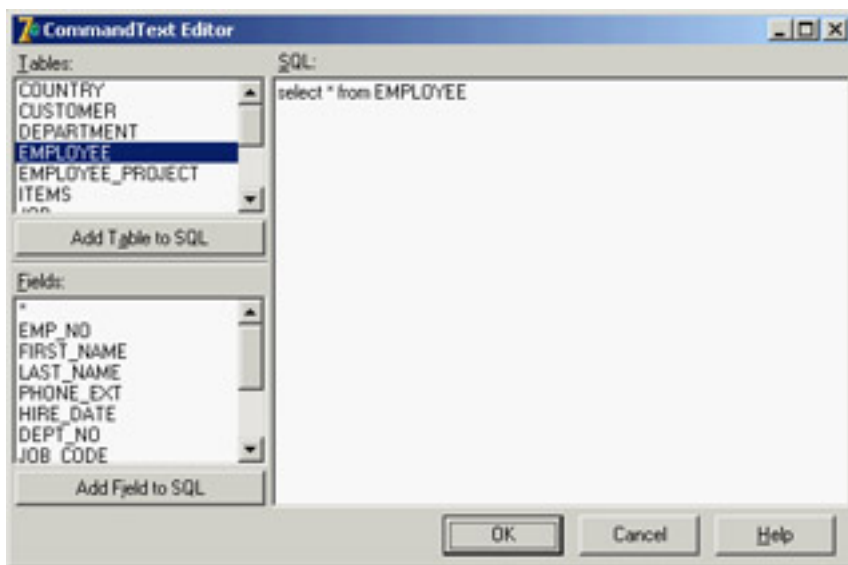
Last, Locate, the use of bookmarks, and all other navigational features are disabled.

**Note** Technically, some of the moving operations call the *CheckBiDirectional* internal function and eventually raise an exception. *CheckBiDirectional* refers to the public *IsUnidirectional* property of the *TDataSet* class, which you can eventually use in your own code to disable operations that are illegal on unidirectional datasets.

In addition to having limited navigational capabilities, these datasets have no editing support, so a lot of methods and events common to other datasets are not available. For example, there is no *AfterEdit* or *BeforePost* event.

As I mentioned earlier, of the four dataset components for dbExpress, the fundamental one is *TSQLDataSet*, which can be used both to retrieve a dataset and to execute a command. The two alternatives are activated by calling the *Open* method (or setting the *Active* property to *True*) and by calling the *ExecSQL* method.

The *SQLDataSet* component can retrieve an entire table, or it can use a SQL query or a stored procedure to read a dataset or issue a command. The *CommandType* property determines one of the three access modes. The possible values are *ctQuery*, *ctStoredProc*, and *ctTable*, which determine the value of the *CommandText* property (and also the behavior of the related property editor in the Object Inspector). For a table or stored procedure, the *CommandText* property indicates the name of the related database element, and the editor provides a drop-down list containing the possible values. For a query, the *CommandText* property stores the text of the SQL command, and the editor provides a little help in building the SQL query (in case it is a *SELECT* statement). You can see the editor in [Figure 14.5](#).



**Figure 14.5:** The *CommandText* Editor used by the *SQLDataSet* component for queries

When you use a table, the component will generate a SQL query for you, because dbExpress targets only SQL databases. The generated query will include all the fields of the table, and if you specify the *SortFieldNames* property, it will include a *sort by* directive.

The three *specific* dataset components offer similar behavior, but you specify the SQL query in the *SQL* string list property, the stored procedure in the *StoredProcName* property, and the table name in the *TableName* property (as in the three corresponding BDE components).

## The Delphi 7 SimpleDataSet Component

The *SimpleDataSet* component is new in Delphi 7. It is a combination of four existing components: *SQLConnection*, *SQLDataSet*, *DataSetProvider*, and *ClientDataSet*. The component is meant to be a helper—you need only one component instead of four (which must also be connected). The component is basically a client dataset with two compound components (the two dbExpress ones), plus a hidden provider. (The fact that the provider is hidden is odd, because it is created as a compound component.)

The component allows you to modify the properties and events of the compound components (besides the provider) and replace the internal connection with an external one, so that multiple datasets share the same database connection. In addition to this issue, the component has other limitations, including difficulty manipulating the dataset fields of the data access dataset (which is important for setting key fields and can affect the way updates are generated) and unavailability of some provider events. So, other than for simple applications, I don't recommend using the SimpleDataSet component.

**Note** Delphi 6 shipped with an even simpler and more limited component called SQLClientDataSet. Similar components were available for the BDE and IBX data access technologies. Now Borland has indicated that all these components are obsolete. However, the directory *Demos\Db\SQLClientDataSet* contains a copy of the original component, and you can install it in Delphi 7 for compatibility purposes. But just as the SimpleDataSet component is somewhat limited, I found the SQLClientDataSet component totally unusable.

## The SQLMonitor Component

The final component in the dbExpress group is SQLMonitor, which is used to log requests sent from dbExpress to the database server. This monitor lets you see the commands sent to the database and the low-level responses you receive, monitoring the client/server traffic at a low level.

---

### The TimeStamp Field Type

Along with dbExpress, Delphi 6 introduced the `TSQLTimeStampField` field type, which is mapped to the time stamp data type that many SQL servers have (InterBase included). A *time stamp* is a record-based representation of a date or time, and it's quite different from the floating-point representation used by the `TDateTime` data type. A time stamp is defined as follows:

```
TSQLTimeStamp = packed record
  Year : SmallInt;
  Month : Word;
  Day : Word;
  Hour : Word;
  Minute : Word;
  Second : Word;
  Fractions : LongWord;
end;
```

A time stamp field can automatically convert standard date and time values using the `AsDateTime` property (as opposed to the native `AsSQLTimeStamp` property). You can also do custom conversions and further manipulation of time stamps using the routines provided by the `SqITimSt` unit, including functions like `DateTimeToSQLTimeStamp`, `SQLTimeStampToStr`, and `VarSQLTimeStampCreate`.

---

## A Few dbExpress Demos

Let's look at a demonstration that highlights the key features of these components and shows how to use the `ClientDataSet` to provide caching and editing support for the unidirectional datasets. Later, I'll show you an example of native use of the unidirectional query, with no caching and editing support required.

The standard visual application based on dbExpress uses this series of components:

- The `SQLConnection` component provides the connection with the database and the proper dbExpress driver.
- The `SQLDataSet` component, which is hooked to the connection (via the `SQLConnection` property), indicates which SQL query to execute or table to open (using the `CommandType` and `CommandText` properties discussed earlier).
- The `DataSetProvider` component, connected with the dataset, extracts the data from the `SQLDataSet` and can generate the proper SQL update statements.
- The `ClientDataSet` component reads from the data provider and stores all the data (if its `PacketRecords` property is set to `-1`) in memory. You'll need to call its `ApplyUpdates` method to send the updates back to the database server (through the provider).
- The `DataSource` component allows you to surface the data from the `ClientDataSet` to the visual data-aware controls.

As I mentioned earlier, the picture can be simplified by using the `SimpleDataSet` component, which replaces the two datasets and the provider (and possibly even the connection). The `SimpleDataSet` component combines most of the properties of the components it replaces.

## Using a Single Component or Many Components

For this first example, drop a `SimpleDataSet` component on a form and set the connection name of its `Connection` subcomponent. Set the `CommandType` and `CommandText` properties to specify which data to fetch, and set the `PacketRecords` property to indicate how many records to retrieve in each block.

These are the key properties of the component in the `DbxSingle` example:

```
object SimpleDataSet1: TSimpleDataSet
  Connection.ConnectionName = 'IBLocal'
  Connection.LoginPrompt = False
  DataSet.CommandText = 'EMPLOYEE'
  DataSet.CommandType = ctTable
end
```

As an alternative, the `DbxMulti` example uses the entire sequence of components:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText = 'select * from EMPLOYEE'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
object ClientDataSet1: TClientDataSet
```



```

    ProviderName = 'DataSetProvider1'
end
object DataSource1: TDataSource
    DataSet = ClientDataSet1
end

```

Both examples include some visual controls: a grid and a toolbar based on the action manager architecture.

## Applying Updates

In every example based on a local cache, like the one provided by the ClientDataSet and SimpleDataSet components, it's important to write the local changes back to the database server. This is typically accomplished by calling the `ApplyUpdates` method. You can either keep the changes in the local cache for a while and then apply multiple updates at once, or you can post each change right away. In these two examples, I've gone for the latter approach, attaching the following event handler to the `AfterPost` (fired after an edit or an insert operation) and `AfterDelete` events of the ClientDataSet components:

```

procedure TForm1.DoUpdate(DataSet: TDataSet);
begin
    // immediately apply local changes to the database
    SQLClientDataSet1.ApplyUpdates(0);
end;

```

If you want to apply all the updates in a single batch, you can do so either when the form is closed or when the program ends, or you can let a user perform the update operation by selecting a specific command, possibly using the corresponding predefined action provided by Delphi 7. We'll explore this approach when discussing the update caching support of the ClientDataSet component in more detail later in this chapter.

## Monitoring the Connection

Another feature I've added to the DbxSingle and DbxMulti examples is the monitoring capability offered by the SQLMonitor component. In the example, the component is activated as the program starts. In the DbxSingle example, because the SimpleDataSet embeds the connection, the monitor cannot be hooked to it at design time, but only when the program starts:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    SQLMonitor1.SQLConnection := SimpleDataSet1.Connection;
    SQLMonitor1.Active := True;
    SimpleDataSet1.Active := True;
end;

```

Every time a tracing string is available, the component fires the `OnTrace` event to let you choose whether to include the string in the log. If the `LogTrace` parameter of this event is `True` (the default value), the component logs the message in the `TraceList` string list and fires the `OnLogTrace` event to indicate that a new string has been added to the log.

The component can also automatically store the log into the file indicated by its `FileName` property, but I haven't used this feature in the example. All I've done is handle the `OnTrace` event, copying the entire log in the memo with the following code (producing the output shown in [Figure 14.6](#)):

```

procedure TForm1.SQLMonitor1Trace(Sender: TObject;
    CBIInfo: pSQLTRACEDesc; var LogTrace: Boolean);
begin
    Mem1.Lines := SQLMonitor1.TraceList;
end;

```

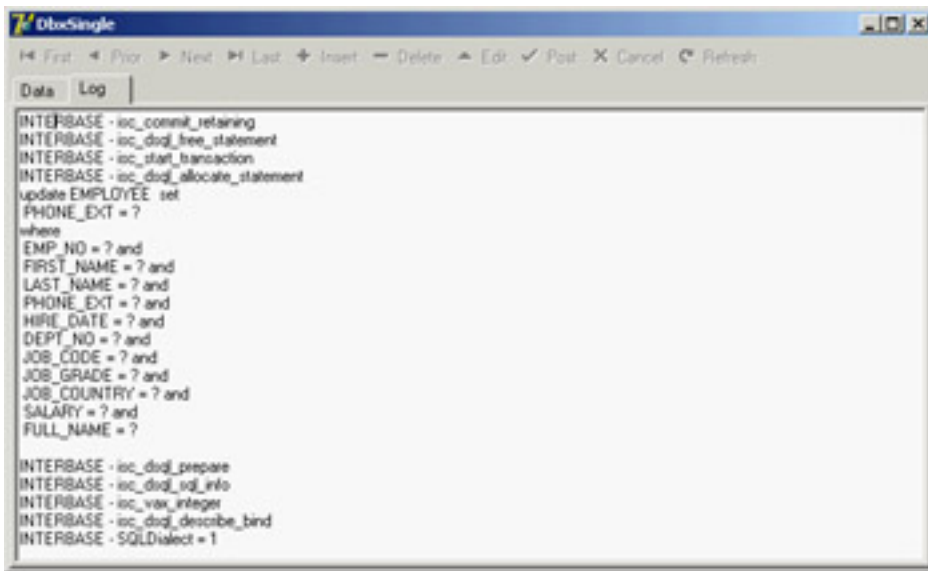


Figure 14.6: A sample log obtained by the SQLMonitor in the DbxSingle example

## Controlling the SQL Update Code

If you run the DbxSingle program and change, for example, an employee's telephone number, the monitor will log this update operation:

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  FIRST_NAME = ? and
  LAST_NAME = ? and
  PHONE_EXT = ? and
  HIRE_DATE = ? and
  DEPT_NO = ? and
  JOB_CODE = ? and
  JOB_GRADE = ? and
  JOB_COUNTRY = ? and
  SALARY = ? and
  FULL_NAME = ?
```

By setting the SimpleDataSet's properties there is no way to change how the update code is generated (which happens to be worse than with the SQLClientDataSet component, which had the UpdateMode you could use to tweak the update statements).

In the DbxMulti example, you can use the UpdateMode property of the DataSetProvider component, setting the value to upWhereChanged or upWhereKeyOnly. In this case you'll get the following two statements, respectively:

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  PHONE_EXT = ?

update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ?
```

**Tip** This result is much better than in Delphi 6 (without the patches applied), in which this operation caused an error because the key field was not properly set.



If you want more control over how the update statements are generated, you need to operate on the fields of the underlying dataset, which are available also when you use the all-in-one SimpleDataSet component (which has two field editors, one for the base ClientDataSet component it inherits from and one for the SQLDataSet component it embeds). I have made similar corrections in the DbxMulti example, after adding persistent fields for the SQLDataSet component and modifying the provider options for some of the fields to include them in the key or exclude them from updates.

**Note** We'll discuss this type of problem again when we examine the details of the ClientDataSet component, the provider, the resolver, and other technical details later in this chapter and in [Chapter 16](#).

## Accessing Database Metadata with *SetSchemaInfo*

All RDBMS systems use special-purpose tables (generally called *system tables*) for storing metadata, such as the list of the tables, their fields, indexes, and constraints, and any other system information. Just as dbExpress provides a unified API for working with different SQL servers, it also provides a common way to access metadata. The SQLDataSet component has a *SetSchemaInfo* method that fills the dataset with system information. This *SetSchemaInfo* method has three parameters:

*SchemaType* Indicates the type of information requested. Values include *stTables*, *stSysTables*, *stProcedures*, *stColumns*, and *stProcedureParams*.

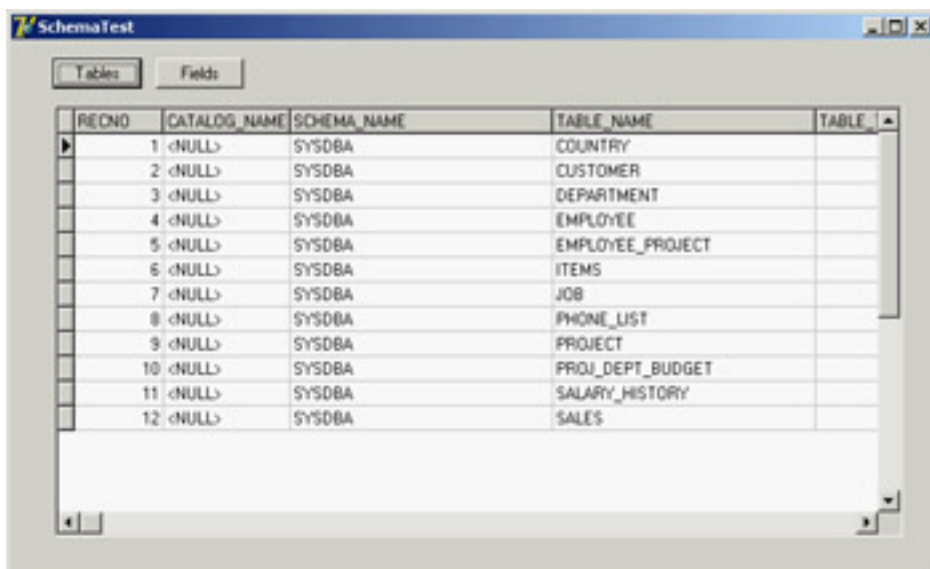
*SchemaObject* Indicates the object you are referring to, such as the name of the table whose columns you are requesting.

*SchemaPattern* A filter that lets you limit your request to tables, columns, or procedures starting with the given letters. This is handy if you use prefixes to identify groups of elements.

For example, in the SchemaTest program, a Tables button reads into the dataset all of the connected database's tables:

```
ClientDataSet1.Close;  
SQLDataSet1.SetSchemaInfo (stTables, '', '');  
ClientDataSet1.Open;
```

The program uses the usual group of dataset provider, client dataset, and data source component to display the resulting data in a grid, as you can see in [Figure 14.7](#). After you're retrieved the tables, you can select a row in the grid and click the Fields button to see a list of the fields of this table:



The screenshot shows a window titled "SchemaTest" with two buttons: "Tables" and "Fields". Below the buttons is a grid displaying the following data:

RECNO	CATALOG_NAME	SCHEMA_NAME	TABLE_NAME	TABLE
1	<NULL>	SYSDBA	COUNTRY	
2	<NULL>	SYSDBA	CUSTOMER	
3	<NULL>	SYSDBA	DEPARTMENT	
4	<NULL>	SYSDBA	EMPLOYEE	
5	<NULL>	SYSDBA	EMPLOYEE_PROJECT	
6	<NULL>	SYSDBA	ITEMS	
7	<NULL>	SYSDBA	JOB	
8	<NULL>	SYSDBA	PHONE_LIST	
9	<NULL>	SYSDBA	PROJECT	
10	<NULL>	SYSDBA	PROJ_DEPT_BUDGET	
11	<NULL>	SYSDBA	SALARY_HISTORY	
12	<NULL>	SYSDBA	SALES	

**Figure 14.7:** The SchemaTest example allows you to see a database's tables and the columns of a given table.

```
SQLDataSet1.SetSchemaInfo (stColumns, ClientDataSet1['Table_Name'], '');  
ClientDataSet1.Close;  
ClientDataSet1.Open;
```

In addition to letting you access database metadata, dbExpress provides a way to access its own configuration information, including the installed drivers and the configured connections. The unit `DbConnAdmin` defines a `TConnectionAdmin` class for this purpose, but the aim of this support is limited to dbExpress add-on utilities for developers (end users aren't commonly allowed to access multiple databases in a totally dynamic way).

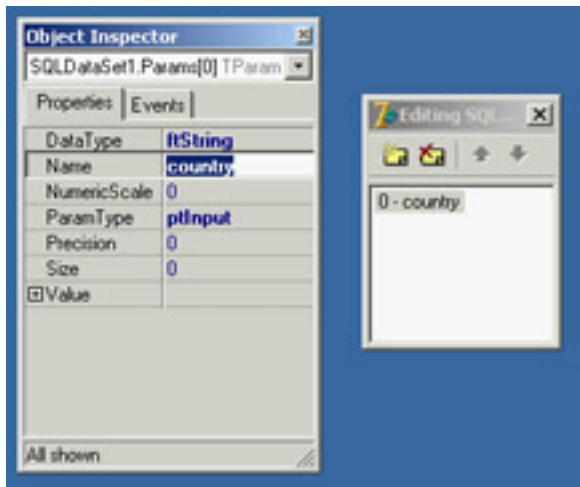
**Tip** The `DbxExplorer` demo included in Delphi shows how to access both dbExpress administration files and schema information. Also check the help file under "The structure of metadata datasets" within the section "Developing database applications."

## A Parametric Query

When you need slightly different versions of the same SQL query, instead of modifying the text of the query itself each time, you can write a query with a parameter and change the value of the parameter. For example, if you decide to have a user choose the employees in a given country (using the `employee` table), you can write the following parametric query:

```
select *  
from employee  
where job_country = :country
```

In this SQL clause, `:country` is a parameter. You can set its data type and startup value using the editor of the `SQLDataSet` component's `Params` property collection. When the `Params` collection editor is open, as shown in [Figure 14.8](#), you see a list of the parameters defined in the SQL statement; you can set the data type and the initial value of these parameters in the Object Inspector.



**Figure 14.8:** Editing a query component's collection of parameters

The form displayed by this program, called `ParQuery`, uses a combo box to provide all the available values for the parameters. Instead of preparing the combo box items at design time, you can extract the available contents from the same database table as the program starts. This is accomplished using a second query component, with this SQL statement:

```
select distinct job_country  
from employee
```

After activating this query, the program scans its result set, extracting all the values and adding them to the list box:

```

procedure TQueryForm.FormCreate(Sender: TObject);
begin
  SqlDataSet2.Open;
  while not SqlDataSet2.EOF do
  begin
    ComboBox1.Items.Add (SqlDataSet2.Fields [0].AsString);
    SqlDataSet2.Next;
  end;
  ComboBox1.Text := ComboBox1.Items[0];
end;

```

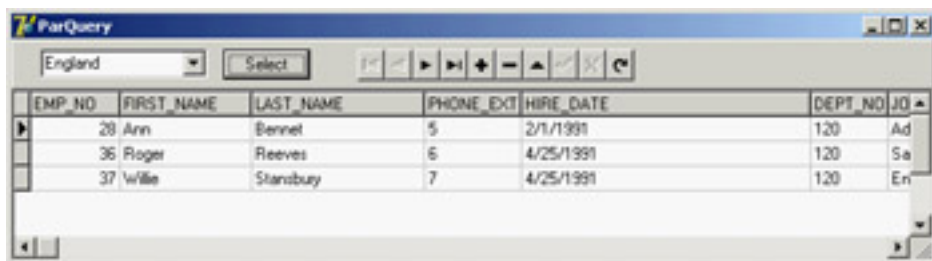
The user can select a different item in the combo box and then click the Select button (Button1) to change the parameter and activate (or re-activate) the query:

```

procedure TQueryForm.Button1Click(Sender: TObject);
begin
  SqlDataSet1.Close;
  ClientDataSet1.Close;
  Query1.Params[0].Value := ListBox1.Items [ListBox1.ItemIndex];
  SqlDataSet1.Open;
  ClientDataSet1.Open;
end;

```

This code displays the employees from the selected country in the DBGrid, as you can see in [Figure 14.9](#). As an alternative to using the elements of the Params array by position, you should consider using the ParamByName method, to avoid any problem in case the query gets modified over time and the parameters end up in a different order.



**Figure 14.9:** The ParQuery example at run time

By using parametric queries, you can usually reduce the amount of data moved over the wire from the server to the client and still use a DBGrid and the standard user interface common in local database applications.

**Tip** Parametric queries are generally also used to obtain master-detail architectures with SQL queries—at least, this is what Delphi tends to do. The *DataSource* property of the SQLDataSet component, automatically replaces parameter values with the fields of the master dataset having the same name as the parameter.

## When One-Way Is Enough: Printing Data

You have seen that one of the key elements of the dbExpress library is that it returns unidirectional datasets. In addition, you can use the ClientDataSet component (in one of its incarnations) to store the records in a local cache. Now, let's discuss an example in which a unidirectional dataset is all you need.

Such a situation is common in *reporting*—that is, producing information for each record in sequence without needing any further access to the data. This broad category includes producing printed reports (via a set of reporting components or using the printer directly), sending data to another application such as Microsoft Excel or Word, saving data to files (including HTML and XML formats), and more.

I don't want to delve into HTML and XML, so I'll present an example of printing—nothing fancy and nothing based on reporting components, just a way to produce a draft report on your monitor and printer. For this reason, I've used

Delphi's most straightforward technique to produce a printout: assigning a file to the printer with the `AssignPrn` RTL procedure.

The example, called `UniPrint`, has a unidirectional `SQLDataSet` component hooked to an `InterBase` connection and based on the following SQL statement, which joins the employee table with the department table to display the name of the department where each employee works:

```
select d.DEPARTMENT, e.FULL_NAME, e.JOB_COUNTRY, e.HIRE_DATE
from EMPLOYEE e
inner join DEPARTMENT d on d.DEPT_NO = e.DEPT_NO
```

To handle printing, I've written a somewhat generic routine, requiring as parameters the data to print, a progress bar for status information, the output font, and the maximum format size of each field. The entire routine uses file-print support and formats each field in a fixed-size, left-aligned string, to produce a columnar type of report. The call to the `Format` function has a parametric format string that's built dynamically using the size of the field.

In [Listing 14.1](#) you can see the code of the core `PrintOutDataSet` method, which uses three nested `try/finally` blocks to release all the resources properly:

### Listing 14.1: The Core Method of the UniPrint Example

---

```
procedure PrintOutDataSet (data: TDataSet;
    progress: TProgressBar; Font: TFont; toFile: Boolean; maxSize: Integer = 30);
var
    PrintFile: TextFile;
    I: Integer;
    sizeStr: string;
    oldFont: TFontRecall;
begin
    // assign the output to a printer or a file
    if toFile then
    begin
        SelectDirectory ('Choose a folder', '', strDir);
        AssignFile (PrintFile,
            IncludeTrailingPathDelimiter(strDir) + 'output.txt');
    end
    else
        AssignPrn (PrintFile);
    // assign the printer to a file
    AssignPrn (PrintFile);
    Rewrite (PrintFile);

    // set the font and keep the original one
    oldFont := TFontRecall.Create (Printer.Canvas.Font);
    try
        Printer.Canvas.Font := Font;
        try
            data.Open;
            try
                // print header (field names) in bold
                Printer.Canvas.Font.Style := [fsBold];
                for I := 0 to data.FieldCount - 1 do
                begin
                    sizeStr := IntToStr (min (data.Fields[I].DisplayWidth, maxSize));
                    Write (PrintFile, Format ('%-' + sizeStr + 's',
                        [data.Fields[I].FieldName]));
                end;
                Writeln (PrintFile);
            end;
        end;
    end;
```

```

// for each record of the dataset
Printer.Canvas.Font.Style := [];
while not data.EOF do
begin
  // print out each field of the record
  for I := 0 to data.FieldCount - 1 do
  begin
    sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
    Write (PrintFile, Format ('%-' + sizeStr + 's',
      [data.Fields[i].AsString]));
  end;
  Writeln (PrintFile);
  // advance ProgressBar
  progress.Position := progress.Position + 1;
  data.Next;
end;
finally
  // close the dataset
  data.Close;
end;
finally
  // reassign the original printer font
  oldFont.Free;
end;
finally
  // close the printer/file
  CloseFile (PrintFile);
end;
end;

```

---

The program invokes this routine when you click the Print All button. It executes a separate query (`select count(*) from EMPLOYEE`), which returns the number of records in the employee table. This query is necessary to set up the progress bar (the unidirectional dataset has no way of knowing how many records it will retrieve until it has reached the last one). Then it sets the output font, possibly using a fixed-width font, and calls the `PrintOutDataSet` routine:

```

procedure TNavigator.PrintAllButtonClick(Sender: TObject);
var
  Font: TFont;
begin
  // set ProgressBar range
  EmplCountData.Open;
  try
    ProgressBar1.Max := EmplCountData.Fields[0].AsInteger;
  finally
    EmplCountData.Close;
  end;

  Font := TFont.Create;
  try
    Font.Name := 'Courier New';
    Font.Size := 9;
    PrintOutDataSet (EmplData, ProgressBar1, Font, cbFile.Checked);
  finally
    Font.Free;
  end;
end;

```



## The Packets and the Cache

The `ClientDataSet` component reads data in packets containing the number of records indicated by the `PacketRecords` property. The default value of this property is `-1`, which means the provider will pull all the records at once (this is reasonable only for a small dataset). Alternatively, you can set this value to zero to ask the server for only the field descriptors and no data, or you can use any positive value to specify a number.

If you retrieve only a partial dataset, then as you browse past the end of the local cache, if the `FetchOnDemand` property is set to `True` (the default value), the `ClientDataSet` component will get more records from its source. This property also controls whether BLOB fields and nested datasets of the current records are fetched automatically (these values might not be part of the data packet, depending on the dataset provider's `Options` value).

If you turn off this property, you'll need to fetch more records manually by calling the `GetNextPacket` method until the method returns zero. (You call `FetchBlobs` and `FetchDetails` for these other elements.)

**Warning** Notice, by the way, that before you set an index for the data, you should retrieve the entire dataset (either by going to its last record or by setting the `PacketRecords` property to `-1`). Otherwise you'll have an odd index based on partial data.

## Manipulating Updates

One of the core ideas behind the `ClientDataSet` component is that it is used as a local cache to collect input from a user and then send a batch of update requests to the database. The component has both a list of the changes to apply to the database server, stored in the same format used by the `ClientDataSet` (accessible through the `Delta` property), and a complete update log that you can manipulate with a few methods (including an Undo capability).

**Tip** In Delphi 7, the `ClientDataSet` component's `ApplyUpdates` and `Undo` operations are also accessible through predefined actions.

## The Status of the Records

The component lets you monitor what's going on within the data packets. The `UpdateStatus` method returns one of the following indicators for the current record:

```
type TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

To easily check the status of every record in the client dataset, you can add a string-type calculated field to the dataset (I've called it `ClientDataSet1Status`) and compute its value with the following `OnCalcFields` event handler:

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);  
begin  
    ClientDataSet1Status.AsString := GetEnumName (TypeInfo(TUpdateStatus),  
        Integer (ClientDataSet1.UpdateStatus));  
end;
```

This method (based on the RTTI `GetEnumName` function) converts the current value of the `TUpdateStatus` enumeration to a string, with the effect you can see in [Figure 14.10](#).

The screenshot shows a window titled "Client 3 Tier" with buttons for "Update", "Show Delta", and "Undo". Below the buttons is a "Data" tab and a table with the following columns: Status, DEPT\_NO, EMP\_NO, FIRST\_NAME, LAST\_NAME, PHONE\_EXT, and SALARY. The table contains 10 rows of employee data.

Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	SALARY
usUnmodified	115	118	Takashi	Yanamoto	23	
usUnmodified	125	121	Roberto	Fenali	1	
usUnmodified	100	127	Michael	Yanowski	492	
usModified	123	134	Jacques	Glon	937	
usUnmodified	623	136	Scott	Johnson	265	
usUnmodified	621	138	T.J.	Green	218	
usUnmodified	672	144	John	Montgomery	820	
usModified	622	145	Mark	Guckenheimer	931	
usInserted	622	146	John	Roland	932	

Figure 14.10: The CdsDelta program displays the status of each record of a ClientDataSet.

## Accessing the Delta

Beyond examining the status of each record, the best way to understand which changes have occurred in a given ClientDataSet (but haven't been uploaded to the server) is to look at the *delta*—the list of changes waiting to be applied to the server. This property is defined as follows:

```
property Delta: OleVariant;
```

The format used by the `Delta` property is the same as that used for the data of a client dataset. You can add another ClientDataSet component to an application and connect it to the data in the `Delta` property of the first client dataset:

```
if ClientDataSet1.ChangeCount > 0 then
begin
  ClientDataSet2.Data := ClientDataSet1.Delta;
  ClientDataSet2.Open;
```

In the CdsDelta example, I've added a data module with the two ClientDataSet components and a source of data: a SQLDataSet mapped to InterBase's EMPLOYEE demo table. Both client datasets have the extra status calculated field, with a slightly more generic version than the code discussed earlier, because the event handler is shared between them.

**Tip** To create persistent fields for the ClientDataSet hooked to the delta (at run time), I've temporarily connected it at design time to the main ClientDataSet's provider. The delta's structure is the same as the dataset it refers to. After creating the persistent fields, I removed the connection.

The application's form has a page control with two pages, each of which has a DBGrid, one for the data and one for the delta. Code hides or shows the second tab depending on the existence of data in the change log, as returned by the `ChangeCount` method, and updates the delta when the corresponding tab is selected. The core of the code used to handle the delta is similar to the previous code snippet, and you can study the example source code on the CD to see more details.

Figure 14.11 shows the change log of the CdsDelta application. Notice that the delta dataset has two entries for each modified record (the original values and the modified fields) unless this is a new or deleted record, as indicated by its status.



Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	SALARY
usUnmodified	600	2	Robert	Nelson	250	105900
usModified					251	
usInserted	622	146	John	Poland	532	32000
usUnmodified	622	145	Mark	Guckenheimer	321	32000
usModified					531	
usDeleted	121	141	Piers	Osborne		110000
usUnmodified	123	134	Jacques	Glen		390500
usModified					537	

**Figure 14.11:** The CdsDelta example allows you to see the temporary update requests stored in the `Delta` property of the `ClientDataSet`.

**Tip** You can filter the delta dataset (or any other `ClientDataSet`) depending on its update status, using the `StatusFilter` property. This property allows you to show new, updated, and deleted records in separate grids or in a grid filtered by selecting an option in a `TabControl`.

## Updating the Data

Now that you have a better understanding of what goes on during local updates, you can try to make this program work by sending the local update (stored in the delta) back to the database server. To apply all the updates from a dataset at once, pass `-1` to the `ApplyUpdates` method.

If the provider (or the Resolver component inside it) has trouble applying an update, it triggers the `OnReconcileError` event. This can take place because of a concurrent update by two different people. We tend to use optimistic locking in client/server applications, so this should be regarded as a normal situation.

The `OnReconcileError` event allows you to modify the `Action` parameter (passed by reference), which determines how the server should behave:

```
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
```

This method has three parameters: the client dataset component (in case there is more than one client dataset in the current application), the exception that caused the error (with the error message), and the kind of operation that failed (`ukModify`, `ukInsert`, or `ukDelete`). The return value, which you'll store in the `Action` parameter, can be any one of the following:

```
type TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,
  raRefresh);
```

**raSkip** Specifies that the server should skip the conflicting record, leaving it in the delta (this is the default value).

**raAbort** Tells the server to abort the entire update operation and not try to apply the remaining changes listed in the delta.

**raMerge** Tells the server to merge the client data with the data on the server, applying only the modified fields of this client (and keeping the other fields modified by other clients).

**raCorrect** Tells the server to replace its data with the current client data, overriding all field changes already made by other clients.

**raCancel** Cancels the update request, removing the entry from the delta and restoring the values originally fetched from the database (thus ignoring changes made by other clients).

**raRefresh** Tells the server to dump the updates in the client delta and to replace them with the values currently on the server (thus keeping the changes made by other clients).

To test a collision, you can launch two copies of the client application, change the same record in both clients, and then post the updates from both. We'll do this later to generate an error, but let's first see how to handle the `OnReconcileError` event.

Handling this event is not too difficult, but only because you'll receive a little help. Because building a specific form to handle an `OnReconcileError` event is common, Delphi provides such a form in the Object Repository (available with the File → New → Other menu command of the Delphi IDE). Go to the Dialogs page and select the Reconcile Error Dialog item. This unit exports a function you can use directly to initialize and display the dialog box, as I've done in the `CdsDelta` example:

```
procedure TDmCds.cdsEmployeeReconcileError (DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

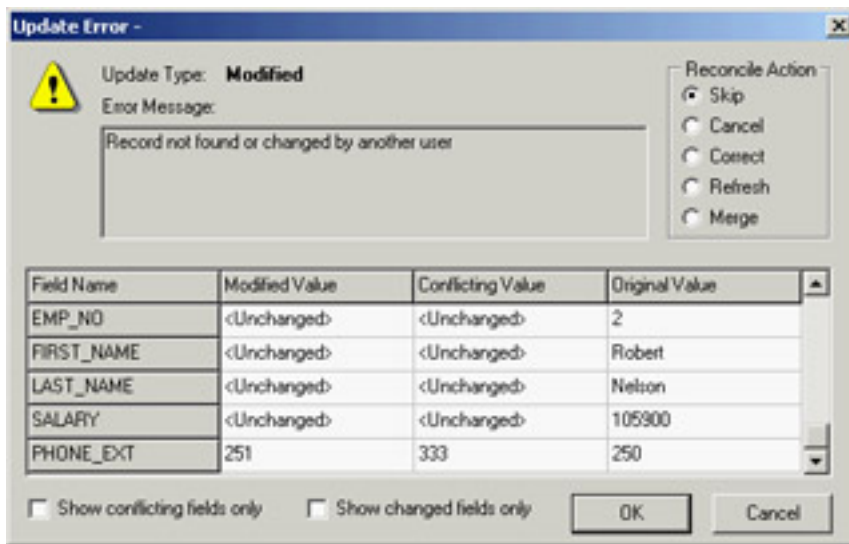
**Warning** As the source code of the Reconcile Error Dialog unit suggests, you should use the Project Options dialog to remove this form from the list of automatically created forms (if you don't, an error will occur when you compile the project). Of course, you need to do this only if you haven't set up Delphi to skip the automatic form creation.

The `HandleReconcileError` function creates the dialog box form and shows it, as you can see in the code provided by Borland:

```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
  ReconcileError: EReconcileError): TReconcileAction;
var
  UpdateForm: TReconcileErrorForm;
begin
  UpdateForm := TReconcileErrorForm.CreateForm(DataSet, UpdateKind,
    ReconcileError);
  with UpdateForm do
  try
    if ShowModal = mrOK then
    begin
      Result := TReconcileAction(ActionGroup.Items.Objects[
        ActionGroup.ItemIndex]);
      if Result = raCorrect then
        SetFieldValues(DataSet);
    end
    else
      Result := raAbort;
  finally
    Free;
  end;
end;
```

The `Reconc` unit, which hosts the Reconcile Error dialog (a window titled Update Error to be more understandable by end-users of your programs), contains more than 350 lines of code, so I can't describe it in detail. However, you should be able to understand the source code by studying it carefully. Alternatively, you can use it without caring how everything works.

The dialog box will appear in case of an error, reporting the requested change that caused the conflict and allowing the user to choose one of the possible `TReconcileAction` values. You can see this form at run time in [Figure 14.12](#).



**Figure 14.12:** The Reconcile Error dialog provided by Delphi in the Object Repository and used by the CdsDelta example

**Tip** When you call *ApplyUpdates*, you start a complex update sequence, which is discussed in more detail in [Chapter 16](#) for multitier architectures. In short, the delta is sent to the provider, which fires the *OnUpdateData* event and then receives a *BeforeUpdateRecord* event for every record to update. These are two chances you have to look at the changes and force specific operations on the database server.

## Using Transactions

Whenever you are working with a SQL server, you should use *transactions* to make your applications more robust. You can think of a transaction as a series of operations that are considered a single, "atomic" whole that cannot be split.

An example may help to clarify the concept. Suppose you have to raise the salary of each employee of a company by a fixed rate, as you did in the Total example in [Chapter 13](#). A typical program would execute a series of SQL statements on the server, one for each record to update. If an error occurred during the operation, you might want to undo the previous changes. If you consider the operation "raise the salary of each employee" as a single transaction, it should either be completely performed or completely ignored. Or, consider the analogy with financial transactions—if an error causes only part of the operation to be performed, you might end up with a missed credit or with some extra money.

Working with database operations as transactions serves a useful purpose. You can start a transaction and do several operations that should all be considered parts of a single larger operation; then, at the end, you can either commit the changes or *roll back* the transaction, discarding all the operations done up to that moment. Typically, you might want to roll back a transaction if an error occurred during its operations.

There is another important element to underline: Transactions also serve a purpose when reading data. Until data is committed by a transaction, other connections and/or transactions should not see it. Once the data is committed from a transaction, others should see the change when reading the data—that is, unless you need to open a transaction and read the same data over and over for data analysis or complex reporting operations. Different SQL servers allow you to read data in transaction according to some or all of these alternatives, as you'll see when we discuss transaction isolation levels.

Handling transactions in Delphi is simple. By default, each edit/post operation is considered a single *implicit* transaction, but you can alter this behavior by handling the operations explicitly. Use the following three methods of the dbExpress SQLConnection component (other database connection components have similar methods):

*StartTransaction* Marks the beginning of a transaction

*Commit* Confirms all the updates to the database done during the transaction

*Rollback* Returns the database to its state prior to starting the transaction

You can also use the `InTransaction` property to check whether a transaction is active. You'll often use a `try` block to roll back a transaction when an exception is raised, or you can commit the transaction as the last operation of the `try` block, which is executed only when there is no error. The code might look like this:

```
var
    TD: TTransactionDesc;
begin
    TD.TransactionID := 1;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
    try
        // -- operations within the transaction go here --
        SQLConnection1.Commit(TD);
    except
        SQLConnection1.Rollback(TD);
    end;
```

Each transaction-related method has a parameter describing the transaction it is working with. The parameter uses the record type `TTransactionDesc` and accounts for a transaction isolation level and a transaction ID. The transaction isolation level is an indication of how the transaction should behave when other transactions make changes to the data. The three predefined values are as follows:

**tiDirtyRead** Makes the transaction's updates immediately visible to other transactions, even before they are committed. This is the only possibility in a few databases and corresponds to the behavior of databases with no transaction support.

**tiReadCommitted** Makes available to other transactions only the updates already committed by this transaction. This setting is recommended for most databases, to preserve efficiency.

**tiRepeatableRead** Hides changes made by every transaction started after the current one, even if the changes have been committed. Subsequent repeat calls within a transaction will always produce the same result, as if the database took a snapshot of the data when the current transaction started. Only InterBase and few other database servers work efficiently with this model.

**Tip** As a general suggestion, for performance reasons transactions should involve a minimal number of updates (only those strictly related and part of a single atomic operation) and should be kept short in time. You should avoid transactions that wait for user input to complete them, because the user might be temporarily gone, and the transaction might remain active for a long time. Caching changes locally, as the `ClientDataSet` allows, can help you make the transactions small and fast, because you can open a transaction for reading, close it, and then open a transaction for writing out the entire batch of changes.

The other field of the `TTransactionDesc` record holds a transaction ID. It is useful only in conjunction with a database server supporting multiple concurrent transactions over the same connection, like InterBase does. You can ask the connection component whether the server supports multiple transactions or doesn't support transactions at all, using the `MultipleTransactionsSupported` and `TransactionsSupported` properties.

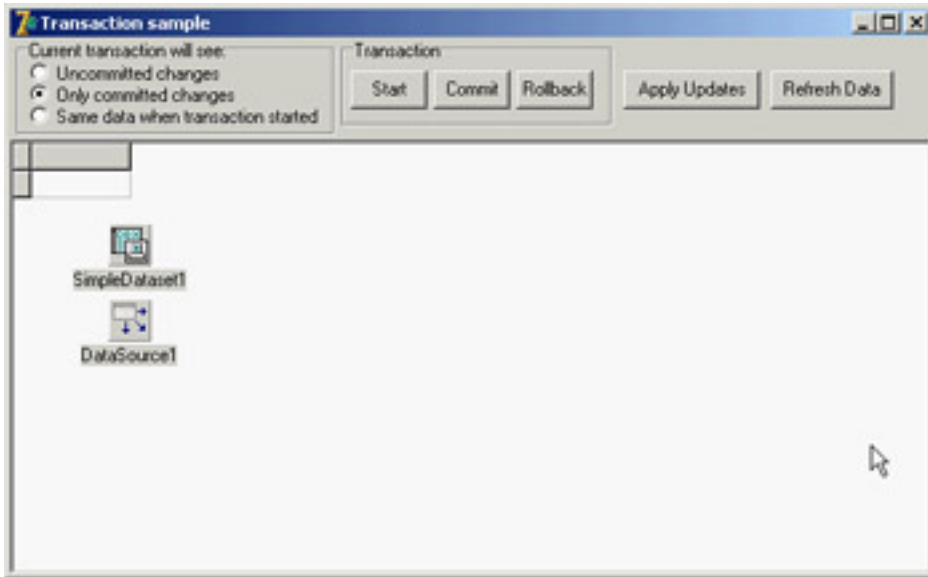
When the server supports multiple transactions, you must supply each transaction with a unique identifier when calling the `StartTransaction` method:

```
var
    TD: TTransactionDesc;
begin
    TD.TransactionID := GetTickCount;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
```

```
SQLDataSet1.TransactionLevel := TD.TransactionID;
```

You can also indicate which datasets belong to which transaction by setting the `TransactionLevel` property of each dataset to a transaction ID, as shown in the last statement.

To further inspect transactions and to experiment with transaction isolation levels, you can use the `TranSample` application. As you can see in [Figure 14.13](#), radio buttons let you choose the various isolation levels and buttons let you work on the transactions and apply updates or refresh data. To get a real idea of the different effects, you should run multiple copies of the program (provided you have enough licenses on your InterBase server).



**Figure 14.13:** The form of the `TranSample` application at design time. The radio buttons let you set different transaction isolation levels.

**Note** InterBase doesn't support the "dirty read" mode, so in the `TranSample` program you cannot use the last option unless you work with a different server.

## Using InterBase Express

The examples built earlier in this chapter were created with the new dbExpress database library. Using this server-independent approach allows you to switch the database server used by your application, although in practice doing so is often far from simple. If the application you are building will invariably use a given database, you can write programs that are tied directly to the API of the specific database server. This approach will make your programs intrinsically non-portable to other SQL servers.

Of course, you won't generally use these APIs directly, but rather base your development on dataset components that wrap these APIs and fit into Delphi and the architecture of its class library. An example of such a family of components is InterBase Express (IBX). Applications built using these components should work better and faster (even if only marginally), giving you more control over the specific features of the server. For example, IBX provides a set of administrative components specifically built for InterBase 6.

**Note** I'll examine the IBX components because they are tied to InterBase (the database server discussed in this chapter) and because they are the only set of components available in the standard Delphi installation. Other similar sets of components (for InterBase, Oracle, and other database servers) are equally powerful and well regarded in the Delphi programmers' community. A good example (and an alternative to IBX) is InterBase Objects ([www.ibobjects.com](http://www.ibobjects.com)).

## IBX Dataset Components

The IBX components include custom dataset components and a few others. The dataset components inherit from the base `TDataSet` class, can use all the common Delphi data-aware controls, and provide a field editor and all the usual design-time features. You can choose among multiple dataset components. Three IBX datasets have a role and a set of properties similar to the table/query/storedproc components in the dbExpress family:

- `IBTable` resembles the `Table` component and allows you to access a single table or view.
- `IBQuery` resembles the `Query` component and allows you to execute a SQL query, returning a result set. The `IBQuery` component can be used together with the `IBUpdateSQL` component to obtain a live (or editable) dataset.
- `IBStoredProc` resembles the `StoredProc` component and allows you to execute a stored procedure.

These components, like the related dbExpress ones, are intended for compatibility with older BDE components you might have used in your applications. For new applications, you should generally use the `IBDataSet` component, which allows you to work with a live result set obtained by executing a `select` query. It basically merges `IBQuery` with `IBUpdateSQL` in a single component. The three components in the previous list are provided mainly for compatibility with existing Delphi BDE applications.

Many other components in InterBase Express don't belong to the dataset category, but are still used in applications that need to access to a database:

- `IBDatabase` acts like the `DBX SQLConnection` component and is used to set up the database connection. The BDE also uses the specific `Session` component to perform some global tasks done by the `IBDatabase` component.
- `IBTransaction` allows complete control over transactions. It is important in InterBase to use transactions explicitly and to isolate each transaction properly, using the `Snapshot` isolation level for reports and the `Read Committed` level for interactive forms. Each dataset explicitly refers to a given transaction, so you can have multiple concurrent transactions against the same database and choose which datasets take part in which transaction.
- `IBSQL` lets you execute SQL statements that don't return a dataset (for example, DDL requests, or `update` and `delete` statements) without the overhead of a dataset component.
- `IBDatabaseInfo` is used for querying the database structure and status.



- IBSQLMonitor is used for debugging the system, because the SQL Monitor debugger provided by Delphi is a BDE-specific tool.
- IBEvents receives events posted by the server.

This group of components provides greater control over the database server than you can achieve with dbExpress. For example, having a specific transaction component allows you to manage multiple concurrent transactions over one or multiple databases, as well as a single transaction spanning multiple databases. The IBDatabase component allows you to create databases, test the connection, and generally access system data, something the Database and Session BDE components don't fully provide.

**Tip** IBX datasets let you set up the automatic behavior of a generator as a sort of auto-increment field. You do so by setting the *GeneratorField* property using its specific property editor. An example is discussed later in this chapter in the section "[Generators and IDs.](#)"

## IBX Administrative Components

The InterBase Admin page of Delphi's Component Palette hosts InterBase administrative components. Although your aim is probably not to build a full InterBase console application, including some administrative features (such as backup handling or user monitoring) can make sense in applications meant for power users.

Most of these components have self-explanatory names: IBConfigService, IBBackupService, IBRestoreService, IBValidationService, IBStatisticalService, IBLogService, IBSecurityService, IBServerProperties, IBInstall, and IBUninstall. I won't build any advanced examples that use these components, because they are more focused toward the development of server management applications than client programs. However, I'll embed a couple of them in the IbxMon example discussed later in this chapter.

## Building an IBX Example

To build an example that uses IBX, you'll need to place in a form (or data module) at least three components: an IBDatabase, an IBTransaction, and a dataset component (in this case an IBQuery). Any IBX application requires at least an instance of the first two components. You cannot set database connections in an IBX dataset, as you can do with other datasets. And, at least a transaction object is required even to read the result of a query.

Here are the key properties of these components in the IbxEmp example:

```
object IBTransaction1: TIBTransaction
  Active = False
  DefaultDatabase = IBDatabase1
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  SQL.Strings = (
    'SELECT * FROM EMPLOYEE')
end
object IBDatabase1: TIBDatabase
  DatabaseName = 'C:\Program Files\Common Files\Borland Shared\Data\employee.gdb'
  Params.Strings = (
    'user_name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  SQLDialect = 1
end
```

Now you can hook a DataSource component to IBQuery1 and easily build a user interface for the application. I had to

type in the pathname of the Borland sample database. However, not everyone has the `Program Files` folder, which depends on the local version of Windows, and the Borland sample data files could be installed elsewhere on the disk. You'll solve these problems in the next example.

**Warning** Notice that I've embedded the password in the code—a naïve approach to security. Not only can anyone run the program, but someone can extract the password by looking at the hexadecimal code of the executable file. I used this approach so I wouldn't need to keep typing in my password while testing the program, but in a real application you should require your users to do so to ensure the security of their data.

## Building a Live Query

The `IbxEmp` example includes a query that doesn't allow editing. To activate editing, you need to add an `IBUpdateSQL` component to the query, even if the query is trivial. Using an `IBQuery` that hosts the SQL `select` statement together with an `IBUpdateSQL` component that hosts the `insert`, `update`, and `delete` SQL statements is a typical approach from BDE applications. The similarities among these components make it easier to port an existing BDE application to this architecture. Here is the code for these components (edited for clarity):

```
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  SQL.Strings = (
    'SELECT Employee.EMP_NO, Department.DEPARTMENT, Employee.FIRST_NAME, '+
    ' Employee.LAST_NAME, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, '+
    ' Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY'
    'FROM EMPLOYEE Employee'
    ' INNER JOIN DEPARTMENT Department'
    ' ON (Department.DEPT_NO = Employee.DEPT_NO) '
    ' INNER JOIN JOB Job'
    ' ON (Job.JOB_CODE = Employee.JOB_CODE) '
    ' AND (Job.JOB_GRADE = Employee.JOB_GRADE) '
    ' AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY) '
    'ORDER BY Department.DEPARTMENT, Employee.LAST_NAME')
  UpdateObject = IBUpdateSQL1
end
object IBUpdateSQL1: TIBUpdateSQL
  RefreshSQL.Strings = (
    'SELECT Employee.EMP_NO, Employee.FIRST_NAME, Employee.LAST_NAME, '+
    ' Department.DEPARTMENT, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, '+
    ' Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY'
    'FROM EMPLOYEE Employee'
    'INNER JOIN DEPARTMENT Department'
    'ON (Department.DEPT_NO = Employee.DEPT_NO)'
    'INNER JOIN JOB Job'
    'ON (Job.JOB_CODE = Employee.JOB_CODE)'
    'AND (Job.JOB_GRADE = Employee.JOB_GRADE)'
    'AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)'
    'WHERE Employee.EMP_NO=:EMP_NO')
  ModifySQL.Strings = (
    'update EMPLOYEE'
    'set'
    ' FIRST_NAME = :FIRST_NAME,'
    ' LAST_NAME = :LAST_NAME,'
    ' SALARY = :SALARY,'
    ' DEPT_NO = :DEPT_NO,'
    ' JOB_CODE = :JOB_CODE,'
    ' JOB_GRADE = :JOB_GRADE,'
    ' JOB_COUNTRY = :JOB_COUNTRY'
```



```

'where'
  ' EMP_NO = :OLD_EMP_NO')
InsertSQL.Strings = (
  'insert into EMPLOYEE'
  '(FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY)'
  'values'
  '(:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, :JOB_GRADE, :JOB_COUNTRY)')
DeleteSQL.Strings = (
  'delete from EMPLOYEE '
  'where EMP_NO = :OLD_EMP_NO')
end

```

For new applications, you should consider using the `IBDataSet` component, which sums up the features of `IBQuery` and `IBUpdateSQL`. The differences between using the two components and the single component are minimal. Using `IBQuery` and `IBUpdateSQL` is a better approach when you're porting an existing application based on the two equivalent BDE components, even if porting the program directly to the `IBDataSet` component doesn't require much extra work.

In the `lbxUpdSql` example, I've provided both alternatives so you can test the differences yourself. Here is the skeleton of the DFM description of the single dataset component:

```

object IBDataSet1: TIBDataSet
  Database = IBDatabase1
  Transaction = IBTransaction1
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    ' (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, ' +
    ' JOB_COUNTRY)'
    'values'
    ' (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, ' +
    ' :JOB_GRADE, :JOB_COUNTRY)')
  SelectSQL.Strings = (...)
  UpdateRecordTypes = [cusUnmodified, cusModified, cusInserted]
  ModifySQL.Strings = (...)
end

```

If you connect the `IBQuery1` or the `IBDataSet1` component to the data source and run the program, you'll see that the behavior is identical. Not only do the components have a similar effect; the available properties and events are also similar.

In the `lbxUpdSql` program, I've also made the reference to the database a little more flexible. Instead of typing in the database name at design time, I've extracted the Borland shared data folder from the Windows Registry (where Borland saves it while installing Delphi). Here is the code executed when the program starts:

```

uses
  Registry;

procedure TForm1.FormCreate(Sender: TObject);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    Reg.OpenKey('\Software\Borland\Borland Shared\Data', False);
    IBDatabase1.DatabaseName := Reg.ReadString('Rootdir') + '\employee.gdb';
  finally

```

```

finally
  Reg.Free;
end;
  EmpDS.DataSet.Open;
end;

```

**Note** For more information about the Windows Registry and INI files, see the related sidebar in [Chapter 8](#), "The Architecture of Delphi Applications."

Another feature of this example is the presence of a transaction component. As I've said, the InterBase Express components make the use of a transaction component compulsory, explicitly following a requirement of InterBase. Simply adding a couple of buttons to the form to commit or roll back the transaction would be enough, because a transaction starts automatically as you edit any dataset attached to it.

I've also improved the program by adding an ActionList component. This component includes all the standard database actions and adds two custom actions for transaction support: Commit and Rollback. Both actions are enabled when the transaction is active:

```

procedure TForm1.ActionUpdateTransactions(Sender: TObject);
begin
  acCommit.Enabled := IBTransaction1.InTransaction;
  acRollback.Enabled := acCommit.Enabled;
end;

```

When executed, they perform the main operation but also need to reopen the dataset in a new transaction (which can also be done by "retaining" the transaction context). `CommitRetaining` doesn't really reopen a new transaction, but it allows the current transaction to remain open. This way, you can keep using your datasets, which won't be refreshed (so you won't see edits already committed by other users) but will keep showing the data you've modified. Here is the code:

```

procedure TForm1.acCommitExecute(Sender: TObject);
begin
  IBTransaction1.CommitRetaining;
end;

procedure TForm1.acRollbackExecute(Sender: TObject);
begin
  IBTransaction1.Rollback;
  // reopen the dataset in a new transaction
  IBTransaction1.StartTransaction;
  EmpDS.DataSet.Open;
end;

```

**Warning** Be aware that InterBase closes any opened cursors when a transaction ends, which means you have to reopen them and refetch the data even if you haven't made any changes. When committing data, however, you can ask InterBase to retain the *transaction context*—not to close open datasets—by issuing a `CommitRetaining` command, as mentioned before. InterBase behaves this way because a transaction corresponds to a snapshot of the data. Once a transaction is finished, you are supposed to read the data again to refetch records that may have been modified by other users. Version 6.0 of InterBase includes a `RollbackRetaining` command, but I've decided not to use it because in a rollback operation, the program should refresh the dataset data to show the original values on screen, not the updates you've discarded.

The last operation refers to a generic dataset and not a specific one, because I'm going to add a second alternate dataset to the program. The actions are connected to a text-only toolbar, as you can see in [Figure 14.14](#). The program opens the dataset at startup and automatically closes the current transaction on exit, after asking the user what to do, with the following `OnClose` event handler:

EMP_NO	FIRST_NAME	LAST_NAME	DEPARTMENT	JOB_TITLE	SALARY
05	Sue Anne	O'Brien	Consumer Electronics Div.	Administrative Assistant	31275 6
107	Kevin	Cook	Consumer Electronics Div.	Director	111262 5 6
105	Oliver H.	Bender	Corporate Headquarters	Chief Executive Officer	212650 0
12	Terri	Lee	Corporate Headquarters	Administrative Assistant	53793 0
144	John	Montgomery	Customer Services	Engineer	35000 6
94	Randy	Williams	Customer Services	Manager	56295 6
25	Roger	De Souza	Customer Support	Engineer	59482 63 6
44	Leslie	Phong	Customer Support	Engineer	56034 38 6
114	Bill	Parker	Customer Support	Engineer	35000 6
15	Katherine	Young	Customer Support	Manager	67241 25 6
136	Scott	Johnson	Customer Support	Technical Writer	60000 6
2	Robert	Nelson	Engineering	Vice President	105900 6
105	Kelly	Brown	Engineering	Administrative Assistant	27000 6
36	Roger	Reeves	European Headquarters	Sales Co-ordinator	33620 63 1
26	Ann	Bennet	European Headquarters	Administrative Assistant	22335 1
37	Wille	Stansbury	European Headquarters	Engineer	35224 06 1

Figure 14.14: The output of the IbxUpdSql example

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
    nCode: Word;
begin
    if IBTransaction1.InTransaction then
    begin
        nCode := MessageDlg ('Commit Transaction? (No to rollback)',
            mtConfirmation, mbYesNoCancel, 0);
        case nCode of
            mrYes: IBTransaction1.Commit;
            mrNo: IBTransaction1.Rollback;
            mrCancel: Action := caNone; // don't close
        end;
    end;
end;

```

## Monitoring InterBase Express

Like the dbExpress architecture, IBX also allows you to monitor a connection. You can embed a copy of the IBSQLMonitor component in your application and produce a custom log.

You can even write a more generic monitoring application, as I've done in the IbxMon example. I've placed in its form a monitoring component and a RichEdit control, and written the following handler for the OnSQL event:

```

procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
    if Assigned (RichEdit1) then
        RichEdit1.Lines.Add (TimeToStr (Now) + ': ' + EventText);
    end;

```

The `if Assigned` test can be useful when receiving a message during shutdown, and it is required when you add this code directly inside the application you are monitoring.

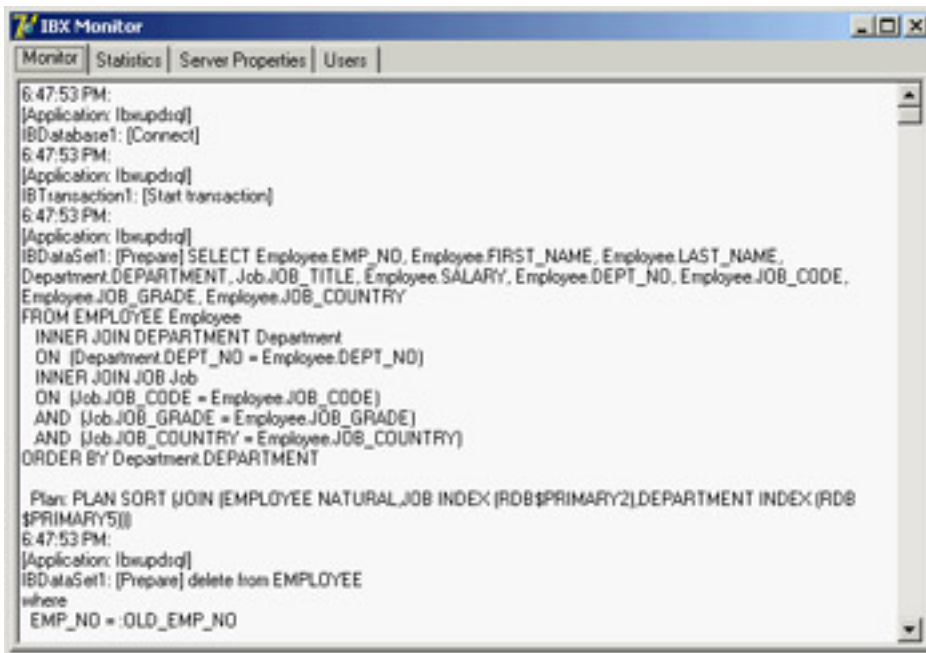
To receive messages from other applications (or from the current application), you have to turn on the IBDatabase component's tracing options. In the IbxUpdSql example (discussed in the preceding section, "[Building a Live Query](#)"), I turned them all on:

```

object IBDatabase1: TIBDatabase
    ...
    TraceFlags = [tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
        tfConnect, tfTransact, tfBlob, tfService, tfMisc]

```

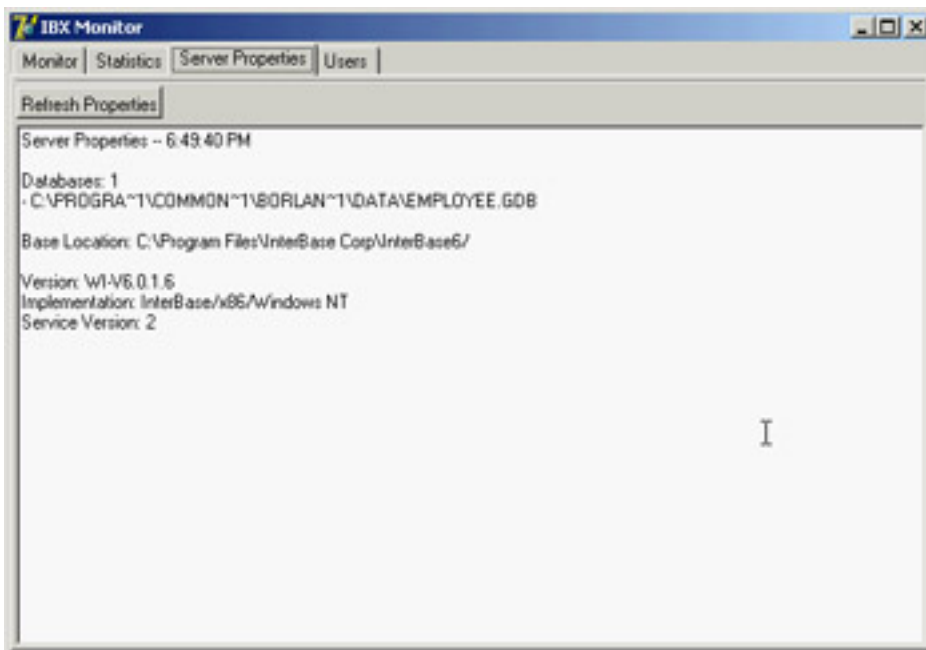
If you run the two examples at the same time, the output of the IbxMon program will list details about the IbxUpdSql program's interaction with InterBase, as you can see in [Figure 14.15](#).



**Figure 14.15:** The output of the IbxMon example, based on the IBMonitor component

## Getting More System Data

In addition to letting you monitor the InterBase connection, the IbxMon example allows you to query some server settings using the various tabs on its page control. The example embeds a few IBX administrative components, showing server statistics, server properties, and all connected users. You can see an example of the server properties in [Figure 14.16](#). The code for extracting the users appears in the following code fragment.



**Figure 14.16:** The server information displayed by the IbxMon application

```
// grab the user's data
IBSecurityService1.DisplayUsers;
```

```
// display the name of each user
for i := 0 to IBSecurityService1.UserInfoCount - 1 do
  with IBSecurityService1.UserInfo[i] do
    RichEdit4.Lines.Add (Format ('User: %s, Full Name: %s, Id: %d',
      [UserName, FirstName + ' ' + LastName, UserId]));
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Real-World Blocks

Up to now, we've discussed specific techniques related to InterBase programming, but we haven't delved into the development of an application and the problems this presents in practice. In the following subsections, I'll discuss a few practical techniques, in no specific order.

Nando Dessena (who knows InterBase much better than I do) and I have used all of these techniques in a seminar discussing the porting of an internal Paradox application to InterBase. The application we discussed in the seminar was large and complex, and I've trimmed it down to only a few tables to make it fit into the space I have for this chapter.

**Tip** The database discussed in this section is called *mastering.gdb*. You can find it in the *data* subfolder of the code folder for this chapter. You can examine it using InterBase Console, possibly after making a copy to a writable drive so that you can fully interact with it.

## Generators and IDs

I mentioned in [Chapter 13](#) that I'm a fan of using IDs extensively to identify the records in each table of a database.

**Note** I tend to use a single sequence of IDs for an entire system, something often called an Object ID (OID) and discussed in a sidebar earlier in this chapter. In such a case, however, the IDs of the two tables must be unique. Because you might not know in advance which objects could be used in place of others, adopting a global OID allows you more freedom later. The drawback is that if you have lots of data, using a 32-bit integer as the ID (that is, having only 4 billion objects) might not be sufficient. For this reason, InterBase 6 supports 64-bit generators.

How do you generate the unique values for these IDs when multiple clients are running? Keeping a table with a *latest* value will create troubles, because multiple concurrent transactions (from different users) will see the same values. If you don't use tables, you can use a database-independent mechanism, including the rather large Windows GUIDs or the so-called *high-low technique* (the assignment of a base number to each client at startup—the high number—that is combined with a consecutive number—the low number—determined by the client).

Another approach, bound to the database, is the use of internal mechanisms for sequences, indicated with different names in each SQL server. In InterBase they are called *generators*. These sequences operate and are incremented outside of transactions, so that they provide unique numbers even to concurrent users (remember that InterBase forces you to open a transaction to read data).

You've already seen how to create a generator. Here is the definition for the one in my demo database, followed by the definition of the view you can use to query for a new value:

```
create generator g_master;

create view v_next_id (
    next_id
) as
select gen_id(g_master, 1) from rdb$database
;
```

Inside the RWBlocks application, I've added an IBQuery component to a data module (because I don't need it to be an editable dataset) with the following SQL:

```
select next_id from v_next_id;
```

The advantage, compared to using the direct statement, is that this code is easier to write and maintain, even if the underlying generator changes (or you switch to a different approach behind the scenes). Moreover, in the same data module, I've added a function that returns a new value for the generator:

```
function TDmMain.GetNewId: Integer;
begin
```

```

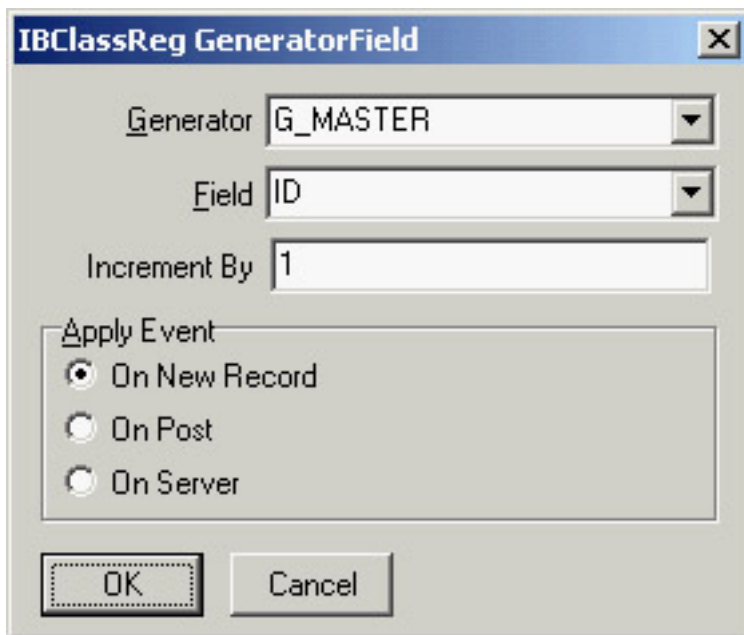
// return the next value of the generator
QueryId.Open;
try
    Result := QueryId.Fields[0].AsInteger;
finally
    QueryId.Close;
end;
end;

```

This method can be called in the `AfterInsert` event of any dataset to fill in the value for the ID:

```
mydataset.FieldName ('ID').AsInteger := data.GetNewId;
```

As I've mentioned, the IBX datasets can be tied directly to a generator, thus simplifying the overall picture. Thanks to the specific property editor (shown in [Figure 14.17](#)), connecting a field of the dataset to the generator becomes trivial.



**Figure 14.17:** The editor for the `GeneratorField` property of the IBX datasets

Notice that both these approaches are much better than the approach based on a server-side trigger, discussed earlier in this chapter. In that case, the Delphi application didn't know the ID of the record sent to the database and so was unable to refresh it. Not having the record ID (which is also the only key field) on the Delphi side means it is almost impossible to insert such a value directly inside a `DBGrid`. If you try, you'll see that the value you insert gets lost, only to reappear in case of a full refresh.

Using client-side techniques based on the manual code or the `GeneratorField` property causes no trouble. The Delphi application knows the ID (the record key) before posting it, so it can easily place it in a grid and refresh it properly.

## Case-Insensitive Searches

An interesting issue with SQL servers in general, not specifically InterBase, has to do with case-insensitive searches. Suppose you don't want to show a large amount of data in a grid (which is a bad idea for a client/server application). You instead choose to let the user type the initial portion of a name and then filter a query on this input, displaying only the smaller resulting record set in a grid. I've done this for a table of companies.

This search by company name will be executed frequently and will take place on a large table. However, if you search using the `starting with` or `like` operator, the search will be case sensitive, as in the following SQL statement:

```
select * from companies
where name starting with 'win';
```

To make a case-insensitive search, you can use the `upper` function on both sides of the comparison to test the uppercase values of each string, but a similar query will be very slow, because it won't be based on an index. On the other hand, saving the company names (or any other name) in uppercase letters would be silly, because when you print those names, the result will be unnatural (even if common in old information systems).

If you can trade off some disk space and memory for the extra speed, you can use a trick: Add an extra field to the table to store the uppercase value of the company name, and use a server-side trigger to generate it and update it. You can then ask the database to maintain an index on the uppercase version of the name, to speed the search operation even further.

In practice, the table definition looks like this:

```
create domain d_uid as integer;
create table companies
(
  id          d_uid not null,
  name       varchar(50),
  tax_code   varchar(16),
  name_upper varchar(50),
  constraint companies_pk primary key (id)
);
```

To copy the uppercase name of each company into the related field, you cannot rely on client-side code, because an inconsistency would cause problems. In a case like this, it is better to use a trigger on the server, so that each time the company name changes, its uppercase version is updated accordingly. Another trigger is used to insert a new company:

```
create trigger companies_bi for companies
active before insert position 0
as
begin
  new.name_upper = upper(new.name);
end;
```

```
create trigger companies_bu for companies
active before update position 0
as
begin
  if (new.name <> old.name) then
    new.name_upper = upper(new.name);
  end;
```

Finally, I've added an index to the table with this DDL statement:

```
create index i_companies_name_upper on companies(name_upper);
```

With this structure behind the scenes, you can now select all the companies starting with the text of an edit box (`edSearch`) by writing the following code in a Delphi application:

```
dm.DataCompanies.Close;
dm.DataCompanies.SelectSQL.Text :=
  'select c.id, c.name, c.tax_code,' +
  ' from companies c ' +
  ' where name_upper starting with ''' +
  UpperCase (edSearch.Text) + ''';
dm.DataCompanies.Open;
```



**Tip** Using a prepared parametric query, you might be able to make this code even faster.

As an alternative, you could create a server-side calculated field in the table definition, but doing so would prevent you from having an index on the field, which speeds up your queries considerably:

```
name_upper varchar(50) computed by (upper(name))
```

## Handling Locations and People

You might notice that the table describing companies is quite bare. It has no company address, nor any contact information. The reason is that I want to be able to handle companies that have multiple offices (or locations) and list contact information about multiple employees of those companies.

Every location is bound to a company. Notice, though, that I've decided not to use a location identifier related to the company (such as a progressive location number for each company), but rather a global ID for all the locations. This way, I can refer to a location ID (let's say, for shipping goods) without having to also refer to the company ID. This is the definition of the table that stores company locations:

```
create table locations
(
  id            d_uid not null,
  id_company   d_uid not null,
  address      varchar(40),
  town         varchar(30),
  zip          varchar(10),
  state        varchar(4),
  phone        varchar(15),
  fax          varchar(15),
constraint locations_pk primary key (id),
constraint locations_uc unique (id_company, id)
);

alter table locations add constraint locations_fk_companies
foreign key (id_company) references companies (id)
on update no action on delete no action;
```

The final definition of a foreign key relates the `id_company` field of the `locations` table with the `ID` field of the `companies` table. The other table lists names and contact information for people at specific company locations. To follow the database normalization rules, I should have added to this table only a reference to the location, because each location relates to a company. However, to make it simpler to change the location of a person within a company and to make my queries much more efficient (avoiding an extra step), I've added to the `people` table both a reference to the location and a reference to the company.

The table has another unusual feature: One of the people working for a company can be set as the key contact. You obtain this functionality with a Boolean field (defined with a domain, because the Boolean type is not supported by InterBase) and by adding triggers to the table so that only one employee of each company can have this flag active:

```
create domain d_boolean as char(1)
default 'F'
check (value in ('T', 'F')) not null

create table people
(
  id            d_uid not null,
  id_company   d_uid not null,
  id_location  d_uid not null,
  name         varchar(50) not null,
  phone        varchar(15),
  fax          varchar(15),
```

```

email          varchar(50),
key_contact    d_boolean,
constraint people_pk primary key (id),
constraint people_uc unique (id_company, name)
);

alter table people add constraint people_fk_companies
foreign key (id_company) references companies (id)
on update no action on delete cascade;
alter table people add constraint people_fk_locations
foreign key (id_company, id_location)
references locations (id_company, id);

create trigger people_ai for people
active after insert position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;

create trigger people_au for people
active after update position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T' and old.key_contact = 'F') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;

```

## Building a User Interface

The three tables discussed so far have a clear master/detail relation. For this reason, the RWBlocks example uses three IBDataSet components to access the data, hooking up the two secondary tables to the main one. The code for the master/detail support is that of a standard database example based on queries, so I won't discuss it further (but I suggest you study the example's source code).

Each of the datasets has a full set of SQL statements, to make the data editable. Whenever you enter a new detail element, the program hooks it to its master tables, as in the two following methods:

```

procedure TDmCompanies.DataLocationsAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataLocationsID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
end;

procedure TDmCompanies.DataPeopleAfterInsert(DataSet: TDataSet);

```

```

begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataPeopleID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
  // the suggested location is the active one, if available
  if not DataLocations.IsEmpty then
    DataPeopleID_LOCATION.AsInteger := DataLocationsID.AsInteger;
  // the first person added becomes the key contact
  // (checks whether the filtered dataset of people is empty)
  DataPeopleKEY_CONTACT.AsBoolean := DataPeople.IsEmpty;
end;

```

As this code suggests, a data module hosts the dataset components. The program has a data module for every form (hooked up dynamically, because you can create multiple instances of each form). Each data module has a separate transaction so that the various operations performed in different pages are totally independent. The database connection, however, is centralized. A main data module hosts the corresponding component, which is referenced by all the datasets. Each of the data modules is created dynamically by the form referring to it, and its value is stored in the form's dm private field:

```

procedure TFormCompanies.FormCreate(Sender: TObject);
begin
  dm := TDmCompanies.Create (Self);
  dsCompanies.Dataset := dm.DataCompanies;
  dsLocations.Dataset := dm.DataLocations;
  dsPeople.Dataset := dm.DataPeople;
end;

```

This way, you can easily create multiple instances of a form, with an instance of the data module connected to each of them. The form connected to the data module has three DBGrid controls, each tied to a data module and one of the corresponding datasets. You can see this form at run time, with some data, in [Figure 14.18](#).

ID	ID_COMPANY	ADDRESS	FAX	PHONE	STATE	TOWN
11	9	100, Barland Way			CA	Scotts Valley
12	9	Fryland			UK	Reading

ID	ID_COMPANY	ID_LOCATION	KEY_CONTACT	NAME	EMAIL
12	9	11	F	Chuck J	
14	9	11	F	David I	

**Figure 14.18:** A form showing companies, office locations, and people (part of the RWBlocks example)

The form is hosted by a main form, which in turn is based on a page control, with the other forms embedded. Only the form connected with the first page is created when the program starts. The ShowForm method I've written takes care of parenting the form to the tab sheet of the page control, after removing the form border:

```

procedure TFormMain.FormCreate(Sender: TObject);
begin

```

```

ShortDateFormat := 'dd/mm/yyyy';
ShowForm (TFormCompanies.Create (Self), TabCompanies);
end;

procedure TFormMain.ShowForm (Form: TForm; Tab: TTabSheet);
begin
  Form.BorderStyle := bsNone;
  Form.Align := alClient;
  Form.Parent := Tab;
  Form.Show;
end;

```

The other two pages are populated at run time:

```

procedure TFormMain.PageControllChange(Sender: TObject);
begin
  if PageControll.ActivePage.ControlCount = 0 then
    if PageControll.ActivePage = TabFreeQ then
      ShowForm (TFormFreeQuery.Create (self), TabFreeQ)
    else if PageControll.ActivePage = TabClasses then
      ShowForm (TFormClasses.Create (self), TabClasses);
end;

```

The companies form hosts the search by company name (discussed in the [previous section](#)) plus a search by location. You enter the name of a town and get back a list of companies having an office in that town:

```

procedure TFormCompanies.btnTownClick(Sender: TObject);
begin
  with dm.DataCompanies do
    begin
      Close;
      SelectSQL.Text :=
        'select c.id, c.name, c.tax_code' +
        ' from companies c ' +
        ' where exists (select loc.id from locations loc ' +
        ' where loc.id_company = c.id and upper(loc.town) = ''' +
        UpperCase(edTown.Text) + ''' )';
      Open;
      dm.DataLocations.Open;
      dm.DataPeople.Open;
    end;
  end;

```

The form includes a lot more source code. Some of it is related to closing permission (as a user cannot close the form while there are pending edits not posted to the database), and quite a bit relates to the use of the form as a lookup dialog, as described later.

## Booking Classes

Part of the program and the database involves booking training classes and courses. (Although I built this program as a showcase, it also helps me run my own business.) The database includes a classes table that lists all the training courses, each with a title and the planned date. Another table hosts registration by company, including the classes registered for, the ID of the company, and some notes. Finally, a third table lists people who've signed up, each hooked to a registration for his or her company, with the amount paid.

The rationale behind this company-based registration is that invoices are sent to companies, which book the classes for programmers and can receive specific discounts. In this case the database is more normalized, because the people registration doesn't refer directly to a class, but only to the company registration for that class. Here are the definitions

of the tables involved (I've omitted foreign key constraints and other elements):

```
create table classes
(
  id            d_uid not null,
  description   varchar(50),
  starts_on    timestamp not null,
constraint classes_pk primary key (id)
);
create table classes_reg
(
  id            d_uid not null,
  id_company   d_uid not null,
  id_class     d_uid not null,
  notes        varchar(255),
constraint classes_reg_pk primary key (id),
constraint classes_reg_uc unique (id_company, id_class)
);
create domain d_amount as numeric(15, 2);
create table people_reg
(
  id            d_uid not null,
  id_classes_reg d_uid not null,
  id_person     d_uid not null,
  amount        d_amount,
constraint people_reg_pk primary key (id)
);
```

The data module for this group of tables uses a master/detail/detail relationship, and has code to set the connection with the active master record when a new detail record is created. Each dataset has a generator field for its ID, and each has the proper update and insert SQL statements. These statements are generated by the corresponding component editor using only the ID field to identify existing records and updating only the fields in the original table. Each of the two secondary datasets retrieves data from a lookup table (either the list of companies or the list of people). I had to edit the RefreshSQL statements manually to repeat the proper inner join. Here is an example:

```
object IBClassReg: TIBDataSet
  Database = DmMain.IBDatabase1
  Transaction = IBTransaction1
  AfterInsert = IBClassRegAfterInsert
  DeleteSQL.Strings = (
    'delete from classes_reg'
    'where id = :old_id')
  InsertSQL.Strings = (
    'insert into classes_reg (id, id_class, id_company, notes)'
    'values (:id, :id_class, :id_company, :notes)')
  RefreshSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id = :id')
  SelectSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id_class = :id')
  ModifySQL.Strings = (
    'update classes_reg'
```

```

'set'
' id = :id,'
' id_class = :id_class,'
' id_company = :id_company,'
' notes = :notes'
'where id = :old_id')
GeneratorField.Field = 'id'
GeneratorField.Generator = 'g_master'
DataSource = dsClasses
end

```

To complete the discussion of IBClassReg, here is its only event handler:

```

procedure TDmClasses.IBClassRegAfterInsert(DataSet: TDataSet);
begin
  IBClassReg.FieldName ('id_class').AsString :=
    IBClasses.FieldName ('id').AsString;
end;

```

The IBPeopleReg dataset has similar settings, but the IBClasses dataset is simpler at design time. At run time, this dataset's SQL code is dynamically modified, using three alternatives to display scheduled classes (whenever the date is after today's date), classes already started or finished in the current year, and classes from past years. A user selects one of the three groups of records for the table with a tab control, which hosts the DBGrid for the main table (see [Figure 14.19](#)).

ID	DESCRIPTION	STARTS_ON
18	Learning 201	11/10/2002

ID	NAME
19	Buford Cop
20	Whitech Lake St

ID	NAME	AMOUNT
21	David J	300
24	Chuck J	300

**Figure 14.19:** The RWBlocks example form for class registrations

The three alternative SQL statements are created when the program starts, or when the class registrations form is created and displayed. The program stores the final portion of the three alternative instructions (the *where* clause) in a string list and selects one of the strings when the tab changes:

```

procedure TFormClasses.FormCreate(Sender: TObject);
begin
  dm := TDmClasses.Create (Self);
  // connect the datasets to the data sources
  dsClasses.Dataset := dm.IBClasses;
  dsClassReg.DataSet := dm.IBClassReg;
  dsPeopleReg.DataSet := dm.IBPeopleReg;

```

```

// open the datasets
dm.IBClasses.Active := True;
dm.IBClassReg.Active := True;
dm.IBPeopleReg.Active := True;

// prepare the SQL for the three tabs
SqlCommands := TStringList.Create;
SqlCommands.Add (' where Starts_On > ''now'' ');
SqlCommands.Add (' where Starts_On <= ''now'' and ' +
  ' extract (year from Starts_On ) >= extract(year from current_timestamp)');
SqlCommands.Add (' where extract (year from Starts_On) < ' +
  ' extract(year from current_timestamp)');
end;
procedure TFormClasses.TabChange(Sender: TObject);
begin
  dm.IBClasses.Active := False;
  dm.IBClasses.SelectSQL [1] := SqlCommands [Tab.TabIndex];
  dm.IBClasses.Active := True;
end;

```

## Building a Lookup Dialog

The two detail datasets of this class registration form display lookup fields. Instead of showing the ID of the company that booked the class, for example, the form shows the company name. You obtain this functionality with an inner join in the SQL statement and by configuring the DBGrid columns so they don't display the company ID. In a local application, or one with a limited amount of data, you could use a lookup field. However, copying the entire lookup dataset locally or opening it for browsing should be limited to tables with about 100 records at most, embedding some search capabilities.

If you have a large table, such as a table of companies, an alternative solution is to use a secondary dialog box to perform the lookup selection. For example, you can choose a company by using the form you've already built and taking advantage of its search capabilities. To display this form as a dialog box, the program creates a new instance of it, shows some hidden buttons already there at design time, and lets the user select a company to refer to from the other table.

To simplify the use of this lookup, which can happen multiple times in a large program, I've added to the companies form a class function that has as output parameters the name and ID of the selected company. An initial ID can be passed to the function to determine its initial selection. Here is the complete code of this class function, which creates an object of its class, selects the initial record if requested, shows the dialog box, and finally extracts the return values:

```

class function TFormCompanies.SelectCompany (
  var CompanyName: string; var CompanyId: Integer): Boolean;
var
  FormComp: TFormCompanies;
begin
  Result := False;
  FormComp := TFormCompanies.Create (Application);
  FormComp.Caption := 'Select Company';
  try
    // activate dialog buttons
    FormComp.btnCancel.Visible := True;
    FormComp.btnOK.Visible := True;
    // select company
    if CompanyId > 0 then
      FormComp.dm.DataCompanies.SelectSQL.Text :=
        'select c.id, c.name, c.tax_code' +
        ' from companies c ' +

```

```

    ' where c.id = ' + IntToStr (CompanyId)
else
    FormComp.dm.DataCompanies.SelectSQL.Text :=
        'select c.id, c.name, c.tax_code' +
        ' from companies c ' +
        ' where name_upper starting with ''a''';
FormComp.dm.DataCompanies.Open;
FormComp.dm.DataLocations.Open;
FormComp.dm.DataPeople.Open;

if FormComp.ShowModal = mrOK then
begin
    Result := True;
    CompanyId := FormComp.dm.DataCompanies.FieldByName ('id').AsInteger;
    CompanyName := FormComp.dm.DataCompanies.FieldByName ('name').AsString;
end;
finally
    FormComp.Free;
end;
end;
end;

```

Another slightly more complex class function (available with the example's source code, but not listed here) lets you select a person from a given company to register people for classes. In this case, the form is displayed after disallowing searching another company or modifying the company's data.

In both cases, you trigger the lookup by adding an ellipsis button to the column of the DBGrid—for example, the grid column listing the names of companies registered for classes. When this button is clicked, the program calls the class function to display the dialog box and uses its result to update the hidden ID field and the visible name field:

```

procedure TFormClasses.DBGridClassRegEditButtonClick(Sender: TObject);
var
    CompanyName: string;
    CompanyId: Integer;
begin
    CompanyId := dm.IBClassReg.FieldByName ('id_Company').AsInteger;
    if TFormCompanies.SelectCompany (CompanyName, CompanyId) then
    begin
        dm.IBClassReg.Edit;
        dm.IBClassReg.FieldByName ('Name').AsString := CompanyName;
        dm.IBClassReg.FieldByName ('id_Company').AsInteger := CompanyId;
    end;
end;

```

## Adding a Free Query Form

The program's final feature is a form where a user can directly type in and run a SQL statement. As a helper, the form lists in a combo box the available tables of the database, obtained when the form is created by calling

```
DmMain.IBDatabase1.GetTableNames (ComboTables.Items);
```

Selecting an item from the combo box generates a generic SQL query:

```
MemoSql.Lines.Text := 'select * from ' + ComboTables.Text;
```

The user (if an expert) can then edit the SQL, possibly introducing restrictive clauses, and then run the query:

```

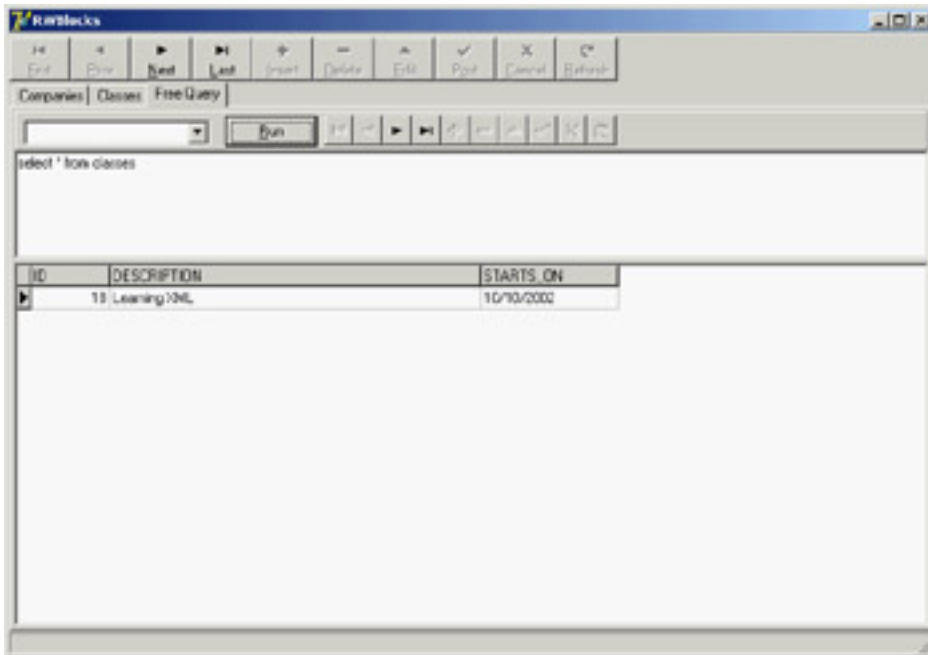
procedure TFormFreeQuery.ButtonRunClick(Sender: TObject);
begin

```



```
QueryFree.Close;  
QueryFree.SQL := MemoSql.Lines;  
QueryFree.Open;  
end;
```

You can see this third form of the RWBlocks program in [Figure 14.20](#). Of course, I'm not suggesting that you add SQL editing to programs intended for all your users—this feature is intended for power users or programmers. I basically wrote it for myself!



**Figure 14.20:** The free query form of the RWBlocks example is intended for power users.

## What's Next?

This chapter has presented a detailed introduction to client/server programming with Delphi. We discussed the key issues and delved a little into interesting areas of client/server programming. After a general introduction, I discussed the use of the dbExpress database library. I also briefly covered the InterBase server and the InterBase Express (IBX) components. At the end of the chapter I presented a real-world example, going beyond the typical examples in the book to focus on a single feature at a time.

There is more I can say about client/server programming in Delphi. [Chapter 15](#) will focus on Microsoft's ADO database engine. It's followed by a chapter on Delphi's multitier architecture, DataSnap; this chapter highlights the use of the dbExpress library and the ClientDataSet component, although in a slightly different context.

## Chapter 15: Working with ADO

### Overview

Since the mid-1980s, database programmers have been on a quest for the "holy grail" of *database independence*. The idea is to use a single API that applications can use to interact with many different sources of data. The use of such an API would release developers from dependence on a single database engine and allow them to adapt to the world's changing demands. Vendors have produced many solutions to this goal, the two most notable early solutions being Microsoft's Open Database Connectivity (ODBC) and Borland's Integrated Database Application Programming Interface (IDAPI), more commonly known as the Borland Database Engine (BDE).

Microsoft started to replace ODBC with OLE DB in the mid-1990s with the success of the Component Object Model (COM). However, OLE DB is what Microsoft would class a *system-level* interface and is intended to be used by system-level programmers. It is very large, complex, and unforgiving. It makes greater demands on the programmer and requires a higher level of knowledge in return for lower productivity. ActiveX Data Objects (ADO) is a layer on top of OLE DB and is referred to as an *application-level* interface. It is considerably simpler than OLE DB and more forgiving. In short, it is designed for use by application programmers.

As you saw in [Chapter 14](#), "Client/Server with dbExpress," Borland has also replaced the BDE with a newer technology called dbExpress. ADO shares greater similarities with the BDE than with the lightweight dbExpress technology. BDE and ADO support navigation and manipulation of datasets, transaction processing, and cached updates (called *batch updates* in ADO), so the concepts and issues involved in using ADO are similar to those of the BDE.

**Note** I wish to acknowledge and thank Guy Smith-Ferrier for originally writing this chapter for *Mastering Delphi 6*. Guy is a programmer, author, and speaker. He is the author of several commercial software products and countless internal systems for independent and blue-chip companies alike. He has written many articles for *The Delphi Magazine* and for others on topics beyond Delphi, and he has spoken at numerous conferences in North America and Europe. Guy lives in England with his wife, his son, and his cat.

In this chapter we will look at ADO. We will also discuss *dbGo*—a set of Delphi components initially called ADOExpress, but renamed in Delphi 6 because Microsoft objects to the use of the term *ADO* in third-party product names. It is possible to use ADO in Delphi without using dbGo. By importing the ADO type library, you can gain direct access to the ADO interfaces; this is how Delphi programmers used ADO before the release of Delphi 5. However, this path bypasses Delphi's database infrastructure and ensures that you are unable to make use of other Delphi technologies such as the data-aware controls or DataSnap. This chapter uses dbGo for all of its examples, not only because it is readily available and supported but also because it is a very viable solution. Regardless of your final choice, you will find the information here useful.

**Note** Alternatively, you can turn to Delphi's active third-party market for other ADO component suites such as Adonis, AdoSolutio, Diamond ADO, and Kamiak.

## Microsoft Data Access Components (MDAC)

ADO is part of a bigger picture called Microsoft Data Access Components (MDAC). MDAC is an umbrella for Microsoft's database technologies and includes ADO, OLE DB, ODBC, and RDS (Remote Data Services). Often you will hear people use the terms MDAC and ADO interchangeably, but incorrectly. Because ADO is only distributed as part of MDAC, we talk of ADO versions in terms of MDAC releases. The major releases of MDAC have been versions 1.5, 2.0, 2.1, 2.5, and 2.6. Microsoft releases MDAC independently and makes it available for free download and virtually free distribution (there are distribution requirements, but most Delphi developers will have no trouble meeting them). MDAC is also distributed with most Microsoft products that have database content. Delphi 7 ships with MDAC 2.6.

There are two consequences of this level of availability. First, it is highly likely that your users will already have MDAC installed on their machines. Second, whatever version your users have, or you upgrade them to, it is also virtually certain that someone—you, your users, or other application software—will upgrade their existing MDAC to the current release of MDAC. You can't prevent this upgrade, because MDAC is installed with such commonly used software as Internet Explorer. Add to this the fact that Microsoft supports only the current release of MDAC and the release before it, and you arrive at this conclusion: Applications must be designed to work with the current release of MDAC or the release before it.

As an ADO developer, you should regularly check the MDAC pages on Microsoft's website at [www.microsoft.com/data](http://www.microsoft.com/data). From there you can download the latest version of MDAC for free. While you are on this website, you should take the opportunity to download the MDAC SDK (13 MB) if you do not already have it or the Platform SDK (the MDAC SDK is part of the Platform SDK). The MDAC SDK is your bible: Download it, consult it regularly, and use it to answer your ADO questions. You should treat it as your first port of call when you need MDAC information.

## OLE DB Providers

OLE DB providers enable access to a source of data. They are ADO's equivalent to the dbExpress drivers and the BDE SQL Links. When you install MDAC, you automatically install the OLE DB providers shown in [Table 15.1](#):

**Table 15.1: OLE DB Providers Included with MDAC**

Driver	Provider	Description
MSDASQL	ODBC Drivers	ODBC drivers (default)
Microsoft.Jet.OLEDB.3.5	Jet 3.5	MS Access 97 databases only
Microsoft.Jet.OLEDB.4.0	Jet 4.0	MS Access and other databases
SQLOLEDB	SQL Server	MS SQL Server databases
MSDAORA	Oracle	Oracle databases
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	Example of an OLE DB provider for CSV files
MSDAOXP	Simple provider	For creating your own providers for simple text data

- The ODBC OLE DB provider is used for backward compatibility with ODBC. As you learn more about ADO, you will discover the limitations of this provider.
- The Jet OLE DB providers support MS Access and other desktop databases. We will return to these providers later.
- The SQL Server provider supports SQL Server 7, SQL Server 2000, and Microsoft Database Engine (MSDE). MSDE is a reduced version of SQL Server, with most of the tools removed and some code added to deliberately

degrade performance when there are more than five active connections. MSDE is important because it is free and it is fully compatible with SQL Server.

- The OLE DB provider for OLAP can be used directly but is more often used by ADO Multi-Dimensional (ADOMD). ADOMD is an additional ADO technology designed to provide Online Analytical Processing (OLAP). If you have used Delphi's Decision Cube, Excel's Pivot Tables, or Access's Cross Tabs, then you have used some form of OLAP.

In addition to these MDAC OLE DB providers, Microsoft supplies other OLE DB providers with other products or with downloadable SDKs:

- The Active Directory Services OLE DB provider is included with the ADSI SDK; the AS/400 and VSAM OLE DB provider is included with SNA Server; and the Exchange OLE DB provider is included with Microsoft Exchange 2000.
- The OLE DB provider for Indexing Service is part of Microsoft Indexing Service, a Windows mechanism that speeds up file searches by building catalogs of file information. Indexing Service is integrated into IIS and, consequently, is often used for indexing websites.
- The OLE DB provider for Internet Publishing allows developers to manipulate directories and files using HTTP.
- Still more OLE DB providers come in the form of *service providers*. As their name implies, OLE DB service providers provide a service to other OLE DB providers and are often invoked automatically as needed without programmer intervention. The Cursor Service, for example, is invoked when you create a client-side cursor, and the Persisted Recordset provider is invoked to save data locally.

MDAC includes many providers that I'll discuss, but many more are available from Microsoft and from the third-party market. It is impossible to reliably list all available OLE DB providers, because the list is so large and changes constantly. In addition to independent third parties, you should consider most database vendors, because the majority now supply their own OLE DB providers. For example, Oracle supplies the ORAOLEDB provider.

**Tip** A notable omission from the vendors that supply OLE DB providers is InterBase. In addition to accessing it using the ODBC driver, you can use Dmitry Kovalenko's IBProvider ([www.lcpi.lipetsk.ru/prog/eng/index.html](http://www.lcpi.lipetsk.ru/prog/eng/index.html)). Also check Binh Ly's OLE DB Provider Development Toolkit ([www.techvanguards.com/products/optk/install.htm](http://www.techvanguards.com/products/optk/install.htm)). If you want to write your own OLE DB provider, this tool is easier to use than most.

## Using dbGo Components

Programmers familiar with the BDE, dbExpress, or IBExpress should recognize the set of components that make up dbGo ([Table 15.2](#)).

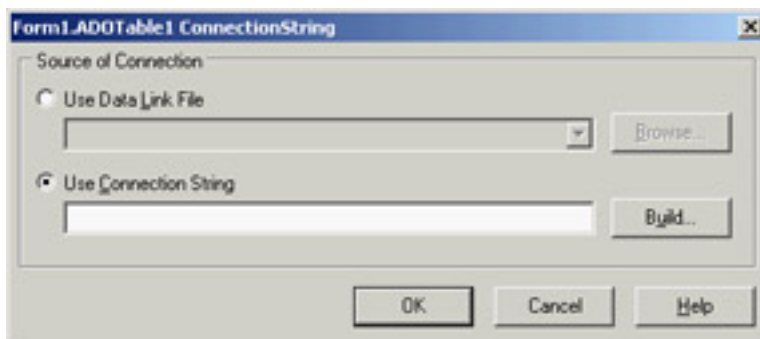
**Table 15.2: dbGo Components**

dbGo Component	Description	BDE Equivalent Component
ADOConnection	Connection to a database	Database
ADOCommand	Executes an action SQL command	No equivalent
ADODataset	All-purpose descendant of TDataSet	No equivalent
ADOTable	Encapsulation of a table	Table
ADOQuery	Encapsulation of SQL <code>SELECT</code>	Query
ADOStoredProc	Encapsulation of a stored procedure	StoredProc
RDSCConnection	Remote Data Services connection	No equivalent

The four dataset components (ADODataset, ADOTable, ADOQuery, and ADOStoredProc) are implemented almost entirely by their immediate ancestor class, TCustomADODataset. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component. As such, the components have a lot in common. In general, however, ADOTable, ADOQuery, and ADOStoredProc are viewed as "compatibility" components and are used to aid the transition of knowledge and code from their BDE counterparts. Be warned, though: These compatibility components are similar to their counterparts but not identical. You will find differences in any application except the most trivial. ADODataset is the component of choice partly because of its versatility but also because it is closer in appearance to the ADO Recordset interface upon which it is based. Throughout this chapter, I'll use all the dataset components to give you the experience of using each.

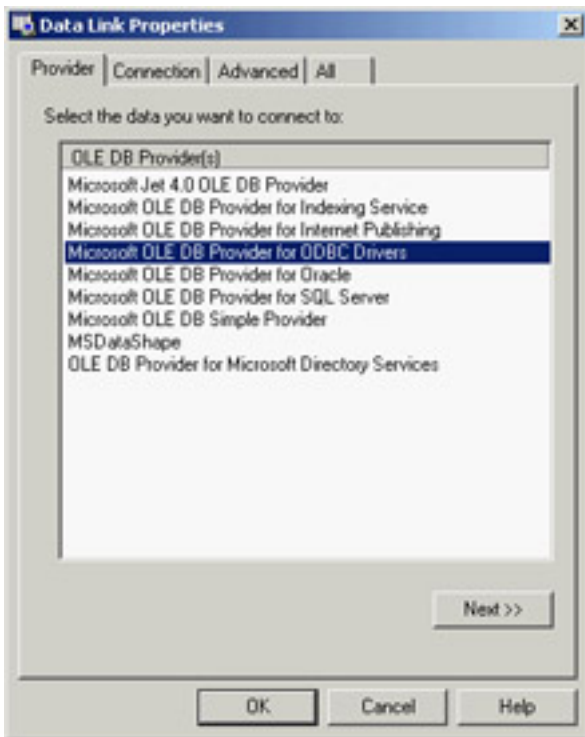
## A Practical Example

Enough theory: Let's see some action. Drop an ADOTable onto a form. To indicate the database to connect to, ADO uses *connection strings*. You can type in a connection string by hand if you know what you are doing. In general, you'll use the connection string editor (the property editor of the `ConnectionString` property), shown in [Figure 15.1](#).



**Figure 15.1:** Delphi's connection string editor

This editor adds little value to the process of entering a connection string, so you can click Build to go straight to Microsoft's connection string editor, shown in [Figure 15.2](#).



**Figure 15.2:** The first page of Microsoft's connection string editor

This is a tool you need to understand. The first tab shows the OLE DB providers and service providers installed on your computer. The list will vary according to the version of MDAC and other software you have installed. In this example, select the Jet 4.0 OLE DB provider. Double-click Jet 4.0 OLE DB Provider, and you will be presented with the Connection tab. This page varies according to the provider you select; for Jet, it asks you for the name of the database and your login details. You can choose an Access MDB file installed by Borland with Delphi 7: the `dbdemos.mdb` file available in the shared data folder (by default, `C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb`). Click the Test Connection button to test the validity of your selections.

The Advanced tab handles access control to the database; here you specify exclusive or read-only access to the database. The All tab lists all the parameters in the connection string. The list is specific to the OLE DB provider you selected on the first page. (You should make a mental note of this page, because it contains many parameters that are the answers to many problems.) After closing the Microsoft connection string editor you'll see in the Borland ConnectionString property editor the value that will be returned to the ConnectionString property (here split on multiple lines for readability):

```
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb;
Persist Security Info=False
```

Connection strings are just strings with many parameters delimited by semicolons. To add, edit, or delete any of these parameter values programmatically, you must write your own routines to find the parameter in the list and amend it appropriately. A simpler approach is to copy the string into a Delphi string list and use its name/value pairs capability: This technique will be demonstrated in the JetText example covered in the section "[Text Files through Jet.](#)"

Now that you have set the connection string, you can select a table. Drop down the list of tables using the `TableName` property in the Object Inspector. Select the Customer table. Add a `DataSource` component and a `DBGrid` control and connect them all together; you are now using ADO in an actual—though trivial—program (available in the source code as `FirstAdoExample`). To see the data, set the `Active` property of the dataset to `True` or open the dataset in the `FormCreate` event (as in the example) to avoid design-time errors if the database is not available.



**Tip** If you are going to use dbGo as your primary database access technology, you might want to move the DataSource component to the ADO page of the Component Palette to avoid moving back and forth between the ADO page and the Data Access page. If you use both ADO and another database technology, then you can simulate installing DataSource on multiple pages by creating a Component Template for a DataSource and installing it on the ADO page.

## The ADOConnection Component

When you use an ADOTable component this way, it creates its own connection component behind the scenes. You do not have to accept the default connection it creates. In general, you should create your own connection using the ADOConnection component, which has the same purpose as the dbExpress SQLConnection component and the BDE Database component. It allows you to customize the login procedure, control transactions, execute action commands directly, and reduce the number of connections in an application.

Using an ADOConnection is easy. Place one on a form and set its `ConnectionString` property the same way you would for the ADOTable. Alternatively, you can double-click an ADOConnection component (or use a specific item of its Component Editor, in its shortcut menu) to invoke the connection string editor directly. With the `ConnectionString` set to the proper database, you can disable the login dialog box by setting `LoginPrompt` to `False`. To use the new connection in the previous example, set the `Connection` property of `ADOTable1` to `ADOConnection1`. You will see `ADOTable1`'s `ConnectionString` property reset because the `Connection` and `ConnectionString` properties are mutually exclusive. One of the benefits of using an ADOConnection is that the connection string is centralized instead of scattered throughout many components. Another, more important, benefit is that all the components that share the ADOConnection share a single connection to the database server. Without your own ADOConnection, each ADO dataset has a separate connection.

## Data Link Files

So, an ADOConnection allows you to centralize the definition of a connection string within a form or data module. However, even though this is a worthwhile step forward from scattering the same connection string throughout all ADO datasets, it still suffers from a fundamental flaw: If you use a database engine that defines the database in terms of a filename, then the path to the database file(s) is hard-coded in the EXE. This makes for a fragile application. To overcome this problem, ADO uses Data Link files.

A Data Link file is a connection string in an INI file. For example, Delphi's installation adds to the system the `dbdemos.udl` file, with the following text:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;
  Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb
```

Although you can give a Data Link file any extension, the recommended extension is `.UDL`. You can create a Data Link using any text editor, or you can right-click in Windows Explorer, select `New`, select `Text Document`, rename the file with a `.UDL` extension (assuming extensions are displayed in your configuration of Explorer), and then double-click the file to invoke the Microsoft connection string editor.

When you select a file in the Connection editor, the `ConnectionString` property will be set to `'FILE NAME ='` followed by the actual filename, as demonstrated by the `DataLinkFile` example. You can place your Data Link files anywhere on the hard disk, but if you are looking for a common, shared location, then you can use the `DataLinkDir` function in the `ADODB` Delphi unit. If you haven't altered MDAC's defaults, `DataLinkDir` will return the following:

```
C:\Program Files\Common Files\System\OLE DB\Data Links
```

## Dynamic Properties

Imagine that you are responsible for designing a new database middleware architecture. You have to reconcile two



opposing goals of a single API for all databases and access to database-specific features. You could take the approach of designing an interface that is the sum of all the features of every database ever created. Each class would have every property and method imaginable, but it would only use the properties and methods it had support for. It doesn't take much discussion to realize that this isn't a good solution. ADO has to solve these apparently mutually exclusive goals, and it does so using *dynamic properties*.

Almost all ADO interfaces and their corresponding dbGo components have a property called `Properties` that is a collection of database-specific properties. These properties can be accessed by their ordinal position, like this:

```
ShowMessage(ADOTable1.Properties[1].Value);
```

But they are more usually accessed by name:

```
ShowMessage(ADOConnection1.Properties['DBMS Name'].Value);
```

Dynamic properties depend on the type of object and also on the OLE DB providers. To give you an idea of their importance, a typical ADO Connection or Recordset has approximately 100 dynamic properties. As you will see throughout this chapter, the answers to many ADO questions lie in dynamic properties.

**Tip** An important event related to the use of dynamic properties is *OnRecordsetCreate*, which was introduced in a Delphi 6 update and is available in Delphi 7. *OnRecordsetCreate* is called immediately after the recordset has been created, but before it is opened. This event is useful when you're setting dynamic properties that can be set only when the recordset is closed.

## Getting Schema Information

In ADO, you can retrieve schema information using the `ADOConnection` component's `OpenSchema` method. This method accepts four parameters:

- The kind of data `OpenSchema` should return. It is a `TSchemaInfo` value: a set of 40 values including those for retrieving a list of tables, indexes, columns, views, and stored procedures.
- A filter to place on the data before it is returned. You will see an example of this parameter in a moment.
- A GUID for a provider-specific query. This parameter is used only if the first parameter is `siProviderSpecific`.
- An `ADODataset` into which the data is returned. This parameter illustrates a common theme in ADO: Any method that needs to return more than a small amount of data will return its data as a `Recordset`, or, in Delphi terms, an `ADODataset`.

To use `OpenSchema`, you need an open `ADOConnection`. The following code (part of the `OpenSchema` example) retrieves a list of primary keys for every table into an `ADODataset`:

```
ADOConnection1.OpenSchema(siPrimaryKeys, EmptyParam, EmptyParam, ADODataset1);
```

Each field in a primary key has a single row in the result set. So, a table with a composite key of two fields has two rows. The two `EmptyParam` values indicate that these parameters are given empty values and are ignored. The result of this code is shown in [Figure 15.3](#), after resizing the grid with some custom code.

The screenshot shows a Delphi dialog box titled 'TADOConnection.OpenSchema'. It has two buttons: 'Get Primary Keys' (selected) and 'Get Customer Primary Keys'. Below the buttons is a table with the following data:

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	COLUMN_GUID	COLUMN_PROPID	ORDIN
		country	Name			
		customer	CustNo			
		employee	EmpNo			
		items	ItemNo			
		items	OrderNo			
		MSysAccessObjects	ID			
		orders	OrderNo			
		parts	PartNo			
		vendors	VendorNo			

**Figure 15.3:** The OpenSchema example retrieves the primary keys of the database tables.

When `EmptyParam` is passed as the second parameter, the result set includes all information of the requested type for the entire database. For many kinds of information, you will want to filter the result set. You can, of course, apply a traditional Delphi filter to the result set using the `Filter` and `Filtered` properties or the `OnFilterRecord` event. However, doing so applies the filter on the client side in this example. Using the second parameter, you can apply a more efficient filter at the source of the schema information. The filter is specified as an array of values. Each element of the array has a specific meaning relevant to the kind of data being returned. For example, the filter array for primary keys has three elements: the catalog (*catalog* is ANSI-speak for the database), the schema, and the table name. This example returns a list of primary keys for the Customer table:

```
var
  Filter: OLEVariant;
begin
  Filter := VarArrayCreate([0, 2], varVariant);
  Filter[2] := 'CUSTOMER';
  ADOConnection1.OpenSchema(
    siPrimaryKeys, Filter, EmptyParam, ADODataSet1);
end;
```

**Note** You can retrieve the same information using ADOX, and this warrants a brief comparison between *OpenSchema* and ADOX. ADOX is an additional ADO technology that allows you to retrieve and update schema information. It is ADO's equivalent to SQL's Data Definition Language (DDL—*CREATE*, *ALTER*, *DROP*) and Data Control Language (DCL—*GRANT*, *REVOKE*). ADOX is not directly supported in dbGo, but you can import the ADOX type library and use it successfully in Delphi applications. Unfortunately, ADOX is not as universally implemented as *OpenSchema* so there are greater gaps. To just retrieve information and not update it, *OpenSchema* is usually a better choice.

## Using the Jet Engine

Now that you have some of the MDAC and ADO basics under your belt, let's take a moment out to look at the Jet engine. This engine is of great interest to some and of no interest to others. If you're interested in Access, Paradox, dBase, text, Excel, Lotus 1-2-3, or HTML, then this section is for you. If you have no interest in any of these formats, you can safely skip this section.

The Jet database engine is usually associated with Microsoft Access databases, and this is its forte. However, the Jet engine is also an all-purpose desktop database engine, and this lesser-known attribute is where much of its strength lies. Because using the Jet engine with Access is its default mode and is straightforward, this section mostly covers use of non-Access formats, which are not so obvious.

**Note** The Jet engine has been included in some (but not all) versions of MDAC. It is not included in version 2.6. There has been a long debate about whether programmers using a non-Microsoft development tool have the right to distribute the Jet engine. The official answer is positive, and the Jet engine is available as a free download (in addition to being distributed with many Microsoft software products).

There are two Jet OLE DB providers: the Jet 3.51 OLE DB provider and the Jet 4.0 OLE DB provider. The Jet 3.51 OLE DB provider uses the Jet 3.51 engine and supports Access 97 databases only. If you intend to use Access 97 and not Access 2000, then you will get better performance using this OLE DB provider in most situations than using the Jet 4.0 OLE DB provider.

The Jet 4.0 OLE DB provider supports Access 97, Access 2000, and Installable Indexed Sequential Access Method (IISAM) drivers. Installable ISAM drivers are those written specifically for the Jet engine to support access to ISAM formats such as Paradox, dBase, and text, and it is this facility that makes the Jet engine so useful and versatile. The complete list of ISAM drivers installed on your machine depends on what software you have installed. You can find this list by looking in the Registry at

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM Formats
```

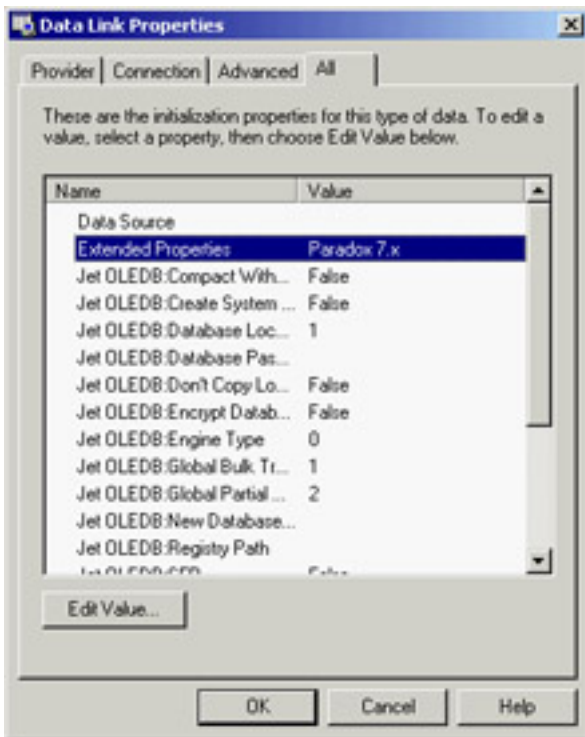
However, the Jet engine includes drivers for Paradox, dBase, Excel, text, and HTML.

## Paradox through Jet

The Jet engine expects to be used with Access databases. To use it with any database other than Access, you need to tell it which IISAM driver to use. This is a painless process that involves setting the Extended Properties connection string argument in the connection string editor. Let's work through a quick example.

Add an ADOTable component to a form and invoke the connection string editor. Select the Jet 4.0 OLE DB Provider. Select the All page, locate the Extended Properties property, and double-click it to edit its value.

Enter **Paradox 7.x** in the Property Value, as illustrated in [Figure 15.4](#), and click OK. Now go back to the Connection tab and enter the name of the directory containing the Paradox tables directly, because the Browse button won't help you (it lets you enter a filename, not a folder name). At this point you can select a table in the ADOTable's `TableName` and open it either at design time or at run time. You are now using Paradox through ADO, as demonstrated by the JetParadox example.



**Figure 15.4:** Setting extended properties

I have some bad news for Paradox users: Under certain circumstances, you will need to install the BDE in addition to the Jet engine. Jet 4.0 requires the BDE in order to be able to update Paradox tables, but it doesn't require the BDE just to read them. The same is true for most releases of the Paradox ODBC Driver. Microsoft has received justified criticism about this point and has made a new Paradox IISAM available that does not require the BDE; you can get these updated drivers from Microsoft Technical Support.

**Note** As you learn more about ADO, you will discover how much it depends on the OLE DB provider and the RDBMS (relational database management system) in question. Although you can use ADO with a local file format, as demonstrated in this and following examples, the general suggestion is to install a local SQL engine whenever possible. Access and MSDE are good choices if you have to use ADO; otherwise you might want to consider InterBase or Firebird as alternatives, as discussed in [Chapter 14](#).

## Excel through Jet

Excel is easily accessed using the Jet OLE DB provider. Once again, you set the Extended Properties property to Excel 8.0. Assume that you have an Excel spreadsheet called `ABCCompany.xls` with a sheet called `Employees`, and you want to open and read this file using Delphi. With a little knowledge of COM, you can do so by automating Excel. However, the ADO solution is considerably easier to implement and doesn't require Excel to be available on the computer.

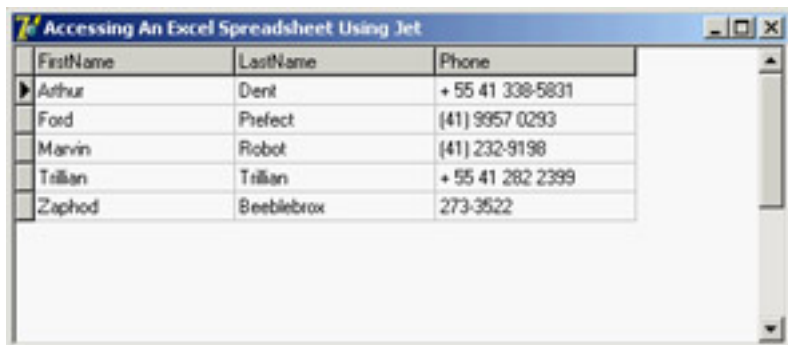
**Tip** You can also read an Excel file using the `XLSReadWrite` component (available from [www.axolot.com](http://www.axolot.com)). It doesn't require Excel to be available on the computer or the time to start it (like OLE Automation techniques do).

Ensure that your spreadsheet is not open in Excel, because ADO requires exclusive access to the file. Add an `ADODataset` component to a form. Set its `ConnectionString` to use the Jet 4.0 OLE DB provider and set Extended Properties to Excel 8.0. In the Connection tab, set the database name to the full file and path specification of the Excel spreadsheet (or use a relative path if you plan to deploy the file along with the program).

The `ADODataset` component works by opening or executing a value in its `CommandText` property. This value might be the name of a table, a SQL statement, a stored procedure, or the name of a file. You specify how this value is interpreted by setting the `CommandType` property. Set `CommandType` to `cmdTableDirect` to indicate that the value in `CommandText` is the name of a table and that all columns should be returned from this table. Select `CommandText` in

the Object Inspector, and you will see a drop-down arrow. Drop down the arrow and a single pseudo-table will be displayed: Employees\$. (Excel workbooks are suffixed with a \$.)

Add a DataSource and a DBGrid and connect them altogether, and you'll obtain the output of the JetExcel example, shown in [Figure 15.5](#) at design time. By default it would be a little difficult to view the data in the grid, because each column has a width of 255 characters. You can change the field display size either by adding columns to the grid and changing their width properties or by adding persistent fields and changing their Size or DisplayWidth properties.



FirstName	LastName	Phone
Arthur	Dent	+ 55 41 338-5831
Ford	Prefect	((41) 9957 0293
Marvin	Robot	((41) 232-9198
Trillian	Trillian	+ 55 41 282 2399
Zaphod	Beeblebrox	273-3522

**Figure 15.5:** ABCCompany.xls in Delphi—a small tribute to Douglas Adams

Notice that you cannot keep the dataset open at design time and run the program, because the Excel IISAM driver opens the XLS file in exclusive mode. Close the dataset and add to the program a line of code to open it at startup. When you run the program, you will notice another limitation of this IISAM driver: You can add new rows and edit existing rows, but you cannot delete rows.

Incidentally, you could have used either an ADOTable or an ADOQuery component, instead of the ADODataset, but you need to be aware of how ADO treats symbols in things like table names and field names. If you use an ADOTable and drop down the list of tables, you will see the Employees\$ table as you expect. Unfortunately, if you attempt to open the table, you will receive an error. The same is true for `SELECT * FROM Employees$` in a TADOQuery. The problem lies with the dollar sign in the table name. If you use characters such as dollar signs, dots, or, more importantly, spaces in a table name or field name, then you must enclose the name in square brackets (for example, [Employees\$]).

## Text Files through Jet

One of the most useful IISAM drivers that comes with the Jet engine is the Text IISAM. This driver allows you to read and update text files of almost any structured format. We will begin with a simple text file and then cover the variations.

Assume you have a text file called `NightShift.TXT` that contains the following text:

```
CrewPerson ,HomeTown
Neo ,Cincinnati
Trinity ,London
Morpheus ,Milan
```

Add an ADOTable component to a form, set its `ConnectionString` to use the Jet 4.0 OLE DB provider, and set `Extended Properties` to `Text`. The Text IISAM provider considers a directory a database, so you need to enter as the database name the directory that contains the `NightShift.TXT` file. In the Object Inspector and drop down the list of tables in the `TableName` property. You will notice that the dot in the filename has been converted to a hash, as in `NightShift#TXT`. Set `Active` to `True`, add a DataSource and a DBGrid and connect them altogether, and you will see the contents of the text file in a grid.

**Warning** If your computer's settings are such that the decimal separator is a comma instead of a period (so that 1,000.00 is displayed as 1.000,00), then you will need to either change your Regional Settings (Start → Settings → Control Panel → Regional Settings → Numbers) or take advantage of `SCHEMA.INI`, described shortly.

The grid indicates that the widths of the columns are 255 characters. You can change these values just as you did in the JetExcel program by adding persistent fields or columns to the grid and then setting the relevant width property. Alternatively, you can define the structure of the text file more specifically using `SCHEMA.INI`.

In the JetText example, the database folder is determined at run time depending on the folder hosting the program. To modify the connection string at run time, first load it into a string list (after converting the separators) and then use the `Values` property to change only one of the elements of the connection string. This is the code from the example:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    sl: TStringList;  
begin  
    sl := TStringList.Create;  
    sl.Text := StringReplace (ADOTable1.ConnectionString,  
        ';', sLineBreak, [rfReplaceAll]);  
    sl.Values ['Data Source'] := ExtractFilePath (Application.ExeName);  
    ADOTable1.ConnectionString := StringReplace (sl.Text,  
        sLineBreak, ';', [rfReplaceAll]);  
    ADOTable1.Open;  
    sl.Free;  
end;
```

Text files come in all shapes and sizes. Often you do not need to worry about the format of a text file because the Text IISAM takes a peek at the first 25 rows to see whether it can determine the format for itself. It uses this information and additional information in the Registry to decide how to interpret the file and how to behave. If you have a file that doesn't match a regular format the Text IISAM can determine, then you can provide this information using a `SCHEMA.INI` file located in the same directory as the text files to which it refers. This file contains *schema* information, also called metadata, about any or all of the text files in the same directory. Each text file is given its own section, identified by the name of the text file, such as `[NightShift.TXT]`.

Thereafter you can specify the format of the file; the names, types, and sizes of columns; any special character sets to use; and any special column formats (such as date/time or currency). Let's assume that you change your `NightShift.TXT` file to the following format:

```
Neo           |Cincinnati  
Trinity       |London  
Morpheus      |Milan
```

In this example, the column names are not included in the text file, and the delimiter is a vertical bar. An associated `SCHEMA.INI` file might look something like the following:

```
[NightShift.TXT]  
Format=Delimited(|)  
ColNameHeader=False  
Col1=CrewPerson Char Width 10  
Col2=HomeTown Char Width 30
```

Regardless of whether you use a `SCHEMA.INI` file, you will encounter two limitations of the Text IISAM: Rows cannot be deleted, and rows cannot be edited.

## Importing and Exporting

The Jet engine is particularly adept at importing and exporting data. The process of exporting data is the same for each export format and consists of executing a `SELECT` statement with a special syntax. Let's begin with an example of exporting data from the Access version of the `DBDemos` database back to a Paradox table. You will need an active `ADOConnection`, called `ADOConnection1` in the `JetImportExport` example, which uses the Jet engine to open the database. The following code exports the `Customer` table to a Paradox `Customer.db` file:

```
SELECT * INTO Customer IN "C:\tmp" "Paradox 7.x;" FROM CUSTOMER
```

Let's look at the pieces of this `SELECT` statement. The `INTO` clause specifies the new table that will be created by the `SELECT` statement; this table must not already exist. The `IN` clause specifies the database to which the new table is added; in Paradox, this is a directory that already exists. The clause immediately following the database is the name of the IISAM driver to be used to perform the export. *You must include the trailing semicolon at the end of the driver name.* The `FROM` clause is a regular part of any `SELECT` statement. In the sample program, the operation is executed through the `ADOConnection` component and uses the program's folder instead of a fixed one:

```
ADOConnection1.Execute ('SELECT * INTO Customer IN "' +  
    CurrentFolder + '" "Paradox 7.x;" FROM CUSTOMER');
```

All export statements follow these same basic clauses, although IISAM drivers have differing interpretations of what a database is. Here, you export the same data to Excel:

```
ADOConnection1.Execute ('SELECT * INTO Customer IN "' +  
    CurrentFolder + 'dbdemos.xls" "Excel 8.0;" FROM CUSTOMER');
```

A new Excel file called `dbdemos.xls` is created in the application's current directory. A workbook called `Customer` is added, containing all the data from the `Customer` table in `dbdemos.mdb`.

This last example exports the same data to an HTML file:

```
ADOConnection1.Execute ('SELECT * INTO [Customer.htm] IN "' +  
    CurrentFolder + '" "HTML Export;" FROM CUSTOMER');
```

In this case, the database is the directory, as it was for Paradox but not for Excel. The table name must include the `.htm` extension and, therefore, it must be enclosed in square brackets. Notice that the name of the IISAM driver is `HTML Export`, not just `HTML`, because this driver can only be used for exporting to HTML.

The last IISAM driver we'll look at in this investigation of the Jet engine is the partner to HTML Export: HTML Import. Add an `ADOTable` to a form, set its `ConnectionString` to use the Jet 4.0 OLE DB provider, and set `Extended Properties` to HTML Import. Set the database name to the name of the HTML file created by the export a few moments ago—that is, `Customer.htm`. Now set the `TableName` property to `Customer`. Open the table—you have just imported the HTML file. Bear in mind, though, that if you attempt to update the data, you'll receive an error because this driver is intended for import only. Finally, if you create your own HTML files containing tables and want to open these tables using this driver, then remember that the name of the table is the value of the `caption` tag of the HTML table.



## Working with Cursors

Two properties of ADO datasets have a fundamental impact on your application and are inextricably linked with each other: `CursorLocation` and `CursorType`. If you want to understand your ADO dataset behavior, you must understand these two properties.

### Cursor Location

The `CursorLocation` property allows you to specify what is in control of the retrieval and update of your data. You have two choices: `client` (`clUseClient`) or `server` (`clUseServer`). Your choice affects your dataset's functionality, performance, and scalability.

A client cursor is managed by the ADO Cursor Engine. This engine is an excellent example of an OLE DB service provider: It provides a service to other OLE DB providers. The ADO Cursor Engine manages the data from the client side of the application. All data in the result set is retrieved from the server to the client when the dataset is opened. Thereafter, the data is held in memory, and updates and manipulation are managed by the ADO Cursor Engine. This is similar to using the `ClientDataSet` component in a `dbExpress` application. One benefit is that manipulation of the data, after the initial retrieval, is considerably faster. Furthermore, because the manipulation is performed in memory, the ADO Cursor Engine is more versatile than most server-side cursors and offers extra facilities. I'll examine these benefits later, as well as other technologies that depend on client-side cursors (such as disconnected and persistent recordsets).

A server-side cursor is managed by the RDBMS. In a client/server architecture based on a database such as SQL Server, Oracle, or InterBase, this means the cursor is managed physically on the server. In a desktop database such as Access or Paradox, the "server" location is a logical location, because the database is running on the desktop. Server-side cursors are often faster to load than client-side cursors because not all the data is transferred to the client when the dataset is opened. This behavior also makes them more suitable for very large result sets where the client has insufficient memory to hold the entire result set in memory. Often you can determine what kinds of features will be available to you with each cursor location by thinking through how the cursor works. Locking is a good example of how features determine the cursor type; I will discuss locking in more detail later. (To place a lock on a record requires a server-side cursor, because there must be a conversation between the application and the RDBMS.)

Another issue that will affect your choice of cursor location is scalability. Server-side cursors are managed by the RDBMS; in a client/server database, this will be located on the server. As more users use your application, the load on the server increases with each server-side cursor. A greater workload on the server means that the RDBMS becomes a bottleneck more quickly, so the application is less scalable. You can achieve better scalability by using client-side cursors. The initial hit on opening the cursor is often heavier, because all the data is transferred to the client, but the maintenance of the open cursor can be lower. As you can see, many conflicting issues are involved in choosing the correct cursor location for your datasets.

### Cursor Type

Your choice of cursor location directly affects your choice of cursor type. To all intents and purposes there are four cursor types, but one value is unused: a cursor type that means *unspecified*. Many values in ADO signify an unspecified value, and I will cover them all here and explain why you won't have much to do with them. They exist in Delphi because they exist in ADO. ADO was primarily designed for Visual Basic and C programmers. In these languages, you use objects directly without the assistance `dbGo` provides. As such, you can create and open *recordsets*, as they are called in ADO-speak, without having to specify every value for every property. The properties for which a value has not been specified have an unspecified value. However, in `dbGo` you use components. These components have constructors, and these constructors initialize the properties of the components. So, from the moment you create a `dbGo` component, it will usually have a value for every property. As a consequence, you have little need for the unspecified values in many enumerated types.

Cursor types affect how your data is read and updated. As I mentioned, there are four choices: forward-only, static, keyset, and dynamic. Before we get too involved in all the permutations of cursor locations and cursor types, you should be aware that there is only one cursor type available for client-side cursors: the static cursor. All other cursor



types are available only to server-side cursors. I'll return to the subject of cursor type availability after we have looked at the various cursor types, in increasing order of expense:

**Forward-Only Cursor** The forward-only cursor is the least expensive cursor type, and therefore the type with the best performance. As the name implies, the forward-only cursor lets you navigate forward. The cursor reads the number of records specified by `CacheSize` (default of 1); each time it runs out of records, it reads another `CacheSize` set. Any attempt to navigate backward through the result set beyond the number of records in the cache results in an error. This behavior is similar to that of a `dbExpress` dataset. A forward-only cursor is not suitable for use in the user interface where the user can control the direction through the result set. However, it is eminently suitable for batch operations, reports and stateless Web applications, because these situations start at the top of the result set and work progressively toward the end, and then the result set is closed.

**Static Cursor** A static cursor works by reading the complete result set and providing a window of `CacheSize` records into the result set. Because the complete result set has been retrieved by the server, you can navigate both forward and backward through the result set. However, in exchange for this facility, the data is static—updates, insertions, and deletions made by other users cannot be seen, because the cursor's data has already been read.

**Keyset Cursor** A keyset cursor is best understood by breaking *keyset* into the two words *key* and *set*. *Key*, in this context, refers to an identifier for each row. Often this will be a primary key. A keyset cursor, therefore, is a set of keys. When the result set is opened, the complete list of keys for the result set is read. If, for example, the dataset was a query like `SELECT * FROM CUSTOMER`, then the list of keys would be built from `SELECT CUSTID FROM CUSTOMER`. This set of keys is held until the cursor is closed. When the application requests data, the OLE DB provider reads the rows using the keys in the set of keys. Consequently, the data is always up to date. If another user changes a row in the result set, then the changes will be seen when the data is reread. However, the set of keys is static; it is read only when the result set is first opened. If another user adds new records, these additions will not be seen. Deleted records become inaccessible, and changes to primary keys (you don't let your users change primary keys, do you?) are also inaccessible.

**Dynamic Cursor** The most expensive cursor type, a dynamic cursor is almost identical to a keyset cursor. The sole difference is that the set of keys is reread when the application requests data that is not in the cache. Because the default for `ADODataset.CacheSize` is 1, such requests occur frequently. You can imagine the additional load this behavior places on the DBMS and the network, and why this is the most expensive cursor. However, the result set can see and respond to additions and deletions made by other users.

## Ask and Ye Shall Not Receive

Now that you know about cursor locations and cursor types, a word of warning: Not all combinations of cursor location and cursor type are possible. Usually, this limitation is imposed by the RDBMS and/or the OLE DB provider as a result of the functionality and architecture of the database. For example, client cursors always force the cursor type to static. You can see behavior this for yourself. Add an `ADODataset` component to a form, set its `ConnectionString` to any database, and set the `ClientLocation` property to `clUseCursor` and the `CursorType` property to `ctDynamic`. Now set `Active` to `True` and keep your eye on the `CursorType`; it changes to `ctStatic`. You learn an important lesson from this example: What you ask for is not necessarily what you get. Always check your properties after opening a dataset to see the actual effect of your requests.

Each OLE DB provider will make different changes according to different requests and circumstances, but here are a few examples to give you an idea of what to expect:

- The Jet 4.0 OLE DB provider changes most cursor types to keyset.
- The SQL Server OLE DB provider often changes keyset and static to dynamic.

- The Oracle OLE DB provider changes all cursor types to forward-only.
- The ODBC OLE DB provider makes various changes according to the ODBC driver in use.

## No Record Count

ADO datasets sometimes return `-1` for their `RecordCount`. A forward-only cursor cannot know how many records are in the result set until it reaches the end, so it returns `-1` for the `RecordCount`. A static cursor always knows how many records are in the result set, because it reads the entire set when it is opened, so it returns the number of records in its result set. A keyset cursor also knows how many records are in the result set, because it has to retrieve a fixed set of keys when the result set is opened, so it also returns a useful value for `RecordCount`. A dynamic cursor does not reliably know how many records are in the result set, because it is regularly rereading the set of keys, so it returns `-1`. You can avoid using `RecordCount` altogether and execute `SELECT COUNT(*) FROM tablename`, but the result will be an accurate reflection of the number of records in the database table—which is not necessarily the same as the number of records in the dataset.

## Client Indexes

One of the many benefits of client-side cursors is the ability to create local, or *client*, indexes. Assuming you have an ADO client-side dataset for the DBDemos Customer table, which has a grid attached to it, set the dataset's `IndexFieldNames` property to `CompanyName`. The grid will immediately show that the data is in `CompanyName` order. There is an important point here: In order to index the data, ADO did not have to reread the data from its source. The index was created from the data in memory. So, not only is the index created as quickly as possible, but the network and the DBMS are not overloaded by transferring the same data over and over in different orders.

The `IndexFieldNames` property has more potential. Set it to `Country;CompanyName` and you will see the data ordered first by country and then, within country, by company name. Now set `IndexFieldNames` to `CompanyName DESC`. Be sure to write `DESC` in capitals (and not `desc` or `Desc`). The data is now sorted in descending order.

This simple but powerful feature allows you to solve one of the great bugbears of database developers. Users ask the inevitable, and quite reasonable, question, "Can I click the columns of the grid to sort my data?" Answers like replacing grids with non-data-aware controls such as `ListView` that have the sorting built into the control or like trapping the `DBGrid`'s `OnTitleClick` event and reissuing the SQL `SELECT` statement after including an appropriate `ORDER BY` clause are far from satisfactory.

If you have the data cached on the client side (as you've also seen in the use of the `ClientDataSet` component), you can use a client index computed in memory. Add the following `OnTitleClick` event to the grid (the code is available in the `ClientIndexes` example):

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
    if ADODataSet1.IndexFieldNames = Column.Field.FieldName then
        ADODataSet1.IndexFieldNames := Column.Field.FieldName + ' DESC'
    else
        ADODataSet1.IndexFieldNames := Column.Field.FieldName
end;
```

This simple event checks to see whether the current index is built on the same field as the column. If it is, then a new index is built on the column, but in descending order. If not, then a new index is built on the column. When the user clicks the column for the first time, it is sorted in ascending order; when it is clicked for the second time, it is sorted in descending order. You could extend this functionality to allow the user to `Ctrl+click` several column titles to build up more complicated indexes.

**Note** All of this can be achieved using `ClientDataSet`, but that solution is not as elegant because `ClientDataSet` does not support the `DESC` keyword so you have to create an index collection item to obtain a descending index, something that requires more code. Moreover when changing an ascending index to a descending version, the `ClientDataSet` will rebuild the index, performing an unnecessary and possibly slow operation.

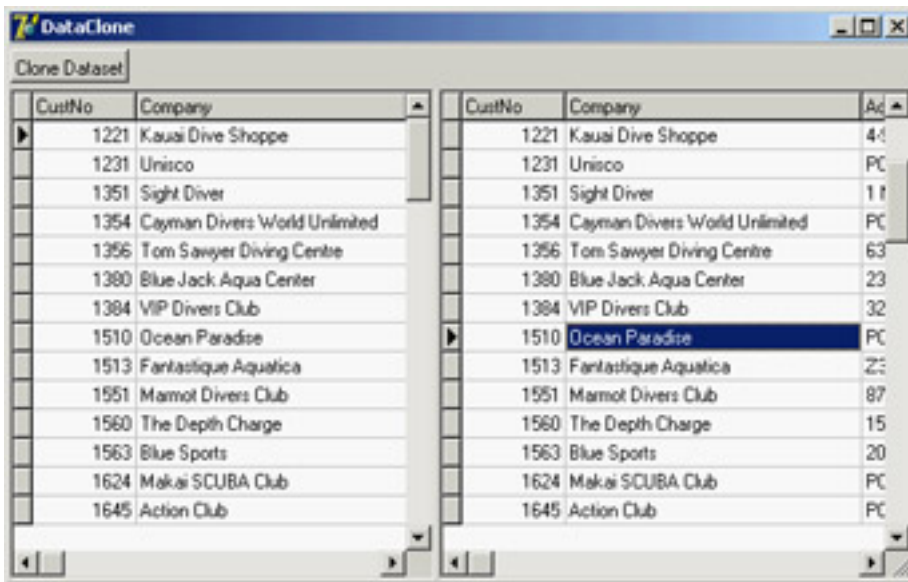
## Cloning

ADO is crammed with features. You can argue that *feature-rich* can translate into *footprint-rich*, but it also translates into more powerful and reliable applications. One such powerful feature is cloning. A cloned recordset is a new recordset that has all the same properties as the original from which it is cloned. First I'll explain how you can create and use a clone, and then I'll explain why clones are so useful.

**Note** The `ClientDataSet` also supports cloning; this feature was not discussed in [Chapter 13](#), "Delphi's Database Architecture."

You can clone a recordset (or, in dbGo-speak, a dataset) using the `Clone` method. You can clone any ADO dataset, but you will use `ADOTable` in this example. The `DataClone` example (see [Figure 15.6](#)) uses two `ADOTable` components, one hooked to database data and the other empty. Both datasets are hooked to a `DataSource` and grid. A single line of code, executed when the user clicks a button, clones the dataset:

```
ADOTable2.Clone(ADOTable1);
```



**Figure 15.6:** The form of the `DataClone` example, with two copies of a dataset (the original and a clone)

This line clones `ADOTable1` and assigns the clone to `ADOTable2`. In the program, you'll see a second view of the data. The two datasets have their own record pointers and other status information, so the clone does not interfere with its original copy. This behavior makes clones ideal for operating on a dataset without affecting the original data. Another interesting feature is that you can have multiple different active records, one for each of the clones—functionality you cannot achieve in Delphi with a single dataset.

**Tip** A recordset must support bookmarks in order to be cloned, so forward-only and dynamic cursors cannot be cloned. You can determine whether a recordset supports bookmarks using the `Supports` method (for example, `ADOTable1.Supports([coBookMark])`). One of the useful side effects of clones is that the bookmarks created by one clone are usable by all other clones.

## Transaction Processing

As we saw in section "[Using Transactions](#)" of [Chapter 14](#), transaction processing allows developers to group individual updates to a database into a single logical unit of work.

ADO's transaction processing support is controlled with the `ADOConnection` component, using the `BeginTrans`, `CommitTrans`, and `RollbackTrans` methods, which have effects similar to those of the corresponding `dbExpress` and `BDE` methods. To investigate ADO transaction processing support, you will build a test program called `TransProcessing`. The program has an `ADOConnection` component with the `ConnectionString` set to the Jet 4.0 OLE DB provider and the `dbdemos.mdb` file. It has an `ADOTable` component hooked to the `Customer` table and a `DataSource` and `DBGrid` for displaying the data. Finally, it has three buttons to execute the following commands:

```
ADOConnection1.BeginTrans;  
ADOConnection1.CommitTrans;  
ADOConnection1.RollbackTrans;
```

With this program, you can make changes to the database table and then roll them back, and they will be rolled back as expected. I emphasize this point because transaction support varies depending on the database and on the OLE DB provider you are using. For example, if you connect to Paradox using the ODBC OLE DB provider, you will receive an error indicating that the database or the OLE DB provider is not capable of beginning a transaction. You can find out the level of transaction processing support you have using the `Transaction DDL` dynamic property of the connection:

```
if ADOConnection1.Properties['Transaction DDL'].Value > DBPROPVAL_TC_NONE then  
    ADOConnection1.BeginTrans;
```

If you are trying to access the same Paradox data using the Jet 4.0 OLE DB provider, you won't receive an error but you also won't be able to roll back your changes, due to limitations of the OLE DB provider.

Another strange difference becomes evident when you're working with Access: If you use the ODBC OLE DB provider, you'll be able to use transactions—but not nested transactions. Opening a transaction when another is active will result in an error. Using the Jet engine, however, Access supports nested transactions.

## Nested Transactions

Using the `TransProcessing` program, you can try this test:

1. Begin a transaction.
2. Change the `ContactName` of the `Around The Horn` record from `Thomas Hardy` to `Dick Solomon`.
3. Begin a nested transaction.
4. Change the `ContactName` of the `Bottom-Dollar Markets` record from `Elizabeth Lincoln` to `Sally Solomon`.
5. Roll back the inner transaction.
6. Commit the outermost transaction.

The net effect is that only the change to the `Around The Horn` record is permanent. If, however, the inner transaction had been committed and the outer transaction rolled back, then the net effect would have been that *none* of the changes were permanent (even the changes in the inner transaction). This is as you would expect, with the only limit being that Access supports five levels of nested transactions.

ODBC does not support nested transactions, the Jet OLE DB provider supports up to five levels of nested transactions, and the SQL Server OLE DB provider doesn't support nesting at all. You might get a different result depending on the version of SQL server or the driver, but the documentation and my experiments with the servers seem to indicate that this is the case. Apparently only the outermost transaction decides whether all the work is committed or rolled back.

## ADOConnection Attributes

There is another issue you should consider if you intend to use nested transactions. The `ADOConnection` component has an `Attributes` property that determines how the connection should behave when a transaction is committed or rolled back. It is a set of `TXActAttributes` that, by default, is empty. `TXActAttributes` contains only two values: `xaCommitRetaining` and `xaAbortRetaining` (this value is often mistakenly written as `xaRollbackRetaining`—a more logical name for it). When `xaCommitRetaining` is included in `Attributes` and a transaction is committed, a new transaction is automatically started. When `xaAbortRetaining` is included in `Attributes` and a transaction is rolled back, a new transaction is automatically started. Thus if you include these values in `Attributes`, a transaction will always be in progress, because when you end one transaction another will always be started.

Most programmers prefer to be in greater control of their transactions and not to allow them to be automatically started, so these values are not commonly used. However, they have a special relevance to nested transactions. If you nest a transaction and set `Attributes` to `[xaCommitRetaining, xaAbortRetaining]`, then the outermost transaction can never be ended. Consider this sequence of events:

1. An outer transaction is started.
2. An inner transaction is started.
3. The inner transaction is committed or rolled back.
4. A new inner transaction is automatically started as a consequence of the `Attributes` property.

The outermost transaction can never be ended, because a new inner transaction will be started when one ends. The conclusion is that the use of the `Attributes` property and the use of nested transactions should be considered mutually exclusive.

## Lock Types

ADO supports four different approaches to locking your data for update: `ItReadOnly`, `ItPessimistic`, `ItOptimistic`, and `ItBatchOptimistic` (there is also an `ItUnspecified` option, but for the reasons mentioned earlier, we will ignore unspecified values). The four approaches are made available through the dataset's `LockType` property. In this section I will provide an overview of the four approaches, and in subsequent sections we will take a closer look at them.

The `ItReadOnly` value specifies that data is read-only and cannot be updated. As such, there is effectively no locking control required, because the data cannot be updated.

The `ItPessimistic` and `ItOptimistic` values offer the same pessimistic and optimistic locking control as the BDE. One important benefit that ADO offers over the BDE in this respect is that the choice of locking control remains yours. If you use the BDE, the decision to use pessimistic or optimistic locking is made for you by your BDE driver. If you use a desktop database such as dBase or Paradox, then the BDE driver uses pessimistic locking; if you use a client/server database such as InterBase, SQL Server, or Oracle, the BDE driver uses optimistic locking.

## Pessimistic Locking

The words *pessimistic* and *optimistic* in this context refer to the developer's expectation of conflict between user updates. Pessimistic locking assumes that there is a high probability that users will attempt to update the same records at the same time and that a conflict is likely. In order to prevent such a conflict, the record is locked when the edit begins. The record lock is maintained until the update is completed or cancelled. A second user who attempts to edit the same record at the same time will fail in their attempt to place their record lock and will receive a "Could not update; currently locked" exception.

This approach to locking will be familiar to developers who have worked with desktop databases such as dBase and Paradox. The benefit is that the user knows that if they can begin editing a record, then they will succeed in saving their update. The disadvantage of pessimistic locking is that the user is in control of when the lock is placed and when it is removed. If the user is skilled with the application, then this lock could be as short as a couple of seconds.

However, in database terms, a couple of seconds is an eternity. On the other hand, the user might begin an edit and go to lunch, and the record would be locked until the user returns. As a consequence, most proponents of pessimistic locking guard against this eventuality by using a Timer or other such device to time out locks after a certain amount of keyboard and mouse inactivity.

Another problem with pessimistic locking is that it requires a server-side cursor. Earlier we looked at cursor locations and saw that they have an impact on the availability of the different cursor types. Now you can see that cursor locations also have an impact on locking types. Later in this chapter, we will discuss more benefits of client-side cursors; if you choose to take advantage of these benefits, then you'll be unable to use pessimistic locking.

Pessimistic locking is an area of dbGo that changed in Delphi 6 (compared to Delphi 5). This section describes the way pessimistic locking works in versions 6 and 7. To highlight how it works, I've built the PessimisticLocking example. It is similar to other examples in this chapter, but the `CursorLocation` property is set to `clUseServer` and the `LockType` property is set to `ltPessimistic`. To use it, run two copies from Windows Explorer and attempt to edit the same record in both running instances of the program: You will fail because the record is locked by another user.

**Team LiB**

◀ PREVIOUS

NEXT ▶



## Updating the Data

One of the reasons people use the ClientDataSet component (or turn to cached updates in the BDE) is to make a SQL join updatable. Consider the following SQL equi-join:

```
SELECT * FROM Orders, Customer
WHERE Customer.CustNo=Orders.CustNo
```

This statement provides a list of orders and the customers that placed those orders. The BDE considers any SQL join to be read-only because inserting, updating, and deleting rows in a join is ambiguous. For example, should the insert of a row into the previous join result in a new product and also a new supplier, or just a new product? The ClientDataSet/Provider architecture allows you to specify a primary update table (and advanced features actually not covered in the book) and also customize the updates' SQL, as we partially saw in [Chapter 14](#) and we'll further explore in [Chapter 16](#), "Multitier DataSnap Applications."

ADO supports an equivalent to cached updates called *batch updates*, which are similar to the BDE approach. In the [next section](#) we will take a closer look at ADO's batch updates, what they can offer you, and why they are so important. However, in this section you won't need them to solve the problem of updating a join, because in ADO, joins are naturally updatable.

For example, the JoinData example is built around an ADODataset component that uses the previous SQL join. If you run it, you can edit one of the fields and save the changes (by moving off the record). No error occurs, because the update has been applied successfully. ADO, compared to the BDE, has taken a more practical approach to the problem. In an ADO join, each field object knows which underlying table it belongs to. If you update a field in the Products table and post the change, then a SQL UPDATE statement is generated to update the field in the Products table. If you change a field in the Products table and a field in the Suppliers table, then two SQL UPDATE statements are generated, one for each table.

The insertion of a row into a join follows a similar behavior. If you insert a row and enter values for the Products table only, then a SQL INSERT statement is generated for the Products table. If you enter values for both tables, two SQL INSERT statements are generated, one for each table. The order in which the statements are executed is important, because the new product might relate to the new supplier, so the new supplier is inserted first.

The biggest problem with the ADO solution can be seen when a row in a join is deleted. The deletion attempt will appear to fail. The exact message you see depends on the version of ADO you are using and the database, but it will be along the lines that you cannot delete the row because other records relate to it. The error message can be confusing. In the current scenario, the error message implies that a product cannot be deleted because there are records that relate to the product, but the error occurs whether the product has any related records or not. The explanation can be found by following the same logic for deletions as for insertions. Two SQL DELETE statements are generated: one for the Suppliers table and then another for the Products table. Contrary to appearances, the DELETE statement for the Products table succeeds. It is the DELETE statement for the Suppliers table that fails, because the supplier cannot be deleted while it still has dependent records.

**Tip** If you are curious about the SQL statements that are generated, and you use SQL Server, you can see these statements using SQL Server Profiler.

Even if you understand how this process works, it's helpful to look at this problem through the users' eyes. From their point of view, when users delete a row in the grid, I would wager that 99 percent of them intend to delete just the product—not both the product and the supplier. Fortunately, you can achieve this result using another dynamic property—in this case, the Unique Table dynamic property. You can specify that deletes refer to just the Products table and not to Suppliers using the following line of code:

```
ADOQuery1.Properties['Unique Table'].Value := 'Products';
```

This value cannot be assigned at design time, so the next best alternative is to place this line in the form's OnCreate event.

## Batch Updates

When you use batch updates, any changes you make to your records can be made in memory; later, the entire "batch" of changes can be submitted as one operation. This approach offers some performance benefits, but there are more practical reasons why this technology is a necessity: The user might not be connected to the database at the time they make their updates. This would be the case in a briefcase application (which we will return to in the section "[The Briefcase Model](#)"), but it can also be the case in web applications that use another ADO technology, Remote Data Services (RDS).

You can enable batch updates in any ADO dataset by setting `LockType` to `ltBatchOptimistic` before the dataset is opened. In addition, you will need to set the `CursorLocation` to `clUseClient`, because batch updates are managed by ADO's cursor engine. Hereafter, changes are all made to a *delta* (a list of changes). The dataset looks to all intents and purposes as if the data has changed, but the changes have only been made in memory; they have not been applied to the database. To make the changes permanent, use `UpdateBatch` (equivalent to cached updates' `ApplyUpdates`):

```
ADODataset1.UpdateBatch;
```

To reject the entire batch of updates, use either `CancelBatch` or `CancelUpdates`. There are many similarities in method and property names between ADO batch updates, BDE cached updates, and `ClientDataSet.UpdateStatus`, for example, can be used exactly the same way as for cached updates to identify records according to whether they have been inserted, updated, deleted, or unmodified. This approach is particularly useful for highlighting records in different colors in a grid or showing their status on a status bar. Some differences between the syntaxes are slight, such as changing `RevertRecord` to `CancelBatch(arCurrent)`; others require more effort.

One useful cached update feature that is not present in ADO batch updates is the dataset's `UpdatesPending` property. This property is true if changes have been made but not yet applied. It's particularly useful in a form's `OnCloseQuery` event:

```
procedure TForm1.FormCloseQuery(  
    Sender: TObject; var CanClose: Boolean);  
begin  
    CanClose := True;  
    if ADODataset1.UpdatesPending then  
        CanClose := (MessageDlg('Updates are still pending' #13 +  
            'Close anyway?', mtConfirmation, [mbYes, mbNo], 0) = mrYes);  
end;
```

However, with a little knowledge and ingenuity you can implement a suitable `ADOUpdatesPending` function. The necessary knowledge is that ADO datasets have a property called `FilterGroup`, which is a kind of filter. Unlike a dataset's `Filter` property, which filters the data based on a comparison of the data against a condition, `FilterGroup` filters based on the status of the record. One such status is `fgPendingRecords`, which includes all records that have been modified but not yet applied. So, to allow the user to look through all the changes they have made so far, you need only execute two lines:

```
ADODataset1.FilterGroup := fgPendingRecords;  
ADODataset1.Filtered := True;
```

Naturally, the result set will now include the records that have been deleted. The effect you will see is that the fields are left blank, which is not very helpful because you don't know which record has been deleted. (This was not the behavior in the first version of ADOExpress, which displayed the field values of deleted records.)

The ingenuity you need in order to solve the `UpdatesPending` problem involves clones, discussed earlier. The `ADOUpdatesPending` function will set the `FilterGroup` to restrict the dataset to only those changes that have not yet been applied. All you need to do is see whether there are any records in the dataset once the `FilterGroup` has been applied. If there are, then some updates are pending. However, if you do this with the actual dataset, then the setting of the `FilterGroup` will move the record pointer, and the user interface will be updated. The best solution is to



use a clone:

```
function ADOUpdatesPending(ADODataset: TCustomADODataset): boolean;  
var  
    Clone: TADODataset;  
begin  
    Clone := TADODataset.Create(nil);  
    try  
        Clone.Clone(ADODataset);  
        Clone.FilterGroup := fgPendingRecords;  
        Clone.Filtered := True;  
        Result := not (Clone.BOF and Clone.EOF);  
        Clone.Close;  
    finally  
        Clone.Free;  
    end;  
end;
```

In this function, you clone the original dataset, set the `FilterGroup`, and check to see whether the dataset is at both beginning of the file and also the end of the file. If it is, then no records are pending.

## Optimistic Locking

Earlier we looked at the `LockType` property and saw how pessimistic locking works. In this section, we'll look at optimistic locking, not only because it is the preferred locking type for medium- to high-throughput transactions, but also because it is the locking scheme employed by batch updates.

Optimistic locking assumes there is a low probability that users will attempt to update the same records at the same time and that a conflict is unlikely. As such, the attitude is that all users can edit any record at any time, and you deal with the consequences of conflicts between different users' updates to the same records when the changes are saved. Thus, conflicts are considered an exception to the rule. This means there are no controls to prevent two users from editing the same record at the same time. The first user to save their changes will succeed; the second user's attempt to update the same record might fail. This behavior is essential for briefcase applications (discussed later in the chapter) and web applications, where there is no permanent connection to the database and, therefore, no way to implement pessimistic locking. In contrast with pessimistic locking, optimistic locking has the additional considerable benefit that resources are consumed only momentarily; therefore, the average resource usage is much lower, making the database more scalable.

Let's consider an example. Assume you have an `ADODataset` connected to the `Customer` table of the `dbdemos.mdb` database, with `LockType` set to `ltBatchOptimistic`, and the contents are displayed in a grid. Assume that you also have a button to call `UpdateBatch`. Run the program twice (it is the `BatchUpdates` example if you don't want to rebuild it) and begin editing a record in the first copy of the program. Although for the sake of simplicity I'll demonstrate a conflict using just a single machine, the scenario and subsequent events are unchanged when using multiple machines:

1. Choose the Bottom-Dollar Markets company in Canada and change the name to Bottom-Franc Markets.
2. Save the change, move off the record to post it, and click the button to update the batch.
3. In the second copy of the program, locate the same record and change the company name to Bottom-Pound Markets.
4. Move off the record and click the button to update the batch. It will fail.

As with many ADO error messages, the exact message you receive will depend not only on the version of ADO you are using but also on how closely you followed the example. In ADO 2.6, the error message is "Row cannot be located for updating. Some values may have been changed since it was last read." This is the nature of optimistic locking. The update to the record is performed by executing the following SQL statement:

```
UPDATE CUSTOMER SET CompanyName="Bottom-Pound Markets"
```

```
WHERE CustomerID="BOTTM" AND CompanyName="Bottom-Dollar Markets"
```

The number of records affected by this `UPDATE` statement is expected to be one, because it locates the original record using the primary key and the contents of the `CompanyName` field as it was when the record was first read. In this example, however, the number of records affected by the `UPDATE` statement is zero. This result can occur only if the record has been deleted, the record's primary key has changed, or the field that you are changing was changed by someone else. Hence, the update fails.

If the "second user" had changed the `ContactName` field and not the `CompanyName` field, then the `UPDATE` statement would have looked like this:

```
UPDATE CUSTOMER SET ContactName="Liz Lincoln"  
WHERE CustomerID="BOTTM" AND ContactName="Elizabeth Lincoln"
```

In the example scenario, this statement would have succeeded because the other user didn't change the primary key or the contact name. This behavior is similar to the BDE with the *update where changed* update mode. The `UpdateMode` property of the BDE in ADO is replaced by the `Update Criteria` dynamic property of a dataset. The following list shows the possible values that can be assigned to this dynamic property:

Constant	Locate Records By
<code>adCriteriaKey</code>	Primary key columns only
<code>adCriteriaAllCols</code>	All columns
<code>adCriteriaUpdCols</code>	Primary key columns and changed columns only
<code>adCriteriaTimeStamp</code>	Primary key columns and a timestamp column only

Don't fall into the trap of thinking that one of these settings is better than another for your whole application. In practice, your choice of setting will be influenced by the contents of each table. Say that the `Customer` table has just `CustomerID`, `Name`, and `City` fields. In this case, the update of any one of these fields is logically not mutually exclusive with the update of any of the other fields, so a good choice for this table would be `adCriteriaUpdCols` (the default). If, however, the `Customer` table included a `PostalCode` field, then the update of a `PostalCode` field would be mutually exclusive with the update of the `City` field by another user (because if the city changes, then so should the postal code, and possibly vice versa). In this case, you could argue that `adCriteriaAllCols` would be a safer solution.

Another issue to be aware of is how ADO deals with errors during the update of multiple records. Using the BDE's cached updates and `ClientDataSet`, you can use the `OnUpdateError` event to handle each update error as the error occurs and resolve the problem before moving on to the next record. In ADO, you cannot establish such a dialog. You can monitor the progress and success or failure of the updating of the batch using the dataset's `OnWillChangeRecord` and `OnRecordChangeComplete`, but you cannot revise the record and resubmit it during this process as you can with the BDE and `ClientDataSet`. There's more: If an error occurs during the update process, the updating does not stop. It continues to the end, until all updates have been applied or have failed. This process can produce an unhelpful and incorrect error message. If more than one record cannot be updated, or the single record that failed is not the last record to be applied, then the error message in ADO 2.6 is "Multiple-step OLE DB operation generated errors. Check each OLE DB status value, if available. No work was done." The last sentence is the problem; it states that "No work was done," but this is incorrect. It is true that no work was done on the record that failed, but other records were successfully applied, and their updates stand.

## Resolving Update Conflicts

As a consequence of the nature of applying updates, the approach that you need to take to update the batch is to update the batch, let the individual records fail, and then deal with the failed records once the process is over. You can determine which records have failed by setting the dataset's `FilterGroup` to `fgConflictingRecords`:

```
ADODataset1.FilterGroup := fgConflictingRecords;  
ADODataset1.Filtered := True;
```

For each failed record, you can inform the user of three critical pieces of information about each field using the

following TField properties:

Property	Description
NewValue	The value this user changed it to
CurValue	The new value from the database
OldValue	The value when first read from the database

Users of the ClientDataSet component will be aware of the handy `ReconcileErrorForm` dialog, which wraps up the process of showing the user the old and new records and allows them to specify what action to take. Unfortunately, there is no ADO equivalent to this form, and `TReconcileErrorForm` has been written with `ClientDataSet` so much in mind that it is difficult to convert it for use with ADO datasets.

I'll point out one last gotcha about using these `TField` properties: They are taken straight from the underlying ADO Field objects to which they refer. This means, as is common in ADO, that you are at the mercy of your chosen OLE DB provider to support the features you hope to use. All is well for most providers, but the Jet OLE DB provider returns the same value for `CurValue` as it does for `OldValue`. In other words, if you use Jet, you cannot determine the value to which the other user changed the field unless you resort to your own measures. Using the OLEDB provider for SQL Server, however, you can access the `CurValue` only after calling the `Resync` method of the dataset with the `AffectRecords` parameter set to `adAffectGroup` and `ResyncValues` set to `adResyncUnderlyingValues`, as in the following code:

```
adoCustomers.FilterGroup := fgConflictingRecords;  
adoCustomers.Filtered := true;  
adoCustomers.Recordset.Resync(adAffectGroup, adResyncUnderlyingValues);
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Disconnected Recordsets

This knowledge of batch updates allows you to take advantage of the next ADO feature: disconnected recordsets. A *disconnected recordset* is a recordset that has been disconnected from its connection. This feature is impressive because the user cannot tell the difference between a regular recordset and a disconnected one; their feature sets and behavior are almost identical. To disconnect a recordset from its connection, you must set the `CursorLocation` to `clUseClient` and the `LockType` to `ltBatchOptimistic`. You then tell the dataset that it no longer has a connection:

```
ADODataset1.Connection := nil;
```

Hereafter, the recordset will continue to contain the same data, support the same navigational features, and allow records to be added, edited, and deleted. The only relevant difference is that you cannot update the batch because you need to be connected to the server to update the server. You can reconnect the connection (and use `UpdateBatch`) as follows:

```
ADODataset1.Connection := ADOConnection1;
```

This feature is also available to the BDE and other database technologies by switching over to `ClientDataSets`, but the beauty of the ADO solution is that you can build your entire application using `dbGo` dataset components and be unaware of disconnected recordsets. When you discover this feature and want to take advantage of it, you can continue to use the same components you always used.

You might want to disconnect your recordsets for two reasons:

- To keep the total number of connections lower
- To create a briefcase application

I'll discuss keeping down the number of connections in this section and return to briefcase applications later.

Most regular client/server business applications open tables and maintain a permanent connection to their database while the table is open. However, there are usually only two reasons to be connected to the database: to retrieve data and to update data. Suppose you change your regular client/server application so that after the table is opened and the data is retrieved, the dataset is disconnected from the connection and the connection is dropped; your user will be none the wiser, and the application will not need to maintain an open database connection. The following code shows the two steps:

```
ADODataset1.Connection := nil;  
ADOConnection1.Connected := False;
```

The only other point at which a connection is required is when the batch of updates needs to be applied. The update code looks like this:

```
ADOConnection1.Connected := True;  
ADODataset1.Connection := ADOConnection1;  
try  
    ADODataset1.UpdateBatch;  
finally  
    ADODataset1.Connection := nil;  
    ADOConnection1.Connected := False;  
end;
```

If you followed this approach throughout the application, the average number of open connections at any one time would be minimal—the connections would be open only for the brief time they were required. The consequence of this change is scalability; the application can cope with significantly more simultaneous users than an application that maintains an open connection. The downside is that reopening the connection can be a lengthy process on some (but not all) database engines, so the application will be slower to update the batch.

## Connection Pooling

All this talk about dropping and reopening connections brings us to the subject of connection pooling. Connection pooling—not to be confused with session pooling—allows connections to a database to be reused once you have finished with them. This process happens automatically; if your OLE DB provider supports it and it is enabled, no action is necessary for you to take advantage of connection pooling.

There is a single reason to pool your connections: performance. The problem with database connections is that it can take time to establish a connection. In a desktop database such as Access, this time is typically brief. However, in a client/server database such as Oracle used on a network, this time could be measured in seconds. It makes sense to promote the reuse of such an expensive (in performance terms) resource.

With ADO connection pooling enabled, ADO Connection objects are placed in a pool when the application "destroys" them. Subsequent attempts to create an ADO connection will automatically search the connection pool for a connection with the same connection string. If a suitable connection is found, it is reused; otherwise, a new connection is created. The connections themselves stay in the pool until they are reused, the application closes, or they time out. By default, connections will time out after 60 seconds, but from MDAC 2.5 onward you can set this time-out period using the `HKEY_CLASSES_ROOT\CLSID\ <ProviderCLSID>\SPTimeout` Registry key. The connection pooling process occurs seamlessly, without the intervention or knowledge of the developer. This process is similar to the BDE database pooling under Microsoft Transaction Server (MTS) and COM+, with the important exception that ADO performs its own connection pooling without the aid of MTS or COM+.

By default, connection pooling is enabled on all MDAC OLE DB providers for relational databases (including SQL Server and Oracle), with the notable exception of the Jet OLE DB provider. If you use ODBC, you should choose between ODBC connection pooling and ADO connection pooling, but you should not use both. From MDAC 2.1 on, ADO connection pooling is enabled and ODBC is disabled.

**Note** Connection pooling does not occur on Windows 95 regardless of the OLE DB provider.

To be comfortable with connection pooling, you need to see the connections being pooled and timed out. Unfortunately, no adequate ADO connection pool spying tools are available at the time of this writing; but you can use SQL Server's Performance Monitor, which can accurately spy on SQL Server database connections.

You can enable or disable connection pooling either in the Registry or in the connection string. The key in the Registry is `OLEDB_SERVICES`, which can be found at `HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>`. It is a bit mask that allows you to disable several OLE DB services, including connection pooling, transaction enlistment, and the cursor engine. To disable connection pooling using the connection string, include `" ;OLE DB Services=-2"` at the end of the connection string. To enable connection pooling for the Jet OLE DB provider, you can include `" ;OLE DB Services=-1"` at the end of the connection string, which enables all OLE DB services.

## Persistent Recordsets

The persistent recordset is a useful feature that contributes to the briefcase model (discussed in the [next section](#)). Persistent recordsets allow you to save the contents of any recordset to a local file, which can be loaded later. In addition to aiding with the briefcase model, this feature allows developers to create true single-tier applications—you can deploy a database application without having to deploy a database. This makes for a very small footprint on your client's machine.

You can "persist" your datasets using the `SaveToFile` method:

```
ADODataset1.SaveToFile('Local.ADTG');
```

This method saves the data and its delta in a file on your hard disk. You can reload this file using the `LoadFromFile` method, which accepts a single parameter indicating the file to load. The format of the file is Advanced Data Table Gram (ADTG), which is a proprietary Microsoft format. It does, however, have the advantage of being very efficient. If you prefer, you can save the file as XML by passing a second parameter to `SaveToFile`:

```
ADODataset1.SaveToFile('Local.XML', pfXML);
```

However, ADO does not have a built-in XML parser (as the ClientDataSet does), so it must use the MSXML parser. Your user must either install Internet Explorer 5 or later or download the MSXML parser from the Microsoft website.

If you intend to persist your files locally in XML format, be aware of a few disadvantages:

- Saving and loading XML files is slower than saving and loading ADTG files.
- ADO's XML files (and XML files in general) are significantly larger than their ADTG counterparts (XML files are typically twice as large as their ADTG counterparts).
- ADO's XML format is specific to Microsoft, like most companies' XML implementations. This means the XML generated in ADO is not readable by the ClientDataSet and vice versa. Fortunately this problem can be overcome using Delphi's XMLTransform component, which can be used to translate between different XML structures.

If you intend to use these features solely for single-tier applications and not as part of the briefcase model, then you can use an ADODataset component and set its `CommandType` to `cmdFile` and its `CommandText` to the name of the file. Doing so will save you the effort of calling `LoadFromFile` manually. However, you will still have to call `SaveToFile`. In a briefcase application this approach is too limiting, because the dataset can be used in two different modes.

## The Briefcase Model

Using this knowledge of batch updates, disconnected recordsets, and persistent recordsets, you can take advantage of the *briefcase model*. The idea behind the briefcase model is that your users want to be able to use your application while they are on the road—they want to take the same application they use on their office desktops and use it on their laptops at client sites. Traditionally, the problem with this scenario is that when your users are at client sites, they are not connected to the database server, because the database server is running on the network back at their office. Consequently, there is no data on the laptop (and the data cannot be updated anyway).

This is where your newfound understanding comes in handy. Assume the application has been written; the user has requested a new briefcase enhancement, and you have to retrofit it into your existing application. You need to add a new option for your users to allow them to prepare the briefcase application by executing `SaveToFile` for every table in the database. The result is a collection of ADTG or XML files that mirror the contents of the database. These files are then copied to the laptop, where a copy of the application has previously been installed.

The application needs to be sensitive to whether it is running locally or connected to the network. You can determine this by attempting to connect to the database and seeing whether the connection fails, by detecting the presence of a local briefcase file, or by creating a flag of your own design. If the application is running in briefcase mode, then it needs to use `LoadFromFile` for each table instead of setting `Connected` to `True` for the ADOConnections and `Active` to `True` for the ADO datasets. Thereafter, the briefcase application needs to use `SaveToFile` instead of `UpdateBatch` whenever data is saved. When the user returns to the office, they need to follow an update process that loads each table from its local file, connects the dataset to the database, and applies the changes using `UpdateBatch`.

**Tip** To see a complete implementation of the briefcase model, refer to the BatchUpdates example mentioned earlier.

## A Word on ADO.NET

ADO.NET is part of Microsoft's new .NET architecture—the company's redesign of application development tools to better suit the needs of web development. ADO.NET is a significant evolution of ADO. It looks at the problems of web development and addresses shortcomings of ADO's solution. The problem with ADO's solution is that it is based on COM. For one- and two-tier applications, COM imposes few problems, but in the world of web development, it is unacceptable as a transport mechanism. COM suffers from three primary problems that limit its use in web development: It (mostly) runs only on Windows, the transmission of recordsets from one process requires COM marshalling, and COM calls cannot penetrate corporate firewalls. ADO.NET's solution to all these problems is to use XML.

Some other redesign issues focus on breaking the ADO recordset into separate classes. The resulting classes are adept at solving a single problem instead of multiple problems. For example, the ADO.NET class currently called `DataSetReader` is similar to a read-only, forward-only, server-side recordset and, as such, is best suited to reading a result set very quickly. A `DataTable` is most like a disconnected, client-side recordset. A `DataRelation` shares similarities with the `MSDataShape` OLE DB provider. You can see that your knowledge of how ADO works is of great benefit in understanding the basic principles of ADO.NET.



## What's Next?

This chapter described ActiveX Data Objects (ADO) and dbGo, the set of Delphi components for accessing the ADO interfaces. You've seen how to take advantage of Microsoft Data Access Components (MDAC) and various server engines, and I've described some of the benefits and hurdles you'll encounter in using ADO.

[Chapter 16](#) will take you into the world of Delphi's DataSnap architecture, which is used to develop custom client and server applications in a three-tier environment. You can do this using ADO, but because this is a book about Delphi I prefer to show you the *native* solution to the problem. After we delve into DataSnap, we will continue examining Delphi's database architecture by covering the development of custom data-aware controls and dataset components.



## Chapter 16: Multitier DataSnap Applications

### Overview

Large companies often have needs that are broader than applications using local database and SQL servers can meet. In the past few years, Borland Software Corporation has been addressing the needs of large corporations, and it even temporarily changed its own name to Inprise to underline this enterprise focus. The name was eventually changed back to Borland, but the focus on enterprise development remains.

Delphi targets many different technologies: three-tier architectures based on Windows NT and DCOM, TCP/IP and socket applications, and—most of all—SOAP- and XML-based web services. This chapter focuses on database-oriented multitier architectures; XML-oriented solutions will be discussed in [Chapters 22](#) and [23](#), which are devoted to XML, SOAP, and web services.

Before proceeding, I should emphasize two important elements. First, the tools to support this kind of development are available only in the Enterprise version of Delphi; and second, with Delphi 7 you don't have to pay a deployment fee for DataSnap applications. You buy the development environment and then deploy your programs on as many servers as you want, without owing Borland any money. This is a very significant change (the most significant in Delphi 7) to the distribution policy of DataSnap, which used to require a per-server fee (initially very high, then significantly lowered over time). This new deployment license will certainly increase the appeal of DataSnap to developers, which is a good reason to cover it in some detail.

## One, Two, Three Levels in Delphi's History

Initially, database PC applications were client-only solutions: The program and the database files were on the same computer. From there, adventuresome programmers moved the database files onto a network file server. The client computers still hosted the application software and the entire database engine, but the database files were now accessible to several users at the same time. You can still use this type of configuration with a Delphi application and Paradox files (or, of course, Paradox itself), but the approach was much more widespread just few years ago.

The next big transition was to client/server development, embraced by Delphi since its first version. In the client/server world, the client computer requests the data from a server computer, which hosts both the database files and a database engine to access them. This architecture downplays the role of the client, but it also reduces the requirements for processing power on the client machine. Depending on how the programmers implement client/server, the server can do most (if not all) of the data processing. In this way, a powerful server can provide data services to several less powerful clients.

Naturally, there are many other reasons for using centralized database servers, such as concern for data security and integrity, simpler backup strategies, central management of data constraints, and so on. The database server is often called a SQL server, because SQL is the language most commonly used for making queries into the data; but it may also be called a RDBMS (relational database management system), reflecting the fact that the server provides tools for managing the data, such as support for backup and replication.

Of course, some applications you build may not need the benefits of a full RDBMS, so a simple client-only solution might be sufficient. On the other hand, you might need some of the robustness of a RDBMS system, but on a single, isolated computer. In this case, you can use a local version of a SQL server, such as InterBase. Traditional client/server development is done with a two-tier architecture. However, if the RDBMS is primarily performing data storage instead of data- and number-crunching, the client might contain both user interface code (formatting the output and input with customized reports, data-entry forms, query screens, and so on) and code related to managing the data (also known as *business rules*). In this case, it's generally a good idea to try to separate these two sections of the program and build a logical three-tier architecture. The term *logical* here means that there are still just two computers (that is, two physical tiers), but you've now partitioned the application into three distinct elements.

Delphi 2 introduced support for a logical three-tier architecture with data modules. As you should know by now, a *data module* is a nonvisual container for the data access components of an application (or indeed any other nonvisual components), but it often includes several handlers for database-related events. You can share a single data module among several different forms and provide different user interfaces for the same data; there might be one or more data-input forms, reports, master/detail forms, and various charting or dynamic output forms.

The logical three-tier approach solves many problems, but it also has a few drawbacks. First, you must replicate the data-management portion of the program on different client computers; doing so may hamper performance, but the bigger issue is the complexity it adds to code maintenance. Second, when multiple clients modify the same data, there's no simple way to handle the resulting update conflicts. Finally, for logical three-tier Delphi applications, you must install and configure the database engine (if any) and SQL server client library on every client computer.

The next logical step up from client/server is to move the data-module portion of the application to a separate server computer and design all the client programs to interact with it. This is exactly the purpose of remote data modules, which were introduced in Delphi 3. Remote data modules run on a server computer—generally called the *application server*. The application server in turn communicates with the DBMS (which can run on the application server or on another dedicated computer). Therefore, the client machines don't connect to the SQL server directly, but indirectly via the application server.

At this point there is a fundamental question: Do we still need to install the database access software? The traditional Delphi client/server architecture (even with three logical tiers) requires you to install the database access on each client, which is quite troublesome when you must configure and maintain hundreds of machines. In the physical three-tier architecture, you need to install and configure the database access only on the application server, not on the client computers. Because the client programs have only user interface code and are extremely simple to install, they now fall into the category of so-called *thin clients*. To use marketing-speak, we might even call this a *zero-configuration thin-client architecture*. But let's focus on technical issues instead of marketing terminology.

## The Technical Foundation of DataSnap

When Borland introduced this physical multitier architecture in Delphi, it was called MIDAS (Middle-tier Distributed Application Services). For example, Delphi 5 included the third version of this technology, MIDAS 3. Afterward Borland renamed the technology *DataSnap* and extended its capabilities.

DataSnap requires the installation of specific libraries on the server (actually the middle-tier computer), which provides your client computers with the data extracted from the SQL server database or other data sources. DataSnap does not require a SQL server for data storage. DataSnap can serve up data from a wide variety of sources, including SQL, other DataSnap servers, or data computed on the fly.

As you would expect, the client side of DataSnap is extremely thin and easy to deploy. The only file you need is `Midas.dll`, a small DLL that implements the `ClientDataSet` and `RemoteServer` components and provides the connection to the application server. As an alternative to distributing the DLL, you can embed the code of this library in your executable file by including the `MidasLib` unit in your `uses` statements, as discussed in [Chapter 13](#), "Delphi's Database Architecture."

## The IAppServer Interface

The two sides of a DataSnap application communicate using the `IAppServer` interface; this interface's definition appears in [Listing 16.1](#). You'll seldom need to call the methods of the `IAppServer` interface directly, because Delphi includes components implementing this interface on the server side applications and components calling the interface on the client side applications. These components simplify the support of the `IAppServer` interface and at times even hide it completely. In practice, the server will make objects implementing this interface available to the client, possibly along with other custom interfaces.

### Listing 16.1: The Definition of the IAppServer Interface

---

```
type
  IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant;
      MaxErrors: Integer; out ErrorCount: Integer;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer;
      out RecsOut: Integer; Options: Integer; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString;
      Data: OleVariant): OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
      RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString;
      const CommandText: WideString; var Params: OleVariant;
      var OwnerData: OleVariant); safecall;
  end;
```

---

**Note** A DataSnap server exposes an interface using a COM type library, a technology covered in [Chapter 12](#), "From COM to COM+."

## The Connection Protocol

DataSnap defines only the higher-level architecture and can use different technologies for moving the data from the middle tier to the client side. DataSnap supports many different protocols, including the following:

**Distributed COM (DCOM) and Stateless COM (MTS or COM+)** DCOM is directly available in Windows NT/2000/XP and 98/Me, and it requires no additional run-time applications on the server. DCOM is basically an extension of COM technology that allows a client application to use server objects that exist and execute on a separate computer. The DCOM infrastructure allows you to use stateless COM objects, available in the COM+ and in the older MTS (Microsoft Transaction Server) architectures. Both COM+ and MTS provide features such as security, component management, and database transactions, and are available in Windows NT/2000/XP and in Windows 98/Me.

Due to the complexity of DCOM configuration and its problems in passing through firewalls, even Microsoft is abandoning DCOM in favor of SOAP-based solutions.

**TCP/IP Sockets** These are available on most systems. Using TCP/IP, you might distribute clients over the Web, where DCOM cannot be taken for granted, and you'll have far fewer configuration headaches. To use sockets, the middle-tier computer must run the `ScktSrvr.exe` application provided by Borland, a single program that can run either as an application or as a service. This program receives the client requests and forwards them to the remote data module (executing on the same server) using COM. Sockets provide no protection against failure on the client side, because the server is not informed and might not release resources when a client unexpectedly shuts down.

**HTTP and SOAP** The use of HTTP as a transport protocol over the Internet simplifies connections through firewalls or proxy servers (which generally don't like custom TCP/IP sockets). You need a specific web server application, `httpsrvr.dll`, which accepts client requests and creates the proper remote data modules using COM. These web connections can also use SSL security. Finally, web connections based on HTTP transport can use DataSnap object-pooling support.

**Note** The DataSnap HTTP transport can use XML as the data packet format, enabling any platform or tool that can read XML to participate in a DataSnap architecture. This is an extension of the original DataSnap data packet format, which is also platform independent. The use of XML over HTTP is also the foundation of SOAP. There's more on SOAP in DataSnap in [Chapter 23](#), "Web Services and SOAP."

Until Delphi 6, you could also use CORBA (Common Object Request Broker Architecture) as a transport mechanism for DataSnap applications. Due to compatibility issues with the newer versions of Borland's VisiBroker CORBA solution, this feature has been discontinued in Delphi 7.

Finally, notice that as an extension to this architecture, you can transform the data packets into XML and deliver them to a web browser. In this case, you basically have one extra tier: the web server gets the data from the middle tier and delivers it to the client. I'll discuss this architecture, called Internet Express, in [Chapter 22](#), "Using XML Technologies."

## Providing Data Packets

The entire Delphi multitier data-access architecture centers around the idea of *data packets*. In this context, a data packet is a block of data that moves from the application server to the client or from the client back to the server. Technically, a data packet is a sort of subset of a dataset. It describes the data it contains (usually a few records of data), and it lists the names and types of the data fields. Even more important, a data packet includes the constraints—that is, the rules to be applied to the dataset. You'll typically set these constraints in the application server, and the server sends them to the client applications along with the data.

All communication between the client and the server occurs by exchanging data packets. The provider component on the server manages the transmission of several data packets within a big dataset, with the goal of responding faster to the user. As the client receives a data packet in a `ClientDataSet` component, the user can edit the records it contains. As mentioned earlier, during this process the client also receives and checks the constraints, which are applied during the editing operations.

When the client has updated the records and sends back a data packet, that packet is known as a *delta*. The delta packet tracks the difference between the original records and the updated ones, recording all the changes the client requested from the server. When the client asks to apply the updates to the server, it sends the delta to the server, and the server tries to apply each of the changes. I say *tries* because if a server is connected to several clients, the data may have changed already, and the update request may fail.

Because the delta packet includes the original data, the server can quickly determine if another client has already changed the data. If so, the server fires an `OnReconcileError` event, which is one of the vital elements for thin-client applications. In other words, the three-tier architecture uses an update mechanism similar to the one Delphi uses for cached updates. As you saw in [Chapter 14](#), "Client/Server with dbExpress," the `ClientDataSet` manages data in a memory cache; it typically reads only a subset of the records available on the server side, loading more elements only as they're needed. When the client updates records or inserts new ones, it stores these pending changes in another local cache on the client, the *delta cache*.

The client can also save the data packets to disk and work offline, thanks to the MyBase support discussed in [Chapter 13](#). Even error information and other data moves using the data packet protocol, so it is truly one of the foundation elements of this architecture.

**Note** It's important to remember that data packets are protocol-independent. A data packet is merely a sequence of bytes, so anywhere you can move a series of bytes, you can move a data packet. This functionality was provided to make the architecture suitable for multiple transport protocols (including DCOM, HTTP, and TCP/IP) and for multiple platforms.

## Delphi Support Components (Client-Side)

Now that we've examined the general foundations of Delphi's three-tier architecture, let's focus on the components that support it. For developing client applications, Delphi provides the `ClientDataSet` component, which provides all the standard dataset capabilities and embeds the client side of the `IAppServer` interface. In this case, the data is delivered through the remote connection.

The connection to the server application is made via another component you'll also need in the client application. You should use one of the three specific connection components (available in the [DataSnap](#) page):

- The `DCOMConnection` component can be used on the client side to connect to a DCOM and MTS server, located either on the current computer or on another computer indicated by the `ComputerName` property. The connection is with a registered object having a given `ServerGUID` or `ServerName`.
- The `SocketConnection` component can be used to connect to the server via a TCP/IP socket. You should indicate the IP address or the host name, and the GUID of the server object (in the `InterceptGUID` property). This connection component has an extra property, `SupportCallbacks`, which you can disable if you are not using callbacks and want to deploy your program on Windows 95 client computers that don't have Winsock 2 installed.

**Note** In the [WebServices](#) page, you can also find the `SoapConnection` component, which requires a specific type of server and will be discussed in [Chapter 23](#).

- The `WebConnection` component is used to handle an HTTP connection that can easily get through a firewall. You should indicate the URL where your copy of `httpsrvr.dll` is located and the name or GUID of the remote object on the server.

A few more client-side components were added to the [DataSnap](#) architecture in Delphi 6, mainly for managing connections:

- The `ConnectionBroker` component can be used as an alias of an actual connection component, which is useful when you have a single application with multiple client datasets. To change the physical connection of each dataset, you only need to change the `Connection` property of the `ConnectionBroker`. You can also use the events of this virtual connection component in place of those of the actual connections, so you don't have to change any code if you change the data transport technology. For the same reason, you can refer to the

`AppServer` object of the `ConnectionBroker` instead of the corresponding property of a physical connection.

- The `SharedConnection` component can be used to connect to a secondary (or child) data module of a remote application, piggy-backing on an existing physical connection to the main data module. In other words, an application can connect to multiple data modules of the server with a single, shared connection.
- The `LocalConnection` component can be used to target a local dataset provider as the source of the data packet. The same effect can be obtained by hooking the `ClientDataSet` directly to the provider. However, using the `LocalConnection`, you can write a local application with the same code as a complete multitier application, using the `IAppServer` interface of the "fake" connection. Doing so will make the program easier to scale up, compared to a program with a direct connection.

A few other components of the `DataSnap` page relate to the transformation of the `DataSnap` data packet into custom XML formats. These components (`XMLTransform`, `XMLTransformProvider`, and `XMLTransformClient`) will be discussed in [Chapter 22](#).

## Delphi Support Components (Server-Side)

On the server side (actually the middle tier), you'll need to create an application or a library that embeds a remote data module, a special version of the `TDataModule` class. A second alternative is the use of a specialized remote data module for transactional COM. In the Multitier page of the New Items dialog box (obtained from the File → New → Other menu) there are specific wizards to create both types of remote data module.

The only specific component you need on the server side is the `DataSetProvider`. You need one of these components for every table or query you want to make available to the client applications, which will then use a separate `ClientDataSet` component for every exported dataset. The `DataSetProvider` was introduced in [Chapter 13](#).

Team LiB

◀ PREVIOUS

NEXT ▶



## Building a Sample Application

Now you're ready to build a sample program. Doing so will let you observe some of the components I've just described in action, and will also allow you to focus on some other problems, shedding light on other pieces of the Delphi multitier puzzle. I'll build the client and application server portions of a three-tier application in two steps. The first step will simply test the technology using a minimum of elements. These programs will be very simple.

From that point, I'll add more power to the client and the application server. In each example, I'll display data from a local InterBase table using dbExpress and set up everything to allow you to test the programs on a stand-alone computer. I won't cover the steps you have to follow to install the examples on multiple computers with various technologies—that would be the subject of at least one other book.

### The First Application Server

The server side of the basic example is easy to build. Simply create a new application and add a remote data module to it using the corresponding icon in the Multitier page of the Object Repository. The Remote Data Module Wizard (see [Figure 16.1](#)) will ask you for a class name and the instancing style. When you enter a class name, such as `TAppServerOne`, and click the OK button, Delphi will add a data module to the program. This data module will have the usual properties and events, but its class will have the following Delphi language declaration:

```
type
  TAppServerOne = class(TRemoteDataModule, IAppServerOne)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
  public
    { Public declarations }
  end;
```

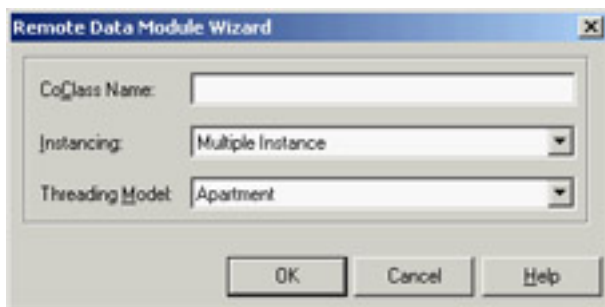


Figure 16.1: The Remote Data Module Wizard

In addition to inheriting from the `TRemoteDataModule` base class, this class implements the custom `IAppServerOne` interface, which derives from the standard `DataSnap` interface (`IAppServer`). The class also overrides the `UpdateRegistry` method to add support for enabling the socket and web transports, as you can see in the code generated by the wizard. At the end of the unit, you'll find the class factory declaration, which should be clear if you read [Chapter 12](#):

#### initialization

```
TComponentFactory.Create(ComServer, TAppServerOne,
  Class_AppServerOne, ciMultiInstance, tmApartment);
end.
```

Now you can add a dataset component to the data module (I've used the dbExpress `SQLDataSet`), connect it to a database and a table or query, activate it, and finally add a `DataSetProvider` and hook it to the dataset component.

You'll obtain a DFM file like this:

```
object AppServerOne: TAppServerOne
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
    Constraints = True
  end
end
```

The main form of this program is almost useless, so you can simply add a label to it indicating that it's the form of the server application. When you've built the server, you should compile it and run it once. This operation will automatically register it as an Automation server on your system, making it available to client applications. Of course, you should register the server on the computer where you want it to run, either the client or the middle tier.

## The First Thin Client

Now that you have a working server, you can build a client that will connect to it. You'll again begin with a standard Delphi application and add a DCOMConnection component to it (or the proper component for the specific type of connection you want to test). This component defines a `ComputerName` property that you'll use to specify the computer that hosts the application server. If you want to test the client and application server from the same computer, you can leave this property blank.

Once you've selected an application server computer, you can simply display the `ServerName` property's combo box list to view the available DataSnap servers. This combo box shows the servers' registered names, by default the name of the executable file of the server followed by the name of the remote data module class, as in `AppServ1.AppServerOne`. Alternatively, you can enter the GUID of the server object as the `ServerGUID` property. Delphi will automatically fill this property as you set the `ServerName` property, determining the GUID by looking it up in the Registry.

At this point, if you set the DCOMConnection component's `Connected` property to `True`, the server form will appear, indicating that the client has activated the server. You don't usually need to perform this operation, because the `ClientDataSet` component typically activates the `RemoteServer` component for you. I've suggested this step simply to emphasize what's happening behind the scenes.

**Tip** You should generally leave the DCOMConnection component's `Connected` property set to `False` at design time, so you can open the project in Delphi even on a computer where the DataSnap server is not already registered.

As you might expect, the next step is to add a `ClientDataSet` component to the form. You must connect the `ClientDataSet` to the `DCOMConnection1` component via the `RemoteServer` property, and thereby to one of the providers it exports. You can see the list of available providers in the `ProviderName` property, via the usual combo box. In this example, you'll be able to select only `DataSetProvider1`, because it is the only provider available in the server you've just built. This operation connects the dataset in the client's memory with the dbExpress dataset on the server. If you activate the client dataset and add a few data-aware controls (or a `DBGrid`), you'll immediately see the server data appear in them, as illustrated in [Figure 16.2](#).



DEPT_NO	EMP_NO	FIRST_NAME	FULL_NAME	HIRE
600	2	Robert	Nelson, Robert	12/28
621	4	Bruce	Young, Bruce	12/28
130	5	Kim	Lambert, Kim	2/6/1
180	8	Leslie	Johnson, Leslie	4/5/1
622	9	Phil	Forest, Phil	4/17/
130	11	K. J.	Weston, K. J.	1/17/
000	12	Teri	Lee, Teri	5/1/1
900	14	Stewart	Hall, Stewart	6/4/1
623	15	Katherine	Young, Katherine	6/14/
671	20	Chris	Papadopoulos, Chris	1/1/1
671	24	Pete	Fisher, Pete	9/12/
120	28	Ann	Bennet, Ann	2/1/1
623	29	Roger	De Souza, Roger	2/18/
110	34	Janet	Baldwin, Janet	3/21/

**Figure 16.2:** When you activate a ClientDataSet component connected to a remote data module at design time, the data from the server becomes visible as usual.

Here is the DFM file for the minimal client application, ThinCli1:

```

object Form1: TForm1
  Caption = 'ThinClient1'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object DCOMConnection1: TDCOMConnection
    ServerGUID = '{09E11D63-4A55-11D3-B9F1-00000100A27B}'
    ServerName = 'AppServ1.AppServerOne'
  end
  object ClientDataSet1: TClientDataSet
    Aggregates = <>
    Params = <>
    ProviderName = 'DataSetProvider1'
    RemoteServer = DCOMConnection1
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
end

```

Obviously, the programs for this first three-tier application are quite simple, but they demonstrate how to create a dataset viewer that splits the work between two different executable files. At this point, the client is only a viewer. If you edit the data on the client, it won't be updated on the server. To accomplish this, you'll need to add more code to the client. However, before you do that, let's add some features to the server.

## Adding Constraints to the Server

When you write a traditional data module in Delphi, you can easily add some of the application logic, or business rules, by handling the dataset events and by setting field object properties and handling their events. You should avoid doing this work on the client application; instead, write your business rules on the middle tier.

In the DataSnap architecture, you can send some constraints from the server to the client and let the client program impose those constraints during the user input. You can also send field properties (such as minimum and maximum values and the display and edit masks) to the client and (using some of the data access technologies) process updates through the dataset used to access the data (or a companion UpdateSql object).

### Field and Dataset Constraints

When the provider interface creates data packets to send to the client, it includes the field definitions, the table and field constraints, and one or more records (as requested by the ClientDataSet component). This implies that you can customize the middle tier and build distributed application logic by using SQL-based constraints.

The constraints you create using SQL expressions can be assigned to an entire dataset or to specific fields. The provider sends the constraints to the client along with the data, and the client applies them before sending updates back to the server. This process reduces network traffic, compared to having the client send updates back to the application server and eventually up to the SQL server, only to find that the data is invalid. Another advantage of coding the constraints on the server side is that if the business rules change, you need to update the single server application and not the many clients on multiple computers.

But how do you write constraints? You can use several properties:

- BDE datasets have a `Constraints` property, which is a collection of `TCheckConstraint` objects. Every object has a few properties, including the expression and the error message.
- Each field object defines the `CustomConstraint` and `ConstraintErrorMessage` properties. There is also an `ImportedConstraint` property for constraints imported from the SQL server.
- Each field object has a `DefaultExpression` property, which can be used locally or passed to the `ClientDataSet`. This is not an actual constraint; it's only a suggestion to the end user.

The next example, `AppServ2`, adds a few constraints to a remote data module connected to the sample `EMPLOYEE` InterBase database. After connecting the table to the database and creating the field objects for it, you can set the following special properties:

```
object SQLDataSet1: TSQLDataSet
...
object SQLDataSet1EMP_NO: TSmallintField
  CustomConstraint = 'x > 0 and x < 10000'
  ConstraintErrorMessage =
    'Employee number must be a positive integer below 10000'
  FieldName = 'EMP_NO'
end
object SQLDataSet1FIRST_NAME: TStringField
  CustomConstraint = 'x <> '#39#39
  ConstraintErrorMessage = 'The first name is required'
  FieldName = 'FIRST_NAME'
  Size = 15
end
object SQLDataSet1LAST_NAME: TStringField
  CustomConstraint = 'not x is null'
  ConstraintErrorMessage = 'The last name is required'
  FieldName = 'LAST_NAME'
```

```
end
end
```

**Note** The expression `'x <> '#39#39` is the DFM transposition of the string `x <> ' '`, indicating that you don't want to have an empty string. The final constraint, *not x is null*, instead allows empty strings but not null values.

## Including Field Properties

You can control whether the properties of the field objects on the middle tier are sent to the `ClientDataSet` (and copied into the corresponding field objects of the client side) by using the `polncFieldProps` value of the `Options` property of the `DataSetProvider`. This flag controls the download of the field properties `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, and `DisplayValues`, if they are available in the field. Here is an example of another field of the `AppServ2` example with some custom properties:

```
object SQLDataSet1SALARY: TBCDField
  DefaultExpression = '10000'
  FieldName = 'SALARY'
  DisplayFormat = '#,###'
  EditFormat = '####'
  Precision = 15
  Size = 2
end
```

With this setting, you can write your middle tier the way you usually set the fields of a standard client/server application. This approach also makes it faster to move existing applications from a client/server to a multitier architecture. The main drawback of sending fields to the client is that transmitting all the extra information takes time. Turning off `polncFieldProps` can dramatically improve network performance of datasets with many columns.

A server can generally filter the fields sent to the client; it does so by declaring persistent field objects with the `Fields` editor and omitting some of the fields. Because a field you're filtering out might be required to identify the record for future updates (if the field is part of the primary key), you can also use the field's `ProviderFlags` property on the server to send the field value to the client but make it unavailable to the `ClientDataSet` component (this provides some extra security, compared to sending the field to the client and hiding it there).

## Field and Table Events

You can write middle-tier dataset and field event handlers as usual and let the dataset process the updates received by the client in the traditional way. This means updates are considered to be operations on the dataset, exactly as when a user is directly editing, inserting, or deleting fields locally.

This update process is requested by setting the `ResolveToDataSet` property of the `TDataSetProvider` component, again connecting either the dataset used for input or a second dataset used for the updates. This approach is possible with datasets supporting editing operations. These include BDE, ADO, and InterBase Express datasets, but not those of the new `dbExpress` architecture.

With this technique, the updates are performed by the dataset, which implies a lot of control (the standard events are being triggered) but generally slower performance. Flexibility is much greater, because you can use standard coding practices. Also, porting existing local or client/server database applications, which use dataset and field events, is much more straightforward with this model. However, keep in mind that the user of the client program will receive your error messages only when the local cache (the delta) is sent back to the middle tier. Saying to the user that some data prepared half an hour ago is not valid might be a little awkward. If you follow this approach, you'll probably need to apply the updates in the cache at every `AfterPost` event on the client side.

Finally, if you decide to let the dataset and not the provider do the updates, Delphi helps you a lot in handling possible exceptions. Any exceptions raised by the middle-tier update events (for example, `OnBeforePost`) are automatically transformed by Delphi into update errors, which activate the `OnReconcileError` event on the client side (more on

this event later in this chapter). No exception is shown on the middle tier, but the error travels back to the client.

**Team LiB**

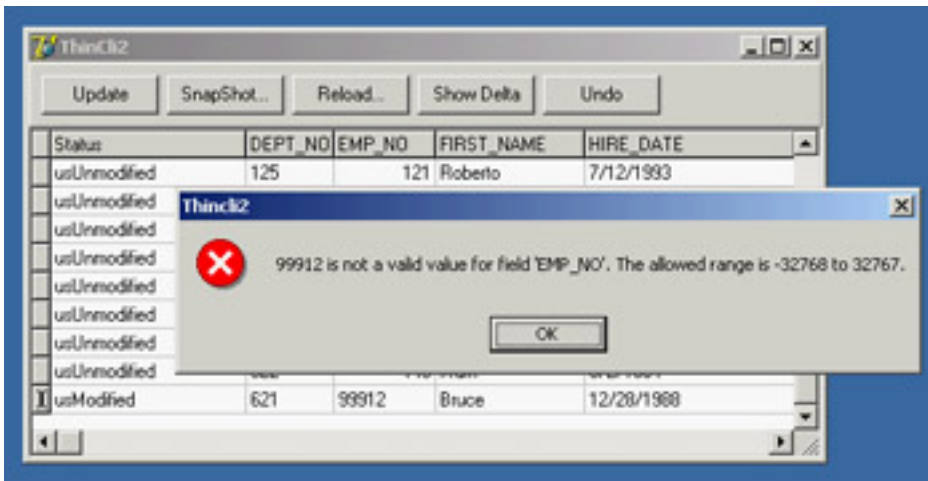
◀ PREVIOUS

NEXT ▶

## Adding Features to the Client

After adding some constraints and field properties to the server, let's return our attention to the client application. The first version was very simple, but now you can add several features to make it work well. In the ThinCli2 example, I've embedded support for checking the record status and accessing the delta information (the updates to be sent back to the server), using some of the ClientDataSet techniques already discussed in [Chapter 13](#). The program also handles reconcile errors and supports the briefcase model.

Keep in mind that while you're using this client to edit the data locally, you'll be reminded of any failure to match the business rules of the application, which are set up on the server side using constraints. The server will also provide you with a default value for the Salary field of a new record and pass along the value of its `DisplayFormat` property. In [Figure 16.3](#), you can see one of the error messages this client application can display, which it receives from the server. This message is displayed when you're editing the data locally, not when you send it back to the server.



**Figure 16.3:** The error message displayed by the ThinCli2 example when the employee ID is too large

## The Update Sequence

This client program includes a button to apply the updates to the server and a standard reconcile dialog. Here is a summary of the complete sequence of operations related to an update request and the possible error events:

1. The client program calls the `ApplyUpdates` method of a `ClientDataSet`.
2. The delta is sent to the provider on the middle tier. The provider fires the `OnUpdateData` event, where you have a chance to look at the requested changes before they reach the database server. At this point you can modify the delta, which is passed in a format compatible with the data of a `ClientDataSet`.
3. The provider (technically, a part of the provider called the *resolver*) applies each row of the delta to the database server. Before applying each update, the provider receives a `BeforeUpdateRecord` event. If you've set the `ResolveToDataSet` flag, this update will eventually fire local events of the dataset in the middle tier.
4. In case of a server error, the provider fires the `OnUpdateError` event (on the middle tier) and the program has a chance of fixing the error at that level.
5. If the middle-tier program doesn't fix the error, the corresponding update request remains in the delta. The error is returned to the client side at this point or after a given number of errors have been collected, depending on the value of the `MaxErrors` parameter of the `ApplyUpdates` call.
6. The delta packet with the remaining updates is sent back to the client, firing the `OnReconcileError` event of the `ClientDataSet` for each remaining update. In this event handler, the client program can try to fix the problem (possibly prompting the user for help), modifying the update in the delta, and later reissuing it.

## Refreshing Data

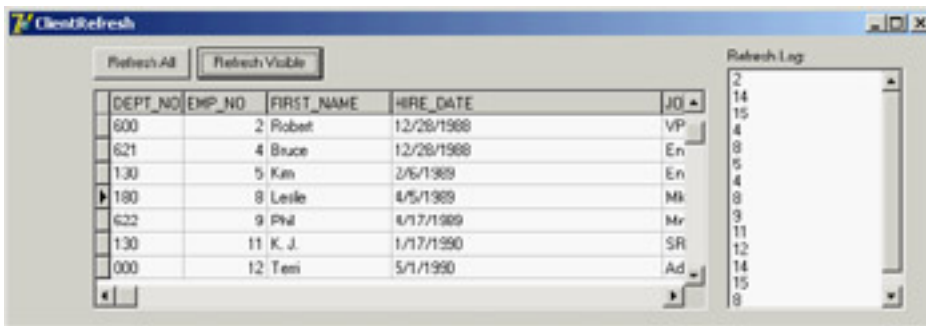
You can obtain an updated version of the data, which other users might have modified, by calling the `Refresh` method of the `ClientDataSet`. However, this operation can be done only if there are no pending update operations in the cache, because calling `Refresh` raises an exception when the change log is not empty:

```
if cds.ChangeCount = 0 then
    cds.Refresh;
```

If only some records have been changed, you can refresh the others by calling `RefreshRecords`. This method refreshes only the current record, but it should be used only if the user hasn't modified the current record. In this case, `RefreshRecords` leaves the unapplied changes in the change log. As an example, you can refresh a record every time it becomes the active one, unless it has been modified and the changes have not yet been posted to the server:

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);
begin
    if cds.UpdateStatus = usUnModified then
        cds.RefreshRecord;
end;
```

When the data is subject to frequent changes by many users and each user should see changes right away, you should apply any change immediately in the `AfterPost` and `AfterDelete` methods, and call `RefreshRecords` for the active record (as shown previously) or each of the records visible inside a grid. This code is part of the `ClientRefresh` example, connected to the `AppServ2` server. For debugging purposes, the program also logs the `EMP_NO` field for each record it refreshes, as you can see in [Figure 16.4](#).



**Figure 16.4:** The form of the `ClientRefresh` example, which automatically refreshes the active record and allows more extensive updates by clicking the buttons

I've done this by adding a button to the `ClientRefresh` example. The handler of this button moves from the current record to the first visible record of the grid and then to the last visible record. This is accomplished by noting that there are `RowCount - 1` rows visible, assuming that the first row is the fixed one hosting the field names. The program doesn't call `RefreshRecord` every time, because each movement will trigger an `AfterScroll` event with the code shown previously. The following code refreshes the visible rows, which might be triggered by a timer:

```
// protected access hack
type
    TMyGrid = class (TDBGrid)
    end;

procedure TForm1.Button2Click(Sender: TObject);
var
    i: Integer;
    bm: TBookmarkStr;
begin
    // refresh visible rows
```

```

cds.DisableControls;
// start with the current row
i := TMyGrid (DbGrid1).Row;
bm := cds.Bookmark;
try
  // get back to the first visible record
  while i > 1 do
    begin
      cds.Prior;
      Dec (i);
    end;
  // return to the current record
  i := TMyGrid(DbGrid1).Row;
  cds.Bookmark := bm;
  // go ahead until the grid is complete
  while i < TMyGrid(DbGrid1).RowCount do
    begin
      cds.Next;
      Inc (i);
    end;
  finally
    // set back everything and refresh
    cds.Bookmark := bm;
    cds.EnableControls;
  end;
end;

```

This approach generates a huge amount of network traffic, so you might want to trigger updates only when there are actual changes. You can implement this process by adding a callback technology to the server, so that it can inform all connected clients that a given record has changed. The client can determine whether it is interested in the change and eventually trigger the update request.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Advanced DataSnap Features

DataSnap includes many more features than I've covered up to now. Here is a quick tour of some of the more advanced features of the architecture, partially demonstrated by the AppSPlus and ThinPlus examples. Unfortunately, demonstrating every piece of functionality would turn this chapter into an entire book, so I'll limit myself to an overview.

**Warning** The ThinPlus example requires Delphi's socket server (provided in Delphi's *bin* folder) to run. Without this program, you'll see a socket error when the client tries to connect with the server. The plus side is that you can easily deploy the program over a network by modifying the IP address of the server in the client program.

Besides the features discussed in the following sections, the AppSPlus and ThinPlus examples demonstrate the use of a socket connection, limited logging of events and updates on the server side, and direct fetching of a record on the client side. The last feature is accomplished with this call:

```
procedure TClientForm.ButtonFetchClick(Sender: TObject);  
begin  
    ButtonFetch.Caption := IntToStr (cds.GetNextPacket);  
end;
```

This allows you to get more records than are required by the client user interface (the DBGrid). In other words, you can fetch records directly, without waiting for the user to scroll down in the grid. I suggest you study the details of these complex examples after reading the rest of this section.

## Parametric Queries

If you want to use parameters in a query or stored procedure, then instead of building a custom solution (with a custom method call to the server), you can let Delphi help you. First define the query on the middle tier with a parameter:

```
select * from customer where Country = :Country
```

Use the `Params` property to set the type and default value of the parameter. On the client side, you can use the `Fetch Params` command from the `ClientDataSet`'s shortcut menu, after connecting the component to the proper provider. At run time, you can call the equivalent `FetchParams` method of the `ClientDataSet` component.

Now you can provide a local default value for the parameter by acting on the `Params` property. The value of the parameter will be sent to the middle tier when you fetch the data. The ThinPlus example refreshes the parameter with the following code:

```
procedure TFormQuery.btnParamClick(Sender: TObject);  
begin  
    cdsQuery.Close;  
    cdsQuery.Params[0].AsString := EditParam.Text;  
    cdsQuery.Open;  
end;
```

You can see the secondary form of this example, which shows the result of the parametric query in a grid, in [Figure 16.5](#). The figure also shows some custom data sent by the server, as explained in the section "[Customizing the Data Packets](#)."



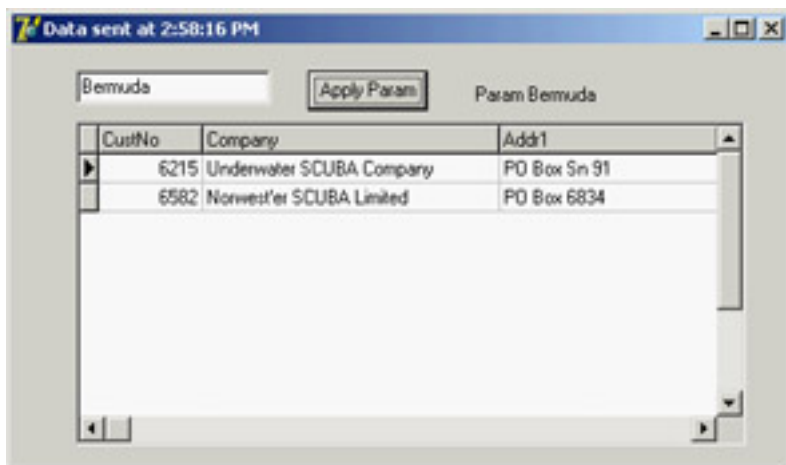


Figure 16.5: The secondary form of the ThinPlus example, showing the data of a parametric query

## Custom Method Calls

Because the server has a normal COM interface, you can add more methods or properties to it and call them from the client. Simply open the server's type library editor and use it as with any other COM server. In the AppSPlus example, I've added a custom `Login` method with the following implementation:

```
procedure TAppServerPlus.Login(const Name, Password: WideString);
begin
  // TODO: add actual login code...
  if Password <> Name then
    raise Exception.Create ('Wrong name/password combination received')
  else
    Query.Active := True;
    ServerForm.Add ('Login:' + Name + '/' + Password);
end;
```

The program performs a simple test, instead of checking the name/password combination against a list of authorizations as a real application should do. Also, disabling the `Query` doesn't really work, because it can be activated by the provider; disabling the `DataSetProvider` is a more robust approach. The client has a simple way to access the server: the `AppServer` property of the remote connection component. Here is a sample call from the ThinPlus example, which takes place in the `AfterConnect` event of the connection component:

```
procedure TClientForm.ConnectionAfterConnect(Sender: TObject);
begin
  Connection.AppServer.Login (Edit2.Text, Edit3.Text);
end;
```

Note that you can call extra methods of the COM interface through DCOM and also using a socket-based or HTTP connection. Because the program uses the `safecall` calling convention, the exception raised on the server is automatically forwarded and displayed on the client side. This way, when a user selects the `Connect` check box, the event handler used to enable the client datasets is interrupted, and a user with the wrong password won't be able to see the data.

**Note** Besides direct method calls from the client to the server, you can also implement callbacks from the server to the client. You can use this approach, for example, to notify every client of specific events. COM events are one way to do this. As an alternative, you can add a new interface, implemented by the client, which passes the implementation object to the server. This way, the server can call the method on the client computer. Callbacks are not possible with HTTP connections, though.

## Master/Detail Relations

If your middle-tier application exports multiple datasets, you can retrieve them using multiple `ClientDataSet`

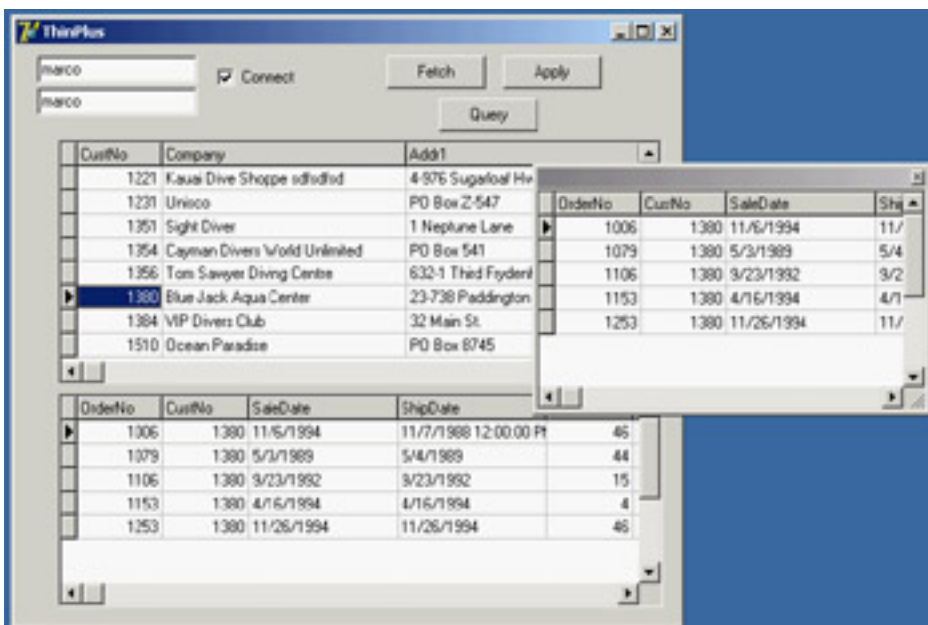
components on the client side and connect them locally to form a master/detail structure. This approach will create quite a few problems for the detail dataset unless you retrieve all the records locally.

This solution also makes it quite complex to apply the updates; you cannot usually cancel a master record until all related detail records have been removed, and you cannot add detail records until the new master record is properly in place. (Different servers handle this situation differently, but in most cases where a foreign key is used, this is the standard behavior.) To solve this problem, you can write complex code on the client side to update the records of the two tables according to the specific rules.

A completely different approach is to retrieve a single dataset that already includes the detail as a dataset field, a field of type `TDataSetField`. To accomplish this, you need to set up the master/detail relation on the server application:

```
object TableCustomer: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object TableOrders: TTable
  DatabaseName = 'DBDEMOS'
  MasterFields = 'CustNo'
  MasterSource = DataSourceCust
  TableName = 'ORDERS.DB'
end
object DataSourceCust: TDataSource
  DataSet = TableCustomer
end
object ProviderCustomer: TDataSetProvider
  DataSet = TableCustomer
end
```

On the client side, the detail table will show up as an extra field of the `ClientDataSet`, and the `DBGrid` control will display it as an extra column with an ellipsis button. Clicking the button will display a secondary form with a grid presenting the detail table (see [Figure 16.6](#)). If you need to build a flexible user interface on the client, you can then add a secondary `ClientDataSet` connected to the dataset field of the master dataset, using the `DataSetField` property. Simply create persistent fields for the main `ClientDataSet` and then hook up the property:



**Figure 16.6:** The ThinPlus example shows how a dataset field can either be displayed in a grid in a floating window or extracted by a `ClientDataSet` and displayed in a second form. You'll generally do one of the two things, not both!

```

object cdsDet: TClientDataSet
    DataSetField = cdsTableOrders
end

```

With this setting, you can show the detail dataset in a separate DBGrid placed as usual in the form (the bottom grid of [Figure 16.6](#)) or any other way you like. Note that with this structure, the updates relate only to the master table, and the server should handle the proper update sequence even in complex situations.

## Using the Connection Broker

I've already mentioned that the ConnectionBroker component can be helpful in case you might want to change the physical connection used by many ClientDataSet components of a single program. By hooking each ClientDataSet to the ConnectionBroker, you can change the physical connection of all the ClientDataSets by updating the physical connection of the broker.

The ThinPlus example uses these settings:

```

object Connection: TSocketConnection
    ServerName = 'AppSPlus.AppServerPlus'
    AfterConnect = ConnectionAfterConnect
    Address = '127.0.0.1'
end
object ConnectionBroker1: TConnectionBroker
    Connection = Connection
end
object cds: TClientDataSet
    ConnectionBroker = ConnectionBroker1
end
// in the secondary form
object cdsQuery: TClientDataSet
    ConnectionBroker = ClientForm.ConnectionBroker1
end

```

That's basically all you have to do. To change the physical connection, drop a new DataSnap connection component to the main form and set the Connection property of the broker to it.

## More Provider Options

I've already mentioned the Options property of the DataSetProvider component, noting that you can use it to add the field properties to the data packet. There are several other options you can use to customize the data packet and the behavior of the client program. Here is a short list:

- You can minimize downloading BLOB data with the poFetchBlobsOnDemand option. In this case, the client application can download BLOBs by setting the FetchOnDemand property of the ClientDataSet to True or by calling the FetchBlobs method for specific records. Similarly, you can disable the automatic downloading of detail records by setting the poFetchDetailsOnDemand option. Again, the client can use the FetchOnDemand property or call the FetchDetails method.
- When you are using a master/detail relation, you can control cascades with either of two options. The poCascadeDeletes flag controls whether the provider should delete detail records before deleting a master record. You can set this option if the database server performs cascaded deletes for you as part of its referential integrity support. Similarly, you can set the poCascadeUpdates option when the server can automatically update key values of a master/detail relation.
- You can limit the operations on the client side. The most restrictive option, poReadOnly, disables any update. If you want to give the user a limited editing capability, use poDisableInserts, poDisableEdits, or poDisableDeletes.

- You can use `poAutoRefresh` to resend to the client a copy of the records the client has modified; doing so is useful in case other users have simultaneously made other, nonconflicting changes. You can also specify the `poPropagateChanges` option to send back to the client changes done in the `BeforeUpdateRecord` or `AfterUpdateRecord` event handler. This option is also handy when you are using autoincrement fields, triggers, and other techniques that modify data on the server or middle tier beyond the changes requested from the client tier.
- Finally, if you want the client to drive the operations, you can enable the `poAllowCommandText` option. It lets you set the SQL query or table name of the middle tier from the client, using the `GetRecords` or `Execute` method.

## The Simple Object Broker

The `SimpleObjectBroker` component provides an easy way to locate a server application among several server computers. You provide a list of available computers, and the client will try each of them in order until it finds one that is available.

Moreover, if you enable the `LoadBalanced` property, the component will randomly choose one of the servers; when many clients use the same configuration, the connections will be automatically distributed among the multiple servers. If this seems like a "poor man's" object broker, consider that some highly expensive load-balancing systems don't offer much more than this.

## Object Pooling

When multiple clients connect to your server at the same time, you have two options. The first is to create a remote data module object for each of them and let each request be processed in sequence (the default behavior for a COM server with the `ciMultiInstance` style). Alternatively, you can let the system create a different instance of the server application for every client (`ciSingleInstance`). This approach requires more resources and more SQL server connections (and possibly licenses).

An alternate approach is offered by `DataSnap`'s support for object pooling. All you need to do to request this feature is add a call to `RegisterPooled` in the overridden `UpdateRegistry` method. Combined with the stateless support built in to this architecture, the pooling capability allows you to share some middle-tier objects among a much larger number of clients. A pooling mechanism is built in to COM+, but `DataSnap` makes it available for HTTP and socket-based connections as well.

The users on the client computers will spend most of their time reading data and typing in updates, and they generally don't continue asking for data and sending updates. When the client is not calling a method of the middle-tier object, this same remote data module can be used for another client. Being stateless, every request reaches the middle tier as a brand-new operation, even when a server is dedicated to a specific client.

## Customizing the Data Packets

There are many ways to include custom information within the data packet handled by the `IAppServer` interface. The simplest is to handle the `OnGetDataSetProperties` event of the provider. This event has a `Sender` parameter, a dataset parameter indicating where the data is coming from, and an `OleVariant` array `Properties` parameter, in which you can place the extra information. You need to define one variant array for each extra property and include the name of the extra property, its value, and whether you want the data to return to the server along with the update delta (the `IncludeInDelta` parameter).

Of course, you can pass properties of the related dataset component, but you can also pass any other value (extra fake properties). In the `AppSPlus` example, I pass to the client the time the query was executed and its parameters:

```
procedure TAppServerPlus.ProviderQueryGetDataSetProperties(
  Sender: TObject; DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
```

```
Properties[0] := VarArrayOf(['Time', Now, True]);
Properties[1] := VarArrayOf(['Param', Query.Params[0].AsString, False]);
end;
```

On the client side, the `ClientDataSet` component has a `GetOptionalParameter` method to retrieve the value of the extra property with the given name. The `ClientDataSet` also has the `SetOptionalParameter` method to add more properties to the dataset. These values will be saved to disk (in the briefcase model) and eventually sent back to the middle tier (by setting the `IncludeInDelta` member of the variant array to `True`). Here is a simple example of the retrieval of the dataset in the previous code:

```
Caption := 'Data sent at ' + TimeToStr (TDateTime (
    cdsQuery.GetOptionalParam('Time')));
Label1.Caption := 'Param ' + cdsQuery.GetOptionalParam('Param');
```

The effect of this code was visible in [Figure 16.5](#). An alternative and more powerful approach for customizing the data packet sent to the client is to handle the `OnGetData` event of the provider, which receives the outgoing data packet in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client. For example, you might encode some of the data or filter out sensitive records.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

Borland originally introduced its multitier technology in Delphi 3 and has kept extending it from version to version. In addition to further updates and the change of the MIDAS name to DataSnap, Delphi 6 saw the introduction of SOAP support, introducing an alternate and extended architecture for multitier applications. We'll fully explore this topic in [Chapter 23](#). Delphi 7, on the other hand, cut back on CORBA support.

However, with the introduction of a new licensing scheme (basically, *free deployment*), Delphi 7 has paved the way for increased adoption of this technology. This is particularly true of DataSnap's SOAP variant, but socket connections also provide a very good balance of data transfer efficiency and ease of deployment.

For the moment, we'll continue with database programming, discussing data-aware controls and custom datasets. In the final part of the book we'll explore sockets, Internet programming, and SOAP, so you'll have a complete picture of possible multitier architectures based on Delphi—not to mention the availability of third-party tools providing features similar to DataSnap.

## Chapter 17: Writing Database Components

### Overview

In [Chapter 9](#), "Writing Delphi Components," we explored the development of Delphi components in depth. Now that I've discussed database programming, we can get back to the earlier topic and focus on the development of database-related components.

There are basically two families of such components: data-aware controls you can use to present the data of a field or an entire record to the users of a program; and dataset components you can define to provide data to existing data-aware controls, reading the data from a database or any other data source. In this chapter, I'll cover both topics.

## The Data Link

When you write a Delphi database program, you generally connect some data-aware controls to a `DataSource` component, and then connect the `DataSource` component to a dataset. The connection between the data-aware control and the `DataSource` is called a *data link* and is represented by an object of class `TDataLink` or descendant. The data-aware control creates and manages this object and represents its only connection to the data. From a more practical perspective, to make a component data-aware, you need to add a data link to it and surface some of the properties of this internal object, such as the `DataSource` and `DataField` properties.

Delphi uses the `DataSource` and `DataLink` objects for bidirectional communication. The dataset uses the connection to notify the data-aware controls that new data is available (because the dataset has been activated, or the current record has changed, and so on). Data-aware controls use the connection to ask for the current value of a field or to update it, notifying the dataset of this event.

The relations among all these components are complicated by the fact that some of the connections can be one-to-many. For example, you can connect multiple data sources to the same dataset, you generally have multiple data links to the same data source (simply because you need one link for every data-aware component), and in most cases you connect multiple data-aware controls to each data source.

### The `TDataLink` Class

We'll work for much of this chapter with `TDataLink` and its derived classes, which are defined in the `DB` unit. This class has a set of protected virtual methods, which have a role similar to events. They are "almost-do-nothing" methods you can override in a specific subclass to intercept user operations and other data-source events. Here is a list, extracted from the class's source code:

**type**

```
TDataLink = class(TPersistent)
protected
  procedure ActiveChanged; virtual;
  procedure CheckBrowseMode; virtual;
  procedure DataSetChanged; virtual;
  procedure DataSetScrolled(Distance: Integer); virtual;
  procedure FocusControl(Field: TFieldRef); virtual;
  procedure EditingChanged; virtual;
  procedure LayoutChanged; virtual;
  procedure RecordChanged(Field: TField); virtual;
  procedure UpdateData; virtual;
```

All these virtual methods are called by the `DataEvent` private method, which is a sort of window procedure for a data source, triggered by several data events (see the `TDataEvent` enumeration). These events originate in the dataset, fields, or data source, and are generally applied to a dataset. The `DataEvent` method of the dataset component dispatches the events to the connected data sources. Each data source calls the `NotifyDataLinks` method to forward the event to each connected data link, and then the data source triggers its own `OnDataChange` or `OnUpdateData` event.

### Derived Data Link Classes

The `TDataLink` class is not technically an abstract class, but you'll seldom use it directly. When you need to create data-aware controls, you'll need to use one of its derived classes or derive a new one yourself. The most important class derived from `TDataLink` is the `TFieldDataLink` class, which is used by data-aware controls that relate to a single field of the dataset. Most data-aware controls fall into this category, and the `TFieldDataLink` class solves the most common problems of this type of component.



All the table- or record-oriented data-aware controls define specific subclasses of `TDataLink`, as we'll do later. The `TFieldDataLink` class has a list of events corresponding to the virtual methods of the base class it overrides. This makes the class simpler to customize, because you can use event handlers instead of having to inherit a new class from it. Here's an example of an overridden method, which fires the corresponding event, if available:

```
procedure TFieldDataLink.ActiveChanged;  
begin  
    UpdateField;  
    if Assigned(FOnActiveChange) then FOnActiveChange(Self);  
end;
```

The `TFieldDataLink` class also contains the `Field` and `FieldName` properties that let you connect the data-aware control to a specific field of the dataset. The link keeps a reference to the current visual component, using the `Control` property.

[Team LiB](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

## Writing Field-Oriented Data-Aware Controls

Now that you understand the theory of how the data link classes work, let's begin building some data-aware controls. The first two examples are data-aware versions of the `ProgressBar` and `TrackBar` common controls. You can use the first to display a numeric value, such as a percentage, in a visual way. You can use the second to allow a user to change the numeric value.

**Note** The code for all of the components built in this chapter is in the `MdDataPack` folder, which also includes a similarly named package for installing them all. Other folders include sample programs that use these components.

### A Read-Only ProgressBar

A data-aware version of the `ProgressBar` control is a relatively simple case of a data-aware control, because it is read-only. This component is derived from the version that's not data-aware and adds a few properties of the data link object it encapsulates:

```
type
  TMdDbProgress = class(TProgressBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
  protected
    // data link event handler
    procedure DataChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

As with every data-aware component that connects to a single field, this control makes available the `DataSource` and `DataField` properties. There is little code to write; simply export the properties from the internal data link object, as follows:

```
function TMdDbProgress.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TMdDbProgress.SetDataField (Value: string);
begin
  FDataLink.FieldName := Value;
end;

function TMdDbProgress.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;
```

```

procedure TMdDbProgress.SetDataSource (Value: TDataSource);
begin
    FDataLink.DataSource := Value;
end;

function TMdDbProgress.GetField: TField;
begin
    Result := FDataLink.Field;
end;

```

Of course, to make this component work, you must create and destroy the data link when the component itself is created or destroyed:

```

constructor TMdDbProgress.Create (AOwner: TComponent);
begin
    inherited Create (AOwner);
    FDataLink := TFieldDataLink.Create;
    FDataLink.Control := Self;
    FDataLink.OnDataChange := DataChange;
end;

destructor TMdDbProgress.Destroy;
begin
    FDataLink.Free;
    FDataLink := nil;
    inherited Destroy;
end;

```

In the preceding constructor, notice that the component installs one of its own methods as an event handler for the data link. This is where the component's most important code resides. Every time the data changes, you modify the output of the progress bar to reflect the value of the current field:

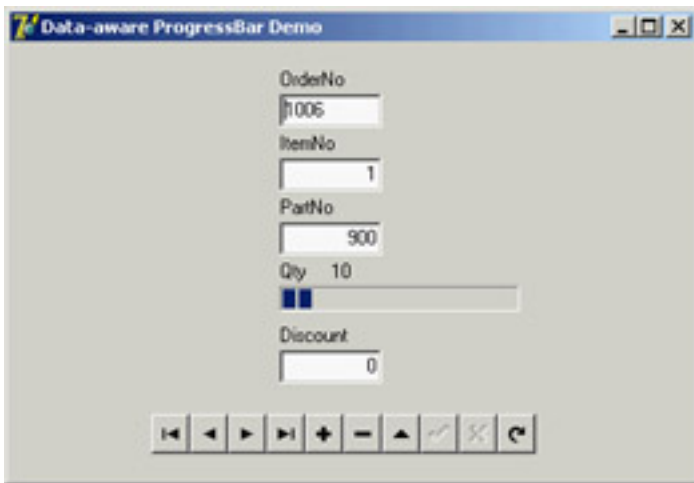
```

procedure TMdDbProgress.DataChange (Sender: TObject);
begin
    if FDataLink.Field is TNumericField then
        Position := FDataLink.Field.AsInteger
    else
        Position := Min;
end;

```

Following the convention of the VCL data-aware controls, if the field type is invalid, the component doesn't display an error message—it disables the output. Alternatively, you might want to check the field type when the `SetDataField` method assigns it to the control.

In [Figure 17.1](#) you can see an example of the DbProgr application's output, which uses both a label and a progress bar to display an order's quantity information. Thanks to this visual clue, you can step through the records and easily spot orders for many items. One obvious benefit to this component is that the application contains almost no code, because all the important code is in the MdProgr unit that defines the component.



**Figure 17.1:** The data-aware ProgressBar in action in the DbProgr example

As you've seen, a read-only data-aware component is not difficult to write. However, it becomes extremely complex to use such a component inside a `DBCtrlGrid` container.

**Note** If you remember the discussion of the *Notification* method in [Chapter 9](#), you might wonder what happens if the data source referenced by the data-aware control is destroyed. The good news is that the data source has a destructor that removes itself from its own data links. So, there is no need for a *Notification* method for data-aware controls, although you'll see books and articles suggesting it, and VCL includes plenty of this useless code.

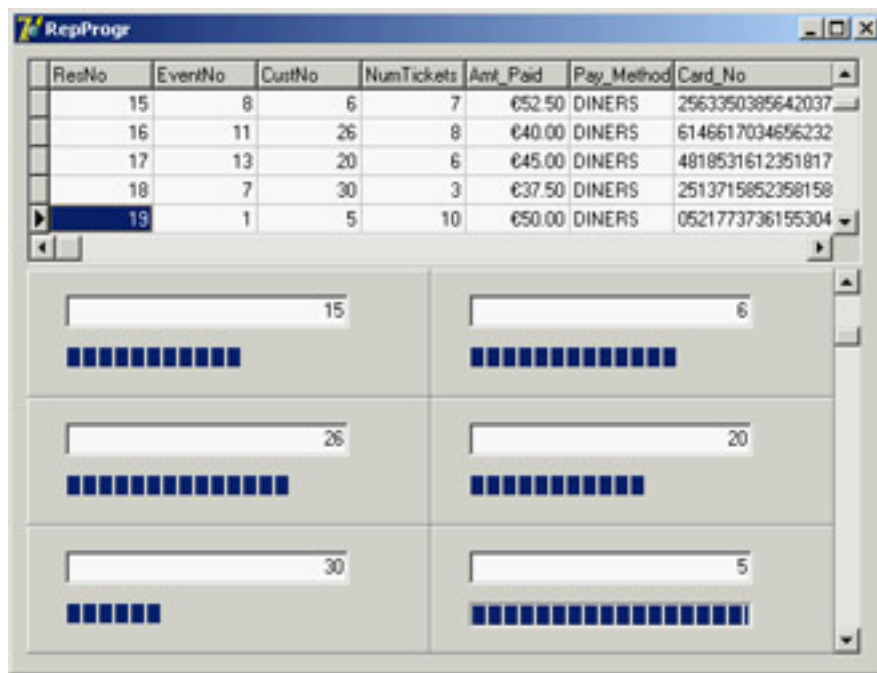
---

### Replicable Data-Aware Controls

Extending a data-aware control to support its use inside a `DBCtrlGrid` component is complex and not well documented. You can find a complete replicable version of the progress bar in the `MdRepPr` unit of the `MdDataPack` package and an example of its use in the `RepProgr` folder, along with an HTML file describing its development. The `DBCtrlGrid` component has a peculiar behavior: It displays on screen multiple versions of the same physical control, using smoke and mirrors. The grid can attach the control to a data buffer other than the current record and redirects the control paint operations to another portion of the monitor.

In short, to appear in the `DBCtrlGrid`, a component's `csReplicable` control style must be set; this flag indicates that your component supports being hosted by a control grid. First, the component must respond to the `cm_GetDataLink` Delphi message and return a pointer to the data link, so that the control grid can use and change it. Second, it needs a custom `Paint` method to draw the output in the appropriate canvas object, which is provided in a parameter of the `wm_Paint` message if the `ControlState` property's `csPaintCopy` flag is set.

The example code is involved, and the `DBCtrlGrid` component is not heavily used, so I decided not to give you full details here; you can find the full code and more information in the source code. Here's the output of a test program that uses this component:



## A Read-Write TrackBar

The next step is to write a component that allows a user to modify the data in a database, not just browse it. The overall structure of this type of component isn't very different from the previous version, but there are a few extra elements. In particular, when the user begins interacting with the component, the code should put the dataset into edit mode and then notify the dataset that the data has changed. The dataset will then use a `FieldDataLink` event handler to ask for the updated value.

To demonstrate how you can create a data-aware component that modifies the data, I extended the `TrackBar` control. This isn't the simplest example, but it demonstrates several important techniques.

Here's the definition of the component's class (from the `MdTrack` unit of the `MdDataPack` package):

```

type
  TMdDbTrack = class(TTrackBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
    procedure CMExit(var Message: TCMExit); message CM_EXIT;
  protected
    // data link event handlers
    procedure DataChange (Sender: TObject);
    procedure UpdateData (Sender: TObject);
    procedure ActiveChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;

```

```

published
  property DataField: string read GetDataField write SetDataField;
  property DataSource: TDataSource read GetDataSource write SetDataSource;
end;

```

Compared to the read-only data-aware control you built earlier, this class is more complex, because it has three message handlers, including component notification handlers, and two new event handlers for the data link. The component installs these event handlers in the constructor, which also disables the component:

```

constructor TMDbTrack.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := Self;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
  FDataLink.OnActiveChange := ActiveChange;
  Enabled := False;
end;

```

The get and set methods and the `DataChange` event handler are similar to those in the `TMDbProgress` component. The only difference is that whenever the data source or data field changes, the component checks the current status to see whether it should enable itself:

```

procedure TMDbTrack.SetDataSource (Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;

```

This code tests three conditions: the data link should be active, the link should refer to an actual field, and the field shouldn't be read-only.

When the user changes the field, the component should consider that the field name might be invalid; to test for this condition, the component uses a `try/finally` block:

```

procedure TMDbTrack.SetDataField (Value: string);
begin
  try
    FDataLink.FieldName := Value;
  finally
    Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
      not FDataLink.Field.ReadOnly;
  end;
end;

```

The control executes the same test when the dataset is enabled or disabled:

```

procedure TMDbTrack.ActiveChange (Sender: TObject);
begin
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;

```

The most interesting portion of this component's code is related to its user interface. When a user begins moving the scroll thumb, the component puts the dataset into edit mode, lets the base class update the thumb position, and alerts the data link (and therefore the data source) that the data has changed. Here's the code:

```

procedure TMDbTrack.CNHScroll(var Message: TWMHScroll);
begin

```

```

// enter edit mode
FDataLink.Edit;
// update data
inherited;
// let the system know
FDataLink.Modified;
end;

procedure TMdDbTrack.CNVScroll(var Message: TWMVScroll);
begin
    // enter edit mode
    FDataLink.Edit;
    // update data
    inherited;
    // let the system know
    FDataLink.Modified;
end;

```

When the dataset needs new data—for example, to perform a `Post` operation—it requests it from the component via the `TFieldDataLink` class's `OnUpdateData` event:

```

procedure TMdDbTrack.UpdateData (Sender: TObject);
begin
    if FDataLink.Field is TNumericField then
        FDataLink.Field.AsInteger := Position;
end;

```

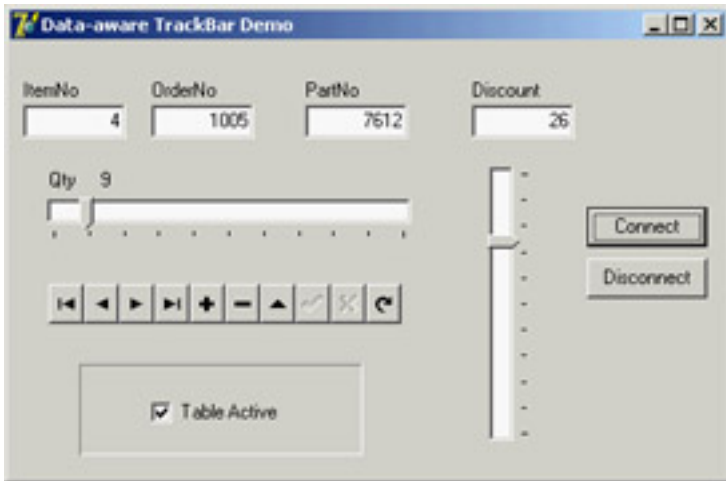
If the proper conditions are met, the component updates the data in the proper table field. Finally, if the component loses the input focus, it should force a data update (if the data has changed) so that any other data-aware components showing the value of that field will display the correct value as soon as the user moves to a different field. If the data hasn't changed, the component doesn't bother updating the data in the table. This is the standard `CMExit` code for components used by VCL and borrowed for this component:

```

procedure TMdDbTrack.CMExit(var Message: TCMExit);
begin
    try
        FDataLink.UpdateRecord;
    except
        SetFocus;
        raise;
    end;
    inherited;
end;

```

A demo program is available for testing this component; you can see its output in [Figure 17.2](#). The `DbTrack` program contains a check box to enable and disable the table, the visual components, and a couple of buttons you can use to detach the vertical `TrackBar` component from the field it relates to. I placed these on the form to test enabling and disabling the track bar.



**Figure 17.2:** The DbTrack example's track bars let you enter data in a database table. The check box and buttons test the enabled status of the components.



## Creating Custom Data Links

The data-aware controls I've built up to this point all refer to specific fields of the dataset, so I used a `TFieldDataLink` object to establish the connection with a data source. Now let's build a data-aware component that works with a dataset as a whole: a record viewer.

Delphi's database grid shows the value of several fields and several records simultaneously. My record viewer component lists all the fields of the current record, using a customized grid. This example will show you how to build a customized grid control and a custom data link to go with it.

### A Record Viewer Component

In Delphi there are no data-aware components that manipulate multiple fields of a single record without displaying other records. The only two components that display multiple fields from the same table are the `DBGrid` and the `DbCtrlGrid`, which generally display multiple fields and multiple records.

The record viewer component I'll describe in this section is based on a two-column grid; the first column displays the table's field names, and the second column displays the corresponding field values. The number of rows in the grid corresponds to the number of fields, with a vertical scroll bar in case they can't fit in the visible area.

The data link you need in order to build this component is a class connected only to the record viewer component and declared directly in the implementation portion of its unit. This is the same approach used by VCL for some specific data links. Here's the definition of the new class:

```
type
  TMdRecordLink = class (TDataLink)
  private
    RView: TMdRecordView;
  public
    constructor Create (View: TMdRecordView);
    procedure ActiveChanged; override;
    procedure RecordChanged (Field: TField); override;
  end;
```

As you can see, the class overrides the methods related to the principal event—in this case, the activation and data (or record) change. Alternatively, you could export events and then let the component handle them, as the `TFieldDataLink` does.

The constructor requires the associated component as its only parameter:

```
constructor TMdRecordLink.Create (View: TMdRecordView);
begin
  inherited Create;
  RView := View;
end;
```

After you store a reference to the associated component, the other methods can operate on it directly:

```
procedure TMdRecordLink.ActiveChanged;
var
  I: Integer;
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // repaint all...
  RView.Invalidate;
end;
```

```

procedure TMdRecordLink.RecordChanged;
begin
    inherited;
    // repaint all...
    RView.Invalidate;
end;

```

The record link code is simple. Most of the difficulties in building this example result from the use of a grid. To avoid dealing with useless properties, I derived the record viewer grid from the `TCustomGrid` class. This class incorporates much of the code for grids, but most of its properties, events, and methods are protected. For this reason, the class declaration is quite long, because it needs to publish many existing properties. Here is an excerpt (excluding the base class properties):

```

type
    TMdRecordView = class(TCustomGrid)
    private
        // data-aware support
        FDataLink: TDataLink;
        function GetDataSource: TDataSource;
        procedure SetDataSource (Value: TDataSource);
    protected
        // redefined TCustomGrid methods
        procedure DrawCell (ACol, ARow: Longint; ARect: TRect;
            AState: TGridDrawState); override;
        procedure ColWidthsChanged; override;
        procedure RowHeightsChanged; override;
    public
        constructor Create (AOwner: TComponent); override;
        destructor Destroy; override;
        procedure SetBounds (ALeft, ATop, AWidth, AHeight: Integer); override;
        // public parent properties (omitted...)
    published
        // data-aware properties
        property DataSource: TDataSource read GetDataSource write SetDataSource;
        // published parent properties (omitted...)
    end;

```

In addition to redeclaring the properties to publish them, the component defines a data link object and the `DataSource` property. There's no `DataField` property for this component, because it refers to an entire record. The component's constructor is very important. It sets the values of many unpublished properties, including the grid options:

```

constructor TMdRecordView.Create (AOwner: TComponent);
begin
    inherited Create (AOwner);
    FDataLink := TMdRecordLink.Create (self);
    // set numbers of cells and fixed cells
    RowCount := 2; // default
    ColCount := 2;
    FixedCols := 1;
    FixedRows := 0;
    Options:= [goFixedVertLine, goFixedHorzLine,
        goVertLine, goHorzLine, goRowSizing];
    DefaultDrawing := False;
    ScrollBars := ssVertical;
    FSaveCellExtents := False;
end;

```

The grid has two columns (one of them fixed) and no fixed rows. The fixed column is used for resizing each row of the grid. Unfortunately, a user cannot drag the fixed row to resize the columns, because you can't resize fixed elements,

and the grid already has a fixed column.

**Note** An alternative approach could be to have an extra empty column, like the DBGrid control. You could resize the two other columns after adding a fixed row. Overall, though, I prefer my implementation.

I used an alternative approach to resize the columns. The first column (holding the field names) can be resized either using programming code or visually at design time, and the second column (holding the values of the fields) is resized to use the remaining area of the component:

```
procedure TMDRecordView.SetBounds (ALeft, ATop, AWidth, AHeight: Integer);  
begin  
    inherited;  
    ColWidths [1] := ClientWidth - ColWidths[0];  
end;
```

This resizing takes place when the component size changes and when either of the columns change. With this code, the `DefaultColWidth` property of the component becomes, in practice, the fixed width of the first column.

After everything has been set up, the key method of the component is the overridden `DrawCell` method, detailed in [Listing 17.1](#). In this method, the control displays the information about the fields and their values. It needs to draw three things. If the data link is not connected to a data source, the grid displays an *empty element* sign ([ ]). When drawing the first column, the record viewer shows the `DisplayName` of the field, which is the same value used by the DBGrid for the heading. When drawing the second column, the component accesses the textual representation of the field value, extracted with the `DisplayText` property (or with the `AsString` property for memo fields).

### **Listing 17.1: The *DrawCell* Method of the Custom RecordView Component**

---

```
procedure TMDRecordView.DrawCell(ACol, ARow: Longint; ARect: TRect;  
    AState: TGridDrawState);  
var  
    Text: string;  
    CurrField: TField;  
    Bmp: TBitmap;  
begin  
    CurrField := nil;  
    Text := '[]'; // default  
    // paint background  
    if (ACol = 0) then  
        Canvas.Brush.Color := FixedColor  
    else  
        Canvas.Brush.Color := Color;  
    Canvas.FillRect (ARect);  
    // leave small border  
    InflateRect (ARect, -2, -2);  
    if (FDataLink.DataSource <> nil) and FDataLink.Active then  
    begin  
        CurrField := FDataLink.DataSet.Fields[ARow];  
        if ACol = 0 then  
            Text := CurrField.DisplayName  
        else if CurrField is TMemoField then  
            Text := TMemoField (CurrField).AsString  
        else  
            Text := CurrField.DisplayText;  
    end;  
    if (ACol = 1) and (CurrField is TGraphicField) then  
    begin  
        Bmp := TBitmap.Create;  
        try
```

```

    Bmp.Assign (CurrField);
    Canvas.StretchDraw (ARect, Bmp);
finally
    Bmp.Free;
end;
end
else if (ACol = 1) and (CurrField is TMemoField) then
begin
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_WordBreak or dt_NoPrefix)
end
else // draw single line vertically centered
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_vcenter or dt_SingleLine or dt_NoPrefix);
if gdFocused in AState then
    Canvas.DrawFocusRect (ARect);
end;

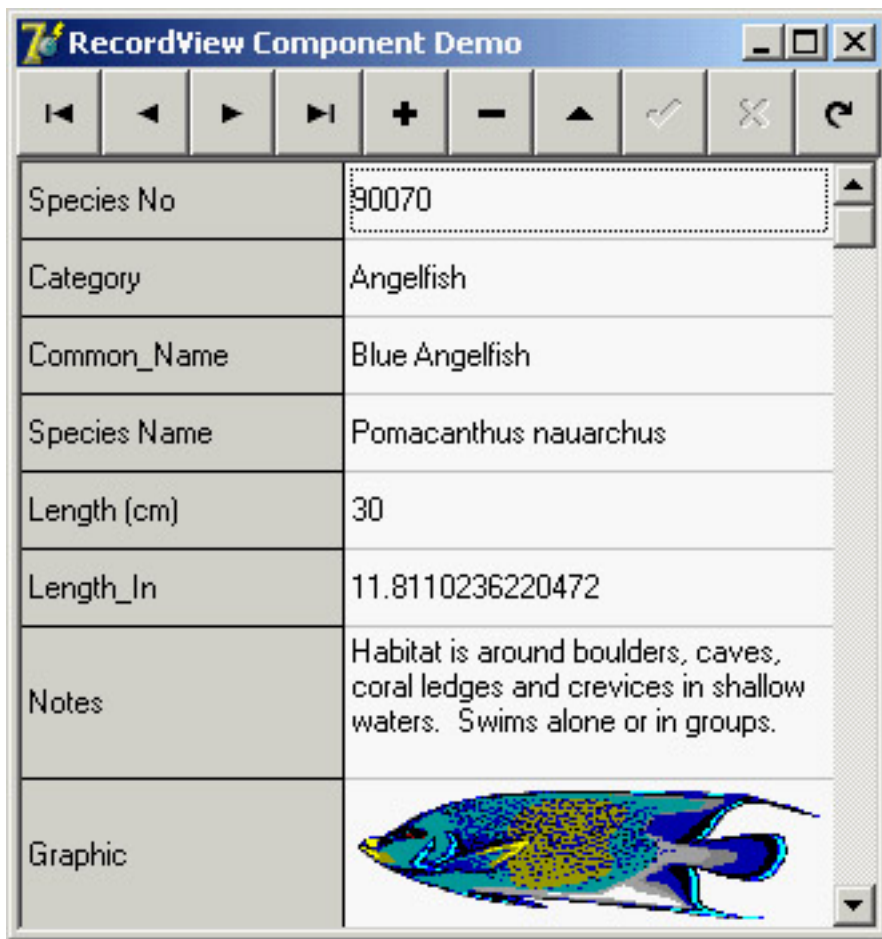
```

---

In the final portion of the method, the component considers memo and graphic fields. If the field is a `TMemoField`, the `DrawText` function call doesn't specify the `dt_SingleLine` flag, but uses `dt_WordBreak` flag to wrap the words when there's no more room. For a graphic field, the component uses a completely different approach, assigning the field image to a temporary bitmap and then stretching it to fill the surface of the cell.

Notice that the component sets the `DefaultDrawing` property to `False`, so it's also responsible for drawing the background and the focus rectangle, as it does in the `DrawCell` method. The component also calls the `InflateRect` API function to leave a small area between the cell border and the output text. The output is produced by calling another Windows API function, `DrawText`, which centers the text vertically in its cell.

This drawing code works both at run time, as you can see in [Figure 17.3](#), and at design time. The output may not be perfect, but this component can be useful in many cases. To display the data for a single record, instead of building a custom form with labels and data-aware controls, you can easily use this record viewer grid. It's important to remember that the record viewer is a read-only component. It's possible to extend it to add editing capabilities (they're already part of the `TCustomGrid` class); however, instead of adding this support, I decided to make the component more complete by adding support for displaying BLOB fields.



**Figure 17.3:** The ViewGrid example demon-strates the output of the RecordView component, using Borland's sample BioLife database table.

To improve the graphical output, the control makes the lines for BLOB fields twice as high as those for plain text fields. This operation is accomplished when the dataset connected to the data-aware control is activated. The data link's `ActiveChanged` method is also triggered by the `RowHeightsChanged` methods connected to the `DefaultRowHeight` property of the base class:

```

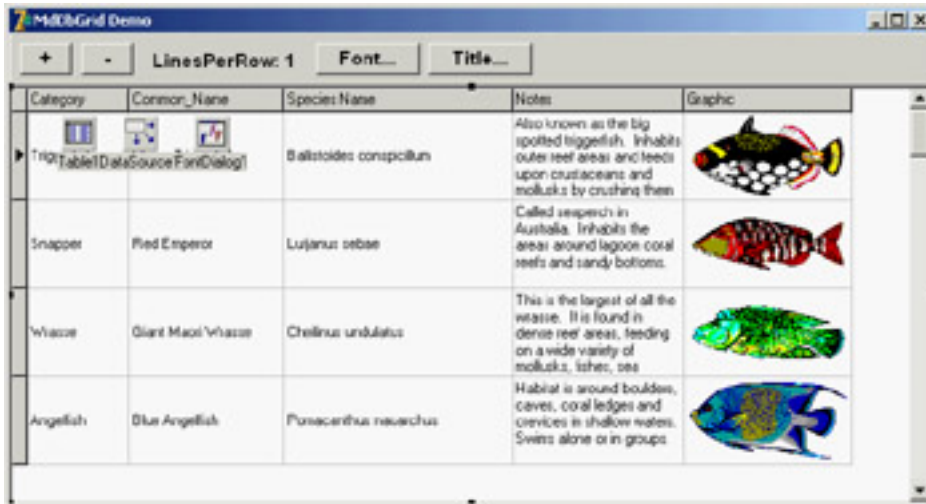
procedure TMdRecordLink.ActiveChanged;
var
    I: Integer;
begin
    // set number of rows
    RView.RowCount := DataSet.FieldCount;
    // double the height of memo and graphics
    for I := 0 to DataSet.FieldCount - 1 do
        if DataSet.Fields [I] is TBlobField then
            RView.RowHeights [I] := RView.DefaultRowHeight * 2;
    // repaint all...
    RView.Invalidate;
end;

```

At this point, you stumble into a minor problem. In the `DefineProperties` method, the `TCustomGrid` class saves the values of the `RowHeights` and `ColHeights` properties. You could disable this streaming by overriding the method and not calling `inherited` (which is generally a bad technique), but you can also toggle the `FSaveCellExtents` protected field to disable this feature (as I've done in the component's code).

## Customizing the DBGrid Component

In addition to writing new, custom, data-aware components, Delphi programmers commonly customize the DBGrid control. The goal for the next component is to enhance the DBGrid with the same kind of custom output I used for the RecordView component, directly displaying graphic and memo fields. To do this, the grid needs to make the row height resizable, to allow space for graphics and a reasonable amount of text. You can see an example of this grid at design time in [Figure 17.4](#).



**Figure 17.4:** An example of the MdDbGrid component at design time. Notice the output of the graphics and memo fields.

Although creating the output was a simple matter of adapting the code used in the record viewer component, setting the height of the grid cells ended up being a difficult problem to solve. The lines of code for that operation may be few, but they cost me hours of work!

**Note** Unlike the generic grid used earlier, a DBGrid is a virtual view on the dataset—there is no relation between the number of rows shown on the screen and the number of rows of data in the dataset. When you scroll up and down through the data records of the dataset, you are not scrolling through the rows of the DBGrid; the rows are stationary, and the data moves from one row to the next to give the appearance of movement. For this reason, the program doesn't try to set the height of an individual row to suit its data; it sets the height of all the data rows to a multiline height value.

This time the control doesn't have to create a custom data link, because it is deriving from a component that already has a complex connection with the data. The new class has a new property to specify the number of lines of text for each row and overrides a few virtual methods:

```
type
  TMdDbGrid = class(TDbGrid)
  private
    FLinesPerRow: Integer;
    procedure SetLinesPerRow (Value: Integer);
  protected
    procedure DrawColumnCell(const Rect: TRect; DataCol: Integer;
      Column: TColumn; State: TGridDrawState); override;
    procedure LayoutChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
  published
    property LinesPerRow: Integer
      read FLinesPerRow write SetLinesPerRow default 1;
  end;
```

The constructor sets the default value for the `FLinesPerRow` field. Here is the set method for the property:

```
procedure TMdDbGrid.SetLinesPerRow(Value: Integer);  
begin  
  if Value <> FLinesPerRow then  
    begin  
      FLinesPerRow := Value;  
      LayoutChanged;  
    end;  
end;
```

The side effect of changing the number of lines is a call to the `LayoutChanged` virtual method. The system calls this method frequently when one of the many output parameters changes. In the method's code, the component first calls the inherited version and then sets the height of each row. As a basis for this computation, it uses the same formula as the `TCustomDBGrid` class: The text height is calculated using the sample word *Wg* in the current font (this text is used because it includes both a full-height uppercase character and a lowercase letter with a descender). Here's the code:

```
procedure TMdDbGrid.LayOutChanged;  
var  
  PixelsPerRow, PixelsTitle, I: Integer;  
begin  
  inherited LayOutChanged;  
  
  Canvas.Font := Font;  
  PixelsPerRow := Canvas.TextHeight('Wg') + 3;  
  if dgRowLines in Options then  
    Inc (PixelsPerRow, GridLineWidth);  
  
  Canvas.Font := TitleFont;  
  PixelsTitle := Canvas.TextHeight('Wg') + 4;  
  if dgRowLines in Options then  
    Inc (PixelsTitle, GridLineWidth);  
  
  // set number of rows  
  RowCount := 1 + (Height - PixelsTitle) div (PixelsPerRow * FLinesPerRow);  
  
  // set the height of each row  
  DefaultRowHeight := PixelsPerRow * FLinesPerRow;  
  RowHeights [0] := PixelsTitle;  
  for I := 1 to RowCount - 1 do  
    RowHeights [I] := PixelsPerRow * FLinesPerRow;  
  
  // send a WM_SIZE message to let the base component recompute  
  // the visible rows in the private UpdateRowCount method  
  PostMessage (Handle, WM_SIZE, 0, MakeLong(Width, Height));  
end;
```

**Warning** `Font` and `TitleFont` are the grid defaults, which can be overridden by properties of the individual `DBGrid` column objects. This component currently ignores those settings.

The difficult part of this method was getting the final statements right. You can set the `Default-RowHeight` property, but in that case the title row will probably be too high. First I tried setting the `DefaultRowHeight` and then the height of the first row, but this approach complicated the code used to compute the number of visible rows in the grid (the read-only `VisibleRowCount` property). If you specify the number of rows (to avoid having rows hidden beneath the lower edge of the grid), the base class keeps recomputing it. Here's the code used to draw the data, ported from the `RecordView` component and adapted slightly for the grid:

```
procedure TMdDbGrid.DrawColumnCell (const Rect: TRect; DataCol: Integer;
```

```

Column: TColumn; State: TGridDrawState);
var
  Bmp: TBitmap;
  OutRect: TRect;
begin
  if FLinesPerRow = 1 then
    inherited DrawColumnCell(Rect, DataCol, Column, State)
  else
    begin
      // clear area
      Canvas.FillRect (Rect);
      // copy the rectangle
      OutRect := Rect;
      // restrict output
      InflateRect (OutRect, -2, -2);
      // output field data
      if Column.Field is TGraphicField then
        begin
          Bmp := TBitmap.Create;
          try
            Bmp.Assign (Column.Field);
            Canvas.StretchDraw (OutRect, Bmp);
          finally
            Bmp.Free;
          end;
        end
      else if Column.Field is TMemoField then
        begin
          DrawText (Canvas.Handle, PChar (Column.Field.AsString),
            Length (Column.Field.AsString), OutRect, dt_WordBreak or dt_NoPrefix)
        end
      else // draw single line vertically centered
        DrawText (Canvas.Handle, PChar (Column.Field.DisplayText),
          Length (Column.Field.DisplayText), OutRect,
            dt_vcenter or dt_SingleLine or dt_NoPrefix);
    end;
  end;
end;

```

In this code you can see that if the user displays a single line, the grid uses the standard drawing technique with no output for memo and graphic fields. However, as soon as you increase the line count, you'll see better output.

To see this code in action, run the GridDemo example. This program has two buttons you can use to increase or decrease the row height of the grid, and two more buttons to change the font. This is an important test because the height in pixels of each cell is the height of the font multiplied by the number of lines.



## Building Custom Datasets

When discussing the `TDataSet` class and the alternative families of dataset components available in Delphi in [Chapter 13](#), "Delphi's Database Architecture," I mentioned the possibility of writing a custom dataset class. Now it's time to look at an example. The reasons for writing a custom dataset relate to the fact that you won't need to deploy a database engine, but you'll still be able to take full advantage of Delphi's database architecture, including things like persistent database fields and data-aware controls.

Writing a custom dataset is one of the most complex tasks for a component developer, so this is one of the most advanced areas (as far as low-level coding practices, including tons of pointers) in the book. Moreover, Borland hasn't released any official documentation about writing custom datasets. If you are early in your experience with Delphi, you might want to skip the rest of this chapter and come back later.

`TDataSet` is an abstract class that declares several virtual abstract methods—23 in Delphi 5, now only a handful, as most have been replaced by empty virtual methods (which you still have to override). Every subclass of `TDataSet` must override all those methods.

Before discussing the development of a custom dataset, we need to explore a few technical elements of the `TDataSet` class—in particular, record buffering. The class maintains a list of buffers that store the values of different records. These buffers store the data, but they also usually store further information for the dataset to use when managing the records. These buffers don't have a predefined structure, and each custom dataset must allocate the buffers, fill them, and destroy them. The custom dataset must also copy the data from the record buffers to the various fields of the dataset, and vice versa. In other words, the custom dataset is entirely responsible for handling these buffers.

In addition to managing the data buffers, the component is responsible for navigating among the records, managing the bookmarks, defining the structure of the dataset, and creating the proper data fields. The `TDataSet` class is nothing more than a framework; you must fill it with the appropriate code. Fortunately, most of the code follows a standard structure, which the `TDataSet`-derived VCL classes use. Once you've grasped the key ideas, you'll be able to build multiple custom datasets borrowing quite a lot of code.

To simplify this type of reuse, I've collected the common features required by any custom dataset in a `TMdCustomDataSet` class. However, I'm not going to discuss the base class first and the specific implementation later, because it would be difficult to understand. Instead, I'll detail the code required by a dataset, presenting methods of the generic and specific classes at the same time, according to a logical flow.

### The Definition of the Classes

The starting point, as usual, is the declaration of the two classes discussed in this section: the generic custom dataset I've written and a specific component storing data in a file stream. The declaration of these classes is available in [Listing 17.2](#). In addition to virtual methods, the classes contain a series of protected fields used to manage the buffers, track the current position and record count, and handle many other features. You'll also notice another record declaration at the beginning: a structure used to store the extra data for every data record you place in a buffer. The dataset places this information in each record buffer, following the data.

#### Listing 17.2: The Declaration of `TMdCustomDataSet` and `TMdDataSetStream`

---

```
// in the unit MdDsCustom
type
  EMdDataSetError = class (Exception);

  TMdRecInfo = record
    Bookmark: Longint;
    BookmarkFlag: TBookmarkFlag;
  end;
  PMdRecInfo = ^TMdRecInfo;
```

```

TMDCustomDataSet = class(TDataSet)
protected
  // status
  FIsTableOpen: Boolean;
  // record data
  FRecordSize,           // the size of the actual data
  FRecordBufferSize,    // data + housekeeping (TRecInfo)
  FCurrentRecord,       // current record (0 to FRecordCount - 1)
  BofCrack,             // before the first record (crack)
  EofCrack: Integer;    // after the last record (crack)
  // create, close, and so on
  procedure InternalOpen; override;
  procedure InternalClose; override;
  function IsCursorOpen: Boolean; override;
  // custom functions
  function InternalRecordCount: Integer; virtual; abstract;
  procedure InternalPreOpen; virtual;
  procedure InternalAfterOpen; virtual;
  procedure InternalLoadCurrentRecord(Buffer: PChar); virtual; abstract;
  // memory management
  function AllocRecordBuffer: PChar; override;
  procedure InternalInitRecord(Buffer: PChar); override;
  procedure FreeRecordBuffer(var Buffer: PChar); override;
  function GetRecordSize: Word; override;
  // movement and optional navigation (used by grids)
  function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):
    TGetResult; override;
  procedure InternalFirst; override;
  procedure InternalLast; override;
  function GetRecNo: Longint; override;
  function GetRecordCount: Longint; override;
  procedure SetRecNo(Value: Integer); override;
  // bookmarks
  procedure InternalGotoBookmark(Bookmark: Pointer); override;
  procedure InternalSetToRecord(Buffer: PChar); override;
  procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
  procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
  procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
  function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
  // editing (dummy versions)
  procedure InternalDelete; override;
  procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
  procedure InternalPost; override;
  procedure InternalInsert; override;
  // other
  procedure InternalHandleException; override;
published
  // redeclared dataset properties
  property Active;
  property BeforeOpen;
  property AfterOpen;
  property BeforeClose;
  property AfterClose;
  property BeforeInsert;
  property AfterInsert;
  property BeforeEdit;

```

```

property AfterEdit;
property BeforePost;
property AfterPost;
property BeforeCancel;
property AfterCancel;
property BeforeDelete;
property AfterDelete;
property BeforeScroll;
property AfterScroll;
property OnCalcFields;
property OnDeleteError;
property OnEditError;
property OnFilterRecord;
property OnNewRecord;
property OnPostError;
end;

```

```
// in the unit MdDsStream
```

```

type
  TMdDataFileHeader = record
    VersionNumber: Integer;
    RecordSize: Integer;
    RecordCount: Integer;
  end;

  TMdDataSetStream = class(TMdCustomDataSet)
  private
    procedure SetTableName(const Value: string);
  protected
    FDataFileHeader: TMdDataFileHeader;
    FDataFileHeaderSize, // optional file header size
    FRecordCount: Integer; // current number of records
    FStream: TStream; // the physical table
    FTableName: string; // table path and file name
    FFieldOffset: TList; // field offsets in the buffer
  protected
    // open and close
    procedure InternalPreOpen; override;
    procedure InternalAfterOpen; override;
    procedure InternalClose; override;
    procedure InternalInitFieldDefs; override;
    // edit support
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
    procedure InternalPost; override;
    procedure InternalInsert; override;
    // fields
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    // custom dataset virtual methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord(Buffer: PChar); override;
  public
    procedure CreateTable;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
  published
    property TableName: string read FTableName write SetTableName;
  end;

```

---

When I divided the methods into sections (as you can see by looking at the source code files), I marked each one with a roman number. You'll see those numbers in a comment describing the method, so that while browsing this long listing you'll immediately know which of the four sections you are in.

## Section I: Initialization, Opening, and Closing

The first methods I'll examine are responsible for initializing the dataset and for opening and closing the file stream used to store the data. In addition to initializing the component's internal data, these methods are responsible for initializing and connecting the proper `TFields` objects to the dataset component. To make this work, all you need to do is to initialize the `FieldsDef` property with the definitions of the fields for your dataset, and then call a few standard methods to generate and bind the `TField` objects. This is the general `InternalOpen` method:

```
procedure TMdCustomDataSet.InternalOpen;
begin
    InternalPreOpen; // custom method for subclasses

    // initialize the field definitions
    InternalInitFieldDefs;

    // if there are no persistent field objects, create the fields dynamically
    if DefaultFields then
        CreateFields;
    // connect the TField objects with the actual fields
    BindFields (True);

    InternalAfterOpen; // custom method for subclasses

    // sets cracks and record position and size
    BofCrack := -1;
    EofCrack := InternalRecordCount;
    FCurrentRecord := BofCrack;
    FRecordBufferSize := FRecordSize + sizeof (TMdRecInfo);
    BookmarkSize := sizeof (Integer);

    // everything OK: table is now open
    FIsTableOpen := True;
end;
```

You'll notice that the method sets most of the local fields of the class, and also the `BookmarkSize` field of the base `TDataSet` class. Within this method, I call two custom methods I introduced in my custom dataset hierarchy: `InternalPreOpen` and `InternalAfterOpen`. The first, `InternalPreOpen`, is used for operations required at the very beginning, such as checking whether the dataset can be opened and reading the header information from the file. The code checks an internal version number for consistency with the value saved when the table is first created, as you'll see later. By raising an exception in this method, you can eventually stop the open operation.

Here is the code for the two methods in the derived stream-based dataset:

```
const
    HeaderVersion = 10;

procedure TMdDataSetStream.InternalPreOpen;
begin
    // the size of the header
    FDataFileHeaderSize := sizeof (TMdDataFileHeader);

    // check if the file exists
```

```

if not FileExists (FTableName) then
    raise EMdDataSetError.Create ('Open: Table file not found');

// create a stream for the file
FStream := TFileStream.Create (FTableName, fmOpenReadWrite);

// initialize local data (loading the header)
FStream.ReadBuffer (FDataFileHeader, FDataFileHeaderSize);
if FDataFileHeader.VersionNumber <> HeaderVersion then
    raise EMdDataSetError.Create ('Illegal File Version');
// let's read this, double check later
FRecordCount := FDataFileHeader.RecordCount;
end;

procedure TMdDataSetStream.InternalAfterOpen;
begin
    // check the record size
    if FDataFileHeader.RecordSize <> FRecordSize then
        raise EMdDataSetError.Create ('File record size mismatch');
    // check the number of records against the file size
    if (FDataFileHeaderSize + FRecordCount * FRecordSize) <> FStream.Size then
        raise EMdDataSetError.Create ('InternalOpen: Invalid Record Size');
end;

```

The second method, `InternalAfterOpen`, is used for operations required after the field definitions have been set and is followed by code that compares the record size read from the file against the value computed in the `InternalInitFieldDefs` method. The code also checks that the number of records read from the header is compatible with the size of the file. This test can fail if the dataset wasn't closed properly: You might want to modify this code to let the dataset refresh the record size in the header anyway.

The `InternalOpen` method of the custom dataset class is specifically responsible for calling `InternalInitFieldDefs`, which determines the field definitions (at either design time or run time). For this example, I decided to base the field definitions on an external file—an INI file that provides a section for every field. Each section contains the name and data type of the field, as well as its size if it is string data. [Listing 17.3](#) shows the `Contrib.INI` file used in the component's demo application.

### **Listing 17.3: The *Contrib.INI* File for the Demo Application**

---

```

[Fields]
Number = 6

[Field1]
Type = ftString
Name = Name
Size = 30

[Field2]
Type = ftInteger
Name = Level

[Field3]
Type = ftDate
Name = BirthDate

[Field4]
Type = ftCurrency
Name = Stipend

```

```
[Field5]
Type = ftString
Name = Email
Size = 50
```

```
[Field6]
Type = ftBoolean
Name = Editor
```

---

This file, or a similar one, must use the same name as the table file and must be in the same directory. The `InternalInitFieldDefs` method (shown in [Listing 17.4](#)) will read it, using the values it finds to set up the field definitions and determine the size of each record. The method also initializes an internal `TList` object that stores the offset of every field inside the record. You use this `TList` to access fields' data within the record buffer, as you can see in the code listing.

#### **Listing 17.4: The `InternalInitFieldDefs` Method of the Stream-Based Dataset**

---

```
procedure TmDataSetStream.InternalInitFieldDefs;
var
  IniFileName, FieldName: string;
  IniFile: TIniFile;
  nFields, I, TmpFieldOffset, nSize: Integer;
  FieldType: TFieldType;
begin
  FFieldOffset := TList.Create;
  FieldDefs.Clear;
  TmpFieldOffset := 0;
  IniFilename := ChangeFileExt(FTableName, '.ini');
  Inifile := TIniFile.Create (IniFilename);
  // protect INI file
  try
    nFields := IniFile.ReadInteger (' Fields', 'Number', 0);
    if nFields = 0 then
      raise EDataSetOneError.Create (' InitFieldsDefs: 0 fields?');
    for I := 1 to nFields do
      begin
        // create the field
        FieldType := TFieldType (GetEnumValue (TypeInfo (TFieldType),
          IniFile.ReadString ('Field' + IntToStr (I), 'Type', '')));
        FieldName := IniFile.ReadString ('Field' + IntToStr (I), 'Name', '');
        if FieldName = '' then
          raise EDataSetOneError.Create (
            'InitFieldsDefs: No name for field ' + IntToStr (I));
        nSize := IniFile.ReadInteger ('Field' + IntToStr (I), 'Size', 0);
        FieldDefs.Add (FieldName, FieldType, nSize, False);
        // save offset and compute size
        FFieldOffset.Add (Pointer (TmpFieldOffset));
        case FieldType of
          ftString:           Inc (TmpFieldOffset, nSize + 1);
          ftBoolean, ftSmallInt, ftWord:  Inc (TmpFieldOffset, 2);
          ftInteger, ftDate, ftTime:     Inc (TmpFieldOffset, 4);
          ftFloat, ftCurrency, ftDateTime:  Inc (TmpFieldOffset, 8);
        else
          raise EDataSetOneError.Create (
```

```

        'InitFieldsDefs: Unsupported field type');
    end;
end; // for
finally
    IniFile.Free;
end;
FRecordSize := TmpFieldOffset;
end;

```

---

Closing the table is a matter of disconnecting the fields (using some standard calls). Each class must dispose of the data it allocated and update the file header, the first time records are added and each time the record count has changed:

```

procedure TmdCustomDataSet.InternalClose;
begin
    // disconnect field objects
    BindFields (False);
    // destroy field object (if not persistent)
    if DefaultFields then
        DestroyFields;
    // close the file
    FIsTableOpen := False;
end;

procedure TmdDataSetStream.InternalClose;
begin
    // if required, save updated header
    if (FDataFileHeader.RecordCount <> FRecordCount) or
        (FDataFileHeader.RecordSize = 0) then
        begin
            FDataFileHeader.RecordSize := FRecordSize;
            FDataFileHeader.RecordCount := FRecordCount;
            if Assigned (FStream) then
                begin
                    FStream.Seek (0, soFromBeginning);
                    FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
                end;
            end;
        end;
    // free the internal list field offsets and the stream
    FFieldOffset.Free;
    FStream.Free;
    inherited InternalClose;
end;

```

Another related function is used to test whether the dataset is open, something you can solve using the corresponding local field:

```

function TmdCustomDataSet.IsCursorOpen: Boolean;
begin
    Result := FIsTableOpen;
end;

```

These are the opening and closing methods you need to implement in any custom dataset. However, most of the time, you'll also add a method to create the table. In this example, the `CreateTable` method creates an empty file and inserts information in the header: a fixed version number, a dummy record size (you don't know the size until you initialize the fields), and the record count (which is zero to start):

```

procedure TmdDataSetStream.CreateTable;

```

```

begin
  CheckInactive;
  InternalInitFieldDefs;

  // create the new file
  if FileExists (FTableName) then
    raise EMdDataSetError.Create ('File ' + FTableName + ' already exists');
  FStream := TFileStream.Create (FTableName, fmCreate or fmShareExclusive);
  try
    // save the header
    FDataFileHeader.VersionNumber := HeaderVersion;
    FDataFileHeader.RecordSize := 0;    // used later
    FDataFileHeader.RecordCount := 0;  // empty
    FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
  finally
    // close the file
    FStream.Free;
  end;
end;

```

## Section II: Movement and Bookmark Management

As mentioned earlier, every dataset must implement *bookmark management*, which is necessary for navigating through the dataset. Logically, a bookmark is a reference to a specific dataset record, something that uniquely identifies the record so a dataset can access it and compare it to other records. Technically, bookmarks are pointers. You can implement them as pointers to specific data structures that store record information, or you can implement them as record numbers. For simplicity, I'll use the latter approach.

Given a bookmark, you should be able to find the corresponding record; but given a record buffer, you should also be able to retrieve the corresponding bookmark. This is the reason for appending the `TMdRecInfo` structure to the record data in each record buffer. This data structure stores the bookmark for the record in the buffer, as well as some bookmark flags defined as follows:

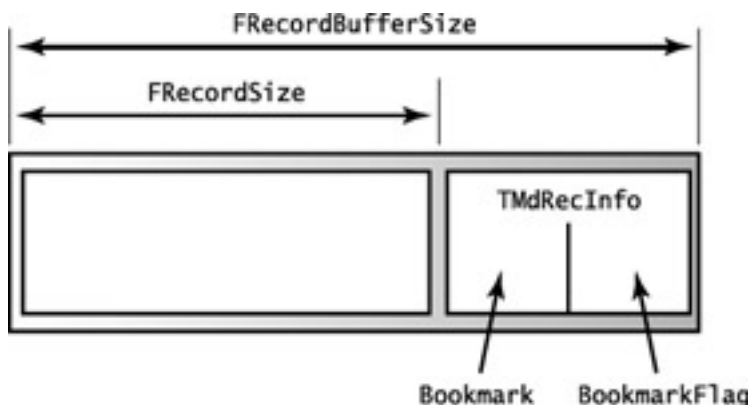
```

type
  TBookmarkFlag = (bfCurrent, bfBOF, bfEOF, bfInserted);

```

The system will request that you store these flags in each record buffer and will later ask you to retrieve the flags for a given record buffer.

To summarize, the structure of a record buffer stores the record data, the bookmark, and the bookmark flags, as you can see in [Figure 17.5](#).



**Figure 17.5:** The structure of each buffer of the custom dataset, along with the various local fields referring to its subportions



To access the bookmark and flags, you can use as an offset the size of the data, casting the value to the `PMdRecInfo` pointer type, and then access the proper field of the `TMdRecInfo` structure via the pointer. The two methods used to set and get the bookmark flags demonstrate this technique:

```
procedure TMdCustomDataSet.SetBookmarkFlag (Buffer: PChar;
    Value: TBookmarkFlag);
begin
    PMdRecInfo(Buffer + FRecordSize).BookmarkFlag := Value;
end;

function TMdCustomDataSet.GetBookmarkFlag (Buffer: PChar): TBookmarkFlag;
begin
    Result := PMdRecInfo(Buffer + FRecordSize).BookmarkFlag;
end;
```

The methods you use to set and get a record's current bookmark are similar to the previous two, but they add complexity because you receive a pointer to the bookmark in the `Data` parameter. Casting the value referenced by this pointer to an integer, you obtain the bookmark value:

```
procedure TMdCustomDataSet.GetBookmarkData (Buffer: PChar; Data: Pointer);
begin
    Integer(Data^) := PMdRecInfo(Buffer + FRecordSize).Bookmark;
end;

procedure TMdCustomDataSet.SetBookmarkData (Buffer: PChar; Data: Pointer);
begin
    PMdRecInfo(Buffer + FRecordSize).Bookmark := Integer(Data^);
end;
```

The key bookmark management method is `InternalGotoBookmark`, which your dataset uses to make a given record the current one. This isn't the standard navigation technique—it's much more common to move to the next or previous record (something you can accomplish using the `GetRecord` method presented in the [next section](#)), or to move to the first or last record (something you'll accomplish using the `InternalFirst` and `InternalLast` methods described shortly).

Oddly enough, the `InternalGotoBookmark` method doesn't expect a bookmark parameter, but a pointer to a bookmark, so you must dereference it to determine the bookmark value. You use the following method, `InternalSetToRecord`, to jump to a given bookmark, but it must extract the bookmark from the record buffer passed as a parameter. Then, `InternalSetToRecord` calls `InternalGotoBookmark`. Here are the two methods:

```
procedure TMdCustomDataSet.InternalGotoBookmark (Bookmark: Pointer);
var
    ReqBookmark: Integer;
begin
    ReqBookmark := Integer (Bookmark^);
    if (ReqBookmark >= 0) and (ReqBookmark < InternalRecordCount) then
        FCurrentRecord := ReqBookmark
    else
        raise EMdDataSetError.Create ('Bookmark ' +
            IntToStr (ReqBookmark) + ' not found');
end;

procedure TMdCustomDataSet.InternalSetToRecord (Buffer: PChar);
var
    ReqBookmark: Integer;
begin
    ReqBookmark := PMdRecInfo(Buffer + FRecordSize).Bookmark;
    InternalGotoBookmark (@ReqBookmark);
```

```
end;
```

In addition to the bookmark management methods just described, you use several other navigation methods to move to specific positions within the dataset, such as the first or last record. These two methods don't really move the current record pointer to the first or last record, but move it to one of two special locations before the first record and after the last one. These are not actual records: Borland calls them *cracks*. The beginning-of-file crack, or `BofCrack`, has the value `-1` (set in the `InternalOpen` method), because the position of the first record is zero. The end-of-file crack, or `EofCrack`, has the value of the number of records, because the last record has the position `FRecordCount - 1`. I used two local fields, called `EofCrack` and `BofCrack`, to make this code easier to read:

```
procedure TMdCustomDataSet.InternalFirst;
begin
    FCurrentRecord := BofCrack;
end;
```

```
procedure TMdCustomDataSet.InternalLast;
begin
    EofCrack := InternalRecordCount;
    FCurrentRecord := EofCrack;
end;
```

The `InternalRecordCount` method is a virtual method introduced in my `TMdCustomDataSet` class, because different datasets can either have a local field for this value (as in case of the stream-based dataset, which has an `FRecordCount` field) or compute it on the fly.

Another group of optional methods is used to get the current record number (used by the `DBGrid` component to show a proportional vertical scroll bar), set the current record number, or determine the number of records. These methods are easy to understand, if you recall that the range of the internal `FCurrentRecord` field is from 0 to the number of records minus 1. In contrast, the record number reported to the system ranges from 1 to the number of records:

```
function TMdCustomDataSet.GetRecordCount: Longint;
begin
    CheckActive;
    Result := InternalRecordCount;
end;
```

```
function TMdCustomDataSet.GetRecNo: Longint;
begin
    UpdateCursorPos;
    if FCurrentRecord < 0 then
        Result := 1
    else
        Result := FCurrentRecord + 1;
end;
```

```
procedure TMdCustomDataSet.SetRecNo(Value: Integer);
begin
    CheckBrowseMode;
    if (Value > 1) and (Value <= FRecordCount) then
        begin
            FCurrentRecord := Value - 1;
            Resync([]);
        end;
end;
```

Notice that the generic custom dataset class implements all the methods of this section. The derived stream-based dataset doesn't need to modify any of them.

## Section III: Record Buffers and Field Management

Now that we've covered all the support methods, let's examine the core of a custom dataset. In addition to opening and creating records and moving around between them, the component needs to move the data from the stream (the persistent file) to the record buffers, and from the record buffers to the `TField` objects that are connected to the data-aware controls. The management of record buffers is complex, because each dataset also needs to allocate, empty, and free the memory it requires:

```
function TmCustomDataSet.AllocRecordBuffer: PChar;
begin
    GetMem (Result, FRecordBufferSize);
end;

procedure TmCustomDataSet.FreeRecordBuffer (var Buffer: PChar);
begin
    FreeMem (Buffer);
end;
```

You allocate memory this way because a dataset generally adds more information to the record buffer, so the system has no way of knowing how much memory to allocate. Notice that in the `AllocRecordBuffer` method, the component allocates the memory for the record buffer, including both the database data and the record information. In the `InternalOpen` method, I wrote the following:

```
FRecordBufferSize := InternalRecordSize + sizeof (TmRecInfo);
```

The component also needs to implement a function to reset the buffer (`InternalInitRecord`), usually filling it with numeric zeros or spaces.

Oddly enough, you must also implement a method that returns the size of each record, but only the data portion—not the entire record buffer. This method is necessary for implementing the read-only `RecordSize` property, which is used only in a couple of peculiar cases in the entire VCL source code. In the generic custom dataset, the `GetRecordSize` method returns the value of the `FRecordSize` field.

Now we've reached the core of the custom dataset component. The methods in this group are `GetRecord`, which reads data from the file; `InternalPost` and `InternalAddRecord`, which update or add new data to the file; and `InternalDelete`, which removes data and is not implemented in the sample dataset.

The most complex method of this group is `GetRecord`, which serves multiple purposes. The system uses this method to retrieve the data for the current record, fill a buffer passed as a parameter, and retrieve the data of the next or previous records. The `GetMode` parameter determines its action:

```
type
    TGetMode = (gmCurrent, gmNext, gmPrior);
```

Of course, a previous or next record might not exist. Even the current record might not exist—for example, when the table is empty (or in case of an internal error). In these cases you don't retrieve the data but return an error code. Therefore, this method's result can be one of the following values:

```
type
    TGetResult = (grOK, grBOF, grEOF, grError);
```

Checking to see if the requested record exists is slightly different than you might expect. You don't have to determine if the current record is in the proper range, only if the requested record is. For example, in the `gmCurrent` branch of the case statement, you use the standard expression `CurrentRecord >= InternalRecordCount`. To fully understand the various cases, you might want to read the code a couple of times.

It took me some trial and error (and system crashes caused by recursive calls) to get the code straight when I wrote my first custom dataset a few years back. To test it, consider that if you use a `DBGrid`, the system will perform a series of `GetRecord` calls, until either the grid is full or `GetRecord` return `grEOF`. Here's the entire code for the `GetRecord`

method:

```
// III: Retrieve data for current, previous, or next record
// (moving to it if necessary) and return the status
function TmdCustomDataSet.GetRecord(Buffer: PChar;
  GetMode: TGetMode; DoCheck: Boolean): TGetResult;
begin
  Result := grOK; // default
  case GetMode of
    gmNext: // move on
      if FCurrentRecord < InternalRecordCount - 1 then
        Inc (FCurrentRecord)
      else
        Result := grEOF; // end of file
    gmPrior: // move back
      if FCurrentRecord > 0 then
        Dec (FCurrentRecord)
      else
        Result := grBOF; // begin of file
    gmCurrent: // check if empty
      if FCurrentRecord >= InternalRecordCount then
        Result := grError;
  end;
  // load the data
  if Result = grOK then
    InternalLoadCurrentRecord (Buffer)
  else if (Result = grError) and DoCheck then
    raise EMdDataSetError.Create ('GetRecord: Invalid record');
end;
```

If there's an error and the DoCheck parameter was True, GetRecord raises an exception. If everything goes fine during record selection, the component loads the data from the stream, moving to the position of the current record (given by the record size multiplied by the record number). In addition, you need to initialize the buffer with the proper bookmark flag and bookmark (or record number) value. This is accomplished by another virtual method I introduced, so that derived classes will only need to implement this portion of the code, while the complex GetRecord method remains unchanged:

```
procedure TmdDataSetStream.InternalLoadCurrentRecord (Buffer: PChar);
begin
  FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
  FStream.ReadBuffer (Buffer^, FRecordSize);
  with PMdRecInfo(Buffer + FRecordSize)^ do
    begin
      BookmarkFlag := bfCurrent;
      Bookmark := FCurrentRecord;
    end;
end;
```

You move data to the file in two different cases: when you modify the current record (that is, a post after an edit) or when you add a new record (a post after an insert or append). You use the InternalPost method in both cases, but you can check the dataset's State property to determine which type of post you're performing. In both cases, you don't receive a record buffer as a parameter; so, you must use the ActiveRecord property of TDataSet, which points to the buffer for the current record:

```
procedure TmdDataSetStream.InternalPost;
begin
  CheckActive;
  if State = dsEdit then
```

```

begin
    // replace data with new data
    FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
end
else
begin
    // always append
    InternalLast;
    FStream.Seek (0, soFromEnd);
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
    Inc (FRecordCount);
end;
end;

```

In addition, there's another related method: `InternalAddRecord`. This method is called by the `AddRecord` method, which in turn is called by `InsertRecord` and `AppendRecord`. These last two are public methods a user can call. This is an alternative to inserting or appending a new record to the dataset, editing the values of the various fields, and then posting the data, because the `InsertRecord` and `AppendRecord` calls receive the values of the fields as parameters. All you must do at that point is replicate the code used to add a new record in the `InternalPost` method:

```

procedure TMDDataSetOne.InternalAddRecord(Buffer: Pointer; Append: Boolean);
begin
    // always append at the end
    InternalLast;
    FStream.Seek (0, soFromEnd);
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
    Inc (FRecordCount);
end;

```

I should also have implemented a file operation that removes the current record. This operation is common, but it is complex. If you take a simple approach, such as creating an empty spot in the file, then you'll need to keep track of that spot and make the code for reading or writing a specific record work around that location. An alternate solution is to make a copy of the entire file without the given record and then replace the original file with the copy. Given these choices, I felt that for this example I could forgo supporting record deletion.

## Section IV: From Buffers to Fields

In the last few methods, you've seen how datasets move data from the data file to the memory buffer. However, there's little Delphi can do with this record buffer, because it doesn't yet know how to interpret the data in the buffer. You need to provide two more methods: `GetData`, which copies the data from the record buffer to the field objects of the dataset, and `SetData`, which moves the data back from the fields to the record buffer. Delphi will automatically move the data from the field objects to the data-aware controls and back.

The code for these two methods isn't difficult, primarily because you saved the field offsets inside the record data in a `TList` object called `FFieldOffset`. By incrementing the pointer to the initial position in the record buffer of the current field's offset, you can get the specific data, which takes `Field.DataSize` bytes.

A confusing element of these two methods is that they both accept a `Field` parameter and a `Buffer` parameter. At first, you might think the buffer passed as a parameter is the record buffer. However, I found out that the `Buffer` is a pointer to the field object's raw data. If you use one of the field object's methods to move that data, it will call the dataset's `GetData` or `SetData` method, probably causing an infinite recursion. Instead, you should use the `ActiveBuffer` pointer to access the record buffer, use the proper offset to get to the data for the current field in the record buffer, and then use the provided `Buffer` to access the field data. The only difference between the two methods is the direction you move the data:

```

function TmDataSetOne.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
    FieldOffset: Integer;
    Ptr: PChar;
begin
    Result := False;
    if not IsEmpty and (Field.FieldNo > 0) then
        begin
            FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
            Ptr := ActiveBuffer;
            Inc (Ptr, FieldOffset);
            if Assigned (Buffer) then
                Move (Ptr^, Buffer^, Field.DataSize);
            Result := True;
            if (Field is TDateTimeField) and (Integer(Ptr^) = 0) then
                Result := False;
            end;
        end;

procedure TmDataSetOne.SetFieldData(Field: TField; Buffer: Pointer);
var
    FieldOffset: Integer;
    Ptr: PChar;
begin
    if Field.FieldNo >= 0 then
        begin
            FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
            Ptr := ActiveBuffer;
            Inc (Ptr, FieldOffset);
            if Assigned (Buffer) then
                Move (Buffer^, Ptr^, Field.DataSize)
            else
                raise Exception.Create (
                    'Very bad error in TmDataSetStream.SetField data');
                DataEvent (deFieldChange, Longint(Field));
            end;
        end;

```

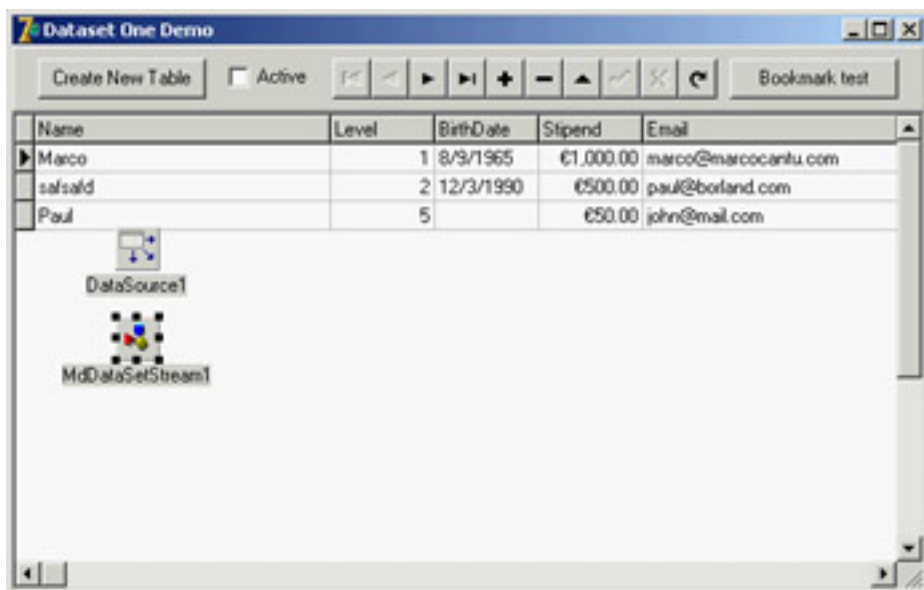
The `GetField` method should return `True` or `False` to indicate whether the field contains data or is empty (a null field, to be more precise). However, unless you use a special marker for blank fields, it's difficult to determine this condition, because you're storing values of different data types. For example, a test such as `Ptr^<>#0` makes sense only if you are using a string representation for all the fields. If you use this test, zero integer values and empty strings will show as null values (the data-aware controls will be empty), which may be what you want. The problem is that `Boolean False` values won't show up. Even worse, floating-point values with no decimals and few digits won't be displayed, because the exponent portion of their representation will be zero. However, to make this example work, I had to consider as empty each date/time field with an initial zero. Without this code, Delphi tries to convert the illegal *internal* zero date (internally, date fields don't use a `TDateTime` data type but a different representation), raising an exception. The code used to work with past versions of Delphi.

**Warning** While trying to fix this problem, I also found out that if you call `IsNull` for a field, this request is resolved by calling `GetFieldData` without passing any buffer to fill but looking only for the result of the function call. This is the reason for the `if Assigned (Buffer)` test within the code.

There's one final method, which doesn't fall into any category: `InternalHandleException`. Generally, this method silences the exception, because it is activated only at design time.

## Testing the Stream-Based DataSet

After all this work, you're ready to test an application example of the custom dataset component, which is installed in the component's package for this chapter. The form displayed by the StreamDSDemo program is simple, as you can see in [Figure 17.6](#). It has a panel with two buttons, a check box, and a navigator component, plus a DBGrid filling its client area.



**Figure 17.6:** The form of the StreamDSDemo example. The custom dataset has been activated, so you can already see the data at design time.

[Figure 17.6](#) shows the example's form at design time, but I activated the custom dataset so that its data is visible. I already prepared the INI file with the table definition (the file listed earlier when discussing the dataset initialization), and I executed the program to add some data to the file.

You can also modify the form using Delphi's Fields editor and set the properties of the various field objects. Everything works as it does with one of the standard dataset controls. However, to make this work, you must enter the name of the custom dataset's file in the `TableName` property, using the complete path.

**Warning** The demo program defines the absolute path of the table file at design time, so you'll need to fix it if you copy the examples to a different drive or directory. In the example, the `TableName` property is used only at design time. At run time, the program looks for the table in the current directory.

The example code is simple, especially compared to the custom dataset code. If the table doesn't exist yet, you can click the Create New Table button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    MdDataSetStream1.CreateTable;
    MdDataSetStream1.Open;
    CheckBox1.Checked := MdDataSetStream1.Active;
end;
```

You create the file first, opening and closing it within the `CreateTable` call, and then open the table. This is the same behavior as the `TTable` component (which accomplishes this step using the `CreateTable` method). To open or close the table, you can click the check box:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
    MdDataSetStream1.Active := CheckBox1.Checked;
end;
```

Finally, I created a method that tests the custom dataset's bookmark management code (it works).

**Team LiB**

◀ PREVIOUS

NEXT ▶



## A Directory in a Dataset

An important idea related to datasets in Delphi is that they represent a set of data, regardless of where this data comes from. A SQL server and a local file are examples of traditional datasets, but you can use the same technology to show a list of a system's users, a list of a folder's files, the properties of objects, XML-based data, and so on.

As an example, the second dataset presented in this chapter is a list of files. I've built a generic dataset based on a list of objects in memory (using a `TObjectList`), and then derived a version in which the objects correspond to a folder's files. The example is simplified by the fact that it is a read-only dataset, so you might find it more straightforward than the previous dataset.

**Note** Some of the ideas presented here were discussed in an article I wrote for the Borland Community website, published in June 2000 at the URL [bdn.borland.com/article/0,1410,20587,00.html](http://bdn.borland.com/article/0,1410,20587,00.html).

## A List as a Dataset

The generic list-based dataset is called `TMDListDataSet` and contains the list of objects, a list that is created when you open the dataset and freed when you close it. This dataset doesn't store the record data within the buffer; rather, it saves in the buffer only the position in the list of the entry corresponding to the record's data. This is the class definition:

```
type
  TMDListDataSet = class (TMDCustomDataSet)
  protected
    // the list holding the data
    FList: TObjectList;
    // dataset virtual methods
    procedure InternalPreOpen; override;
    procedure InternalClose; override;
    // custom dataset virtual methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord (Buffer: PChar); override;
  end;
```

You can see that by writing a generic custom data class, you can override a few virtual methods of the `TDataSet` class and of this custom dataset class, and have a working dataset (although this is still an abstract class, which requires extra code from subclasses to work). When the dataset is opened, you have to create the list and set the record size, to indicate you're saving the list index in the buffer:

```
procedure TMDListDataSet.InternalPreOpen;
begin
  FList := TObjectList.Create (True); // owns the objects
  FRecordSize := 4; // an integer, the list item id
end;
```

Further derived classes at this point should also fill the list with objects.

**Tip** Like the `ClientDataSet`, my list dataset keeps its data in memory. However, using some smart techniques, you can also create a list of fake objects and then load the actual objects only when you are accessing them.

Closing is a matter of freeing the list, which has a record count corresponding to the list size:

```
function TMDListDataSet.InternalRecordCount: Integer;
begin
  Result := fList.Count;
end;
```

The only other method saves the current record's data in the record buffer, including the bookmark information. The core data is the position of the current record, which matches the list index (and also the bookmark):

```
procedure TMdListDataSet.InternalLoadCurrentRecord (Buffer: PChar);  
begin  
  PInteger (Buffer)^ := fCurrentRecord;  
  with PMdRecInfo(Buffer + FRecordSize)^ do  
    begin  
      BookmarkFlag := bfCurrent;  
      Bookmark := fCurrentRecord;  
    end;  
end;
```

## Directory Data

The derived directory dataset class has to provide a way to load the objects in memory when the dataset is opened, to define the proper fields, and to read and write the value of those fields. It also has a property indicating the directory to work on—or, to be more precise, the directory plus the file mask used for filtering the files (such as `c:\docs\*.txt`):

```
type  
  TMdDirDataset = class(TMdListDataSet)  
    private  
      FDirectory: string;  
      procedure SetDirectory(const NewDirectory: string);  
    protected  
      // TDataSet virtual methods  
      procedure InternalInitFieldDefs; override;  
      procedure SetFieldData(Field: TField; Buffer: Pointer); override;  
      function GetCanModify: Boolean; override;  
      // custom dataset virtual methods  
      procedure InternalAfterOpen; override;  
    public  
      function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;  
    published  
      property Directory: string read FDirectory write SetDirectory;  
    end;
```

The `GetCanModify` function is another virtual method of `TDataSet`, used to determine if the dataset is read-only. In this case, it returns `False`. You don't have to write any code for the `SetFieldData` procedure, but you must define it because it is an abstract virtual method.

Because you are dealing with a list of objects, the unit includes a class for those objects. In this case, the file data is extracted from a `TSearchRec` buffer by the `TFileData` class constructor:

```
type  
  TFileData = class  
    public  
      ShortFileName: string;  
      Time: TDateTime;  
      Size: Integer;  
      Attr: Integer;  
      constructor Create (var FileInfo: TSearchRec);  
    end;  
  
constructor TFileData.Create (var FileInfo: TSearchRec);  
begin  
  ShortFileName := FileInfo.Name;  
  Time := FileDateToDateTime (FileInfo.Time);
```

```

Size := FileInfo.Size;
Attr := FileInfo.Attr;
end;

```

This constructor is called for each folder while opening the dataset:

```

procedure TMdDirDataset.InternalAfterOpen;
var
  Attr: Integer;
  FileInfo: TSearchRec;
  FileData: TFileData;
begin
  // scan all files
  Attr := faAnyFile;
  FList.Clear;
  if SysUtils.FindFirst(fDirectory, Attr, FileInfo) = 0 then
  repeat
    FileData := TFileData.Create (FileInfo);
    FList.Add (FileData);
  until SysUtils.FindNext(FileInfo) <> 0;
  SysUtils.FindClose(FileInfo);
end;

```

The next step is to define the fields of the dataset, which in this case are fixed and depend on the available directory data:

```

procedure TMdDirDataset.InternalInitFieldDefs;
begin
  if fDirectory = '' then
    raise EMdDataSetError.Create ('Missing directory');

  // field definitions
  FieldDefs.Clear;
  FieldDefs.Add ('FileName', ftString, 40, True);
  FieldDefs.Add ('TimeStamp', ftDateTime);
  FieldDefs.Add ('Size', ftInteger);
  FieldDefs.Add ('Attributes', ftString, 3);
  FieldDefs.Add ('Folder', ftBoolean);
end;

```

Finally, the component has to move the data from the list object referenced by the current record buffer (the `ActiveBuffer` value) to each field of the dataset, as requested by the `GetFieldData` method. This function uses either `Move` or `StrCopy`, depending on the data type, and it does some conversions for the attribute codes (*H* for hidden, *R* for read-only, and *S* for system) extracted from the related flags and used to determine whether a file is a folder. Here is the code:

```

function TMdDirDataset.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FileData: TFileData;
  Bool1: WordBool;
  strAttr: string;
  t: TDateTimeRec;
begin
  FileData := fList [Integer(ActiveBuffer^)] as TFileData;
  case Field.Index of
    0: // filename
      StrCopy (Buffer, pchar(FileData.ShortFileName));
    1: // timestamp

```

```

begin
  t := DateTimeToNative (ftdatetime, FileData.Time);
  Move (t, Buffer^, sizeof (TDateTime));
end;
2: // size
  Move (FileData.Size, Buffer^, sizeof (Integer));
3: // attributes
begin
  strAttr := '   ';
  if (FileData.Attr and SysUtils.faReadOnly) > 0 then
    strAttr [1] := 'R';
  if (FileData.Attr and SysUtils.faSysFile) > 0 then
    strAttr [2] := 'S';
  if (FileData.Attr and SysUtils.faHidden) > 0 then
    strAttr [3] := 'H';
  StrCopy (Buffer, pchar(strAttr));
end;
4: // folder
begin
  Bool1 := FileData.Attr and SysUtils.faDirectory > 0;
  Move (Bool1, Buffer^, sizeof (WordBool));
end;
end; // case
Result := True;
end;

```

The tricky part in writing this code was figuring out the internal format of dates stored in date/time fields. This is not the common TDateTime format used by Delphi, and not even the internal TTimeStamp, but what is internally called the *native date and time format*. I've written a conversion function by cloning one I found in the VCL code for the date/time fields:

```

function DateTimeToNative(DataType: TFieldType; Data: TDateTime): TDateTimeRec;
var
  TimeStamp: TTimeStamp;
begin
  TimeStamp := DateTimeToTimeStamp(Data);
  case DataType of
    ftDate: Result.Date := TimeStamp.Date;
    ftTime: Result.Time := TimeStamp.Time;
  else
    Result.DateTime := TimeStampToMsecs(TimeStamp);
  end;
end;

```

With this dataset available, building the demo program (shown in [Figure 17.7](#)) was just a matter of connecting a DBGrid component to the dataset and adding a folder-selection component, the sample ShellTreeView control. This control is set up to work only on files, by setting its Root property to C:\. When the user selects a new folder, the OnChange event handler of the ShellTreeView control refreshes the dataset:

```

procedure TForm1.ShellTreeView1Change(Sender: TObject; Node: TTreeNode);
begin
  MdDirDataset1.Close;
  MdDirDataset1.Directory := ShellTreeView1.Path + '\*.*';
  MdDirDataset1.Open;
end;

```

FileName	TimeStamp	Size	Attributes	Folder
.	9/19/2002 13:49:18 AM			True
..	9/19/2002 13:49:18 AM			True
delphi32.bpl	8/9/2002 2:30:00 PM	14,272		False
delphi32.exe	8/9/2002 2:30:00 PM	545,792		False
DELPHI32.BCF	8/9/2002 2:30:00 PM	6,644		False
delphi32.dac	9/14/2002 6:28:32 PM	3,742		False
delphi32.dcl	8/9/2002 4:30:00 PM	66,560		False
delphi32.exe_80_70.bpl	8/9/2002 4:30:00 PM	140,256		False
TIIDelphi32.dac	8/9/2002 4:30:00 PM	4,449		False
Delphi.vpp	8/9/2002 4:30:00 PM	3,320		False
ETH1-arcoc.dcl	8/9/2002 4:30:00 PM	18,796		False
ETH2.dcl	8/9/2002 4:30:00 PM	19,340		False
ETH3.2.dcl	8/9/2002 4:30:00 PM	21,076		False
ETH.lan1.exe	8/9/2002 4:30:00 PM	11,564		False
ETH-special.exe	8/9/2002 4:30:00 PM	4,114		False
ETH-special.exe	8/9/2002 4:30:00 PM	14,447		False

**Figure 17.7:** The output of the DirDemo example, which uses an unusual dataset that shows directory data

**Warning** If your version of Windows has problems with the sample shell controls available in Delphi, you can use the DirDemoNoShell version of the example, which uses the old-fashioned Windows 3.1-compatible Delphi file controls.

## A Dataset of Objects

As you saw in the previous example, a list of objects is conceptually similar to the rows of a table in a dataset. In Delphi, you can build a dataset wrapping a list of objects, as in the case of the `TFileData` class. It's intriguing to extend this example to build a dataset that supports generic objects, which you can do thanks to the extended RTTI available in Delphi.

This dataset component inherits from `TMDListDataSet`, as in the previous example. You must provide a single setting: the target class, stored in the `ObjClass` property (see the complete definition of the `TMdObjDataSet` class in [Listing 17.5](#)).

### Listing 17.5: The Complete Definition of the *TMdObjDataSet* Class

```

type
  TMdObjDataSet = class(TMDListDataSet)
  private
    PropList: PPropList;
    nProps: Integer;
    FObjClass: TPersistentClass;
    ObjClone: TPersistent;
    FChangeToClone: Boolean;
    procedure SetObjClass (const Value: TPersistentClass);
    function GetObjects (I: Integer): TPersistent;
    procedure SetChangeToClone (const Value: Boolean);
  protected
    procedure InternalInitFieldDefs; override;
    procedure InternalClose; override;
    procedure InternalInsert; override;
    procedure InternalPost; override;
    procedure InternalCancel; override;
    procedure InternalEdit; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    function GetCanModify: Boolean; override;
    procedure InternalPreOpen; override;
  public
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
    property Objects [I: Integer]: TPersistent read GetObjects;
    function Add: TPersistent;
  published
    property ObjClass: TPersistentClass read FObjClass write SetObjClass;
    property ChangesToClone: Boolean read FChangeToClone
      write SetChangeToClone default False;
  end;

```

The class is used by the `InternalInitFieldDefs` method to determine the dataset fields based on the published properties of the target class, which are extracted using RTTI:

```

procedure TMdObjDataSet.InternalInitFieldDefs;
var
  i: Integer;
begin
  if FObjClass = nil then
    raise Exception.Create ('TMdObjDataSet: Unassigned class');

  // field definitions

```

```

FieldDefs.Clear;
nProps := GetTypeData(fObjClass.ClassInfo)^.PropCount;
GetMem(PropList, nProps * SizeOf(Pointer));
GetPropInfos (fObjClass.ClassInfo, PropList);

for i := 0 to nProps - 1 do
  case PropList [i].PropType^.Kind of
    tkInteger, tkEnumeration, tkSet:
      FieldDefs.Add (PropList [i].Name, ftInteger, 0);
    tkChar: FieldDefs.Add (PropList [i].Name, ftFixedChar, 0);
    tkFloat: FieldDefs.Add (PropList [i].Name, ftFloat, 0);
    tkString, tkLString:
      FieldDefs.Add (PropList [i].Name, ftString, 50); // TODO: fix size
    tkWString: FieldDefs.Add (PropList [i].Name, ftWideString, 50);
      // TODO: fix size
  end;
end;

```

Similar RTTI-based code is used in the `GetFieldData` and `SetFieldData` methods to access the properties of the current object when a dataset field access operation is requested. The huge advantage in using properties to access the dataset data is that read and write operations can be mapped directly to data but also use the corresponding method. This way, you can write the *business rules* of your application by implementing rules in the read and write methods of the properties—definitely a sounder OOP approach than hooking code to field objects and validating them.

Here is a slightly simplified version of `GetFieldData` (the other method is symmetric):

```

function TObjDataSet.GetFieldData (
  Field: TField; Buffer: Pointer): Boolean;
var
  Obj: TPersistent;
  TypeInfo: PTypeInfo;
  IntValue: Integer;
  FlValue: Double;
begin
  if FList.Count = 0 then
    begin
      Result := False;
      exit;
    end;
  Obj := fList [Integer(ActiveBuffer^)] as TPersistent;
  TypeInfo := PropList [Field.FieldNo-1]^ .PropType^;
  case TypeInfo.Kind of
    tkInteger, tkChar, tkWChar, tkClass, tkEnumeration, tkSet:
      begin
        IntValue := GetOrdProp(Obj, PropList [Field.FieldNo-1]);
        Move (IntValue, Buffer^, sizeof (Integer));
      end;
    tkFloat:
      begin
        FlValue := GetFloatProp(Obj, PropList [Field.FieldNo-1]);
        Move (FlValue, Buffer^, sizeof (Double));
      end;
    tkString, tkLString, tkWString:
      StrCopy (Buffer, pchar(GetStrProp(Obj, PropList [Field.FieldNo-1])));
  end;
  Result := True;
end;

```

This pointer-based code may look terrible, but if you've endured the discussion of the technical details of developing a custom dataset, it won't add much complexity to the picture. It uses some of the data structures defined (and briefly commented) in the `TTypeInfo` unit, which should be your reference for any questions about the previous code.

Using this naïve approach of editing the object's data directly, you might wonder what happens if a user cancels the editing operation (something Delphi generally accounts for). My dataset provides two alternative approaches, controlled by the `ChangesToClone` property and based on the idea of cloning objects by copying their published properties. The core `DoClone` procedure uses RTTI code similar to what you have already seen to copy all the published data of an object into another object, creating an effective copy (or a clone).

This cloning takes place in both cases. Depending on the value of the `ChangesToClone` property, either the edit operations are performed on the clone object, which is then copied over the actual object during the Post operation; or the edit operations are performed on the actual object, and the clone is used to get back the original values if editing terminates with a Cancel request. This is the code of the three methods involved:

```
procedure TObjDataSet.InternalEdit;
begin
    DoClone (fList [FCurrentRecord] as TDbPers, ObjClone);
end;

procedure TObjDataSet.InternalPost;
begin
    if FChangeToClone and Assigned (ObjClone) then
        DoClone (ObjClone, TDbPers (fList [fCurrentRecord]));
end;

procedure TMdObjDataSet.InternalCancel;
begin
    if not FChangeToClone and Assigned (ObjClone) then
        DoClone (ObjClone, TPersistent(fList [fCurrentRecord]));
end;
```

In the `SetFieldData` method, you have to modify either the cloned object or the original instance. To make things more complicated, you must also consider this difference in the `GetFieldData` method: If you are reading fields from the current object, you might have to use its modified clone (otherwise, the user's changes to other fields will disappear).

As you can see in [Listing 17.5](#), the class also has an `Objects` array that accesses the data in an OOP way and an `Add` method that's similar to the `Add` method of a collection. By calling `Add`, the code creates a new empty object of the target class and adds it to the internal list:

```
function TMdObjDataSet.Add: TPersistent;
begin
    if not Active then
        Open;
    Result := fObjClass.Create;
    fList.Add (Result);
end;
```

To demonstrate the use of this component, I wrote the `ObjDataSetDemo` example. It has a demo target class with a few fields and buttons to create objects automatically, as you can see in [Figure 17.8](#). The most interesting feature of the program, however, is something you have to try for yourself. Run the program and look at the `DbGrid` columns. Then edit the target class, `TDemo`, adding a new published property to it. Run the program again, and the grid will include a new column for the property.



The screenshot shows a window titled "ObjDataSetDemo" with a toolbar containing buttons for "Add Object", "Descriptions", and navigation controls. Below the toolbar is a table with the following data:

Name	Age	Amount	Value
John	21	3242.43	12
Mark	33	6716.54	32
Joseph	28	3722.38	33
Bill	24	4747.94	23
John	62	597.24	14

Figure 17.8: The ObjDataSet-Demo example showcases a dataset mapped to objects using RTTI.

## What's Next?

In this chapter we've delved into Delphi's database architecture by first examining the development of data-aware controls and then studying the internals of the `TDataSet` class to write a couple of custom dataset components. With this information and the other ideas presented in this part of the book, you should be able to choose the architecture of your database applications, depending on your needs.

Database programming is a core element of Delphi, which is why I've devoted several chapters to this topic. I'll continue to cover it in the [next chapter](#), which is devoted to the new reporting engine available in Delphi 7. We'll get back to databases when focusing on presenting data over the Web in [Chapters 20](#) and [21](#), and also when discussing XML and SOAP in [Chapters 22](#) and [23](#).

## Chapter 18: Reporting with Rave

### Overview

Database applications let you view and edit data, but quite often their output should be physically printed to paper. Technically, Delphi supports printing in many different ways, from direct text output to the use of the printer `Canvas`, from sophisticated database reporting to the generation of documents in various formats (from Microsoft's Word to Sun's OpenOffice). In this chapter we'll focus exclusively on reports and in particular on the use of the third-party reporting Rave engine included in Delphi 7. If you are interested in other Delphi techniques for driving the printer, see the related material on my website (discussed in [Appendix C](#), "Free Companion Books on Delphi").

Reporting tools are important because they can perform complex processing by themselves: the reporting subsystem can become a stand-alone application. Although this chapter focuses on how you can produce a report from the dataset within your Delphi programs, you should always keep in mind the autonomous nature of reports evaluating such a tool.

Reports must be created with care, because they represent a user interface for your applications that goes beyond and is sometimes more significant than the software itself. Many more people will probably look at the printed reports than just the users who produce the reports using the programs. For this reason, it's important to have good-quality reports and a flexible architecture to let users customize them.

**Note** This chapter was written with the help of Jim Gunkel of Nevrona Designs, the company that developed the Rave engine.

## Introducing Rave

Reports are a primary means of retrieving information from the data being managed by an application. To solve the problems associated with presenting a visual report of data in a meaningful and informative manner, traditional visual reporting applications have offered banded layout tools geared toward table-style data listings. Today, however, more complex reporting requirements exist that are not easily handled by banded layout tools.

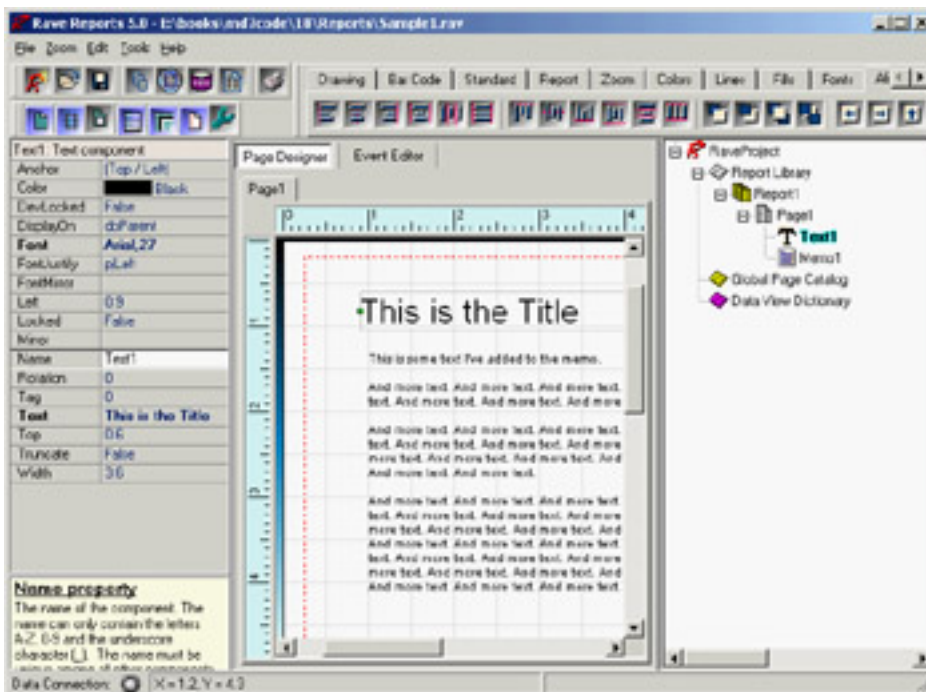
Rave Reports is a visual report design environment offering many unique features that help make the reporting process simpler, quicker, and more efficient. Rave can handle a wide variety of report formats and includes advanced technologies such as mirroring, to encourage the reuse of report contents for quicker changes and easier maintenance.

This chapter presents a brief introduction to Rave's features. Additional information about Rave can be found in the online Help files, in the PDF documentation on Delphi's CD, in several demo projects, and on the producer's website, [www.nevrona.com](http://www.nevrona.com).

**Note** A key feature of Rave—and one of the reasons Borland chose it over other solutions—is that it is a completely cross-platform solution you can use on both Windows and Linux. Not only do Rave components integrate with both the VCL and CLX, but the Rave Designer is itself a cross-platform application written in CLX.

## Rave: The Report Authoring Visual Environment

To start the Rave visual report design environment, double-click on a `TRvProject` component dropped on a form or select `Tools` → `Rave Designer` in the Delphi IDE. When you activate this environment, you'll see a window like that shown in [Figure 18.1](#). As you can see, the Rave Designer includes several sections: the Page Designer, Event Editor, Property panel, Project Tree panel, toolbars, Toolbar Palette, and status bar.



**Figure 18.1:** The Rave Designer with a simple report

**Note** Any time you want to see the result of your efforts, press F9 in the Rave Designer for a preview of the current report.

Keep in mind that Rave allows your end users to create or modify their own reports. You can set the Rave Designer level to Beginner, Intermediate, or Advanced in the `Edit` → `Preferences` dialog box (in the `Environment` section) so your

end users will be working at the level they're comfortable with and won't have more power than you want to give them. You can also lock features of the report so they cannot be modified.

## Page Designer and Event Editor

The central portion of the Rave Designer window hosts the Page Designer (where you lay out the report) and the Event Editor (where you can provide scripts to customize the report at run time).

The Page Designer is the most noticeable aspect of Rave. This page is the foundation of a report, where you perform all the designing actions. The page displays a grid pattern, although you can change the look and feel of the page with the preference settings. The names of the current pages being designed appear in the tabs above the Page Designer (*Page1* in the figure).

The Event Editor allows you to define custom scripting code for the report components. Each component has several different types of events that can be used for calculations, string manipulation, or custom report logic. The Event Editor is an advanced feature of Rave; I'll cover it briefly at the end of this chapter.

## The Property Panel

The Property panel at left in the Rave Designer helps you customize the way components appear or behave. This panel has a role similar to that of Delphi's Object Inspector: When a component is selected on the page, the Property panel reflects the selection by displaying the different properties associated with that component. If no component is selected, the Property panel is blank.

Also as in the Delphi IDE, you can change a property value by editing the contents of the edit box, selecting an option from a drop-down list, or bringing up an editor dialog. You can double-click on any property that has a list of choices (instead of clicking on the down arrow button and selecting the option) to advance to the next item in the list.

## The Project Tree Panel

The Project Tree panel on the right side of the designer is very informative. It also provides an easy way to navigate a report project's structure. The Project Tree contains three main nodes: Report Library, Global Page Catalog, and Data View Dictionary:

**Report Library** Contains all the reports within the project. Each report has one or more pages. Each of those pages will normally include one or more components.

**Global Page Catalog** Manages reporting templates. The reporting templates can contain one or more components and can then be reused via Rave's mirroring technology. They can include items such as letter headings and footers, pre-printed forms, watermark designs, or complete page definitions that can be the foundation for other reports. You can think of the Global Page Catalog as a repository—a central location for you to store reporting items you want to be able to access from multiple reports.

**Data View Dictionary** Defines all the data views and other data-related objects for reports.

## Toolbars and the Toolbar Palette

As in Delphi, Rave includes two types of toolbars: component toolbars and IDE toolbars. However, unlike in Delphi (which uses the tabbed Component Palette exclusively for components), in Rave you can place either type of toolbar in the open area at the top of the designer or dock them in the tabbed Toolbar Palette. At first this process can be confusing, but after you arrange the toolbars to suit your needs (using the Dock and Undock shortcut menu commands, and not by dragging them as you'd expect) you'll probably find it quite flexible.

The default component toolbars in Rave are Standard, Drawing, Report, and Bar Code. I'll describe the component toolbars in more detail later in this chapter. For the moment, suffice to say that other component toolbars may be present if you've installed add-on packages in the Rave Designer, and that components can be rearranged in the

toolbars.

Editor toolbars have the ability to change or modify the project or existing components. Here is summary of the commands they host:

**Project Toolbar** Similar to a component toolbar, because it lets you create new report, page, and data object components in the report project. The Project toolbar can also be used to create new report projects, save and load existing report projects, and send the current report to the printer or preview.

**Alignment Toolbar** Has numerous tools for aligning and positioning components on a page. The first component selected often determines the benchmark for an alignment operation. You can change the printing order of components using the order buttons: Move Forward, Move Behind, Bring to Front, and Send to Back. The tap buttons let you move components in very small increments.

**Fonts Toolbar** Can be used to change font attributes such as Name, Size, Style, and Alignment.

**Fills Toolbar** Provides the fill style for shapes such as rectangles and circles.

**Lines Toolbar** Allows you to change the border widths and styles of lines and borders.

**Colors Toolbar** Determines primary and secondary colors (normally foreground and background, respectively). The left mouse button selects the primary color, and the right mouse button selects the secondary color. Eight custom color boxes are available for commonly used custom colors. Double-clicking on the primary or secondary color box at right on the toolbar opens the Color Editor dialog, which provides additional color selection tools.

**Zoom Toolbar** Provides many tools that allow the Page Designer to zoom in or out for easier editing.

**Designer Toolbar** Lets you customize the Page Designer and the Rave Designer through the preferences dialog.

## The Status Bar

At the bottom of the Rave Designer is the status bar. The status bar provides information about the direct data view connection status and the mouse position and sizing. The data connection LED's color tells you the status of the Rave data system (DirectDataViews): gray and green indicate an inactive or active connection, respectively; yellow and red indicate specific data access situations (respectively, waiting for response or a time-out).

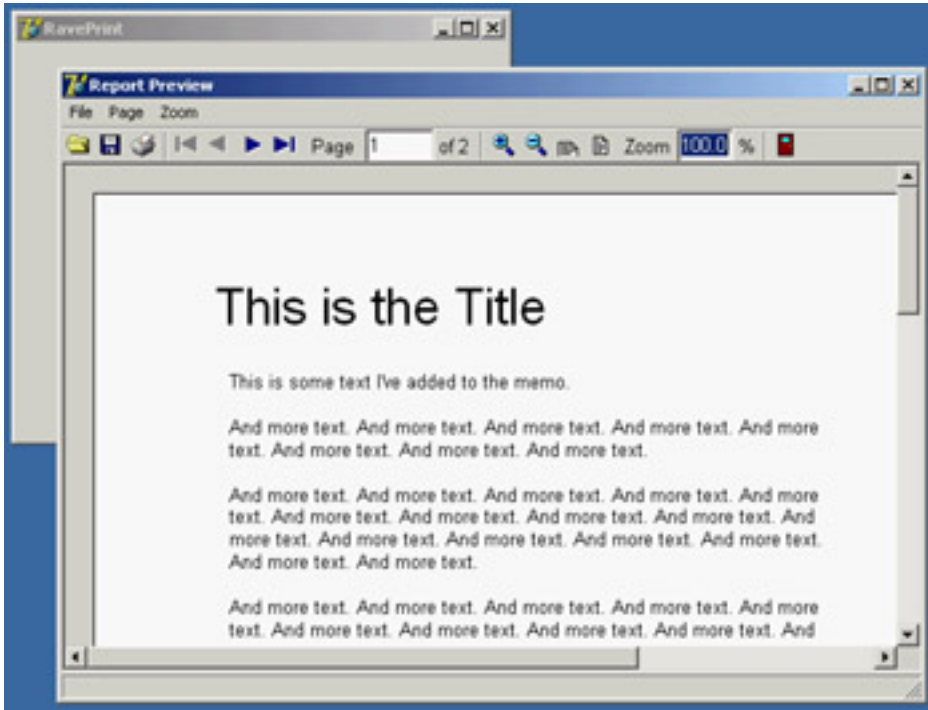
The X and Y values are the coordinates of the mouse pointer in page units. When you're dropping a component on the page, if you don't release the mouse button, the size of the component will be shown by the dX and dY values (*d* stands for *delta*).

## Using the RvProject Component

[Figure 18.1](#) shows a trivial Rave report, which is stored in a `.rav` file. To connect this report to your Delphi 7 application you use the Rave page, which provides a series of components. The central component is RvProject. Place this component on a form or data module, set its `ProjectFile` property to a Rave file, and write this event handler code for a button:

```
RvProject1.Execute;
```

You now have a working application (the RavePrint example among the source code files) that can print a report and that includes an embedded print preview feature, as you can see in [Figure 18.2](#). The referenced file should be distributed separately, and can be modified without having to change the Delphi program. As an alternative, you can also embed the `.rav` file into the Delphi executable by loading it in the DFM file. To accomplish this, use the `StoreRAV` property of the Rave project component.



**Figure 18.2:** The Rave Report Preview window for a report

**Note** As I mentioned earlier, the RvProject component, as well as all the components available in Rave Reports, can be used in either a VCL or a CLX application. The entire Rave engine is fully cross-platform.

To control the most important settings of the report and the preview, you can hook an RvNDRWriter or an RvSystem component to the `Engine` property of the RvProject component:

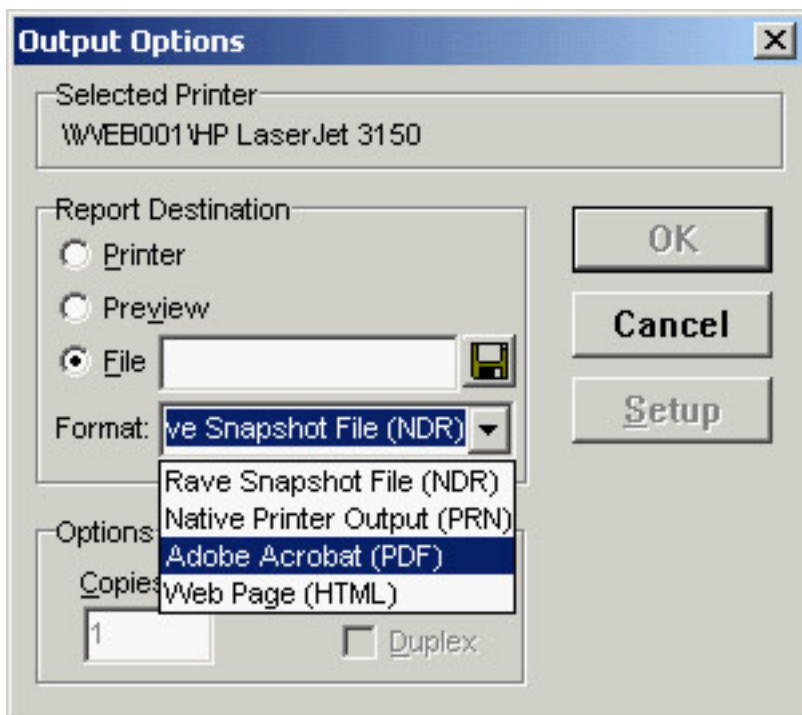
**RvNDRWriter Component** Generates an NDR formatted file when the report is executed. NDR formatted files are of a proprietary binary format and store all information needed by the rendering components to reproduce the report in a wide variety of formats.

**RvSystem Component** Combines the RvNDRWriter component with a standard print, preview, and rendering user interface. Many properties are available to customize the user interface, and override events can be defined to replace the standard dialogs with custom versions.

## Rendering Formats

The Rave engine produces an NDR file or stream, generated by the RvNDRWriter. Rave's rendering engine can convert this internal representation to a wide variety of formats. To let a user choose one of the target file formats, drop the desired render components within a form of your Delphi program. When running the RvProject component's `Execute` method (as in the RavePrint example), you can choose one of the file formats in the dialog box displayed by Rave and visible in [Figure 18.3](#).





**Figure 18.3:** After executing a Rave project, a user can choose the output format or rendering engine.

These are the available rendering components on the Rave page of Delphi's Component Palette:

**RvRenderPreview** Can be used to display an NDR file or stream to the screen. A ScrollBox component is used to display the report pages, and many methods and properties are available to create custom preview dialogs. Unless you need a custom print preview, you should use the RvSystem component for previewing instead of RvRenderPreview.

**RvRenderPrinter** Sends an NDR file or stream to the printer. Unless you need a custom printing or previewing interface, you do not need to use this component—RvSystem provides standard printing functionality.

**RvRenderPDF** Converts an NDR file or stream to PDF (Adobe Acrobat) form. You can use properties and events to customize the type of output that is created and to provide compression support.

**RvRenderHTML** Converts an NDR file or stream to HTML (DHTML) form. Each page is rendered to a separate page and can be combined with templates for greater output control.

**RvRenderRTF** Converts an NDR file or stream to RTF form. The RTF format uses field areas for accurate positioning and reproduction of the report.

**RvRenderText** Converts an NDR file or stream to text form. Many graphical commands and components, such as lines and rectangles, are ignored during text rendering.

In addition to letting a user choose a file format in the dialog, rendering can be done programmatically. For example, to convert a report to a PDF file directly, you can write the following code (taken again from the RavePrint example):

```

procedure TFormRave.btnPdfClick(Sender: TObject);
begin
  RvSystem1.DefaultDest := rdFile;
  RvSystem1.DoNativeOutput := False;
  RvSystem1.RenderObject := RvRenderPDF1;
  RvSystem1.OutputFileName := 'Simple2.pdf';
  RvSystem1.SystemSetups := RvSystem1.SystemSetups - [ssAllowSetup];
  RvProject1.Engine := RvSystem1;
  RvProject1.Execute;

```



end;

## Data Connections

Data connection components provide a link between data contained in a Delphi application and the DirectDataViews available in the Rave Designer. Notice that the value defined in the `Name` property for each data connection component is used to provide the link with the Rave report. For this reason, take care to avoid changing the component names after the DirectDataViews are created in Rave.

The data connection components are as follows:

**RvCustomConnection** Provides data to Rave reports using programmed events and can be used to send non-database data to a visual report.

**RvDataSetConnection** Connects any `TDataSet` class descendent component to a Rave DirectDataView. Using the `FieldAliasList` property, you can also modify the names of the dataset fields, replacing them with more legible names for the developers or end users building the report. If you need sorting or filtering for lookups or master-detail relationships within the Rave reports, you can handle the `OnSetSort` and `OnSetFilter` events.

**RvTableConnection and RvQueryConnection** Connect BDE Table and Query components to a Rave DirectDataView. Rave natively provides sorting and filtering support for table connections.

As a first example of how to construct a database-related report, I've created the RaveSingle example, which is an update of the DbxSingle program (built in [Chapter 14](#), "Client/ Server with dbExpress") with a Rave project and a connection:

```
object RvDataSetConnection1: TRvDataSetConnection
  RuntimeVisibility = rtDeveloper
  DataSet = SimpleDataSet1
end
object RvProject1: TRvProject
  ProjectFile = 'RaveSingle.rav'
end
```

In the Rave Designer, I created a new project, the file `RaveSingle.rav` in the `RaveSingle` folder. To refer to the data exposed by the Delphi program, you need to add a data view by clicking the New Data Object button on the Project toolbar, selecting the Direct Data View option (more details on the other alternatives later), and choosing an available connection. The list you'll see depends on the connections available in the project currently active in the Delphi IDE.

You can now create a report with the help of a wizard. From the Rave Designer menu, choose Tools → Report Wizards → Simple Table. Select the data view (you should have only one if you've followed these steps), and in the following window choose the dataset fields you want to include in your report. You should see a report like the one shown in [Figure 18.4](#).



**Figure 18.4:** The RaveSingle report (generated with the help of a wizard) at design time

As you can see in the Project Tree, the wizard generated a report page with a Region containing three Band components: one for the title, one for the table header, and a data-aware band for the items. I'll cover the details of using these components in the section "[Regions and Bands](#)," later in this chapter; for the moment, I wanted to build and let you experiment with a working example.

## Components of the Rave Designer

The [previous section](#) discussed the Rave components available in Delphi, because when you use this reporting engine you'll do most of your work in the report designer. The core element of a report (available at the top of the Project Tree view) is the RaveProject component, also called the Project Manager. This is not a component you drop on a report, because there is always only one object of this type in every report (as there is one global `Application` object in a Delphi program).

The RaveProject component is the owner of all the other report components. Like every component in Rave, the Project Manager has properties. To see them, select RaveProject in the Project Tree, and then look in the Property panel. To create a new RaveProject, click the New Project button on the Project toolbar. Doing so will create a new file, as well as the new Project Manager.

Every project can host multiple reports, represented by the Report component. A Report component contains the pages of a report. There can be multiple Reports in one project, and each Report can have multiple Pages. The Page component is the base visual component upon which you can place the report's visual components. You complete the designing and layout of a report here.

In addition to the list of reports available under the Report Library node, a Rave project has a Global Page Catalog, introduced earlier, and a Data View Dictionary. The Data View Dictionary is a detailed list of the data conduits made available by the hosting Delphi application (using the Delphi Rave connection components covered earlier) or activated directly from the report to a database.

You design the report by placing visual components directly on the page or on another container, such as a Band or a Region. Some of these components are not connected with database data—for example, the Text, Memo, and Bitmap components in the designer's Standard toolbar. Other components can be connected with a field in a database table (or are *data-aware*, to use a Delphi term), such as the DataText and DataMemo components on the Report toolbar.

### Basic Components

The Standard toolbar contains seven components: Text, Memo, Section, Bitmap, Metafile, FontMaster, and PageNumInit. Many of the standard components are used frequently when designing reports.

#### Text and Memo Components

The Text component is useful for displaying a single line of text on the report. It acts like a label that can contain simple text (not data). When placed on the report, the Text box is surrounded by a box that indicates its boundaries.

Memo components are similar to Text components, but they can contain multiple lines of text. When you set the Memo's font, all text in the Memo will have the same font, as in a Delphi Memo component. Once you've entered the text, the Memo box can be resized, and the text within the Memo box will be repositioned accordingly. If text seems to be missing from the Memo box after it has been entered in the Memo Editor, resize the box to allow all text to be visible.

#### The Section Component

The Section component is used to group components, like a Panel in Delphi. It provides advantages such as letting you move all components that form part of the Section with one mouse click, as opposed to making you move each component individually or try to select all components before moving.

The Project Tree is helpful when you're dealing with the Section component. From an expanded node, it is easy to see what components are in each Section, because compound components can form a tree with parent-child relationships.

**Tip** The Section component is important when you're mirroring, which allows visual design inheritance within your report designs.

### Graphical Components

The Bitmap and Metafile components let you place images in a report. Bitmap supports raster image files with the extension `.bmp`, and Metafile supports vector image files with the extensions `.wmf` and `.emf`. Metafiles are not supported when you use Rave in a CLX application, because they are based on a specific Windows technology.

## The FontMaster Component

Each Text component in a report has a `Font` property. By setting this property, you can assign a specific font to the component. In many cases, it may be useful and necessary to set the same font properties for more than one object. Although you can do so by selecting more than one component at a time, this method has a drawback: You have to keep track of which fonts must be the same typeface, size, and style, which is not an easy task for more than one report.

The FontMaster component lets you define standard fonts for different parts of the report, such as the headers, body, and footers. The FontMaster is a non-visual component (designated by the button's green color), so there will be no visual reference to it on the page (unlike in Delphi); and, like other non-visual components, it can only be accessed using the Project Tree.

Once you've set the FontMaster component's `Font` property, linking it to a body of text is simple. On the report, select a Text/Memo component, and then use the down arrow button by the `FontMirror` property in the Property panel to choose a FontMaster link. Any component whose `FontMirror` property is set to the FontMaster will be affected by and changed to the FontMaster's `Font` property.

When you set a component's `FontMirror` property, the component's `Font` property will be overridden by the FontMaster's `Font` property. Another side effect of using the FontMaster is that the Font toolbar is disabled when the `FontMirror` property is set for that component.

There can be more than one FontMaster per page; however, it is good practice to rename FontMaster components to describe their function. You should also locate them on a Global Page, so they can be used by all reports within the project and provide a more consistent typographical layout.

## Page Numbers

`PageNumInit` is a non-visual component that lets you restart page numbering within a Report. Its use is similar to that of other non-visual components. You use `PageNumInit` when more advanced formatting is required.

For example, consider a customer statement report for a checking account. The account statements customers receive every month may vary in the number of pages. Suppose the first page defines the account summary page layout, the second defines the customer's credits/deposits, and the third defines withdrawals and debits. The first two reports may need only one page; but if account activity is high for a customer, then the withdrawals section may be several pages long. If the user producing the report wants each section's pages numbered individually, the summary and deposits pages should both be marked "1 of 1". If an active customer account has three pages of withdrawals and debits, this section of the statement should be numbered "1 of 3", "2 of 3", and "3 of 3". `PageNumInit` comes in handy for this kind of page numbering.

## Drawing Components

Like the standard components, drawing components aren't related to data. Rave reports can include three components for lines: Generic lines are drawn in any direction and include angled lines; horizontal and vertical lines have a fixed direction. Available geometric shapes include squares, rectangles, circles, and ellipses. You can drop a shape on a report and then move it behind another element. For example, you can place a rectangle around a `DataBand`, set it so that it is sized to completely fill the band, and then move it behind the other components placed in the band.

## Bar Code Components

Bar code components are used to create many different kinds of bar codes in a report. Bar codes are for users who know exactly what they need, you must have background knowledge about bar codes and how they are used. To

define a bar code value, go to the Property panel and type the value in the `Text` property box.

Rave bar codes support includes the following:

- POSTNET (POSTal Numeric Encoding Technique) bar code, specifically used by the U.S. Postal Service
- I2of5BarCode (interleaved 2 of 5), for numeric information only
- Code39BarCode, an alphanumeric bar code that can encode decimal numbers, the uppercase alphabet, and a few special symbols
- Code128BarCode, a high-density alphanumeric bar code designed to encode all 128 ASCII characters
- UPCBarCode (Universal Product Code), which has a fixed length of 12 digits and was designed for coding products
- EANBarCode (European Article Numbering System), which is identical to the UPC but has 13 digits, 10 numeric characters, 2 country code characters, and a check digit

## Data Access Objects

In most cases, a report will be based on a data feed, either directly from a database or from a Delphi application. The data might come from a dataset connected to a database or from some internal processing of a Delphi program. When you choose the New Data Object button on the Project toolbar, you'll see the alternatives shown here and described in the following list:



**RaveDatabase Component (Database Connection)** Provides database connection parameters for the DriverDataView component. Only database connections for which DataLink drivers are installed will be allowed.

**RaveDirectDataView Component (Direct Data View)** Provides a means to retrieve data from a data connection component located within the hosting Delphi application, as in the last example. (The selection is called Direct Data View even if this is not the direct database connection from the report, but is instead the indirect connection based on data fetched from the database by a hosting application.)

**RaveDriverDataView Component (Driver Data View)** Provides a means to define a query for a specific database connection using a query language such as SQL. A query builder will be displayed to define the query.

**RaveSimpleSecurity Component (Simple Security Controller)** Implements the most basic form of security by using a simple list of username and password pairs in the `UserList` property. `UserList` contains one username and password pair per line in this format: `Username = password`. `CaseMatters` is a Boolean property that controls whether the password is case sensitive.

**RaveLookupSecurity Component (Data Lookup Security)** Allows username and password pairs to be checked against entries in a database table. The `DataView` property specifies the `DataView` to use to look up the username and password. The `UserField` and `PasswordField` properties are used to look up the username and the password to verify against.

## Regions and Bands

A Region component is a container for Band components. In its simplest form, the Region could be the whole Page component. This would be true for a report that is a list type. Many master-detail reports could be made to fit a single-Region design. However, do not be limited by thinking of a Region as being the entire Page; the properties for a Region deal with its size and location on the Page. Creative use of Regions will give you more flexibility when you're designing complex reports. Multiple Regions can be placed on a single Page; they can be side-by-side, one above the other, or staggered about the Page.

**Tip** Do not confuse a Region with a Section. Region components contain Bands and only Bands. A Section component can contain any group of components, including Region components, but cannot directly contain Bands.

When you're working with Bands, you must follow a simple rule: Bands must be in a Region. Notice that the number of Regions on a Page is not limited, nor is the number of Bands within a Region. As long as you can visualize the report mentally, you can use a combination of Regions and Bands to solve any difficulties faced when putting those thoughts into design. There are two band types:

**DataBand** Used to display iterating information from a `DataView`. In general, a `DataBand` will contain several `DataText` components. A `DataBand`'s `DataView` property must be set to the `DataView` component over which it will iterate and will typically contain other data-aware components that operate on that same `DataView`.

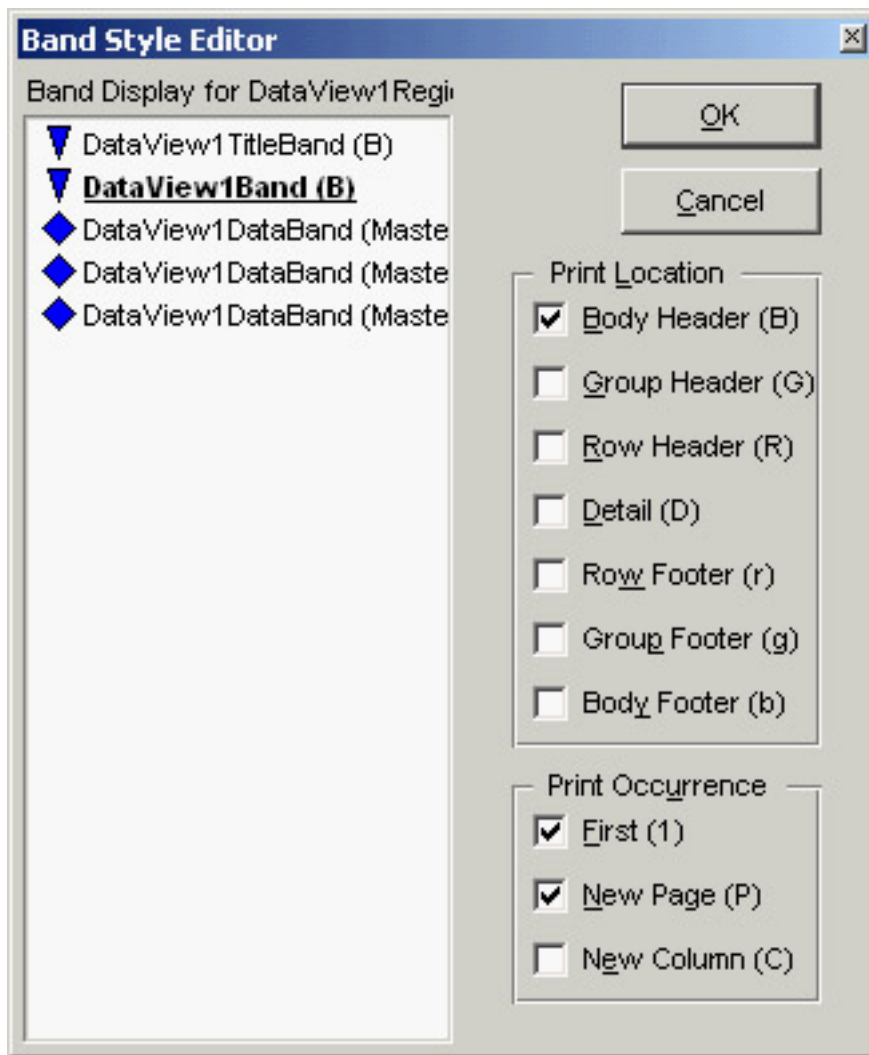
**Band** Used to print header and footer bands within a Region. Supported header and footer types include Body, Group, and Row; they are selected using the `BandStyle` property. Page headers and footers are not needed in a Band because you can drop them directly on the page outside the Region component.

`ControllingBand` is an important property for the Band component. This property determines which `DataBand` a Band belongs to (or is controlled by). When the controlling `DataBand` has been set, notice that the graphics symbol on the Band points in the direction of that controlling Band and that the color of the symbols matches. The letter codes shown on the Band are explained in the following section, "[The Band Style Editor](#)."

## The Band Style Editor

Go to the `BandStyle` property of a Band or `DataBand` component and click the ellipsis button to open the Band Style Editor:





This editor provides a simple method to select the features you want for a Band by using the check boxes. Note that a Band can have several different features active at a time. This means it is possible for a Band to be both a Body Header and Body Footer simultaneously.

The display area in the Band Style Editor is designed to represent the flow of a report in pseudo-layout style. DataBands are duplicated three times to show that they are repeated. The current Band that is being edited is highlighted: Although you see the other Bands in the editor, you can only modify the settings of the current one.

The Band Style Editor uses symbols and letters in the display area and in the Page Layout area (for example, see [Figure 18.6](#)) to inform you about each Band's behavior. The major difference between these two representations is that the Band Style Editor display arranges the bands in a pseudo flow according to the definition of each Band. In the Band Style Editor the Bands are arranged according to the logical flow and the order they are placed in the report at design time. The Bands sequence in the report's output is mainly controlled by this order.

Headers (capital letters *BGR*, which stand for Body, Group, and Row, respectively) will print first, followed by the DataBand, and then the footers (lower case letters *bgr*) for each level. However, if more than one header is defined for a particular level, then the header Bands are processed in the order they are arranged in the Region. So, it is possible to put all the headers at the top, all the DataBands in the middle, and all the footers at the bottom of a Region for all levels of a master-detail report. Alternatively, you can group each level, with the appropriate headers, footers, and DataBands together for each level. Rave lets you use the Region layout in a way that makes the most sense to the design flow. Just remember that the order of precedence for Bands at the same level is controlled by their order within the Region.

Two symbols show the parent/child or master/detail relationships of the various bands:

- The *triangle* symbol (up/down arrows) indicates that the band is controlled by a master band with the same color (level), which can be found in the direction of the arrow.

- The *diamond* symbol represents a master or controlling band.

These symbols are both color-coded and indented to represent the level of master-detail flow. Remember that you can have a master/detail/detail relationship in which both details are both controlled by the same master or one of the details is controlled by the other detail.

## Data-Aware Components

You can place different data-aware Rave components within a DataBand. The most common choice is the DataText component, used to display a text field from a dataset, as you saw in the RaveSingle example.

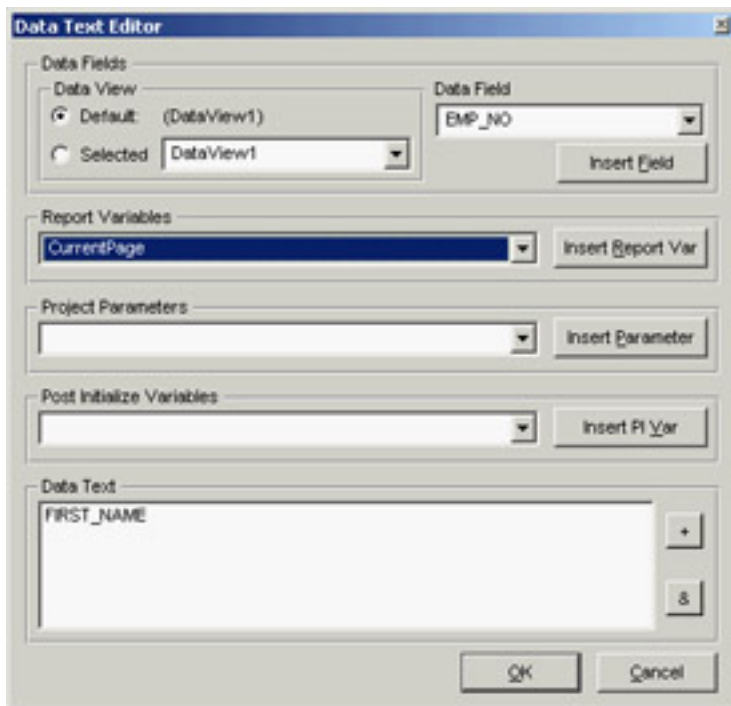
Two options are available for entering data in a DataField property. The first is to select a single field using the drop-down list; this approach is fine for normal database reporting where only a single data field for each DataText item may be needed. The second is to use the Data Text Editor.

### The Data Text Editor

Many reports require various fields to be combined. Two common examples are city, state, and zip code, or first name and last name combinations. In code, you can accomplish these combinations using statements like the following:

```
City + ', ' + State + ' ' + Zip
FirstName & LastName
```

The DataField property has a Data Text Editor, shown in [Figure 18.5](#), which assists you in building composite fields. Click on the ellipsis to open the DataText Editor; it gives you the power to concatenate fields, parameters, or variables to build a complex data-aware text field by selecting items from the list boxes. This editor includes a lot of combinations; I will cover them quickly here, and you can try them in practice.



**Figure 18.5:** The Rave Designer's Data Text Editor

Note that the dialog box is divided into five groups: Data Fields, Report Variables, Project Parameters, Post Initialize Variables, and Data Text. Data Text is the result window: Watch this window when you're inserting items. The two buttons on the right side of this window are plus (+) and ampersand (&). The + button adds two items together with no spaces, and the & button concatenates them with a single space (as long as the previous field was not blank). So, the first step is to decide whether you want to use + or &, and then select the text from one of the three groups above the



Data Text window.

The DataText component is not limited to printing database data: You can also use project parameters and report variables. Go to the Report Variables group, look in the list box, note the variables that are available. The Project Parameters list could contain `UserName`, `ReportTitle`, or `UserOption` parameters initialized by the application. To create the list of project parameters, select the Project node in the Project Tree (the top item). Then, in the Properties panel, click the ellipsis button by the Parameters property to open typical string editor, where you can enter the different parameters you want to pass to Rave from the application (such as the `UserName`).

## From Text to Memo

The DataMemo component displays a memo field from a DataView. The main difference between the DataMemo component and the DataText component is that the DataMemo component is used to print text that may require more than one line and will thus need to be wrapped. For example, it could be used to print remarks about a customer at the bottom of each Page of an invoice.

One use for the DataMemo component is mail-merge functions. The easiest way to accomplish this is to set the `DataView` and `DataField` properties to the source of the Memo field. Then, launch the Mail Merge Editor by clicking the ellipsis button by the `MailMergeItems` property. This editor lets you set the items in the Memo that will be changed.

To use the Mail Merge Editor, click the Add button. In the Search Token window, type the item that is in the Memo and that will be replaced. Then, either type the replacement string in the Replacement window or click the Edit button to start the Data Text Editor that will help you select different DataViews and fields.

## Calculating Totals

The CalcText component is a data-aware totaling component. The main difference between the DataText component and the CalcText component is that the CalcText component is specially designed to perform calculations and display the results.

The `CalcType` property determines the type of calculation being performed; values include Average, Count, Maximum, Minimum, and Sum. For example, you can use this component to print the totals of an invoice at the top of each page.

The `CountBlanks` property determines whether blank field values are included in the Average and Count calculation methods. If `RunningTotal` is True, then the calculation will not be reset to 0 each time it is printed.

## Cycling on Data inside Pages

The DataCycle component is basically an invisible DataBand that adds data-iterating capabilities directly to a Page. This component is useful for form-style reports where using Regions and Band components would be cumbersome. It is important to make sure the DataCycle component appears before any components on the Page that process information from the same DataView.

## Advanced Rave

In this long introduction to Rave, you've seen that this reporting system is so complex I could devote an entire book to it. I've already built a couple of examples, and I could continue by showing a master/detail relationship or other reports with a complex structure. However, with the wizards available and the information provided so far, you should be able to build similar examples yourself. So, in this section I'll build only one example of this type, and then provide information about a few relevant features of Rave that aren't easy to understand by trial and error.

**Note** Among other features of Rave not discussed here is one that allows you to distribute the report designer to end users (to let them customize the reports), either bound to a Delphi program or as a stand-alone tool. Rave also has a server version that allows you to surface reports from a web server.

**Tip** You can learn more about these features and other aspects of Rave on Nevrona's website. Specifically, you should browse the tip collection located at [www.nevrona.com/rave/tips.shtml](http://www.nevrona.com/rave/tips.shtml).

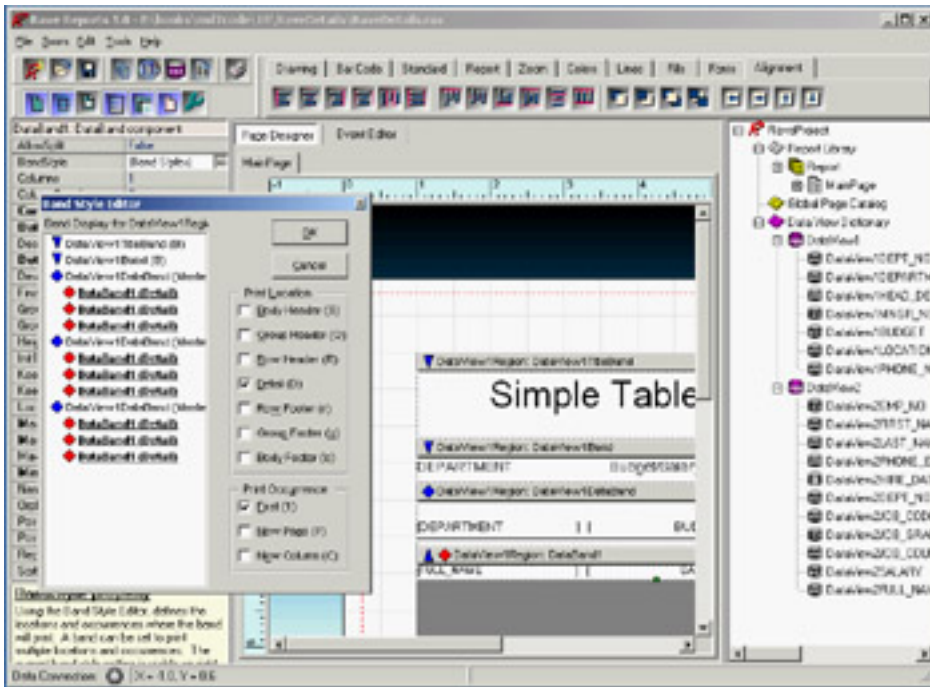
## Master/Detail Reports

To create a master/detail report in Rave, you need two datasets in the corresponding Delphi application, but these datasets don't need to have a master/detail relationship defined in the program—the report can define such a relationship. In the RaveDetails demo program, each of the two datasets is exposed through a Rave connection:

```
object dsDepartments: TSimpleDataSet
  Connection = SQLConnection1
  DataSet.CommandText = 'select * from DEPARTMENT'
end
object dsEmployee: TSimpleDataSet
  Connection = SQLConnection1
  DataSet.CommandText = 'select * from EMPLOYEE'
end
object RvConnectionDepartments: TRvDataSetConnection
  DataSet = dsDepartments
end
object RvConnectionEmployee: TRvDataSetConnection
  DataSet = dsEmployee
end
```

The report has two corresponding data views, each connected to a DataBand component (both hosted within a Region). The first DataBand, used for the main dataset, has no special settings. The secondary DataBand defines the master/detail relationship using a few properties. The `MasterDataView` property refers to the data view of the master dataset, and the `MasterKey` and `DetailKey` properties refer to the fields that define the join (in this case, they both refer to the field `DEPT_NO`). The `ControllerBand` property refers to the DataBand showing the data of the master dataset.

In case master/details report, the most important settings are managed by the Band Style Editor, where you must set the band as a detail. You can see this editor for the RaveDetails example in [Figure 18.6](#). The master data view's `KeepRowTogether` property is set to `True`, to avoid having details end up in a different page than the master.



**Figure 18.6:** The master/detail report. The Band Style Editor appears in front of it.

**Warning** To create a master/detail report, you may want to use the corresponding wizard available in Rave. In the version shipping with Delphi 7, this wizard doesn't work. At the time of this writing an update to fix this and other issues is still not available.

## Scripting Reports

At the beginning of this chapter I mentioned the Event Editor window of the Rave Designer, but so far I haven't used it. This tool is used to write code (scripts) within a report, responding to events of the various components as you'd do in Delphi. Writing scripts in the Rave Designer allows you to customize, or fine-tune, the output of a report in sophisticated ways. The language Rave uses for the scripts is Pascal based and a variation of the Delphi language, so you should be able to pick it up easily.

The RaveDetails example shows in bold font salaries that are greater than a specified amount. The obvious way to accomplish this is to write scripting code to be executed for each instance of the detail band—that is, for each record in the employee database. Instead of modifying the `Font` property directly, I decided to add two different `FontManager` components to the report page and name them to make their role understandable: `fmPlainFont` and `fmBoldFont`. You can open the report to see their properties and layout.

In the report, to highlight values outside a given range, you handle the `DataText` component's `BeforePrint` event. To do so, move to the Event Editor page, choose the `DataText` component connected to the Salary field, and select the event. In the event's code editor window, write the following code:

```
if DataView2Salary.AsFloat > 100000 then
  self.FontMirror := fmBoldFont;
else
  self.FontMirror := fmPlainFont;
end if;
```

The script changes the `FontMirror` property of the current object (`self`) to refer to one of the two `FontManager` components on the page, depending on the field value. Notice that `DataView2Salary` is a reference to one of the fields of the data view—the one connected to the current `DataText` component. Compile the script and run the report to see its effect, shown in [Figure 18.7](#).

DEPARTMENT	Budget/Salary	LOCATION
Corporate Headquarters	1,000,000.00	Monterey
Lee, Terri	53,793.00	
Bender, Oliver H.	<b>212,850.00</b>	
Sales and Marketing	2,000,000.00	San Francisco
MacDonald, Mary S.	<b>111,262.59</b>	
Yanowski, Michael	44,000.00	
Engineering	1,100,000.00	Monterey
Nelson, Robert	<b>105,960.00</b>	
Brown, Kelly	27,000.00	

**Figure 18.7:** The bold text in the report is determined at run time by a script.

**Warning** Every time you edit a script, remember to click the Compile button; otherwise your changes won't take effect!

## Mirroring

Reporting templates can contain one or more components and can be reused via Rave's mirroring technology. The DataMirrorSection component mirrors other Sections based on the contents of a DataField. The use of mirroring sections allows the DataMirrorSection to be very flexible. Remember that Sections can contain any other component including graphics, regions, text, and so on.

For example, you could use a DataMirrorSection component to make a single report produce different envelope formats for international or U.S. addresses. The template for international customers could include a line for the country have the text centered on the envelope, whereas the U.S. format would not include the country line would offset the text to the right of center and lower on the envelope.

Normally, one of the settings will be defined as the default. If a default is *not* defined and the field value does not match any of the other settings, then the format used will be the normal contents of the DataMirrorSection component.

## Calculations to the Max

In addition to the simple CalcText component discussed earlier, the Rave Designer includes three components for handling more complex situations: CalcTotal, CalcController, and CalcOp.

### CalcTotal

The CalcTotal is a non-visual version of the CalcText component. When this component is printed, its value is typically stored in a project parameter (defined by the DestParam property) and formatted according to the DisplayFormat property. It can be useful when you're performing totaling calculations that will be used by other calculations before being printed. If the value of the CalcTotal will only be used by other calculation components, such as CalcOp, you should leave the DestParam property blank.

### CalcController

CalcController is a non-visual component that acts as a controller for CalcText and CalcTotal components through their Controller properties. When the controller component is printed, it signals all calculation components that it controls to perform their operation. This process allows a report to recalculate totals on group bands, detail bands, or whole

pages depending upon the location of the CalcController component.

The CalcController component can also initialize a CalcText or CalcTotal component to a specific value (through the `InitCalcVar`, `InitDataField`, and `InitValue` properties). The CalcController component will initialize values only if it is used in the `Initializer` property of CalcText or CalcTotal components.

## CalcOp

CalcOp is a non-visual component that allows an operation (defined by the `Operator` property) to be performed on values from different data sources. The result can then be saved in a project parameter like CalcTotal, as indicated by the `DestParam` and `DisplayFormat` properties.

For example, suppose you need to add two DataText components, as in  $A + B = C$  (where A and B represent the two DataText component values and C represents the result stored in a project parameter). The three source types have many different values associated with them.

The calculation can start with different types of data sources:

- A DataField source is a field in a table, or a DataView in Rave terms. Thus in order to choose a field, you must first select a DataView.
- For a Value source, you fill in the property with a numeric value.
- A CalcVar source represents another calculation variable; you can choose one from the drop-down menu that lists all the calculation variables available in the page. This value can be from another CalcOp component or from some other calculation component.

After choosing the data sources, you select the operation to be used between them. The `Operator` property has a drop-down menu you can use to make the appropriate choice. In the example  $A + B = C$ , the operator is `coAdd`.

Sometimes a function needs to be performed on a value before it is processed with the second value. In this case the source's `Function` property is handy. With a function, you can converted a value (such as hours to minutes), compute a trigonometric function (like the sin of the value), or perform many others calculations (such as a square root or absolute value).

Just as it is important to do the calculations in order, it is important to make sure the components are in order in the Project Tree. A report executes components down the Project Tree. For CalcOp components or any calculation component, this means they must be in the correct order. It is also important to note that if a source value is dependent on any other components (like other CalcOp components or DataText components), those components must come first in the Project Tree.

## What's Next?

This chapter has focused using Rave to add reporting capabilities to Delphi applications. I've covered basic reports and data-driven reports, describing both the components you can use in a Delphi application to connect with a report and the components you can use in the report to display data passed by the application. I also touched on some of the more advanced features of Rave, like the use of the Band Style Editor, scripting, mirroring, calculations, and a few others.

If you are looking for more documentation, refer to the PDF files that are available with Delphi but not installed with the product. These manuals and many others are available on the Delphi 7 Companion CD.

This chapter ends the book's coverage of Delphi database technologies, although you'll also use databases in the following part of the book (devoted to Internet and web programming). I'll begin by covering core technologies, such as the use of sockets, and then proceed to HTML generation, traditional Delphi web development, and the use of IntraWeb. I'll then cover the XML and SOAP technologies.

## Part IV: Delphi, the Internet, and a .NET Preview

### Chapter List

[Chapter 19](#): Internet Programming: Sockets and Indy

[Chapter 20](#): Web Programming with WebBroker and WebSnap

[Chapter 21](#): Web Programming with IntraWeb

[Chapter 22](#): Using XML Technologies

[Chapter 23](#): Web Services and SOAP

[Chapter 24](#): The Microsoft .NET Architecture from the Delphi Perspective

[Chapter 25](#): Delphi for .NET Preview: The Language and the RTL

## Chapter 19: Internet Programming: Sockets and Indy

### Overview

With the advent of the Internet era, writing programs based on Internet protocols has become commonplace, so I've devoted five chapters to this topic. This chapter focuses on low-level socket programming and Internet protocols, [Chapter 20](#) is devoted to server-side web programming, [Chapter 21](#) covers IntraWeb, and [Chapters 22](#) and [23](#) discuss XML and web services.

In this chapter I'll begin by looking at the sockets technology in general; then I'll move to the use of the Internet Direct (Indy) components supporting both low-level socket programming and the most common Internet protocols. I will introduce some elements of the HTTP protocol, leading up to building HTML files from database data.

Although you probably just want to use a high-level protocol, the discussion of Internet programming starts from the core concepts and low-level applications. Understanding TCP/ IP and sockets will help you grasp most of the other concepts more easily.



## Building Socket Applications

Delphi 7 ships with two sets of TCP components—Indy socket components (IdTCPClient and IdTCPServer) and native Borland components—which are also available in Kylix and are hosted in the Internet page of the Component palette. The Borland components, TcpClient and TcpServer, were probably developed to replace the ClientSocket and ServerSocket components available in past versions of Delphi. However, now that the ClientSocket and ServerSocket components have been declared obsolete (although they are still available), Borland suggests using the corresponding Indy components instead.

In this chapter I'll focus on using Indy during my discussion of low-level socket programming, not only when I cover support for high-level Internet protocols. To learn more about the Indy project, refer to the sidebar "[Internet Direct \(Indy\) Open Source Components](#)"; keep reading to see how you can use these components for low-level socket programming.

Before I present an example of a low-level socket-based communication, let's take a tour of the core concepts of TCP/IP so you understand the foundations of the most widespread networking technology.

---

### Internet Direct (Indy) Open Source Components

Delphi ships with a collection of open-source Internet components called Internet Direct (Indy). The Indy components, previously called WinShoes (a pun on the term WinSock, the name of the Windows socket library), are built by a group of developers led by Chad Hower and are also available in Kylix. You can find more information and the most recent versions of the components at [www.nevrona.com/indy](http://www.nevrona.com/indy).

Delphi 7 ships with Indy 9, but you should check the website for updated versions. The components are free and are complemented by many examples and a reasonable help file. Indy 9 includes many more components than the previous version (Indy 8, available in Delphi 6), and it has two new pages on the component palette (Indy Intercepts and Indy I/O Handlers).

With more than 100 components installed on Delphi's palette, Indy has an enormous number of features, ranging from the development of client and server TCP/IP applications for various protocols to encoding and security. You can recognize Indy components from the Id prefix. Rather than list the various components here, I'll touch on a few of them throughout this chapter.

#### ***Blocking and Non-Blocking Connections***

When you're working with sockets in Windows, reading data from a socket or writing to it can happen asynchronously, so that it does not block the execution of other code in your network application. This is called a *non-blocking connection*. The Windows socket support sends a message when data is available. An alternative approach is the use of *blocking connections*, where your application waits for the reading or writing to be completed before executing the next line of code. When you're using a blocking connection, you must use a thread on the server, and you'll generally also use a thread on the client.

The Indy components use blocking connections exclusively. So, any client socket operation that might be lengthy should be performed within a thread or by using Indy's IdAntiFreeze component as a simpler but limited alternative. Using blocking connections to implement a protocol has the advantage of simplifying the program logic, because you don't have to use the state-machine approach of non-blocking connections.

All the Indy servers use a multithreaded architecture that you can control with the IdThreadMgrDefault and IdThreadMgrPool components. The first is used by default; the second supports thread pooling and should account for faster connections.

---

To understand the behavior of the socket components, you need to be confident with several terms related to the Internet in general and with sockets in particular. The heart of the Internet is the Transmission Control Protocol/Internet Protocol (TCP/IP), a combination of two separate protocols that work together to provide connections over the Internet (and that can also provide connection over a private intranet). In brief, IP is responsible for defining and routing the *datagrams* (Internet transmission units) and specifying the addressing scheme. TCP is responsible for higher-level transport services.

## Configuring a Local Network: IP Addresses

If you have a local network available, you'll be able to test the following programs on it; otherwise, you can use the same computer as both client and server. In this case, as I've done in the examples, use the address 127.0.0.1 (or *localhost*), which is invariably the address of the current computer. If your network is complex, ask your network administrator to set up proper IP addresses for you. If you want to set up a simple network with a couple of spare computers, you can set up the IP address yourself; it's a 32-bit number usually represented with each of its four components (called *octets*) separated by dots. These numbers have a complex logic underlying them, and the first octet indicates the class of the address.

Specific IP addresses are reserved for unregistered internal networks. Internet routers ignore these address ranges, so you can freely do your tests without interfering with an actual network. For example, the "free" IP address range 192.168.0.0 through 192.168.0.255 can be used for experiments on a network of fewer than 255 machines.

## Local Domain Names

How does the IP address map to a name? On the Internet, the client program looks up the values on a domain name server. But it is also possible to have a local *hosts* file, which is a text file you can easily edit to provide local mappings. Look at the `HOSTS.SAM` file (installed in a subdirectory of the `Windows` directory, depending on the version of Windows you have) to see a sample; you can eventually rename the file `HOSTS`, without the extension, to activate local host mapping.

You may wonder whether to use an IP or a hostname in your programs. Hostnames are easier to remember and won't require a change if the IP address changes (for whatever reason). On the other hand, IP addresses don't require any resolution, whereas hostnames must be resolved (a time-consuming operation if the lookup takes place on the web).

## TCP Ports

Each TCP connection takes place through a *port*, which is represented by a 16-bit number. The IP address and the TCP port together specify an Internet connection, or a *socket*. Different processes running on the same machine cannot use the same socket (the same port).

Some TCP ports have a standard usage for specific high-level protocols and services. In other words, you should use those port numbers when implementing those services and stay away from them in any other case. Here is a short list:

Protocol	Port
HTTP (Hypertext Transfer Protocol)	80
FTP (File Transfer Protocol)	21
SMTP (Simple Mail Transfer Protocol)	25
POP3 (Post Office Protocol, version 3)	110
Telnet	23

The `Services` file (another text file similar to the `Hosts` file) lists the standard ports used by services. You can add your own entry to the list, giving your service a name of your own choosing. Client sockets always specify the port number or the service name of the server socket to which they want to connect.

## High-Level Protocols

I've used the term *protocol* many times now. A protocol is a set of rules the client and server agree on to determine the communication flow. The low-level Internet protocols, such as TCP/IP, are usually implemented by an operating system. But the term *protocol* is also used for high-level Internet standard protocols (such as HTTP, FTP, or SMTP). These protocols are defined in standard documents available on the Internet Engineering Task Force website ([www.ietf.org](http://www.ietf.org)).

If you want to implement a custom communication, you can define your own (possibly simple) protocol, a set of rules determining which request the client can send to the server and how the server can respond to the various possible requests. You'll see an example of a custom protocol later. Transfer protocols are at a higher level than transmission protocols, because they abstract from the transport mechanism provided by TCP/IP. This makes the protocols independent not only from the operating system and the hardware but also from the physical network.

## Socket Connections

To begin communication through a socket, the server program starts running first; but it simply waits for a request from a client. The client program requests a connection indicating the server it wishes to connect to. When the client sends the request, the server can accept the connection, starting a specific server-side socket, which connects to the client-side socket.

To support this model, there are three types of socket connections:

- *Client connections* are initiated by the client and connect a local client socket with a remote server socket. Client sockets must describe the server they want to connect to, by providing its hostname (or its IP address) and its port.
- *Listening connections* are passive server sockets waiting for a client. Once a client makes a new request, the server spawns a new socket devoted to that specific connection and then gets back to listening. Listening server sockets must indicate the port that represents the service they provide. (The client will connect through that port.)
- *Server connections* are activated by servers; they accept a request from a client.

These different types of connections are important only for establishing the link from the client to the server. Once the link is established, both sides are free to make requests and to send data to the other side.

## Using Indy's TCP Components

To let two programs communicate over a socket (either on a local area network or over the Internet), you can use the `IdTCPClient` and `IdTCPServer` components. Place one of them on a program form and the other on another form in a different program; then, make them use the same port, and let the client program refer to the host of the server program, and you'll be able to open a connection between the two applications. For example, in the `IndySock1` project group, I've used the two components with these settings:

```
// server program
object IdTCPServer1: TIdTCPServer
  DefaultPort = 1050
end
// client program
object IdTCPClient1: TIdTCPClient
  Host = 'localhost'
  Port = 1050
end
```

**Note** The Indy server sockets allow binding to multiple IP addresses and/or ports, using the *Bindings* collection.

As this point, in the client program you can connect to the server by executing

```
IdTCPClient1.Connect;
```

The server program has a list box used to log information. When a client connects or disconnects, the program lists the

IP of that client along with the operation, as in the following `OnConnect` event handler:

```
procedure TFormServer.IdTCPServer1Connect(AThread: TIdPeerThread);  
begin  
  lbLog.Items.Add ('Connected from: ' +  
    AThread.Connection.Socket.Binding.PeerIP);  
end;
```

Now that you have set up a connection, you need to make the two programs communicate. Both the client and server sockets have read and write methods you can use to send data, but writing a multithreaded server that can receive many different commands (usually based on strings) and operate differently on each of them is far from trivial.

However, Indy simplifies the development of a server by means of its command architecture. In a server, you can define a number of commands, which are stored in the `CommandHandlers` collection of the `IdTCPServer`. In the `IndySock1` example the server has three handlers, all implemented differently to show you some of the possible alternatives.

The first server command, called `test`, is the simplest one, because it is fully defined in its properties. I've set the command string, a numeric code, and a string result in the `ReplyNormal` property of the command handler:

```
object IdTCPServer1: TIdTCPServer  
  CommandHandlers = <  
    item  
      Command = 'test'  
      Name = 'TIdCommandHandler0'  
      ParseParams = False  
      ReplyNormal.NumericCode = 100  
      ReplyNormal.Text.Strings = (  
        'Hello from your Indy Server')  
      ReplyNormal.TextCode = '100'  
    end
```

The client code used to execute the command and show its response is as follows:

```
procedure TFormClient.btnTestClick(Sender: TObject);  
begin  
  IdTCPClient1.SendCmd ('test');  
  ShowMessage (IdTCPClient1.LastCmdResult.TextCode + ' : ' +  
    IdTCPClient1.LastCmdResult.Text.Text);  
end;
```

For more complex cases, you should execute code on the server and read and write directly over the socket connection. This approach is shown in the second command of the trivial protocol I've come up with for this example. The server's second command is called `execute`; and it has no special property set (only the command name), but has the following `OnCommand` event handler:

```
procedure TFormServer.IdTCPServer1TIdCommandHandler1Command(  
  ASender: TIdCommand);  
begin  
  ASender.Thread.Connection.WriteLine ('This is a dynamic response');  
end;
```

The corresponding client code writes the command name to the socket connection and then reads a single-line response, using different methods than the first one:

```
procedure TFormClient.btnExecuteClick(Sender: TObject);  
begin  
  IdTCPClient1.WriteLine('execute');  
  ShowMessage (IdTCPClient1.ReadLn);  
end;
```

```
end;
```

The effect is similar to the previous example, but because it uses a lower-level approach, it should be easier to customize it to your needs. One such extension is provided by the third and last command in the example, which allows the client program to request a bitmap file from the server (in a sort of file-sharing architecture). The server command has parameters (the filename) and is defined as follows:

```
object IdTCPServer1: TIdTCPServer
  CommandHandlers = <
    item
      CmdDelimiter = ' '
      Command = 'getfile'
      Name = 'TIdCommandHandler2'
      OnCommand = IdTCPServer1TIdCommandHandler2Command
      ParamDelimiter = ' '
      ReplyExceptionCode = 0
      ReplyNormal.NumericCode = 0
      Tag = 0
    end>
```

The code uses the first parameter as filename and returns it in a stream. In case of error, it raises an exception, which will be intercepted by the server component, which in turn will terminate the connection (not a very realistic solution, but a safe approach and a simple one to implement):

```
procedure TFormServer.IdTCPServer1TIdCommandHandler2Command(
  ASender: TIdCommand);
var
  filename: string;
  fstream: TFileStream;
begin
  if Assigned (ASender.Params) then
    filename := HttpDecode (ASender.Params [0]);
  if not FileExists (filename) then
    begin
      ASender.Response.Text := 'File not found';
      lbLog.Items.Add ('File not found: ' + filename);
      raise EIdTCPServerError.Create ('File not found: ' + filename);
    end
  else
    begin
      fstream := TFileStream.Create (filename, fmOpenRead);
      try
        ASender.Thread.Connection.WriteStream(fstream, True, True);
        lbLog.Items.Add ('File returned: ' + filename +
          ' (' + IntToStr (fStream.Size) + ')');
      finally
        fstream.Free;
      end;
    end;
  end;
end;
```

The call to the `HttpDecode` utility function on the parameter is required to encode a pathname that includes spaces as a single parameter, at the reverse the client program calls `HttpEncode`. As you can see, the server also logs the files returned and their sizes, or an error message. The client program reads the stream and copies it into an `Image` component, to show it directly (see [Figure 19.1](#)):

```
procedure TFormClient.btnGetFileClick(Sender: TObject);
var
```

```

stream: TStream;
begin
  IdTCPClient1.WriteLine('getfile ' + HttpEncode (edFileName.Text));
  stream := TMemoryStream.Create;
  try
    IdTCPClient1.ReadStream(stream);
    stream.Position := 0;
    Image1.Picture.Bitmap.LoadFromStream (stream);
  finally
    stream.Free;
  end;
end;

```

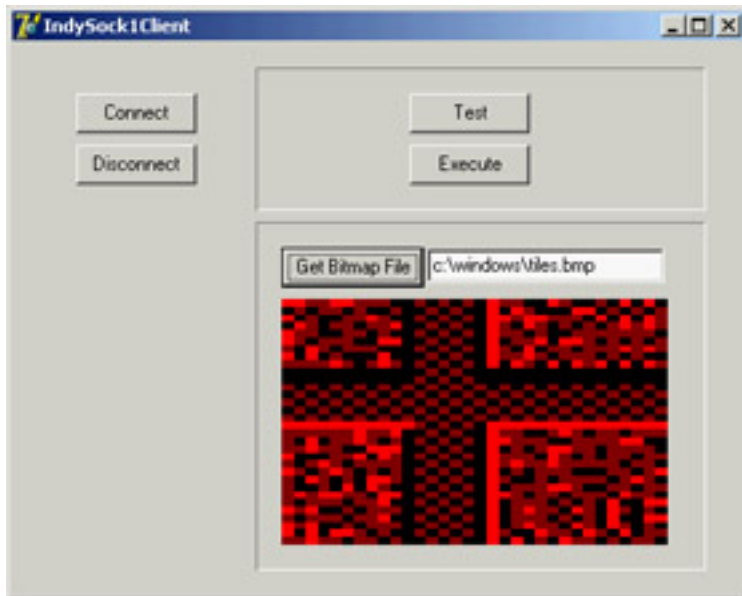


Figure 19.1: The client program of the IndySock1 example

## Sending Database Data over a Socket Connection

Using the techniques you've seen so far, you can write an application that moves database records over a socket. The idea is to write a front end for data input and a back end for data storage. The client application will have a simple data-entry form and use a database table with string fields for Company, Address, State, Country, Email, and Contact, and a floating-point field for the company ID (called CompID).

**Note** Moving database records over a socket is exactly what you can do with DataSnap and a socket connection component (as covered in [Chapter 16](#), "Multitier DataSnap Applications") or with SOAP support (discussed in [Chapter 23](#), "Web Services and SOAP").

The client program I've come up with works on a ClientDataSet with this structure saved in the current directory. (You can see the related code in the `OnCreate` event handler.) The core method on the client side is the handler of the Send All button's `OnClick` event, which sends all the new records to the server. A new record is determined by looking to see whether the record has a valid value for the CompID field. This field is not set up by the user but is determined by the server application when the data is sent.

For all the new records, the client program packages the field information in a string list, using the structure *FieldName=FieldValue*. The string corresponding to the entire list, which is a record, is then sent to the server. At this point, the program waits for the server to send back the company ID, which is then saved in the current record. All this code takes place within a thread, to avoid blocking the user interface during the lengthy operation. By clicking the Send button, a user starts a new thread:

```

procedure TForm1.btnSendClick(Sender: TObject);

```

```

var
    SendThread: TSendThread;
begin
    SendThread := TSendThread.Create(cds);
    SendThread.OnLog := OnLog;
    SendThread.ServerAddress := EditServer.Text;
    SendThread.Resume;
end;

```

The thread has a few parameters: the dataset passed in the constructor, the address of the server saved in the `ServerAddress` property, and a logging event to write to the main form (within a safe `Synchronize` call). The thread code creates and opens a connection and keeps sending records until it's finished:

```

procedure TSendThread.Execute;
var
    I: Integer;
    Data: TStringList;
    Buf: String;
begin
    try
        Data := TStringList.Create;
        fIdTcpClient := TIdTcpClient.Create (nil);
        try
            fIdTcpClient.Host := ServerAddress;
            fIdTcpClient.Port := 1051;
            fIdTcpClient.Connect;
            fDataSet.First;
            while not fDataSet.Eof do
                begin
                    // if the record is still not logged
                    if fDataSet.FieldName('CompID').IsNull or
                        (fDataSet.FieldName('CompID').AsInteger = 0) then
                        begin
                            FLogMsg := 'Sending ' + fDataSet.FieldName('Company').AsString;
                            Synchronize(DoLog);
                            Data.Clear;
                            // create strings with structure "FieldName=Value"
                            for I := 0 to fDataSet.FieldCount - 1 do
                                Data.Values [fDataSet.Fields[I].FieldName] :=
                                    fDataSet.Fields [I].AsString;
                            // send the record
                            fIdTcpClient.Writeln ('senddata');
                            fIdTcpClient.WriteString (Data, True);
                            // wait for reponse
                            Buf := fIdTcpClient.ReadLn;
                            fDataSet.Edit;
                            fDataSet.FieldName('CompID').AsString := Buf;
                            fDataSet.Post;
                            FLogMsg := fDataSet.FieldName('Company').AsString +
                                ' logged as ' + fDataSet.FieldName('CompID').AsString;
                            Synchronize(DoLog);
                        end;
                    fDataSet.Next;
                end;
            finally
                fIdTcpClient.Disconnect;
                fIdTcpClient.Free;
            end;

```

```

    Data.Free;
end;
except
    // trap exceptions in case of dataset errors
    // (concurrent editing and so on)
end;
end;
end;

```

Now let's look at the server. This program has a database table, again stored in the local directory, with two more fields than the client application's table: `LoggedBy`, a string field; and `LoggedOn`, a data field. The values of the two extra fields are determined automatically by the server as it receives data, along with the value of the `CompID` field. All these operations are done in the handler of the `senddata` command:

```

procedure TForm1.IdTCPServer1TIdCommandHandler0Command(
    ASender: TIdCommand);
var
    Data: TStrings;
    I: Integer;
begin
    Data := TStringList.Create;
    try
        ASender.Thread.Connection.ReadStrings(Data);
        cds.Insert;
        // set the fields using the strings
        for I := 0 to cds.FieldCount - 1 do
            cds.Fields [I].AsString :=
                Data.Values [cds.Fields[I].FieldName];
        // complete with ID, sender, and date
        Inc(ID);
        cdsCompID.AsInteger := ID;
        cdsLoggedBy.AsString := ASender.Thread.Connection.Socket.Binding.PeerIP;
        cdsLoggedOn.AsDateTime := Date;
        cds.Post;
        // return the ID
        ASender.Thread.Connection.WriteLine(cdsCompID.AsString);
    finally
        Data.Free;
    end;
end;

```

Except for the fact that some data might be lost, there is no problem when fields have a different order and if they do not match, because the data is stored in the *FieldName=FieldValue* structure. After receiving all the data and posting it to the local table, the server sends back the company ID to the client. When receiving feedback, the client program saves the company ID, which marks the record as sent. If the user modifies the record, there is no way to send an update to the server. To accomplish this, you might add a modified field to the client database table and make the server check to see if it is receiving a new field or a modified field. With a modified field, the server should not add a new record but update the existing one.

As shown in [Figure 19.2](#), the server program has two pages: one with a log and the other with a DBGrid showing the current data in the server database table. The client program is a form-based data entry, with extra buttons to send the data and delete records already sent (and for which an ID was received back).



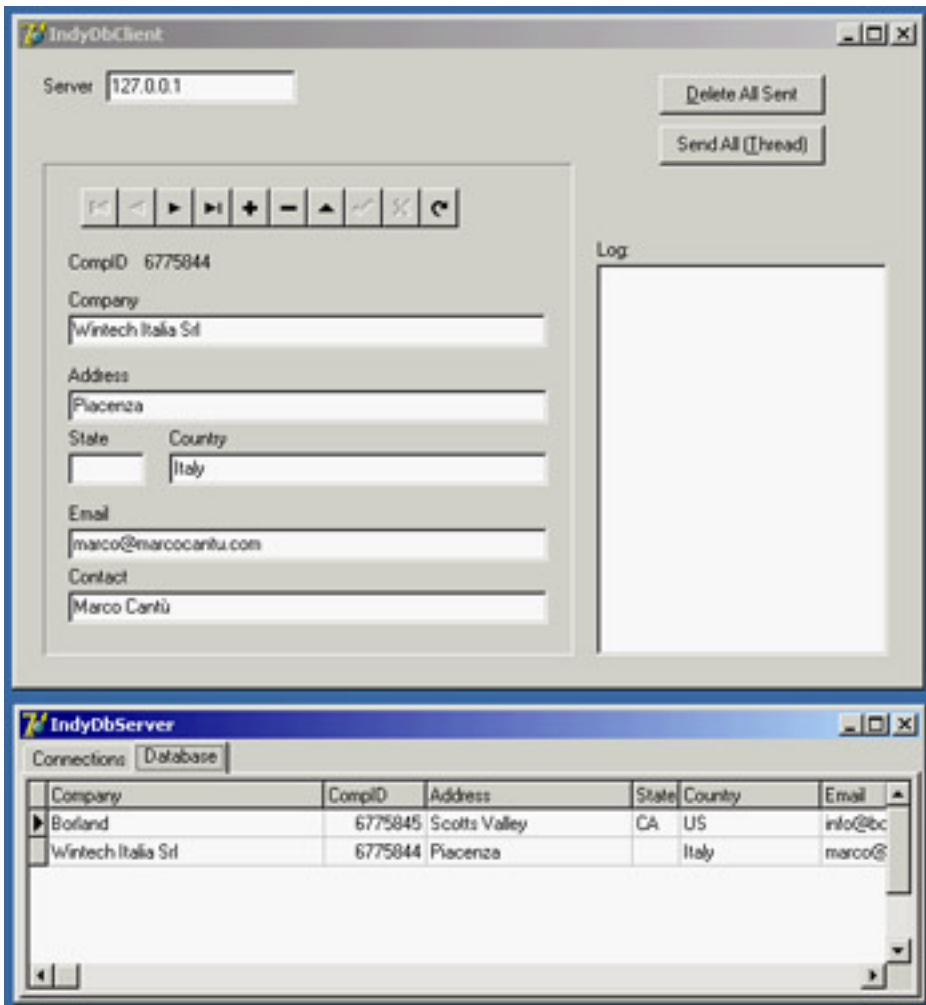


Figure 19.2: The client and server programs of the data-base socket example (IndyDbSock)

## Sending and Receiving Mail

Probably the most common operation you do on the Internet is to send and receive e-mail. There is generally little need to write a complete application to handle e-mail, because some of the existing programs are rather complete. For this reason, I have no intention of writing a general-purpose mail program here. You can find some examples of those among Indy demos. Other than creating a general-purpose mail application, you can do many things with the mail components and protocols; I've grouped the possibilities into two areas:

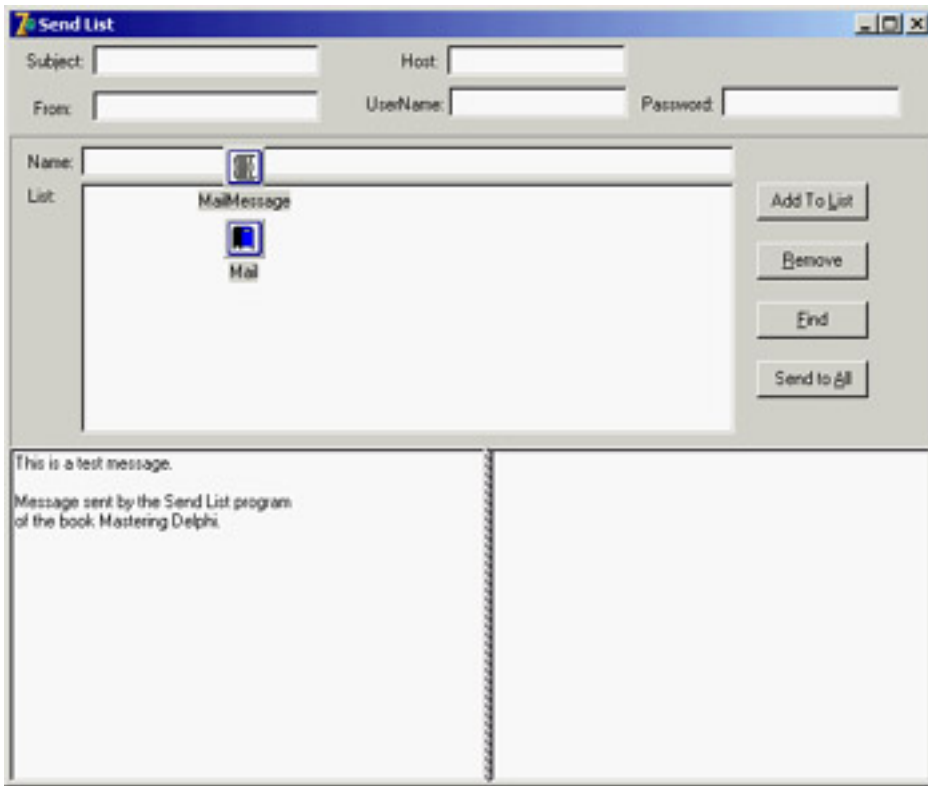
**Automatic Generation of Mail Messages** An application you've written can have an About box for sending a registration message back to your marketing department or a specific menu item for sending a request to your tech support. You might even decide to enable a tech-support connection whenever an exception occurs. Another related task could automate the dispatching of a message to a list of people or generate an automatic message from your website (an example I'll show you toward the end of this chapter).

**Use of Mail Protocols for Communication with Users Who Are Only Occasionally Online** When you must move data between users who are not always online, you can write an application on a server to synchronize among them, and you can give each user a specialized client application for interacting with the server. An alternative is to use an existing server application, such as a mail server, and write the two specialized programs based on the mail protocols. The data sent over this connection will generally be formatted in special ways, so you'll want to use a specific e-mail address for these messages (not your primary e-mail address). As an example, you could rewrite the earlier IndyDbSock example to dispatch mail messages instead of using a custom socket connection. This approach has the advantage of being firewall-friendly and allowing the server to be temporarily offline, because the requests will be kept on the mail server.

## Mail In and Out

Using the mail protocols with Indy means placing a message component (`IdMessage`) in your application, filling it with data, and then using the `IdSMTP` component to send the mail message. To *retrieve* a mail message from your mailbox, use the `IdPop3` component, which will return an `IdMessage` object. To give you an idea how this process works, I've written a program for sending mail to multiple people at once, using a list stored in an ASCII file. I originally used this program to send mail to people who sign up on my website, but later I extended the program by adding database support and the ability to read subscriber logs automatically. The original version of the program is still a good introduction to the use of the Indy SMTP component.

The `SendList` program keeps a list of names and e-mail addresses in a local file, which is displayed in a list box. A few buttons allow you to add and remove items, or to modify an item by removing it, editing it, and then adding it again. When the program closes, the updated list is automatically saved. Now let's get to the interesting portion of the program. The top panel, shown in [Figure 19.3](#) at design time, allows you to enter the subject, the sender address, and the information used to connect to the mail server (hostname, username, and eventually a password).



**Figure 19.3:** The SendList program at design time

You'll probably want to make the value of these edit boxes persistent, possibly in an INI file. I haven't done this, only because I don't want you to see my mail connection details! The value of these edit boxes, along with the list of addresses, allows you to send the series of mail messages (after customizing them) with the following code:

```

procedure TMainForm.BtnSendAllClick(Sender: TObject);
var
    nItem: Integer;
    Res: Word;
begin
    Res := MessageDlg ('Start sending from item ' +
        IntToStr (ListAddr.ItemIndex) + ' (' +
        ListAddr.Items [ListAddr.ItemIndex] + ')?'#13 +
        '(No starts from 0)', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    if Res = mrCancel then
        Exit;
    if Res = mrYes then
        nItem := ListAddr.ItemIndex
    else
        nItem := 0;
    // connect
    Mail.Host := eServer.Text;
    Mail.UserName := eUserName.Text;
    if ePassword.Text <> '' then
        begin
            Mail.Password := ePassword.Text;
            Mail.AuthenticationType := atLogin;
        end;
    Mail.Connect;
    // send the messages, one by one, prepending a custom message
    try
        // set the fixed part of the header
        MailMessage.From.Name := eFrom.Text;
        MailMessage.Subject := eSubject.Text;

```

```
MailMessage.Body.SetText (reMessageText.Lines.GetText);
MailMessage.Body.Insert (0, 'Hello');
while nItem < ListAddr.Items.Count do
begin
    // show the current selection
    Application.ProcessMessages;
    ListAddr.ItemIndex := nItem;
    MailMessage.Body [0] := 'Hello ' + ListAddr.Items [nItem];
    MailMessage.Recipients.EmailAddresses := ListAddr.Items [nItem];
    Mail.Send(MailMessage);
    Inc (nItem);
end;
finally // we're done
    Mail.Disconnect;
end;
end;
```

Another interesting example of using mail is to notify developers of problems within applications (a technique you might want to use in an internal application rather than in one you'll distribute widely). You can obtain this effect by modifying the [ErrorLog](#) example from [Chapter 2](#) and sending mail when an exception (or one of a given type only) occurs.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Working with HTTP

Handling mail messages is certainly interesting, and mail protocols are probably still the most widespread Internet protocols. The other popular protocol is HTTP, which is used by web servers and web browsers. I'll devote the rest of this chapter to this protocol (along with a discussion of HTML); the following two chapters also discuss it.

On the client side of the Web, the main activity is browsing—reading HTML files. Besides building a custom browser, you can embed the Internet Explorer ActiveX control within your program (as I've done in the WebDemo example in [Chapter 12](#), "From COM to COM+"). You can also directly activate the browser installed on the user's computer—for example, opening an HTML page by calling the `ShellExecute` method (defined in the `ShellApi` unit):

```
ShellExecute(Handle, 'open', FileName, '', '', sw_ShowNormal);
```

Using `ShellExecute`, you can simply execute a document, such as a file. Windows will start the program associated with the `HTM` extension, using the action passed as the parameter (in this case, `open`, but passing `nil` would have invoked the standard action producing the same effect). You can use a similar call to view a website, by using a string like `'http://www.example.com'` instead of a filename. In this case, the system recognizes the `http` section of the request as requiring a web browser and launches it.

On the server side, you generate and make available the HTML pages. At times, it may be enough to have a way to produce static pages, occasionally extracting new data from a database table to update the HTML files as needed. In other cases, you'll need to generate pages dynamically based on a request from a user.

As a starting point, I'll discuss HTTP by building a simple but complete client and server; then I'll move on to discuss HTML producer components. In [Chapter 20](#), I'll move from this "core technology" level to the RAD development style for the web supported by Delphi, introducing the web server extension technologies (CGI, ISAPI, and Apache modules) and discussing the WebBroker and WebSnap architectures.

## Grabbing HTTP Content

As an example of the use of the HTTP protocols, I've written a specific search application. The program hooks onto the Google website, searches for a keyword, and retrieves the first 100 sites found. Instead of showing the resulting HTML file, the program parses it to extract only the URLs of the related sites to a list box. The description of these sites is kept in a separate string list and is displayed as you click a list-box item. So, the program demonstrates two techniques at once: retrieving a web page and parsing its HTML code.

To demonstrate how you should work with blocking connections, such as those used by Indy, I've implemented the program using a background thread for the processing. This approach also gives you the advantage of being able to start multiple searches at once. The thread class used by the WebFind application receives as input a URL to look for, `strUrl`.

The class has two output procedures, `AddToList` and `ShowStatus`, to be called inside the `Synchronize` method. The code of these two methods sends some results or some feedback to the main form, respectively adding a line to the list box and changing the status bar's `SimpleText` property. The key method of the thread is `Execute`. Before we look at it, however, here is how the thread is activated by the main form:

```
const
  strSearch = 'http://www.google.com/search?as_q=';

procedure TForm1.BtnFindClick(Sender: TObject);
var
  FindThread: TFindWebThread;
begin
  // create suspended, set initial values, and start
  FindThread := TFindWebThread.Create(True);
  FindThread.FreeOnTerminate := True;
```

```

// grab the first 100 entries
FindThread.strUrl := strSearch + EditSearch.Text + '&num=100';
FindThread.Resume;
end;

```

The URL string is made of the main address of the search engine, followed by some parameters. The first, `as_q`, indicates the words you are looking for. The second, `num=100`, indicates the number of sites to retrieve; you cannot use numbers at will but are limited to a few alternatives, with 100 being the largest possible value.

**Warning** The WebFind program works with the server on the Google website at the time this book was written and tested. The custom software on the site can change, however, which might prevent WebFind from operating correctly. This program was also in *Mastering Delphi 6*; however, it was missing the user agent HTTP header, and after a while Google changed its server software and blocked the requests. Adding any value for the user agent fixed the problem.

The thread's `Execute` method, activated by the `Resume` call, calls the two methods doing the work (shown in [Listing 19.1](#)). In the first, `GrabHtml`, the program connects to the HTTP server using a dynamically created `IdHttp` component and reads the HTML with the result of the search. The second method, `HtmlToList`, extracts the URLs referring to other websites from the result, the `strRead` string.

### Listing 19.1: The TFindWebThread Class (of the WebFind Program)

---

```

unit FindTh;

interface

uses
  Classes, IdComponent, SysUtils, IdHTTP;

type
  TFindWebThread = class(TThread)
  protected
    Addr, Text, Status: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (Sender: TObject; AWorkMode: TWorkMode;
      const AWorkCount: Integer);
  public
    strUrl: string;
    strRead: string;
  end;

implementation

{ TFindWebThread }

uses
  WebFindF;

procedure TFindWebThread.AddToList;
begin
  if Form1.ListBox1.Items.IndexOf (Addr) < 0 then
  begin
    Form1.ListBox1.Items.Add (Addr);
  end;
end;

```

```

    Form1.DetailsList.Add (Text);
end;
end;

procedure TFindWebThread.Execute;
begin
    GrabHtml;
    HtmlToList;
    Status := 'Done with ' + StrUrl;
    Synchronize (ShowStatus);
end;

procedure TFindWebThread.GrabHtml;
var
    Http1: TIdHTTP;
begin
    Status := 'Sending query: ' + StrUrl;
    Synchronize (ShowStatus);
    Http1 := TIdHTTP.Create (nil);
    try
        Http1.Request.UserAgent := 'User-Agent: NULL';
        Http1.OnWork := HttpWork;
        strRead := Http1.Get (StrUrl);
    finally
        Http1.Free;
    end;
end;

procedure TFindWebThread.HtmlToList;
var
    strAddr, strText: string;
    nText: integer;
    nBegin, nEnd: Integer;
begin
    Status := 'Extracting data for: ' + StrUrl;
    Synchronize (ShowStatus);
    strRead := LowerCase (strRead);
    repeat
        // find the initial part HTTP reference
        nBegin := Pos ('href=http', strRead);
        if nBegin <> 0 then
            begin
                // get the remaining part of the string, starting with 'http'
                strRead := Copy (strRead, nBegin + 5, 1000000);
                // find the end of the HTTP reference
                nEnd := Pos ('>', strRead);
                strAddr := Copy (strRead, 1, nEnd - 1);
                // move on
                strRead := Copy (strRead, nEnd + 1, 1000000);
                // add the URL if 'google' is not in it
                if Pos ('google', strAddr) = 0 then
                    begin
                        nText := Pos ('</a>', strRead);
                        strText := copy (strRead, 1, nText - 1);
                        // remove cached references and duplicates
                        if (Pos ('cached', strText) = 0) then

```

```

begin
  Addr := strAddr;
  Text := strText;
  AddToList;
end;
end;
end;
until nBegin = 0;
end;

procedure TFindWebThread.HttpWork(Sender: TObject; AWorkMode: TWorkMode;
  const AWorkCount: Integer);
begin
  Status := 'Received ' + IntToStr (AWorkCount) + ' for ' + strUrl;
  Synchronize (ShowStatus);
end;

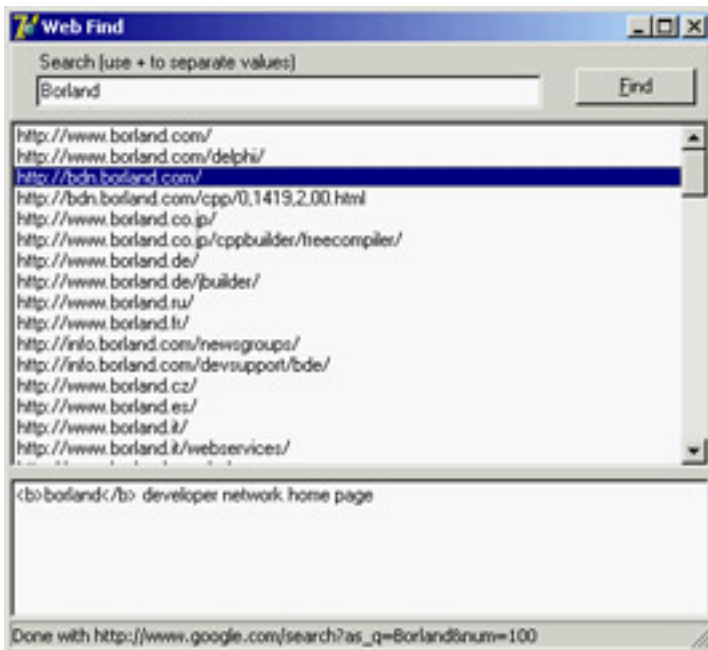
procedure TFindWebThread.ShowStatus;
begin
  Form1.StatusBar1.SimpleText := Status;
end;

end.

```

---

The program looks for subsequent occurrences of the href=http substring, copying the text up to the closing > character. If the found string contains the word *google*, or its target text includes the word *cached*, it is omitted from the result. You can see the effect of this code in the output shown in [Figure 19.4](#). You can start multiple searches at the same time, but be aware that the results will be added to the same memo component.



**Figure 19.4:** The WebFind application can be used to search for a list of sites on the Google search engine.

## The WinInet API

When you need to use the FTP and HTTP protocols, as alternatives to using particular VCL components, you can use a specific API provided by Microsoft in the WinInet DLL. This library is part of the core operating system and implements the FTP and HTTP protocols on top of the Windows sockets API.



With just three calls—`InternetOpen`, `InternetOpenURL`, and `InternetReadFile`—you can retrieve a file corresponding to any URL and store a local copy or analyze it. Other simple methods can be used for FTP; I suggest you look for the source code of the `WinInet.pas` Delphi unit, which lists all the functions.

**Tip** The help file of the WinInet library is not part of the SDK Help shipping with Delphi, but can be found online on MSDN at [msdn.microsoft.com/library/en-us/wininet/wininet/wininet\\_reference.asp](http://msdn.microsoft.com/library/en-us/wininet/wininet/wininet_reference.asp).

The `InternetOpen` function establishes a generic connection and returns a handle you can use in the `InternetOpenURL` call. This second call returns a handle to the URL that you can pass to the `InternetReadFile` function in order to read blocks of data. In the following sample code, the data is stored in a local string. When all the data has been read, the program closes the connection to the URL and the Internet session by calling the `InternetCloseHandle` function twice:

```
var
  hHttpSession, hReqUrl: HInternet;
  Buffer: array [0..1023] of Char;
  nRead: Cardinal;
  strRead: string;
  nBegin, nEnd: Integer;
begin
  strRead := '';
  hHttpSession := InternetOpen ('FindWeb', INTERNET_OPEN_TYPE_PRECONFIG,
    nil, nil, 0);
  try
    hReqUrl := InternetOpenURL (hHttpSession, PChar(StrUrl), nil, 0,0,0);
    try // read all the data
      repeat
        InternetReadFile (hReqUrl, @Buffer, sizeof (Buffer), nRead);
        strRead := strRead + string (Buffer);
      until nRead = 0;
    finally
      InternetCloseHandle (hReqUrl);
    end;
  finally
    InternetCloseHandle (hHttpSession);
  end;
end;
```

## Browsing on Your Own

Although I doubt you are interested in writing a new web browser, it might be interesting to see how you can grab an HTML file from the Internet and display it locally, using the HTML viewer available in CLX (the `TextBrowser` control). Connecting this control to an Indy HTTP client, you can quickly come up with a simplistic text-only browser with limited navigation. The core is

```
TextBrowser1.Text := IdHttp1.Get (NewUrl);
```

where `NewUrl` is complete location of the web resource you want to access. In the `BrowseFast` example, this URL is entered in a combo box, which keeps track of recent requests. The effect of a similar call is to return the textual portion of a web page (see [Figure 19.5](#)), because grabbing the graphic content requires much more complex coding. The `TextBrowser` control really is better defined as a local file viewer than as a browser.



Figure 19.5: The output of the BrowseFast text-only browser

I've added to the program only very limited support for hyperlinks. When a user moves the mouse over a link, its link text is copied to a local variable (`NewRequest`), which is then used in case of a click on the control to compute the new HTTP request to forward. Merging the current address (`LastUrl`) with the request, though, is far from trivial, even with the help of the `IdUrl` class provided by Indy. Here is my code, which handles only the simplest cases:

```

procedure TForm1.TextBrowser1Click(Sender: TObject);
var
    Uri: TIdUri;
begin
    if NewRequest <> '' then
    begin
        Uri := TIdUri.Create (LastUrl);
        if Pos ('http:', NewRequest) > 0 then
            GoToUrl (NewRequest)
        else if NewRequest [1] = '/' then
            GoToUrl ('http://' + Uri.Host + NewRequest)
        else
            GoToUrl ('http://' + Uri.Host + Uri.Path + NewRequest);
    end;
end;

```

Again, this example is trivial and far from usable, but building a browser involves little more than the ability to connect via HTTP and display HTML files.

## A Simple HTTP Server

The situation with the development of an HTTP server is quite different. Building a server to deliver static pages based on HTML files is far from simple, although one of the Indy demos provides a good starting point. However, a custom HTTP server might be interesting when building a totally dynamic site, something I'll focus on in more detail in [Chapter 20](#).

To show you how to begin the development of a custom HTTP server, I've built the `HttpServ` example. This program has a form with a list box used for logging requests and an `IdHTTPServer` component with these settings:

```

object IdHTTPServer1: TIdHTTPServer
  Active = True
  DefaultPort = 8080
  OnCommandGet = IdHTTPServer1CommandGet
end

```

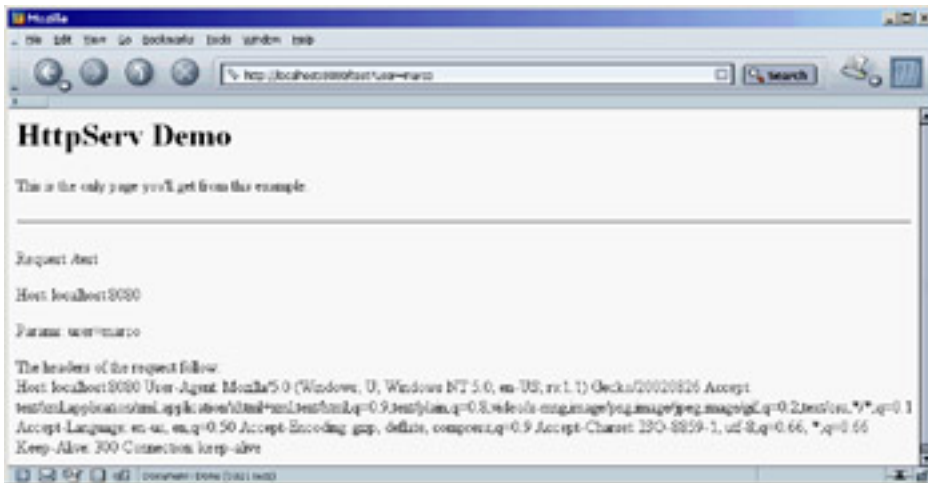
The server uses port 8080 instead of the standard port 80, so that you can run it alongside another web server. All the custom code is in the `OnCommandGet` event handler, which returns a fixed page plus some information about the request:

```

procedure TForm1.IdHTTPServer1CommandGet(AThread: TIdPeerThread;
  RequestInfo: TIdHTTPRequestInfo; ResponseInfo: TIdHTTPResponseInfo);
var
  HtmlResult: String;
begin
  // log
  Listbox1.Items.Add (RequestInfo.Document);
  // respond
  HtmlResult := '<h1>HttpServ Demo</h1>' +
    '<p>This is the only page you''ll get from this example.</p><hr>' +
    '<p>Request: ' + RequestInfo.Document + '</p>' +
    '<p>Host: ' + RequestInfo.Host + '</p>' +
    '<p>Params: ' + RequestInfo.UnparsedParams + '</p>' +
    '<p>The headers of the request follow: <br>' +
    RequestInfo.RawHeaders.Text + '</p>';
  ResponseInfo.ContentText := HtmlResult;
end;

```

By passing a path and some parameters in the command line of the browser, you'll see them reinterpreted and displayed. For example, [Figure 19.6](#) shows the effect of this command line:



**Figure 19.6:** The page displayed by connecting a browser to the custom `HttpServ` program

`http://localhost:8080/test?user=marco`

If this example seems too trivial, you'll see a slightly more interesting version in the [next section](#), where I discuss the generation of HTML with Delphi's producer components.

**Note** If you plan to build an advanced web server or other Internet servers with Delphi, then as an alternative to the Indy components, you should look at the `DXSock` components from Brain Patchwork DX ([www.dxsock.com](http://www.dxsock.com)).

## Generating HTML

The Hypertext Markup Language, better known by its acronym HTML, is the most widespread format for content on the Web. HTML is the format web browsers typically read; it is a standard defined by the W3C (World Wide Web Consortium, [www.w3.org](http://www.w3.org)), which is one of the bodies controlling the Internet. The HTML standard document is available on [www.w3.org/MarkUp](http://www.w3.org/MarkUp) along with and some interesting links.

### Delphi's HTML Producer Components

Delphi's HTML producer components (on the Internet page of the Component Palette) can be used to generate the HTML files and particularly to turn a database table into an HTML table. Many developers believe that the use of these components makes sense only when writing a web server extension. Although they were introduced for this purpose and are part of the WebBroker technology, you can still use three out of the five producer components in any application in which you must generate a static HTML file.

Before looking at the HtmlProd example, which demonstrates the use of these HTML producer components, let me summarize their role:

- The simplest HTML producer component is the PageProducer, which manipulates an HTML file in which you've embedded special tags. The HTML can be stored in an external file or an internal string list. The advantage of this approach is that you can generate such a file using the HTML editor you prefer. At run time, the PageProducer converts the special tags to HTML code, giving you a straightforward method for modifying sections of an HTML document. The special tags have the basic format `<#tagname>`, but you can also supply named parameters within the tag. You'll process the tags in the `OnTag` event handler of the PageProducer.
- The DataSetPageProducer extends the PageProducer by automatically replacing tags corresponding to field names of a connected data source.
- The DataSetTableProducer component is generally useful for displaying the contents of a table, query, or other dataset. The idea is to produce an HTML table from a dataset, in a simple yet flexible way. The component has a nice preview, so you can see how the HTML output will look in a browser directly at design time.
- The QueryTableProducer and the SQLQueryTableProducer components are similar to the DataSetTableProducer, but they are specifically tailored for building parametric queries (for the BDE or dbExpress, respectively) based on input from an HTML search form. This component makes little sense in a stand-alone program, and for this reason, I'll delay covering these components until [Chapter 20](#).

### Producing HTML Pages

A very simple example of using tags (introduced by the # symbol) is creating an HTML file that displays fields with the current date or a date computed relative to the current date, such as an expiration date. If you examine the HtmlProd example, you'll find a PageProducer1 component with internal HTML code, specified by the `HTMLDOC` string list:

```
<html>
<head>
<title>Producer Demo</title>
</head>
<body>
<h1>Producer Demo</h1>
<p>This is a demo of the page produced by the <b><#appname></b> application on
<b><#date></b>.</p>
<hr>
<p>The prices in this catalog are valid until <b><#expiration
days=21></b>.</p>
</body>
```

</html>

**Warning** If you prepare this file with an HTML editor (something I suggest you do), it may automatically place quotes around tag parameters, as in `days="21"`, because this format is required by HTML 4 and XHTML. The PageProducer component has a `StripParamQuotes` property you can activate to remove those extra quotes when the component parses the code (before calling the `OnHTMLTag` event handler).

The Demo Page button copies the PageProducer component's output to the `Text` of a Memo. As you call the `Content` function of the PageProducer component, it reads the input HTML code, parses it, and triggers the `OnTag` event handler for every special tag. In the handler for this event, the program checks the value of the tag (passed in the `TagString` parameter) and returns a different HTML text (in the `ReplaceText` reference parameter), producing the output shown in [Figure 19.7](#).



**Figure 19.7:** The output of the HtmlProd example, a simple demonstration of the Page-Producer component, when the user clicks the Demo Page button

```
procedure TFormProd.PageProducer1HTMLTag(Sender: TObject;
Tag: TTag; const TagString: String; TagParams: TStrings;
var ReplaceText: String);
var
nDays: Integer;
begin
if TagString = 'date' then
    ReplaceText := DateToStr (Now)
else if TagString = 'appname' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
else if TagString = 'expiration' then
begin
    nDays := StrToIntDef (TagParams.Values['days'], 0);
if nDays <> 0 then
        ReplaceText := DateToStr (Now + nDays)
    else
        ReplaceText := '<i>{expiration tag error}</i>';
    end;
end;
```

Notice in particular the code I've written to convert the last tag, `#expiration`, which requires a parameter. The PageProducer places the entire text of the tag parameter (in this case, `days=21`) in a string that's part of the `TagParams` list. To extract the value portion of this string (the portion after the equal sign), you can use the `Values` property of the `TagParams` string list and search for the proper entry at the same time. If it can't locate the parameter or if the parameter's value isn't an integer, the program displays an error message.

**Tip** The PageProducer component supports user-defined tags, which can be any string you like, but you should first review the special tags defined by the `TTags` enumeration. The possible values include `tgLink` (for the `link` tag), `tgImage` (for the `img` tag), `tgTable` (for the `table` tag), and a few others. If you create a custom tag, as in the PageProd example, the value of the `Tag` parameter to the HTMLTag handler will be `tgCustom`.

## Producing Pages of Data

The HtmlProd example also has a `DataSetPageProducer` component, which is connected with a database table and with the following HTML source code:

```
<html><head><title>Data for <#name></title></head>
<body>
<h1><center>Data for <#name></center></h1>
<p>Capital: <#capital></p>
<p>Continent: <#continent></p>
<p>Area: <#area></p>
<p>Population: <#population></p><hr>
<p>Last updated on <#date><br>
HTML file produced by the program <#program>.</p>
</body></html>
```

By using tags with the names of the connected dataset's fields (the usual `COUNTRY.DB` database table), the program automatically gets the value of the current record's fields and replaces them automatically. This produces the output shown in [Figure 19.8](#); the browser is connected to the HtmlProd example working as an HTTP server, as I'll discuss later. In the source code of the program related to this component, there is no reference to the database data:

```
procedure TFormProd.DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'program' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'date' then
    ReplaceText := DateToStr (Date);
end;
```



**Figure 19.8:** The output of the HtmlProd example for the Print Line button

## Producing HTML Tables

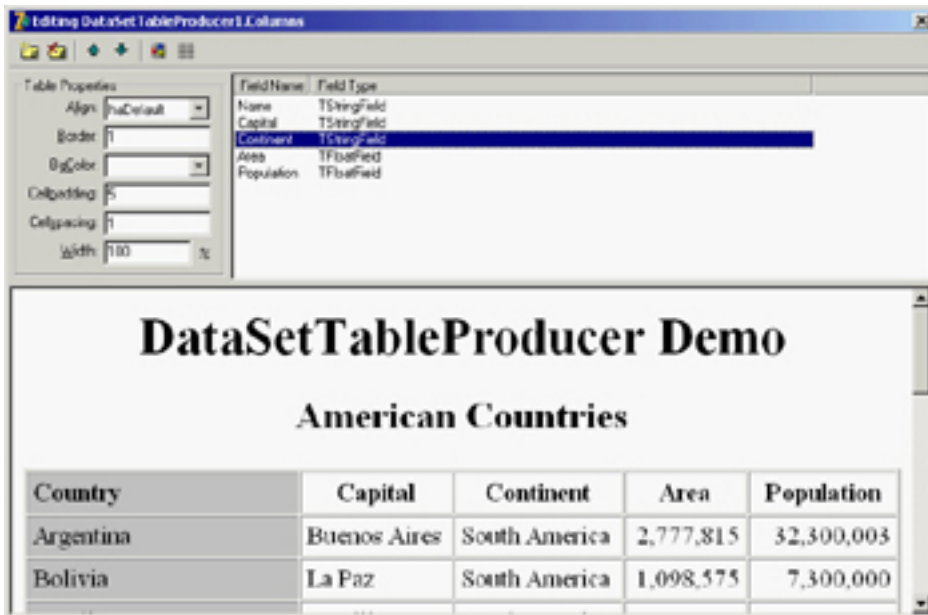
The third button in the HtmlProd example is Print Table. This button is connected to a DataSetTableProducer component, again calling its `Content` function and copying its result to the `Text` of the Memo. By connecting the `DataSet` property of the DataSetTableProducer to `ClientDataSet1`, you can produce a standard HTML table.

The component by default generates only 20 rows, as indicated by the `MaxRows` property. If you want to get all of the table's records, you can set this property to -1—a simple but undocumented setting.

**Tip** The DataSetTableProducer component starts from the current record rather than from the first one. So, the second time you click the Print Table button, you'll see no records in the output. Adding a call to the dataset's `First` method before calling the producer component's `Content` method fixes the problem.

To make the output of this producer component more complete, you can perform two operations. The first is to provide some `Header` and `Footer` information (to generate the HTML heading and closing elements) and add a `Caption` to the HTML table. The second is to customize the table by using the setting specified by the `RowAttributes`, `TableAttributes`, and `Columns` properties. The property editor for the columns, which is also the default component editor, allows you to set most of these properties, providing at the same time a nice preview of the output, as you can see in [Figure 19.9](#). Before using this editor, you can set up properties for the dataset's fields using the Fields editor. This is how, for example, you can format the output of the population and area fields to use thousands separators.





**Figure 19.9:** The editor of the DataSetTable-Producer component's Columns property provides you with a preview of the final HTML table (if the data-base table is active).

You can use three techniques to customize the HTML table, and it's worth reviewing each of them:

- You can use the table producer component's Column's property to set properties, such as the text and color of the title, or the color and the alignment for the cells in the rest of the column.
- You can use the TField properties, particularly those related to output. In the example, I've set the DisplayFormat property of the ClientDataSet1Area field object to ###,###,###. This is the approach to use if you want to determine the output of each field. You might go even further and embed HTML tags in the output of a field.
- You can handle the DataSetTableProducer component's OnFormatCell event to customize the output further. In this event, you can set the various column attributes uniquely for a given cell, but you can also customize the output string (stored in the CellData parameter) and embed HTML tags. You can't do this using the Columns property.

In the HtmlProd example, I've used a handler for this event to turn the text of the Population and Area columns to bold font and to a red background for large values (unless it is the header row). Here is the code:

```

procedure TFormProd.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and
    (((CellColumn = 3) and (Length (CellData) > 8)) or
    ((CellColumn = 4) and (Length (CellData) > 9))) then
    begin
      BgColor := 'red';
      CellData := '<b>' + CellData + '</b>';
    end;
end;

```

The rest of the code is summarized by the settings of the table producer component, including its header and footer, as you can see by opening the source code of the HtmlProd example.

## Using Style Sheets



The latest incarnations of HTML include a powerful mechanism for separating content from presentation: Cascading Style Sheets (CSS). Using a style sheet, you can separate the formatting of the HTML (colors, fonts, font sizes, and so on) from the text displayed (the content of the page). This approach makes your code more flexible and your website easier to update. In addition, you can separate the task of making the site graphically appealing (the work of a web designer) from automatic content generation (the work of a programmer). Style sheets are a complex technique, in which you give formatting values to the main types of HTML sections and to special "classes" (which have nothing to do with OOP). Again, see an HTML reference for the details.

You can update table generation in the `HtmlProd` example to include style sheets by providing a link to the style sheet in the `Header` property of a second `DataSetTableProducer` component:

```
<link rel="stylesheet" type="text/css" href="test.css">
```

You can then update the code of the `OnFormatCell` event handler with the following action (instead of the two lines changing the color and adding the bold font tag):

```
CustomAttrs := 'class="highlight"';
```

The style sheet I've provided (`test.css`, available in the source code of the example) defines a highlight style, which has the bold font and red background that were hard-coded in the first `DataSetTableProducer` component.

The advantage of this approach is that now a graphic artist can modify the CSS file and give your table a nicer look without touching its code. When you want to provide many formatting elements, using a style sheet can also reduce the total size of the HTML file. This is an important element that can reduce download time.

## Dynamic Pages from a Custom Server

The `HtmlProd` component can be used to generate static HTML files; it doubles as a web server, using an approach similar to that demonstrated in the `HttpServ` example, but in a more realistic context. The program accesses the request of one of the possible page producers, passing the name of the component in a request. This is a portion of the `IdHTTPServer` component's `OnCommandGet` event handler, which uses the `FindComponent` method to locate the proper producer component:

```
var
  Req, Html: String;
  Comp: TComponent;
begin
  Req := RequestInfo.Document;
  if Req [1] = '/' then
    Req := Copy (Req, 2, 1000); // skip '/'
  Comp := FindComponent (Req);
  if (Req <> '') and Assigned (Comp) and
    (Comp is TCustomContentProducer) then
    begin
      ClientDataSet1.First;
      Html := TCustomContentProducer (Comp).Content;
    end;
  ResponseInfo.ContentText := Html;
end;
```

In case the parameter is not there (or is not valid), the server responds with an HTML-based menu of the available components:

```
Html := '<h1>HtmlProd Menu</h1><p><ul>';
for I := 0 to ComponentCount - 1 do
  if Components [i] is TCustomContentProducer then
    Html := Html + '<li><a href="/" + Components [i].Name + "' +
      Components [i].Name + '</a></li>';
```

```
Html := Html + '</ul></p>';
```

Finally, if the program returns a table that uses CSS, the browser will request the CSS file from the server; so, I've added some specific code to return it. With the proper generalizations, this code shows how a server can respond by returning files, and also how to indicate the MIME type of the response (`ContentType`):

```
if Pos ('test.css', Req) > 0 then
begin
  CssTest := TStringList.Create;
  try
    CssTest.LoadFromFile(ExtractFilePath(Application.ExeName) + 'test.css');
    ResponseInfo.ContentText := CssTest.Text;
    ResponseInfo.ContentType := 'text/css';
  finally
    CssTest.Free;
  end;
  Exit;
end;
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

In this chapter, I've focused on some core Internet technologies, including the use of sockets and core Internet protocols. I've discussed the main idea and shown a few examples of using the mail and HTTP protocols. You can find many more examples that use the Indy components in the demos written by their developers (which are not installed in Delphi 7).

After this introduction to the world of the Internet, you are now ready to delve into two key areas: the present and the future. The present is represented by the development of web applications, and I'll explore the development of dynamic websites in the next two chapters. I'll focus first on the old WebBroker technology and then move to the new WebSnap architecture. Then, in [Chapter 21](#), I'll discuss IntraWeb. The future is represented by the development of web services and the use of XML and related technology, discussed in [Chapters 22](#) and [23](#).

## Chapter 20: Web Programming with WebBroker and WebSnap

### Overview

The Internet has a growing role in the world, and much of it depends on the success of the World Wide Web, which is based on the HTTP protocol. In [Chapter 19](#), "Internet Programming: Sockets and Indy," we discussed HTTP and the development of client- and server-side applications based on it. With the availability of several high-performance, scalable, flexible web servers, you'll rarely want to create your own. Dynamic web server applications are generally built by integrating scripting or compiled programs within web servers, rather than by replacing them with custom software.

This chapter is entirely focused on the development of server-side applications, which extend existing web servers. I introduced the dynamic generation of HTML pages toward the end of the [last chapter](#). Now you will learn how to integrate this dynamic generation within a server. This chapter is a logical continuation of the last one, but it won't complete this book's coverage of Internet programming; [Chapter 21](#) is devoted to the IntraWeb technology available in Delphi 7, and [Chapter 22](#) gets back to Internet programming from the XML perspective.

**Warning** To test some of the examples in this chapter, you'll need access to a web server. The simplest solution is to use the version of Microsoft's IIS or Personal Web Server already installed on your computer. My personal preference, however, is to use the free open-source Apache Web Server available at [www.apache.org](http://www.apache.org). I won't spend much time giving you details of the configuration of your web server to enable the use of applications; you can refer to its documentation for this information.

## Dynamic Web Pages

When you browse a website, you generally download static pages—HTML-format text files—from the web server to your client computer. As a web developer, you can create these pages manually, but for most businesses, it makes more sense to build the static pages from information in a database (a SQL server, a series of files, and so on). Using this approach, you're basically generating a snapshot of the data in HTML format, which is reasonable if the data isn't subject to frequent changes. This approach was discussed in [Chapter 19](#).

As an alternative to static HTML pages, you can build dynamic pages. To do this, you extract information directly from a database in response to the browser's request, so that the HTML sent by your application displays current data, not an old snapshot of the data. This approach makes sense if the data changes frequently.

As mentioned earlier, there are a couple of ways you can program custom behavior at the web server, and these are ideal techniques you can use to generate HTML pages dynamically. In addition to script-based techniques, which are very popular, two common protocols for programming web servers are CGI (the Common Gateway Interface) and the web server APIs.

**Note** Keep in mind that Delphi's WebBroker technology (available in both the Enterprise Studio and Professional editions) flattens the differences between CGI and server APIs by providing a common class framework. This way, you can easily turn a CGI application into an ISAPI library or integrate it into Apache.

## An Overview of CGI

CGI is a standard protocol for communication between the client browser and the web server. It's not a particularly efficient protocol, but it is widely used and is not platform specific. This protocol allows the browser both to ask for and to send data, and it is based on the standard command-line input and output of an application (usually a console application). When the server detects a page request for the CGI application, it launches the application, passes command-line data from the page request to the application, and then sends the application's standard output back to the client computer.

You can use many tools and languages to write CGI applications, and Delphi is only one of them. Despite the obvious limitation that your web server must be an Intel-based Windows or Linux system, you can build some fairly sophisticated CGI programs in Delphi and Kylix. CGI is a low-level technique, because it uses the standard command-line input and output along with environment variables to receive information from the web server and pass it back.

To build a CGI program without using support classes, you can create a Delphi console application, remove the typical project source code, and replace it with the following statements:

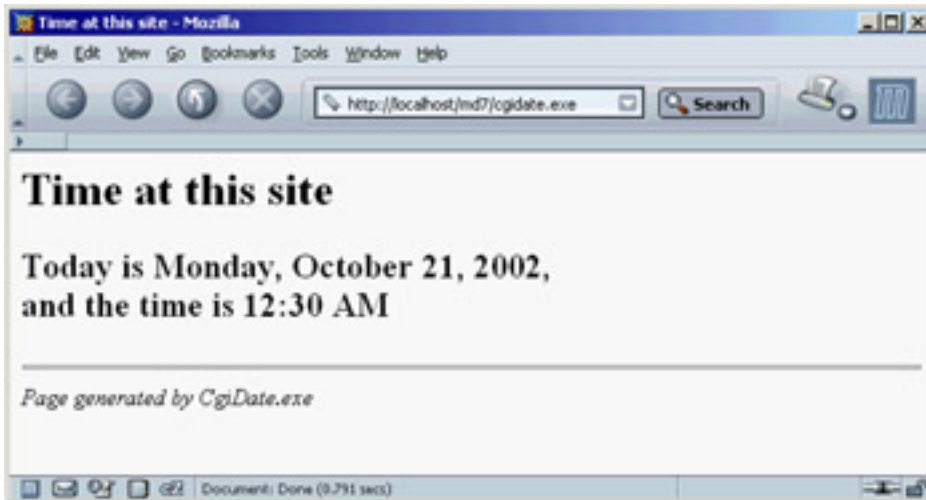
```
program CgiDate;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
  writeln ('content-type: text/html');
  writeln;
  writeln ('<html><head>');
  writeln ('<title>Time at this site</title>');
  writeln ('</head><body>');
  writeln ('<h1>Time at this site</h1>');
  writeln ('<hr>');
  writeln ('<h3>');
  writeln (FormatDateTime('"Today is " dddd, mmmm d, yyyy,' +
    '"<br> and the time is" hh:mm:ss AM/PM', Now));
  writeln ('</h3>');
```

```
writeln ('<hr>');
writeln ('<i>Page generated by CgiDate.exe</i>');
writeln ('</body></html>');
end.
```

CGI programs produce a header followed by the HTML text using the standard output. If you execute this program directly, you'll see the text in a terminal window. If you run it instead from a web server and send the output to a browser, the formatted HTML text will appear, as shown in [Figure 20.1](#).



**Figure 20.1:** The output of the CgiDate application, as seen in a browser

Building advanced applications with plain CGI requires a lot of work. For example, to extract status information about the HTTP request, you need to access the relevant environment variables, as in the following:

```
// get the pathname
GetEnvironmentVariable ('PATH_INFO', PathName, sizeof (PathName));
```

## Using Dynamic Libraries

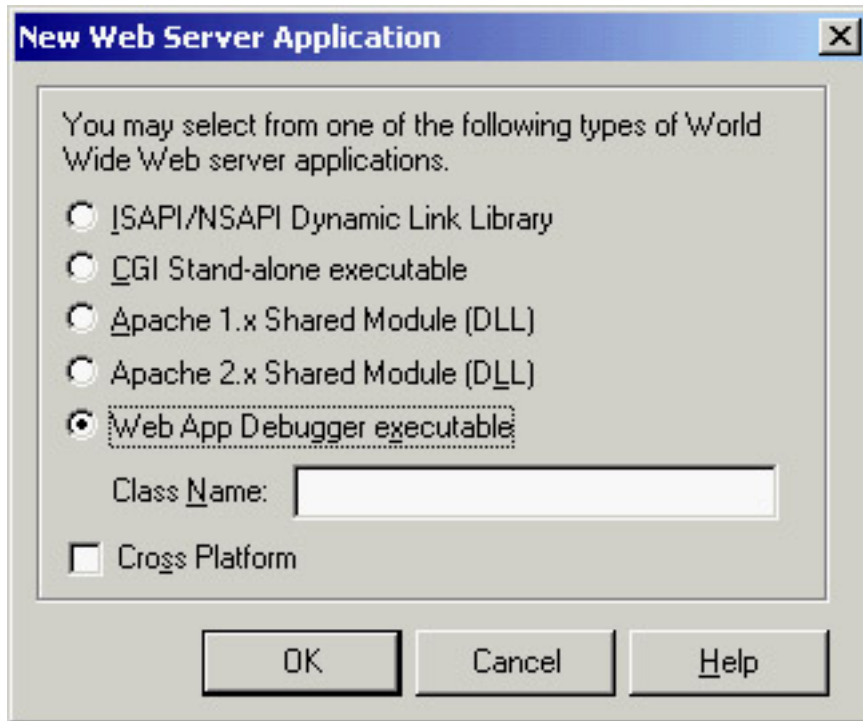
A completely different approach is the use of the web server APIs: the popular ISAPI (Internet Server API, introduced by Microsoft), the less common NSAPI (Netscape Server API), or the Apache API. These APIs allow you to write a library that the server loads into its own address space and keeps in memory. Once it loads the library, the server can execute individual requests via threads within the main process, instead of launching a new EXE for every request (as it must in CGI applications).

When the server receives a page request, it loads the DLL (if it hasn't done so already) and executes the appropriate code, which may launch a new thread or use an existing one to process the request. The library then sends the appropriate HTTP data back to the client that requested the page. Because this communication generally occurs in memory, this type of application tends to be faster than CGI.

## Delphi's WebBroker Technology

The CGI code snippet I've shown you demonstrates the plain, direct approach to this protocol. I could have provided similar low-level examples for ISAPI or Apache modules, but in Delphi it's more interesting to use the WebBroker technology. This comprises a class hierarchy within VCL and CLX (built to simplify server-side development on the Web) and a specific type of data modules called *WebModules*. Both the Enterprise Studio and Professional editions of Delphi include this framework (as opposed to the more advanced and newer WebSnap framework, which is available only in the Enterprise Studio version).

Using the WebBroker technology, you can begin developing an ISAPI or CGI application or an Apache module easily. On the first page (New) of the New Items dialog box, select the Web Server Application icon. The subsequent dialog box will offer you a choice among ISAPI, CGI, Apache 1 or 2 module, and the Web App Debugger:



In each case, Delphi will generate a project with a WebModule, which is a non-visual container similar to a data module. This unit will be identical, regardless of the project type; only the main project file changes. For a CGI application, it will look like this:

```

program Project2;

{$APPTYPE CONSOLE}

uses
  WebBroker,
  CGIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.

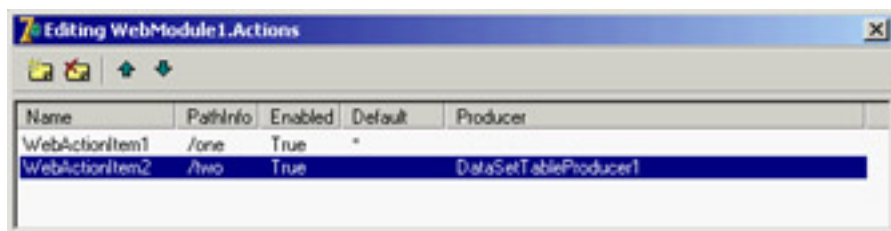
```

Although this is a console CGI program, the code looks similar to that of a standard Delphi application. However, it

uses a trick—the `Application` object used by this program is not the typical global object of class `TApplication` but an object of a new class. This `Application` object is of class `TCGIApplication` or another class derived from `TWebApplication`, depending on your web project type.

The most important operations take place in the `WebModule`. This component derives from `TCustomWebDispatcher`, which provides support for all the input and output of your programs. The `TCustomWebDispatcher` class defines `Request` and `Response` properties, which store the client request and the response you're going to send back to the client. Each of these properties is defined using a base abstract class (`TWebRequest` and `TWebResponse`), but an application initializes them using a specific object (such as the `TISAPIRequest` and `TISAPIResponse` subclasses). These classes make available all the information passed to the server, so you have a single approach to accessing all the information. The same is true of a response, which is easy to manipulate. The key advantage of this approach is that the code written with `WebBroker` is independent of the type of application (CGI, ISAPI, Apache module); you'll be able to move from one to the other, modifying the project file or switching to another one, but you won't need to modify the code written in a `WebModule`.

This is the structure of Delphi's framework. To write the application code, you can use the Actions editor in the `WebModule` to define a series of actions (stored in the `Actions` array property) depending on the *pathname* of the request:



This pathname is a portion of the CGI or ISAPI application's URL, which comes after the program name and before the parameters, such as `path1` in the following URL:

```
http://www.example.com/scripts/cgitest.exe/path1?param1=date
```

By providing different actions, your application can easily respond to requests with different pathnames, and you can assign a different producer component or call a different `OnAction` event handler for every possible pathname. Of course, you can omit the pathname to handle a generic request. Also consider that instead of basing your application on a `WebModule`, you can use a plain data module and add a `WebDispatcher` component to it. This is a good approach if you want to turn an existing Delphi application into a web server extension.

**Warning** The `WebModule` inherits from the base `WebDispatcher` class and doesn't require it as a separate component. Unlike `WebSnap` applications, `WebBroker` programs cannot have multiple dispatchers or multiple web modules. Also note that the actions of the `WebDispatcher` have nothing to do with the actions stored in an `ActionList` or `ActionManager` component.

When you define the accompanying HTML pages that launch the application, the links will make page requests to the URLs for each of those paths. Having a single library that can perform different operations depending on a parameter (in this case, the pathname) allows the server to keep a copy of this library in memory and respond much more quickly to user requests. The same is partially true for a CGI application: The server has to run several instances but can cache the file and make it available more quickly.

In the `OnAction` event, you write the code to specify the *response* to a given *request*, the two main parameters passed to the event handler. Here is an example:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;  
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
begin  
  Response.Content :=  
    '<html><head><title>Hello Page</title></head><body>' +  
    '<h1>Hello</h1>' +
```



```
'<hr><p><i>Page generated by Marco</i></p></body></html>';  
end;
```

In the `Content` property of the `Response` parameter, you enter the HTML code you want users to see. The only drawback of this code is that the output in a browser will be correctly displayed on multiple lines, but looking at the HTML source code, you'll see a single line corresponding to the entire string. To make the HTML source code more readable by splitting it onto multiple lines, you can insert the `#13` newline character (or, even better, the cross-platform `sLineBreak` value).

To let other actions handle this request, you set the last parameter, `Handled`, to `False`. The default value is `True`; if this value is set, then once you've handled the request with your action, the `WebModule` assumes you're finished. Most of a web application's code will be in the `OnAction` event handlers for the actions defined in the `WebModule` container. These actions receive a request from the client and return a response using the `Request` and `Response` parameters.

When you're using the producer components, your `OnAction` event often returns as `Response.Content` the `Content` of the producer component, with an assignment operation. You can shortcut this code by assigning a producer component to the `Producer` property of the action, and you won't need to write these event handlers any more (but don't do both things, because doing so might get you into trouble).

**Tip** As an alternative to the `Producer` property, you can use the `ProducerContent` property. This property allows you to connect custom producer classes that don't inherit from the `TCustomContentProducer` class but implement the `IProduceContent` interface. The `ProducerContent` property is almost an interface property: It behaves the same way, but this behavior is due to its property editor and is not based on Delphi's support for interfaced properties.

## Debugging with the Web App Debugger

Debugging web applications written in Delphi is often difficult. You cannot simply run the program and set breakpoints in it, but must convince the web server to run your CGI program or library within the Delphi debugger. You can do so by indicating a host application in Delphi's Run Parameters dialog box, but this approach implies letting Delphi run the web server (which is often a Windows service, not a stand-alone program).

To solve these issues, Borland has developed a specific Web App Debugger program. This program, which is activated by the corresponding item on the Tools menu, is a web server that waits for requests on a port you can set up (1024 by default). When a request arrives, the program can forward it to a stand-alone executable. In Delphi 6, this communication was based on COM techniques; in Delphi 7 it is based on Indy sockets. In both cases, you can run the web server application from within the Delphi IDE, set all the breakpoints you need, and then (when the program is activated through the Web App Debugger) debug the program as you would a plain executable file.

The Web App Debugger does a good job of logging all the received requests and the responses returned to the browser. The program also has a Statistics page that tracks the time required for each response, allowing you to test the efficiency of an application in different conditions. Another new feature of the Web App Debugger in Delphi 7 is that it is now a CLX application instead of a VCL application. This user interface change and the conversion from COM to sockets were both done to make the Web App Debugger available in Kylix.

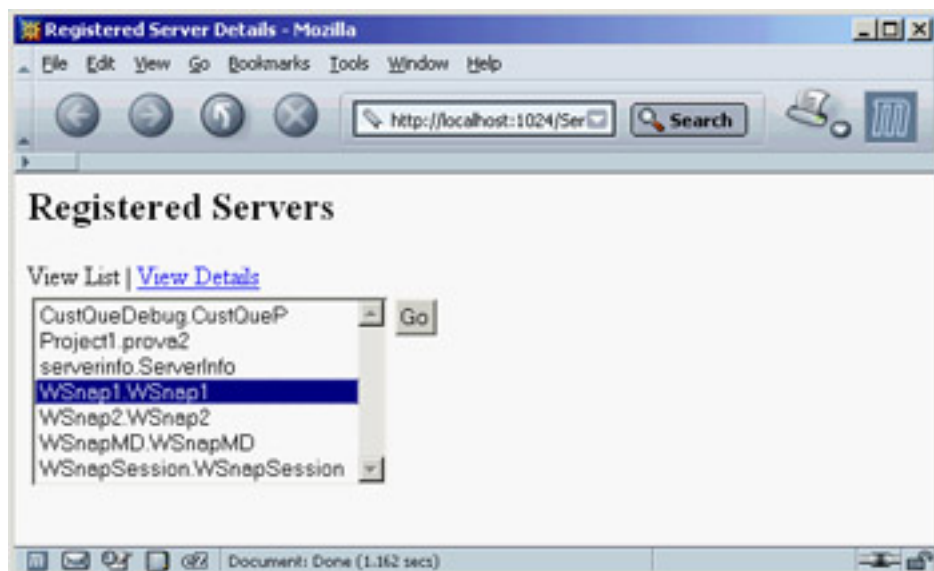
**Warning** Because the Web App Debugger uses Indy sockets, your application will receive frequent exceptions of type `EidConnClosedGracefully`. For this reason, this exception is automatically disabled in all Delphi 7 projects.

By using the corresponding option in the New Web Server Application dialog, you can easily create a new application compatible with the debugger. This option defines a standard project, which creates both a main form and a web module. The (useless) form includes code for providing initialization code and adding the application to the Windows Registry:

### initialization

```
TWebAppSockObjectFactory.Create('program_name');
```

The Web App Debugger uses this information to get a list of the available programs. It does so when you use the default URL for the debugger, indicated in the form as a link, as you can see (for example) in [Figure 20.2](#). The list includes all the registered servers, not only those that are running, and can be used to activate a program. This is not a good idea, though, because you have to run the program within the Delphi IDE to be able to debug it. (Notice that you can expand the list by clicking View Details; this view includes a list of the executable files and many other details.)



**Figure 20.2:** A list of applications registered with the Web App Debugger is displayed when you hook to its home page.

The data module for this type of project includes initialization code:

```
uses WebReq;
```

#### initialization

```
if WebRequestHandler <> nil then  
    WebRequestHandler.WebModuleClass := TWebModule2;
```

The Web App Debugger should be used only for debugging. To deploy the application, you should use one of the other options. You can create the project files for another type of web server program and add to the project the same web module as the debug application.

The reverse process is slightly more complex. To debug an existing application, you have to create a program of this type, remove the web module, add the existing one, and patch it by adding a line to set the `WebModuleClass` of the `WebRequestHandler`, as in the preceding code snippet.

**Warning** Although in most cases you'll be able to move a program from one web technology to another, this is not always the case. For example, in the `CustQueP` example (discussed later), I had to avoid the request's `ScriptName` property (which is fine for a CGI program) and use the `InternalScriptName` property instead.

There are two other interesting elements involved in using the Web App Debugger. First, you can test your program without having a web server installed and without having to tweak its settings. In other words, you don't have to deploy your programs to test them—you can try them right away. Second, instead of doing early development of an application as CGI, you can begin experimenting with a multithreaded architecture immediately, without having to deal with the loading and unloading of libraries (which often implies shutting down the web server and possibly even the computer).

## Building a Multipurpose WebModule

To demonstrate how easily you can build a feature-rich server-side application using Delphi's support, I created the

BrokDemo example. I built this example using the Web App Debugger technology, but it should be relatively simple to recompile as a CGI or a web server library.

A key element of the WebBroker example is the list of actions. The actions can be managed in the Actions editor or directly in the Object TreeView. Actions are also visible in the Designer page of the editor, so you can graphically see their relationship with database objects. If you examine the source code, you'll notice that every action has a specific name. I also gave meaningful names to the `OnAction` event handlers. For instance, `TimeAction` as a method name is much more understandable than the `WebModule1WebActionItem1Action` name automatically generated by Delphi.

Every action has a different pathname, and one is marked as a default and executed even if no pathname is specified. The first interesting idea in this program is the use of two `PageProducer` components, `PageHead` and `PageTail`, which are used for the initial and final portion of every page. Centralizing this code makes it easier to modify, particularly if it is based on external HTML files. The HTML produced by these components is added at the beginning and the end of the resulting HTML in the web module's `OnAfterDispatch` event handler:

```
procedure TWebModule1.WebModule1AfterDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageHead.Content + Response.Content + PageTail.Content;
end;
```

You add the initial and final HTML at the end of the page generation because doing so allows the components to produce the HTML as if they were making all of it. Starting with HTML in the `OnBeforeDispatch` event means that you cannot directly assign the producer components to the actions, or the producer component will override the `Content` you've already provided in the response.

The `PageTail` component includes a custom tag for the script name, replaced by the following code, which uses the current request object available in the web module:

```
procedure TWebModule1.PageTailHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'script' then
    ReplaceText := Request.ScriptName;
end;
```

This code is activated to expand the `<#script>` tag of the `PageTail` component's `HTMLDoc` property.

The code for the time and date actions is straightforward. The really interesting part begins with the `Menu` path, which is the default action. In its `OnAction` event handler, the application uses a `for` loop to build a list of the available actions (using their names without the first two letters, which are always `wa` in this example), providing a link to each of them with an anchor (an `<a>` tag):

```
procedure TWebModule1.MenuAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<h3>Menu</h3><ul>'#13;
  for I := 0 to Actions.Count - 1 do
    Response.Content := Response.Content + '<li> <a href="' +
      Request.ScriptName + Action[I].PathInfo + '"> ' +
      Copy(Action[I].Name, 3, 1000) + '</a>'#13;
  Response.Content := Response.Content + '</ul>';
end;
```

The BrokDemo example also provides users with a list of the system settings related to the request, which is useful for

debugging. It is also instructive to learn how much information (and exactly what information) the HTTP protocol transfers from a browser to a web server and vice versa. To produce this list, the program looks for the value of each property of the `TWebRequest` class, as this snippet demonstrates:

```
procedure TWebModule1.StatusAction(Sender: TObject; Request: TWebRequest;
Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<h3>Status</h3>'#13 +
    'Method: ' + Request.Method + '<br>'#13 +
    'ProtocolVersion: ' + Request.ProtocolVersion + '<br>'#13 +
    'URL: ' + Request.URL + '<br>'#13 +
    'Query: ' + Request.Query + '<br>'#13 + ...
```

## Dynamic Database Reporting

The BrokDemo example defines two more actions, indicated by the `/table` and `/record` pathnames. For these two last actions, the program produces a list of names and then displays the details of one record, using a `DataSetTableProducer` component to format the entire table and a `DataSetPageProducer` component to build the record view. Here are the properties of these two components:

```
object DataSetTableProducer1: TDataSetTableProducer
  DataSet = dataEmployee
  OnFormatCell = DataSetTableProducer1FormatCell
end
object DataSetPage: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<h3>Employee: <#LastName></h3>'
    '<ul><li> Employee ID: <#EmpNo>'
    '<li> Name: <#FirstName> <#LastName>'
    '<li> Phone: <#PhoneExt>'
    '<li> Hired On: <#HireDate>'
    '<li> Salary: <#Salary></ul>')
  OnHTMLTag = PageTailHTMLTag
  DataSet = dataEmployee
end
```

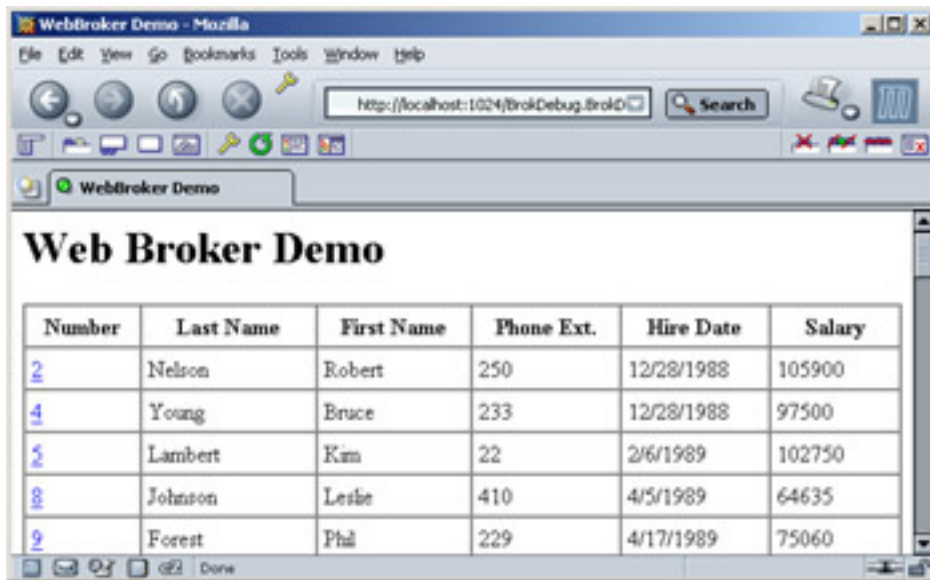
To produce the entire table, you connect the `DataSetTableProducer` to the `Producer` property of the corresponding actions without providing a specific event handler. The table is made more powerful by adding internal links to the specific records. The following code is executed for each cell of the table but a link is created only for the first column and not for the first row (the one with the title):

```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign;
  var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + ScriptName + '/record?LastName=' +
      dataEmployee['Last_Name'] + '&FirstName=' +
      dataEmployee['First_Name'] + '"> '
      + CellData + ' </a>';
end;
```

You can see the result of this action in [Figure 20.3](#). When the user selects one of the links, the program is called again, and it can check the `QueryFields` string list and extract the parameters from the URL. It then uses the values

corresponding to the table fields used for the record search (which is based on the `FindNearest` call):

```
procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  dataEmployee.Open;
  // go to the requested record
  dataEmployee.Locate ('LAST_NAME;FIRST_NAME',
    VarArrayOf([Request.QueryFields.Values['LastName'],
      Request.QueryFields.Values['FirstName']]), []);
  // get the output
  Response.Content := Response.Content + DataSetPage.Content;
end;
```



The screenshot shows a Mozilla browser window titled 'WebBroker Demo'. The address bar contains 'http://localhost:1024/BrokDebug.BrokD'. The main content area displays a table with the following data:

Number	Last Name	First Name	Phone Ext.	Hire Date	Salary
<a href="#">2</a>	Nelson	Robert	250	12/28/1988	105900
<a href="#">4</a>	Young	Bruce	233	12/28/1988	97500
<a href="#">5</a>	Lambert	Kim	22	2/6/1989	102750
<a href="#">8</a>	Johnson	Leslie	410	4/5/1989	64635
<a href="#">9</a>	Forest	Phil	229	4/17/1989	75060

**Figure 20.3:** The output corresponding to the table path of the BrokDemo example, which produces an HTML table with internal hyperlinks

## Queries and Forms

The previous example used some of the HTML producer components introduced earlier in this chapter. This group includes another component you haven't used yet: `QueryTableProducer` (for the BDE) and its sister `SQLQueryTableProducer` (for dbExpress). As you'll see in a moment, this component makes building even complex database programs a breeze.

Suppose you want to search for customers in a database. You might construct the following HTML form (embedded in an HTML table for better formatting):

```
<h4>Customer QueryProducer Search Form</h4>
<form action="<#script>/search" method="POST">
<table>
<tr><td>State:</td>
  <td><input type="text" name="State"></td></tr>
<tr><td>Country:</td>
  <td><input type="text" name="Country"></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

**Note** As in Delphi, an HTML form hosts a series of controls. There are visual tools to help you design these forms, or you can manually enter the HTML code. The available controls include buttons, input text (or edit boxes), selections (or combo boxes), and input buttons (or radio buttons). You can define buttons as specific types, such as Submit or Reset, which imply standard behaviors. An important element of forms is the request method, which can be either *POST* (data is passed behind the scenes, and you receive it in the *ContentFields* property) or *GET* (data is passed as part of the URL, and you extract it from the *QueryFields* property).

You should notice a very important element in the form: the names of the input components (State and Country), which should match the parameters of a SQLQuery component:

```
SELECT Customer, State_Province, Country
FROM CUSTOMER
WHERE
    State_Province = :State OR Country = :Country
```

This code is used in the CustQueP (customer query producer) example. To build it, I placed a SQLQuery component inside the WebModule and generated the field objects for it. In the same WebModule, I added a SQLQueryTableProducer component connected to the *Producer* property of the */search* action. The program generates the proper response. When you activate the SQLQuery-TableProducer component by calling its *Content* function, it initializes the SQLQuery component by obtaining the parameters from the HTTP request. The component can automatically examine the request method and then use either the *QueryFields* property (if the request is a *GET*) or the *ContentFields* property (if the request is a *POST*).

One problem with using a static HTML form as you did earlier is that it doesn't tell you which states and countries you can search for. To address this issue, you can use a selection control instead of an edit control in the HTML form. However, if the user adds a new record to the database table, you'll need to update the element list automatically. As a final solution, you can design the ISAPI DLL to produce a form on-the-fly, and you can fill the selection controls with the available elements.

You'll generate the HTML for this page in the */form* action, which you connect to a PageProducer component. The PageProducer contains the following HTML text, which embeds two special tags:

```
<h4>Customer QueryProducer Search Form</h4>
<form action=" <#script>/search" method="POST">
<table>
<tr><td>State:</td>
    <td><select name="State"><option></option><#State_Province></select></td></tr>
<tr><td>Country:</td>
    <td><select name="Country"><option></option><#Country></select></td></tr>
<tr><td></td>
    <td><center><input type="Submit"></center></td></tr>
</table></form>
```

You'll notice that the tags have the same name as some of the table's fields. When the PageProducer encounters one of these tags, it adds an *<option>* HTML tag for every distinct value of the corresponding field. Here's the *OnTag* event handler's code, which is generic and reusable:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
    const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    ReplaceText := '';
    SQLQuery2.SQL.Clear;
    SQLQuery2.SQL.Add ('select distinct ' + TagString + ' from customer');
    try
        SQLQuery2.Open;
        try
            SQLQuery2.First;
            while not SQLQuery2.EOF do
```

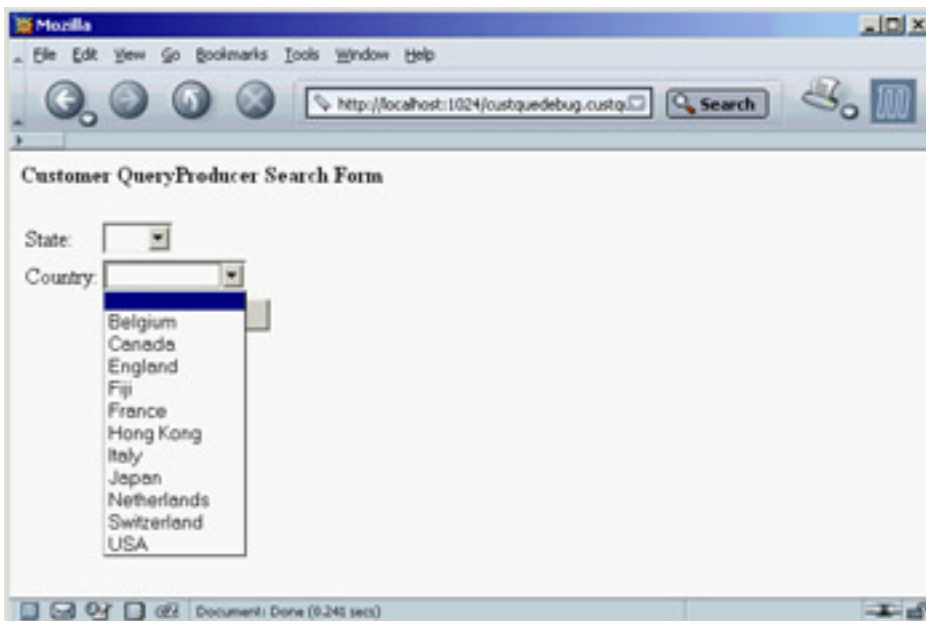


```

begin
  ReplaceText := ReplaceText +
    '<option>' + SQLQuery2.Fields[0].AsString + '</option>'#13;
  SQLQuery2.Next;
end;
finally
  SQLQuery2.Close;
end;
except
  ReplaceText := '{wrong field: ' + TagString + '}';
end;
end;

```

This method uses a second SQLQuery component, which I manually placed on the form and connected to a shared SQLConnection component. It produces the output shown in [Figure 20.4](#).



**Figure 20.4:** The form action of the CustQueP example produces an HTML form with a selection component dynamically updated to reflect the current status of the database.

This web server extension, like many others, allows the user to view the details of a specific record. As in the previous example, you can accomplish this by customizing the output of the first column (column zero), which is generated by the QueryTableProducer component:

```

procedure TWebModule1.QueryTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + Request.ScriptName + '/record?Company=' + CellData +
      '">' + CellData + '</a>'#13;
  if CellData = '' then
    CellData := '&nbsp;';
end;

```

**Tip** When you have an empty cell in an HTML table, most browsers render it without the border. For this reason, I added a *non-breaking* space symbol (&nbsp;) in each empty cell. You'll have to do this in each HTML table generated with Delphi's table producers.

The action for this link is `/record`, and you pass a specific element after the `?` parameter (without the parameter name, which is slightly nonstandard). The code you use to produce the HTML tables for the records doesn't use the producer components as you've been doing; instead, it shows the data of every field in a custom-built table:

```
procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  if Request.QueryFields.Count = 0 then
    Response.Content := 'Record not found'
  else
    begin
      Query2.SQL.Clear;
      Query2.SQL.Add ('select * from customer ' +
        'where Company="' + Request.QueryFields.Values['Company'] + '"');
      Query2.Open;
      Response.Content :=
        '<html><head><title>Customer Record</title></head><body>'#13 +
        '<h1>Customer Record: ' + Request.QueryFields[0] + '</h1>'#13 +
        '<table border>'#13;
      for I := 1 to Query2.FieldCount - 1 do
        Response.Content := Response.Content +
          '<tr><td>' + Query2.Fields [I].FieldName + '</td>'#13'<td>' +
          Query2.Fields [I].AsString + '</td></tr>'#13;
      Response.Content := Response.Content + '</table><hr>'#13 +
        // pointer to the query form
        '<a href="' + Request.ScriptName + '/form">' +
        ' Next Query </a>'#13 + '</body></html>'#13;
    end;
  end;
```

## Working with Apache

If you plan to use Apache instead of IIS or another web server, you can take advantage of the CGI technology to deploy your applications on almost any web server. However, using CGI means reduced speed and some trouble handling state information (because you cannot keep any data in memory). This is a good reason to write an ISAPI application or a dynamic Apache module. Using Delphi's WebBroker technology, you can also easily compile the same code for both technologies, so that moving your program to a different web platform becomes much simpler. You can also recompile a CGI program or a dynamic Apache module with Kylix and deploy it on a Linux server.

As I've mentioned, Apache can run traditional CGI applications but also has a specific technology for keeping the server extension program loaded in memory at all times for faster response. To build such a program in Delphi, you can use the Apache Shared Module option in the New Web Server Application dialog box; choose Apache 1 or Apache 2, depending on the version of the web server you plan to use.

**Warning** While D7 supports version 2.0.39 of Apache, it doesn't support the current popular version 2.0.40 due to a change in library interfaces. Use of version 2.0.39 is discouraged because it has a security problem. Information on how to fix the VCL code and make your modules compatible with Apache 2.0.40 and above have been posted by Borland R&D members on the newsgroups and are currently available also on Bob Swart's website at the URL [www.drbob42.com/delphi7/apache2040.htm](http://www.drbob42.com/delphi7/apache2040.htm).

If you choose to create an Apache module, you end up with a library having this type of source code for its project:

```
library Apache1;
```



```

uses
    WebBroker,
    ApacheApp,
    ApacheWm in 'ApacheWm.pas' {WebModule1: TWebModule};

{$R *.res}

exports
    apache_module name 'apache1_module';

begin
    Application.Initialize;
    Application.CreateForm(TWebModule1, WebModule1);
    Application.Run;
end.

```

Notice in particular the `exports` clause, which indicates the name used by Apache configuration files to reference the dynamic module. In the project source code, you can add two more definitions—the module name and the content type—in the following way:

```

ModuleName := 'Apache1_module';
ContentType:= 'Apache1-handler';

```

If you don't set these values, Delphi will assign them defaults, which are built adding the `_module` and `-handler` strings to the project name (resulting in the two names I used here).

An Apache module is generally not deployed within a script folder, but within the `modules` subfolder of the server (by default, `c:\Program Files\Apache\modules`). You have to edit the `http.conf` file, adding a line to load the module, as follows:

```
LoadModule apache1_module modules/apache1.dll
```

Finally, you must indicate when the module is invoked. The handler defined by the module can be associated with a given file extension (so that your module will process all the files having a given extension) or with a physical or virtual folder. In the latter case, the folder doesn't exist, but Apache pretends it is there. This is how you can set up a virtual folder for the Apache1 module:

```
<Location /Apache1>SetHandler Apache1-handler</Location>
```

Because Apache is inherently case sensitive (because of its Linux heritage), you also might want to add a second, lowercase virtual folder:

```
<Location /apache1>SetHandler Apache1-handler</Location>
```

Now you can invoke the sample application with the URL `http://localhost/Apache1`. A great advantage of using virtual folders in Apache is that a user doesn't really distinguish between the physical and dynamic portions of your site, as you can see by experimenting with the Apache1 example (which includes the code discussed here).

## Practical Examples

After this general introduction to developing server-side applications with WebBroker, I'll end this part of the chapter with two practical examples. The first is a classic web counter. The second is an extension of the WebFind program presented in [Chapter 19](#), which produces a dynamic page instead of filling a list box.

### A Graphical Web Hit Counter

The server-side applications you've built up to now were based only on text. Of course, you can easily add references to existing graphics files. What's more interesting, however, is to build server-side programs capable of generating graphics that change over time.

A typical example is a *page hit counter*. To write a web counter, you save the current number of hits to a file and then read and increase the value every time the counter program is called. To return this information, all you need is HTML text with the number of hits. The code is straightforward:

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  nHit: Integer;
  LogFile: Text;
  LogFileName: string;
begin
  LogFileName := 'WebCont.log';
  System.Assign (LogFile, LogFileName);
  try
    // read if the file exists
    if FileExists (LogFileName) then
      begin
        Reset (LogFile);
        Readln (LogFile, nHit);
        Inc (nHit);
      end
    else
      nHit := 0;
    // saves the new data
    Rewrite (LogFile);
    Writeln (LogFile, nHit);
  finally
    Close (LogFile);
  end;
  Response.Content := IntToStr (nHit);
end;

```

**Warning** This simple file handling does not scale. When multiple visitors hit the page at the same time, this code may return false results or fail with a file I/O error because a request in another thread has the file open for reading while this thread tries to open the file for writing. To support a similar scenario, you'll need to use a mutex (or a critical section in a multithreaded program) to let each subsequent thread wait until the thread currently using the file has completed its task.

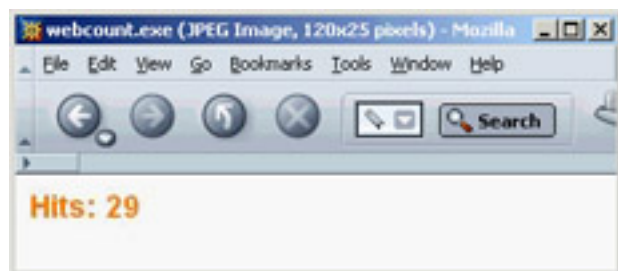
It's more interesting to create a graphical counter that can be easily embedded into any HTML page. There are two approaches to building a graphical counter: You can prepare a bitmap for each digit up front and then combine them in the program, or you can let the program draw over a memory bitmap to produce the graphic you want to return. In the WebCount program, I chose the second approach.

Basically, you can create an Image component that holds a memory bitmap, which you can paint on with the usual

methods of the `TCanvas` class. Then you can attach this bitmap to a `TJpegImage` object. Accessing the bitmap through the `JpegImage` component converts the image to the JPEG format. Then, you can save the JPEG data to a stream and return it. As you can see, there are many steps, but the code is not complicated:

```
// create a bitmap in memory
Bitmap := TBitmap.Create;
try
  Bitmap.Width := 120;
  Bitmap.Height := 25;
  // draw the digits
  Bitmap.Canvas.Font.Name := 'Arial';
  Bitmap.Canvas.Font.Size := 14;
  Bitmap.Canvas.Font.Color := RGB (255, 127, 0);
  Bitmap.Canvas.Font.Style := [fsBold];
  Bitmap.Canvas.TextOut (1, 1, 'Hits: ' +
    FormatFloat ('###,###,###', Int (nHit)));
  // convert to JPEG and output
  Jpeg1 := TJpegImage.Create;
  try
    Jpeg1.CompressionQuality := 50;
    Jpeg1.Assign(Bitmap);
    Stream := TMemoryStream.Create;
    Jpeg1.SaveToStream (Stream);
    Stream.Position := 0;
    Response.ContentStream := Stream;
    Response.ContentType := 'image/jpeg';
    Response.SendResponse;
    // the response object will free the stream
  finally
    Jpeg1.Free;
  end;
finally
  Bitmap.Free;
end;
```

The three statements responsible for returning the JPEG image are the two that set the `ContentStream` and `ContentType` properties of the `Response` and the final call to `SendResponse`. The content type must match one of the possible MIME types accepted by the browser, and the order of these three statements is relevant. The `Response` object also has a `SendStream` method, but it should be called only after sending the type of the data with a separate call. Here you can see the effect of this program:



To embed the program in a page, add the following code to the HTML:

```

```

## Searching with a Web Search Engine

In [Chapter 19](#), I discussed the use of the Indy HTTP client component to retrieve the result of a search on the Google

website. Let's extend the example, turning it into a server-side application. The WebSearcher program, available as a CGI application or a Web App Debugger executable, has two actions: The first returns the HTML retrieved by the search engine; and the second parses the HTML filling a client data set component, which is hooked to a table page producer for generating the final output. Here is the code for the second action:

```
const
  strSearch = 'http://www.google.com/search?as_q=borland+delphi&num=100';

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  I: integer;
begin
  if not cds.Active then
    cds.CreateDataSet
  else
    cds.EmptyDataSet;
  for i := 0 to 5 do // how many pages?
  begin
    // get the data form the search site
    GrabHtml (strSearch + '&start=' + IntToStr (i*100));
    // scan it to fill the cds
    HtmlStringToCds;
  end;
  cds.First;
  // return producer content
  Response.Content := DataSetTableProducer1.Content;
end;
```

The GrabHtml method is identical to the WebFind example. The HtmlStringToCds method is similar to the corresponding method of the WebFind example (which adds the items to a list box); it adds the addresses and their textual descriptions by calling

```
cds.InsertRecord ([0, strAddr, strText]);
```

The ClientDataSet component is set up with three fields: the two strings plus a line counter. This extra empty field is required in order to include the extra column in the table producer. The code fills the column in the cell-formatting event, which also adds the hyperlink:

```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if CellRow <> 0 then
  case CellColumn of
    0: CellData := IntToStr (CellRow);
    1: CellData := '<a href="' + CellData + '">' + SplitLong(CellData) + '</a>';
    2: CellData := SplitLong (CellData);
  end;
end;
```

The call to SplitLong is used to add extra spaces in the output text, to avoid having grid columns that are too large—the browser won't split the text on multiple lines unless it contains spaces or other special characters. The result of this program is a rather slow application (because of the multiple HTTP requests it must forward) that produces output like that shown in [Figure 20.5](#).

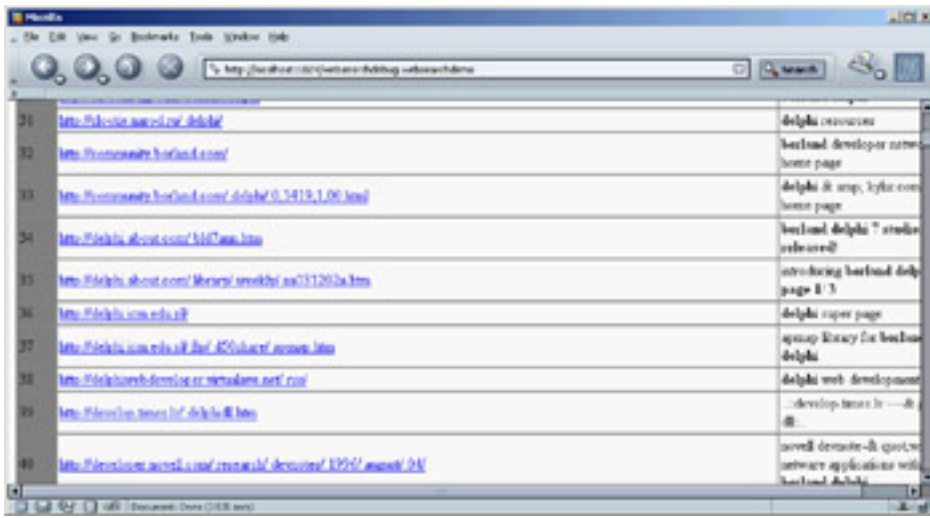
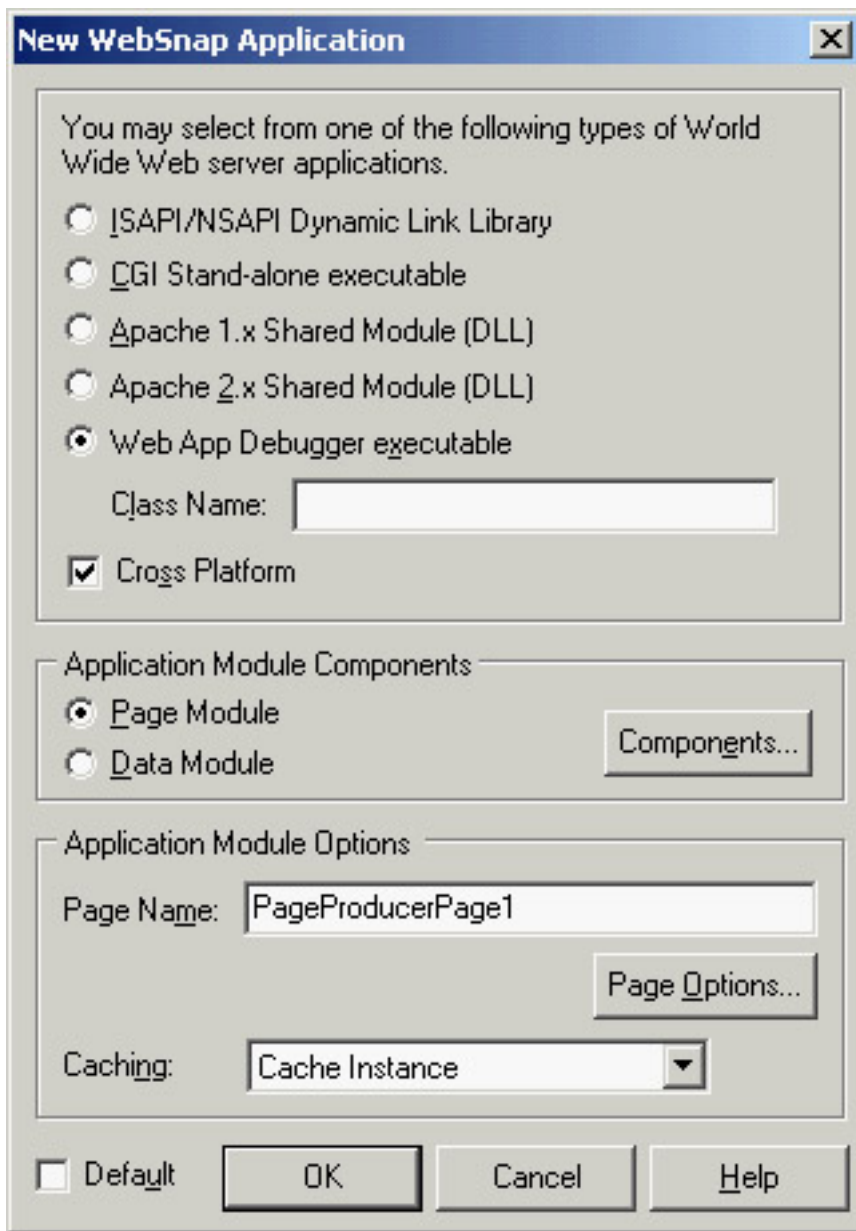


Figure 20.5: The WebSearch program shows the result of multiple searches done on Google.

Now that I've introduced the core elements of developing web server applications with Delphi, let's move to a more complex architecture available since Delphi 6: WebSnap. There are two good reasons I didn't jump into this topic at the beginning of this chapter. First, WebSnap builds on the foundation offered by WebBroker, so you cannot learn how to use the new features if you don't understand the underlying ones. For example, a WebSnap application is technically a CGI program, or an ISAPI or Apache module. Second, because WebSnap is included only in the Enterprise Studio version of Delphi, not all Delphi programmers have the chance to use it.

WebSnap has a few definite advantages over WebBroker, such as allowing for multiple web modules each corresponding to a page, integrating server-side scripting, XSL, and the Internet Express technology (these last two elements will be covered in [Chapter 22](#), "Using XML Technologies"). Moreover, many ready-to-use components are available for handling common tasks, such as user logins, session management, and so on. Instead of listing all the features of WebSnap, though, I've decided to cover them in a sequence of simple and focused applications. I built these applications using the Web App Debugger, for testing purposes, but you can easily deploy them using one of the other available technologies.

When you're developing a WebSnap application, the starting point is a dialog box you can invoke either in the WebSnap page of the New Items dialog box (File → New → Other) or using the Internet toolbar in the IDE (which is not visible by default). The resulting dialog box, shown in [Figure 20.6](#), allows you to choose the type of application (as in a WebBroker application) and to customize the initial application components (you'll be able to add more later). The bottom portion of the dialog determines the behavior of the first page (usually the default or home page of the program). A similar dialog box is displayed for subsequent pages.



**Figure 20.6:** The options offered by the New Web-Snap Application dialog box include the type of server and a button that lets you select the core application components.

If you choose the defaults and type in a name for the home page, the dialog box will create a project and open a `TWebAppPageModule`. This module contains the components you've chosen, by default:

- A `WebAppComponents` component is a container for all the centralized services of the WebSnap application, such as the user list, core dispatcher, session services, and so on. Not all of its properties must be set up, because an application might not need all the available services.
- One of these core services is offered by the `PageDispatcher` component, which (automatically) holds a list of the application's available pages and defines the default page.
- Another core service is given by the `AdapterDispatcher` component, which handles HTML form submissions and image requests.
- The `ApplicationAdapter` is the first component you encounter from the *adapters* family. These components offer fields and actions to the server-side scripts evaluated by the program. Specifically, the `ApplicationAdapter` is a field adapter that exposes the value of its own `ApplicationTitle` property. If you enter a value for this property, it will be made available to the scripts.
- The module hosts a `PageProducer` that includes the HTML code of the page—in this case, the program's default

page. Unlike in WebBroker applications, the HTML for this component is not stored inside its `HTMLDoc` string list property or referenced by its `HTMLFile` property. The HTML file is an external file, stored by default in the folder hosting the project's source code and referenced from the application using a statement similar to a resource include statement: `{*.html}`.

- Because the HTML file included by the PageProducer is kept as a separate file (the `LocateFileService` component will eventually help you for its deployment), you can edit it to change the output of a page of your program without having to recompile the application. These possible changes relate not only to the fixed portion of the HTML file but also to some of its dynamic content, thanks to the support for server-side scripting. The default HTML file, based on a standard template, already contains some scripting.

**Warning** The similarity between resource inclusion and HTML reference is mostly syntactical. The HTML reference is used only for the design-time location of the file, whereas a resource include directive links the data it refers into the executable file.

You can view the HTML file in the Delphi editor thanks to that directive (with reasonably good syntax highlighting) by selecting the corresponding lower tab. The editor also has pages for a WebSnap module, including by default an HTML Result page where you can see the HTML generated after evaluating the scripts, and a Preview page hosting what a user will see in a browser. The Delphi 7 editor for a WebSnap module includes a much more powerful HTML editor than Delphi 6 had; it includes better syntax highlighting and code completion. If you prefer to edit your web application's HTML with a more sophisticated editor, you can set up your choice in the Internet page of the Environment Options dialog box. Click the Edit button for the HTML extension, and you can choose an external editor from the editor's shortcut menu or a specific button on Delphi's Internet toolbar.

The standard HTML template used by WebSnap adds to any page of the program its title and the application title, using script lines such as these:

```
<h1><%= Application.Title %></h1>
<h2><%= Page.Title %></h2>
```

We'll get back to the scripting in a while; we'll begin developing the `WSnap1` example in the [next section](#) by creating a program with multiple pages. But first, I'll finish this overview by showing you the source code for a sample web page module:

```
type
  Thome = class(TWebAppPageModule)
    ...
  end;

function home: Thome;

implementation

{$R *.dfm} {*.html}

uses WebReq, WebCntxt, WebFact, Variants;

function home: Thome;

begin
  Result := Thome(WebContext.FindModuleClass(Thome));
end;

initialization
  if WebRequestHandler <> nil then

    WebRequestHandler.AddWebModuleFactory(TWebAppPageModuleFactory.Create(
      Thome, TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html'),
      caCache));
```



end.

The module uses a global function instead of a form's typical global object to support page caching. This application also has extra code in the initialization section (particularly registration code) to let the application know the role of the page and its behavior.

**Tip** Unlike this chapter's WebBroker examples, the WebSnap examples compile each in its own folder. This is because the HTML files are needed at runtime and I preferred simplifying the deployment as much as possible.

## Managing Multiple Pages

The first notable difference between WebSnap and WebBroker is that instead of having a single data module with multiple actions eventually connected to producer components, WebSnap has multiple data modules, each corresponding to an action and having a producer component with an HTML file attached to it. You can add multiple actions to a page/module, but the idea is that you structure applications around pages rather than actions. As is the case with actions, the name of the page is indicated in the request path.

As an example, I added two more pages to the WebSnap application (which was built with default settings). For the first page, in the New WebSnap Page Module dialog (see [Figure 20.7](#)) choose a standard page producer and name it *date*. For the second, choose a DataSetPageProducer and name it *country*. After saving the files, you can begin testing the application. Thanks to some of the scripting I'll discuss later, each page lists all the available pages (unless you've unchecked the Published check box in the New WebSnap Page Module dialog).

**New WebSnap Page Module**

Producer

Type: PageProducer

Script Engine: JScript

HTML

New File:

Template: Standard

Page

Name: PageProducerPage1

Title: PageProducerPage1

Published:  Login Required:

Module Options

Creation: On Demand

Caching: Cache Instance

Default OK Cancel Help

**Figure 20.7:** The New WebSnap Page Module dialog box

The pages will be rather empty, but at least you have the structure in place. To complete the home page, you'll edit its linked HTML file directly. For the date page, use the same approach as a WebBroker application. Add some custom tags to the HTML text, like the following:

```
<p>The time at this site is <#time>.</p>
```

I also added code to the producer component's `OnTag` event handler to replace this tag with the current time.

For the third page (the country page), modify the HTML to include tags for the various fields of the country table, as in:

```
<h3>Country: <#name></h3>
```

Then attach the `ClientDataSet` to the page producer:

```
object DataSetPageProducer: TDataSetPageProducer
  DataSet = cdsCountry
end
object cdsCountry: TClientDataSet
  FileName = 'C:\Program Files\Common Files\Borland
  Shared\Data\country.cds'
end
```

To open this dataset when the page is first created and reset it to the first record in further invocations, you handle the `OnBeforeDispatchPage` event of the web page module, adding this code to it:

```
cdsCountry.Open;
cdsCountry.First;
```

The fact that a WebSnap page can be very similar to a portion of a WebBroker application (basically, an action tied to a producer) is important, if you want to port existing WebBroker code to this new architecture. You can even port your existing `DataSetTableProducer` components to the new architecture. Technically, you can generate a new page, remove its producer component, replace it with a `DataSetTableProducer`, and hook this component to the `PageProducer` property of the web page module. In practice, this approach would cut out the HTML file of the page and its scripts.

In the `WSnap1` program, I used a better technique. I added a custom tag (`<#htmltable>`) to the HTML file and used the `OnTag` event of the page producer to add to the HTML the result of the data set table:

```
if TagString = 'htmltable' then
  ReplaceText := DataSetTableProducer1.Content;
```

## Server-Side Scripts

Having multiple pages in a server-side program—each associated with a different page module—changes the way you write a program. Having the server-side scripts at hand offers an even more powerful approach. For example, the standard scripts of the `WSnap1` example account for the application and page titles, and for the index of the pages. This index is generated by an *enumerator*, the technique used to scan a list from within WebSnap script code. Let's look at it:

```
<table cellpadding="0" cellspacing="0"><td>
<% e = new Enumerator(Pages)
   s = ''
   c = 0
   for (; !e.atEnd(); e.moveNext())
   {
     if (e.item().Published)
     {
       if (c > 0) s += '&nbsp;|&nbsp;';
```

```

    if (Page.Name != e.item().Name)
        s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
    else
        s += e.item().Title
    c++
}
}
if (c>1) Response.Write(s)
%>
</td></table>

```

**Note** Typically, WebSnap scripts are written in JavaScript, an object-based language common for Internet programming because it is the only scripting language generally available in browsers (on the client side). JavaScript (technically indicated as ECMAScript) borrows the core syntax of the C language and has almost nothing to do with Java. WebSnap uses Microsoft's ActiveScripting engine, which supports both JScript (a variation of JavaScript) and VBScript.

Inside the single cell of this table (which, oddly enough, has no rows), the script outputs a string with the `Response.Write` command. This string is built with a `for` loop over an enumerator of the application's pages, stored in the `Pages` global entity. The title of each page is added to the string only if the page is published. Each title uses a hyperlink with the exclusion of the current page. Having this code in a script, instead of hard-coded into a Delphi component, allows you to pass it to a good web designer, who can turn it into something a more visually appealing.

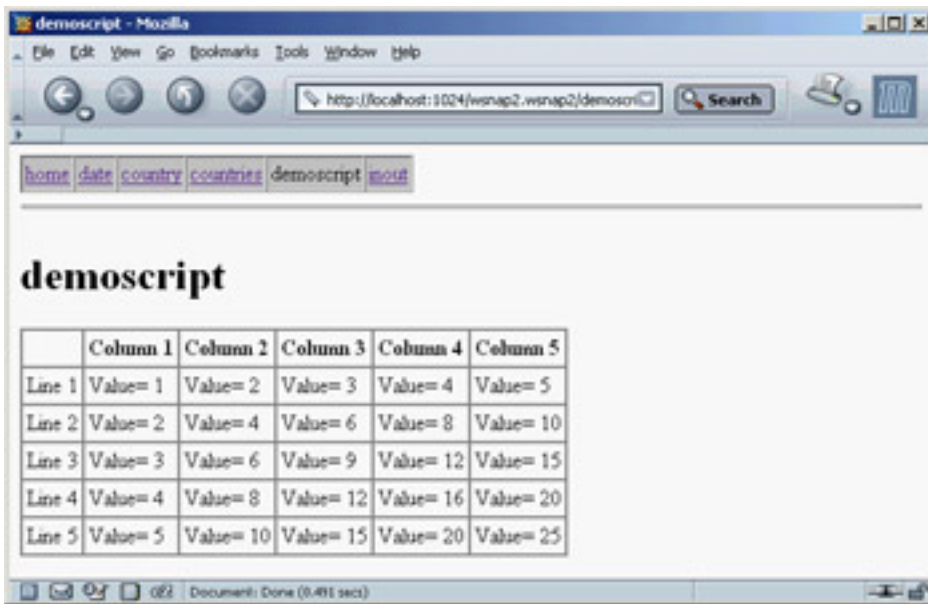
**Tip** To publish or unpublish a page, don't look for a property in the web page module. This status is controlled by a flag of the `AddWebModuleFactory` method called in the web page module initialization code. You can comment or uncomment this flag to obtain the desired effect.

As a sample of what you can do with scripting, I added to the WSnap2 example (an extension of the WSnap1 example) a *demoscript* page. The page's script can generate a full table of multiplied values with the following scripting code (see [Figure 20.8](#) for its output):

```

<table border=1 cellspacing=0>
<tr>
  <th>&nbsp;</th>
  <% for (j=1;j<=5;j++) { %>
  <th>Column <%=j %></th>
  <% } %>
</tr>
<% for (i=1;i<=5;i++) { %>
<tr>
  <td>Line <%=i %></td>
  <% for (j=1;j<=5;j++) { %>
  <td>Value= <%=i*j %></td>
  <% } %>
</tr>
<% } %>
</table>

```



**Figure 20.8:** The WSnap2 example features a plain script and a custom menu stored in an include file.

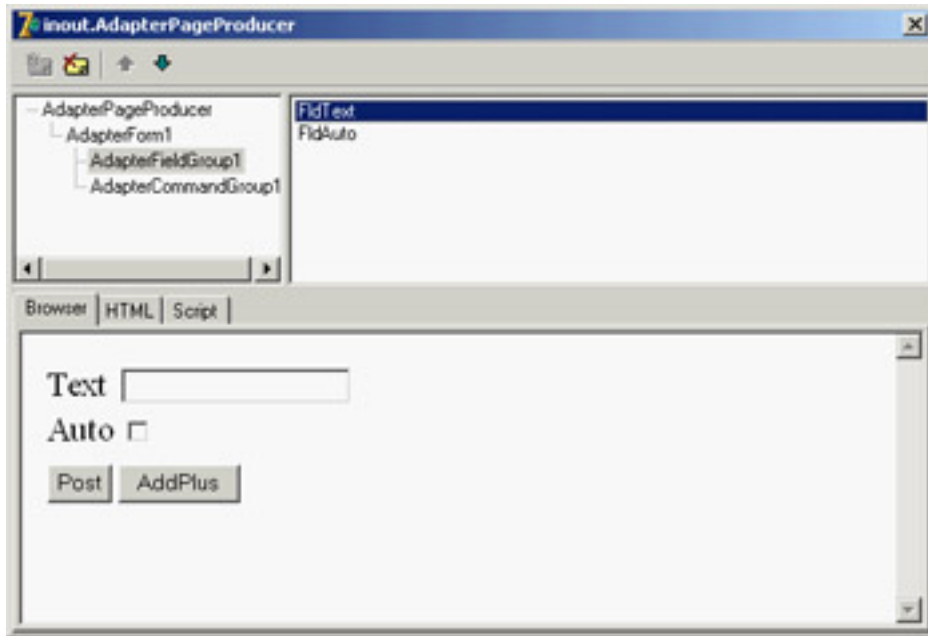
In this script, the `<%=` symbol replaces the longer `Response.Write` command. Another important feature of server-side scripting is the inclusion of pages within other pages. For example, if you plan to modify the menu, you can include the related HTML and script in a single file, instead of changing it and maintaining it in multiple pages. File inclusion is generally done with a statement like this:

```
<!-- #include file="menu.html" -->
```

In [Listing 20.1](#), you can find the complete source code of the include file for the menu, which is referenced by all of the project's other HTML files. [Figure 20.9](#) shows an example of this menu, which is displayed across the top of the page using the table-generation script mentioned earlier.

### Listing 20.1: The `menu.html` File Included in Each Page of the WSnap2 Example

```
<html>
<head>
<title><%= Page.Title %></title>
</head>
<body>
<h2><%= Application.Title %></h2>
<table cellspacing="0" cellpadding="2" border="1" bgcolor="#c0c0c0">
<tr>
<% e = new Enumerator(Pages)
for (; !e.atEnd(); e.MoveNext())
{
if (e.item().Published)
{
if (Page.Name != e.item().Name)
Response.Write ('<td><a href="' + e.item().HREF + '"' +
e.item().Title + '</a></td>')
else
Response.Write ('<td>' + e.item().Title + '</td>')
}
}
%>
</tr>
</table>
<hr>
<h1><%= Page.Title %></h1>
```



**Figure 20.9:** The Web Surface Designer for the inout page of the WSnap2 example, at design time

This script for the menu uses the `Pages` list and the `Page` and `Application` global scripting objects. WebSnap makes available a few other global objects, including `EndUser` and `Session` objects (in case you add the corresponding adapters to the application), the `Modules` object, and the `Producer` object, which allows access to the `Producer` component of the web page module. The script also has available the `Response` and `Request` objects of the web module.

## Adapters

In addition to these global objects, within a script you can access all the adapters available in the corresponding web page module. (Adapters in other modules, including shared web data modules, must be referenced by prefixing their name with the `Modules` object and the corresponding module.) Adapters allow you to pass information from your compiled Delphi code to the interpreted script, providing a scriptable interface to your Delphi application. Adapters contain fields that represent data and host actions that represent commands. The server-side scripts can access these values and issue these commands, passing specific parameters to them.

## Adapter Fields

For simple customizations, you can add new fields to specific adapters. For instance, in the WSnap2 example, I added a custom field to the application adapter. After selecting this component, you can either open its Fields editor (accessible via its shortcut menu) or work in the Object TreeView. After adding a new field (called `AppHitCount` in the example), you can assign a value to it in its `OnGetValue` event. Because you want to count the hits (or requests) on any page of the web application, you can also handle the `OnBeforePageDispatch` event of the *global* `PageDispatcher` component to increase the value of a local field, `HitCount`. Here is the code for the two methods:

```
procedure Thome.PageDispatcherBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
begin
  Inc (HitCount);
end;
```

```
procedure Thome.AppHitCountGetValue(Sender: TObject; var Value: Variant);
begin
```

```
Value := HitCount;  
end;
```

Of course, you could use the page name to also count hits on each page (and you could add support for persistency, because the count is reset every time you run a new instance of the application). Now that you've added a custom field to an existing adapter (corresponding to the `Application` script object), you can access it from within any script, like this:

```
<p>Application hits since last activation:  
<%= Application.AppHitCount.Value %></p>
```

## Adapter Components

In the same way, you can add custom adapters to specific pages. If you need to pass along a few fields, use the generic Adapter component. Other custom adapters (besides the global `ApplicationAdapter` you've already used) include these:

- The `PagedAdapter` component has built-in support for showing its content over multiple pages.
- The `DataSetAdapter` component is used to access a Delphi dataset from a script and is covered in the section "[WebSnap and Databases](#)."
- The `StringValuesList` holds a list of name/value pairs, such as a string list, and can be used directly or to provide a list of values to an adapter field. The inherited `DataSetValuesList` adapter has the same role but grabs the list of name/value pairs from a dataset, providing support for lookups and other selections.
- User-related adapters, such as the `EndUser`, `EndUserSession`, and `LoginForm` adapters, are used to access user and session information and to build a login form for the application that is automatically tied to the users list. I'll discuss these adapters in the section "[Sessions, Users, and Permissions](#)" later in this chapter.

## Using the AdapterPageProducer

Most of these components are used in conjunction with an `AdapterPageProducer` component. The `AdapterPageProducer` can generate portions of script after you visually design the desired result. As an example, I've added to the `WSnap2` application the `inout` page, which has an adapter with two fields, one standard and one Boolean:

```
object Adapter1: TAdapter  
  OnBeforeExecuteAction = Adapter1BeforeExecuteAction  
  object TAdapterActions  
    object AddPlus: TAdapterAction  
      OnExecute = AddPlusExecute  
    end  
    object Post: TAdapterAction  
      OnExecute = PostExecute  
    end  
  end  
end  
object TAdapterFields  
  object Text: TAdapterField  
    OnGetValue = TextGetValue  
  end  
  object Auto: TAdapterBooleanField  
    OnGetValue = AutoGetValue  
  end  
end  
end
```

The adapter also has a couple of actions that post the current user input and add a plus sign (+) to the text. The same plus sign is added when the `Auto` field is enabled. Developing the user interface for this form and the related scripting

would take some time using plain HTML. But the AdapterPageProducer component (used in this page) has an integrated HTML designer, which Borland calls Web Surface Designer. Using this tool, you can visually add a form to the HTML page and add an AdapterFieldGroup to it. Connect this field group to the adapter to automatically display editors for the two fields. Then you can add an AdapterCommandGroup and connect it to the AdapterFieldGroup, to provide buttons for all of the adapter's actions. You can see an example of this designer in [Figure 20.9](#).

To be more precise, the fields and buttons are automatically displayed if the `AddDefaultFields` and `AddDefaultCommands` properties of the field group and command group are set. The effect of these visual operations to build this form are summarized in the following DFM snippet:

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = Adapter1
      object FldText: TAdapterDisplayField
        FieldName = 'Text'
      end
      object FldAuto: TAdapterDisplayField
        FieldName = 'Auto'
      end
    end
  object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterFieldGroup1
    object CmdPost: TAdapterActionButton
      ActionName = 'Post'
    end
    object CmdAddPlus: TAdapterActionButton
      ActionName = 'AddPlus'
    end
  end
end
```

Now that you have an HTML page with some scripts to move data back and forth and issue commands, let's look at the source code required to make this example work. First, you must add to the class two local fields to store the adapter fields and manipulate them, and you need to implement the `OnGetValue` event for both, returning the field values. When each button is clicked, you must retrieve the text passed by the user, which is not automatically copied into the corresponding adapter field. You can obtain this effect by looking at the `ActionValue` property of these fields, which is set only if something was entered (for this reason, when nothing is entered you set the Boolean field to `False`). To avoid repeating this code for both actions, place it in the `OnBeforeExecuteAction` event of the web page module:

```
procedure Tinout.Adapter1BeforeExecuteAction(Sender, Action: TObject;
  Params: TStrings; var Handled: Boolean);
begin
  if Assigned (Text.ActionValue) then
    fText := Text.ActionValue.Values [0];
    fAuto := Assigned (Auto.ActionValue);
  end;
```

Notice that each action can have multiple values (in case components allow multiple selections); but this is not the case, so you can grab the first element. Finally, here is the code for the `OnExecute` events of the two actions:

```
procedure Tinout.AddPlusExecute(Sender: TObject; Params: TStrings);
begin
  fText := fText + '+';
end;
```



```

procedure TInout.PostExecute(Sender: TObject; Params: TStrings);
begin
    if fAuto then
        AddPlusExecute (Self, nil);
end;

```

As an alternative, adapter fields have a public `EchoActionFieldValue` property you can set to get the value entered by the user and place it in the resulting form. This technique is typically used in case of errors, to let the user change the input starting with the values already entered.

**Note** The `AdapterPageProducer` component has specific support for Cascading Style Sheets (CSS). You can define the CSS for a page using either the `StylesFile` property or the `Styles` string list. Any element of the editor of the producer's items can define a specific style or choose a style from the attached CSS. You accomplish this last operation (which is the suggested approach) using the `StyleRule` property.

## Scripts Rather Than Code?

Even this example of the combined use of an adapter and an adapter page producer, with its visual designer, shows the power of this architecture. However, this approach also has a drawback: By letting the components generate the script (in the HTML, you have only the `<#SERVERSCRIPT>` tag), you save a lot of development time; but you end up mixing the script with the code, so changes to the user interface will require updating the program. The division of responsibilities between the Delphi application developer and the HTML/script designer is lost. And, ironically, you end up having to run a script to accomplish something the Delphi program could have done right away, possibly much faster.

In my opinion, `WebSnap` is a powerful architecture and a huge step forward from `WebBroker`, but it must be used with care to avoid misusing some of these technologies just because they are simple and powerful. For example, it might be worth using the `AdapterPageProducer` designer to generate the first version of a page, and then grabbing the generated script and copying it to a plain Page-Producer's HTML, so that a web designer can modify the script with a specific tool.

For nontrivial applications, I prefer the possibilities offered by XML and XSL, which are available within this architecture even if they don't have a central role. You'll find more on this topic in [Chapter 22](#).

## Locating Files

When you have written an application like the one just described, you must deploy it as a CGI or dynamic library. Instead of placing the templates and include files in the same folder as the executable file, you can devote a subfolder or custom folder to host all the files. The `LocateFileService` component handles this task.

The component is not intuitive to use. Instead of having you specify a target folder as a property, the system fires one of this component's events any time it has to locate a file. (This approach is much more powerful.) There are three events: `OnFindIncludeFile`, `OnFindStream`, and `OnFindTemplateFile`. The first and last events return the name of the file to use in a `var` parameter. The `OnFindStream` event allows you to provide a stream directly, using one you already have in memory or that you've created on the fly, extracted from a database, obtained via an HTTP connection, or gotten any other way you can think of. In the simplest case of the `OnFindIncludeFile` event, you can write code like the following:

```

procedure TPageProducerPage2.LocateFileService1FindIncludeFile(
    ASender: TObject; AComponent: TComponent; const AFileName: String;
    var AFoundFile: String; var AHandled: Boolean);
begin
    AFoundFile := DefaultFolder + AFileName;
    AHandled := True;
end;

```



## WebSnap and Databases

Delphi has always shone in the area of database programming. For this reason, it is not surprising to see a lot of support for handling datasets within the WebSnap framework. Specifically, you can use the `DataSetAdapter` component to connect to a dataset and display its values in a form or a table using the `AdapterPageProducer` component's visual editor.

### A WebSnap Data Module

As an example, I built a new WebSnap application (called `WSnapTable`) with an `AdapterPage-Producer` as its main page to display a table in a grid and another `AdapterPageProducer` in a secondary page to show a form with a single record. I also added to the application a `WebSnap Data Module`, as a container for the dataset components. The data module has a `ClientDataSet` that's wired to a `dbExpress` dataset through a provider and based on an `InterBase` connection, as shown here:

```
object ClientDataSet1: TClientDataSet
  Active = True
  ProviderName = 'DataSetProvider1'
end
object SQLConnection1: TSQLConnection
  Connected = True
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText =
    'select CUST_NO, CUSTOMER, ADDRESS_LINE1, CITY, STATE_PROVINCE, ' +
    ' COUNTRY from CUSTOMER'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
```

### The DataSetAdapter

Now that you have a dataset available, you can add a `DataSetAdapter` to the first page and connect it to the web module's `ClientDataSet`. The adapter automatically makes available all of the dataset's fields and several predefined actions for operating on the dataset (such as `Delete`, `Edit`, and `Apply`). You can add them explicitly to the `Actions` and `Fields` collections to exclude some of them and customize their behavior, but this step is not always required.

Like the `PagedAdapter`, the `DataSetAdapter` has a `PageSize` property you can use to indicate the number of elements to display in each page. The component also has commands you can use to navigate among pages. This approach is particularly suitable when you want to display a large dataset in a grid. These are the adapter settings for the main page of the `WSnapTable` example:

```
object DataSetAdapter1: TDataSetAdapter
  DataSet = WebDataModule1.ClientDataSet1
  PageSize = 6
end
```

The corresponding page producer has a form containing two command groups and a grid. The first command group (displayed above the grid) has the predefined commands for handling pages: `CmdPrevPage`, `CmdNextPage`, and `CmdGotoPage`. The last command generates a list of numbers for the pages, so that a user can jump to each of them directly. The `AdapterGrid` component has the default columns plus an extra column hosting `Edit` and `Delete`

commands. The bottom command group provides a button used to create a new record. You can see an example of the table's output in [Figure 20.10](#) and the complete settings of the AdapterPageProducer in [Listing 20.2](#).

The screenshot shows a web browser window with the URL `http://localhost:1024/wsnaptable-unsnaptable/`. The page title is "WSnapTable" and the content is labeled "table". Below the title, there are navigation links: "Previous Page", "1", "2", "3", and "Next Page". The main content is a table with the following data:

CUST_NO	CUSTOMER	ADDRESS_LINE1	CITY	STATE_PROVINCE	COUNTRY	COMMANDS
1001	Signature Design	15500 Pacific Heights Blvd.	San Diego	CA	USA	<a href="#">Edit</a> <a href="#">Delete</a>
1002	Dallas Technologies	P. O. Box 47000	Dallas	TX	USA	<a href="#">Edit</a> <a href="#">Delete</a>
1003	Furde, Griffith and Co.	2300 Newbury Street	Boston	MA	USA	<a href="#">Edit</a> <a href="#">Delete</a>
1004	Central Bank	66 Lloyd Street	Manchester		England	<a href="#">Edit</a> <a href="#">Delete</a>
1005	IT Systems, LTD	400 Connaught Road	Central Hong Kong		Hong Kong	<a href="#">Edit</a> <a href="#">Delete</a>
1005	DataServe International	2000 Carling Avenue	Ottawa	ON	Canada	<a href="#">Edit</a> <a href="#">Delete</a>

Below the table, there is a "New" button.

**Figure 20.10:** The page shown by the WSnapTable example at startup includes the initial portion of a paged table.

### **Listing 20.2: AdapterPageProducer Settings for the WSnapTable Main Page**

```

object AdapterPageProducer: TAdapterPageProducer
object AdapterForm1: TAdapterForm
object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterGrid1
object CmdPrevPage: TAdapterActionButton
    ActionName = 'PrevPage'
    Caption = 'Previous Page'
end
object CmdGotoPage: TAdapterActionButton...
object CmdNextPage: TAdapterActionButton
    ActionName = 'NextPage'
    Caption = 'Next Page'
end
end
object AdapterGrid1: TAdapterGrid
    TableAttributes.CellSpacing = 0
    TableAttributes.CellPadding = 3
    Adapter = DataSetAdapter1
    AdapterMode = 'Browse'
object ColCUST_NO: TAdapterDisplayColumn
    ...
object AdapterCommandColumn1: TAdapterCommandColumn
    Caption = 'COMMANDS'
object CmdEditRow: TAdapterActionButton
    ActionName = 'EditRow'
    Caption = 'Edit'
    PageName = 'formview'
    DisplayType = ctAnchor
end

```

```

    object CmdDeleteRow: TAdapterActionButton
        ActionName = 'DeleteRow'
        Caption = 'Delete'
        DisplayType = ctAnchor
    end
end
end
object AdapterCommandGroup2: TAdapterCommandGroup
    DisplayComponent = AdapterGrid1
    object CmdNewRow: TAdapterActionButton
        ActionName = 'NewRow'
        Caption = 'New'
        PageName = 'formview'
    end
end
end
end
end
end

```

---

There are a couple of things to notice in this listing. First, the grid's `AdapterMode` property is set to `Browse` (other possibilities are `Edit`, `Insert`, and `Query`). This dataset display mode for adapters determines the type of user interface (text or edit boxes and other input controls) and the visibility of other buttons (for example, `Apply` and `Cancel` buttons are only present in the edit view; the opposite is true for the `Edit` command).

**Note** You can also modify the adapter mode by using server-side script and accessing `Adapter.Mode`.

Second, I modified the display of the commands in the grid using the `ctAnchor` value for the `DisplayType` property instead of the default button style. Similar properties are available for most components of this architecture to tweak the HTML code they produce.

## Editing the Data in a Form

Some of the commands are connected to a different page, which will be displayed after the commands are invoked. For example, the `edit` command's `PageName` property is set to `formview`. This second page of the application has an `AdapterPageProducer` with components hooked to the same `DataSetAdapter` as of the other table, so that all the requests will be automatically synchronized. If you click the `Edit` command, the program will open the secondary page that displays the data of the record corresponding to the command.

[Listing 20.3](#) shows the details of the page producer for the program's second page. Again, building the HTML form visually using the Delphi-specific designer (see [Figure 20.11](#)) was a very fast operation.

### Listing 20.3: AdapterPageProducer Settings for the formview Page

---

```

object AdapterPageProducer: TAdapterPageProducer
    object AdapterForm1: TAdapterForm
        object AdapterErrorList1: TAdapterErrorList
            Adapter = table.DataSetAdapter1
        end
        object AdapterCommandGroup1: TAdapterCommandGroup
            DisplayComponent = AdapterFieldGroup1
            object CmdApply: TAdapterActionButton
                ActionName = 'Apply'
                PageName = 'table'
            end
            object CmdCancel: TAdapterActionButton

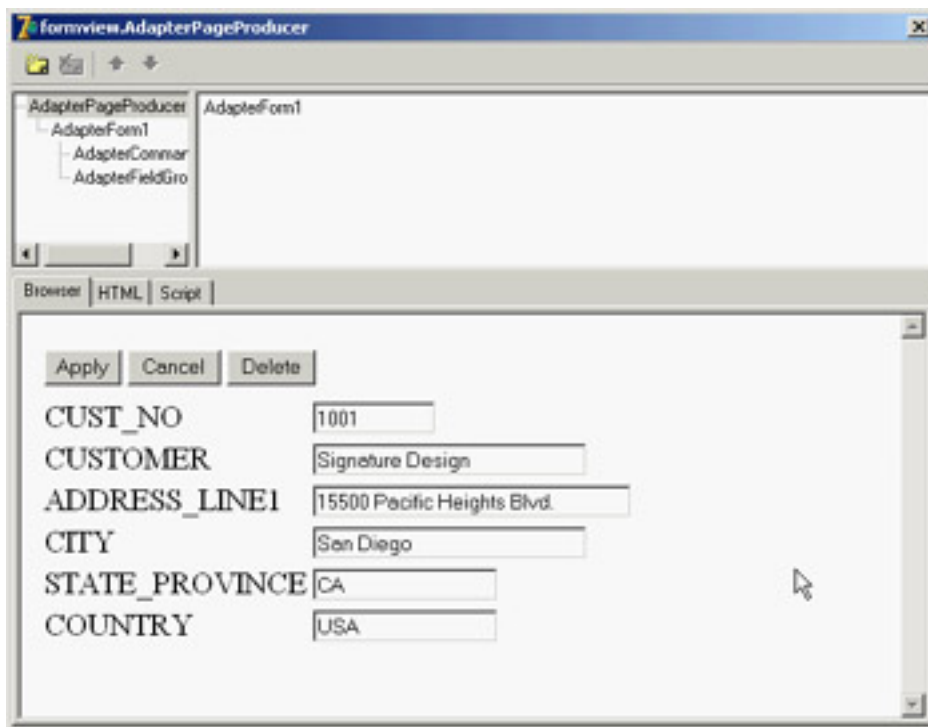
```

```

    ActionName = 'Cancel'
    PageName = 'table'
end
object CmdDeleteRow: TAdapterActionButton
    ActionName = 'DeleteRow'
    Caption = 'Delete'
    PageName = 'table'
end
end
object AdapterFieldGroup1: TAdapterFieldGroup
    Adapter = table.DataSetAdapter1
    AdapterMode = 'Edit'
    object FldCUST_NO: TAdapterDisplayField
        DisplayWidth = 10
        FieldName = 'CUST_NO'
    end
    object FldCUSTOMER: TAdapterDisplayField
        ...
    end
end
end
end
end

```

---



**Figure 20.11:** The formview page shown by the WSnapTable example at design time, in the Web Surface Designer (or AdapterPageProducer editor)

In the listing, you can see that all the operations send the user back to the main page and that the `AdapterMode` is set to `Edit`, unless there are update errors or conflicts. In this case, the same page is displayed again, with a description of the errors obtained by adding an `AdapterErrorList` component at the top of the form.

The second page is not published, because selecting it without referring to a specific record would make little sense. To unpublish the page, you comment the corresponding flag in the initialization code. Finally, to make the changes to the database persistent, you can call the `ApplyUpdates` method in the `OnAfterPost` and `OnAfterDelete` events of the `ClientDataSet` component hosted by the data module.

Another problem (which I haven't fixed) relates to the fact that the SQL server assigns the ID of each customer, so that when you enter a new record, the data in the ClientDataSet and in the database are no longer aligned. This situation can cause Record Not Found errors.

## Master/Detail in WebSnap

The DataSetAdapter component has specific support for master/detail relationships between datasets. After you've created the relationship among the datasets, as usual, define an adapter for each dataset and then connect the `MasterAdapter` property of the detail dataset's adapter. Setting up the master/detail relationship between the adapters makes them work in a more seamless way. For example, when you change the work mode of the master or enter new records, the detail automatically enters into Edit mode or is refreshed.

The WSnapMD example uses a data module to define such a relationship. It includes two ClientDataSet components, each connected to a SQLDataSet through a provider. The data access components each refer to a table, and the ClientDataSet components define a master/detail relationship. The same data module hosts two dataset adapters that refer to the two datasets and again define the master/detail relationship:

```
object dsaDepartment: TDataSetAdapter
  DataSet = cdsDepartment
end
object dsaEmployee: TDataSetAdapter
  DataSet = cdsEmployee
  MasterAdapter = dsaDepartment
end
```

**Warning** I originally tried to use a SimpleDataSet component to avoid cluttering the data module, but this approach didn't work. The master/detail portion of the program was correct, but moving from a page to the next or a previous page with the related buttons kept failing. The reason is that if you use a SimpleDataSet, a bug closes the dataset at each interaction, losing the status information.

This WebSnap application's only page has an AdapterPageProducer component hooked to both dataset adapters. The page's form has a field group hooked to the master and a grid connected with the detail. Unlike other examples, I tried to improve the user interface by adding custom attributes for the various elements. I used a gray background, displayed some of the grid borders (HTML grids are often used by Web Surface Designer), centered most of the elements, and added spacing. Notice that I added extra spaces to the button captions to prevent them from being too small. You can see the related code in the following detailed excerpt and the effect in [Figure 20.12](#):



**Figure 20.12:** The WSnapMD example shows a master/detail structure and has some customized output.

```

object AdapterPageProducer: TAdapterPageProducer
object AdapterForm1: TAdapterForm
    Custom = 'Border="1" CellSpacing="0" CellPadding="10" ' +
        'BgColor="Silver" align="center"'
object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterFieldGroup1
    Custom = 'Align="Center"'
object CmdFirstRow: TAdapterActionButton...
object CmdPrevRow: TAdapterActionButton...
object CmdNextRow: TAdapterActionButton...
object CmdLastRow: TAdapterActionButton...
end
object AdapterFieldGroup1: TAdapterFieldGroup
    Custom = 'BgColor="Silver"'
    Adapter = WDataMod.dsaDepartment
    AdapterMode = 'Browse'
end
object AdapterGrid1: TAdapterGrid
    TableAttributes.BgColor = 'Silver'
    TableAttributes.CellSpacing = 0
    TableAttributes.CellPadding = 3
    HeadingAttributes.BgColor = 'Gray'
    Adapter = WDataMod.dsaEmployee
    AdapterMode = 'Browse'
object ColEMP_NO: TAdapterDisplayColumn...
object ColFIRST_NAME: TAdapterDisplayColumn...
object ColLAST_NAME: TAdapterDisplayColumn...
object ColDEPT_NO: TAdapterDisplayColumn...
object ColJOB_CODE: TAdapterDisplayColumn...
object ColJOB_COUNTRY: TAdapterDisplayColumn...
object ColSALARY: TAdapterDisplayColumn...
end
end
end

```

## Sessions, Users, and Permissions

Another interesting feature of the WebSnap architecture is its support for sessions and users. Sessions are supported using a classic approach: temporary cookies. These cookies are sent to the browser so that following requests from the same user can be acknowledged by the system. By adding data to a session instead of an application adapter, you can have data that depends on the specific session or user (although a user can run multiple sessions by opening multiple browser windows on the same computer). For supporting sessions, the application keeps data in memory, so this feature is not available in CGI programs.

### Using Sessions

To underline the importance of this type of support, I built a WebSnap application with a single page showing both the total number of hits and the total number of hits for each session. The program has a `SessionService` component with default values for its `MaxSessions` and `DefaultTimeout` properties. For every new request, the program increases both an `nHits` private field of the page module and the `SessionHits` value of the current session:

```
procedure TSessionDemo.WebAppPageModuleBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
begin
  // increase application and session hits
  Inc (nHits);
  WebContext.Session.Values ['SessionHits'] :=
    Integer (WebContext.Session.Values ['SessionHits']) + 1;
end;
```

**Note** The `WebContext` object (of type `TWebContext`) is a thread variable created by WebSnap for each request. It provides thread-safe access to other global variables used by the program.

The associated HTML displays status information by using both custom tags evaluated by the `OnTag` event of the page producer and script evaluated by the engine. Here is the core portion of the HTML file:

```
<h3>Plain Tags</h3>
<p>Session id: <#SessionID>
<br>Session hits: <#SessionHits></p>
<h3>Script</h3>
<p>Session hits (via application): <%=Application.SessionHits.Value%>
<br>Application hits: <%=Application.Hits.Value%></p>
```

The parameters of the output are provided by the `OnTag` event handler and the fields' `OnGetValue` events:

```
procedure TSessionDemo.PageProducerHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'SessionID' then
    ReplaceText := WebContext.Session.SessionID
  else if TagString = 'SessionHits' then
    ReplaceText := WebContext.Session.Values ['SessionHits']
end;
```

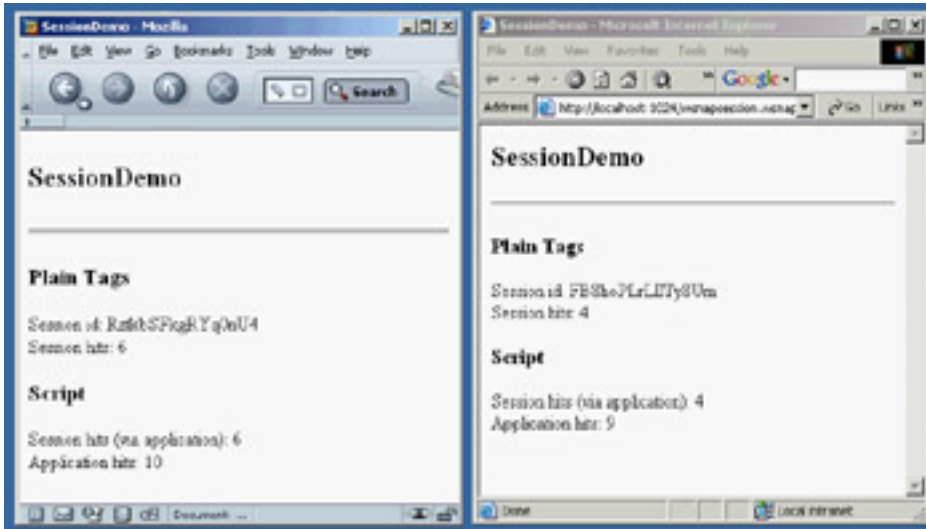
```
procedure TSessionDemo.HitsGetValue(Sender: TObject; var Value: Variant);
begin
  Value := nHits;
end;
```

```
procedure TSessionDemo.SessionHitsGetValue(Sender: TObject; var Value: Variant);
begin
```



```
Value := Integer (WebContext.Session.Values ['SessionHits']);  
end;
```

The effect of this program is visible in [Figure 20.13](#), where I've activated two sessions in two different browsers.



**Figure 20.13:** Two instances of the browser operate on two different sessions of the same WebSnap application.

**Tip** In this example, I used both the traditional WebBroker tag replacement and the newer WebSnap adapter fields and scripting, so that you can compare the two approaches. Keep in mind that they are both available in a WebSnap application.

## Requesting Login

In addition to generic sessions, WebSnap also has specific support for users and login-based authorized sessions. You can add to an application a list of users (with the WebUserList component), each with a name and a password. This component is rudimentary in the data it can store. However, instead of filling it with your list of users, you can keep the list in a database table (or in another proprietary format) and use the WebUserList component's events to retrieve your custom user data and check the user passwords.

You'll generally also add to the application the SessionService and EndUserSessionAdapter components. At this point, you can ask the users to log in, indicating for each page whether it can be viewed by everyone or only by logged-in users. This is accomplished by setting the wpLoginRequired flag in the constructor of the TWebPageModuleFactory and TWebAppPageModuleFactory classes in the web page unit's initialization code.

**Note** Rights and publication information is included in the factory rather than in the WebPageModule because the program can check the access rights and list the pages even without loading the module.

When a user tries to see a page that requires user identification, the login page indicated in the EndUserSessionAdapter component is displayed. You can create such a page easily by creating a new web page module based on an AdapterPageProducer and adding to it the LoginFormAdapter. In the page's editor, add a field group within a form, connect the field group to the LoginFormAdapter, and add a command group with the default Login button. The resulting login form will have fields for the username and its password, and also for the requested page. This last value is automatically filled with the requested page, in case this page required authorization and the user wasn't already logged in. This way, a user can immediately reach the requested page without being bounced back to a generic menu.

The login form is typically not published, because the corresponding Login command is already available when the user isn't logged in to the system; when the user logs in, it is replaced by a Logout command. This command is obtained by the standard script of the web page module, particularly the following:

```
<% if (EndUser.Logout != null) { %>
```



```
<% if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<% } %>
<% if (EndUser.Logout.Enabled) { %>
  <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<% } %>
<% if (EndUser.LoginForm.Enabled) { %>
  <a href="<%=EndUser.LoginForm.AsHREF%>">Login</a>
<% } %>
<% } %>
```

There isn't much else to say about the WSnapUsers application, because it has almost no custom code and settings. This script for the standard template demonstrates the access to the user data.

## Single Page Access Rights

In addition to making pages require a login for access, you can give specific users the right to see more pages than others. Any user has a set of rights separated by semicolons or commas. The user must have all the rights defined for the requested page. These rights, which are generally listed in the `ViewAccess` and `ModifyAccess` properties of the adapters, indicate respectively whether the user can see the given elements while browsing or can edit them. These settings are granular and can be applied to entire adapters or specific adapter fields (notice I'm referring to the adapter fields, not the user interface components within the designer). For example, you can hide some of a table's columns from given users by hiding the corresponding fields (and also in other cases, as specified by the `HideOptions` property).

The global `PageDispatcher` component also has `OnCanViewPage` and `OnPageAccessDenied` events, which you can use to control access to various pages of a program within the program code, allowing for even greater control.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

In this chapter, I've covered web server applications, using multiple server-side techniques and two different frameworks of the Delphi class library: WebBroker and WebSnap. This wasn't an in-depth presentation, because I could write an entire book on this topic alone. It was intended as a starting point, and (as usual) I've tried to make the core concepts clear rather than building complex examples.

If you want to learn more about the WebSnap framework and see examples in actions, refer to the extensive Delphi demos in the `\Demos\WebSnap` folder. Some other options, relating to XML, XSL, and client-side scripts, will be examined in the [Chapter 22](#).

[Chapter 21](#) is devoted to another alternative framework for developing web server applications in Delphi: IntraWeb. This third-party tool has been around for a few years, but it has now become more relevant because Borland included it in Delphi 7. As you'll see, IntraWeb can be integrated in a WebBroker or WebSnap program, or it can be used as a separate architecture.

## Chapter 21: Web Programming with IntraWeb

### Overview

Since Delphi 2 days, Chad Z. Hower has been building a Delphi architecture for simplifying the development of web applications, with the idea of making web programming as simple and as visual as standard Delphi form programming. Some programmers are fully acquainted with dynamic HTML, JavaScript, Cascading Style Sheets, and the latest Internet technologies. Other programmers just want to build web applications in Delphi the way they build VCL or CLX applications.

IntraWeb is intended for this second category of developers, although it is so powerful that even expert web programmers can benefit from its use. In Chad's words, IntraWeb is for building web applications, not websites. Moreover, IntraWeb components can be used in a specific application, or they can be used in a WebBroker or WebSnap program.

In this chapter I cannot cover every detail of IntraWeb—with 50 components installed on Delphi's palette and several module designers, it's a very large library. My plan is to cover its foundations, so that you can choose whether to use it for your forthcoming projects or for portions of those projects, as it fits best.

**Tip** For documentation on IntraWeb, refer to the PDF manuals on the Delphi 7 Companion CD. If you cannot find them, these manuals are also available on Atozed Software's website for download. For support on IntraWeb, refer to the Borland newsgroups.

**Note** This chapter has had a special review by Chad Z. Hower, a.k.a. "Kudzu," the original author and project coordinator for Internet Direct (Indy; see [Chapter 19](#), "Internet Programming: Sockets and Indy") and author of IntraWeb. Chad's areas of specialty include TCP/IP networking and programming, interprocess communication, distributed computing, Internet protocols, and object-oriented programming. When not programming, he likes to cycle, kayak, hike, downhill ski, drive, and do just about anything outdoors. Chad also posts free articles, programs, utilities, and other oddities at Kudzu World at <http://www.Hower.org/Kudzu/>. Chad is an American expatriate who currently spends his summers in St. Petersburg, Russia, and his winters in Limassol, Cyprus. Chad can be reached at [cpub@Hower.org](mailto:cpub@Hower.org).

## Introducing IntraWeb

IntraWeb is a component library currently produced by Atozed Software ([www.atozedsoftware.com](http://www.atozedsoftware.com)). In Delphi 7 Professional and Enterprise, you can find the corresponding version of IntraWeb. The professional version can be used only in *Page mode*, as you'll see later in this chapter. Although Delphi 7 is the first version of the Borland IDE to include this set of components, IntraWeb has been around for several years; it has received appraisal and support, including the availability of several third-party components.

**Tip** IntraWeb third-party components include IWChart by Steema (the makers of TeeChart), IWBold by Centillex (for integration with Bold), IWOpenSource, IWTranslator, IWDialogs, IWDataModulePool by Arcana, IW Component Pack by TMS, and IWGranPrimo by GranPrimo. You'll find an updated list of third parties on [www.atozedsoftware.com](http://www.atozedsoftware.com).

Although you don't have the source code for the core library (available on request and upon a specific payment), the IntraWeb architecture is fairly open, and the full source code of the components is readily available. IntraWeb is part of the Delphi standard installation now, but it is also available for Kylix. Written with care, IntraWeb applications can be fully cross-platform.

**Note** In addition to the Delphi and Linux versions, C++ Builder and Java versions of IntraWeb are available. A .NET version is in the works and will probably become available along with a future Delphi for .NET version.

As an owner of Delphi 7, you're entitled to receive the first significant update release (version 5.1) and can upgrade your license to a full IntraWeb Enterprise edition including upgrades and support from Atozed Software (see their website for details). More serious documentation (help and PDF files) will be available in this 5.1 upgrade.

## From Websites to Web Applications

As I mentioned earlier, the idea behind IntraWeb is to build web applications rather than websites. When you work with WebBroker or WebSnap (covered in [Chapter 20](#), "Web Programming with WebBroker and WebSnap"), you think in terms of web pages and page producers, and work closely at the HTML generation level. When you work with IntraWeb, you think in terms of components, their properties, and their event handlers, as you do in Delphi visual development.

For example, create a new IntraWeb application by selecting File → New → Other, moving to the IntraWeb page of the New Items dialog box, and choosing Stand Alone Application. In the following dialog box (which is part of Delphi, not of the IntraWeb wizard) you can choose an existing folder or enter a new one that will be created for you (I'm mentioning this because it is far from clear in the dialog). The resulting program has a project file and two different units (I'll cover this structure later).

For the moment, let's create an example (called IWSimpleApp in the book's source code). To build it, follow these steps:

1. Move to the main form of the program and add to it a button, an edit box, and a list box from the IW Standard page of the Components palette. That is, don't add the VCL components from the Standard page of the palette—instead, use the corresponding IntraWeb components: IWButton, IWEdit, and IWListBox.
2. Slightly modify their properties as follows:

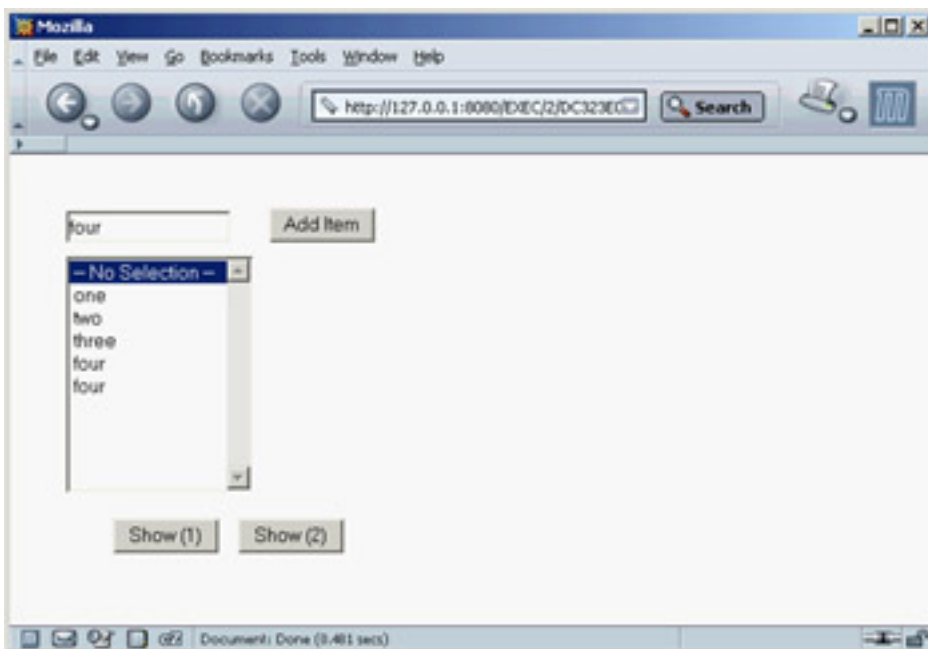
```
object IWButton1: TIWButton
  Caption = 'Add Item'
end
object IWEdit1: TIWEdit
  Text = 'four'
end
object IWListBox1: TIWListBox
  Items.Strings = (
    'one'
```

```
    'two'  
    'three' )  
end
```

3. Handle the `OnClick` event of the button by double-clicking on the component at design time as usual and writing this familiar code:

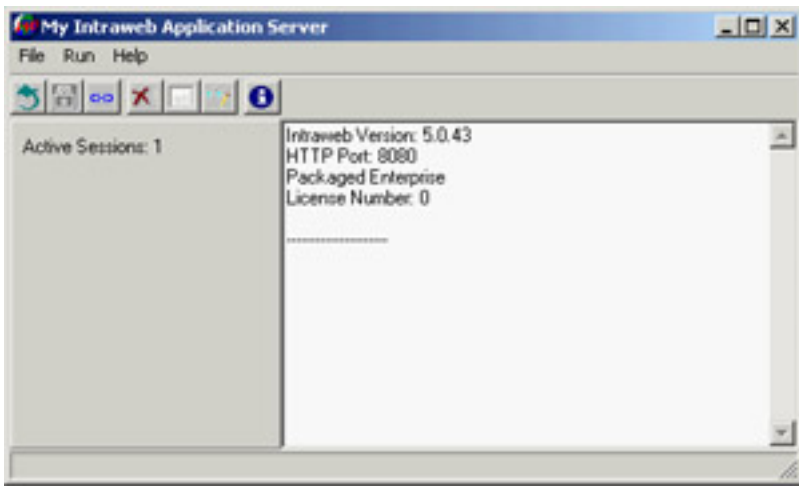
```
procedure TFormMain.IWButton1Click(Sender: TObject);  
begin  
    IWListBox1.Items.Add (IWEdit1.Text);  
end;
```

That's all it takes to create a web-based application capable of adding text to a list box, as you can see in [Figure 21.1](#) (which shows the final version of the program, with a couple more buttons). The important thing to notice when you run this program is that every time you click the button, the browser sends a new request to the application, which runs the Delphi event handler and produces a new HTML page based on the new status of the components on the form.



**Figure 21.1:** The IWSimpleApp program in a browser

As you execute the application, you won't see the program output in the browser, but rather IntraWeb's controller form, shown in [Figure 21.2](#). A stand-alone IntraWeb application, is a full-blown HTTP server, as you'll see in more detail in the [next section](#). The form you see is managed by the `IWRun` call in the project file created by default in each stand-alone IntraWeb application. The debug form allows you to select a browser and run the application through it or copy the URL of the application to the Clipboard, so you can paste it within your browser. It's important to know that the application will by default use a random port number, which is different for each execution; so, you'll have to use a different URL every time. You can modify this behavior by selecting the server controller's designer (similar to a data module) and setting the `port` property. In the example I've used 8080 (one of the common HTTP ports), but any other value will do.



**Figure 21.2:** The controller form of a stand-alone IntraWeb application

IntraWeb code is mainly server side, but IntraWeb also generates JavaScript to control some application features; you can also execute extra code on the client side. You do so by using specific client-side IntraWeb components or by writing your own custom JavaScript code. As a comparison, the two buttons at the bottom of the form in the IWSimpleApp example show a message box using two different approaches.

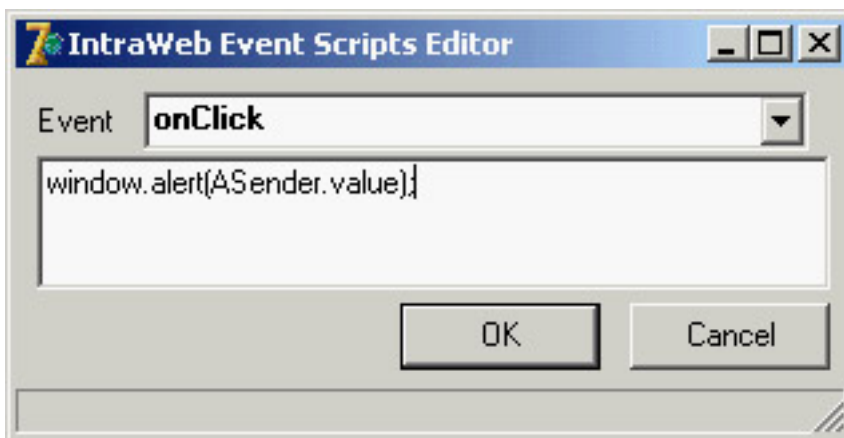
The first of these two buttons (IWButton2) shows a message using a server-side event, with this Delphi code:

```

procedure TFormMain.IWButton2Click(Sender: TObject);
var
    nItem: Integer;
begin
    nItem := IWListbox1.ItemIndex;
    if nItem >= 0 then
        WebApplication.ShowMessage (IWListBox1.Items [nItem])
    else
        WebApplication.ShowMessage ('No item selected');
end;

```

The second of these two buttons (IWButton3) uses JavaScript, which is inserted in the Delphi program by setting the proper JavaScript event handler in the special property editor for the ScriptEvents property:



## A First Look Behind the Scenes

You have seen that creating an IntraWeb application is as simple as creating a Delphi form-based program: You place components on a form and handle their events. Of course, the effect is rather different, because the application runs in a browser. To give you a feel for what's going on, let's briefly look behind the scenes of this simple program. Doing so should help you understand the effect of setting the component properties and of working with them in general.

This is a browser-based program, so there is no better way to understand it than by looking at the HTML it sends to the browser. Open the source of the IWSimpleApp program's page (not listed here, because it would waste too much book space), and you'll notice it is divided into three main sections. The first is a list of styles (based on the HTTP `style` tag) with lines like the following:

```
.IWEDIT1CSS {position:absolute;left:40;top:40;z-index:100;
font-style:normal;font-size:10pt;text-decoration:none;}
```

IntraWeb uses styles to determine not only the visual appearance of each component, such as its font and color, but also the component's position, using default absolute positioning. Each style is affected by a number of the IntraWeb component's properties, so you can easily experiment if you have some knowledge of style sheets. If you aren't familiar with style sheets, just use the properties and trust that IntraWeb will do its best to render the components in the web page.

The second block consists of JavaScript scripting. The main script block contains initialization code and the code of client-side event handlers for the components, like the following:

```
function IWBUTTON1_OnClick(ASender) {
    return SubmitClickConfirm('IWBUTTON1','', true, '');
}
```

This handler invokes the corresponding server-side code. If you've provided the JavaScript code directly in the IntraWeb application, as discussed earlier, you'll see this code:

```
function IWBUTTON3_onClick(ASender) {
    window.alert(ASender.value);
}
```

The scripting section of the page has also references to other files required by the browser and made available by IntraWeb. Some of these files are generic; others are tied to the specific browser: IntraWeb detects the browser being used and returns different JavaScript code and base JavaScript files.

**Note** Because JavaScript is not identical on all browsers, IntraWeb supports only some of them, including all recent versions of Microsoft Internet Explorer, Netscape Navigator, and the open-source Mozilla (which I've used while writing this chapter). Opera has more limited JavaScript support, so by default if it is recognized IntraWeb will issue an error (depending on the *SupportedBrowsers* property of the controller). Opera can be used with the free Arcana components and will officially be supported in IW 5.1. Keep in mind that a browser can fake its identity: For example, Opera is often set to identify itself as Internet Explorer, preventing a proper identification to make it possible to use sites restricted to other browsers, but possibly leading to run time errors or inconsistencies.

The third part of the generated HTML is the definition of the page structure. Inside the `body` tag is a `form` tag (on the same line) with the next action to be executed:

```
<form onsubmit="return FormDefaultSubmit();" name="SubmitForm"
action="/EXEC/3/DC323E01B09C83224E57E240" method="POST">
```

The `form` tag hosts the specific user interface components, such as buttons and edit boxes:

```
<input type="TEXT" name="IWEDIT1" size="17" value="four"
id="IWEDIT1" class="IWEDIT1CSS">
<input value="Add Item" name="IWBUTTON1" type="button"
onclick="return IWBUTTON1_OnClick(this);"
id="IWBUTTON1" class="IWBUTTON1CSS">
```

The form has also a few hidden components that IntraWeb uses to pass information back and forth. However, the URL is the most important way to pass information in IntraWeb; in the program it looks like this:

```
http://127.0.0.1:8080/EXEC/2/DC323E01B09C83224E57E240
```

The first part is the IP address and port used by the stand-alone IntraWeb application (it changes when you use a

different architecture to deploy the program), followed by the `EXEC` command, a progressive request number, and a session ID. We'll get back to session later on, but suffice to say that IntraWeb uses a URL token instead of cookies to make its applications available regardless of browser settings. If you prefer, you can use cookies instead of URLs by changing the `TrackMode` property in the server controller.

**Warning** The version of IntraWeb that shipped with Delphi 7 had a bug involving cookies and some time zone settings. It has been fixed, and the patch is available as a free update on the Atozed software website.

## IntraWeb Architectures

Before I write more examples to demonstrate the use of other IntraWeb components available in Delphi 7, let's discuss another key element of IntraWeb: the different architectures you can use to create and deploy applications based on this library. You can create IntraWeb projects in Application mode (which accounts for all of the features of IntraWeb) or Page mode (which is a simplified version you can plug into existing Delphi WebBroker or WebSnap applications). Application mode applications can be deployed as ISAPI libraries, Apache modules, or by using IntraWeb Standalone mode (a variation of the Application mode architecture). Page mode programs can be deployed as any other WebBroker application (ISAPI, Apache module, CGI, etc.). IntraWeb features three different but partially overlapping architectures:

**Standalone Mode** Provides you with a custom web server, as in the first example you built. This is handy for debugging the application (because you can run it from the Delphi IDE and place breakpoints anywhere in the code). You can also use Standalone mode to deploy applications on internal networks (intranets) and to let users work in offline mode on their own computers, with a web interface. If you run a stand-alone IntraWeb program with the `-install` flag, it will run as a service, and the dialog box won't appear. Standalone mode gives you a way to deploy an Application mode IntraWeb program using IntraWeb itself as web server.

**Application Mode** Allows you to deploy an IntraWeb application on a commercial server, building an Apache module or an IIS library. Application mode includes session management and all the IntraWeb features and is the preferred way to deploy a scalable application for use on the Web. To be precise, Application mode IntraWeb programs can be deployed as stand-alone programs, ISAPI libraries, or Apache modules.

**Page Mode** Opens a way to integrate IntraWeb pages in WebBroker and WebSnap applications. You can add features to existing programs or rely on other technologies for portions of a dynamic site based on HTML pages, while providing the interactive portions with IntraWeb. Page mode is the only choice for using IntraWeb in CGI applications, but it lacks session-management features. Stand-alone IntraWeb servers do not support Page mode.

In the examples in the rest of the chapter, I'll use Standalone mode for simplicity and easier debugging, but I'll also cover Page mode support.



## Building IntraWeb Applications

When you build an IntraWeb application, a number of components are available. For example, if you look at the IW Standard page of Delphi's Component Palette, you'll see an impressive list of core components, from the obvious button, check box, radio button, edit box, list box, memo, and so on to the intriguing tree view, menu, timer, grid, and link components. I won't list each component and describe its use with an example—I'd rather use some of the components in a few demos and underline the architecture of IntraWeb rather than specific details.

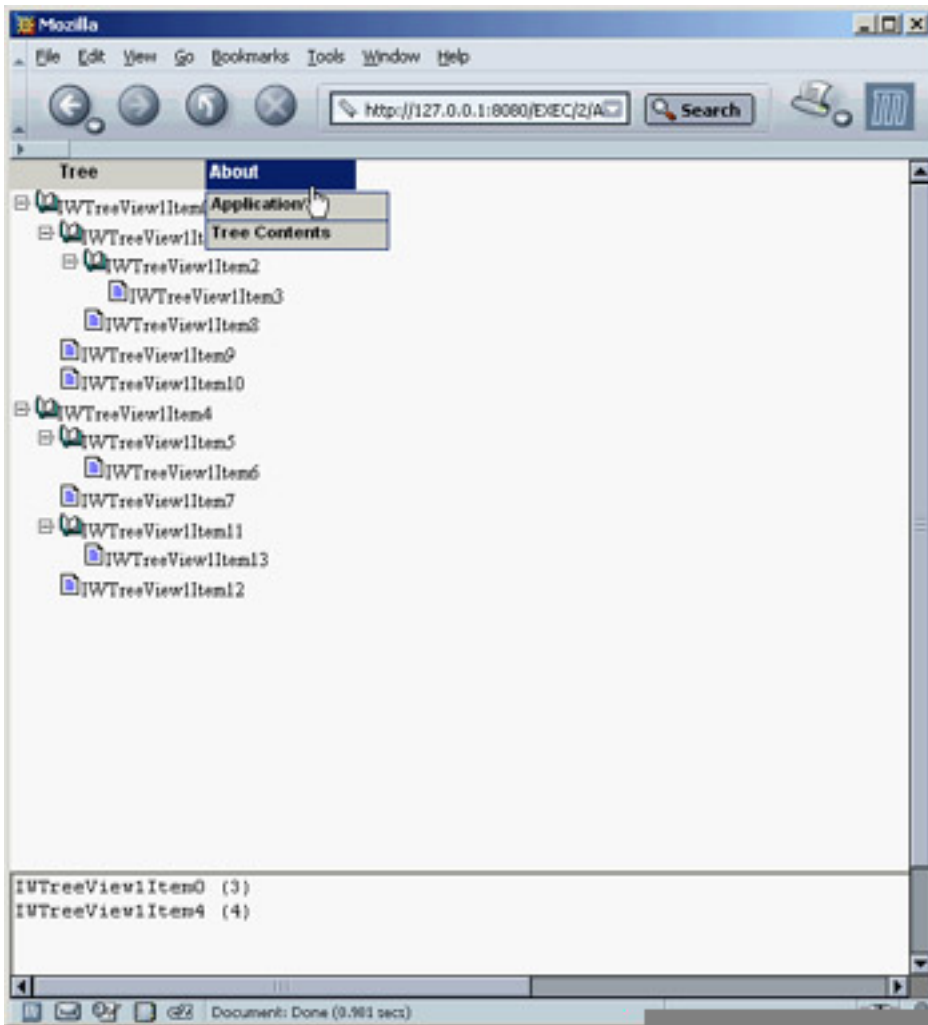
I've built an example (called IWTree) showcasing the menu and tree view components of IntraWeb but also featuring the creation of a component at run time. This handy component makes available in a dynamic menu the content of a standard Delphi menu, by referring its `AttachedMenu` property to a `TMenu` component:

```
object MainMenu1: TMainMenu
  object Tree1: TMenuItem
    object ExpandAll1: TMenuItem
    object CollapseAll1: TMenuItem
    object N1: TMenuItem
    object EnlargeFont1: TMenuItem
    object ReduceFont1: TMenuItem
  end
  object About1: TMenuItem
    object Application1: TMenuItem
    object TreeContents1: TMenuItem
  end
end
object IWMenu1: TIWMenu
  AttachedMenu = MainMenu1
  Orientation = iwOHorizontal
end
```

If the menu items handle the `OnClick` event in the code, they become links at run time. You can see an example of a menu in a browser in [Figure 21.3](#). The example's second component is a tree view with a set of predefined nodes. This component has a lot of JavaScript code to let you expand and collapse nodes directly in the browser (with no need to call back the server). At the same time, the menu items allow the program to operate on the menu by expanding or collapsing nodes and changing the font. Here is the code for a couple of event handlers:

```
procedure TFormTree.ExpandAll1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to IWTreeView1.Items.Count - 1 do
    IWTreeView1.Items [i].Expanded := True;
  end;

procedure TFormTree.EnlargeFont1Click(Sender: TObject);
begin
  IWTreeView1.Font.Size := IWTreeView1.Font.Size + 2;
end;
```



**Figure 21.3:** The IWTree example features a menu, a tree view, and the dynamic creation of a memo component.

Thanks to the similarity of IntraWeb components to standard Delphi VCL components, the code is easy to read and understand.

The menu has two submenus, which are slightly more complex. The first displays the application ID, which is an application execution/session ID. This identifier is available in the `AppID` property of the `WebApplication` global object. The second submenu, Tree Contents, shows a list of the first tree nodes of the main level along with the number of direct subnodes. What's interesting, though, is that the information is displayed in a memo component created at run time (see again [Figure 21.3](#)), exactly as you'll do in a VCL application:

```

procedure TFormTree.TreeContents1Click(Sender: TObject);
var
    i: Integer;
begin
    with TIWMemo.Create(Self) do
    begin
        Parent := Self;
        Align := alBottom;
        for i := 0 to IWTreeView1.Items.Count - 1 do
            Lines.Add (IWTreeView1.Items [i].Caption + ' (' +
                IntToStr (IWTreeView1.Items [i].SubItems.Count) + ')');
    end;
end;

```

**Tip** Notice that alignment in IntraWeb works similarly to its VCL counterpart. For example, this program's menu with alTop alignment, the tree view has alClient alignment, and the dynamic memo is created with alBottom alignment. As an alternative, you can use anchors (again working as in the VCL): You can create bottom-right buttons, or components in the middle of the page, with all four anchors set. See the following demos for examples of this technique.

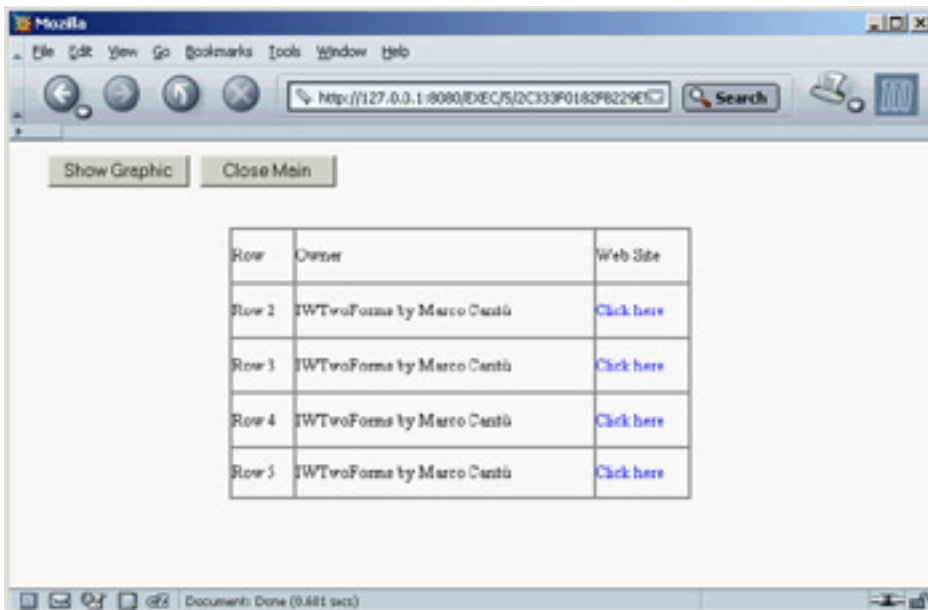
## Writing Multipage Applications

All the programs you have built so far have had a single page. Now let's create an IntraWeb application with a second page. As you'll see, even in this case, IntraWeb development resembles standard Delphi (or Kylix) development, and is different than most other Internet development libraries. This example will also serve as an excuse to delve into some of the source code automatically generated by the IntraWeb application wizard.

Let's start from the beginning. The main form of the IWTwoForms example features an IntraWeb grid. This powerful component allows you to place within an HTML grid both text and other components. In the example, the grid content is determined at startup (in the OnCreate event handler of the main form):

```
procedure TFormMain.IWAppFormCreate(Sender: TObject);  
var  
    i: Integer;  
    link: TIWURL;  
begin  
    // set grid titles  
    IWGrid1.Cell[0, 0].Text := 'Row';  
    IWGrid1.Cell[0, 1].Text := 'Owner';  
    IWGrid1.Cell[0, 2].Text := 'Web Site';  
    // set grid contents  
    for i := 1 to IWGrid1.RowCount - 1 do  
    begin  
        IWGrid1.Cell [i,0].Text := 'Row ' + IntToStr (i+1);  
        IWGrid1.Cell [i,1].Text := 'IWTwoForms by Marco Cantù';  
        link := TIWURL.Create(Self);  
        link.Text := 'Click here';  
        link.URL := 'http://www.marcocantu.com';  
        IWGrid1.Cell [i,2].Control := link;  
    end;  
end;
```

The effect of this code is shown in [Figure 21.4](#). In addition to the output, there are a few interesting things to notice. First, the grid component uses Delphi anchors (all set to False) to generate code that keeps it centered in the page, even if a user resizes the browser window. Second, I've added an IWURL component to the third column, but you could add any other component (including buttons and edit boxes) to the grid.



**Figure 21.4:** The IWTwoForms example uses an IWGrid component, embedded text, and IWURL components.

The third and most important consideration is that an IWGrid is translated into an HTML grid, with or without a frame around it. Here is a snippet of the HTML generated for one of the grid rows:

```
<tr>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;">Row 2</font>
  </td>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;">IWTwoForms by Marco Cantù</font>
  </td>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;"></font>
    <a href="#" onclick="parent.LoadURL('http://www.marcocantu.com')"
      id="TIWURL1" name="TIWURL1"
      style="z-index:100;font-style:normal;font-size:10pt;text-decoration:none;">
      Click here</a>
  </td>
</tr>
```

**Tip** In the previous listing, notice that the linked URL is activated via JavaScript, not with a direct link. This happens because all actions in IntraWeb allow for extra client-side operations, such as validations, checks, and submits. For example, if you set the *Required* property for a component, if the field is empty the data won't be submitted, and you'll see a JavaScript error message (customizable by setting the descriptive *FriendlyName* property).

The core feature of the program is its ability to show a second page. To accomplish this, you first need to add a new IntraWeb page to the application, using the Application Form option on the IntraWeb page of Delphi's New Items dialog box (File → New → Other). Add to this page a few IntraWeb components, as usual, and then add to the main form a button or other control you'll use to show the secondary form (with the reference *anotherform* stored in a field of the main form):

```
procedure TFormMain.btnShowGraphicClick(Sender: TObject);
begin
  anotherform := TAnotherForm.Create(WebApplication);
  anotherform.Show;
end;
```

Even if the program calls the *Show* method, it can be considered like a *ShowModal* call, because IntraWeb considers visible pages as a stack. The last page displayed is on the top of the stack and is displayed in the browser. By closing

this page (hiding or destroying it), you re-display the previous page. In the program, the secondary pages closes itself by calling the `Release` method, which as in the VCL is the proper way to dispose of a currently executing form. You can also hide the secondary form and then display it again, to avoid re-creating it each time (particularly if doing so implies losing the user's editing operations).

**Warning** In the program I added a Close button to the main form. It should not call `Release`, but rather should invoke the `WebApplication` object's `Terminate` method, passing the output message, as in `WebApplication .Terminate('Goodbye!')`. The demo uses an alternative call: `TerminateAndRedirect`.

Now that you have seen how to create an IntraWeb application with two forms, let's briefly examine how IntraWeb creates the main form. The relevant code, generated by the IntraWeb wizard as you create a new program, is in the project file:

```
begin
  IWRun(TFormMain, TIWServerController);
```

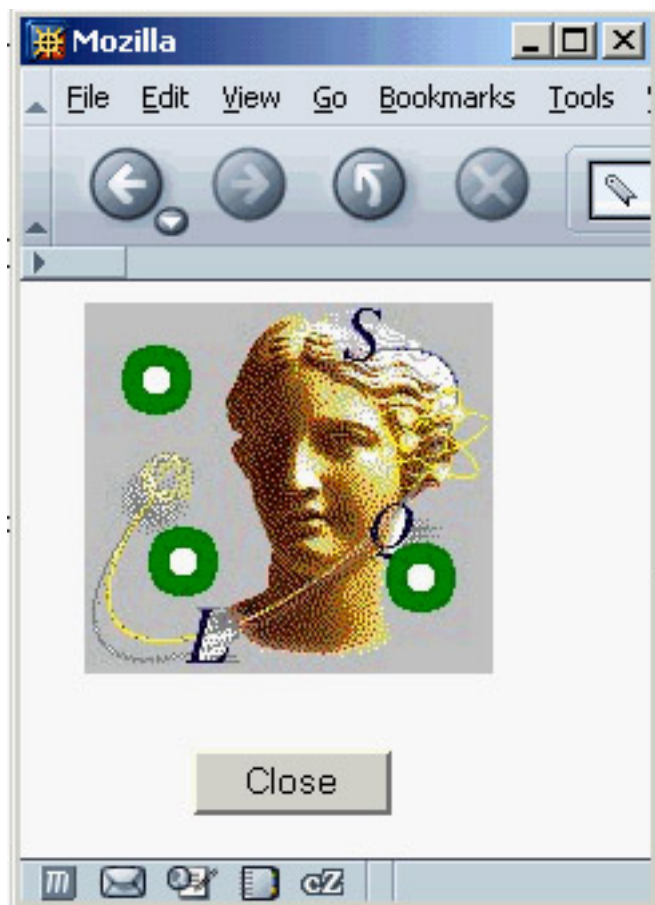
This is different from Delphi's standard project file, because it calls a global function instead of applying a method to a global object representing the application. The effect, though, is quite similar. The two parameters are the classes of the main form and of the IntraWeb controller, which handle sessions and other features as you'll see in a while.

The secondary form of the IWTwoForms example shows another interesting feature of IntraWeb: its extensive graphics support. The form has a graphical component with the classic Delphi Athena image. This is accomplished by loading a bitmap into the an `IWImage` component: IntraWeb converts the bitmap into a JPEG, stores it in a cache folder created under the application folder, and returns a reference to it, with the following HTML:

```

```

The extra feature provided by IntraWeb and exploited by the program is that a user can click on the image with the mouse to modify the image by launching server-side code. In this program, the effect is to draw small green circles.



This effect is obtained with the following code:

```

procedure Tanotherform.IWImageMouseDown(ASender: TObject;
  const AX, AY: Integer);
var
  aCanvas: TCanvas;
begin
  aCanvas := IWImage1.Picture.Bitmap.Canvas;
  aCanvas.Pen.Width := 8;
  aCanvas.Pen.Color := clGreen;
  aCanvas.Ellipse(AX - 10, AY - 10, AX + 10, AY + 10);
end;

```

**Warning** The painting operation takes place on the bitmap canvas. Do not try to use the Image canvas (as you can do with the VCL's TImage component), and do not try to use a JPEG in the first place, or you'll see either no effect or a run-time error.

## Sessions Management

If you've done any web programming, you know that session management is a complex issue. IntraWeb provides predefined session management and simplifies the way you work with sessions. If you need session data for a specific form, all you have to do is add a field to that form. The IntraWeb forms and their components have an instance for each user session. For example, in the IWSession example, I've added to the form a field called `FormCount`. As a contrast, I've also declared a global unit variable called `GlobalCount`, which is shared by all the instances (or sessions) of the application.

To increase your control over session data and let multiple forms share it, you can customize the `TUserSession` class that the IntraWeb Application Wizard places in the `ServerController` unit. In the IWSession example, I've customized the class as follows:

```

type
  TUserSession = class
  public
    UserCount: Integer;
  end;

```

IntraWeb creates an instance of this object for each new session, as you can see in the `IWServerControllerBaseNewSession` method of the `TIWServerController` class in the default `ServerController` unit:

```

procedure TIWServerController.IWServerControllerBaseNewSession(
  ASession: TIWApplication; var VMainForm: TIWAppForm);
begin
  ASession.Data := TUserSession.Create;
end;

```

In an application's code, the session object can be referenced by accessing the `Data` field of the `RWebApplication` global variable, used to access the current user's session.

**Note** `RWebApplication` is a *threadvar* variable, defined in the `IWInit` unit. It gives you access to the session data in a thread-safe way: you need to take special care to access it even in a multi-threading environment. This variable can be used outside of a form or control (which are natively session-based), which is why it is primarily used inside data modules, global routines, and non-IntraWeb classes.

Again, the default `ServerController` unit provides a helper function you can use:

```

function UserSession: TUserSession;
begin
  Result := TUserSession(RWebApplication.Data);
end;

```

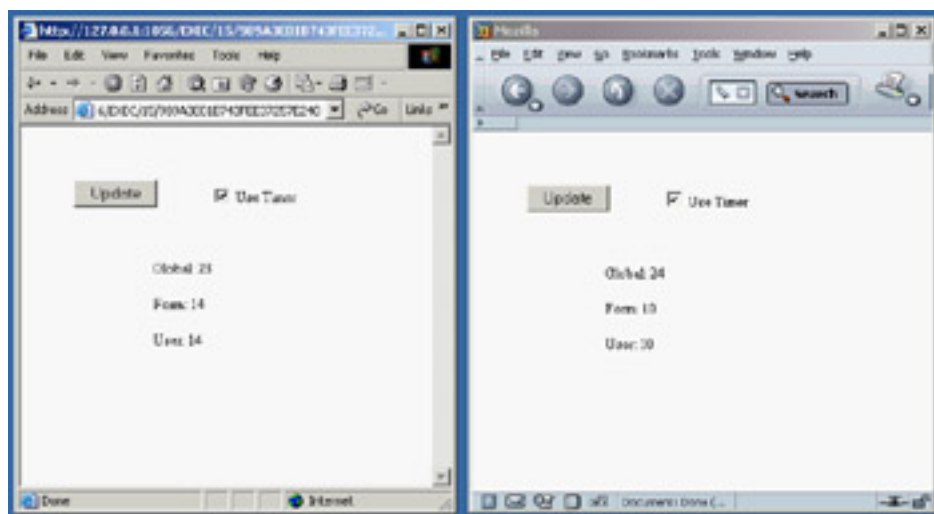


Because most of this code is generated for you, after adding data to the `TUserSession` class you simply use it through the `UserSession` function, as in the following code extracted from the `IWSession` example. When you click a button, the program increases several counters (one global and two session-specific) and shows their values in labels:

```
procedure TFormMain.IWButton1Click(Sender: TObject);  
begin  
  InterlockedIncrement (GlobalCount);  
  Inc (FormCount);  
  
  Inc (UserSession.UserCount);  
  
  IWLabel1.Text := 'Global: ' + IntToStr (GlobalCount);  
  IWLabel2.Text := 'Form: ' + IntToStr (FormCount);  
  IWLabel3.Text := 'User: ' + IntToStr (UserSession.UserCount);  
end;
```

Notice that the program uses Windows' `InterlockedIncrement` call to avoid concurrent access to the global shared variable by multiple threads. Alternative approaches include using a critical section or Indy's `TidThreadSafeInteger` (found in the `IdThreadsafe` unit).

[Figure 21.5](#) shows the output of the program (with two sessions running in two different browsers). The program has also a check box that activates a timer. Odd as it sounds, in an IntraWeb application, timers work almost the same as in Windows. When the timer interval expires, code is executed. Over the Web, this means refreshing the page by triggering a refresh in the JavaScript code:



**Figure 21.5:** The `IWSession` application has both session-specific and global counters, as you can see by running two sessions in two different browsers (or even in the same browser).

```
IWTIMER1=setTimeout('SubmitClick("IWTIMER1","", false)',5000);
```

## Integrating with WebBroker (and WebSnap)

Up to now, you have built stand-alone IntraWeb applications. When you create an IntraWeb application in a library to be deployed on IIS or Apache, you are basically in the same situation. Things change considerably, however, if you want to use IntraWeb Page mode, which is integrated in an IntraWeb page in a WebBroker (or WebSnap) Delphi application.

The bridge between the two worlds is the `IWPageProducer` component. This component hooks to a WebBroker action like any other page producer component and has a special event you can use to create and return an IntraWeb form:

```
procedure TWebModule1.IWPageProducer1GetForm(ASender: TIWPageProducer;  
  AWebApplication: TIWApplication; var VForm: TIWPageForm);
```

```

begin
  VForm := TFormMain.Create(AWebApplication);
end;

```

With this single line of code (plus the addition of an `IWebModuleController` component in the web module), the WebBroker application can embed an IntraWeb page, as the `Cgilntra` program does. The `IWebModuleController` component provides *core services* for IntraWeb support. A component of this type must exist in every project for IntraWeb to work properly.

**Warning** The release that comes with Delphi 7 has a problem with Delphi's Web App Debugger and the `IWebModuleController` component. This issue has been fixed and is a free update.

Here is a summary of the DFM of the example program's web module:

```

object WebModule1: TWebModule1
  Actions = <
    item
      Default = True
      Name = 'WebActionItem1'
      PathInfo = '/show'
      OnAction = WebModule1WebActionItem1Action
    end
    item
      Name = 'WebActionItem2'
      PathInfo = '/iwdemo'
      Producer = IWPageProducer1
    end>
  object IWebModuleController1: TIWebModuleController
  object IWPageProducer1: TIWPageProducer
    OnGetForm = IWPageProducer1GetForm
  end
end
end

```

Because this is a Page mode CGI application, it has no session management. Moreover, the status of the components in a page is not automatically updated by writing event handlers, as in a standard IntraWeb program. To accomplish the same effect you need to write specific code to handle further parameters of the HTTP request. It should be clear even from this simple example that Page mode does less for you than Application mode, but it's more flexible. In particular, IntraWeb Page mode allows you to add visual RAD design capabilities to your WebBroker and WebSnap applications.

## Controlling the Layout

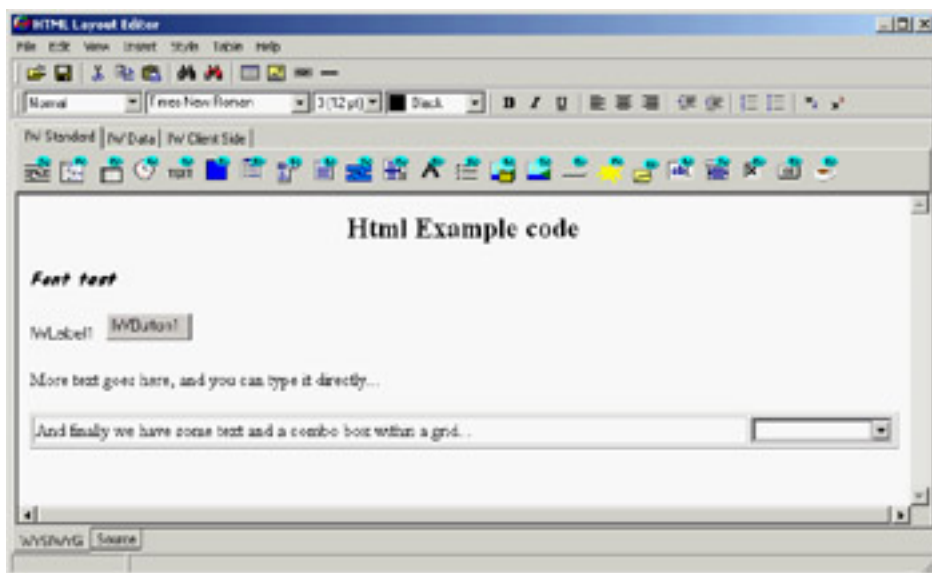
The `Cgilntra` program features another interesting technology available in IntraWeb: the definition of a custom layout based on HTML. (That topic isn't really related, because HTML layouts work also in Application mode—but I've happened to use these two techniques in a single example.) In the programs built so far, the resulting page is the mapping of a series of components placed at design time on a form, in which you can use properties to modify the resulting HTML. But what if you want to embed a data-entry form within a complex HTML page? Building the entire page contents with IntraWeb components is awkward, even if you can use the `IWText` component to embed a custom piece of HTML within an IntraWeb page.

The alternative approach is represented by the use of IntraWeb's layout managers. In IntraWeb you invariably use a layout manager; the default is the `IWLayoutMgrForm` component. The other two alternatives are `IWTemplateProcessorHTML` for working with an external HTML template file and `IWLayoutMgrHTML` for working with internal HTML.

This second component includes a powerful HTML editor you can use to prepare the generic HTML as well as embed the required IntraWeb components (something you'll have to do manually with an external HTML editor). Moreover, as you select an IntraWeb component from this editor (which is activated by double-clicking on an `IWLayoutMgrHTML`



component), you'll be able to use Delphi's Object Inspector to customize the component's properties. As you can see in [Figure 21.6](#), the HTML Layout Editor available in IntraWeb is a powerful visual HTML editor; the HTML text it generates is available in a separate page. (The HTML editor will be improved in a coming upgrade, and a few quirks will be fixed.)



**Figure 21.6:** IntraWeb's HTML Layout Editor is a full-blown visual HTML editor.

In the generated HTML, the HTML defines the structure of the page. The components are marked only with a special tag based on curly braces, as in the following of the example:

```
<P> { %IWLabel1% } { %IWButton1% } </P>
```

**Tip** Notice that when you're using HTML the components don't use absolute positioning but flow along with the HTML. Thus the form becomes only a component holder, because the size and position of the form's components are ignored.

Needless to say, the HTML you see in the visual designer of the HTML Layout Editor corresponds almost perfectly to the HTML you'll see when running the program in a browser.

## Web Database Applications

As in Delphi's libraries, a significant portion of IntraWeb's available controls relates to the development of database applications. The IntraWeb Application Wizard has a version that allows you to create an application with a data module—a good starting point for the development of a database application. In such a case, the application's predefined code creates an instance of the data module for each session, saving it in the session's data.

Here is the predefined `TUserSession` class (and its constructor) for an IntraWeb application with a data module:

```

type
  TUserSession = class(TComponent)
  public
    DataModule1: TDataModule1;
    constructor Create(AOwner: TComponent); override;
  end;

constructor TUserSession.Create(AOwner: TComponent);
begin
  inherited;
  DataModule1 := TDataModule1.Create(AOwner);
end;

```

The unit of the data module doesn't have a global variable for it; if it did, all the data would be shared among all sessions, with severe risks of trouble in case of concurrent requests in multiple threads. However, the data module already exposes a global function having the same name as the global variable Delphi would use, accessing the current session's data module:

```

function DataModule1: TDataModule1;
begin
  Result := TUserSession(RWebApplication.Data).DataModule1;
end;

```

This means you can write code like the following:

```
DataModule1.SimpleDataSet1
```

But instead of accessing a global data module, you are using the current session's data module.

In the first sample program featuring database data, called `IWScrollData`, I've added to the data module a `SimpleDataSet` component and to the main form an `IWDBGrid` component with the following configuration:

```

object IWDBGrid1: TIWDBGrid
  Anchors = [akLeft, akTop, akRight, akBottom]
  BorderSize = 1
  CellPadding = 0
  CellSpacing = 0
  Lines = tlRows
  UseFrame = False
  DataSource = DataSource1
  FromStart = False
  Options = [dgShowTitles]
  RowAlternateColor = clSilver
  RowLimit = 10
  RowCurrentColor = clTeal
end

```

The most important settings are the removal of a frame hosting the control with its own scroll bars (the `UseFrame` property), the fact that the data is displayed from the current data set position (the `FromStart` property), and the number of rows to be displayed in the browser (the `RowLimit` property). In the user interface, I've removed vertical

lines and colored alternate rows. I also had to set up a color for the current row (the `RowCurrentColor` property); otherwise the alternate colors won't appear to work properly, since the current row is the same color as the background rows, regardless of its position (set the `RowCurrentColor` property to `clNone` to see what I mean). These settings produce the effect you can see in [Figure 21.7](#) or by running the `IWScrollData` example.

EMP_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER	HIRE_DATE	DEPT_ID	JOB_CODE	JOB_TITLE	SALARY
24	Ford	Ford	917	11/09/90	60	Analyst	3	USA
25	Rogers	De Silva	288	11/09/90	60	Analyst	3	USA
26	Jones	Bell	3	08/09/90	110	Sales	3	USA
27	Rogers	Brown	4	02/09/90	120	Sales	3	England
28	Wales	Shaw	7	02/09/90	120	Analyst	4	England
41	Levy	Fung	219	09/09/91	60	Analyst	4	USA
42	Abel	Perkins	209	01/09/91	60	Analyst	3	USA
43	Walker	Shelton	210	08/09/91	90	CPA	1	USA
50	Grant	Hollman	430	08/09/90	130	Prod	4	USA

**Figure 21.7:** The data-aware grid of the `IWScrollData` example

The program opens the data set when the form is created, using the data set hooked to the current data source:

```

procedure TFormMain.IWAppFormCreate(Sender: TObject);
begin
    DataSource1.DataSet.Open;
end;

```

The example's relevant code is in the button code, which can be used to move through the data showing the following page or returning to the previous one. Here is the code for one of the two methods (the other is omitted, because it's very similar):

```

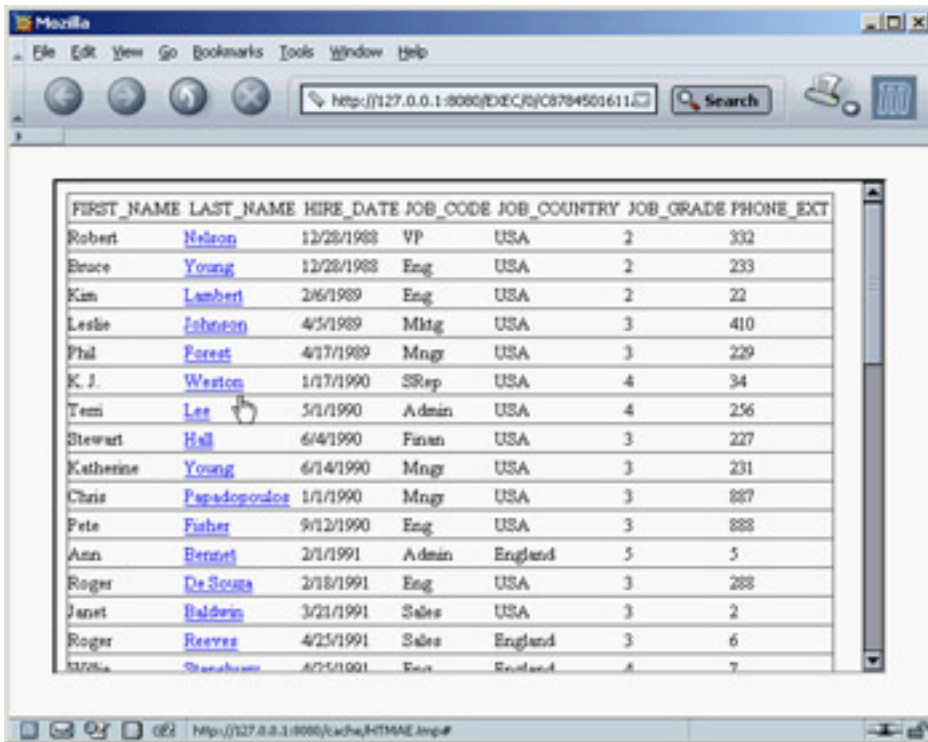
procedure TFormMain.btnNextClick(Sender: TObject);
var
    i: Integer;
begin
    nPos := nPos + 10;
    if nPos > DataSource1.DataSet.RecordCount - 10 then
        nPos := DataSource1.DataSet.RecordCount - 10;
    DataSource1.DataSet.First;
    for i := 0 to nPos do
        DataSource1.DataSet.Next;
end;

```

## Linking to Details

The grid of the `IWScrollData` example shows a single page of a table's data; buttons let you scroll up and down the pages. An alternative grid style in IntraWeb is offered by framed grids, which can move larger amounts of data to the web browser within a screen area of a fixed size using a frame and an internal scroll bar, as a Delphi `ScrollBox` control does. This is demonstrated by the `IWGridDemo` example.

The example customizes the grid in a second powerful way: It sets the `Columns` collection property of the grid. This setting allows you to fine-tune the output and behavior of specific columns, for example by showing hyperlinks or handling clicks on items or title cells. In the `IWGridDemo` example, one of the columns (the last name) is turned into a hyperlink; the employee number is passed as a parameter to the follow-up command, as you can see in [Figure 21.8](#).



**Figure 21.8:** The main form of the IWGridDemo example uses a framed grid with hyperlinks to the secondary form.

[Listing 21.1](#) shows a summary of the grid's key properties. Notice in particular the last name column, which as mentioned has a linked field (which turns the cell's text into a hyperlink) and an event handler responding to its selection. In this method, the program creates a secondary form in which a user can edit the data:

```

procedure TGridForm.IWDBGrid1Columns1Click(ASender: TObject;
  const AValue: String);
begin
  with TRecordForm.Create (WebApplication) do
    begin
      StartID := AValue;
      Show;
    end;
end;

```

### **Listing 21.1: Properties of the IWDBGrid in the IWGridDemo Example**

```

object IWDBGrid1: TIWDBGrid
  Anchors = [akLeft, akTop, akRight, akBottom]
  UseFrame = True
  UseWidth = True
  Columns = <
    item
      Alignment = taLeftJustify
      BGColor = clNone
      DoSubmitValidation = True
      Font.Color = clNone
      Font.Enabled = True
      Font.Size = 10
      Font.Style = []
      Header = False
      Height = '0'
      VAlign = vaMiddle
      Visible = True
      Width = '0'

```

```

Wrap = False
BlobCharLimit = 0
CompareHighlight = hcNone
DataField = 'FIRST_NAME'
Title.Alignment = taCenter
Title.BGColor = clNone
Title.DoSubmitValidation = True
Title.Font.Color = clNone
Title.Font.Enabled = True
Title.Font.Size = 10
Title.Font.Style = []
Title.Header = False
Title.Height = '0'
Title.Text = 'FIRST_NAME'
Title.VAlign = vaMiddle
Title.Visible = True
Title.Width = '0'
Title.Wrap = False
end
item
  DataField = 'LAST_NAME'
  LinkField = 'EMP_NO'
  OnClick = IWDBGrid1Columns1Click
end
item
  DataField = 'HIRE_DATE'
end
item
  DataField = 'JOB_CODE'
end
item
  DataField = 'JOB_COUNTRY'
end
item
  DataField = 'JOB_GRADE'
end
item
  DataField = 'PHONE_EXT'
end>
DataSource = DataSource1
Options = [dgShowTitles]
end

```

---

By setting the second form's `StartID` property, you can locate the proper record:

```

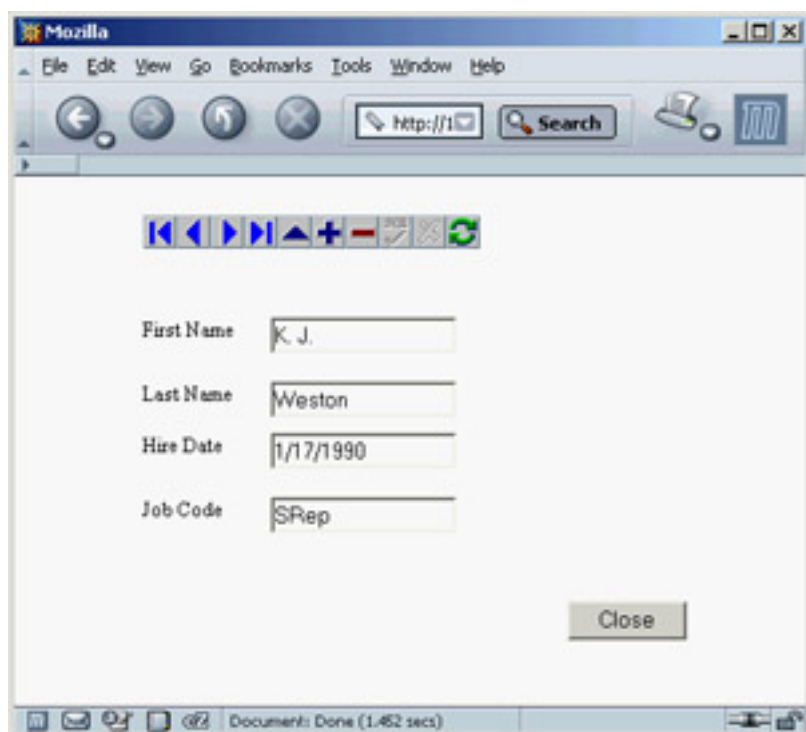
procedure TRecordForm.SetStartID(const Value: string);
begin
  FStartID := Value;
  DataSource1.DataSet.Locate('EMP_NO', Value, []);
end;

```

**Tip** The IWDBGrid columns have also an `OnTitleClick` event you can handle to sort the data or perform other operations on the column.

The secondary form is hooked to the same data module as the main form. So, after the database data is updated, you

can see it in the grid (but the updates are kept only in memory, because the program doesn't have an `ApplyUpdates` call). The secondary form uses a few edit controls and a navigator, provided by IntraWeb. You can see this form at run time in [Figure 21.9](#).



**Figure 21.9:** The secondary form of the IWGridDemo example allows a user to edit the data and navigate through records.

## Moving Data to the Client Side

Regardless of how you use it, the IWDBGrid component produces HTML with the database data embedded in the cells, but it cannot work on the data on the client side. A different component (or a set of IntraWeb components) allows you to follow a different model. The data is sent to the browser in a custom format, and the JavaScript code on the browser populates a grid and operates on the data, moving from record to record without asking more data to the server.

**Note** This architecture is similar to Delphi's native Internet Express architecture, which I'll cover in [Chapter 22](#) ("Using XML Technologies").

You can use several IntraWeb components for a client-side application, but these are the most important ones:

**IWClientSideDataSet** An in-memory dataset you define by setting the `ColumnNames` and `Data` properties within your program's code. In future updates, you will be able to edit client-side data, sort it, filter it, define master-detail structures, and more.

**IWClientSideDataSetDBLink** A data provider you can connect to any Delphi dataset, connecting it with the `DataSource` property.

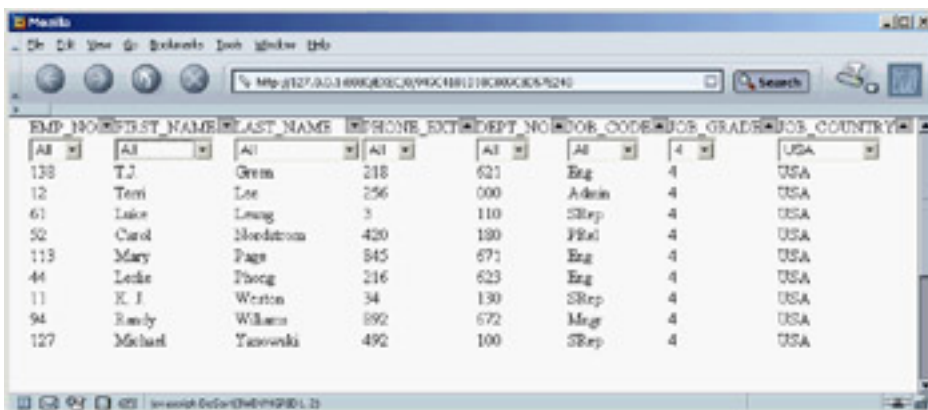
**IWDynGrid** A dynamic grid component connected to one of the two previous components using the `Data` property. This component moves all the data to the browser and can operate on it on the client via JavaScript.

There are other client-side components in IntraWeb, such as IWCSLabel, IWCSNavigator, and IWDynamicChart (which works only with Internet Explorer). As an example of using this approach, I've built the IWClientGrid example. The program has little code, because there is a lot available in the components being used. Here are the core

elements of its main form:

```
object formMain: TFormMain
  SupportedBrowsers = [brIE, brNetscape6]
  OnCreate = IWebAppFormCreate
  object IWDynGrid1: TIWDynGrid
    Align = alClient
    Data = IWClientSideDatasetDBLink1
  end
  object DataSource1: TDataSource
    Left = 72
    Top = 88
  end
  object IWClientSideDatasetDBLink1: TIWClientSideDatasetDBLink
    DataSource = DataSource1
  end
end
```

The dataset from the data module is hooked to the DataSource when the form is created. The resulting grid, shown in [Figure 21.10](#), allows you to sort the data on any cell (using the small arrow *after* the column title) and filter the data displayed on one of the field's possible values. In the figure, for example, you can sort the employee data by last name and filter it by country and job grade.



EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	DEPT_NO	JOB_CODE	JOB_GRADE	JOB_COUNTRY
138	TJ	Green	218	621	Eng	4	USA
12	Toni	Lee	256	000	Admin	4	USA
61	Luise	Leung	3	110	SRep	4	USA
52	Carol	Nordstrom	420	190	Fin	4	USA
113	Mary	Page	345	671	Eng	4	USA
44	Lodie	Thong	216	623	Eng	4	USA
11	K. J.	Watson	34	130	SRep	4	USA
94	Randy	Williams	892	672	Mgr	4	USA
127	Michael	Yasvski	492	100	SRep	4	USA

**Figure 21.10:** The grid of the IWClientGrid example supports custom sorting and filtering without re-fetching the data on the web server.

This functionality is possible because the data is moved to the browser within the JavaScript code. Here is a snippet of one of the scripts embedded in the page's HTML:

```
<script language="Javascript1.2">
var IWDYNGRID1_TitleCaptions =
  [ "EMP_NO", "FIRST_NAME", "LAST_NAME", "PHONE_EXT",
    "DEPT_NO", "JOB_CODE", "JOB_GRADE", "JOB_COUNTRY" ];
var IWDYNGRID1_CellValues = new Array();
IWDYNGRID1_CellValues[0] = [ 2, 'Robert', 'Nelson', '332', '600', 'VP', 2, 'USA' ];
IWDYNGRID1_CellValues[1] = [ 4, 'Bruce', 'Young', '233', '621', 'Eng', 2, 'USA' ];
IWDYNGRID1_CellValues[2] = [ 5, 'Kim', 'Lambert', '22', '130', 'Eng', 2, 'USA' ];
IWDYNGRID1_CellValues[3] = [ 8, 'Leslie', 'Johnson', '410', '180', 'Mktg', 3, 'USA' ];
IWDYNGRID1_CellValues[4] = [ 9, 'Phil', 'Forest', '229', '622', 'Mgr', 3, 'USA' ];
```

**Note** The reason to use this JavaScript-based approach, instead of an XML-based approach featured by other similar technologies, is that only Internet Explorer has support for *XML data islands*. Mozilla and Netscape lack this feature and have limited XML support in general. Mimicking it in JavaScript, as Internet Explorer does, is very expensive at run time.

## What's Next?

This chapter's description of IntraWeb's features has been far from complete, but my aim was mainly to let you evaluate this technology so you can choose whether to use it in your forthcoming Delphi projects. IntraWeb is so powerful that you now have a good reason to build web applications in Delphi, instead of resorting to other development tools.

You can read much more about IntraWeb in the documentation, found on the Delphi Companion CD (not on the main CD). Delphi's default installation includes the IntraWeb demos, including the extensive Features example that shows at once most of the features of this component library. Also refer to the IntraWeb website ([www.atozedsoftware.com](http://www.atozedsoftware.com)) for updates and further documentation and examples.

In this book, we have another alternative web development approach to cover, based on XML and XSLT. [Chapter 22](#) is devoted to a complete roundup of XML-related technologies from the Delphi perspective. So, we'll have another chance to cover web development in Delphi, using one of the techniques I like best—but that's also one of the most complex.



## Chapter 22: Using XML Technologies

### Overview

Building applications for the Internet means using protocols and creating browser-based user interfaces, as you've done in the preceding two chapters, but also opens an opportunity for exchanging business documents electronically. The emerging standards for this type of activity center around the XML document format and include the SOAP transmission protocol, XML schemas for the validation of documents, and XSL for rendering documents as HTML.

In this chapter, I'll discuss the core XML technologies and the extensive support Delphi has offered for them since version 6. Because XML knowledge is far from widespread, I'll provide a little introduction about each technology, but you should refer to books specifically devoted to them to learn more. In [Chapter 23](#), I'll focus specifically on web services and SOAP.

## Introducing XML

Extensible Markup Language (XML) is a simplified version of SGML and is getting a lot of attention in the IT world. XML is a *markup language*, meaning it uses symbols to describe its own content—in this case, *tags* consisting of specially defined text enclosed in angle brackets. It is *extensible* because it allows for free markers (in contrast, for example, to HTML, which has predefined markers). The XML language is a standard promoted by the World Wide Web Consortium (W3C). The XML Recommendation is at [www.w3.org/TR/REC-xml](http://www.w3.org/TR/REC-xml).

XML has been touted as the ASCII of the year 2000, to indicate a simple and widespread technology and also to indicate that an XML document is a plain-text file (optionally with Unicode characters instead of plain ASCII text). The important characteristic of XML is that it is descriptive, because every tag has an almost human-readable name. Here is an example, in case you've never seen an XML document:

```
<book>
  <title>Mastering Delphi 7</title>
  <author>Cantu</author>
  <publisher>Sybex</publisher>
</book>
```

XML has a few disadvantages I want to underline from the beginning. The biggest is that without a formal description, a document is worth little. If you want to exchange documents with another company, you must agree on what each tag means and also on the semantic meaning of the content. (For example, when you have a quantity, you have to agree on the measurement system or include it in the document.) Another disadvantage is that XML documents are much larger than other formats; using strings for numbers, for example, is far from efficient, and the repeated opening and closing tags eat up a lot of space. The good news is that XML compresses well, for the same reason.

## Core XML Syntax

A few technical elements of XML are worth knowing before we discuss its usage in Delphi. Here is a short summary of the key elements of the XML syntax:

- White space (including the space character, carriage return, line feed, and tabs) is generally ignored (as in an HTML document). It is important to format an XML document to make it readable by a human being, but your programs won't care much.
- You can add comments within `<!--` and `-->` markers, which are basically ignored by XML processors. There are also directives and processing instructions, enclosed within `<?>` and `?>` markers.
- There are a few special or reserved characters you cannot use in the text. The only two symbols you can *never* use are the less-than character (or left angle bracket, `<`, used to delimit a marker), which is replaced by `&lt;`, and the ampersand character (`&`), which is replaced by `&amp;`. Other optional special characters are `&gt;` for the greater-than symbol (right angle bracket, `>`), `&apos;` for the single quote (`'`), and `&quot;` for the double quote (`"`).
- To add non-XML content (for example, binary information or a script), you can use a CDATA section, enclosed within `<![CDATA[` and `]]>`.
- All markers are enclosed by angle brackets, `<` and `>`. Markers are case sensitive (in contrast to HTML).
- For each opening marker, you must have a matching closing marker, denoted by an initial slash character:  
`<node>value</node>`
- Markers must not overlap—they must be *properly nested*, as in the first line here (the second line is not correct):  
`<node>xx <nested> yy</nested> </node> // OK`  
`<node>xx <nested> yy</node> </nested> // WRONG`

- If a marker has no content (but its presence is important), you can replace the opening and closing markers with a single marker that includes a final (*trailing*) slash: `<node/>`.

- Markers can have attributes, using multiple attribute names followed by a value enclosed within quotes:

```
<node attrib1="aaa">
```

- Any XML node can have multiple attributes, multiple embedded tags, and only one block of text, representing the value of the node. It is common practice for XML nodes to have either a textual value or embedded tags, and not both. Here is an example of the full syntax of a node:

```
<node attrib1="aaa" attrib2="bbb">
  value1
  <child1>
    value2
  </child1>
</node>
```

- A node can have multiple child nodes with the same tag (tags need not be unique). Attribute names are unique for each node.

## Well-Formed XML

The elements discussed in the [previous section](#) define the syntax of an XML document, but they are not enough. An XML document is considered syntactically correct, or *well formed*, if it follows a few extra rules. Notice that this type of check doesn't guarantee that the content of the document is meaningful—only that the tags are properly laid out.

Each document should have a prologue indicating that it is indeed an XML document, which version of XML it complies with, and possibly the type of character encoding. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Possible encodings include Unicode character sets (such as UTF-8, UTF-16, and UTF-32) and some ISO encodings (such as ISO-10646-xxx or ISO-8859-xxx). The prologue can also include external declarations, the schema used to validate the document, namespace declarations, an associated XSL file, and some internal entity declarations. Refer to XML documentation or books for more information about these topics.

An XML document is well formed if it has a prologue, has a proper syntax (see the rules in the [previous section](#)), and has a tree of nodes with a single root. Most tools (including Internet Explorer) check whether a document is well formed when loading it.

**Note** XML is more formal and precise than HTML. The W3C is working on an XHTML standard that will make HTML documents XML compliant, for better processing with XML tools. This implies many changes in a typical HTML document, such as avoiding attributes with no values, adding all the closing markers (as in `</p>` and `</li>`), adding the slash to stand-alone markers (as `<hr/>` and `<br/>`), proper nesting, and more. An HTML-to-XHTML converter called HTML Tidy is hosted by the W3C website at [www.w3.org/People/Raggett/tidy/](http://www.w3.org/People/Raggett/tidy/).

## Working with XML

To get acquainted with the format of XML, you can use one of the existing XML editors available on the market (including Delphi itself and Context, a programmer's editor written in Delphi). When you load an XML document into Internet Explorer, you'll see whether it is correct and, in this case, you'll see it within the browser in a tree-like structure. (At the time I'm writing this, other browsers have more limited XML support.)

To speed up this type of operation, I've built the simplest XML editor I could come up with—basically a memo with XML syntax-checking and a browser attached to it. The XmlEditOne example has a PageControl with three pages. The first page, Settings, hosts a couple of components in which you can insert the path and the name of the file you want to

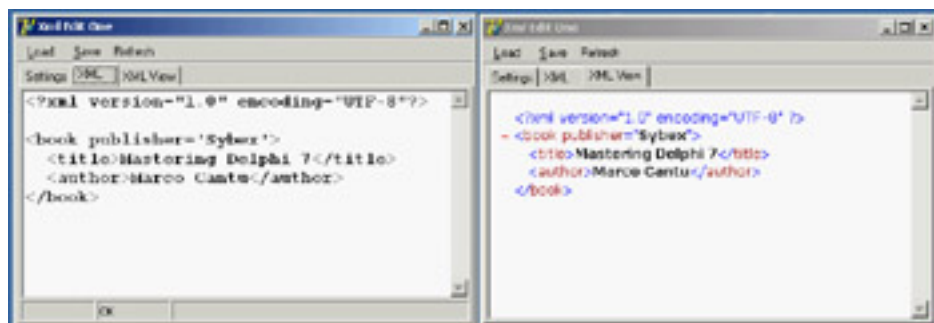
work with. (The reason for not using a standard dialog will become clear when I show you an extension of the program.) The edit box hosting the complete filename is automatically updated with the path and filename, provided the AutoUpdate check box is selected.

The second page hosts a Memo control; the text of the XML file is loaded and saved by clicking the two toolbar buttons. As soon as you load the file, or each time you modify its text, its content is loaded into a DOM to let a parser check for its correctness (something that would be complex to do with your own code). To parse the code, I've used the XMLDocument component available in Delphi, which is basically a wrapper around a DOM available on the computer and indicated by its `DOMVendor` property. I'll discuss the use of this component in more detail in the [next section](#). For the moment, suffice to say you can assign a string list to its `XML` property and activate it to let it parse the XML text and eventually report an error with an exception.

For this example, this behavior is far from good, because while typing the XML code you'll have temporarily incorrect XML. Still, I prefer not to ask the user to click a button to do the validation, but rather to let it run continuously. Because it is not possible to disable the parse exception raised by the XMLDocument component, I had to work at a lower level, extracting the `DOMPersist` property (referring to the persistency interface of the DOM) after extracting the `IXMLDocumentAccess` interface from the XMLDocument component (called `XmlDoc` in this code). You can also extract the `IDOMParseError` interface from the document component, to display any error message in the status bar:

```
procedure TFormXmlEdit.MemoXmlChange(Sender: TObject);  
var  
    eParse: IDOMParseError;  
begin  
    XmlDoc.Active := True;  
    xmlBar.Panels[1].Text := 'OK';  
    xmlBar.Panels[2].Text := '';  
    (XmlDoc as IXMLDocumentAccess).DOMPersist.loadxml(MemoXml.Text);  
    eParse := (XmlDoc.DOMDocument as IDOMParseError);  
    if eParse.errorCode <> 0 then  
        with eParse do  
            begin  
                xmlBar.Panels[1].Text := 'Error in: ' + IntToStr(Line) + '.' +  
                    IntToStr(LinePos);  
                xmlBar.Panels[2].Text := SrcText + ': ' + Reason;  
            end;  
    end;  
end;
```

You can see an example of the output of the program in [Figure 22.1](#), alongside the XML tree view provided by the third page (for a correct document). The third page of the program is built using the WebBrowser component, which embeds Internet Explorer's ActiveX control. Unfortunately, there is no direct way to assign a string with the XML text to this control, so you'll have to save the file first and then move to its page to trigger the loading of the XML in the browser (after manually clicking the Refresh button at least once).



**Figure 22.1:** The XmlEditOne example allows you to enter XML text in a memo, indicating errors as you type, and shows the result in the embedded browser.

**Note** I've used this code as a starting point to build a full-fledged XML editor called XmiTypist for a company I work with. It includes syntax highlighting, XSLT support, and a number of extra features. Refer to [Appendix A](#), "Extra Delphi Tools by the Author" for the availability of this free XML editor.

## Managing XML Documents in Delphi

Now that you know the core elements of XML, we can begin discussing how to manage XML documents in Delphi programs (or in programs in general; some of the techniques discussed here go beyond the language used). There are two typical techniques for manipulating XML documents: using a Document Object Model (DOM) interface or using the Simple API for XML (SAX). The two approaches are quite different:

- The DOM loads the entire document into a hierarchical tree of nodes, allowing you to read them and manipulate them to change the document. For this reason, the DOM is suitable when you want to navigate the XML structure in memory and edit it, or for creating new documents from scratch.
- The SAX parses the document, firing an event for each element of the document without building a structure in memory. Once parsed by the SAX, the document is lost, but this operation is generally much faster than the construction of the DOM tree. Using the SAX is good for reading a document once—for example, if you're looking for a portion of its data.

There is a third classic way to manipulate (and specifically create) XML documents: string management. Creating a document by adding strings is the fastest operation, particularly if you can do a single pass (and don't need to modify nodes already generated). Even reading documents by means of string functions is very fast, but this process can become difficult for complex structures.

Besides these classic XML processing approaches, which are also available for other programming languages, Delphi provides two more techniques you should consider. The first is the definition of interfaces that map the document structure and are used to access the document instead of the generic DOM interface. As you'll see, this approach makes for faster coding and more robust applications. The second technique is the development of transformations that allow you to read a generic XML document into a ClientDataSet component or save the dataset into an XML file of a given structure (not the specific XML structure natively supported by the ClientDataSet or MyBase).

I won't try to fully assess which option is better suited for each type of document and manipulation, but I will highlight some of the advantages and disadvantages while discussing examples of each approach in the following sections. At the end of the chapter, I'll discuss the relative speed of techniques for processing large files.

## Programming with the DOM

Because an XML document has a tree-like structure, loading an XML document into a tree in memory is a natural fit. This is what the DOM does. The DOM is a standard interface, so when you have written code that uses a DOM, you can switch DOM implementations without changing your source code (at least, if you haven't used any non-custom extensions).

In Delphi, you can install several DOM implementations, available as COM servers, and use their interfaces. One of the most commonly used DOM engines on Windows is the one provided by Microsoft as part of the MSXML SDK but also installed by Internet Explorer (and for this reason in all recent versions of Windows) and many other Microsoft applications. (With the full MSXML SDK also containing documentation and examples you don't get in other embedded installations of the same library.) Other DOM engines directly available in Delphi 7 include Apache Foundation's Xerces and the open-source OpenXML.

**Tip** OpenXML is a native Object Pascal DOM available at [www.philo.de/xml](http://www.philo.de/xml). Another native Delphi DOM is offered by TurboPower. These solutions offer two advantages: They don't require an external library for the program to execute, because the DOM component is compiled into your application; and they are cross-platform.

Delphi embeds the DOM implementations into a wrapper component called XMLDocument. I used this component in the preceding example, but here I will examine its role in a more general way. The idea behind using this component instead of the DOM interface is that you remain more independent from the implementations and can work with some simplified methods, or helpers.

The DOM interface is complex to use. A *document* is a collection of nodes, each having a name, a text element, a collection of attributes, and a collection of child nodes. Each collection of nodes lets you access elements by position or search for them by name. Notice that the text within the tags of a node, if any, is rendered as a child of the node and listed in its collection of child nodes. The root node has some extra methods for creating new nodes, values, or attributes.

With Delphi's XMLDocument, you can work at two levels:

- At a lower level, you can use the `DOMDocument` property (of the `IDOMDocument` interface type) to access a standard W3C Document Object Model interface. The official DOM is defined in the `xmlDom` unit and includes interfaces like `IDOMNode`, `IDOMNodeList`, `IDOMAttr`, `IDOMElement`, and `IDOMText`. With the official DOM interfaces, Delphi supports a lower-level but standard programming model. The DOM implementation is indicated by the XMLDocument component in the `DOMVendor` property.
- As a higher-level alternative, the XMLDocument component also implements the `IXMLDocument` interface. This is a custom DOM-like API defined by Borland in the `XMLIntf` unit and comprising interfaces like `IXMLNode`, `IXMLNodeList`, and `IXMLNodeCollection`. This Borland interface simplifies some of the DOM operations by replacing multiple method calls, which are repeated often in sequence, with a single property or method.

In the following examples (particularly the `DomCreate` demo), I'll use both approaches to give you a better idea of the practical differences between the two approaches.

### An XML Document in a TreeView

The starting point generally consists of loading a document from a file or creating it from a string, but you can also start with a new document. As a first example of using the DOM, I've built a program that can load an XML document into a DOM and show its structure in a TreeView control. I've also added to the `XmlDomTree` program a few buttons with sample code used to access to the elements of a sample file, as an example of accessing the DOM data. Loading the document is simple, but showing it in a tree requires a recursive function that navigates the nodes and subnodes. Here is the code for the two methods:

```
procedure TFormXmlTree.btnLoadClick(Sender: TObject);
```

```

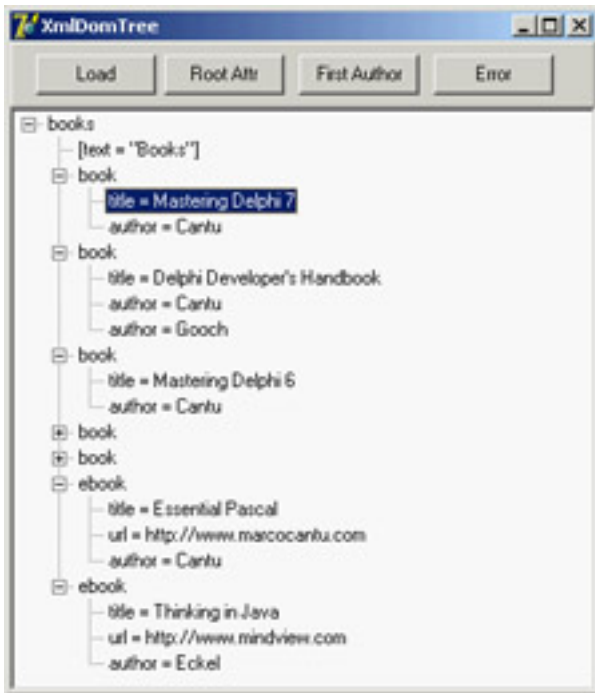
begin
  OpenFileDialog1.InitialDir := ExtractFilePath (Application.ExeName);
  if OpenFileDialog1.Execute then
    begin
      XMLDocument1.LoadFromFile(OpenDialog1.FileName);
      Treeview1.Items.Clear;
      DomToTree (XMLDocument1.DocumentElement, nil);
      TreeView1.FullExpand;
    end;
  end;

procedure TFormXmlTree.DomToTree (XmlNode: IXMLNode; TreeNode: TTreeNode);
var
  I: Integer;
  NewTreeNode: TTreeNode;
  NodeText: string;
  AttrNode: IXMLNode;
begin
  // skip text nodes and other special cases
  if XmlNode.NodeType <> ntElement then
    Exit;
  // add the node itself
  NodeText := XmlNode.NodeName;
  if XmlNode.IsTextElement then
    NodeText := NodeText + ' = ' + XmlNode.NodeValue;
  NewTreeNode := TreeView1.Items.AddChild(TreeNode, NodeText);
  // add attributes
  for I := 0 to XmlNode.AttributeNodes.Count - 1 do
    begin
      AttrNode := XmlNode.AttributeNodes.Nodes[I];
      TreeView1.Items.AddChild(NewTreeNode,
        '[' + AttrNode.NodeName + ' = "' + AttrNode.Text + '" ]');
    end;
  // add each child node
  if XmlNode.HasChildNodes then
    for I := 0 to XmlNode.ChildNodes.Count - 1 do
      DomToTree (XmlNode.ChildNodes.Nodes [I], NewTreeNode);
    end;
end;

```

This code is interesting because it highlights some of the operations you can do with a DOM. First, each node has a `NodeType` property you can use to determine whether the node is an element, attribute, text node, or special entity (such as CDATA and others). Second, you cannot access the textual representation of the node (its `NodeValue`) unless it has a text element (notice that the text node will be skipped, as per the initial test). After displaying the name of the item, and then the text value if available, the program shows the content of each attribute directly and of each subnode by calling the `DomToTree` method recursively (see [Figure 22.2](#)).





**Figure 22.2:** The XmlDomTree example can open a generic XML document and show it inside a TreeView common control.

Once you have loaded the sample document that accompanies the XmlDomTree program (shown in [Listing 22.1](#)) into the XmlDocument component, you can use the various methods to access generic nodes, as in the previous tree-building code, or fetch specific elements. For example, you can grab the value of the attribute *text* of the root node by writing:

```
XMLDocument1.DocumentElement.Attributes ['text']
```

Notice that if there is no attribute called *text*, the call will fail with a generic error message, "Invalid variant type conversion," which helps neither you nor the end user to understand what's wrong. If you need to access to the first attribute of the root without knowing its name, you can use the following code:

```
XMLDocument1.DocumentElement.AttributeNodes.Nodes[0].NodeValue
```

To access the nodes, you use a similar technique, possibly taking advantage of the *ChildValues* array. This is a Delphi extension to the DOM, which allows you to pass as parameter either the name of the element or its numeric position:

```
XMLDocument1.DocumentElement.ChildNodes.Nodes[1].ChildValues['author']
```

This code gets the (first) author of the second book. You cannot use the *ChildValues['book']* expression, because there are multiple nodes with the same name under the root node.

### **Listing 22.1: The Sample XML Document Used by Examples in this Chapter**

```
<?xml version="1.0" encoding="UTF-8"?>
<books text="Books">
  <book>
    <title>Mastering Delphi 7</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi Developer's Handbook</title>
    <author>Cantu</author>
    <author>Gooch</author>
  </book>
  <book>
    <title>Delphi COM Programming</title>
```



```

    <author>Harmon</author>
</book>
<book>
    <title>Thinking in C++</title>
    <author>Eckel</author>
</book>
<ebook>
    <title>Essential Pascal</title>
    <url>http://www.marcocantu.com</url>
    <author>Cantu</author>
</ebook>
<ebook>
    <title>Thinking in Java</title>
    <url>http://www.mindview.com</url>
    <author>Eckel</author>
</ebook>
</books>

```

---

## Creating Documents Using the DOM

Although I mentioned earlier that you can create an XML document by chaining together strings, this technique is far from robust. Using a DOM to create a document ensures that the XML will be well formed. Also, if the DOM has a schema definition attached, you can validate the structure of the document while adding data to it.

To highlight different cases of document creation, I've built the DomCreate example. This program can create XML documents within the DOM, showing their text on a memo and optionally in a TreeView.

**Warning** The XmlDocument component uses the doAutoIndent option to improve the output of the XML text to the memo by formatting the XML in a slightly better way. You can choose the type of indentation by setting the *NodeIndentStr* property. To format generic XML text, you can also use the global *FormatXMLData* function using the default setting (two spaces) as indentation. Oddly, there doesn't seem a way to pass a different parameter to the function.

The Simple button on the form creates simple XML text using the low-level, official DOM interfaces. The program calls the document's *createElement* method for each node, adding them as children of other nodes:

```

procedure TForm1.btnSimpleClick(Sender: TObject);
var
    iXml: IDOMDocument;
    iRoot, iNode, iNode2, iChild, iAttribute: IDOMNode;
begin
    // empty the document
    XMLDoc.Active := False;
    XMLDoc.XML.Text := '';
    XMLDoc.Active := True;

    // root
    iXml := XmlDoc.DOMDocument;
    iRoot := iXml.appendChild (iXml.createElement ('xml'));
    // node "test"
    iNode := iRoot.appendChild (iXml.createElement ('test'));
    iNode.appendChild (iXml.createElement ('test2'));
    iChild := iNode.appendChild (iXml.createElement ('test3'));
    iChild.appendChild (iXml.createTextNode('simple value'));
    iNode.insertBefore (iXml.createElement ('test4'), iChild);

```

```

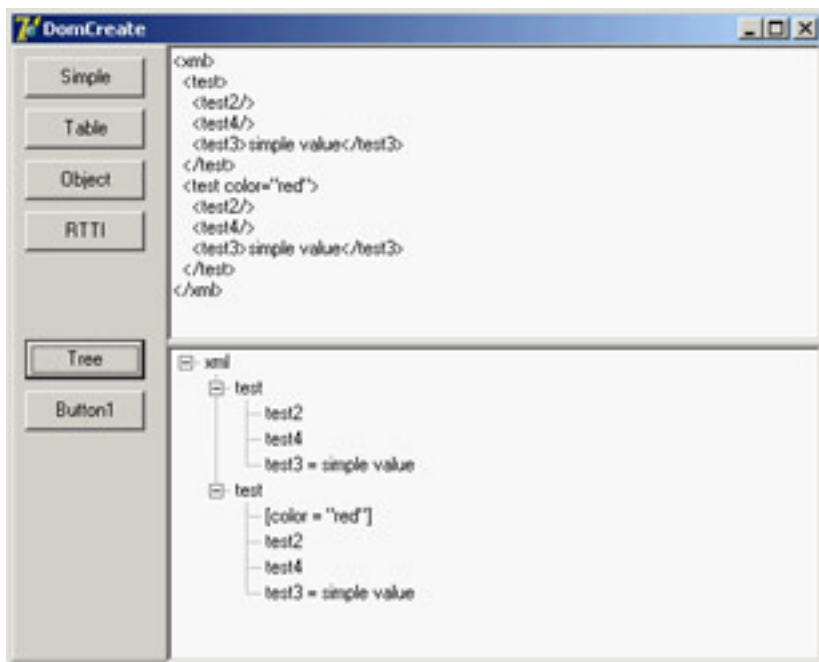
// node replication
iNode2 := iNode.cloneNode (True);
iRoot.appendChild (iNode2);

// add an attribute
iAttribute := iXml.createAttribute ('color');
iAttribute.nodeValue := 'red';
iNode2.attributes.setNamedItem (iAttribute);

// show XML in memo
Memo1.Lines.Text := FormatXMLData (XMLDoc.XML.Text);
end;

```

Notice that text nodes are added explicitly, attributes are created with a specific create call, and the code uses `cloneNode` to replicate an entire branch of the tree. Overall, the code is cumbersome to write, but after a while you may get used to this style. The effect of the program is shown (formatted in the memo and in the tree) in [Figure 22.3](#).



**Figure 22.3:** The DomCreate example can generate various types of XML documents using a DOM.

The second example of DOM creation relates to a dataset. I've added to the form a `dbExpress` dataset component (but any other dataset would do). I also added to a button the call to my custom `DataSetToDOM` procedure, like this:

```
DataSetToDOM ('customers', 'customer', XMLDoc, SQLDataSet1);
```

The `DataSetToDOM` procedure creates a root node with the text of the first parameter, grabs each record of the dataset, defines a node with the second parameter, and adds a subnode for each field of the record, all using extremely generic code:

```

procedure DataSetToDOM (RootName, RecordName: string; XMLDoc: TXMLDocument;
  DataSet: TDataSet);
var
  iNode, iChild: IXMLNode;
  i: Integer;
begin
  DataSet.Open;
  DataSet.First;
  // root

```

```

XMLDoc.DocumentElement := XMLDoc.CreateNode (RootName);

// add table data
while not DataSet.EOF do
begin
  // add a node for each record
  iNode := XMLDoc.DocumentElement.AddChild (RecordName);
  for I := 0 to DataSet.FieldCount - 1 do
  begin
    // add an element for each field
    iChild := iNode.AddChild (DataSet.Fields[i].FieldName);
    iChild.Text := DataSet.Fields[i].AsString;
  end;
  DataSet.Next;
end;
DataSet.Close;
end;

```

The preceding code uses the simplified DOM access interfaces provided by Borland, which include an `AddChild` node that creates the subnode, and the direct access to the `Text` property for defining a child node with textual content. This routine extracts an XML representation of your dataset, also opening up opportunities for web publishing, as I'll discuss later in the section on XSL.

Another interesting opportunity is the generation of XML documents describing Delphi objects. The `DomCreate` program has a button that describes a few properties of an object, again using the low-level DOM:

```

procedure AddAttr (iNode: IDOMNode; Name, Value: string);
var
  iAttr: IDOMNode;
begin
  iAttr := iNode.ownerDocument.createAttribute (name);
  iAttr.nodeValue := Value;
  iNode.attributes.setNamedItem (iAttr);
end;

procedure TForm1.btnObjectClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('Button1'));

  // a few properties as attributes (might also be nodes)
  AddAttr (iRoot, 'Name', Button1.Name);
  AddAttr (iRoot, 'Caption', Button1.Caption);
  AddAttr (iRoot, 'Font.Name', Button1.Font.Name);
  AddAttr (iRoot, 'Left', IntToStr (Button1.Left));
  AddAttr (iRoot, 'Hint', Button1.Hint);

  // show XML in memo

```

```

Memol.Lines := XmlDoc.XML;
end;

```

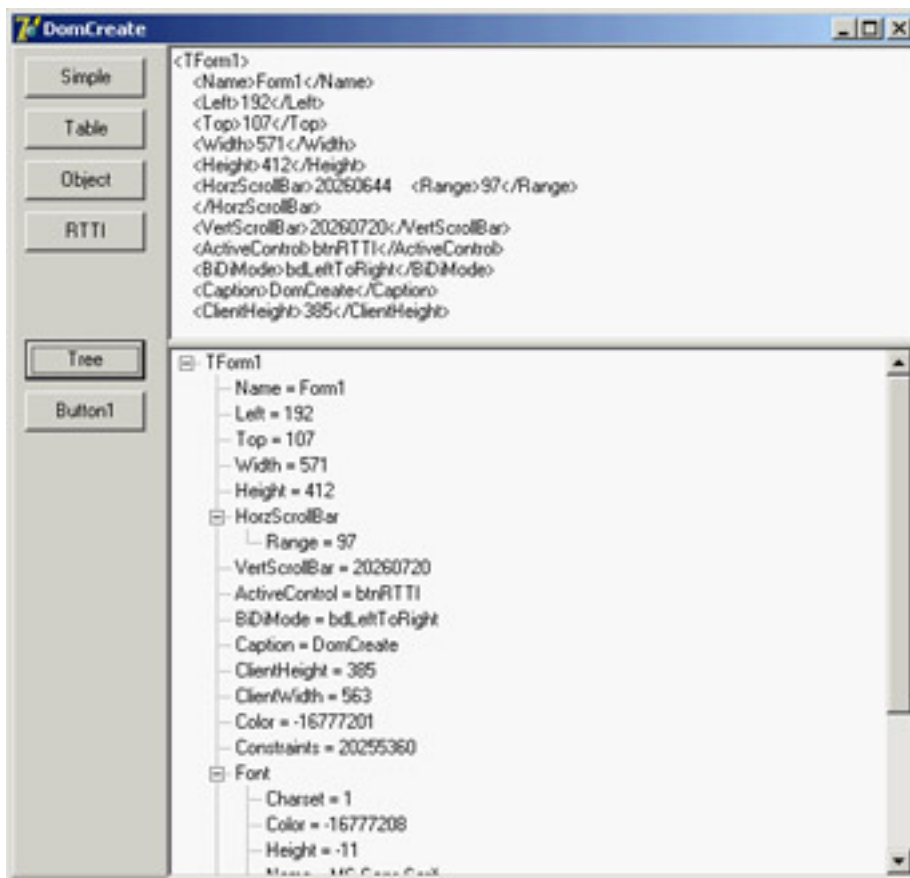
Of course, it is more interesting to have a generic technique capable of saving the properties of each Delphi component (or *persistent object*, to be more precise), recursing on persistent subobjects and indicating the names of referenced components. I've done this in the `ComponentToDOM` procedure, which uses the low-level RTTI information provided by the `TyplInfo` unit, including the extraction of the list of component properties not having a default value. Once more, the program uses the simplified Delphi XML interfaces:

```

procedure ComponentToDOM (iNode: IXmlNode; Comp: TPersistent);
var
  nProps, i: Integer;
  PropList: PPropList;
  Value: Variant;
  newNode: IXmlNode;
begin
  // get list of properties
  nProps := GetTypeData (Comp.ClassInfo)^.PropCount;
  GetMem (PropList, nProps * SizeOf(Pointer));
  try
    GetPropInfos (Comp.ClassInfo, PropList);
    for i := 0 to nProps - 1 do
      if not IsDefaultPropertyValue(Comp, PropList [i], nil) then
        begin
          Value := GetPropValue (Comp, PropList [i].Name);
          NewNode := iNode.AddChild(PropList [i].Name);
          NewNode.Text := Value;
          if (PropList [i].PropType^.Kind = tkClass) and (Value <> 0) then
            if TObject (Integer(Value)) is TComponent then
              NewNode.Text := TComponent (Integer(Value)).Name
            else
              // TPersistent but not TComponent: recurse
              ComponentToDOM (newNode, TObject (Integer(Value)) as TPersistent);
          end;
        finally
          FreeMem (PropList);
        end;
      end;
    end;
  end;

```

These two lines of code, in this case, trigger the creation of the XML document (shown in [Figure 22.4](#)):



**Figure 22.4:** The XML generated to describe the form of the DomCreate program. Notice (in the tree and in the memo text) that properties of class types are further expanded.

```
XMLDoc.DocumentElement := XMLDoc.CreateNode( Self.ClassName );
ComponentToDOM (XMLDoc.DocumentElement, Self);
```

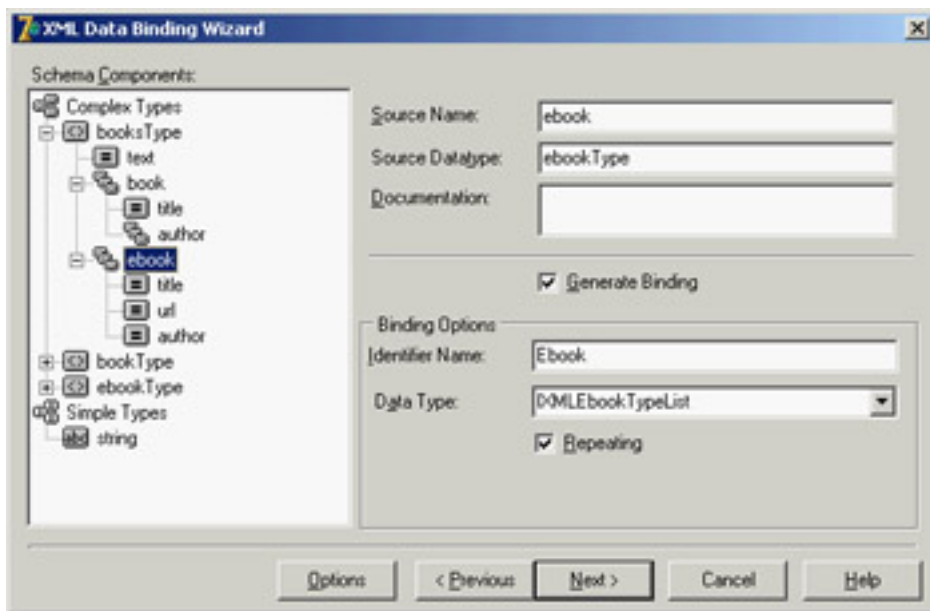
## XML Data-Binding Interfaces

You have seen that working with the DOM to access or generate a document is tedious, because you must use positional information and not logical access to the data. Also, handling series of repeated nodes of different possible types (as shown in the XML sample in [Listing 22.1](#), describing books) is far from simple. Moreover, using a DOM, you can create any well-formed document; but (unless you use a validating DOM) you can add any subnode to any node, coming up with almost useless documents, because no one else will be able to manage them.

To solve these issues, Borland added to Delphi an XML Data Binding Wizard, which can examine an XML document or a document definition (a schema, a Document Type Definition (DTD), or another type of definition) and generate a set of interfaces for manipulating the document. These interfaces are specific to the document and its structure and allow you to have more readable code, but they are certainly less generic as far as the types of documents you can handle (and this is more positive than it might sound at first).

You can activate the XML Data Binding Wizard by using the corresponding icon in the first page of the IDE's New Items dialog box or by double-clicking the XMLDocument component. (It is odd that the corresponding command is not in the shortcut menu of the component.)

After a first page in which you can select an input file, this wizard shows you the structure of the document graphically, as you can see in [Figure 22.5](#) for the sample XML file from [Listing 22.1](#). In this page, you can give a name to each entity of the generated interfaces, if you don't like the defaults suggested by the wizard. You can also change the rules used by the wizard to generate the names (an extended flexibility I'd like to have in other areas of the Delphi IDE). The final page gives you a preview of the generated interfaces and offers options for generating schemas and other definition files.



**Figure 22.5:** Delphi's XML Data Binding Wizard can examine the structure of a document or a schema (or another document definition) to create a set of interfaces for simplified and direct access to the DOM data.

For the sample XML file with the author names, the XML Data Binding Wizard generates an interface for the root node, two interfaces for the elements lists of two different types of nodes (books and e-books), and two interfaces for the elements of the two types. Here are a few excerpts of the generated code, available in the `XmlIntfDefinition` unit of the `XmlInterface` example:

```

type
  IXMLBooksType = interface(IXMLNode)
    ['{C9A9FB63-47ED-4F27-8ABA-E71F30BA7F11}']
    { Property Accessors }
    function Get_Text: WideString;
    function Get_Book: IXMLBookTypeList;
    function Get_Ebook: IXMLEbookTypeList;
    procedure Set_Text(Value: WideString);
    { Methods & Properties }
    property Text: WideString read Get_Text write Set_Text;
    property Book: IXMLBookTypeList read Get_Book;
    property Ebook: IXMLEbookTypeList read Get_Ebook;
  end;

  IXMLBookTypeList = interface(IXMLNodeCollection)
    ['{3449E8C4-3222-47B8-B2B2-38EE504790B6}']
    { Methods & Properties }
    function Add: IXMLBookType;
    function Insert(const Index: Integer): IXMLBookType;
    function Get_Item(Index: Integer): IXMLBookType;
    property Items[Index: Integer]: IXMLBookType read Get_Item; default;
  end;

  IXMLBookType = interface(IXMLNode)
    ['{26BF5C51-9247-4D1A-8584-24AE68969935}']
    { Property Accessors }
    function Get_Title: WideString;
    function Get_Author: IXMLString_List;
    procedure Set_Title(Value: WideString);
    { Methods & Properties }
    property Title: WideString read Get_Title write Set_Title;
    property Author: IXMLString_List read Get_Author;
  
```

```
end;
```

For each interface, the XML Data Binding Wizard also generates an implementation class that provides the code for the interface methods by translating the requests into DOM calls. The unit includes three initialization functions, which can return the interface of the root node from a document loaded in an `IXMLDocument` component (or a component providing a generic `IXMLDocument` interface), or return one from a file, or create a brand new DOM:

```
function Getbooks(Doc: IXMLDocument): IXMLBooksType;  
function Loadbooks(const FileName: WideString): IXMLBooksType;  
function Newbooks: IXMLBooksType;
```

After generating these interfaces using the wizard in the `XmlInterface` example, I've repeated XML document access code that's similar to the `XmlDomTree` example but is much simpler to write (and to read). For example, you can get the attribute of the root node by writing

```
procedure TForm1.btnAttrClick(Sender: TObject);  
var  
    Books: IXMLBooksType;  
begin  
    Books := Getbooks (XmlDocument1);  
    ShowMessage (Books.Text);  
end;
```

It is even simpler if you recall that while typing this code, Delphi's code insight can help by listing the available properties of each node, thanks to the fact that the parser can read in the interface definitions (although it cannot understand the format of a generic XML document). Accessing a node of one of the sublists is a matter of writing one of the following statements (possibly the second, with the default array property):

```
Books.Book.Items[1].Title // full  
Books.Book[1].Title       // further simplified
```

You can use similarly simplified code to generate new documents or add new elements, thanks to the customized `Add` method available in each list-based interface. Again, if you don't have a predefined structure for the XML document, as in the dataset-based and RTTI-based examples of the previous demonstration, you won't be able to use this approach.

## Validation and Schemas

The XML Data Binding Wizard can work from existing schemas or generate a schema for an XML document (and eventually save it in a file with the `.XDB` extension). An XML document describes some data, but to exchange this data among companies, it must stick to some agreed structure. A schema is a document definition against which a document can be checked for correctness, an operation usually indicated with the term *validation*.

The first—and still widespread—type of validation available for XML used *document type definitions (DTDs)*. These documents describe the structure of the XML but cannot define the possible content of each node. Also, DTDs are not XML document themselves, but use a different, awkward notation.

At the end of 2000, the W3C approved the first official draft of XML *schemas* (already available in an incompatible version called XML-Data within Microsoft's DOM). An XML schema is an XML document that can validate both the structure of the XML tree and the content of the node. A schema is based on the use and definition of simple and complex data types, similar to what happens in an OOP language.

A schema defines complex types, indicating for each the possible nodes, their optional sequence (*sequence*, *all*), the number of occurrences of each subnode (*minOccurs*, *maxOccurs*), and the data type of each specific element. Here is the schema defined by the XML Data Binding Wizard for the sample books file:

```
<?xml version="1.0"?>  
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema "  
    xmlns:xdb="http://www.borland.com/schemas/delphi/7.0/XMLDataBinding">  
    <xs:element name="books" type="booksType"/>  
    <xs:complexType name="booksType">
```

```

<xs:annotation>
  <xs:appinfo xdb:docElement="books" />
</xs:annotation>
<xs:sequence>
  <xs:element name="book" type="bookType" maxOccurs="unbounded" />
  <xs:element name="ebook" type="ebookType" maxOccurs="unbounded" />
</xs:sequence>
<xs:attribute name="text" type="xs:string" />
</xs:complexType>
<xs:complexType name="bookType">
  <xs:annotation>
    <xs:appinfo xdb:repeated="True" />
  </xs:annotation>
  <xs:sequence>
    <xs:element name="title" type="xs:string" />
    <xs:element name="author" type="xs:string" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ebookType">
  <xs:annotation>
    <xs:appinfo xdb:repeated="True" />
  </xs:annotation>
  <xs:sequence>
    <xs:element name="title" type="xs:string" />
    <xs:element name="url" type="xs:string" />
    <xs:element name="author" type="xs:string" />
  </xs:sequence>
</xs:complexType>
</xs:schema>

```

Microsoft and Apache DOM engines have good support for schemas. Another tool I've used for validation is XML Schema Validator (XSV), an open-source attempt at a conformant schema-aware processor, which can be used either directly via the Web or after downloading a command-line executable (see the links to the current website of this tool in the W3C's XML Schema pages).

**Note** The Delphi editor supports code completion for XML files via DTDs. Dropping a DTD file in Delphi's *bin* directory and referring to it with a *DOCTYPE* tag should enable this feature, which is not officially supported by Borland.

## Using the SAX API

The Simple API for XML (SAX) doesn't create a tree for the XML nodes, but parses the node—firing events for each node, attribute, value, and so on. Because it doesn't keep the document in memory, using the SAX allows you to manage much larger documents. Its approach is also useful for one-time examination of a document or retrieval of specific information. This is a list of the most important events fired by the SAX:

- `StartDocument` and `EndDocument` for the entire document
- `StartElement` and `EndElement` for each node
- `Characters` for the text within the nodes

It is common to use a stack to handle the current path within the nodes tree, and push and pop elements to and from the stack for every `StartElement` and `EndElement` event.

Delphi does not include specific support for the SAX interface, but you can import Microsoft's XML support (the MSXML library). In particular, for the `SaxDemo1` example I've used version 2 of MSXML, because this version is widely



available. I've generated a Pascal type library import unit from the type library, and the import unit is available within the source code of the program, but you must have that specific COM library registered on your computer to run the program successfully.

**Note** Another example at the end of this chapter (LargeXml) demonstrates, among other things, the use of the SAX API, including the OpenXml engine.

To use the SAX, you must install a SAX event handler within a SAX reader, and then load a file and parse it. I've used the SAX reader interface provided by MSXML for VB programmers. The official (C++) interface had a few errors in its type library that prevented Delphi from importing it properly. The main form of the SaxDemo1 example declares

```
sax: IVBSAXXMLReader;
```

In the `FormCreate` method, the `sax` variable is initialized with the COM object:

```
sax := CoSAXXMLReader.Create;  
sax.ErrorHandler := TMySaxErrorHandler.Create;
```

The code also sets an error handler, which is a class implementing a specific interface (`IVBSAXErrorHandler`) with three methods that are called depending on the severity of the problem: `error`, `fatalError`, and `ignorableWarning`.

Simplifying the code a little, the SAX parser is activated by calling the `parseURL` method after assigning a content handler to it:

```
sax.ContentHandler := TMySaxHandler.Create;  
sax.parseURL (filename)
```

So, the code ultimately resides in the `TMySaxHandler` class, which has the SAX events. Because I have multiple SAX content handlers in this example, I've written a base class with the core code and a few specialized versions for specific processing. Following is the code of the base class, which implements both the `IVBSAXContentHandler` interface and the `IDispatch` interface the `IVBSAXContentHandler` interface is based on:

```
type  
TMySaxHandler = class (TInterfacedObject, IVBSAXContentHandler)  
protected  
    stack: TStringList;  
public  
    constructor Create;  
    destructor Destroy; override;  
    // IDispatch  
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;  
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):  
        HRESULT; stdcall;  
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;  
        NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;  
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;  
        Flags: Word; var Params; var Result, ExcepInfo, ArgErr: Pointer):  
        HRESULT; stdcall;  
    // IVBSAXContentHandler  
    procedure Set_documentLocator(const Param1: IVBSAXLocator);  
        virtual; safecall;  
    procedure startDocument; virtual; safecall;  
    procedure endDocument; virtual; safecall;  
    procedure startPrefixMapping(var strPrefix: WideString;  
        var strURI: WideString); virtual; safecall;  
    procedure endPrefixMapping(var strPrefix: WideString); virtual; safecall;  
    procedure startElement(var strNamespaceURI: WideString;  
        var strLocalName: WideString; var strQName: WideString;
```

```

    const oAttributes: IVBSAXAttributes); virtual; safecall;
procedure endElement(var strNamespaceURI: WideString;
    var strLocalName: WideString; var strQName: WideString);
    virtual; safecall;
procedure characters(var strChars: WideString); virtual; safecall;
procedure ignorableWhitespace(var strChars: WideString);
    virtual; safecall;
procedure processingInstruction(var strTarget: WideString;
    var strData: WideString); virtual; safecall;
procedure skippedEntity(var strName: WideString); virtual; safecall;
end;

```

The most interesting portion, of course, is the final list of SAX events. All this base class does is emit information to a log when the parser starts (`startDocument`) and finishes (`endDocument`) and keep track of the current node and its parent nodes with a stack:

```

// TMySaxHandler.startElement
stack.Add (strLocalName);
// TMySaxHandler.endElement
stack.Delete (stack.Count - 1);

```

An implementation is provided by the `TMySimpleSaxHandler` class, which overrides the `startElement` event triggered for any new node to output the current position in the tree with the following statement:

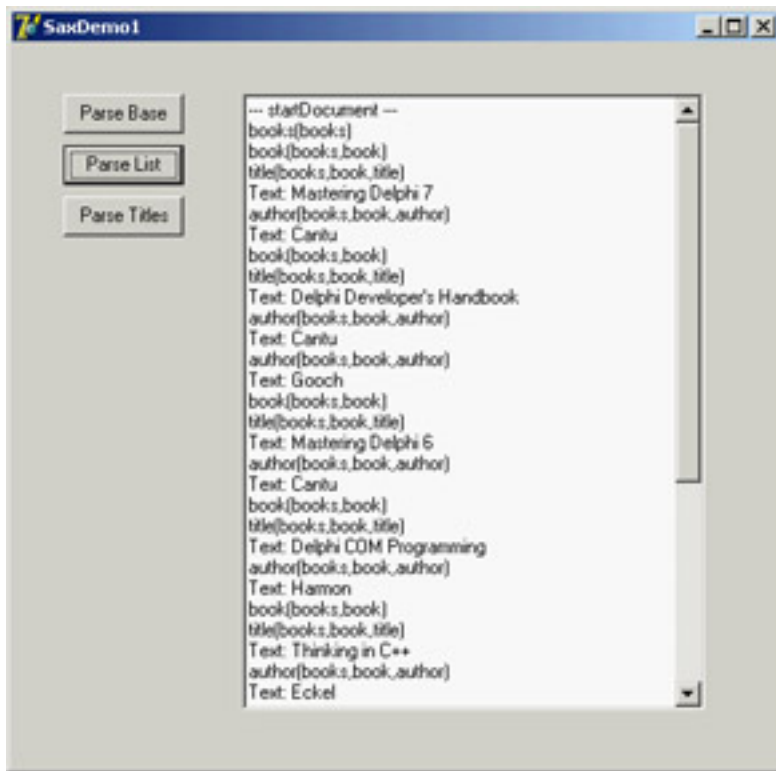
```
Log.Add (strLocalName + '(' + stack.CommaText + ')');
```

The second method of the class is the `characters` event, which is triggered when a node value (or a text node) is encountered and outputs its content (as you can see in [Figure 22.6](#)):

```

procedure TMySimpleSaxHandler.characters(var strChars: WideString);
var
    str: WideString;
begin
    inherited;
    str := RemoveWhites (strChars);
    if (str <> '') then
        Log.Add ('Text: ' + str);
end;

```



**Figure 22.6:** The log produced by reading an XML document with the SAX in the Sax-Demo1 example

This is a generic parsing operation affecting the entire XML file. The second derived SAX content handler class refers to the specific structure of the XML document, extracting only nodes of a given type. In particular, the program looks for nodes of the *title* type. When a node has this type (in `startElement`), the class sets the `isbook` Boolean variable. The text value of the node is considered only immediately after a node of this type is encountered:

```
procedure TMyBooksListSaxHandler.startElement(var strNamespaceURI,
    strLocalName, strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
    inherited;
    isbook := (strLocalName = 'title');
end;

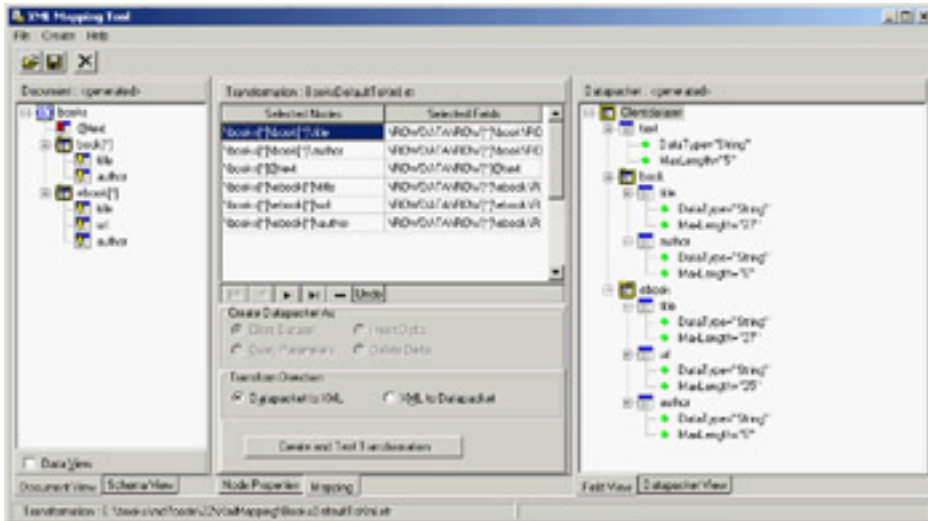
procedure TMyBooksListSaxHandler.characters(var strChars: WideString);
var
    str: string;
begin
    inherited;
    if isbook then
        begin
            str := RemoveWhites (strChars);
            if (str <> '') then
                Log.Add (stack.CommaText + ': ' + str);
        end;
end;
end;
```

## Mapping XML with Transformations

You can use one more technique in Delphi to handle some XML documents: You can create a *transformation* to translate the XML of a generic document into the format used natively by the ClientDataSet component when saving data to a MyBase XML file. In the reverse direction, another transformation can turn a dataset available within a ClientDataSet (through a DataSetProvider component) into an XML file of a required format (or schema).

Delphi includes a wizard to generate such transformations. Called the XML Mapping Tool, or XML Mapper for short, it

can be invoked from the IDE's Tools menu or executed as a stand-alone application. The XML Mapper, shown in [Figure 22.7](#), is a design-time helper that assists you in defining transformation rules between the nodes of a generic XML document and fields of the ClientDataSet data packet.



**Figure 22.7:** The XML Mapper shows the two sides of a transformation to define a mapping between them (with the rules indicated in the central portion).

The XML Mapper window has three areas:

- On the left is the XML document section, which displays information about the structure of the XML document (and eventually its data, if the related check box is active) in the Document View or an XML schema in the Schema View, depending on the selected tab.
- On the right is the data packet section, which displays information about the metadata in the data packet, either in the Field View (indicating the dataset structure) or in the Datapacket View (reporting the XML structure). The XML Mapper can also open files in the native ClientDataSet format.
- The central portion is used by the mapping section. It contains two pages: Mapping, where you can see the correspondence between selected elements of the two sides that will be part of the mapping; and Node Properties, where you can modify the data types and other details of each possible mapping.

The Mapping page of the central pane also hosts the shortcut menu used to generate the transformation. The other panes and views have specific shortcut menus you can use to perform the various actions (besides the few commands in the main menu).

You can use XML Mapper to map an existing schema (or extract it from a document) to a new data packet, an existing data packet to a new schema or document, or an existing data packet into an existing XML document (if a match is reasonable). In addition to converting the data of an XML file into a data packet, you can also convert to a delta packet of the ClientDataSet. This technique is useful for merging a document to an existing table, as if a user had inserted the modified table records. In particular, you can transform an XML document into a delta packet for records to be modified, deleted, or inserted.

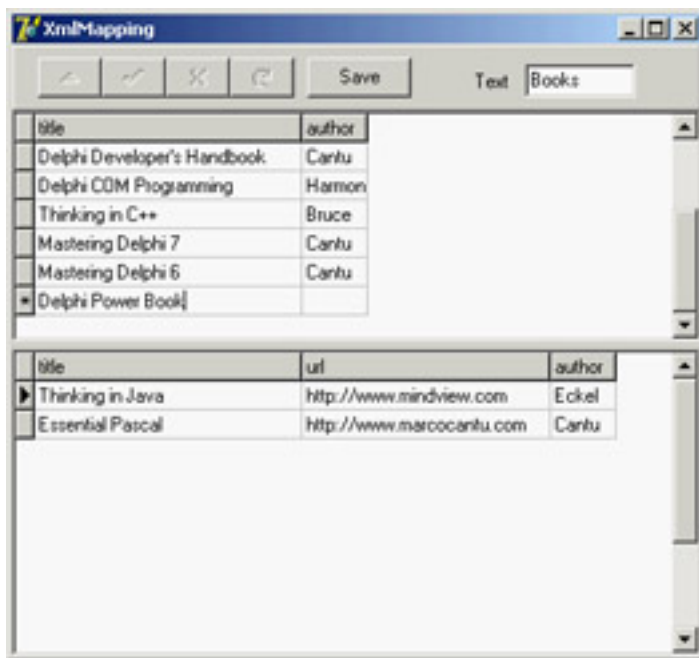
The result of using the XML Mapper is one or more transformation files, each representing a one-way conversion (so you need at least two transformation files to convert data back and forth). These transformation files are then used at design time and at run time by the XMLTransform, XMLTransformProvider, and XMLTransformClient components.

As an example, I opened the books XML document, which has a structure that doesn't easily match a table, because it includes two lists of values of different types (I've skipped easier examples in which the XML has a plain rectangular structure). After opening the `Sample.XML` file in the XML Document section, I used its shortcut menu to select all of its elements (Select All) and to create the data packet (Create Datapacket From XML). This command automatically fills the right pane with the data packet and the central portion with the proposed transformation. You can also view its effect in a sample program by clicking the Create And Test Transformation button. Doing so opens a generic

application that can load a document into the dataset using the transformation you've just created.

In this case, the XML Mapper generates a table with two dataset fields: one for each possible list of subelements. This was the only possible standard solution, because the two sublists have different structures, and it is the only solution that allows you to edit the data in a DBGrid attached to the ClientDataSet and save it back to a complete XML file, as demonstrated by the XmlMapping example. This program is basically a Windows-based editor for a complex XML document.

The example uses a TransformProvider component with two transformation files attached to read in an XML document and make it available to a ClientDataSet. As the name suggests, this component is a dataset provider. To build the user interface, I didn't connect the ClientDataSet directly to a grid, because it has a single record with a text field plus two detailed datasets. For this reason, I added to the program two more ClientDataSet components attached to the dataset fields and connected to two DBGrid controls. This explanation is easier to understand by looking at the definition of the non-visual components from DFM source code in the following excerpt and at its output in [Figure 22.8](#).



**Figure 22.8:** The XmlMapping example uses a TransformProvider component to make a complex XML document available for editing within multiple ClientDataSet components.

```
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'BooksDefault.xtr'
  TransformWrite.TransformationFile = 'BooksDefaultToXml.xtr'
  XMLDataFile = 'Sample.xml'
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'XMLTransformProvider1'
  object ClientDataSet1text: TStringField
  object ClientDataSet1book: TDataSetField
  object ClientDataSet1lebook: TDataSetField
end
object ClientDataSet2: TClientDataSet
  DataSetField = ClientDataSet1book
end
object ClientDataSet3: TClientDataSet
  DataSetField = ClientDataSet1lebook
end
```

This program allows you to edit the data of the various sublists of nodes within the grids, modifying them and also

adding or deleting records. As you apply the changes to the dataset (clicking the Save button, which calls `ApplyUpdates`), the transform provider saves an updated version of the file to disk.

As an alternative approach, you can also create transformations that map only portions of the XML document into a dataset. As an example, see the `BooksOnly.xtr` file in the folder of the `XmlMapping` example. The modified XML document you'll generate will have a different structure and content from the original, including only the portion you've selected. So, it can be useful for viewing the data, but not for editing it.

**Note** It is not surprising that the transformation files are themselves XML documents, as you can see by opening one in the editor. This XML document uses a custom format.

At the opposite side, you can see how a transformation can be used to take a database table or the result of a query and produce an XML file with a more readable format than that provided by default by the `ClientDataSet` persistence mechanism. To build the `MapTable` example, I placed a `dbExpress SimpleDataSet` component on a form and attached a `DataSetProvider` to it and a `ClientDataSet` to the provider. After opening the table and the client dataset, I saved its content to an XML file.

At that point, I opened the XML Mapper, loaded the data packet file into it, selected all the data packet nodes (with the `Select All` command from the shortcut menu) and invoked the `Create XML From Datapacket` command. In the following dialog box, I accepted the default name mappings for fields and only changed the suggested name for record nodes (`ROW`) into something more readable (`Customer`). If you now test the transformation, the XML Mapper will display the contents of the resulting XML document in a custom tree view.

After saving the transformation file, I was ready to resume developing the program, removing the `ClientDataSet` and adding a `DataSource` and a `DBGrid` (as a user might edit in on an attached `DBGrid` before transforming it), and an `XMLTransformClient` component. This component has the transformation file connected to it, but not an XML file. Instead, it refers to the data through the provider. Clicking the button shows the XML document within a memo (after formatting it) instead of saving it to a file, something you can do by calling the `GetDataAsXml` method (even if the Help file is far from clear about the use of this method):

```
procedure TForm1.btnMapClick(Sender: TObject);  
begin  
    Mem1.Lines.Text := FormatXmlData(XMLTransformClient1.GetDataAsXml(''));  
end;
```

This is the only code for the program visible at run time in [Figure 22.9](#); you can see the original dataset in the `DBGrid` and the resulting XML document in the memo control below the grid. The application has much simpler code than the `DomCreate` example I used to generate a similar XML document, but it requires the design-time definition of the transformation. The `DomCreate` example could work on any dataset at run time without any connection to a specific table, because it has rather generic code. In theory, it is possible to produce similar dynamic mappings by using the events of the generic `XMLTransform` component, but I find it easier to use the DOM-based approach discussed earlier. Notice also that the `FormatXmlData` call produces nicer output but slows down the program, because it involves loading the XML into a DOM.

The screenshot shows a window titled "MapTable" with a table of customer data and an XML output area below it.

CUSTNO	COMPANY	ADDR1	ADDR2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Thid Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	

Map

```

<?xml version="1.0"?>
<Document>
  <Customer>
    <CustNo>1221</CustNo>
    <Company>Kauai Dive Shoppe</Company>
    <Addr1>4-976 Sugarloaf Hwy</Addr1>
    <Addr2>Suite 103</Addr2>
    <City>Kapaa Kauai</City>
    <State>HI</State>
    <Zip>94766-1234</Zip>
    <Country>America</Country>
    <Phone>808-555-0269</Phone>
    <FAO>808-555-0278</FAO>
    <TaxRate>8.5</TaxRate>
  
```

**Figure 22.9:** The MapTable example generates an XML document from a database table using a custom transformation file.



## XML and Internet Express

Once you have defined the structure of an XML document, you might want to let users see and edit the data in a Windows application or over the Web. This second case is interesting because Delphi provides specific support for it. Delphi 5 introduced an architecture called Internet Express, which is now available as part of the WebSnap platform. WebSnap also offers support for XSL, which I'll discuss later in this chapter.

In [Chapter 16](#), "Multitier DataSnap Applications," I discussed the development of DataSnap applications. Internet Express provides a client component called XMLBroker for this architecture, which can be used in place of a client dataset to retrieve data from a middle-tier DataSnap program and make it available to a specific type of page producer called InetXPageProducer. You can use these components in a standard WebBroker application or in a WebSnap program. The idea behind Internet Express is that you write a web server extension (as discussed in [Chapter 20](#), "Web Programming with WebBroker and WebSnap"), which in turn produces web pages hooked to your DataSnap server. Your custom application acts as a DataSnap client and produces pages for a browser client. Internet Express offers the services required to build this custom application easily.

I know this sounds confusing, but Internet Express is a four-tier architecture: SQL server, application server (the DataSnap server), web server with a custom application, and web browser. Of course, you can place the database access components within the same application handling the HTTP request and generating the resulting HTML, as in a client/ server solution. You can even access a local database or an XML file, in a two-tier structure (the server program and the browser).

In other words, Internet Express is a technology for building clients based on a browser, which lets you send the entire dataset to the client computer along with the HTML and some JavaScript for manipulating the XML and showing it into the user interface defined by the HTML. The JavaScript enables the browser to show the data and manipulate it.

### The XMLBroker Component

Internet Express uses multiple technologies to accomplish this result. The DataSnap data packets are converted into the XML format to let the program embed this data into the HTML page for web client-side manipulation. The delta data packet is also represented in XML. These operations are performed by the XMLBroker component, which can handle XML and provide data to the new JavaScript components. Like the ClientDataSet, the XMLBroker has the following:

- A `MaxRecords` property indicating the number of records to add to a single page
- A `Params` property hosting the parameters that components will forward to the remote query through the provider
- A `WebDispatch` property indicating the update request the broker responds to

The InetXPageProducer allows you to generate HTML forms from datasets in a visual way similar to the development of an AdapterPageProducer user interface. The Internet Express architecture, the interfaces it uses internally, and some of its IDE editor can together be considered the parent of the WebSnap architecture. With the notable difference of generating scripts to be executed on the server side and on the client side, they both provide an editor for placing visual components and generating such scripts. Personally, I'm not terribly happy that the older Internet Express is more XML-oriented than the newer WebSnap.

**Tip** Another common feature of the InetXPageProducer and the AdapterPageProducer is the support for Cascading Style Sheets (CSS). These components have the `Style` and `StylesFile` properties for defining the CSS, and each visual element has a `StyleRule` property you can use to select the style name.

### JavaScript Support

To make the editing operations on the client side powerful, the InetXPageProducer uses special JavaScript components and code. Delphi embeds a large JavaScript library, which the browser must download. This process might seem a nuisance, but it is the only way the browser interface (which is based on dynamic HTML) can be rich



enough to support field constraints and other business rules with the browser. This functionality is impossible with plain HTML. The JavaScript files provided by Borland, which you should make available on the website hosting the application, are the following:

File	Description
XmlDom.js	DOM-compatible XML parser (for browsers lacking native XML DOM support)
XmlDb.js	JavaScript classes for the HTML controls
XmlDisp.js	JavaScript classes for binding XML data with the HTML controls
XmlErrDisp.js	Classes for reconciling errors
XmlShow.js	JavaScript functions to display data and delta packets (for debugging purposes)

HTML pages generated by Internet Express usually include references to these JavaScript files, such as:

```
<script language=Javascript type="text/javascript"
  src="IncludePathURL/xmlDb.js"></script>
```

You can customize the JavaScript by adding code directly to the HTML pages or by creating new Delphi components written to fit with the Internet Express architecture that emits JavaScript code (possibly along with HTML). As an example, the sample `TPromptQueryButton` class of `INetXCustom` generates the following HTML and JavaScript code:

```
<script language=javascript type="text/javascript">
  function PromptSetField(input, msg) {
    var v = prompt(msg);
    if (v == null || v == "")
      return false;
    input.value = v
    return true;
  }
  var QueryForm3 = document.forms['QueryForm3'];
</script>
<input type=button value="Prompt..."
  onclick="if (PromptSetField(PromptResult, 'Enter some text\n'))
  QueryForm3.submit();">
```

**Tip** If you plan to use Internet Express, look at the `INetXCustom` extra demo components, available in the `\Demos\Midas\InternetExpress\INetXCustom` folder. Follow the detailed instructions in the `readme.txt` file to install these components, which are provided by Borland with no support but allow you to add many more features to your Internet Express applications with little extra effort.

To deploy this architecture you don't need anything special on the client side, because any browser up to the HTML 4 standard can be used, on any operating system. The web server, however, must be a Win32 server (this technology is not available in Kylix), and you must deploy the `DataSnap` libraries on it.

## Building an Example

To better understand what I'm talking about, and as a way to cover more technical details, let's try a demo called `leFirst`. To avoid configuration issues, this is a CGI application accessing a dataset directly—in this case, a local table retrieved via a `ClientDataSet` component. Later, I'll show you how to turn an existing `DataSnap` Windows client into a browser-based interface. To build `leFirst`, I created a new CGI application and added to its data module a `ClientDataSet` component hooked to a local `.CDS` file and a `DataSetProvider` component connected with the dataset. The next step is to add an `XMLBroker` component and connect it to the provider:

```
object ClientDataSet1: TClientDataSet
  FileName = 'C:\Program Files\Common Files\Borland
```

```

    Shared\Data\employee.cds'
end
object DataSetProvider1: TDataSetProvider
    DataSet = ClientDataSet1
end
object XMLBroker1: TXMLBroker
    ProviderName = 'DataSetProvider1'
    WebDispatch.MethodType = mtAny
    WebDispatch.PathInfo = 'XMLBroker1'
    ReconcileProducer = PageProducer1
    OnGetResponse = XMLBroker1GetResponse
end

```

The `ReconcileProducer` property is required to show a proper error message in case of an update conflict. As you'll see later, one of the Delphi demos includes some custom code, but in this example I've connected a traditional `PageProducer` component with a generic HTML error message. After setting up the XML broker, you can add an `InetXPageProducer` to the web data module. This component has a standard HTML skeleton; I've customized it to add a title, without touching the special tags:

```

<HTML><HEAD>
    <title>IeFirst</title>
</HEAD><BODY>
    <h1>Internet Express First Demo (IeFirst.exe)</h1>
    <#INCLUDES><#STYLES><#WARNINGS><#FORMS><#SCRIPT>
</BODY>

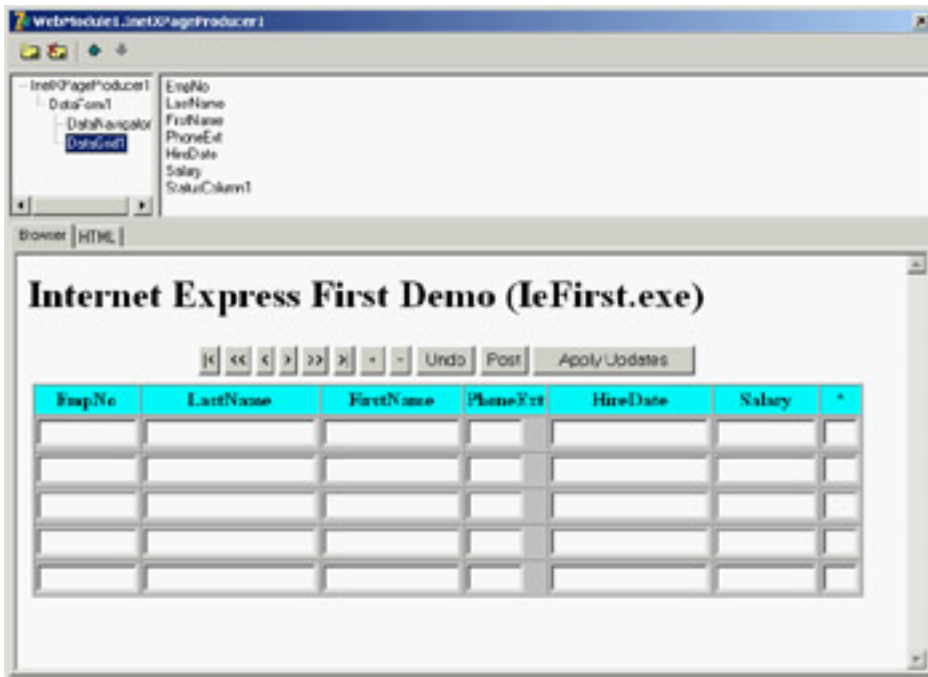
```

The special tags are automatically expanded using the JavaScript files of the directory specified by the `IncludePathURL` property. You *must* set this property to refer to the web server directory where these files reside. You can find them in the `\Delphi7\Source\WebMidas` directory. The five tags have the following effect:

Tag	Effect
<#INCLUDES>	Generates the instructions to include the JavaScript libraries
<#STYLES>	Adds the embedded style sheet definition
<#WARNINGS>	Used at design time to show errors in the <code>InetXPageProducer</code> editor
<#FORMS>	Generates the HTML code produced by the components of the web page
<#SCRIPT>	Adds a JavaScript block used to start the client-side script

**Note** The `InetXPageProducer` component also handles a few more internal tags. `<#BODYELEMENTS>` corresponds to all five tags of the predefined template. `<#COMPONENT Name=WebComponentName>` is part of the generated HTML code used to declare the components generated visually. `<#DATA PACKET XMLBroker=BrokerName>` is replaced with the XML of the data packet.

To customize the resulting HTML of the `InetXPageProducer`, you can use its editor, which again is similar to the one for `WebSnap` server-side scripting. Double-click the `InetXPageProducer` component, and Delphi opens a window like the one shown in [Figure 22.10](#) (with the example's final settings). In this editor, you can create complex structures starting with a query form, data form, or generic layout group. In the example data form, I added a `DataGrid` and a `DataNavigator` component without customizing them any further (an operation you do by adding child buttons, columns, and other objects, which fully replace the defaults).



**Figure 22.10:** The InetXPage-Producer editor allows you to build complex HTML forms visually, similarly to the AdapterPageProducer.

The DFM code for the InetXPageProducer and its internal components in my example is as follows. You can see the core settings plus some limited graphical customizations:

```

object InetXPageProducer1: TInetXPageProducer
  IncludePathURL = '/jssource/'
  HTMLDoc.Strings = (...)
object DataForm1: TDataForm
  object DataNavigator1: TDataNavigator
    XMLComponent = DataGrid1
    Custom = 'align="center"'
  end
object DataGrid1: TDataGrid
  XMLBroker = XMLBroker1
  DisplayRows = 5
  TableAttributes.BgColor = 'Silver'
  TableAttributes.CellSpacing = 0
  TableAttributes.CellPadding = 2
  HeadingAttributes.BgColor = 'Aqua'
  object EmpNo: TTextColumn...
  object LastName: TTextColumn...
  object FirstName: TTextColumn...
  object PhoneExt: TTextColumn...
  object HireDate: TTextColumn...
  object Salary: TTextColumn...
  object StatusColumn1: TStatusColumn...
end
end
end

```

The value of these components is in the HTML (and JavaScript) code they generate, which you can preview by selecting the HTML tab of the InetXPageProducer editor. Here are a few pieces of the definitions in the HTML, for the buttons, the data grid heading, and one of the grid's cells:

```

// buttons
<table align="center">
  <tr><td colspan="2">

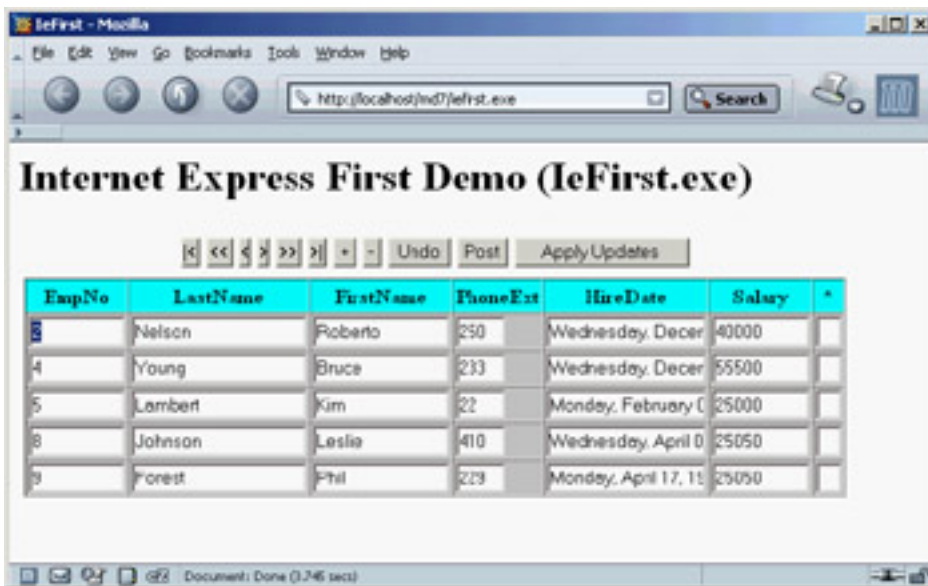
```

```

<input type="button" value="|<"
  onclick='if (xml_ready) DataGrid1_Dispatch.first();'>
<input type="button" value="<<"
  onclick='if (xml_ready) DataGrid1_Dispatch.pgup();'>
...
// data grid heading
<table cellspacing="0" cellpadding="2" border="1" bgcolor="silver">
  <tr bgcolor="aqua">
    <th>EmpNo</th>
    <th>LastName</th>
    ...
  </tr>
  <tr>
    // a data cell
    <td><div>
      <input type="text" name="EmpNo" size="10"
        onfocus='if(xml_ready)DataGrid1_Dispatch.xfocus(this);'
        onkeydown='if(xml_ready)DataGrid1_Dispatch.keys(this);'>
    </div></td>...

```

When the HTML generator is set up, you can go back to the web data module, add an action to it, and connect the action with the InetXPageProducer via the `Producer` property. This should be enough to make the program work through a browser, as you can see in [Figure 22.11](#).



**Figure 22.11:** The IeFirst application sends the browser some HTML components, an XML document, and the JavaScript code to show the data in the visual components.

If you look at the HTML file received by the browser, you'll find the table mentioned in the preceding definition, some JavaScript code, and the database data in the data packet XML format. This data is assembled by the XML broker and passed to the producer component to be embedded in the HTML file. The number of records sent to the client depends on the XMLBroker, not on the number of lines in the grid. Once the XML data is sent to the browser, you can use the navigator component's buttons to move around in the data without requiring further access to the server to fetch more. This is different from the paging behavior of WebSnap. One is not necessarily better than the other; it depends on the specific application you are building.

At the same time, the JavaScript classes in the system allow the user to type in new data, following the rules imposed by the JavaScript code hooked to dynamic HTML events. By default, the grid has an extra asterisk column indicating which records have been modified. The update data is collected in an XML data packet in the browser and sent back to the server when the user clicks the Apply Updates button. At this point, the browser activates the action specified by

the `WebDispatch.PathInfo` property of the `XMLBroker`. There is no need to export this action from the web data module, because this operation is automatic (although you can disable it by setting `WebDispatch.Enable` to `False`).

The `XMLBroker` applies the changes to the server, returning the content of the provider connected to the `ReconcileProvider` property (or raising an exception if this property is not defined). When everything works fine, the `XMLBroker` redirects the control to the main page that contains the data. However, I've experienced some problems with this technique, so the `leFirst` example handles the `OnGetResponse`, indicating this is an update view:

```
procedure TWebModule1.XMLBroker1GetResponse(Sender: TObject;  
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
begin  
    Response.Content := '<h1>Updated</h1><p>' + InetXPageProducer1.Content;  
    Handled := True;  
end;
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Using XSLT

Another approach for generating HTML starting from an XML document is the use of the Extensible Stylesheet Language (XSL) or, to be more precise, its XSL Transformations (XSLT) subset. The aim of XSLT is to transform an XML document into another document, generally an XML document. One of the most frequent uses of the technology is to turn an XML document into an XHTML document to be sent to a browser from a web server. Another interesting related technology is XSL-FO (XSL Formatting Objects), which can be used to turn an XML document into a PDF or another formatted document.

An XSLT document is a well-formed XML document. The structure of an XSLT file requires a root node like the following:

```
<xsl:stylesheet version="1.0" xmlns:xsl="...">
```

The content of the XSLT file is based on one or more templates (or rules or functions), which will be processed by the engine. Their node is `xsl:template`, usually with a `match` attribute. In the simplest case, a template operates on nodes with a given name; you invoke the template by passing to it one or more nodes with an XPath expression:

```
xsl:apply-templates select="node_name"
```

The starting point for this operation is a template that processes the root node, which can be the only template of the XSLT file. Within templates, you can find any of the other commands, such as the extraction of a value from an XML document (`xsl:value-of select`), looping statements (`xsl:for-each`), conditional expressions (`xsl:if`, `xsl:choose`), sorting requests (`xsl:sort`), and numbering requests (`xsl:number`), just to mention a few common XSLT commands.

## Using XPath

XSLT uses other XML technologies, notably XPath to identify portions of documents. XPath defines a set of rules to locate one or more nodes within a document. The rules are based on a path-line structure of the node within the XML tree, so that `/books/book` identifies any *book* node under the *books* document root. XPath uses special symbols to identify nodes:

- A star (\*) stands for any node; for example, `book/*` indicates any subnode under a *book* node.
- A dot (.) stands for the current node.
- The pipe symbol (|) indicates alternatives, as in `book|ebook`.
- A double slash (//) stands for any path, so that `//title` indicates all the title nodes, whatever their parent nodes, and `books//author` indicates any *author* node under a *books* node regardless of the nodes in between.
- The at sign (@) indicates an attribute instead of a node, as in the hypothetical `author/@lastname`.
- Square brackets can be used to choose only nodes or attributes having a given value. For example, to select all authors with a given first name attribute, you can use `author[@name="marco"]`.

There are many other cases, but this short introduction to the rules of XPath should get you started and help you understand the following examples. An XSLT document is an XML document that works on the structure of a source XML document and generates in output another XML document, such as an XHTML document you can view in a web browser.

**Note** Commonly used XSLT processors include MS-XML, Xalan from the Apache XML project ([xml.apache.org](http://xml.apache.org)), and the Java-based Xt from James Clarke. From Delphi, you can also use the XSLT engine, included in TurboPower's XML Partner Pro ([www.turbopower.com](http://www.turbopower.com)).

## XSLT in Practice

Let's discuss a couple of examples. As a starting point, you should study XSL by itself, and then focus on its activation from within a Delphi application.

For your initial tests, you can connect an XSL file directly to an XML file. As you load the XML file in Internet Explorer, you will see the resulting XHTML transformation. The connection is indicated in the heading of the XML document with a command like this:

```
<?xml-stylesheet type="text/xsl" href="sampleembedded.xsl"?>
```

This is what I've done in the `sampleembedded.xml` file available in the `XslEmbed` folder. The related XSL embeds various XSL snippets that I don't have space to discuss in detail. For example, it grabs the entire list of authors or filters a specific group of them with this code:

```
<h2>All Authors</h2>
<ul>
  <xsl:for-each select="books//author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
<h3>E-Authors</h3>
<ul>
  <xsl:for-each select="books/ebook/author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
```

More complex code is used to extract nodes only when a specific value is present in a subnode or attribute, regardless of the higher-level nodes. The following XSL snippet also has an `if` statement and produces an attribute in the resulting node, as a way to build an `href` hyperlink in the HTML:

```
<h3>Marco's works (books + ebooks)</h3>
<ul>
  <xsl:for-each select="books/*[author = 'Cantu']">
    <li> <xsl:value-of select="title"/>
      <xsl:if test="url">
        (<a><xsl:attribute name="href"><xsl:value-of select="url"/>
          </xsl:attribute>Jump to document</a>)
      </xsl:if>
    </li>
  </xsl:for-each>
</ul>
```

## XSLT with WebSnap

Within the code of a program, you can execute the `TransformNode` method of a DOM node, passing to it another DOM hosting the XSL document. Instead of using this low-level approach, however, you can let WebSnap help you to create an XSL-based example. You can create a new WebSnap application (I've built a CGI program called `XslCust` in this case) and choose an `XSLPageProducer` component for its main page, to let Delphi help you begin the application code. Delphi also includes a skeleton XSL file for manipulating a `ClientDataSet` data packet and adds many new views to the editor. The XSL text replaces the HTML file; the XML Tree page shows the data, if any; the XSL Tree page shows the XSL within the Internet Explorer ActiveX; the HTML Result page shows the code produced by the transformation; and the Preview page shows what a user will see in a browser.

**Tip** In Delphi 7, the editor provides full-blown code completion for XSLT, which makes editing this code in the editor as powerful as it is in some sophisticated and specific XML editors.



To make this example work, you must provide data to the XSLPageProducer component via its `XMLData` property. This property can be hooked up to an `XMLDocument` or directly to an `XMLBroker` component, as I've done in this case. The broker takes its data from a provider connected to a local table attached to the classic `Customer.cds` component table of the classic `DBDEMOS`. The effect is that, with the following Delphi-generated XSL, you get (even at design time) the output of shown in [Figure 22.12](#):

```
<?xml version="1.0"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="DATAPACKET">
    <table border="1">
      <xsl:apply-templates select="METADATA/FIELDS"/>
      <xsl:apply-templates select="ROWDATA/ROW"/>
    </table>
  </xsl:template>

  <xsl:template match="FIELDS">
    <tr>
      <xsl:apply-templates/>
    </tr>
  </xsl:template>

  <xsl:template match="FIELD">
    <th>
      <xsl:value-of select="@attrname"/>
    </th>
  </xsl:template>

  <xsl:template match="ROWDATA/ROW">
    <xsl:variable name="fieldDefs" select="//METADATA/FIELDS"/>
    <xsl:variable name="currentRow" select="current()"/>
    <tr>
      <xsl:for-each select="$fieldDefs/FIELD">
        <td>
          <xsl:value-of select="$currentRow/@*[name()=current()/@attrname]"/><br/>
        </td>
      </xsl:for-each>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```



CustNo	Company	Addr1	Addr2	City	State	Zip	Country	PI
1221	Kana Dwe Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kanae	HI	94766-1234	US	801 55: 02
1231	Unisco	PO Box Z-547		Freeport			Bahamas	801 55: 39
1351	Sight Diver	1 Neptune Lane		Kato Paphos			Cyprus	35 87
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman			British West Indies	01 69
1356	Tom Sawyer	632-1 Third		Christianssted	St. Croix	00020	US Virgin	50 79

**Figure 22.12:** The result of an XSLT transformation generated (even at design time) by the XSLPageProducer component in the XslCust example

**Note** The standard XSL template has been extended since Delphi 6, because the original versions didn't account for null fields omitted from the XML data packet. I presented several extensions to the original XSL code at the 2002 Borland Conference, and some of my suggestions have been incorporated in the template.

This code generates an HTML table consisting of the expansion of field metadata and row data. The fields are used to generate the table heading, with a <th> cell for each entry in a single row. The row data is used to fill in the other rows of the table. Taking the value of each attribute (`select="@"`) wouldn't be enough, because an attribute might be missing. For this reason, the list of fields and the current row are saved in two variables; then, for each field, the XSL code extracts the value of a row item having an attribute name (`@*[name()=...]`) corresponding to the name of the current field stored in its *attrname* attribute (`@attrname`). This code is far from simple, but it is a compact and portable way to examine different portions of an XML document at the same time.

## Direct XSL Transformations with the DOM

Using the XSLPageProducer can be handy, but generating multiple pages based on the same data just to handle different possible XSL styles with WebSnap isn't the best approach. I've built a plain CGI application called CdsXslt that can transform a ClientDataSet data packet into different types of HTML, depending on the name of the XSL file passed as a parameter. The advantage is that I can modify the existing XSL files and add new XSL files to the system without having to recompile the program.

To obtain the XSL transformation, the program loads both the XML and the XSL files into two XMLDocument components called `xmlDom` and `XslDom`. Then it invokes the `transformNode` method of the XML document, passing the XSL document as a parameter and filling in a third XMLDocument component called `HtmlDom`:

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  xslfile, xslfolder: string;
  attr: IDOMAttr;
begin
  // open the client dataset and load its XML in a DOM
  ClientDataSet1.Open;
  XmlDom.Xml.Text := ClientDataSet1.XMLData;
  XmlDom.Active := True;
  // load the requested xsl file
  xslfile := Request.QueryFields.Values ['style'];
  if xslfile = '' then
    xslfile := 'customer.xsl';

```

```

xslfolder := ExtractFilePath (ParamStr (0)) + 'xsl\';
if FileExists (xslfolder + xslfile) then
  xslDom.LoadFromFile (xslfolder + xslfile)
else
  raise Exception.Create('Missing file: ' + xslfolder + xslfile);
XSLDom.Active := True;
if xslfile = 'single.xsl' then
begin
  attr := xslDom.DOMDocument.createAttribute('select');
  attr.value := '//ROW[@CustNo="' + Request.QueryFields.Values ['id'] + '"']';
  xslDom.DOMDocument.getElementsByTagName ('xsl:apply-templates').
    item[0].attributes.setNamedItem(attr);
end;
// do the transformation
HTMLDom.Active := True;
xmlDom.DocumentElement.transformNode (xslDom.DocumentElement, HTMLDom);
Response.Content := HTMLDom.XML.Text;
end;

```

The code uses the DOM to modify the XSL document for displaying a single record, adding the XPath statement for selecting the record indicated by the `id` query field. This `id` is added to the hyperlink by the XSL with the list of records, but I'll skip listing more XSL files. They are available for study in the XSL subfolder of this example's folder.

**Warning** To run this program, deploy the XSL files in a folder called XSL under the one where the script is located. You can find the demo files in the XSL subfolder of the scripts folder of this chapter. To deploy these files in a different location, change the code above that extracts the XSL folder name from the program name available in the first common line parameter (as the global *Application* object defined in the Forms unit is not accessible in a CGI application).

## Processing Large XML Documents

As you have seen, there are often many different techniques to accomplish the same task with XML. In many cases you can choose any solution with the goal of writing less and more maintainable code; but when you need to process a large number of XML documents or very large XML documents, you must consider efficiency.

Discussing theory by itself is not terribly useful, so I've built an example you can use (and modify) to test different solutions. The example is called LargeXml, and it covers a specific area: moving data from a database to an XML file and back. The example can open a dataset (using dbExpress) and then replicate the data many times in a ClientDataSet in memory. The structure of the in-memory ClientDataSet is cloned from that of the data access component:

```
SimpleDataSet1.Open;
ClientDataSet1.FieldDefs := SimpleDataSet1.FieldDefs;
ClientDataSet1.CreateDataSet;
```

After using a radio group to determine the amount of data you want to process (some options require minutes on a slow computer), the data is cloned with this code:

```
while ClientDataSet1.RecordCount < nCount do
begin
    SimpleDataSet1.RecNo := Random (SimpleDataSet1.RecordCount) + 1;
    ClientDataSet1.Insert;
    ClientDataSet1.Fields [0].AsInteger := Random (10000);
    for I := 1 to SimpleDataSet1.FieldCount - 1 do
        ClientDataSet1.Fields [i].AsString := SimpleDataSet1.Fields [i].AsString;
    ClientDataSet1.Post;
end;
```

## From a ClientDataSet to an XML Document

Now that the program has a (large) dataset in memory, it provides three different ways to save the dataset to a file. The first is to save the XMLData of the ClientDataSet directly to a file, obtaining an attribute-based document. This is probably not the format you want, so the second solution is to apply an XmlMapper transformation with an XMLTransformClient component. The third solution involves processing the dataset directly and writing out each record to a file:

```
procedure TForm1.btnSaveCustomClick(Sender: TObject);
var
    str: TFileStream;
    s: string;
    i: Integer;
begin
    str := TFileStream.Create ('data3.xml', fmCreate);
    try
        ClientDataSet1.First;
        s := '<?xml version="1.0" standalone="yes" ?><employee>';
        str.Write(s[1], Length (s));

        while not ClientDataSet1.EOF do
            begin
                s := '';
                for i := 0 to ClientDataSet1.FieldCount - 1 do
                    s := s + MakeXmlstr (ClientDataSet1.Fields[i].FieldName,
                        ClientDataSet1.Fields[i].AsString);
                s := MakeXmlStr ('employeeData', s);
                str.Write(s[1], length (s));
            end;
    finally
        str.Free;
    end;
end;
```

```

ClientDataSet1.Next
end;
s := '</employee>';
str.Write(s[1], length (s));
finally
str.Free;
end;
end;

```

This code uses a simple (but effective) support function to create XML nodes:

```

function MakeXmlstr (node, value: string): string;
begin
Result := '<' + node + '>' + value + '</' + node + '>';
end;

```

If you run the program, you can see the time taken by each operation, as shown in [Figure 22.13](#). Saving the ClientDataSet data is the fastest approach, but you probably don't get the result you want. Custom streaming is only slightly slower; but you should consider that this code doesn't require you to first move the data to a ClientDataSet, because you can apply it directly even to a unidirectional dbExpress dataset. You should forget using the code based on the XmlMapper for a large dataset, because it is hundreds of times slower, even for a small dataset (I haven't been able to try a large dataset, because the process takes too long). For example, the 50 milliseconds required by custom streaming for a small dataset become more than 10 seconds when I use the mapping, and the result is very similar.

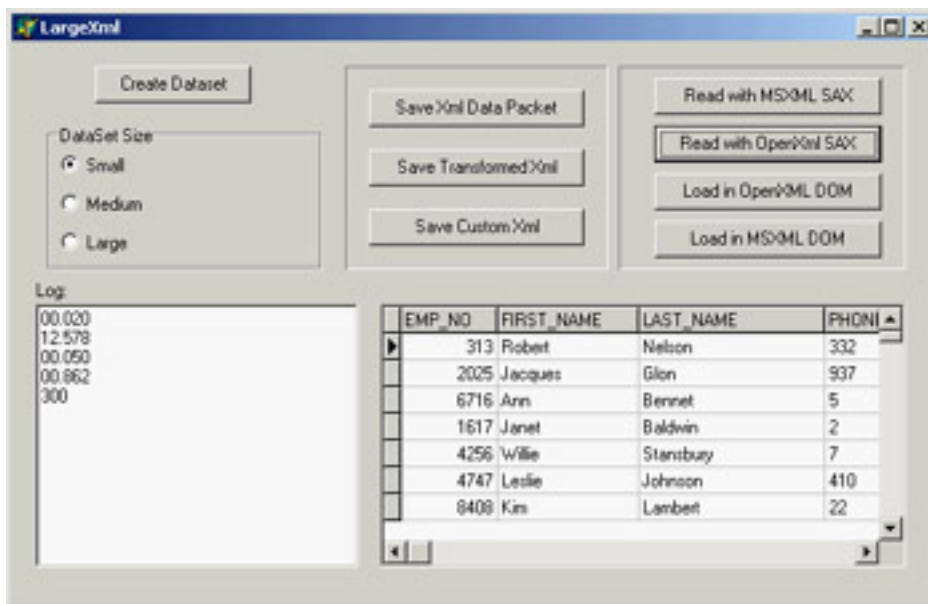


Figure 22.13: The LargeXml example in action

## From an XML Document to a ClientDataSet

Once you have a large XML document, obtained by a program (as in this case) or from an external source, you need to process it. As you have seen, XmlMapper support is far too slow, so you are left with three alternatives: an XSL transformation, a SAX, or a DOM. XSL transformations will probably be fast enough, but in this example I've opened the document with a SAX; it's the fastest approach and doesn't require much code. The program can also load a document in a DOM, but I haven't written the code to navigate the DOM and save the data back to a ClientDataSet.

In both cases, I've tested the OpenXml engine versus the MSXML DOM. This allows you to see the two SAX solutions compared, because (unluckily) the code is slightly different. I can summarize the results here: Using the MSXML SAX is slightly faster than using the OpenXml SAX (the difference is about 20 percent), whereas loading in the DOM marks a large advantage in favor of MSXML.

The MSXML SAX code uses the same architecture discussed in the SaxDemo1 example, so here I've listed only the code of the handlers you use. As you can see, at the beginning of an employeeData element you insert a new record, which is posted when the same node is closed. Lower-level nodes are added as fields of the current record. Here is the code:

```

procedure TMyDataSaxHandler.startElement(var strNamespaceURI, strLocalName,
    strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
    inherited;
    if strLocalName = 'employeeData' then
        Form1.clientdataset2.Insert;
    strCurrent := '';
end;

procedure TMyDataSaxHandler.characters(var strChars: WideString);
begin
    inherited;
    strCurrent := strCurrent + RemoveWhites(strChars);
end;

procedure TMyDataSaxHandler.endElement(var strNamespaceURI, strLocalName,
    strQName: WideString);
begin
    if strLocalName = 'employeeData' then
        Form1.clientdataset2.Post;
    if stack.Count > 2 then
        Form1.ClientDataSet2.FieldByName (strLocalName).AsString := strCurrent;
    inherited;
end;

```

The code for the event handlers in the OpenXml version is similar. All that changes are the interface of the methods and the names of the parameters:

```

type
    TDataSaxHandler = class (TXmlStandardHandler)
    protected
        stack: TStringList;
        strCurrent: string;
    public
        constructor Create(aowner: TComponent); override;
        function endElement(const sender: TXmlCustomProcessorAgent;
            const locator: TdomStandardLocator;
            namespaceURI, tagName: WideString): TXmlParserError; override;
        function PCDATA(const sender: TXmlCustomProcessorAgent;
            const locator: TdomStandardLocator; data: WideString):
            TXmlParserError; override;
        function startElement(const sender: TXmlCustomProcessorAgent;
            const locator: TdomStandardLocator; namespaceURI, tagName: WideString;
            attributes: TdomNameValueList): TXmlParserError; override;
        destructor Destroy; override;
    end;

```

It is also more difficult to invoke the SAX engine, as shown in the following code (from which I've removed the code for the creation of the dataset, the timing, and the logging):

```

procedure TForm1.btnReadSaxOpenClick(Sender: TObject);
var
    agent: TXmlStandardProcessorAgent;
    reader: TXmlStandardDocReader;

```

```
filename: string;
begin
  Log := memoLog.Lines;
  filename := ExtractFilePath (Application.Exename) + 'data3.xml';
  agent := TXmlStandardProcessorAgent.Create(nil);
  reader:= TXmlStandardDocReader.Create (nil);
  try
    reader.NextHandler := TDataSaxHandler.Create (nil); // our custom class
    agent.reader := reader;
    agent.processFile(filename, filename);
  finally
    agent.free;
    reader.free;
  end;
end;
```

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

In this chapter I've covered XML and related technologies, including DOM, SAX, XSLT, XML schemas, XPath, and a few more. You've seen how Delphi simplifies DOM programming with XML access using interfaces and XML transformations. I've also discussed the use of XSL for web programming, introducing the XSLT support of WebSnap and the Internet Express architecture.

[Chapter 23](#) will continue the discussion of XML with one of the most interesting and promising technologies of the last few years: web services. I'll cover SOAP and WSDL, and also introduce UDDI and other related technologies. If you are interested in the topics discussed here from the Delphi perspective, you should refer to books specifically devoted to XML, XML schemas, and XSLT.

## Chapter 23: Web Services and SOAP

### Overview

Of all the recent features of Delphi, one stands out clearly: the support for web services built into the product. The fact that I'm discussing it near the end of the book has nothing to do with its importance, but only with the logical flow of the text, and with the fact that this is not the starting point from which to learn Delphi programming.

The topic of web services is broad and involves many technologies and business-related standards. As usual, I'll focus on the underlying Delphi implementation and the technical side of web services, rather than discuss the larger picture and business implications.

This chapter is also relevant because Delphi 7 adds a lot of power to the implementation of web services provided by Delphi 6, including support for attachments, custom headers, and much more. You'll see how to create a web service client and a web service server, and also how to move database data over SOAP using the DataSnap technology.



## Web Services

The rapidly emerging web services technology has the potential to change the way the Internet works for businesses. Browsing web pages to enter orders is fine for individuals (business-to-consumer [B2C] applications) but not for companies (business-to-business [B2B] applications). If you want to buy a few books, going to a book vendor website and punching in your requests is probably fine. But if you run a bookstore and want to place hundreds of orders a day, this is far from a good approach, particularly if you have a program that helps you track your sales and determine reorders. Grabbing the output of this program and reentering it into another application is ridiculous.

Web services are meant to solve this issue: The program used to track sales can automatically create a request and send it to a web service, which can immediately return information about the order. The next step might be to ask for a tracking number for the shipment. At this point, your program can use another web service to track the shipment until it is at its destination, so you can tell your customers how long they have to wait. As the shipment arrives, your program can send a reminder via SMS or pager to the people with pending orders, issue a payment with a bank web service, and ... I could continue but I think I've given you the idea. Web services are meant for computer interoperability, much as the Web and e-mail let people interact.

## SOAP and WSDL

Web services are made possible by the Simple Object Access Protocol (SOAP). SOAP is built over standard HTTP, so that a web server can handle the SOAP requests and the related data packets can pass through firewalls. SOAP defines an XML-based notation for requesting the execution of a method by an object on the server and passing parameters to it; another notation defines the format of a response.

**Note** SOAP was originally developed by DevelopMentor (the training company run by COM expert Don Box) and Microsoft, to overcome weaknesses involved with using DCOM in web servers. Submitted to the W3C for standardization, it is being embraced by many companies, with a particular push from IBM. It is too early to know whether there will be standardization to let software programs from Microsoft, IBM, Sun, Oracle, and many others truly interoperate, or whether some of these vendors will try to push a private version of the standard. In any case, SOAP is a cornerstone of Microsoft's .NET architecture and also of the current platforms by Sun and Oracle.

SOAP will replace COM invocation, at least between different computers. Similarly, the definition of a SOAP service in the Web Services Description Language (WSDL) format will replace the IDL and type libraries used by COM and COM+. WSDL documents are another type of XML document that provides the metadata definition of a SOAP request. As you get a file in this format (generally published to define a service), you'll be able to create a program to call it.

Specifically, Delphi provides a bi-directional mapping between WSDL and interfaces. This means you can grab a WSDL file and generate an interface for it. You can then create a client program, embedding SOAP requests via these interfaces, and use a special Delphi component that lets you convert your local interface requests into SOAP calls (I doubt you want to manually generate the XML required for a SOAP request).

The other way around, you can define an interface (or use an existing one) and let a Delphi component generate a WSDL description for it. Another component provides you with a SOAP-to-Pascal mapping, so that by embedding this component and an object implementing the interface within a server-side program, you can have your web service up and running in a matter of minutes.

## BabelFish Translations

As a first example of the use of web service, I've built a client for the BabelFish translation service offered by AltaVista. You can find this and many other services for experimentation on the XMethods website ([www.xmethods.com](http://www.xmethods.com)).

After downloading the WSDL description of this service from XMethods (also available among the source code files for this chapter), I invoked Delphi's Web Services Importer in the Web Services page of the New Items dialog box and selected the file. The wizard lets you preview the structure of the service (see [Figure 23.1](#)) and generate the proper Delphi-language interfaces in a unit like the following (with many of the comments removed):

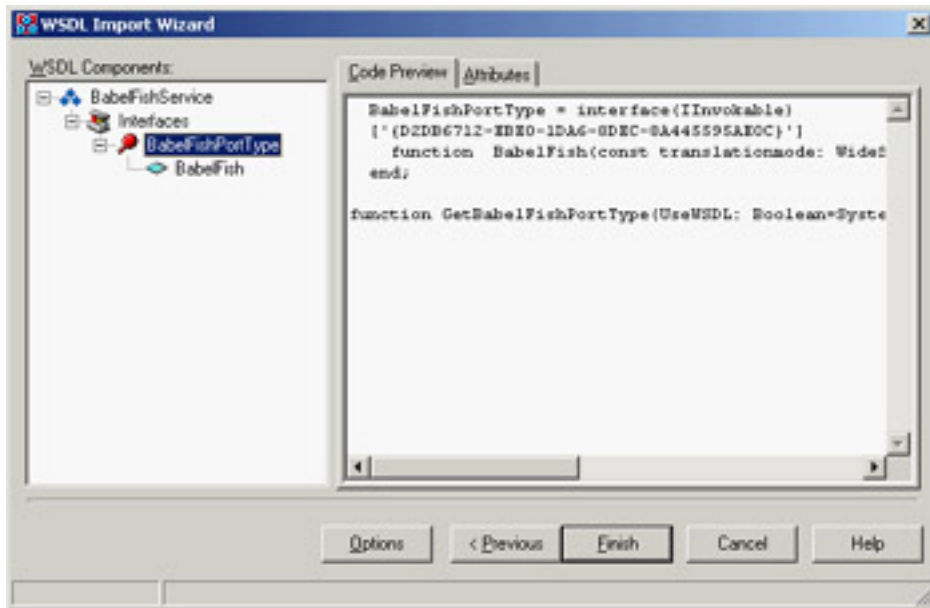


Figure 23.1: The WSDL Import Wizard in action

```
unit BabelFishService;
```

```
interface
```

```
uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;
```

```
type
```

```
  BabelFishPortType = interface(IInvokable)
  ['{D2DB6712-EBE0-1DA6-8DEC-8A445595AEOC}']
    function BabelFish(const translationmode: WideString;
      const sourcedata: WideString): WideString; stdcall;
  end;
```

```
function GetBabelFishPortType(UseWSDL: Boolean=System.False;
  Addr: string=''; HTTPRIO: THTTPRIO = nil): BabelFishPortType;
```

```
implementation
```

```
// omitted
```

```
initialization
```

```
  InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType),
    'urn:xmethodsBabelFish', '');
  InvRegistry.RegisterDefaultSOAPAction(TypeInfo(BabelFishPortType),
    'urn:xmethodsBabelFish#BabelFish');
```

```
end.
```

Notice that the interface inherits from the `IInvokable` interface. This interface doesn't add anything in terms of methods to Delphi's `IInterface` base interface, but it is compiled with the flag used to enable RTTI generation, `{ $M+ }`, like the `TPersistent` class. In the `initialization` section, you notice that the interface is registered in the global invocation registry (or `InvRegistry`), passing the type information reference of the interface type.

**Note** Having RTTI information for interfaces is the most important technological advance underlying SOAP invocation. Not that SOAP-to-Pascal mapping isn't important—it is crucial to simplify the process—but having RTTI for an interface makes the entire architecture powerful and robust.

The third element of the unit generated by the WSDL Import Wizard is a global function named after the service, introduced in Delphi 7. This function helps simplify the code used to call the web service. The `GetBabelFishPortType` function returns an interface of the proper type, which you can use to issue a call directly. For instance, the following code translates a short sentence from English into Italian (as indicated by the value of the first parameter, `en_it`) and shows it on screen:

```
ShowMessage (GetBabelFishPortType.BabelFish('en_it', 'Hello, world!'));
```

If you look at the code for the `GetBabelFishPortType` function, you'll see that it creates an internal invocation component of the class `THTTTPRIO` to process the call. You can also place this component manually on the client form (as I've done in the example program) to gain better control over its various settings (and handle its events).

This component can be configured in two basic ways: You can refer to the WSDL file or URL, import it, and extract from it the URL of the SOAP call; or, you can provide a direct URL to call. The example has two components that provide the alternative approaches (with exactly the same effect):

```
object HTTPRIO1: THTTTPRIO
  WSDLLocation = 'BabelFishService.xml'
  Service = 'BabelFish'
  Port = 'BabelFishPort'
end
object HTTPRIO2: THTTTPRIO
  URL = 'http://services.xmethods.net:80/perl/soaplite.cgi'
end
```

At this point, there is little left to do. You have information about the service that can be used for its invocation, and you know the types of the parameters required by the only available method as they are listed in the interface. The two elements are merged by extracting the interface you want to call directly from the `HTTTPRIO` component, with an expression like `HTTPRIO1 as BabelFishPortType`. It might seem astonishing at first, but it is also outrageously simple.

This is the web service call done by the example:

```
EditTarget.Text := (HTTPRIO1 as BabelFishPortType).
  BabelFish(ComboBoxType.Text, EditSource.Text);
```

The program output, shown in [Figure 23.2](#), allows you to learn foreign languages (although the teacher has its shortcomings!). I haven't replicated the same example with stock options, currencies, weather forecasts, and the many other services available, because they would look much the same.



**Figure 23.2:** An example of a translation from English to German obtained by Alta-Vista's BabelFish via a web service

**Warning**

Although the web service interface provides you with the types of the parameters, in many cases you need to refer to some actual documentation of the service to know what the values of the parameters really mean and how they are interpreted by the service. The BabelFish web service is an example of this issue, as I had to look at some textual documentation to find out the list of translation types, available in the demo in a combo box.

[Team LiB](#)[◀ PREVIOUS](#)[NEXT ▶](#)

## Building Web Services

If calling a web service in Delphi is straightforward, the same can be said of developing a service. If you go into the Web Services page of the New Items dialog box, you can see the SOAP Server Application option. Select it, and Delphi presents you with a list that's quite similar to what you see if you select a WebBroker application. A web service is typically hosted by a web server using one of the available web server extension technologies (CGI, ISAPI, Apache modules, and so on) or the Web App Debugger for your initial tests.

After completing this step, Delphi adds three components to the resulting web module, which is just a plain web module with no special additions:

- The HTTPSoapDispatcher component receives the web request, as any other HTTP dispatcher does.
- The HTTPSoapPascalInvoker component does the reverse operation of the HTTPRIO component; it can translate SOAP requests into calls of Pascal interfaces (instead of shifting interface method calls into SOAP requests).
- The WSDLHTMLPublish component can be used to extract the WSDL definition of the service from the interfaces it support, and performs the opposite role of the Web Services Importer Wizard. Technically, this is another HTTP dispatcher.

## A Currency Conversion Web Service

Once this framework is in place—something you can also do by adding the three components listed in the [previous section](#) to an existing web module—you can begin writing a service. As an example, I've taken the euro conversion example from [Chapter 3](#), "The Run-Time Library," and transformed it into a web service called ConvertService. First, I've added to the program a unit defining the interface of the service, as follows:

```
type
  IConvert = interface(IInvokable)
    ['{FF1EAA45-0B94-4630-9A18-E768A91A78E2}']
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

Defining an interface directly in code, without having to use a tool such as the Type Library Editor, provides a great advantage, as you can easily build an interface for an existing class and don't have to learn using a specific tool for this purpose. Notice that I've given a GUID to the interface, as usual, and used the `stdcall` calling convention, because the SOAP converter does not support the default `register` calling convention.

In the same unit that defines the interface of the service, you should also register it. This operation will be necessary on both the client and server sides of the program, because you will be able to include this interface definition unit in both:

```
uses InvokeRegistry;
```

### initialization

```
InvRegistry.RegisterInterface(TypeInfo(IConvert));
```

Now that you have an interface you can expose to the public, you have to provide an implementation for it, again by means of the standard Delphi code (and with the help of the predefined `TInvokableClass` class:

```
type
  TConvert = class (TInvokableClass, IConvert)
  protected
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
```

```

    stdcall;
function ToEuro (Source: string; Amount: Double): Double; stdcall;
function FromEuro (Dest: string; Amount: Double): Double; stdcall;
function TypesList: string; stdcall;
end;

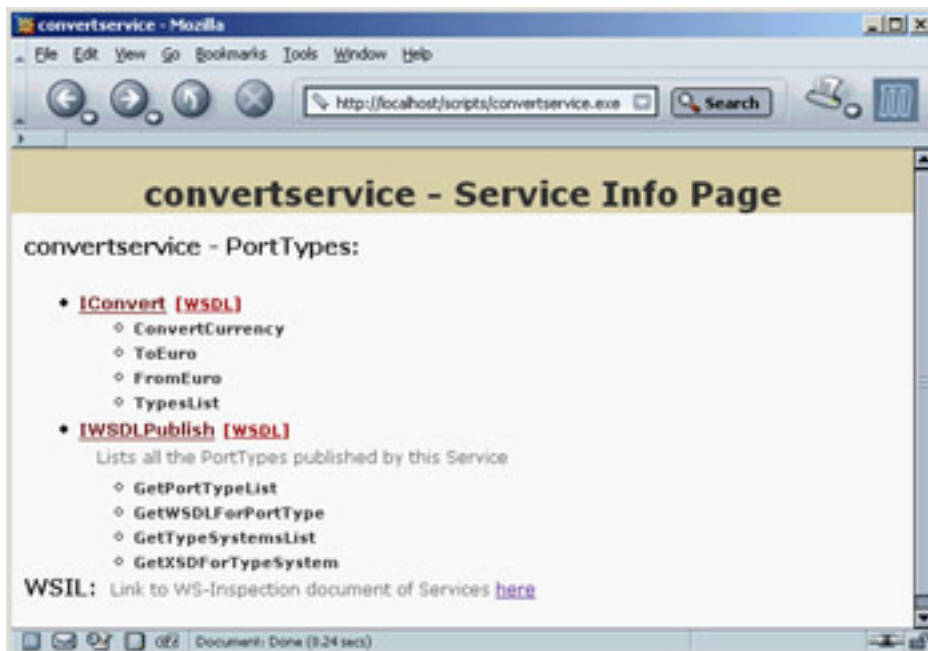
```

The implementation of these functions, which call the code of the euro conversion system from [Chapter 3](#), is not discussed here because it has little to do with the development of the service. However, it is important to notice that this implementation unit also has a registration call in its initialization section:

```
InvRegistry.RegisterInvokableClass (TConvert);
```

## Publishing the WSDL

By registering the interface, you make it possible for the program to generate a WSDL description. The web service application (since the Delphi 6.02 update) is capable of displaying a first page describing its interfaces and the detail of each interface, and returning the WSDL file. By connecting to the web service via a browser, you'll see something similar to [Figure 23.3](#).



**Figure 23.3:** The description of the Convert-Service web service provided by Delphi components

**Note** Although other web service architectures automatically provide you with a way to execute the web service from the browser, this technique is mostly meaningless, because using web services makes sense in an architecture where different applications interoperate. If all you need to do is show data on a browser, you should build a website!

This auto-descriptive feature was not available in web services produced in Delphi 6 (which provided only the lower-level WSDL listing), but it is quite easy to add (or customize). If you look at the Delphi 7 SOAP web module you'll notice a default action with an `OnAction` event handler invoking the following default behavior:

```
WSDLHTMLPublish1.ServiceInfo(Sender, Request, Response, Handled);
```

This is all you have to do to retrofit this feature into an existing Delphi web service that lacks it. To provide similar functionality manually, you must call into the invocation registry (the `InvRegistry` global object), with calls like `GetInterfaceExternalName` and `GetMethExternalName`.

What's important is the web service application's ability to document itself to any other programmer or programming tool, by exposing the WSDL.

## Creating a Custom Client

Let's move to the client application that calls the service. I don't need to start from the WSDL file, because I already have the Delphi interface. This time the form doesn't even have the HTTPRIO component, which is created in code:

```
private
    Invoker: THTTPrIo;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Invoker := THTTPrIo.Create(nil);
    Invoker.URL := 'http://localhost/scripts/ConvertService.exe/soap/iconvert';
    ConvIntf := Invoker as IConvert;
end;
```

As an alternative to using a WSDL file, the SOAP invoker component can be associated with an URL. Once this association has been done and the required interface has been extracted from the component, you can begin writing straight Pascal code to invoke the service, as you saw earlier.

A user fills the two combo boxes, calling the `TypesList` method, which returns a list of available currencies within a string (separated by semicolons). You extract this list by replacing each semicolon with a line break and then assigning the multiline string directly to the combo items:

```
procedure TForm1.Button2Click(Sender: TObject);
var
    TypeNames: string;
begin
    TypeNames := ConvIntf.TypesList;
    ComboBoxFrom.Items.Text := StringReplace (TypeNames, ';', sLineBreak,
        [rfReplaceAll]);
    ComboBoxTo.Items := ComboBoxFrom.Items;
end;
```

After selecting two currencies, you can perform the conversion with this code ([Figure 23.4](#) shows the result):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    LabelResult.Caption := Format ('%n', [(ConvIntf.ConvertCurrency(
        ComboBoxFrom.Text, ComboBoxTo.Text, StrToFloat(EditAmount.Text))]);
end;
```





**Figure 23.4:** The ConvertCaller client of the Convert-Service web service shows how few German marks you used to get for so many Italian liras, before the euro changed everything.

## Asking for Database Data

For this example, I built a web service (based on the Web App Debugger) capable of exposing data about employees of a company. This data is mapped to the EMPLOYEE table of sample InterBase database we've used so often throughout the book. The Delphi interface of the web service is defined in the SoapEmployeeIntf unit as follows:

```

type
  ISoapEmployee = interface (IInvokable)
    ['{77D0D940-23EC-49A5-9630-ADE0751E3DB3}']
    function GetEmployeeNames: string; stdcall;
    function GetEmployeeData (EmpID: string): string; stdcall;
  end;

```

The first method returns a list of the names of all the employees in the company, and the second returns the details of a given employee. The implementation of this interface is provided in the Soap-EmployeeImpl unit with the following class:

```

type
  TSoapEmployee = class(TInvokableClass, ISoapEmployee)
  public
    function GetEmployeeNames: string; stdcall;
    function GetEmployeeData (EmpID: string): string; stdcall;
  end;

```

The implementation of the web service lies in the two previous methods and some helper functions to manage the XML data being returned. But before we get to the XML portion of the example, let me briefly discuss the database access section.

## Accessing the Data

All the connectivity and SQL code in this example are hosted in a separate data module. Of course, I could have created some connection and dataset components dynamically in the methods, but doing so is contrary to the approach of a visual development tool like Delphi. The data module has the following structure:



```

object DataModule3: TDataModule3
  object SQLConnection: TSQLConnection
    ConnectionName = 'IBConnection'
    DriverName = 'Interbase'
    LoginPrompt = False
    Params.Strings = // omitted
  end
  object dsEmplList: TSQLDataSet
    CommandText = 'select EMP_NO, LAST_NAME, FIRST_NAME from EMPLOYEE'
    SQLConnection = SQLConnection
    object dsEmplListEMP_NO: TStringField
    object dsEmplListLAST_NAME: TStringField
    object dsEmplListFIRST_NAME: TStringField
  end
  object dsEmpData: TSQLDataSet
    CommandText = 'select * from EMPLOYEE where Emp_No = :id'
    Params = <
      item
        DataType = ftFixedChar
        Name = 'id'
        ParamType = ptInput
      end>
    SQLConnection = SQLConnection
  end
end

```

As you can see, the data module has two SQL queries hosted by SQLDataSet components. The first is used to retrieve the name and ID of each employee, and the second returns the entire set of data for a given employee.

## Passing XML Documents

The problem is how to return this data to a remote client program. In this example, I've used the approach I like best: I've returned XML documents, instead of working with complex SOAP data structures. (I don't get how XML can be seen as a messaging mechanism for SOAP invocation—along with the transport mechanism provided by HTTP—but then, it is not used for the data being transferred. Still, very few web services return XML documents, so I'm beginning to wonder if it's me or many other programmers who can't see the full picture.)

In this example, the `GetEmployeeNames` method creates an XML document containing a list of employees, with their first and last names as values and the related database ID as an attribute, using two helper functions `MakeXmlStr` (already described in the [last chapter](#)) and `MakeXmlAttribute` (listed here):

```

function TSoapEmployee.GetEmployeeNames: string;
var
  dm: TDataModule3;
begin
  dm := TDataModule3.Create (nil);
  try
    dm.dsEmplList.Open;
    Result := '<employeeList>' + sLineBreak;
    while not dm.dsEmplList.EOF do
      begin
        Result := Result + ' ' + MakeXmlStr ('employee',
          dm.dsEmplListLASTNAME.AsString + ' ' +
          dm.dsEmplListFIRSTNAME.AsString,
          MakeXmlAttribute ('id', dm.dsEmplListEMPNO.AsString)) + sLineBreak;
        dm.dsEmplList.Next;
      end;
  end;

```

```

    Result := Result + '</employeeList>';
finally
    dm.Free;
end;
end;

function MakeXmlAttribute (attrName, attrValue: string): string;
begin
    Result := attrName + '=' + attrValue + '';
end;

```

Instead of the manual XML generation, I could have used the XML Mapper or some other technology; but as you should know from [Chapter 22](#) ("Using XML Technologies"), I rather prefer creating XML directly in strings. I'll use the XML Mapper to process the data received on the client side.

**Note** You may wonder why the program creates a new instance of the data module each time. The negative side of this approach is that each time, the program establishes a new connection to the database (a rather slow operation); but the plus side is that you have no risk related to the use of a multithreaded application. If two web service requests are executed concurrently, you can use a shared connection to the database, but you must use different dataset components for the data access. You could move the datasets in the function code and keep only the connection on the data module, or have a global shared data module for the connection (used by multiple threads) and a specific instance of a second data module hosting the datasets for each method call.

Let's now look at the second method, `GetEmployeeData`. It uses a parametric query and formats the resulting fields in separate XML nodes (using another helper function, `FieldsToXml`):

```

function TSoapEmployee.GetEmployeeData(EmpID: string): string;
var
    dm: TDataModule3;
begin
    dm := TDataModule3.Create (nil);
    try
        dm.dsEmpData.ParamByName('ID').AsString := EmpID;
        dm.dsEmpData.Open;
        Result := FieldsToXml ('employee', dm.dsEmpData);
    finally
        dm.Free;
    end;
end;

function FieldsToXml (rootName: string; data: TDataSet): string;
var
    i: Integer;
begin
    Result := '<' + rootName + '>' + sLineBreak;;
    for i := 0 to data.FieldCount - 1 do
        Result := Result + ' ' + MakeXmlStr (
            LowerCase (data.Fields[i].FieldName),
            data.Fields[i].AsString) + sLineBreak;
    Result := Result + '</' + rootName + '>' + sLineBreak;;
end;

```

## The Client Program (with XML Mapping)

The final step for this example is to write a test client program. You can do so as usual by importing the WSDL file defining the web service. In this case, you also have to convert the XML data you receive into something more

manageable—particularly the list of employees returned by the `GetEmployeeNames` method. As mentioned earlier, I've used Delphi's XML Mapper to convert the list of employees received from the web service into a dataset I can visualize using a DBGrid.

To accomplish this, I first wrote the code to receive the XML with the list of employees and copied it into a memo component and from there to a file. Then, I opened the XML Mapper, loaded the file, and generated from it the structure of the data packet and the transformation file. (You can find the transformation file among the source code files of the SoapEmployee example.) To show the XML data within a DBGrid, the program uses an XMLTransformProvider component, referring to the transformation file:

```
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'EmplListToDataPacket.xtr'
end
```

The ClientDataSet component is not hooked to the provider, because it would try to open the XML data file specified by the transformation. In this case, the XML data doesn't reside in a file, but is passed to the component after calling the web service. For this reason the program moves the data to the ClientDataSet directly in code:

```
procedure TForm1.btnGetListClick(Sender: TObject);
var
  strXml: string;
begin
  strXml := GetISoapEmployee.GetEmployeeNames;
  strXML := XMLTransformProvider1.TransformRead.TransformXML(strXml);
  ClientDataSet1.XmlData := strXml;
  ClientDataSet1.Open;
end;
```

With this code, the program can display the list of employees in a DbGrid, as you can see in [Figure 23.5](#). When you retrieve the data for the specific employee, the program extracts the ID of the active record from the ClientDataSet and then shows the resulting XML in a memo:

```
procedure TForm1.btnGetDetailsClick(Sender: TObject);
begin
  Memo2.Lines.Text := GetISoapEmployee.GetEmployeeData(
    ClientDataSet1.FieldName ('id').AsString);
end;
```

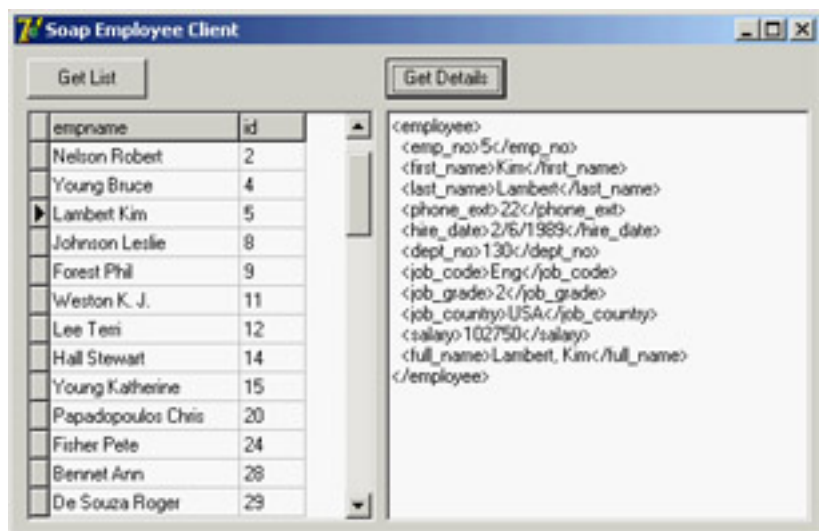
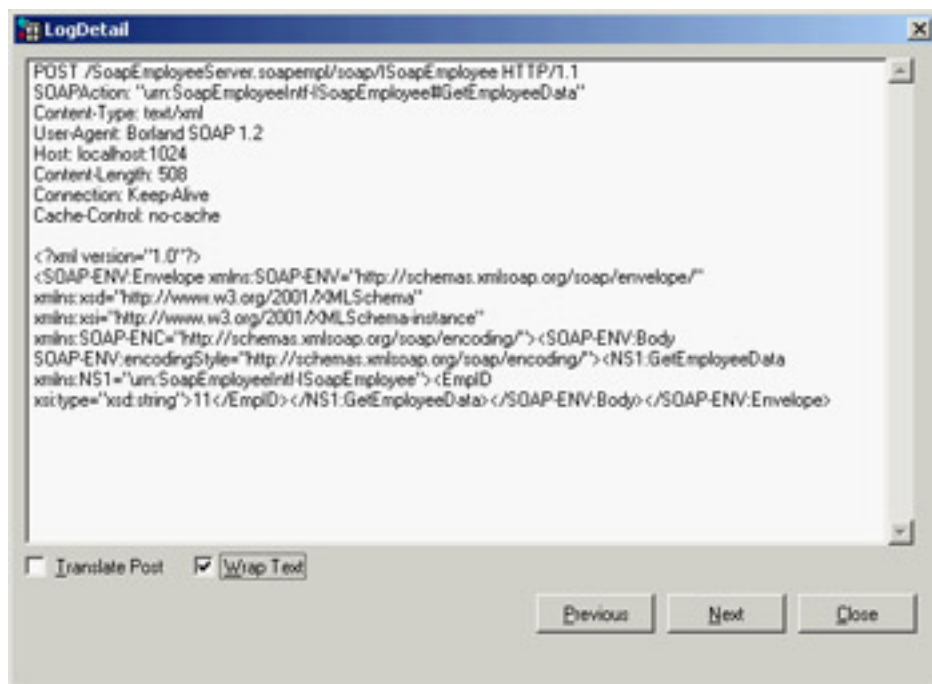


Figure 23.5: The client program of the SoapEmployee web service example

## Debugging the SOAP Headers

One final note for this example relates to the use of the Web App Debugger for testing SOAP applications. Of course, you can run the server program from the Delphi IDE and debug it easily, but you can also monitor the SOAP headers passed on the HTTP connection. Although looking at SOAP from this low-level perspective can be far from simple, it is the ultimate way to check if something is wrong with either a server or a client SOAP application. As an example, in [Figure 23.6](#) you can see the HTTP log of a SOAP request from the last example.



**Figure 23.6:** The HTTP log of the Web App Debugger includes the low-level SOAP request.

The Web App Debugger might not always be available, so another common technique is to handle the events of the HTTPRIO component, as the BabelFishDebug example does. The program's form has two memo components in which you can see the SOAP request and the SOAP response:

```
procedure TForm1.HTTPRIO1BeforeExecute(const MethodName: String;
var SOAPRequest: WideString);
begin
    MemoRequest.Text := SoapRequest;
end;
```

```
procedure TForm1.HTTPRIO1AfterExecute(const MethodName: String;
    SOAPResponse: TStream);
begin
    SOAPResponse.Position := 0;
    MemoResponse.Lines.LoadFromStream(SOAPResponse);
end;
```

## Exposing an Existing Class as a Web Service

Although you might want to begin developing a web service from scratch, in some cases you may have existing code to make available. This process is not too complex, given Delphi's open architecture in this area. To try it, follow these steps:

1. Create a web service application or add the related components to an existing WebBroker project.
2. Define an interface inheriting from `IInvokable` and add to it the methods you want to make available in the web service (using the `stdcall` calling convention). The methods will be similar to those of the class you want to make available.

3. Define a new class that inherits from the class you want to expose and implements your interface. The methods will be implemented by calling the corresponding methods of the base class.
4. Write a factory method to create an object of your implementation class any time a SOAP request needs it.

This last step is the most complex. You could define a factory and register it as follows:

```
procedure MyObjFactory (out Obj: TObject);  
begin  
    Obj := TMyImplClass.Create;  
end;
```

#### **initialization**

```
InvRegistry.RegisterInvokableClass(TMyImplClass, MyObjFactory);
```

However, this code creates a new object for every call. Using a single global object would be equally bad: Many different users might try to use it, and if the object has state or its methods are not concurrent, you might be in for big problems. You're left with the need to implement some form of session management, which is a variation on the problem we had with the earlier web service connecting to the database.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## DataSnap over SOAP

Now that you have a reasonably good idea how to build a SOAP server and a SOAP client, let's look at how to use this technology in building a multitier DataSnap application. You'll use a Soap Server Data Module to create the new web service and the SoapConnection component to connect a client application to it.

### Building the DataSnap SOAP Server

Let's look at the server side first. Go to the Web Services page of the New Items dialog box and use the Soap Server Application icon to create a new web service, and then use the Soap Server Data Module icon to add a DataSnap server-side data module to the SOAP server. I did this in the SoapDataServer7 example (which uses the Web App Debugger architecture for testing purposes). From this point on, all you do is write a normal DataSnap server (or a middle-tier DataSnap application), as discussed in [Chapter 16](#) ("Multitier DataSnap Applications"). In this case, I added InterBase access to the program by means of dbExpress, resulting in the following structure:

```
object SoapTestDm: TSoapTestDm
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
  end
end
```

The data module built for a SOAP-based DataSnap server defines a custom interface (so you can add methods to it) inheriting from `IAppServerSOAP`, which is defined as a published interface (even though it doesn't inherit from `IInvokable`).

**Tip** Delphi 6 used DataSnap's standard `IAppServer` interface for exposing data via SOAP. Delphi 7 has replaced the default with the inherited `IAppServerSOAP` interface, which is functionally identical but allows the system to discriminate the type of call depending on the interface name. You'll see shortly how to call an older application from a client built in Delphi 7, because this process is not automatic.

The implementation class, `TSoapTestDm`, is the data module, as in other DataSnap servers. Here is the code Delphi generated, with the addition of the custom method:

```
type
  ISampleDataModule = interface(IAppServerSOAP)
    ['{D47A293F-4024-4690-9915-8A68CB273D39}']
    function GetRecordCount: Integer; stdcall;
  end;

  TSampleDataModule = class(TSoapDataModule, ISampleDataModule,
    IAppServerSOAP, IAppServer)
    DataSetProvider1: TDataSetProvider;
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
  public
    function GetRecordCount: Integer; stdcall;
  end;
```

The base `TSoapDataModule` doesn't inherit from `TInvokableClass`. This is not a problem as long as you provide

an extra factory procedure to create the object (which is what `TInvokableClass` does for you) and add it to the registration code (as discussed earlier, in the section "[Exposing an Existing Class as a Web Service](#)):

```
procedure TSampleDataModuleCreateInstance(out obj: TObject);  
begin  
    obj := TSampleDataModule.Create(nil);  
end;  
  
initialization  
    InvRegistry.RegisterInvokableClass(  
        TSampleDataModule, TSampleDataModuleCreateInstance);  
    InvRegistry.RegisterInterface(TypeInfo(ISampleDataModule));
```

The server application also publishes the `IAppServerSOAP` and `IAppServer` interfaces, thanks to the (little) code in the `SOAPMidas` unit. As a comparison, you can find a SOAP `DataSnap` server built with Delphi 6 in the `SoapDataServer` folder. The example can still be compiled in Delphi 7 and works fine, but you should write new programs using the structure of the newer; an example is in the `SoapDataServer7` folder.

**Tip** Web service applications in Delphi 7 can include more than one SOAP data module. To identify a specific SOAP data module, use the `SOAPServerIID` property of the `SoapConnection` component or add the data module interface name to the end of the URL.

The server has a custom method (the Delphi 6 version of the program also had one, but it never worked) that uses a query with the `select count(*) from EMPLOYEE` SQL statement:

```
function TSampleDataModule.GetRecordCount: Integer;  
begin  
    // read in the record count by running a query  
    SQLDataSet2.Open;  
    Result := SQLDataSet2.Fields[0].AsInteger;  
    SQLDataSet2.Close;  
end;
```

## Building the DataSnap SOAP Client

To build the client application, called `SoapDataClient7`, I began with a plain program and added a `SoapConnection` component to it (from the Web Services page of the palette), hooking it to the URL of the `DataSnap` web service and referring to the specific interface I'm looking for:

```
object SoapConnection1: TSoapConnection  
    URL = 'http://localhost:1024/SoapDataServer7.soapdataserver/' +  
        'soap/Isampledatabodule'  
    SOAPServerIID = 'IAppServerSOAP - {C99F4735-D6D2-495C-8CA2-E53E5A439E61}'  
    UseSOAPAdapter = False  
end
```

Notice the last property, `UseSOAPAdapter`, which indicates you are working against a server built with Delphi 7. As a comparison, the `SoapDataClient` (again, no 7) example, which uses a server created with Delphi 6 and recompiled with Delphi 7, must have this property set to `True`. This value forces the program to use the plain `IAppServer` interface instead of the new `IAppServerSOAP` interface.

From this point on, you proceed as usual, adding a `ClientDataSet` component, a `DataSource`, and a `DBGrid` to the program; choosing the only available provider for the client dataset; and hooking the rest. Not surprisingly, for this simple example, the client application has little custom code: a single call to open the connection when a button is clicked (to avoid startup errors) and an `ApplyUpdates` call to send changes back to the database.

## SOAP versus Other DataSnap Connections

Regardless of the apparent similarity of this program to all the other DataSnap client and server programs built in [Chapter 16](#), there is a very important difference worth underlining: The SoapDataServer and SoapDataClient programs do not use COM to expose or call the `IAppServerSOAP` interface. Quite the opposite—the socket- and HTTP-based connections of DataSnap still rely on local COM objects and a registration of the server in the Windows Registry. The native SOAP-based support, however, allows for a totally custom solution that's independent of COM and that offers many more chances to be ported to other operating systems. (You can recompile this server in Kylix, but not the programs built in [Chapter 16](#).)

The client program can also call the custom method I've added to the server to return the record count. This method could be used in a real-world application to show only a limited number of records but inform the user how many haven't yet been downloaded from the server. The client code to call the method relies on an extra HTTPRIO component:

```
procedure TFormSDC.Button3Click(Sender: TObject);  
var  
    SoapData: ISampleDataModule;  
begin  
    SoapData := HttpRio1 as ISampleDataModule;  
    ShowMessage (IntToStr (SoapData.GetRecordCount));  
end;
```

**Team LiB**

◀ PREVIOUS

NEXT ▶



## Handling Attachments

One of the most important features Borland added to Delphi 7 is full support for SOAP attachments. Attachments in SOAP allow you to send data other than XML text, such as binary files or images. In Delphi, attachments are managed through streams. You can read or indicate the type of attachment encoding, but the transformation of a raw stream of bytes into and from a given encoding is up to your code. This process isn't too complex, though, if you consider that Indy includes a number of encoding components.

As an example of how to use attachments, I've written a program that forwards the binary content of a ClientDataSet (also hosts images) or one of the images alone. The server has the following interface:

```
type
  ISoapFish = interface(IInvokable)
  [ '{4E4C57BF-4AC9-41C2-BB2A-64BCE470D450}' ]
    function GetCds: TSoapAttachment; stdcall;
    function GetImage(fishName: string): TSoapAttachment; stdcall;
end;
```

The implementation of the GetCds method uses a ClientDataSet that refers to the classic BIOLIFE table, creates a memory stream, copies the data to it, and then attaches the stream to the TSoapAttachment result:

```
function TSoapFish.GetCds: TSoapAttachment; stdcall;
var
  memStr: TMemoryStream;
begin
  Result := TSoapAttachment.Create;
  memStr := TMemoryStream.Create;
  WebModule2.cdsFish.SaveToStream(memStr); // binary
  Result.SetSourceStream(memStr, soReference);
end;
```

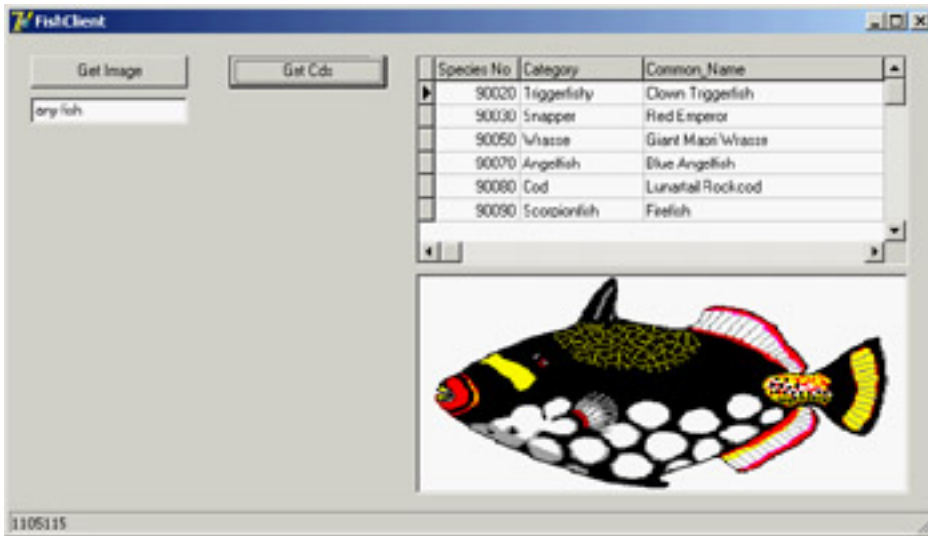
On the client side, I prepared a form with a ClientDataSet component connected to a DBGrid and a DBImage. All you have to do is grab the SOAP attachment, save it to a temporarily in-memory stream, and then copy the data from the memory stream to the local ClientDataSet:

```
procedure TForm1.btnGetCdsClick(Sender: TObject);
var
  sAtt: TSoapAttachment;
  memStr: TMemoryStream;
begin
  nRead := 0;
  sAtt := (HttpRiol as ISoapFish).GetCds;
  try
    memStr := TMemoryStream.Create;
    try
      sAtt.SaveToStream(memStr);
      memStr.Position := 0;
      ClientDataSet1.LoadFromStream(memStr);
    finally
      memStr.Free;
    end;
  finally
    DeleteFile(sAtt.CacheFile);
    sAtt.Free;
  end;
end;
```

**Warning** By default, SOAP attachments received by a client are saved to a temporary file, referenced by the *CacheFile* property of the *TSOAPAttachment* object. If you don't delete this file it will remain in a folder that hosts temporary files.

This code produces the same visual effect as a client application loading a local file into a *ClientDataSet*, as you can see in [Figure 23.7](#). In this SOAP client I used an *HTTPRIO* component explicitly to be able to monitor the incoming data (which will possibly be very large and slow); for this reason, I set a global *nRead* variable to zero before invoking the remote method. In the *OnReceivingData* event of the *HTTPRIO* object's *HTTPWebNode* property, you add the data received to the *nRead* variable. The *Read* and *Total* parameters passed to the event refer to the specific block of data sent over a socket, so they are almost useless by themselves to monitor progress:

```
procedure TForm1.HTTPRIO1HTTPWebNode1ReceivingData(  
    Read, Total: Integer);  
begin  
    Inc (nRead, Read);  
    StatusBar1.SimpleText := IntToStr (nRead);  
    Application.ProcessMessages;  
end;
```



**Figure 23.7:** The FishClient example receives a binary *ClientDataSet* within a SOAP attachment.

## Supporting UDDI

The increasing popularity of XML and SOAP opens the way for business-to-business communication applications to interoperate. XML and SOAP provide a foundation, but they are not enough—standardization in the XML formats, in the communication process, and in the availability of information about a business are all key elements of a real-world solution.

Among the standards proposed to overcome this situation, the most notable are Universal Description, Discovery, and Integration (UDDI, [www.uddi.org](http://www.uddi.org)) and Electronic Business using eXtensible Markup Language (ebXML, [www.ebxml.org](http://www.ebxml.org)). These two solutions partially overlap and partially diverge and are now being further worked on by the OASIS consortium (Organization for the Advancement of Structured Information Standards, [www.oasis-open.org](http://www.oasis-open.org)). I won't get into the problems with business processes; instead I'll only discuss some of the technical elements of UDDI, because it is specifically supported by Delphi 7.

### What Is UDDI?

The Universal Description, Discovery, and Integration (UDDI) specification is an effort to create a catalog of web services offered by companies throughout the world. The goal of this initiative is to build an open, global, platform-independent framework to let business entities find each other, define how they interact with the Internet network, and share a global business registry. Of course, the idea is to speed up the adoption of e-commerce, in the form of B2B applications.

UDDI is basically a global business registry. Companies can register themselves on the system, describing their organization and the web services they offer (in UDDI the term *web services* is used in a very wide sense, including e-mail addresses and websites). The information in the UDDI registry for each company is divided into three areas:

**White Pages** Include contact information, addresses, and the like.

**Yellow Pages** Register the company in one or more taxonomies, including industrial categories, products sold by the company, geographical information, and other (possibly customizable) taxonomies.

**Green Pages** List the web services offered by the company. Each service is listed under a service type (called a *tModel*), which can be predefined or a type specifically described by the company (for example, in terms of WSDL).

Technically, the UDDI registry should be perceived like today's DNS, and should have a similar distributed nature: multiple servers, mirrored and caching data. Clients can cache data following given rules.

UDDI defines specific data models for a business entity, a business service, and a binding template. The *BusinessEntity* type includes core information about the business, such as its name, the category it belongs to, and contact information. It supports the taxonomies of the yellow pages, with industry information, product types, and geographic details.

The *BusinessService* type includes descriptions of web services (used for the green pages). The main type is only a container for the related services. The services can be bound to a taxonomy (geographical area, product, and so on). Every *BusinessService* structure includes one or more *BindingTemplates* (the reference to the service).

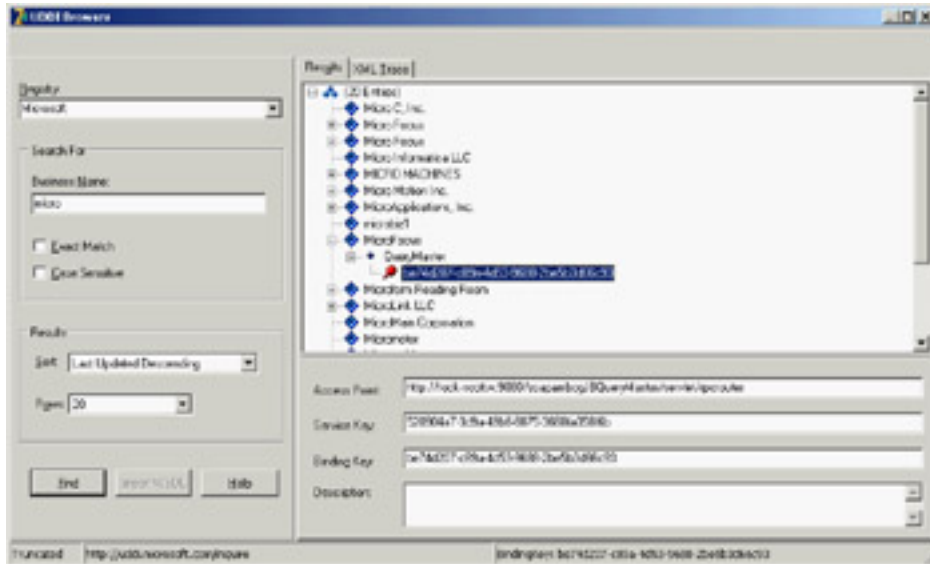
The *BindingTemplate* has a *tModel*. The *tModel* includes information about formats, protocols, and security, and references to technical specifications (possibly using the WSDL format). If multiple companies share a *tModel*, a program can interact with all of them with the same code. A given business program, for example, can offer a *tModel* for other software programs to interact with it, regardless of the company that has adopted the software.

The UDDI API is based on SOAP. Using SOAP, you can both register data and query a registry. Microsoft also offers a COM-based SDK, and IBM has a Java Open Source toolkit for UDDI. UDDI APIs include inquiry (*find\_xx* and

get\_xx) and publishing (save\_xx and delete\_xx) on each of the four core data structures (businessEntity, businessService, bindingTemplate, and tModel).

## UDDI in Delphi 7

Delphi 7 includes a UDDI browser you can use to find a web service while importing a WSDL file. The UDDI browser, shown in [Figure 23.8](#), is activated by the WSDL Import Wizard. This browser uses only UDDI version 1 servers (a newer interface is available, but it isn't supported) and has a few predefined UDDI registries. You can add predefined settings in the UDDIBrow.ini file in the Delphi bin folder.



**Figure 23.8:** The UDDI Browser embedded in the Delphi IDE

This is a handy way to access web services information, but it is not all that Delphi provides. Although the UDDI Browser is not available as a stand-alone application, the UDDI interface units are available (and they are not trivial to import). So, you can write your own UDDI browser.

I'll sketch a simple solution, which is a starting point for a full-blown UDDI browser. The UddiInquiry example, shown in action in [Figure 23.9](#), has a number of features, but not all of them work smoothly (particularly the category search features). The reason lies in the fact that using UDDI implies navigating complex data structures, which are not always mapped in the most obvious way by the WSDL importer. This makes the example code fairly involved; so, I'll show you only the code for a plain search, and not even all of it (another reason is that some readers may not be particularly interested in UDDI).

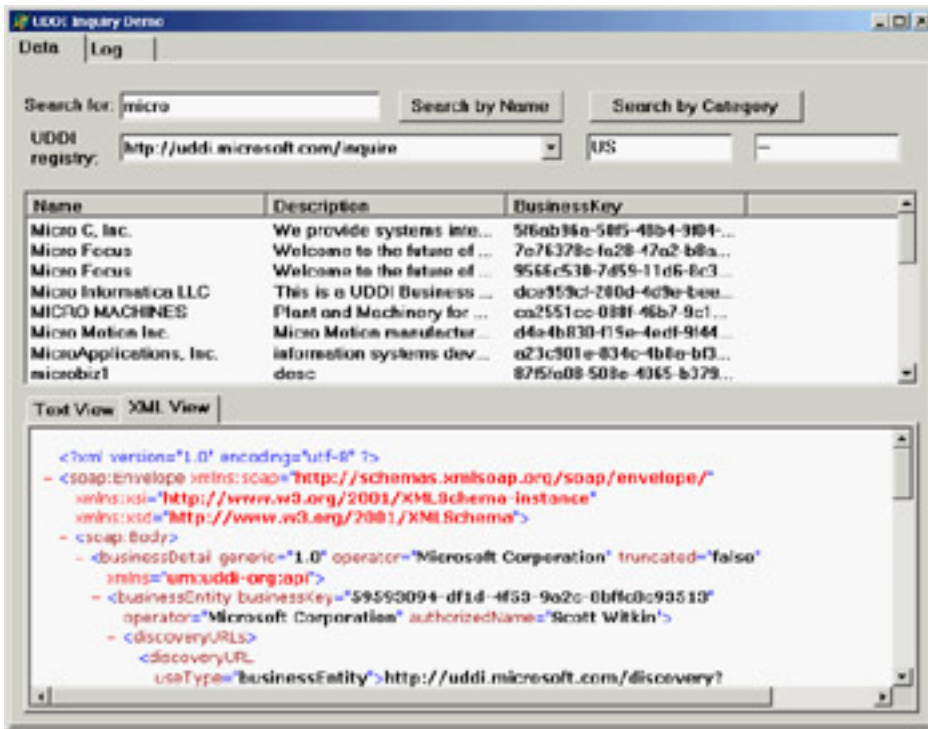


Figure 23.9: The UddiInquiry example features a limited UDDI browser.

As the program starts, it binds the HTTPRIO component it hosts with the InquireSoap UDDI interface, defined in the inquire\_v1 unit provided with Delphi 7:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    httprio1.Url := comboRegistry.Text;
    inftInquire := httprio1 as InquireSoap;
end;

```

Clicking the Search button makes the program call the find\_business UDDI API. Because most UDDI functions require many parameters, it has been imported using a single record-based parameter of type FindBusiness; it returns a businessList2 object:

```

procedure TForm1.btnSearchClick(Sender: TObject);
var
    findBusinessData: Findbusiness;
    businessListData: businessList2;
begin
    httprio1.Url := comboRegistry.Text;

    findBusinessData := FindBusiness.Create;
    findBusinessData.name := edSearch.Text;
    findBusinessData.generic := '1.0';
    findBusinessData.maxRows := 20;

    businessListData := inftInquire.find_business(findBusinessData);
    BusinessListToListView (businessListData);
    findBusinessData.Free;
end;

```

The businessList2 object is a list that is processed by the businessListToListView method of the program's main form, showing the most relevant details in a list view component:

```

procedure TForm1.businessListToListView(businessList: businessList2);
var
    i: Integer;

```

```
begin
  ListView1.Clear;
  for i := 0 to businessList.businessInfos.Len do
  begin
    with ListView1.Items.Add do
    begin
      Caption := businessList.businessInfos [i].name;
      SubItems.Add (businessList.businessInfos [i].description);
      SubItems.Add (businessList.businessInfos [i].businessKey);
    end;
  end;
end;
```

By double-clicking on a list view item you can further explore its details, although the program shows the resulting XML information in a plain textual format (or a `TWebBrowser`-based XML view) and doesn't further process it. As mentioned, I don't want to get into technical details; if you're interested, you can find them by looking at the source code.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

In this chapter, I've focused on web services, covering SOAP, WSDL, and UDDI. Refer to the W3C website and to the UDDI ([www.uddi.org](http://www.uddi.org)) and ebXML ([www.ebxml.org](http://www.ebxml.org)) sites for more information in the area of business-oriented web services. I didn't delve much into these non-technical issues, but they were worth mentioning.

You should have noticed in this chapter that Delphi is strong player in the area of web services, with a powerful, open architecture. You can use web services to interact with applications written for the Microsoft .NET platform; Delphi has much to offer for this architecture, because it includes a Delphi for .NET preview (covered in the two final chapters of the book).

[Chapter 24](#), "The Microsoft .NET Architecture from the Delphi Perspective," is focused on the .NET platform; [Chapter 25](#), "Delphi for .NET Preview: The Language and the RTL," covers the new Delphi compiler.

## Chapter 24: The Microsoft .NET Architecture from the Delphi Perspective

### Overview

Every few years a new technology comes along that turns our industry upside down. Some of these technologies thrive, some mutate into something else, and many are revealed to be half-baked marketing fluff. Almost in military fashion, the arrival of a new technology is invariably preceded by an artillery barrage of hype. Experienced programmers know to keep their heads down until the hype barrage subsides. After all, the proof will be in the technical pudding, so to speak.

Your reaction to Microsoft's .NET initiative will depend somewhat on your background. If you have previous experience with Java or Delphi, you might wonder what all the fuss is about. But, if you have been slogging it out in the trenches writing applications for Windows in C++ (or, heaven forbid, in C), you might be overwhelmed with joy.

The purpose of this chapter is to explain some of the technologies that make up the .NET initiative and to show how they fit into the world of Windows programming in general and Delphi programming in particular. We will begin by installing and configuring the Delphi for .NET Preview compiler. Then we will broadly cover some of the .NET technologies. Finally, we will discuss these technologies in greater detail, weaving in some Delphi code for illustration.

**Note** This chapter and the following one were by Marco with extensive help from John Bushakra, who works for Borland's RAD tools documentation department.



## Installing the Delphi for .NET Preview

The Delphi for .NET Preview requires but does not install .NET Framework Runtime, which is freely downloadable from Microsoft's MSDN website. As a developer, you'll probably want to take an extra step and install the more complete .NET Framework SDK, which includes documentation and tools for developers (but is also much larger). You should install the .NET Framework Runtime or SDK before you install the Delphi for .NET Preview.

The Preview compiler is compatible with the .NET Framework SDK and service pack 1. If you have applied service pack 2 or later, then you will have to perform the extra step of rebuilding the precompiled units (the `dcuil` files) installed with the Preview. This requirement might go away in subsequent updates to the Delphi for .NET Preview compiler.

**Note** A few months after Delphi 7 shipped (in November 2002), Borland released a significant update of the Delphi for .NET Preview. As a registered Delphi user, you can download updates to the Preview from the Borland website. Do this even before installing the version that comes in the Delphi box, as you'll need to uninstall it to update to a newer version.

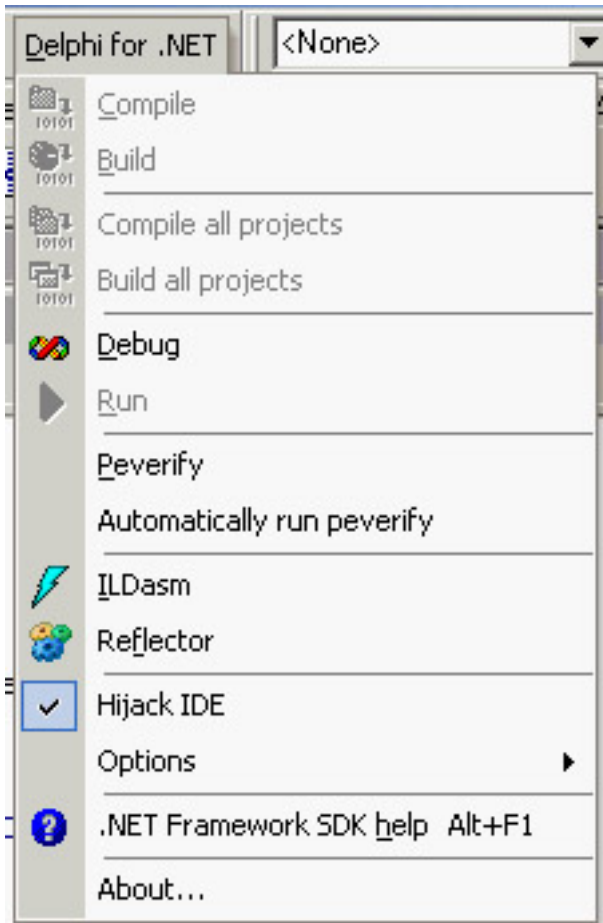
**Warning** I've used the November 2002 version of the Delphi for .NET Preview to test the examples of this and the .NET chapter, although most of them will also compile against the original version that shipped with Delphi 7. By the time you read this, a newer version might be available; check the author's website for updates of the examples.

After installing the .NET Framework SDK, insert the Delphi for .NET Preview CD and run the setup program. The Preview will install itself in a separate directory from your Delphi 7 installation, and none of your Delphi 7 settings are affected by installing the Preview.

Previously I mentioned that you would need to rebuild the precompiled units shipped with the compiler, if you have installed service pack 2 for the .NET Framework. If this applies to you, open a command window and navigate to the `source\rtl` directory under the root of the compiler installation directory. There, you will find a file called `rebuild.bat`, which you must execute to perform the rebuild. You might see some errors and warnings; these are known issues at the time of this writing, and might be cleared up in subsequent updates to the compiler. Once the batch file has completed, you will be ready to go.

The `dccil` compiler is located in the `bin` directory. Along with the compiler, in this directory is a file called `dccil.cfg` that specifies the default unit search path (the `-U` compiler switch), which is the `units` directory under the root installation directory. As packaged, the Delphi for .NET Preview compiler is strictly for use with a command window.

However, Borland has made available an unsupported plug-in to the Delphi 7 IDE on the Borland Developers Network website, referred to as Delphi for .NET common line compiler IDE integration. Using the plug-in, you can control the `dccil` compiler from within the IDE, as you can see in the menu installed by this plug-in.



Note, however, that the plug-in does not give you the full capability of designing forms and all the other things Delphi is famous for. The Delphi for .NET Preview compiler is intended to give you some advance warning about new language features and a glimpse into how the Delphi run-time library might look in the .NET context. You can download this plug-in from the Code Central area of the Borland Developer Network website, [bdn.borland.com](http://bdn.borland.com) (if you cannot find it, look under the number 18889).

Another valuable resource you might want to work with is the Reflector. This tool was written by Lutz Roeder (who happens to work at Microsoft, by the way) and is available on his own website, [www.aisto.com/roeder/dotnet](http://www.aisto.com/roeder/dotnet).

**Note** Reflector is like Microsoft's Intermediate Language Disassembler (ILDASM) tool—it allows you to inspect .NET assemblies (executables and DLLs) and their types and members.

## Testing the Installation

Now it's time to test your Delphi for .NET Preview installation by compiling a simple console program that prints a message.

You can start this project using Delphi 7, or you can type in the text with your favorite editor:

```
program HelloWorld;  
  
{ $APPTYPE CONSOLE }  
  
uses  
  Borland.Delphi.SysUtils;  
  
begin  
  WriteLn ('Hello, Delphi! - Today is ' +  
    DateToStr (Now));  
end.
```

Notice how similar the code looks to the Delphi applications you're used to. The only difference is the `uses` clause. Borland has organized the units of the Delphi library into namespaces similar to those of the Common Language Runtime (CLR). I will discuss this in more detail in the [next chapter](#).

Save the file as `HelloWorld.dpr` and open a command window. Navigate to the location of the file and type  
`dccil HelloWorld.dpr`

The result should be an executable image, `HelloWorld.exe`, which you can run from the command line to produce a single line of output. If you are using the IDE plug-in, compiling and running the program will involve pressing `Ctrl+F9` or `F9` after you've enabled IDE *hijacking* (see the Hijack IDE menu item in the screenshot above).

If this program seems boring, you can begin your exploration of .NET using the windowed version of the hello world program instead of the console version. Called `HelloWin`, this program creates and shows a window on the screen, writing into its caption. It also shows the call to `Application.Run`, used to activate the application's message processing loop:

```
program HelloWin;  
  
uses  
    System.Windows.Forms,  
    Borland.Delphi.SysUtils;  
  
var  
    aForm: Form;  
  
begin  
    aForm := Form.Create;  
    aForm.Text := 'Hello Delphi';  
    Application.Run (aForm);  
end.
```

In .NET presentations, the previous code raises the enthusiasm of programmers used to Microsoft development tools. As a Delphi programmer, you might have used almost this same code since 1995, so you might wonder where this excitement comes from. The output of these programs is far from interesting, but you can at least prove they are not classic Win32 applications by running `ILDASM` on them (directly from the plug-in's menu). You can see the output for the `HelloWorld` executable in [Figure 24.1](#). Notice that the unit's global code is wrapped in a pseudo class called `uUnit`. Unlike Delphi, the .NET runtime doesn't account for global procedures and functions, but only class methods. But I'm getting ahead of myself; let's start from the beginning and look at the NET platform.

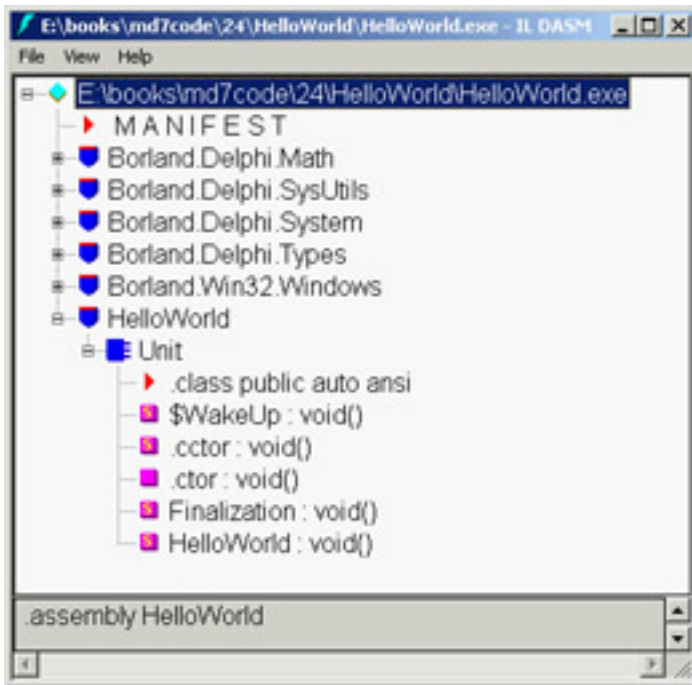


Figure 24.1: The HelloWorld demo as seen in ILDASM

## Microsoft's .NET Platform

Microsoft's .NET platform comprises many different specifications and initiatives. The core functionality of the .NET platform has been given over to the European Computer Manufacturer's Association (ECMA) and is undergoing the standardization process. At the time of this writing, the C# language specification has just passed ECMA's process and is heading for ISO standardization.

**Note** You can find the standard documents for the C# programming language and the Common Language Infrastructure (CLI) at [www.ecma.ch](http://www.ecma.ch). An interesting element of the CLI specification is the standard naming convention for variables and methods.

From the programmer's standpoint, the core feature of the .NET platform is its managed environment, which allows the execution of intermediate code compiled by many different programming languages (provided they conform to a common definition of the base data types). This environment embeds many features ranging from sophisticated memory management up to integrated security, to name just two. On top of this managed environment, Microsoft has built a large class library, covering diverse areas of development (Windows-based forms, web development, web services development, XML processing, database access, and many more).

This is only a short overview. To get more into the details we need to learn the precise terms used in the .NET Platform, most of which are indicated by three-letter acronyms, introduced in the following subsections.

### The Common Language Infrastructure (CLI)

The CLI is the main pillar of the .NET platform. It was submitted to the ECMA in December 2001. The CLI comprises many different specifications:

**Common Type System (CTS)** The CTS lays the groundwork for integrating disparate programming languages. It specifies how types are declared and used, and language compilers must conform to this specification in order to take advantage of the .NET platform's cross-language integration.

**Extensible Metadata** The CLI specifies that every unit of deployment (an *assembly*) must be a self-describing entity. Therefore, an assembly must carry with it data that fully identifies the assembly (its name, version, and optional culture and public key), data that describes the types defined within the assembly, data listing other files referenced, and any special security permissions that are required. Furthermore, the metadata system is extensible, so an assembly might also contain user-defined descriptive elements, known as custom attributes.

**Note** The term culture is used by Microsoft as an extension of the term's language (the language used for the user messages of a library) and locale (the dates and number formatting settings for a country). The idea is to cover anything peculiar to a country, or portions of a country.

**Common Intermediate Language (CIL)** CIL is the programming language of a virtual execution environment with an abstract microprocessor. Compilers targeting the .NET platform do not generate native CPU instructions, but instead generate instructions in the intermediate language of the abstract processor. In this respect, the CLI is similar to Java byte code.

**P/Invoke** Programs that execute in the .NET runtime each play in their own private sandbox. Unlike the traditional Win32 programming model, a large and complicated layer of software exists between them and the operating system. But the .NET runtime does not completely replace the Win32 API, so there must be a way to bridge between the two worlds. This bridge is called Platform Invocation Service (in short, P/Invoke or PInvoke).

**Framework Class Library (FCL)** The .NET Framework Class Library (FCL), or .NET Framework for short (or even .NET Fx), is a class hierarchy with a design similar to that of Borland's VCL. Features also available in the VCL are actually quite similar, although the .NET Framework has a much larger

set of classes (covering many more areas of programming). The architectural difference between the .NET Framework and the VCL is that the .NET Framework is not only callable from other languages, it can be directly extended by other languages. This means that, as a Delphi programmer, you have the ability to directly inherit from classes in the .NET Framework, just as you would inherit from any other Delphi class. Moreover, the CLI gives you the ability to extend a class written in any language that has a compiler that targets the .NET runtime. The *factorable* part of the FCL means that parts of the class hierarchy can be factored out; for example, to create a stripped-down version for use on a handheld device.

**Extended Portable Executable (PE) File Format** Microsoft is using its standard Portable Executable (PE) file format (the file format used by standard Win32 executable files) to support the metadata requirements of the CLI. The advantage of using this format is that the operating system can load a .NET application the same way it loads a native Win32 application. Common loading is about where the similarity ends though, since all managed code is in a special section. The framework modifies the loader so when it finds it is dealing with a .NET entity, it passes control over to the CLR that then works out how to call the managed entry points.

## The Common Language Runtime (CLR)

The CLI is a specification, and the CLR is Microsoft's implementation of that specification. Not surprisingly, the CLR is a superset of the specification. To a programmer, the CLR is an all-encompassing run-time library that unifies the vast array of services provided by the Windows operating system and presents them to you in an object-oriented framework.

On a larger scale, the CLR is responsible for every aspect of the .NET universe: loading the executable image, verifying its identity and type safety, compiling CIL code into native CPU instructions, and managing the application during its lifetime. CIL code that is intended to be run in the CLR is called *managed code*, whereas all other code (such as Intel executable code produced by Delphi 7) is *unmanaged*.

## Common Language Specification (CLS)

Closely related to the Common Type System, the CLS is a subset of that specification that defines the rules that govern how types created in different programming languages can interoperate. Not all languages are created equal, and some languages have features that can't be found elsewhere. The CLS tries to find a happy medium, specifying those items that a large set of languages can implement. Microsoft tried to make the CLS as small as possible, yet still accommodate a large set of languages.

Some features of Delphi are not CLS compliant. This does not mean the code cannot be executed by the CLR and work on the .NET platform; it only means you are using a language feature that can't be supported by other languages, so that particular part of your code cannot be used by other .NET applications if they are written in languages other than Delphi.

**Note** If you have experience with Java, you might find some of these items familiar. The .NET platform shares many concepts with the Java platform, in particular the intermediate language and virtual execution system, although with the core difference of Java being pcode interpreted by default, whereas .NET is invariably JIT compiled. There are also relevant analogies in the class libraries. In most cases, however, don't take corresponding Java concepts for granted, because differences in the implementation details might significantly affect how you use an apparently similar feature.

The various specifications of the CLI give a glimmer of hope about cross-platform development. But, you probably won't hear the "write once, run anywhere," mantra from Microsoft. This is due to the fact that they consider the user interface to be a key part of an application, and a normal PC screen compared to, say, a mobile phone screen has inherently different capabilities. There are two major ongoing efforts to implement the CLI on other operating systems. The Rotor project (officially called Microsoft Shared Source CLI or MS SSCLI) is written by the Microsoft Research team. Rotor consists of a large donation of software, under Microsoft's Shared Source license, and the necessary tools to build it on the FreeBSD, Win2, and Mac OS X 10.2 operating systems.

The second well-known CLI implementation on another operating system is the Mono Project, which supports Win32 and Linux. Mono is building what is essentially a clean-room implementation of the CLI on Linux and is released with a

much more open license. Borland has a wait-and-see approach about Mono; there has been no word about whether Delphi will be a player in that arena.

The Rotor project seems to be aimed toward educators and people who are just curious about how the CLR is implemented, because its license is very open to academia but prohibits any commercial use. The Mono project, however, may give Microsoft some serious competition. The most serious stumbling block to true portability will probably continue to be the graphical user interface. The Rotor project does not include any GUI elements, while Mono has started developing WinForms using the WINE library. There is also a GTK# project associated with Mono, which is a binding of the GTK+ toolkit to the C# language.

Today's computing environment is a nebulous mass of diverse possibilities. Handheld devices have complex operating systems of their own, and Microsoft has a keen interest in this arena. Microsoft is working on a version of the .NET platform called the .NET Compact Framework, which is intended for handheld devices. The .NET platform could also serve as a springboard for tomorrow's technologies. Sixty-four bit processors are just around the corner, and .NET will no doubt be running there.

Does this mean you can run your managed application on Linux, 64-bit Windows, and your PDA? No. It is not reasonable to expect a user interface geared for a 1600×1200 pixel display on a 21-inch monitor to port to a handheld device. So, regarding .NET as a cross-platform tool, you will gain some advantage, but you should not expect your application to be a straight port especially if you are porting to another operating system or a handheld device.

In the following sections, we will examine some components of the CLI in more detail.

## Assemblies

Microsoft coined the term *assembly* to denote a "unit of deployment" in the .NET runtime environment. In manufacturing, an assembly is a group of separate but related parts that are put together to form a single functional unit. Sometimes an assembly consists of only one part, but it might consist of many. So it is with assemblies in .NET.

Typically, an assembly consists of only one file—either a single executable or a dynamic link library. An assembly could consist of a group of related DLLs, or a DLL and associated resources like images or strings. Going back to the manufacturing analogy, each assembly, whether it consists of one part or multiple parts, contains a packing slip called a *manifest*. A manifest describes the contents of something, and an assembly manifest contains the metadata for an assembly. The manifest is the implementation of the extensible metadata system described in the CLI.

As you look through the installation directory for the Delphi for .NET Preview compiler, you will find a `units` folder. In this folder are a number of files with the extensions `.dcua` and `.dcuil`. These are not PE files, so they cannot be examined with ILDASM.

A `.dcua` file catalogs the namespaces and types in a .NET assembly (which, keep in mind, can consist of multiple executable files). A `.dcuil` file corresponds to a namespace. The `.dcuil` file contains all the compiler symbols for a namespace, as well as references to the `.dcua` files that contribute to that namespace (an assembly can contribute types to more than one namespace).

**Note** A *namespace* is a hierarchical containment mechanism that organizes classes and other types. We will discuss namespaces in more detail in [Chapter 25](#), "Delphi for .NET Preview: The Language and the RTL."

An application is built against a certain set of assemblies; the application is said to *reference* these assemblies. When the set of assemblies changes, the compiler must rebuild any `.dcuil` files that contain references to `.dcua` files that are no longer in the set. Likewise, if a change is made to an assembly, the compiler must rebuild the corresponding `.dcua` and `.dcuil` files for that assembly. A `.dcuil` file is roughly equivalent to a Delphi `.dcu` file, but a `.dcua` file does not have a corresponding file type in the Delphi for Win32 universe.

Different project types produce different types of .NET assemblies: A Delphi `program` produces an executable assembly; a `library` project produces a DLL assembly.



## The Intermediate Language

The Common Language Runtime is an implementation of a virtual execution system, or virtual machine. Like all virtual machines, the CLR has its own abstract microprocessor. As already mentioned, the assembly language of the virtual processor is called Common Intermediate Language (CIL), although before being promoted as a standard it was called Microsoft Intermediate Language (MSIL), a terms you'll still see around often. Compilers that target the CLR do not generate code in the native instruction set of any specific, real microprocessor. Instead, .NET compilers target the abstract processor of the CLR. The hardware abstraction built into the CLR hints at some cross-platform viability. Remember, Microsoft's CLR is but one implementation of the CLI; any hardware/operating system platform that has a CLI-compliant execution environment built for it could be targeted.

Of course, there is no real microprocessor that can execute CIL directly, so it must be compiled to the instruction set of the native hardware prior to execution. This is the job of the Just in Time (JIT) compiler. Here is where the CLR differs from other virtual machine implementations (like Java). The CLR is not an interpreter, nor does it execute bytecode. On the .NET platform, CIL is always compiled to native CPU instructions, and once compiled it is cached in memory; so chances are good that it will never have to be recompiled.

**Note** In some memory-constrained environments (such as a PDA), compiled code can be discarded. In this case, the code would need to be recompiled if it was ever reloaded.

Compiling IL is not a very expensive operation (MS Research has spent years developing technology to allow the JIT compilation to be as negligible as possible) but does imply a little overhead and it must be repeated every time you run even the same program. Most applications will see a little increase in startup time (what's particularly slow is loading all the .NET framework itself with the first .NET application run in a Windows session); however, this is limited because not all code in an application is compiled at once. The JIT compiler works in conjunction with the loader, and so IL code is not compiled until it is called (on a method by method basis).

JIT compilation is the normal case on the .NET platform, but it is possible to compile a managed executable into native instructions and store the native image on disk. By doing so, you avoid the negative impact of JIT compilation on your application's startup time. The .NET Framework runtime contains a utility called the Native Image Generator (`Ngen.exe`) to accomplish this.

The native image created by Ngen is stored in the native image cache. The next time the CLR tries to load the assembly, it looks in the native image cache first. If a native image of the assembly is found, it is preferred over the IL version. Note that you must also deploy the IL version of the assembly, because the native image does not contain any metadata. In addition, the end user or administrator could remove your native image from the cache. In this case, there would be no native image to find, and the CLR would revert back to the usual JIT compilation of the IL version.

The ability to generate a native executable image can be helpful, but you should profile your application under both environments (JIT and native image) to see whether the detrimental effects of the JIT compiler are that bad. The JIT compiler is, after all, a real compiler for a specific microprocessor, and as such it can do some performance optimizations of its own. It employs good algorithms to reduce its overhead and also introduces optimizations to the compiled code (somewhat like Delphi does in its compilations).

Looking over the IL code generated by your compiler can be highly educational. Microsoft provides a utility called the IL Disassembler (`ildasm.exe`) that you can use to dissect your assemblies at the lowest level. Located in the `bin` directory of your .NET Framework SDK installation, `ildasm` can load any assembly and its metadata: the manifest, the classes, their methods and properties, and, of course, the IL code generated by the compiler. We will look more at ILDASM in this chapter and the next. The Reflector tool mentioned earlier is also useful in this regard.

## Managed and Safe Code

Simply put, *managed code* is any code that is loaded, JIT compiled, and executed under the auspice of the CLR. Like all executable and library modules on the Windows platform, managed code modules are stored in Microsoft's Portable Executable (PE) file format. A managed PE file contains additional header information and, when loaded, jumps into the runtime's execution engine (to a function in `MSCorEE.dll`).



The runtime initializes, and then it looks for the module's entry point. The IL code in the entry point is JIT compiled to native CPU instructions. Finally, execution begins at the module's entry point. The situation is similar for a library module; the PE file directs the loader to jump to a different function in `MSCorEE.dll`.

In contrast, *unmanaged code* consists not of IL, but rather of traditional, native CPU instructions. Unmanaged code executes outside of the runtime and therefore can't take advantage of the services provided by the CLR—at least not without special measures.

Unmanaged code can create .NET Framework classes using COM Interop services. The .NET Framework class is wrapped in a COM proxy and exposed to unmanaged code as if it were a COM object. The COM Interop bridge goes the other way too, allowing a COM objects in a COM server to be accessed by managed code. Finally, the Platform Invoke services of the CLR allow managed code to call the Win32 API directly.

**Note** The Delphi for .NET Preview compiler produces fully managed code. There is currently no support for mixing managed and unmanaged (native) code within the same module, as you can with Microsoft's Visual C++ .NET (which uses a mechanism called IJW: It Just Works).

A module is completely self-describing, because it contains both IL code and metadata that describes the data elements used by the code. Taking the IL code and the metadata together, the CLR can perform another level of verification beyond the static checking done by the compiler. This process, which is always performed unless a system administrator turns it off, verifies that code is type safe. Verifiably type-safe code is known as *safe code*. Safe code passes the following type-safety checks:

- Only valid operations are invoked on objects. This includes parameter validation, return type verification, and visibility checks.
- Objects are always assigned to compatible types.
- The code uses no explicit pointers, as they might refer to invalid memory locations.

As you'd expect, *unsafe code* fails to pass these checks. But, just because code is not verifiably type safe does not mean it is unsafe; it simply means the code could not be verified, either due to a limitation in the verification process, or perhaps in the compiler itself. When it will be released as a finished product, the Delphi for .NET compiler is expected to generate verifiably type-safe code.

Some Delphi language constructs are not CLS compliant, but this is different than not being verifiably type safe. Non-CLS-compliant language constructs will be covered more in [Chapter 25](#).

The .NET Framework SDK contains a PEVerify utility that exhaustively analyzes a managed PE for type safety (`peverify.exe`). The Delphi 7 IDE plug-in mentioned previously lets you automatically run PEVerify on your code after every build.

## The Common Type System

The Common Type System (CTS) is the bulldozer that levels the playing field for programming languages in the .NET framework. The CTS fully specifies the primitive types and object types known to the CLR. These types are used to define an object model that is shared among all languages that target the CLR.

The Component Object Model (COM) has been the usual way to achieve binary compatibility and language interoperability on the Windows platform. The CTS goes beyond that, allowing languages as different as Eiffel, C#, and Delphi to integrate with each other. Components written in these disparate languages can pass objects among themselves and directly extend their capabilities through inheritance. This level of integration of programming languages is unprecedented.

All types defined by the CTS fall into two categories: value types and reference types. Value types, as their name implies, have pass-by-value semantics. For example, say you have a variable that is a value type. If you pass this variable as a parameter to a function and modify the parameter within the function, the original variable will be

unaffected. Examples of value types include scalar types, enumerations, and records. Aggregate types such as Delphi records (or C# structures) are known as value classes within the CTS.

On the other hand, reference types have alias, or pass-by-reference, semantics. If you have a variable that is a reference type (for example, an instance of a class) and you pass that variable as a parameter to a function, any changes you make to the parameter will also affect the variable. Examples of reference types include class types and interfaces. Pointer types are also reference types, as are delegates, which will be discussed shortly.

## Objects and Properties

Like Delphi, the CTS implements a single-inheritance model. A class must inherit from one and only one ancestor and may declare itself to be an implementer of zero or more interfaces. Other familiar object-oriented attributes of the CTS are private, public, and protected visibility of classes and class members (with other visibility specifiers available, as discussed in the [next chapter](#)). These CTS visibility specifiers have meanings similar to those in Delphi; however, they are more restricted, in line with C++ semantics. (In [Chapter 25](#) we'll look at how Delphi's visibility specifiers map to the CTS versions, and examine the specific changes made to the language thus far to accommodate additional features of the CTS.)

As you peruse the .NET literature, you will notice similarities between the capabilities of CTS class types and Delphi classes. The traditional object-oriented features of fields and methods are supported, of course. In addition, the CTS implements properties in a way that is conceptually similar to the familiar Delphi notion. Properties in CTS can have read and write access methods that restrict or compute values on the fly, or they can simply mask private fields. However there are also many differences, including the fact that property get/set methods must have the same visibility as the property itself to ensure languages that don't support property syntax can still access the property. Although Delphi doesn't enforce this in the source code, it modifies compiled code behind the scenes if necessary.

## Events and Delegates

One reason the Win32 API has survived so long is that at its lowest levels it is based on fundamental concepts, such as using the address of a function as a callback mechanism. The entire Windows user interface event system is based on callback functions (and some of events in the VCL framework are built on top of even that system). The callback mechanism is so powerful that it surely must make its way into the CTS. Using callbacks in a type-safe, language-neutral way relies on a reference type called a *delegate*.

CTS delegates are different from ordinary function pointers, in that they can reference both static and instance methods of a class. The declaration of a delegate must match the signature of the methods the delegate will reference. In Delphi for .NET, usage of delegates is similar to that of familiar procedural types:

```
type
  TMyClass = class
  public
    procedure myMethod;
  end;

var
  threadDelegate: System.Threading.ThreadStart;
  tmc: TMyClass;
  aThread: System.Threading.Thread;

begin
  tmc := TMyClass.Create;
  threadDelegate := @tmc.myMethod;
  aThread := Thread.Create (threadDelegate);
  aThread.Start;
```

The `threadDelegate` variable is of type `System.Threading.ThreadStart`, which is a CLR delegate class. Methods you assign to a delegate have a signature matching the delegate's, which in this case is a procedure taking no parameters. (You can find this code snippet in the `Delegate` sample folder.)

The compiler is hiding a lot of complexity here. Behind the scenes, the compiler must create an instance of the class `System.MulticastDelegate`. The delegate—the function being encapsulated (`myMethod` in this case)—is invoked using methods on the `MulticastDelegate` class. This class supports the simultaneous encapsulation of multiple functions in a single delegate. In a user interface event model, this translates to having multiple listeners for an event.

**Note** Incidentally, the fact that delegates are classes tells you why you can declare a delegate outside the scope of a class. Because delegates are instances of `System.MulticastDelegate` (or a compiler-generated descendant class), you can declare them anywhere you can declare a class.

Each specific language compiler implements some form of semantic sugar to make the creation of events and the addition and removal of event listeners less painful. For example, in C#, Microsoft used the `+=` and `-=` operators to add and remove functions from the underlying delegate. Delphi for .NET uses set semantics for the same purpose. In [Chapter 25](#) we'll explore this topic, showing how the functions `Include` and `Exclude` are used to assign event handlers. We'll also look at how Delphi's `:=` assignment operator works, with regard to assigning event handlers in the .NET universe.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## Garbage Collection

The garbage collector is that part of the CLR that performs automatic management of memory allocation and deallocation. All Delphi 7 offers in this respect is reference counting for interface-based variables (see [Chapter 2](#) for details). When there are no more references to an object, the memory for it is reclaimed. This is also the basic idea of the CLR garbage collector, but the similarities stop there. The CLR's garbage collector can detect that two objects are still referring one to the other, but there is no other reference to them, so they can be discarded, something a reference counting system (like the one in Delphi) fails to handle.

Using a GC means you can create objects, reference one from another as you need, and simply forget about the hassles of deleting those objects. The system will do all that's needed for you. That's all you really need to know. Note that this implies that you don't need to *balance* object creation and destruction in your code architecture, nor do you need to use `try/finally` blocks to make sure objects are destroyed; these are just a couple of specific circumstances in which you'll benefit from a GC.

Only when you need to write low-level classes will you have to remember to free unmanaged resources, such as window or file handles. The CLR class `System.Object` contains a protected `Finalize` method that you can override to free up unmanaged resources. You must be aware of some important things before you begin overriding `Finalize`. (If you want some in-depth information on this topic, see the sidebar "[Issues with Overriding Finalize.](#)")

---

### Issues with Overriding Finalize

Overriding the `Finalize` method is inefficient because it requires your object to make at least two trips through the garbage collector. The job of the garbage collector is to reclaim memory, and it can't do this until it runs your object's `Finalize` method. So, the garbage collector must give special treatment to all objects with `Finalize` methods. It puts them on a special list, which has the side effect of keeping them alive through this round of garbage collection (because there is now another reference to the object). Eventually, a thread walks through the special list, runs each object's `Finalize` method, and removes the object from the list. Removing the object from the list takes away the last reference to it, so next time the garbage collector runs, the object's memory can be reclaimed.

But inefficiency is not the only problem with finalizers. You also can never be sure when your object's finalizer will run. If you are relying on the `Finalize` method to give back unmanaged resources, the object might hold onto these resources longer than you expect.

When `Finalize` runs, it does not run in the same thread of execution as your object. This means all thread synchronization and blocking must be avoided, because it's not your thread you are blocking, it's the CLR's finalizer thread. On a related note, if an exception is thrown from `Finalize` the CLR catches it, not you, and the exception will be swallowed.

---

The recommended approach for freeing up unmanaged resources is to implement the dispose pattern. This is a strategy for providing an object both an automatic way to dispose the external resources it refers to when it is garbage collected and also a way to free the same resources upon the direct invocation of a method. In real terms, coding this pattern equates to implementing an interface called `IDisposable`. `IDisposable` consists of one method: `Dispose`. The `IDisposable` interface gives you deterministic control over freeing up resources used in your objects. Unlike `Finalize`, the `Dispose` method is public and is meant to be called by you (or users of your class), not by the garbage collector.

You may have read .NET literature explaining how the C# language implements destructor semantics by having the compiler create a `Finalize` method on your class behind the scenes. To implement the `IDisposable` interface in C#, you must do so directly and wire everything so that the `Dispose` method can be called directly or from the automatically generated `Finalize` method. This is not the way things work in Delphi for .NET.

Creating a destructor for your class does not cause the compiler to implement a finalizer behind the scenes. Instead, it causes the compiler to implement the `IDisposable` interface. But nothing is stopping you from implementing `IDisposable` directly, so if you want to rely on the compiler's behind-the-scenes magic, you must follow a strict pattern. Your class destructor must be declared exactly as shown here:

```
destructor Destroy; override;
```

When the compiler sees this declaration, it generates code to mark your class as an implementor of `IDisposable`. Then, using a feature of CIL, your destructor is marked as the implementation of the `Dispose` method (this can be done even though the name of the method is `Destroy`, not `Dispose`).

The usual way to dispose of objects in Delphi is to call the `Free` method on the object. In Delphi for .NET, the `Free` method is implemented so that it tests to see whether the object implements the `IDisposable` interface. If you follow the previous destructor signature, the compiler implements it for you; `Free` then calls `Dispose`, which winds up in your `Destroy` method.

Let's create a project and declare a class with a destructor that follows the pattern. Then you'll use ILDASM to inspect the generated code. The code is available in [Listings 24.1](#) and [24.2](#).

### **Listing 24.1: The Project of the DestructorTest Example**

---

```
program DestructorTest;  
  
{ $APPTYPE CONSOLE }  
  
uses  
  MyClass in 'MyClass.pas';  
  
var  
  test : TMyClass;  
  
begin  
  WriteLn ('DestructorTest starts');  
  test := TMyClass.Create;  
  test.Free;  
  WriteLn ('DestructorTest ends');  
end.
```

---

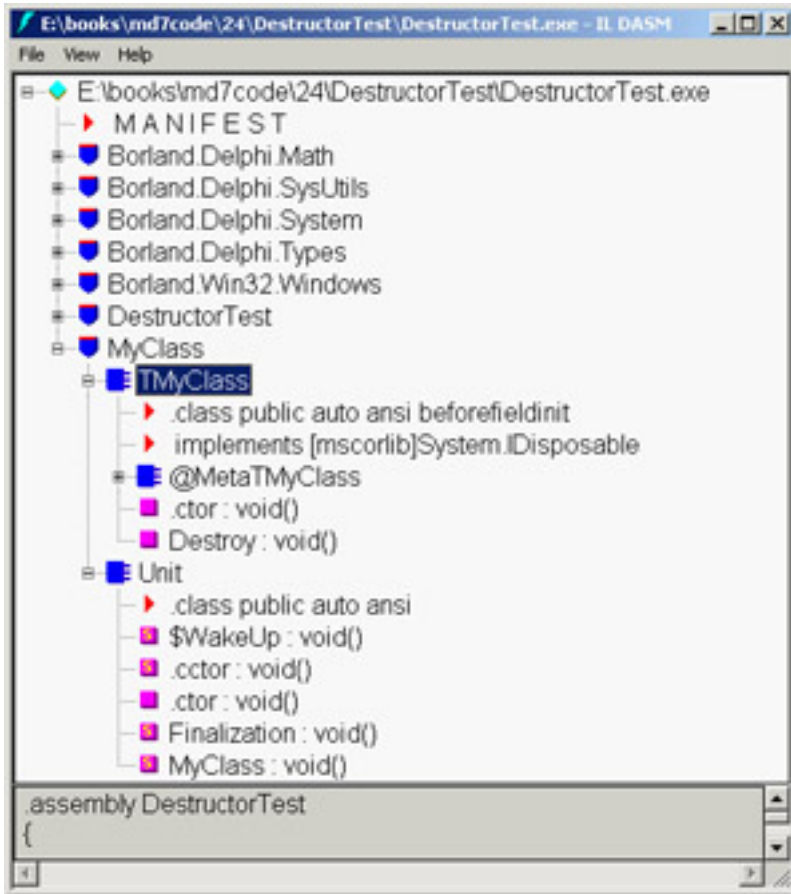
### **Listing 24.2: The Unit of the DestructorTest Example**

---

```
unit MyClass;  
  
interface  
  
type  
  TMyClass = class  
    public  
      destructor Destroy; override;  
    end;  
  
implementation  
  
destructor TMyClass.Destroy;  
begin  
  WriteLn ('In destructor (which is actually the IDisposable.Dispose method)');  
end;
```

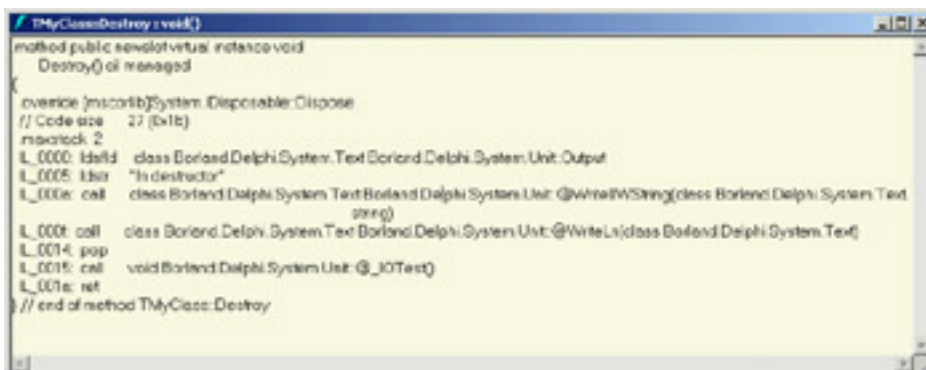
end.

After compiling this program you can run it; but we are not so much interested in running the program as in inspecting it with ILDASM. Start ILDASM and select File | Open. Navigate to the directory where `DestructorTest.exe` is located, and open it. You will see a window similar to that shown in [Figure 24.2](#).



**Figure 24.2:** The ILDASM output for the DestructorTest example

The tree entry labeled `MyClass` represents the namespace created to hold all the symbols in the unit. Expand the `MyClass` node and notice the entries for `TMyClass` and `Unit`. Delphi for .NET creates a CLR class for each unit to implement initialization and finalization and also to let you still write global routines; they become methods of that `Unit` class. Expand the node for `TMyClass` and notice the `Destroy` node, which has a pink block. This node represents the `Destroy` method in `TMyClass`. Double-click the `Destroy` node to open a window displaying the full IL code for this method, as shown in [Figure 24.3](#).





**Figure 24.3:** The ILDASM IL window for the `Destroy` destructor

The line you're looking for uses the `.override` directive:

```
.override [mscorlib]System.IDisposable::Dispose
```

This is an explicit override, and it says that the `Destroy` method is the implementation of `Dispose` in the `IDisposable` interface. The `.override` directive is how the thread of execution winds up in your destructor when `Dispose` is called from `Free`.

**Note** The way `IDisposable` was dealt with illustrates a familiar pattern in the ongoing work on the Delphi for .NET compiler. Unlike C#, Delphi is an old language with a large and loyal following and a sizable existing code base. Borland could not sweep everything aside and mandate a new way of doing things that would break all existing code. The implementation of `IDisposable` was done in a way that should allow Delphi programmers to continue using familiar paradigms and minimize the impact of porting their existing code to the .NET platform.

## Garbage Collection and Efficiency

Among programmers, the use of a garbage collector (GC) is the topic of many debates. Most programmers like the fact that a GC helps them reduce the chance of memory management errors. Some programmers, though, fear that a GC might not be efficient enough, a fear that often prevented the widespread use of this technology. The inefficient GC of the early Java virtual machines didn't really help in this respect. Even Microsoft has a hard time convincing all programmers to use its GC, up to the point that they overrate their solution depicting it as perfect. Even in Microsoft technical documentation about the GC, you'll find lots of hype and little facts.

**Note** This is not to say that the GC doesn't do an adequate job; quite the contrary. But it works differently than it is presented. At present, your only way to know how the GC works is to write test case programs and study their effect.

As a starting point for your exploration, you can use the `GarbageTest` example. Use a Windows memory analysis tool to see how the memory is affected by program execution. The `GarbageTest` example declares the following class of large objects (about 10 KB):

```
type
  TMyClass = class
  private
    data: Integer;
    list: array [1..10240] of Char;
    s: string;
  public
    constructor Create;
  end;
```

The simplest test code is the following:

```
for i := 1 to 10000 do
begin
  mc := TMyClass.Create;
  WriteLn (mc.s);
end;
```

Writing this simple program in Delphi will flood the memory, because there is no call to `Free`. In .NET, however, the GC reuses the single memory block, so memory consumption is flat. If you save each object in an array declared as `objlist: array [1..10000] of TMyClass;`

the memory consumption will increase at each cycle of the loop, causing problems! Keeping the object references in memory and then releasing them regularly or randomly can provide more complex tests. You'll find a few snippets in

the program code, but you'll have to exercise your knowledge and imagination to build other significant test cases.

**Team LiB**

◀ PREVIOUS

NEXT ▶



## Deployment and Versioning

Most Windows software developers have at some point felt the pain caused by deploying shared libraries. The .NET Framework can adapt to a number of different deployment scenarios. In the simplest scenario, the end user can copy files to a directory and go. When it's time to remove the application from the machine, the end user can remove the files, or the entire directory, if it doesn't contain any data. In this scenario there is no installer, no uninstaller, and no registry to deal with.

**Tip** You can package your application for the Microsoft Installer if you want to and still deploy to a single directory, with no Registry impact.

Outside of the easy scenario, the deployment options become complex and a bit bewildering, because the problem itself is complicated. How do you enable developers to deploy new versions of shared components, while at the same time ensuring that the new version won't break an application that depends on behavior provided by a prior version? The only answer is to develop a system that allows differing versions of the same component to be deployed *side-by-side*.

The first thing you need to learn about deployment in the .NET Framework is that the days of deploying shared components to `C:\Windows\System32` are gone. Instead, the .NET Framework includes two kinds of assemblies, well-defined rules governing how the system searches for them, and an infrastructure that supports the deployment of multiple versions of the same assembly on the same machine.

Before talking about the two kinds of assemblies, I will explain the two kinds of deployment:

**Public, or Global** A publicly deployed assembly is one you intend to share either among your own applications or with applications written by other people. The .NET Framework specifies a well-known location in which you are to deploy such assemblies.

**Private** A privately deployed assembly is one you do not intend to share. It is typically deployed to your application's base installation directory.

The type of assembly you create depends on how you intend to deploy it. The two types of assemblies are as follows:

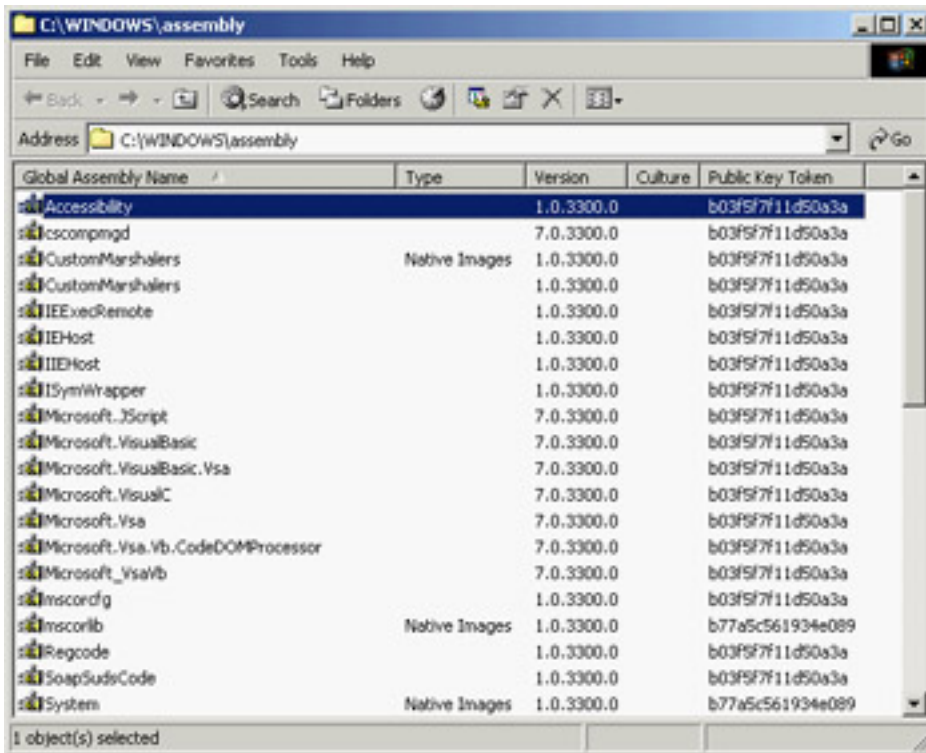
**Strong Name Assemblies** A strong name assembly is digitally signed. The signature consists of the assembly's identity (name, version, and optional culture plus a key pair with public and private components). All global assemblies must be strong name assemblies. The name, version, and culture are attributes of the assembly, stored in its metadata. How you create and update these attributes depends largely on your development tools.

**Everything Else** There is no official name for an assembly that does not have a strong name. An assembly that does not have a strong name can only be deployed privately, and it cannot take advantage of the side-by-side versioning features of the CLR.

The key pair that completes the signing of a strong name assembly is generated by a tool provided by the .NET Framework SDK, called SN (`sn.exe`). SN creates a key file, which is referenced by the assembly in an attribute. The signing of an assembly can only take place when it is created, or *built*. However, in the form of signing called *delayed signing*, developers work only with the public part of the key. Later, when the final build of the assembly is prepared, the private key is added to the signature. Currently the Delphi for .NET compiler is weak on custom attributes. Because attributes are required to express the identity and reference the key file, support for strong name assemblies isn't quite complete at the time of writing.

Strong name assemblies are usually intended to be shared, so you deploy them in the Global Assembly Cache (GAC) (but they can be deployed privately as well). Because the GAC is a system folder with a specialized structure, you can't just copy the assembly there. Instead, you must use another tool contained in the .NET Framework runtime, called GACUTIL (`gacutil.exe`). GACUTIL installs files into and removes files from the GAC.

The GAC is a special system folder with a hierarchy that supports side-by-side deployment of assemblies, as you can see in [Figure 24.4](#). The intent is to hide the structure of the GAC so you don't have to worry about the complexities involved. GACUTIL reads the assembly's identity from its metadata and uses that information to construct a folder within the GAC to hold the assembly.



**Figure 24.4:** .NET's Global Assembly Cache as seen in Windows Explorer

**Note** You can see the true folder hierarchy by opening a command window and navigating to the `C:\Windows\assembly` directory. This hierarchy is hidden from you if you use Windows Explorer to view it.

When you create a reference to a strong name assembly in the GAC, you are creating a reference to a specific version of that assembly. If a later version of that same assembly is deployed on the machine, it will be put into a separate location in the GAC, leaving the original intact. Because your assembly referenced a specific version, that's the version it always gets.

## What's Next?

Your reaction to the .NET initiative will depend on your programming background. If you come from the Java environment, from a traditional Windows tool (even if object-oriented), or from Delphi, you may find many new features or many concepts you are already familiar with. The .NET platform is a technical achievement that offers a staggering array of options and possibilities. Compared with the earlier Microsoft system interfacing technologies (the Windows API and COM), the progress is significant. The best part is that you can take advantage of the platform using the tools and languages you feel are best for the job, soon including Delphi.

In [Chapter 25](#) we will examine specific changes to the Delphi language. We will look at deprecated features and types, new features that have already been added to the language, and examples demonstrating the use of .NET Framework classes from Delphi for .NET.

## Chapter 25: Delphi for .NET Preview: The Language and the RTL

### Overview

The [last chapter](#) introduced Microsoft's .NET architecture. Now it is time to focus on the Delphi for .NET Preview that ships with Delphi 7. This chapter will cover specific changes that were made to the Delphi language to make it compatible with the Common Language Runtime. There have been some important additions to the language, such as namespaces and new visibility specifiers. Other long-standing features of the language had to be dropped because they are not supported in a type-safe environment.

This chapter discusses areas of the new compiler including the use of some of Microsoft's own class libraries for .NET and ASP support.

Bear in mind that the Delphi for .NET compiler is still being crafted as I write this. Borland's plan was to release the Preview compiler with Delphi 7 and to provide periodic updates to registered Delphi customers throughout the remainder of 2002 and in the early 2003 (before the Galileo project is completed). Some of the language features discussed here may not be implemented, or may be in varying states of completeness, depending on which version of the compiler you are working with. At the end of this chapter I list web resources you can monitor to keep abreast of the changes and latest news regarding Borland's Delphi for .NET compiler.

**Note** As was true in [Chapter 24](#), most of this material comes from John Bushakra.

## Deprecated Delphi Language Features

We will begin by looking at some of the Delphi features that had to be dropped (deprecated) in order to be compatible with the Common Language Runtime (CLR). Then we'll look at new features added (or planned) in Delphi for .NET, which will shape the Delphi language in the near future, possibly also on the Win32 and Linux platforms.

### Deprecated Types

Some of Delphi's types will not survive the transition to a managed, virtual execution system. So far, the following types are either known to be deprecated or have uncertain fates:

**Pointers** Pointers are considered unsafe types by the CLR, and all forms of pointer arithmetic are forbidden. *Unsafe* means the code cannot be verified for type safety. The final version of the Delphi for .NET compiler may support unsafe, unmanaged pointers, but in the meantime you can use dynamic arrays to get back some of the functionality of the (also deprecated) `GetMem`, `FreeMem`, and `ReallocMem` functions.

**Types Based on *file of <type>*** File types based on the old Pascal *file of <type>* syntax cannot be supported because the compiler has no way to determine the size of a given type on the target platform.

**Pre-Delphi *Object Syntax*** The pre-Delphi *object* syntax has been deprecated and will not be supported in the final release of the compiler. This syntax was introduced back in the Turbo Pascal era and allowed you to declare a new class with the syntax `type MyClass = object;`. Variables of this type were stack-based, contrary to the heap-based class type objects.

**Real48 and Comp** These types won't be supported. Real48 type is a 6-byte floating-point type. The Comp type will be replaced by Int64 in the future, as noted in the Delphi 7 Language Reference.

### Strings and Other Types

The following types, although not candidates for deprecation, will have changes made to their underlying implementation. These changes are almost transparent, but you can expect slightly different behavior in some circumstances:

**Strings** In Delphi for .NET, strings map to the CLR type `System.String` and are wide by default. This means they use 16 bits per character, like the `WideString` type in Delphi 7. In addition, all characters are wide by default.

**Records** Records are mapped to value types. [Chapter 24](#), "The Microsoft .NET Architecture from the Delphi Perspective," talked about the two main categories of types specified by the Common Type System (CTS): reference types and value types. A record will be a value type on the .NET platform. The CLR demands that value types cannot have inheritance, but you can define methods on them (which, of course, is completely new to the Delphi language). Methods defined on value types must be declared as `final`. (The new `final` keyword is discussed in "[New Delphi Language Features](#)" section.)

**TDateTime** In Delphi, this type is based on the same implementation as Microsoft's DATE type (see [Chapter 2](#), "The Delphi Language," for more details). The .NET platform uses a different implementation. The `System.DateTime` structure (a descendant of `System.ValueType`) measures time from midnight, January 1, 0001 C.E. (Common Era) to 11:59:59 p.m., December 31, 9999 C.E. On this clock, one tick equals 100 nanoseconds. Going forward, Delphi for .NET will transition to the .NET platform standard for measuring time. Date calculations that depend on the floating-point implementation will require your attention when porting to the .NET platform. In particular, if you are

using Delphi's `Trunc` and `Frac` functions to separate the date and time portions of the floating-point value, then you could be in for some interesting bugs in the future.

**Currency** Currency will be mapped to the CLR type `System.Decimal`.

## Deprecated Code Features

As is the case with the types listed in the [previous section](#), some Delphi features that have been part of the language a long time cannot be ported to the .NET platform:

**Variant Records** Variant records with overlapping fields are not supported by the CLR. In general, you can't make any assumptions regarding the layout of fields in a record declaration, because the Just In-Time (JIT) compiler reserves the right to optimize things to suit the underlying platform.

*ExitProc* Things don't always happen when you'd like them to, or in the order you'd like them to. Such problems with unit initialization and finalization have been overcome (although there are still issues to be aware of, as I discuss later in this chapter), but `ExitProc` is not supported.

**Dynamic Aggregation of Interfaces** The CLR doesn't support dynamic aggregation of interfaces using the `implements` keyword, because it cannot be verified for type safety. A class must declare all interfaces it will implement.

**ASM Statements** ASM statements and inline assembly language are not supported by the Delphi for .NET Preview compiler. The future of `asm` in the final version is doubtful. The compiler would have to be able to mix managed IL (Intermediate Language) and unmanaged native CPU instructions, like Microsoft's Visual C++ compiler.

*automated* **Keyword** The `automated` keyword was created to support OLE Automation. It is not needed in the .NET environment. The same is true for the `dispid` keyword, which is used to dispatch COM Automation methods by number instead of by name. Notice that although they're no longer explicitly required, GUIDs are supported on the .NET platform; they appear as custom attributes on a type.

**Direct Memory Access Functions** Direct memory access functions like `BlockRead`, `BlockWrite`, `GetMem`, `FreeMem`, and `ReallocMem`, as well as `Absolute`, and `Addr`, all deal with unmanaged pointers, and so cannot be used with managed, safe code. The `@` operator is available in the current Preview version of the compiler (but is not expected to remain in the final version), although you cannot type cast pointers or do any pointer arithmetic.

**Note** As discussed in [Chapter 2](#), Delphi 7 provides a new set of compiler warnings to help you get ready to port your code. These warnings flag certain features and language constructs that are known to be unsafe on the .NET platform and therefore should be avoided. The warnings are turned off by default for a new Delphi 7 project but are active when you recompile an existing project. You can also turn them on with the `{ $WARN UNSAFE_CODE ON }` compiler directive and similar directives. Refer to [Chapter 2](#) for more details.

## New Delphi Language Features

The first release of the dccil compiler included new features required by the CLR, and more have been added in subsequent updates.

### Unit Namespaces

Namespaces play an important role in the .NET Framework. They allow the class hierarchy to be extended by multiple third parties without fear of conflicting symbol names. Windows and COM use a 16-byte GUID to uniquely identify components, and this magic number must be recorded in the system registry. On the .NET platform, the concept of namespaces—plus metadata and the hard-and-fast rules about locating assemblies—makes GUIDs obsolete.

Ironically, the idea of a Delphi unit is similar to the CLR's namespaces. It's not too far a leap, if you think of a unit as a container of symbols, and a namespace as a container of units. In Delphi for .NET, the namespace to which a unit belongs is declared in the `unit` clause:

```
unit NamespaceA.NamespaceB.UnitA;
```

The dots indicate the containment of one namespace within another, and ultimately of the unit within the namespace. The dots separate the declaration into components, and each component—up to but not including the rightmost one—is a namespace. The entire declaration taken as a whole, dots and all, is the unit name. The dots simply serve as separators; no new symbols are introduced by the declaration. In this example, `NamespaceA.NamespaceB` is the namespace, and `NamespaceA.NamespaceB.UnitA` is the name of the unit. `NamespaceA.NamespaceB.UnitA.pas` would be the name of the source file, and the compiler would produce an output file called `NamespaceA.NamespaceB.UnitA.dcuil`.

The `program` statement (and eventually the `package` and `library` statements) optionally declares the default namespace for the entire project. Otherwise, the project is called a *generic project*, and the default namespace is that specified by the `-ns` compiler option. If no default project namespace is specified with compiler options, then behavior reverts to not using namespaces, like in Delphi 7 (and prior releases).

The `unit` clause does not have to declare membership in any explicit namespace. It might look like a traditional Delphi statement:

```
unit UnitA;
```

A unit that does not declare membership in a namespace is called a *generic unit*. Generic units automatically become members of the project namespace. Note, however, that this does not affect the source filename.

**Warning** At the time of this writing, namespace support is very limited; this section describes how it should work in the future, rather than how it works now.

In the project file, you can specify a `namespaces` clause to list a set of namespaces for the compiler to search when it is trying to resolve references to generic units. The `namespaces` clause must appear immediately after the `program` (or `package` or `library`) statement and before any other clause or block type. The namespaces are separated by commas, and the list is terminated with a semicolon. For example:

```
program NamespaceA.MyProgram
  namespaces Foo.Bar, Foo.Frob, Foo.Nitz;
```

This example adds the namespaces `Foo.Bar`, `Foo.Frob`, and `Foo.Nitz` to the generic unit search space.

This discussion leads up to showing you how the compiler searches for generic units when you build your program. When you use a unit and fully qualify its name with the full namespace declaration, there is no problem:

```
uses Foo.Frob.Gizmos;
```



The compiler knows the name of the `dcuil` file (or the `.pas` file) in this case. But suppose you only said the following:

```
uses Gizmos;
```

This is called a *generic unit reference*, and the compiler must have a way to find its `dcuil` file.

The compiler searches namespaces in the following order:

1. The current unit namespace (if any)
2. The default project namespace (if any)
3. The namespaces listed in the project's `namespaces` clause (if any)
4. The namespaces specified by compiler options

For the first item, if the current unit specifies a namespace, then subsequent generic unit references in the current unit's `uses` clause are looked for first in the current unit's namespace. Consider this example:

```
unit Foo.Frob.Gizmos;  
uses doodads;
```

The first search location for the unit `doodads` would be in the namespace `Foo.Frob`. So, the compiler would try to open `Foo.Frob.Doodads.dcuil`. Failing this, the compiler would move on and prefix the unit name `doodads` with the default project namespace, and so on down the list.

The same symbol name can appear in different namespaces. When such ambiguity occurs, you must refer to the symbol by its full namespace and unit name. If you have a symbol named `Hoozitz` in unit `Foo.Frob.Gizmos`, you can refer to the symbol with either

```
Hoozitz; // if the name is unambiguous  
Foo.Frob.Gizmos.Hoozitz;
```

but not with

```
Gizmos.Hoozitz; // error!  
Frob.Gizmos.Hoozitz; // error!
```

Unit and namespace names can become quite long and unwieldy. You can create an alias for the fully qualified name with the `as` keyword in the `uses` clause:

```
uses Foo.Frob.DepartmentOfRedundancyDepartment.UIToys as ToyUnit;
```

Unit aliases introduce new identifiers, so their names cannot conflict with any other identifiers in the same unit (aliases are local to their unit). Even if you declare an alias, you can still use the original, longer name to refer to the unit.

**Note** The case of a namespace declaration is preserved and emitted into assembly metadata as is. However, as far as Delphi is concerned, two namespaces that differ only in case are equivalent.

## Extended Identifiers

The cross-language integration of the CTS and CLR brings up some interesting situations for compiler developers. For example, what if the name of an identifier in an assembly is the same as one of your language keywords? Consider the Delphi language keyword `type`. `Type` is also the name of a CLR class. Because `type` is a language keyword, it cannot be used as the name of an identifier. You can avoid this problem two ways in Delphi for .NET (these techniques were not implemented in Delphi 7 and previous versions).

First, you can use the fully qualified name of the identifier:

```
var  
  T: System.Type;
```



The second, shorter way is to use the new ampersand operator (&) to prefix the identifier. The following has the same effect as the previous example:

```
var
  T: &Type;
```

In this statement the ampersand tells the compiler to look for a symbol with the name `Type` and to not consider it as a keyword. The compiler will look for the `Type` symbol in the available units, finding it in `System` (the same mechanism works regardless of the unit defining the symbol).

## The *final* and *sealed* Keywords

Two more concepts specified by the Common Language Infrastructure (CLI) have been added to the Delphi for .NET compiler: the class attribute `sealed` and the method attribute `final`. Putting the `sealed` attribute on a class effectively ends the class's ability to be used as a base class. Here is a sample code snippet:

```
type
  TDeriv1 = class (TBase)
    procedure A; override;
  end sealed;
```

A class cannot derive from a class that has been sealed. Similarly, a virtual method marked with the `final` attribute cannot be overridden in any descendant class, as in the following sample code.

```
type
  TDeriv1 = class (TBase)
    procedure A; override; final;
  end;

  TDeriv2 = class (TDeriv1)
    procedure A; override; // error: "cannot override a final method"
  end;
```

Borland added the `sealed` and `final` keywords to map an existing feature of .NET, but why did Microsoft introduce these attributes? The `final` and `sealed` attributes give users of your code important insights into how you intend your classes to be used. Moreover, these attributes give the compiler hints that allow it to generate more efficient Common Intermediate Language (CIL).

## New Visibility and Access Specifiers

Delphi's notion of visibility—public, protected, and private—is a bit different from that of the CLI. In languages like C++ and Java, when you specify a visibility of private or protected on a class member, that class member is only visible to descendants of the class in which it is defined. As you saw in [Chapter 2](#), however, Delphi enforces the idea of private and protected only for classes in different units, because everything is visible within a single unit. To be CTS compliant, the language required new visibility specifiers:

*class private* A member declared with class private visibility follows the C++ and Java rules. That is, class private members can be accessed only in methods or properties of the declaring class. Procedures and functions declared at the unit level and methods of other classes do not have access.

*class protected* Similarly, class protected members are visible only within the declaring class, and to descendants of the declaring class. Other classes in the same unit have access only if they inherit from this class.

See the `ProtectedPrivate` example in the `LanguageTest` folder of the chapter's source code for a trivial test case.

## Class Static Members

Delphi has long supported class methods—methods you can apply to a class as a whole and also as a specific instance, even if the methods' code cannot refer to the current object (the `Self` parameter of a class methods references the current class, not the current object). Delphi for .NET extends this idea by adding the `class static` specifier, class properties, class static fields, and class constructors:

**Class Static Methods** Like Delphi 7 class methods, class static members can be called without an object instance, and no `Self` parameter refers to an object. Unlike in Delphi 7, however, you cannot refer to the class itself. For example, calling the `ClassName` method will fail. Also unlike in Delphi 7, you cannot use the `virtual` keyword with class static methods.

**Class Static Properties** Like class methods, class static properties can be accessed without an object instance. The access methods or backing fields for class static properties must be declared class static themselves. Class static properties cannot be published, nor can they have stored or default value definitions.

**Class Static Fields** A class static field can be accessed without an object instance. Class static fields and properties are typically used as design tools; they allow you to declare variables and constants within the meaningful context of a class declaration.

**Class Constructor** A class constructor is a private constructor (it must be declared with class private visibility) that runs prior to the first use of the declaring class. The CLR offers no guarantee of when this will happen, except to say it will happen before the first use of the class. In CLR terms, this can get a bit tricky, because code is not considered "used" unless (and until) it is executed. A class can declare only one class constructor. Descendants can declare their own class constructors, but only one can be declared in any class.

You can't call a class constructor from source code; it is called automatically as a way to initialize class static fields and properties. Even the `inherited` keyword is prohibited, because the compiler takes care of this for you.

The following example class declaration illustrates the syntax for these new specifiers:

```
TMyClass = class
class private // can only be accessed within TMyClass
    // Class constructor must have class private visibility
    class constructor Create;
class protected // can be accessed in TMyClass and in descendants
    // Class static accessors for class static property P1, below
    class static function getP1 : Integer;
    class static procedure setP1(val : Integer);
public
    // fx can be called without an object instance
    class static function fx(p : Integer) : Integer;
    // Class static property P1 must have class static accessors
    class static property P1 : Integer read getP1 write setP1;
end;
```

## Nested Types

Nested types are similar to class fields, in that they can be accessed through a class reference; an object instance is not needed. Declared within the scope of a class, nested types give you a way to use the enclosing class as a kind of namespace for the type.

## Multicast Events

Delphi has always had the ability to set an event listener—a function that is called when an event is fired. The CLR supports the use of multiple event listeners so that more than one function can respond when an event is fired. These are called *multicast events*. Delphi for .NET introduces two new property access methods, `add` and `remove`, to support multicast events. The `add` and `remove` methods can be used only on properties that are events.

To support multicast events, you must have a way to store all the functions that register themselves as listeners. As stated in [Chapter 24](#), multicast events are implemented using the CLR `MulticastDelegate` class. And, as discussed there, the compiler hides a lot of complexity behind the scenes. The `add` and `remove` keywords handle the storage and removal of event listeners, but the containment mechanism is an implementation detail you aren't expected to deal with. The compiler automatically generates `add` and `remove` methods for you, and these methods implement storage of event listeners in an efficient way.

In the final release of Delphi for .NET, the `add` and `remove` methods should work hand in hand with an overloaded version of the standard functions `Include` and `Exclude`. In your source code, when you'd want to register a method as an event listener, you call `Include`. To remove a method, call `Exclude`. For example:

```
Include(EventProp, eventHandler);  
Exclude(EventProp, eventHandler);
```

Behind the scenes, `Include` and `Exclude` will call the methods assigned to the `add` and `remove` access functions, respectively. At the time of this writing, this technology wasn't working, so the book examples don't use it.

To support legacy code, the Delphi assignment operator (`:=`) still works as a way to assign a single event handler. The compiler generates code to go back and replace the last event handler (and only that event handler) that was set with the assignment operator. The assignment operator works separately and independently from the `add/remove` (or `Include/Exclude`) mechanism. In other words, the use of the assignment operator does not affect the list of event handlers that have been added to the `MulticastDelegate`.

As an example, you can refer to the `XmlDemo` program. The following code snippet (the working code at the time of this writing) creates a button at run time and installs two event handlers for its `Click` event:

```
MyButton := Button.Create;  
MyButton.Location := Point.Create (  
    Width div 2 - MyButton.Width div 2, 2);  
MyButton.Text := 'Load';  
MyButton.add_Click (OnButtonClick);  
MyButton.add_click (OnButtonClick2);  
Controls.Add (MyButton);
```

## Custom Attributes

Recall from [Chapter 24](#) that one of the requirements of the CLI is an extensible metadata system. All .NET language compilers are required to emit metadata for the types defined within an assembly. The *extensible* part of extensible metadata means that programmers can define their own attributes and apply them to just about anything: assemblies, classes, methods, and more. The compiler emits these into the assembly's metadata. At run time, you can query for the attributes that were applied to an entity (assembly, class, method, and so on) using the methods of the CLR class `System.Type`.

Custom attributes are reference types derived from the CLR class `System.Attribute`. Declaring a custom attribute class is just like declaring any other class (this code snippet is extracted from the trivial `NetAttributes` project part of the `LanguageTest` folder):

```
type  
    TMyCustomAttribute = class(TCustomAttribute)  
    private
```

```

    FAttr : Integer;
public
    constructor Create(val: Integer);
    property customAttribute : Integer read FAttr write FAttr;
end;
...
constructor TMyCustomAttribute.Create(val: Integer)
begin
    inherited Create;
    customAttribute := val;
end;

```

The syntax for applying the custom attribute is similar to that of C#:

```

type
    [TMyCustomAttribute(17)]
    TFoo = class
public
    function Pl(X : Integer) : Integer;
end;

```

The custom attribute is applied to the construct immediately following it. In the example, it is applied to the class `TFoo`. No doubt you noticed that the custom attribute syntax is nearly identical to that of Delphi's GUID syntax. Here we have a problem: GUIDs are applied to interfaces; they must immediately *follow* the interface declaration. Custom attributes, on the other hand, must immediately *precede* the declaration to which they apply. How can the compiler determine whether the thing in the square brackets is a traditional Delphi-style GUID (which should be applied to the preceding interface declaration) or a .NET-style custom attribute (which should be applied to the first member of the interface)?

There is no way to tell, so you have to punt—make a special case for custom attributes and interfaces. If you apply a GUID to an interface, it must immediately follow the declaration of the interface, and it must follow the established Delphi syntax:

```

type
    interface IMyInterface
    ['(12345678-1234-1234-1234-1234567890ab)']

```

CLR's `GuidAttribute` custom attribute is used to apply GUIDs; it is part of the `System.Runtime.InteropServices` namespace. If you use this custom attribute to apply a GUID, then you must follow the CLR standard and put the attribute declaration before the interface.

## Class Helpers

Class helpers are an intriguing new language feature added to Delphi for .NET. The main reason for supporting class helpers is the way Borland maps .NET core classes with its own RTL classes, as covered later in the section "[Class Helpers for the RTL](#)." Here I will focus on this feature from a language perspective.

A class helper gives you a way to extend a class without using derivation, by adding new methods (but not new data). The odd fact, compared to inheritance, is that you can create objects of the original class, which is extended maintaining the same name. This means you can *plug-in* methods to an existing object of an existing class. A simple example will help clarify the idea.

Suppose you have a class (probably one you haven't written yourself—otherwise you could have extended it right away) like this:

```

type
    TMyObject = class
private
    Value: Integer;

```

```
Text: string;  
public  
  procedure Increase;  
end;
```

Now you can add a `Show` method to objects of this class by writing a class helper to extend it:

```
type  
  TMyObjectHelper = class helper for TMyObject  
  public  
    procedure Show;  
  end;  
  
procedure TMyObjectHelper.Show;  
begin  
  WriteLn (Text + ' ' + IntToStr (Value) + ' -- ' +  
    Self.ClassType.ClassName + ' -- ' + ToString);  
end;
```

Notice that `Self` in the class helper method is the object of the class that the helper is for. You can use it like this:

```
Obj := TMyObject.Create;  
...  
Obj.Show;
```

You'll end up seeing the name of the `TMyObject` class in the output. If you inherit from the class, however, the class helper will also be usable on the derived class (so you end up adding a method to an entire hierarchy), and everything will work properly. For your experiments, refer to the `ClassHelperDemo` example in the `LanguageTest` folder.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## The Run-time Library and VCL

Located in the `source\rtl` directory of the Delphi for .NET Preview installation, you will find the source files for the run-time library (RTL). You can already see the migration of units into CLR namespaces, as reflected in the source filenames.

Borland is taking the approach (for the most part) of preserving the original unit name and prefixing it with the namespace name `Borland.Delphi`. Operating system–specific things (such as registry and `ini` file utilities) go in the `Borland.Win32` namespace, because these classes, procedures, and functions are Borland-specific wrappers of Windows-specific features. The naming trend should continue, although not all units will make the transition, and some will have their contents reorganized into an appropriate namespace.

Perusing the RTL source files is both highly educational and highly recommended; however, remember that you are looking at a preview release of a product, not the final version. The contents of the RTL source files are still subject to change—you should not make any assumptions, and you should definitely not introduce dependencies into your own code based on what you see there.

### Class Helpers for the RTL

With the warnings out of the way, let's see what has been done to the RTL so far. The most interesting change is perhaps the introduction of class helpers.

`Borland.Delphi.System.pas` includes the following declaration:

```
type
  TObject = System.Object;
```

It tells you that Delphi's `TObject` class is an alias for the CLR class `System.Object`. This is important: `TObject` is not a descendant of `System.Object`—it is semantically equivalent. What happened to the methods that used to be defined in `TObject`, such as `ClassName` and `ClassParent`? That's where the class helper comes in.

The methods that used to be directly declared and implemented in `TObject` are now declared and implemented in a class called `TObjectHelper`. `TObjectHelper` is then declared to be a class helper for `TObject`. In `Borland.Delphi.System.pas` is the following:

```
type
  TObjectHelper = class helper for TObject
    procedure Free;
    function ClassType: TClass;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: TObject;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: string): TObject;
    class function SystemType: System.Type;
    function FieldAddress(const Name: string): TObject;
    procedure Dispatch(var Message);
  end;
```

A class helper gives you a way to extend a class without using derivation.

You might want to extend a CLR class but not derive from it in order to use the CLR class with existing Delphi code. No doubt you've noticed that Delphi's class framework and the .NET Framework share a fair amount of functionality. In some cases there are name clashes between the two—for example, Borland's `Exception` class and the CLR's `System.Exception` class. On one hand, the two classes do basically the same thing, but they expose that

functionality in different ways. On the other hand, large amounts of existing Delphi code have been using Borland's `Exception` class for a long time.

The only workable solution was to create a mechanism that would allow developers (including Borland) to leverage the CLR classes, and that would also allow the CLR classes to be extended to include long-standing behaviors expected by existing Delphi code.

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## The VCL

The .NET Framework classes in the `System.Windows.Forms` namespace are not a replacement for the GUI portion of the Win32 API. The same happens with most other portions of the .NET Framework: its role is to make the underlying API easier to use by providing an easier object-oriented interface compatible with the core services of the .NET environment. The GUI subset of Win32 is still there, occupying the same place it always has. `System.Windows.Forms` organizes this GUI subset of the Win32 API, presents it in an object-oriented way, and layers an event model on top of it; but classes in `System.Windows.Forms` call the unmanaged code in Win32. When you use `System.Windows.Forms`, you are still calling the Win32 APIs, but now you have a large layer of software called the CLR sitting between your code and Win32.

This entire preamble is important to acknowledge why the VCL takes this same approach. `TObject` is rooted (if you will) from `System.Object` through the use of a class helper. `TPersistent` and `TComponent` still descend from there. Thus the VCL class `TForm`, for example, is not a descendant of the class `System.Windows.Forms.Form`. Instead, the entire VCL hierarchy remains much as it is today. `TForm` will ultimately descend from `TWinControl`, which itself is a descendant of `TControl` and then `TComponent`.

If you consider the entire `System.Windows.Forms` namespace as a single entity, the VCL then becomes a kind of sibling to it, rather than a child of it. Both frameworks ultimately rely on the native, unmanaged Win32 API for the underlying implementation of the user interface controls.

## Looking into the VCL.NET Source

The update of the Delphi .NET Preview compiler made available by Borland in November 2002, although still preliminary, provides details about the architecture the company is planning. If you open the `Borland.Vcl.Controls` unit, you'll be surprised by its similarity to the Win32 version. The source code is almost identical; the differences exist behind the scenes at the `TObject` and `TComponent` level. I've already covered the former, so let's focus on the core component class, which is defined in three steps:

### type

```
TComponent = System.ComponentModel.Component;  
TComponentHelper = class helper (TPersistentHelper) for TComponent  
TComponentSite = class(TObject, ISite, IServiceProvider)
```

The `TComponent` class corresponds to the .NET Framework class, with a helper providing extra methods and properties and a further class offering the extra data required by the helper class. The situation is complex, and I don't want to get into the details because the `TComponent` class is marked as experimental and may change in further updates.

Getting back to the VCL, a large set of components is already available, so you can begin porting code. The only trouble you'll face with the November 2002 update is that streaming is not supported; so, you must add the component-creation code in the form constructor (an action that will not be required by the final version of Delphi for .NET).

As an interesting example to help you figure out the architecture of the VCL classes, I've ported to .NET the `ClassInfo` example from [Chapter 3](#), "The Run-Time Library." The `NetClassInfo` example uses this modified code in the project source (again, something you won't have to do in the future):

```
Application.Initialize;  
Form1 := TForm1.Create (Application);  
Application.MainForm := Form1;
```

The code for the form, as I've mentioned, has an extra method called by the constructor and used to initialize the controls. This method is quite long, so I'll provide only a few excerpts here:

```
procedure TForm1.InitializeControls;  
begin
```



```

// creating all controls...
Label3:= TLabel.Create(Self);
Panell:= TPanel.Create(Self);
Label1:= TLabel.Create(Self);
Label2:= TLabel.Create(Self);
...

// setting form properties and events
Left:= 217;
Top:= 109;
Caption:= 'Class Info';
OnCreate:= FormCreate;

// initializing controls (only one is listed here)
with Label3 do
begin
    Parent:= Self;
    Left:= 8;
    Top:= 8;
    Width:= 56;
    Height:= 13;
    Caption:= 'Class Name';
end;

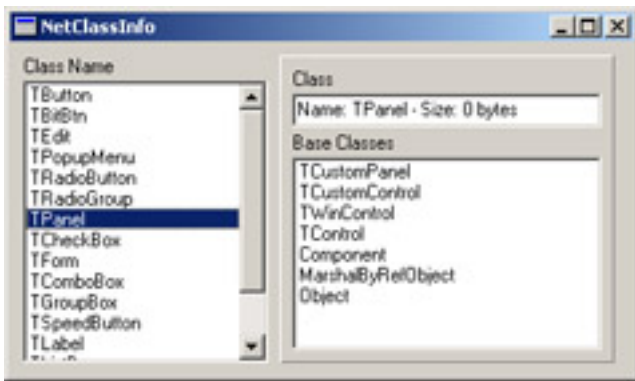
```

The rest of the application's code remains almost identical, which is surprising considering that this is a low-level example. I had to remove the call to `InstanceSize`, because the compiler cannot resolve the size of an object given the architecture of .NET, and I had to test for the base class against `Object` instead of `TObject`. Here is the code snippet that produces the output shown in [Figure 25.1](#):

```

procedure TForm1.ListClassesClick(Sender: TObject);
var
    MyClass: TClass;
begin
    MyClass := ClassArray [ListClasses.ItemIndex];
    EditInfo.Text := Format ('Name: %s - Size: %d bytes',
        [MyClass.ClassName, 0 {MyClass.InstanceSize}]);
    with ListParent.Items do
    begin
        Clear;
        while MyClass.ClassName <> 'Object' do
        begin
            MyClass := MyClass.ClassParent;
            Add (MyClass.ClassName);
        end;
    end;
end;

```



**Figure 25.1:** The NetClassInfo example shows the base classes of a given component.

## Further VCL Examples

To provide starting points for your own experiments with the VCL under .NET, I've built two more examples. NetEuroConv is a port of the EuroConv example from [Chapter 3](#) based on the RTL's conversion engine. NetLibSpeed is a port of the LibSpeed example used in [Chapter 5](#) ("Visual Controls") to compare the VCL and VisualCLX libraries' speed in creating visual components. The number you'll see makes little sense in such a preliminary version of a library, although the fact that VCL.NET takes four to five times as long for the same purpose may worry you.

As I mentioned, these examples are meant to be only starting points for your experiments. They may not work with further updates of the Delphi for .NET Preview.

**Note** Stay tuned to my website for updates of this section of the book and the related examples.

Team LiB

◀ PREVIOUS

NEXT ▶

## Using Microsoft Libraries

The VCL is not quite ready, but you can use the .NET Framework class library as a basis for experimentation with the Delphi for .NET Preview compiler. It can be educational to build programs with the compiler and then inspect them with Intermediate Language Disassembler (ILDASM), for instance. This will be the aim of this section. If you want to look at a simpler example using XML support, refer to the `XmlDemo` mentioned earlier in the chapter.

The `CLRReflection` program opens an assembly and then uses reflection to inspect the modules and types defined within that assembly. This program demonstrates using a common dialog box (the `OpenFileDialog`), constructing menus, handling events, using Delphi's dynamic arrays, and, of course, reflection. Let's look at the project file first:

```
program CLRReflection;

uses
  System.Windows.Forms,
  ReflectionUnit;

var
  reflectForm : ReflectionForm;
begin
  reflectForm := ReflectionForm.Create;
  System.Windows.Forms.Application.Run(reflectForm);
end.
```

The code looks almost like a good old VCL application. You define a variable for your main form, and then you create the form. Then you use the `Run` method of the .NET Framework class `System.Windows.Forms.Application`. Here the code is analogous (at least in concept) to the way it is done in the VCL.

Note that throughout this example I have given the fully qualified name for .NET Framework classes. I did so to make sure you know where these classes are located. Because the `uses` clause includes `System.Windows.Forms`, you could shorten the expression

```
System.Windows.Forms.Application.Run(reflectForm);
```

to

```
Application.Run(reflectForm);
```

Now, look at [Listing 25.1](#), which shows the unit where the main form is defined. Note that this code compiles with the November 2002 update of the Delphi for .NET Preview, but not with the version originally shipping with Delphi 7.

### Listing 25.1: The `ReflectionUnit` Unit of the `CLRReflection` Example

---

```
unit ReflectionUnit;

interface

uses
  System.Windows.Forms,
  System.Reflection,
  System.Drawing,
  Borland.Delphi.SysUtils;

type
  ReflectionForm = class(System.Windows.Forms.Form)
  private
    mainMenu: System.Windows.Forms.MainMenu;
    fileMenu: System.Windows.Forms.MenuItem;
    separatorItem: System.Windows.Forms.MenuItem;
```

```

openItem: System.Windows.Forms.MenuItem;
exitItem: System.Windows.Forms.MenuItem;

showFileLabel: System.Windows.Forms.Label;
typesListBox: System.Windows.Forms.ListBox;
openFileDialog: System.Windows.Forms.OpenFileDialog;
protected
  procedure InitializeMenu;
  procedure InitializeControls;
  procedure PopulateTypes(fileName: String);
  { Event Handlers }
  procedure exitItemClick(sender: TObject; Args: System.EventArgs);
  procedure openItemClick(sender: TObject; Args: System.EventArgs);
public
  constructor Create;
end;

implementation

constructor ReflectionForm.Create;
begin
  inherited Create;

  SuspendLayout;
  InitializeMenu;
  InitializeControls;

  { Initialize the form and other member variables }
  openFileDialog := System.Windows.Forms.OpenFileDialog.Create;
  openFileDialog.Filter := 'Assemblies (*.dll;*.exe)|*.dll;*.exe';
  openFileDialog.Title := 'Open an assembly';

  AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
  ClientSize := System.Drawing.Size.Create(631, 357);
  Menu := mainMenu;
  Name := 'reflectionForm';
  Text := 'Reflection in Delphi for .NET';

  { Add the controls to the form's collection. }
  Controls.Add(showFileLabel);
  Controls.Add(typesListBox);
  ResumeLayout;
end;

{ Build the main menu }
procedure ReflectionForm.InitializeMenu;
var
  menuItemArray : array of System.Windows.Forms.MenuItem;
begin
  mainMenu := System.Windows.Forms.MainMenu.Create;
  fileMenu := System.Windows.Forms.MenuItem.Create;
  openItem := System.Windows.Forms.MenuItem.Create;
  separatorItem := System.Windows.Forms.MenuItem.Create;
  exitItem := System.Windows.Forms.MenuItem.Create;

  { Initialize mainMenu }
  mainMenu.MenuItems.Add(fileMenu);

```

```

{ Initialize fileMenu }
fileMenu.Index := 0;
SetLength(menuItemArray, 3);
menuItemArray[0] := openItem;
menuItemArray[1] := separatorItem;
menuItemArray[2] := exitItem;
fileMenu.MenuItems.AddRange(menuItemArray);
fileMenu.Text := '&File';

// openItem
openItem.Index := 0;
openItem.Text := '&Open...';
openItem.add_Click(openItemClick);

// separatorItem
separatorItem.Index := 1;
separatorItem.Text := '-';

// exitItem
exitItem.Index := 2;
exitItem.Text := 'E&xit';
exitItem.add_Click(exitItemClick);
end;

{ Create the controls and populate the form }
procedure ReflectionForm.InitializeControls;
begin
  { Initialize showFileLabel }
  showFileLabel := System.Windows.Forms.Label.Create;
  showFileLabel.Location := System.Drawing.Point.Create(5, 6);
  showFileLabel.Name := 'showFileLabel';
  showFileLabel.Size := System.Drawing.Size.Create(616, 37);
  showFileLabel.TabIndex := 0;
  showFileLabel.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right;
  showFileLabel.Text := 'Showing types in: ';

  { Initialize typesListBox }
  typesListBox := System.Windows.Forms.ListBox.Create;
  typesListBox.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Bottom or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right;
  typesListBox.Location := System.Drawing.Point.Create(8, 46);
  typesListBox.Name := 'typesListBox';
  typesListBox.Size := System.Drawing.Size.Create(610, 303);
  typesListBox.Font := System.Drawing.Font.Create('Lucida Console', 8.25,
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, 0);
  typesListBox.TabIndex := 1;
end;

{ Event handler for the Exit menu item }
procedure ReflectionForm.exitItemClick(sender: TObject; Args: System.EventArgs);
begin
  System.Windows.Forms.Application.Exit;

```

```

end;

{ Event handler for the Open menu item }
procedure ReflectionForm.openItemClick(sender: TObject; Args: System.EventArgs);
begin
    if openFileDialog.ShowDialog = DialogResult.OK then
        begin
            showFileLabel.Text := 'Showing types in: ' + openFileDialog.FileName;
            PopulateTypes(openFileDialog.FileName);
        end;
    end;

{ Open the given assembly, and reflect over its modules }
{ and types. }
procedure ReflectionForm.PopulateTypes(fileName : String);
var
    assy: System.Reflection.Assembly;
    modules: array of System.Reflection.Module;
    module: System.Reflection.Module;
    types: array of System.Type;
    t: System.Type;
    members: array of System.Reflection.MemberInfo;
    m: System.Reflection.MemberInfo;
    i,j,k: Integer;
    s: String;
begin
    try
        { Clear the listbox }
        typesListBox.BeginUpdate;
        typesListBox.Items.Clear;

        { Load the assembly and get its modules }
        assy := System.Reflection.Assembly.LoadFrom(fileName);
        modules := assy.GetModules;

        {For every module, get all types }
        for i := 0 to High(modules) do
            begin
                module := modules[i];
                types := module.GetTypes;

                { For every type, get all of its members }
                for j := 0 to High(types) do
                    begin
                        t := types[j];
                        members := t.GetMembers;

                        { for every member, get type information and add to list box }
                        for k := 0 to High(members) do
                            begin
                                m := members[k];
                                s := module.Name + ':' + t.Name + ':' + m.Name +
                                    ' (' + m.MemberType.ToString + ')';
                                typesListBox.Items.Add(s);
                            end;
                        end;
                    end;
                end;
            end;
        typesListBox.EndUpdate;
    except
    end;

```

```
except
    System.Windows.Forms.MessageBox.Show('Could not load the assembly.');
```

---

```
end;
end;

end.
```

The unit begins by declaring its dependency on .NET Framework `dcuil` files and on the `Borland.Delphi.SysUtils` unit. From there it goes straight into declaring the class for the main form, which is a descendent of the .NET Framework class, `System.Windows.Forms.Form`. The form class layout looks familiar: You have member variables for all the controls, and these are declared to be of types found in the .NET Framework class library.

The functions `exitItemClick` and `openItemClick` are event handler declarations. The signature of event handler methods is specified by the CLR. All event handlers are procedures that take two parameters: the object that fired the event (a derivative of `System.Object`) and the event arguments, which are wrapped in the `System.EventArgs` (or a derived) class. (You will see how to hook up these event handlers in a moment.)

Let's move on to the class constructor. I must call attention to the first statement in the constructor, which calls `inherited Create`.

**Warning** Here you see a major departure from the .NET Framework way of life, compared to what you are used to with the VCL and with Delphi in general. In Delphi, the constructor initializes member variables, putting the object instance into a known-good state; it does not do any memory allocation. So, it is not uncommon to see a constructor make assignments and then call the inherited constructor. Indeed, you might not call the inherited constructor at all. In Delphi for .NET, you can't get away with this approach. In your constructor, you must call `inherited Create`, and it must be the method's first executable statement. Currently, if you fail to do so, you will get a compiler error saying that `Self` is uninitialized and that the inherited constructor must be called prior to accessing any ancestor fields.

After calling the inherited constructor, you are back in familiar territory. Although this code uses a different class hierarchy, it should be clear to any Delphi programmer. You make an instance of `System.Windows.Forms.OpenFileDialog` by calling the `Create` constructor—this is how you create an instance of any .NET Framework class.

The next few lines demonstrate setting properties, both of the `OpenFileDialog` object instance and of the form itself. Finally, you add two controls (a label for the filename and the `ListBox` that will hold the assembly) to the form's `Controls` collection, which is a property of type `Control.ControlCollection`.

The `InitializeMenu` procedure demonstrates allocation and layout of a `System.Windows.Forms.MainMenu` object instance. Where the File menu is initialized, a dynamic array holds each menu item. The dynamic array is then passed to the `AddRange` method. This code could have been accomplished by calling the `Add` method separately for each menu item.

The next interesting thing in `InitializeMenu` is the wiring of the menu item event handlers. In [Chapter 24](#) and earlier in this chapter, I mentioned the behind-the-scenes complexity involved with delegates and multicast events. Here you see some of that complexity coming to the foreground.

You can't do it yet in Delphi for .NET, but in other .NET languages such as C#, you can use the language keyword `event` to introduce an event handler delegate. The event declaration specifies a delegate to use as a callback mechanism. Because the event is a `System.MulticastDelegate` derivative (a `System.EventHandler` delegate in this case), other objects can add and remove event handlers, and these handlers are called when the event fires.

The C# language adds a bit of syntactic sugar to help this pill go down more easily. C# defines `+=` and `-=` operators for adding and removing event handlers, respectively. Eventually Delphi will get its own spoonful of sugar, with the `Include/Exclude` mechanism mentioned previously. CTS mandates that all .NET compilers targeting this event

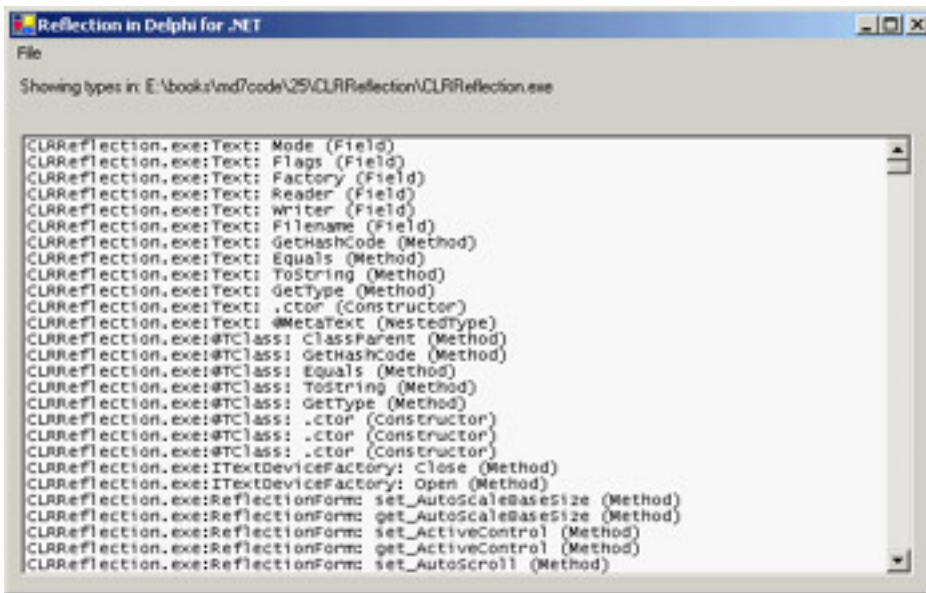
model must generate methods named `add_<Event>` and `remove_<Event>`. These `add_` and `remove_` methods wrap the `Combine` and `Remove` methods declared in `System.Delegate`.

For now, to assign an event handler, you must use these `add_` and `remove_` methods; ordinarily, you would not concern yourself with them, because the compiler would hide this complexity. In the current class declaration, you introduce two methods whose signatures match the `System.EventHandler` delegate: `openItemClick` and `exitItemClick`. You then call the `add_Click` method on the respective menu item, passing your event handler as the callback method.

Now that the setup is out of the way, let's look at the code that reflects over the types defined within an assembly. You can load any assembly (thus creating an object instance), given its filename, with the static `LoadFrom` method. Once you have an assembly object, the keys to the kingdom are yours; you can use reflection to look over the assembly from any angle.

The collection of modules contained within an assembly is available with the `GetModules` method. From there you can drill down to the types defined in the module with `GetTypes`. As you saw in the `InitializeMenu` procedure, you can use dynamic arrays for properties that expose a collection with a `System.Array`.

Finally, each individual member of the module and types arrays contains a `Name` property, which you can use to build a string to display in the `ListBox`. The final effect of the code is visible in [Figure 25.2](#).



```
CLRReflection.exe:Text: Mode (Field)
CLRReflection.exe:Text: Flags (Field)
CLRReflection.exe:Text: Factory (Field)
CLRReflection.exe:Text: Reader (Field)
CLRReflection.exe:Text: Writer (Field)
CLRReflection.exe:Text: Filename (Field)
CLRReflection.exe:Text: GetHashCode (Method)
CLRReflection.exe:Text: Equals (Method)
CLRReflection.exe:Text: ToString (Method)
CLRReflection.exe:Text: GetType (Method)
CLRReflection.exe:Text: .ctor (Constructor)
CLRReflection.exe:Text: #MetaText (NestedType)
CLRReflection.exe:@TClass: ClassParent (Method)
CLRReflection.exe:@TClass: GetHashCode (Method)
CLRReflection.exe:@TClass: Equals (Method)
CLRReflection.exe:@TClass: ToString (Method)
CLRReflection.exe:@TClass: GetType (Method)
CLRReflection.exe:@TClass: .ctor (Constructor)
CLRReflection.exe:@TClass: .ctor (Constructor)
CLRReflection.exe:@TClass: .ctor (Constructor)
CLRReflection.exe:ITextDeviceFactory: Close (Method)
CLRReflection.exe:ITextDeviceFactory: Open (Method)
CLRReflection.exe:ReflectionForm: set_AutoScaleBaseSize (Method)
CLRReflection.exe:ReflectionForm: get_AutoScaleBaseSize (Method)
CLRReflection.exe:ReflectionForm: set_ActiveControl (Method)
CLRReflection.exe:ReflectionForm: get_ActiveControl (Method)
CLRReflection.exe:ReflectionForm: set_AutoScroll (Method)
```

**Figure 25.2:** The `CLRReflection` example, with an assembly loaded



## ASP.NET with the Delphi Language

You might like it or not (I'm not that fond of it), but Microsoft's ASP technology plays a significant role in the development of web applications, as least on the Windows platform. With the transition to ASP.NET, this technology has fully embraced the .NET Framework; now, with the availability of a Delphi compiler for .NET, the Delphi language can be your language of choice in the development of ASP applications.

To create a test, configure IIS to support ASP.NET (I won't cover these steps, which are beyond the scope of this book, but you can find more information on [www.asp.net](http://www.asp.net)) and then place in the target folder the `web.config` file distributed by Borland along with the Delphi for .NET Preview (and available in the `aspx` subfolder). This configuration file defines the mapping of the language to a specific library, again provided by Borland. The core of the file (which uses XML format) has the following elements:

```
<compilation debug="true">
  <assemblies>
    <add assembly="DelphiProvider" />
  </assemblies>
  <compilers>
    <compiler language="Delphi" extension=".pas"
      type="Borland.Delphi.DelphiCodeProvider,DelphiProvider" />
  </compilers>
</compilation>
```

To test that the configuration is correct, nothing is better than trying an example. Create a new file (I've called mine `asppbase.aspx`, available in the `DelphiAsp` folder of the chapter source code) and type something like the following:

```
<html>
<body>

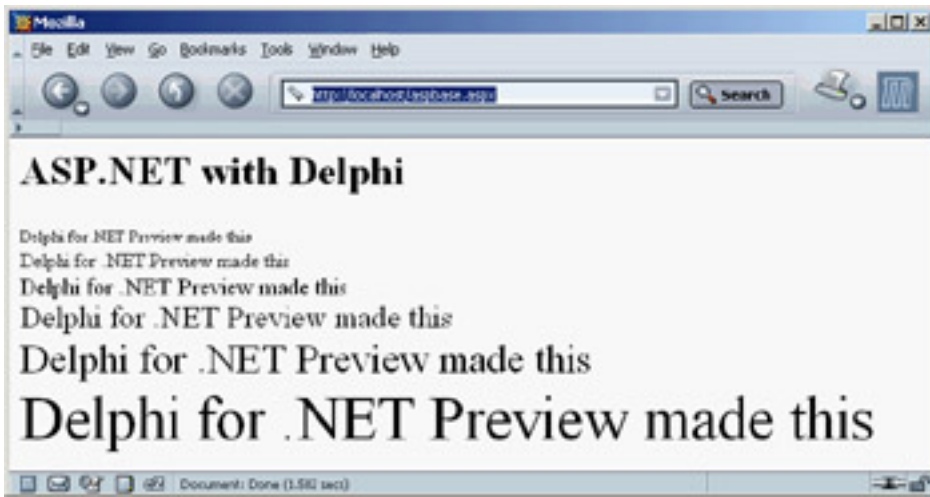
<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
procedure HelloMessage(msg: string);
var
  i: Integer;
begin
  for i := 2 to 7 do
    Response.Write ('<font size=' + inttostr (i) +
      '>' + msg + '</font> <br>')
end;
</script>

<% HelloMessage('Delphi for .NET Preview made this'); %>

</body>
</html>
```

The effect is to execute the Delphi code after transforming this file into a .NET source, compiling it with the Delphi preview compiler, and compiling the IL into assembly code (remember, even scripts in .NET are compiled before they are executed). If everything goes well, the browser should display output like that shown in [Figure 25.3](#).



**Figure 25.3:** The `aspbase.aspx` example in a browser

From this point on, you can do anything an ASP.NET application provides. The only other example I want to show you is the use of controls with event handlers, which I've covered in a different situation for .NET applications based on Windows forms.

This example, saved in the `aspui.aspx` file in the `AspDelphi` folder, uses HTML to define a form with a textbox (that is, an edit control) and a button, plus an output label. The button has a Delphi language event handler attached to it, which moves the user input to the label (the output of the program appears in [Figure 25.4](#)):

```
<html>
<body>

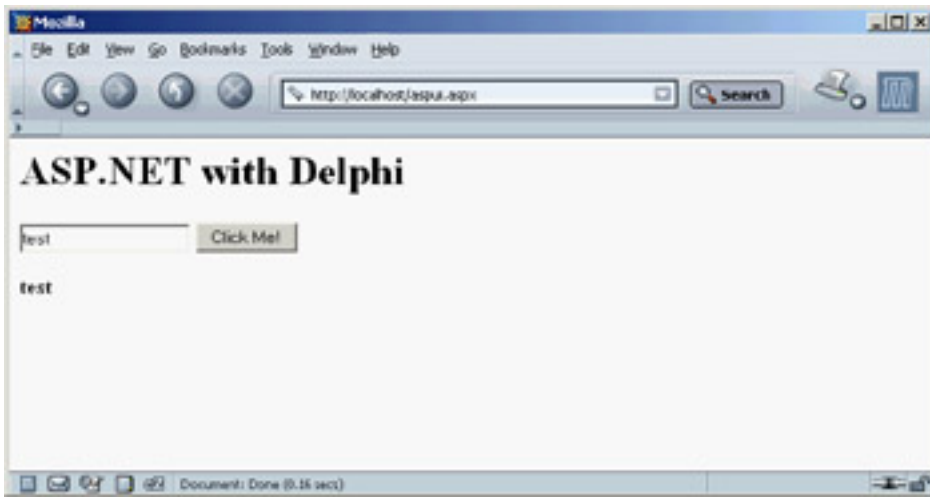
<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
  procedure ButtonClick(Sender: System.Object; E: EventArgs);
  begin
    Message.Text := Edit1.Text;
  end;
</script>

<form runat="server">
  <asp:textbox id="Edit1" runat="server"/>
  <asp:button text="Click Me!" OnClick="ButtonClick" runat="server"/>
</form>

<p><b><asp:label id="Message" runat="server" text="message"/></b></p>

</body>
</html>
```



**Figure 25.4:** The output of the `aspui.aspx` example, after typing in the edit box and clicking the button

This has been a limited introduction to ASP.NET with the Delphi language provider. However, it should give you a feeling for the possibilities opening up for Delphi programmers in this new world of .NET.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## What's Next?

While you're waiting for the Delphi for .NET product, currently code-named Galileo, you can begin experimenting with the Delphi for .NET Preview that ships with Delphi 7 (and the subsequent updates made available by Borland). Of course, you should stay tuned to Borland's Developer Network website ([bdn.borland.com](http://bdn.borland.com)) and newsgroups and to the author's site for update information about this area of Delphi, which is definitely a work in progress.

Just as Borland wants to provide the best tools to developers, I hope this book has helped you master Delphi, the most successful tool Borland has brought to the market in the last few years. Remember to check from time to time the reference, foundations, and advanced material I've collected on my website ([www.marcocantu.com](http://www.marcocantu.com)). Much of this material could not be included in this book, because of space constraints; see [Appendix C](#), "Free Companion Books on Delphi," for more information.

[Appendixes A](#) and [B](#) discuss some of the add-ins I've built, which are freely available on my site, and a few other notable free Delphi tools. Also check my site for updates and integration of the material in the book, and feel free to use the newsgroups hosted there for your questions about the book and about Delphi in general.

## Appendix A: Extra Delphi Tools by the Author

### Overview

During the last few years I have developed a number of small components and Delphi add-in tools. Some of these tools were written for books or were the result of extending examples from books. Others were written as a helper for repetitive operations. All these tools are available for free, and some include the source code. This appendix provides a list, including in particular tools mentioned in the book. Support for all these tools is available on my newsgroups (see [www.marcocantu.com](http://www.marcocantu.com) for directions).

## CanTools Wizards

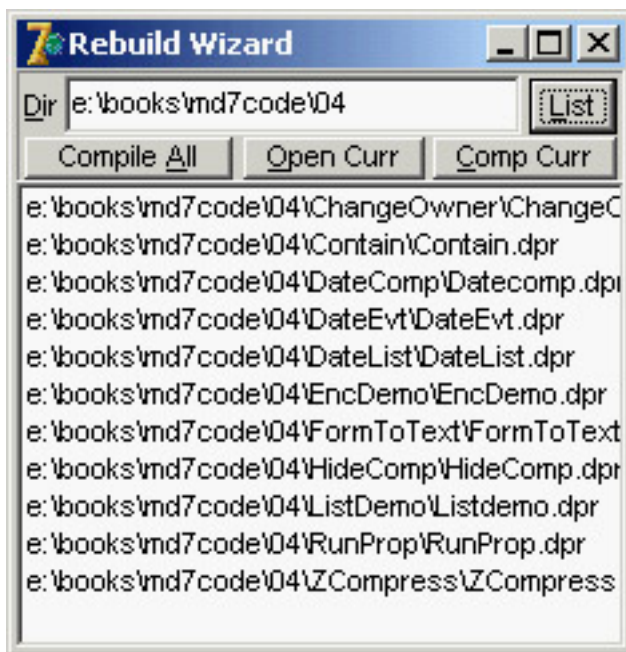
This is a set of wizards you can install in Delphi, either within an extra pull-down menu or as a submenu of the Tools menu). The wizards (freely available at [www.marcocantu.com/cantoolsw](http://www.marcocantu.com/cantoolsw)) are unrelated and offer disparate features:

**List Template Wizard** Streamlines the development of similar list-based classes, each with its own type of objects. This wizard is mentioned in [Chapter 4](#). Because it does a search-and-replace operation on a base source file, it can be used any time you need repeated code and the name of a class (or other entity) varies.

**OOP Form Wizard** (Mentioned in [Chapter 4](#).) Allows you to hide the published components of a form, making your form more object-oriented and providing a better encapsulation mechanism. Start it when a form is active, and it will fill the `OnCreate` event handler. Then you must manually move part of the code into the unit initialization section.

**Object Inspector Font Wizard** Lets you change the font of the Object Inspector (something particularly useful for presentations, because the Object Inspector's font is too small to be seen easily on a projection screen). Another option available in the settings dialog allows you to toggle an internal feature of the Object Inspector and display the font names (in the drop-down combo box for that property) using a specific font.

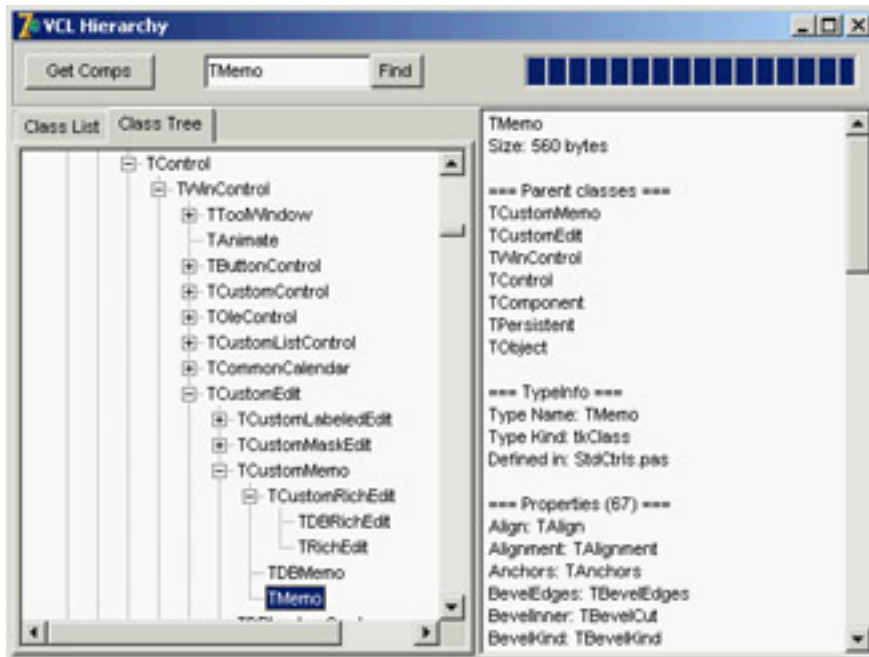
**Rebuild Wizard** Allows you to rebuild all the Delphi projects in a given subfolder after loading each of them in sequence in the IDE. You can use this wizard to grab a series of projects (like those in a book) and open the one you are interested in by clicking on the list:



You can also automatically compile a given project or start a (slow) multiproject build: In the compiler results dialog, you click a button to proceed only if the corresponding environment option is set. If this environment option is not set, you won't see the compiler errors, because the compiler messages are superseded at every compilation.

**Clip History Viewer** Keeps track of a list of text items you've copied to the Clipboard. A memo in the viewer's window shows the last 100 clipped lines. Editing the memo (and clicking Save) modifies this Clipboard history. If you keep Delphi open, the Clipboard will also get text from other programs (but only text, of course). I've seen occasional Clipboard-related error messages caused by this wizard.

**VCL Hierarchy Wizard** Shows the (almost) complete hierarchy of the VCL, including third-party components you've installed, and allows you to search one class and see many details (base and subclasses, published properties, and so on). Clicking the button regenerates both the list and the tree (in sequence—hence the progress bar runs twice):



The list of classes is generated by using predefined core classes (some are still missing; feel free to send suggestions) and then adding each component of the installed packages (Delphi's, yours, third parties'...) along with the classes of all the published properties having a class type. However, classes used only as public properties are not included.

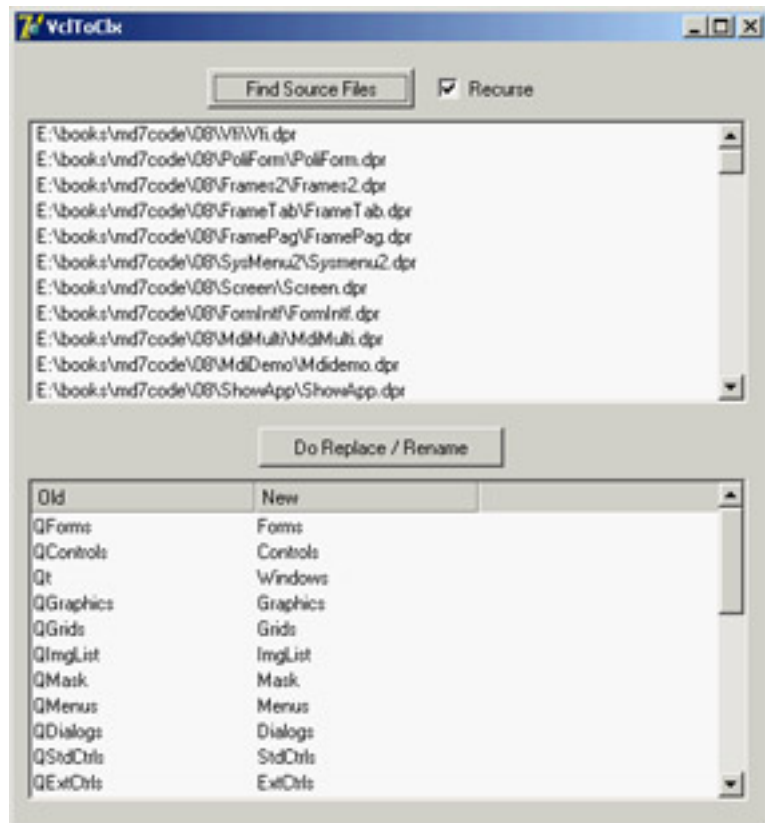
**Extended Database Forms Wizard** Does much more than the Database Forms Wizard available in the Delphi IDE, allowing you to choose the fields to place on a form and also to use a dataset other than those based on the BDE.

**Multiline Palette Manager** Allows you to turn Delphi's Component Palette into a TabControl with multiple lines of tabs:



## VclToClx Conversion Program

You can use this stand-alone tool to convert a Delphi project from VCL to CLX (and vice versa, if you configure it to do so). It can simultaneously convert all the files in a given folder and its subfolders. Here's an example of its output:

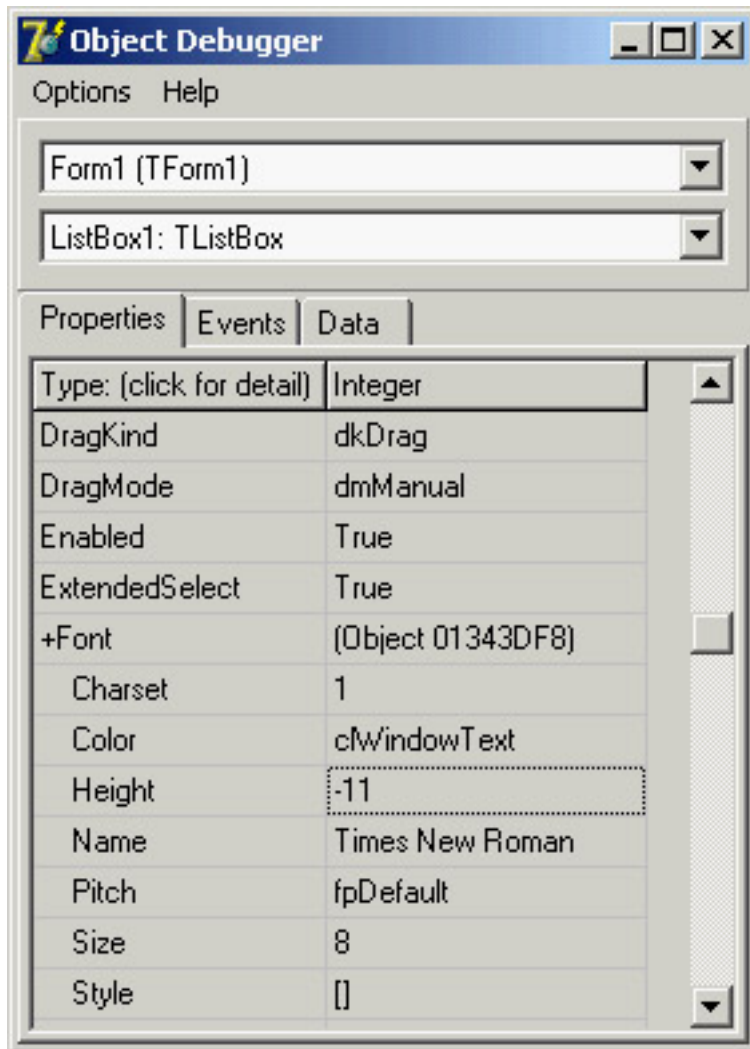


The program's source code is available in the `Tools` folder of the book's source code. The `VclToClx` program converts unit names (based on a configuration file) and handles the DFM issue by renaming the DFM files to XFM and fixing the references in the source code. The program is unsophisticated—it doesn't parse the source code but instead looks for occurrences of the unit names followed by a comma or semicolon, as happens in a `uses` statement. It also requires the unit name to be preceded by a space, but you can modify the program to look for a comma. Don't skip this extra test; otherwise the Forms unit will be converted to `QForms`, but the `QForms` unit will be re-converted to `QQForms`!



## Object Debugger

At design time, you can use the Object Inspector to set the properties of the components of your forms and other designers. In Delphi 4, Borland introduced a run-time Debug Inspector, which has a similar interface and shows similar information. Before Borland added this feature, I implemented a run-time clone of the Object Inspector meant for debugging programs:



It allows read-write access to all the published properties of a component, and has two combo boxes that let you select a form and a component within the form. Some of the property types have custom property editors (lists and so on).

You can place the Object Debugger component on a program's main form (or you can create it dynamically in code): It will appear in its own window. There is room for improvement, but even in its current form this tool is handy and has numerous users.

This component's source code is freely available in the `Tools` folder of the book's source code.

## Memory Snap

There are many tools to track the memory status of a Delphi application. During the development of a project, I had to write such a tool, and afterward I made it available.

I've written a custom memory manager that plugs into Delphi's default memory manager, keeping track of all memory allocations and de-allocations. In addition to reporting the total number (something Delphi also does now by default), it can save a detailed description of the memory status to a file.

Memory Snap keeps in memory a list of allocated blocks (up to a given total number, which can be easily varied), so that it can dump the contents of the heap to a file with a low-level perspective. This list is generated by examining each memory block and determining its nature with empirical techniques you can see in the source code (although they are not easy to understand). The output is saved in a file, because this is the only activity that doesn't require a memory allocation that would affect the results. Here is a snippet of a sample file:

```
157) 00C035CC: object: [TList - 16]
158) 00C035E0: buffer with heap pointer [00C032B0]
159) 00C03730: string: [5-1]: Edit1
160) 00C03744: object: [TEdit - 544]
161) 00C03968: object: [TFont - 36]
162) 00C03990: object: [TSizeConstraints - 32]
163) 00C039B4: object: [TBrush - 24]
164) 00C039F4: buffer with heap pointer [00C01FE4]
165) 00C03B34: buffer with heap pointer [00C01F18]
166) 00C03B48: string: [0-0]: dD
167) 00C03B58: string: [11-2]: c:\mman.log
```

The program could be extended to profile memory usage by type (strings, objects, other blocks), track unreleased blocks, and keep memory allocations under control.

Again, the source code of this component is freely available in the `Tools` folder of the book's source code.

## Licensing and Contributions

As you have seen, some of these tools are available with full source code. They are licensed under the LGPL (Lesser General Public License, [www.gnu.org/copyleft/lesser.html](http://www.gnu.org/copyleft/lesser.html)), which means you can freely use and redistribute them in any way, including making modifications, while the author retains the original copyright. The LGPL doesn't allow you to close-source your extensions, but you can use this library code within programs you sell, regardless of the source code availability. If you extend these tools by fixing bugs or adding new features, I ask you to send your updates to me so I can further distribute the enhancements and we can avoid forking the code into too many versions. The license does not require you to do so, though.

## Appendix B: Extra Delphi Tools from Other Sources

Thousands of Delphi add-on components and tools are available on the market, ranging from simple freebies to large open-source projects, from shareware programs to highly professional components. This appendix provides a list of notable open-source projects I've mentioned elsewhere in the book.

### Delphi Preinstalled Open-Source Components

Delphi 7 includes the source code for two notable open-source projects:

**Internet Direct (Indy)** This project is covered in detail in [Chapter 19](#), "Internet Programming: Sockets and Indy." The official website for Indy is [www.nevrona.com/indy](http://www.nevrona.com/indy), but you should also refer to the Indy Portal at [www.atozedsoftware.com/indy](http://www.atozedsoftware.com/indy). Support is available on Borland's newsgroups.

**Open XML** This is a Delphi-based XML DOM and SAX engine, which I covered in [Chapter 22](#) ("Using XML Technologies"). Refer to [www.philo.de/xml](http://www.philo.de/xml) for updates and more information. Support for Open XML is available on a mailing list you can sign up for on the website.

Delphi also includes a third open-source project: the ZLib compression library, covered in [Chapter 4](#) ("Core Library Classes"). It isn't listed here because it is not Delphi based.

## Other Open-Source Projects

The Delphi programmers' community has been very active since its inception and has produced numerous tools that were and are freely distributed among programmers. A subset of these tools is available with complete source code, as is often true for commercial Delphi components as well.

### Project JEDI

The Joint Endeavor of Delphi Innovators, better known as Project JEDI ([www.delphi-jedi.org](http://www.delphi-jedi.org)), is not a single project but rather is the largest community of open-source Delphi developers. The site hosts its own projects, plus others contributed and maintained by members on separate websites.

Project JEDI started as an effort to translate APIs for specific Windows libraries distributed by Microsoft or other companies. Making available the Delphi units with the declarations of those APIs allowed any Delphi developer to use them easily. More recently, the goals of Project JEDI have been extended with the definition of many subprojects and groups. In addition to an ever-growing API library, you can find projects including the JEDI Visual Component Library (JVCL), the JEDI Code Library (JCL, a set of utility functions and non-visual classes, including a nice un-handled exception stack tracer), plus many projects in the area of graphics, multimedia, and game programming. Other activities range from a JEDI Version Control System client to the DARTH header conversion kit, from a programmer's editor to online tutorials.

Other JEDI-associated projects include Indy (covered earlier), GExperts, and the Delphree site (covered next).

### GExperts

GExperts ([www.gexperts.org](http://www.gexperts.org)) is probably the most widespread add-in for the Delphi IDE, providing features ranging from a multiline Component Palette to source code navigation aids. Self-described as "a set of tools built to increase the productivity of Delphi and C++Builder programmers," it includes a large collection of wizards, including Procedure List, Clipboard History, Expert Manager, Grep Search, Grep Regular Expressions, Grep Results, Message Dialog, Backup Project, Set Tab Order, Clean Directories, Favorite Files, Class Browser, Source Export, Code Librarian, ASCII Chart, PE Information, Replace Components, Component Grid, IDE Menu Shortcuts, Project Dependencies, Perfect Layout, To Do List, Code Proofreader, Project Option Sets, and Components to Code.

### The Delphree Site

There are many other Delphi-related projects in the open-source and community development areas. Although I cannot list all the relevant ones here, I can refer you to a website that tries to track all such projects. The Delphree (Delphi Free) website is available at [delphree.clexpert.com](http://delphree.clexpert.com).

This site has a comprehensive list of Delphi open-source projects, ranging from libraries to programmers' tools to end user applications.

### DUnit

Among the many ideas promoted by extreme programming, I find unit testing particularly interesting. *Unit Testing* is the continuous development of test code even before a program is written. To do this, it is important to have a proper test framework. A group of Delphi programmers created such an architecture, called DUnit. You can find it on SourceForge at [dunit.sourceforge.net](http://dunit.sourceforge.net).

## Appendix C: Free Companion Books on Delphi

### Overview

This is the seventh edition of *Mastering Delphi*, and I've also written other books, material for classes and conference presentations, and much more; so, in addition to the thousand or so pages this book can accommodate, I have a lot of other material about Delphi programming, particularly introductory material. Over the last few years, I've turned this information into electronic books (in HTML or PDF format) available for free on my website. This appendix outlines the titles and their contents, and provides references to the download pages (all hosted on [www.marcocantu.com](http://www.marcocantu.com)).

## Essential Pascal

*Essential Pascal* is an introduction to the foundations of the language invented by Professor Nicklaus Wirth and spread by Borland with its world-famous Turbo Pascal compilers during the 1980s. The 100-page book doesn't cover the OOP extensions, but details many features Borland added to the core language over the years. The following is a list of this e-book's chapters, available at [www.marcocantu.com/epascal](http://www.marcocantu.com/epascal):

Chapter 1: "Pascal History"

Chapter 2: "Coding in Pascal"

Chapter 3: "Types, Variables, and Constants"

Chapter 4: "User-Defined Data Types"

Chapter 5: "Statements"

Chapter 6: "Procedures and Functions"

Chapter 7: "Handling Strings"

Chapter 8: "Memory (and Dynamic Arrays)"

Chapter 9: "Windows Programming"

Chapter 10: "Variants"

Chapter 11: "Programs and Units"

Chapter 12: "Files in the Pascal Language"

Appendix A: "Glossary of Terms"

Appendix B: "Examples"

As you can see on the website, the book has been translated by volunteers into quite a few languages.

## Essential Delphi

*Mastering Delphi* has changed from an introductory text to an intermediate one as I've tried to cover most of the features Borland has added to Delphi over the years. In the process, I had to cut some of the material aimed at programmers who've never used Delphi or a visual tool. This material is now contained in the *Essential Delphi* e-book, freely available in PDF format on my site at [www.marcocantu.com/edelphi](http://www.marcocantu.com/edelphi). Here is the table of contents (which is still under development):

Chapter 1: "A Form Is a Window"

Chapter 2: "Highlights of the Delphi Environment"

Chapter 3: "The Object Repository and the Delphi Wizards"

Chapter 4: "A Tour of the Basic Components"

Chapter 5: "Creating and Handling Menus"

Chapter 6: "Multimedia Fun"

Chapter 7: "Saving Settings: From INI Files to the Registry"

Chapter 8: "More on Forms"

Chapter 9: "Delphi Database 101 (with Paradox)"

Chapter 10: "Printing"

Appendix A: "Essential SQL"

Appendix B: "Common VCL Properties"



# Index

**Note to the Reader:** Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

## A

- About boxes, [289–292](#), [290](#)
- Absolute function, [921](#)
- abstract classes, [69](#), [338](#)
- abstract methods, [66–67](#)
- AbstractAnimals example, [67](#)
- acBoldUpdate method, [229](#)
- accelerator keys, [181–182](#)
- Access databases, [624](#)
- access rights in WebSnap, [808](#)
- access specifiers, [48–50](#), [925](#)
- acCountcharsUpdate method, [229](#)
- acCutUpdate method, [229](#)
- AcManTest example, [244–247](#), [244](#)
- acPasteUpdate method, [229–230](#)
- acSaveUpdate method, [229](#)
- action objects, [222](#)
- Action parameter in close, [279](#)
- Action property, [223](#)
- ActionBars collection, [244](#), [246](#)
- ActionBoldExecute method, [227](#)
- ActionCount object, [227](#)
- ActionCountExecute method, [227](#)
- ActionEnableExecute method, [227](#)
- ActionFont object, [313](#)
- ActionFontExecute method, [313](#)
- ActionFontUpdate method, [314](#)
- ActionIncreaseExecute method, [539](#)
- ActionLink object, [223](#)
- ActionList component, [173](#), [222–224](#), [223](#)
  - example, [225–228](#), [226](#), [228](#)
  - predefined actions in, [224–225](#)
  - with toolbars, [229–230](#)
- ActionList editor, [226](#), [226](#)

ActionListUpdate method, [230](#)

ActionMainMenuBar control, [242](#)

ActionManager architecture, [241–242](#)  
example, [242–245](#), [243–244](#)  
for least-recently used menu items, [245–247](#), [245](#)  
for list actions, [248–250](#), [249](#)  
for porting programs, [247–248](#)

ActionManager component, [242](#)

ActionNewExecute method, [314](#)

ActionOpenExecute method, [314](#)

actions, [173](#)  
ActionList for, [222–224](#), [223](#)  
defining, [385–388](#)  
example, [225–228](#), [226](#), [228](#)  
list, [248–250](#), [249](#)  
predefined, [224–225](#)  
for toolbars, [229–230](#)  
for transactions, [596](#)

Actions example, [225–228](#), [226](#), [228](#)

Actions property, [771–772](#)

ActionSaveAsExecute method, [314](#)

ActionSaveExecute method, [315](#)

ActionSaveUpdate method, [314](#)

ActionShowStatusExecute method, [246](#)

ActionToolBar control, [242](#)

ActionTotalExecute method, [539](#)

ActivApp example, [298](#), [298](#)

activating  
applications and forms, [298](#), [298](#)  
controls, [159](#)

activation messages, [374](#)

active documents, [456](#)

Active property, [350](#), [509](#)

ActiveButton component, [372–373](#), [373](#)

ActiveChange method, [676](#)

ActiveChanged method, [678–679](#), [683](#)

ActiveForm property, [299](#)

ActiveForm Wizard dialog box, [491](#)

ActiveForms, [491–492](#)

ActiveMDIChild property, [312](#)

ActiveScripting engine, [791](#)

ActiveX Control Wizard, [487–488](#), [487](#)

ActiveX controls, [456](#), [483–484](#)

ActiveForms, [491–492](#)

arrow, [487–488](#), [487](#)

vs. components, [484–485](#)

properties for, [488–489](#), [488](#)

property pages for, [489–491](#), [490](#)

in Web pages, [492–493](#), [493](#)

WebBrowser control, [485–486](#), [486](#)

XClock, [492–493](#), [493](#)

ActiveX Data Objects. See [ADO \(ActiveX Data Objects\)](#)

ActiveX libraries, [34](#), [460–461](#), [488](#)

ActiveX page, [461](#), [487](#), [489](#)

activity diagrams, [434](#)

actors in use case diagrams, [434](#)

AdapterCommandGroup, [795](#)

AdapterDispatcher component, [787](#)

AdapterErrorList component, [803](#)

AdapterFieldGroup component, [795](#)

AdapterGrid component, [799](#)

AdapterMode property, [801](#), [803](#)

AdapterPageProducer component, [795–797](#), [796](#)

for formview page, [802–803](#)

for logins, [807](#)

for main page, [798–801](#)

for master/detail relationships, [803](#)

adapters, [794](#)

AdapterPageProducer for, [795–797](#), [796](#)

components for, [794–795](#)

fields for, [794](#)

for locating files, [798](#)

scripts for, [797–798](#)

adCriteria constants, [640](#)

Add Field command, [531](#)

Add Fields dialog box, [526](#), [526](#)

Add Files to Model option, [437](#)

Add method

for collections, [382](#)

in Delphi for .NET Preview, [926–927](#)

in TBucketList, [132](#)

in TList, [133](#)

in TMdObjDataSet, [713](#)

- in TreeView, [194](#)
- in TStringList and TString, [128](#)
- Add To-Do Item command, [9](#)
- Add To Interface dialog box, [488–489](#), [488](#)
- Add to Model option, [437](#)
- Add To Repository command, [40](#)
- AddChild method, [194](#)
- AddChild node, [844](#)
- AddColors method, [187–188](#)
- AddDefaultCommands property, [796](#)
- AddDefaultFields property, [796](#)
- AddFilesToList function, [103](#)
- Additional page, [180](#)
- AddNode method, [197](#)
- AddObject method, [187](#)
- Addr function, [921](#)
- AddRecord method, [701](#)
- \_AddRef method, [70](#), [457](#)
- addresses
  - of functions, [63](#)
  - of methods, [115](#)
  - in socket programming, [739](#)
- AddToList method, [751](#), [753](#)
- AddWebModuleFactory method, [792](#)
- AdjustLineBreaks function, [87](#)
- administrative components in IBX, [593](#)
- ADO (ActiveX Data Objects), [508](#), [615–616](#)
  - briefcase model in, [645](#)
  - client indexes in, [632](#)
  - cloning in, [633](#), [633](#)
  - connection pooling in, [643–644](#)
  - cursors in, [629–632](#)
  - Data Link files for, [621](#)
  - dbGo for, [618–620](#), [619–620](#)
  - dynamic properties in, [622](#)
  - Jet engine for, [624–629](#), [625–626](#)
  - locks in, [635–636](#), [639–641](#)
  - in MDAC, [616–617](#)
  - OLE DB providers, [616–617](#)
  - recordsets in
    - disconnected, [642–643](#)
    - persistent, [644–645](#)

schema information in, [622–623](#), [623](#)

transaction processing in, [634–636](#)

updates in, [615](#)

batch, [638–642](#)

joins, [636–637](#)

ADO Cursor Engine, [629](#)

ADO Multi-Dimensional (ADOMD), [617](#)

ADO.NET, [645–646](#)

ADOCCommand component, [508](#), [618](#)

ADOCConnection component, [508](#), [618](#), [621](#), [628](#), [634–635](#)

ADODataSet component, [508](#), [618–619](#), [626](#)

ADOExpress, [616](#)

ADOMD (ADO Multi-Dimensional), [617](#)

ADOQuery component, [508](#), [618–619](#), [626](#)

ADOSToredProc component, [508](#), [618–619](#)

ADOTable component, [508](#), [618–619](#), [621](#), [624](#), [626](#)

ADOUpdatesPending function, [638](#)

ADOX technology, [623](#)

ADTG (Advanced Data Table Gram) format, [644–645](#)

Advanced page, [619](#)

AfterEdit event, [524](#)

AfterInsert event, [547](#)

aggregates, [550–552](#), [550](#)

Aggregates property, [550](#)

aggregation of interfaces, [921](#)

Align property, [181](#)

Alignment dialog box, [19](#)

Alignment property, [528](#), [658](#)

Alignment toolbar, [718](#)

All page, [619](#), [624](#)

AllocMemCount variable, [84](#), [335](#)

AllocMemSize variable, [84](#), [335](#)

AllocRecordBuffer method, [699](#)

AllowAllUp property, [230](#)

AllowGrayed property, [165–166](#)

AllowHiding property, [244](#)

alpha blending, [267](#), [267](#)

AlphaBlend function, [267](#)

AlphaBlendValue property, [267–268](#)

AlphaDIBBlend function, [267](#)

alter procedure statement, [565](#)

alter trigger statement, [566](#)

alternatives in XSLT, [864](#)

ampersands (&) for accelerator keys, [181](#)

ancestor classes, [63](#)

AncestorIsValid function, [148](#)

Anchor example, [179–180](#), [179](#)

anchors for controls, [179–180](#), [179](#)

Anchor property, [179](#)

angle brackets (<>) in XML, [834](#)

animated displays, [268](#)

AnimateWindow API, [268](#)

annotations on diagrams, [435](#)

AnsiContainsText function, [91](#)

AnsiDequotedStr function, [87](#)

AnsiIndexText function, [91–92](#)

AnsiMatchText function, [91–92](#)

AnsiQuoteStr function, [87](#)

AnsiReplaceText function, [91](#)

AnsiResembleText function, [91](#)

Apache Web Server, [767](#), [781–872](#)

apartment threading model, [463](#)

AppendRecord method, [701](#)

AppID property, [817](#)

Application Form option, [820](#)

application-level interfaces, [615](#)

Application mode in IntraWeb, [815](#)

Application object, [183](#), [219](#), [295–297](#), [793](#)

- for activating applications and forms, [298](#), [298](#)
- application window in, [297–298](#)
- with DLLs, [411](#)
- from TCGIApplication, [771](#)
- for tracking forms, [299–302](#), [300](#)

Application page, [295](#)

application window, [297–298](#)

Application Wizard, [825](#)

ApplicationAdapter component, [788](#)

ApplicationEvents component, [296](#)

applications

activating, [298](#), [298](#)

MDI. See [MDI \(Multiple Document Interface\) applications](#)

messages for, [374](#)

titles for, [297](#)

upsizing, [556](#)

Web. See [IntraWeb library](#)

Applications page, [28](#)

ApplicationTitle property, [788](#)

ApplyBitBtnClick method, [287](#)

ApplyUpdates method, [575](#), [586](#), [589](#), [660](#), [803](#)

AppServ2 example, [657–658](#)

AppServer property, [664](#)

AppSPlus example, [662–664](#), [668](#)

Architect Studio edition, [4](#)

Arrange option, [21](#)

Arrangelons method, [312](#)

array-based properties, [52](#)

arrays in ModelMaker, [450](#)

Arrow component, [358](#)

ActiveX, [487–488](#), [487](#)

enumerated properties for, [358–360](#)

events for, [364–366](#)

paint method for, [361–362](#), [362](#)

persistent properties for, [362–364](#), [364](#)

registering property categories for, [366–367](#), [367](#)

arrow keys for moving components, [18](#)

ArrowDbIClick method, [365](#)

ArrowHeight property, [359](#), [489](#)

AS methods in IAppServer, [650](#)

as operator, [68](#), [70](#), [465](#)

As properties, [528](#)

ASCII text files, source files as, [38](#)

AsDateTime property, [574](#)

ASM statements, [921](#)

ASP.NET, [940–942](#), [942–943](#)

aspbase.aspx example, [941–942](#), [942](#)

aspui.aspx example, [942](#), [943](#)

- assemblies
  - in .NET architecture, [906–907](#)
  - type of, [916](#)
- Assign method, [57–58](#)
  - in TClipboard and TBitmap, [208](#)
  - in TList, [128](#)
  - in TPersistent, [113](#)
- assigning objects, [56–58](#), [56](#)
- AssignPrn method, [582](#)
- AssignTo method, [113](#)
- Associate property, [172](#), [275](#)
- association symbols in diagrams, [435](#)
- associative lists, [132–133](#)
- AsSQLTimeStamp property, [574](#)
- AsText method, [495](#)
- asymmetric arithmetic rounding, [90](#)
- atomic operations, [589](#)
- AttachedMenu property, [816](#)
- attachments, [892–893](#), [894](#)
- Attribute class, [928](#)
- attributes
  - in Delphi for .NET Preview, [927–928](#)
  - in XML, [835](#)
  - in XSLT, [865](#)
- Attributes property, [635](#)
- AutoActivate property, [482](#)
- AutoCheck property, [173](#), [222](#)
- AutoComplete property, [168](#)
- AutoConnect property, [477](#)
- AutoDock property, [235](#)
- AutoDrag property, [235](#)
- AutoHotkeys property, [182](#)
- automated keyword, [921](#)
- automatic operations
  - form scaling, [276–277](#)
  - form scrolling, [272–273](#)
  - mail message generation, [748](#)
- Automation, [456](#), [467–468](#)
  - call dispatching in, [468–470](#), [469](#)
  - clients in, [474–475](#)
  - compound documents, [479–483](#), [482](#)



internal objects in, [482–483](#), [483](#)

object scope in, [476–477](#)

Office programs with, [478–479](#)

servers, [470](#), [470](#)

code for, [472–473](#)

in components, [477–478](#), [477](#)

registering, [473–474](#)

type-library editor for, [470–472](#), [471](#)

speed differences in, [476](#)

types in, [478](#)

Automation Object Wizard, [470](#), [470](#)

AutoSave feature, [11](#)

AutoScroll property, [270](#), [272](#), [275](#), [277](#)

AutoSize property, [234–235](#)

AutoSnap property, [181](#)

availability in InterBase, [561](#)

AxBorderStyle property, [492](#)

AxCtrls unit, [478](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

# Index

## B

B2B (business-to-business) applications, [875](#)

BabelFish translations, [876–879](#), [877](#), [879](#)

background processing, [304](#)

BackPages object, [212](#)

BackTask example, [305–307](#)

band components, [726–728](#), [727](#)

Band Style Editor, [727–728](#), [727](#)

Bands property, [231](#)

BandStyle property, [726](#)

banker's rounding, [90](#)

bar code components, [725](#)

BarMenuClick method, [234](#)

base classes, [330–333](#), [340](#)

base forms

inheritance from, [319–321](#), [319–320](#)

and interfaces, [330](#), [333–334](#)

base types, [84](#)

BaseCLX, [110–111](#)

batch updates, [615](#), [638–642](#)

BDE (Borland Database Engine), [507](#), [615](#)

BeforeDestruction method, [279](#)

BeforeEdit event, [524](#)

BeforePost event, [546](#)

beginning-of-file cracks, [697](#)

BeginThread function, [84](#)

BeginTrans method, [634](#)

Beveled property, [180](#)

Beyond Compare utility, [11](#)

bi constants, [256–257](#)

Blcons example, [256–257](#), [257](#)

binary large object (BLOB) fields

in DataSnap, [666](#)

in record viewer, [682](#)

streams for, [136](#), [139](#)

binding

late. See [late binding](#)

- resource files into executables, [30](#)
- binding interfaces in XML, [846–850](#), [847](#)
- Bitmap component, [724](#)
- bitmaps
  - for Component Palette, [348](#), [349](#)
  - for components, [340](#), [348](#)
  - for ListView, [188](#)
  - for Rave, [724](#)
  - for TabControl, [209](#)
  - viewer for, [207–210](#), [207](#)
- black box programming, [48](#)
- BLOB (binary large object) fields
  - in DataSnap, [666](#)
  - in record viewer, [682](#)
  - streams for, [136](#), [139](#)
- blocking sockets, [738](#)
- BlockRead function, [921](#)
- BlockWrite function, [921](#)
- BMP files, [34](#). See also [bitmaps](#)
- BmpSide constant, [210](#)
- BmpViewer example, [207–210](#), [207](#)
- Bof tests, [537](#)
- BofCracks, [697](#)
- BoldExecute method, [215](#)
- booking classes, [608–611](#), [610](#)
- Bookmark property, [538](#), [542](#)
- bookmarks, [537–538](#), [633](#), [695–698](#), [696](#)
  - shortcut keys for, [16](#)
  - in TDataSet, [687](#)
- Boolean conversions, [86](#)
- BoolToStr function, [86](#)
- border icons, [256–257](#), [257](#)
- BorderIcons property, [255–256](#)
- Borders example, [254–256](#), [254](#)
- borders for forms, [254–256](#), [254](#)
- BorderStyle property, [254–255](#)
- Borland Database Engine, [507](#), [615](#)
- Borland Registry Cleanup Utility, [33](#)
- Borland resource compiler tool, [33](#)
- BorIndMM.DLL file, [405–406](#)

BoundsRect method, [159](#)

BPG files, [28](#), [34](#)

BPL files, [34](#), [338](#), [347](#)

BPL folder, [347](#)

BRC32.exe and BRCC32.exe tools, [33](#)

Break property, [174](#)

briefcase model, [645](#)

Bring To Front command, [204](#)

BringToFront method, [302](#), [328](#)

BrokDemo example, [774–778](#), [777](#)

BrowseFast example, [756–757](#), [757](#)

browsers

- creating, [756–757](#), [757](#)
- in UDDI, [895–898](#), [896](#)

browsing in editor, [12–13](#)

bs constants, [255](#)

btnSendClick method, [745](#)

BtnTodayClick method, [56–58](#)

BucketCount property, [133](#)

BucketFor function, [133](#)

buffers and caches

- with ClientDataSet, [584](#)
- for dataset records, [698–703](#)
- in TDataSet, [687](#)

Build Later command, [27](#)

Build Sooner command, [27](#)

building projects, [30–32](#)

Bushakra, John, [899](#)

business rules, [648](#), [656–657](#), [711](#)

business-to-business (B2B) applications, [875](#)

BusinessEntity type, [895](#)

BusinessService type, [895](#)

BusinessTemplate type, [895](#)

button wheels, [261](#)

ButtonLoadClick method, [323](#)

ButtonStyle property, [516](#), [533](#)

Byte type, [478](#)

# Index

## C

C# programming language, [10](#)

C++ programming language, [401–402](#)

CAB files, [34](#), [493](#)

cached updates, [638–642](#)

CacheFile property, [893](#)

caches and buffers

with ClientDataSet, [584](#)

for dataset records, [698–703](#)

in TDataSet, [687](#)

CacheSize property, [630–631](#)

caFree value, [279](#)

caHide value, [279](#)

Calc example, [531–533](#), [533](#)

CalcController component, [733](#)

CalcOp component, [733–734](#)

CalcText component, [729–730](#)

CalcTotal component, [733](#)

CalcType property, [730](#)

calculated fields

adding, [530–533](#), [531](#), [533](#)

internally calculated, [513](#)

calculations in Rave, [733–734](#)

callback functions, [308](#)

callback interfaces, [497–498](#)

callback mechanisms, [910](#)

CallFirst example, [406](#), [407](#)

calling

DLLs, [406](#), [407](#), [408–410](#)

procedures, [409–410](#)

caMinimize value, [279](#)

CancelBatch method, [638](#)

CancelUpdates method, [638](#)

caNone value, [279](#)

CanTools wizards, [945–947](#)

Canvas property, [262](#)

CaptionPlus method, [57](#)

captions and Caption property  
for actions, [222](#)  
inputting, [260](#)  
for menu items, [174](#)  
for property pages, [490](#)  
removing, [258](#)  
in TabSheet, [203](#)  
toolbar, [236](#)

Capture property, [264–265](#)

Cascade method, [312](#)

Cascading Style Sheets (CSS), [764–765](#), [797](#), [859](#)

case-insensitive searches, [602–603](#)

case toggling, shortcut keys for, [16](#)

casting

RTTI for, [67–69](#)

type-safe down-casting, [67–69](#)

categories, property

in Object Inspector, [21–22](#)

registering, [366–367](#), [367](#)

Category property, [222](#), [224](#)

CCLibDir entry, [26](#)

CDATA section, [834](#)

CDK (Component Development Kit), [340](#)

CDS formats, [511](#), [512](#)

cdsCalcFields method, [531–532](#)

CdsCalcs example, [549–553](#), [550](#)

CdsDelta example, [586–588](#), [586](#)

CdsXstl example, [868–869](#)

CFG files, [34](#)

CGI programs, [768–769](#), [769](#)

CgiDate example, [768–769](#), [769](#)

CgiIntra example, [823–824](#)

ChangeAlignment method, [230](#)

ChangeColor method, [476](#)

Changed method, [344](#), [383](#)

ChangedAllowed property, [244](#)

ChangeFamily method, [97](#)

ChangeFormFont property, [344](#), [348](#)

ChangeOwner example, [119–120](#), [120](#)

ChangesToClone property, [712](#)

- character sets in XML, [835](#)
- characters event, [850](#), [852](#)
- characters method, [872](#)
- CheckBiDirectional function, [572](#)
- CheckBox component, [165–166](#)
- CheckBoxList component, [169](#)
- CheckCapsLock method, [218](#)
- Checked property
  - for actions, [222](#), [230](#)
  - in CheckBoxList, [169](#)
- child components, messages for, [374](#)
- child forms in MDI applications, [315–316](#), [315](#)
- child windows in MDI applications, [311–315](#), [313](#)
- CIL (Common Intermediate Language), [904](#), [907–908](#)
  - CTS in, [909–911](#)
  - managed and safe code in, [908–909](#)
- class completion
  - in editor, [13–14](#)
  - for methods, [45](#)
- Class\_DColorPropPage constant, [489](#)
- Class\_DFontPropPage constant, [489](#)
- Class\_DPImagePropPage constant, [489](#)
- Class\_DStringPropPage constant, [489](#)
- class factories, [460](#)
- class helpers, [929–931](#)
- class interfaces, [48](#)
- class references, [76–78](#), [78](#)
- classes, [109](#). See *also* [components](#)
  - in Delphi, [44–48](#)
  - diagrams for, [431–432](#), [432–433](#)
  - for exceptions, [73–75](#)
  - for fields, [529–530](#)
  - information for, [104–108](#), [108](#)
  - interposer, [332–333](#)
  - moving, [446](#)
  - in Project Explorer, [32](#)
  - reparenting, [446](#)
- Classes unit, [147–148](#)
- ClassHelperDemo example, [929](#)
- ClassInfo example, [107–108](#), [108](#)
- ClassInfo method, [106](#)

ClassName method, [104–105](#)

ClassParent method, [105](#)

ClassRef example, [77–78](#), [78](#)

ClassType method, [105](#)

clear method for dialog boxes, [283–284](#)

CLI (Common Language Infrastructure), [903–904](#)

client areas

in controls, [159](#)

in forms, [269](#)

for mouse events, [262](#)

client cursors, [629](#)

client indexes in ADO, [632](#)

client/server programming, [555–557](#)

with ClientDataSet

aggregates in, [550–552](#), [550](#)

grouping in, [549](#), [550](#)

packets and caches in, [584](#)

transactions in, [589–591](#), [591](#)

updates in, [584–589](#), [585–586](#), [588](#)

database design for, [557–561](#)

with dbExpress library, [566](#)

components, [569–574](#), [571](#), [573](#)

metadata in, [578–579](#), [579](#)

parametric queries in, [579–581](#), [580–581](#)

platforms and databases, [567–569](#)

printing in, [581–584](#)

single and multiple components with, [575–578](#), [577](#)

unidirectional cursors, [566–567](#)

with InterBase, [561–566](#), [563–564](#)

unidirectional cursors in, [560–561](#)

client socket connections, [740](#)

client windows, subclassing, [317–318](#)

ClientDataSet component, [508–509](#), [567](#), [575](#)

client/server programming with

aggregates in, [550–552](#), [550](#)

grouping in, [549](#), [550](#)

packets and caches in, [584](#)

transactions in, [589–591](#), [591](#)

updates in, [584–589](#), [585–586](#), [588](#)

for DataSnap, [649](#), [655–656](#), [656](#)

filtering for, [513](#)

formats in, [511](#)

indexing for, [512–513](#)

local tables for

connections for, [509–510](#), [510](#)

defining, [511–512](#), [512](#)



locating records in, [514–515](#)  
master/detail structures in, [552–553](#), [553](#)  
Midas library for, [510–511](#)  
for updates, [576](#)  
in WebSearch program, [785](#)  
in WebSnap, [799](#), [803](#)  
in XML, [838](#), [854–856](#), [870–873](#)

ClientHandle property, [312](#)

ClientHeight property, [269](#)

ClientRefresh example, [661](#), [661](#)

clients

Automation, [474–475](#)  
for DataSnap, [652–653](#)  
MDI, [310–311](#)  
thin, [649](#)  
in Web database applications, [831–832](#), [832](#)

ClientToScreen method, [175](#), [183](#)

ClientWidth property, [269](#)

Clip History Viewer, [946](#)

Clipboard

for actions, [229–230](#)  
for components, [24–25](#)  
for DDE, [456](#)  
with TabControl, [208–209](#)  
for toolbar operations, [214–215](#)

ClipBrd unit, [208](#)

ClipCursor function, [263](#)

clocks

digital, [349–352](#)  
XClock, [492–493](#), [493](#)

Clone method, [633](#)

cloneNode, [843](#)

clones

in ADO, [633](#), [633](#)  
property, [187](#)  
recordset, [633](#)

Close method, [279–280](#)

closing

datasets, [694–695](#)  
forms, [279–280](#)

closing markers in XML, [834–835](#)

CLR (Common Language Runtime), [905–906](#)

CLRReflection example, [934–940](#), [940](#)

CLS (Common Language Specification), [905–906](#)

CLX (Component Library for Cross-Platform), [5](#), [110–112](#)

structure of, [111–112](#)

styles for, [219–220](#), [220](#)

cm\_ prefix, [373–378](#)

cm\_DialogChar message, [377](#)

cm\_DialogKey message, [260](#), [377](#)

cm\_FocusChanged message, [378](#)

cm\_MouseEnter message, [372](#)

cm\_MouseExit message, [372](#)

CmdGotoPage command, [799](#)

CmdLine global variable, [84](#)

CmdNextPage command, [799](#)

CmdPrevPage command, [799](#)

CmExit method, [677](#)

CMNTest example, [377–378](#)

cn\_ prefix, [373](#), [377](#)

CNHScroll method, [676](#)

CNVScroll method, [676–677](#)

CoCreateGuid function, [459](#)

code completion, [14–15](#)

Code Explorer window, [11–12](#), [12](#)

code generation vs. streaming, [113–114](#)

code in Delphi for .NET Preview, [921–922](#)

code insight technology, [14](#)

code parameters, [15](#)

Code Template Parameters dialog box, [450](#), [450](#)

code templates, [15](#)

CoFirstServer class, [475](#)

ColHeights property, [683](#)

collaboration diagrams, [434](#)

collections

classes in, [131](#)

for properties, [381–385](#), [383](#)

colons (:) in to-do items, [9](#)

color

for controls, [160–161](#)

converting to strings, [187–188](#)

in DLLs, [404](#), [410–411](#)

list box of, [186–188](#)  
messages for, [377](#)  
in Object Inspector, [20](#)  
transparent, [267](#), [267](#)

Color property  
in Splitter, [180](#)  
in TControl, [159–160](#)

ColorBox component, [169](#)

ColorDialog component, [288](#)

ColorDrawItem method, [185–186](#)

ColorKeyHole example, [267–268](#), [267](#)

ColorRef type, [404](#)

Colors toolbar in Rave, [718](#)

ColorToString function, [187–188](#)

Column property, [763](#)

ColumnLayout property, [167](#)

columns and Columns property  
in DataSetTableProducer, [763](#)  
in DBGrid, [382](#), [516](#)  
editing, [539–540](#)  
in ListBox, [167](#)  
in RadioGroup, [166](#)  
totals of, [537](#), [537](#)

Columns property editor, [533](#)

COM (Component Object Model), [455–456](#)

and ADO.NET, [645–646](#)

class factories in, [460](#)

data types in, [478](#)

in Delphi, [500–502](#), [502](#)

GUIDs in, [458–460](#), [459](#)

history of, [456–457](#)

instanting and threading methods in, [463](#)

Interop services in, [908](#)

IUnknown for, [457–458](#)

server for, [460–461](#)

interface properties for, [465–466](#)

interfaces and objects for, [461–463](#), [462](#)

object initialization for, [464](#)

testing, [464–466](#)

virtual methods in, [466–467](#)

COM+, [456](#), [494](#), [650](#)

components for, [494–496](#), [495–496](#)

events for, [497–500](#), [499](#)

transactional data modules for, [497](#)

COM Component Install Wizard, [498](#)

- COM+ Event Object wizard, [498](#)
- COM Object Wizard, [461–462](#), [462](#), [498](#)
- combo boxes, [216–217](#), [341–348](#), [345–346](#)
- ComboBox component, [168](#)
- ComboBoxEx component, [169](#)
- ComConts unit, [96](#)
- ComDlg32.DLL file, [287](#)
- CommandHandlers class, [741](#)
- commands, [173](#)
  - and actions, [173](#)
  - Menu Designer for, [174](#), [174](#)
  - pop-up menus, [174–176](#)
- CommandText Editor, [573](#), [573](#)
- CommandText property
  - in SQLDataSet, [573](#)
  - in TADODataset, [626](#)
- CommandType property
  - in SQLDataSet, [573](#)
  - in TADODataset, [626](#)
- CommDlg example, [287–288](#), [288](#)
- comments
  - in ModelMaker, [444–445](#), [444](#)
  - TODO, [7–9](#)
  - in XML, [834](#)
- Commit action, [596](#)
- Commit method, [589](#)
- CommitRetaining command, [596–597](#)
- CommitTrans method, [634](#)
- common dialog boxes, [287–288](#), [288](#)
- Common Intermediate Language (CIL), [904](#), [907–908](#)
  - CTS in, [909–911](#)
  - managed and safe code in, [908–909](#)
- Common Language Infrastructure (CLI), [903–904](#)
- Common Language Runtime (CLR), [905–906](#)
- Common Language Specification (CLS), [905–906](#)
- Common Object Request Broker Architecture (CORBA), [651](#)
- Common Type System (CTS), [903–904](#), [909](#)
  - events and delegates in, [910–911](#)
  - objects and properties in, [909–910](#)
- ComObj unit, [96](#), [457](#)
- Comp type, [920](#)

CompareValue function, [89](#)

comparisons

for ClientDataSet, [513](#)

with floating-point numbers, [89](#)

with strings, [91–92](#)

compatibility

in DLLs, [404–405](#)

with Linux, [88](#)

type, in inheritance, [62–63](#)

compilation

conditional, [156](#)

for libraries, [156](#)

.NET Preview, [900](#)

for projects, [30–32](#)

compiler messages, [9–10](#), [31](#), [31](#)

Compiler Messages page, [31](#), [31](#)

Compiler page, [28](#)

complex numbers, [95](#)

Component Development Kit (CDK), [340](#)

component diagrams, [434](#)

Component Library for Cross-Platform (CLX), [5](#), [110–112](#)

structure of, [111–112](#)

styles for, [219–220](#), [220](#)

Component Object Model. See [COM \(Component Object Model\)](#)

Component Palette, [23–24](#)

bitmaps for, [348](#), [349](#)

for copying and pasting components, [24–25](#)

for databases, [507](#)

submenus on, [5](#)

templates and frames in, [25–26](#), [26](#)

Component property, [395](#)

component streaming system, [52](#)

Component Template Information dialog box, [25](#)

component templates, [25–26](#)

Component Wizard, [341–342](#)

ComponentCount property, [119](#)

ComponentIndex property, [118](#)

components, [109](#), [162](#). See also [classes](#); controls

vs. ActiveX controls, [484–485](#)

for adapters, [794–795](#)

Automation servers in, [477–478](#), [477](#)

base classes for, [340](#)

class references for, [76–78](#), [78](#)

- collection properties, [381–385](#), [383](#)
- for COM+, [494–496](#), [495–496](#)
- commands, [173–176](#)
- compound, [349](#)
  - external components in, [352–354](#), [353](#)
  - frames for, [357–358](#)
  - graphical. See [Arrow component](#)
  - interfaces for, [354–356](#)
  - internal components in, [349–350](#)
  - publishing subcomponents of, [350–352](#), [352](#)
- copying and pasting, [24–25](#)
- dynamic creation of, [47–48](#), [47](#)
- editors for, [392–395](#), [394](#)
- extending libraries for, [337–340](#)
- Fonts combo box, [341–348](#), [341](#), [345–346](#)
- hiding, [159](#)
- input focus, [176–178](#), [176–177](#)
- lists, [167–172](#), [170](#)
- messages for, [376](#)
- without names, [122](#)
- nonvisual dialog, [378–381](#), [381](#)
- for options, [165–166](#)
- packages for, [337–339](#)
- property editors, [388–392](#), [390–391](#)
- ranges, [172–173](#)
- text-input, [162–165](#), [165](#)
- windows controls. See [windows](#)
- writing, [339–340](#)

Components property, [118–120](#)

ComponentState property, [300](#)

ComponentStyle property, [351](#)

ComponentToDOM method, [845](#)

- compound components, [349](#)
  - external components in, [352–354](#), [353](#)
  - frames for, [357–358](#)
  - graphical. See [Arrow component](#)
  - interfaces for, [354–356](#)
  - internal components in, [349–350](#)
  - publishing subcomponents of, [350–352](#), [352](#)

compound documents, [456](#), [479–483](#), [482](#)

compressed CAB files, [493](#)

compressing streams, [145–146](#), [145](#)

CompressStream function, [146](#)

ComputePoints method, [360–361](#), [365](#)

ComputerName property, [652](#), [655](#)

ComServ unit, [96](#)

concurrency

access in, [557](#)

in COM+, [496](#)

InterBase for, [561](#)

conditional compilation, [156](#)

conflicts in ADO, [641–642](#)

Connected property, [655](#)

Connection component, [575](#)

connection protocols for DataSnap, [650–651](#)

connection string editor, [618–619](#), [619–620](#)

connection strings, [618–619](#)

ConnectionBroker component, [652–653](#), [666](#)

ConnectionString property, [571](#)

connections

ClientDataSet, [509–510](#), [510](#)

dbExpress, [576–577](#), [577](#)

pooling, [643–644](#)

socket, [740](#), [744–747](#)

connections.ini file, [570](#), [572](#)

ConnectionString property, [618–619](#), [621](#), [629](#)

ConnectKind property, [478](#)

constant strings, [85](#)

ConstraintErrorMessage property, [657](#)

constraints and Constraints property, [179](#), [657](#)

in DataSnap example, [657–658](#)

for forms, [270](#)

in Splitter, [180](#)

constructor keyword, [54](#)

constructors, [54–55](#), [926](#)

container classes, [131–132](#)

containers, type-safe, [133–135](#)

contains keyword, [346](#)

contains lists, [345–346](#)

Contains section, [348](#)

Content method

in DataSetTableProducer, [762](#)

in PageProducer, [760](#)

in QueryTableProducer, [778](#)

Content property, [772](#)

ContentFields property, [778](#)

- ContentStream property, [784](#)
- ContentType property, [784](#)
- Contrrs unit, [131](#)
- Contrib.INI file, [692–693](#)
- ControlBar component, [231–234](#), [232](#)
  - docking toolbars in, [235–239](#), [236](#)
  - menus ins, [234](#)
- ControlBarLowerDockOver method, [247–248](#)
- ControllerBand property, [727](#), [731](#)
- controllers in Automation, [467](#)
- controls, [111](#), [149](#). See also [components](#)
  - activation and visibility of, [159](#)
  - ActiveX. See [ActiveX controls](#)
  - anchors for, [179–180](#), [179](#)
  - colors for, [160–161](#)
  - converting, [156–158](#)
  - data-aware. See [data-aware controls](#)
  - dual libraries support for, [151–155](#), [152](#)
  - fonts, [159–160](#)
  - forms. See [forms](#)
  - ListView, [188–193](#), [191](#)
  - menus. See [menus](#)
  - owner-draw, [184–185](#)
    - list box of colors, [186–188](#)
    - menu items, [185–186](#), [186](#)
  - parents of, [158–159](#)
  - size and position of, [159](#)
  - status bars, [217–219](#), [218](#)
  - TControl for, [158–162](#)
  - ToolBar, [213–219](#), [214](#)
  - TreeView, [193–197](#), [194](#), [199](#)
  - TWidgetControl, [161–162](#)
  - TWinControl, [161](#)
  - VCL vs. VisualCLX, [149–158](#), [152](#)
  - windows. See [windows](#)
- Controls property, [159](#)
- ConvDemo example, [97](#), [98](#)
- conversions, [96–99](#), [98](#)
  - Booleans, [86](#)
  - colors, [187–188](#)
  - controls, [156–158](#)
  - currency, [99–102](#), [101](#), [880–882](#), [881](#), [883](#)
  - DFM files, [33](#), [153](#)
  - floating-point numbers, [87](#)
  - members, [447](#)



- stream data, [140–142](#)
- strings, [86–87](#), [187–188](#)
- temperatures, [96](#)
- types, [67–69](#)
- to uppercase names, [602–603](#)

CONVERT.EXE utility, [33](#), [140](#)

Convert function, [96](#)

Convert Project to Model option, [437](#)

Convert to Model option, [437](#)

Convert tool, [33](#), [140](#)

ConvertAndShow method, [141](#)

ConvertIt demo, [100](#)

ConvertService service, [880–882](#), [881](#), [883](#)

ConvUtils unit, [91](#), [96](#)

CoolBar component, [230–231](#)

coordinates

- for controls, [159](#)
- in dragging, [544](#)
- for mouse-related keys, [262](#)
- in Rave, [719](#)
- in scrolling, [273–274](#), [273](#)

Copy1Click method, [209](#)

CopyFile method, [138–139](#)

copying

- components, [24–25](#)
- files, [138–139](#)

CORBA (Common Object Request Broker Architecture), [651](#)

core library classes, [109](#), [148](#)

- Classes unit, [147–148](#)
- collections, [131](#)
- containers, [131–135](#)
- events, [124–128](#)
- lists, [128–131](#), [130](#), [133–135](#)
- RTL package, [110–112](#), [110](#)
- streaming, [135–148](#), [141](#)
- TComponent, [117–124](#), [120](#)
- TPersistent, [112–117](#), [117](#)

core syntax of XML, [834–835](#)

Count command, [316](#)

Count property, [128](#)

count(\*) wildcard, [561](#)

CountBlanks property, [730](#)

- counters
  - hit, [782–785](#)
  - in Interbase 6, [565](#)
- CountSubstr function, [93](#)
- CountSubstrEx function, [93](#)
- CppDll example, [401–402](#), [402](#)
- cracks in datasets, [697](#)
- Create constructors, [54](#)
  - for file streams, [138](#)
  - overloaded, [54–55](#)
  - in TMDClock, [351](#)
  - in TObject, [54](#)
- Create XML From Datapacket command, [856–857](#)
- CreateComObject function, [460](#), [465](#), [475](#)
- CreateComps example, [47–48](#), [47](#)
- CreateDataSet method, [511](#)
- createElement method, [842](#)
- CreateFileMapping function, [413–414](#)
- CreateForm method, [77](#), [277](#)
- CreateGUID function, [87](#)
- CreateHandle method, [161](#)
- CreateInstance method
  - in IClassFactory, [460](#)
  - in IObjectContext, [497](#)
- CreateMutex method, [308](#)
- CreateOleObject function, [475](#)
- CreateParams method, [161](#), [258](#)
- CreatePolygonalRgn method, [365](#)
- CreateTable method, [695](#)
- CreateWindowEx function, [258](#)
- CreateWindowHandle method, [161](#)
- CreateWnd method, [161](#), [327](#), [342–343](#)
- creation classes, [475](#)
- Creational Wizard, [452](#)
- CreatOrd program, [279](#)
- CSS (Cascading Style Sheets), [764–765](#), [797](#), [859](#)
- CTS (Common Type System), [903–904](#), [909](#)
  - events and delegates in, [909–910](#)
  - objects and properties in, [909–910](#)
- culture, [904](#)

CUR files, [34](#)

currency

converting, [99–102](#), [101](#), [880–882](#), [881](#), [883](#)

in Delphi for .NET Preview, [921](#)

floating-point numbers with, [87](#)

Currency property, [658](#)

CurrentYear function, [87](#)

Cursor Service, [618](#)

CursorLocation property, [629](#)

cursors

in ADO, [629](#)

locations, [629–630](#)

types, [630–631](#)

unidirectional, [560–561](#), [566–567](#)

CurValue property, [641–642](#)

CustHint example, [183–184](#), [184](#)

CustLookup example, [519](#), [519](#)

custom attributes, [927–928](#)

custom classes for streaming, [142–145](#)

custom drawing technique, [185](#)

custom method calls, [663–664](#)

custom variant types, [94–95](#), [94](#)

CustomConstraint property, [657](#)

Customizable option, [244](#)

CustomizeDlg component, [242](#), [244](#)

CustomNodes example, [197–198](#), [199](#)

CustQueP example, [774](#), [778–780](#), [780](#)

◀ PREVIOUS

NEXT ▶

# Index

## D

- D7RegClean.exe utility, [33](#)
- data access components, [725–726](#), [725](#)
- data-aware components, [728–730](#), [729](#)
- data-aware controls, [162](#), [515](#)
  - DBNavigator, [516](#)
  - graphical, [520](#)
  - grids, [516](#)
  - list-based, [517–518](#), [518](#)
  - lookup, [519–520](#), [519](#)
  - mimicking, [545–547](#), [545](#)
  - replicable, [674](#)
  - text-based, [516](#)
- data binding interfaces, [846–850](#), [847](#)
- data connections in Rave, [721–722](#), [722](#)
- data links, [669–671](#)
  - for ADO, [621](#)
  - record viewer component, [678–683](#), [682](#)
- Data Lookup Security option, [726](#)
- data modules, [535](#), [648](#)
  - for COM+, [497](#)
  - for WebSnap, [798–799](#)
- data packets, [651–652](#), [667–668](#)
- Data property
  - for OLE objects, [481](#)
  - in TBucketList, [132](#)
- Data Text Editor, [728–729](#), [729](#)
- data types, COM, [478](#)
- Data View Dictionary, [717](#), [723](#)
- DataBand component, [726–727](#)
- Database Connections option in Rave, [726](#)
- Database Desktop tool, [33](#)
- Database Explorer tool, [33](#)
- database independence, [615](#)
- databases
  - ADO for, [508](#)
  - BDE for, [507](#), [615](#)
  - client/server programming for. See [client/server programming](#)
  - ClientDataSet for. See [ClientDataSet component](#)
  - data-aware controls for. See [data-aware controls](#)

DataSet for. See [datasets and DataSet component](#); [fields](#)

dbExpress library for, [506–507](#)

design issues

entities and relations, [557–558](#)

primary keys and OIDs, [558–560](#)

error management for, [553–554](#)

grids for, [540–544](#), [542–543](#)

multitier. See [DataSnap](#)

platforms for, [567–569](#)

reports for. See [Rave](#)

sending data over socket connections, [744–747](#), [747](#)

standard controls, [544–545](#)

data-aware-like, [545–547](#), [545](#)

for sending requests, [547–549](#), [548](#)

Web applications, [825–827](#), [827](#)

client side, [831–832](#), [832](#)

linking to details, [827–830](#), [828](#), [830](#)

Web services for, [883–887](#), [887](#)

WebSnap application, [798](#)

data editing, [801–803](#), [802](#)

data module for, [798–799](#)

DataSetAdapter for, [799–801](#), [800](#)

master/detail in, [803–805](#), [805](#)

DataChange method, [673](#)

DataClone example, [633](#), [633](#)

DataCLX, [110–112](#)

DataCycle component, [730](#)

DataEvent method, [670](#)

DataField property

for data-aware controls, [515](#), [669](#), [672](#)

in Data Text Editor, [728–729](#)

datagrams, [739](#)

DataLinkDir function, [621](#)

DataMemo component, [729](#)

DataMirrorSection component, [732–733](#)

DataRelation class, [646](#)

DataSet component, See [datasets and DataSet component](#)

DataSet property

for ClientDataSet connections, [509](#)

in DataSetTableProducer, [762](#)

DataSetAdapter component, [794](#), [799–801](#), [800](#), [803–805](#), [805](#)

DataSetPageProducer component, [759](#), [761–762](#), [776](#), [789](#)

DataSetProvider component, [575](#), [653](#)

DataSetReader class, [646](#)

datasets and DataSet component, [520–524](#)

actions in, [224](#), [516](#)

class definitions for, [687–690](#)

closing, [694–695](#)

custom, [686–687](#)

in dbExpress, [572–574](#), [573](#)

directories in, [705–709](#), [709](#)

fields of. See [fields](#)

in IBX, [592–593](#)

initializing, [690–691](#)

navigating, [536–540](#), [695–698](#), [696](#)

bookmarks for, [537–538](#)

editing columns, [539–540](#)

totals of columns, [537](#), [537](#)

of objects, [710–713](#), [713](#)

opening, [690–694](#)

record buffers in, [698–703](#)

status of, [524–525](#)

testing, [703–704](#), [704](#)

DataSetTableProducer component, [759](#), [762–764](#), [763](#), [776](#), [791](#)

DataSetToDOM method, [843–844](#)

DataSnap, [647](#)

ConnectionBroker in, [666](#)

custom method calls in, [663–664](#)

data packets in, [651–652](#), [667–668](#)

example, [653](#)

client features in, [659–662](#), [659](#), [661](#)

first server, [653–655](#), [654](#)

first thin client, [655–656](#), [656](#)

server constraints in, [657–658](#)

IAppServer interface for, [649–650](#)

levels in, [647–649](#)

master/detail relations in, [664–665](#), [665](#)

object pooling in, [667](#)

parametric queries in, [663](#), [663](#)

protocols for, [650–651](#)

provider options in, [666–667](#)

SimpleObjectBroker in, [667](#)

over SOAP, [889–892](#)

support components for, [652–653](#)

technical foundation of, [649](#)

DataSource component, [575](#), [669](#)

DataSource property

for data-aware controls, [515](#), [669](#)

in SQLDataSet, [581](#)

in TDataSet, [552](#)

in TMdDbProgress, [672](#)

DataTable class, [646](#)

DataView property  
in Data Text Editor, [729](#)  
in DataBand, [726](#)

DateCopy example, [57–58](#)

DateList example, [134–135](#)

DateProp example, [51](#), [51](#)

dates  
conversions with, [87](#)  
native formats for, [709](#)  
unit for, [91](#)

DateTimeToNative function, [709](#)

DateTimeToSQLTimeStamp function, [574](#)

DateUtils unit, [91](#)

DAX (Delphi ActiveX) framework, [112](#)

DaysBetween function, [91](#)

DbAware example, [517–518](#), [518](#)

DBCheckBox component, [515](#), [517](#)

DBComboBox component, [517](#)

DBCtrlGrid component, [674](#)

DBEdit component, [515–516](#)

DBError example, [553–554](#)

dbExpress Connection Editor, [571](#), [571](#)

dbExpress Draft Specification, [567](#)

dbExpress library, [506–507](#)

client/server programming with, [566](#)  
components, [569–574](#), [571](#), [573](#)  
metadata in, [578–579](#), [579](#)  
parametric queries in, [579–581](#), [580–581](#)  
platforms and databases, [567–569](#)  
printing in, [581–584](#)  
single and multiple components with, [575–578](#), [577](#)  
unidirectional cursors, [566–567](#)

components  
dataset, [572–574](#), [573](#)  
SQLConnection, [569–574](#), [571](#)  
SQLMonitor, [574](#)  
drivers for, [567–569](#)

dbGo package, [508](#), [616](#), [618–620](#), [619–620](#)

DBGrid control, [515–516](#), [544](#)  
customizing, [683–686](#), [683](#)  
dragging with, [544](#)  
with multiple selections, [542–543](#), [543](#)

- painting, [540–542](#), [542](#)
- DBI files, [38](#)
- DBImage component, [520](#)
- DBListBox component, [517](#)
- DBLookupComboBox component, [518–519](#)
- DBLookupListBox component, [518](#)
- DBMemo component, [516](#)
- DBNavigator component, [515–516](#)
- DbProgr example, [673](#), [673](#)
- DBRadioGroup component, [517–518](#), [518](#)
- DBText component, [516](#)
- DbTrack example, [677](#), [677](#)
- DBX. See [dbExpress library](#)
- DbxExplorer example, [579](#)
- DbxMulti example, [568](#), [575–578](#)
- DbxSingle example, [575–578](#), [577](#)
- DCC.exe tool, [33](#)
- dccil compiler, [900](#)
- DCI files, [37](#)
- DCOM (Distributed COM), [650](#)
- DCOMConnection component, [652](#), [655](#)
- DCOMConnection1 class, [655](#)
- DCP files, [35](#), [347](#)
- DCR (Delphi Component Resource) project type, [348](#)
- DCT files, [38](#)
- DCU (Delphi Compiled Unit) files, [30](#), [35](#), [338–339](#), [416](#)
- .dcua files, [906–907](#)
- .dcuil files, [906–907](#)
- DDE (Dynamic Data Exchange), [456](#)
- DDP (Delphi Diagram Portfolio) files, [35](#)
- debugger, [42](#)
- Debugger Options dialog box, [74](#)
- debugging, [33](#)
  - and exceptions, [74](#)
  - SOAP headers, [887–888](#), [888](#)
  - in WebBroker technology, [772–774](#), [774](#)
- Decision Cube components, [112](#)
- declarations, external, [398](#), [406](#)



- [\\_declspec declaration, 401](#)
- [DecodeDateFully function, 87](#)
- [DecompressStream function, 146](#)
- [Decorator pattern, 449](#)
- [DefAttributes property, 164](#)
- [default exception handlers, 74](#)
- [default keyword for styles, 343](#)
- [default values for methods, 46](#)
- [DefaultColumnWidth property, 680](#)
- [DefaultDrawColumnCell method, 540](#)
- [DefaultDrawing property, 540, 682](#)
- [DefaultExpression property, 657](#)
- [DefaultRowHeight property, 683, 685](#)
- [DefaultStyle property, 219](#)
- [DefaultTextLineBreakStyle variable, 85](#)
- [DefaultTimeout property, 805](#)
- [Define command in Fields editor, 526](#)
- [DefineProperties method, 113, 187, 683](#)
- [DefinePropertyPage method, 491](#)
- [DefinePropertyPages method, 491](#)
- defining
  - [actions, 385–388](#)
  - [events and event handlers, 364–365](#)
  - [symbols, 156](#)
- definitions
  - [external, 400](#)
  - [links to, 13](#)
- [delayed signing, 917](#)
- [delegates in CTS, 909–910](#)
- [delegation in event handling, 124](#)
- [Delete method, 128](#)
- [delete operations, trigger firing from, 565](#)
- [Delete1Click method, 209](#)
- [Delphi ActiveX \(DAX\) framework, 112](#)
- [Delphi Compiled Unit \(DCU\) files, 30, 35, 338–339, 416](#)
- [Delphi compiler tool, 33](#)
- [Delphi Component Palette, 287](#)
- [Delphi Component Resource \(DCR\) project type, 348](#)

Delphi Diagram Portfolio (DDP) files, [35](#)

Delphi for .NET Preview, [900–902](#), [903](#), [919](#)

ASP.NET with, [940–942](#), [942–943](#)

class helpers in, [929–931](#)

custom attributes in, [927–928](#)

deprecated features for, [920–922](#)

extended identifiers in, [924](#)

final and sealed keywords in, [924–925](#)

Microsoft libraries for, [934–940](#), [940](#)

multicast events in, [926–927](#)

nested types in, [926](#)

run-time library for, [930–931](#)

static members in, [925–926](#)

unit namespaces in, [922–924](#)

VCL for, [931–933](#), [933](#)

visibility and access specifiers in, [925](#)

Delphi internal messages, [374](#)

Delphi language, [43–44](#)

class references in, [76–78](#), [78](#)

classes and objects in, [44–48](#)

for components, [339](#)

constructors in, [54–55](#)

dynamic objects in, [47–48](#), [47](#)

encapsulation in, [48](#)

access specifiers for, [48–50](#), [60–62](#)

and forms, [52–54](#), [53](#)

with properties, [50–52](#), [51](#)

exceptions in, [71–75](#), [76](#)

inheritance in, [59–63](#), [60](#)

interfaces in, [69–71](#)

late binding in, [63–67](#), [64](#)

type-safe down-casting in, [67–69](#)

Delphi Power Book, [954](#)

Delphi project (DPR) files, [39](#)

DELPHI32.DCT file, [26](#)

DelphiMM unit, [95–96](#)

Delphree site, [952](#)

delta caches, [652](#)

Delta memory area, [514](#)

delta packets, [651](#)

Delta property, [585–586](#), [586](#)

deltas

in aggregates, [550](#)

in batch updates, [638](#)

in ClientDataSet, [585–586](#), [586](#), [652](#)

- DEM files, [38](#)
- democript pages, [792–793](#), [792](#)
- dependencies, [339](#)
- deployment
  - in library selection, [155](#)
  - in .NET architecture, [916–917](#), [917](#)
- deployment diagrams, [434](#)
- deprecated features, [920–922](#)
- descendant classes, [63](#)
- DESCRIPTION directive, [346](#)
- Description property, [353](#)
- design critics, [452](#)
- design-only component packages, [338](#)
- design patterns, [447–450](#)
- design-time packages, [338](#)
- design-time properties, [52](#)
- Designer page, [19](#), [280](#)
- Designer toolbar in Rave, [718](#)
- Desktop (DSK) files, [36](#), [39](#)
- desktop settings, saving, [5–6](#)
- Dessena, Nando, [600](#)
- Destination property, [249](#)
- DestParan property, [733](#)
- Destroy method, [55](#), [59](#), [912–914](#), [914](#)
- DestroyComponents method, [118](#)
- destroying objects, [58–59](#)
- destructors, [55](#), [59](#)
- DestructorTest class, [912–914](#), [914](#)
- DetailKey property, [731](#)
- details, linking, in Web databases, [827–830](#), [828](#), [830](#)
- Details view in Object Repository, [40](#)
- ~DF files, [35](#)
- DFM files, [19](#), [33](#), [35](#), [38](#), [152–154](#)
- DFN files, [35](#)
- Diagram view, [16–18](#), [17–18](#)
- diagrams
  - for classes, [431–432](#), [432–433](#)
  - common elements in, [435–436](#), [436](#)

non-UML, [435](#)  
for sequences, [433–434](#), [433](#)  
for use cases, [434](#)

dialog actions, [224](#)

dialog boxes, [280–281](#)  
creating, [282–287](#), [284–285](#)  
modeless, [285–287](#), [285](#)  
nonvisual, [378–381](#), [381](#)  
predefined, [287–289](#), [288](#)

dialog menu items, [173](#)

diamond symbol in Band Style Editor, [728](#)

Difference view, [442–443](#), [443](#)

digital clock, [349–352](#)

DirDemo example, [709](#), [709](#)

Direct Data View option, [726](#)

Direct Driver View option, [726](#)

direct form input, [259](#)  
keyboard, [259–261](#), [260](#)  
mouse, [261–265](#)

direct memory access functions, [921](#)

Direction property, [489](#)

directives, [44](#), [347](#)

directories  
for components, [347](#)  
in datasets, [705–709](#), [709](#)

Directories/Conditionals page, [477](#)

DisableCommit method, [497](#)

DisableControls method, [538–539](#)

DisabledImages property, [213](#)

DisableIfNoHandler property, [223](#), [227](#)

disabling aggregates, [551](#)

disconnected recordsets, [642–643](#)

dispatching calls in Automation, [468–470](#), [469](#)

dispid keyword, [470](#)

dispinterface keyword, [469–470](#), [472](#), [488](#)

Display method, [422](#)

DisplayFormat property, [658–659](#), [733](#)  
in DataSetTableProducer, [763](#)  
in TFloatField, [527](#), [532](#)

DisplayLabel property, [527–528](#), [658](#)

DisplayType property, [801](#)

DisplayValues property, [658](#)

DisplayWidth property, [528](#), [658](#)

Dispose method, [912](#)

Distributed COM (DCOM), [650](#)

distribution of updated packages, [416](#)

DivideTwicePlusOne example, [73–75](#)

DivMod function, [89](#)

DlgApply example, [285–287](#), [285](#)

DllCanUnloadNow function, [461](#)

DLLGetClassObject method, [460](#)

DllMem example, [412–413](#)

DllRegisterServer function, [461](#)

DLLs (dynamic link libraries), [35](#), [397](#)

- for ActiveX controls, [484](#)
- C++, [401–402](#)
- calling, [406](#), [407](#), [408–410](#)
- creating, [402–406](#)
- dynamic linking in, [397–398](#)
- exporting strings from, [404–406](#)
- forms in, [410–411](#)
- with ISAPI, [769–770](#)
- in memory, [411–414](#), [415](#)
- overloaded functions in, [404](#)
- and packages, [30](#), [338](#), [345](#), [415–416](#)
- purpose of, [398–399](#)
- rules for, [399–400](#)
- using, [400–402](#), [402](#)
- wizards for, [41](#)

DllUnregisterServer function, [461](#)

DMT files, [37](#)

doAutoIndent option, [842](#)

DoChange method, [126–127](#)

Dock method, [235](#)

DockClientCount property, [235](#)

DockClients property, [235](#)

docking support, [234–235](#)

- in control bars, [235–239](#), [236](#)
- messages for, [376](#)
- for page controls, [239–241](#), [241](#)

DockPage example, [239–241](#), [241](#)

DockSite property, [234](#)

- DoConvert method, [98–99](#)
- DoCreate method, [330–331](#)
- document management in XML, [837–838](#), [840](#), [843](#), [846](#)
- Document Object Model (DOM)
  - programming with, [838–846](#), [840](#), [843](#), [846](#)
  - XSL transformations with, [868–869](#)
- document type definitions (DTDs), [849](#)
- documentation in ModelMaker, [444–445](#), [444](#)
- DoDestroy method, [331](#)
- Does Not Support Transaction option, [495](#)
- DOF files, [28](#), [35](#)
- DOM (Document Object Model)
  - programming with, [838–846](#), [840](#), [843](#), [846](#)
  - XSL transformations with, [868–869](#)
- domain names, [739](#)
- DomCreate example, [839](#), [842–846](#), [843](#), [846](#)
- DOMDocument property, [838](#)
- DOMPersist property, [836](#)
- DOMVendor property, [836](#)
- dot-notation for methods, [46](#)
- Double type, [478](#)
- DoVerb method, [481](#)
- down-casting, type-safe, [67–69](#)
- ~DP files, [36](#)
- DPK files, [35](#), [345](#)
- DPKL files, [35](#)
- DPKW files, [35](#)
- DPR (Delphi project) files, [36](#), [39](#)
- dragging, [98](#)
  - with DBGrid control, [544](#)
  - with mouse, [262–265](#), [265](#)
  - in Object TreeView, [23](#)
  - source code files, [27](#)
- DragKind property, [234–235](#)
- DragMode property, [98](#)
  - in TControl, [234–235](#)
  - in TreeView, [195](#)
- DragToGrid example, [544](#)
- DragTree example, [194–197](#), [194](#)
- DrawCell method, [680–682](#)

[DrawColumnCell](#), [685–686](#)  
[DrawData](#) program, [540–542](#), [542](#)  
[DrawFocusRect](#) method, [264](#)  
drawing  
    with mouse, [262–265](#), [265](#)  
    Rave components for, [725](#)  
[DrawPoint](#) method, [263](#)  
[DrawText](#) function, [209](#), [682](#)  
[DriverName](#) property, [570](#)  
drivers for dbExpress, [567–569](#)  
drivers.ini file, [569–570](#)  
Drivers Settings window, [571](#), [571](#)  
DRO files, [37](#)  
drop-down fonts, [21](#)  
drop procedure statement, [565](#)  
drop trigger statement, [566](#)  
[DropDownMenu](#) property, [213](#), [215](#)  
[DropDownRows](#) property, [516](#)  
[DropDownWidth](#) property, [519](#)  
[dsBrowse](#) value, [524](#)  
[dsCalcFields](#) value, [524](#)  
[dsCurValue](#) value, [524](#)  
[dsEdit](#) value, [524](#)  
[dsFilter](#) value, [524](#)  
[dsInactive](#) value, [524](#)  
[dsInsert](#) value, [524](#)  
[DSIntf.pas](#) unit, [510](#)  
DSK (Desktop) files, [36](#), [39](#)  
DSM files, [36](#)  
[dsNewValue](#) value, [524](#)  
[dsOldValue](#) value, [524](#)  
DST files, [6](#), [38](#)  
DTDs (document type definitions), [849](#)  
dual interfaces, [470](#)  
dual libraries support, [151–155](#), [152](#)  
DUnit architecture, [952](#)  
DynaCall example, [408–410](#)

DynaForm example, [252–253](#), [253](#)

dynamic aggregation of interfaces, [921](#)

dynamic binding. See [late binding](#)

dynamic components, [47–48](#), [47](#)

dynamic cursors, [631](#)

Dynamic Data Exchange (DDE), [456](#)

dynamic database reporting, [776–777](#), [777](#)

dynamic link libraries. See [DLLs \(dynamic link libraries\)](#)

dynamic linking, [397–398](#)

dynamic methods vs. virtual, [65–66](#)

dynamic pages, [764–765](#)

dynamic properties, [622](#)

dynamic Web pages, [768](#)

and CGI, [768–769](#), [769](#)

DLLs, [769–770](#)

DynaPackForm example, [419–420](#)

◀ PREVIOUS

NEXT ▶



# Index

## E

e-mail, [748–750](#), [749](#)

E object, [75](#)

early binding, [63](#)

ebXML (Electronic Business using Extensible Markup Language), [894](#)

EchoActionFieldValue property, [797](#)

EchoMode property, [163](#)

ECMAScript, [791](#)

EDBClient class, [553](#)

Edit command, [520](#)

Edit component, [163](#)

Edit method

in IComponentEditor, [393](#)

in TSoundProperty, [389–390](#)

Edit Tab Order dialog box, [176](#), [176](#)

Edit To-Do Item window, [8](#), [8](#)

edit verbs, [480](#)

EditFirstNameExit method, [178](#)

EditFormat property, [658](#)

editing

actions for, [224](#)

columns, [539–540](#)

connection properties, [571](#)

HTML, [788](#)

in-place, [480](#)

records, [520](#)

textual form descriptions, [38](#)

visual, [484](#)

WebSnap data, [801–803](#), [802](#)

editions, [3–4](#)

EditLabel property, [163](#)

EditMask property, [163–164](#), [658](#)

Editor Properties dialog box, [10–11](#), [10](#)

editors, [10–11](#), [10](#)

ActionManager, [242–245](#), [243–245](#)

for columns, [533](#)

for commands, [573](#), [573](#)

for components, [392–395](#), [394](#)

for connection strings, [618–619](#), [619–620](#)

Delphi

- browsing in, [12–13](#)
- class completion in, [13–14](#)
- Code Explorer window in, [11–12](#), [12](#)
- code insight technology in, [14](#)
- loadable views in, [16–18](#), [17–18](#)
- shortcut keys in, [16](#)
- for fields, [525–527](#), [526](#), [550](#)
- for images, [33](#), [348](#)
- for input masks, [163–164](#)
- Method Implementation Code Editor, [441–442](#), [442](#)
- for packages, [345](#), [345](#), [347–348](#), [417](#), [417](#)
- for properties, [388–392](#), [390–391](#), [533](#)
- type-library, [470–472](#), [471](#)
- for unit code, [439–441](#)

EDivByZero exception, [73](#)

Electronic Business using Extensible Markup Language (ebXML), [894](#)

ellipsis (...) on menus, [173](#)

embedded units, [568–569](#)

embedding, [480–481](#)

EMDataSetError class, [687–688](#)

Empty Project template, [40–41](#)

empty values with field events, [535](#)

Enable property, [863](#)

EnableCommit method, [497](#)

EnableControls method, [538–539](#)

Enabled property

- for actions, [222](#), [227](#)
- in TControl, [159](#)
- in Timer, [350](#)

encapsulation, [48](#)

- access specifiers for, [48–50](#), [60–62](#)
- and forms, [52–54](#), [53](#)
- with properties, [50–52](#), [51](#)

end-of-file cracks, [697](#)

endDocument event, [850](#), [852](#)

endElement event, [850](#), [872](#)

EndThread function, [84](#)

EndUser object, [793](#), [795](#)

EndUserSession adapter, [795](#)

EndUserSessionAdapter component, [807–808](#)

EnlargeFont1Click method, [816](#)

EnsureRange function, [89](#)

Enterprise Studio edition, [4](#)

entities in database design, [557–558](#)

enumerated properties, [358–360](#)

EnumModules function, [84](#), [425–426](#)

EnumWindows functions, [308](#)

EnumWndProc method, [308–309](#)

Environment Options dialog box, [6–7](#)

- for AutoSave feature, [11](#)
- for Code Explorer, [11–12](#)
- for compiler, [30](#)
- for DFM file saving, [19](#)
- for HTML editing, [788](#)
- for Project Explorer, [32](#)
- for secondary forms, [280](#)
- for type libraries, [471](#)

environment variables in CGI programs, [769](#)

Environment Variables page, [7](#)

Eof tests, [536](#)

EofCracks, [697](#)

EqualsValue constant, [89](#)

equi-joins, [636](#)

error method, [851](#)

ErrorLog example, [75–76](#), [76](#)

errors. See also [exceptions](#)

- in ADO, [641](#)
- database, [553–554](#)
- logging, [75–76](#), [76](#)

Essential Delphi, [954](#)

Essential Pascal, [953–954](#)

etm60.exe tool, [33](#)

EuroConv example, [100–102](#), [101](#)

EuroRound function, [102](#)

euros, converting, [100–102](#), [101](#)

event-driven programming, [302–303](#)

Event Editor in Rave, [717](#), [731–732](#)

Event Types view, [443](#)

EventHandler method, [162](#)

events, [124](#)

- for ActiveX controls, [484](#)
- for applications, [296](#)
- for Arrow component, [364–366](#)

- for COM+, [497–500](#), [499](#)
- in CTS, [909–910](#)
- in DataSnap example, [658–659](#)
- defining, [364–365](#)
- in Delphi for .NET Preview, [926–927](#)
- in Interbase [6](#), [565–566](#)
- method pointers for, [125–126](#)
- null values with, [535–536](#), [536](#)
- as properties, [126–128](#)
- in Qt, [162](#)

EvtSubscriber library, [498](#)

Excel

- with Automation, [479](#)
- Jet engine for, [625–626](#), [626](#)

Excel IISAM, [626](#)

except blocks, [71–73](#)

Exception class, [75](#)

Exception1 example, [73–75](#)

exceptions, [71](#)

- in calculated fields, [532](#)
- classes for, [73–75](#)
- for components, [339](#)
- and debugging, [74](#)
- error logging, [75–76](#), [76](#)
- program flow and finally block in, [72–73](#)

Exclude method, [927](#)

EXE files, [36](#)

ExecSQL method, [572](#)

executable files, [30](#), [82–83](#)

executable viewer tool, [33](#)

Execute method

- in TFindWebThread, [753](#)
- in TMdListBoxDialog, [380](#)
- in TMdListCompEditor, [395](#)
- in TPrimeAdder, [305–306](#)

ExecuteTarget method, [385–387](#)

ExecuteVerb method

- in IComponentEditor, [393](#)
- in TMdListCompEditor, [394](#)

exitItemClick method, [937](#), [939–940](#)

ExitProc function, [921](#)

ExpandAll1lick method, [816](#)

ExpandFileName function, [87](#)

- expanding component references in-place, [20](#)
- explicit transactions, [589](#)
- Explorer page, [32](#)
- exploring projects, [32](#), [32](#)
- exporting
  - Jet engine for, [628–629](#)
  - strings from DLLs, [404–406](#)
- exports clause
  - in Apache, [782](#)
  - for DLLs, [399](#), [404](#)
- expression evaluation, tooltip, [16](#)
- ExtCtrls unit, [163](#)
- Extended Database Form Wizard, [947](#)
- extended identifiers, [924](#)
- extended ListBox selections, [167](#)
- Extended Properties property, [624](#), [626](#), [629](#)
- ExtendedSelect property, [167](#)
- extending libraries, [337–340](#)
- Extensible Stylesheet Language (XSLT), [864](#)
  - examples, [865–866](#)
  - with WebSnap, [866–868](#), [866](#)
  - XPath in, [864–865](#)
- external components
  - in compound components, [352–354](#), [353](#)
  - interfaces for, [354–356](#)
- external declarations, [398](#), [406](#)
- external definitions for DLLs, [400](#)
- external keys, [559](#)
- external symbols, [15](#)
- External Translation Manager tool, [33](#)
- \$EXTERNALSYM directive, [400](#)
- ExtraCode property, [197](#)
- ExtractStrings method, [147](#)
- Extras key for hot-track activation, [24](#)

# Index

## F

- fake implementation code, [498](#)
- fArrowDbClick method pointer, [364–365](#)
- fatalError method, [851](#)
- FCL (Framework Class Library), [904](#)
- FetchBlobs method, [584](#), [666](#)
- FetchDetails method, [584](#), [666](#)
- FetchOnDemand property, [666](#)
- FetchParams method, [663](#)
- field-oriented data-aware controls
  - read-only progress bar, [671–673](#), [673](#)
  - read-write TrackBar, [674–677](#), [677](#)
- Field property, [671](#)
- FieldAcc example, [527–528](#), [528](#)
- FieldAddress method, [115](#)
- FieldByName method, [525](#), [527](#)
- FieldDefs property, [511](#), [690](#)
- FieldLookup example, [534](#), [534](#)
- FieldName property
  - in TField, [527](#)
  - in TFieldDataLink, [671](#)
- fields
  - for adapters, [794](#)
  - in databases, [525–527](#), [526](#), [698–702](#)
    - calculated, [530–533](#), [531](#), [533](#)
    - class hierarchy for, [529–530](#)
    - constraints on, [657–658](#)
    - events for, [658–659](#)
    - lookup, [533–534](#), [534](#)
    - null values with events for, [535–536](#), [536](#)
    - objects for, [527–528](#), [528](#)
    - properties for, [658](#)
  - in Delphi for .NET Preview, [926](#)
  - hiding, [122–123](#)
  - names for, [360](#)
  - removing, [121–122](#)
  - virtual, [50](#)
- Fields editor, [525–527](#), [526](#), [550](#)
- Fields property, [525](#), [527](#)
- FieldsToXml function, [886](#)

FieldValues property, [525](#), [527](#)

file management, [102–104](#), [103–104](#)

File Transfer Protocol (FTP)

API for, [754–755](#)

ports for, [739](#)

FileCreate function, [87](#)

FileName property, [244](#)

files

actions for, [224](#)

copying, [138–139](#)

locating, [798](#)

produced by system, [34–38](#)

source code, [38–39](#)

streaming, [138–139](#)

FilesList example, [102–104](#), [103–104](#)

Filled property, [359](#), [366–367](#), [489](#)

FillFormList method, [299–300](#)

FillFormsList method, [300](#)

Fills toolbar in Rave, [718](#)

Filter property, [513](#)

FilterGroup property, [638](#), [641](#)

filtering for ClientDataSet, [513](#)

final keyword, [924–925](#)

Finalize method, [911–912](#)

finally blocks, [71–73](#)

Find and Replace dialog boxes, [288](#)

find\_business API, [897](#)

Find method, [132](#)

FindClose function, [102](#)

FindComponent method, [121](#), [286–287](#), [765](#)

FindFirst function, [102](#)

FindGlobalComponent function, [121](#)

FindNearest method, [777](#)

FindNext function, [102](#)

FindNextPage function, [205–206](#)

FindWindow function, [307](#)

FirstDll example, [405–406](#)

FishClient example, [893](#), [894](#)

Flat property, [213](#)

- floating-point numbers
  - comparisons with, [89](#)
  - conversions with, [87](#)

FloatingDockSiteClass property, [234](#)

FloatToCurr function, [87](#)

FloatToDateTime function, [87](#)

fly-by-hints, [182–184](#), [184](#), [218](#), [218](#)

fm flags, [138](#)

FocusControl property, [177](#)

Font dialog box, [288](#), [288](#)

Font property, [21](#), [159](#)

FontDialog component, [288](#)

FontMaster component in Rave, [724](#)

FontMirror property, [724](#)

FontNamePropertyDisplayFontNames variable, [21](#)

fonts

- in Automation, [478](#)
- for controls, [159–160](#)
- for dialog boxes, [288](#), [288](#)
- listing, [299](#)
- for menu items, [216](#)
- in Object Inspector, [21](#)
- size of, [276–277](#)
- with toolbars, [215](#)

Fonts combo box, [341–344](#)

- package for, [344–347](#), [345–346](#)
- using, [347–348](#)

Fonts toolbar in Rave, [718](#)

ForEach method, [133](#)

ForEachFactory method, [462](#)

ForEachModule function, [425–426](#)

Form component, [119](#)

Form Designer, [18–19](#)

Format function, [85](#), [582](#)

FormatDateTime function, [60](#)

formats for Rave, [720–721](#), [720](#)

FormatXMLData method, [842](#), [857](#)

FormCount property, [821](#)

FormCreate method, [127](#)

- for pages, [204](#)
- in TDateForm, [127](#)



in TForm1, [196–197](#)

in TFormUserIntf, [423](#)

FormDestroy method, [131](#)

FormDLL example, [411](#)

FormIntf demo, [330–331](#)

FormMouseDown method, [263–264](#)

FormMouseMove method, [263–264](#)

FormMouseUp method, [263](#), [265](#)

FormProp example, [53](#), [53](#)

forms, [251](#)

About boxes and splash screens for, [289–292](#), [290](#)

activating, [298](#), [298](#)

border icons for, [256–257](#), [257](#)

borders for, [254–256](#), [254](#)

client area in, [269](#)

closing, [279–280](#)

constraints in, [270](#)

creating, [277–279](#), [278](#)

dialog boxes

creating, [282–287](#), [284–285](#)

predefined, [287–289](#), [288](#)

direct input

keyboard, [259–261](#), [260](#)

mouse, [261–265](#)

displaying, [267–268](#), [267](#)

in DLLs, [410–411](#)

and encapsulation, [52–54](#), [53](#)

fields in

hiding, [122–123](#)

removing, [121–122](#)

inheritance in

from base forms, [319–321](#), [319–320](#)

polymorphic, [321–324](#), [322](#)

multiple-page, [202](#)

image viewer in, [207–210](#), [207](#)

PageControl and TabSheet for, [202–207](#), [204–205](#)

for user interface wizard, [210–213](#), [211](#)

in packages, [417–419](#), [417](#)

painting in, [265–267](#)

plain, [252–253](#), [253](#)

position of, [268–269](#)

properties for, [53–54](#), [53](#)

scaling, [274–277](#)

scrolling, [270–271](#)

automatic, [272–273](#)

coordinates in, [273–274](#), [273](#)

example, [271–272](#), [272](#)

secondary, [280–281](#)  
size of, [269](#)  
snapping in, [269](#)  
splitting, [180–181](#), [180](#)  
style of, [254](#)  
TForm class, [251–259](#), [253–254](#), [257–258](#)  
tracking, [299–302](#), [300](#)  
WebBroker technology for, [777–781](#), [780](#)  
windows for, [257–259](#), [258](#)

Forms page, [28](#), [277](#), [278](#), [280](#)

FormStyle property, [254](#), [257](#), [311](#)

FormToText program, [140–141](#), [141](#)

forward-only cursors, [630–631](#)

Frac function, [921](#)

FramePag example, [327–328](#), [327](#)

frames, [324–326](#), [325](#)

in Component Palette, [26](#), [26](#)

for compound components, [357–358](#)

in MDI applications, [311–315](#), [313](#)

with pages, [326–328](#), [327](#)

without pages, [328–330](#), [329](#)

Frames1 example, [26](#), [26](#)

Frames2 example, [324–326](#), [325](#)

FrameTab example, [328–330](#), [329](#)

Framework Assembly Registration Utility (regasm), [501](#)

Framework Class Library (FCL), [904](#)

Free method, [55–56](#), [58–59](#), [912](#)

free query forms, [613](#), [613](#)

Free threading model, [463](#)

FreeAndNil method, [55](#), [59](#)

FreeInstance class, [335](#)

FreeLibrary function, [408–409](#)

FreeMem method, [84](#), [921](#)

FreeNotification method, [302](#), [354](#), [356](#)

FreeRecordBuffer method, [699](#)

FROM clause in SELECT, [628](#)

FromStart property, [826](#)

fs constants, [254](#), [311](#)

FTimer class, [351](#)

FTP (File Transfer Protocol)

API for, [754–755](#)

ports for, [739](#)

FullExpand method, [194](#)

functions

addresses of, [63](#)

names for, [360](#)

overloaded, [46](#), [404](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

# Index

## G

GAC (Global Assembly Cache), [917](#), [917](#)

GACUTIL utility, [917](#)

garbage collection, [911–916](#)

Garbage Test example, [915–916](#)

gen\_id function, [566](#)

GeneratorField property, [593](#), [601](#)

generators, [565–566](#), [600–602](#), [601](#)

generic projects, [922](#)

generic unit references, [923](#)

generic units, [922](#)

Get methods

in TAXForm1, [491](#)

in WebBroker, [778](#)

Get\_Value method, [473](#)

GetAttributes function, [388–389](#)

GetBookmarkData method, [696](#)

GetBookmarkFlag method, [696](#)

GetBufStart method, [194](#)

GetCanModify method, [706](#)

GetCds method, [892](#)

GetClass function, [419](#)

GetCollString method, [384](#)

GetColor function, [410–411](#)

GetControl function, [386–387](#)

GetData method, [547](#), [702](#)

GetDataAsXml method, [857](#)

GetDataField method, [672](#)

GetDataSource method, [672](#)

GetDriverFunc function, [570](#)

GetDriverVersion method, [568](#)

GetEmployeeData method, [885–886](#)

GetEmployeeNames method, [884–885](#)

GetEnumeratorName function, [585](#)

GetEnvironmentVariable function, [87](#)

GetExtraCode function, [197](#)

GetField method  
in TMdDataSetOne, [703](#)  
in TMdDbProgress, [672](#)

GetFieldData method  
in TMdDataSetOne, [702–703](#)  
in TMdDirDataset, [708](#)  
in TObjDataSet, [711–712](#)

GetFileVersion function, [88](#)

GetHeapStatus function, [84](#)

GetIDsOfNames method, [468](#)

GetInterfaceExternalName, [882](#)

GetKeyState function, [218](#)

GetLines method, [379–380](#)

GetLocaleFormatSettings function, [88](#)

GetLongHint method, [218](#)

GetMem method, [84](#), [921](#)

GetMemo function, [335](#)

GetMemoryManager function, [84](#), [334](#)

GetMethExternalName, [882](#)

GetModuleFilename, [309](#)

GetNamePath method, [113](#)

GetNewId function, [601](#)

GetNextPacket method, [584](#)

GetObjectContext method, [497](#)

GetOleFont method, [478](#)

GetOleStrings function, [478](#)

GetOptionalParameter method, [668](#)

GetOwner method  
in TMdMyCollection, [382–383](#)  
in TPersistent, [113](#)

GetPackageDescription function, [424](#)

GetPackageInfo function, [424](#), [426–427](#)

GetPackageInfoTable function, [84](#), [424](#)

GetProcAddress function, [408](#), [412](#)

GetPropInfo function, [116](#)

GetPropValue function, [116](#)

GetRecNo method, [698](#)

GetRecord method, [699–700](#)

GetRecordCount method, [698](#), [891](#)

- GetRecordSize method, [699](#)
- GetSellItem function, [380](#)
- GetShareData method, [414](#)
- GetShortHint method, [218](#)
- GetStrProp function, [116](#)
- GetSubDirs function, [103](#)
- GetSystemMetrics function, [259](#)
- GetText method, [59–60](#)
- GetTextHeight method, [187](#)
- GetTickCount function, [476](#)
- GetValue method, [390](#)
- GetValues method, [389](#)
- GetVerb method
  - in IComponentEditor, [393](#)
  - in TMDListCompEditor, [394](#)
- GetVerbCount method, [393](#)
- GetWindowLong function, [298](#)
- GetYear method, [51](#)
- GExperts tool, [952](#)
- Global Assembly Cache (GAC), [917](#), [917](#)
- global data
  - for DLLs, [400](#)
  - in Project Explorer, [32](#)
  - in System unit, [84](#)
- global deployment, [916](#)
- global handlers, [72](#)
- Global Page Catalog node, [717](#), [723](#)
- GlobalCount variable, [821](#)
- GlobalEnter method, [178](#)
- Globally Unique Identifiers (GUIDs)
  - conversions with, [87](#)
  - in DCOMConnection, [655](#)
  - in Delphi for .NET Preview, [928](#)
  - for interfaces, [69](#)
  - in IUnknown, [458–460](#), [459](#)
- Google website, [751–752](#)
- GotoPage method, [485](#)
- GrabHtml method
  - in TFindWebThread, [752–753](#)
  - in WebSearch program, [785](#)
- graphic toolbars, [213](#)

graphical components in Rave, [724](#)

graphical controls, [158](#). See also [images](#)

compound components. See [Arrow component](#)

data-aware, [520](#)

hit counters, [783–784](#)

GreaterThanValue constant, [89](#)

Green Pages in UDDI, [895](#)

Grep.exe tool, [33](#)

GridDemo example, [686](#)

grids

data-aware controls for, [516](#)

for databases. See [DBGrid control](#)

GroupBox component, [119](#), [166](#)

Grouped property, [230](#)

GroupIndex property

for actions, [230](#)

for menu items, [481](#)

for radio items, [173](#)

groups

in ClientDataSet, [549](#), [550](#)

project, [26–27](#)

GUIDs (Globally Unique Identifiers)

conversions with, [87](#)

in DCOMConnection, [655](#)

in Delphi for .NET Preview, [928](#)

for interfaces, [69](#)

in IUnknown, [458–460](#), [459](#)

GUIDToString function, [459](#)

◀ PREVIOUS

NEXT ▶

# Index

## H

H files, [401–402](#)

hacks, [62](#)

Handle property

in Application, [297](#)

in TWinControl, [161](#)

HandleMessages function, [303–304](#)

HandleNeeded method, [161](#)

HandleReconcileError function, [588](#)

HandlesPackages lists, [422](#)

HandlesTarget method, [385](#)

hashed associative lists, [132–133](#)

header files, [401–402](#)

headers in CGI programs, [769](#)

Height property

for components, [18](#)

for forms, [268–269](#)

in TControl, [159](#)

in TMdArrow, [359](#)

HelloWorld example, [901–902](#), [903](#)

Help actions, [224](#)

Help files, [33](#), [340](#)

HelpContext property, [222](#), [256](#)

Hide method, [159](#)

HideComp example, [122–123](#)

HideOptions property, [808](#)

HideUnused property, [246](#)

hiding

components, [159](#)

form fields, [122–123](#)

forms, [279](#)

high-level protocols in socket programming, [740](#)

high-low technique, [600](#)

hint directives, [44](#)

Hint property

for actions, [222](#)

in Application, [217](#)

for toolbars, [236](#)

HintColor property, [182](#)



HintHidePause property, [182](#)

HintPause property, [182](#)

hints in ToolBar, [182–184](#), [184](#), [218](#), [218](#), [236](#)

HintShortPause property, [182](#)

hit counters, [782–785](#)

HitsGetValue method, [806](#)

HookEvents method, [162](#)

horizontal splitting of forms, [181](#)

HorzScrollBar property, [270](#), [273](#)

hosts file in IP addresses, [739](#)

hosts in socket programming, [741](#)

HOSTS.SAM file, [739](#)

hot-track activation, [24](#)

HotImages property, [213](#)

Hower, Chad Z., [809](#)

HTM files, [36](#)

HTML (Hypertext Markup Language), [758–759](#)

- for data, [761–762](#), [762](#)
- for dynamic pages, [764–765](#)
- for pages, [759–761](#), [760](#)
- producer components for, [759](#)
- style sheets for, [764–765](#)
- for tables, [762–764](#), [763](#)
- templates for, [788](#)

HTML editor, [788](#)

HTML Import, [629](#)

HTML tab, [862](#)

HTMLDoc property, [788](#)

HtmlDom component, [868](#)

HtmlEdit example, [165](#), [165](#)

HTMLFile property, [788](#)

HtmlProd example, [759–764](#), [760](#), [762](#)

HtmlStringToCds method, [785](#)

HtmlToList method, [752–754](#)

HTTP (Hypertext Transfer Protocol), [651](#), [750–751](#)

- API for, [754](#)
- for browsers, [756–757](#), [757](#)
- grabbing content in, [751–754](#), [755](#)
- ports for, [739](#)
- server, [757–758](#), [758](#)

WinInet API for, [754–756](#)

HttpDecode function, [743](#)

HttpEncode function, [743](#)

HTTPRIO component, [879](#)

HttpServ example, [757–758](#), [758](#)

HTTPSoapDispatcher component, [879](#)

HTTPSoapPascalInvoker component, [879](#)

httpssvr.dll file, [651](#)

HTTPWebNode property, [893](#)

HttpWork method, [754](#)

Hypertext Markup Language. See [HTML \(Hypertext Markup Language\)](#)

Hypertext Transfer Protocol. See [HTTP \(Hypertext Transfer Protocol\)](#)

◀ PREVIOUS

NEXT ▶

# Index

## I

IAppServer interface, [649–650](#), [667](#), [890–891](#)

IAppServerOne interface, [654](#)

IAppServerSOAP interface, [891](#)

IB provider, [618](#)

IB\_WISQL tool, [564](#)

IBClasses dataset, [610](#)

IBClassReg dataset, [609–610](#)

IBConsole application, [562–564](#), [563–564](#)

IBDatabase component, [592](#)

IBDatabasInfo component, [592](#)

IBDataSet component, [592](#)

IBEvents component, [592](#)

IBM DB2 drivers, [567](#)

IBPeopleReg dataset, [610](#)

IBQuery component, [592](#)

IBSQL component, [592](#)

IBSQLMonitor component, [592](#), [598](#)

IBStoredProc component, [592](#)

IBTable component, [592](#)

IBTransaction component, [592](#)

IBUpdateSQL component, [592](#)

IBX. See [InterBase Express \(IBX\)](#)

IbxEmp example, [593–595](#)

IbxMon example, [598](#), [599](#)

IbxUpdSql example, [595–597](#), [598](#)

ICO files, [34](#)

IComponentEditor interface, [393](#)

icons

for applications, [297](#)

for borders, [256–257](#), [257](#)

IdAntiFreeze component, [738](#)

IDAPI (Independent Database Application Programming Interface), [615](#)

IDE (Integrated Development Environment), [3–4](#), [5](#)

compiler messages, [9–10](#)

Component Palette. See [Component Palette](#)

debugger, [42](#)

- desktop settings in, [5–6](#)
- editor. See [editors](#)
- environment settings in, [6–7](#)
- Form Designer, [18–19](#)
- libraries in, [5](#)
- menus, [7](#)
- Object Inspector, [19–23](#)
- project management in, [26–32](#), [27](#), [32](#)
- to-do lists, [7–9](#), [8](#)
- tools, [33–34](#)

identifiers, [924](#)

IDispatch interface, [84](#), [458](#), [467–469](#), [851](#)

IDisposable interface, [912](#), [915](#)

idle state, [302](#)

IdMessage class, [748](#)

IDockManager interface, [235](#)

IDOMAttr interface, [838](#)

IDOMDocument interface, [838](#)

IDOMEElement interface, [838](#)

IDOMNode interface, [838](#)

IDOMNodeList interface, [838](#)

IDOMParseError interface, [836](#)

IDOMText interface, [838](#)

IdPop3 component, [748](#)

IDs

- GUIDs. See [GUIDs \(Globally Unique Identifiers\)](#)
- in Interbase, [600–602](#), [601](#)

IdSMTP component, [748](#)

IdTCPClient component, [741](#)

IdTCPServer component, [741](#)

idThreadMgrDefault component, [738](#)

idThreadMgrPool component, [738](#)

leFirst example, [860–864](#), [861](#), [863](#)

\$IF directive, [44](#)

IFirstServer interface, [471–473](#)

IFontDisp interface, [478](#)

IFormOperations interface, [333](#)

IfSender example, [106–107](#), [106](#)

IfThen function, [88–89](#), [92](#)

ignorableWarning method, [851](#)

- IIDs (interface IDs), [459](#)
- IInterface interface, [69](#), [84](#), [457](#)
- IInvokable interface, [84](#), [878](#)
- IISAM (Installable Indexed Sequential Access Method) drivers, [624](#)
- IL Disassembler (ildasm.exe), [908](#)
- ildasm.exe (IL Disassembler), [908](#)
- Image Editor tool, [33](#), [348](#)
- image viewer in multiple-page forms, [207–210](#), [207](#)
- ImagEdit.exe tool, [33](#)
- ImageIndex property
  - for actions, [222](#)
  - in ListView, [189](#)
  - for menu items, [174](#)
- ImageList class, [189](#), [203](#), [213](#)
- images
  - for dialog boxes, [288](#)
  - in ListView controls, [188–193](#), [191](#)
  - on menus, [185](#)
- Images property
  - for menu items, [174](#)
  - for toolbars, [213](#)
- IMdInform interface, [498](#)
- IMdWArrowX interface, [488](#)
- implementation diagrams, [434](#)
- Implementation File Update Wizard, [495](#)
- implementation section
  - for DLLs, [400](#)
  - uses statement in, [256](#)
- implicit transactions, [589](#)
- Import Source File dialog box, [438](#)
- Import Type Library dialog box, [477](#)
- ImportedConstraint property, [657](#)
- importing
  - Jet engine for, [628–629](#)
  - in ModelMaker, [437–438](#)
- IN clause in SELECT, [628](#)
- in-place activation, [484](#)
- in-place editing, [480](#)
- in-place expansion of component references, [20](#)
- in-process servers, [457](#)
- Include method, [927](#)

[IncludePathURL](#) property, [860](#)

[IncludeTrailingPathDelimiter](#) function, [87](#)

[Increase](#) method, [49](#), [495](#)

[Increment](#) property, [270](#)

incremental searches, shortcut keys for, [16](#)

indenting, shortcut keys for, [16](#)

Independent Database Application Programming Interface (IDAPI), [615](#)

[IndexDefs](#) property, [513](#)

indexes

- in ADO, [632](#)
- in [ClientDataSet](#), [512–513](#)

[IndexFieldNames](#) property, [512](#), [632](#)

[Indexing Service](#), [618](#)

[IndexOf](#) method, [128](#)

Indy (Internet Direct) components, [738](#), [741–744](#), [744](#)

- HTML. See [HTML \(Hypertext Markup Language\)](#)
- HTTP. See [HTTP \(Hypertext Transfer Protocol\)](#)
- for mail, [748–750](#), [749](#)

Indy (Internet Direct) project, [951](#)

[IndyDbSock](#) example, [747](#), [747](#)

[IndySock1](#) example, [741–744](#), [744](#)

[INetXCustom](#), [859](#)

[INetXPageProducer](#), [858–859](#), [861–863](#), [861](#)

[INetXPageProducer](#) editor, [863](#)

infinite constants, [88](#)

Infinity constant, [88](#)

[InflateRect](#) function, [682](#)

[InFocus](#) example, [176–178](#), [177](#)

information hiding, [48](#)

inheritance

- from existing types, [59–63](#), [60](#)
- form, [318–319](#)
  - from base forms, [319–321](#), [319–320](#)
  - polymorphic, [321–324](#), [322](#)
- in [ModelMaker](#), [437–438](#)
- restricting, [447](#)
- type compatibility in, [62–63](#)

inherited keyword, [65](#), [272](#), [320–321](#)

[InheritsFrom](#) method, [67](#), [105](#)

INI files, [331–332](#)

- initialization
  - of COM objects, [464](#)
  - constructors for, [54](#)
  - of datasets, [690–691](#)
- initialization section, [473–474](#)
- Initialize method
  - for automation servers, [474](#)
  - in TNumber, [464](#)
- InitializeControls method, [936–937](#)
- InitializeMenu method, [936](#), [939–940](#)
- InitWidget method, [162](#)
- inline keyword, [325](#)
- inner transactions, [634–635](#)
- input, forms
  - keyboard, [259–261](#), [260](#)
  - mouse, [261–265](#)
- input focus
  - handling, [176–178](#), [176–177](#)
  - messages for, [374](#)
- Input Mask editor, [164](#)
- InputBox function, [289](#)
- InputQuery function, [289](#)
- InquireSoap interface, [897](#)
- InRange function, [89](#)
- Insert command for records, [520](#)
- Insert method, [128](#)
- insert operations, trigger firing from, [565](#)
- InsertComponent method, [118–119](#)
- InsertObjectDialog method, [481](#)
- InsertRecord method, [701](#)
- inside-out activation, [484](#)
- Install COM+ Object dialog box, [495–496](#)
- Install Into New Application page, [496](#)
- Installable Indexed Sequential Access Method (IISAM) drivers, [624](#)
- installing
  - DLL wizards, [41](#)
  - property editors, [391–392](#)
- instances, checking for, [307–310](#)
- InstanceSize method, [105](#)
- instancing in COM, [463](#)

Integer type, [478](#)

Integrated Development Environment. See [IDE \(Integrated Development Environment\)](#)

Integrated Translation Environment (ITE), [368](#), [399](#)

## InterBase

generators and triggers in, [565–566](#)

history of, [562](#)

IBConsole application, [562–564](#), [563–564](#)

IBX. See [InterBase Express \(IBX\)](#)

operation of, [561–562](#)

real-world examples

booking classes, [608–611](#), [610](#)

case-insensitive searches, [602–603](#)

free query forms, [613](#), [613](#)

generators and IDs, [600–602](#), [601](#)

locations and people, [603–605](#)

lookup dialogs, [611–613](#)

user interfaces, [605–608](#), [607](#)

server-side programming in, [564–565](#)

InterBase Express (IBX), [507](#), [591](#)

administrative components in, [593](#)

dataset components in, [592–593](#)

example, [593–594](#)

live queries in, [594–597](#), [598](#)

monitoring, [598](#), [599](#)

system data in, [599–600](#), [599](#)

InterBase Objects program, [564](#)

InterBase Workbench tool, [564](#)

InterceptGUID property, [652](#)

interface IDs (IIDs), [459](#)

interface section, [400](#)

Interface Wizard, [432](#), [433](#)

Interfaced Component Reference model, [21](#)

interfaces, [48](#), [69–71](#)

and base forms, [330](#), [333–334](#)

for COM server, [461–463](#), [462](#), [465–466](#)

for compound components, [354–356](#)

of controls, [483](#)

in packages, [420–424](#)

user. See [user interface](#)

InterfaceSupportsErrorInfo method, [458](#)

InterlockedDecrement function, [458](#)

InterlockedIncrement function, [458](#), [822](#)

internal components in compound components, [349–350](#)

internal instancing, [463](#)



internal messages, [302](#), [372–374](#), [373](#)

internal objects in Automation, [482–483](#), [483](#)

InternalAddRecord method  
in TMdCustomDataSet, [699](#)  
in TMdDataSetOne, [701](#)

InternalAfterOpen method  
in TMdCustomDataSet, [691–692](#)  
in TMdDirDataset, [707](#)

InternalCancel method, [712](#)

InternalClose method, [694](#)

InternalDelete method, [699](#)

InternalEdit method, [712](#)

InternalFieldDefs method, [710–711](#)

InternalFirst method, [697](#)

InternalGotoBookmark method, [697](#)

InternalHandleException method, [703](#)

InternalInitFieldDefs method  
in TMdCustomDataSet, [692–694](#)  
in TMdDirDataset, [707–708](#)

InternalInitRecord method, [699](#)

InternalLast method, [698](#)

InternalLoadCurrentRecord method  
in TMdDataSetStream, [700–701](#)  
in TMdListDataSet, [706](#)

internally calculated fields, [513](#)

InternalOpen method, [690–691](#)

InternalPost method, [699](#), [701](#)

InternalPreOpen method  
in TMdCustomDataSet, [691](#)  
in TMdListDataSet, [705](#)

InternalRecordCount method  
in TMdCustomDataSet, [698–699](#)  
in TMdListDataSet, [706](#)

InternalRethinkHotkey method, [182](#)

InternalScriptName property, [774](#)

InternalSetToRecord method, [697](#)

Internet, actions for, [224](#)

Internet Direct (Indy) components, [738](#), [741–744](#), [744](#)  
HTML. See [HTML \(Hypertext Markup Language\)](#)  
HTTP. See [HTTP \(Hypertext Transfer Protocol\)](#)  
for mail, [748–750](#), [749](#)

Internet Direct (Indy) project, [951](#)

Internet Express, [858](#)

JavaScript support in, [859–864](#), [861](#), [863](#)

XMLBroker component in, [858](#)

Internet page, [7](#), [788](#)

Internet programming, [737](#)

HTML. See [HTML \(Hypertext Markup Language\)](#)

HTTP. See [HTTP \(Hypertext Transfer Protocol\)](#)

for mail, [748–750](#), [749](#)

with sockets. See [socket programming](#)

XML. See [XML](#)

Internet Server API (ISAPI), [769–770](#)

InternetCloseHandle function, [755](#)

InternetOpen function, [755](#)

InternetOpenURL function, [755](#)

InternetReadFile function, [755](#)

Interop services, [908](#)

interposer classes, [332–333](#)

IntfColSel unit, [421–422](#)

IntfFormPack package, [421–422](#)

IntfFormPack2 package, [421](#)

IntfPack example, [421](#)

IntfPack package, [421](#)

INTO clause in SELECT, [628](#)

InTransaction property, [589](#)

IntraWeb library, [809–810](#)

architectures in, [815](#)

for Web applications, [810–815](#), [812](#)

building, [815–818](#), [817](#)

databases, [825–832](#), [827–828](#), [830](#), [832](#)

layout, [824–825](#), [825](#)

multipage, [818–821](#), [819](#)

sessions management, [821–823](#), [823](#)

with WebBroker, [823–824](#)

IntraWeb page, [810](#)

IntToStr function, [85](#)

INumber interface, [465](#)

invalid areas, [266](#)

Invalidate method, [266–267](#)

InvalidateRect function, [267](#)

InvalidateRegion function, [267](#)

Invoke method, [468–469](#)

IObjectContext interface, [497](#)

IP addresses, [739](#)

IPictureDisp interface, [478](#)

IProduceContent interface, [772](#)

IProperty interface, [388](#)

is operator, [67–68](#)

ISAPI (Internet Server API), [769–770](#)

IsCallerInRole method, [497](#)

IsConsole variable, [84](#)

IsCursorOpen method, [695](#)

IsDefaultPropertyValue function, [148](#)

IsEqualGUID function, [87](#)

IsInfinite function, [88](#)

IsInTransaction method, [497](#)

IsLibrary variable, [84](#)

IsMultiThread variable, [84](#)

IsNan function, [88](#)

IsNull method, [703](#)

isolation levels in transactions, [590](#)

IsSecurityEnabled method, [497](#)

IStrings interface, [478](#)

IsUniDirectional function, [572](#)

ISupportErrorInfo interface, [458](#)

IsZero function, [89](#)

ITE (Integrated Translation Environment), [368](#), [399](#)

ItemHeight property, [187](#)

ItemIndex property, [250](#)

- in ListBox, [167](#)
- for radio group components, [259](#)

ItemProps property, [171](#)

Items property

- in ActionBars, [246](#)
- for fonts, [343](#)
- for images, [189](#)
- in ListBox, [167](#)
- in TBookMarkStr, [542](#)
- in TList, [133](#)
- in TreeView, [194](#)

Items string list, [169](#)

ItemsEx property, [169](#)

IUnknown interface, [69](#), [84](#), [457–458](#)  
class factories in, [460](#)  
GUIDs in, [458–460](#), [459](#)

IVBSAXContentHandler interface, [851](#)

IVBSAXErrorHandler interface, [851](#)

IW Component Park, [810](#)

IWAppFormCreate method, [818](#), [827](#)

IWButton component, [811](#)

IWChart component, [810](#)

IWClientGrid example, [831–832](#), [832](#)

IWClientSideDataSet component, [831](#)

IWClientSideDataSetDBLink component, [831](#)

IWCSLabel component, [831](#)

IWCSNavigator component, [831](#)

IWDataModulePool component, [810](#)

IWDialogs component, [810](#)

IWDynamicChart component, [831](#)

IWDynGrid component, [831](#)

IWEdit component, [811](#)

IWGranPrimo component, [810](#)

IWGridDemo example, [828–830](#), [828](#), [830](#)

IWLayoutMgrHTML component, [824](#)

IWListbox component, [811](#)

IWModuleController component, [823](#)

IWOpenSource component, [810](#)

IWPageProducer component, [823](#)

IWScrollData example, [826–827](#), [827](#)

IWServerControllerBaseNewSession method, [822](#)

IWSession example, [821–823](#), [823](#)

IWSimpleApp example, [811–815](#), [812](#)

IWTranslator component, [810](#)

IWTree example, [816–818](#), [817](#)

IWTwoForms example, [818–821](#), [819](#)

IXMLDocument interface, [839](#), [848](#)

IXMLDocumentAccess interface, [836](#)

IXMLNode interface, [839](#)

IXMLNodeCollection interface, [839](#)

IXMLNodeList interface, [839](#)

◀ PREVIOUS

NEXT ▶

# Index

## J

JavaScript

in Internet Express, [859–864](#), [861](#), [863](#)

for WebSnap, [791](#)

JavaScript language, [814](#)

JEDI (Joint Endeavor of Delphi Innovators), [952](#)

Jet engine, [624](#)

for Excel, [625–626](#), [626](#)

for importing and exporting, [628–629](#)

for Paradox, [624–625](#), [625](#)

for text files, [626–628](#)

JIT (Just in Time) compiler, [907–908](#)

joins, [636–637](#)

Joint Endeavor of Delphi Innovators (JEDI), [952](#)

JPEG file format, [784](#)

JpegImage component, [783–784](#)

JScript, [791](#)

Just in Time (JIT) compiler, [907–908](#)

# Index

## K

KeepRowTogether property, [731](#)

keyboard

for form input, [259–261](#), [260](#)

messages for, [377](#)

KeyField property, [519](#)

KeyOptions property, [172](#)

KeyPreview property, [259](#)

keys

in associative lists, [132–133](#)

in database design, [558–560](#)

in keyset cursors, [631](#)

keyset cursors, [631](#)

Kind property, [286](#)

KPreview example, [259–260](#), [260](#)

Kylix, CLX in, [110](#)

# Index

## L

LabelDoubleClick method, [285–286](#)

LabeledEdit component, [163](#)

LabelPosition property, [163](#)

LabelSpacing property, [163](#)

Language Exceptions page, [74](#)

large documents with XML, [869–873](#), [871](#)

LargeImages property, [189](#)

LargeXML example, [869–871](#), [871](#)

LastSession property, [246](#)

late binding

abstract methods in, [66–67](#)

for message handlers, [66](#)

overriding and redefining methods in, [64–65](#)

and polymorphism, [63–64](#), [64](#)

virtual vs. dynamic methods in, [65–66](#)

layout for Web applications, [824–825](#), [825](#)

Layout Manager, [825](#), [825](#)

LayoutChanged method, [684–685](#)

least-recently used menu items, [245–247](#), [245](#)

Left property

for components, [18](#)

for forms, [268](#)

in TControl, [159](#)

Lesser General Public License (LGPL), [949](#)

LessThanValue constant, [89](#)

LGPL (Lesser General Public License), [949](#)

LIB file, [401](#)

LibComp example, [153–154](#)

\$LIBPREFIX directive, [407](#)

libraries

ActiveX, [34](#), [460–461](#)

for controls

dual, [151–155](#), [152](#)

visual, [155–156](#), [156](#)

DataSnap, [649](#)

for Delphi for .NET Preview, [934–940](#), [940](#)

DLL. See [DLLs \(dynamic link libraries\)](#)

extending, [337–340](#)

in IDE, [5](#)



IntraWeb. See [IntraWeb library](#)  
names for, [407–408](#)  
type, [468–470](#)

library path setting, [347](#)

library statement, [403](#)

LibSpeed example, [155](#)

\$LIBSUFFIX directive, [407](#)

\$LIBVERSION directive, [407](#)

LIC files, [36](#)

licenses

for ActiveX controls, [488](#)

for author tools, [949](#)

for Qt libraries, [150](#)

line breaks in text files, [85](#)

Lines class, [379](#)

Lines string list, [164](#)

Lines toolbar in Rave, [718](#)

LineStart method, [147](#)

LinkedActionLists property, [247](#)

linking, [480](#)

to definitions, [13](#)

dynamic vs. static, [397–398](#)

to Web database details, [827–830](#), [828](#), [830](#)

Linux

compatibility functions for, [88](#)

libraries for, [155–156](#), [156](#)

list actions, [224](#), [248–250](#), [249](#)

list-based data-aware controls, [517–518](#), [518](#)

list boxes, [167](#), [186–188](#)

list components. See [lists and list components](#)

List Template Wizard, [945](#)

ListActions example, [248–250](#), [249](#)

ListBox component, [167](#)

ListBoxForm form, [378–379](#)

ListCli example, [478](#)

ListControl property, [249](#)

ListDate example, [132](#)

ListDemo example, [129–131](#), [130](#)

ListDialDemo example, [381](#), [381](#)

ListDialog component, [393–395](#), [394](#)

- listening socket connections, [740](#)
- lists and list components, [128–131](#), [130](#)
  - CheckBoxList, [169](#)
  - ComboBox, [168](#)
  - ComboBoxEx and ColorBox, [169](#)
  - as datasets, [705–706](#)
  - hashed associative, [132–133](#)
  - ListBox, [167](#)
  - Listview and TreeView, [169–170](#)
  - type-safe, [133–135](#)
  - ValueListEditor, [170–172](#), [170](#)
- ListServ example, [478](#)
- ListSource property, [519](#)
- ListTest example, [387](#)
- Listview component, [169–170](#), [188–193](#), [191](#)
- Listview Item Editor, [189](#), [191](#)
- literals in input masks, [164](#)
- live queries in IBX, [594–597](#), [598](#)
- LIVE\_SERVER\_AT\_DESIGN\_TIME directive, [477](#)
- loadable views, [16–18](#), [17–18](#)
- LoadBalanced property, [667](#)
- LoadDynaPackage method, [423](#)
- LoadFile method, [322](#)
- LoadFromFile method
  - in datasets, [510–511](#), [645](#)
  - in TreeView, [194](#)
  - in TStringList and TStringList, [128](#)
- LoadFromStream method
  - in TBlobField, [139](#)
  - in TStream, [137](#)
- loading packages, [419–420](#)
- LoadLibrary function, [408](#)
- LoadPackage function, [419](#)
- LoadParamsOnConnect property, [572](#)
- local domain names, [739](#)
- local tables for ClientDataSet
  - connections for, [509–510](#), [510](#)
  - defining, [511–512](#), [512](#)
- LocalConnection component, [653](#)
- localhosts in IP addresses, [739](#)
- Localizable property, [368](#)

Locate method, [514](#)

LocateFileService component, [788](#), [798](#)

locating  
files, [798](#)  
records, [514–515](#)

locations  
ADO cursor, [629–630](#)  
in InterBase example, [603–605](#)

Locked property, [481](#)

locking  
control position, [19](#)  
records, [587](#)  
in ADO, [635–636](#), [639–641](#)  
in client/server programming, [557](#)  
recursive, [86](#)

LockType property, [635](#), [638](#)

Log Changes property, [515](#)

LogException method, [75–76](#)

logging  
for ClientDataSet, [515](#)  
errors, [75–76](#), [76](#)

logical identifiers, [559](#)

logical keys, [558](#)

logical operators, [513](#)

logical three-tier architecture, [648](#)

LoginForm adapter, [795](#)

LoginFormAdapter component, [807](#)

LoginPrompt property, [572](#)

logins in WebSnap, [807–808](#)

lookup controls, [519–520](#), [519](#)

lookup dialogs, [611–613](#)

lookup fields, [533–534](#), [534](#)

ItBatchOptimistic value, [635](#), [638](#)

ItOptimistic value, [635](#)

ItPessimistic value, [635](#)

# Index

## M

\$M+ compiler directive, [114](#)

macros

for ModelMaker, [445–446](#)

shortcut keys for, [16](#)

mail, [748–750](#), [749](#)

Mail Merge Editor, [729](#)

MailMergeItems property, [729](#)

Main Form option, [41](#)

main forms in MDI applications, [316–317](#), [316](#)

Main window, looking for, [307–308](#)

MainForm object, [301](#)

MainForm property, [277](#), [297](#)

MainInstance variable, [84](#)

maintained aggregates, [550](#)

maintenance releases of packages, [417](#)

major releases of packages, [417](#)

MakeObjectInstance method, [317](#)

MakeSplash method, [290–292](#)

MakeXmlAttribute function, [884–885](#)

MakeXmlStr function, [884](#)

managed code, [905](#), [908–909](#)

manifest files for themes, [220–221](#)

manifests, [906](#)

ManualDock method, [235](#)

ManualFloat method, [235](#)

many-to-many relations, [557–558](#)

mapping

object-relational, [559](#)

in XML, [854–857](#), [854](#), [856–857](#), [886–887](#), [887](#)

Mapping page, [854](#)

MapTable example, [856–857](#), [857](#)

MapViewOfFile function, [413–414](#)

markers in XML, [834–835](#)

markup languages, [833](#)

marshaling

in Automation, [467](#)

- in COM, [457](#)
- MaskEdit component, [163–164](#)
- MastDet example, [552–553](#), [553](#)
- master/detail relations
  - in ClientDataSet, [552–553](#), [553](#)
  - in DataSnap, [664–665](#), [665](#)
  - in Rave reports, [730–731](#), [731](#)
  - in WebSnap, [803–805](#), [805](#)
- MasterAdapter property, [803](#)
- MasterDataView property, [731](#)
- MasterFields property, [552](#)
- Mastering Delphi Com+ Test example, [496](#)
- mastering.gdb database, [600](#)
- MasterKey property, [731](#)
- Math unit, [88–90](#), [90](#)
- MaxRecords property, [858](#)
- MaxRows property, [762](#)
- MaxSessions property, [805](#)
- MaxValue property, [658](#)
- MBParade example, [289](#)
- MDAC (Microsoft Data Access Components), [616–617](#)
- MdComEvents library, [498](#)
- MdDbGrid component, [683](#), [683](#)
- MdDesPk package, [392](#), [395](#)
- MdEdit1 example, [229–230](#)
- MdEdit2 example, [232–236](#), [232](#), [236](#)
- MdEdit3 example, [247–248](#)
- MDI (Multiple Document Interface) applications, [310](#)
  - child forms and merging menus in, [315–316](#), [315](#)
  - frame and child windows in, [311–315](#), [313](#)
  - main forms in, [316–317](#), [316](#)
  - style property for, [254](#)
  - subclassing client windows in, [317–318](#)
- MDI window actions, [224](#)
- MDIChildCount property, [312](#)
- MDIChildren property, [312](#), [316](#)
- MdiDemo example, [313–315](#), [313](#)
- MdiMulti example, [315–316](#), [315](#)
- MdiMulti Demo4 example, [316–317](#), [316](#)

MdListBoxForm class, [378–379](#)

MdListDialog class, [381](#)

MdPack package, [347–348](#)

MdWArrowImpl1 example, [488](#)

Mediator pattern, [449](#)

members, converting and moving, [447](#)

Memo component, [723](#)

Memo controls, [836](#)

memo fields in Rave, [729](#)

MemoRich component, [164–165](#)

memory

allocation routines for, [84](#)

DLLs in, [411–414](#), [415](#)

garbage collection for, [911–916](#)

for methods, [115](#)

for objects, [58–59](#)

for screen, [265–266](#)

for strings, [58](#)

units for, [95–96](#)

memory leaks, [56](#)

memory management records, [334–335](#)

memory manager, [334–336](#)

memory-mapped files, sharing data with, [413–414](#), [415](#)

Memory Snap tool, [948–949](#)

Menu Designer, [174](#), [174](#)

menu.html file, [793](#)

menus, [173](#)

accelerator keys for, [181–182](#)

in control Bars, [234](#)

images on, [185](#)

items on

least-recently used, [245–247](#), [245](#)

owner-draw, [185–186](#), [186](#)

Menu Designer for, [174](#), [174](#)

merging

in MDI applications, [315–316](#), [315](#)

in OLE, [481](#)

new, [7](#)

pop-up, [174–176](#)

in ToolBar, [216–217](#)

MergeChangesLog method, [515](#)

merging menus

in MDI applications, [315–316](#), [315](#)

in OLE, [481](#)

message boxes, [289](#)

\$MESSAGE directive, [44](#)

message directive, [66](#)

message keyword, [369](#)

message-response methods, [369](#)

message spy program, [33](#)

MessageBox method, [289](#)

MessageDlg function, [289](#)

MessageDlgPos function, [289](#)

messages and message handlers

  compiler, [31–32](#)

  internal, [372–373](#), [373](#)

  late binding for, [66](#)

  mail, [748–750](#), [749](#)

  overriding, [368–372](#)

  queues for, [302](#)

  in sequence diagrams, [433](#)

  for triggering events, [124](#)

  user-defined, [309–310](#)

  in Windows, [303–304](#)

  in windows controls, [373–378](#)

metadata

  in CLI, [904](#)

  in dbExpress, [578–579](#), [579](#)

  in Jet, [628](#)

Metabase component, [724](#)

Method Implementation Code Editor, [441–442](#), [442](#)

method pointers for events, [125–126](#)

MethodAddress method, [115](#)

MethodName method, [115](#)

methods

  abstract, [66–67](#)

  for ActiveX controls, [484](#)

  characteristics of, [46–47](#)

  code completion for, [14–15](#)

  in DataSnap, [663–664](#)

  in Delphi for .NET Preview, [925](#)

  messages for, [376](#)

  names for, [46](#)

  overloaded, [404](#)

  overriding and redefining, [64–65](#), [70](#)

  virtual, [70](#)

    in COM server, [466–467](#)

- vs. dynamic, [65–66](#)
- MGA (Multi-Generational Architecture), [561](#)
- Microsoft Access databases, [624](#)
- Microsoft Data Access Components (MDAC), [616–617](#)
- Microsoft Database Engine (MSDE), [617](#)
- Microsoft Indexing Service, [618](#)
- Microsoft Intermediate Language (MSIL). See [CIL \(Common Intermediate Language\)](#)
- Microsoft.Jet driver, [617](#)
- Microsoft libraries, [934–940](#), [940](#)
- Microsoft Shared Source CLI, [905](#)
- Microsoft Transaction Server (MTS), [494](#)
- Microsoft Word
  - with Automation, [479](#)
  - DLLs for, [411](#)
- MIDAS (Middle-tier Distributed Application Services), [649](#)
- Midas library for ClientDataSet, [510–511](#)
- MidasLib unit, [511](#), [569](#)
- MillisecondOfMonth function, [91](#)
- mind-map diagrams, [435](#)
- Minimize method, [269](#)
- MiniSize program, [82–83](#)
- MinSize property, [180–181](#)
- MinValue property, [658](#)
- mirroring in Rave, [732–733](#)
- MltGrid example, [543](#), [543](#)
- modal forms, [410–411](#)
- modal windows, [280](#)
- ModalResult property, [285](#), [379](#)
- modeless dialog boxes, [280](#), [285–287](#), [285](#)
- ModelMaker, [429–430](#)
  - class diagrams in, [431–432](#), [432–433](#)
  - coding features of, [435](#)
    - Delphi integration with, [436–437](#)
    - Difference view, [442–443](#), [443](#)
    - Event Types view, [443](#)
    - managing, [438–439](#), [439](#)
    - Method Implementation Code Editor, [441–442](#), [442](#)
    - refactoring, [446–447](#)
    - templates for, [450–452](#), [450](#)
    - Unit Code Editor, [439–441](#)



- common diagram elements in, [435–436](#), [436](#)
- design patterns in, [447–450](#)
- documentation in, [444–445](#), [444](#)
- features in, [452](#)
- inheritance in, [437–438](#)
- internal model in, [430](#)
- macros for, [445–446](#)
- non-UML diagrams in, [435](#)
- sequence diagrams in, [433–434](#), [433](#)
- use case diagrams in, [434](#)

#### Modified method

- in TPropertyPage, [490](#)
- in TSoundProperty, [390](#)

Modified property, [313](#)

ModifyAccess property, [808](#)

ModuleIsPackage variable, [424](#)

#### modules

- data modules, [535](#), [648](#)
  - for COM+, [497](#)
  - for WebSnap, [798–799](#)
- System unit support for, [84](#)

Modules object, [793–794](#)

monitor resolution, [299](#)

#### monitoring

- dbExpress connections, [576–577](#), [577](#)
- IBX, [598](#), [599](#)

Mono project, [905–906](#)

#### mouse input

- dragging and drawing, [262–265](#), [265](#)
- for forms, [261–262](#)

mouse messages, [374](#)

mouse-related virtual keys, [262](#)

MouseCoord method, [544](#)

MouseOne example, [262–265](#), [265](#)

MoveBy method, [536](#)

movement in datasets, [536–537](#), [695–698](#), [696](#)

- bookmarks for, [537–538](#)
- editing columns, [539–540](#)
- totals of columns, [537](#), [537](#)

MoveTo method, [195](#), [212](#)

#### moving

- classes, [446](#)
- components, [18](#)
- members, [447](#)

MS-XML processor, [865](#)

MSCorEE.dll file, [908](#)

MSDAORA driver, [617](#)

MSDAOAOSP driver, [617](#)

MSDASQL driver, [617](#)

MSDE (Microsoft Database Engine), [617](#)

MSIL (Microsoft Intermediate Language). See [CIL \(Common Intermediate Language\)](#)

MSOLAP driver, [617](#)

MSXML DOM, [871–872](#)

MSXML library, [850–851](#)

MSXML parser, [644](#)

MSXML SDK, [838](#)

MTS (Microsoft Transaction Server), [494](#)

Multi-Generational Architecture (MGA), [561](#)

multi-read synchronizer, [86](#)

multi-threaded apartment model, [463](#)

multicast events, [926–927](#)

MulticastDelegate class, [910–911](#), [927](#)

Multiline Palette Manager, [947](#)

multipage Web applications, [818–821](#), [819](#)

Multiple Document Interface (MDI) applications, [310](#)

- child forms and merging menus in, [315–316](#), [315](#)
- frame and child windows in, [311–315](#), [313](#)
- main forms in, [316–317](#), [316](#)
- style property for, [254](#)
- subclassing client windows in, [317–318](#)

multiple frames without pages, [328–330](#), [329](#)

multiple instancing, [463](#)

multiple-page forms, [202](#)

- image viewer in, [207–210](#), [207](#)
- PageControl and TabSheet for, [202–207](#), [204–205](#)
- for user interface wizard, [210–213](#), [211](#)

multiple pages, WebSnap, [789–791](#), [790](#)

multiple selections

- DBGrid for, [542–543](#), [543](#)
- in ListBox, [167](#)

MultipleTransactionsSupported property, [590](#)

multipurpose WebModule, [774–776](#)

MultiSect property, [167](#)

multitasking, [302](#), [304](#)

multithreading, [304–307](#)

class for, [86](#)

with socket connections, [738](#)

multitier databases. See [DataSnap](#)

Multitier page, [653](#)

MultiWin example, [281–282](#)

mutexes (mutual exclusion objects), [308](#)

MyBase file, [508](#). See also [ClientDataSet component](#)

MyBase1 example, [509–510](#), [510](#)

MyBase2 example, [511–513](#), [512](#)

MySQL drivers, [567](#)

◀ PREVIOUS

NEXT ▶

# Index

## N

name directive, [402](#)

name mangling, [401–402](#)

name-value pairs for lists, [129](#)

names and Name property

accessing properties by, [115–117](#), [117](#)

for action objects, [222](#)

in C++, [401–402](#)

for components, [339](#)

components without, [122](#)

for domains, [739](#)

for fields, [360](#)

for fonts, [21](#)

for functions, [360](#)

for methods, [46](#), [115](#)

in ModelMaker code, [446](#)

for projects and library, [407–408](#)

for properties, [360](#)

in TComponent, [121](#)

in TField, [527](#)

for units, [44](#)

namespaces

in .NET architecture, [906](#)

for units, [922–924](#)

namespaces clause, [923](#)

NameValues example, [170–171](#), [170](#)

NameValueSeparator property, [129](#)

native formats

in ClientDataSet, [511](#)

date time, [709](#)

native look-and-feel in library selection, [155](#)

navigating datasets, [536–537](#), [695–698](#), [696](#)

bookmarks for, [537–538](#)

editing columns, [539–540](#)

totals of columns, [537](#), [537](#)

NegInfinity constant, [88](#)

nested markers, [834](#)

nested transactions, [634–635](#)

nested try blocks, [72](#)

nested types, [926](#)

.NET architecture, [500–502](#), [502](#), [899](#)

assemblies in, [906–907](#)

CIL, [907–911](#)

CLI in, [903–904](#)

CLR in, [905–906](#)

Delphi for .NET Preview. See [Delphi for .NET Preview](#)  
deployment and versioning in, [916–917](#), [917](#)

garbage collection in, [911–916](#)

.NET Framework, [900](#)

NetClassInfo example, [932–933](#), [933](#)

NetCLX, [110](#), [112](#)

NetEuroConv example, [933](#)

NetImport example, [500–502](#), [502](#)

NetLibrary library, [500–502](#)

NetLibSpeed example, [933](#)

NetNumberClass unit, [501](#)

Netscape Server API (NSAPI), [769–770](#)

Neutral threading model, [463](#)

New Component dialog box, [341](#), [341](#)

New Field dialog box, [531](#), [531](#), [534](#)

New Form option, [41](#)

New Items dialog box, [39–40](#), [40](#)

for ActiveX controls, [489](#)

for COM+, [494](#)

for DataSnap, [653](#)

for form inheritance, [319–320](#), [319](#)

for IntraWeb, [810](#)

for Web services, [877](#), [889](#)

for WebBroker, [770](#)

for WebSnap, [787](#)

for XML Data Binding Wizard, [847](#)

New Project option, [41](#)

New Transactional Object dialog box, [495](#), [495](#)

New Web Server Application dialog box, [770](#), [773](#), [781](#)

New WebSnap Application dialog box, [787](#), [787](#)

New WebSnap Page Module dialog box, [789](#), [790](#)

NewButtonClick method, [301](#)

NewDate example, [59–60](#), [60](#)

NewGuid example, [459–460](#), [459](#)

NewInstance class, [335](#)

NewItem function, [174](#)

NewLine function, [174](#)

NewMenu function, [174](#)

NewPopupMenu function, [174](#)

NewSubMenu function, [174](#)

NewValue property, [641](#)

NewWinProcedure method, [318](#)

nil value

- for forms, [282](#)
- with Free, [55](#), [58–59](#)

NoAction object, [227](#)

NodeIndentStr property, [842](#)

nodes

- in TreeView, [197–198](#), [199](#)
- in XSLT, [864–865](#)

NodeType property, [840](#)

NonAware example, [545–547](#), [545](#)

nonblocking socket connections, [738](#)

nonvisual components, [111](#), [378–381](#), [381](#)

nonwindowed controls, [158](#)

normalization rules, [558](#)

Notebook component, [202](#)

Notification method

- for data-aware controls, [673](#)
- in TComponent, [301–302](#)
- in TMdPersonalData, [354](#)
- in TMdViewer, [355–356](#)

notifications

- messages for, [374](#)
- in windows controls, [376–377](#)

NotifyDataLinks method, [670](#)

NoTitle example, [258–259](#), [258](#)

NSAPI (Netscape Server API), [769–770](#)

null values with field events, [535–536](#), [536](#)

NullAsStringValue variable, [94](#)

NullDates example, [535–536](#), [536](#)

NullEqualityRule variable, [94](#)

NullMagnitudeRule variable, [94](#)

NullStrictConvert variable, [94](#)

Numeric edit box

- creating, [368–370](#)
- with thousands separators, [370–371](#)

# Index

## O

OBJ files, [36](#)

ObjAddRef method, [458](#)

ObjAuto unit, [148](#)

ObjClass property, [710](#)

ObjDataSetDemo example, [713](#), [713](#)

Object Debugger, [948](#)

object identifiers (OIDs)  
 in database design, [558–560](#)  
 in Interbase, [600–602](#), [601](#)

Object Inspector, [19–21](#)  
 for components, [351–352](#), [352](#)  
 drop-down fonts in, [21](#)  
 Object TreeView in, [22–23](#)  
 property categories in, [21–22](#)

Object Inspector Font Wizard, [945](#)

object linking, [480](#)

Object Linking and Embedding (OLE), [456–457](#)

object-oriented programming (OOP), [43](#), [64](#)  
 and modeling. See [ModelMaker](#)  
 for reusable components, [483](#)

object reference model, [55–56](#), [55](#)  
 assigning objects in, [56–58](#), [56](#)  
 memory for objects in, [58–59](#)

object-relational mapping, [559](#)

Object Repository, [39–41](#), [40](#)

Object TreeView, [22–23](#)

ObjectBinaryToText method, [140](#)

ObjectPropertiesDialog method, [482](#)

objects  
 assigning, [56–58](#), [56](#)  
 in Automation, [476–477](#), [482–483](#), [483](#)  
 code completion for, [14–15](#)  
 for COM server, [461–463](#), [462](#)  
 in CTS, [909–910](#)  
 datasets of, [710–713](#), [713](#)  
 in DataSnap, [667](#)  
 in Delphi, [44–48](#)  
 field, [527–528](#), [528](#)  
 memory for, [58–59](#)  
 scope of, [476–477](#)

ObjectTextToBinary method, [140](#)

ObjQueryInterface method, [458](#)

ObjRelease method, [458](#)

OBJREPOS subdirectory, [41](#)

ObjsLeft example, [335–336](#)

Observer pattern, [449](#)

octets in IP addresses, [739](#)

OCX (OLE Controls), [456](#), [483](#)

OCX files, [36](#)

ODBC (Open Database Connectivity), [615](#), [644](#)

ODList example, [187–188](#)

ODMenu example, [185–186](#), [186](#)

Office programs with Automation, [478–479](#)

oh.exe tool, [33](#)

OIDs (object identifiers)  
    in database design, [558–560](#)  
    in Interbase, [600–602](#), [601](#)

OLAP (Online Analytical Processing), [617](#)

OldCreateOrder property, [153](#), [279](#)

OldValue property, [641–642](#)

OLE (Object Linking and Embedding), [456–457](#)

OLE Automation. See [Automation](#)

OLE container messages, [376](#)

OLE Controls (OCX), [456](#), [483](#)

OLE DB Provider, [617–618](#)

OLE DB Provider Development Toolkit, [618](#)

OLE DB Provider For Indexing Service, [617](#)

OLE DB Provider For Internet Publishing, [617](#)

OLE DB Provider For OLAP, [617](#)

OLE DB providers, [616–617](#)

OLE Insert Object dialog box, [480–481](#)

OleCont example, [482](#), [482](#)

OleContainer component, [480–482](#), [482](#)

OnAction event, [771–772](#)

OnActionExecute event, [228](#), [296](#)

OnActionUpdate event, [296](#)

OnActivate event, [278](#), [296](#), [298](#)



OnActiveFormChange event, [299–300](#)

OnAdvancedDrawItem event, [186](#)

OnArrowDbClick event, [364–365](#)

OnBeforeDispatchPage event, [790](#)

OnBeforeExecuteAction event, [797](#)

OnBeforePageDispatch event, [794](#)

OnCalcFields event, [531](#), [585](#)

OnCanResize event, [270](#)

OnCanViewPage event, [808](#)

OnChange event

in TabControl, [208](#)

in TDate, [126](#)

in TMdArrow, [363](#)

OnClick event, [125](#), [261](#)

OnClose event, [278–279](#), [300](#)

OnCloseQuery event, [278–279](#)

OnCloseUp event, [168](#)

OnColumnClick event, [191](#)

OnCommand event, [742](#)

OnCommandGet event, [757](#), [765](#)

OnConstrainedResize event, [270](#)

OnContextMenu event, [175](#)

OnContextPopup event, [176](#)

OnCreate event, [277–279](#), [299](#)

OnCreateNodesClass event, [197](#)

OnDataChange event, [546](#)

OnDeactivate event, [296](#), [298](#)

OnDeleteError event, [553](#)

OnDestroy event, [279](#), [301](#)

OnDockDrop event, [236](#)

OnDockOver event, [235–236](#)

OnDoubleClick event, [285](#)

OnDragDrop event, [98](#)

for docking, [234](#)

for TreeView, [195](#)

OnDragOver event, [98](#)

for docking, [234](#)

for TreeView, [195](#)

OnDrawColumnCell event, [540–541](#)

OnDrawItem event, [185](#)

OnDrawTab event, [209](#)

OnDropDown event, [168](#)

one-to-many relations, [557](#)

one-to-one relations, [557](#)

OneCopy example, [308–310](#)

OnEditButtonClick event, [533](#)

OnEditError event, [553](#)

OnEndDock event, [234](#), [236](#)

OnEnter event, [177–178](#)

OnException event, [75](#), [296](#), [553](#)

OnExecute event

- in ActionFont, [313](#)
- in ActionLink, [223–224](#)
- for actions, [228](#), [228](#)

OnExecuteAction, [223](#)

OnExit event, [177–178](#)

OnFind event, [288](#)

OnFindIncludeFile event, [798](#)

OnFindStream event, [798](#)

OnFindTemplateFile event, [798](#)

OnFormatCell event, [763](#)

OnGetData event, [668](#)

OnGetDataSetProperties event, [667](#)

OnGetItem event, [249](#)

OnGetItemCount event, [249](#)

OnGetResponse event, [864](#)

OnGetText event, [535](#), [541](#), [549](#)

OnGetValue event, [794](#), [797](#)

OnHelp event, [257](#), [296](#)

OnHint event, [217](#), [296](#)

OnHTMLTag event, [760](#)

OnIdle event, [215](#), [296](#), [304](#)

OnItemSelected event, [250](#)

OnKeyPress event, [163](#), [259](#)

Online Analytical Processing (OLAP), [617](#)

OnLogTrace event, [576](#)

OnMeasureItem event, [185](#), [187](#)

OnMessage event, [296](#)

OnMinimize event, [296](#)

OnMouseDown event, [261](#)

OnMouseMove event, [261–262](#)

OnMouseUp event, [261](#)

OnMouseWheel event, [261](#)

OnMouseWheelDown event, [261](#)

OnMouseWheelUp event, [261](#)

OnMove event, [181](#)

OnNewRecord event, [547](#)

OnPageAccessDenied event, [808](#)

OnPaint event, [265–266](#), [274](#)

OnPostError event, [553](#)

OnReceivingData event, [893](#)

OnReconcileError event, [587](#), [651](#), [659–660](#)

OnRecordChangeComplete event, [641](#)

OnRecordSetCreate event, [622](#)

OnReplace event, [288](#)

OnResize event, [269–271](#)

OnRestore event, [296](#)

OnScroll event, [271](#)

OnSetText event, [535](#), [541](#)

OnShortCut event, [296](#)

OnShow event, [278](#)

OnShowHint event, [183](#), [296](#)

OnStartDock event, [234](#)

OnStateChange event, [525](#), [546](#)

OnStatusTextChange event, [486](#)

OnTag event, [760](#)  
    in PageProducer, [779](#)  
    in WebSnap, [791](#)

OnTimer event, [492](#)

OnTitleChange event, [486](#)

OnTitleClick event, [830](#)

OnTrace event, [576](#)

OnUpdate event  
    in ActionCount, [227](#)  
    in ActionFont, [313](#)  
    and actions, [224](#), [229](#)

- in MDI applications, [314](#)
- OnUpdateData event, [660](#)
- OnUpdateError event, [553](#), [641](#), [660](#)
- OnValidate event, [535](#)
- OnWillChangeRecord event, [641](#)
- OOP (object-oriented programming), [43](#), [64](#)
  - and modeling. See [ModelMaker](#)
  - for reusable components, [483](#)
- OOP Form Wizard, [945](#)
- opaque strings, [538](#)
- Open Database Connectivity (ODBC), [615](#), [644](#)
- open-source projects, [951–952](#)
- Open Tools API, [452](#)
- Open XML project, [951](#)
- OpenDialog component, [287](#)
- OpenHelp tool, [33](#)
- opening datasets, [690–694](#)
- opening markers in XML, [834–835](#)
- openItemClick method, [937](#), [939–940](#)
- OpenPictureDialog component, [288](#)
- OpenSchema method, [622–623](#), [623](#)
- OpenTools API, [388](#)
- OpenXML DOM, [838](#)
- Operator property, [734](#)
- optimistic locking, [587](#)
  - in ADO, [639–641](#)
  - in client/server programming, [557](#)
- options, components for, [165–166](#)
- Options property
  - in DataSetProvider, [658](#)
  - in DBGrid, [516](#)
- Oracle drivers, [567](#)
- orthogonal lines in ModelMaker, [452](#)
- out-of-process servers, [457](#)
- outer transactions, [634–635](#)
- OutLabel property, [353](#)
- Output Options dialog box, [720](#), [720](#)
- overload directive, [404](#)
- overload keyword, [46](#), [65](#)

overloaded constructors, [54–55](#)

overloaded methods, [46](#), [404](#)

override keyword, [64](#)

overriding

Finalize, [911–912](#)

message handlers, [368–372](#)

methods

in late binding, [64–65](#)

virtual, [70](#)

owner components, [58](#)

owner-draw controls, [184–185](#)

list box of colors, [186–188](#)

menu items, [185–186](#), [186](#)

messages for, [377](#)

Owner property, [119–120](#)

OwnerDraw property, [185](#), [209](#)

ownership in TComponent, [117–120](#), [120](#)

OwnHandle property, [162](#)

OwnsObjects property, [131](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

# Index

## P

- P/Invoke (Platform Invocation Service), [904](#)
- pa attributes, [388](#)
- ~PA files, [37](#)
- Package Collection Editor tool, [33](#)
- Package Editor, [345](#), [345](#), [347–348](#), [349](#), [417](#), [417](#)
- PackageInfo pointer type, [84](#)
- packages, [415–416](#)
  - for components, [337–339](#)
  - for Fonts combo box, [344–347](#), [345–346](#)
  - forms in, [417–419](#), [417](#)
  - interfaces in, [420–424](#)
  - loading at run time, [419–420](#)
  - and program size, [30–31](#)
  - structure of, [424–427](#), [425](#)
  - System unit support for, [84](#)
  - versioning in, [416–417](#)
- Packages page, [346](#)
- PackAuto package, [477](#)
- PacketRecords property, [575](#), [584](#)
- packets
  - with ClientDataSet, [584](#)
  - in DataSnap, [651–652](#), [667–668](#)
- PackInfo example, [425–427](#), [425](#)
- PackWithForm package, [418](#), [420](#)
- Page mode in IntraWeb, [815](#)
- page numbers in Rave, [724](#)
- Page object, [793](#)
- PageControl component, [119](#), [202–207](#), [204–205](#)
  - actions for, [224](#)
  - docking to, [239–241](#), [241](#)
  - in XML, [836](#)
- PageControl property, [206](#)
- PagedAdapter component, [794](#)
- PageDesigner page, [717](#)
- PageDispatcher component, [787](#), [794](#), [808](#)
- PageHead component, [774](#)
- PageNumInit component in Rave, [724](#)
- PageProducer component, [759–761](#), [774](#), [779](#), [788](#)

pages

frames with, [326–328](#), [327](#)

frames without, [328–330](#), [329](#)

Web, ActiveX controls in, [492–493](#), [493](#)

Pages example, [202–207](#), [204–205](#)

Pages list, [793](#)

PageScroller component, [172–173](#)

PageSize property, [799](#)

PageTail component, [774–775](#)

Paint method, [358](#), [361–363](#), [362](#)

painting

DBGrids, [540–542](#), [542](#)

in windows, [265–267](#)

Panel component, [119](#)

Panels property, [382](#)

Paradox, Jet engine for, [624–625](#), [625](#)

ParamByName method, [581](#)

ParamCount global variable, [84](#)

parameters

code, [15](#)

for DLLs, [399](#)

for mouse events, [262](#)

in overloaded methods, [46](#)

in stored procedures, [565](#)

parametric queries

in DataSnap, [663](#), [663](#)

in dbExpress, [579–581](#), [580–581](#)

Params property

in queries, [663](#)

in SQLConnection, [572](#)

in SQLDataSet, [580–581](#)

in XMLBroker, [858](#)

ParamStr global variable, [84](#)

Parent property, [158–159](#)

ParentColor property, [160](#)

ParentFont property, [160](#)

ParentXxx properties, [374](#)

ParQuery example, [580–581](#), [581](#)

parsers, MSXML, [644](#)

parseURL method, [851](#)

PAS files, [36](#)

Pascal language, [38](#), [44](#)

passing XML documents, [884–886](#)

PasswordChar property, [163](#)

Paste Link operation, [480](#)

PasteSpecialClick event, [481](#)

PasteSpecialDialog method, [481](#)

pasting components, [24–25](#)

patch files, [415](#)

path delimiter functions, [87](#)

PathInfo property, [863](#)

patterns in ModelMaker, [447–450](#)

PCE.exe tool, [33](#)

PChar type, [405–406](#)

PE (Portable Executable) format, [904](#), [908](#)

Peek method, [131](#)

people in InterBase example, [603–605](#)

performance in InterBase, [561](#)

Performance Monitor for connections, [644](#)

permissions in WebSnap, [808](#)

Persisted Recordset provider, [618](#)

persistence

- in ADO recordsets, [644–645](#)
- in Arrow component properties, [362–364](#), [364](#)
- class for, [113](#)
- in streaming, [140–142](#), [141](#)

Personal edition, [4](#)

pessimistic locking in ADO, [635–636](#)

PEVerify utility, [909](#)

physical keys, [558–559](#)

PickList property, [516](#)

PixelsPerInch property, [275–276](#)

plain component packages, [338](#)

plain forms, [252–253](#), [253](#)

Platform Invocation Service (P/Invoke), [904](#)

platform-specific symbols, [156](#)

platforms for databases, [567–569](#)

playing compound objects, [480](#)

PlaySound function, [64](#), [372](#), [391](#)

PMdReclInfo pointer type, [696](#)



po options, [666–667](#)

pointers

for bookmarks, [696](#)

in Delphi for .NET Preview, [920](#)

in object reference model, [55](#), [55](#)

for procedures, [409–410](#)

PoliForm example, [322–324](#), [322](#)

PolyAnimals example, [64](#), [64](#)

polymorphic forms, [321–324](#), [322](#)

polymorphism, [63–66](#), [64](#)

abstract methods in, [66–67](#)

in interfaces, [70](#)

pooling

connection, [643–644](#)

object, [667](#)

Pop method, [131](#)

pop-up menus, [173–176](#)

POP3 (Post Office Protocol, version 3), [740](#)

PopulateTypes method, [937–938](#)

Popup method, [175](#)

PopupMenuActionBarEx component, [242](#)

PopupMenu property, [174–175](#)

portability

of database engines, [506](#)

in library selection, [155](#)

of text DFM files, [19](#)

Portable Executable (PE) format, [904](#), [908](#)

PortableDragTree example, [196–197](#)

porting programs, [247–248](#)

ports in socket programming, [739–740](#)

Pos function, [93](#)

PosEx function, [93](#)

position and Position property

of controls, [159](#), [163](#)

of Form Designer components, [18](#)

of forms, [255](#), [268–269](#)

of mouse, [262](#)

of scroll bars, [271](#)

in TStream, [136](#)

Post command, [520](#)

POST method, [778](#)

Post Office Protocol, version 3 (POP3), [740](#)

PostMessage method, [66](#), [303–304](#), [310](#)

predefined items

actions, [224–225](#)

dialog boxes, [287–289](#), [288](#)

preemptive multitasking, [302](#)

Preferences page

for AutoSave feature, [11](#)

for compiler, [30](#)

for docking, [6](#)

prefixes

for functions, [360](#)

for names, [121](#)

pressed-down state, [173](#)

Preview compiler, [900](#)

Preview pages, HTML, [788](#)

previewing in Rave, [719](#), [719](#)

primary keys, [558–560](#)

printing in dbExpress, [581–584](#)

PrintOutDataSet method, [582–583](#)

PrioritySchedule property, [246](#)

private specifier, [48–50](#), [925](#)

ProcessMessages function, [303–304](#)

Producer object, [793](#)

Producer property, [772](#), [776](#), [863](#)

ProducerContent property, [772](#)

Professional Studio edition, [4](#)

program flow in exceptions, [72–73](#)

program statement, [922](#)

programmability, client/server programming for, [557](#)

ProgressBar component, [172](#), [671–673](#), [673](#)

Project JEDI, [952](#)

Project Manager, [26–28](#), [27](#), [723](#)

Project Options dialog box, [28](#)

for applications, [296](#)

for Automation, [477](#)

for forms, [277](#), [278](#), [280](#)

for messages, [31](#), [31](#)

for ModelMaker documentation, [445](#)

for packages, [346](#), [346](#)

Project toolbar in Rave, [718](#)

Project Tree panel, [717](#)

project-wide to-do items, [9](#)

ProjectFile property, [719](#)

projects, [26–28](#), [27](#)

  compiling and building, [30–32](#)

  exploring, [32](#), [32](#)

  names for, [407–408](#)

  options for, [28–30](#)

prologues in XML, [835](#)

PropCom example, [466](#)

properties

  accessing by name, [115–117](#), [117](#)

  for ActiveX controls, [489–491](#), [490](#)

  categories

    in Object Inspector, [21–22](#)

    registering, [366–367](#), [367](#)

  code completion for, [14–15](#)

  collections for, [381–385](#), [383](#)

  in CTS, [909–910](#)

  in Delphi for .NET Preview, [925–926](#)

  editors

    installing, [391–392](#)

    for sound, [388–391](#), [390–391](#)

    writing, [388](#)

  encapsulation with, [50–52](#), [51](#)

  events as, [126–128](#)

  for forms, [53–54](#), [53](#)

  names for, [360](#)

  in Object Inspector, [20–21](#)

Properties dialog box, [20](#)

Property Editor dialog box, [438–439](#), [439](#)

property keyword, [53](#)

Property panel, [717](#)

protected specifier, [48–50](#), [60–62](#), [925](#)

protection, client/server programming for, [557](#)

Protection example, [61–62](#)

protocols

  for DataSnap, [650–651](#)

  in socket programming, [740](#)

provider options in DataSnap, [666–667](#)

ProviderFlags property, [658](#)

ProviderName property, [655](#)

proxy forms, [114](#)

PtInRegion function, [365](#)

public deployment, [916](#)

public specifier, [48–50](#), [52](#)

published keyword, [52](#), [114–115](#)

Publisher example, [499–500](#)

publishing

subcomponents, [350–352](#), [352](#)

WSDL, [881–883](#), [881](#)

pull-down menus, [173](#)

purely abstract classes, [69](#)

Push method, [131](#)

PWideChar type, [460](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

# Index

## Q

QBlcons example, [257](#)

QDynaForm example, [252](#)

QLibComp example, [153–154](#), [156](#), [156](#)

QLibSpeed example, [155](#)

QMbParade example, [289](#)

QMouseOne example, [262–263](#)

QMultiWin example, [281–282](#)

QRefList example, [192–193](#)

QRefList2 example, [283](#)

QRegion class, [366](#)

QScale example, [275–276](#)

QStdCtrls unit, [152](#)

Qt library, [150](#)

queries

live, [594–597](#), [598](#)

parametric, [663](#), [663](#)

WebBroker technology for, [777–781](#), [780](#)

Query component, [507](#)

query forms, [613](#), [613](#)

QueryFields property, [777–778](#)

QueryInterface method

for as casts, [465](#)

in IUnknown, [457–458](#)

QueryTableProducer component, [759](#), [777](#), [779](#)

queues, [132](#), [302](#)

# Index

## R

radio items on menus, [173](#)

RadioButton component, [165–166](#)

RadioGroup component, [166](#)

RadioButton property, [173](#), [230](#)

raise keyword, [71](#)

raising exceptions, [71](#)

RandomFrom function, [89](#)

RandomRange function, [89](#)

Range property, [270](#), [272–273](#)

ranges, components for, [172–173](#)

Rave, [715–716](#)

calculations in, [733–734](#)

components of, [722–723](#)

basic, [723–725](#)

data access, [725–726](#), [725](#)

data-aware, [728–730](#), [729](#)

regions and bands, [726–728](#), [727](#)

data connections in, [721–722](#), [722](#)

design environment in, [716–719](#), [716](#)

master/detail reports in, [730–731](#), [731](#)

mirroring in, [732–733](#)

rendering formats for, [720–721](#), [720](#)

RVProject component in, [719–720](#), [719](#)

scripts for, [731–732](#), [732](#)

RaveDatabase component, [726](#)

RaveDetails example, [730–732](#), [731–732](#)

RaveDirectDataView component, [726](#)

RaveDriverDataView component, [726](#)

RaveLookupSecurity component, [726](#)

RaveProject component, [723](#)

RaveSimpleSecurity component, [726](#)

RaveSingle example, [721–722](#), [722](#)

RC files, [37](#)

RDSConnection component, [508](#), [619](#)

re-raising exceptions, [71](#)

read clauses, [50](#), [52](#), [465](#)

Read method, [143](#)

read-only ProgressBars, [671–673](#), [673](#)

read-only properties  
in Object Inspector, [20](#)  
setting, [52](#)

read-write TrackBar, [674–677](#), [677](#)

ReadBool method, [332](#)

ReadBuffer method, [137](#)

ReadComponent method, [136](#)

ReadComponentRes method, [140](#)

ReadInteger method, [332](#)

ReadString method, [332](#)

Real48 type, [920](#)

ReallocMem function, [921](#)

rebuild.bat file, [900](#)

Rebuild Wizard, [946](#)

RecNo property, [537](#)

Reconc unit, [588](#)

Reconcile Error Dialog unit, [588](#), [588](#)

ReconcileProducer property, [860](#)

ReconcileProvider property, [863](#)

record buffers, [698–703](#)

record viewer component, [678–683](#), [682](#)

RecordChanged method, [679](#)

RecordCount property, [537](#), [560](#), [631–632](#)

recording macros, shortcut keys for, [16](#)

records

in Delphi for .NET Preview, [920–921](#)

editing, [520](#)

locating, [514–515](#)

locking, [587](#)

in ADO, [635–636](#), [639–641](#)

in client/server programming, [557](#)

recursive, [86](#)

status of, [585](#), [585](#)

recordsets in ADO

clone, [633](#)

disconnected, [642–643](#)

persistent, [644–645](#)

recursive locks, [86](#)

redefining methods, [64–65](#)

refactoring ModelMaker code, [446–447](#)

reference-counting techniques, [476](#)

references, class, [76–78](#), [78](#)

referential integrity, [559](#)

ReflectionForm class, [935](#)

Reflector tool, [901](#)

RefList example, [189–193](#), [191](#)

RefList2 example, [283–285](#), [284](#)

Refresh in Model option, [438](#)

Refresh method, [266](#), [660](#)

refreshing

data, [660–662](#), [661](#)

screen, [266–267](#), [660](#)

RefreshRecords method, [660–661](#)

regasm (Framework Assembly Registration Utility), [501](#)

region components, [726–728](#), [727](#)

Register ActiveX Server command, [488](#)

Register method

for component editors, [395](#)

for components, [342](#), [357](#)

for packages, [345](#)

for property categories, [366–367](#)

for property editors, [391](#)

RegisterActions method, [387](#)

RegisterClass method, [330](#)

RegisterColorSelect method, [421](#)

registering

ActiveX libraries, [34](#), [488](#)

Automation servers, [473–474](#)

component editors, [395](#)

components, [342](#)

conversion rates, [99–101](#)

property categories, [366–367](#), [367](#)

RegisterPooled method, [667](#)

RegisterPropertyInCategory function, [366–367](#)

Registry

for automation servers, [473–474](#)

cleaning up, [33](#)

for COM servers, [464](#)

for connection pooling, [644](#)

for GUIDs, [458–459](#)

for hot-track activation, [24](#)

and INI files, [331–332](#)

for shared component templates, [26](#)



- for sound properties, [389–390](#)
- status information in, [39](#)
- RegSvr32.exe program, [464](#)
- relational databases, [560](#)
- relations
  - in database design, [557–558](#)
  - master/detail. See [master/detail relations](#)
- relationships in diagrams, [435](#)
- Release method, [292](#), [820](#)
- \_Release method, [71](#), [457](#)
- ReleaseCapture function, [265](#)
- Remote Data Module Wizard, [654](#), [654](#)
- RemoteServer component, [649](#)
- RemoteServer property, [655](#)
- Remove method
  - in Delphi for .NET Preview, [926–927](#)
  - for lists, [128](#)
- RemoveComponent method, [119](#)
- removing
  - captions, [258](#)
  - form fields, [121–122](#)
- renaming in ModelMaker code, [446](#)
- rendering formats for Rave, [720–721](#), [720](#)
- Repaint method, [266–267](#)
- RepaintRequest method, [363](#)
- reparenting classes, [446](#)
- Repeatable Read transaction isolation mode, [561](#), [590](#)
- replicable data-aware controls, [674](#)
- ReplyNormal property, [741](#)
- Report Library node, [717](#)
- reporting
  - in dbExpress, [581–584](#)
  - dynamic database, [776–777](#)
  - in Rave. See [Rave](#)
- request methods in WebBroker, [778](#)
- Request object, [793](#)
- Request property, [771](#)
- Requires a New Transaction option, [495](#)
- Requires a Transaction option, [494](#)
- requires keyword, [346](#)

- requires lists, [345–346](#)
- RES files, [37](#)
- reserved characters in XML, [834](#)
- resize cursors, [255](#)
- ResizeStyle property, [181](#)
- resolution of monitors, [299](#)
- resolvers, [660](#)
- ResolveToDataSet property, [658](#), [660](#)
- Resource Explorer tool, [34](#)
- Resource Workshop tool, [34](#)
- resources and resource files
  - binding into executables, [30](#)
  - in COM+, [494](#)
  - DLLs for, [399](#)
  - Windows, [27–28](#)
- resourcestring keyword, [85](#)
- Response object, [793](#)
- Response property, [771](#)
- Restore method, [269](#)
- Result pages, HTML, [788](#)
- Resume method, [752](#)
- Resync method, [642](#)
- RethinkHotkeys method, [182](#)
- RethinkLines method, [182](#)
- return values in stored procedures, [565](#)
- reusability
  - design patterns for, [448–449](#)
  - object-oriented programming for, [64](#), [483](#)
- Revert to Inherited command, [319](#)
- RGB color, [160–161](#)
- RGB function, [160–161](#)
- RichBar example, [214–215](#), [214](#), [217–219](#), [218](#)
- RichEdit actions, [224](#)
- RichEdit component, [164–165](#)
- RichEditSelectionChange method, [214–215](#), [229](#)
- right mouse button, [261](#)
- robustness diagrams, [435](#)
- Roeder, Lutz, [901](#)
- role-based security, [494](#)

- Rollback action, [596](#)
- Rollback method, [589](#)
- RollbackRetaining command, [597](#)
- RollbackTrans method, [634](#)
- rolling back transactions, [589](#), [634–635](#)
- Rotor project, [905–906](#)
- Round function, [89–90](#)
- rounding
  - in currency conversions, [101–102](#)
  - numbers, [89–90](#), [90](#)
- Rounding example, [90](#), [90](#)
- RoundTo function, [89–90](#)
- Row property, [167](#)
- RowAttributes property, [763](#)
- RowCurrentColor property, [826–827](#)
- RowHeights property, [683](#)
- RowHeightsChanged method, [683](#)
- RowLayout property, [167](#)
- RowLimit property, [826](#)
- RPS files, [37](#)
- RTL (run-time library) package, [81](#), [83](#), [110–112](#), [110](#)
  - converting data, [96–102](#), [98](#), [101](#)
  - file management, [102–104](#), [103–104](#)
  - TObject, [104–108](#), [106](#), [108](#)
  - units of. See [units](#)
- RTTI (run-time type information)
  - for casting, [67–69](#)
  - for SOAP, [878](#)
- Run method, [296](#)
- run-only component packages, [338](#)
- Run Parameters dialog box, [42](#), [773](#)
- run time
  - calling DLLs at, [408–410](#)
  - loading packages at, [419–420](#)
- run-time library for Delphi for .NET Preview, [930–931](#)
- run-time library (RTL) package, [81](#), [83](#), [110–112](#), [110](#)
  - converting data, [96–102](#), [98](#), [101](#)
  - file management, [102–104](#), [103–104](#)
  - TObject, [104–108](#), [106](#), [108](#)
  - units of. See [units](#)
- run-time packages, [30–31](#), [338](#)

run-time only properties, [52](#)

run-time type information (RTTI)

for casting, [67–69](#)

for SOAP, [878](#)

RunningTotal property, [730](#)

RunProp example, [116–117](#), [117](#)

RvCustomConnection component, [721](#)

RvDataSetConnection component, [721](#)

RvNDR Writer component, [720](#)

RVProject component, [719–720](#), [719](#)

RvQueryConnection component, [721](#)

RvRenderHTML component, [721](#)

RvRenderPDF component, [720](#)

RvRenderPreview component, [720](#)

RvRenderPrinter component, [720](#)

RvRenderRTF component, [721](#)

RvRenderText component, [721](#)

RvSystem Writer component, [720](#)

RvTableConnection component, [721](#)

RWBlocks example, [601](#), [605–613](#), [607](#), [610](#)

# Index

## S

- safe code, [908–909](#)
- safecall calling convention, [471](#), [664](#)
- SafeLoadLibrary function, [408](#)
- SameValue function, [89](#)
- sample1embedded.xml file, [865–866](#)
- SampProv driver, [617](#)
- Save Code Template dialog box, [451](#)
- Save method in MDI applications, [314](#)
- SaveDialog component, [288](#)
- SavePictureDialog component, [288](#)
- SavePoint property, [515](#)
- SaveStatusForm unit, [331](#), [333](#)
- SaveToFile method, [128](#), [510–511](#), [645](#)
- SaveToStream method
  - in TBlobField, [139](#)
  - in TStream, [137](#)
- saving
  - application status information, [331–332](#)
  - compiler options, [28–29](#)
  - desktop settings, [5–6](#)
  - DFM files, [19](#)
  - docking status, [238](#)
- SAX (Simple API for XML), [850–853](#), [853](#), [872–873](#)
- SaxDemo1 example, [850–853](#), [853](#), [872](#)
- scalability and cursor location, [630](#)
- Scale example, [275–276](#)
- ScaleBy method, [275–276](#)
- Scaled property, [275–276](#)
- scaling forms, [274–277](#)
- schema information
  - in ADO, [622–623](#), [623](#)
  - in Jet, [628](#)
  - in XML, [849–850](#)
- SCHEMA.INI file, [627](#)
- SchemaObject parameter, [578](#)
- SchemaPattern parameter, [578](#)
- SchemaTest example, [579](#), [579](#)

- SchemaType parameter, [578](#)
- ScktSrvr.exe application, [650](#)
- scope in Automation, [476–477](#)
- screen
  - refreshing, [266–267](#), [660](#)
  - size of, [299](#)
- Screen example, [299–302](#), [300](#)
- Screen object, [299–302](#), [300](#)
- ScreenSnap property, [269](#)
- scripts
  - for adapters, [797–798](#)
  - for Rave, [731–732](#), [732](#)
  - server-side, [791–793](#), [792](#)
- Scroll1 example, [271–272](#), [272](#)
- Scroll2 example, [273–274](#), [273](#)
- ScrollBar component, [172](#)
- ScrollBar component, [173](#)
- scrolling forms, [270–271](#)
  - automatic, [272–273](#)
  - coordinates in, [273–274](#), [273](#)
  - example, [271–272](#), [272](#)
  - messages for, [377](#)
- SDI (Single Document Interface), [310](#)
- sealed keyword, [924–925](#)
- search actions, [224](#)
- search application for HTTP, [751–754](#)
- search engines, [785–786](#), [786](#)
- search path setting, [347](#)
- search results, [9–10](#)
- search utility, [33](#)
- searches, case-insensitive, [602–603](#)
- searching window lists, [308–309](#)
- second-level message handlers, [369](#)
- secondary forms, [280](#)
  - adding, [280](#)
  - creating, [281–282](#)
- SecondaryShortCut property, [222](#)
- SecondOfWeek function, [91](#)
- Section component, [723–724](#)
- security

- client/server programming for, [557](#)
- in COM+, [494](#), [496](#)
- in Web pages, [493](#)
- SelAttributes property, [164](#)
- SelCount property, [167](#)
- SELECT statement, [628–629](#)
- SelectDirectory function, [103](#)
- Selected property, [167](#)
- SelectedColor property, [419](#)
- SelectedRows property, [542](#)
- selecting components
  - in Form Designer, [18](#)
  - referenced by properties, [20](#)
- SelectNextPage method, [204](#)
- Self keyword, [46–47](#)
- self pointer, [70](#), [125](#)
- SellItem property, [379](#)
- semicolons (;)
  - for connection strings, [620](#)
  - in SELECT, [628](#)
- Sender object, [228](#)
- sending data
  - database requests, [547–549](#), [548](#)
  - to Excel tables, [479](#)
  - mail, [748](#)
  - over socket connections, [744–747](#), [747](#)
  - to Word, [479](#)
- SendList program, [748](#), [749](#)
- SendMessage function, [304](#)
- SendResponse method, [784](#)
- SendStream method, [784](#)
- SendToDb example, [548–549](#), [548](#)
- separators for toolbars, [213](#)
- sequences
  - diagrams for, [433–434](#), [433](#)
  - in Oracle, [565](#)
- server constraints, [657–658](#)
- server-side cursors, [629](#), [636](#)
- server-side programming, [564–565](#)
- server-side scripts, [791–793](#), [792](#)
- server-side support components for DataSnap, [653](#)

server socket connections, [740](#)

ServerAddress property, [745](#)

ServerGUID property, [655](#)

ServerName property, [655](#)

servers

Automation, [470](#), [470](#)

code for, [472–473](#)

in components, [477–478](#), [477](#)

registering, [473–474](#)

type-library editor for, [470–472](#), [471](#)

COM. See [COM \(Component Object Model\)](#)

service providers, [618](#)

Services file, [740](#)

Session component, [793](#)

SessionCount property, [246](#)

sessions

in Web applications, [821–823](#), [823](#)

in WebSnap, [805–806](#), [807](#)

SessionService component, [805–806](#)

Set methods in TAXForm1, [491](#)

set operations on lists, [128](#)

Set\_Value method, [473](#)

SetAbort method, [497](#)

SetBookmarkData method, [696–697](#)

SetBookmarkFlag method, [696](#)

SetBounds method

in TControl, [159](#)

in TMdArrow, [361](#)

in TMdRecordView, [680](#)

SetCapture function, [264](#)

SetChangeFormFont method, [344](#)

SetComplete method, [497](#)

SetData method, [414](#), [702](#)

SetDataField method

in TMdDbProgress, [672–673](#)

in TMdDbTrack, [676](#)

SetDataSource method

in TMdDbProgress, [672](#)

in TMdDbTrack, [676](#)

SetDirection method, [359–360](#)

SetFieldData method

in TMdDataSetOne, [702–703](#)



- in TMdDirDataset, [706](#)
- SetFirstName method, [353](#)
- SetForegroundWindow function, [307](#), [309](#)
- SetLabel method, [354](#)
- SetLines method, [379–380](#)
- SetLinesPerRow method, [684](#)
- SetMemoryManager function, [84](#), [334](#)
- SetMoreData method, [384](#)
- SetOleFont method, [478](#)
- SetOleStrings function, [478](#)
- SetOptionalParameter method, [668](#)
- SetPen method, [363](#)
- SetPropValue method, [419](#)
- SetRecNo method, [698](#)
- SetRoundMode function, [90](#)
- SetSchemaInfo method, [578–579](#), [579](#)
- SetShareData method, [414](#)
- SetSubComponent method, [351](#), [357–358](#)
- SetText method, [356](#)
- SetTextLineBreakStyle function, [85](#)
- setTimeout function, [823](#)
- Settings page, [836](#)
- SetValue method
  - in TDate, [51](#)
  - in TSoundProperty, [390](#)
- SetViewer method, [356](#)
- SetWindowLong function, [298](#), [317](#)
- SetWindowOrgEx function, [274](#)
- SetYear method, [51](#)
- SharedConnection component, [653](#)
- ShareMem unit, [95–96](#), [405–406](#), [409](#)
- sharing data with memory-mapped files, [413–414](#), [415](#)
- ShellExecute function, [212](#), [750](#)
- shift-state modifiers, [262](#)
- shortcut keys, [16](#)
- ShortCut property, [222](#)
- Show Compiler Progress option, [30](#)
- Show Component Captions setting, [19](#)

Show method, [280](#), [286](#), [820](#)

ShowApp example, [297–298](#)

ShowColumnHeaders property, [191](#)

ShowException method, [75–76](#)

ShowForm method, [606](#)

ShowFrame method, [329](#)

ShowHints property, [183](#)

ShowInfoProc function, [426–427](#)

Showing property, [159](#)

ShowMainForm property, [297](#)

ShowMessage function, [289](#), [403](#)

ShowMessageFmt method, [289](#)

ShowMessagePos method, [289](#)

ShowModal method, [280](#), [282](#), [285](#), [411](#)

ShowSender method, [106](#)

ShowStatus method, [751](#), [754](#)

ShowStringForm method, [252](#)

ShowTotal method, [305–306](#)

Sign function, [89](#)

signals in Qt, [162](#)

Simple API for XML (SAX), [850–853](#), [853](#), [872–873](#)

Simple Mail Transfer Protocol (SMTP), [740](#)

Simple Object Access Protocol (SOAP), [876](#), [889–892](#)

Simple Security Controller option, [726](#)

SimpleDataSet component, [573–574](#), [576](#)

SimpleMemTest unit, [335–336](#)

SimpleObjectBroker component, [667](#)

SimplePanel property, [217](#)

SimpleRoundTo function, [90](#)

SimpleText property, [217](#), [751](#)

Single Document Interface (SDI), [310](#)

single-instance secondary forms, [281–282](#)

single instancing, [463](#)

single page access rights, [808](#)

single-threaded Apartment model, [463](#)

Single threading model, [463](#)

Single type, [478](#)

size

components, [19](#)

controls, [159](#)

executables, [82–83](#)

fonts, [276–277](#)

forms, [269](#)

screen, [299](#)

Size property, [136](#)

sLineBreak constant, [85](#)

SmallImages property, [189](#)

SmallInt type, [478](#)

Smith-Ferrier, Guy, [615](#)

SMTP (Simple Mail Transfer Protocol), [740](#)

SMTP component, [748](#)

SN SDK, [916](#)

Snap To Grid option, [18](#)

SnapBuffer property, [269](#)

snapping to screen, [269](#)

SOAP (Simple Object Access Protocol), [651](#), [876](#), [889–892](#)

SOAP-based solutions, [651](#)

SOAP headers, debugging, [887–888](#), [888](#)

Soap Server Data Module, [889](#)

SoapConnection component, [889](#)

SoapDataClient example, [891](#)

SoapDataServer example, [890](#)

SoapEmployee example, [886–887](#), [887](#)

SoapEmployeeIntf unit, [883](#)

SOAPServersIID property, [891](#)

socket programming, [737–739](#)

connections in, [740](#)

for database data, [744–747](#), [747](#)

domain names in, [739](#)

high-level protocols in, [740](#)

Indy components in, [741–744](#), [744](#)

IP addresses in, [739](#)

ports in, [739–740](#)

SocketConnection component, [652](#)

sort by directive, [573](#)

SortFieldNames property, [573](#)

sorting in ListView, [191](#)

SortType property, [191](#)

Sound button, [371–372](#)

sound properties editor, [388–392](#), [390–391](#)

SoundDown property, [388](#)

soundex algorithm, [91–92](#)

SoundUp property, [388](#)

source code files, [27–28](#), [38–39](#)

Source Doc Generation page, [445](#)

Source Doc Import page, [445](#)

Source Options page, [10](#), [15](#)

special characters in XML, [834](#)

special keys, messages for, [376](#)

speed

- in Automation, [476](#)
- in library selection, [155](#)

Splash example, [290](#), [292](#)

splash screens, [289–292](#), [290](#)

Splash1 example, [290–291](#)

Splash2 example, [291–292](#)

Split1 example, [180](#), [180](#)

Split2 example, [181](#)

SplitH example, [181](#)

SplitLong method, [786](#)

Splitter component, [180](#)

splitting forms, [180–181](#), [180](#)

SQL Links drivers, [507](#)

SQL Monitor tool, [33](#)

SQLClientDataSet component, [574](#)

SQLConnection component, [569–572](#), [571](#), [575](#), [589](#)

SQLDataSet component, [573](#), [803](#)

SqlMon.exe tool, [33](#)

SQLMonitor component, [574](#), [576–577](#), [577](#)

SQLOLEDB driver, [617](#)

SQLTimeStampToStr function, [574](#)

stacks, [131–132](#), [398](#)

Standalone mode in IntraWeb, [815](#)

Starkey, Jim, [561–562](#)

startDocument event, [850](#), [852](#)

- startElement event, [850](#), [852](#), [872](#)
- StartID property, [830](#)
- StartTransaction method, [589–590](#)
- state diagrams, [434](#)
- state of windows, [268–269](#)
- State property
  - in CheckBoxList, [169](#)
  - in datasets, [524–525](#), [701](#)
- state-setter commands, [173](#)
- StateImages property, [189](#)
- static binding, [63](#)
- static cursors, [630–631](#)
- static linking, [397–398](#)
- static members, [925–926](#)
- static methods, [70](#)
- StaticListAction component, [248–249](#)
- StaticSynchronize method, [305](#)
- Statistics page, [773](#)
- status
  - of ClientDataSet records, [585](#), [585](#)
  - of datasets, [524–525](#)
- status bars, [217–219](#), [218](#), [718–719](#)
- StatusBar component, [217](#)
- StatusFilter property, [586](#)
- stdcall calling convention
  - for DLLs, [399](#)
  - for Web services, [880](#)
- stdcall directive, [402](#)
- StdConvs unit, [91](#), [96](#)
- StdCtrls unit, [152](#)
- StdVCL library, [478](#)
- Stop on Delphi Exceptions options, [74](#)
- store and paint approach, [266](#)
- stored directive, [344](#)
- stored procedures, [564–565](#)
- StoreDefs property, [511](#)
- StoredProc component, [507](#)
- StoredProcName property, [573](#)
- StoreRAV property, [719](#)

Str function, [83](#)

StrDemo example, [92](#)

StreamDSDemo example, [703–704](#), [704](#)

streaming, [135](#)

classes for, [137–138](#)

compressing, [145–146](#), [145](#)

custom, [142–145](#)

TReader and TWriter, [139–140](#)

TStream, [135–137](#)

vs. code generation, [113–114](#)

files, [138–139](#)

persistence in, [140–142](#), [141](#)

string lists

in Automation, [478](#)

for combo boxes, [169](#)

for CSS, [797](#)

for memo lines, [164](#)

for properties, [52](#)

StringFromGuid2 function, [459](#)

StringReplace method, [512](#)

strings

comparisons with, [91–92](#)

conversions with, [86–87](#), [187–188](#)

in Delphi for .NET Preview, [920–921](#)

exporting from DLLs, [404–406](#)

for lists, [128–129](#)

memory for, [58](#)

opaque, [538](#)

unit for, [91–93](#)

URL, [752](#)

in XML, [837](#)

StringToColor function, [188](#)

StringToFloatSkipping method, [370–371](#)

StringToGUID function, [459](#)

StringValuesList component, [794](#)

StripParamQuotes property, [760](#)

strong name assemblies, [916](#)

StrToBool function, [86](#)

StrUtils unit, [91–93](#)

stub routines, [70](#)

style sheets, [764–765](#), [797](#), [859](#)

StyleRule property, [797](#), [859](#)

styles and Style property

- in Application, [219](#)
- for CLX, [219–220](#), [220](#)
- in ColorBox, [169](#)
- in ComboBox, [168](#)
- for forms, [254](#)
  - borders, [254–256](#), [254](#)
  - windows, [257–259](#), [258](#)
- in InetXPageProducer, [859](#)
- for list boxes, [187](#)
- in ModelMaker, [452](#)
- in TMdFontCombo, [343](#)
- in TToolButton, [213](#)

Styles string list, [797](#)

StylesDemo program, [219–220](#), [220](#)

StylesFile property, [797](#), [859](#)

subclassing, [317–318](#)

subcomponents, publishing, [350–352](#), [352](#)

SubMenuImages property, [174](#)

Support function, [87](#)

SupportCallbacks property, [652](#)

SupportedBrowsers property, [814](#)

Supports function, [424](#)

Supports transactions option, [495](#)

surrogate keys, [559](#)

symbols

- code completion for, [15](#)

- defining, [156](#)

- in diagrams, [435](#)

synchronization classes, [86](#)

Synchronize method

- in TFindWebThread, [751](#)

- in TThread, [305–306](#)

SyncObjs unit, [148](#)

Syntax Helper option, [488](#)

syntax of XML, [834–835](#)

SysConst unit, [85–86](#)

SysInit unit, [84–85](#)

system colors, [160](#)

system data in IBX, [599–600](#), [599](#)

system-level interfaces, [615](#)

system tables, [578](#)

System unit, [84–85](#)

SysUtils unit, [85–88](#), [102–104](#), [103–104](#)

◀ PREVIOUS

NEXT ▶



# Index

## T

tab actions, [224](#)

TabbedNotebook component, [202](#)

TabBmp object, [210](#)

TabControl component, [202](#)

for image viewer, [207–210](#), [207](#)

for pages, [328–329](#), [329](#)

table bookmarks, [538](#)

Table component, [507](#)

TableAttributes property, [763](#)

TableName property

in datasets, [573](#), [704](#)

in TADOTable, [624](#)

tables

constraints on, [657–658](#)

editing columns in, [539–540](#)

events for, [658–659](#)

in HTML, [762–764](#), [763](#)

system, [578](#)

totals of columns in, [537](#), [537](#)

virtual method tables, [66](#), [70](#), [466](#)

TabOrder property, [176](#)

TAboutBox class, [290](#)

tabs in editor, [11](#)

Tabs menu, [24](#)

TabSet component, [202](#)

TabSheet components, [202–207](#), [204–205](#)

TabStop property, [176](#)

TabVisible property, [205](#), [210](#)

TAction class, [222](#), [385](#)

TActionBar collection, [244](#)

TActionClient class, [244](#)

TActiveXControl class, [458](#), [488](#)

TActiveXControlFactory class, [460](#)

TADTField class, [529](#)

Tag property, [124](#)

TAggregateField class, [529](#)

TagParams list, [761](#)

tags in XML, [833](#)

TAlignment class, [230](#)

TAnimal class, [62–64](#), [67](#)

TApplication class, [295–296](#)

TAppServerOne class, [654](#)

TargetList function, [386–387](#)

TArrayField class, [529](#)

TAutoIncField class, [529](#)

TAutoObject class, [458](#)

TAutoObjectFactory class, [460](#)

TAXForm1 class, [491–492](#)

TBasicAction class, [222](#)

TBCDField class, [529](#)

TBinaryField class, [529](#)

TBitBtn class, [184](#), [373](#)

TBitmap class, [208](#)

TBits class, [147](#)

TBlobField class, [139](#), [529](#)

TBlobStream class, [138](#)

TBookmark class, [538](#)

TBookmarkList class, [542](#)

TBookmarkStr class, [542–543](#)

TBooleanField class, [529](#)

TBrush class, [358](#)

TBucketList class, [132](#)

TButton class, [152](#), [173](#)

TBytesField class, [529](#)

TCanTest class, [384](#)

TCanvas class, [150](#), [262–263](#)

TCGIApplication class, [771](#)

TCharProperty class, [388](#)

TCheckConstraint class, [657](#)

TClass class, [77](#)

TClassFinder class, [147](#)

TClassList class, [131](#)

TClipboard class, [208](#)

TCollection class, [131](#), [147](#)

TCollectionItem class, [131](#), [147](#), [382](#)

TColor class, [160–161](#)

TComObject class, [457–458](#), [464](#)

TComObjectFactory class, [460](#), [462](#)

TComponent class, [110–111](#), [110](#), [117](#), [340](#)

in Delphi for .NET Preview, [931–932](#)

form fields in

hiding, [122–123](#)

removing, [121–122](#)

Name property in, [121](#)

ownership in, [117–120](#), [120](#)

Tag property in, [124](#)

TComponentEditor class, [393](#)

TComponentList class, [131](#)

TCompressStream class, [145](#)

TConnectionAdmin class, [579](#)

TContainedAction class, [222](#)

TControl class, [158](#)

activation and visibility, [159](#)

colors, [160–161](#)

fonts, [159–160](#)

Parent property, [158–159](#)

size and position, [159](#)

TCoolBand class, [231](#)

TCoolBar component, [231](#)

TCP/IP (Transmission Control Protocol/Internet Protocol), [652](#), [737–739](#)

TCP/IP sockets, [651](#)

TCP ports, [739–740](#)

TcpClient class, [737–738](#)

TcpServer class, [737–738](#)

TCriticalSection class, [305](#)

TCurrencyField class, [529](#)

TCustomAction class, [222](#)

TCustomActionToolBar class, [247–248](#)

TCustomADODataset, [618](#)

TCustomConnection class, [569](#)

TCustomControl class, [158](#), [487](#)

TCustomDockForm class, [234](#)

TCustomEdit class, [369](#)

TCustomForm class, [251–252](#)

[TCustomGrid class, 679](#)  
[TCustomLabel class, 350](#)  
[TCustomMemoryStream class, 137](#)  
[TCustomSQLDataSet class, 572](#)  
[TCustomVariantType class, 94](#)  
[TCustomWebDispatcher class, 771](#)  
[TDataLink class, 669–670](#)  
[TDataModule class, 147, 535, 653](#)  
[TDataSaxHandler class, 872–873](#)  
[TDataSet class, 520–524, 536, 552, 592, 687](#)  
[TDataSetField class, 529](#)  
[TDatasetProvider class, 658](#)  
[TDate class, 51, 51](#)  
[TDateField class, 529](#)  
[TDateForm class, 56–58](#)  
[TDateListI class, 134–135](#)  
[TDateListW class, 134–135](#)  
[TDateTime class, 49–50](#)  
    in DateUtils unit, [91](#)  
    in Delphi for .NET Preview, [921](#)  
    in System unit, [84](#)  
[TDateTimeField class, 529](#)  
[TDBGHack class, 544](#)  
[TDecompressStream class, 145](#)  
[TDump.exe tool, 33](#)  
[TeamSource tool, 33](#)  
[Telnet ports, 740](#)  
[temperature conversions, 96](#)  
templates  
    code, [15, 450–452, 450](#)  
    in Component Palette, [25–26, 26](#)  
    HTML, [788](#)  
    in Object Repository, [40–41](#)  
    in UDDI, [895](#)  
[TEncodedStream class, 142–143](#)  
[TEnumProperty class, 388](#)  
[TerminateAndRedirect method, 820](#)  
[TestCom example, 464–465](#)  
testing  
    COM server, [464–466](#)

datasets, [703–704](#), [704](#)

debugger for, [774](#)

TestStreamFormat method, [140](#)

TEvent class, [305](#)

text

for case diagrams, [434](#)

DFM files stored as, [19](#)

in Rave, [723](#)

text-based data-aware controls, [516](#)

Text component, [723](#)

text files

Jet engine for, [626–628](#)

line breaks in, [85](#)

Text IISAM, [626–627](#)

text-input components

Edit, [163](#)

LabeledEdit, [163](#)

MaskEdit, [163–164](#)

MemoRich and RichEdit, [164–165](#)

TextViewer, [165](#), [165](#)

Text property

in ComboBox, [168](#)

in DOM, [844](#)

in Edit, [162](#)

in MemoRich, [164](#)

for panels, [217](#)

TextBrowser control, [756](#)

TextHeight attribute, [187](#)

TextViewer component, [165](#), [165](#)

TField class, [525–527](#), [641–642](#)

TFieldDataLink class, [670–671](#)

TFileData class, [707](#)

TFile class, [139](#), [147](#)

TFileRec class, [85](#)

TFileStream class, [137](#)

TFindWebThread class, [752–754](#)

TFirstServer class, [473](#), [477](#)

TFloatField class, [527](#), [529](#)

TFMTBCDField class, [529](#)

TForm class, [251–252](#), [333](#)

for border icons, [256–257](#), [257](#)

for form style, [254](#)

- borders, [254–256](#), [254](#)
- windows, [257–259](#), [258](#)
- for plain forms, [252–253](#), [253](#)
- TFormBitmap class, [331](#), [333–334](#)
- TFormBorderStyle class, [256](#)
- TFormClass class, [77](#)
- TFormScrollBar class, [270](#)
- TFormSimpleColor class, [422](#)
- TFrameClass class, [329–330](#)
- TFrameList class, [324–325](#)
- TGraphicControl class, [158](#), [340](#), [359](#), [487](#)
- TGraphicField class, [530](#)
- TGUID class, [458–459](#)
- TGuidField class, [530](#)
- THandleStream class, [137](#)
- THashedStringList class, [133](#)
- themes, [220–221](#), [222](#)
- thick frame border style, [255](#)
- thin clients, [649](#)
- ThinCli1 example, [656](#)
- ThinCli2 example, [659](#), [659](#)
- ThinPlus example, [662–665](#), [663](#), [665](#)
- THintInfo structure, [183](#)
- THome class, [788–789](#)
- thousands separators, [370–371](#), [533](#)
- ThousandSeparator character, [533](#)
- threading methods in COM, [463](#)
- threads, [84](#), [304–307](#)
- threadvar variables, [822](#)
- three-tier architecture, [648](#)
- tiDirtyRead value, [590](#)
- TIDispatchField class, [530](#)
- TidThreadSafeInteger class, [822](#)
- Tile method, [312](#)
- TileMode property, [312](#)
- time
  - native formats for, [709](#)
  - unit for, [91](#)

Timer class, [350–351](#)

Timer property, [351](#)

timers

for messages, [302](#)

in Web applications, [823](#)

XClock, [492–493](#), [493](#)

TimeStamp field, [574](#)

timing, [476](#)

TInformSubscriber class, [498–499](#)

TIniFile class, [331–332](#)

TIntegerField class, [530](#)

TIntegerProperty class, [388](#)

TInterfacedField class, [530](#)

TInterfacedObject class, [70](#), [84](#), [457](#)

TInterfacedPersistent class, [147](#)

TInterfaceList class, [147](#)

TInvokableClass class, [880](#)

tiReadCommitted value, [590](#)

tiRepeatableRead value, [590](#)

TISAPIRequest class, [771](#)

TISAPIResponse class, [771](#)

TItemProp class, [171](#)

Title property, [379](#)

titles for applications, [297](#)

TIWServerController class, [822](#)

TJpegImage class, [783](#)

TLargeIntField class, [530](#)

TLB (type library) file, [37](#), [470](#)

TLibCli example, [474–475](#)

TlibdemoLib\_TLB file, [477](#)

tlibimp utility, [501](#)

TList class, [128–131](#), [130](#), [133](#)

TListControlAction class, [385](#)

TListItems class, [189](#)

TMdActiveButton class, [372–373](#)

TMdArrow class, [359–360](#), [363–365](#)

TMdClock class, [350–351](#)

TMdCustomDataSet class, [687–689](#)

TMdCustomListAction class, [385–386](#)

TMdDataSetStream class, [689–690](#)

TMdDbGrid class, [684](#)

TMdDbProgress class, [671–672](#)

TMdDbTrack class, [675](#)

TMdDirDataset class, [706](#)

TMdFontCombo class, [341–344](#)

TMdListBoxDialog class, [379](#)

TMdListCompEditor class, [393–395](#)

TMdListCopyAction class, [386](#)

TMdListCutAction class, [386](#)

TMdListDataSet class, [705](#)

TMdListPasteAction class, [386](#)

TMdMyColleciton class, [382–383](#)

TMdMyItem class, [382](#)

TMdNumEdit class, [369](#)

TMdObjDataSet class, [710](#)

TMdPersonalData class, [352–354](#)

TMdRecInfo class, [695–696](#)

TMdRecordLink class, [678](#)

TMdRecordView class, [679](#)

TMdSoundButton class, [371–372](#), [392](#)

TMdThousandEdit class, [370–371](#)

TMdViewer class, [355–356](#)

TMdWArrowX class, [488](#)

TMemoField class, [530](#)

TMemoryManager class, [84](#), [334–335](#)

TMemoryStream class, [137–138](#)

TMethod class, [115](#), [125](#)

TMultiReadExclusiveWriteSynchronizer class, [86](#)

TMySaxHandler class, [851–852](#)

TMySimpleSaxHandler class, [852](#)

TNewDate class, [59–60](#)

TNotifyEvent type, [364](#)

TNumber class, [462](#), [464](#)

TNumericField class, [530](#)

to-do lists, [7–9](#), [8](#)



TObject class, [54](#), [84](#), [104–108](#), [106](#), [108](#)

TObjectBucketList class, [132](#)

TObjectField class, [530](#)

TObjectHelper class, [930](#)

TObjectList class, [131–133](#)

TObjectQueue class, [132](#)

TObjectStack class, [131–132](#)

TODO comments, [7–9](#)

TODO files, [37](#)

TOutputStream class, [138](#)

toolbars and ToolBar control, [213](#), [230–231](#)

- actions with, [224](#), [229–230](#)
- captions for, [236](#)
- ControlBar, [231–234](#), [232](#)
- CoolBar, [230–231](#)
- docking in, [235–239](#), [236](#)
- hints in, [182–184](#), [184](#), [218](#), [218](#), [236](#)
- menus and combo boxes in, [216–217](#)
- in Rave, [718](#)
- RichBar example, [214–215](#), [214](#)

Tools API, [388](#)

Tooltip symbol insight feature, [12–13](#)

tooltips, [182–184](#), [184](#)

- expression evaluation, [16](#)
- hints, [19](#)

Top property

- for components, [18](#)
- for forms, [268](#)
- in TControl, [159](#)

Total example, [537](#), [537](#)

totals

- of columns, [537](#), [537](#)
- in Rave, [729–730](#)

TOwnedCollection class, [147](#)

TParser class, [147](#)

TPen class, [358](#)

TPersistent class, [110](#), [112–113](#)

- for Arrow component, [362–364](#), [364](#)
- property access in, [115–117](#), [117](#)
- published keyword for, [114–115](#)

TPoint structure, [93](#)

TPromptQueryButton class, [859](#)

TPropertyEditor class, [388](#)

TPropertyPage class, [490](#)

TPropertyPage1 class, [490](#)

TQueue class, [132](#)

TrackBar component, [172](#), [674–677](#), [677](#)

tracking forms, [299–302](#), [300](#)

TrackMode property, [815](#)

training classes, [608–611](#), [610](#)

transactional data modules, [497](#)

TransactionLevel property, [591](#)

transactions

actions for, [596](#)

in ADO, [634–636](#)

in client/server programming, [589–591](#), [591](#)

in COM+, [494](#)

InterBase for, [561](#)

TransactionsSupported property, [590](#)

TranSample example, [591](#), [591](#)

transformations

XML, [854–857](#), [854](#), [856–857](#), [886–887](#), [887](#)

XSL, [868–869](#)

TransformNode method, [866](#), [868](#)

TransformProvider component, [855](#), [856](#)

translations, BabelFish, [876–879](#), [877](#), [879](#)

Transmission Control Protocol/Internet Protocol (TCP/ IP), [652](#), [737–739](#)

transparent color in forms, [267](#), [267](#)

TransparentColor property, [267](#)

TransparentColorValue property, [267](#)

TReader class, [139–140](#), [147](#)

TRecall class, [147](#)

TReconcileAction class, [588](#)

TReconcileErrorForm dialog box, [641](#)

TRect structure, [93](#)

trees in XML, [835](#)

TreeView, [22–23](#)

TreeView component, [169–170](#), [193–197](#), [194](#)

custom nodes in, [197–198](#), [199](#)

for XML documents, [839–841](#), [840](#)

TreeView Items property editor, [193](#)

[TReferenceField class, 530](#)  
[TRegIniFile class, 332](#)  
[TRegistry class, 332](#)  
[TRegSvr.exe example, 464](#)  
[TRegSvr.exe tool, 34](#)  
[TRemoteDataModule class, 654](#)  
[TResourceStream class, 137](#)  
[triangle symbol in Band Style Editor, 728](#)  
[triggered events, 124](#)  
[triggers in Interbase 6, 565–566](#)  
[trigonometric functions, 89](#)  
[Trunc function, 921](#)  
[try blocks, 71–73](#)  
[TryEncodeDate function, 87](#)  
[TryEncodeTime function, 87](#)  
[TryFinally example, 72–73](#)  
[TryStrToCurr function, 87](#)  
[TryStrToDate function, 87](#)  
[TryStrToFloat function, 87](#)  
[TSaveStatusForm class, 331, 334](#)  
[TSchemaInfo class, 622](#)  
[TScreen class, 299](#)  
[TScrollingWidget class, 252](#)  
[TScrollingWinControl class, 252](#)  
[TSetProperty class, 388](#)  
[TSingleEvent class, 305](#)  
[TSmallIntField class, 530](#)  
[TSmallPoint class, 93](#)  
[TSOAPAttachment class, 893](#)  
[TSoapDataModule class, 890](#)  
[TSoapTestDm class, 890](#)  
[TSoundProperty class, 389](#)  
[TSQLConnection class, 569](#)  
[TSQLDataSet component, 572, 578](#)  
[TSQLTimeStampField class, 530, 574](#)  
[TStack class, 131–132](#)  
[TStream class, 135–137](#)

TStringField class, [530](#)  
TStringHash class, [133](#)  
TStringList class, [128–129](#), [148](#)  
TStringProperty class, [388](#)  
TStrings class, [128–129](#)  
TStringStream class, [137](#)  
TTags class, [761](#)  
TTextRec class, [85](#)  
TThread class, [147](#), [305](#)  
TThreadList class, [147](#)  
TTimeField class, [530](#)  
TToolButton class, [173](#), [213](#)  
TTransactionDesc type, [590](#)  
TTreeltems class, [197](#)  
TTreeNode class, [195](#)  
TTypedComObject class, [458](#)  
TTypedComObjectFactory class, [460](#)  
tuples, [560](#)  
Turbo Grep tool, [33](#)  
Turbo Register Server tool, [34](#)  
TUserSession class, [821–822](#), [825–826](#)  
TVarBytesField class, [530](#)  
TVarData record class, [84](#)  
TVariantField class, [530](#)  
TVariantManager class, [84](#)  
TViewerForm class, [322–323](#)  
TWebAppPageModule class, [787](#)  
TWebAppPageModuleFactory class, [807](#)  
TWebBrowser class, [485](#)  
TWebContext class, [806](#)  
TWebPageModuleFactory class, [807](#)  
TWebRequest class, [771](#), [775](#)  
TWebResponse class, [771](#)  
TWideStringField class, [530](#)  
TWidgetControl class, [158](#), [161–162](#), [251](#)  
TWinControl class, [158](#), [161](#), [251](#), [340](#)  
TWindowArrange class, [312](#)

TWindowCascade class, [312](#)

TWindowClose class, [312](#)

TWindowMinimizeAll class, [312](#)

TWindowTileHorizontal class, [312](#)

TWindowTileVertical class, [312](#)

TWinSocketStream class, [138](#)

TWordField class, [530](#)

TWriter class, [139–140](#), [147](#)

TXActAttributes class, [635](#)

type libraries

in Automation, [468–470](#)

editors for, [470–472](#), [471](#)

type library (TLB) file, [37](#), [470](#)

Type Library Importer, [501](#)

Type Library page, [471](#)

type-safe containers and lists, [133–135](#)

type-safe down-casting, [67–69](#)

typed parameters, [565](#)

TypeInfo unit, [116–117](#)

types

checking, [458](#)

COM, [478](#)

in Delphi for .NET Preview, [920](#), [926](#)

in inheritance, [62–63](#)

Types unit, [93](#)

TypesList method, [882](#)

TypInfo unit, [148](#)

◀ PREVIOUS

NEXT ▶

# Index

## U

UDDI (Universal Description, Discovery, and Integration) specification, [894–898](#), [896](#)

UDDIBrow.ini file, [895](#)

UddiInquiry example, [895–898](#), [896](#)

UDL files, [37](#)

UML diagrams, [429–430](#)

class, [431–432](#), [432–433](#)

common elements in, [435–436](#), [436](#)

sequence, [433–434](#), [433](#)

use case, [434](#)

UndoLastChange method, [514](#)

Unicode character sets, [835](#)

unidirectional cursors, [560–561](#), [566–567](#)

UniPrint example, [582–584](#)

Unit Code Editor, [439–441](#)

unit dependency diagrams, [435](#)

units

code editors for, [439–441](#)

in dbExpress, [568–569](#)

in inheritance, [60–61](#)

names for, [44](#)

namespaces for, [922–924](#)

in Project Explorer, [32](#)

in RTL, [81–82](#)

COM-related, [96](#)

ConvUtils and StdConvs, [91](#)

DateUtils, [91](#)

DelphiMM and ShareMem, [95–96](#)

Math, [88–90](#), [90](#)

StrUtils, [91–93](#)

SysConst, [85–88](#)

System and SysInit, [84–85](#)

SysUtils, [85–88](#), [102–104](#), [103–104](#)

Types, [93](#)

Variants and VarUtils, [94–95](#), [94](#)

in VCL and CLX, [157](#)

Universal Description, Discovery, and Integration (UDDI) specification, [894–898](#), [896](#)

UnloadPackage function, [419–420](#)

unmanaged code, [905](#)

unsafe casts, [44](#)

unsafe code, [44](#), [909](#)

unsafe types, [44](#), [920](#)

Update Criteria property, [640](#)

Update method

in painting, [266](#)

in TMDMyColleciton, [383](#)

update regions, [267](#)

UpdateBatch method, [638](#)

UpdateClock method, [352](#)

UpdateData method, [677](#)

UpdateLabel method, [353](#)

UpdateList method, [135](#)

UpdateMode property, [640](#)

UpdateOleObject method, [490](#)

UpdateRegistry method

in DataSnap, [667](#)

in TRemoteDataModule, [654](#)

updates

in ADO, [615](#)

batch, [638–642](#)

conflicts in, [641–642](#)

joins, [636–637](#)

in ClientDataSet, [584–589](#), [585–586](#), [588](#)

in DataSnap example, [659–662](#)

in dbExpress, [576–578](#)

DLLs, [416](#)

triggers fired by, [565](#)

UpdatesPending property, [638](#)

UpdateStatus method, [585](#), [638](#)

UpdateTarget method

in TAction, [385](#)

in TMDCustomListAction, [385–386](#)

UpDown component, [172](#)

uppercase names, converting to, [602–603](#)

upsizing applications, [556](#)

URL strings, [752](#)

URLs

in IntraWeb, [814–815](#)

in WinInet, [755](#)

UsageCount property, [246](#)

use cases, diagrams for, [434](#)

UseCol example, [411](#)

UseDockManager property, [235](#), [240](#)

UseFrame property, [826](#)

UseMem example, [413–414](#), [415](#)

user-defined window messages, [309–310](#)

user interface

ActionList component for, [222–224](#), [223](#)

example, [225–228](#), [226](#), [228](#)

predefined actions in, [224–225](#)

with toolbars, [229–230](#)

ActionManager architecture, [241–250](#), [243–245](#), [249](#)

CLX styles, [219–220](#), [220](#)

docking support, [234–241](#), [236](#)

in InterBase, [605–608](#), [607](#)

multiple-page forms. See [multiple-page forms](#)

themes, [220–221](#), [222](#)

toolbars, [213–219](#), [230–234](#), [232](#)

for wizard, [210–213](#), [211](#)

user logins, [807–808](#)

user-related adapters, [794](#)

uses statement

for controls, [154](#)

in Delphi for .NET Preview, [923–924](#)

for external symbols, [15](#)

in implementation section, [256](#)

UseSOAPAdapter property, [891](#)

◀ PREVIOUS

NEXT ▶



# Index

## V

validation in XML, [849–850](#)

Value property, [525](#)

ValueChecked property, [517](#)

ValueFromIndex property, [129](#)

ValueListEditor component, [170–172](#), [170](#)

ValueUnchecked property, [517](#)

VarArrayCreate function, [514](#)

VarArrayOf function, [514](#)

VarComplexCreate functions, [95](#)

varEmpty type, [477](#)

variables

code completion for, [14–15](#)

pointers to, [55](#), [55](#)

variant records, [921](#)

Variant type, [468](#), [478](#)

custom, [94–95](#), [94](#)

System unit support for, [84](#)

Variants unit, [94–95](#), [94](#)

varNull type, [477](#)

VarSQLTimeStampCreate function, [574](#)

VarUtils unit, [94–95](#), [94](#)

VBX standard, [483](#)

VCL (Visual Components Library), [110–112](#), [110](#)

VCL controls

converting, [156–158](#)

for Delphi for .NET Preview, [931–933](#), [933](#)

dual libraries support for, [151–155](#), [152](#)

visual libraries for, [155–156](#), [156](#)

vs. VisualCLX, [149–158](#), [152](#)

VCL.DCP file, [408](#)

VCL Hierarchy Wizard, [946–947](#)

VCL package, [83](#)

VCL70.BPL file, [408](#)

VclToClx tool, [158](#), [947–948](#)

verbs

in compound documents, [480](#)

in IComponentEditor, [393](#)

version tags in DCU files, [416](#)

versions

dbExpress, [568–569](#)

DLL, [399](#)

in InterBase, [561](#)

in .NET architecture, [916–917](#), [917](#)

in packages, [416–417](#)

TeamSource control system, [33](#)

VertScrollBar property, [270](#)

ViewAccess property, [808](#)

ViewGrid example, [682](#), [682](#)

views, loadable, [16–18](#), [17–18](#)

virtual action items, [250](#)

virtual destructors, [55](#)

virtual fields, [50](#)

virtual functions, [64–65](#)

virtual keys, [262](#)

virtual keyword, [64–65](#)

virtual method tables (VMTs), [66](#), [70](#), [466](#)

virtual methods, [398](#)

in COM server, [466–467](#)

vs. dynamic, [65–66](#)

in interfaces, [70](#)

VirtualListAction component, [248–249](#)

VirtualListAction1GetItem method, [250](#)

visibility and Visible property

for actions, [222](#)

for controls, [159](#)

in Delphi for .NET Preview, [925](#)

for fields, [658](#)

for inherited forms, [321](#)

for scroll bars, [270](#)

for secondary forms, [280](#)

in TControl, [159](#)

in TField, [528](#)

VisibleButtons property, [516](#)

VisibleRowCount property, [685](#)

Visitor pattern, [449](#)

Visual Basic controls, [483](#)

Visual Components Library (VCL), [110–112](#), [110](#)

visual editing, [484](#)

visual form inheritance, [318–319](#)

from base forms, [319–321](#), [319–320](#)

polymorphic, [321–324](#), [322](#)

visual libraries, [155–156](#), [156](#)

visual styles in ModelMaker, [452](#)

VisualCLX controls, [110–111](#)

converting, [156–158](#)

dual libraries support for, [151–155](#), [152](#)

vs. VCL, [149–158](#), [152](#)

visual libraries for, [155–156](#), [156](#)

VMTs (virtual method tables), [66](#), [70](#), [466](#)

Voice method, [63–64](#)

vtables, [66](#), [70](#), [466](#)

◀ PREVIOUS

NEXT ▶

# Index

## W

W3C (World Wide Web Consortium)

for HTML, [759](#)

for XML, [833](#)

WaitForSingleObject function, [308](#)

\$WARN directive, [44](#)

warnings, compiler, [31–32](#), [44](#)

Watch List, [42](#)

Web and Web pages. See also [Internet programming](#)

actions for, [224](#)

ActiveX controls in, [492–493](#), [493](#)

hit counter for, [782–785](#)

search engines for, [785–786](#), [786](#)

services for, [875–876](#)

attachments, [892–893](#), [894](#)

BabelFish translations, [876–879](#), [877](#), [879](#)

currency conversion, [880–882](#), [881](#), [883](#)

database data, [883–887](#), [887](#)

debugging SOAP headers, [887–888](#), [888](#)

existing classes as, [888–889](#)

SOAP and WSDL, [876](#)

UDDI, [894–898](#), [896](#)

WebSnap for. See [WebSnap application](#)

Web App Debugger tool, [33](#), [772–774](#), [774](#), [887–889](#), [888](#)

Web Services Description Language (WSDL), [876](#), [881–883](#), [881](#)

Web Services page, [889](#)

WebAppComponents component, [787](#)

WebAppDbg.exe tool, [33](#), [772–774](#), [774](#)

WebApplication class, [817](#)

WebBroker technology, [770–772](#)

with Apache, [781–872](#)

debugging in, [772–774](#), [774](#)

for dynamic database reporting, [776–777](#), [777](#)

with IntraWeb, [823–824](#)

for multipurpose WebModule, [774–776](#)

for queries and forms, [777–781](#), [780](#)

Web hit counter, [782–785](#)

Web searches, [785–786](#), [786](#)

WebBrowser control, [485–486](#), [486](#)

WebConnection component, [652](#)

WebContext object, [806](#)

WebDemo example, [485–486](#), [486](#)

WebDispatch property, [858](#)

WebDispatcher component, [771–772](#)

WebFind example, [751–754](#), [755](#)

WebModules, [770–772](#)

WebRequestHandler object, [773](#)

WebSearcher example, [785–786](#), [786](#)

WebSnap application, [786–789](#), [787](#)  
for adapters, [794–798](#), [796](#)  
for databases, [798](#)  
data editing, [801–803](#), [802](#)  
data module for, [798–799](#)  
DataSetAdapter for, [799–801](#), [800](#)  
master/detail in, [803–805](#), [805](#)  
with IntraWeb, [823–824](#)  
logins in, [807–808](#)  
for multiple pages, [789–791](#), [790](#)  
permissions in, [808](#)  
for server-side scripts, [791–793](#), [792](#)  
sessions in, [805–806](#), [807](#)  
XSLT with, [866–868](#), [866](#)

WebSnap page, [787](#)

well-formed XML, [835](#)

What's this? Help, [256](#)

wheel events, [261](#)

White Pages in UDDI, [894](#)

white space in XML, [834](#)

wide string support, [87](#)

WideCompareStr function, [87](#)

WideCompareText function, [87](#)

WideFormat function, [87](#)

WideLowerCase function, [87](#)

WideSameStr function, [87](#)

WideSameText function, [87](#)

WideString type, [87](#), [459–460](#), [478](#)

WideUpperCase function, [87](#)

Width property  
for components, [18](#)  
for forms, [268–269](#)  
in Splitter, [180](#)  
in TControl, [159](#)  
in TMdArrow, [359](#)

- WINAPI modifier, [401](#)
- window lists, searching, [308–309](#)
- window procedures, [161](#)
- WindowMenu property, [311](#)
- WindowProc property, [161](#)
- windows, [161](#)
  - controls in, [158](#), [368](#)
    - message handlers in, [368–373](#), [373](#)
    - messages in, [373–378](#)
    - notifications in, [376–377](#)
    - overriding message handlers in, [371–372](#)
  - for forms, [257–259](#), [258](#)
  - painting in, [265–267](#)
- Windows Interactive SQL (WISQL), [562](#)
- Windows menu, [7](#)
- Windows operating system
  - background processing and multitasking in, [304](#)
  - common dialogs, [287–288](#), [288](#)
  - events in, [302–303](#)
  - messages in, [303–304](#), [309–310](#), [374](#)
  - resource files from, [28](#)
  - themes in, [220–221](#), [222](#)
- WindowState property, [268–269](#)
- WindowsXP.res file, [220](#)
- WinInet API, [754–756](#)
- WinShoes component, [739](#)
- WinSight tool, [33](#), [303](#)
- WinVersion example, [85–86](#)
- WISQL (Windows Interactive SQL), [562](#)
- WithinPastDays function, [91](#)
- wizards
  - DLL, [41](#)
  - user interface for, [210–213](#), [211](#)
- WizardUI example, [210–213](#), [211](#)
- wm\_Char message, [368](#)
- wm\_Command message, [376](#)
- wm\_EraseBkgnd message, [317](#)
- wm\_HScroll message, [272](#)
- wm\_LButtonDbtClk message, [365](#)
- wm\_LButtonDown message, [371](#)
- wm\_LButtonUp message, [371](#)

- wm\_NCCalcSize message, [269](#)
- wm\_NCHitTest message, [258](#), [269](#)
- wm\_NCLButtonDown message, [269](#)
- wm\_NCPaint message, [269](#)
- wm\_Size message, [270](#)
- wm\_User message, [66](#), [301](#)
- WMHScroll method, [272](#)
- WNDCLASS, [307](#)
- WndProc method, [161](#), [317](#)
- Word
  - with Automation, [479](#)
  - DLLs for, [411](#)
- WordBool type, [478](#)
- WORDCALL.TXT file, [411](#)
- WordCont example, [482–483](#), [483](#)
- World Wide Web. See [Internet programming](#); [IntraWeb library](#); [Web and Web pages](#)
- World Wide Web Consortium (W3C)
  - for HTML, [759](#)
  - for XML, [833](#)
- Wrapper pattern, [449](#)
- wrapper properties, [350](#)
- wrappers
  - for ActiveX controls, [485](#)
  - for XML, [838](#)
- write clauses, [50](#), [52](#), [465](#)
- Write method, [143](#)
- write-only properties, [52](#)
- WriteBool method, [332](#)
- WriteBuffer method, [137](#)
- WriteComponent method, [136](#), [140](#), [142](#)
- WriteComponentRes method, [140](#)
- WriteInteger method, [332](#)
- WriteString method, [332](#)
- ws constants, [269](#)
- Ws.exe tool, [33](#), [303](#)
- WSDL (Web Services Description Language), [876](#), [881–883](#), [881](#)
- WSDLHTMLPublish component, [880](#)
- WSnap1 example, [791–793](#)
- WSnap2 example, [792–793](#), [792](#), [795–796](#), [796](#)

WSnapMD example, [803–805](#), [805](#)

WSnapTable example, [798–803](#), [800](#), [802](#)

WWW (World Wide Web). See [Internet programming](#); [IntraWeb library](#); [Web and Web pages](#)

◀ PREVIOUS

NEXT ▶



# Index

## X

Xalan processor, [865](#)

XArrow example, [489](#)

XClock control, [492–493](#), [493](#)

Xerces DOM, [838](#)

XFM files, [152–154](#)

XForm1 example, [492](#)

XLSReadWrite component, [625](#)

XML, [833–834](#)

core syntax of, [834–835](#)

data binding interfaces in, [846–850](#), [847](#)

for DataSnap, [653](#)

document management in, [838–846](#), [840](#), [843](#), [846](#)

file format for, [644–646](#)

and Internet Express, [858–864](#), [861](#), [863](#)

large documents with, [869–873](#), [871](#)

mapping with transformations, [854–857](#), [854](#), [856–857](#), [886–887](#), [887](#)

passing documents in, [884–886](#)

SAX API for, [850–853](#), [853](#)

well-formed, [835](#)

working with, [836–837](#), [837](#)

and XSLT, [864–869](#), [866](#)

XML-based formats in ClientDataSet, [511](#), [512](#)

XML Data Binding Wizard, [846–849](#), [847](#)

XML Mapper tool, [33](#), [854–857](#), [854](#), [856–857](#), [886–887](#), [887](#)

XML Schema Validator (XSV), [850](#)

XMLBroker component, [858](#)

XMLData property, [511](#), [866](#)

Xmldb.js file, [859](#)

XmlDemo example, [927](#)

Xmldisp.js file, [859](#)

XMLDocument component, [836](#), [838](#), [842](#), [848](#)

xmlDom component, [868](#)

Xmldom.js file, [859](#)

XmlDomTree example, [839–841](#), [840](#)

XmlEditOne example, [836–837](#), [837](#)

Xmlerrdisp.js file, [859](#)

XmlInterface example, [848](#)

[XMLIntf unit, 839](#)

[XmlIntfDefinition unit, 848](#)

[XmlMapper.exe tool, 33](#)

[XMLMapping example, 855–857, 856](#)

[XmlShow.js file, 859](#)

[XMLTransform component, 855, 857](#)

[XMLTransformClient component, 855](#)

[XMLTransformProvider component, 855](#)

[XPath technology, 864–865](#)

[XpManifest component, 221](#)

[XSL transformations, 868–869](#)

[XslCust example, 866–868, 866](#)

[XslDom component, 868](#)

[XslEmbed project, 865](#)

[XSLPageProducer component, 866](#)

[XSLT \(Extensible Stylesheet Language\), 864](#)  
examples, [865–866](#)  
with WebSnap, [866–868, 866](#)  
XPath in, [864–865](#)

[XSV \(XML Schema Validator\), 850](#)

[Xt processor, 865](#)

[◀ PREVIOUS](#)

[NEXT ▶](#)

# Index

## Y

Yellow Pages in UDDI, [895](#)

## Index

### Z

z-order, [312](#)

ZCompress example, [145–146](#), [145](#)

zero-configuration thin-client architecture, [649](#)

ZLib utility, [145–146](#), [145](#)

Zoom toolbar in Rave, [718](#)

# List of Figures

## Chapter 1: Delphi 7 and Its IDE

[Figure 1.1:](#) A form and a data module in the Delphi 7 IDE

[Figure 1.2:](#) The Preferences page of the Environment Options dialog box

[Figure 1.3:](#) The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code.

[Figure 1.4:](#) The multiple languages supported by the Delphi IDE can be associated with various file extensions in the Source Options page of the Editor Properties dialog box.

[Figure 1.5:](#) You can configure the Code Explorer in the Environment Options dialog box.

[Figure 1.6:](#) The Diagram view shows relationships among components (and even allows you to set them up).

[Figure 1.7:](#) The Print Options for the Diagram view

[Figure 1.8:](#) The Frames1 example demon-strates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.

[Figure 1.9:](#) Delphi's multitarget Project Manager

[Figure 1.10:](#) The new Compiler Messages page of the Project Options dialog box

[Figure 1.11:](#) The Project Browser

[Figure 1.12:](#) The first page of the New Items dialog box, generally known as the Object Repository

## Chapter 2: The Delphi Programming Language

[Figure 2.1:](#) The output of the CreateComps example, which creates Button components at run time

[Figure 2.2:](#) The DateProp example's form

[Figure 2.3:](#) Two forms of the FormProp example at run time

[Figure 2.4:](#) A representation of the structure of an object in memory, with a variable referring to it

[Figure 2.5:](#) A representation of the operation of assigning an object reference to another object. This is different from copying the actual content of an object to another.

[Figure 2.6:](#) The output of the NewDate program, with the name of the month and of the day depending on Windows regional settings

[Figure 2.7:](#) The output of the PolyAnimals example

[Figure 2.8:](#) The ErrorLog example and the log it produces

[Figure 2.9:](#) An example of the output of the ClassRef example

## Chapter 3: The Run-Time Library

[Figure 3.1:](#) The Rounding example, demonstrating banker's rounding and arithmetic rounding

[Figure 3.2:](#) The form of the VariantComp example at design time

[Figure 3.3:](#) The ConvDemo example at run time

[Figure 3.4:](#) The output of the EuroConv unit, showing the use of Delphi's conversion engine with a custom measurement unit

[Figure 3.5:](#) An example of the output of the FilesList application

[Figure 3.6:](#) The dialog box of the SelectDirectory procedure, displayed by the FilesList application

[Figure 3.7:](#) The output of the IfSender example

[Figure 3.8:](#) The output of the ClassInfo example

## Chapter 4: Core Library Classes

[Figure 4.1:](#) A graphical representation of the main groups of VCL components

[Figure 4.2:](#) The output of the RunProp example, which accesses properties by name at run time

[Figure 4.3:](#) In the ChangeOwner example, clicking the Change button moves the `Button1` component to the second form.

[Figure 4.4:](#) The list of dates shown by the ListDemo example

[Figure 4.5:](#) The textual description of a form component, displayed inside itself by the FormToText example

[Figure 4.6:](#) The ZCompress example can compress a file using the ZLib library.

## Chapter 5: Visual Controls

[Figure 5.1:](#) A comparison of the first three pages of the Component Palette for a CXL-based application (above) and a VCL-based application (below)

[Figure 5.2:](#) An application written with CLX can be directly recompiled under Linux with Kylix (displayed in the background).

[Figure 5.3:](#) The HtmlEdit example at run time: When you add new HTML text to the memo, you get an immediate preview.

[Figure 5.4:](#) The NameValues example uses the ValueListEditor component, which shows the name/value or key/value pairs of a string list, also visible in a plain memo.

[Figure 5.5:](#) Delphi's Menu Designer in action

[Figure 5.6:](#) The Edit Tab Order dialog box

[Figure 5.7:](#) The InFocus example at run time

[Figure 5.8](#): The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only proper use of the `Anchors` property.

[Figure 5.9](#): The Split1 example's splitter component determines the minimum size for each control on the form, even those not adjacent to the splitter.

[Figure 5.10](#): The ListBox control of the CustHint example shows a different hint, depending on which list item the mouse is over.

[Figure 5.11](#): The owner-draw menu of the ODMenu example

[Figure 5.12](#): Different examples of the output of a ListView component in the RefList program, obtained by changing the `ViewStyle` property and adding the check boxes

[Figure 5.13](#): The DragTree example after loading the data and expanding the branches

[Figure 5.14](#): The CustomNodes example has a tree view with node objects based on different custom classes, thanks to the `OnCreateNodes-Class` event.

## **Chapter 6: Building the User Interface**

[Figure 6.1](#): The first sheet of the PageControl of the Pages example, with its shortcut menu

[Figure 6.2](#): The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control.

[Figure 6.3](#): The interface of the bitmap viewer in the BmpViewer example. Notice the owner-draw tabs.

[Figure 6.4](#): The first page of the WizardUI example at design time

[Figure 6.5](#): The RichBar example's toolbar. Notice the drop-down menu.

[Figure 6.6](#): The StatusBar of the RichBar example displays a more detailed description than the fly-by hint.

[Figure 6.7](#): The StylesDemo program, a Windows application that currently has an unusual Motif layout

[Figure 6.8](#): The Pages example uses the current Windows XP theme, as it includes a manifest file (compare the figure with 6.1)

[Figure 6.9](#): The ActionList component editor, with a list of predefined actions you can use

[Figure 6.10](#): The ActionList editor of the Actions example

[Figure 6.11](#): The Actions example, with a detailed description of the `Sender` of an Action object's `OnExecute` event

[Figure 6.12](#): The MdEdit2 example at run time, while a user is rearranging the toolbars in the control bar

[Figure 6.13](#): The MdEdit2 example allows you to dock the toolbars (but not the menu) at the top or bottom of the form or to leave them floating.

[Figure 6.14](#): The DockTest example with three controls docked in the main form

[Figure 6.15](#): The main form of the DockPage example after a form has been docked to the page control on the left.

[Figure 6.16](#): The three pages of the ActionManager editor dialog box

[Figure 6.17:](#) Using the `CustomizeDlg` component, you can let a user customize the toolbars and the menu of an application by dragging items from the dialog box or moving them around in the action bars.

[Figure 6.18:](#) The `ActionManager` disables least-recently used menu items that you can still see by selecting the menu extension command.

[Figure 6.19:](#) The `ListActions` application has a toolbar hosting a static list and a virtual list.

## **Chapter 7: Working with Forms**

[Figure 7.1:](#) The dynamic form generated by the `DynaForm` example is completely created at run time, with no design-time support.

[Figure 7.2:](#) Sample forms with the various border styles, created by the `Borders` example

[Figure 7.3:](#) The `BIcons` example. By selecting the Help border icon and clicking the button, you get the help displayed in the figure.

[Figure 7.4:](#) The `NoTitle` example has no real caption but a fake one made with a label.

[Figure 7.5:](#) The `KPreview` program at design time

[Figure 7.6:](#) During a dragging operation, the `MouseOne` example uses a dotted line to indicate the final area of a rectangle.

[Figure 7.7:](#) The output of the `ColorKeyHole`, showing the effect of the new `TransparentColor` and `AlphaBlend` properties and the `Animate-Window` API

[Figure 7.8:](#) The output of the `Scroll1` example

[Figure 7.9:](#) The lines to draw on the virtual surface of the form

[Figure 7.10:](#) The Forms page of the Delphi Project Options dialog box

[Figure 7.11:](#) The dialog box of the `RefList2` example used in edit mode. Notice the `ComboBoxEx` graphical component in use.

[Figure 7.12:](#) The three forms (a main form and two dialog boxes) of the `DlgApply` example at run time

[Figure 7.13:](#) The Font selection dialog box with an Apply button

[Figure 7.14:](#) The main form of the `Splash` example, with the splash screen (this is the `Splash2` version)

## **Chapter 8: The Architecture of Delphi Applications**

[Figure 8.1:](#) The `ActivApp` example shows whether the application is active and which of the application's forms is active.

[Figure 8.2:](#) The output of the `Screen` example with some secondary forms

[Figure 8.3:](#) The `MdiDemo` program uses a series of predefined Delphi actions connected to a menu and a toolbar.

[Figure 8.4:](#) The output of the `MdiMulti` example, with a child window that displays circles



[Figure 8.5](#): The menu bar of the MdiMulti application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of Figure 8.4.

[Figure 8.6](#): The New Items dialog box allows you to create an inherited form.

[Figure 8.7](#): The two forms of the VFI example at run time

[Figure 8.8](#): The base-class form and the two inherited forms of the PoliForm example at design time

[Figure 8.9](#): A frame and two instances of it at design time, in the Frames2 example

[Figure 8.10](#): Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks.

[Figure 8.11](#): The first page of the FrameTab example at run time. The frame inside the tab is created at run time.

## **Chapter 9: Writing Delphi Components**

[Figure 9.1](#): The Package Editor

[Figure 9.2](#): The Project Options for packages

[Figure 9.3](#): The Contains section of the Package Editor shows both the units that are included in the package and the component resource files.

[Figure 9.4](#): The Object Inspector can automatically expand sub-components, showing their properties, as in the case of the `Timer` property of the `TMdClock` component.

[Figure 9.5](#): A component referencing an external label at design time

[Figure 9.6](#): The output of the Arrow component

[Figure 9.7](#): The output of the Arrow component with a thick pen and a special hatch brush

[Figure 9.8](#): The Arrow component defines a custom property category, `Arrow`, as you can see in the Object Inspector. Notice that properties can be visible in multiple section, such as the `Filled` property in this case.

[Figure 9.9](#): An example of the use of the `ActiveButton` component

[Figure 9.10](#): The `ListDialDemo` example shows the dialog box encapsulated in the `ListDial` component.

[Figure 9.11](#): The collection editor, with the Object `TreeView` and the Object Inspector for the collection item

[Figure 9.12](#): The list of sounds provides a hint for the user, who can also type in the property value or double-click to activate the editor (shown later, in Figure 9.13).

[Figure 9.13](#): The Sound Property Editor's form displays a list of available sounds and lets you load a file and hear the selected sound.

[Figure 9.14](#): The custom menu items added by the component editor of the `ListDialog` component

## **Chapter 10: Libraries and Packages**

[Figure 10.1](#): The output of the `CallCpp` example when you have clicked each of the buttons

[Figure 10.2](#): The output of the CallFrst example, which calls the DLL you've built in Delphi

[Figure 10.3](#): The Application page of the Project Options dialog box now has a Library Name section.

[Figure 10.4](#): If you run two copies of the UseMem program, you'll see that the global data in its DLL is not shared.

[Figure 10.5](#): The structure of the package hosting a form in Delphi's Package Editor

[Figure 10.6](#): The output of the PackInfo example, with details of the packages it uses

## **Chapter 11: Modeling and OOP Programming (with ModelMaker)**

[Figure 11.1](#): A class diagram in ModelMaker

[Figure 11.2](#): A class diagram with interfaces, classes, and interface delegation

[Figure 11.3](#): ModelMaker's Interface Wizard

[Figure 11.4](#): A sequence diagram for an event handler of the NewDate example

[Figure 11.5](#): The organizational possibilities of the Diagrams view

[Figure 11.6](#): The Property Editor dialog

[Figure 11.7](#): ModelMaker with the Implementation tab active

[Figure 11.8](#): ModelMaker's Difference view

[Figure 11.9](#): The Documentation tab of a Class Symbol

[Figure 11.10](#): ModelMaker's Code Template Parameters dialog box

## **Chapter 12: From COM to COM+**

[Figure 12.1](#): An example of the GUIDs generated by the NewGuid example. Values depend on my computer and the time I run this program.

[Figure 12.2](#): The COM Object Wizard

[Figure 12.3](#): The Word document is being created and composed by the WordTest Delphi application.

[Figure 12.4](#): The type-library editor, showing the details of an interface

[Figure 12.5](#): Delphi's Type Library Import dialog box.

[Figure 12.6](#): The second toolbar of the OleCont example (above) is replaced by the server's toolbar (below).

[Figure 12.7](#): The WordCont example shows how to use Automation with an embedded object.

[Figure 12.8](#): The WebDemo program after choosing a page that's well known by Delphi developers

[Figure 12.9](#): The XArrow ActiveX control and its property page, hosted by the Delphi environment

[Figure 12.10](#): The XClock control in the sample HTML page

[Figure 12.11](#): The New Trans-ational Object dialog box, used to create a COM+ object

[Figure 12.12](#): The newly installed COM+ component in a custom COM+ application (as shown by Microsoft's Component Services tool)

[Figure 12.13](#): A COM+ event with two subscriptions in the Component Services management console

[Figure 12.14](#): The NetImport program uses a .NET object to sum numbers.

## **Chapter 13: Delphi's Database Architecture**

[Figure 13.1](#): A sample local table active at design time in the Delphi IDE

[Figure 13.2](#): The XML display of a CDS file in the MyBase2 example. The table structure is defined by the program, which creates a file for the dataset on its first execution.

[Figure 13.3](#): The data-aware controls of the DbAware example at design time in Delphi

[Figure 13.4](#): The output of the CustLookup example, with the DBLookupCombo-Box showing multiple fields in its drop-down list

[Figure 13.5](#): The Fields Editor with the Add Fields dialog box

[Figure 13.6](#): The output of the FieldAcc example after the Center and Format buttons have been clicked

[Figure 13.7](#): The definition of a calculated field in the Calc example

[Figure 13.8](#): The output of the Calc example. Notice the Population Density calculated column and the ellipsis button displayed when you edit it.

[Figure 13.9](#): The output of the FieldLookup example, with the drop-down list inside the grid displaying values taken from another database table

[Figure 13.10](#): By handling the *OnGetText* and *OnSetText* events of a date field, the NullDates example displays specific output for null values.

[Figure 13.11](#): The output of the Total program, showing the total salaries of the employees

[Figure 13.12](#): The DrawData program displays a grid that includes the text of a memo field and the ubiquitous Borland fish.

[Figure 13.13](#): The MltGrid example has a DBGrid control that allows the selection of multiple rows.

[Figure 13.14](#): The output of the NonAware example in Browse mode. The program manually fetches the data every time the current record changes.

[Figure 13.15](#): In the SendToDb example, you can use a combo box to select the record you want to see.

[Figure 13.16](#): The CdsCalcs example demonstrates that by writing a little code, you can have the DBGrid control visually show the grouping defined in the ClientDataSet.

[Figure 13.17](#): The bottom portion of a ClientDataSet's Fields Editor displays aggregate fields.

[Figure 13.18](#): The MastDet example at run time

## **Chapter 14: Client/Server with dbExpress**

[Figure 14.1:](#) IBConsole lets you manage, from a single computer, InterBase databases hosted by multiple servers.

[Figure 14.2:](#) IBConsole can open separate windows to show you the details of each entity—in this case, a table.

[Figure 14.3:](#) IBConsole's Interactive SQL window lets you try in advance the queries you plan to include in your Delphi programs.

[Figure 14.4:](#) The dbExpress Connection Editor with the dbExpress Drivers Settings dialog box

[Figure 14.5:](#) The CommandText Editor used by the SQLDataSet component for queries

[Figure 14.6:](#) A sample log obtained by the SQLMonitor in the DbxSingle example

[Figure 14.7:](#) The SchemaTest example allows you to see a database's tables and the columns of a given table.

[Figure 14.8:](#) Editing a query component's collection of parameters

[Figure 14.9:](#) The ParQuery example at run time

[Figure 14.10:](#) The CdsDelta program displays the status of each record of a ClientDataSet.

[Figure 14.11:](#) The CdsDelta example allows you to see the temporary update requests stored in the `Delta` property of the ClientDataSet.

[Figure 14.12:](#) The Reconcile Error dialog provided by Delphi in the Object Repository and used by the CdsDelta example

[Figure 14.13:](#) The form of the TranSample application at design time. The radio buttons let you set different transaction isolation levels.

[Figure 14.14:](#) The output of the IbxUpdSql example

[Figure 14.15:](#) The output of the IbxMon example, based on the IBMonitor component

[Figure 14.16:](#) The server information displayed by the IbxMon application

[Figure 14.17:](#) The editor for the `GeneratorField` property of the IBX datasets

[Figure 14.18:](#) A form showing companies, office locations, and people (part of the RWBlocks example)

[Figure 14.19:](#) The RWBlocks example form for class registrations

[Figure 14.20:](#) The free query form of the RWBlocks example is intended for power users.

## **Chapter 15: Working with ADO**

[Figure 15.1:](#) Delphi's connection string editor

[Figure 15.2:](#) The first page of Microsoft's connection string editor

[Figure 15.3:](#) The OpenSchema example retrieves the primary keys of the database tables.

[Figure 15.4:](#) Setting extended properties

[Figure 15.5:](#) ABCCompany.xls in Delphi—a small tribute to Douglas Adams

[Figure 15.6:](#) The form of the DataClone example, with two copies of a dataset (the original and a clone)

## **Chapter 16: Multitier DataSnap Applications**

[Figure 16.1:](#) The Remote Data Module Wizard

[Figure 16.2:](#) When you activate a ClientDataSet component connected to a remote data module at design time, the data from the server becomes visible as usual.

[Figure 16.3:](#) The error message displayed by the ThinCli2 example when the employee ID is too large

[Figure 16.4:](#) The form of the ClientRefresh example, which automatically refreshes the active record and allows more extensive updates by clicking the buttons

[Figure 16.5:](#) The secondary form of the ThinPlus example, showing the data of a parametric query

[Figure 16.6:](#) The ThinPlus example shows how a dataset field can either be displayed in a grid in a floating window or extracted by a ClientDataSet and displayed in a second form. You'll generally do one of the two things, not both!

## **Chapter 17: Writing Database Components**

[Figure 17.1:](#) The data-aware ProgressBar in action in the DbProgr example

[Figure 17.2:](#) The DbTrack example's track bars let you enter data in a database table. The check box and buttons test the enabled status of the components.

[Figure 17.3:](#) The ViewGrid example demonstrates the output of the RecordView component, using Borland's sample BioLife database table.

[Figure 17.4:](#) An example of the MdDbGrid component at design time. Notice the output of the graphics and memo fields.

[Figure 17.5:](#) The structure of each buffer of the custom dataset, along with the various local fields referring to its subportions

[Figure 17.6:](#) The form of the StreamDSDemo example. The custom dataset has been activated, so you can already see the data at design time.

[Figure 17.7:](#) The output of the DirDemo example, which uses an unusual dataset that shows directory data

[Figure 17.8:](#) The ObjDataSet-Demo example showcases a dataset mapped to objects using RTTI.

## **Chapter 18: Reporting with Rave**

[Figure 18.1:](#) The Rave Designer with a simple report

[Figure 18.2:](#) The Rave Report Preview window for a report

[Figure 18.3:](#) After executing a Rave project, a user can choose the output format or rendering engine.

[Figure 18.4:](#) The RaveSingle report (generated with the help of a wizard) at design time

[Figure 18.5:](#) The Rave Designer's Data Text Editor

[Figure 18.6:](#) The master/detail report. The Band Style Editor appears in front of it.

[Figure 18.7:](#) The bold text in the report is determined at run time by a script.

## **Chapter 19: Internet Programming: Sockets and Indy**

[Figure 19.1:](#) The client program of the IndySock1 example

[Figure 19.2:](#) The client and server programs of the data-base socket example (IndyDbSock)

[Figure 19.3:](#) The SendList program at design time

[Figure 19.4:](#) The WebFind application can be used to search for a list of sites on the Google search engine.

[Figure 19.5:](#) The output of the BrowseFast text-only browser

[Figure 19.6:](#) The page displayed by connecting a browser to the custom HttpServ program

[Figure 19.7:](#) The output of the HtmlProd example, a simple demonstration of the Page-Producer component, when the user clicks the Demo Page button

[Figure 19.8:](#) The output of the HtmlProd example for the Print Line button

[Figure 19.9:](#) The editor of the DataSetTable-Producer component's `Columns` property provides you with a preview of the final HTML table (if the data-base table is active).

## **Chapter 20: Web Programming with WebBroker and WebSnap**

[Figure 20.1:](#) The output of the CgiDate application, as seen in a browser

[Figure 20.2:](#) A list of applications registered with the Web App Debugger is displayed when you hook to its home page.

[Figure 20.3:](#) The output corresponding to the table path of the BrokDemo example, which produces an HTML table with internal hyperlinks

[Figure 20.4:](#) The form action of the CustQueP example produces an HTML form with a selection component dynamically updated to reflect the current status of the database.

[Figure 20.5:](#) The WebSearch program shows the result of multiple searches done on Google.

[Figure 20.6:](#) The options offered by the New Web-Snap Application dialog box include the type of server and a button that lets you select the core application components.

[Figure 20.7:](#) The New WebSnap Page Module dialog box

[Figure 20.8:](#) The WSnap2 example features a plain script and a custom menu stored in an include file.

[Figure 20.9:](#) The Web Surface Designer for the inout page of the WSnap2 example, at design time

[Figure 20.10:](#) The page shown by the WSnapTable example at startup includes the initial portion of a paged table.

[Figure 20.11](#): The formview page shown by the WSnapTable example at design time, in the Web Surface Designer (or AdapterPageProducer editor)

[Figure 20.12](#): The WSnapMD example shows a master/detail structure and has some customized output.

[Figure 20.13](#): Two instances of the browser operate on two different sessions of the same WebSnap application.

## **Chapter 21: Web Programming with IntraWeb**

[Figure 21.1](#): The IWSimpleApp program in a browser

[Figure 21.2](#): The controller form of a stand-alone IntraWeb application

[Figure 21.3](#): The IWTree example features a menu, a tree view, and the dynamic creation of a memo component.

[Figure 21.4](#): The IWTwoForms example uses an IWGrid component, embedded text, and IWURL components.

[Figure 21.5](#): The IWSession application has both session-specific and global counters, as you can see by running two sessions in two different browsers (or even in the same browser).

[Figure 21.6](#): IntraWeb's HTML Layout Editor is a full-blown visual HTML editor.

[Figure 21.7](#): The data-aware grid of the IWScrollData example

[Figure 21.8](#): The main form of the IWGridDemo example uses a framed grid with hyperlinks to the secondary form.

[Figure 21.9](#): The secondary form of the IWGridDemo example allows a user to edit the data and navigate through records.

[Figure 21.10](#): The grid of the IWClientGrid example supports custom sorting and filtering without re-fetching the data on the web server.

## **Chapter 22: Using XML Technologies**

[Figure 22.1](#): The XmlEditOne example allows you to enter XML text in a memo, indicating errors as you type, and shows the result in the embedded browser.

[Figure 22.2](#): The XmlDomTree example can open a generic XML document and show it inside a TreeView common control.

[Figure 22.3](#): The DomCreate example can generate various types of XML documents using a DOM.

[Figure 22.4](#): The XML generated to describe the form of the DomCreate program. Notice (in the tree and in the memo text) that properties of class types are further expanded.

[Figure 22.5](#): Delphi's XML Data Binding Wizard can examine the structure of a document or a schema (or another document definition) to create a set of interfaces for simplified and direct access to the DOM data.

[Figure 22.6](#): The log produced by reading an XML document with the SAX in the Sax-Demo1 example

[Figure 22.7](#): The XML Mapper shows the two sides of a transformation to define a mapping between them (with the rules indicated in the central portion).

[Figure 22.8](#): The XmlMapping example uses a TransformProvider component to make a complex XML document available for editing within multiple ClientData-Set components.



[Figure 22.9:](#) The MapTable example generates an XML document from a database table using a custom transformation file.

[Figure 22.10:](#) The InetXPage-Producer editor allows you to build complex HTML forms visually, similarly to the AdapterPageProducer.

[Figure 22.11:](#) The leFirst application sends the browser some HTML components, an XML document, and the JavaScript code to show the data in the visual components.

[Figure 22.12:](#) The result of an XSLT transformation generated (even at design time) by the XSLPageProducer component in the XslCust example

[Figure 22.13:](#) The LargeXml example in action

## **Chapter 23: Web Services and SOAP**

[Figure 23.1:](#) The WSDL Import Wizard in action

[Figure 23.2:](#) An example of a translation from English to German obtained by Alta-Vista's BabelFish via a web service

[Figure 23.3:](#) The description of the Convert-Service web service provided by Delphi components

[Figure 23.4:](#) The ConvertCaller client of the Convert-Service web service shows how few German marks you used to get for so many Italian liras, before the euro changed everything.

[Figure 23.5:](#) The client program of the SoapEmployee web service example

[Figure 23.6:](#) The HTTP log of the Web App Debugger includes the low-level SOAP request.

[Figure 23.7:](#) The FishClient example receives a binary ClientDataSet within a SOAP attachment.

[Figure 23.8:](#) The UDDI Browser embedded in the Delphi IDE

[Figure 23.9:](#) The UddiInquiry example features a limited UDDI browser.

## **Chapter 24: The Microsoft .NET Architecture from the Delphi Perspective**

[Figure 24.1:](#) The HelloWorld demo as seen in ILDASM

[Figure 24.2:](#) The ILDASM output for the DestructorTest example

[Figure 24.3:](#) The ILDASM IL window for the `Destroy` destructor

[Figure 24.4:](#) .NET's Global Assembly Cache as seen in Windows Explorer

## **Chapter 25: Delphi for .NET Preview: The Language and the RTL**

[Figure 25.1:](#) The NetClassInfo example shows the base classes of a given component.

[Figure 25.2:](#) The CLRReflection example, with an assembly loaded

[Figure 25.3:](#) The `aspbase.aspx` example in a browser



Figure 25.4: The output of the `aspui.aspx` example, after typing in the edit box and clicking the button

Team Lib

◀ PREVIOUS

NEXT ▶

# List of Tables

## Chapter 1: Delphi 7 and Its IDE

[Table 1.1:](#) Delphi Project File Extensions

[Table 1.2:](#) Selected Delphi IDE Customization File Extensions

## Chapter 5: Visual Controls

[Table 5.1:](#) Names of Equivalent VCL and CLX Units

## Chapter 13: Delphi's Database Architecture

[Table 13.1:](#) The Dataset Fields in the DbAware Example

[Table 13.2:](#) The Subclasses of TField

## Chapter 15: Working with ADO

[Table 15.1:](#) OLE DB Providers Included with MDAC

[Table 15.2:](#) dbGo Components

# List of Listings

## Chapter 3: The Run-Time Library

[Listing 3.1](#): The definition of the *TObject* class (in the System RTL unit)

## Chapter 4: Core Library Classes

[Listing 4.1](#): The Definition of the *TPersistent* Class, from the Classes Unit

[Listing 4.2](#): The Public Portion of the Definition of the *TStream* Class

## Chapter 5: Visual Controls

[Listing 5.1](#): An XFM File (Left) and an Equivalent DFM File (Right)

## Chapter 6: Building the User Interface

[Listing 6.1](#): Key Portions of the DFM of the Pages Example

[Listing 6.2](#): A Sample Manifest File (pages.exe.manifest)

[Listing 6.3](#): The Actions of the Actions Example

## Chapter 8: The Architecture of Delphi Applications

[Listing 8.1](#): A Simple Unit for Testing Memory Leaks, from the ObjLeft Example

## Chapter 9: Writing Delphi Components

[Listing 9.1](#): Code of the *TMdFontCombo* Class, Generated by the Component Wizard

[Listing 9.2](#): A Component that Refers to an External Component Using an Interface

[Listing 9.3](#): The Classes for a Collection and Its Items

## Chapter 10: Libraries and Packages

[Listing 10.1](#): The *IntfColSel* Unit of the *IntfPack* Package

## Chapter 13: Delphi's Database Architecture

[Listing 13.1](#): The DFM File of the MyBase1 Sample Program

[Listing 13.2](#): The Public Interface of the *TDataSet* Class (Excerpted)

## Chapter 14: Client/Server with dbExpress

[Listing 14.1](#): The Core Method of the UniPrint Example

## **Chapter 16: Multitier DataSnap Applications**

[Listing 16.1](#): The Definition of the `IAppServer` Interface

## **Chapter 17: Writing Database Components**

[Listing 17.1](#): The `DrawCell` Method of the Custom RecordView Component

[Listing 17.2](#): The Declaration of `TMdCustomDataSet` and `TMdDataSetStream`

[Listing 17.3](#): The `Contrib.INI` File for the Demo Application

[Listing 17.4](#): The `InternallnitFieldDefs` Method of the Stream-Based Dataset

[Listing 17.5](#): The Complete Definition of the `TMdObjDataSet` Class

## **Chapter 19: Internet Programming: Sockets and Indy**

[Listing 19.1](#): The `TFindWebThread` Class (of the WebFind Program)

## **Chapter 20: Web Programming with WebBroker and WebSnap**

[Listing 20.1](#): The `menu.html` File Included in Each Page of the WSnap2 Example

[Listing 20.2](#): AdapterPageProducer Settings for the WSnapTable Main Page

[Listing 20.3](#): AdapterPageProducer Settings for the formview Page

## **Chapter 21: Web Programming with IntraWeb**

[Listing 21.1](#): Properties of the `IWDBGrid` in the `IWGridDemo` Example

## **Chapter 22: Using XML Technologies**

[Listing 22.1](#): The Sample XML Document Used by Examples in this Chapter

## **Chapter 24: The Microsoft .NET Architecture from the Delphi Perspective**

[Listing 24.1](#): The Project of the `DestructorTest` Example

[Listing 24.2](#): The Unit of the `DestructorTest` Example

## **Chapter 25: Delphi for .NET Preview: The Language and the RTL**

[Listing 25.1](#): The `ReflectionUnit` Unit of the `CLRReflection` Example

# List of Sidebars

## Chapter 1: Delphi 7 and Its IDE

[Drop-Down Fonts in the Object Inspector](#)

[The Empty Project Template](#)

[Installing New DLL Wizards](#)

## Chapter 2: The Delphi Programming Language

[Accessing Protected Data of Other Classes \(or, the "Protected Hack"\)](#)

[Debugging and Exceptions](#)

## Chapter 3: The Run-Time Library

[Executable Size under the Microscope](#)

[Simple Dragging in Delphi](#)

## Chapter 4: Core Library Classes

[Object Streaming versus Code Generation](#)

[Accessing Published Fields and Methods](#)

[Writing a Custom Stream Class](#)

## Chapter 5: Visual Controls

[From Qt to CLX](#)

## Chapter 6: Building the User Interface

[A Really Cool Toolbar](#)

## Chapter 7: Working with Forms

[Using Windows without a Mouse](#)

## Chapter 8: The Architecture of Delphi Applications

[Forms in Pages](#)

[INI Files and the Registry in Delphi](#)

## Chapter 9: Writing Delphi Components

[Installing the Components Created in This Chapter](#)

[Building Compound Components with Frames](#)

[Property-Naming Conventions](#)

## **Chapter 11: Modeling and OOP Programming (with ModelMaker)**

[Where's the VCL? Inheritance and Importing Code in ModelMaker](#)

[Design Patterns 101](#)

## **Chapter 12: From COM to COM+**

[COM Instancing and Threading Models](#)

[Interfaces, Variants, and Dispatch Interfaces: Testing the Speed Difference](#)

## **Chapter 13: Delphi's Database Architecture**

[A Data Module for Data-Access Components](#)

## **Chapter 14: Client/Server with dbExpress**

[OIDs to the Extreme](#)

[A Short History of InterBase](#)

[The TimeStamp Field Type](#)

## **Chapter 17: Writing Database Components**

[Replicable Data-Aware Controls](#)

## **Chapter 19: Internet Programming: Sockets and Indy**

[Internet Direct \(Indy\) Open Source Components](#)

## **Chapter 24: The Microsoft .NET Architecture from the Delphi Perspective**

[Issues with Overriding Finalize](#)

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

### **Introduction**

### **Chapter 1—The First Steps**

#### **DIRECTORY STRUCTURE**

#### **PRELIMINARY CODE**

##### **Getting Ready to Start to Begin**

##### **Starting to Begin**

##### **Begin**

#### **DEFINITIONS**

#### **IDENTIFIER REFERENCE**

#### **SUMMARY**

### **Chapter 2—Basics**

#### **VIEW PORT**

##### **Event Handler**

##### **Declaration**

##### **Analogy**

#### **ORTHOGRAPHIC PROJECTION**

##### **Meaning**

##### **Flexibility**

#### **DRAWING**

##### **Command Placement**

##### **State Variables**

##### **Drawing Mode**

## ADDITIONAL CONSIDERATIONS

Naming Convention

Error Handling in Depth

Order of Events

Matrices and Command Placement

View Port Manipulation

Mixing Windows Components

Pushing the (X,Y,Z) Limits

## DEFINITIONS

## IDENTIFIER REFERENCE

## SUMMARY

# Chapter 3—Introducing 3-D

## DEPTH

### CUBE

Square View Port

Distinguish Faces

Depth Test

Matrices Revisited

The ModelView Matrix

## ROTATION

Three Faces

glRotate

Action

## TWO-FACED POLYGONS

Open the Cube

Culling

## VERTEX DIRECTION

## TRANSLATION

Menu

Accumulation

More Clipping

## SCALE

Menu

Description

## ORDER OF TRANSFORMATIONS



A New Interface

Simultaneous Commands

DEFINITIONS

IDENTIFIER REFERENCE

SUMMARY

## Chapter 4—Perspective

ILLUSTRATION

COMMAND

Clipping Volume

Parameters

DEPTH

Previous Cube

Test

Better Cube

MOVEMENT

Rotation

Translation

MULTIPLE OBJECTS

Matrix Stack

Reuse an Object

Make a Scene

INDEPENDENT MOVEMENT

Translate

Rotate

Transformation Order Revisited

Another View

DEFINITIONS

IDENTIFIER REFERENCE

SUMMARY

## Chapter 5—Lighting

ENABLE LIGHTING

AMBIENT LIGHT

Definition

## Commands and Constants

### THE EASY WAY

### THE HARD WAY

### DIFFUSE LIGHT

#### Definition

#### Positional Lamps

#### Different Material Colors

#### Spotlights

#### Directional Spotlights

### EMISSION

#### Definition

#### Implementation

### SPECULAR LIGHT

#### Definition

#### Implementation

### HOW MUCH LIGHT

### DEFINITIONS

### IDENTIFIER REFERENCE

### SUMMARY

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

---

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Table of Contents](#)

# **Introduction**

## **ABOUT OPENGL**

### **What it Is and Is Not**

OpenGL is an Open Graphics Library. Although Silicon Graphics originally developed it for one of their own graphics workstations, it is not a proprietary system for a single platform. OpenGL is now an industry standardized library of graphics commands that enable programs to produce anything from simple lines to 3-D animation. The OpenGL Architecture Review Board works to insure that each implementation complies with the library's defined standard.

OpenGL is strictly about graphics. Unlike Direct X (for Windows) it has nothing to do with sound, joysticks, etc. It does one thing, graphics, and it does that one thing very well. It is not a resource hog; its two Windows libraries consume less than one megabyte of disk space. Although OpenGL gives the graphics programmer a rich and powerful set of commands, it provides a clean and simple Application Programming Interface (API). Just the little bit of programming work introduced in Chapter 1 is sufficient to produce some graphics effects. Most of that work is simply getting Windows to work with OpenGL. Very little effort with OpenGL commands themselves is required to produce some basic graphics.

### **The Windows Implementation**

In Windows, OpenGL is implemented in `opengl32.dll` for the main graphics commands, and `glu32.dll` for the utility library. As those file names imply, OpenGL is only available for 32-bit versions of Windows, such as NT and 95 or later. Windows NT, Windows 98, and OEM versions of Windows 95 (pre-installed by the computer manufacturer) include the OpenGL libraries, but a separately purchased copy of Windows 95 may require downloading the two DLL's from the Internet. Alternatively, you may obtain modules from the appropriate vendors, that take advantage of hardware support in newer video cards. Such hardware-supporting libraries may have names slightly different from the ones provided by

Microsoft.

The two Dynamic Link Libraries (DLL's) contain a powerful set of routines by which one can produce truly impressive three-dimensional graphics scenes and animations, especially with 16-bit palettes or better. The Windows implementation can work with 8-bit palettes, but that requires extra coding the book presents later. For now set the system to work with High Color (16-bit) or True Color (24-bit or 32-bit).

## ABOUT THE READER

The intended audience for this book is the Delphi developer interested in graphics programming. While the reader does not require expert level knowledge to benefit from this book, he or she should be familiar with Delphi's event-driven model, with Object Pascal's Unit structure, and with object-oriented programming. The reader should also be comfortable with Windows programming in general.

Not only should the reader be experienced using the above programming concepts, but should also be familiar enough with Delphi's IDE (Integrated Development Environment) to implement those concepts. The book asks, without detailed explanation, the reader to perform such tasks as generate specific event handlers or save a project to a new location with a different name.

The reader also needs to know enough about the user interface for a suitable 32-bit version of Windows to be able set the color depth to 16 or 24 bits. Of course, the reader must have direct access to Windows NT or 95 or later, as well as to a 32 bit version of Delphi.

Finally, the reader needs to have an interest in graphics programming. That's it! The reader does not need to start out with much graphics knowledge, just an interest.

If you have read this far without cringing, this book must be for you. No longer are you (the reader) treated as if you were off in another room being discussed behind your back. Congratulations! Welcome to the wonderful world of graphics programming with Delphi and OpenGL.

## ABOUT DELPHI

OpenGL produces various graphic images in a window, which in Delphi is a form. The place to perform the OpenGL commands is Delphi event handlers. Since the Windows version of OpenGL is a 32-bit DLL, use at least Delphi 2. Finally, Delphi 3 and 4 bring a required feature. They come with an interface unit for OpenGL and its utility library. Using OpenGL with Delphi requires a 32-bit version of Delphi, an OpenGL interface unit, and a working knowledge of developing Delphi applications.

### The Interface Unit

If your version of Delphi does not have an OpenGL interface unit, you can download one from the Internet. Wherever you obtain the interface unit, simply reference it in the **uses** clause of any unit that invokes OpenGL commands. Be sure to place the .DCU (Delphi Compiled Unit) file in Delphi's library path, otherwise give a full path to it in the .DPR (Delphi PProject) file. For example, in Delphi 3, the interface file is in the LIB directory under the DELPHI 3 directory. To see or modify the library path, go to the Library tab of the Environment Options window.

## Roll Your Own Interface

Actually, this book provides enough information about the OpenGL commands by which to construct an interface unit. A sample format of an interface unit follows. Notice the call in the initialization section that disables floating point exceptions. Keep in mind that Silicon Graphics, Inc., holds the copyright to the OpenGL libraries, and the required unit is an interface to their libraries.

```
unit OpenGL;  
  
interface  
  
uses  
    Windows;  
  
const  
    GL_COLOR_BUFFER_BIT = $4000;  
Type  
    HGLRC = THandle;  
    GLenum = cardinal;  
    GLbitfield = cardinal;  
    GLfloat = single;  
    GLclampf = single;  
  
{regular commands}  
procedure glClear(mask:GLbitfield); stdcall;  
procedure glClearColor(red,green,blue,alpha:GLclampf); stdcall  
function glGetError: GLenum; stdcall;  
  
{utility commands}  
  
implementation  
  
{regular commands}  
procedure glClear; external opengl32;  
procedure glClearColor; external opengl32;  
function glGetError; external opengl32;  
  
{utility commands}  
  
initialization  
    Set8087CW($133F);  
end.
```

## ABOUT THIS BOOK

### Style Convention

The type style used in this book is simple and straightforward. **Bold** type, **larger** font sizes, and **CAPITALIZATION** indicate levels of organization of the topics, in a fairly obvious way. *Italics* introduce new terms whose explanations soon follow. Code examples and

listings use a monospaced font, with certain key words, such as **procedure**, in bold face in a manner similar to the syntax highlighting used in the Delphi editor. The interface example above illustrates these techniques.

## Content

This is a hands-on, learn-by-doing book. For greatest benefit from the book place it near the computer, turn on the computer, and launch Delphi. Create a separate directory or folder in a suitable location for placing various OpenGL projects. Naming the directory “OPENGL” is a good choice.

Some people may prefer to develop a thorough background before they write their first line of code. I am not one of those people. While I do like to have a good understanding of the tools I am using and the tasks I am (hopefully) accomplishing, I also like to dive in and start doing something! I wrote this book the way someone like me would like to see books written.

Code writing begins as soon as possible, and explanation and theory appear as needed, building the concepts a little at a time. Each step has just enough explanation to use the program code under development, without a dumping of the whole load at once. Appendix A is the reference section for each OpenGL command.

This book generally takes each concept from the simple to the complex, making the basic steps obvious before obscuring them with a lot of detail. Do not be alarmed when some portions of code do not work very well. Sometimes the best way to show the reason for doing something a certain way is to take the direct approach first and, by the results, demonstrate why it must be a little different.

In short, this is a learning adventure, so fasten your seat belts, launch Delphi, and start!

[Table of Contents](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

# **Chapter 1**

## **The First Steps**

In an ideal world only a few lines of code would produce graphics right now. Alas, in the real world, some preliminary housekeeping is necessary. In this chapter you will learn how to make the necessary connection between Windows and OpenGL. The plan is to produce some graphics as soon as possible, so be patient with the necessary set up. In the spirit of getting to the interesting code as soon as possible, this chapter is fairly short and ends with actual graphics output. Although it is just a simple colored background, it shows that the code setting it up successfully did its job. Though trivial, this simple scene is not just a token reward for enduring a necessary evil. The code for producing it forms the basis for more advanced output.

## **DIRECTORY STRUCTURE**

To use this book effectively you will be writing code along the way and will create and save a number of projects, so pick a drive for keeping these OpenGL projects. Create a directory and name it OpenGL. Under that directory create subdirectories named Chapter.1, Chapter.2, etc. Some chapters may produce several projects, and these each have their own subdirectories under the chapter directory. You can, of course, organize in some other way, but the book proceeds as if you use the recommended directory structure and names for your projects, so you must translate as you go.

## **PRELIMINARY CODE**

### **Getting Ready to Start to Begin**

OpenGL is intended to be fairly platform-independent, rather than just for Windows. Therefore OpenGL needs a link to Windows, using some special structures and API extensions in Windows to provide this link. In Delphi a good place for the connection is within a form's OnCreate event handler.

Create a new project and save it in a new directory under the OpenGL directory created earlier. Name the new directory "Chapter.1," name the project "First.Dpr," and name the main unit "First1.Pas". Double-click the main form to set up the form's OnCreate event handler. Define a

variable of type `TPixelFormatDescriptor` and fill it in. Defining the pixel format with this structure permits describing some properties that the Windows GDI (graphics device interface) needs in order to work with OpenGL.

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    pfd: TPixelFormatDescriptor;  
    FormatIndex: integer;  
begin  
    fillchar(pfd, SizeOf(pfd), 0);  
    with pfd do  
        begin  
            nSize      := SizeOf(pfd);  
            nVersion   := 1; {The current version of the descriptor is 1}  
            dwFlags    := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL;  
            iPixelFormat := PFD_TYPE_RGBA;  
            cColorBits := 24; {support 24-bit color}  
            cDepthBits := 32; {depth of z-axis}  
            iLayerType := PFD_MAIN_PLANE;  
        end; {with}  
        FormatIndex := ChoosePixelFormat(Canvas.Handle, @pfd);  
end; {FormCreate}
```

Inform the system of the desired properties by calling `ChoosePixelFormat`, passing it form's device context and a pointer to the descriptor. Depending on what the device context can support, the contents of the descriptor may be altered to reflect the best approximation of the request. Windows uses Device Contexts, and OpenGL uses Rendering Contexts. This code attempts to map them to each other.

**DEVICE CONTEXT** The use of a device context is not obvious in this code. In Delphi, the `Canvas` property is a wrapper for a window's device context, and `Canvas.Handle` is the handle to the device context. This code uses native Delphi where possible, leaving out error checking and other details for clarity. Be patient for a while; the code improves later.

**DESCRIPTOR FIELDS** This descriptor has a number of fields that can remain zero or the equivalent, so explicit assignment statements were unnecessary. Here is a quick look at the rest of the contents of the descriptor to meet the current needs:

- 1.nSize.** Windows structures often require the size of the structure as part of the structure itself. This field follows that tradition.
- 2.nVersion.** The version number of the descriptor structure is 1, so a 1 must be stored here.
- 3.dwFlags.** The bits are set by or-ing together some pre-defined constants. `PFD_DRAW_TO_WINDOW` has an obvious meaning; you could be drawing to a bitmap in memory instead. `PFD_SUPPORT_OPENGL` is certainly a desired feature. Keep in mind that this code does not yet do anything with OpenGL; it just gets Windows ready for OpenGL. These are Windows structures and Windows API calls. Windows does not assume the code will work with OpenGL unless the code tells it. `ChoosePixelFormat` attempts to find a pixel format with the same flags set as those passed to it in the descriptor.
- 4.iPixelFormat.** Use RGBA (red, green, blue, alpha) pixels. Explanation of Alpha comes later.
- 5.cColorBits.** Support 24-bit color.
- 6.cDepthBits.** Set the depth of the z-axis to 32. Explanation of depth comes later.
- 7.iLayerType.** The current version only supports the main plane.

`ChoosePixelFormat` is a function that returns an integer, stored in `FormatIndex`. It returns zero to indicate an error, or a positive number as an index to the appropriate pixel format. `SetPixelFormat`



sets the pixel format of the device context, using that index. Now the bottom of the OnCreate event handler looks like this:

```
FormatIndex := ChoosePixelFormat(Canvas.Handle,@pfd);
SetPixelFormat(Canvas.Handle,FormatIndex,@pfd);
end; {FormCreate}
```

The functions receive the handle to the window's device context and change the pixel format of the window. So far none of the code produces visible results other than a standard blank form, but keep coding. OpenGL will shine forth soon.

## Starting to Begin

**RENDERING CONTEXT** Now that the code takes care of the pixel format, proceed to the rendering context. Add GLContext to the private section of the form:

```
type
  TForm1 = class (TForm)
    procedure FormCreate(Sender: TObject);
  private
    GLContext : HGLRC;
  public
    { Public declarations }
  end;
```

The HGLRC type is a handle to an OpenGL Rendering Context type. Here is its declaration for the interface unit:

```
type
  HGLRC = THandle;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

The new variable receives the result of `wglCreateContext`. This is one of several `wgl` (Windows-GL) functions for managing rendering contexts. Place this call at the bottom of the event handler:

```
FormatIndex := ChoosePixelFormat(Canvas.Handle, @pfd);
SetPixelFormat(Canvas.Handle, FormatIndex, @pfd);
GLContext := wglCreateContext(Canvas.Handle);
end; {FormCreate}
```

As the name implies, this function creates the OpenGL rendering context needed by the window. Now make it current:

```
FormatIndex := ChoosePixelFormat(Canvas.Handle, @pfd);
SetPixelFormat(Canvas.Handle, FormatIndex, @pfd);
GLContext := wglCreateContext(Canvas.Handle);
wglMakeCurrent(Canvas.Handle, GLContext);
end; {FormCreate}
```

**CLEAN UP** You should always put away your tools when finished with them, and you should always put away windows resources when finished with them. Go to the events page of the object inspector for the form. Double-click the `OnDestroy` event. Fill in the event handler as follows:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
    wglMakeCurrent(Canvas.Handle, 0);
    wglDeleteContext(GLContext);
end;
```

Passing a zero to `wglMakeCurrent` makes the previously current context no longer current. Now that `GLContext` is no longer current, delete it with `wglDeleteContext`.

## Begin

**USES CLAUSE** So far the code is nothing but Windows and Delphi code. The time has arrived for some OpenGL code. Add OpenGL to a **uses** clause. Put it in the interface section in order to use an OpenGL type in the interface.

**uses**

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, OpenGL, Menus;

**PAINT** Again go to the events page of the form's object inspector. This time double-click the OnPaint event. This event fires every time the window (form) needs repainting, such as when the form has been partially (or fully) covered and then exposed again. It also fires when the form first shows. Make this event handler like the following:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
    {background}
    glClearColor(0.0,0.4,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);

    {error checking}
    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
        raise Exception.Create('Error in Paint'#13+
            gluErrorString(errorCode));
end;
```

The given OpenGL commands have the following declaration:

```
procedure glClear(mask:GLbitfield); stdcall;
procedure glClearColor(red,green,blue,alpha:GLclampf); stdcall;
```

Their implementations are simply references to the DLL which contains the code and look like this:

```
procedure glClear; external opengl32;
procedure glClearColor; external opengl32;
```

**ERROR CHECKING** Notice this code introduces a little bit of error checking. This method is small enough (so far) that including the error checking code at this point does not obscure the main activity.

Use glClearColor to set the background color. Each color component ranges in floating point value from zero to one. In this case the color has no red, 0.4 units of green, and no blue. Set alpha to zero for now. The background color is a dark green (a little less than half the maximum intensity of green, which would be 1.0).

This is a good time to introduce an OpenGL type. Much of this graphics library, including color specifications, uses floating point numbers. To keep itself portable, OpenGL defines a type called GLfloat. Under Windows on Intel processors and compatibles, GLfloat is the same as a Delphi single, which is an IEEE (Institute for Electrical and Electronics Engineers) 32 bit floating point value. In the interface unit is a type declaration like this:

```
GLfloat = single;
```

Next call glClear and pass it a bit mask that tells it what to clear. This clears the buffers enabled for writing colors. For error checking this segment of code introduces the function glGetError, which appears often. It returns a number of type GLenum, which is an alias for Cardinal. The following type declaration appear in the interface unit:

```
GLenum = Cardinal;
```

Store the result in `errorCode`, which you should add to the private section of the form declaration.

```
ErrorCode: GLenum;
```

Compare the result to `GL_NO_ERROR`, a pre-defined constant of obvious meaning. If there is an error, raise an exception. The application handles this exception simply by displaying the message. The message includes another new function, `gluErrorString`, from the OpenGL utility library. It returns a pointer to some text (`PChar`). The text represents the meaning of the error number passed to the function. Delphi's string concatenation knows how to handle that `PChar`.

The constant has this declaration:

```
GL_NO_ERROR = 0;
```

Here is the declaration for the function:

```
function gluErrorString(errCode:GLenum):PChar; stdcall;
```

Since this function name begins with “glu,” its code is found in the other DLL:

```
function gluErrorString; external glu32;
```

Now, save the program (you have been warned!) and compile it. Do not run it from the Delphi IDE (Integrated Development Environment). OpenGL programs sometimes crash if launched from the IDE even though they work fine when run independently. That is why it was so important to save your work before starting, just in case you got stubborn and ran it from the IDE anyway. You might be lucky, but you might not. Use the Windows Run command, giving it the appropriate path. Behold! A dark green form!

**MORE DEVICE CONTEXT** Some of the code just written needs improving. The time has arrived to demonstrate why. Drag the form by its title bar left and right and up and down, so that parts are obscured and re-exposed repeatedly. Partially cover it with some other window, then uncover it. Do this a number of times until something strange happens. Eventually some part of the form fails to show green after exposure.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

While `Canvas.Handle` is the handle to the device context, it is not reliable for these purposes, because it is generated on the fly, when referenced, then released. So the device context matched up with the rendering context at one point may not even exist at another point. The rendering context would not then have a valid association. The application needs a handle it can control. In the private section of the form define `glDC`:

**type**

```
TForm1 = class (TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
private
    GLContext: HGLRC;
    glDC: HDC;
    errorCode: GLenum;
public
    { Public declarations }
end ;
```

In the `FormCreate` method (OnCreate handler), call `getDC` and replace all references to `Canvas.Handle` with `glDC`:

```
glDC := getDC(handle);
FormatIndex := ChoosePixelFormat(glDC, @pfd);
SetPixelFormat(glDC, FormatIndex, @pfd);
GLContext := wglCreateContext(glDC);
wglMakeCurrent(glDC, GLContext);
end; {FormCreate}
```

Save the code again, compile it, then run it (independently). Now it works much

more reliably.

**MORE ERROR HANDLING** In the FormCreate method is the basic setup required to use OpenGL under Windows. Now that the code has appeared in its simplicity, it is time to complicate it with error checking. It is not exciting, but all good software requires some kind of error handling or reporting. With these learning programs, if something fails, at least some kind of clue should appear. Add a boolean to the form's private declaration. As a field of a TObject descendant it is initialized to false. If you are not familiar with this initialization, see Delphi's online help for the InitInstance method under TObject.

**private**

```
    GLContext: HGLRC;  
    glDC: HDC;  
    errorCode: GLenum;  
    openGLReady: boolean;
```

In FormPaint test openGLReady because there is no point in calling OpenGL commands if they were not even set up.

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    if not openGLReady then  
        exit;  
    {background}  
    glClearColor(0.0,0.4,0.0,0.0);  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    {error checking}  
    error := glGetError;  
    if error<>GL_NO_ERROR then  
        raise Exception.Create('Error in Paint'#13+  
            gluErrorString(errorCode));  
end;
```

FormCreate also needs to test some function results. On failure call GetLastError which returns a Windows error code. If execution make it all the way to the end of the method, the reward is setting OpenGLReady to true.

```
    glDC := getDC(handle);  
    FormatIndex := ChoosePixelFormat(glDC,@pfd);  
    if FormatIndex=0 then  
        raise Exception.Create('ChoosePixelFormat failed '+  
            IntToStr(GetLastError));  
  
    if not SetPixelFormat(glDC,FormatIndex,@pfd) then  
        raise Exception.Create('SetPixelFormat failed '+  
            IntToStr(GetLastError));  
  
    GLContext := wglCreateContext(glDC);  
    if GLContext=0 then
```

```
raise Exception.Create('wglCreateContext failed '+
    IntToStr(GetLastError)');

if not wglMakeCurrent(glDC, GLContext) then
    raise Exception.Create('wglMakeCurrent failed '+
        IntToStr(GetLastError)');
OpenGLReady := true;
end; {FormCreate}
```

Be sure to save your work. The next chapter starts with this code as the foundation.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## **DEFINITIONS**

### **Term**

24-bit color

### **Meaning**

On typical PC video hardware colors have three components: red, green, and blue. Each color component can range in integer value from 0 (none) to 255 (maximum intensity), allowing an 8-bit (1-byte) integer to specify each intensity. You can specify 256\*256\*256 color combinations. These 16,777,216 possible colors are more than the human eye can distinguish. In a True Color or 24-bit color mode you can specify the color components directly, using at least a 24-bit number, but with a lesser "color depth" (number of bits to specify a color), the number you use to describe a color is just an index into a "palette" or pre-defined array of colors.

Background color

The color that shows through in a window when nothing else is drawn at a particular location. When a window is cleared, it is filled with the appropriate background color.

Current context

The active (currently being used) rendering context.

Device Context

A set of information about Windows drawing modes and

commands.



GDI	Graphics Device Interface. A standardized way of specifying graphics operations under Windows independently of the hardware.
Pixel	Picture element. The smallest unit of drawing. A dot.
Pixel format	Those characteristics of a device context needed for setting up a rendering context.
Rendering Context	A set of information about OpenGL states and commands under Windows.
RGBA	Red, Green, Blue, Alpha. OpenGL's method of specifying color. Each of the three color components are floating point values that range from 0.0 (none) to 1.0 (maximum intensity). The combination of the three "pure" colors produces the color intended for the viewer. The alpha component is reserved for a later chapter.

## IDENTIFIER REFERENCE

Identifier	Description
ChoosePixelFormat	A Windows function that receives a handle to a device context and a pointer to a pixel format descriptor. It returns an integer index to Windows' best attempt at a pixel format matching the specifications in the descriptor.
getDC	A Windows function that receives a handle to a window and returns a handle to that window's device context.
GetLastError	A Windows function that returns an integer representing the most recent error.
glClear	An OpenGL command that erases a buffer, filling it with the most recently specified background color. If the buffer is the color buffer, then the window is filled with the background color.
glClearColor	An OpenGL command that receives red, blue, green, and alpha floating point values sets the background color to be used by glClear.
GLenum	An OpenGL numeric type that maps to Delphi cardinal;
GLfloat	An OpenGL numeric type that maps to Delphi single.

glGetError	An OpenGL function that returns an integer value representing an error flag set by an OpenGL command, and clears that flag. If no error flags are set when called, the function returns the value of GL_NO_ERROR.
gluErrorString	An OpenGL utility function that receives an error number (such as returned by glGetError) and returns a pointer to a human-readable string (null-terminated) describing the associated OpenGL error flag.
GL_NO_ERROR	An OpenGL constant to represent a successful command.
HDC	A Windows type. Handle to a device context.
HGLRC	A Windows/OpenGL type. Handle to a rendering context.
SetPixelFormat	A Windows function that specifies the pixel format to use with the given device context. While ChoosePixelFormat merely obtains an index to a pixel format, this function sets that pixel format as the one to use. The function receives a handle to the device context, an index to the desired pixel format (such as the one returned by ChoosePixelFormat) and a pointer to the pixel format descriptor (as modified by ChoosePixelFormat). It returns true on success or false on failure. Call GetLastError to learn more about the failure.
TPixelFormatDescriptor	A Windows type. A structure of this type contains the fields needed to select and set a pixel format.
wglCreateContext	A Windows function that creates a rendering context for the specified device context. It receives a handle to the device context and returns a handle to the rendering context.
wglDeleteContext	A Windows function that deletes the specified rendering context. It receives the handle to the rendering context and returns true on success and false on failure.
wglMakeCurrent	A Windows function that makes the specified rendering context current (the one available for use) with the specified device context. It receives the handle to the device context and the handle to the rendering context and returns true on success and false on failure.

## SUMMARY

Chapter 1 showed how to:

1. Link the Windows graphics device context with the OpenGL rendering context.
2. Use actual OpenGL commands to fill a window with a color.

### 3. Perform some basic error checking with OpenGL commands.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 2 Basics

OpenGL majors in three-dimensional graphics, but computer monitors operate with essentially flat, two-dimensional screens. Representing a three-dimensional space on a two-dimensional surface requires *projection*. Each point in the space projects mathematically to an appropriate point on the surface. This chapter introduces OpenGL commands to identify the part of the window to receive the projection, to set up the projection, and to draw an object. Included with the projection is a *clipping* volume, outside of which no drawing happens. These foundational concepts and commands permit drawing something besides a mere background. This chapter also brings another Delphi event handler into use.

Use the first chapter's final code as the starting point in this chapter. If you have not already done so, create a new directory under the "OPENGL" directory and name it "Chapter.2." In Delphi load the First.Dpr project and do a File|Save Project As in the new directory, naming it "Rect.Dpr." Do File|Save As for the "First1.Pas" unit. Name it "Rect1.Pas" in the new directory.

### VIEW PORT

#### Event Handler

Drawing a green background is a wonderful accomplishment, but by itself, a green background is not very exciting. Drawing a shape ON the background is much more interesting. For now, drawing a rectangle is the next great advance in OpenGL knowledge. The place for drawing is the form's OnPaint event handler, but setting up the drawing area also needs the OnResize handler. Go to the events page of the form's object inspector and double-click OnResize. In the event handler place a call to glViewport and a call to glOrtho, and a modicum of error checking:

```
procedure TForm1.FormResize(Sender: TObject);  
begin  
    if not openGLReady then  
        exit;  
    glViewport(0,0,ClientWidth,ClientHeight);  
    glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);  
    errorCode := glGetError;  
    if errorCode<>GL_NO_ERROR then  
        raise Exception.Create('FormResize:'+gluErrorString(errorCode));  
end
```

## Declaration

Remember that OpenGL is fairly platform independent, so it does not inherently know anything about the coordinate system of the operating environment. Use `glViewport` to make a connection between the coordinates used by the windowing system (Microsoft Windows) and the coordinates used by OpenGL. Its declaration looks like this:

```
procedure glViewport(x,y:GLint; width,height:GLsizei); stdcall;
```

`GLint` and `GLsizei` both map to integer. The meaning of `width` and `height` are as expected, but `x` and `y` need more attention. Windows measures from the upper left corner of a region, but OpenGL measures from the lower left, as if it were relating to the first quadrant of a set of Cartesian coordinates. The main thing to note is that the `y`-direction runs opposite to the familiar Windows coordinates, but fits more naturally with coordinates commonly used in mathematics.

If you are building your own interface unit, the Listing 2-1 encompasses the declarations used thus far, plus a few more that are just waiting in the wings. Add other declarations when they appear.

**Listing 2-1.** Beginning of an Interface Unit.

```
unit OpenGL;  
  
interface  
  
uses  
    Windows;  
  
const  
    GL_COLOR_BUFFER_BIT = $4000;  
    GL_NO_ERROR = 0;  
Type  
    HGLRC = THandle;  
    GLenum = cardinal;  
    GLbitfield = cardinal;  
    GLfloat = single;  
    GLclampf = single;  
    GLint = integer;  
    GLsizei = integer;  
  
    {regular commands}  
procedure glClear(mask:GLbitfield); stdcall;  
procedure glClearColor(red,green,blue,alpha:GLclampf); stdcall;  
function glGetError: GLenum; stdcall;  
procedure glViewport(x,y:GLint; width,height: GLsizei); stdcall;  
  
    {utility commands}  
function gluErrorString(errCode:GLenum):PChar; stdcall;  
  
implementation  
  
    {regular commands}  
procedure glClear; external opengl32;  
procedure glClearColor; external opengl32;  
function glGetError; external opengl32;  
procedure glViewport; external opengl32;  
  
    {utility commands}
```

```
function gluErrorString; external glu32;
```

### **initialization**

```
    Set8087CW($133F);  
end.
```

## **Analogy**

Think of the form as a wall. With `glViewport` cut a hole in the wall through which to view a scene. In this case, cut a hole the size of the entire client area of the form, so all that is left is the frame. A smaller “opening” is possible:

```
glViewport(50,30,80,70);
```

The above example starts at 50 pixels from the left side of the form and 30 pixels up from the bottom of the form. It is 80 pixels wide and 70 pixels tall (going up from the bottom).

It may not be a good idea to specify such absolute coordinates without specifying a non-resizable border or limiting the minimum window size.

One side note: `glViewport` limits the portion of the window in which the scene is *rendered*, but does not restrict the area in which the background is cleared. In the wall analogy think of the background color as something sprayed onto the stage for the scene. Not only does the spray go through the hole in the wall, but the spray also covers the entire wall! There are remedies for this problem, but that subject comes later.

In simplest terms rendering is drawing with OpenGL. To render a scene is to use OpenGL constructs and commands to cause pixels to illuminate in a pattern that represents the scene envisioned by the programmer.

## **ORTHOGRAPHIC PROJECTION**

### **Meaning**

The call to `glOrtho` defines a clipping volume, which is a region in space in which objects can be seen. Any part of a scene that lies outside of the region is not visible (clipped). Of course, though the rendered scenes may be conceptually three-dimensional, they must be projected onto a two-dimensional screen. The available projections are orthographic projection and perspective projection. You get one guess as to which kind `glOrtho` uses. In orthographic projection, also known as parallel projection, an object appears the same size whether it is near or far from the viewer. Showing a simple, flat rectangle certainly does not need anything more complicated. Here is the declaration:

```
procedure glOrtho(left,right,bottom,top,zNear,zFar: GLdouble); stdcall;
```

`GLdouble` is the same thing as a Delphi double, an IEEE 64 bit floating point value.

```
GLdouble = double;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Left, right, bottom, and top describe the same area as the `glViewport`, but with programmer-defined values for the edges. The rendering commands use values relative to these. In the above `FormResize` method, `-1.0` is the new value given to the left side of the view port, and `1.0` is the value for the right side. The bottom of the view port is now `-1.0`, and the top is `1.0`. The `z`-values represent directions perpendicular to the screen. The `zNear` value represents the `z`-axis coordinate of the clipping region face nearest the viewer, while `zFar` designates the face farthest from the viewer, deepest into the screen. Those values represent in 3-dimensional space whatever the programmer wants them to mean.

### Flexibility

Here is another example of `glOrtho` with `glViewport`:

```
glViewport(0,0,ClientWidth,ClientHeight);  
glOrtho(3.204,17.2,-11.0,1.45,5.025,41.0);
```

The call to `glViewport` remains the same to isolate the effect, in this example, of changing the values given to `glOrtho`. For example, the clipping region now has a value of `17.2` for its right side. It is still located at the same place as before, which is the right side of the view port, at `ClientWidth`. The value of `17.2` was just convenient. Delphi's Object Pascal language has a similar concept for arrays:

```
var  
  first: array[0..9] of integer;  
  second: array[1..10] of integer;  
  third: array[100..109] of integer;
```

All three of the above variables are equivalent arrays of ten integers, but with different starting points. Why not just use zero-based arrays? After all, that is

all that is available in some programming languages. The answer is that they are declared in a manner convenient for the programmer. An alternate form may allow less typing in the coding of an algorithm, or may more obviously represent some real world concept.

Similarly the programmer has the freedom to assign arbitrary values to the orthographic clipping region. He may choose values for computational convenience or for better conceptual representation of the scene. There is a little bit of order imposed on the glOrtho universe, however. Negative z-values are in front of the view port in the window, and positive z-values are behind the view port. Placing the clipping region entirely in front of the view port strains the hole-in-the-wall analogy, but the comparison has already served its purpose.

## DRAWING

### Command Placement

At last the time has arrived to draw something: a rectangle. As expected, the place to draw the rectangle is in the OnPaint event handler. The drawing commands need to appear between glBegin and glEnd; The latter requires no parameters; the former takes a parameter to set the drawing mode:

```
procedure glBegin(mode:GLenum); stdcall;  
procedure glEnd; stdcall;
```

Between glBegin and glEnd goes a list of vertices, and possibly a few commands related to the vertices. Each vertex is simply a point in space specified by the glVertex command. This command takes many forms, but only a few of them are appropriate here:

```
procedure glVertex2f (x,y: GLfloat); stdcall;  
procedure glVertex3f (x,y,z: GLfloat); stdcall;  
procedure glVertex4f (x,y,z,w: GLfloat); stdcall;
```

### State Variables

In the FormPaint method (Listing 2-2), not only does glClearColor set the clear color (background), but glColor sets the foreground color for drawing. These calls set state variables which remain the same until similar calls explicitly change them. If the command says to draw in blue, all drawing is done in blue until another call changes the state. OpenGL's state variables are in DLLs, not directly accessible to the programmer. You read and write OpenGL's state variables via function and procedure calls. The glColor command is before glBegin in the current event handler method, but it is one of the limited set of commands that can appear between glBegin and glEnd, since color changes are often necessary in the midst of a drawing sequence.

**Listing 2-2.** Setting State Variables in FormPaint.

```
procedure TForm1.FormPaint(Sender: TObject);
```



```

begin
  if not openGLReady then
    exit;
    {green background}
    glClearColor(0.0,0.4,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);

    {red rectangle}
    glColor3f(0.5,0.0,0.0);
    glBegin(GL_POLYGON);
      glVertex2f(0.25,0.25);
      glVertex2f(0.75,0.25);
      glVertex2f(0.75,0.75);
      glVertex2f(0.25,0.75);
    glEnd;
    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
      raise Exception.Create('Error in Paint'#13+
        gluErrorString(errorCode));
end;

```

Here is the declaration of the glColor command used in the FormPaint method:

```
procedure glColor3f(red,green,blue:GLfloat); stdcall;
```

The new constant has this declaration:

```
GL_POLYGON = 9;
```

Since glColor3f has arguments of 0.5, 0.0, and 0.0, OpenGL is drawing in half-intensity red. Alpha is not specified, so it gets 0.0 also. In the calls to glVertex2f, no z-value is passed, so all of the rectangle lies in the z = 0.0 plane. That means the rectangle lies in the same plane as the view port, neither in front nor behind. Recall that glOrtho received arguments of -1.0 for left and 1.0 for right. Since the x-values in glVertex are all positive, the rectangle is entirely on the right side of the viewing area. The same is true of the y-values, so the rectangle lies entirely in the upper half of the viewing area. Remember that all the drawing coordinates are relative to the values defining the clipping region, so changing the arguments to glOrtho could affect the size, shape, and position of the drawing.

## Drawing Mode

```
procedure glBegin(mode:GLenum); stdcall;
```

Note the argument passed to glBegin. GL\_POLYGON informs OpenGL that the subsequent vertex list establishes the corners of a polygon. The number of vertices between glEnd and glBegin are the number of vertices in the polygon,

which in this case is a quadrilateral. The first side of the polygon is a line from the first vertex to the second. The second side of the polygon is a line from the second vertex to the third, and so on until the polygon closes with a line from the last vertex to the first. OpenGL knows to do all that simply by being given `GL_POLYGON`. It further knows to fill in the interior of the polygon with the color previously specified in `glColor`. What if a complete scene needs more than one type of figure? Make multiple `glBegin`, `glEnd` pairs, passing the appropriate mode arguments.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

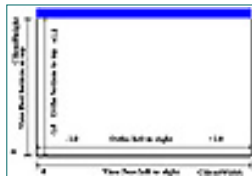
Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Another new command introduced here is `glFlush`. For the sake of efficiency, the graphics engine does not necessarily start drawing as soon as it receives its first command. It may continue to accumulate information in a buffer until given a little kick. The call to `glFlush` forces it to start drawing.

```
procedure glFlush; stdcall;
```

To illustrate another drawing mode, `GL_LINES`, Listing 2-3 is the code for the `OnPaint` event handler that produced Figure 2-1, including a couple of new commands needed for that purpose.



**Figure 2-1.** `glViewport` and `glOrtho`

**Listing 2-3.** Using `GL_LINES`.

```
{white background}
glClearColor(1.0,1.0,1.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);

{top of the form}
glColor3f(0.0,0.0,1.0);
glRectf(-300.0,300.0,300.0,345.0);

glLineWidth(2.0);
glColor3f(0.0,0.0,0.0);
glBegin(GL_LINES);
    {left side}
```

```

    glVertex2f(-300.0,-300.0);
    glVertex2f(-300.0,300.0);

    {bottom}
    glVertex2f(-300.0,-300.0);
    glVertex2f(300.0,-300.0);

    {right side}
    glVertex2f(300.0,-300.0);
    glVertex2f(300.0,300.0);
glEnd;

glLineWidth(1.0);
glBegin(GL_LINES);
    {view port measure left to right}
    glVertex2f(-300.0,-340.0); {line}
    glVertex2f(300.0,-340.0);

    glVertex2f(-300.0,-320.0); {end}
    glVertex2f(-300.0,-360.0);

    glVertex2f(-300.0,-340.0); {end}
    glVertex2f(-290.0,-330.0);

    glVertex2f(-300.0,-340.0); {arrow}
    glVertex2f(-290.0,-350.0);

    glVertex2f(300.0,-320.0); {arrow}
    glVertex2f(300.0,-360.0);

    glVertex2f(300.0,-340.0); {arrow}
    glVertex2f(290.0,-330.0);

    glVertex2f(300.0,-340.0); {arrow}
    glVertex2f(290.0,-350.0);

    {view port measure bottom to top}
    glVertex2f(-325.0,-300.0); {line}
    glVertex2f(-325.0,300.0);

    glVertex2f(-312.0,-300.0); {end}
    glVertex2f(-337.0,-300.0);

    glVertex2f(-312.0,300.0); {end}
    glVertex2f(-337.0,300.0);

    glVertex2f(-325.0,-300.0); {arrow}
    glVertex2f(-332.0,-285.0);

    glVertex2f(-325.0,-300.0); {arrow}

```

```

    glVertex2f(-318.0,-285.0);

    glVertex2f(-325.0,300.0); {arrow}
    glVertex2f(-332.0,285.0);

    glVertex2f(-325.0,300.0); {arrow}
    glVertex2f(-318.0,285.0);

    {ortho measure left to right}
    glVertex2f(-300.0,-260.0); {line}
    glVertex2f(300.0,-260.0);

    glVertex2f(-300.0,-260.0); {arrow}
    glVertex2f(-288.0,-272.0);

    glVertex2f(-300.0,-260.0); {arrow}
    glVertex2f(-290.0,-250.0);

    glVertex2f(300.0,-260.0); {arrow}
    glVertex2f(290.0,-250.0);

    glVertex2f(300.0,-260.0); {arrow}
    glVertex2f(288.0,-272.0);

    {ortho measure bottom to top}
    glVertex2f(-275.0,-300.0); {line}
    glVertex2f(-275.0,300.0);

    glVertex2f(-275.0,-300.0); {arrow}
    glVertex2f(-282.0,-285.0);

    glVertex2f(-275.0,-300.0); {arrow}
    glVertex2f(-268.0,-285.0);

    glVertex2f(-275.0,300.0); {arrow}
    glVertex2f(-282.0,285.0);

    glVertex2f(-275.0,300.0); {arrow}
    glVertex2f(-268.0,285.0);
glEnd;

```

When the drawing mode is `GL_LINES`, each pair of calls to `glVertex` defines a line segment. The declaration for the new mode is:

```
GL_LINES = 1;
```

Rectangles are so common that OpenGL has a special command just for drawing them. The first two parameters define one vertex, and the next two define the opposite vertex. The rectangle assumes  $z = 0$ . The float version just introduced is one of many variations. Also introduced is a command for setting the width of the lines in pixels. The default width is 1.0.

```
procedure glRectf(x1,y1,x2,y2:GLfloat); stdcall;  
procedure glLineWidth(width:GLfloat); stdcall;
```

## ADDITIONAL CONSIDERATIONS

### Naming Convention

You may have noticed by now that all the OpenGL command names begin with “gl.” The graphics library has a well-defined naming convention, which this book mentions from time to time. So far the “gl” prefix has dominated, which does apply to all OpenGL commands. A “glu” prefix has also made a cameo appearance. It is the standard for all commands of the utility library. Chapter 1 presented the “wgl” prefix for the extensions to Microsoft Windows that support OpenGL rendering. Similarly the X Window System (for Unix) extensions use a “glx” prefix, and OS/2 Presentation Manager extensions use “pgl.”

The naming convention does not stop with prefixes. The glVertex commands, for example, have a number suffix, such as 2, 3, or 4, indicating the number of parameters the command requires. A later chapter explains the fourth (w) parameter, but for now, if it receives no explicit value, it receives a value of 1.0. In the two-parameter version, even the z-value goes without assignment, so it gets a 0.0 value by default. The “f” suffix indicates these versions of glVertex take GLfloat parameters. Other parameter types are possible, so as many as 24 different versions of the glVertex command are available.

The FormPaint method used the glColor3f command. The “3” indicates that this version of the command takes three parameters, but there is another version of the command that takes four parameters. The “f” indicates that the parameters are of type GLfloat.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

### Error Handling in Depth

Chapter 1 introduced `glGetError`, but gave it only superficial treatment. Checking it every time a command could possibly produce an error would have a performance penalty. Notice that `FormPaint` performs a single call to `glGetError` at the end of the method. Fortunately the graphics library is capable of storing multiple error codes. If several error codes have accumulated, multiple calls to `glGetError` are required to clear them all. Accordingly this application replaces its default exception handler. Place this at the end of the private section of the form's type declaration:

```
procedure ExceptionGL(Sender:TObject;E:Exception);
```

The application's `OnException` event handler activates whenever an exception climbs all the way up the stack without being handled along the way. The default handler simply displays the corresponding error message. To replace the default one with one more suitable for OpenGL, place this at the beginning of the `FormCreate` method:

```
Application.OnException := ExceptionGL;
```

Finally, here is the body of the new exception handler. By calling `ShowException` it does what the old exception handler would have done, then in the loop it calls `glGetError` and shows all the pending OpenGL errors until they have all been cleared:

```
procedure TForm1.ExceptionGL(Sender:TObject;E:Exception);  
begin  
  ShowException(Sender,E);  
  repeat  
    errorCode := glGetError;  
    if errorCode<>GL_NO_ERROR then
```

```
        showMessage(gluErrorString(errorCode));  
    until errorCode=GL_NO_ERROR;  
end
```

## Order of Events

Now, compile the program and launch it with the Windows “Run” command. Resize it several ways. Watch the shape of the rectangle as you do the resizing. Careful observation reveals that if the form only shrinks, the shape of the rectangle does not change. If the form enlarges either horizontally or vertically or both, the shape of the rectangle changes appropriately. An expected event is not firing. This calls for further investigation.

In the variable declaration section add:

```
    EventFile: TextFile;
```

So that an “Event.Log” text file opens for writing at program start, and closes at program end, make the end of the unit look like this:

```
initialization  
    assignFile(EventFile, 'Event.Log');  
    rewrite(EventFile);  
finalization  
    closeFile(EventFile);  
end.
```

To record each firing of the OnPaint event, at the beginning of FormPaint add the following:

```
    writeln(EventFile, 'Paint  '+IntToStr(TimeGetTime));
```

TimeGetTime is a Windows API call that returns the number of milliseconds since Windows last loaded. To access it the unit needs MMSystem in the uses clause. Now, to log each firing of the OnResize event, at the beginning of FormResize add this line:

```
    writeln(EventFile, 'Resize '+IntToStr(TimeGetTime));
```

Now compile and run as before. Wait about one second. Shrink the form from the right side. Wait another second. Shrink the form from the bottom. Wait another second. Enlarge the form a little at the bottom. Now close the form. Examine the contents of the Event.Log file in the current (project) directory. At the top of the file are “Resize” and “Paint” with nearly identical time stamps, only 10 or 20 milliseconds apart. They fired at program start. About 1000 milliseconds later (depending on the accuracy of the “one second” wait) is another “Resize” line. That fired when the form shrank from the right side. Another 1000 milliseconds later is another “Resize” entry, which fired when the form shrank from the bottom. Still 1000 milliseconds later is another “Resize” followed shortly by a “Paint.”

After careful consideration, this makes sense. When the form shrinks, Windows only sends a Resize message. Why should Windows send a Paint message just for the



covering of some already painted territory? Only form enlargement in one or both directions exposes new area that needs painting. Windows does not know about the special need to adjust the interior of the form. Since the Windows Resize and Paint messages are what trigger the Delphi OnResize and OnPaint events, a form shrinkage does not trigger the necessary adjustment of the rectangle.

The way to remedy this little problem is to call FormPaint on those occasions when it would otherwise be neglected. FormResize is called for every OnResize event, so that is a good place to check whether either dimension enlarged. If not, then no OnPaint event happens, so you must call FormPaint directly. After cleaning out the text file code, add these fields to the private section of the form type declaration:

```
    oldw,  
    oldh: integer;
```

The fields preserve the previous width and height for comparison with new values. Initialize them at the end of the FormCreate method:

```
    yldw := ClientWidth;  
    oldh := ClientHeight;
```

Finally, this code at the end of FormResize insures the FormPaint method always fires when needed:

```
    if (ClientWidth<=oldw) and (ClientHeight<=oldh) then  
        FormPaint(Sender);  
    oldh := ClientHeight;  
    oldw := ClientWidth;
```

Save, compile and run (from Windows). Resize to your heart's content.

## Matrices and Command Placement

At this point the program is ready for a little experimentation with glOrtho. Change left from -1.0 to 0.0, then save, compile and run as before.

```
    glOrtho(0.0,1.0,-1.0,1.0,-1.0,1.0);
```

Remember that the values given to the vertices of the rectangle are relative to the clipping volume. Since the range of horizontal numbers in the call to glOrtho is half the former range, the rectangle should be twice as wide as before. Program execution shows that to be the case. Now, resize the form exactly one time. Something is wrong! The rectangle jumps to the left side. Resize the form again. The rectangle is gone, never to be seen again!

This experience exposes another important concept. OpenGL relies heavily on *matrices*. A matrix is a two-dimensional array of numbers for which a special set of mathematical operations apply. For every call to glOrtho, OpenGL creates a matrix from the parameters and multiplies it by the projection matrix. Each successive call changes the projection matrix. The symmetrical numbers passed to glOrtho just masked the potential strange behavior, but the unbalanced numbers for this version exposed the problem. One solution is to move the call to glOrtho to the last line of

the FormCreate method. Save, compile and run. Resize the form a few times. Now the expected behavior manifests itself.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Go ahead and experiment some more with passing different values to `glOrtho`, leaving everything else the same. These variations should give a sense of the relationship between the values in the projection matrix and the values of the vertices.

### View Port Manipulation

The view port is the next experimental subject. The orthographic projection can change some of its boundary numbers, but still have the same boundary (the view port). Just the arbitrary meaning of the numbers change. The view port boundary numbers have a real meaning, which is window coordinates. Changing these numbers changes the actual boundary of the view port.

Restore the call to `glOrtho` to its symmetrical arrangement, but leave it at the bottom of `FormCreate`.

```
glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);  
end; {FormCreate}
```

In `FormResize` change the vertical extent in the call to `glViewport`.

```
glViewport(0,0,ClientWidth,ClientHeight div 2);
```

Compile and run as before. From this point on using the Windows Run command rather than running from the IDE is taken for granted and needs no further mention. Now the view port only covers the lower half of the window. Remember that OpenGL coordinates normally start from the bottom of the window, even when using “window coordinates.” The red rectangle is still in the upper right quadrant of the view port, but since the view port itself is confined to the lower half of the window, the rectangle is now in the lower half of the window. Since the view port is only half as tall, the rectangle is half as tall as it was.

Now, try it the other way.

```
glViewport(0,0,ClientWidth div 2,ClientHeight);
```

Compile and run. This time everything lies on the left side of the screen. The rectangle is still in the right side of the view port, but it lies in the left side of the window. Notice that with these restrictions on the size of the view port, the background color still covers the whole window. This experiment should be enough to give a good feel for glViewport, so take out the **div** operators to restore the original appearance.

## Mixing Windows Components

One of the really neat features of OpenGL is that it does not prevent using regular Windows components. This opens many possibilities for user interaction with your graphics constructs. The coming chapters will explore a number of different ways users can give information to your applications. Since these are illustrations and exercises, they are not the ultimate in slick, state-of-the-art front ends; they are just enough to demonstrate a technique. For the first such demonstration, drop a MainMenu component on the form. Right-Click the component and select the Menu Designer. Give it one menu item with a caption of “&Z-Value.” Close the Menu Designer, click the menu item, and edit the event handler as follows:

```
procedure TForm1.Zvalue1Click(Sender: TObject);  
var  
    zString: string[7];  
begin  
    zString := InputBox('Z-Value', 'float:', '0.0');  
end;
```

The first argument gives the title of the input box. The second is the prompt. “float:” serves as a reminder to enter a floating point value. The third argument provides a default value to return.

Position the form near the center of the screen or set the form’s Position property to poScreenCenter. Save, compile and run. Click the menu item. Close the input box. The menu showed properly. The input box showed properly. Since the form shows at or near the center of the screen, and the input box shows near the center of the screen, the input box has to obscure part of the form. Since there was no hole left in the form when the input box closed, then the OnPaint event must have fired. The only adverse effect is the fact that the view port is a little shorter, since the presence of the menu reduced the ClientHeight.

## Pushing the (X,Y,Z) Limits

Add zvalue as a new field to the private section of the form declaration.

```
private  
    GLContext: HGLRC;  
    glDC: HDC;  
    errorCode: GLenum;
```

```

    openGLReady: boolean;
    oldw,
    oldh: integer;
    zvalue: GLfloat;
    procedure ExceptionGL(Sender:TObject;E:Exception);

```

Edit the Zvalue1Click event handler.

```

procedure TForm1.Zvalue1Click(Sender: TObject);
var
    zString: string[7];
begin
    zString := InputBox('Z-Value','float:','0.0');
    zvalue := StrToFloat(zString);
end;

```

Change the calls to glVertex from the two-parameter version to the three-parameter version inside FormPaint, using the freshly-minted zvalue as the third argument.

```

    glBegin(GL_POLYGON);
        glVertex3f(0.25,0.25,zvalue);
        glVertex3f(0.75,0.25,zvalue);
        glVertex3f(0.75,0.75,zvalue);
        glVertex3f(0.25,0.75,zvalue);
    glEnd;

```

Save, compile and run. Select the menu item and enter 0.5 in the input box (and press the Ok button, of course). Do it again, but enter 1.0 as the value. Again, but use 1.01. Something different happened! The rectangle disappeared! Every time the input box covers part of form and closes, the form must re-paint. The results of each new zvalue appear as soon as the box closes. As long as zvalue is between the z-axis limits of the clipping volume (the glOrtho projection), the rectangle is visible. The side of the clipping volume toward the viewer is at  $z = 1.0$ . The side of the clipping volume away from the viewer, into the screen is at  $z = -1.0$ . As long as the rectangle lies between  $-1.0$  and  $1.0$  on the z-axis, it is visible. Values of 0.5, 1.0, -0.5, and -1.0 kept the figure in sight, but 1.01 and  $-1.01$  made it vanish.

Use the Menu Designer to add two more main menu items. The captions are &Y-Value and &X-Value. Add the following event handlers:

```

procedure TForm1.Yvalue1Click(Sender: TObject);
var
    yString: string[7];
begin
    yString := InputBox('Y-Value','float:','0.0');
    yvalue := StrToFloat(yString);
end;
procedure TForm1.Xvalue1Click(Sender: TObject);
var

```

```
xString: string[7];  
begin  
  xString := InputBox('X-Value', 'float:', '0.0');  
  xvalue := StrToFloat(xString);  
end;
```

Add xvalue and yvalue to private section of the form declaration:

```
private  
  GLContext: HGLRC;  
  glDC: HDC;  
  errorCode: GLenum;  
  openGLReady: boolean;  
  oldw,  
  oldh: integer;  
  xvalue,  
  yvalue,  
  zvalue: GLfloat;  
procedure ExceptionGL(Sender:TObject;E:Exception);
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Use the new fields in the drawing:

```
glBegin(GL_POLYGON) ;
    glVertex3f(xvalue+0.25,yvalue+0.25,zvalue) ;
    glVertex3f(xvalue+0.75,yvalue+0.25,zvalue) ;
    glVertex3f(xvalue+0.75,yvalue+0.75,zvalue) ;
    glVertex3f(xvalue+0.25,yvalue+0.75,zvalue) ;
glEnd;
```

Save, compile and run. Try 0.5 and  $-1.5$  for `yvalue`, then go back to 0.0. Try the same numbers for `xvalue`. Now the meaning of a clipping volume should be quite clear. If any part of the figure goes outside the boundaries of the clipping volume, that part is clipped off, but the rest remains visible.



**Figure 2-2.** Clipping with `yvalue = 0.5`

Naturally, good code uses a more reliable method of insuring a re-paint than just having an input box go away, but for this little clipping demonstration, it was sufficient.

## DEFINITIONS

Clipping	The act of not drawing the part of a scene that lies beyond a specified plane.
Clipping volume	A volume enclosed by clipping planes. The parts of a scene lying outside the volume are not drawn.
Foreground color	The current drawing color.

Matrix	A two-dimensional array of numbers for which special mathematical operations have been defined.
Orthographic projection	Parallel projection.
Parallel projection	A projection that does not make size correction for distance.
Polygon	A closed two-dimensional figure consisting of three or more vertices with sides connecting one vertex to the next, and finally a side connecting the last vertex to the first.
Projection	The mathematical transformation of points in a three-dimensional scene onto a flat surface, such as the computer screen.
Quadrilateral	A four-sided polygon.
Render	To convert drawing elements into an image. To draw with OpenGL.
State variables	Much OpenGL information is contained in variables whose values remain the same until deliberately changed. This kind of information is a state. The variables that hold such information are state variables.
Vertex	A point in three-dimensional space, usually representing an endpoint of a line segment or a corner of a polygon.
View port	The portion of a window's client area in which to draw the clipping area.

## IDENTIFIER REFERENCE

glBegin	An OpenGL command designating the start of a list of vertices. It receives a mode parameter that tells OpenGL what to do with those vertices.
GLbitfield	An OpenGL type that maps to Delphi cardinal.
GLclampf	An OpenGL type equivalent to Delphi single.
glColor	An OpenGL command that sets the foreground color.
GLdouble	An OpenGL type equivalent to Delphi double.
glEnd	An OpenGL command that terminates a list of vertices started with glBegin.
GLenum	An OpenGL type that maps to Delphi cardinal.
GLfloat	An OpenGL type that maps to Delphi single.
glFlush	An OpenGL command that forces drawing to the screen.
GLint	An OpenGL type that maps to Delphi integer.
glLineWidth	An OpenGL command that specifies the width, in pixels, of all subsequent lines drawn.



glOrtho	An OpenGL command that specifies an orthographic projection and clipping volume.
glRect	An OpenGL command for drawing a rectangle.
GLsizei	An OpenGL type that maps to Delphi integer.
glVertex	An OpenGL command for specifying the coordinates of a vertex.
glViewport	An OpenGL command that specifies the viewport in window coordinates.
GL_COLOR_BUFFER_BIT	An OpenGL constant that identifies the color buffer.
GL_LINES	An OpenGL constant to pass to glBegin. Each pair of vertices then specifies a line segment, separate from any of the other line segments.
GL_POLYGON	An OpenGL constant to pass to glBegin. All the vertices listed before glEnd are connected to form a polygon.
TimeGetTime	A Windows function that returns the number of milliseconds since Windows was started.

## SUMMARY

Chapter 2 presented the following concepts:

1. Perform drawing in the window's OnPaint event handler.
2. Drawing occurs within the bounds of the view port.
3. The view port definition uses the operating system's window coordinates, but counts in the y-direction from the bottom up.
4. Use glColor to set the foreground drawing color, changing as needed. The color is one of many state variables that remain the same until the programmer directly changes them via function or procedure call.
5. Presenting a 3-D scene on a 2-D screen requires projection.
6. Orthographic projection does not change the apparent size of an object to correct for distance. It is also called parallel projection.
7. The projection is a volume in space bounded by the view port.
8. The programmer may assign the numbers defining the projection volume for clarity or convenience.
9. The projection region is a clipping volume. Any part of a scene that lies outside the volume simply does not appear.
10. Adjust the view port in the window's OnResize event handler.
11. Projection involves matrix multiplication. Repeated invocation is repeated multiplication which probably gives unintended results. The solution in this chapter was to invoke projection only once in the FormCreate method.
12. OpenGL does not hinder the use of Windows features such as menus and dialog boxes.
13. The glGetError function returns the error code number for recent

operations. It returns a number equal to GL\_NO\_ERROR if no errors have accumulated. OpenGL stores accumulated errors, so the program must call glGetError repeatedly until GL\_NO\_ERROR comes back.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

# Chapter 3 Introducing 3-D

Chapter 2 introduced drawing a two-dimensional figure and projecting it onto the screen. However, OpenGL is rather under-utilized in 2-D. OpenGL shines in three-dimensional operations. This chapter explores the third (z) dimension and shows a way to draw a cube in three-dimensional space. The cube has six square faces with different colors for each. This chapter shows how to rotate, move, and stretch the cube, as well as cut it open to show what is inside.

The Rect program from chapter 2 makes a good starting point for the next program. Create a new directory as before and name it “Chapter.3.” Under that directory create a directory called “Cube.” In Delphi load the Rect.Dpr project and do a File|Save Project As in the new directory, naming it “Cube.Dpr.” Do File|Save As for the “Rect1.Pas” unit. Name it “Cube1.Pas” in the new directory.

## DEPTH

The introduction of a third dimension calls for an understanding of the direction in which that dimension runs. Construct a new FormPaint method as shown in Listing 3-1.

**Listing 3-1.** FormPaint Method with Depth.

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    if not openGLReady then  
        exit;  
  
    {green background}
```

```

glClearColor(0.0,0.4,0.0,0.0);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glBegin(GL_POLYGON); {near face}
    glColor3f(0.5,0.0,0.0);
    glVertex3f(-0.4,-0.4,-0.4);
    glVertex3f(+0.4,-0.4,-0.4);
    glVertex3f(+0.4,+0.4,-0.4);
    glVertex3f(-0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {back face}
    glColor3f(0.0,0.0,0.5);
    glVertex3f(-0.2,-0.2,+0.4);
    glVertex3f(+0.6,-0.2,+0.4);
    glVertex3f(+0.6,+0.6,+0.4);
    glVertex3f(-0.2,+0.6,+0.4);
glEnd;

glFlush;

errorCode := glGetError;
if errorCode<>GL_NO_ERROR then
    raise Exception.Create('Error in Paint'#13+
        gluErrorString(errorCode));
end; {FormPaint}

```

The z value of the original (red) rectangle is  $-0.4$ . The second rectangle is  $0.8$  units farther along the positive z direction with a z value of  $+0.4$ . The second (blue) rectangle also lies  $0.2$  units farther in the positive x direction and  $0.2$  units farther in the positive y direction, so that one rectangle does not completely obscure the other.

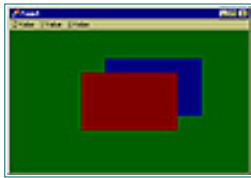
Without a *depth test* an object drawn later may appear on top of an object drawn earlier, no matter how near or far the respective objects actually were. The explanation for the depth test appears just a few paragraphs later, but for now, add some lines to the bottom of the FormCreate method:

```

glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
glEnable(GL_DEPTH_TEST);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity;
end; {FormCreate}

```

Save, compile, and run to see the result. As Figure 3-1 shows, the red rectangle is in front and the blue rectangle is in back. The conclusion is that with an orthographic projection, the farther an object lies in the positive z direction, (or at least less negative), the farther the object is from the viewer.



**Figure 3-1.** Depth

## CUBE

Now you are ready for another illustration. For the sake of this illustration, comment out the `glEnable(GL_DEPTH_TEST)` statement you just put in.

### Square View Port

The new project name certainly suggests the next shape under discussion. A cube provides a fairly simple figure for introducing 3-D. In geometry a cube has identical height, width, and depth. Unfortunately the projection of such a figure onto the computer screen may not look like a cube unless the view port is square. Since the current view port matches the form's client height and client width, those properties should be set equal to each other. Select the form and go to the object inspector. Set `ClientHeight` to 340, and set `ClientWidth` to 340.

### Distinguish Faces

A cube has six square faces. In `FormPaint` draw six colored squares, using different colors for each face for easy distinction. If all the faces of the cube were the same color, they would blend into each other and obscure the shape. The application of lighting effects solves such problems, but that subject comes later. Modify the `FormPaint` method as in listing 3-2.

**Listing 3-2.** Six Colored Faces.

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    if not openGLReady then  
        exit;  
  
    {green background}  
    glClearColor(0.0,0.4,0.0,0.0);  
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);  
  
    glBegin(GL_POLYGON); {near face}  
        glColor3f(0.5,0.0,0.0);  
        glVertex3f(-0.4,-0.4,-0.4);  
        glVertex3f(+0.4,-0.4,-0.4);  
        glVertex3f(+0.4,+0.4,-0.4);  
        glVertex3f(-0.4,+0.4,-0.4);  
    glEnd;  
  
    glBegin(GL_POLYGON); {right face}  
        glColor3f(0.3,0.3,0.8);
```

```

        glVertex3f(+0.4,-0.4,-0.4);
        glVertex3f(+0.4,-0.4,+0.4);
        glVertex3f(+0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,-0.4);
    glEnd;

    glBegin(GL_POLYGON); {left face}
        glColor3f(0.5,0.0,0.5);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(-0.4,+0.4,-0.4);
        glVertex3f(-0.4,+0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {back face}
        glColor3f(0.0,0.0,0.5);
        glVertex3f(+0.4,-0.4,+0.4);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(-0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {bottom face}
        glColor3f(0.5,0.5,0.0);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(+0.4,-0.4,+0.4);
        glVertex3f(+0.4,-0.4,-0.4);
    glEnd;

    glBegin(GL_POLYGON); {top face}
        glColor3f(0.3,0.0,0.1);
        glVertex3f(-0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,+0.4);
        glVertex3f(-0.4,+0.4,+0.4);
    glEnd;

    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
        raise Exception.Create('Error in Paint'#13+
            gluErrorString(errorCode));
    end; {FormPaint}

```

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

**Bookmark It**

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

### Depth Test

Save, compile, and run and admire the beautiful cube! Well, at least tolerate the colored rectangle. Why is there no evidence of a third dimension? Actually the straight-on view guarantees that only the front face shows. Or does it? Notice the color. The front face is red. The back face is blue. The blue back face shows on the screen! Why? OpenGL drew all six faces. The four edge-on faces have no effect, but the back face was drawn after the front face, so it is the one that shows. Fortunately there is a way to tell OpenGL to test depth and not draw pixels that should be hidden. This ability, as well as others, becomes effective by means of the `glEnable` command:

```
procedure glEnable(cap:GLenum); stdcall;
```

Turn on each capability by passing the appropriate constant. Here is the declaration for the depth-testing constant:

```
GL_DEPTH_TEST = $0B71;
```

Enabling depth testing sets up a depth buffer that contains depth information on each pixel within the clipping volume. Of the pixels that map to the same spot on the screen, OpenGL draws only the one nearest the viewer. This process is OpenGL's method of hidden surface removal. The bottom of the `FormCreate` method with depth testing added follows:

```
openGLReady := true;  
oldw := ClientWidth;  
oldh := ClientHeight;  
glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);  
glEnable(GL_DEPTH_TEST);  
end; {FormCreate}
```

## Matrices Revisited

Save, compile, and run. Observe that no change is evident. This means that more must be done to make the depth test work. Therefore it is time to give more consideration to matrices. OpenGL is full of them. Matrices describe the very scene itself, and there are matrices for texture, for projection, etc. There are commands to select among those matrices. There are even matrix stacks. Some commands, such as `glLoadIdentity`, directly manipulate the active matrix, and other commands use their parameters to construct a matrix, then multiply the appropriate matrix by the constructed matrix.

### The ModelView Matrix

Consider a 4x4 matrix that represents the coordinate system of the scene. Call it the Model matrix. Apply this matrix to every vertex in the scene as part of the process of illuminating the appropriate dots on the screen. Now consider a 4x4 matrix that represents the coordinates of the viewer. Call it the View matrix. Finally consider Einstein's Theory of Relativity. What?! Well, anyway, think of two automobiles side by side. The observer is in the first car and sees the second car sliding backwards. Or is the first car with the observer actually moving forward? Or does it matter? In a universe that contains only those two cars, either description is equally valid. Similarly the Model matrix and the View matrix are equally useful. In fact they are the same matrix. Its true name is the ModelView matrix.

Here is the OpenGL command that identifies which matrix other commands will affect:

```
procedure glMatrixMode(mode:GLenum); stdcall;
```

These are the constants to pass for the mode parameter:

```
GL_MODELVIEW    = $1700;  
GL_PROJECTION   = $1701;  
GL_TEXTURE      = $1702;
```

Here is the declaration of the command that sets the current matrix to be the *identity matrix*, which is a matrix that, when multiplied with another matrix, leaves a product the same as the other matrix.

```
procedure glLoadIdentity; stdcall;
```

For the three-dimensional rendering program to work, make the bottom of `FormCreate` look like this:

```
glEnable(GL_DEPTH_TEST);  
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity;  
end; {FormCreate}
```

So save, compile, and run. Ah! The correct face shows. But it still does not



look like a cube.

## ROTATION

### Three Faces

Looking at a cube straight on just does not reveal its third dimension. The cube must have an orientation that shows more than one face. Indeed, a cube showing two faces is not much better, because it still just looks like two rectangles side by side. The obvious answer is to draw the cube skewed at some angle. Not only that, but the rotation must not be about a single coordinate axis, or about any line parallel to a single axis, otherwise no more than two faces can show.

### glRotate

Drawing the cube rotated calls for some ugly math to apply to each vertex. Actually the math involved looks fairly simple when expressed as matrices, but it is ugly because the programmer has to carry out the details. There is a better way! Draw the cube simply, as before, then let OpenGL rotate it. Here is the needed command in two versions:

```
procedure glRotated(angle,x,y,z:GLdouble); stdcall;  
procedure glRotatef(angle,x,y,z:GLfloat); stdcall;
```

**PURPOSE** The purpose of this command is to define an axis of rotation and an amount of rotation. The command then causes the scene to rotate about the axis. Or it causes the coordinate system of the scene, and therefore the viewing angle of the observer, to rotate about the axis. Take your choice; it is relative! What really happens is that OpenGL creates a rotation matrix from the parameters and multiplies it with the active matrix, in this case, the ModelView matrix.

**PARAMETERS** This simple program certainly does not need double precision to accomplish its purpose, so the float version is the proper choice for the command.

Either version uses four parameters. The angle parameter defines the amount of rotation. Interestingly it takes its value in degrees rather than radians, which is fortunate for mere mortals who do not spend their days immersed in trigonometry. The x, y, and z parameters define a point in three-dimensional space with respect to the coordinates of the clipping (projection) volume.

The given point helps define the *vector* that serves as the axis of rotation. Two points can define a line; the other point is (0.0,0.0,0.0), the origin for the projection coordinate system. Think of a vector as a line with direction, from the origin toward the other point, and then continuing on forever.

**DIRECTION** To understand the direction of rotation, imagine an observer located at the origin looking toward the point defined by the x, y, and z parameters. The object turns counter-clockwise with respect to that vantage point.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

### Action

In the Menu Designer add a fourth main menu item. Give it a caption of &Angle and exit from the Menu Designer. Click the menu item to produce the event handler and fill it in as follows:

```
procedure TForm1.Angle1Click(Sender: TObject);  
var  
    astring: string[7];  
begin  
    astring := InputBox('Angle of Rotation', 'float:', '0.0');  
    angle := StrToFloat(astring);  
    glMatrixMode(GL_MODELVIEW);  
    glLoadIdentity;  
  
    glRotatef(angle, xvalue, yvalue, zvalue);  
end;
```

The other three menu choices, left over from the Rect program, set the coordinates for the point (other than the origin) that defines the axis of rotation. The new menu choice defines the angle of rotation and performs the rotation. Notice that `glMatrixMode` makes the `ModelView` matrix active every time, since that is the matrix on which to perform the action. This action is one of several *transformations*. OpenGL enables the programmer to perform three basic `ModelView` transformations or combinations thereof on the `ModelView` matrix, rotation, *translation* (movement in a straight line through space), and *scale* (changing the size of an object in each of the three dimensions independently). The call to `glLoadIdentity` is present to insure that the rotation transformation acts from a known and understood condition. Some situations may naturally call for allowing the transformations to accumulate, but not this simple interactive angle entry.

Save, compile and run. Enter coordinates to define the axis (vector) of rotation. Enter various angles and see the results. Notice that any multiple of 360.0 returns the cube to its original orientation and that negative angles rotate the cube clockwise. All the different colors for the faces help identify which faces come into view.

## TWO-FACED POLYGONS

### Open the Cube

There is no way to avoid it. Polygons are two-faced. That is not a character assassination but a geometric statement. When a polygon facing the viewer rotates 180 degrees, it does not disappear; it has a back face that the viewer can see. To illustrate, expose the inside of the cube by commenting out the right side as in Listing 3-3.

**Listing 3-3.** Right Side Missing.

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    if not openGLReady then  
        exit;  
  
    {green background}  
    glClearColor(0.0,0.4,0.0,0.0);  
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);  
  
    glBegin(GL_POLYGON); {near face}  
        glColor3f(0.5,0.0,0.0);  
        glVertex3f(-0.4,-0.4,-0.4);  
        glVertex3f(+0.4,-0.4,-0.4);  
        glVertex3f(+0.4,+0.4,-0.4);  
        glVertex3f(-0.4,+0.4,-0.4);  
    glEnd;  
    (*  
    glBegin(GL_POLYGON); {right face}  
        glColor3f(0.3,0.3,0.8);  
        glVertex3f(+0.4,-0.4,-0.4);  
        glVertex3f(+0.4,-0.4,+0.4);  
        glVertex3f(+0.4,+0.4,+0.4);  
        glVertex3f(+0.4,+0.4,-0.4);  
    glEnd;  
    *)  
    glBegin(GL_POLYGON); {left face}  
        glColor3f(0.5,0.0,0.5);  
        glVertex3f(-0.4,-0.4,+0.4);  
        glVertex3f(-0.4,-0.4,-0.4);  
        glVertex3f(-0.4,+0.4,-0.4);  
        glVertex3f(-0.4,+0.4,+0.4);  
    glEnd;  
  
    glBegin(GL_POLYGON); {back face}
```

```

        glColor3f(0.0,0.0,0.5);
        glVertex3f(+0.4,-0.4,+0.4);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(-0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {bottom face}
        glColor3f(0.5,0.5,0.0);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(+0.4,-0.4,+0.4);
        glVertex3f(+0.4,-0.4,-0.4);
    glEnd;

    glBegin(GL_POLYGON); {top face}
        glColor3f(0.3,0.0,0.1);
        glVertex3f(-0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,+0.4);
        glVertex3f(-0.4,+0.4,+0.4);
    glEnd;

    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
        raise Exception.Create('Error in Paint'#13+
            gluErrorString(errorCode));
end; {FormPaint}

```

Save, compile, and run. With the menus make xvalue and yvalue each equal to 1.0, but leave zvalue alone. For the angle enter 30.0 (degrees). Note the inside face for the back square and the top square. Experiment with various combinations of positive and negative values for all four parameters until all interior faces have presented themselves.

## Culling

There it is! Those polygons have a back side! What if the cube was not open? The shocking truth is that even if there is no possible way to see the back side of the polygons, OpenGL draws them! Fortunately *depth testing* (hidden surface removal) keeps them from bleeding though. Better yet, an OpenGL command can tell it not even to draw them.

```
procedure glCullFace(mode:GLenum); stdcall;
```

The possible values for mode (with obvious meanings) are:

```

GL_FRONT          = $0404;
GL_BACK           = $0405;

```

```
GL_FRONT_AND_BACK= $0408;
```

To *cull* faces means to remove them from the list of things to draw. Like many other OpenGL options, the programmer must first enable this option before using it. The value to give glEnable is:

```
GL_CULL_FACE= $0B44;
```

Add some code so the bottom of FormCreate looks like this:

```
glEnable(GL_DEPTH_TEST);  
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity;  
  
glEnable(GL_CULL_FACE);  
glCullFace(GL_BACK);  
end; {FormCreate}
```

Save, compile, and run yet again. As before, experiment with various combinations of positive and negative values for all four parameters and see what the inside of the cube looks like when the inner faces simply are not drawn.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

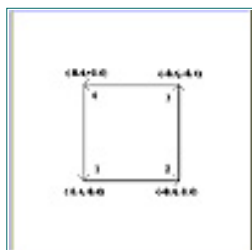
## VERTEX DIRECTION

How does OpenGL know which face of a polygon the programmer intended to be front and which to be back? The answer lies in the order of the vertices.

Consider the front square of this chapter's cube.

```
glBegin(GL_POLYGON); {near face}
  glColor3f(0.5,0.0,0.0);
  glVertex3f(-0.4,-0.4,-0.4);
  glVertex3f(+0.4,-0.4,-0.4);
  glVertex3f(+0.4,+0.4,-0.4);
  glVertex3f(-0.4,+0.4,-0.4);
glEnd;
```

Mentally number the vertices from one to four. Vertex one is the lower left corner. Vertex two is the lower right corner. Vertex three is the upper right corner. Vertex four is the upper left corner. Imagine movement from one to two to three to four and back to one. The direction or *winding* is counter-clockwise, which is the default direction for OpenGL to identify the front face of a polygon. Those same vertices, if viewed from the other side, would present a clockwise winding. Figure 3-2 shows the counter-clockwise winding of the front view.



**Figure 3-2.** Counter-Clockwise Winding.

Consider the back square of the cube. Its vertex order was designed to present a clockwise winding from the viewer's position, making the face inside the cube the back face and the face outside the cube the front face. That is the usual arrangement. If the cube were rotated 180 degrees around the z-axis, the front face of that square would be toward the viewer. The next experiment reverses the back square's winding. Here is the original code for the back square:

```
glBegin(GL_POLYGON); {back face}
  glColor3f(0.0,0.0,0.5);
  glVertex3f(+0.4,-0.4,+0.4);
  glVertex3f(-0.4,-0.4,+0.4);
  glVertex3f(-0.4,+0.4,+0.4);
  glVertex3f(+0.4,+0.4,+0.4);
glEnd;
```

Rearrange the vertices to reverse the winding:

```
glBegin(GL_POLYGON); {back face}
  glColor3f(0.0,0.0,0.5);
  glVertex3f(+0.4,+0.4,+0.4);
  glVertex3f(-0.4,+0.4,+0.4);
  glVertex3f(-0.4,-0.4,+0.4);
  glVertex3f(+0.4,-0.4,+0.4);
glEnd;
```

Save, compile, and run. Give the xvalue 0.3, the yvalue 1.0, and the angle 50.0. While the other interior faces have been culled, the interior face of the back square has been drawn. With a large enough angle, the back square is toward the viewer, but since its exterior face is a "back" face, it does not appear. Indeed, only the bottom of the cube shows at all.

## TRANSLATION

Restore the original winding of the back square and uncomment the right square. While rotation may be the most interesting transformation, it is certainly not the only one of importance. Translation may be the most common transformation.

### Menu

Use the Menu Designer to add &Translate as another main menu item. Click on it and fill in the event handler thusly:

```
procedure TForm1.Translate1Click(Sender: TObject);
begin
  glMatrixMode(GL_MODELVIEW);
  glTranslatef(xvalue,yvalue,zvalue);
  Invalidate;
end;
```

The `glTranslate` command takes three parameters that specify the relative amount of movement in each direction. The translation command has two forms:

```
procedure glTranslated(x,y,z:GLdouble); stdcall;  
procedure glTranslatef x,y,z:GLfloat); stdcall;
```

## Accumulation

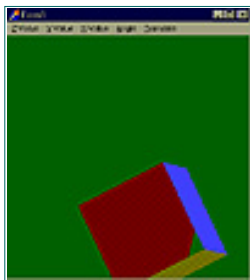
Notice the absence of `glLoadIdentity`. Transformations will accumulate and demonstrate the effects of combining different kinds. Comment out the `glLoadIdentity` in the `Angle1Click` method as well. Notice also the presence of the `Invalidate Windows` API call. This causes Windows to send a `WM_PAINT` message, which triggers the `OnPaint` event. This is certainly superior to the (temporary) reliance on covering and uncovering the window by another window.

## More Clipping

As usual, save, compile, run, and play with the various menu choices and values. The transformations accumulate, and may allow the cube to disappear altogether. Certainly the translation transformation provides more opportunity to experiment with clipping. Enough translation in the positive `z` direction provides some interesting effects. The far side of the cube disappears as if cut off by an invisible wall.

Did you see that flicker each time you selected `Translate`? Ugh! Do not despair; the chapter on animation shows how to redraw images without flicker.

Even more interesting is the effect of translation in the negative `z` direction. With just the right `z` values, a face or a corner nearest the viewer disappears, allowing a peek inside the cube without having to neglect drawing a square. What does the viewer see inside? He sees a mysterious green nothing because of the culling, as in Figure 3-3, where clipping in both the negative `y` direction and the negative `z` direction is evident.



**Figure 3-3.** Clipping and Culling.

## SCALE

### Menu

With the `Menu Designer` add still another main menu item, `&Scale`. Produce the following event handler in the same manner as before:



```
procedure TForm1.Scale1Click(Sender: TObject);  
begin  
    glMatrixMode(GL_MODELVIEW);  
    glScalef(xvalue,yvalue,zvalue);  
    Invalidate;  
end;
```

The only new command has the following declarations:

```
procedure glScaled(x,y,z:GLdouble); stdcall;  
procedure glScalef(x,y,z:GLfloat); stdcall;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

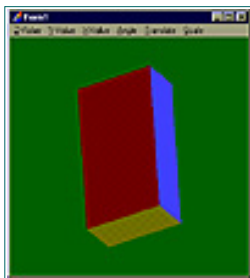
ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

### Description

The scale command multiplies the size of the objects in the scene. The value given for the x parameter multiplies the x-dimension, the y value passed to the command multiplies the y-dimension, and the z parameter's value multiplies the z-dimension. A call to `glScale(2.0,2.0,2.0)` makes the cube twice as wide, twice as tall, and twice as deep as the original specifications. So why are there three parameters? Each dimension's multiplier has a separate specification so each dimension can be scaled differently. A call to `glScale(0.5,3.0,0.8)` makes the cube (or whatever drawing follows) only half as wide, three times as tall, and 80 percent of the original depth. The programmer can draw something and then distort it freely. Of course a cube so mutilated is no longer a cube. Save, compile, run, and experiment. Figure 3-4 shows a distorted (former) cube.



**Figure 3-4.** The Cube After `glScale`.

## ORDER OF TRANSFORMATIONS

### A New Interface

The admittedly clumsy interface used thus far has exhausted its usefulness. It was kept simple in order to minimize distraction from the OpenGL coding, but further development calls for a better user interface. The new interface is nothing elegant, but is a great improvement.

Create a new directory under “Chapter.3” named “Multi.” Do a File|Save Project As of Cube.Dpr into the new directory, calling it Mult.Dpr. Save Cube1.Pas as Multi.Pas. Empty all the menu item event handlers by deleting the lines of code within and save again so Delphi can remove the newly empty event handlers. In the Menu Designer delete the existing menu items and put in &Parameters, &Transformation, &Identity, and &Cull Disable. Create and fill in their event handlers as in Listing 3-4.

**Listing 3-4.** Menu Item Event Handlers.

```

procedure TForm1.Parameters1Click(Sender: TObject);
begin
    FormTransform.ShowModal;
end;

procedure TForm1.Transformation1Click(Sender: TObject);
var
    list: TStringList;
    lno: integer;
begin
    list := TStringList.Create;
    list.sorted := true;
    with list,transData do
    begin
        if translate then
            Add(OrT+'T');
        if scale then
            Add(OrS+'S');
        if rotate then
            Add(OrR+'R');
        if count=0 then
            exit;

        glMatrixMode(GL_MODELVIEW);
        for lno := 0 to count-1 do
        begin
            case strings[lno][2] of
                'R': begin
                    glRotatef(ra,rx,ry,rz);
                end;
                'S': begin
                    glScalef(sx,sy,sz);
                end;
                'T': begin
                    glTranslatef(tx,ty,tz);
                end;
            end; {case}
        end; {for}
    end; {with}
    List.Free;

```

```
        Invalidate;
    end;
procedure TForm1.Identity1Click(Sender: TObject);
begin
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity;
    Invalidate;
end;

procedure TForm1.Cull1Click(Sender: TObject);
begin
    glDisable(GL_CULL_FACE);
    Invalidate;
end;
```

Create a new form and name it FormTransform, then save the unit as Trans1. To the form add three check boxes, four labels, ten edit controls, three radio groups, and two bit buttons to look like Figure 3-5. An efficient way to describe the component names and properties is to show the form as text as in Listing 3-5.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

### **Listing 3-5.** FormTransform as Text.

```
object FormTransform: TFormTransform
  Left = 200
  Top = 100
  Width = 360
  Height = 185
  Caption = 'Transformation Parameters'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 13
object Label1: TLabel
  Left = 76
  Top = 4
  Width = 7
  Height = 13
  Caption = 'X'
end
object Label2: TLabel
  Left = 120
  Top = 4
  Width = 7
  Height = 13
  Caption = 'Y'
end
object Label3: TLabel
  Left = 164
```

```
    Top = 4
    Width = 7
    Height = 13
    Caption = 'Z'
end
object Label4: TLabel
    Left = 208
    Top = 4
    Width = 27
    Height = 13
    Caption = 'Angle'
end
object CheckBoxTranslate: TCheckBox
    Left = 4
    Top = 20
    Width = 64
    Height = 24
    Caption = 'Translate'
    TabOrder = 0
    OnClick = CheckBoxTranslateClick
end
object CheckBoxScale: TCheckBox
    Left = 4
    Top = 56
    Width = 64
    Height = 24
    Caption = 'Scale'
    TabOrder = 5
    OnClick = CheckBoxScaleClick
end
object CheckBoxRotate: TCheckBox
    Left = 4
    Top = 92
    Width = 64
    Height = 24
    Caption = 'Rotate'
    TabOrder = 10
    OnClick = CheckBoxRotateClick
end
object EditTX: TEdit
    Left = 72
    Top = 20
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 1
end
object EditTY: TEdit
    Left = 116
    Top = 20
```

```
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 2
end
object EditTZ: TEdit
    Left = 160
    Top = 20
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 3
end
object EditRA: TEdit
    Left = 204
    Top = 92
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 14
end
object EditSX: TEdit
    Left = 72
    Top = 56
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 6
end
object EditSY: TEdit
    Left = 116
    Top = 56
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 7
end
object EditSZ: TEdit
    Left = 160
    Top = 56
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 8
end
object EditRX: TEdit
    Left = 72
    Top = 92
    Width = 40
    Height = 21
```

```
        Enabled = False
        TabOrder = 11
    end
    object EditRY: TEdit
        Left = 116
        Top = 92
        Width = 40
        Height = 21
        Enabled = False
        TabOrder = 12
    end
    object EditRZ: TEdit
        Left = 160
        Top = 92
        Width = 40
        Height = 21
        Enabled = False
        TabOrder = 13
    end
    object BitBtnOk: TBitBtn
        Left = 60
        Top = 124
        Width = 90
        Height = 25
        Enabled = False
        TabOrder = 16
        OnClick = BitBtnOkClick
        Kind = bkOK
    end
    object BitBtnCancel: TBitBtn
        Left = 220
        Top = 124
        Width = 80
        Height = 25
        TabOrder = 17
        Kind = bkCancel
    end
    object RadioGroupT: TRadioGroup
        Left = 252
        Top = 14
        Width = 97
        Height = 32
        Columns = 3
        Enabled = False
        ItemIndex = 0
        Items.Strings = (
            '1'
            '2'
            '3')
        TabOrder = 4
```



```

        OnClick = RadioGroupClick
    end
    object RadioGroupS: TRadioGroup
        Left = 252
        Top = 50
        Width = 97
        Height = 32
        Columns = 3
        Enabled = False
        ItemIndex = 1
        Items.Strings = (
            '1'
            '2'
            '3')
        TabOrder = 9
        OnClick = RadioGroupClick
    end
    object RadioGroupR: TRadioGroup
        Left = 252
        Top = 86
        Width = 97
        Height = 32
        Columns = 3
        Enabled = False
        ItemIndex = 2
        Items.Strings = (
            '1'
            '2'
            '3')
        TabOrder = 15
        OnClick = RadioGroupClick
    end
end
end

```



**Figure 3-5.** Parameter Data Entry Form for Transformations

The text version of the form identified the names of the event handlers, and the code for unit Trans1 in Listing 3-6 reveals the contents of the event handlers:

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

### **Listing 3-6.** The Trans1 Unit.

```
unit Trans1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, Buttons, OpenGL, ExtCtrls;
```

```
type
```

```
TFormTransform = class(TForm)  
  Label1: TLabel;  
  Label2: TLabel;  
  Label3: TLabel;  
  Label4: TLabel;  
  CheckBoxTranslate: TCheckBox;  
  CheckBoxScale: TCheckBox;  
  CheckBoxRotate: TCheckBox;  
  EditTX: TEdit;  
  EditTY: TEdit;  
  EditTZ: TEdit;  
  EditRA: TEdit;  
  EditSX: TEdit;  
  EditSY: TEdit;  
  EditSZ: TEdit;  
  EditRX: TEdit;  
  EditRY: TEdit;  
  EditRZ: TEdit;  
  BitBtnOk: TBitBtn;  
  BitBtnCancel: TBitBtn;  
  RadioGroupT: TRadioGroup;  
  RadioGroupS: TRadioGroup;  
  RadioGroupR: TRadioGroup;  
procedure BitBtnOkClick(Sender: TObject);
```

```

    procedure CheckBoxTranslateClick(Sender: TObject);
    procedure CheckBoxScaleClick(Sender: TObject);
    procedure CheckBoxRotateClick(Sender: TObject);
    procedure RadioGroupClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    FormTransform: TFormTransform;
    transData: record
        translate,
        scale,
        rotate: boolean;
        orT,
        orS,
        orR: char;
        tx,ty,tz,
        sx,sy,sz,
        rx,ry,rz,ra: GLfloat;
    end;

implementation

{$R *.DFM}

procedure TFormTransform.BitBtnOkClick(Sender: TObject);
begin
    ModalResult := mrNone;

    with transData do
    begin
        translate := checkBoxTranslate.Checked;
        scale := checkBoxScale.Checked;
        rotate := checkBoxRotate.Checked;

        if translate then
        begin
            tx := StrToFloat(EditTX.Text);
            ty := StrToFloat(EditTY.Text);
            tz := StrToFloat(EditTZ.Text);
        end;

        if scale then
        begin
            sx := StrToFloat(EditSX.Text);
            sy := StrToFloat(EditSY.Text);
            sz := StrToFloat(EditSZ.Text);
        end;

        if rotate then
        begin
            rx := StrToFloat(EditRX.Text);

```

```

        ry := StrToFloat(EditRY.Text);
        rz := StrToFloat(EditRZ.Text);
        ra := StrToFloat(EditRA.Text);
    end;

    orT := char(48+RadioGroupT.ItemIndex);
    orS := char(48+RadioGroupS.ItemIndex);
    orR := char(48+RadioGroupR.ItemIndex);
end; {with}

ModalResult := mrOk;
end;

procedure TFormTransform.CheckBoxTranslateClick(Sender: TObject);

var
    chck: boolean;
begin
    with Sender as TCheckBox do
        chck := checked;
        EditTX.Enabled := chck;
        EditTY.Enabled := chck;
        EditTZ.Enabled := chck;
        RadioGroupT.Enabled := chck;
        BitBtnOk.Enabled := chck
    or EditSX.Enabled
    or EditRX.Enabled;
end;

procedure TFormTransform.CheckBoxScaleClick(Sender: TObject);
var
    chck: boolean;
begin
    with Sender as TCheckBox do
        chck := checked;
        EditSX.Enabled := chck;
        EditSY.Enabled := chck;
        EditSZ.Enabled := chck;
        RadioGroupS.Enabled := chck;
        BitBtnOk.Enabled := chck
    or EditTX.Enabled
    or EditRX.Enabled;
end;

procedure TFormTransform.CheckBoxRotateClick(Sender: TObject);
var
    chck: boolean;
begin
    with Sender as TCheckBox do
        chck := checked;
        EditRX.Enabled := chck;
        EditRY.Enabled := chck;
        EditRZ.Enabled := chck;
        EditRA.Enabled := chck;
        RadioGroupR.Enabled := chck;

```

```

    BitBtnOk.Enabled := chck
    or EditSX.Enabled
    or EditTX.Enabled;
end;

procedure TFormTransform.RadioGroupClick(Sender:TObject);
type
    TRange=0..2;
    SRange=Set of TRange;
var
    ch: char;
    Xii,
    Uii,
    Tii,
    Sii,
    Rii: integer;
    Used: SRange;
    ar: array['R'..'T'] of TRadioGroup;
    notif: TNotifyEvent;
begin
    {build set of used values to find what's left}
    Tii := RadioGroupT.ItemIndex;
    Sii := RadioGroupS.ItemIndex;
    Rii := RadioGroupR.ItemIndex;
    Used := [Tii]+[Sii]+[Rii];
    Xii := TRadioGroup(Sender).ItemIndex;

    {build array of rg for convenient searching}
    ar['T'] := RadioGroupT;
    ar['S'] := RadioGroupS;
    ar['R'] := RadioGroupR;

    {find rg not the sender but with same ItemIndex}
    for ch := 'R' to 'T' do
    with ar[ch] do
    if (ar[ch]<>Sender) and (ItemIndex=Xii) then
    begin
        for Uii := 0 to 2 do
        if not (Uii in Used) then
        begin
            notif := OnClick;
            OnClick := nil; {prevent second triggering}

            {set to unused ItemIndex and leave}
            ar[ch].ItemIndex := Uii;
            OnClick := notif;
            exit;
        end; {for if}
    end; {for with if}

end;

```

The scope of this book is to teach OpenGL with Delphi, with the assumption that the reader is at

least somewhat experienced in programming with Delphi. Therefore the ordinary Delphi code is presented with little comment.

## Simultaneous Commands

This program still draws the same old cube, but provides for specifying two or three transformations (with independent parameters) to apply to the cube at the same time. The check boxes allow the user to specify which transformations to apply, and the radio groups allow the user to specify the order in which to apply them. The edit controls allow the user to specify the floating point values for the various parameters.

The event handler for the &Transformation menu item performs the transformations specified on the secondary form. TForm1.Transformation1Click creates a sorted string list that identifies the order in which to perform the transformation. The handler sets the matrix mode to GL\_MODELVIEW and performs the transformations in the appropriate order, using the arguments stored in the transData record. The call to Invalidate insures that FormPaint fires after the transformation calls.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 [EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Save, compile, run, and experiment, not only with the values to give to each transformation, but also with the order of the transformation. Admittedly the difference is subtle for just one “object” in the scene, but many combinations will show a difference according to the sequence of the transformations. Order does matter; try it and see.

## **DEFINITIONS**

Culling	Removing faces from the list of things to draw based on front or back designation.
Depth buffer	A list of information about the distance of pixels from the viewer.
Depth test	Determination of whether to draw pixels by comparing their distance from the viewer with that of other pixels at the same screen position.
Hidden surface removal	Not drawing surface that would otherwise be obscured from the viewer anyway.
Identity matrix	A matrix whose product with a second matrix is identical to the second matrix.
ModelView matrix	The matrix representing the coordinate system.
Origin	The point located at (0.0,0.0,0.0).
Scale	Each dimension of an object is adjusted independently by specified factors.
Transformation	Adjusting the coordinate system.
Translation	Moving an object through space.

Vector A line with direction from a starting point toward a second point, then continuing on forever. Vectors are specified as if the first point were the origin and the second point simply sets the direction. With the first point understood to be the origin, the second point alone identifies the vector.

## IDENTIFIER REFERENCE

<code>glCullFace</code>	An OpenGL command that specifies culling of front or back faces or both.
<code>glEnable</code>	An OpenGL command that allows certain features.
<code>glLoadIdentity</code>	An OpenGL command that sets the specified matrix equal to the identity matrix.
<code>glMatrixMode</code>	An OpenGL command that makes the specified matrix the currently active matrix.
<code>glRotate</code>	An OpenGL command that performs the rotation transformation.
<code>glScale</code>	An OpenGL command that performs the scale transformation.
<code>glTranslate</code>	An OpenGL command that performs the translation transformation.
<code>GL_BACK</code>	An OpenGL constant for <code>glCullFace</code> .
<code>GL_CULL_FACE</code>	An OpenGL constant for <code>glEnable</code> .
<code>GL_DEPTH_BUFFER_BIT</code>	An OpenGL constant for <code>glClear</code> .
<code>GL_DEPTH_TEST</code>	An OpenGL constant for <code>glEnable</code> .
<code>GL_FRONT</code>	An OpenGL constant for <code>glCullFace</code> .
<code>GL_FRONT_AND_BACK</code>	An OpenGL constant for <code>glCullFace</code> .
<code>GL_MODELVIEW</code>	An OpenGL constant for <code>glMatrixMode</code> .
<code>GL_PROJECTION</code>	An OpenGL constant for <code>glMatrixMode</code> .
<code>GL_TEXTURE</code>	An OpenGL constant for <code>glMatrixMode</code> .
<code>Invalidate</code>	A Windows function that causes Windows to send <code>WM_PAINT</code> and returns true on success, false on failure.

## SUMMARY

Chapter 3 introduced three-dimensional rendering with these concepts:

1. Left to itself, OpenGL draws all surfaces whether they should show or not.
2. Turn on Depth Testing with `glEnable` to allow hidden surface removal.
3. The `ModelView` matrix represents the coordinate system of the scene or the coordinates of the viewer, whichever is convenient.



4. The ModelView transformations are rotation, transformation, and scale.
5. Use `glRotate` to rotate the scene counter-clockwise by a certain angle about a given axis.
6. Polygons have two faces. Cull back or front faces with `glCullFace`. Pass it `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.
7. Determine clockwise or counter-clockwise winding of a polygon by the order of specifying the vertices. By default the front face of a polygon has counter-clockwise winding.
8. Successive transformations before painting accumulate, and changing the order of the transformations can produce different results.
9. Use the Invalidate Windows API call to insure that a window redraws.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## Chapter 4 Perspective

Orthographic, or parallel, projection is fine for many purposes, but when the goal is realism in the graphics, *perspective* projection may be more appropriate. In the real world the fact that objects appear smaller when more distant is universally familiar. Artists recognize that elements of a scene appear to come together to a vanishing point at some great distance. This chapter shows how to produce this realistic effect and to position vertices in such an environment.

### ILLUSTRATION

With OpenGL the programmer can construct a (you guessed it!) matrix to project a scene onto the computer screen in a manner that simulates perspective. This process is called (appropriately) perspective projection. Consider two parallel lines as in the following code.

```
glBegin(GL_LINES);  
    glVertex3f(-0.5,0.1,-2.2);  
    glVertex3f(-0.5,15.0,-32.2);  
    glVertex3f(+0.5,0.1,-2.2);  
    glVertex3f(+0.5,15.0,-32.2);  
glEnd;
```

The two line segments run side by side off into the z distance, rising in the y direction. Figure 4-1 shows how they appear with perspective projection. Although reproducing this effect requires more code than given here, this is enough to show that the two lines are indeed parallel. The apparent convergence of the lines is due entirely to the perspective projection.



**Figure 4-1.** Parallel Lines in Perspective Projection

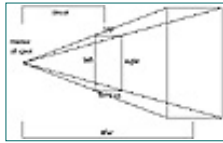
## COMMAND

To set up the perspective projection matrix, use this command:

```
procedure glFrustum(left, right, bottom, top, zNear, zFar:GLdouble); stdcall;
```

### Clipping Volume

What in the world is a frustum? Basically a frustum is a pyramid with the top chopped off. Since the clipping volume looks like a frustum with a rectangular base, lying horizontally, the command name makes sense. Figure 4-2 shows the shape of the clipping volume.



**Figure 4-2.** Clipping Volume from glFrustum.

### Parameters

Notice that the parameters for glFrustum are the same as the parameters for glOrtho. In general these parameters have a similar usage, but there are enough differences between them to warrant caution. Just as in glOrtho, left, right, bottom, and top define a rectangle. In glOrtho, however, the front rectangle and the rear rectangle are identical except for their z value, so only one rectangle specification is sufficient. In glFrustum, the near and the far rectangles are very different sizes, but the command only takes the values for the near rectangle.

How in the world does this command define the far rectangle? The answer lies in the usage of the znear and zfar parameters. The znear value is the distance from the viewer to the near face of the frustum, which is the “cut” across the pyramid. The znear distance must be expressed as a positive number. The zfar value is the distance to the farther face of the frustum (base of the pyramid), which must be a positive number larger than znear. In effect the viewer is at the apex of the pyramid, so znear and zfar are enough to define the base of the frustum from its front face, as Figure 4-2 suggests.

## DEPTH

### Previous Cube

Create a directory (under the OpenGL directory, of course) and name it Chapter.4. Under that directory create a directory named Multi. Copy the files from OpenGL\Chapter.3\Multi to OpenGL\Chapter.4\Multi. Open this project in Delphi and edit the FormCreate method in Multi.Pas. Replace

```
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

with

```
glMatrixMode(GL_PROJECTION);  
glFrustum(-1.0, 1.0, -1.0, 1.0, 2.0, 37.0);
```

This command change maps the same near face rectangle, but specifies the z range differently, as well as providing a different projection. The clipping volume starts at 2.0 units from the viewer and extends to 37.0 units from the viewer. In glOrtho znear and zfar just provided a range of values whose meaning was entirely up to the programmer. With parallel projection apparent sizes are not affected by distance from the viewer, so that distance is not a consideration. With perspective projection, distance from the viewer is a primary ingredient.

Compile and Run. Nothing but the green background shows because under the new coordinate system the cube was constructed entirely in front of the front clipping face. Use transformations to bring the cube

into view. Fill out the parameters as in Figure 4-3, click Ok, and select Transformation from the menu.



**Figure 4-3.** Parameters to Bring the Cube into View

Now the cube shows, but it looks a little strange. It is like the back is larger than the front. Test this theory with a rotation transformation. Set it up as in Figure 4-4.



**Figure 4-4.** Parameters to Rotate the Cube

Sure enough, the back of the cube appears larger than the front. Since this is the opposite of what perspective projection should produce, something must be wrong. This is a good time to test the coordinates.

## Test

Like the depth test in Chapter 3 draw two rectangles in the FormPaint method, replacing the current contents. The reason for the choice of z coordinates will soon become evident. Save the project as Depth.Dpr in OpenGL\Chapter.4\Depth and save the main unit as Depth1.Pas in the Depth directory also. Remove the Trans1 unit from the project, remove the uses Trans1 statement from the main unit, remove the Main Menu control and empty the contents of the menu item event handlers. Adjust the form's ClientHeight back to 340. Here is what FormPaint should look like:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
    if not openGLReady then
        exit;

    {green background}
    glClearColor(0.0,0.4,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

    glBegin(GL_POLYGON); {near face}
        glColor3f(0.5,0.0,0.0);
        glVertex3f(-0.4,-0.4,-2.4);
        glVertex3f(+0.4,-0.4,-2.4);
        glVertex3f(+0.4,+0.4,-2.4);
        glVertex3f(-0.4,+0.4,-2.4);
    glEnd;

    glBegin(GL_POLYGON); {back face}
        glColor3f(0.0,0.0,0.5);
        glVertex3f(-0.2,-0.2,-2.8);
        glVertex3f(+0.6,-0.2,-2.8);
        glVertex3f(+0.6,+0.6,-2.8);
        glVertex3f(-0.2,+0.6,-2.8);
    glEnd;

    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
        raise Exception.Create('Error in Paint'#13+
```

```
        gluErrorString(errorCode));  
end; {FormPaint}
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Save, Compile, and Run. Again the red rectangle is in front of the blue rectangle. But look again at the z coordinates of the two rectangles. The farther (blue) rectangle is more negative in the z direction than the near rectangle. In the orthographic projection clipping volume a more distant point is more positive (or less negative) than a nearer point, and the specifications of the cube vertices took that fact into account. So when the same vertices appeared in a perspective projection clipping volume whose z coordinates run in the opposite direction, naturally the cube was distorted.

### Better Cube

In Delphi load the previous project under Chapter.4, Multi.Dpr. Adapt the coordinates of the cube vertices to the new z direction reality. Remember that moving away from the viewer is to travel in the negative z direction, even though the znear and zfar parameters are positive and more positive respectively. Make the FormPaint method look like this:

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    if not openGLReady then  
        exit;  
  
    {green background}  
    glClearColor(0.0,0.4,0.0,0.0);  
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);  
  
    glBegin(GL_POLYGON); {near face}  
        glColor3f(0.5,0.0,0.0);  
        glVertex3f(-0.4,-0.4,+0.4);  
        glVertex3f(+0.4,-0.4,+0.4);  
        glVertex3f(+0.4,+0.4,+0.4);
```

```

        glVertex3f(-0.4,+0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {right face}
        glColor3f(0.3,0.3,0.8);
        glVertex3f(+0.4,-0.4,+0.4);
        glVertex3f(+0.4,-0.4,-0.4);
        glVertex3f(+0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {left face}
        glColor3f(0.5,0.0,0.5);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(-0.4,+0.4,+0.4);
        glVertex3f(-0.4,+0.4,-0.4);
    glEnd;

    glBegin(GL_POLYGON); {back face}
        glColor3f(0.0,0.0,0.5);
        glVertex3f(+0.4,-0.4,-0.4);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(-0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,-0.4);
    glEnd;

    glBegin(GL_POLYGON); {bottom face}
        glColor3f(0.5,0.5,0.0);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(+0.4,-0.4,-0.4);
        glVertex3f(+0.4,-0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {top face}
        glColor3f(0.3,0.0,0.1);
        glVertex3f(-0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,-0.4);
        glVertex3f(-0.4,+0.4,-0.4);
    glEnd;

    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
        raise Exception.Create('Error in      Paint'#13+
            gluErrorString(errorCode));
    end; {FormPaint}

```

For convenience, move the cube inside the clipping volume with a call to `glTranslatef` near the bottom of the `FormCreate` method. Place it there so it will only happen once.

```
glLoadIdentity();
glTranslatef(0.0,0.0,-2.4);

glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
end; {FormCreate}
```

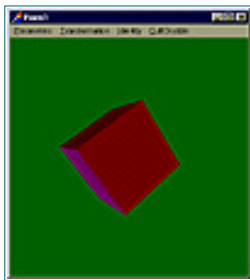
## MOVEMENT

### Rotation

Set up a transformation using the parameters of Figure 4-4, then perform the rotation. Notice that since the cube is so close to the front clipping face, the rotation sends a corner right outside the clipping volume. Now it is apparent that clipping works just like in the orthographic projection clipping volume. Notice too that back face culling also works as expected. The exposed interior of the cube reveals a green nothing. Select Cull Disable from the menu and behold the interior faces of the cube.

### Translation

In the parameter screen uncheck rotation and check translation. Set  $z$  to  $-0.1$  and the rest to  $0.0$ . Click Ok, and select Transformation from the menu repeatedly until the entire cube is inside the clipping volume. Now the image, shown in Figure 4-5, demonstrates clearly that the far side of the cube appears smaller than the near side.



**Figure 4-5.** Near and Far Sides of a Cube in Perspective

Next hold down `Alt+T` so that the keyboard starts repeating. Watch the cube retreat into the distance. Now that's perspective!

## MULTIPLE OBJECTS

### Matrix Stack

Most scenes built with OpenGL consist of more than just a single object on a plain background. Building various objects in the right places is an obvious way to build more complex scenes, but it can get tedious. Suppose the scene has a number of objects that are similar to each other and the programmer has to



describe them over and over. Isn't computer software supposed to relieve people of certain tiresome repetitive tasks? OpenGL provides a way!

Put a commonly used object in its own procedure or method. Build it around the origin (0.0,0.0,0.0) for ease of coding. Rotate it, translate it, even scale it, to put it in the right place, with the right orientation, and in the right shape. Then (this is important!) restore the ModelView matrix to its former condition and build and place another object.

What programming structure useful for saving and restoring something comes to mind? A stack would be nice. Indeed, as mentioned previously, OpenGL even provides a matrix stack. The commands for storing the current matrix on the stack, and retrieving it back, are:

```
procedure glPopMatrix; stdcall;  
procedure glPushMatrix; stdcall;
```

## Reuse an Object

Save the Mult project as Obj.Dpr in a new directory named Objects under Chapter.4. Remove Trans1.Pas from the project and save Multi.Pas as Objct.Pas in the new directory. Delete the menu object from the form, and empty all the menu item event handlers, and adjust the ClientHeight of the form back to 340.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

The object of choice these days is a cube. To make it available for reuse, collect the code to draw the cube and put it in a separate method.

```
procedure TForm1.Cube;  
begin  
    glBegin(GL_POLYGON); {near face}  
        glColor3f(0.5,0.0,0.0);  
        glVertex3f(-0.4,-0.4,+0.4);  
        glVertex3f(+0.4,-0.4,+0.4);  
        glVertex3f(+0.4,+0.4,+0.4);  
        glVertex3f(-0.4,+0.4,+0.4);  
    glEnd;  
  
    glBegin(GL_POLYGON); {right face}  
        glColor3f(0.3,0.3,0.8);  
        glVertex3f(+0.4,-0.4,+0.4);  
        glVertex3f(+0.4,-0.4,-0.4);  
        glVertex3f(+0.4,+0.4,-0.4);  
        glVertex3f(+0.4,+0.4,+0.4);  
    glEnd;  
  
    glBegin(GL_POLYGON); {left face}  
        glColor3f(0.5,0.0,0.5);  
        glVertex3f(-0.4,-0.4,-0.4);  
        glVertex3f(-0.4,-0.4,+0.4);  
        glVertex3f(-0.4,+0.4,+0.4);  
        glVertex3f(-0.4,+0.4,-0.4);  
    glEnd;  
  
    glBegin(GL_POLYGON); {back face}  
        glColor3f(0.0,0.0,0.5);  
        glVertex3f(+0.4,-0.4,-0.4);
```

```

        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(-0.4,+0.4,-0.4);
        glVertex3f(+0.4,+0.4,-0.4);
    glEnd;

    glBegin(GL_POLYGON); {bottom face}
        glColor3f(0.5,0.5,0.0);
        glVertex3f(-0.4,-0.4,+0.4);
        glVertex3f(-0.4,-0.4,-0.4);
        glVertex3f(+0.4,-0.4,-0.4);
        glVertex3f(+0.4,-0.4,+0.4);
    glEnd;

    glBegin(GL_POLYGON); {top face}
        glColor3f(0.3,0.0,0.1);
        glVertex3f(-0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,+0.4);
        glVertex3f(+0.4,+0.4,-0.4);
        glVertex3f(-0.4,+0.4,-0.4);
    glEnd;
end;

```

Declare the method at the end of the private section of the form:

```

private
    GLContext: HGLRC;
    glDC: HDC;
    errorCode: GLenum;
    openGLReady: boolean;
    oldw,
    oldh: integer;
    procedure ExceptionGL(Sender:TObject;E:Exception);
    procedure Cube;
public

```

## Make a Scene

This is enough information to build a scene with multiple copies of the cube. Make the FormPaint method look like this:

```

procedure TForm1.FormPaint(Sender: TObject);
begin
    if not openGLReady then
        exit;

    {background}
    glClearColor(0.6,0.8,0.8,0.0);
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

    glPushMatrix;
    glTranslatef(-0.7,0.7,-2.8);

```

```

Cube;
glPopMatrix;

glPushMatrix;
glTranslatef(1.9, -1.6, -5.4);
Cube;
glPopMatrix;

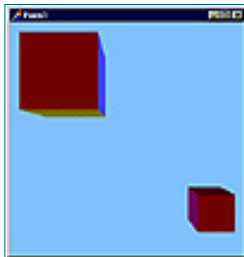
glFlush;

errorCode := glGetError;
if errorCode <> GL_NO_ERROR then
    raise Exception.Create('Error in Paint'#13+
        gluErrorString(errorCode));
end; {FormPaint}

```

Notice the sequence: push the ModelView matrix, translate to the desired location, call the routine that builds the object about the origin, and pop the matrix. Do this for each object in the scene; even if a given object is not repeated, it is usually simpler to construct at the origin, especially if it has symmetry. In this simple example only a translation appears before each cube, but any combination the three transformations could take its place.

Save, Compile, and Run. Behold a 3-D, two-object scene! Notice the marvelous new background color. Perhaps its name should be “Clear Blue Sky with Light Smog.” Figure 4-6 shows how it looks. The new color, though far from pretty, provides better contrast for the cubes.



**Figure 4-6.** Two Cubes in Perspective

## INDEPENDENT MOVEMENT

The techniques for creating multiple copies of an object and placing them in different places also lend themselves to moving the objects separately. To demonstrate, add some fields to the private section of the form type declaration:

```

x1, y1, z1, x2, y2, z2, xr1, yr1, zr1,
xr2, yr2, zr2, angle1, angle2, sign: GLfloat;

```

Initialize sign at the very bottom of the FormCreate method:

```

sign := 0.1;

```

### Translate

In the FormPaint method replace

```
glTranslatef(-0.7,0.7,-2.8);
```

with

```
glTranslatef(x1,y1,z1);
```

Also replace

```
glTranslatef(1.9,-1.6,-5.4);
```

with

```
glTranslatef(x2,y2,z2);
```

Now that the arguments in the `glTranslatef` calls are variables, code to manipulate the variables is appropriate. Using the Object Inspector set the form's `KeyPreview` property to true. On the Events page double click the `OnKeyPress` event and fill in the following event handler:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);  
var  
    ok: boolean;  
begin  
    ok := true;  
    case Key of  
        '+': sign := 0.1;  
        '-': sign := -0.1;  
        'X': x1 := x1+sign;  
        'x': x2 := x2+sign;  
        'Y': y1 := y1+sign;  
        'y': y2 := y2+sign;  
        'Z': z1 := z1+sign;  
        'z': z2 := z2+sign;  
    else  
        ok := false;  
    end; {case}  
    if ok then  
        Invalidate;  
end;
```

Whenever the user executes a keystroke the `FormKeyPress` method tests the character in the case statement and adjusts the variables accordingly. Pressing “+” or “-” sets the sign variable accordingly. The letters “x”, “y”, and “z” set the corresponding variables, with the capital letters setting x1, etc., and the lowercase letters setting x2, etc. If a desired letter came through, then a call to `Invalidate` causes the `FormPaint` method to fire again.

The sign variable starts with a positive value of one tenth. Pressing “-” changes it to negative one tenth and pressing “+” changes it back. This value adds to the other variables in response to the appropriate letter. Save, Compile, and Run. Start by pressing the minus key, then pressing and holding a capital “Z” until a cube comes into view. Since the variables are fields of the form, they start as 0.0 unless explicitly changed. That means both cubes remain

out of site at the origin until the user gives them a lot of negative z. Capital letters affect one cube and lower case letters affect the other cube. Experiment with pressing the desired keys and enjoy the power of moving the two cubes independently.

## Rotate

Being able to move two cubes independently represents a definite advance in the available OpenGL power, but rotating the cubes separately is an even greater advance. Better yet is the power to combine rotation and translation for each cube (independently). The variables have already been declared. After each `glTranslatef` goes a `glRotatef`:

```
glTranslatef(x1,y1,z1);  
glRotatef(angle1,xr1,yr1,zr1);
```

```
glTranslatef(x2,y2,z2);  
glRotatef(angle2,xr2,yr2,zr2);
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

The case statement in the FormKeyPress method needs an addition so that it can manipulate the rotation variables. The letters to check are upper and lower case “r,” “s,” and “t.”

```
'a': angle2 := angle2+10.0*sign;  
'r': xr2 := xr2+sign;  
's': yr2 := yr2+sign;  
't': zr2 := zr2+sign;  
'A': angle1 := angle1+10.0*sign;  
'R': xr1 := xr1+sign;  
'S': yr1 := yr1+sign;  
'T': zr1 := zr1+sign;
```

The letters “r,” “s,” and “t” set up the axis of rotation for each cube. Each axis is a *vector* (line segment with direction) from (0.0,0.0,0.0) toward the vertex constructed from (xr1,yr1,zr1) or (xr2,yr2,zr2). The letter “a” adjusts the amount of rotation (angle1 or angle2) about the appropriate vector. It would be nice to have a little help visualizing the axes of rotation. Add the following method to draw a representation of the coordinate axes along with the two rotation vectors:

```
procedure TForm1.Axes;  
begin  
    glLineWidth(3.0);  
    glBegin(GL_LINES);  
    glColor3f(0.0,0.0,0.0);  
  
    glVertex3f(+0.0,+0.0,+0.4);  
    glVertex3f(+0.0,+0.0,-0.4);  
  
    glVertex3f(+0.4,+0.0,+0.0);  
    glVertex3f(-0.4,+0.0,+0.0);  
end
```

```

        glVertex3f(+0.0,+0.4,+0.0);
        glVertex3f(+0.0,-0.4,+0.0);
    glEnd;

    glLineWidth(1.0);
    glBegin(GL_LINES);
        glColor3f(1.0,1.0,0.0);
        glVertex3f(0.0,0.0,0.0);
        glVertex3f(xr1,yr1,zr1);

        glColor3f(0.0,1.0,1.0);
        glVertex3f(0.0,0.0,0.0);
        glVertex3f(xr2,yr2,zr2);
    glEnd;
end;

```

Declare the new method at the bottom of the form's private section:

```

        procedure ExceptionGL(Sender:TObject;E:Exception);
        procedure Cube;
        procedure Axes;
public
    { Public declarations }
end;

```

Call the Axes method inside the FormPaint method, just before the call to glFlush. Translate it along the z axis to bring it inside the clipping volume. Translate it a little along the x and y axes so that the viewer will see it slightly from above and from the side, to make its three-dimensional nature more evident.

```

    glPushMatrix;
    glTranslatef(-0.6,-0.6,-2.8);
    Axes;
    glPopMatrix;

    glFlush;

```

Save, Compile, and Run. In addition to all the previous abilities of the program, it can now use the three new letters to rotate the cubes. Whew!

## Transformation Order Revisited

About the only thing left for this structure is to add scaling as well, but that would not add much to the OpenGL knowledge because the results are easy to predict. There is something else that can contribute significantly to the level of OpenGL understanding. The ability to switch back and forth between rotating before translation and rotating after translation reveals a dramatic difference between the two. Add the following to the case statement:

```

    'B': before := true;

```



```
'b': before := false;
```

The form declaration needs to include the before field:

```
errorCode: GLenum;  
before,  
openGLReady: boolean;  
oldw,  
oldh: integer;  
x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,  
xr2,yr2,zr2,angle1,angle2,sign: GLfloat;  
procedure ExceptionGL(Sender:TObject;E:Exception);  
procedure Cube;  
procedure Axes;
```

FormPaint needs an enhancement using the value of the before field to control whether the call to glRotatef happens before or after the call to glTranslatef:

```
glPushMatrix;  
if before then  
    glRotatef(angle1,xr1,yr1,zr1);  
glTranslatef(x1,y1,z1);  
if not before then  
    glRotatef(angle1,xr1,yr1,zr1);  
Cube;  
glPopMatrix;  
  
glPushMatrix;  
if before then  
    glRotatef(angle2,xr2,yr2,zr2);  
glTranslatef(x2,y2,z2);  
if not before then  
    glRotatef(angle2,xr2,yr2,zr2);  
Cube;  
glPopMatrix;
```

Again Save, Compile, and Run. Bring the cubes into view and position and orient them as you please, but be sure to include some rotation for each of them. Now press “B” and “b” to show the difference between rotation before translation and rotation after translation. The experiment with order of transformations at the end of Chapter 3 gave small distinctions, but this time the difference is exceedingly obvious. For example, a small rotation about the y axis changes the direction of the cube’s own personal z axis, so a subsequent translation along the z axis sends it off in a different direction than if the translation happens first. Try it and see.

## Another View

Frankly, transformations in OpenGL can be very confusing, so any aid to comprehension is welcome. For this purpose think of the translations as applying to the objects rather than the coordinate system. Then think of the transformations as happening in reverse order. When you push and pop the ModelView matrix several

times, treat this reverse order concept separately between each push and pop.

Different people may need to consider OpenGL transformations in different ways to understand them. Possibly this alternate view will be the best one for many readers of this book.

## DEFINITIONS

Frustum	A geometric figure formed when the top of a cone or pyramid is cut off by a plane parallel to the base.
Perspective projection	A projection in which the apparent size of each object is adjusted for distance from the viewer.

[Previous](#) [Table of Contents](#) [Next](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## **IDENTIFIER REFERENCE**

glFrustum	An OpenGL command that sets up perspective projection and the clipping volume.
glPopMatrix	An OpenGL command that saves the current OpenGL matrix on the matrix stack.
glPushMatrix	An OpenGL command that retrieves a matrix from the matrix stack and makes it the current OpenGL matrix.

## **SUMMARY**

Here are the high points of Chapter 4:

1. The shape of the clipping region for perspective projection is a frustum, which is a pyramid with the top chopped off.
2. The command is glFrustum; the parameters are the same as for glOrtho, but the znear and zfar parameters have different effects and must be positive.
3. To move away from the viewer, deeper into the screen, is to move in the negative z direction.
4. The effects of clipping and back face culling are the same as with glOrtho.
5. When viewed in perspective, parallel lines appear to converge in the distance, and objects appear smaller as they retreat into the distance.
6. Use glPushMatrix to save the ModelView matrix and glPopMatrix to restore it. Placing transformations between those calls allows for

independent positioning and movement of objects in a scene.

**7.** Rotation changes the orientation of the axes so that a subsequent translation may go off in a different direction. When planning a positioning or movement within a scene, be careful about the order of the transformations.

**8.** OpenGL transformations seem to take effect in the reverse order of their specification in program code.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

# **Chapter 5 Lighting**

In the real world things are visible because of light. All light comes from various sources, usually identifiable, but even when the source is unknown, the light was actually emitted from somewhere. Not only does the visibility of everything (except those objects that emit their own light) depend on light emitted from somewhere and reflected to the viewer, but the detailed appearance of each object depends on properties of the material composing the illuminated surface. A realistic graphics system should incorporate these facts. This chapter explores lighting and material properties.

Building on the work of Chapter 4, Save Obj.Dpr as ALight.Dpr in OpenGL\Chapter.5\Ambient. Save the main unit as ALight1.Pas.

## **ENABLE LIGHTING**

One of the great features of OpenGL is the ability to produce realistic effects through simulation of lighting. In the code in the first four chapters, OpenGL provided the equivalent of lighting so that the objects were visible. Now is the time to provide lighting specifications directly. Here is a necessary OpenGL constant:

```
GL_LIGHTING = $0B50;
```

At the bottom of the FormCreate method enable lighting:

```
glEnable(GL_LIGHTING);
```

Save, Compile and Run. Maneuver one of the cubes into the viewing volume (clipping region). Ugh! The red square is black! Rotate the cube so that two more faces show at least a little. They are black, too! It is difficult to distinguish one face from another because they have exactly the same color. Now that the program is taking responsibility for making the objects in the scene visible, OpenGL is taking no responsibility at all. Obviously the program needs to do more than just enable lighting.

Basically three kinds of lighting are available through OpenGL, *Ambient*, *Diffuse*, and *Specular*. This chapter will examine each of them in turn.

## AMBIENT LIGHT

### Definition

Ambient light has no apparent source, but seems to come from all directions. In the real world, all light has a source, but some light may be reflected and scattered from several surfaces before finally reaching the viewer. The source of such light may be difficult or impossible to trace. This kind of light is called ambient light. OpenGL simulates this effect by providing illumination equally from all directions. It can be associated with a particular light or it can have no specific source, just magically appearing (at the programmer's command).

### Commands and Constants

The command for defining a light is a variation of `glLight`. Use the “f” suffix to use `GLfloat` parameters. Many commands, including ones already discussed, have a vector version, indicated by a “v” suffix. The vector version substitutes a pointer to an array in place of several scalar parameters. The `glLight` command only takes three parameters in any version, so only the vector version meets the current need. The `glLightModel` command takes three parameters and also has a vector version suitable for the occasion. Then there is the `glColorMaterial` command, which introduces the idea of material properties. The relevant declarations follow. First a necessary type:

```
PGLfloat = ^GLfloat;
```

Here are some useful constants:

```
GL_LIGHT_MODEL_AMBIENT = $0B53;
GL_COLOR_MATERIAL      = $0B57;
GL_AMBIENT              = $1200;
GL_LIGHT0               = $4000;
GL_LIGHT1               = $4001;
GL_LIGHT2               = $4002;
GL_LIGHT3               = $4003;
GL_LIGHT4               = $4004;
GL_LIGHT5               = $4005;
GL_LIGHT6               = $4006;
GL_LIGHT7               = $4007;
```

Finally, the commands:

```
procedure glColorMaterial(face,mode:GLenum); stdcall;
procedure glLightModelfv(pname:GLenum;params:PGLfloat); stdcall;
procedure glLightfv(light,pname:GLenum;params:PGLfloat); stdcall;
```

Consider the parameters in the commands. The `face` parameter can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. Right now `GL_FRONT` is sufficient. The `mode` parameter specifies the kind of lighting to which the material property relates. The example under development uses `GL_AMBIENT`. The `params` pointer could be pointing to a single `GLfloat`, or the first element of an array of them. While this is very C-like, Delphi can handle it nicely. The `pname` parameter identifies the purpose for two of the commands. The command for the simplest method of implementing ambient light is `glLightModel`. Pass it

GL\_LIGHT\_MODEL\_AMBIENT for the first parameter.

## THE EASY WAY

Lighting introduces a new level of complexity to the specification of an object's appearance. A simple color specification does not tell the whole story about how a surface responds to light. *Material* properties allow a more complete way to describe responses to light. The present example avoids some of the complexity by having the material color track the current color by means of the glColorMaterial command. This is one of the options that requires enabling before use.

At the bottom of the FormCreate method, after the enabling of lighting , initialize the array and place some of the new commands with the appropriate arguments.

```
AmbientDef[0] := 1.0; {red}
AmbientDef[1] := 1.0; {green}
AmbientDef[2] := 1.0; {blue}
AmbientDef[4] := 1.0; {alpha}
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @AmbientDef);
```

In this version of the glLightModel command, using GL\_LIGHT\_MODEL\_AMBIENT, means the array contains the color specification of the ambient light. Here, giving maximum intensity (1.0) to the red, green, and blue components specifies a white ambient light. A later chapter covers alpha; just set it to 1.0 for now. Declare the array in the private section of the form.

```
AmbientDef: array[0..3] of GLfloat;
```

Save, Compile, and Run. Now the cubes look like they did before. So what did the new commands accomplish? One benefit is control. Experiment with decreasing the red, green, and blue components of AmbientDef (positions 0, 1, and 2 in the array). Reducing the overall intensity of the ambient light or giving it a color bias produces effects that may be desirable in some applications.

[Previous](#) [Table of Contents](#) [Next](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## THE HARD WAY

A more flexible approach uses the `glLight` command. Make a new directory named `OpenGL\Chapter.5\Ambient0`. In that directory save the program as `Amb0.Dpr` and the main unit as `Ambient0.Pas`. Change the bottom of the `FormCreate` method to this:

```
glEnable(GL_LIGHTING);
AmbientDef[0] := 1.0; {red}
AmbientDef[1] := 1.0; {green}
AmbientDef[2] := 1.0; {blue}
AmbientDef[3] := 1.0; {alpha}

glLightfv(GL_LIGHT0, GL_AMBIENT, @AmbientDef);
glEnable(GL_LIGHT0);
```

This approach defines a particular light (`LIGHT0`) instead of just setting up a light model. At least eight lights are available; some implementations of OpenGL may allow more. Notice that you must turn on any light you want to use. This example switches on `LIGHT0` with `glEnable(GL_LIGHT0)`. To use a light, the simple color tracking of `glColorMaterial` is not adequate. So instead of `glColor` commands inside the `Cube` method, use `glMaterial` commands. A vector version is most convenient, so define an array at the top of `Cube`. For this example there is no attempt to produce the same colors as before.

```
procedure TForm1.Cube;
var
    ambient: array[0..3] of GLfloat;
begin
    fillchar(ambient, sizeof(ambient), 0);
    ambient[3] := 1.0; {alpha}
```



```

glBegin(GL_POLYGON); {near face}
    ambient[0] := 0.5; {red}
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glVertex3f(-0.4, -0.4, +0.4);
    glVertex3f(+0.4, -0.4, +0.4);
    glVertex3f(+0.4, +0.4, +0.4);
    glVertex3f(-0.4, +0.4, +0.4);
glEnd;

glBegin(GL_POLYGON); {right face}
    ambient[1] := 0.5; {green} {total=yellow}
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glVertex3f(+0.4, -0.4, +0.4);
    glVertex3f(+0.4, -0.4, -0.4);
    glVertex3f(+0.4, +0.4, -0.4);
    glVertex3f(+0.4, +0.4, +0.4);
glEnd;

glBegin(GL_POLYGON); {left face}
    ambient[0] := 0.0; {red} {total=green}
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glVertex3f(-0.4, -0.4, -0.4);
    glVertex3f(-0.4, -0.4, +0.4);
    glVertex3f(-0.4, +0.4, +0.4);
    glVertex3f(-0.4, +0.4, -0.4);
glEnd;

glBegin(GL_POLYGON); {back face}
    ambient[2] := 0.5; {blue} {total=turquoise}
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glVertex3f(+0.4, -0.4, -0.4);
    glVertex3f(-0.4, -0.4, -0.4);
    glVertex3f(-0.4, +0.4, -0.4);
    glVertex3f(+0.4, +0.4, -0.4);
glEnd;

glBegin(GL_POLYGON); {bottom face}
    ambient[1] := 0.0; {green} {total=blue}
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glVertex3f(-0.4, -0.4, +0.4);
    glVertex3f(-0.4, -0.4, -0.4);
    glVertex3f(+0.4, -0.4, -0.4);
    glVertex3f(+0.4, -0.4, +0.4);
glEnd;

glBegin(GL_POLYGON); {top face}
    ambient[0] := 0.5; {red} {total=magenta/magenta}
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glVertex3f(-0.4, +0.4, +0.4);
    glVertex3f(+0.4, +0.4, +0.4);

```

```

        glVertex3f(+0.4,+0.4,-0.4);
        glVertex3f(-0.4,+0.4,-0.4);
    glEnd;
end;

```

Save, Compile, and Run. Bring the cubes into view and move them around. Notice something rather strange. A side of the cube facing directly toward the viewer is brighter than the same side with a significantly different angle to the viewer.

That is not the behavior of ambient light. Perhaps it has been set too bright. Ambient light is often a minor part of the total light in a scene. In the FormCreate method cut it down as follows:

```

AmbientDef[0] := 0.3; {red}
AmbientDef[1] := 0.3; {green}
AmbientDef[2] := 0.3; {blue}
AmbientDef[3] := 1.0; {alpha}

```

Save, Compile, and Run. Manipulate the scene as before. Alas, there is little noticeable difference between ambient light at 30% and ambient light at 100%. Something is very wrong.

Actually, LIGHT0 has a special characteristic different from LIGHT1, LIGHT2 or any of the others. LIGHT0 by default has a white light diffuse component and a white light specular component. The default ambient component is black, but the present code replaced the default value. To see the effect of ambient light alone requires replacing the diffuse and specular parts (with black). Make these changes to the FormCreate method:

```

glEnable(GL_LIGHTING);
fillchar(AmbientDef,SizeOf(AmbientDef),0);
glLightfv(GL_LIGHT0,GL_DIFFUSE,@AmbientDef);
glLightfv(GL_LIGHT0,GL_SPECULAR,@AmbientDef);

AmbientDef[0] := 0.3; {red}
AmbientDef[1] := 0.3; {green}
AmbientDef[2] := 0.3; {blue}
AmbientDef[3] := 1.0; {alpha}

glLightfv(GL_LIGHT0,GL_AMBIENT,@AmbientDef);
glEnable(GL_LIGHT0);

```

The new constants are:

```

GL_DIFFUSE = $1201;
GL_SPECULAR = $1202;

```

Now Save, Compile, and Run. Bring the cubes into view. What a difference! The cubes are rather dim, which is natural for only 30% ambient illumination, but most importantly, they are uniformly dim. This is how ambient light should appear!

Can OpenGL handle bright white ambient light, or did the earlier supposition that the ambient light was set out of natural range have merit? Find out by replacing all the 0.3 values in AmbientDef with 1.0. Save, Compile, Run, and move cubes around. Now uniform illumination abounds. Though it is quite bright, the ambient light appears evenly distributed regardless of angle, which is the right behavior for ambient light.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

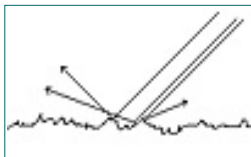
Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## **DIFFUSE LIGHT**

### **Definition**

Imagine a lamp lighting a large room. Much of the light bounces off many surfaces before reaching a viewer. While the light actually came from the lamp, it seems simply to be everywhere. This is the ambient portion of the light. Obviously, however, some of the light proceeds directly from the source to the surface of some object in the room and reflects directly to the viewer. However, unless the surface is a mirror, the light tends to scatter rather than bounce off at a predictable angle. In the real world nothing is perfectly smooth, but some things have been polished enough that the surface irregularities are too small to matter. Such surfaces are “shiny.” Most surfaces have irregularities large enough to scatter most light as in Figure 5-1. These surfaces are “flat.” Unless the surface is white, it will absorb some colors from the light and reflect the rest, thus defining its color in terms of the portion of the light it reflects. This scattering effect determines the diffuse portion of the light. Not only does OpenGL allow the definition of at least eight light sources, but it also allows independent specification of the ambient portion and diffuse portion of each light.



**Figure 5-1.** Diffuse Reflection from a Flat Surface

### **Positional Lamps**

Save the project as Difu.Dpr in a new directory called OpenGL\Chapter.5\Dif1 and save the main unit as Diffus.Pas. For the lamp under development, most

light shines directly on an object from the lamp (the diffuse portion), and a lesser contribution comes from multiple reflections (the ambient portion). Prepare to change the AmbientDef assignments in the FormCreate method as follows, but do not type it in just yet:

```
glEnable(GL_LIGHTING);
AmbientDef[0] := 0.3; {red}
AmbientDef[1] := 0.3; {green}
AmbientDef[2] := 0.3; {blue}
AmbientDef[3] := 1.0; {alpha}
```

While bland, homogenized arrays like this provide a simple, fairly language-independent way of passing data to the OpenGL DLL's, Delphi allows more meaningful structures that use memory exactly the same way. The following type declaration of a record structure is more intuitive and does not require the programmer to memorize the order of the colors:

```
TLightDesc=record
    red,
    green,
    blue,
    alpha: GLfloat;
end;
```

Wait! Don't type that in either! Arrays use contiguous memory, but record structures may have "holes" in the data due to memory alignment by the compiler. This particular record structure is safe (for the moment) because each of the elements is four bytes in size, but to guarantee that the record memory layout is exactly the same as the array memory layout, use the **packed** reserved word and place the declaration just above the form's class declaration:

```
type
    TLightDesc=packed record
        red,
        green,
        blue,
        alpha: GLfloat;
    end;
TForm1 = class(TForm)
```

Of course the type for the AmbientDef field within the form declaration should become:

```
AmbientDef: TLightDesc;
```

Now the new assignments to AmbientDef in the FormCreate method can proceed:

```
glEnable(GL_LIGHTING);
```

```

with AmbientDef do
begin
    red := 0.3;
    green := 0.3;
    blue := 0.3;
    alpha := 1.0;
end;

```

The key distinction between diffuse light and ambient light is that the source of diffuse light has a position. Therefore a demonstration of diffuse light requires the ability to set the position of the lamp. Accordingly add these lines to the case statement in the FormKeyPress method:

```

    'l': x1 := x1+sign;
    'm': y1 := y1+sign;
    'n': z1 := z1+sign;

```

In the form's class declaration add these "L" variable declarations to describe the lamp position:

```

    x1,y1,z1,
    x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
    xr2,yr2,zr2,angle1,angle2,sign: GLfloat;
    AmbientDef: TLightDesc;
    LightPlace: TLightPlace;

```

LightPlace is also new and needs a type declaration.

```

type
    TLightDesc=packed record
        red,
        green,
        blue,
        alpha: GLfloat;
    end;
    TLightPlace=packed record
        x,
        y,
        z,
        w: GLfloat;
    end;
TForm1 = class(TForm)

```

In the FormPaint method, place some code to manipulate the light position:

```

    glPushMatrix;
    if before then
        glRotatef(angle2,xr2,yr2,zr2);
        glTranslatef(x2,y2,z2);
    if not before then
        glRotatef(angle2,xr2,yr2,zr2);

```

```

Cube;
glPopMatrix;
glPushMatrix;
glTranslatef(x1,y1,z1);
glLightfv(GL_LIGHT0, GL_POSITION, @LightPlace);
glPopMatrix;

```

Save, Compile, Run, and bring a cube barely into view, then go two keystrokes more down the negative z-axis. Move the cube near the upper left corner of the form. The cube appears very bright, but that will change as the lamp moves.

Press the minus key, then press the lowercase “n” repeatedly. After about five such presses, a slight difference is noticeable. As you continue to advance the lamp down the negative z-axis, watch the shading develop on the cube!

Now it should be possible to have a uniformly colored cube that still shows its three-dimensional nature by its shading in a diffuse light source. In the Cube method change the type of the ambient local variable:

```

procedure TForm1.Cube;
var
    ambient: TLightDesc;
begin

```

Remove all assignments to ambient inside the Cube method and replace them with one set of assignments as follows:

```

procedure TForm1.Cube;
var
    ambient: TLightDesc;
begin
    ambient.red := 0.2;
    ambient.green := 0.5;
    ambient.blue := 0.8;
    ambient.alpha := 1.0;
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glBegin(GL_POLYGON); {near face}

```

Save, Compile, Run, and perform movements as before. Around the 22<sup>nd</sup> “n” press the shading of the cube is more than sufficient to distinguish the faces. Enjoy! This is tremendous progress from the first step of making a colored background.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

### Different Material Colors

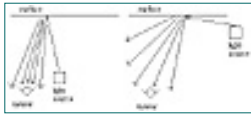
Various material properties can be set independently, including colors for ambient reflectivity, specular reflectivity, and diffuse reflectivity. Allowing the surface to reflect ambient light differently from diffuse light provides for some interesting effects. Adjust the top of the Cube method as follows:

```
procedure TForm1.Cube;  
var  
    diffuse,  
    ambient: TLightDesc;  
begin  
    ambient.red := 0.2;  
    ambient.green := 0.5;  
    ambient.blue := 0.8;  
    ambient.alpha := 1.0;  
    diffuse.red := 0.8;  
    diffuse.green := 0.5;  
    diffuse.blue := 0.2;  
    diffuse.alpha := 1.0;  
    glMaterialfv(GL_FRONT, GL_DIFFUSE, @diffuse);  
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);  
    glBegin(GL_POLYGON); {near face}
```

Save, etc. as before. This version more clearly demonstrates the ambient contribution and the diffuse contribution from the light source. At first the diffuse light reflection color clearly dominates. As the light advances down the negative z-axis until it is almost past the near face of the cube, ambient reflection clearly shows one color, and diffuse reflection another color. When the light moves a little more, the ambient reflection color will completely dominate. Why does the diffuse contribution wane before the light is past the



cube? Another behavior of real light reflecting off real surfaces is the fact that as the angle from the light source to the surface flattens, less of the diffuse light reflects to the viewer. Figure 5-2 illustrates light shining on a surface from an acute angle (left) and from a more obtuse angle (right). The right side of the illustration shows less of the scattered (diffuse) light reaching the viewer. OpenGL reproduces this behavior nicely.



**Figure 5-2.** Diffuse Light at Different Angles.

So far the exercises have depended on the fact that LIGHT0 has a white diffuse component by default. The next exercise illustrates colored light on surfaces of different color. Accordingly, at the bottom of the FormCreate method, make the light pure red and explicitly define the diffuse component.

```

glEnable(GL_LIGHTING);
with AmbientDef do
begin
    red := 0.6;
    green := 0.0;
    blue := 0.0;
    alpha := 1.0;
end;
LightPlace.w := 1.0;

glLightfv(GL_LIGHT0, GL_AMBIENT, @AmbientDef);
glLightfv(GL_LIGHT0, GL_DIFFUSE, @AmbientDef);
glEnable(GL_LIGHT0);
end; {FormCreate}

```

Save ... well, you know the drill. Now, since the only light is red, that is the only light reflected, so the cube looks pure red. What will happen if the material property has no red component? Find out by removing the red at the top of the Cube method. At the same time return the cube to having only one color response to the light.

```

procedure TForm1.Cube;
var
    ambient: TLightDesc;
begin
    ambient.red := 0.0;
    ambient.green := 0.5;
    ambient.blue := 0.8;
    ambient.alpha := 1.0;
    glMaterialfv(GL_FRONT, GL_DIFFUSE, @ambient);
    glMaterialfv(GL_FRONT, GL_AMBIENT, @ambient);
    glBegin(GL_POLYGON); {near face}

```

Repeat the steps and see the result. The cube is black. Only red light is

available. The cube reflects no red light. Therefore the cube reflects no virtually light.

## Spotlights

The light source discussed so far can be a spotlight. First, this naked light needs some clothes. For this stage of the learning adventure, dress it in a ... cube. Save the project as Spot.Dpr in OpenGL\Chapter.5\Spot, and save the main unit as Spot1.pas in that directory. Give the Cube method the ability to set its material color from a passed parameter. Adjust its declaration inside the form class declaration and add a few fields.

```
CubeColor,  
SpotColor,  
DiffDef,  
AmbientDef: TLightDesc;  
LightPlace: TLightPlace;  
procedure ExceptionGL(Sender:TObject;E:Exception);  
procedure Cube(matColor:TLightDesc);  
procedure Axes;
```

Now change its implementation code to handle the parameter.

```
procedure TForm1.Cube(matColor:TLightDesc);  
begin  
    glMaterialfv(GL_FRONT,GL_AMBIENT,@matColor);  
    glMaterialfv(GL_FRONT,GL_DIFFUSE,@matColor);  
    glBegin(GL_POLYGON); {near face}
```

Set the color records in the FormCreate method.

```
glEnable(GL_LIGHTING);  
with AmbientDef do  
begin  
    red := 0.3;  
    green := 0.3;  
    blue := 0.3;  
    alpha := 1.0;  
end;  
with DiffDef do  
begin  
    red := 1.0;  
    green := 1.0;  
    blue := 1.0;  
    alpha := 1.0;  
end;  
with CubeColor do  
begin  
    red := 0.5;  
    green := 0.0;
```

```
    blue := 0.8;
    alpha := 1.0;
end;
with SpotColor do
begin
    red := 0.0;
    green := 0.5;
    blue := 0.5;
    alpha := 1.0;
end;
```

```
LightPlace.w := 1.0;
```

```
glLightfv(GL_LIGHT0, GL_AMBIENT, @AmbientDef);
glLightfv(GL_LIGHT0, GL_DIFFUSE, @DiffDef);
```

Make the two calls to Cube in the FormPaint method use the parameter.

```
Cube(CubeColor);
```

Also adjust the code that positions the light Be sure to make this cube smaller with glScalef.

```
glPushMatrix;
glScalef(0.33, 0.33, 0.33);
glTranslatef(x1, y1, z1);
glLightfv(GL_LIGHT0, GL_POSITION, @LightPlace);
Cube(SpotColor);
glPopMatrix;
```

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

Now, go back to the bottom of the `FormCreate` method and initialize some positions so you will not have to do so much work to position the objects. Note that the “l” in the variable names is an ell not a one.

```
z2 := -2.7;  
x2 := -0.7;  
y2 := 0.7;  
z1 := -6.7;  
x1 := -1.8;  
y1 := -2.1;  
end; {FormCreate}
```

Since the `glScale` command affects translation (and rotation), as well as the object size, adjust the case statement in the `FormKeyPress` method.

```
'l': x1 := x1+3.0*sign;  
'm': y1 := y1+3.0*sign;  
'n': z1 := z1+3.0*sign;
```

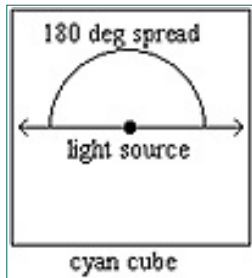
Save, Compile, and Run. (You are remembering not to run from the IDE, aren't you?) The magenta (or purple) cube and the cyan (blue-green, turquoise, whatever) “spotlight” are visible from the start. Use the “l,” “m,” and “n” keys to move the spotlight around. The center of the small cyan cube tracks the position of the light source so that the cube appears to BE the light source. Notice the effect different light positions have on illumination of the magenta cube.

A really interesting effect is the illumination of the small cube from the inside, so that the sides of the cube appear to be translucent. This brings up an important fact. OpenGL knows nothing about shadows. When you “shine” an OpenGL light source on an OpenGL object, you see a very good representation of real illumination. However, if you interpose another seemingly solid OpenGL

object between the light source and the first object, the more distant object receives illumination just as if there were nothing in the way. The consequence of this fact is that if you want shadows, you will have to program them yourself. Do not despair; it can be done. More about that comes later.

Notice that only half of the cyan cube lights up. By default an OpenGL diffuse light source has a 180 degree spread as shown in figure 5-3, which includes the cyan cube that represented the spotlight. A real spotlight is more focused, and OpenGL permits this effect with an appropriate constant in a familiar command.

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, cutoff);
```

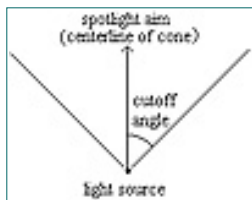


**Figure 5-3.** Default Spread of a Diffuse Light Source.

The new constant has this value:

```
GL_SPOT_CUTOFF = $1206;
```

Express the value of the cutoff variable in degrees. Think of the light spreading from a spotlight as a cone of light. The angle from the center of the cone to the edge of the cone is the cutoff angle, as shown in figure 5-4.



**Figure 5-4.** The Spotlight Cutoff Angle.

Add the cutoff field to the “ell” line in the form’s class declaration.

```
x1,y1,z1,xr1,yr1,zr1,cutoff,
x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
xr2,yr2,zr2,angle1,angle2,sign: GLfloat;
```

Use the cutoff field in the FormCreate method.

```
glLightfv(GL_LIGHT0, GL_AMBIENT, @AmbientDef);
glLightfv(GL_LIGHT0, GL_DIFFUSE, @DiffDef);
cutoff := 55.0;
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, cutoff);
glEnable(GL_LIGHT0);
```

Save, Compile, Run. Move the spotlight around with “l,” “m,” and “n.” Notice that you must move the spotlight closer to the magenta cube in the x and y

directions or farther in the positive z direction in order to illuminate the magenta cube.

## Directional Spotlights

A feature still missing from the spotlight illustrated thus far in the chapter is the ability to aim it. What kind of spotlight is it that you can't ... spot? One thing needed for that purpose is to add some more "ell" fields in the form class declaration.

```
x1,y1,z1,xr1,yr1,zr1,angle1,cutoff,  
x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,  
xr2,yr2,zr2,angle1,angle2,sign: GLfloat;
```

The `xr1`, `yr1`, and `zr1` fields set up the axis of rotation for the cyan cube and the aiming direction of the spotlight. Similarly the `angle1` field is the number of degrees of rotation about the axis for both the cube and the aiming of the spotlight. The default aim of a spotlight is a vector from the origin (0.0,0.0,0.0) to (0.0,0.0,-1.0) down the negative z-axis, which points toward the far face of the cube at its starting position. Therefore aiming the far face of the cube effectively aims the spotlight.

To manipulate those fields, add to the case statement in the `FormKeyPress` method.

```
'k': angle1 := angle1+30.0*sign;  
'o': xr1 := xr1+3.0*sign;  
'p': yr1 := yr1+3.0*sign;  
'q': zr1 := zr1+3.0*sign;
```

Add a new type:

```
TLightDesc=packed record  
  red,  
  green,  
  blue,  
  alpha: GLfloat;  
end;  
TLightPlace=packed record  
  x,  
  y,  
  z,  
  w: GLfloat;  
end;  
TPosition=packed record  
  x,  
  y,  
  z: GLfloat;  
end;
```

Add a new field of that type in the form's class declaration:

```
AmbientDef: TLightDesc;  
LightPlace: TLightPlace;  
SpotDirect: TPosition;
```

Initialize SpotDirect in the FormCreate method to match the default value of a spotlight direction. Except for the z field, the zeros that come automatically from creating an instance of the form are sufficient.

```
with SpotColor do  
begin  
    red := 0.0;  
    green := 0.5;  
    blue := 0.5;  
    alpha := 1.0;  
end;  
SpotDirect.z := -1.0;  
LightPlace.w := 1.0;
```

Finally, to the FormPaint method add the use of SpotDirect and the rotational “ell” fields.

```
glPushMatrix;  
glScalef(0.33,0.33,0.33);  
if before then  
    glRotatef(angle1,xr1,yr1,zr1);  
glTranslatef(xl,yl,zl);  
if not before then  
    glRotatef(angle1,xr1,yr1,zr1);  
glLightfv(GL_LIGHT0,GL_POSITION,@LightPlace);  
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,@SpotDirect);  
Cube(SourceColor);
```

Save, Compile, Run, and press the plus key. Press the lower-case “o” to set the axis of rotation for the spotlight in the positive x direction. Press and hold the lower-case “k” to produce rotation of the spotlight about the x-axis. Watch the change in lighting on the magenta cube as the spotlight direction changes. That is not a bad effect for just a little code. There is certainly a lot of power in simple OpenGL commands! Notice the brilliance the face of the cyan cube from which the light seems to emanate.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## EMISSION

### Definition

Here is a profound revelation. Light sources emit light. They glow. A surface that glows looks different from a surface that merely reflects light. OpenGL provides a way to produce this effect. Notice that the subject is the appearance of surfaces. Therefore the appropriate command is `glMaterialfv` rather than `glLightfv`. The new constant is:

```
GL_EMISSION = $1600;
```

The purpose of the use of `GL_EMISSION` is to intensify the surface color so that it seems to glow. It does not add to the light shed on other surfaces.

### Implementation

Save the spot project as `Emission.Dpr` in `OpenGL\Chapter.5\Emit`. Save the main unit as `Emit1.Pas` in the same directory. Since the use of `GL_EMISSION` applies to the material properties of a face in the spotlight cube, some special purpose code is necessary for this illustration. In the `Cube` method surround the `{back face}` part with material commands. The `if` statement tests for a red component so the magenta cubes will not be affected, but the cyan cube will be affected. Remember this is just special code for a brief illustration.

```
if matColor.red=0.0 then
    glMaterialfv(GL_FRONT, GL_EMISSION, @matColor);
glBegin(GL_POLYGON); {back face}
    glVertex3f(+0.4, -0.4, -0.4);
    glVertex3f(-0.4, -0.4, -0.4);
    glVertex3f(-0.4, +0.4, -0.4);
```



```

        glVertex3f(+0.4,+0.4,-0.4);
    glEnd();
    glMaterialfv(GL_FRONT, GL_EMISSION, @black);

```

The black emission statement confines the effect of the previous emission statement to the one face, otherwise everything would eventually end up with GL\_EMISSION. Of course the black field requires declaration within the form's class declaration. Happily it will automatically be initialized to all zeros.

```

        black: TLightDesc;
        LightPlace: TLightPlace;
        SpotDirect: TPosition;

```

Frankly, since the back face of the cyan cube already almost seems to glow, using matColor with GL\_EMISSION will have no noticeable effect. Replace it with the emit field.

```

if matColor.red=0.0 then
    glMaterialfv(GL_FRONT, GL_EMISSION, @emit);

```

Declare emit in the form's class declaration:

```

        emit: TLightDesc;
        black: TLightDesc;
        LightPlace: TLightPlace;
        SpotDirect: TPosition;

```

Initialize emit in the FormCreate method:

```

with emit do
begin
    red := 0.6;
    green := 1.0;
    blue := 1.0;
    alpha := 1.0;
end;
end; {FormCreate}

```

Save, Compile, Run, press plus, and press little "o." Hold down little "k" until the back side partially faces the viewer. Notice the intensity of the back face. This illustrates a feature that is seldom used, but you may find handy on special occasions.

To see why you needed glMaterialfv(GL\_FRONT, GL\_EMISSION, @black), comment it out, re-compile, and re-run. GL\_EMISSION affected everything. Ugh! The effect of a glMaterial command remains until explicitly changed.

Remember that emission is just a surface effect in OpenGL; it is not one of the three kinds of light that come from a light source. The next topic deals with such a type of light.

# SPECULAR LIGHT

## Definition

The third component of light from a light source is *specular* light. Ambient light has been scattered from many surfaces and seems to come from all directions. Diffuse light comes from a specific direction (the source, of course), but scatters from surfaces, giving a soft, “flat” appearance to surfaces. Ambient and Diffuse components of a light source commonly share the same color. Specular light is also directional, but much “tighter” in reflection than diffuse light, often with a different color. Specular light more closely follows the classical rule of reflection wherein the angle of reflection equals the angle of incidence, as in Figure 5-5.



**Figure 5-5.** Specular Reflection Angles.

As you might expect, the result of reflection of specular light depends greatly on the material properties of the surface. You may have noticed that in the real world surfaces that are smooth and “shiny” often show a small bright spot in bright, direct light. The bright spot is known as the specular highlight. With OpenGL you can give a surface the appearance of being smooth and shiny by producing such a bright spot. While the color of the surface with respect to ambient and diffuse light may vary, the color of the specular highlight tends toward white.

## Implementation

To demonstrate the specular component of a light source, load the Spot.Dpr project from OpenGL\Chapter.5\Spot. Create a new directory called OpenGL\Chapter.5\Spec and save the project as Specular.Dpr into the new directory. Save its main unit as Spec1.Pas in the same directory. At the bottom of the FormCreate method add a new glLightfv command.

```
glLightfv(GL_LIGHT0, GL_SPECULAR, @SpecDef);  
end; {FormCreate}
```

Actually, LIGHT0 by default has a pure white specular component, but this illustration needs something a little less intense. Assign a medium gray to it.

```
with SpecDef do  
begin  
  red := 0.5;  
  green := 0.5;  
  blue := 0.5;  
  alpha := 1.0;  
end;  
glLightfv(GL_LIGHT0, GL_SPECULAR, @SpecDef);
```

```
end; {FormCreate}
```

Oh yes! Be sure to declare SpecDef in the form's class declaration.

```
    CubeColor ,  
    SpotColor ,  
    DiffDef ,  
    SpecDef ,  
    AmbientDef: TLightDesc;  
    LightPlace: TLightPlace;  
    SpotDirect: TPosition;
```

For convenience, change the initial value of z2 from -2.7 to -3.7, moving the larger cube farther back, so that the spotlight cube will not bury itself in the larger cube in the coming illustration.

```
    z2 := -3.7;  
    x2 := -0.7;  
    y2 := 0.7;  
    z1 := -6.7;  
    x1 := -1.8;  
    y1 := -2.1;
```

To the specular light component add the surface color response to that light.

```
    glLightfv(GL_LIGHT0, GL_SPECULAR, @SpecDef);  
    glMaterialfv(GL_FRONT, GL_SPECULAR, @SpecDef);  
end; {FormCreate}
```

Notice that the new glMaterialfv command is in the FormCreate method. Remember that such a command remains in effect until another command changes it. By placing it here, with no contradictory command elsewhere, you give all surfaces the same response to specular light for all time. Well, it lasts for as long as the program runs, anyway.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## Delphi Developer's Guide to OpenGL

(Publisher: Wordware Publishing, Inc.)

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

The specular effect needs one more specification. Remember that specular highlights appear on shiny surfaces. Naturally OpenGL defines a constant for setting shininess in a command.

```
GL_SHININESS = $1601;
```

The shininess property is an integer that can range in value from 0 to 128. Therefore the property needs an integer form of the `glMaterial` command. Give it maximum shininess for this illustration.

```
glLightfv(GL_LIGHT0, GL_SPECULAR, @SpecDef);  
glMaterialfv(GL_FRONT, GL_SPECULAR, @SpecDef);  
glMateriali(GL_FRONT, GL_SHININESS, 128);  
end; {FormCreate}
```

Save, Compile, Run, and press plus. Press little “m” four times to raise the spotlight toward the magenta cube. Do you see the faint light streak running from the lower left corner to the upper right corner of the near face? It is not much, but that is as good as it gets with a face of a cube. To see a better specular highlight, you need a surface with some curvature. A later chapter introduces curved surfaces.

## HOW MUCH LIGHT

So far you have seen three light source components: Ambient, Diffuse, and Specular. Each of these has three color components: red, green, and blue. There is also an alpha component, which still must be put off until later. Never fear! You will see alpha when the time is right. You have also seen material properties which specify the response to light. Material properties also have Ambient, Diffuse, and Specular parts. Each of these also has the three color components.

So what determines how much light of each color reaches the viewer? Would you believe a simple multiplication? Figure 5-6 illustrates.



**Figure 5-6.** Multiplying a Light Component.

## DEFINITIONS

<b>Term</b>	<b>Meaning</b>
Ambient light	Light that has reflected from various surfaces and seems to come from all directions with no readily identifiable source. The component of an OpenGL light source used to simulate that effect.
Diffuse light	Light that comes from an identifiable source and scatters when reflected. The component of an OpenGL light source used to simulate that effect.
Flat surface	A surface with enough roughness to scatter most light it reflects, producing a dull, non-glossy appearance. A surface that is not shiny.
Material properties	Characteristics of a surface that affect its appearance.
Shiny surface	A mirror-like surface. A surface that is smooth enough that much of the light it reflects is not scattered, giving a glossy appearance. Such a surface may be characterized by a specular highlight under a suitable light source.
Specular highlight	A bright spot that appears on a smooth, shiny surface under a strong light source.
Specular light	The component of an OpenGL light source used to determine the color and intensity of a specular highlight.
Spotlight	A light source with both position and direction. The space illuminated by the source is shaped like a cone. The cone is defined by an angle from the “aim” direction of the light.

[Previous](#) | 
 [Table of Contents](#) | 
 [Next](#)

[Products](#) | 
 [Contact Us](#) | 
 [About Us](#) | 
 [Privacy](#) | 
 [Ad Info](#) | 
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#)  
 All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

To access the contents, click the chapter and section titles.

## **Delphi Developer's Guide to OpenGL**

*(Publisher: Wordware Publishing, Inc.)*

Author(s): Jon Jacobs

ISBN: 1556226578

Publication Date: 08/01/99

[Previous](#) [Table of Contents](#) [Next](#)

## **IDENTIFIER REFERENCE**

### **Identifier**

glColorMaterial

### **Description**

An OpenGL command that causes material properties to track the current color set by glColor. It takes two parameters. The first tells which face or faces will be affected. The second tells which properties will track the current color. You must enable color tracking for this to work.

glLight

An OpenGL command that defines characteristics of an OpenGL light source. You must enable lighting and enable the specific light.

glLightModel

An OpenGL command that provides a simplified method of setting up a lighting environment instead of defining specific lights.

glMaterial

An OpenGL command for defining material properties.

GL\_AMBIENT

An OpenGL constant specifying the ambient portion of light or material properties in various OpenGL commands.

GL\_BACK

An OpenGL constant specifying the back face of polygons to be affected in various OpenGL commands.

GL\_COLOR\_MATERIAL

The OpenGL constant to pass to enable or disable color tracking.

GL_DIFFUSE	An OpenGL constant specifying the diffuse portion of light or material properties in various OpenGL commands.
GL_EMISSION	An OpenGL constant to pass to the glMaterial command, so that the affected surfaces appear to be glowing.
GL_FRONT	An OpenGL constant specifying the front face of polygons to be affected in various OpenGL commands.
GL_FRONT_AND_BACK	An OpenGL constant specifying both the front and back faces of polygons to be affected in various OpenGL commands.
GL_LIGHT0 .. GL_LIGHT7	OpenGL constants for specifying specific OpenGL light sources.
GL_LIGHTING	An OpenGL constant to pass to glEnable or glDisable to turn on or turn off lighting calculations.
GL_LIGHT_MODEL_AMBIENT	An OpenGL constant to pass to glLightModel to specify global ambient light.
GL_SPECULAR	An OpenGL constant specifying the specular portion of light or material properties in various OpenGL commands.
GL_SPOT_CUTOFF	An OpenGL constant to pass to glLight to specify the angle that defines the cone of illumination from a spotlight.
Packed	An Object Pascal reserved word used to make a data structure take the minimum amount of memory. The elements of the structure fit together with no unused memory between them because the compiler does not attempt to align the data elements on word or double-word boundaries.
PGLfloat	A data type which is a pointer to a GLfloat.

## SUMMARY

Chapter 5 presented new levels of realism with lighting. Here are the high points:

1. Use glEnable(GL\_LIGHTING) to inform OpenGL that you will take responsibility for lighting specification.
2. OpenGL allows at least eight separate light sources.
3. All light has a source. Each light source has three components whose color compositions can be separately identified:
  - a. Ambient light seems to come from all directions.
  - b. Diffuse light comes from a specific direction and scatters upon

reflection, giving a soft, flat appearance to surfaces.

c. Specular light reflects sharply and appears as a specular highlight on smooth surfaces.

4. Specify the color response of a surface to each light component with the `glMaterial` commands.

5. Emission is also a material property for OpenGL.

6. In addition to the color response to specular light, you must also specify the smoothness or shininess of the surface with the `GL_SHININESS` constant in the `glMateriali` command.

<a href="#">Previous</a>	<a href="#">Table of Contents</a>	<a href="#">Next</a>
--------------------------	-----------------------------------	----------------------

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc.

All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



# Database Application Developer's Guide

## Delphi for Windows

### Chapter 1 Introduction

#### Copyright Agreement



Delphi enables you to create robust database applications quickly and easily. Delphi database applications can work directly with desktop databases like Paradox, dBASE, the Local InterBase Server, and ODBC data sources. The Delphi Client/Server edition also works with remote database servers such as Oracle, Sybase, Microsoft SQL Server, Informix, InterBase, and ODBC data sources. Delphi client applications can be scaled easily between mission critical network-based client/server databases, and local databases on a single machine.

This chapter introduces Delphi's database tools, including the Data Access and Data Controls component pages, the Fields Editor, the Database Desktop, and the Database Forms Expert.

### What you should know first

Building a database application is similar to building any other Delphi application. This book assumes you understand the basic application development techniques covered in the *Delphi User's Guide*, including:

- Creating and managing *projects*
- Creating *forms* and managing *units*
- Working with *components, properties, and events*
- Writing simple Object Pascal source code

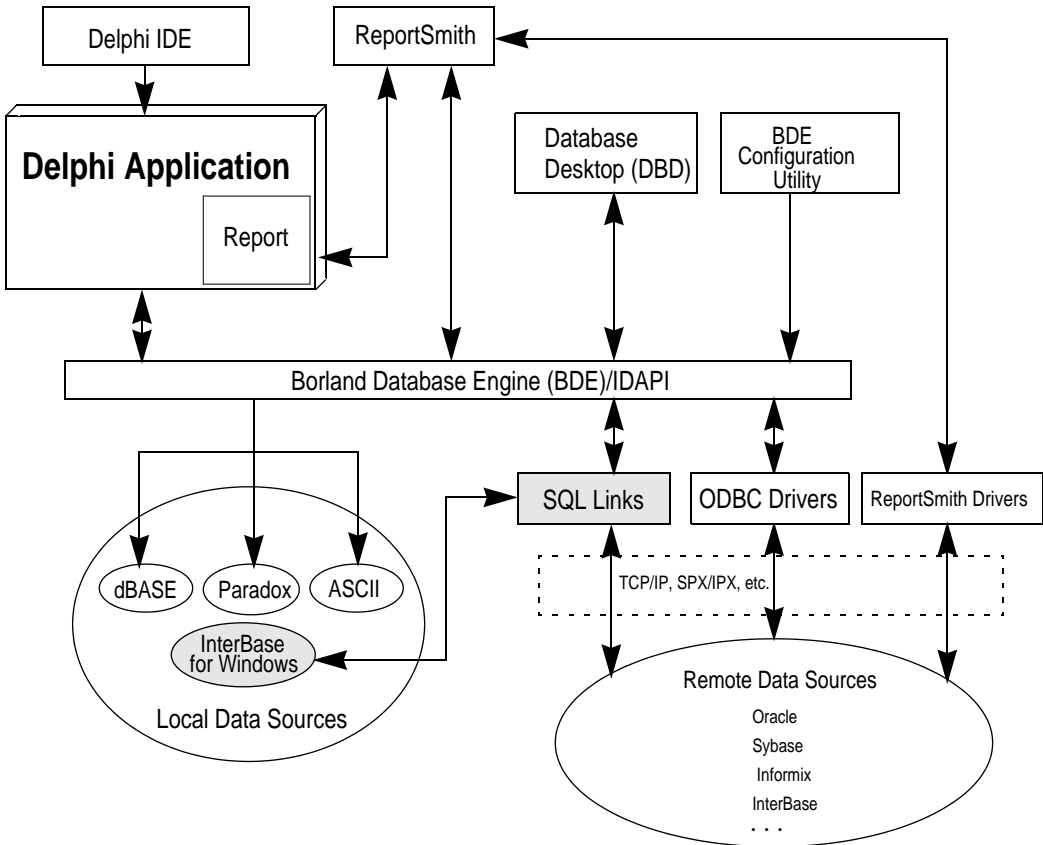
You also need to have a working knowledge of the Database Management System (DBMS) your Delphi database applications access, whether it is a desktop database such as dBASE or Paradox, or an SQL server. For information specific to building client/server applications with Delphi, see Chapter 6, "Building a client/server application."

This book assumes you have a basic understanding of relational databases, database design, and data management. There are many third-party books covering these topics if you need to learn more about them.

# Overview of Delphi's database features and capabilities

A Delphi database application is built using Delphi database development tools, Delphi data-access components, and data-aware GUI components. A database application uses Delphi components to communicate with the Borland Database Engine (BDE), which in turn communicates with databases. The following figure illustrates the relationship of Delphi tools and Delphi database applications to the BDE and data sources:

**Figure 1.1** Delphi database architecture



The following table summarizes Delphi's database features.

**Table 1.1** Database features summary

Tool	Purpose
Data Access components	Access databases, tables, stored procedures, and custom component editors.
Data Control components	Provide user interface to database tables.
Database Desktop (DBD)	Create, index, and query Paradox and dBASE tables, and SQL databases. Access and edit data from all sources.

**Table 1.1** Database features summary (continued)

Tool	Purpose
ReportSmith	Create, view, and print reports.
Borland Database Engine (BDE)	Access data from file-based Paradox and dBASE tables, and from local InterBase server databases.
BDE Configuration Utility	Create and manage database connection Aliases used by the BDE.
Local InterBase Server	Provides a single-user, multi-instance desktop SQL server for building and testing Delphi applications, before scaling them up to a production database, such as Oracle, Sybase, Informix, or InterBase on a remote server.
InterBase SQL Link	Native driver that connect Delphi applications to the Local InterBase Server.

These features enable you to build database applications with live connections to Paradox and dBASE tables, and the Local InterBase Server through the BDE. In many cases, you can create simple data access applications with these components and their properties without writing a line of code.

The BDE is built into Delphi components so you can create database applications without needing to know anything about the BDE. The Delphi installation program installs drivers and sets up configuration for Paradox, dBASE, and the Local InterBase Server, so you can begin working with tables native to these systems immediately. The BDE Configuration Utility enables you to tailor database connections and manage database aliases.

Advanced BDE features are available to programmers who need more functionality. These features include local SQL, which is a subset of the industry-standard SQL that enables you to issue SQL statements against Paradox and dBASE tables; low-level API function calls for direct engine access; and ODBC support for communication with other ODBC-compliant databases, such as Access and Btrieve.

Delphi includes Borland ReportSmith, so you can embed database report creation, viewing, and printing capabilities in Delphi database applications. Delphi also includes the Database Desktop (DBD), a tool that enables you to create, index, and query desktop and SQL databases, and to copy data from one source to another. For more information about ReportSmith, see *Creating Reports*. For more information about the DBD, see Appendix A, "Using Database Desktop."

The Local InterBase Server is a single-user, multi-instance, 16-bit, ANSI SQL-compliant, Windows-based version of Borland's 32-bit InterBase SQL server that is available for Novell NetWare, Windows NT, and Unix. For more information, see the *Local InterBase Server User's Guide*.

The following table lists the additional database features available in the Client/server edition of Delphi. These features extend Delphi's database capabilities to access remote

SQL database servers such as Sybase, Microsoft SQL Server, Oracle, Informix, and InterBase.

**Table 1.2** Additional Delphi Client/Server database features

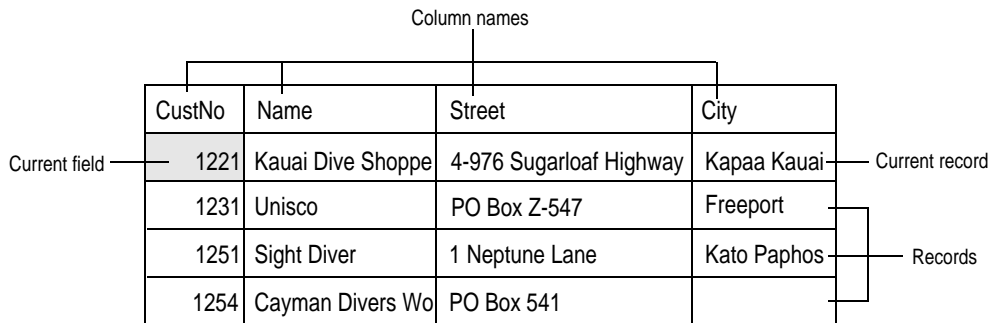
Tool	Purpose
SQL Drivers	Both SQL Links and ReportSmith provide native drivers that connect Delphi database applications to remote SQL database servers, such as Oracle, Sybase, Microsoft SQL Server, Informix, and InterBase.
Visual Query Builder	Creates SQL statements by visually manipulating tables and columns.

SQL Links provide Delphi applications with SQL access to data residing on remote servers, including Sybase, Microsoft SQL Server, Oracle, and Informix. When an SQL Link driver is installed, SQL statements are passed directly to the server for parsing and execution. For more information about using passthrough SQL, see Chapter 5, "Using SQL in applications."

## What is a database?

Delphi programmers should understand some basic concepts about databases, data, and data access, before building database applications. A database consists of one or more tables, where each table contains a series of columns into which records (also called "rows") are stored. Each record is identical in structure. For example, a database of addresses consists of a table with name, street address, city, state, and zipcode columns. The intersection of a single column and row is referred to as a field. Fields contain values. The following figure illustrates these concepts:

**Figure 1.2** Structure of a table



The current *field* is one field in a single record. The current *record* is a single record in a multi-record set that is the focus of attention. For example, some Delphi database applications display multiple columns and records in a grid format for editing. As far as Delphi controls are concerned, only one field in a single record is "current," meaning that editing tasks affect only the data in that field.

Different databases vary widely in structure. A database in Paradox consists of one or more files, each of which contains a single table or index, but an SQL relational database on a remote server generally consists of a single file that contains all tables, indices, and

other database structures. Delphi's Data Access and Data Control components encapsulate the underlying structure of the databases your application uses, so that your application can present the same interface to an end user whether it accesses a local Paradox file or a database on a remote SQL server.

The information in most databases is constantly changing. When you access a networked database from a Delphi application, many users may be accessing and updating the database at the same time. When any database application accesses a database, whether to process a query or generate a report, the application receives a snapshot of the database as it was at the time the application accessed the database. An application's view of data may differ from the data currently in the database, so database applications should always be robust enough to react to such data changes. For more information about building client/server applications that access remote data, see Chapter 6, "Building a client/server application."

## What is data?

---

In this book, "data" refers to information stored in a database. Data may be a single item in a field, a record that consists of a series of fields, or a set of records. Delphi applications can retrieve, add, modify, or delete data in a database.

## What is data access?

---

Delphi applications can access data from desktop database tables on a file server or local disk drive and from remote database servers. To access a data source, a Delphi application uses Data Access components to establish a connection through the BDE. The installation program for Delphi installs drivers and sets up configurations for Paradox, dBASE, and the Local InterBase Server so you can begin working with tables native to these systems immediately.

To connect to another data source requires the installation of a driver for that specific database and subsequent configuration of the BDE to recognize the driver. Connecting to remote database servers requires the Client/Server edition of Delphi that includes SQL Links to access to Sybase, Microsoft SQL Server, Oracle, Informix, and InterBase on NT, NetWare, and Unix servers. For more information about installing and configuring the SQL Link drivers, see the *SQL Links User's Guide*.

The BDE uses *aliases* as convenient shorthand names for often-used data sources, whether local or remote. The BDE Configuration Utility enables you to define and modify aliases that Delphi applications can use immediately. For more information about defining aliases, see Appendix B, "Using the BDE configuration utility."

Once drivers are installed and network connections established, Delphi applications can access data from any authorized server. The examples in Chapter 2 demonstrate techniques for accessing data from a database—specifically, sample data tables that are shipped and installed as part of the Delphi package. Although the example project in the next chapter deals with local desktop data, the techniques for accessing remote data are essentially the same, as subsequent chapters demonstrate.

**Note** An SQL version of the example project is provided in the DEMOS directory.

## Data sources

---

Delphi database applications get their data through the BDE. The different data sources (not to be confused with the *TDataSource* component) that the BDE can use are shown in Table 1.3.

**Table 1.3** Delphi data sources

Data source	Description	File extension
Paradox	Tables created with Paradox or Database Desktop. Each table is in a separate file.	.DB
dBASE	Tables created with dBASE or Database Desktop. Each table is in a separate file.	.DBF
ASCII files	Tables created with Database Desktop. Each table is in a separate file.	.TXT
Local InterBase Server	Database created with InterBase Windows ISQL. Multiple tables in a single database file.	.GDB
SQL Database Server: Oracle, Sybase, Microsoft SQL Server, Informix, InterBase	Database created with server-specific tools, or the DBD, accessed across network with SQL Links. Delphi Client/Server Edition only.	Depends on server
ODBC data sources	Databases such as Microsoft Access, Btrieve, FoxPro, etc.	Depends on data source

---

## Understanding Delphi database architecture

---

Delphi uses object-oriented components to create database applications, just as it does with non-database applications. Like standard components, database components have attributes, or *properties*, that are set by the programmer at design time. These properties can also be set programmatically at run time.

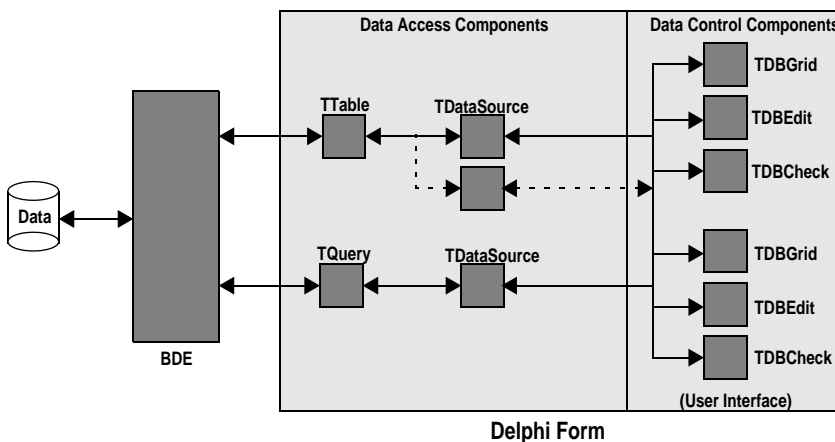
Database components have default behavior that enables them to perform useful functions with little or no programming. The Delphi Component palette provides two database component pages:

- The *Data Access page* contains Delphi objects that simplify database access by encapsulating database source information, such as the database to connect to, the tables in that database to access, and specific field references within those tables. Examples of the most frequently used data access objects include *TTable*, *TQuery*, *TDataSource*, and *TReport*.
- The *Data Controls page* contains data-aware user interface components for displaying database information in forms. Data Control components are like standard user interface components, except that their contents can be derived from or passed to database tables. Examples of the most frequently used data control components include *TDBEdit*, *TDBNavigator*, and *TDBGrid*.

Datasets, such as *TTable*, *TQuery*, and *TStoredProc* components, are not visible at run time, but provide applications their connection to data through the BDE. Data Control components are attached to dataset components by a *TDataSource* component, to provide a visual interface to data.

The following figure illustrates how Data Access and Data Control components relate to data, to one another, and to the user interface in a Delphi database application:

**Figure 1.3** Database components architecture

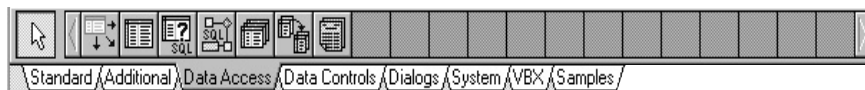


As this figure illustrates, a form usually contains at least three database components: a dataset component (*TTable* and *TQuery* in the figure) that communicates with the BDE; a *TDataSource* component that acts as a conduit between a dataset component and the user interface; and one or more data control components, such as *TDBEdit* or *TDBGrid*, that enable a user to browse, edit, or enter data.

## Overview of the Data Access page

The Data Access page of the Delphi Component palette provides a set of database encapsulation objects that simplify database access.

**Figure 1.4** Data Access page of the Component palette



When building a database application, you place data access components on a form, then assign them properties that specify the database, table, and records to access. They provide the connection between a data source and Data Control components.

At run time, after an application is built and compiled, data access objects are not visible, but are “under the hood,” where they manage data access.

The following table lists the data access objects on the Data Access page, and briefly describes how they are used:

**Table 1.4** Data Access components

Component	Purpose
TDataSource	Acts as a conduit between a TTable, TQuery, TStoredProc component and data-aware components, such as TDBGrid.
TTable	Retrieves data from a database table via the BDE and supplies it to one or more data-aware components through a TDataSource component. Sends data received from a component to a database via the BDE.
TQuery	Uses SQL statements to retrieve data from a database table via the BDE and supplies it to one or more data-aware components through a TDataSource component, or uses SQL statements to send data from a component to a database via the BDE.
TStoredProc	Enables an application to access server stored procedures. Sends data received from a component to a database via the BDE.
TDatabase	Sets up a persistent connection to a database, especially a remote database requiring a user login and password.
TBatchMove	Copies a table structure or its data. Can be used to move entire tables from one database format to another.
TReport	Enables printing and viewing of database reports through ReportSmith.

Four data access components deserve special mention. Most forms provide a link to a database with a TTable or TQuery component (or through a user-defined component based on the normally hidden abstract class, TDataSet, of which TTable and TQuery are descendents). Other forms provide a link to a database with TStoredProc, also a descendent of TDataSet. In turn, all forms must provide a TDataSource component to link a TTable, TQuery, or TStoredProc component to data control components that provide the visible user interface to the data.

TTable, TQuery, (and TStoredProc, when it returns a result set) contain a collection of TField components. Each TField corresponds to a column or field in the table or query. TFields are created

- Automatically, when TTable, TQuery, or TStoredProc are activated.
- At design time, using the Fields editor.

For more information about TFields and the Fields editor, see Chapter 3, “Using data access components and tools.” For more information about TStoredProc, see Chapter 6, “Building a client/server application.”

## Understanding TTable

The TTable component is the easiest way for a programmer to specify a database table for access. To put a TTable component on a form:

- 1 Select the Data Access page from the Component palette.
- 2 Click the Table icon.
- 3 Click on the form to drop the TTable component.
- 4 Enter the directory where the database resides in the DatabaseName property of the Object Inspector window. For SQL databases, enter an alias name.



**Note** An alias can also be used for local Paradox and dBASE tables. You can choose an alias from a drop-down list in the Object Inspector.

- 5 Enter the name of the table to use in the *TableName* property of the Object Inspector window, or you can also choose a table from the drop-down list instead of entering the name.

By default, a *TTable* component accesses every column in a table when you activate it. When a visual component, such as *TDBEdit*, is associated with a *TTable* object, it can display any field in the table. Multi-column visual components, such as *TDBGrid*, access and display columns in the table using the table's *TField* list.

If you double-click a *TTable* component on a form, you invoke the Fields Editor. The Fields Editor enables you to control the way Data Control components display data. It can

- Create a static model of a table's columns, column order, and column type that does not change even if changes are made to the underlying physical table in the database.
- Provide convenient, readable, and efficient component names for programmatic access.
- Specify the order in which fields are displayed and which fields to include.
- Specify all display characteristics of fields.
- Add custom validation code.
- Create new fields for display, including calculated fields.

For complete information about the Fields Editor, see Chapter 3, "Using data access components and tools."

## Understanding TQuery

The *TQuery* component provides a tool for data access using SQL statements, such as a SELECT statement, to specify a set of records and a subset of columns from a table. *TQuery* is useful for building local SQL queries against Paradox and dBASE data, and for building client/server applications that run against SQL servers.

To put a *TQuery* component on a form:

- 1 Select the Data Access page from the Component palette.
- 2 Choose the Query icon.
- 3 Click on the form to drop the *TQuery* component.
- 4 Enter the directory where the database resides (or select an alias for SQL databases) in the *DatabaseName* property of the Object Inspector window.
- 5 Enter the SQL statement to use for data access in the SQL property of the Object Inspector window by clicking the list button to open the String Editor.

The Object Inspector window for *TQuery* does not contain a separate property for specifying a table name. Instead, a table name must always be specified as part of the SQL statement in the SQL property.

With Delphi Client/Server, you can right-click a *TQuery* component on a form, then select the Visual Query Builder from the pop-up menu. The Visual Query Builder enables you to connect to a database and build an SQL statement interactively. For complete information about the Visual Query Builder, see the online Help.

If you double-click a *TQuery* component, you invoke the Fields Editor. The Fields Editor enables you to control the way Data Control components display data. For complete information on *TQuery* and the Fields Editor, see Chapter 3, “Using data access components and tools.”

## Understanding TDataSource

Every dataset that supplies a data control component must have at least one *TDataSource* component. *TDataSource* acts as a bridge between one *TTable*, *TQuery*, or *TStoredProc* component and one or more data control components that provide a visible user interface to data.

*TTable* and *TQuery* can establish connections to a database through the BDE, but they cannot display database information on a form. Data Control components provide the visible user interface to data, but are unaware of the structure of the table from which they receive (and to which they send) data. A *TDataSource* component bridges the gap.

To put a *TDataSource* component on a form:

- 1 Select the Data Access page from the Component palette.
- 2 Choose the DataSource icon.
- 3 Click on the form to create the *TDataSource* component.
- 4 Enter the name of the *TTable* or *TQuery* component to use as a database connection source in the DataSet property of the Object Inspector. If the form contains any *TTable* or *TQuery* components, you can choose a component from the drop-down list instead.

**Note** *TDataSource* is also used to link tables or queries in a master/detail form. For more information about master/detail forms, see Chapter 2, “Building a sample database application: MASTAPP.”

## Overview of the Data Controls page

---

The Data Controls page provides a set of data-aware user-interface components that you can use to create forms-based database applications.

**Figure 1.5** The Data Controls page of the Component palette



Many data controls are data-aware versions of component classes available on the Standard page of the Component palette. In addition to standard component functionality, data controls can display data from a field in a database table, or send new or modified data from a form to a database table.

The following table lists the data controls on the Data Control page.

**Table 1.5** Data Controls components

Component	Purpose
TDBNavigator	Data-aware navigation buttons that move a table's current record pointer forward or backward; start Insert or Edit mode; post new or modified records; cancel Edit mode; and refresh display to retrieve updated data.
TDBText	Data-aware label that can display a field from a currently active record.
TDBEdit	Data-aware edit box that can display or edit a field from a currently active record.
TDBCheckBox	Data-aware check box that can display or edit a Boolean data field from a currently active record.
TDBListBox	Data-aware list box that can display values from a column in a table.
TDBComboBox	Data-aware combo box that can display or edit values from a column in a table.
TDBRadioGroup	Data-aware radio group populated with radio buttons that can display or set column values.
TDBGrid	Data-aware custom grid that enables viewing and editing data in a tabular form similar to a spreadsheet; makes extensive use of <i>TField</i> properties (set in the Fields Editor) to determine a column's visibility, display format, ordering, etc.
TDBMemo	Data-aware memo box that can display or edit text BLOB data from a currently active record.
TDBImage	Data-aware image box that can display, cut, or paste bitmapped BLOB images to and from a currently active record.
TDBLookupList	Data-aware list box that displays values mapped through another table at run time.
TDBLookupCombo	Data-aware combo box that displays values mapped through another table at run time.

Data control components make up a consistent visual user interface for Delphi database applications, whether the application accesses a local database file, or a remote database server. To see how data control components are used in an application, see the subsequent chapters of this book. For a complete description of each data control component and its properties, see the online VCL Reference.

## Overview of the Database Forms Expert

The Database Forms Expert automates many of the tasks necessary for creating data-entry or tabular forms from an existing database table. It can generate simple or master/detail forms using *TTable* or *TQuery* components. The Database Forms Expert automates such form building tasks as:

- Placing database components on a form.
- Connecting *TDataSet* components (e.g., *TTable* and *TQuery*) to a database.
- Connecting *TDataSource* components to interactive data control components and *TTable* or *TQuery* data access objects.
- Writing SQL statements for *TQuery* objects.
- Defining a tab order for components.

Inexperienced database applications programmers can use the Database Forms Expert to learn how to build database forms. Experienced database applications programmers can use it to speed application development. To learn how to use the Database Forms Expert when building an application, see Chapter 2, “Building a sample database application: MASTAPP.”

## Overview of the Database Desktop

---

The Database Desktop (DBD) is a database maintenance and data definition tool. It enables programmers to query, create, restructure, index, modify, and copy database tables, including Paradox and dBASE files, and SQL tables. You do not have to own Paradox or dBASE to use the DBD with desktop files in these formats.

The DBD can copy data and data dictionary information from one format to another. For example, you can copy a Paradox table to an existing database on a remote SQL server. For a complete description of the DBD, see Appendix A, “Using Database Desktop.”

## Developing applications for desktop and remote servers

---

Delphi Client/Server enables programmers to develop and deploy database client applications for both desktop and remote servers. One of Delphi’s strengths is the ease with which an application developed for the desktop can be adapted to access data on a remote SQL server. The user interface need not change even if the source of the data changes. To an end user, a Delphi database application looks the same whether it accesses a local database file or a remote SQL database.

For simple applications that use *TQuery* components to access desktop data, the transition to a remote server may be as simple as changing the data source. For other applications, more significant changes may be in order. Some of these changes are the result of differing conventions and concurrency issues between desktop and SQL databases.

For example, desktop databases like Paradox and dBASE are record-oriented. They always display records in ascending or descending alphabetic or numeric order. They lock and access a single record at a time. Each time a user changes a record, the changes are immediately written to the database. Desktop database users can see a range of records, and can efficiently navigate forward and backward through that range.

In contrast, data in SQL databases is set-oriented, and designed for simultaneous multiuser access. Record ordering must be specified as part of an SQL query. To accommodate multiuser access to data, SQL relies on transactions to govern access. For more information about working with transactions, see Chapter 6, “Building a client/server application.”

# Database application development methodology

---

Developing database applications with Delphi is similar to developing other types of software, but there are important distinctions and challenges that must be addressed. The methodology presented in this section should be used as a guideline that you can adapt to meet your specific business needs.

## Development scenarios

---

Since an application's design usually depends on the structure of the database it will access, the database must be defined before the application can be developed.

**Note** Database development (also called *data definition*) is a part of the overall development process, but is beyond the scope of this manual. For more information, refer to the numerous books about relational database design.

There are four possible scenarios for Delphi database application development:

- The database does not yet exist or must be re-defined.
  - Use the Database Desktop utility to define Paradox and dBASE tables. For more information, see Appendix A, "Using Database Desktop."
  - For SQL servers, use the tools provided with the server or the Database Desktop. For example, for the Local InterBase Server or an InterBase Workgroup Server, use Windows ISQL. For more information, see the *Local InterBase Server User's Guide* and the *InterBase Data Definition Guide*.
- The database exists on a desktop or LAN data source (Paradox or dBASE) and the database will access it there. If the BDE and the data source are on the same machine as the application, then the application is a standalone (not client/server) application.
- The database exists on a desktop data source, and is being upsized to an SQL server. This scenario is discussed in Appendix C, "Using local SQL."
- The database exists on an SQL server and the application will access it there. This is a standard client/server application. For information specific to developing a client/server application, Chapter 6, "Building a client/server application."

## Database application development cycle

---

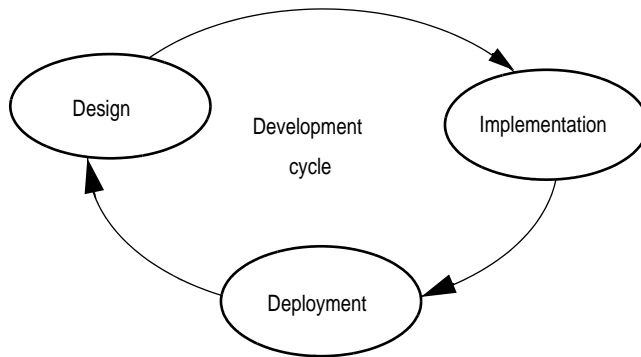
The goal of database application development is to build a product which meets end users' long-term needs. While this goal may seem obvious, it is important not to lose sight of it throughout the complexities and often conflicting demands of the development process. To create a successful application it is critical to define the end users' needs in detail early in the development process.

The three primary stages of database application development are

- Design and prototyping
- Implementation
- Deployment and maintenance

There are database and application tasks in each of these phases. Depending on the size and scope of the development project, the database and application tasks may be performed by different individuals or by the same individual. Often, one team or individual will be responsible for the database tasks of the project, and another team or individual will be responsible for the application tasks.

**Figure 1.6** Development cycle



For client/server applications, the database and application tasks become more distinct, since they run on different platforms, often with different operating systems (for example, a Unix server and Windows 3.1 client).

When development responsibilities are thus divided it is important to clearly delineate in the design phase which functions will be performed by the database server and which will be performed by the client application. Usually, the functional lines are clear cut. But database processes such as stored procedures can sometimes perform functions that can also be performed by the client application. Depending on the expected deployment configuration, application requirements, and other considerations, the design can allocate such functions to either client or server.

It is also important to realize that database application development is by its nature an iterative process. Users may not fully understand their own needs, or may define additional needs as development proceeds. User interface elements are always refined as they are used. Also, changing business needs will change requirements over time. Generally, a number of iterations through the development cycle will be required before an application can meet a significant portion of its requirements.

## Design phase

The design phase begins with requirements definition. In consultation with knowledgeable end users, define the functional specifications for the database and applications. Determine which aspects of the functional requirements will be implemented in the database design, and which aspects will be implemented in the applications.

For client/server applications, often certain functions can be performed either by the server or by the application; for example, a complex mathematical transform function could be performed either by the client application or by a stored procedure on the server. The hardware deployment configuration will generally determine whether such

functions are best performed on the server or client. For example, if the client platforms are expected to be low-end desktop PCs, and the server platform is expected to be a high-end workstation, then it will probably be best to run computation-intensive functions on the server. If the hardware configuration changes, then it is possible to move the function between client and server in a later iteration.

## Implementation phase

In the implementation phase, you use Delphi to build and test the application conceived in the design phase. During the implementation phase, you should use a duplicate data source, that is, a data source that has the same essential structure as the production database, but with a small subset of representative data. It is not recommended to develop an application against a production database, since the untested application may corrupt the data or otherwise interfere with normal database activities.

If your application will ultimately be deployed to use a desktop data source, make copies of the required tables with the Database Desktop, and populate them with representative “dummy” data.

If the application will ultimately be deployed to use a remote data source (an SQL server), then you can take two approaches during the implementation phase:

- Develop and test the application against a non-production database on the Local InterBase Server.
- Develop and test the application against a non-production database on the server.

The first approach has the advantage that is isolated on the development platform(s), and so will not interfere with other server activities. It will not consume server resources or increase network traffic. Its primary disadvantage is that only standard SQL server features can be used and tested during this phase, if you are using a server other than InterBase for the deployed application.

The second approach enables you to surface all server-specific features, but will consume network and server resources during testing. This approach can be dangerous, since it is conceivable that a programmer error could cause a server to crash during testing.

## Deployment phase

In the deployment phase, the client/server application is put to the acid test: it is handed over to end users. To ensure that the application’s basic functionality is error-free, deploy a prototype application before attempting to deploy a production application.

Since the ultimate judges of an application’s efficacy are its users, developers must be prepared to incorporate changes to applications arising from their suggestions, changing business needs, and for general enhancement (for example, for usability). Sometimes application changes may require changes to the database, and conversely, changes to the database may require application changes. For this reason, application developers and database developers should work together closely during this phase. As features and enhancements are incorporated into the application, the application moves iteratively closer to completion.

Deploying a client/server application requires addressing a number of special issues, including connectivity and multiuser access. These issues are discussed in Chapter 6, “Building a client/server application.”

## Deploying an application

---

Deploying an application means giving it to the end users, and providing the necessary software they need to use the application in a production environment. Non-database applications require only an .EXE file to run—Delphi applications do not require a run time interpreter or DLL.

Typically, when deploying a database application, you will create a package that includes all the files that end users need to run the application and access data sources. These files include

- The application .EXE file and .DLL files (if any)
- Required ancillary files (for example, a README file or .HLP files for online help)
- BDE support for database access (desktop or server)
- ReportSmith Runtime for running and printing reports
- If the application uses VBX controls, include each VBX along with BIVBX11.DLL

If you are distributing the files on disks, you will generally want to compress them with a standard file compression utility, and provide the utility on the disk. You may also want to build a simple installation application to install the files for your users. For complex applications, you may want to use one of the many commercially-available installation programs.

**Important** Before distributing any files, ensure that you have the proper redistribution rights. As described in the Delphi license agreement, Delphi provides distribution rights for the BDE (including Paradox and dBASE support). Delphi Client/Server includes distribution rights for Borland SQL Links for Windows. Licenses for distribution of the Local InterBase Server are available from Borland.

For information on deploying support for remote server access, see Chapter 6, “Building a client/server application.” For client/server applications, you also must ensure that the necessary communications software (for example, TCP/IP interface) is installed on the client platforms. This software is provided with databases servers. For more information, see your server documentation.

## Deploying BDE support

---

When you deploy a database application, you must ensure that the client platform has the correct version of the BDE installed. Delphi includes Redistributable BDE, with its own installation utility, that can you can redistribute with your applications. When you deploy an application, simply include a copy of the Redistributable BDE disk.

The Delphi license agreement requires you to make *all* the files in Redistributable BDE available to your application users. This requirement enables users to install the new version of the BDE for Delphi without interfering with existing Paradox and dBASE applications. You can advise your users to save disk space and install only the drivers



required to run your application, but you must still distribute all the files in the Redistributable BDE.

For example, if your application needs access only to Paradox files, you can advise your users not to deploy the dBASE driver. The minimum BDE configuration for accessing a Paradox database requires about 500 Kbytes.

**Note** For more information on deployment, refer to the file DEPLOY.TXT installed to the DELPHI\DOC directory by default.

**Table 1.6** Redistributable Borland Database Engine files

File name	Description
IDAPI01.DLL	BDE API DLL
IDBAT01.DLL	BDE Batch Utilities DLL
IDQRY01.DLL	BDE Query DLL
IDASCI01.DLL	BDE ASCII Driver DLL
IDPDX01.DLL	BDE Paradox Driver DLL
IDDBAS01.DLL	BDE dBASE Driver DLL
IDR10009.DLL	BDE Resources DLL
ILD01.DLL	Language Driver DLL
IDODBC01.DLL	BDE ODBC Socket DLL
ODBC.NEW	Microsoft ODBC Driver Manager DLL, version 2.0
ODBCINST.NEW	Microsoft ODBC Driver installation DLL, version 2.0
TUTILITY.DLL	BDE Tutility DLL
BDECFG.EXE	BDE Configuration Utility
BDECFG.HLP	BDE Configuration Utility Help
IDAPI.CFG	BDE (IDAPI) Configuration File

## Language drivers

The BDE provides the ability to localize applications with language drivers. The language driver DLL loads the drivers specified by Paradox or dBASE tables or in IDAPI.CFG for server databases. The language drivers are files with extension .LD installed to the LANGDRV sub-directory of the BDE directory.

**Important** For language drivers to load correctly, the WIN.INI file must have the following entry, assuming the default installation directory:

```
[Borland Language Drivers]
LDPath = C:\DELPHI\IDAPI\LANGDRV
```

## ODBC Socket

The BDE comes with an ODBC Socket. It has been certified with Microsoft's 2.0 ODBC Driver Manager. If you have a different version of the ODBC Driver Manager:

- Back up your existing ODBC.DLL and ODBCINST.DLL
- Copy the version 2.0 files, ODBC.NEW and ODBCINST.NEW, from your BDE directory to your WINDOWS\SYSTEM directory.
- Rename these files to ODBC.DLL and ODBCINST.DLL.

**Note** The ODBC 2.0 Driver Manager does work with ODBC 1.x ODBC drivers.

# Building a sample database application: MASTAPP

This chapter is a tutorial and introduction to building Delphi database applications. Examples show how to perform database tasks using Delphi interactively and by programming in Object Pascal. In each example you build a single form, self-contained and independent of the others. You can save yourself some work by doing the examples in sequence. Several examples use the same basic form as a starting point.

**Note** This material assumes you know how to use Delphi; it tells you *what* to do to perform certain tasks. For more details (that is, to find out *why*), follow the cross-references in the “For more information” section that follows each example. In particular, see Chapter 3, “Using data access components and tools.”

The tutorial consists of the following sections:

- “Building forms” describes how to use the Database Form Expert to create database forms, including single-table and master-detail forms. It also describes how to enhance forms by adding components and code by hand.
- “Working with fields” describes how to read and write field values, how to search for values and do table lookups, and how to format data displayed to the user. It also describes how to work with calculated fields.
- “Using queries and ranges” describes how to use SQL queries and set ranges to select a subset of the data in one or more tables.
- “Printing reports and forms” describes how to print ReportSmith reports and Delphi forms.

## Building forms

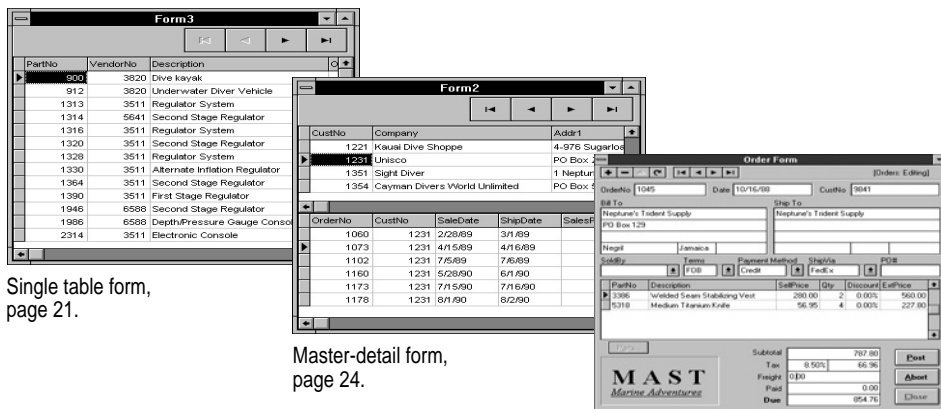
---

The material in this section focuses on database issues. To learn about general application building with Delphi, see the *User’s Guide*. The forms described here are the

basis for a database application called MASTAPP, designed to meet the record-keeping needs of the fictitious Marine Adventures & Sunken Treasures company (MAST). MAST sells diving equipment and arranges diving expeditions. MASTAPP tracks information about customers, orders, inventory, and vendors.

The tutorial starts with a simple “codeless” form for viewing and editing table data, and works up to a full-featured invoice form containing several tables, data-aware components, and other advanced Delphi features. All the forms, tables, and related files are installed by default in C:\DELPHI\DEMOS\DB\MASTAPP. During a default installation of Delphi, an alias, DBDEMOS, that points to the MASTAPP directory is created for you, or you can create your own alias using the BDE configuration utility (see Appendix B). The following figure shows the forms and tells where they are described in this chapter:

**Figure 2.1** Database forms described in the tutorial



Single table form,  
page 21.

Master-detail form,  
page 24.

One-many-many form,  
page 27.

**Note** For general information about building Delphi forms, see the *User's Guide*.

## MASTAPP aliases

All *TTable* and *TQuery* components used in example code in this chapter set their *DatabaseName* property to DBDEMOS. In contrast, the complete demo in the MASTAPP directory does the following to facilitate porting:

- 1 The main form (MAIN.PAS) has a *TDatabase* component with its *AliasName* property set to DBDEMOS and *DatabaseName* property set to MAST.
- 2 All datasets on all forms have their *Database* properties set to MAST. Now all forms can use a different BDE alias simply by changing the main form's *TDatabase* component's *AliasName* property.

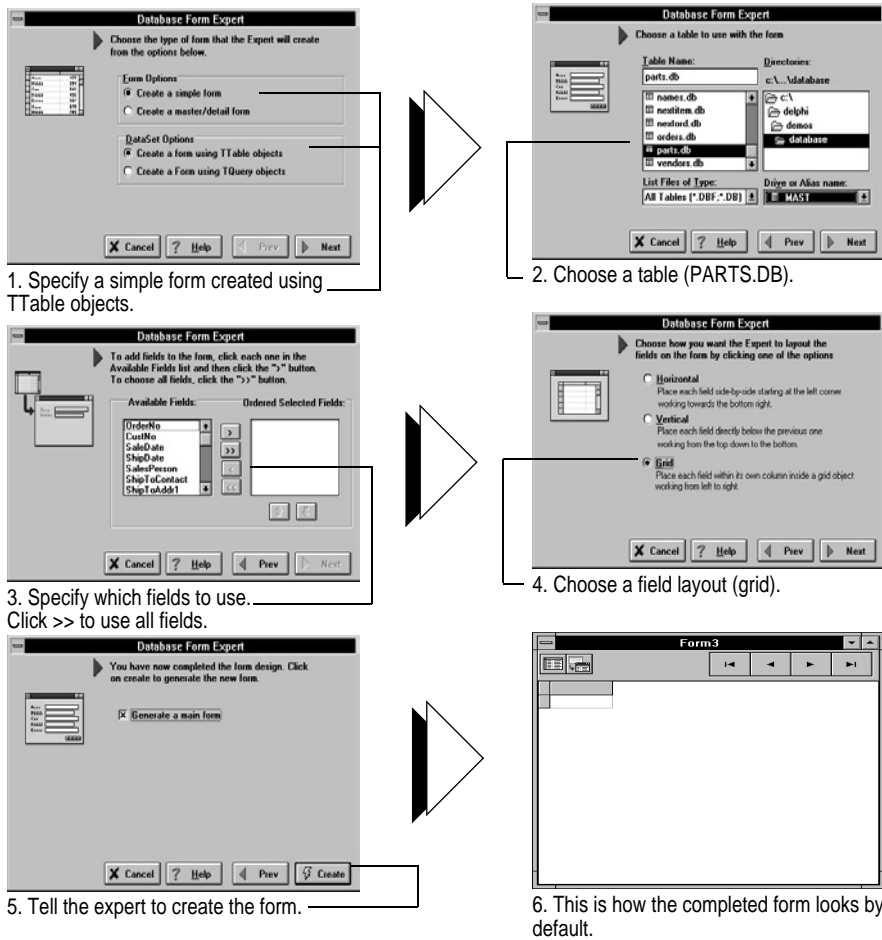
# Building a single-table form

The steps in this section show how to use the Database Form Expert to build a single-table form. Of course, anything the expert does, you can do by hand, but the expert saves a lot of time.

## What to do

- 1 Choose Help | Database Form Expert to open the Form Expert.
- 2 Specify a table, fields, and field layout as shown in the following figure. The Form Expert creates the form.
- 3 Press *F9* to run the form. Click the navigator control buttons to move through the records in the table.

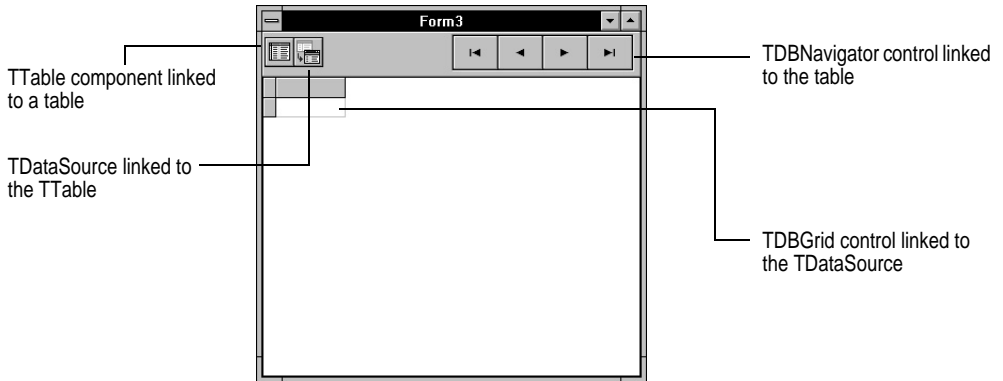
**Figure 2.2** Building a single-table form using the Database Form Expert



## How it works

The Database Form Expert builds a single-table form to match your specifications, and adds a tool bar of navigation controls. The form shown in the following figure is the basis for the full-featured BRPARTS.DFM form in the MASTAPP application. The important relationships in a data-aware form like this one are the links between the underlying data, the nonvisual components, and the data controls that display data to the user.

**Figure 2.3** A single-table form



The form contains one *TTable* component. The expert links it to the *Parts* table by setting the properties listed in the following table. A *TTable* component establishes a connection to a table in a database.

**Table 2.1** Important *TTable* properties for a single-table form

Property	Value	Remarks
<i>Active</i>	False	When <i>Active</i> is False, data-aware controls do not display data at design time. To make controls display data at design time, set <i>Active</i> to True.
<i>DataBaseName</i>	MAST	MAST is an alias that points to where the table resides. Use aliases, not hard-coded paths, to make applications portable and easy to upsize.
<i>Name</i>	Form1	Use the Object Inspector to change names.
<i>TableName</i>	PARTS.DB	Tells the component which table to link to.

The form contains one *TDataSource* component. The expert links it to the *TTable* component by setting the properties listed in the following table. A *TDataSource* component acts as a bridge between one dataset component (*TTable* in this case) and one or more data-aware controls that provide a visible interface to data.

**Table 2.2** Important *TDataSource* properties for a single-table form

Property	Value	Remarks
<i>AutoEdit</i>	True (default)	When <i>AutoEdit</i> is True, Delphi puts the <i>TDataSource</i> into Edit state automatically when the user changes a value in a linked control. To make a <i>TDataSource</i> read-only, or to control when to enter Edit state, set <i>AutoEdit</i> to False.

**Table 2.2** Important TDataSource properties for a single-table form (continued)

Property	Value	Remarks
<i>DataSet</i>	Table1	Specifies which <i>TTable</i> (or <i>TQuery</i> ) is supplying the data.
<i>Name</i>	DataSource1	Use the Object Inspector to change names.

The form contains one *TDBGrid* control. The expert links it to a *TDataSource* component by setting the properties listed in the following table. By default, a *TDBGrid* includes all the fields (columns) in a table. To limit the columns displayed by a *TDBGrid*, double-click on its associated *TTable* component to invoke the Fields Editor (see page 30).

**Table 2.3** Important TDBGrid properties for a single-table form

Property	Value	Remarks
<i>DataSource</i>	DataSource1	Links the <i>DBGrid</i> control to a <i>TDataSource</i> component, which supplies the data.

The form contains one *TDBNavigator* control. This control moves a table's current record pointer forward or backward, starts Insert or Edit state, posts new or modified records, etc. The expert links this control to the *Parts* table by setting the properties listed in the following table.

**Table 2.4** Important TDBNavigator properties for a single-table form

Property	Value	Remarks
<i>DataSource</i>	DataSource1	Links the control to a <i>TDataSource</i> component.
<i>VisibleButtons</i>	A list of button identifiers and a corresponding Boolean value that specifies whether that button is visible. Example: <i>nbFirst</i> True.	For example, by default <i>nbNext</i> is True, so the Next Record button is visible; <i>nbDelete</i> is False, so the Delete button is invisible.

The expert does more than create a form and components. It also generates a line of code to open the table at run time in case you do not activate the table at design time. For example, the expert creates *TTable* components with the *Active* property set to False. That's why the various *TDBEdit* controls aren't displaying data. You could set *Active* to True, and the controls would display data from the first record. Instead, the Form Expert generates the following code to open the table at run time.

```

procedure TEditPartsForm.FormCreate(Sender: TObject);
begin
    Table1.Open;
end;

```

This code is hooked to the form's *OnCreate* event, so Delphi executes it before creating the form. As a result, the table is opened before the form is displayed.

**Note** Code created by the Form Expert is like code you type yourself. If you rename components created by the expert, be sure to update your source file everywhere the renamed component occurs. Name changes to components that Delphi originates (for example, in the **type** declaration) are changed automatically for you by Delphi.

## For more information

For more information about

- Building Delphi forms, see the *User's Guide*.
- Naming components, see the *User's Guide*.
- The Delphi database architecture, see Chapter 1, "Introduction."
- *TTable* components, see page 8.
- The Fields Editor, see page 30.
- *TDataSource* components, see page 10.

## Building a master-detail form

---

The steps in this section show how to use the Database Form Expert to build a form containing two tables: a master table and a detail table, linked one-to-many. This form is the basis for the CUSTORD.DFM form in MASTAPP. The master table is CUSTOMER.DB and the detail table is ORDERS.DB. You can access both tables using the MAST alias. The expert links these tables and creates components to display data for one customer at a time, and for each customer, to display many orders.

### What to do

When you use the Database Form Expert, building a master-detail form is much like building a single-table form (for details, see page 21).

- 1 Choose Help | Database Form Expert to open the Form Expert.
- 2 In the first panel, specify a master-detail form that uses *TTable* objects.
- 3 In subsequent panels, specify the master table (CUSTOMER.DB), fields (use them all), and field layout (grid).
- 4 Specify the detail table (ORDERS.DB), fields (all), and field layout (grid).
- 5 Specify fields to link the master and detail tables as shown in the following figure, then tell the expert to create the form.



- 6 Press *F9* to run the form. Click the navigator control buttons to move through the records in the table.

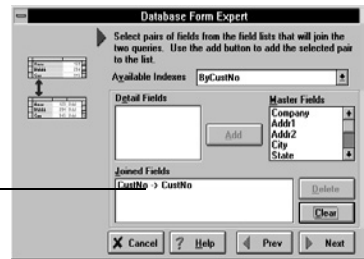
**Figure 2.4** Linking fields in a master-detail form



1. Choose an index. All Paradox tables have a primary index by default. ORDERS.DB was also created to have a secondary index named ByCustNo that orders records by customer number. Choose ByCustNo from the combo box.



2. When you choose an index, Delphi updates the lists of possible linking fields. In each list, choose CustNo, then click Add.

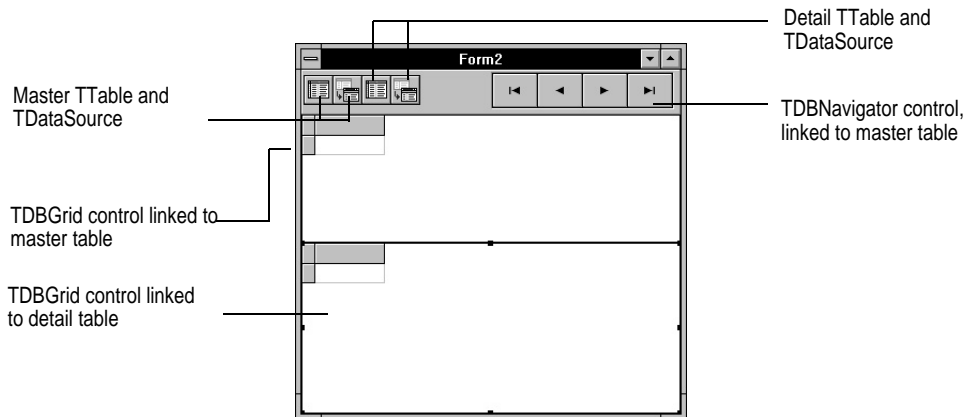


3. When you click Add, Delphi shows how the fields are linked. Click Next to continue.

## How it works

The expert builds a master-detail form much as it builds a single-table form (for details, see page 22). It creates *TTable* components and *TDataSource* components for the master table and the detail table and links them to the underlying data by setting properties. The expert creates controls to display the data from each table, and sets properties to link them to the corresponding *TDataSource* component. The expert also creates a *TDBNavigator* control linked to the master table.

**Figure 2.5** A master-detail form



What distinguishes a master-detail form is the link between the tables. For each record in the master table, the form displays all corresponding records in the detail table. Such a link is called a *one-to-many relationship* (see page 76 for more information). Delphi creates this relationship by setting properties of the *TTable* component linked to the detail table.

**Table 2.5** Important detail table properties

Property	Remarks
<i>IndexFieldNames</i>	Specifies the columns used to order the records in the table. You can also specify an index by name using the <i>IndexFields</i> property. For more information, see page 74.
<i>MasterFields</i>	Specifies which fields in the master table to link to. Use a semicolon to separate multiple field names. For more information, see page 76.
<i>MasterSource</i>	Specifies the <i>TDataSource</i> component linked to the master table. This data source provides a restricted view of the data in the detail table based on the values of the fields specified in <i>MasterFields</i> . For more information, see page 76.

The following table lists important properties for each component and control.

**Table 2.6** Important component properties for a master-detail form

Component	Property	Value	Remarks
<b>TTable (master)</b>	<i>Active</i>	False	When <i>Active</i> is False, data from this table is not displayed.
	<i>DataBaseName</i>	MAST	An alias that specifies where to find the table. Use aliases, not hard-coded paths, to make applications portable and easy to upsize.
	<i>Name</i>	Table1 (by default).	Use the Object Inspector to rename it (for example, Cust).
	<i>TableName</i>	CUSTOMER.DB	Specifies the master table.
<b>TDataSource (master)</b>	<i>DataSet</i>	Table1 (by default)	Specifies which <i>TTable</i> is supplying the data to this component.
	<i>Name</i>	DataSource1 (by default)	Use the Object Inspector to rename it (for example, CustSource).
<b>TTable (detail)</b>	<i>Active</i>	False	Data from this table is not displayed at design time.

**Table 2.6** Important component properties for a master-detail form (continued)

Component	Property	Value	Remarks
	<i>DataBaseName</i>	MAST	An alias that specifies where to find the table. Use aliases, not hard-coded paths, to make applications portable and easy to upsize.
	<i>IndexFieldNames</i>	CustNo	Specifies a column to use to order records in the table. <i>ByCustNo</i> is a secondary index based on <i>CustNo</i> . You can also set the <i>IndexFields</i> property to <i>ByCustNo</i> .
	<i>MasterFields</i>	CustNo	A list of one or more master table fields to link to. Use a semicolon to separate field names in the list.
	<i>MasterSourceName</i>	DataSource1 (by default) Table2 (by default)	Identifies a <i>TDataSource</i> linked to the master table. Use the Object Inspector to rename it (for example, Orders).
	<i>TableName</i>	ORDERS.DB	Specifies the detail table.
<b>TDataSource (detail)</b>	<i>DataSet</i>	Table2 (by default)	Specifies which <i>TTable</i> is supplying the data to this component.
	<i>Name</i>	DataSource2 (by default)	Use the Object Inspector to rename it (for example, OrdersSource).
<b>TDBGrid (master)</b>	<i>DataSource</i>	DataSource1 (by default)	Links the grid control to a <i>TDataSource</i> control.
<b>TDBGrid (detail)</b>	<i>DataSource</i>	DataSource2 (by default)	This data source provides a restricted view of the data in the <i>Orders</i> (detail) table, based on the values of the fields linked to the <i>Customer</i> (detail) table.
<b>TDBNavigator</b>	<i>DataSource</i>	DataSource1 (by default)	The expert links the <i>TDBNavigator</i> control to the master table.

### For more information

For more information about

- Linking tables, see step 6 on page 25.
- One to many relationships, see “Creating a master-detail form,” on page 76.
- Using the Database Form Expert, see page 21.

## Building a one-many-many form

This section describes how to build a form that displays data from three tables linked one-many-many. For example, one customer may place many orders, and each order may have many items. Use the Database Form Expert to create a master-detail form linking the *Customer* table to the *Orders* table as described on page 24. Then, to display and link the *Items* table, place components and set properties by hand. You can also use the techniques described here to build a master-detail form from scratch.

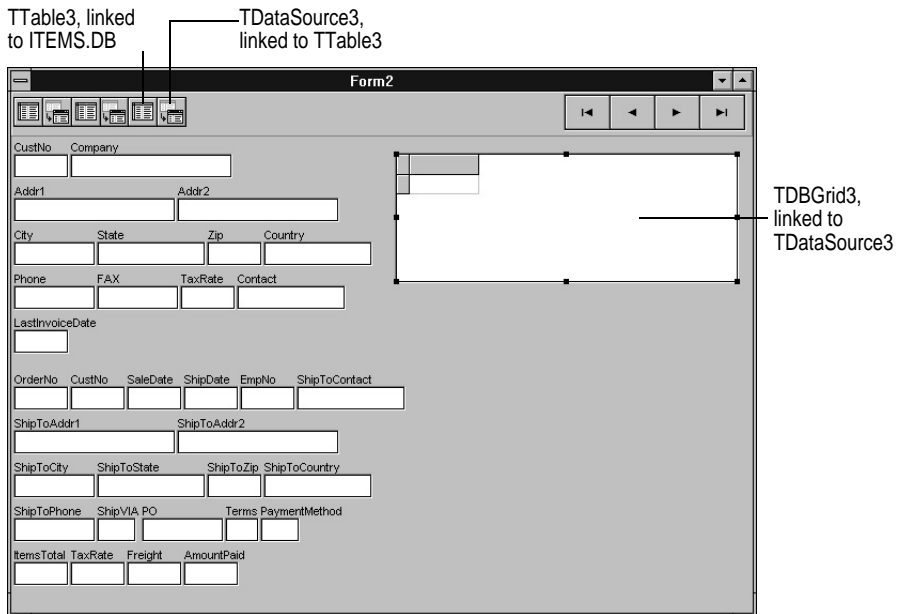
### What to do

- 1 Choose Help | Database Form Expert to open the Form Expert.
- 2 In the first panel, specify a master-detail form that uses *TTable* objects.

- 3 In subsequent panels, specify the master table (CUSTOMER.DB), fields (use them all), and field layout (horizontal).
- 4 Specify the detail table (ORDERS.DB), fields (all), and field layout (horizontal).
- 5 Specify fields to link the master and detail tables: choose CustNo for the *IndexFieldName* property and link the CustNo fields in each table. For details, see step 6 on page 25.
- 6 Tell the expert to create the form.
- 7 Move components around to make room at the bottom of the form. You may have to change the *Align* property of some controls from *alClient* to *alNone*.
- 8 Place a *TTable* component, a *TDataSource* component, and a *TDBGrid* component as shown in the following figure. (The *TTable* and *TDataSource* are on the Data Access components page; the *TDBGrid* is on the Data Controls page.) These components represent the third table in the one-many-many link. In this example its the *Items* table.

- 9 To create the link, set properties of these new components as shown in the following table.

**Figure 2.6** One-many-many form



**Table 2.7** Important component properties for a one-many-many form

Component	Property	Value	Remarks
<b>TTable</b> (third table)	<i>Active</i>	True	False by default. Set <i>Active</i> to True after setting all other properties to display data in linked controls.
	<i>DataBaseName</i>	MAST	An alias that specifies where to find the table. Use aliases, not hard-coded paths, to make an application portable and easier to upsize.
	<i>IndexFieldNames</i>	OrderNo	Specifies a column to use to order records in the table. <i>ByOrderNo</i> is a secondary index based on <i>OrderNo</i> . You can also set the <i>IndexFields</i> property to <i>ByOrderNo</i> .
	<i>MasterFields</i>	OrderNo	A list of one or more master table fields to link to. Use a semicolon to separate field names in the list.
	<i>MasterSource</i>	DataSource2 (by default)	Identifies a <i>TDataSource</i> component linked to the master table.
	<i>Name</i>	Table3 (by default).	Use the Object Inspector to rename it (for example, Items).
	<i>TableName</i>	ITEMS.DB	The name of the third table in the link.
<b>TDataSource</b> (third table)	<i>DataSet</i>	Table3 (by default)	Specifies which <i>TTable</i> is supplying the data to this component.
	<i>Name</i>	DataSource3 (by default)	Use the Object Inspector to rename it (for example, ItemsSource).
<b>TDBGrid</b> (third table)	<i>DataSource</i>	DataSource3 (by default)	This data source provides a restricted view of the data in the <i>Items</i> table, based on the values of the fields linked to the <i>Orders</i> table.

## How it works

A one-many-many form links data from three tables. The first table in the link (the *Customer* table) is the master table. The second table (*Orders*) does double duty: it's the detail table for the first table and the master table for the third table. The third table (*Items*) is a detail table for the second table. In this example, you can link these tables quickly and easily by setting properties because secondary indexes were specified for the tables when they were created.

## For more information

For more information about

- Using the Database Form Expert, see page 21.
- Creating tables and adding secondary indexes, see Appendix A, "Using Database Desktop."
- Linking tables, see step 6 on page 25.

# Working with fields

---

Examples in this section show how use code to control field values and display attributes. Delphi's data-aware controls enable end users to view and edit table data, but to access the underlying data or control how it is displayed, you want to create field components. Use the Fields Editor to define a list of fields and work with invisible components of type *TField*.

## Creating Tfield components

---

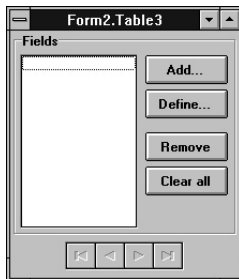
This example explains how to use the Fields Editor to define a list of some or all of the fields (columns) in a table. For each field, Delphi creates a corresponding *TField* component. *TField* components are invisible components that provide access to field values and display attributes. The following example shows how to use the Fields Editor to make a grid display four fields selected from the *Customer* table. It also shows how to specify the field order.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21.
- 2 To make the grid display data at design time, use the Object Inspector to change the *TTable* component's *Active* property to True.

- 3 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty, as shown in the following figure:

**Figure 2.7** The Fields Editor



- 4 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Click CustNo to select it, then control-click to select the Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box. In the form, the grid changes: instead of displaying all fields, it displays the only the fields you selected.

**Figure 2.8** Adding fields to a data set



- 5 Use the Fields Editor to change the field order as follows: Click LastInvoiceDate in the list of fields, then drag it to the third place in the list, between Company and Phone. In the form, the grid changes to display columns in their new order.
- 6 Close the Fields Editor by choosing Close from the Control menu.
- 7 Press *F9* to run the form. The grid displays the four fields in the order you specified.

## How it works

This form is the basis for the full-featured CUSTORD.DFM form in the MASTAPP application. By choosing fields in the Fields Editor, you can tell a *TTable* component which fields to make available to the components that are linked to it. In effect, the Fields Editor changes the logical structure of the table. The Fields Editor also adds *TField* objects to the unit's **type** section (for example, `CustCustNo: TFloatField;`). More accurately, it adds a descendant of the *TField* type appropriate for the data type of the field. For example, when you add the CustNo field to the data set, Delphi adds a *TFloatField* object; when you add the Company field, Delphi adds a *TStringField*, and so on. This tutorial uses the general term *TField* when the specific data type is unimportant.

**Note** Specifying a list of fields in the Database Form Expert is not the same as using the Fields Editor to define a dataset. The expert places components and controls in a form to create an initial layout, but you must use the Fields Editor to specify the fields in a data set. Use *TField* components, not visual controls, to access fields programmatically. Understanding the relationship between a *TTable* component, *TField* components, and data-aware controls is crucial to building database applications with Delphi.

### For more information

For more information about

- Data sets, the Fields Editor, and *TField* components, see “Using TFields and the Fields Editor” on page 79.
- Using the Database Form Expert, see page 21.

## Setting Tfield properties at design time

---

Because they are invisible, the only way to set the properties of *TField* components at design time is by using the Fields Editor. The following example shows how to set the properties of a *TField* component at design time.

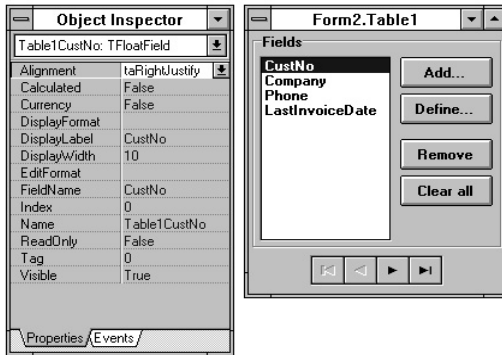
### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21. (If you’re working through these examples in sequence, you can use the form from the previous example and skip to step 5.)
- 2 To make the grid display data at design time, use the Object Inspector to change the *TTable* component’s *Active* property to True. This step is optional: it lets you see what happens to the form as you use the Fields Editor.
- 3 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 4 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Control-click to select only the CustNo, Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box. In the form, the grid changes: instead of displaying all fields, it displays the only the fields you selected.
- 5 Click the CustNo field in the Fields Editor’s field list and view the properties and values displayed in the Object Inspector. (To open the Object Inspector, choose View | Object Inspector.) Notice that the component’s *Name* property is Table1CustNo. Delphi generates this name automatically by appending the field name to the name of the associated *TTable* component. Use this name to refer to the *TField* component in code.



- 6 Change the *Align* property from *taRightJustify* (the default) to *taCenter*. In the form, the grid changes: values in the *CustNo* column are centered.

**Figure 2.9** TField component properties



To display a TField component's properties in the Object Inspector, choose a field name from the list in the Fields Editor.

Delphi generates a TField component's name by appending the field name to the name of the associated TTable component. Use this name to refer to the TField in code.

### How it works

When you select a field listed in the Fields Editor, you can use the Object Inspector to set an invisible *TField* component's properties, just as you would a visible control. Note that a field's display attributes are properties of the *TField* component, not of the control that displays the value! You only have to set a property once for the *TField* component. Linked *Cust* controls use the settings automatically. This is a guiding principle of Delphi database applications: use *TField* components to work with database fields.

The following table lists important *TField* design-time properties.

**Table 2.8** Important TField design-time properties

Property	Remarks
<i>Alignment</i>	Specifies how to display the field value: left-justified, right-justified, or centered.
<i>Calculated</i>	When True, this field isn't stored in the table but instead is calculated, record by record, by Object Pascal code. Write this code in the table's <i>OnCalcFields</i> event handler.
<i>DisplayLabel</i>	The string used by grids as the column header for the corresponding field.
<i>DisplayWidth</i>	The number of characters that a grid column uses to display this field in a grid.
<i>DisplayFormat</i> and <i>EditMask</i>	Provides control over input characters for some fields types as they are displayed and edited, for example, in working with commonly formatted values such as dates and telephone numbers.
<i>FieldName</i>	The name of the field in the underlying table. Example: <i>CustNo</i> .
<i>Index</i>	The field's logical position in the data set (first position = 0).
<i>Name</i>	Formed by appending the field name to the name of the <i>TTable</i> component. Example: Given a <i>TTable</i> name of <i>Customer</i> and a field name of <i>CustNo</i> , the <i>TField</i> 's name is <i>CustomerCustNo</i> . This name is used for programmatic access to field's properties (for example: <i>CustomerCustNo.Value</i> ).
<i>ReadOnly</i>	When <i>ReadOnly</i> is True, data in the corresponding field cannot be modified.
<i>Visible</i>	When True, the corresponding field appears in grids linked to this <i>TField</i> .

### For more information

For more information about

- Setting a *TField* component's properties, see page 79.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

## Reading field values

---

Every line in the Fields Editor's list of fields represents a Delphi component type *TField*. When you add a field to the list in the Fields Editor, you add a new component to the form (the code appears in the unit's **type** section). Delphi provides *TField* descendents for every available field type. For example, in the *Customer* table, *CustNo* is of type *TFloatField*, and *Company* is of type *TStringField*. This example shows how to build a form so you can read field values at run time. You get a field value by reading the *Value* property of the corresponding *TField* component.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of *CUSTOMER.DB* in a grid. For detailed instructions, see page 21.
- 2 Use the Object Inspector to change the form's *Name* property to *TutorialForm*.
- 3 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 4 Click **Add** to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Control-click to select only the *CustNo*, *Company*, *Phone*, and *LastInvoiceDate* fields, then click **OK** to confirm your choices and close the dialog box. The Fields Editor remains open.
- 5 Place a button control and a standard edit box anywhere in the panel that contains the *TDBNavigator* control (you may need to resize the panel), then set the edit box *Name* property to *OutputField*.
- 6 Attach the following code to the button's *OnClick* event:

```
procedure TTutorialForm.Button1Click(Sender: TObject);
begin
    OutputField.Text := Table1Company.Value;
end;
```

- 7 Press **F9** to run the form. Click a field in the grid, then click the button. The text in the edit control displays the value of the *Company* field for the current record.

### How it works

The code in step 7 does a simple assignment: the edit control's *Text* property gets the value of the *TField* component named *Table1Company*, where the *TField*'s value is the value of the corresponding field in the current record of the table.

**Note** In code, refer to a *TField* by its component name, don't use the name of the field in the table. For example, use *Table1Company*, not *Company*.

You can assign the *TField* component's value directly to the *Text* property because *Table1Company* is of type *TStringField*. An edit box's *Text* property and a *TStringField*'s

*Value* property are of compatible types, so you don't need to do a conversion. However, the following code generates a type mismatch error at compile time, because `Table1CustNo` is of type *TFloatField*.

```
OutputField.Text := Table1CustNo.Value; {Causes a type mismatch error.}
```

To display a numeric field's value in an edit box, convert the value as follows.

```
OutputField.Text := Table1CustNo.AsString;
```

This code uses the *TField* property *AsString* to read the field value and convert it to a string before assigning it to the edit box's *Text* property. *TField* components have the following properties for converting values: *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger*, and *AsString*.

The following code shows examples of how to read field values, display them in edit boxes, and assign them to variables.

```
var
    CustNoDouble: Double;
    CustNoInt: Integer;
    CustNoString: String;
begin
    {Display field value in edit control.}
    OutputField.Text := Table1Company.Value; {Compatible types, no conversion.}
    OutputField.Text := Table1CustNo.AsString; {Convert field value to compatible type.}

    {Assign field value to variables.}
    CustNoDouble := Table1CustNo.Value; {Compatible types, no conversion.}
    CustNoInt := Table1CustNo.AsInteger; {Convert field value to compatible type.}
    CustNoString := Table1CustNo.AsString; {Convert field value to compatible type.}
end;
```

## For more information

For more information about

- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

## Assigning values to fields

---

This example shows how to build a form so you can assign field values at run time. You write to a field by assigning a value to the corresponding *TField* component.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of `CUSTOMER.DB` in a grid. For detailed instructions, see page 21. (If you're working through these examples in sequence, you can use the form from the previous example and skip to step 5.)
- 2 Use the Object Inspector to change the form's *Name* property to `TutorialForm`.

- 3 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 4 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Control-click to select only the CustNo, Company, Phone, and LastInvoiceDate fields, then click OK to confirm your choices and close the dialog box.
- 5 Place a button control and a standard edit box anywhere in the panel that contains the *TDBNavigator* control (you may need to resize the panel).
- 6 Attach the following code to the button's *OnClick* event to change a company's name in the CUSTOMER table based on the current value in the standard edit box:

```

procedure TTutorialForm.Button1Click(Sender: TObject);
begin
    Table1.Edit;
    Table1Company.Value := Edit1.Text;
    Table1.Post;
end;

```

- 7 Press *F9* to run the form. Type a value into the edit box, then click the button and notice the value of the Company field in the current record in the grid. It should be the same as the text in the edit box.

### How it works

The code in step 6 does three things: it puts the table into Edit state, assigns a value to the Company field of the current record, and posts the modified record back to the table (which takes the table out of Edit state). As this example shows, there is a difference between editing field values interactively using controls and editing field values in code. By default, the *AutoEdit* property of a *TDataSource* component is set to True. A user can type into a data-aware control to that data source and modify the value immediately. When the user moves off that record, changes are posted automatically. But, to modify *TField* component values based on values entered in a standard edit control, or to perform table modifications in code, you need to explicitly switch to Edit state, set the value, and post changes.

**Note** You can safely call *Edit* even if you're already in Edit or Insert state. In such cases, calls to *Edit* have no effect.

The *TField* component and the value you assign must be compatible. In the example, the edit control's *Text* property is compatible with *Table1Company*, a *TStringField*. However, the following code generates a type mismatch error at compile time, because *Table1CustNo* is a *TFloatField*.

```

Table1CustNo.Value := OutputField.Text; {Causes a type mismatch error.}

```

To assign the text of an edit control to a numeric field, convert the text as follows.

```

Table1CustNo.AsString := OutputField.Text;

```

This code uses the *TField* property *AsString* to convert the text before assigning it to *Table1CustNo*. *TField* components have the following properties for converting values: *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger*, and *AsString*.

The following code shows examples of how to assign field values, converting them if necessary.

```
Table1CustNo.Value := 12340;  
Table1CustNo.AsInteger := 4321;  
Table1CustNo.AsString := '5678';  
  
Table2CustNo.Value := Table1CustNo.Value; {Assign value of one TField to another.}  
Table2.Fields[0] := Table1.Fields[0]; {Also assign value of one TField to another.}
```

### For more information

For more information about

- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

## Defining a calculated field

---

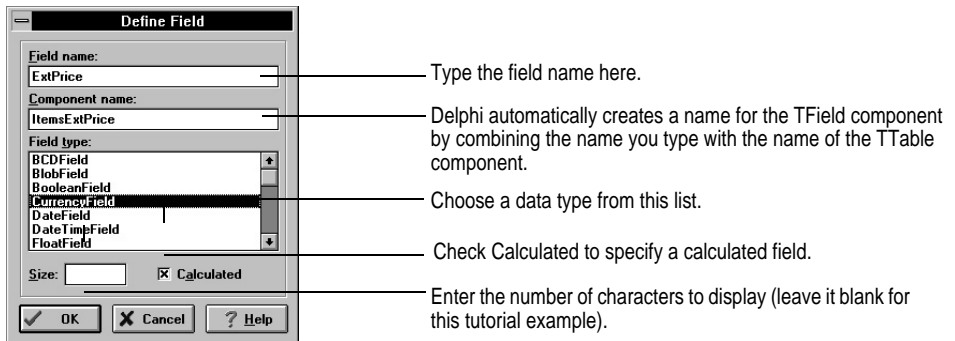
This example shows how to use the Fields Editor and *TField* components to define a calculated field. It calculates the extended price of an item, based on the unit price, quantity ordered, and discount values in the *Items* table. For more information about defining calculated fields, see page 82.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of *ITEMS.DB* in a grid. For detailed instructions, see page 21.
- 2 Use the Object Inspector to change the form's *Name* property to *OrderForm*, change the *TTable* component's *Name* property to *Items*, and change the *TTable* component's *Active* property to *True*.
- 3 In this form's *CreateForm* procedure, change *Table1* to *Items*.
- 4 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 5 Click *Add* to open a dialog box listing the fields in the *Items* table. By default, all fields are selected. Click *OK* to add all the fields to the data set and close the dialog box. In the form, the grid displays a column for each field in the data set.

- Click Define to open the Define Field dialog box, then enter specifications for a calculated field named ExtPrice as shown in the following figure:

**Figure 2.10** Defining a calculated field



- Click OK to accept values and close the dialog box. In the form, an empty ExtPrice column appears in the grid. You may have to scroll the grid to the right to see it.
- Click Define again, then enter specifications for a second calculated field named ItemsSellPrice. It, too, should be given a CurrencyField data type. Click OK.
- Double click the *OnCalcFields* event of the *TTable* component to open the code window, then enter the following code.

```

procedure TOrderForm.ItemsCalcFields(DataSet: TDataSet);
begin
    if Parts.FindKey([ItemsPartNo]) then
    begin
        ItemsDescription.Value := PartsDescription.Value;
        ItemsSellPrice.Value := PartsListPrice.Value;
    end;
    ItemsExtPrice.Value := ItemsQty.Value *
        ItemsSellPrice.Value * (100 - ItemsDiscount.Value) / 100;
end;

```

- Type *F9* to run the form. The ExtPrice column fills with calculated values. A calculated field does not display values at design time.

### How it works

This form is the basis for the full-featured EDORDERS.DFM form in MASTAPP. It demonstrates the two key aspects of creating a calculated field in Delphi: adding the calculated field to a *TTable* component's data set, and writing code to handle the *TTable* component's *OnCalcFields* event. Delphi sends an *OnCalcFields* event each time the cursor for that table changes. The calculation accesses data through the *TField* components created in the Fields Editor, not the controls placed in the form.

The example attaches code to the *OnCalcFields* event to update the value of the calculated field. You can use the *OnCalcFields* event for other purposes; for example, to do a lookup into another table, perform a complex calculation, or do real-time data

acquisition. However, in an *OnCalcFields* event, you can only assign values to calculated fields.

### For more information

For more information about

- Programming calculated fields, see page 82.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

## Formatting field values at design time

---

This example shows how to format a field value by setting the *DisplayFormat* property at design time.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB. Specify a vertical layout and labels aligned left. For detailed instructions, see page 21.
- 2 Use the Object Inspector to change the form's *Name* property to *EdCustForm*, change the *TTable* component's *Name* property to *Cust*, and change the *TTable* component's *Active* property to *True*. In the form, notice the Tax Rate field's display format (example: 8.5).
- 3 In this form's *CreateForm* procedure, change *Table1* to *Cust*.
- 4 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 5 Click *Add* to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Click *OK* to add all the fields to the data set and close the dialog box.
- 6 In the Field Editor's list of fields, choose *Tax Rate*, then use the Object Inspector to view the properties of the *TFloatField* named *CustTaxRate*. The *DisplayFormat* property is blank.
- 7 In the Object Inspector, enter the following value for the *DisplayFormat* property of *CustTaxRate*: *0.00%*. In the form, the Tax Rate field's display format changes (example: 8.50%).

## How it works

Delphi provides several properties that specify a field's display format. The following table lists some of the most frequently-used properties. Any data-aware control linked to a *TField* component uses the *TField*'s display properties.

**Table 2.9** Important TField design-time properties

Property	Remarks
<i>Alignment</i>	Displays field value left justified, right justified, or centered within a data-aware control.
<i>Currency</i>	True, numeric field displays monetary values. False, numeric field does not display monetary values.
<i>DisplayFormat</i>	Specifies the format of data displayed in a data-aware component. For more information about valid patterns, see the online help.
<i>DisplayWidth</i>	The number of characters that a grid column uses to display this field in a grid.
<i>DisplayFormat</i> and <i>EditMask</i>	Provides control over input characters for some fields types as they are displayed and edited; for example, in working with commonly formatted values such as dates and telephone numbers.
<i>FieldName</i>	The name of the field in the underlying table; for example, CustNo.
<i>Index</i>	The field's logical position in the data set (first position = 0).
<i>Name</i>	Formed by appending the field name to the name of the <i>TTable</i> component. Example: Given a <i>TTable</i> name of Customer and a field name of CustNo, the <i>TField</i> 's name is CustomerCustNo. This name is used for programmatic access to field's properties (for example, CustomerCustNo.Value).
<i>Visible</i>	When True, the corresponding field can appear in linked grids.

## For more information

For more information about

- Formatting field values, see page 83.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

## Formatting field values at run time

This example shows how to format a field value by setting the *DisplayText* property in code.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB. (If you're working through these examples in sequence, you can use the form from the previous example and skip to step 6.) Specify a vertical layout and labels aligned left. For detailed instructions, see page 21.
- 2 Use the Object Inspector to change the form's *Name* property to EdCustForm, change the *TTable* component's *Name* property to Cust, and change the *TTable* component's *Active* property to True. In the form, notice the Phone field displays U.S phone numbers (example: 808-555-0269).
- 3 In this form's *CreateForm* procedure, change Table1 to Cust.



- 4 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 5 Click Add to open a dialog box listing the fields in the *Customer* table. By default, all fields are selected. Click OK to add all the fields to the data set and close the dialog box.
- 6 In the Fields Editor's list of fields, choose Phone, then use the Object Inspector to view the events of the *TStringField* named *CustPhone*.
- 7 Double-click the *OnGetText* event to open the code window, then add code to handle the event as follows.

```

procedure TEdCustForm.CustPhoneGetText (Sender: TField;
    var Text: OpenString; DisplayText: Boolean);
begin
    if DisplayText then
        begin
            Text := CustPhone.Value;
            Delete(Text, 4, 1);
            Insert('(', Text, 1);
            Insert(')', Text, 5);
        end;
    end;
end;

```

In this sample code, *DisplayText* is True, meaning you're not actually editing the field in question, but modifying the display of the field. When *DisplayText* is True, characters that would be invalid for the user to enter, such as parentheses in this case, can be inserted into the field for display purposes only.

- 8 Press *F9* to run the form. The Phone field's display format changes (example: (808)555-0269).

### How it works

Use the *OnGetText* event to format field values. Delphi calls a *TField* component's *OnGetText* method whenever it's about to display the value of a field onscreen, for example, when redrawing a data-aware component. Delphi ignores a *TField* component's *DisplayFormat* property when you add code to its *OnGetText* event.

In this example, *CustPhoneGetText* copies the phone number string to variable *Text*, then uses the *Insert* and *Delete* procedures to place parentheses around the first three digits. This gives U.S. phone numbers a more traditional format: (713) 555-1212, instead of 713-555-1212 as stored in the file. The *DisplayText* property is True when a *TField* is displaying data but is not available for editing. When a *TField* is in Edit state, *DisplayText* is *False*.

**Note** *DisplayFormat* and *EditMask* are ignored if an *OnGetText* event handler exists.

### For more information

For more information about

- Formatting fields, see page 85.
- Editing display properties, see page 83.

- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.
- Setting a *TField* component's properties, see page 79.

## Searching for field values

---

This example shows how to search for values in keyed (indexed) fields in a table. To search for values in unkeyed fields, use a query. See “Using queries and ranges” on page 47 for more information.

### What to do

This example shows how to display a record in the Customer table by searching for a customer number entered by the user.

- 1 Use the Database Form Expert to build a single-table form that displays fields from CUSTOMER.DB. Specify a vertical layout and labels aligned left. For detailed instructions, see page 21.
- 2 Place a button control in the panel that contains the *TDBNavigator* control.
- 3 Double-click the button to open a code window, then add code to handle the button's *OnClick* event as follows.

```
procedure TForm2.Button1Click(Sender: TObject);
var UserCustNo: String;
begin
  UserCustNo := InputBox('Search', 'Enter a Customer Number:', ' ');
  if not Table1.FindKey([UserCustNo]) then
    MessageDlg('Not found.', mtInformation, [mbOK], 0);
end;
```

- 4 Add the *Dialogs* unit to the **uses** clause of your form unit. The *InputBox* function used in the previous step resides in the *Dialogs* unit.
- 5 Press *F9* to run the form. Click the button and enter a number in the input dialog box (try 1560). If that number is a customer number in the table, Delphi displays the corresponding record. Otherwise, it displays a dialog box.

### How it works

*FindKey* takes an array, where each array value represents a value to search for in the corresponding key field in the table. *FindKey* searches for the first array value in the first key field, the second array value in the second key field, and so on. In this example, the array contains one value, so *FindKey* searches for it in the first key field of the table, which is *CustNo*.

The following example shows how to search for the name Frank P. Borland in a table where the key fields are *LastName*, *FirstName*, and *MiddleInitial*.

```
begin
  if not Table1.FindKey(['Borland', 'Frank', 'P.']) then
    MessageDlg('Not found.', mtInformation, [mbOK], 0);
end;
```

## For more information

For more information about

- Searching for field values, see page 70.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

## Validating data entry

---

This section describes ways to make sure the user doesn't enter invalid data into tables. There are three different approaches to validating data: table-based, field-based, and record-based.

Most forms use a combination of validation techniques.

- **Table-based validation.** Build in validity checks when you create tables. Using the Database Desktop, you can specify a wide range of validity checks, including minimum and maximum values, formatting pictures, and referential integrity, all without writing a line of code. Table-based validation rules are enforced by the database when the data is posted. Multiple exceptions will be raised, one after another, until all fields match the validation criteria. Delphi enforces some table-based validation rules before posting (for example, whether a field is required or not).

For more information about the DBD, see Appendix A, "Using Database Desktop."

- **Field-based validation.** There are two methods:

Create an *OnValidate* event handler. The handler is called when the field's value is modified, whether by user input or when the field is assigned a value in code.

For fields that must get values from a user (for example, a password or part number) set the *TField's Required* property to True, and write an *OnValidate* event handler for the field. Before a record is posted, an exception occurs for any fields that have a **nil** value.

- **Record-based validation.** Use this approach when other fields are involved in determining if a field is valid. For example, if it is invalid to use a credit card for purchases under \$10, you want to give a user the chance to enter both a payment method and an invoice amount before performing validation on the payment method. Otherwise, since a new order starts with a zero invoice amount, you would require the entry of all line items before permitting specification of a payment method.

Record-based validation should be performed in a table's *BeforePost* event handler. Raising an exception there prevents posting from occurring. Use the *TField FocusControl* method to inform the user which field is invalid. For an example of this technique, see "Writing code to check field values" on page 46.

## Using lists and lookups

The following examples show two techniques for using lists to restrict user input. The first example presents a list of items entered by hand. The second example presents a lookup filled with items from a field (column) of a table.

**Figure 2.11** Lists and lookups

The Terms field is a *TDBComboBox* control. The list items are hard-coded. See page 44.

The SoldBy field is a *TDBLookupCombo* control. It reads list items from a field (column) of a table. See page 45.

### What to do (list)

The following steps describe how to create the drop-down list labeled Terms in the Edorders form in MASTAPP. The Terms field specifies the payment terms for an order. Allowed values are Prepaid, Net 30, and COD.

- 1 Open the MASTAPP project, then open the Edorders form.
- 2 Place a *TTable* component. Set its *DatabaseName* and *TableName* properties to link it to the *Orders* table. Change its *Name* property to *Orders*.
- 3 Place a *TDataSource* component. Set its *DataSet* property to the *Orders TTable*. Change its *Name* property to *OrdersSource*.
- 4 Place a *TDBComboBox* control. Set its properties as shown in the following table.

**Table 2.10** Important *TDBComboBox* properties

Property	Value	Remarks
<i>DataField</i>	Terms	The name of the field in the table this control is linked to.
<i>DataSource</i>	OrdersSource	The <i>TDataSource</i> component this control is linked to.
<i>Items</i>	Prepaid Net 30 COD	Each list item is on its own line. In other words, when you enter list items, press <i>Enter</i> after each item.

## How it works

A *TDBComboBox* control works like a standard combo box control, but it's linked to a field in a table. Its *Items* property stores the list items. When the user chooses an item from the list, that value is assigned to the corresponding data field in the current record. You can add list items by hand at design time or specify them in code at run time.

## For more information

For more information about

- Using lists and combo boxes, see “Using list and combo boxes” on page 109.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

## What to do (lookup)

The following steps describe how to create the drop-down lookup list labeled *SoldBy* in the *Edorders* form in *MASTAPP*. The combo box labeled *SoldBy* displays a list of employee names, but it stores employee numbers. When the user chooses a name from the list, Delphi looks up the corresponding employee number and writes that value to the *Orders* tables.

- 1 Open the *MASTAPP* project, then open the *Edorders* form.
- 2 Place two *TTable* components. Set the first component's *DatabaseName* and *TableName* properties to link it to the *Orders* table, and change its *Name* property to *Orders*. Set the second component's *DatabaseName* and *TableName* properties to link it to the *Employee* table, and change its *Name* property to *Emps*.
- 3 Place two *TDataSource* components. Set the first component's *DataSet* property to the *Orders TTable* and change its *Name* property to *OrdersSource*. Set the second component's *DataSet* property to the *Emps TTable* and change its *Name* property to *EmpsSource*.
- 4 Place a *TDBLookupCombo* control. Set its properties as shown in the following table.

**Table 2.11** Important *TDBComboBox* properties

Property	Value	Remarks
<i>DataSource</i>	<i>OrdersSource</i>	The <i>TDataSource</i> component this control is linked to.
<i>DataField</i>	<i>EmpNo</i>	The name of the field in the table this control is linked to.
<i>LookupSource</i>	<i>EmpsSource</i>	The <i>TDataSource</i> component used to identify the table from which to look up field values.
<i>LookupDisplay</i>	<i>FullName</i>	The field (column) this control reads from to display list values to the user. <i>FullName</i> is a calculated field in the table pointed to by <i>LookupSource</i> .
<i>LookupField</i>	<i>EmpNo</i>	The field to search for a value corresponding to the value in the <i>LookupDisplay</i> field. This value is assigned to the field specified in <i>DataField</i> .

## How it works

The *TDBLookupCombo* control dynamically accesses a column from a table (specified in *LookupDisplay*) and displays it to the user. When the user chooses an item, Delphi finds

the value in the table specified in *LookupSource*. Then it reads the value of the field specified in *LookupField*. Finally, it assigns that value to the field specified in *DataField* in the table specified in *DataSource*.

In this example, Delphi fills a drop-down box with employee names from the *FullName* field of the *Employee* table. *FullName* is a calculated field defined for the *Employee* table. To see how *FullName* is calculated, select *Employee TTable* component, then examine its *OnCalcFields* event. When the user chooses a name from the list, Delphi reads the *EmpNo* field to get the employee number for that name. Then it assigns that value to the *EmpNo* field of the *Orders* table.

### For more information

For more information about

- Using lists and combo boxes, see “Using list and combo boxes” on page 109.
- Using the Fields Editor, see page 30.
- Using the Database Form Expert, see page 21.

## Writing code to check field values

---

This example shows how to validate a field value after the user has entered it. The following steps show how to add code to the Edorders form in MASTAPP to disallow a sale date later than the current date. The example uses the standard function *Now* to get the current timestamp.

### What to do

- 1 Open the MASTAPP project, then open the Edorders form.
- 2 Double-click the *TTable* component named Orders to open the Fields Editor.
- 3 In the Fields Editor’s list of fields, click *SaleDate*, then use the Object Inspector to view the events of the *TDateTimeField* component named *OrdersSaleDate*.
- 4 Double-click the *OnValidate* event to open the code window, then add code to handle the *OnValidate* event as follows.

```
procedure TOrderForm.OrdersSaleDateValidate(Sender: TField);
begin
  if OrdersSaleDate.Value > Now then
    raise Exception.Create('Cannot enter a future date');
end;
```

### How it works

To check field values after the user enters them, write code to handle the *OnValidate* event of the appropriate *TField* component. The example code raises an exception if the date a user enters is later than today’s date. Make sure all fields are either initialized to a valid value in an *OnNewRecord* event, or have their *TField Required* properties set to *True*.

### For more information

For more information about

- Working with *TField* components, see page 34.
- Using the Fields Editor, see page 30.

## Using queries and ranges

---

The examples in this section show how to select a set of records from a table. In Delphi, you can do this by issuing SQL statements or by setting a range (filter). SQL statements can be either static or dynamic; that is, they can be fixed or include parameters where values are provided at run time. For more information, see Chapter 5, “Using SQL in applications.”

### What to do (static query)

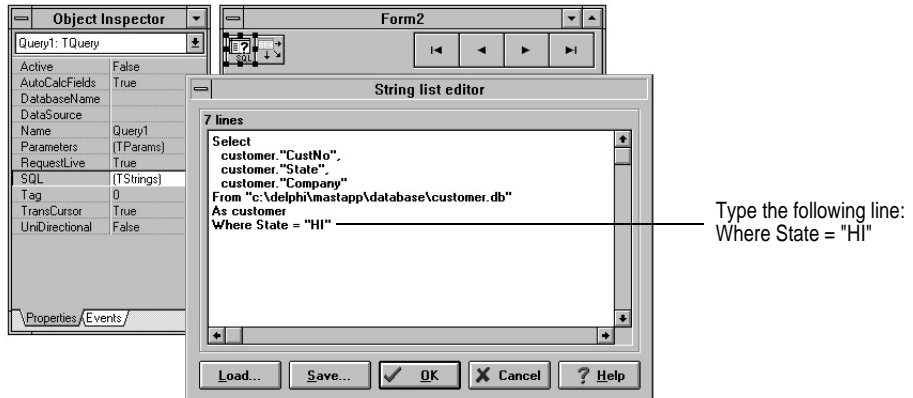
This example shows how to create and execute a static query and display the results in a form.

- 1 Use Form Expert to build a form based on a query of the *Customer* table (for detailed instructions, see page 21). Use only the *CustNo*, *Company*, and *State* fields (for simplicity). Specify a grid layout.
- 2 Use the Object Inspector to set the *TQuery* component's *Active* property to True. The grid displays the query results (by default, the query selects all records).
- 3 Set the *TQuery* component's *Active* property to False. The grid empties. (*Active* must be False to change the query.)
- 4 Use the Object Inspector to display the *TQuery* component's *SQL* property, and type the following statement after the last line in the SQL statement in the String List Editor window:  

```
where State = "HI"
```
- 5 Click OK to close the String List Editor.

- 6 Set the *TQuery* component's *Active* property to True. The grid displays only those records where the value of the *State* field is "HI." (Optional: Press *F9* to run the form.)

**Figure 2.12** Setting a *TQuery*'s SQL property



### How it works

Delphi reads the SQL statements assigned to the *TQuery* component's *SQL* property and passes them to the server without any intermediate interpretation. (Use double quotes in SQL statements, not single quotes as in Pascal code.) The server executes the query and returns the results to Delphi, and Delphi displays the result set in the grid.

**Note:** By default, a *TQuery* component's *RequestLive* property is set to False. When *RequestLive* is False, the result set returned by a query is read-only. Users cannot modify data in the form. If you want users to be able to update data returned by a query of a single table, then you can set *RequestLive* to True to request a live result set. If the query is updateable, *CanModify* returns True.

### For more information

For more information about

- Using *TQuery* components and SQL in applications, see Chapter 5.
- *RequestLive* behavior and restrictions, see Chapter 5, "Using SQL in applications."
- Using the Database Form Expert, see page 21.

### What to do (dynamic query)

This example shows how to create and execute a dynamic query and display the results in a form. A dynamic query consists of SQL statements that include parameters (variables) whose values can be assigned at design time or run time. Steps 1 through 6 show how to set parameters at design time. Steps 7 through 9 show how to do it at run time.

- 1 Use Form Expert to build a form based on a query of the *Customer* table (for detailed instructions, see page 21). Use only the *CustNo*, *Company*, and *State* fields (for simplicity). Specify a grid layout. If you're working through these examples in sequence, you can use the form from the previous example.

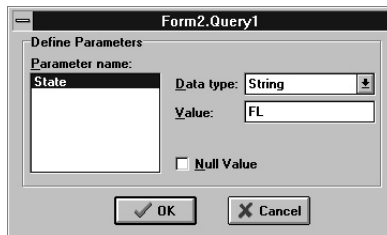


- 2 Use the Object Inspector to display the *TQuery* component's *SQL* property, and type the following statement after the last line in the SQL statement in the String List Editor window:

**where State = :State**

- 3 The SQL statement in the previous step contains the field name *State* and the parameter *State* (a parameter is an arbitrary string preceded by a colon). Click OK to close the String List Editor.
- 4 Right-click the *TQuery* component, then choose Define Parameters from the pop-up menu. The Define Parameters dialog box opens, ready for you to assign a field type and a value to the parameter *State*.
- 5 Choose a field type of String, and enter a value of FL, then click OK to close the dialog box.

**Figure 2.13** Defining a query parameter



To define the parameter *State*, choose a field type of String and enter a value of FL.

- 6 Set the *TQuery* component's *Active* property to True. The grid displays the query results (all records where *State* = FL).
- 7 Place a button control in the panel that contains the *TDBNavigator* control.
- 8 Double-click the button to open a code window, then write the following code to handle its *OnClick* event.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    Query1.DisableControls;
    try
        Query1.Active := False;
        Query1.Params[0].AsString := 'HI';
        Query1.Active := True;
    finally
        Query1.EnableControls;
    end;
end;
```

- 9 Press *F9* to run the form, then click the button. The grid displays the query results (all records where *State* = HI).

## How it works

In a SQL statement, a string preceded by a colon (like *:State* in the example) represents a parameter. At design time, Delphi recognizes parameters and displays them in the

Define Parameters dialog box. Then at run time, it assigns the specified values to the corresponding parameters.

You can assign values to query parameters at run time using the *Params* property. It stores parameter values in an array, where an index of 0 represents the first parameter, an index of 1 represents the second parameter, and so on. So, this statement assigns the value HI to the first parameter.

```
Query1.Params[0].AsString := 'HI';
```

You need to set the *TQuery* component's *Active* property to False before setting parameter values, then set it to True again to update the query. The calls to *DisableControls* and *EnableControls*, respectively, freeze and restore display capabilities of controls linked to the *TQuery* component. This disables data-aware control while the result set is updated.

### For more information

For more information about

- Dynamic SQL statement, see page 121.
- Using SQL in applications, see Chapter 5.
- Using TQuery components, see page 115.
- The SQL property, see page 117.
- Using the Database Form Expert, see page 21.

## Setting a range

---

This example explains how to select a set of records by setting a range (also called a filter). To set a range on a Paradox or dBASE table, work with keyed (indexed) fields. To set a range on a SQL table, you can specify the fields to be used as indexes using the *IndexFieldNames* property.

### What to do

- 1 Use the Database Form Expert to build a single-table form that displays all the fields of CUSTOMER.DB in a grid. For detailed instructions, see page 21.
- 2 Use the Object Inspector to change the form's *Name* property to CustForm, change the *TTable* component's *Name* property to Cust, and change the *TTable* component's *Active* property to True.
- 3 In this form's *CreateForm* procedure, change Table1 to Cust.
- 4 Open the Fields Editor by double-clicking the *TTable* component. By default, the list of fields is empty.
- 5 Click Add to open a list box for the fields in the *Items* table. By default, all fields are selected. Click OK to add all the fields to the data set and close the list box. Close the Fields Editor. In the form, the grid displays a column for each field in the data set.
- 6 Place a button control in the panel that contains the *TDBNavigator* control.

- 7 Double-click the button to open a code window, then add code to handle the button's *OnClick* event as follows.

```
procedure CustForm.Button1Click(Sender: TObject);
begin
    Cust.DisableControls;
try
    Cust.SetRangeStart;
    CustCustNo.Value := 3000;
    Cust.KeyExclusive := False;

    Cust.SetRangeEnd;
    CustCustNo.Value := 4000;
    Cust.KeyExclusive := True;

    Cust.ApplyRange;
finally
    Cust.EnableControls;
end;
end;
```

- 8 Press *F9* to run the form, then click the button. The grid displays records for customer numbers from 3,000 to 3,999.

### How it works

This example sets a range to include records for customer numbers from 3,000 to 3,999. Setting a range on a table affects the values displayed in data-aware components linked to that table. That's why the routine in step 7 begins by calling *Cust.DisableControls* and ends by calling *Cust.EnableControls*. *DisableControls* disables the display capabilities of controls linked to the specified table; in effect, it freezes them. *EnableControls* restores the controls to an active state.

**Important** Always use a **try...finally** statement, followed by *EnableControls*. Otherwise, after an exception occurs, data controls are inactive.

The call to *Cust.SetRangeStart* marks the beginning of a code block that sets the minimum value of the range. The next line assigns a value to the *TField* component named *CustCustNo*. (*CustNo* is the first and only key field in the *Customer* table.) Setting *Cust.KeyExclusive* to *False* indicates that the value is not to be excluded in the range. So, in this example, a customer number 3,000 would be included in the range.

The call to *Cust.SetRangeEnd* marks the beginning of a code block that sets the maximum value of the range. The next line assigns a value to *CustCustNo*. Setting *Cust.KeyExclusive* to *True* indicates that the value is to be excluded from the range. A customer number 4,000 would not be included.

The call to *Cust.ApplyRange* puts the range settings into effect. (To cancel a range, call *CancelRange*.)

### For more information

For more information about

- Setting ranges and filtering data, see page 72.
- Using the Fields Editor, see page 30.

# Printing reports and forms

---

This section describes how to use Delphi to print reports created using ReportSmith, and how to print a Delphi form. Designing reports with ReportSmith is described in *Creating Reports*.

**Note** To run reports, the *TReport* component needs to locate the RS\_RUN directory. You can put the RS\_RUN directory in your DOS PATH, or you can put an "EXEpath =" entry in the Delphi RS\_RUN.INI file.

## What to do: printing reports

This example show how to print a ReportSmith report that lists MAST customers.

- 1 Choose File | New Form to open the Browse Gallery, then choose Blank form and click OK to create a blank form.
- 2 Place a *TReport* component anywhere in the form. (The *TReport* component is on the Data Access components page.)
- 3 Use the Object Inspector to set the *TReport* component's properties as shown in the following table.

**Table 2.12** Important TReport properties

Property	Value	Remarks
<i>Preview</i>	True	When <i>Preview</i> is True, Delphi displays the report onscreen only; when <i>Preview</i> is False, Delphi sends the report to the printer.
<i>ReportDir</i>	C:\DELPHI\ DEMOS\DB\ MASTAPP\REPORTS	The path and directory where the report file resides.
<i>ReportName</i>	CUSTLIST.RPT	The report file name.

- 4 Place a button control anywhere in the form.
- 5 Double-click the button to open a code window, then write code to handle its *OnClick* event, as follows.

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Report1.Run;  
end;
```

- 6 Press *F9* to run the form.
- 7 Click the button to run the report. If the *TReport* component's *Preview* property is set to True, Delphi displays the report onscreen; if *Preview* is False, Delphi sends the report to the printer.

## How it works

The *Run* method of *TReport* opens the run-time version of ReportSmith, which prints or displays a report as specified by the *TReport* component's properties. When you're designing a form, you can double-click a *TReport* component to open the full version of ReportSmith and build a report.

## For more information

For more information about

- Using *TReport* components, see page 88.
- Using ReportSmith, see *Creating Reports*.

## What to do: printing forms

You can print a form by calling its *Print* method. Following is the code called by the Print button in the main form (MAIN.DFM) of MASTAPP. It prints whichever Delphi form is displayed “on top,” that is, in front of the others.

```
procedure TMainForm.PrintTopForm(Sender: TObject);
var TopForm: TForm;
begin
  TopForm := Screen.Forms[1]; {For a single form app, this should be Screen.Forms[0]!}
  if not TopForm.Visible then ShowMessage('Nothing to print!')
  else if MessageDlg(Format('Print "%s" form?', [TopForm.Caption]),
    mtConfirmation, mbOkCancel, 0) = mrOk then TopForm.Print;
end;
```

## How it works

The *Print* method for the *TForm* class prints a form as it appears onscreen.

## For more information

For more information about

- The *TForm* class, see the online Help.
- Designing forms, see the *User's Guide*.



# Using data access components and tools

This chapter describes how to use key Delphi features and tools when building database applications, including:

- The *TSession* component.
- Dataset components (*TTable* and *TQuery*), their properties, and their methods.
- *TDataSource* components, their properties, and their methods.
- *TField* objects, their properties, and their methods.
- The Fields Editor to instantiate and control *TField* objects.
- *TReport* and *TBatchMove* components.

This chapter provides an overview and general description of data access components in the context of application development. For in-depth reference information on database components, methods, and properties, see the online VCL reference.

## Database components hierarchy

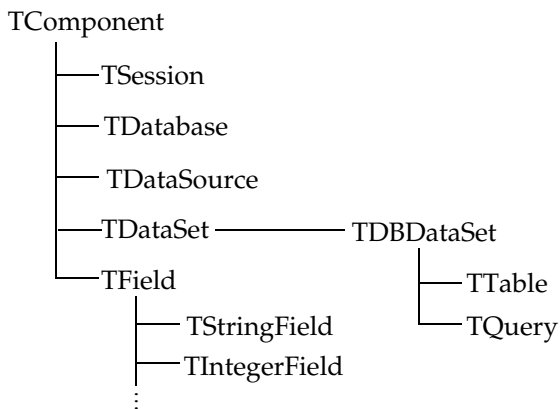
---

The Delphi database component hierarchy is important to show the properties, methods, and events inherited by components from their ancestors. The most important database components are

- *TSession*, a global component created automatically at run time. It is not visible on forms either at design time or run time.
- *TDatabase*, component that provides an additional level of control over server logins, transaction control, and other database features. It appears on the Data Access component page.
- *TDataSet* and its descendents, *TTable* and *TQuery*, collectively referred to as dataset components. *TTable* and *TQuery* components appear on the Data Access component page.

- *TDataSource*, a conduit between dataset components and data-aware components. It appears on the Data Access component page.
- *TFields*, components corresponding to database columns, created either dynamically by Delphi at run time or at design time with the Fields Editor. Data controls use them to access data from a database. In addition, you can define calculated fields whose values are calculated based on the values of one or more database columns.

**Figure 3.1** Delphi Data Access components hierarchy



This chapter describes most of these components and the tools that Delphi provides to work with them. The *TQuery* component is described in Chapter 5, “Using SQL in applications.” The *TDatabase* component is described in Chapter 6, “Building a client/server application.”

## Using the TSession component

---

The *TSession* component is rarely used, but can be useful for some specialized purposes. Delphi creates a *TSession* component named “Session” each time an application runs. You cannot see nor explicitly create a *TSession* component, but you can use its methods and properties to globally affect the application.

## Controlling database connections

---

*TSession* provides global control over database connections for an application. The *Databases* property of *TSession* is an array of all the active databases in the session. The *DatabaseCount* property is an integer specifying the number of active databases (*TDatabase* components) in the Session. For more information on *TDatabase*, see Chapter 6, “Building a client/server application.”

*KeepConnections* is a Boolean property that specifies whether to keep inactive database connections. A database connection becomes inactive when a *TDatabase* component has no active datasets. By default, *KeepConnections* is True, and an application will maintain its connection to a database even if the connection is inactive. This is generally



preferable if an application will be repeatedly opening and closing tables in the database. If *KeepConnections* is *False*, a database connection will be closed as soon as the connection is inactive. The *DropConnections* method will drop all inactive database connections.

The *NetFileDir* property specifies the directory path of the BDE network control directory. The *PrivateDir* property specifies the path of the directory in which to store temporary files (for example, files used to process local SQL statements). You should set this property if there will be only one instance of the application running at a time. Otherwise, the temporary files from multiple application instances will interfere with each other.

## Getting database information

---

*TSession* has a number of methods that enable an application to get database-related information. Each method takes a *TStrings* component as its parameter and returns into a *TStrings* the specified information:

**Table 3.1** TSession methods

Method	Returns
<i>GetAliasNames</i>	Defined BDE alias names.
<i>GetAliasParams</i>	Parameters for the specified BDE alias.
<i>GetDatabaseNames</i>	Database names and BDE aliases defined.
<i>GetDriverNames</i>	Names of BDE drivers installed.
<i>GetDriverParams</i>	Parameters for the specified BDE driver.
<i>GetTableNames</i>	All table names in the specified database.

---

For more information on these methods, see the online *VCL Reference*.

## Using datasets

---



*TTable* and *TQuery* component classes are descended from *TDataSet* through *TDBDataSet*. These component classes share a number of inherited properties, methods, and events. For this reason, it is convenient to refer to them together as datasets, when the discussion applies to both *TTable* and *TQuery*.

This section describes the features of datasets that are common to *TTable* and *TQuery*. A subsequent section discusses features unique to *TTable*. Chapter 5, “Using SQL in applications” describes features unique to *TQuery*.

**Note** *TStoredProc* is also a dataset component since it is descended from *TDBDataSet*. Therefore, much of this section also applies to *TStoredProc* if the stored procedure returns a result set rather than a singleton result. For more information on *TStoredProc*, see Chapter 6, “Building a client/server application.”

## Dataset states

---

A dataset can be in the following states, also referred to as *modes*:

**Table 3.2** Dataset states

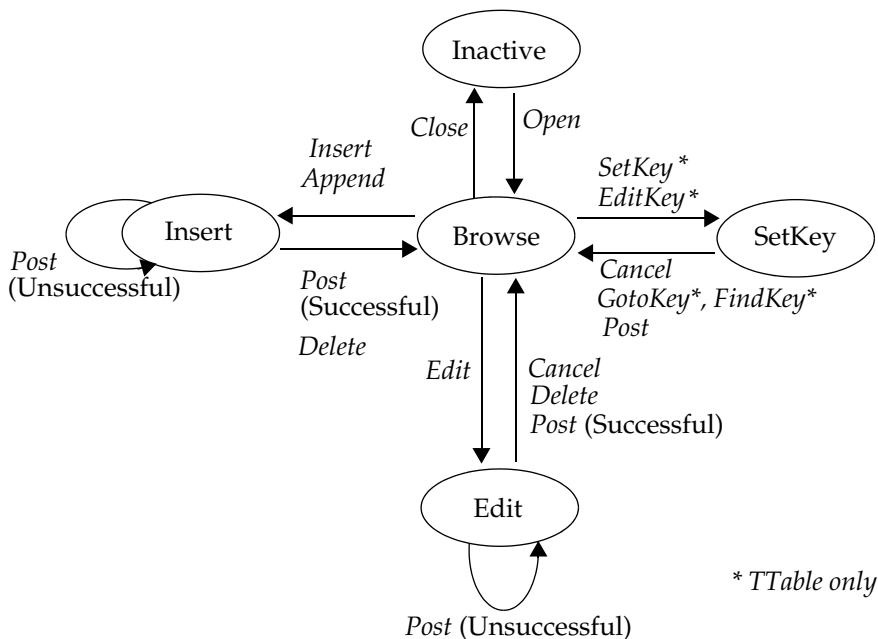
State	Description
Inactive	The dataset is closed.
Browse	The default state when a dataset is opened. Records can be viewed but not changed or inserted.
Edit	Enables the current row to be edited.
Insert	Enables a new row to be inserted. A call to <code>Post</code> inserts a new row.
SetKey	Enables <code>FindKey</code> , <code>GoToKey</code> , and <code>GoToNearest</code> to search for values in database tables. These methods only pertain to <code>TTable</code> components. For <code>TQuery</code> , searching is done with SQL syntax.
CalcFields	Mode when the <code>OnCalcFields</code> event is executed; prevents any changes to fields other than calculated fields. Rarely used explicitly.

An application can put a dataset into most states by calling the method corresponding to the state. For example, an application can put `Table1` in `Insert` state by calling `Table1.Insert` or `Edit` state by calling `Table1.Edit`. A number of methods return a dataset to `Browse` state, depending on the result of the method call. A call to `Cancel` will always return a dataset to `Browse` state.

`CalcFields` mode is a special case. An application cannot explicitly put a dataset into `CalcFields` mode. A dataset automatically goes into `CalcFields` mode when its `OnCalcFields` event is called. In `OnCalcFields`, an exception will occur if an application attempts to assign values to non-calculated fields. After the completion of `OnCalcFields`, the dataset returns to its previous mode.

The following diagram illustrates the primary dataset states and the methods that cause a dataset to change from one mode to another.

**Figure 3.2** Dataset state diagram



The *State* property specifies the current state of a dataset. The possible values correspond to the above states and are *dsInactive*, *dsBrowse*, *dsEdit*, *dsInsert*, *dsSetKey*, and *dsCalcFields*.

The *OnStateChange* event of *TDataSource* is called whenever the state of a data source's dataset changes. For more information, see "Using TDataSource events" on page 78.

## Opening and closing datasets

Before an application can access data through a dataset, the dataset must be open. There are two ways to open a dataset:

- Set the dataset's *Active* property to True, either at design time through the Object Inspector, or programmatically at run time. For example,

```
Table1.Active := True;
```

- Call the dataset's *Open* method at run time. For example,

```
Query1.Open
```

Both of these statements open the dataset and put it into Browse state.

Similarly, there are two ways to close a dataset:

- Set the dataset's *Active* property to False, either at design time through the Object Inspector, or programmatically at run time. For example,

```
Query1.Active := False;
```

- Call the dataset's *Close* method. For example,

```
Table1.Close;
```

Both of these statements return a dataset to Inactive state.

## Navigating datasets

---

There are two important concepts in understanding how Delphi handles datasets: *cursors* and *local buffers*. Each active dataset has a cursor, which is essentially a pointer to the *current row* in the dataset. A number of rows of data before and after the cursor are fetched by Delphi into the local buffer. Delphi will always fetch a number of rows into the local buffer sufficient to display the current row, plus an additional number of rows to reduce the refresh time as the user scrolls up or down in the dataset:

**Table 3.3** Navigational methods and properties

Method or property	Description
First method	Moves the cursor to the first row of a dataset.
Last method	Moves the cursor to the last row of the dataset.
Next method	Moves the cursor to the next row in the dataset.
Prior method	Moves the cursor to the prior row in the dataset.
BOF property	True when cursor is known to be at beginning of dataset, otherwise False.
EOF property	True when cursor is known to be at end of dataset, otherwise False.
MoveBy(n) method	Moves the cursor <i>n</i> rows forward in dataset, where <i>n</i> is a positive or negative integer.

Many of these methods are encapsulated in the *TDBNavigator* component. For more information on *TDBNavigator*, see Chapter 4, "Using Data Controls."

### The Next and Prior methods

The *Next* method moves the cursor down (forward) by one row in the table. For example, the following code for a button's *OnClick* event moves to the next row in Table1:

```
Table1.Next;
```

Similarly, the *Prior* method moves the cursor up (backward) by one row in the dataset. For example, to move to the previous row in the table, a button's *OnClick* text could be:

```
Table1.Prior
```

### The First and Last methods

As their names imply, the *First* and *Last* methods move to a dataset's first and last rows, respectively. For example, the following code for a button's *OnClick* event moves the cursor to the first row in Table1:

```
Table 1.First;
```

Similarly, the *Last* method moves to the last row in the dataset. To move to the last row in the table, a button's *OnClick* text could be:

```
Table1.Last
```

## The BOF and EOF properties

*BOF* is a read-only Boolean property that indicates whether a dataset is known to be at its first row. The *BOF* property returns a value of True only after:

- An application first opens a table
- A call to a Table's *First* method
- A call to a Table's *Prior* method fails

Databases are dynamic; while one application is viewing data, another may be inserting rows before or after the first application's notion of the current row. For this reason, for a non-empty table, it is unsafe to assume *BOF* is True.

For example, consider the following code:

```
Table1.Open; {BOF = True}  
Table1.Next; {BOF = False}  
Table1.Prior; {BOF = False}
```

After this code executes, *BOF* is False, even if there are no records before the current row. Once the table is open, Delphi can only determine *BOF* when an application explicitly calls *First* or a call to *Prior* fails. Similarly, Delphi can only determine *EOF* when an application explicitly calls *Last* or a call to *Next* fails.

The following code sample demonstrates a common technique for using the *BOF* property:

```
while not Table1.BOF do  
begin  
    DoSomething;  
    Table1.Prior;  
end;
```

In this code sample, the hypothetical function *DoSomething* is called on the current record and then on all the records between the current record and the beginning of the dataset. The loop will continue until a call to *Prior* fails to move the current record back. At that point, *BOF* will return a value of True and the program will break out of the loop.

To improve performance during the iteration through the table, call the *DisableControls* method before beginning the loop. This prevents data controls from displaying the iteration through the table, and speeds up the loop. After the loop completes, call the *EnableControls* method. Make sure to use a **try...finally...end** statement with the call to *EnableControls* in the **finally** clause. Otherwise, an exception will leave the application's controls inactive.

The same principles apply to the *EOF* property, which returns a value of True after:

- An application opens an empty dataset
- A call to a Table's *Last* method
- A call to a Table's *Next* fails

The following code sample provides a simple means of iterating over all the records in a dataset:

```
Table1.DisableControls;
try
    Table1.First;
while not Table1.EOF do
begin
    DoSomething;
    Table1.Next;
end;
finally
    Table1.EnableControls;
end.
```

In this case, the *Next* method and the *EOF* property are used together to reach the end of the dataset.

**Caution** A common error in using such properties in navigating a dataset is to use a **repeat. . . until** loop while forgetting to call *Table1.Next*, as in the following example:

```
Table1.First;
repeat
    DoSomething;
until Table1.EOF;
```

If code like this were executed, the application would appear to “freeze,” since the same action would be endlessly performed on the first record of the dataset, and the *EOF* property would never return a value of True.

On an empty table, opening or executing any navigational methods will return True for both *BOF* and *EOF*.

## The MoveBy function

The *MoveBy* function enables an application to move through a dataset backward or forward by a specified number of records. This function takes only one parameter, the number of records by which to move. Positive integers indicate a forward move, while negative integers indicate a backward move.

For example, to move two records forward in *Table1*, use the following:

```
Table1.MoveBy(2);
```

When using this function, keep in mind that datasets are fluid entities, and the record which was five records back a moment ago may now be only four records back, or six records, or an unknown number of records, because multiple users may be simultaneously accessing the database and modifying its data.

**Note** There is no functional difference between calling *Table1.Next* and calling *Table1.MoveBy(1)*, just as there is no functional difference between calling *Table1.Prior* or calling *Table1.MoveBy(-1)*.

## Modifying data in datasets

---

The following methods enable an application to insert, update, and delete data in datasets:

**Table 3.4** Methods to insert, update and delete data in datasets

Method	Description
Edit	Puts the dataset into Edit state. If a dataset is already in Edit or Insert state, a call to <i>Edit</i> has no effect.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in Insert state.
Insert	Posts any pending data, and puts the dataset in Insert state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in Browse state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset into Browse state.
Delete	Deletes the current record and puts the dataset in Browse state.

---

### The CanModify property

*CanModify* is a read-only property that specifies whether an application can modify the data in a dataset. When *CanModify* is False, then the dataset is read-only, and cannot be put into Edit or Insert state. When *CanModify* is True, the dataset can enter Edit or Insert state. Even if *CanModify* is True, it is not a guarantee that a user will be able to insert or update records in a table. Other factors may come in to play, for example, SQL access privileges.

*TTable* has a *ReadOnly* property that requests write privileges when set to False. When *ReadOnly* is True, *CanModify* will automatically be set to False. When *ReadOnly* is False, *CanModify* will be True if the database allows read and write privileges for the dataset and the underlying table. For more information, see “Using *TTable*.”

### Posting data to the database

The *Post* method is central to a Delphi application’s interaction with a database table. *Post* behaves differently depending on a dataset’s state.

- In Edit state, *Post* modifies the current record.
- In Insert state, *Post* inserts or appends a new record.
- In SetKey state, *Post* returns the dataset to Browse state.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, Delphi calls *Post* implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in Edit or Insert state. The *Append* and *Insert* methods also implicitly perform a *Post* of any pending data.

**Note** *Post* is not called implicitly by the *Close* method. Use the *BeforeClose* event to post any pending edits explicitly.

### Editing records

A dataset must be in Edit state before an application can modify records in the underlying table. The *Edit* method puts a dataset in Edit state. When in Edit state, the

*Post* method will change the current record. If a dataset is already in Edit state, a call to *Edit* has no effect.

The *Edit* and *Post* methods are often used together. For example,

```
Table1.Edit;  
Table1.FieldByName('CustNo').AsString := '1234';  
Table1.Post;
```

The first line of code in this example places the dataset in Edit mode. The next line of code assigns the string “1234” to the CustNo field. Finally, the last line posts, or writes to the database, the data just modified.

## Adding new records

To add a new record to a dataset, an application can call either the *Insert* method or the *Append* method. Both methods put a dataset into Insert state. *Insert* opens a new, empty record after the current record. *Append* moves the current record to the end of the dataset and opens a new, empty record.

When an application calls *Post*, the new record will be inserted in the dataset in a position based on its index, if defined. Thus, for indexed tables, *Append* and *Insert* perform similarly. If no index is defined on the underlying table, then the record will maintain its position—so *Append* will add the record to the end of the table, and *Insert* will insert it at the cursor position when the method was called. In either case, posting a new record in a data grid may cause all the rows before and after the new record to change as the dataset follows the new row to its indexed position and then fetches data to fill the grid around it.

## Deleting records

The *Delete* method deletes the current record from a dataset and leaves the dataset in Browse mode. The cursor moves to the following record.

## Canceling changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a Table is in Edit state, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the Table’s *Cancel* method. A call to *Cancel* always returns a dataset to Browse state.

## Working with entire records

The following methods enable an application to work with an entire record in one statement:

**Table 3.5** Methods used to work with entire records

Method	Description
<i>AppendRecord</i> ([array of values])	Appends a record with the specified column values at the end of a table; analogous to <i>Append</i> . Performs an implicit <i>Post</i> .



**Table 3.5** Methods used to work with entire records (continued)

Method	Description
<i>InsertRecord</i> ([array of values])	Inserts the specified values as a record after the current cursor position of a table; analogous to Insert. Performs an implicit Post.
<i>SetFields</i> ([array of values])	Sets the values of the corresponding fields; analogous to assigning values to TFields. Application must perform a Post.

Each method takes a comma-delimited array of values as its argument, where each value corresponds to a column in the underlying table. The values can be literals, variables, null, or **nil**. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be null.

For un-indexed tables, *AppendRecord* adds a record to the end of the table and *InsertRecord* inserts a record after the current cursor position. For indexed tables, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

*SetFields* assigns the values specified in the array of parameters to fields in the dataset. The application must first perform an *Edit* to put the dataset in Edit state. To modify the current record, it must then perform a *Post*.

Since these methods depend explicitly on the structure of the underlying tables, an application should use them only if the table structure will not change.

For example, the COUNTRY table has columns for Name, Capital, Continent, Area, and Population. If Table1 were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
Table1.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

The statement does not specify values for Area and Population, so it will insert Null values for these columns. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
Table1.Edit;
Table1.SetRecord(nil, nil, nil, 344567, 164700000);
Table1.Post;
```

This code assumes that the cursor will be positioned on the record just entered for Japan. It assigns values to the Area and Population fields and then posts them to the database. Notice the three **nil**s that act as place holders for the first three columns, which are not changed.

## Setting the update mode

The *UpdateMode* property of a dataset determines how Delphi will find records being updated in a SQL database. This property is important in a multi-user environment when users may retrieve the same records and make conflicting changes to them.

When a user posts an update, Delphi uses the original values in the record to find the record in the database. This approach is similar to an optimistic locking scheme. *UpdateMode* specifies which columns Delphi uses to find the record. In SQL terms,

*UpdateMode* specifies which columns are included in the WHERE clause of an UPDATE statement. If Delphi cannot find a record with the original values in the columns specified (if another user has changed the values in the database), Delphi will not make the update and will generate an exception.

The *UpdateMode* property may have the following values:

- *WhereAll* (the default): Delphi uses every column to find the record being updated. This is the most restrictive mode.
- *WhereKeyOnly*: Delphi uses only the key columns to find the record being updated. This is the least restrictive mode and should be used only if other users will not be changing the records being updated.
- *WhereChanged*: Delphi uses key columns and columns that have changed to find the record being updated.

For example, consider a COUNTRY table with columns for NAME (the key), CAPITAL, and CONTINENT. Suppose you and another user simultaneously retrieve a record with the following values:

- NAME = "Philippines"
- CAPITAL = "Nairobi"
- CONTINENT = "Africa"

Both you and the other user notice that the information in this record is incorrect and should be changed. Now, suppose the other user changes CONTINENT to "Asia," CAPITAL to "Manila," and posts the change to the database. A few seconds later, you change NAME to "Kenya" and post your change to the database.

If your application has *UpdateMode* set to *WhereKey* on the dataset, Delphi compares the original value of the key column (NAME = "Philippines") to the current value in the database. Since the other user did not change NAME, your update occurs. You think the record is now ["Kenya," "Nairobi," "Africa"] and the other user thinks it is ["Philippines," "Asia," "Manila"]. Unfortunately, it is actually ["Kenya," "Asia," "Manila"], which is still incorrect, even though both you and the other user think you have corrected the mistake. This problem occurred because you had *UpdateMode* set to its least restrictive level, which does not protect against such occurrences.

If your application had *UpdateMode* set to *WhereAll*, the Delphi would check all the columns when you attempt to make your update. Since the other user changed CAPITAL and CONTINENT, Delphi would not let you make the update. When you retrieved the record again, you would see the new values entered by the other user and realize that the mistake had already been corrected.

## Bookmarking data

---

It is often useful to mark a particular location in a table so that you can quickly return to it when desired. Delphi provides this functionality through *bookmark* methods. These methods enable you to put a bookmark in the dataset, and quickly return to it later.

The three bookmarking methods are

- *GetBookmark*
- *GoToBookmark*
- *FreeBookmark*

These are used together. The *GetBookmark* function returns a variable of type *TBookmark*. A *TBookmark* contains a pointer to a particular location in a dataset. When given a bookmark, the *GoToBookmark* method will move an application's cursor to that location in the dataset.

*FreeBookmark* frees memory allocated for the specified bookmark. A call to *GetBookmark* allocates memory for the bookmark, so an application should call *FreeBookmark* before exiting, and before every use of a bookmark.

The following code illustrates a typical use of bookmarking:

```
procedure DoSomething;
var Bookmark: TBookmark;
begin
    Bookmark := Table1.GetBookmark; {allocate}
    Table1.DisableControls; {Disengage data controls}
    try
        Table1.First;
        while not Table1.EOF do
            begin
                {Do Something}
                Table1.Next;
            end;
        finally
            Table1.GotoBookmark(Bookmark);
            Table1.EnableControls;
            Table1.FreeBookmark(Bookmark); {deallocate}
        end;
    end;
```

Notice the careful positioning of statements in this code. If the call to *GetBookmark* fails, controls are not disabled. If it succeeds, the bookmark is always freed and controls are always enabled.

## Disabling, enabling, and refreshing data-aware controls

---

The *DisableControls* method disables all data-aware controls linked to a dataset. This method should be used with caution (for example, when programmatically iterating or searching through a dataset) to prevent “flickering” of the display as the cursor moves. As soon as the cursor is repositioned, an application should call the *EnableControls* method to re-enable data controls. It is important to re-enable controls with *EnableControls* as soon as the application completes its iteration or searching, to keep the form synchronized with the underlying dataset. Use a **try...finally** statement as in the example above.

The *Refresh* method flushes all local buffers and retrieves data from the specified dataset again. The dataset must be open. You can use this function to update a table if you think

the table or the data it contains might have changed. Refreshing a table can sometimes lead to unexpected results. For example, if a user is viewing a record that has been deleted, then it will seem to disappear the moment the application calls *Refresh*. Similarly, data can appear to change while a user is viewing it if another user changes or deletes a record after the data was originally fetched and before a call to *Refresh*.

## Using dataset events

---

Datasets have a number of events that enable an application to perform validation, compute totals, and perform other tasks depending on the method performed by the dataset. The events are listed in the following table.

**Table 3.6** Dataset events

Event	Description
<i>BeforeOpen, AfterOpen</i>	Called before/after a dataset is opened.
<i>BeforeClose, AfterClose</i>	Called before/after a dataset is closed.
<i>BeforeInsert, AfterInsert</i>	Called before/after a dataset enters Insert state.
<i>BeforeEdit, AfterEdit</i>	Called before/after a dataset enters Edit state.
<i>BeforePost, AfterPost</i>	Called before/after changes to a table are posted.
<i>BeforeCancel, AfterCancel</i>	Called before/after the previous state is canceled.
<i>BeforeDelete, AfterDelete</i>	Called before/after a record is deleted.
<i>OnNewRecord</i>	Called when a new record is created; used to set default values.
<i>OnCalcFields</i>	Called when calculated fields are calculated.

---

For more information on these events and methods of the *TDataSet* component, refer to the online VCL reference.

### Abort a method

To abort a method such as an *Open* or *Insert*, raise an exception or call the *Abort* procedure in any of the *Before* methods (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code confirms a delete operation:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataSet);
begin
    if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) = mrYes then
        Abort;
end;
```

### Using OnCalcFields

The *OnCalcFields* event is used to set the values of calculated fields. The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, then *OnCalcFields* is called when:

- The dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a *DBDataGrid*.

- A record is retrieved from the database.

*OnCalcFields* is also called whenever a non-calculated field's value changes, regardless of the setting of *AutoCalcFields*.

Typically, the *OnCalcFields* event will be called often, so it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* will be called again, leading to another *Post*, and so on.

If *AutoCalcFields* is *False*, then *OnCalcFields* is called when the dataset's *Post* method is called (or any method that implicitly calls *Post*, such as *Append* or *Insert*).

While the *OnCalcFields* event is executed, a dataset will be put in *CalcFields* mode. When a dataset is in *CalcFields* mode, you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset will return to its previous mode.

## Using TTable

---



*TTable* is one of the most important database component classes. Along with the other dataset component class, *TQuery*, it enables an application to access a database table. This section describes the most important properties that are unique to *TTable*.

### Specifying the database table

---

*TableName* specifies the name of the database table to which the *TTable* component is linked. You can set this property at design time through the Object Inspector.

The *DatabaseName* property specifies where Delphi will look for the specified database table. It can be a BDE alias, an explicit specification, or the *DatabaseName* defined by any *TDatabase* component in the application. For Paradox and dBASE tables, an explicit specification is a directory path; for SQL tables, it is a directory path and database name.

Instead of an actual directory path or database name, *DatabaseName* can also be a BDE alias. The advantage of this is that you can change the data source for an entire application by simply changing the alias definition in the BDE Configuration Utility. For more information on using the BDE Configuration Utility, see Appendix B, "Using the BDE configuration utility." For more information on the *DatabaseName* property, see the online VCL reference.

**Note** Neither of these properties can be changed when a table is open—that is, when the table's *Active* property is set to a value of *True*.

### The TableType property

---

The *TableType* property specifies the type of the underlying database table. This property is not used for SQL tables.

If *TableType* is set to *Default*, the table's file-name extension determines the table type:

- Extension of .DB or no file-name extension: Paradox table
- Extension of .DBF : dBASE table
- Extension of .TXT : ASCII table

If the value of *TableType* is not *Default*, then the table will always be of the specified *TableType*, regardless of file-name extension.

## Searching a table

---

*TTable* has a number of functions that will search for values in a database table:

- Goto functions
- Find functions

The easiest way to search for values is with the Find functions, *FindKey* and *FindNearest*. These two functions combine the functionality of the basic Goto functions, *SetKey*, *GoToKey*, and *GoToNearest*, which are described first.

In dBASE and Paradox tables, these functions can search only on index fields. In SQL tables, they can search on any fields, if the field name is specified in the *IndexFieldNames* property of the *TTable*. For more information, see “Indexes” on page 74.

To search a dBASE or Paradox table for a value in a non-index field, use SQL SELECT syntax with a *TQuery* component. For more information on using SQL and *TQuery* components, see Chapter 5, “Using SQL in applications.”

## Using Goto functions

The *GoToKey* and *GoToNearest* methods enable an application to search a database table using a key. *SetKey* puts a table in “search mode,” more accurately referred to as SetKey state. In SetKey state, assignments indicate values for which to search for in indexed fields. *GoToKey* then moves the cursor to the first row in the table that matches those field values.

For example, the following code could be used in the *OnClick* event of a button:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  Table1.SetKey; {First field is the key}
  Table1.Fields[0].AsString := Edit1.Text;
  Table1.GoToKey;
end;
```

The first line of code after **begin** puts *Table1* in SetKey state. This indicates that the following assignment to the table’s *Fields* property specifies a search value. The first column in the table, corresponding to *Fields[0]*, is the index. In this example, the value the application searches for is determined by the text the user types into the edit control, *Edit1*. Finally, *GoToKey* performs the search, moving the cursor to the record if it exists.

*GoToKey* is a Boolean function that moves the cursor and returns True if the search is successful. If the search is unsuccessful, it returns False and does not change the position of the cursor. For example,

```

Table1.SetKey;
Table1.Fields[0].AsString := 'Smith';
if not Table1.GotoKey
    then ShowMessage('Record Not Found');

```

If the search does not find a record with a first column matching “Smith,” the *ShowMessage* function displays a dialog box with the “Record Not Found” message.

If a table has more than one key column, and you want to search for values in a sub-set of the keys, set *KeyFieldCount* to the number of columns on which you are searching. For example, if a table has three columns in its primary key, and you want to search for values in just the first, set *KeyFieldCount* to 1. For tables with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. That is, you can search for values in the first column, or the first and second, or the first, second, and third, but not just the first and third.

*GoToNearest* is similar, except it finds the nearest match to a partial field value. It can be used only for columns of string data type. For example,

```

Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GoToNearest;

```

If a record exists with “Sm” as the first two characters, the cursor will be positioned on that record. Otherwise, the position of the cursor does not change and *GoToNearest* returns False.

If it is not searching on the primary index of a local table, then an application must specify the column names to use in the *IndexFieldNames* property or the name of the index to use in the *IndexName* property of the table. For example, if the CUSTOMER table had a secondary index named “CityIndex” on which you wanted to search for a value, you would need to set the value of the table’s *IndexName* property to “CityIndex.” You could then use the following syntax when you searching on this field:

```

Table1.IndexName := 'CityIndex';
Table1.Open;
Table1.SetKey;
Table1.FieldByName('City').AsString := Edit1.Text;
Table1.GoToNearest;

```

Because indexes often have non-intuitive names, you can use the *IndexFieldNames* property instead to specify the names of indexed fields.

Each time an application calls *SetKey*, it must set all the field values for which it will search. That is, *SetKey* clears any existing values from previous searches. To keep previous values, use *EditKey*.

For example, to extend the above search to find a record with the specified city name in a specified country, an application could use the following code:

```

Table1.EditKey;
Table1.FieldByName('Country').AsString := Edit2.Text;

```

## Using Find functions

The Find functions, *FindKey* and *FindNearest* provide easy way to search a table. They combine the functionality of *SetKey*, field assignment, and Goto functions into a single statement.

Each of these methods takes a comma-delimited array of values as its argument, where each value corresponds to a column in the underlying table. The values can be literals, variables, null, or **nil**. If the number of values in an argument is less than the number of columns in the database table, then the remaining values are assumed to be null. The functions will search for values specified in the array in the current index.

*FindKey* is similar to *GotoKey*:

- It will put a table in search mode (SetKey state).
- It will find the record in the table that matches the specified values. If a matching record is found, it moves the cursor there, and returns True.
- If a matching record is not found, it does not move the cursor, and returns False.

For example, if Table1 is indexed on its first column, then the statement:

```
Table1.FindKey([Edit1.Text]);
```

will perform the same function as the three statements:

```
Table1.SetKey; {First field is the key}  
Table1.Fields[0].AsString := Edit1.Text;  
Table1.GoToKey;
```

*FindNearest* is similar to *GotoNearest*, in that it will move the cursor to the row with the nearest matching value. This can be used for columns of string data type only.

Both of these functions work by default on the primary index column. To search the table for values in other indexes, you must specify the field name in the table's *IndexFieldNames* property or the name of the index in the *IndexName* property.

**Note** With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

## The KeyExclusive property

The *KeyExclusive* property indicates whether a search will position the cursor on or after the specified record being searched for. If *KeyExclusive* is False (the default), then *GoToNearest* and *FindNearest* will move the cursor to the record that matches the specified values. If True, then the search functions will go the record immediately following the specified key, if found.

## Limiting records retrieved by an application

---

Tables in the real world can be huge, so applications often need to limit the number of rows they work with. The following methods of *TTable* enable an application to work with a subset of the data in a database table:



- *SetRangeStart* and *EditRangeStart*
- *SetRangeEnd* and *EditRangeEnd*
- *SetRange*([Start Values], [End Values])
- *ApplyRange*
- *CancelRange*

*SetRangeStart* indicates that subsequent assignments to field values will specify the start of the range of rows to include in the application. *SetRangeEnd* indicates that subsequent assignments will specify the end of the range of rows to include. Any column values not specified are not considered. The corresponding methods *EditRangeStart* and *EditRangeEnd* indicate to keep existing range values and update with the succeeding assignments.

*ApplyRange* applies the specified range. If *SetRangeStart* has not been called when *ApplyRange* is called, then the start range will be the beginning of the table; likewise, if *SetRangeEnd* has not been called, the end range will be the end of the table. *CancelRange* cancels the range filter and includes all rows in the table.

The *SetRange* function combines *SetRangeStart*, *SetRangeEnd*, and field assignments into a single statement that takes an array of values as its argument.

**Note** With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

For example, suppose there is a form with a *TTable* component named *Cust*, linked to the CUSTOMER table. CUSTOMER is indexed on its first column (*CustNo*). The form also has two Edit components named *StartVal* and *EndVal*, and you have used the Fields Editor to create a *TField* component for the *CustNo* column. Then these methods could be applied (for example, in a button's *OnClick* event) as follows:

```
Cust.SetRangeStart;
CustCustNo.AsString := StartVal.Text;
Cust.SetRangeEnd;
if EndVal.Text <> '' then
    CustCustNo.AsString := EndVal.Text;
Cust.ApplyRange;
```

Notice that this code first checks that the text entered in *EndVal* is not null before assigning any values to Fields. If the text entered for *StartVal* is null, then all records from the beginning of the table will be included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records will be included, since none are less than null.

This code could be re-written using the *SetRange* function as follows:

```
if EndVal.Text <> '' then
    Cust.SetRange([StartVal.Text], [EndVal.Text]);
Cust.ApplyRange;
```

## Using partial keys

If a key is composed of one or more string fields, these methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
Table1.SetRangeStart;  
Table1.FieldName('LastName').AsString := 'Smith';  
Table1.SetRangeEnd;  
Table1.ApplyRange;
```

This will include all records where LastName greater than or equal to “Smith.” The value specification could also be:

```
Table1.FieldName('LastName').AsString := 'Sm';
```

This would include records which have LastName greater than or equal to “Sm.” The following would include records with a LastName greater than or equal to “Smith” and a FirstName greater than or equal to “J”:

```
Table1.FieldName('LastName').AsString := 'Smith';  
Table1.FieldName('FirstName').AsString := 'J';
```

## The KeyExclusive property

The *KeyExclusive* property determines whether the filtered range excludes the range boundaries. The default is *False*, which means rows will be in the filtered range if they are greater than or equal to the start range specified and less than or equal to the end range specified. If *KeyExclusive* is *True*, the methods will filter strictly greater than and less than the specified values.

## Indexes

---

An *index* determines how records are sorted when a Delphi application displays data. By default, Delphi displays data in ascending order, based on the values of the primary index column(s) of a table.

Delphi supports SQL indexes, maintained indexes for Paradox tables, and maintained .MDX (production) indexes for dBASE tables. Delphi does not support:

- Non-maintained indexes on Paradox tables.
- Non-maintained or .NDX indexes of dBASE tables.
- The *IndexFieldCount* property for a dBASE table opened on an expression index.

The *GetIndexNames* method returns a list of the names of available indexes on the underlying database table. For Paradox tables, the primary index is unnamed and therefore not returned by *GetIndexNames*. To use a primary index on a Paradox table, set the corresponding *TTable's IndexName* to a null string.

*IndexFields* is an array of field names used in the index. *IndexFieldCount* is the number of fields in the index. *IndexFieldCount* and *IndexFields* are read-only properties that are available only during run-time.

Use the *IndexName* property to sort or search a table on an index other than the primary index. In other words, to use the primary index of a table, you need do nothing with the *IndexName* property. To use a secondary index, however, you must specify it in *IndexName*.

For tables in a SQL database, the *IndexFieldNames* property specifies the columns to use in the ORDER BY clause when retrieving data. The entry for this property is a

semicolon-delimited list of field names. Records are sorted by the values in the specified fields. Sorting can be only in ascending order. Case-sensitivity depends on the server being used.

For example, to sort customer records in an SQL table by zip code and then by customer number, enter the following for the *IndexFieldNames* property:

```
ZipCode;CustNo
```

For Paradox and dBASE tables, Delphi will pick an index based on the columns specified in *IndexFieldNames*. An error will occur if you specify a column or columns that cannot be mapped to an existing index.

The *IndexName* and *IndexFieldNames* properties are mutually exclusive. Setting one property clears the value of the other.

## The Exclusive property

---

The *Exclusive* property indicates whether to open the table with an exclusive lock. If True, no other user will be able to access it at the same time. You cannot open a table in Exclusive mode if another user is currently accessing the table.

If the underlying table is in a SQL database, an exclusive table-level lock may allow others to read data from the table but not modify it. Some servers may not support exclusive table-level locks, depending on the server. Refer to your server documentation for more information.

## Other properties and methods

---

In addition to dataset properties shared with *TQuery*, *TTable* has a number of unique methods and properties. For example, the unique methods include

- *EmptyTable*, which deletes all records (rows) in the table.
- *DeleteTable*, which deletes the table.
- *BatchMove*, which copies data and table structures from one table to another, similar to the operation of *TBatchMove*.

A few of the more important properties and methods are discussed in this section. For a complete list and descriptions, see the online *VCL Reference*.

## The ReadOnly and CanModify properties

Before opening a *TTable*, set *ReadOnly* False to request read and write privileges for the dataset. Set *ReadOnly* to True to request read-only privileges for the dataset. Depending on the characteristics of the underlying table, the request for read and write privileges may or may not be granted by the database.

*CanModify* is a read-only property of datasets that reflects the actual rights granted for the dataset. When *ReadOnly* is True, *CanModify* will automatically be set to False. When *ReadOnly* is False, *CanModify* will be True if the database allows read and write privileges for the dataset and the underlying table.

When *CanModify* is False, then the table is read-only, and the dataset cannot be put into Edit or Insert state. When *CanModify* is True, the dataset can enter Edit or Insert state. Even if *CanModify* is True, it is not a guarantee that a user will be able to insert or update records in a table. Other factors may come in to play, for example, SQL access privileges.

## The GoToCurrent method

*GoToCurrent* is a method that synchronizes two *TTable* components linked to the same database table and using the same index. This method takes a *TTable* component as its argument, and sets the cursor position of the *TTable* to the current cursor position of the argument. For example,

```
Table1.GotoCurrent (Table2);
```

## Creating master-detail forms

The *MasterSource* and *MasterFields* are used to define one-to-many relationships between two tables. The *MasterSource* property is used to specify a data source from which the table will get data for the master table. For instance, if you link two tables in a master-detail relationship, then the detail table can track the events occurring in the master table by specifying the master table's *TDataSource* in this property.

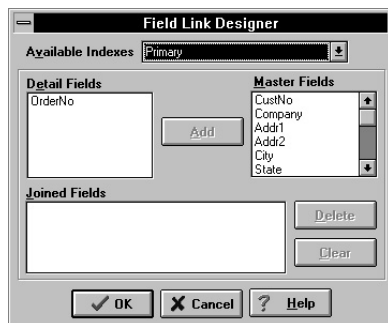
To link tables based on values in multiple column names, use a semicolon delimited list:

```
Table1.MasterFields := 'OrderNo;ItemNo';
```

## The Field Link Designer

At design time, when you double-click (or click on the ellipsis button) on the *MasterFields* property in the Object Inspector, the Field Link Designer dialog box opens.

Figure 3.3 Field Link designer



The Field Link Designer provides a visual way to link master and detail tables. The Available Indexes combo box shows the currently selected index by which to join the two tables. For Paradox tables, this will be “Primary” by default, indicating that the primary index of the detail field will be used. Any other named indices defined on the table will be shown in the drop-down list.

Select the field you want to use to link the detail table in the Detail Fields list, the field to link the master table in the Master Fields list, and then choose Add. The selected fields will be displayed in the Joined Fields list box. For example,

```
OrderNo -> OrderNo
```

For tables on a database server, the Available Indexes combo box will not appear, and you must select the detail and master fields to join manually in the Detail Fields and Master Fields list boxes.

## Using TDataSource

---



*TDataSource* acts as a conduit between datasets and data-aware controls. Often the only thing you will do with a *TDataSource* component is to set its *DataSet* property to an appropriate dataset object. Then you will set data controls' *DataSource* property to the specific *TDataSource*. You also use *TDataSource* components to link datasets to reflect master-detail relationships.

### Using TDataSource properties

---

*TDataSource* has only a few published properties in addition to the standard *Name* and *Tag* properties.

#### The DataSet property

The *DataSet* property specifies the name of the dataset from which the *TDataSource* will get its data. You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on the two forms. For example,

```
procedure TForm2.FormCreate (Sender : TObject);
begin
    DataSource1.DataSet := Form1.Table1;
end;
```

#### The Enabled property

The *Enabled* property can temporarily disconnect a *TDataSource* from its *TDataSet*. When set to *False*, all data controls attached to the data source will go blank and become inactive until *Enabled* is set to *True*.

In general, it is recommended to use datasets' *DisableControls* and *EnableControls* methods to perform this function, because they affect all attached data sources.

#### The AutoEdit property

The *AutoEdit* property of *TDataSource* specifies whether datasets connected to the data source automatically enter Edit state when the user starts typing in data-aware controls linked to the dataset. If *AutoEdit* is *True* (the default), Delphi automatically puts the dataset in Edit state when a user types in a linked data-aware control. Otherwise, a dataset enters Edit state only when the application explicitly calls its *Edit* method. For more information on dataset states, see "Dataset states" on page 58.

## Using TDataSource events

---

*TDataSource* has three events associated with it:

- *OnDataChange*
- *OnStateChange*
- *OnUpdateData*

### The OnDataChange event

*OnDataChange* is called whenever the cursor moves to a new record. In other words, if an application calls *Next*, *Previous*, *Insert*, or any method that leads to a change in the cursor position, then an *OnDataChange* is triggered.

This event is useful if an application is keeping components synchronized manually.

### The OnUpdateData event

*OnUpdateData* is called whenever the data in the current record is about to be updated. For instance, an *OnUpdateData* event will occur after *Post* is called but before the data is actually posted to the database.

This event is useful if an application uses a standard (non-data aware) control and needs to keep it synchronized with a dataset.

### The OnStateChange event

*OnStateChange* is called whenever the mode (state) of a data source's dataset changes. A dataset's *State* property records its current state. This event is useful for performing actions as a *TDataSource*'s state changes, as the following examples illustrate.

During the course of a normal database session, a dataset's state will change frequently. To track these changes, you can use code in *OnStateChange* such as the following example that displays the current state in a Label component:

```
procedure TForm1.DataSource1.StateChange(Sender:TObject);
var
  S:String;
begin
  case Table1.State of
    dsInactive: S := 'Inactive';
    dsBrowse: S := 'Browse';
    dsEdit: S := 'Edit';
    dsInsert: S := 'Insert';
    dsSetKey: S := 'SetKey';
  end;
  Label1.Caption := S;
end;
```

Similarly, *OnStateChange* can be used to enable or disable buttons or menu items based on the current state. For example,

```
procedure TForm1.DataSource1.StateChange(Sender: TObject);
begin
  InsertBtn.Enabled := (Table1.State = dsBrowse);
```

```
CancelBtn.Enabled := Table1.State in [dsInsert, dsEdit, dsSetKey];
...
end;
```

## Using TFields and the Fields Editor

---

*TField* components correspond to database columns. They are created

- At run time by Delphi whenever a dataset component is active. This creates a dynamic set of *TFields* that mirrors the columns in the table at that time.
- At design time through the Fields Editor. This creates a persistent set of *TFields* that does not change, even if the structure of the underlying table changes.

There are *TField* components corresponding to all possible data types, including *TStringField*, *TSmallintField*, *TIntegerField*, *TWordField*, *TBooleanField*, *TFloatField*, *TCurrencyField*, *TBCDField*, *TDateField*, *TTimeField*, and *TDateTimeField*. This chapter discusses *TFields* in general, and the discussion applies to all the different sub-types. For information on the properties of a specific type, see the online *VCL Reference*.

### What are TField components?

---

All Delphi data-aware components rely on an underlying object class, *TField*. Although not visible on forms, *TField* components are important because they provide an application a direct link to a database column. *TFields* contain properties specifying a column's data type, current value, display format, edit format, and other characteristics. *TField* components also provide events, such as *OnValidate*, that can be used to implement field-based validation rules.

Each column retrieved from a table has a corresponding *TField* component. By default, *TField* components are dynamically generated at design time when the *Active* property of a *TTable* or *TQuery* component is set to True. At run time, these components are also dynamically generated. Dynamic generation means Delphi builds *TField* components based on the underlying physical structure of a database table each time the connection to the table is activated. Thus, dynamically generated *TFields* always correspond to the columns in the underlying database tables.

To generate a persistent list of *TField* components for an application, use the Fields Editor. Using the Fields Editor to specify a persistent list of *TField* components is smart programming. Creating *TField* components with the Fields Editor provides efficient, readable, and type-safe programmatic access to underlying data. It guarantees that each time your application runs, it uses and displays the same columns, in the same order, every time, even if the physical structure of the underlying database has changed. Creating *TField* components at design time guarantees that data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent *TField* component is based is deleted or changed, then Delphi generates an exception rather than running the application against a non-existent column or mismatched data.

## Using the Fields Editor

---

When *TTable* and *TQuery* components are connected to a database and their *Active* properties are set to True, they dynamically generate a *TField* component for each column in a table or query. Each *TField* component stores display-related information about a field. The display information is used by data control components, such as *TDBEdit* and *TDBGrid*, to format data for display in a form. You can make *TField* components persistent and edit their display characteristics by invoking the Fields Editor.

The Fields Editor enables you to:

- Generate a persistent list of *TField* components.
- Modify the display properties of persistent *TField* components.
- Remove *TField* components from the list of persistent components.
- Add new *TField* components based on existing columns in a table.
- Define calculated *TField* components that behave just like physical data columns, except that their values are computed programmatically.

### Starting the Fields Editor

To invoke the Fields Editor for a *TTable* or *TQuery* component,

- Double-click the mouse on the component, or
- Select the component, right click the mouse, and select Fields Editor from the pop-up menu.

The Fields Editor opens, with the name of the Form and Table on which it was invoked in the title bar:

**Figure 3.4** Fields Editor



The Fields list box displays the names of persistent *TField* components associated with the data access component. The first time you invoke the Fields Editor on a particular a *TTable* or *TQuery* component, the Fields list is empty because all *TFields* are dynamically created. If any *TField* objects are listed in *Fields*, then data-aware components can only display data from those fields. You can drag and drop individual *TField* objects within the *Field* list box to change the order in which fields are displayed in controls, like *TDBGrid*, that display multiple columns.



The navigator buttons at the bottom of the Fields Editor window enable you to scroll through the records one at a time, and to jump to the first or last record of the dataset if it is active.

The Add button enables you to see a list of column names in the physical dataset but not already included in the Fields list, and to create new *TField* components for them.

The Define button enables you to create calculated fields. Fields created this way are only for display purposes. The underlying physical structure of the table or data is not changed.

The Remove button deletes the selected *TFields*. The Clear All button deletes all the *TFields* shown in the Fields list.

## Adding a TField component

The Add button of the Fields Editor enables you to specify which *TField* components will be included in a dataset. To see a list of fields currently available to a *TTable* or *TQuery* component, click the Add button. The Add Fields dialog box appears.

**Figure 3.5** Fields Editor Add Fields dialog box



The *Available Fields* list box shows all database fields that do not have persistent *TFields* instantiated. Initially, all available fields are selected. Use the mouse to select specific fields and then choose OK. The selected fields move to the *Fields* list box in the main Fields Editor window.

Fields moved to the *Available Fields* list become persistent. Each time the dataset is opened, Delphi verifies that each non-calculated field exists or can be created from data in the database. If it cannot, an exception is raised, warning you that the field is not valid, and the dataset is not opened.

## Deleting a TField component

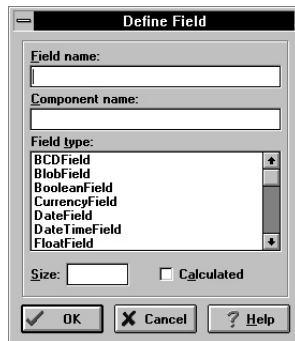
The Remove button of the Fields Editor deletes the selected *TField* components from the *Fields* list box. Fields removed from the *Field* list box are no longer in the dataset and cannot be displayed by data-aware components. Removing a *TField* component is useful to display a subset of available fields within a table, or when you want to define your own field to replace an existing field.

You can re-create deleted *TField* components with the Add button, but any changes previously made to its properties or events will be lost.

## Defining a new TField component

The Define button of the Fields Editor enables you to create new *TField* components for display. You can create a new *TField* based on a column in the underlying table (for example, to change the data type of the field), but its main purpose is to create new *TField* components whose values are calculated programmatically.

Figure 3.6 Define Field dialog box



## Defining a calculated field

A calculated field is used to display values calculated at run time in the dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from two other fields.

To create a calculated field:

- 1 Choose the Define button in the Fields Editor window.
- 2 Enter the name of the new field in the Field Name edit box, or select a field name from the drop-down list. A corresponding *TField* component name appears automatically in the Component edit box as you type. This name is the identifier you use to access the field programmatically.
- 3 Select the data type for the field from the Field Type list box.
- 4 Check the Calculated check box if it is not already checked.
- 5 Choose OK. The newly defined calculated field is automatically added to the Fields list box in the main Fields Editor window, and the component declaration is automatically added to the form's **type** declaration in the source code.

To edit the properties or events associated with the new *TField* component, select the component name in the Fields list box, then edit the properties or events via the Object Inspector.

## Programming a calculated field

Once a calculated field is defined, it has a null value unless you write code to provide values for the field. Code for a calculated field is placed in the *OnCalcFields* event for its dataset. An *OnCalcFields* event handler can only modify fields that have a *Calculated* property of True.

For example, on Form1, the *OnCalcFields* event for a *TTable* component named Table1 is

```
TForm1.Table1CalcFields
```

To program a value for a calculated field:

- 1 Select the *TTable* or *TQuery* component from the Object Inspector drop-down list.
- 2 Choose the Object Inspector Events tab.
- 3 Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the *TTable* or *TQuery* component.
- 4 Write the code that sets the values and other properties of the calculated field as desired.

## Editing a TField component

---

You can customize properties and events of persistent *TField* components at design time. To do this, select the *TField* either in the Fields list box of the Fields Editor or the component list of the Object Inspector. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

### Editing Display properties

To edit the display properties of a selected *TField* component, click the Properties tab on the Object Inspector window. The following table summarizes display properties that can be edited. Not all properties appear for all *TField* descendants.

**Table 3.7** TField properties

Property	Purpose
<i>Alignment</i>	Displays contents of field left justified, right justified, or centered within a data-aware component.
<i>Calculated</i>	True, field value can be calculated by a <i>CalcFields</i> method at run time. False, field value is determined from the current record.
<i>Currency</i>	True, numeric field displays monetary values. False, numeric field does not display monetary values.
<i>DisplayFormat</i>	Specifies the format of data displayed in a data-aware component.
<i>DisplayLabel</i>	Specifies the column name for a field in a <i>TDBGrid</i> .
<i>DisplayWidth</i>	Specifies the width, in characters, of a grid column that display this field.
<i>EditFormat</i>	Specifies the edit format of data in a data-aware component.
<i>EditMask</i>	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, etc.).
<i>FieldName</i>	Specifies the actual name of column in the physical table from which the <i>TField</i> component derives its value and data type.
<i>Index</i>	Specifies the order of the field in a dataset.
<i>MaxValue</i>	Specifies the maximum numeric value that can be entered in an editable numeric field.

**Table 3.7** TField properties (continued)

Property	Purpose
<i>MinValue</i>	Specifies the minimum numeric value that can be entered in an editable numeric field.
<i>Name</i>	Specifies the component name of the <i>TField</i> component within Delphi.
<i>ReadOnly</i>	True: Field can be displayed in a component, but cannot be edited by a user. False: Field can be displayed and edited.
<i>Size</i>	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size of byte and var byte fields.
<i>Tag</i>	Long integer bucket available for programmer use in every component as needed.
<i>Visible</i>	True: Field is displayed by a <i>TDBGrid</i> component. User-defined components can also make display decisions based on this property. False: Field is not displayed by a <i>TDBGrid</i> component.

Not all properties are available to all *TField* components. For example, a component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties. A component of type *TFloatField* does not have a *Size* property.

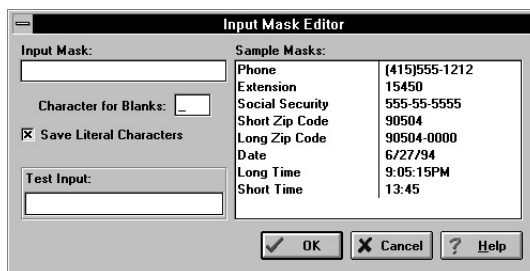
Boolean properties, those that can be toggled between True and False, can be changed by double-clicking the property in the Object Inspector. Other properties require you to enter values or pick from drop-down lists in the Object Inspector. All *TField* properties can also be manipulated programmatically.

While the purpose of most properties is straight-forward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using the *Calculated* property, see “Programming a calculated field.” For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see “Formatting fields.”

## Using the Input Mask Editor

The *EditMask* property provides a way to limit the entries that a user can type into data aware controls tied to a *TField*. You can enter a specific edit mask by hand or use the Input Mask Editor to create a mask.

To use the Input Mask Editor, when you have selected a *TField* component, double-click on the *EditMask* field in the Object Inspector or click the ellipsis button in the Values column. The Input Mask Editor opens:

**Figure 3.7** Input Mask Editor

To use one of the sample masks, select it in the Sample Masks list box. You can then customize the mask as desired by editing it in the Input Mask text field. You can test the allowable input in the Test Input field. For more information, see the online Help.

**Note** For the *TStringField*, the *EditMask* property is also used as a display format.

## Formatting fields

Delphi provides built-in display and edit format routines and intelligent default formatting for *TField* components. These routines and formats require no action on the programmer's part. Default formatting conventions are based on settings in the Windows Control Panel. For example, using default Windows settings in the United States, a *TFloatField* column with the *Currency* property set to True sets the *DisplayFormat* property for the value 1234.56 to \$1234.56, while the *EditFormat* is 1234.56. Only format properties appropriate to the data type of a *TField* component are available for a given component.

All *TField* component formatting is performed by the following routines:

**Table 3.8** TField formatting routines

Routine	Used by . . .
FormatFloat	<i>TFloatField</i> , <i>TCurrencyField</i>
FormatDateTime	<i>TDateField</i> , <i>TTimeField</i> , <i>TDateTimeField</i>
FormatInteger	<i>TIntegerField</i> , <i>TSmallIntField</i> , <i>TWordField</i>

The format routines use the International settings specified in the Windows Control Panel for determining how to display date, time, currency, and numeric values. You can edit the *DisplayFormat* and *EditFormat* properties of a *TField* component to override the default display settings for a *TField*, or you can handle the *OnGetText* and *OnSetText* events for a *TField* to do custom programmatic formatting.

## Handling TField events

To edit the events for a selected *TField* component, click the Events tab on the Object Inspector window. The following table summarizes events that can be edited:

**Table 3.9** Published TField events

Event	Purpose
<i>OnChange</i>	Called when the value for a <i>TField</i> component changes.
<i>OnGetText</i>	Called when the value for a <i>TField</i> component is retrieved for display or editing.
<i>OnSetText</i>	Called when the value for a <i>TField</i> component is set.
<i>OnValidate</i>	Called to validate the value for a <i>TField</i> component whenever the value is changed because of an edit or insert operation.

*OnGetText* and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond Delphi's built-in formatting functions, *FormatFloat*, *FormatDate*, and so on.

*TFields* have a *FocusControl* method that enables an event to set focus to the first data-aware control associated with the *TField*. This is especially important for record-oriented

validation (for example, on *BeforePost* of a *TTable*) since a *TField* may be validated whether or not its associated data control has focus.

## Using TField conversion functions

*TFields* have built-in functions for conversion among data types. Depending on the *TField* type, different conversion functions are available and do different things. The following table summarizes these functions.

**Table 3.10** TField conversion functions

TField Type	AsString	AsInteger	AsFloat	AsDateTime	AsBoolean
<b>TStringField</b>	String type by definition	Convert to Integer if possible	Convert to Float if possible	Convert to Date if possible	Convert to Boolean if possible
<b>TIntegerField</b> <b>TSmallIntField</b> <b>TWordField</b>	Convert to String	Integer type by definition	Convert to Float	Not Allowed	Not Allowed
<b>TFloatField</b> <b>TCurrencyField</b> <b>TBCDField</b>	Convert to String	Round to nearest integer value	Float type by definition	Not Allowed	Not Allowed
<b>TDateTimeField</b> <b>TDateField</b> <b>TTimeField</b>	Convert to String. Content depends on <code>DisplayFormat</code> of Field	Not Allowed	Convert Date to number of days since 01/01/0001 Convert Time to fraction of 24 hours	DateTime type by definition Zero date or time if not specified	Not Allowed
<b>TBooleanField</b>	Convert to String "True" or "False"	Not Allowed	Not Allowed	Not Allowed	Boolean type by definition
<b>TBytesField</b> <b>TVarBytesField</b> <b>TBlobField</b> <b>TMemoField</b> <b>TGraphicField</b>	Convert to String (Generally only makes sense for <code>TMemoField</code> )	Not Allowed	Not Allowed	Not Allowed	Not Allowed

The conversion functions can be used in any expression involving a *TField* component, on either side of an assignment statement. For example, the following statement converts the value of the *TField* named *MyTableMyField* to a string and assigns it to the text of the *Edit1* control:

```
Edit1.Text := MyTableMyField.AsString;
```

Conversely, this statement assigns the text of the *Edit1* control to the *TField* as a string:

```
MyTableMyField.AsString := Edit1.Text;
```

An exception occurs if an unsupported conversion is performed at run time.

## Accessing TField properties programmatically

An application can access the value of a database column through a *TField* component's *Value* property. For example, the following statement assigns the value of the *CustTableCountry* *TField* to the text in the *TEdit* component *Edit3*:

```
Edit3.Text := CustTableCountry.Value;
```

Any properties of *TField* components that are available from the Object Inspector can also be accessed and adjusted programmatically as well. For example, this statement changes field ordering by setting the *Index* property of *CustTableCountry* to 3:

```
CustTableCountry.Index := 3;
```

## Displaying data with standard controls

---

You can display database values at run time with standard components as well as data aware and *TField* components. Besides accessing *TField* components created with the Fields Editor, there are two ways to access column values at run time: the *Fields* property and the *FieldsByName* method. Each accesses the value of the current row of the specified column in the underlying database table at run time. Each requires a dataset component in the form, but not a *TDataSource* component.

In general, you should use the data-aware controls built in to Delphi in database applications. These components have properties and methods built in to them that enable them to be connected to database columns, display the current values in the columns, and make updates to the columns. If you use standard components, you must provide analogous code by hand.

### Using the Fields property

---

You can access the value of a field with the *Fields* property of a dataset component, using as a parameter the ordinal number of the column in the table (starting at 0). To access or change the field's value, convert the result with the appropriate conversion function, such as *AsString* or *AsInteger*.

This method requires you to know the order and data types of the columns in the table. Use this method if you want to iterate over a number of columns or if your application works with tables that are not available at design time.

For example, the following statement assigns the current value of the seventh column (*Country*) in the *CustTable* table to the *Edit1* component:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a column a dataset in Edit mode by assigning the appropriate *Fields* property to the value of a component. For example,

```
CustTable.Fields[6].AsString := Edit1.Text;
```

To make these assignments occur, you must enter them in an event such as a *TButton OnClick* event, or an edit control's *OnEnter* event.

### Using the FieldByName method

---

You can access the value of a field with the *FieldByName* method by specifying the dataset component name, and then passing *FieldByName* the name of the column you want to access. To access or change the field's value, convert the result with the appropriate conversion function, such as *AsString* or *AsInteger*.

This method requires you to know the name of the column you want to access or if your application works with tables that are not available at design time.

For example, the following statement assigns the value of the Country column in the CustTable table to Edit2:

```
Edit2.Text := CustTable.FieldByName('Country').AsString;
```

Conversely, you can assign a value to a column a dataset in Edit mode by assigning the appropriate *FieldByName* property to the value of a component. For example,

```
CustTable.FieldByName('Country').AsString := Edit2.Text;
```

To make these assignments occur, you must enter them in an event such as a button's *OnClick* or a Edit component's *OnExit* event.

## Incorporating reports in an application



Delphi applications can include reports created with ReportSmith with the *TReport* component. *TReport* appears on the Data Access component page. To incorporate a report in an application, simply add a *TReport* component to the desired form as you would any other component. Then specify the name of the report (created with ReportSmith) and other report parameters with properties of the component.

Designing reports with ReportSmith is described in *Creating Reports*. You can invoke ReportSmith at design time by double-clicking on a *TReport* component, or on the ReportSmith icon in the Delphi program group.

Specify the name of an existing report in the *ReportName* property and the directory in the *ReportDir* property. To load ReportSmith Runtime and print the specified report, use the *Run* method (i.e., `Report1.Run`). The report prints on the default printer defined in ReportSmith. *Preview* is a Boolean property that specifies whether to print the report or just display it: If set to True, *Run* will display the report onscreen only; if set to False, *Run* will print the report.

The *AutoUnload* property specifies whether to automatically unload the ReportSmith Runtime executable after a report is run. Generally, if an application runs one report at a time, *AutoUnload* should be True. If an application is going to run a series of reports, then *AutoUnload* should be False.

The *InitialValues* property is of type *TStrings* and specifies the report variables to use with the report. Each line specifies a report variable as follows:

```
REPORTVAR = value
```

For more information about creating and using report variables, see *Creating Reports*.

Some important methods of *TReport* are listed in the following table:

**Table 3.11** Important TReport methods

Method	Purpose
<i>Run</i>	Run a report.
<i>RunMacro</i>	Send a macro command to ReportSmith.



**Table 3.11** Important TReport methods (continued)

Method	Purpose
<i>Connect</i>	Preconnect the report to a database, so it does not prompt for login.
<i>SetVariable</i>	Change a specific report variable.
<i>ReCalcReport</i>	Run a report again. Use this when report variables have changed.

## Using TBatchMove



*TBatchMove* enables you to perform operations on groups of records or entire tables. The primary uses of this component are

- Downloading data from a server to a local data source for analysis or other operations.
- Upsizing a database from a desktop data source to a server. For more information on upsizing, see Appendix C, “Using local SQL.”

*TBatchMove* is powerful because it can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

Two *TBatchMove* properties specify the source and a destination for the batch move operation: *Source* specifies a dataset (a *TQuery* or *TTable* component) corresponding to an existing source table. *Destination* specifies a *TTable* component corresponding to a database table. The destination table may or may not already exist.

### Batch move modes

The *Mode* property specifies what the *TBatchMove* object will do:

**Table 3.12** Batch move modes

Property	Purpose
<i>batAppend</i>	Append records to the destination table. The destination table must already exist. This is the default mode.
<i>batUpdate</i>	Update records in the destination table with matching records from the source table. The destination table must exist and must have an index defined to match records.
<i>batAppendUpdate</i>	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table. The destination table must exist and must have an index defined to match records.
<i>batCopy</i>	Create the destination table based on the structure of the source table. The destination table must not already exist—if it does, the operation will delete it.
<i>batDelete</i>	Delete records in the destination table that match records in the source table. The destination table must already exist and must have an index defined.

The *Transliterate* property specifies whether to transliterate character data to the preferred character set for the destination table.

## Data type mappings

In *Copy* mode, *TBatchMove* will create the destination table based on the column data types of the source table. In moving data between different table types, *TBatchMove* translates the data types as appropriate. The mappings from dBASE, Paradox, and InterBase data types are shown in the following tables.

**Note** To batch move data to an SQL server database, you must have that database server and Delphi Client/Server edition with the appropriate SQL Link installed. For more information, see the *SQL Links for Windows User's Guide*.

**Table 3.13** Physical data type translations from Paradox tables to tables of other driver types

From Paradox type	To dBASE type	To Oracle type	To Sybase type	To InterBase type	To Informix type
Alpha	Character	Character	VarChar	Varying	Character
Number	Float {20.4}	Number	Float	Double	Float
Money	Float {20.4}	Number	Money	Double	Money {16.2}
Date	Date	Date	DateTime	Date	Date
Short	Number {6.0}	Number	SmallInt	Short	SmallInt
Memo	Memo	Long	Text	Blob/1	Text
Binary	Memo	LongRaw	Image	Blob	Byte
Formatted memo	Memo	LongRaw	Image	Blob	Byte
OLE	OLE	LongRaw	Image	Blob	Byte
Graphic	Binary	LongRaw	Image	Blob	Byte
Long	Number {11.0}	Number	Int	Long	Integer
Time	Character {>8}	Character {>8}	Character {>8}	Character {>8}	Character {>8}
DateTime	Character {>8}	Date	DateTime	Date	DateTime
Bool	Bool	Character {1}	Bit	Character {1}	Character
AutoInc	Number{11.0}	Number	Int	Long	Integer
Bytes	Memo	LongRaw	Image	Blob	Byte
BCD	N/A	N/A	N/A	N/A	N/A

**Table 3.14** Physical data type translations from dBASE tables to tables of other driver types

From dBASE type	To Paradox type	To Oracle type	To Sybase type	To InterBase type	To Informix type
Character	Alpha	Character	VarChar	Varying	Character
Number	Short	Number	SmallInt	Short	SmallInt
others	Number	Number	Float	Double	Float
Float	Number	Number	Float	Double	Float
Date	Date	Date	DateTime	Date	Date
Memo	Memo	Long	Text	Blob/1	Text
Bool	Bool	Character {1}	Bit	Character {1}	Character
Lock	Alpha {24}	Character {24}	Character {24}	Character {24}	Character
OLE	OLE	LongRaw	Image	Blob	Byte

**Table 3.14** Physical data type translations from dBASE tables to tables of other driver types (continued)

From dBASE type	To Paradox type	To Oracle type	To Sybase type	To InterBase type	To Informix type
Binary	Binary	LongRaw	Image	Blob	Byte
Bytes	Bytes	LongRaw	Image	Blob	Byte (only for temp tables)

**Table 3.15** Physical data type translations from InterBase tables to tables of other driver types

From Interbase type	To Paradox type	To dBASE type	To Oracle type	To Sybase type	To Informix type
Short	Short	Number {6.0}	Number	Small Int	SmallInt
Long	Number	Number {11.0}	Number	Int	Integer
Float	Number	Float {20.4}	Number	Float	Float
Double	Number	Float {20.4}	Number	Float	Float
Char	Alpha	Character	Character	VarChar	Character
Varying	Alpha	Character	Character	VarChar	Character
Date	DateTime	Date	Date	DateTime	DateTime
Blob	Binary	Memo	LongRaw	Image	Byte
Blob/1	Memo	Memo	Long	Text	Text

By default *TBatchMove* matches columns based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the *Mappings* property. This is a list of column mappings (one per line) in one of two forms. To map the column, *ColName*, in the source table to the column of the same name in the destination table:

```
ColName
```

Or, to map the column named *SourceColName* in the source table to the column named *DestColName* in the destination table:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, *TBatchMove* will perform a “best fit”. It will trim character data types, if necessary, and attempt to perform a limited amount of conversion if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, *TBatchMove* will convert a character value of ‘5’ to the corresponding integer value. Values that cannot be converted will generate errors. See “Handling batch move errors” on page 92.

## Executing a batch move

---

Use the *Execute* method to execute the batch operation at run time. For example, if *BatchMoveAdd* is the name of a *TBatchMove* component, the following statement executes it:

```
BatchMoveAdd.Execute
```

You can also execute a batch move at design time by right clicking the mouse on a *TBatchMove* component and choosing *Execute*. The *MoveCount* property keeps track of the number of records that were moved when a batch move is executed.

## Handling batch move errors

---

There are basically two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. *TBatchMove* has a number of properties that specify how it handles errors. The *AbortOnProblem* property specifies whether to abort the operation when a data type conversion error occurs. The *AbortOnKeyViol* property indicates whether to abort the operation when an integrity (key) violation occurs.

The following properties enable a *TBatchMove* to create additional tables that document the batch move operation:

- *ChangedTableName*, if specified, creates a local (Paradox) table containing all records in the destination table that changed as a result of the batch operation.
- *KeyViolTableName*, if specified, creates a local (Paradox) table containing all records from the source table that caused an integrity violation (such as a key violation) as a result of the batch operation.
- *ProblemTableName*, if specified, creates a local (Paradox) table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table.

## Accessing the BDE directly

---

Delphi provides a wide range of built-in methods, properties, and functions that provide an interface to the Borland Database Engine, but applications are not limited to them. Some advanced applications may require direct access to BDE function calls, cursors, and so on. While direct BDE calls can provide additional functionality, they should be performed with caution, as they bypass Delphi's built-in functionality that keeps data-aware components synchronized with datasets.

If your application requires direct access to the BDE, you should first get the *Borland Database Engine User's Guide* from Borland. This documentation provides a complete reference and user's guide to the BDE.

The application must include the header files that reference the BDE: DBIPROCS.PAS and DBTYPES.PAS. Then the code can make direct calls to the BDE application programming interface.

BDE function calls often require parameters to specify the action to be performed. Delphi provides access to these through the following properties of dataset components:

- *DBHandle* is the handle for the database to which they are connected.
- *Handle* is the handle for the underlying cursor on the database.
- *DBLocale* and *Locale* are used for ANSI/OEM conversion for localization of applications.

After performing a BDE call directly, it is a good idea to call *Refresh* to ensure that all data-aware components are synchronized with their datasets.

## Application examples

---

This section provides some brief examples of specific database tasks, illustrating some of the material presented in the preceding sections.

### Creating a master-detail form

---

In the following example, you will create a simple form in which the user can scroll through customer records, and display all orders for the current customer. Follow these steps to create this application:

- 1 Place two *TTable*, two *TDataSource*, and two *TDBDataGrid* components on a form.
- 2 Set the properties of the first *TTable* component as follows:
  - *DatabaseName*: DBDEMO (the alias for the directory with the MAST database).
  - *TableName*: CUSTOMER (the table containing customer records).
  - *Name*: CustTable (for ease-of-use).
- 3 Name the first *TDataSource* component “CustDataSource,” and set its *Dataset* property to “CustTable.” Set the *DataSource* property of DBGrid1 to “CustDataSource.” When you activate CustTable (by setting its *Active* property to True), the grid displays the data in the CUSTOMER table.
- 4 Analogously, set the properties of the second *TTable* component as follows:
  - *DatabaseName*: DBDEMO.
  - *TableName*: ORDERS (the table containing order records).
  - *Name*: OrdTable (for ease-of-use).
- 5 Name the second *TDataSource* component “OrdDataSource,” and set its *Dataset* property to “OrdTable.” Set the *DataSource* property of DBGrid2 to “OrdDataSource.” When you activate OrdTable (by setting its *Active* property to True), the grid displays the data in the ORDERS table.

- 6 Compile and run the application now. The form displays data from each table independently and should look something like this:

**Figure 3.8** Sample form

CustNo	Company
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver
1354	Cayman Divers World Unlimited

OrderNo	CustNo	SaleDate	ShipDate
1003	1351	4/12/88	
1004	1354	4/17/88	
1005	1356	4/20/88	
1006	1380	11/6/94	

- 7 The next step is to link the ORDERS table (the master table) to the CUSTOMER table (the detail table) so that the form displays only the orders placed by the current customer. To do this, exit the application, return to design mode, and set the *MasterSource* property of *OrdTable* to *CustDataSource*.
- 8 In the Object Inspector, click on the ellipsis button to the right of the *MasterFields* property of *OrdTable*. The Field Link Designer dialog box will open.
  - In the Available Indexes field, choose *ByCustNo* to link the two tables by the *CustNo* field.
  - Select *CustNo* in both the Detail Fields and Master Fields field lists.
  - Click on the Add button to add this join condition. In the Joined Fields list, "*CustNo -> CustNo*" will appear.
  - Choose OK to commit your selections and exit the Field Link Designer.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

The *MasterSource* property specifies the *TDataSource* from which *OrdTable* will take its master column values. This limits the records it retrieves, based on the current record in *CustTable*. To do this, you must specify for *OrdTable*:

- The name of the column that links the two tables. The column must be present in each table, and must be identically named.
- The index of the column in the ORDERS table that will be linked to the CUSTOMER table.

In addition, you must ensure that the ORDERS table has an index on the *CustNo* field. Since it is a primary index, there is no need to specifically name it, and you can safely leave the *IndexName* field blank in both tables. However, if the table were linked through a secondary index, you must explicitly designate that index in the *IndexName* property.

In this example, the CUSTOMER table has a primary index on the *CustNo* column, so there is no need to specify the index name. However, the ORDERS table does not have a

primary index on *CustNo*, so you must explicitly declare it in the *IndexName* property, in this case *ByCustNo*

**Note** You can also set the *IndexFieldNames* property to *CustNo*, and the correct index will be supplied for you.

## Displaying multiple views of a table

Applications often need to display two different views of the same dataset. For instance, if you have two forms, and each displays different columns of the same row, you need to find a way to keep these forms synchronized to ensure accurate and up-to-date views of the data in each form.

There are two main concepts important to working with multiple views of a table:

- The *TDataSource* component provides access for multiple forms sharing a single dataset.
- When you want more than one different view of one table, put a *TDataSource* on each form that contains data aware components. You only need put a single *TTable* or *TQuery* component on the first form.

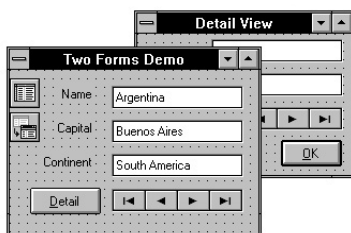
The simplest way to do this is to place a *TDataSource* on each form, and connect the first data source to the dataset you want to access. At run time you can then set the *DataSet* property of the second *TDataSource* to the *TTable* on the first form, for example:

```
Form2.DataSource2.DataSet := Form1.Table1;
```

Once the second *TDataSource* is assigned to a valid dataset, both forms will remain synchronized.

The *TWOFORMS.DPR* example in `\DELPHI\DEMOS\DB\TWOFORMS` demonstrates how to work with multiple forms and a single dataset. The program opens the *COUNTRY* table and shows the *Name*, *Capital*, and *Continent* fields on one form, and the *Area* and *Population* fields on a second form. A button on the first form opens the second form. Both forms have *TDBNavigator* components, so you can navigate through the table. The forms look like this:

**Figure 3.9** Two forms



- ✓ To create the program manually,
  - 1 Place a *TTable*, a *TDataSource*, a *TButton*, three *TDBEdit*, and three *TLabel* components on a form.

- 2 Give the Button the name "Detail."
- 3 Set the Table1's *DatabaseName* property to the DBDemos alias, and open the COUNTRY Table.
- 4 Connect DataSource1 to *Table1*, then connect each of the DBEdit component's *Datasource* properties to DataSource1. By performing these steps in this order, you can drop down a list in the *DataField* property of the data aware controls.
- 5 Create a second form, and place a data source, a *TBitBtn*, two *TDBLabels*, a *TDBNavigator* and two *TDBEdit* controls on the form. Don't yet connect the data source on this form to anything else.
- 6 Connect the DBEdit components to the DataSource, then enter 'Area' for the *DataField* property of DBEdit1, and 'Population' for DBEdit2.
- 7 Set the *Kind* property of the bitbutton to 'OK'.
- 8 Name this second form *DetailView*, then save the whole unit under the name DETAILS.PAS.

**Note** When you are creating the second form, you might find it helpful to temporarily create a *TTable* component and link the data source to it. This enables you to design the form using live data, and gives you access to a list of field names for each edit control. Once you have the form set up properly, you can delete the *TTable* component and at run time reconnect the data source to a table on a separate form.

- ✓ Once you have built the second form, go back to the first form and add DETAILS to the **uses** clause in the form unit's **implementation** part, and create the following event handler for the *OnClick* event of the Detail button:

```
procedure TForm1.DetailClick(Sender: TObject);
begin
    DetailView.DataSource1.DataSet := Table1;
    DetailView.Visible := True;
end;
```

This code first assigns the data source on the second form to the table in the first form. Once this connection is made, then the data source on the second form is "live." That is, it will act exactly like the data source on the first form. Then it makes the form visible.

- ✓ Now, connect the navigators on each form to the appropriate data source.

When you run the program, open the second form by clicking on the Detail button. Notice that whether you use the navigator in either form, the edit controls on the other form remain synchronized with the current record.

- ✓ Close the application and add two more lines of code to the second form's unit. This code is called in response to a click on the OK button:

```
procedure TDetailView.BitBtn1Click(Sender: TObject);
begin
    DataSource1.DataSet := nil;
    Close;
end;
```



This code sets the dataset to `null` whenever the second form is closed. While not necessary, this ensures that the hidden detail view is not responding to events.



## Using Data Controls

To display and edit data from a database, use the components on the Data Controls page of the Component palette. Data controls include components such as *TDBGrid* for displaying and editing all specified records and fields in a table, and *TDBNavigator* for navigating among records, deleting records, and posting records when they change.

**Figure 4.1** Data Controls Component palette



The following table summarizes the data controls in order from left to right as they appear on the Component palette:

**Table 4.1** Data controls

Data control	Description
TDBGrid	Displays information from a data source in a spreadsheet-like grid. Columns in the grid can be specified at design time using the Fields Editor or at run time (dynamically bound).
TDBNavigator	Provides buttons for navigating through data obtained from a data source. At design time, you can choose to include one or more buttons to navigate through records, update records, post records, and refresh the data from the data source.
TDBText	Displays data from a specific column in the current data record.
TDBEdit	Uses an edit box to display and edit data from a specific column in the current data record.
TDBMemo	Displays memo-type database data. Memo fields can contain multiple lines of text or can contain BLOB (binary large object) data.
TDBImage	Displays graphic images and BLOB data from a specific column in the current data record.
TDBListBox	Displays a list of items from which a user can update a specific column in the current data record.
TDBComboBox	Combines a TDBEdit control with an attached list. The application user can update a specific column in the current data record by typing a value or by choosing a value from the drop-down list.

**Table 4.1** Data controls (continued)

Data control	Description
TDBCheckBox	Displays a check box. If the value in the indicated column of the current record matches the text of the check box's <i>ValueChecked</i> property, the box is checked.
TDBRadioGroup	Offers a mutually exclusive set of options to enter into a specific column in the current data record in the form of radio buttons.
TDBLookupList	Displays a list of items from which a user can update a column in the current data record. The list of items is looked up in a specific column in another dataset.
TDBLookupCombo	Combines a TDBEdit control with a dropdown version of TDBLookupList. The application user can update a field in the current database by typing a value or by choosing a value from the drop-down list that is looked up in a specific column in another dataset.

Most data controls are data-aware versions of standard components. Some, such as *TDBGrid* and *TDBNavigator*, are data-aware, but differ from standard components in significant, useful ways. A *data-aware* control derives display data from a database source outside the application, and can also optionally post (or return) data changes to a data source. Data controls are data-aware at *design time*, meaning that when you connect a component to an active data source while building an application, you can immediately see live data in the controls. You can use the Fields Editor to scroll through records at design time to verify that your application is making the right database connections without requiring you to compile and run the application, but you cannot modify data at design time.

This chapter describes basic features common to all Data Control components, then describes how and when to use individual components.

## Data Control component basics

Data controls are linked to database tables through the *DataSource* property. The *DataSource* property specifies the name of the *TDataSource* component from which a control gets its data. The *TDataSource* component is linked to a dataset (for example, *TTable* or *TQuery*) that is, in turn, connected to a database table. A dataset component is a grouping of *TField* components that can be dynamically created for you at run time, or statically specified using the Fields Editor at design time. For more information about *TDataSource*, *TTable*, *TQuery*, and the Fields Editor, see Chapter 3, "Using data access components and tools."

Data controls can only access columns in tables for which there are corresponding *TField* components. If the Fields Editor is used to limit a dataset to a subset of columns in a table, then *TField* components exist only for those columns. Most data controls provide a *DataField* property where you can specify the *TField* component with which it should be associated.

When designing a form that accesses data, you must place at least one dataset component (for example, *TTable* or *TQuery*), at least one *TDataSource* component, and one or more data controls on the form.

To place a data control on a form and link it to a dataset, follow these steps:

- 1 Drop the control on the form.

- 2 Set the *DataSource* property to the name of a *TTable* or *TQuery* component already on the form. You can type the name or choose it from the drop-down list.
- 3 Set the *DataField* property to the name of a *TField* component. You can type the field name or choose it from the drop-down list.

**Note** Two data controls, *TDBGrid* and *TDBNavigator*, access all available *TField* components within a dataset, and therefore do not have *DataField* properties. For these controls, omit step 3.

When a data control is associated with a dataset, its *Enabled* property determines if its attached *TDataSource* component receives data from mouse, keyboard, or timer events. Controls are also disabled if the *Enabled* property of *TDataSource* is *False*, or if the *Active* property of the dataset is *False*.

## Updating fields

---

Most data controls can update fields by default. Update privileges depend on the status of the control's *ReadOnly* property and underlying *TField*'s and dataset's *CanModify* property. *ReadOnly* is set to *False* by default, meaning that data modifications can be made.

In addition, the data source must be in *Edit* state if updates are to be permitted. The data source *AutoEdit* property, set to *True* by default, ensures that the dataset enters *Edit* mode whenever an attached control starts to modify data in response to keyboard or mouse events.

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying *TField* component when you *Tab* from the control. If you press *Esc* before you *Tab* from a field, then Delphi abandons the modifications, and the value of the field reverts to the value it held before any modifications were made. In *TDBGrid*, modifications are copied only when you move to a different record; you can press *Esc* in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware components associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

## Displaying data as labels with TDBText

---



*TDBText* is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. *TDBText* gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*. When *TDBText* is linked to a data field at design time, however, you can see the current value for that field. For example, the following *TDBText* box, from the tutorial example, *MASTAPP*, displays company names.

**Figure 4.2** TDBText component



A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zipcode field from a separate table. A *TDBText* component tied to the zipcode table could be used to display the zipcode field that matches the address entered by the user.

**Note** When you create a *TDBText* component on a form, make sure its *AutoSize* property is True (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is set to False, and the control is too small, data display is truncated.

## Displaying and editing fields with TDBEdit



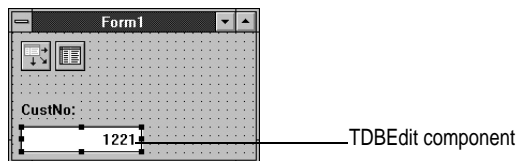
*TDBEdit* is a data-aware version of the *TEdit* component. *TDBEdit* displays the current value of a data field to which it is linked. You can also modify values in this component.

For example, suppose *DataSource1* is a *TDataSource* component that is active and linked to an open *TTable* called *Customer*. You can then create a *TDBEdit* component (*DBEdit1*), and set its properties as follows:

- *DataSource*: *DataSource1*
- *DataField*: *CUSTNO*

The *DBEdit1* component immediately displays the value of the current row in the *CUSTNO* column of the *CUSTOMER* table, both at design time and at run time.

**Figure 4.3** TDBEdit component at design time



## Editing a field

A user can modify a database field in a *TDBEdit* component if:

- 1 The *Dataset* is in Edit state.
- 2 The *Can Modify* property of the *Dataset* is True.
- 3 The *ReadOnly* property of *TDBEdit* is False.

**Note** Edits made to a field must be posted to the database by using a navigation or *Post* button on a *TDBNavigator* component.

# Viewing and editing data with TDBGrid



*TDBGrid* enables you to view and edit all records associated with a dataset in a spreadsheet-like format:

**Figure 4.4** TDBGrid component

The diagram shows a TDBGrid component with a table of data. The table has four columns: VendorName, Address1, City, and State. The first row is highlighted, and a record indicator points to it. The table is labeled with 'Current field' and 'Column titles' above the columns, and 'Record indicator' to the left of the rows.

Current field	Column titles	VendorName	Address1	City	State
Record indicator		Cacor Corporation	161 Southfield Rd	Southfield	OH
		Underwater	50 N 3rd Street	Indianapolis	IN
		J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
		Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
		Divers' Supply Shop	5208 University Dr	Macon	GA
		Techniques	52 Dolphin Drive	Redwood City	CA
		Perry Scuba	3443 James Ave	Hapeville	GA

The appearance of records in *TDBGrid* depends entirely on whether the *TField* components of the dataset are dynamically created by Delphi at run time, or if you use the Fields Editor to create persistent set of *TField* components whose properties you can specify in the Object Inspector at design time.

If Delphi generates a dynamic dataset at run time, then all records are displayed using default record and field ordering, and default display and edit formats. In most cases, however, you will want to control field order and appearance. To do so, use the Fields Editor to instantiate *TField* components and set their properties at design time.

When you use the Fields Editor to instantiate *TField* components, you gain a great deal of flexibility over the appearance of records in a *TDBGrid*. For example, the order in which fields appear from left to right in *TDBGrid* is determined by the way you order *TField* components in the Fields list box of the Fields Editor. Similarly, the *DisplayFormat* and *EditFormat* properties of a *TField* component determine how that field appears in *TDBGrid* during display and editing, respectively. You can also ensure that a value is entered for a field in a new record by setting its *Required* property to True. You can even prevent a *TField* component from being displayed in a grid by setting its *Visible* property to False. For more information about using the Fields Editor to control *TField* properties, see Chapter 3, "Using data access components and tools."

To put a *TDBGrid* on a form and link it to a dataset:

- 1 Create the control on the form.
- 2 Set the *DataSource* property to the name of a *TTable* or *TQuery* component already on the form.

## Setting grid options

You can use the *TDBGrid Options* property at design time to control grid behavior and appearance at run time. When a *TDBGrid* component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean

properties that you can set individually. To view and set these properties, double-click the *Options* property. The list of options that you can set appears in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, indicating that you can collapse the list of properties by double-clicking the *Options* property.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at run time:

**Table 4.2** Expanded TDBGrid Options properties

Option	Purpose
dgEditing	True: (Default). Enables editing, inserting, and deleting records in the grid. False: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	True: When a field is selected, it is in Edit state. False: (Default). A field is not automatically in Edit state when selected.
dgTitles	True: (Default). Displays field names across the top of the grid. False: Field name display is turned off.
dgIndicator	True: (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam. False: The indicator column is turned off.
dgColumnResize	True: (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying TField component. False: Columns cannot be resized in the grid.
dgColLines	True: (Default). Displays vertical dividing lines between columns. False: Does not display dividing lines between columns.
dgRowLines	True: (Default). Displays horizontal dividing lines between records. False: Does not display dividing lines between records.
dgTabs	True: (Default). Enables tabbing between fields in records. False: Tabbing exits the grid control.
dgRowSelect	True: The selection bar spans the entire width of the grid. False: (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	True: (Default). The selection bar in the grid is always visible, even if another control has focus. False: The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	True: (Default). Prompt for confirmation to delete records ( <i>Ctrl+Del</i> ). False: Delete records without confirmation.

For more information about these options, see the online Help reference.

## Editing in the grid

At run time, you can use a grid to modify existing data and enter new records, if all the following default conditions are met:

- 1 The *CanModify* property of the *Dataset* is True.
- 2 The *ReadOnly* property of *TDBGrid* is False.



In most data controls, edits to a field are posted as soon as you *Tab* to another control. By default in *TDBGrid*, edits and insertions within a field are posted only when you move to a different record in the grid. Even if you use the mouse to change focus to another control on a form, the grid does not post changes until you move off the current row. When a record is posted, Delphi checks all data-aware components associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and does not modify the record.

You can cancel all edits for a record by pressing *Esc* in any field before moving to another record.

## Rearranging column order at run time

---

At run time, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. Dragging a column to a new position in the grid affects the order of columns in the grid and the dataset. It does not change the order of columns in the underlying physical table.

To prevent rearrangement of column order, set the *DragMode* property to *dmAutomatic*.

## Controlling grid drawing

---

You can control how individual cells in a grid are drawn. The *DefaultDrawing* property controls drawing of cells. By default, *DefaultDrawing* is set to *True*, meaning that Delphi uses the grid's default drawing method to paint cells and the data they contain. Data is drawn according to the *DisplayFormat* or *EditFormat* properties of the *TField* component associated with a given column.

If you set a grid's *DefaultDrawing* property to *False*, Delphi does not draw cells or cell contents in the grid. You must supply your own drawing routine that is keyed to the *OnDrawDataCell* event for *TDBGrid*.

## Using events to control grid behavior

---

You can modify grid behavior by writing events handlers to respond to specific actions within the grid. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists *TDBGrid* events available in the Object Inspector:

**Table 4.3** TDBGrid events

Event	Purpose
<i>OnColEnter</i>	Specify action to take when a user moves into a column on the grid.
<i>OnColExit</i>	Specify action to take when a user leaves a column on the grid.
<i>OnDblClick</i>	Specify action to take when a user double clicks in the grid.
<i>OnDragDrop</i>	Specify action to take when a user drags and drops in the grid.
<i>OnDragOver</i>	Specify action to take when a user drags over the grid.

**Table 4.3** TDBGrid events (continued)

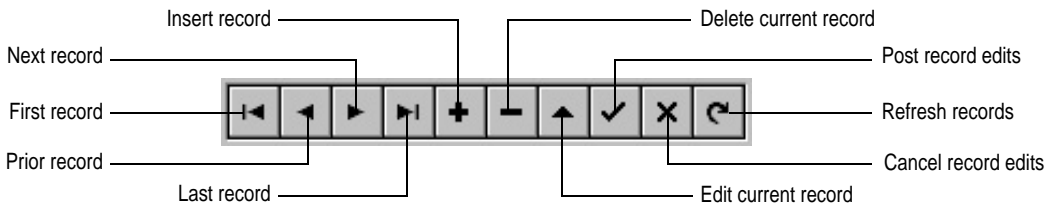
Event	Purpose
<i>OnDrawDataCell</i>	Customize drawing of grid cells.
<i>OnEndDrag</i>	Specify action to take when a user stops dragging on the grid.
<i>OnEnter</i>	Specify action to take when the grid gets focus.
<i>OnExit</i>	Specify action to take when the grid loses focus.
<i>OnKeyDown</i>	Specify action to take when a user presses any key or key combination on the keyboard when in the grid.
<i>OnKeyPress</i>	Specify action to take when a user presses a single alphanumeric key on the keyboard when in the grid.
<i>OnKeyUp</i>	Specify action to take when when a user releases a key when in the grid.

There are many uses for these events. For example, you might write a handler for the *OnDblClick* event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the *SelectedField* property to determine to current row and column. For more information about *TDBGrid* events and properties, see the online Help reference.

## Navigating and manipulating records with TDBNavigator



*TDBNavigator* provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

**Figure 4.5** TDBNavigator component

The following table describes the buttons on the navigator:

**Table 4.4** TDBNavigator buttons

Button	Purpose
First	Calls the dataset's <i>First</i> method to set the current record to the first record.
Prior	Calls the dataset's <i>Prior</i> method to set the current record to the previous record.
Next	Calls the dataset's <i>Next</i> method to set the current record to the next record.
Last	Calls the dataset's <i>Last</i> method to set the current record to the last record.
Insert	Calls the dataset's <i>Insert</i> method to insert a new record before the current record, and set the dataset in Insert state.

**Table 4.4** TDBNavigator buttons (continued)

Button	Purpose
Delete	Deletes the current record. If the <i>ConfirmDelete</i> property is True it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

## Hiding and showing navigator buttons

When you first put a *TDBNavigator* on a form, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, on a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

The *VisibleButtons* property in the Object Inspector is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, double-click the *VisibleButtons* property. The list of buttons that can be turned on or off appears in the Object Inspector below the *VisibleButtons* property. The + sign changes to a – (minus) sign, indicating that you can collapse the list of properties by double-clicking the *VisibleButtons* property.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to True, the button appears in the *TDBNavigator*. If False, the button is removed from the navigator at design and run times.

**Note** As button values are set to False, they are removed from the *TDBNavigator* on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

For more information about buttons and the methods they call, see the online Help reference.

## Displaying fly-by help

By default, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. The *ShowHint* property of *TDBNavigator*, set to False by default, controls this functionality. Set *ShowHints* to True to display Help Hints.

You can provide your own hint text for a navigator by entering separate strings for each hint in the *Hints* property. The strings you enter override the default hints provided by the navigator control.

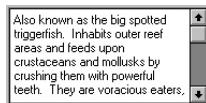
## Displaying and editing BLOB text with TDBMemo



*TDBMemo* is a data-aware component—similar to the Standard *TMemo* component—that can display binary large object (BLOB) data. *TDBMemo* displays multi-line text, and

permits a user to enter multi-line text as well. For example, you can use *TDBMemo* controls to display memo fields from dBASE and Paradox tables and text data contained in BLOB fields.

**Figure 4.6** TDBMemo component



By default, *TDBMemo* permits a user to edit memo text. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. To make a *TDBMemo* component read-only, set its *ReadOnly* property to *True*.

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the *ScrollBars* property. To prevent word wrap, set the *WordWrap* property to *False*. To permit tabs in a memo, set the *WantTabs* property to *True*. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*.

At run time, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at run time. To reduce the time it takes scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should automatically be displayed. If you set *AutoDisplay* to *False*, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

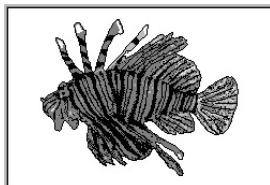
**Note** A *TDBMemo* control raises an exception if an attempt is made to access fields that contain more than 32K of data, or if edited data exceeds 32K in length.

## Displaying BLOB graphics with TDBImage



*TDBImage* is a data-aware component that displays bitmapped graphics contained in BLOB data fields. It captures BLOB graphics images from a dataset, and stores them internally in the Windows .DIB format.

**Figure 4.7** DBImage component



By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard, or if you supply an editing method. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and

*PasteFromClipboard* methods. To make a *TDBImage* component read-only, set its *ReadOnly* property to True.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at run time. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay* to False, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

## Using list and combo boxes

There are four data controls that provide data-aware versions of standard list box and combo box controls. These useful controls provide the user with a set of default data values to choose from at run time.

**Note** These components can be linked only to *TTable* components. They do not work with *TQuery* components.

The following table describes these controls:

**Table 4.5** Data-aware list box and combo box controls

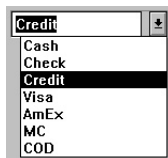
Data control	Description
TDBListBox	Displays a list of items from which a user can update a specific column in the current data record.
TDBComboBox	Combines a TDBEdit control with an attached list. The application user can update a specific column in the current data record by typing a value or by choosing a value from the drop-down list.
TDBLookupList	Displays a list of items from which a user can update a column in the current data record. The list of items is looked up in a specific column in another dataset.
TDBLookupCombo	Combines a TDBEdit control with a dropdown version of TDBLookupList. The application user can update a field in the current database by typing a value or by choosing a value from the drop-down list that is looked up in a specific column in another dataset.

### TDBComboBox



The *TDBComboBox* component is similar to a *TDBEdit* component, except that at run time it has a drop-down list that enables a user to pick from a predefined set of values. Here is an example of what a *TDBComboBox* component looks like at run time:

**Figure 4.8** DBComboBox component



The *Items* property of the component specifies the items contained in the drop-down list. Use the String List Editor to specify the values for the drop-down list.

At run time, the user can choose an item from the list or (depending on the value of the *Style* property) type in a different entry. When the component is linked to a column through its *DataField* property, it displays the value in the current row, regardless of whether it appears in the *Items* list.

The following properties determine how the *Items* list is displayed at run time:

- *Style* determines the display style of the component:
  - *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
  - *csSimple*: Displays the *Items* list at all times instead of in a drop-down list. All items are strings and have the same height.
  - *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at run time.
  - *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps). For more information, see the online VCL reference.
- *DropDownCount*: the maximum number of items displayed in the list. If the number of *Items* is greater than *DropDownCount*, the user can scroll the list. If the number of *Items* is less than *DropDownCount*, the list will be just large enough to display all the *Items*.
- *ItemHeight*: The height of each item when style is *csOwnerDrawFixed*.
- *Sorted*: If True, then the *Items* list will be displayed in alphabetical order.

## TDBListBox



*TDBListBox* is functionally the same as *TDBComboBox*, but instead of a drop-down list, it displays a scrollable list of available choices. When the user selects a value at run time, the component is assigned that value. Unlike *TDBComboBox*, the user cannot type in an entry that is not in the list.

Here is an example of how a *TDBListBox* component appears at run time.

**Figure 4.9** TDBListBox component



While navigating through a dataset, a *TDBListBox* component displays values in the column by highlighting the corresponding entry in its list. If the current row's value is not defined in the *Items* property, no value is highlighted in the *TDBListBox*. Changing the selection changes the underlying value in the database column and is equivalent to typing a value in a *TDBEdit* component.

The *IntegralHeight* property controls the way the list box is displayed. If *IntegralHeight* is True (the default), the bottom of the list box moves up to the bottom of the last

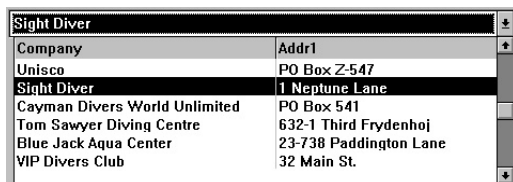
completely-displayed item in the list. If *IntegralHeight* is False, the bottom of the list box is determined by the *ItemHeight* property, and the bottom item might not be completely displayed.

## TDBLookupCombo



The *TDBLookupCombo* component is similar to *TDBComboBox*, except that it derives its list of values dynamically from a second dataset at run time, and it can display multiple columns in its drop-down list. With this control, you can ensure that users enter valid values into forms by providing an interface from which they can choose values. Here is an example of how a *TDBLookupCombo* component appears at run time:

Figure 4.10 TDBLookupCombo component



Sight Diver	
Company	Addr1
Unisco	PO Box Z-547
Sight Diver	1 Neptune Lane
Cayman Divers World Unlimited	PO Box 541
Tom Sawyer Diving Centre	632-1 Third Frydenhoj
Blue Jack Aqua Center	23-738 Paddington Lane
VIP Divers Club	32 Main St.

The lookup list for *TDBLookupCombo* must be derived from a second dataset. To display values from a column in the same table as the first dataset, drop a second data source and dataset component on the form and point them at the same data as the first data source and dataset.

Three properties establish the lookup list for *TDBLookupCombo*, and determine how it is displayed:

- *LookupSource* specifies a second data source from where the control populates its list.
- *LookupField* specifies a field in the *LookupSource* dataset which links that dataset to the primary dataset. This must be a column in the dataset pointed to by *LookupSource*, and it must contain the same values as the column pointed to by the *DataField* property (although the column names do not have to match).
- *LookupDisplay*, if set, defines the columns that *TDBLookupCombo* displays. If you do not specify values in *LookupDisplay*, *TDBLookupCombo* displays the values found in the column specified by *LookupField*. Use this property to display a column other than that specified by *LookupField*, or to display multiple columns in the drop-down list. To specify multiple columns, separate the different column names with a semicolon.

A *TDBLookupCombo* component appears similar to a *TDBComboBox* at both design time and run time, except when you want it to display multiple columns in its lookup list. How the control displays multiple columns depends on the *Options* property settings:

- *loColLines*: When True, uses lines to separate the columns displayed in the lookup list.
- *loRowLines*: When True, uses lines to separate the rows displayed in the lookup list.
- *loTitles*: When True, column names appear as titles above the columns displayed in the lookup list.

As a simple example, an order entry form could have a *TDBLookupCombo* component to specify the customer number of the customer placing the order. The user placing the order can simply click on the drop down “pick list” instead of having to remember the customer number. The value displayed could be the customer name.

To build this form,

- 1 Choose File | New Project to create a new form.
- 2 Create a *TDataSource* component onto the form, then set its *Name* property to “OrdSource.”
- 3 Drop a *TTable* component on the form, and set the *Name* property to “OrdTable,” the *DatabaseName* property to “DBDEMOS,” the *TableName* property to “ORDERS.DB,” and the *Active* property to True.
- 4 Create a second *TDataSource* component on the form, then set its *Name* property to “CustSource.”
- 5 Create a second *TTable* component on the form, and set the *Name* property to “CustTable,” the *DatabaseName* property to “DBDEMOS,” the *TableName* property to “CUSTOMER.DB,” and the *Active* property to True.
- 6 Create a *TDBGrid* component and link it to OrdSource so it displays the contents of the ORDERS table.
- 7 Create a *TDBLookupCombo* component, and set its *DataSource* property to “CustNo.” The database lookup combo box is now linked to the CustNo column of the ORDERS table.
- 8 Specify the lookup values of the *TDBLookupCombo* component:
  - Set *LookupSource* to “CustSource” (so it looks up values in the CUSTOMER table).
  - Set *LookupField* to “CustNo” (so it looks up and gets values from the CustNo column).
  - In *LookupDisplay*, type `Company;Addr1` (this displays the corresponding company name and address in the drop-down list).
- 9 Set the *loColLines* and *LoTitles* properties (under the *Option* property) of the *TDBLookupCombo* to True.

At run time, the *TDBLookupCombo* component displays a drop-down list of company names and addresses. If the user selects a new company from the list, the control is assigned the value of the corresponding customer number (CustNo). When the user scrolls off the current order in the database grid, Delphi posts the new customer number and information to the row.

## TDBLookupList

---



*TDBLookupList* is functionally the same as *TDBLookupCombo*, but instead of a drop-down list, it displays a scrollable list of the available choices. When the user selects one at run time, the component is assigned that value. Like *TDBLookupCombo*, the user cannot type in an entry that is not in the list. Here is an example of how a *TDBLookupList* component appears at run time:



**Figure 4.11** TDBLookupList component



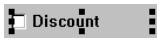
While navigating through a dataset, a *TDBLookupList* component highlights the item in the list that corresponds to the value in the currently selected row. If the current row's value is not defined in the *Items* property, no value is highlighted in the *TDBLookupList* component. Changing the selection changes the underlying value in the database column and is equivalent to typing a value in a *TDBEdit* component.

## TDBCheckBox



*TDBCheckBox* is a data-aware version of the Standard *TCheckBox* component. It can be used to set the values of fields in a dataset. For example, a customer invoice form might have a check box control that when checked, specifies that the customer is entitled to a special discount, or when unchecked means that the customer is not entitled to a discount.

**Figure 4.12** TDBCheckBox component



Like the other data controls, *TDBCheckBox* is attached to a specific field in a dataset through its *DataSource* and *DataField* properties. Use the *Caption* property to display a label for the check box on your form.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to *True*, but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the field for the current record contains values of “True,” “Yes,” or “On,” then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to *False*, but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

If the *DataField* of the check box is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this

case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

A *TDBCheckBox* component is grayed out and disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

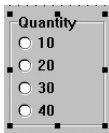
## TDBRadioGroup



*TDBRadioGroup* is a data-aware version of the standard *TRadioGroup* component. It lets you set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group consists of one button for each value a field can accept.

*TDBRadioGroup* is attached to a specific field in a dataset through its *DataSource* and *DataField* properties. A radio button for each string value entered in the *Items* property is displayed on the form, and the string itself is displayed as a label to the right of the button.

**Figure 4.13** A *TDBRadioGroup* component



For the current record, if the field associated with a radio group matches one of the strings in the *Items* or property, that radio button is selected. For example, if three strings, “Red,” “Yellow,” and “Blue,” are listed for *Items*, and the field for the current record contains the value “Blue,” then the third button in the group is selected.

**Note** If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the *Values* property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, to continue the example for the three buttons labeled “Red,” “Yellow,” and “Blue,” if three strings, “Magenta,” “Yellow,” and “Cyan,” are listed for *Values*, and the user selects the first button (labeled “Red”), then Delphi posts “Magenta” to the database.

If strings for *Values* are not provided, the label from a selected radio button (from *Items*) is returned to the database when a record is posted. Users can modify the value of a data field by clicking the appropriate radio button. When the user scrolls off the current row, Delphi posts the value indicated by the radio button string to the database.

# Using SQL in applications

SQL (Structured Query Language) is an industry-standard language for database operations. Delphi enables your application to use SQL syntax directly through the *TQuery* component. Delphi applications can use SQL to access data from:

- Paradox or dBASE tables, using local SQL. The allowable syntax is a sub-set of ANSI-standard SQL and includes basic SELECT, INSERT, UPDATE, and DELETE statements. For more information on local SQL syntax, see Appendix C, “Using local SQL.”
- Databases on the Local InterBase Server. Any statement in InterBase SQL is allowed. For information on syntax and limitations, see the *InterBase Language Reference*.
- Databases on remote database servers (Delphi Client/Server only). You must have installed the appropriate SQL Link. Any standard statement in the server’s SQL is allowed. For information on SQL syntax and limitations, see your server documentation.

Delphi also supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). For more information, see “Creating heterogenous queries” on page 124.

## Using TQuery



*TQuery* is a dataset component, and shares many characteristics with *TTable*, as described in Chapter 3, “Using data access components and tools.” In addition, *TQuery* enables Delphi applications to issue SQL statements to a database engine (either the BDE or a server SQL engine).

The SQL statements can be either static or dynamic, that is, they can be set at design time or include parameters that vary at run time.

## When to use TQuery

---

For simple database operations, *TTable* is often sufficient and provides portable database access through the BDE. However, *TQuery* provides additional capabilities that *TTable* does not. Use *TQuery* for:

- Multi-table queries (joins).
- Complex queries that require sub-SELECTs.
- Operations that require explicit SQL syntax.

*TTable* does not use SQL syntax; *TQuery* uses SQL, which provides powerful relational capabilities but may increase an application's overall complexity. Also, use of non-standard (server-specific) SQL syntax may decrease an application's portability among servers; for more information, see Chapter 6, "Building a client/server application."

## How to use TQuery

---

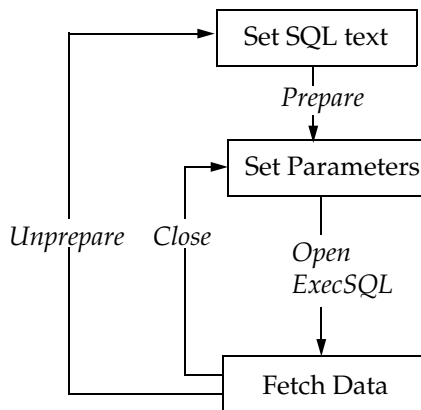
To access a database, set the *DatabaseName* property to a defined BDE alias, a directory path for desktop database files, or a file name for a server database. If the application has a *TDatabase* component, *DatabaseName* can also be set to a local alias that it defines. For more information, see "Using the *TDatabase* component" in Chapter 6, "Building a client/server application."

To issue SQL statements with a *TQuery* component:

- Assign the *TQuery* component's *SQL* property the text of the SQL statement. You can do this:
  - At design time, by editing the *TQuery*'s *SQL* property in the Object Inspector, choosing the *SQL* property, and entering the SQL statements in the String List Editor dialog box. With Delphi Client/Server, you can also use the Visual Query Builder to construct SQL syntax.
  - At run time, by closing any current query with *Close*, clearing the *SQL* property with *Clear*, and then specifying the SQL text with the *Add* method.
- Execute the statement with the *TQuery* component's *Open* or *ExecSQL* method. Use *Open* for SELECT statements. Use *ExecSQL* for all other SQL statements. The differences between *Open* and *ExecSQL* are discussed in a subsequent section.
- To use a dynamic SQL statement, use the *Prepare* method, provide parameters and then call *Open* or *ExecSQL*. *Prepare* is not required, but will improve performance for dynamic queries executed multiple times.

The following diagram illustrates the lifetime of a *TQuery* component and the methods used to work with it:

**Figure 5.1** TQuery methods and flow



**Note** *Prepare* applies only to dynamic queries. It is not required, but is recommended in most cases. For more information, see “Dynamic SQL statements” on page 121.

## The SQL property

---

The *SQL* property contains the text of the SQL statement to be executed by a Query component. This property is of type *TStrings*, which means that it is a series of strings in a list. The list acts very much as if it were an array, but it is actually a special class with unique capabilities. For more information on *TStrings*, see the online VCL reference.

A Query component can execute two kinds of SQL statements:

- Static SQL statements
- Dynamic SQL statements

A *static* SQL statement is fixed at design time and does not contain any parameters or variables. For example, this statement is a static SQL statement:

```
SELECT * FROM CUSTOMER WHERE CUST_NO = 1234
```

A *dynamic* SQL statement, also called a *parameterized* statement, includes parameters for column or table names. For example, this is a dynamic SQL statement:

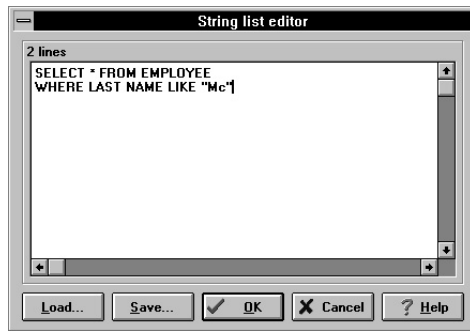
```
SELECT * FROM CUSTOMER WHERE CUST_NO = :Number
```

The variable *Number*, indicated by the leading colon, is a parameter which must be provided at run time and may vary each time the statement is executed.

## Creating the query text

You can enter the SQL text for a *TQuery* at design time by double-clicking on the *SQL* property in the Object Inspector, or choosing the ellipsis button. The String List Editor opens, enabling you to enter an SQL statement.

**Figure 5.2** Editing SQL statements in the String List Editor



Choose OK to assign the text you enter to the *SQL* property of the query. Choose Load to include text from a file or Save to save the text to a file.

To specify SQL text at run time, an application should first close the query with *Close* and clear the *SQL* property with *Clear*. For example,

```
Query1.Close; {This closes the query}
Query1.SQL.Clear; {This clears the contents of the SQL property}
```

It is always safe to call *Close*—if the query is already closed, the call will have no effect. Use the *SQL* property’s *Add* method to add the SQL statements to it. For example,

```
Query1.SQL.Add('SELECT * FROM COUNTRY');
Query1.SQL.Add('WHERE NAME = 'ARGENTINA''');
```

An application should always call *Clear* before specifying an SQL statement. Otherwise, *Add* will simply append the statements to the existing one.

**Note** The *SQL* property may contain only one complete SQL statement at a time. In general, multiple statements are not allowed. Some servers support multiple statement “batch” syntax; if the server supports this, then such statements are allowed.

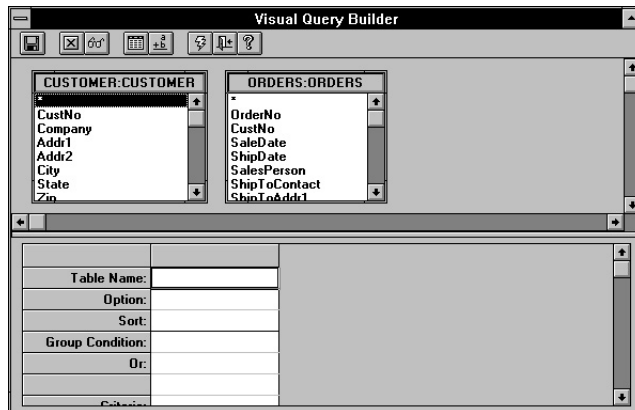
You can also use the *LoadFromFile* method to assign the text in an SQL script file to the *SQL* property. For example,

```
Query1.SQL.LoadFromFile('C:\MYQUERY.TXT');
```

## Using the Visual Query Builder

Delphi Client/Server includes a Visual Query Builder that enables you to construct SQL *SELECT* statements visually. To invoke the Visual Query Builder, right click on a *TQuery* component and select Run Visual Query Builder. A dialog box prompts you to select the database to work with; select the desired database and choose OK. Another dialog box will prompt you to enter the tables you want to query; select the desired tables, choosing Add after each, and then choose Close. The Visual Query Builder window will then become active with the select tables.

**Figure 5.3** Working in the Visual Query Builder



For information on how to use the Visual Query Builder, refer to its online Help. After you have constructed a query and exited the Visual Query Builder, the SQL statement you constructed will be entered in the *SQL* property of the selected *TQuery* component.

## Executing a query

At design time, you can execute a query by changing its *Active* property in the Object Inspector to *True*. The results of the query will be displayed in any data controls connected to the Query component (through a data source).

At run time, an application can execute a query with either the *Open* or the *ExecSQL* methods. Use *Open* for SQL statements that return a result set (SELECT statements). Use *ExecSQL* for all other SQL statements (INSERT, UPDATE, DELETE, and so on). For example,

```
Query1.Open; {Returns a result set}
```

If the SQL statement does not return a cursor and a result set from the database, use *ExecSQL* instead of *Open*. For example,

```
Query1.ExecSQL; {Does not return a result set}
```

If you don't know at design time whether a query will return a result set, use a **try...except** block with *Open* in the **try** part and *ExecSQL* in the **except** part.

## The UniDirectional property

Use the *UniDirectional* property to optimize access to a database table through a *TQuery* component. If you set *UniDirectional* to *True*, you can iterate through a table more quickly, but you will only be able to move in a forward direction. *UniDirectional* is *False* by default.

## Getting a live result set

A *TTable* component always returns a live result set to an application. That is, the user sees the data “live” from the database, and can make changes to it directly through data controls. A *TQuery* can return two kinds of result sets:

- “Live” result sets: As with *TTable* components, users can edit data in the result set with data controls. The changes are sent to the database when a *Post* occurs, or when the user tabs off a control, as described in Chapter 4, “Using Data Controls.”
- “Read only” result sets: Users cannot edit data in the result set with data controls.

By default, a query always returns a read-only result set. To get a live result set, an application must request it by setting the *RequestLive* property of *TQuery* to True. However, for the BDE to be able to return a live result set, the SELECT syntax of the query must conform to the guidelines given below. If an application requests a live result set, but the syntax does not conform to the requirements, the BDE returns a read-only result set (for local SQL) or an error return code (for passthrough SQL). If a query returns a live result set, Delphi will set the *CanModify* property to True.

**Table 5.1** Types of query result sets

<b>RequestLive</b>	<b>CanModify</b>	<b>Type of result set</b>
False	False	Read-only result set
True—SELECT syntax meets requirements	True	Live result set
True—SELECT syntax does not meet requirements	False	Read-only result set

If an application needs to update the data in a read-only result set, it must use a separate *TQuery* to construct an UPDATE statement. By setting the parameters of the update query based on the data retrieved in the first query, the application can perform the desired update operation.

### Syntax requirements for live result sets

To return a live result set, a query must have *RequestLive* set to True. The SQL syntax must conform to that of Local SQL, as described in Appendix C, “Using local SQL.” Additionally, the syntax must meet the requirements described below.

A query of a Paradox or dBASE table can return a live result set if it:

- Involves only a single table.
- Does not have an ORDER BY clause.
- Does not use aggregates such as SUM or AVG.
- Does not use calculated fields in the SELECT list.
- The WHERE clause may consist only of comparisons of column names to scalar constants. The comparison operators may be LIKE, >, <, >=, and <=. The clause may contain any number of such comparisons linked by AND or OR operators.

A query of a server table using passthrough SQL can return a live result set if it:

- Involves a single table.



- Does not have an ORDER BY clause.
- Does not use aggregates such as SUM or AVG.

In addition, if the table is on a Sybase server, it must have a unique index.

## Dynamic SQL statements

---

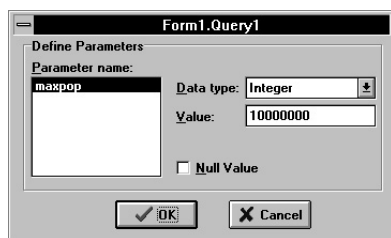
A dynamic SQL statement (also called a parameterized query) contains parameters that can vary at run time.

### Supplying values to parameters

---

At design time, you can supply values to parameters with the Parameters Editor. Invoke the Parameters Editor by selecting a *TQuery* component, right-clicking the mouse, and then selecting Parameters Editor. The Parameters Editor opens.

**Figure 5.4** Parameters Editor



Select the desired data type for the parameter in the Data Type combo box. Enter a value in the Value text field or select Null Value to set the parameter's value to null. When you click OK, the query will be prepared and values will be bound to the parameters. Then, when you set the query's *Active* property to True, the results of the SQL query with the specified parameter values will be shown in any data controls connected to the query.

At run time, an application can supply values to parameters with the following *TQuery* properties:

- The *Params* property, using the order that the parameters appear in the SQL statement.
- The *ParamByName* method, using the parameter names specified in the SQL statement.
- The *DataSource* property to set values from another dataset for columns that match the names of parameters that have no values.

### Preparing a query

---

The *Prepare* method sends a parameterized query to the database engine for parsing and optimization. A call to *Prepare* is not required to use a parameterized query. However, it is strongly recommended, since it will improve performance for dynamic queries that

will be executed more than once. If a query is not explicitly prepared, each time it is executed, Delphi automatically prepares it.

*Prepare* is a Boolean property of *TQuery* that indicates if a query has been prepared. The Parameters Editor automatically prepares a query when you use it to set parameter values at design time.

If a query has been executed, an application must call *Close* before calling *Prepare* again. Generally, an application should call *Prepare* once—for example, in the *OnCreate* event of the form—then set parameters using the *Params* property, and finally call *Open* or *ExecSQL* to execute the query. Each time the query is to be executed with different parameter values, an application must call *Close*, set the parameter values, and then execute the query with *Open* or *ExecSQL*.

Preparing a query consumes some database resources, so it is good practice for an application to unprepare a query once it is done using it. The *UnPrepare* method unprepares a query. When you change the text of a query at run time, Delphi automatically closes and unprepares the query.

## Using the Params property

---

When you enter a query, Delphi creates a *Params* array for the parameters of a dynamic SQL statement. *Params* is a zero-based array of *TParam* objects with an element for each parameter in the query; that is, the first parameter is *Params[0]*, the second *Params[1]*, and so on.

For example, suppose a *TQuery* component named *Query2* has the following statement for its *SQL* property:

```
INSERT
  INTO COUNTRY (NAME, CAPITAL, POPULATION)
  VALUES (:Name, :Capital, :Population)
```

An application could use *Params* to specify the values of the parameters as follows:

```
Query2.Params[0].AsString := 'Lichtenstein';
Query2.Params[1].AsString := 'Vaduz';
Query2.Params[2].AsInteger := 420000;
```

These statements would bind the value “Lichtenstein” to the *:Name* parameter, “Vaduz” to the *:Capital* parameter, and 420000 to the *:Population* parameter.

## Using the ParamByName method

---

*ParamByName* is a function that enables an application to assign values to parameters based on their names. Instead of providing the ordinal location of the parameter, you must supply its name.

For example, an application could use *ParamByName* could specify values for the parameters in the preceding example as follows:

```
Query2.ParamByName('Name').AsString := 'Lichtenstein';
Query2.ParamByName('Capital').AsString := 'Vaduz';
```

```
Query2.ParamByName('Population').AsInteger := 420000;
```

These statements would have the same effect as the three previous statements that used the *Params* array directly.

## Using the DataSource property

For parameters of a query not bound to values at design time, Delphi will check the query's *DataSource* property. This property specifies the name of a *TDataSource* component. If *DataSource* is set, and the unbound parameter names match any column names in the specified *DataSource*, Delphi binds the current values of those fields to the corresponding parameters. This capability enables applications to have linked queries.

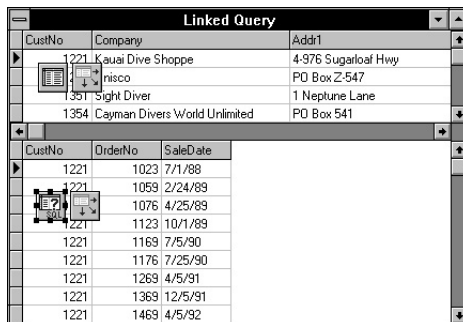
The LINKQRY sample application illustrates the use of the *DataSource* property to link a query in a master-detail form. The form contains a *TQuery* component (named *Orders*) with the following in its *SQL* property:

```
SELECT Orders.CustNo, Orders.OrderNo, Orders.SaleDate
FROM Orders
WHERE Orders.CustNo = :CustNo
```

As illustrated below, the form also contains:

- *ATDataSource* named *OrdersSource*, linked to *Orders* by its *DataSet* property.
- *ATTable* component (named *Cust*).
- *ATDataSource* named *CustSource* linked to *Cust*.
- Two data grids; one linked to *CustSource* and the other to *OrdersSource*.

**Figure 5.5** Form with linked queries



CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1221	nisco	PO Box Z-547
1331	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541

CustNo	OrderNo	SaleDate
1221	1023	7/1/88
1221	1059	2/24/89
1221	1076	4/25/89
1221	1123	10/1/89
1221	1169	7/5/90
1221	1176	7/25/90
1221	1269	4/5/91
1221	1369	12/5/91
1221	1469	4/5/92

*Orders'* *DataSource* property is set to *CustSource*. Because the parameter *:CustNo* does not have any value assigned to it, at run time Delphi will try to match it with a column name in *CustSource*, which gets its data from the Customer table through *Cust*. Because there is a *CustNo* column in *Cust*, the current value of *CustNo* in the *Cust* table is assigned to the parameter, and the two data grids are linked in a master-detail relationship. Each time the *Cust* table moves to a different row, the *Orders* query automatically re-executes to retrieve all the orders for the current customer.

## Dynamic SQL example

---

Here's a simple form that uses a dynamic query and provides substitution parameters programmatically:

- Start a new project and place the required controls on the form. For this example, place an edit control, Edit1, a button, Button1, and a data-aware grid control, DBGrid1, on the form. Now create a *TQuery* component, Query1, and a data source, DataSource1, and set the *DataSource* property of DBGrid1 to "DataSource1" and the *Dataset* property of DataSource1 to "Query1."

Set the *DatabaseName* property of Query1 to DBDEMOS. Double-click on its SQL property in the Object Inspector and enter the following SQL statement. Remember to precede the substitution parameter with a colon:

```
SELECT * FROM country WHERE name LIKE :CountryName
```

- Prepare the query in the *OnCreate* event of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Query1.Prepare;
end;
```

- Provide parameters in response to some event. In this example, double-click on Button1 to edit the *OnClick* event and use the contents of Edit1.Text as a substitution parameter:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Query1.Close;
    Query1.Params[0].AsString := Edit1.Text;
    Query1.Open;
end;
```

## Creating heterogenous queries

---

Some applications may require queries of tables in more than one database. Such queries are called *heterogenous queries*, and are not supported by standard SQL. A heterogenous query may join tables on different servers, and even different types of servers. For example, a heterogeneous query might involve a table in a Oracle database, a table in a Sybase database, and a local dBASE table.

Delphi supports heterogeneous queries, as long as the query syntax conforms to the requirements of local SQL, as described in Appendix C, "Using local SQL."

To perform a heterogeneous query, you must define a BDE standard alias that references a local directory, and use the alias for the *DatabaseName* of the query component. You must also define BDE aliases for each of the databases being queried. In the query text, precede each table name with the alias for its database.

You can define BDE aliases with the BDE Configuration Utility, described in Appendix B, “Using the BDE configuration utility.” For example, suppose you define an alias called *Oracle1* for an Oracle database that has a CUSTOMER table, and *Sybase1* for a Sybase database that has an ORDERS table. A simple query against these two tables would be

```
SELECT CUSTOMER.CUSTNO, ORDERS. ORDERNO  
FROM :Oracle1:CUSTOMER, :Sybase1:ORDERS
```



# Building a client/server application

Delphi Client/Server enables you to develop applications that can access remote SQL servers such as Oracle, Sybase, Informix, and InterBase, as well as local Paradox and dBASE databases. A *remote* server is one that is physically removed from the client machine on which the Delphi application runs. The server and client must be connected by a network. Delphi also includes the Local InterBase Server, a full-featured SQL server that runs on Microsoft Windows.

There are a number of issues that are particularly important when developing a client/server application:

**Portability versus optimization:** Will the application use any server-specific SQL syntax? To what degree will the database be optimized for a particular server?

**Transactions:** What kind of transaction control will the application require?

**Server features:** Will the application require the use of server features such as stored procedures? How will these be surfaced?

**Connectivity:** What communication protocol will the application use? Does the application need to be deployed to support multiple communication protocols?

**Deployment:** What executables, libraries, and other files does the application require and how are these delivered to the end user?

## Portability versus optimization

---

In a client/server system, the database running on the server and the application running on the client define the overall system, referred to as the *database/application*. While these two elements are often designed separately and considered distinct, they must be integrated to build a successful client/server application. One of the important considerations is portability versus optimization.

*Portability* refers to the ease with which an database/application can run on different servers. *Optimization* refers to the extent to which an application takes advantage of the special features of a particular system.

Client portability is not an issue with Delphi, because Delphi applications will run on any 16-bit or 32-bit Windows platform. However, server portability and communications portability can be considerations.

Because Delphi applications use the Borland Database Engine, they can be easily integrated with dBASE and Paradox applications (for desktop data sources) and other clients for server data sources.

## Server portability

---

It may be desirable to design an application so that it can be easily ported to different types of servers, either because the end-users require multiple heterogeneous server support, or because the application will be used by different groups of end-users with different types of servers. In designing a client-server application, there is an inherent trade-off between portability and optimization, because making use of server-specific features results in increased application performance but decreased portability.

A Delphi application that uses only *TTable* components for data access will be fully portable among different server types. An application may benefit from improved performance by using *TQuery* components and passthrough SQL, and as long as the SQL syntax is ANSI standard, there will be little loss of portability.

As soon as SQL syntax departs from the ANSI standard, the application will no longer be fully portable. If server portability is a consideration, you must carefully weigh whether the gain in using server-specific syntax is worth the cost in portability. Maintainability of an application may be reduced by optimization for a specific server type, because each server-specific implementation may require separate maintenance.

An application can be further optimized by using server-specific features such as stored procedures. However, this will usually require server-specific implementation in the database, and perhaps the application, depending on how the features are surfaced.

It is also important to consider that servers' transaction processing may differ in subtle yet important ways. This and other distinctions among SQL servers may complicate portability. Before attempting to create a portable database/application, you should build an application that runs reliably against one type of server database. In some cases, it may be necessary to build the application separately against each of the target server types.

## Client/server communication portability

---

Depending on the application requirements, it may be necessary to support multiple communication protocols, such as TCP/IP and Novell SPX. Providing for multiple communication protocols is simply a matter of ensuring that the client platforms have the proper communication software installed. This portability issue does not typically surface until the deployment phase, but it should be addressed in the implementation



phase to ensure that the initial test deployment packages include the proper client communication software.

## Using the Local InterBase Server

---

The Local InterBase Server (LIBS) is a version of the Borland InterBase server that runs on Windows 3.1. It provides all of the features of a SQL server for local, single-user operation. For more information on the Local Interbase Server, see the *Local InterBase Server User's Guide*. The Local Interbase Server can be used in primarily two ways for client/server development:

- As a local environment for developing a client/server application, regardless of the server to be used.
- As an intermediate step in upsizing, between the desktop and server, providing a local SQL engine for development of SQL-specific features. For more information on upsizing, see "Upsizing" on page 140.
- As a local database engine for deployment of standalone desktop SQL applications.

### Building an application to access any server

---

If you are developing a client/server application, you can use the Local InterBase Server for local development, even if the production database will run on some other server type.

If the production database already exists on a database server, you can use the Local InterBase Server as follows:

- Use your server's tools to create an SQL data definition script for the database. Remove any non-standard SQL syntax that will not work with InterBase. Generally, this means removing stored procedure definitions and other advanced features and mapping data types to InterBase data types.
- Use Windows ISQL to create a local InterBase database then execute the SQL script to define the database. For information on using Windows ISQL, see the *Local InterBase Server User's Guide*.
- Populate the database with representative data using the techniques described in "Upsizing."
- Create a Delphi database application that uses the Local InterBase Server database as a data source.
- Once the application has been sufficiently tested against the development database, change the *TDataSource* to use an alias that points to the target server. It is recommended to first test the application against a duplicate of the production database to surface server-specific features such as stored procedures.

If the production database does not already exist, you can first define a prototype database on the LIBS, and develop the application locally. Simultaneously, you can

define a congruent database on the target server. Finally, you can redirect the application to access the database on the target server.

## Building an application to access InterBase

---

To build an application that accesses a production database that already exists on an InterBase workgroup server, follow this procedure:

- Use the InterBase database backup utility to create a backup of the database, using the “Backup Metadata Only” option. This will save the structure of the database, but not the data (which may be huge). For information on how to do this, see the *Local InterBase Server User’s Guide*.
- Restore the database to the Local InterBase Server. This will be the development platform.
- Populate the database with a modest amount of representative data. Your application will access this data during the development and testing process. Because it is “dummy” data that is not connected to the production database, there is no chance that the production database will be corrupted. Your application can access any server features present in the production database, including stored procedures.
- Once the application has been sufficiently tested against the development database, change the *TDataSource* to use an alias that refers to the production server. It is recommended to first test the application against a duplicate of the production database on the server.

## Using InterBase in upsizing

---

One upsizing path is to use the Local Interbase Server (LIBS) as an intermediate data source. This enables you to address all the SQL development issues separately from connectivity and client/server issues. The steps include

- Defining the database on the Local InterBase Server, using the techniques described in “Upsizing” on page 140.
- Developing the application against the LIBS database.
- Migrating the LIBS database to the target server.
- Redirecting the application to access the target server.

## Connecting to a database server

---

Borland SQL Links for Windows enables a Delphi application to connect through the BDE to a remote database server. SQL Links drivers provide connections to Oracle, Sybase, Informix, Microsoft SQL Server, and InterBase.

Through the BDE Configuration Utility, you can set up an alias for each data source to which your application needs to connect. These aliases then become available to choose as the value of the *DatabaseName* property of *TTable* and *TQuery* components. For more

information on the BDE Configuration Utility, see Appendix B, “Using the BDE configuration utility.”

## Connectivity

---

Delphi client applications can use any network protocol (such as TCP/IP or Novell SPX/IPX) supported by the server, as long as both the server and the client machines have the proper communication software installed. You must configure the SQL Link driver for the desired protocol. For more information, see the *SQL Links for Windows User's Guide*.

Establishing an initial connection between client and server can often be problematic, especially when using TCP/IP, because there are a number of critical factors that must all be in place before a connection can be established.

### Using TCP/IP

TCP/IP is a widely-used communication protocol that enables applications to connect to many different database servers. When using TCP/IP, you must ensure:

- The TCP/IP communication software and the proper Winsock driver are installed on the client.
- The server's IP address is registered in the client's HOSTS file or that Directory Network Services (DNS) is properly configured.
- The server's port number is entered in the client's SERVICES file.
- The application is searching the proper directory paths for the DLLs it needs. Check the PATH statement in AUTOEXEC.BAT.

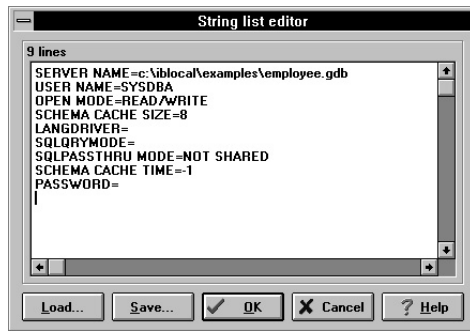
For more information, see the *SQL Links for Windows User's Guide* and your server documentation.

## Connection parameters

---

The *Params* property of a connected *TDatabase* object contains a *TString* list of all the SQL Link parameters required to connect to a server of the specified type. You can edit these parameters by clicking on the ellipsis button to the right of the *Params* property in the Object Inspector. The String List Editor opens with the parameters. For example, here are the parameters for connection to an InterBase server:

**Figure 6.1** InterBase parameters in the String List Editor



You can modify these parameters and add others as needed to customize the connection performed by the application. For more information, see the *SQL Links for Windows User's Guide*.

## Using ODBC

---

A Delphi application can access ODBC data sources such as DB2, Btrieve, or Microsoft Access through the Borland Database Engine (BDE). To do this, you must set up an ODBC driver connection using the BDE Configuration Utility. An ODBC driver connection requires:

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- A BDE alias, established with the BDE Configuration Utility or with Delphi.

The BDE configuration setting AUTO ODBC (on the System page) enables an alias to automatically configure for use of ODBC. When AUTO ODBC is True, datasource and driver information will automatically be imported from the ODBC.INI file.

For more information, see the online help for the BDE Configuration Utility.

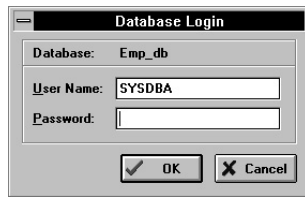
## Handling server security

---

Most database servers include security features to limit database access. Generally, the server will require a user name and password login before a user can access a database. If a server requires a login, then Delphi will prompt you at design time when you attempt to connect to a database on the server (for example, when you specify a *TableName* for a *TTable* component).

By default, a Delphi application opens the standard Login dialog box, whenever an application opens a connection to a database server. If a connection has already been established, the Login dialog box does not appear.

**Figure 6.2** Database Login dialog box



A Delphi application can handle server login several different ways:

- If the *LoginPrompt* property of a *TDatabase* component is True (the default), the standard Delphi Login dialog box will be opened when the application attempts to establish a database connection.
- By setting *LoginPrompt* to False, and including the USERNAME and PASSWORD parameters in the *Params* property of the *TDatabase* component. For example,

```
USERNAME = SYSDBA  
PASSWORD = masterkey
```

This is generally not recommended since it compromises server security.

- Use the *OnLogin* event of *TDatabase* to set login parameters. The *OnLogin* event gets a copy of the *TDatabase*'s login parameters array. Use the *Values* property to change these parameters:

```
LoginParams.Values['SERVER NAME'] := 'MYSERVERNAME';  
LoginParams.Values['USER NAME'] := 'MYUSERNAME';  
LoginParams.Values['PASSWORD'] := 'MYPASSWORD';
```

When control returns from your *DatabaseLogin* event handler, these parameters will be used to establish a connection.

## Using the TDatabase component



The *TDatabase* component is not required for database access, but it provides additional control over factors that are important for client/server applications, including the ability to:

- Create a persistent database connection.
- Customize database server logins.
- Create BDE aliases local to an application.
- Control transactions and specify transaction isolation level.

The *DataSets* property of *TDatabase* is an array of pointers to the active datasets in the *TDatabase*. The *DatasetsCount* property is an integer that specifies the number of active datasets.

## The Connected property

The *Connected* property of a *TDatabase* component specifies whether an application remains connected to a database server even when no tables are open. If an application will be opening and closing several tables in a single database, it will be more efficient to set the Database object's *Connected* property to True. That way, the application will remain connected to the database even if it does not have any tables open. It can then open and close tables repeatedly without incurring the overhead of connecting to the database each time.

A *TDatabase* object's *Connected* property can be overridden by the application-global *TSession* object. This object has a Boolean *KeepConnections* property that specifies whether to maintain database connections even if no tables are open. If *KeepConnections* is False, a *TDatabase* object's *Connected* property determines if connections are maintained when no *TTables* or *TQueries* are open. If *KeepConnections* is True (the default), database connections are always maintained. Specifically,

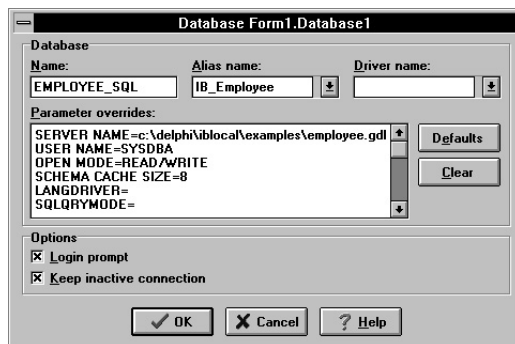
- Database connections will be maintained until the application exits or until the Session's *DropConnections* method is called .
- Setting a *TDatabase* object's *Connected* property to False will have no effect.

## Creating application-specific aliases

The *TDatabase* object enables you to create BDE aliases specific to an application. To name the alias, enter a name in the *DatabaseName* property. Any dataset components can then use the local alias by using the specified *DatabaseName*.

To customize the parameters for a local alias, double-click on the *TDatabase* component. The Database Properties Editor opens:

**Figure 6.3** Database Properties Editor



This tool enables you to customize application-specific aliases local based on existing aliases.

The three text fields at the top of the dialog box correspond to the *DatabaseName*, *AliasName*, and *DriverName* properties.

- *DatabaseName* is the name of the database connection that can be used by dataset components instead of a BDE alias, directory path, or database name. In other words, this is the name of the local alias defined by the component that will show up in the *DatabaseName* drop-down list of dataset components.
- *AliasName* is the name of an existing BDE alias configured with the BDE Configuration Utility. This is where the component gets its default parameter settings. This property will be cleared if *DriverName* is set.
- *DriverName* is the name of a BDE driver, such as STANDARD (for dBASE and Paradox), ORACLE, SYBASE, INFORMIX or INTERBASE. This property will be cleared if *AliasName* is set, because an *AliasName* specifies a driver type.

Choose the Defaults button to retrieve the default parameters for the selected alias. The values will be displayed in the Parameters list box. Any changes you make to the defaults are used instead of the default values for any database connection in the application that uses that *DatabaseName*.

The check boxes labeled “Loginprompt” and “Keep inactive connection” correspond to the *LoginPrompt* and *KeepConnections* properties of the *TDatabase* component.

## Understanding transaction control

---

SQL database servers handle requests in logical units of work called *transactions*. A transaction is a group of SQL statements that must all be performed successfully before the server will finalize (or *commit*) changes to the database. Either all the statements will succeed, or all will fail.

Transaction processing ensures database consistency even if there are hardware failures, and maintains the integrity of data while allowing concurrent multiuser access. For example, an application might update the ORDERS table to indicate that an order for a certain item was taken, and then update the INVENTORY table to reflect the reduction in inventory available. If there were a hardware failure after the first update but before the second, the database would be in an inconsistent state, since the inventory would not reflect the order entered. Under transaction control, both statements would be committed at the same time. Transaction control becomes even more important in a multiuser application.

In SQL, transactions are explicitly ended with a command to either accept or discard the actions performed. The COMMIT statement permanently commits the transaction, making changes visible to all users. The ROLLBACK statement undoes all changes made to the database in the transaction. Different database servers implement transaction processing differently. For the specifics of how your server handles transaction processing, refer to your server documentation.

## Handling transactions in applications

---

Delphi applications can control transactions:

- **Implicitly:** Delphi will automatically start and commit transactions as needed when an application calls the *Post* method (explicitly or implicitly in another method).

- **Explicitly:** depending on the level of control the application requires, either with
  - The *StartTransaction*, *Commit*, and *Rollback* methods of *TDatabase*. This is the recommended approach.
  - Passthrough SQL in a *TQuery* component. The application must use server-specific SQL transaction control statements. You must understand how your server performs transaction handling.

Transaction control statements are only meaningful when the database is on an SQL server. The *StartTrans*, *Commit*, and *Rollback* methods will raise an exception if the underlying database is Paradox or dBASE.

## Implicit transaction control

Delphi applications that use only the built-in methods can rely on implicit transaction control. Any Delphi operations on a server database that are not under explicit transaction control will be under implicit control. Delphi will commit each individual write operation (*Post*, *AppendRecord*, and so on) as a separate transaction, so it will commit database changes on a row-by-row basis. This minimizes update conflicts, but can lead to heavy network traffic.

When using implicit transaction control, keep the `SQLPASSTHRUMODE` setting at `SHARED AUTOCOMMIT`, the default. For more information, see “Setting the SQL passthrough mode.”

Implicit transaction control happens automatically, but does not provide much flexibility. If an application needs multi-row transactions or passthrough SQL, then it should use explicit transaction control.

## Explicit transaction control

The recommended approach for transaction control is to use the methods of *TDatabase*, because this will result in clearer code and a more portable application. The methods for transaction control are

- *StartTransaction*: Begins a transaction at the isolation level specified by the *TransIsolation* property of *TDatabase*. If a transaction is currently active, Delphi will raise an exception.
- *Commit*: Commits the currently active transaction on the database. If no transaction is active, Delphi will raise an exception.
- *Rollback*: Rolls back the currently active transaction. All changes to the database since the last *Commit* will be undone.

Some applications may require additional server-specific transaction control features. In this case, use a *TQuery* component with passthrough SQL statements for transaction control. Ensure that `SQLPASSTHRUMODE` is set to `NOT SHARED` so that the passthrough SQL does not affect other transactions.

## Setting the SQL passthrough mode

`SQLPASSTHRUMODE` in the BDE Configuration utility determines if passthrough SQL and standard BDE calls share the same database connection. For transactions, this



translates to whether passthrough transactions and other transactions “know” about each other. Only applications that use passthrough SQL need be concerned with `SQLPASSTHRUMODE`.

`SQLPASSTHRUMODE` can have the following settings:

- `SHARED AUTOCOMMIT` (the default)
- `SHARED NOAUTOCOMMIT`
- `NOT SHARED`

With `SHARED AUTOCOMMIT`, each operation on a single row is committed. This mode most closely approximates desktop database behavior, but is inefficient on SQL servers because it starts and commits a new transaction for each row, resulting in a heavy load of network traffic.

With `SHARED NOAUTOCOMMIT`, the application must explicitly start and commit transactions. This setting can result in conflicts in busy, multiuser environments where many users are updating the same rows.

`NOT SHARED` means that passthrough SQL and Delphi methods use separate database connections.

**Note** To control transactions with passthrough SQL, you must set `SQLPASSTHRU MODE` to `NOT SHARED`. Otherwise, passthrough SQL and Delphi’s methods may interfere with each other, leading to unpredictable results.

## Transaction isolation levels

---

A transaction’s *isolation level* determines how it interacts with other simultaneous transactions accessing the same tables. In particular, the isolation level affects what a transaction reads from the tables being accessed by other transactions.

Some servers enable you to set the transaction isolation level explicitly in passthrough SQL. If not specified, passthrough SQL operations will use a server’s default isolation level. For more information, see your server documentation.

Transactions (both explicit and implicit) using Delphi’s built-in methods will use the *TransIsolation* property of *TDatabase* to specify transaction isolation level. *TransIsolation* can have the following values:

- *tiDirtyRead*: The transaction can read uncommitted changes to the database by other transactions. This is the lowest isolation level.
- *tiReadCommitted*: The transaction can read only committed changes to the database by other transactions. This is the default isolation level.
- *tiRepeatableRead*: The transaction cannot read other transactions’ changes to previously read data. This guarantees that once a transaction reads a records, it will not change if it reads it again. This the highest isolation level.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, then Delphi will use the next highest isolation level. The actual isolation level used by each type of server is shown in

Table 6.1, “Server transaction isolation levels.” For a detailed description of how each isolation level is implemented, see your server documentation.

**Table 6.1** Server transaction isolation levels

<i>TransIsolation setting</i>	Oracle	Sybase and Microsoft SQL servers	Informix	InterBase
Dirty read	Read committed	Read committed	Dirty Read	Read committed
Read committed (Default)	Read committed	Read committed	Read committed	Read committed
Repeatable read	Repeatable read (READ ONLY)	Read committed	Repeatable Read	Repeatable Read

If an application is using ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information, see your ODBC driver documentation.

## Using stored procedures



A stored procedure is a server-based program that can take input parameters and return output parameters to an application. Stored procedures are associated with a database, and are actually part of metadata, like tables or domains. The *TStoredProc* component enables Delphi applications to execute server stored procedures.

The *DatabaseName* property of *TStoredProc* specifies the database in which the stored procedure is defined. This property is the same as for *TTable* and *TQuery*--it can be a BDE alias or an explicit directory path and database name. The *StoredProcName* specifies the name of the stored procedure. A drop-down list will display a list of all procedures defined in the specified database.

A *TStoredProc* can return either a singleton result or a result set with a cursor, if the server supports it.

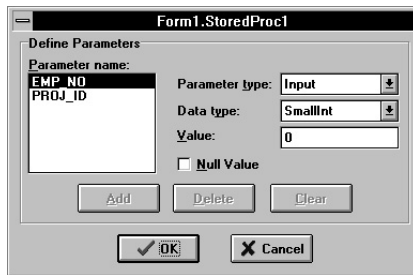
**Note** InterBase “select” procedures are called with the `SELECT` statement as if querying a table. To get output from such procedures, use a *TQuery* component with the appropriate `SELECT` syntax.

## Input and output parameters

A stored procedure has a *Params* array for its input and output parameters similar to a *TQuery* component. The order of the parameters in the *Params* array is determined by the stored procedure definition. An application can set the values of input parameters and get the values of output parameters in the array similar to *TQuery* parameters. You can also use *ParamByName* to access the parameters by name. If you are not sure of the ordering of the input and output parameters for a stored procedure, use the Parameters Editor.

To invoke the Parameters Editor, select the *TStoredProc* component and then right-click the mouse. The following dialog box opens:

**Figure 6.4** TStoredProc Parameters Editor



The Parameters Editor displays the input and output parameters for the procedure. To prepare the stored procedure with the default parameter types and field types, simply choose OK. You can set values of input parameters at design time by choosing the parameter in the Parameters list and entering a value in the Value field. To specify null input parameter values, select the Null value check box. The Parameters Editor is explained in more detail in Chapter 5, “Using SQL in applications.”

**Note** Delphi will attempt to get information on input and output parameters from the server. For some servers (such as Sybase), this information may not be accessible. In such cases, you must enter the names and data types of the input and output parameters in the Parameters Editor at design time.

## Executing a stored procedure

Before an application can execute a stored procedure, you must prepare the stored procedure, which can be done:

- At design time with the Parameters Editor.
- At run time with the *Prepare* method of *TStoredProc*.

To prepare a stored procedure at run time, use the *Prepare* method, before executing it. For example,

```
StoredProc1.Prepare;
```

To execute a prepared stored procedure, use the *ExecProc* method. Values can be assigned to and from a *TStoredProc* component just as for *TQuery* components, by using the *Params* array. For example, the following code could be in a button's *OnClick* event:

```
StoredProc1.Params[0].AsString := Edit1.Text;  
StoredProc1.ExecProc;  
Edit2.Text := StoredProc1.Params[1].AsString;
```

The first parameter, *Params*[0], is an input parameter of type String. It is assigned the text entered by the user in Edit1. Then, assuming StoredProc1 has been prepared at design time with the Parameters Editor, the stored procedure is run with *ExecProc*. Finally, the output parameter, *Params*[1], is displayed by Edit2.

On some servers, stored procedures can return a result set similar to a query. Applications can use data aware controls to display the output of such stored procedures. You do this in the same way as you display output from *TQuery*

components: create a *TDataSource* component and assign its name to a data grid's *DataSource* property.

## Oracle overloaded stored procedures

---

Oracle servers allow overloading of stored procedures; that is, different procedures with the same name. The *Overload* property enables an application to specify the procedure to execute. If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then Delphi will execute the first stored procedure with the overloaded name; if it is two (2), it will execute the second, and so on.

## Upsizing

---

Migrating a desktop application to a client/server application is called *upsizing*. Upsizing is a complex topic and a full treatment of it is beyond the scope of this book. However, this section will address some of the most important aspects of upsizing a Delphi application.

Upsizing has two major facets:

- Upsizing the database from the desktop to the server
- Upsizing the application to address client/server considerations

Upsizing requires a shift in perspective from the desktop world to the client/server world. Desktop databases and SQL server databases are different in many respects. Desktop databases are designed for one user at a time, while servers are designed for multiuser access. Desktop databases are conceptually record-oriented, while server databases are conceptually set-oriented. Desktop databases typically store each table in a separate file, while servers store all the tables in a database together.

Client/server applications must also address some entirely new issues, the most complex of which are connectivity, network usage, and transaction handling.

## Upsizing the database

---

Upsizing a database includes the following steps:

- Defining metadata on the server, based on the existing desktop database structure.
- Migrating the data from the desktop to the server.
- Addressing issues such as:
  - Data type differences
  - Data Security and Integrity
  - Transaction control
  - Data Access Rights
  - Data Validation
  - Locking

Delphi provides two ways to upsize a database:

- Use the Database Desktop utility and choose Tools | Utilities | Copy to copy a table from desktop table to SQL format. For more information, see Appendix A, “Using Database Desktop.”
- Build a Delphi application that uses a *TBatchMove* component. For more information on *TBatchMove*, see Chapter 3, “Using data access components and tools.”

Both of these options will copy table structures and migrate data from the desktop source to the server destination. Depending on the database, it may be necessary to make changes to the tables created by these methods. For example, the datatype mappings may not be exactly as desired.

Additionally, you must add to the database any of the following features if required:

- Integrity constraints (primary and foreign keys)
- Indexes
- Check constraints
- Stored procedures and triggers
- Other server-specific features

Depending on the database, it may be most efficient to define the metadata first by using an SQL script and the server’s data definition tools and then migrate the data using one of the two methods previously mentioned. If you define the table structure manually, then Database Desktop and *TBatchMove* will copy only the data.

## Upsizing the application

---

In principle, a Delphi application designed to access local data can access data on a remote server with few changes to the application itself. If a congruent data source has been defined on an SQL server, you can re-direct the application to access it rather than the local data source, simply by changing the *DatabaseName* property of *TTable* or *TQuery* components in the application.

In practice, however, there are a number of important differences between accessing local and remote data sources. Client/server applications must also address a number of issues that are not relevant to desktop applications.

Any Delphi application can use either *TTable* or *TQuery* for data access. Desktop applications will generally use the *TTable* component. When upsizing to a SQL server, it may be more efficient to use *TQuery* objects instead in some instances. Depending on the specific application, *TQuery* may be preferable if the application will be retrieving a large number of records from database tables.

If the application uses mathematical or aggregate functions, it may be more efficient to perform these functions on the server with stored procedures. The use of stored procedures may be faster because servers are typically more powerful. This also reduces the amount of network traffic required, particularly for functions that process a large number of rows.

For example, an application might need to compute the standard deviation of values of a large number of records. If this function were performed on the client, all the values would have to be retrieved from the server to the client, resulting in a lot of network traffic. If the function were performed by a stored procedure, all the computation would

be performed on the server, so the application would only retrieve the answer from the server.

## Deploying support for remote server access

---

Deployment of general database application is discussed in Chapter 1, "Introduction." In addition to the files required to deploy a desktop database application, deployment of a client/server application requires installation of the appropriate Borland SQL Links. These are not part of the Borland Database Engine, and must be installed separately. They are redistributable, according to the terms of the license agreement.

**Note** For more information on deploying Delphi applications, refer to the file DEPLOY.TXT, installed to the DELPHI\DOC directory by default.

Each server type has a set of files for the SQL Link. In addition, a file used by all the SQL Links is BLROM800.LD, the Roman8 language driver using binary collation sequence.

### Oracle

---

The following files provide the SQL Links interface with Oracle servers. In addition, applications will require Oracle client files for interface to low-level communication protocols such as TCP/IP. Refer to your server documentation.

**Table 6.2** Oracle SQL Link files

File name	Description
SQLD_ORA.DLL	Borland SQL Link Oracle Driver
SQLD_ORA.HLP	Online help file
SQL_ORA.CNF	BDE Configuration File for Oracle Driver
ORA6WIN.DLL	Oracle Version 6.x client-side DLL
ORA7WIN.DLL	Oracle Version 7.x client-side DLL
SQL13WIN.DLL	Oracle client-side DLL
SQLWIN.DLL	Oracle client-side DLL
COREWIN.DLL	Oracle client-side DLL
ORAWE850.LD	Language driver based on DOS code page 850

---

### Sybase and Microsoft SQL servers

---

The following files provide the SQL Links interface with Sybase servers. In addition, applications will require Sybase client files for interface to low-level communication protocols such as TCP/IP. Refer to your server documentation.

**Table 6.3** Sybase SQL Link files

File name	Description
SQLD_SS.DLL	Borland SQL Link Sybase Driver
SQLD_SS.HLP	Borland SQL Link Sybase Driver Help
SQL_SS.CNF	BDE Configuration File for Sybase Driver

**Table 6.3** Sybase SQL Link files (continued)

File name	Description
W3DBLIB.DLL	Sybase/Microsoft SQL Server client-side DLL
DBNMP3.DLL	Sybase/Microsoft SQL Server client-side DLL for Named Pipes
SYDC437.LD	Language driver based on DOS code page 850
SYDC850.LD	Language driver based on DOS code page 437

## Informix

The following files provide the SQL Links interface with Informix servers. In addition, applications will require Informix client files for interface to low-level communication protocols such as TCP/IP. Refer to your server documentation.

**Table 6.4** Informix SQL Link files

File name	Description
SQLD_INF.DLL	Borland SQL Link Informix Driver
SQLD_INF.HLP	Online help file
SQL_INF.CNF	BDE Configuration File for Informix Driver
LDLLSQLW.DLL	Informix client-side DLL
ISAM.IEM	Informix error message file
OS.IEM	Informix error message file
RDS.IEM	Informix error message file
SECURITY.IEM	Informix error message file
SQL.IEM	Informix error message file

## InterBase

The following files provide the SQL Links interface to remote InterBase servers. These files are distinct from those required to access the Local InterBase Server..

**Table 6.5** InterBase SQL Link files

File name	Description
SQLD_IB.DLL	Borland SQL Link InterBase Driver
SQLD_IB.HLP	Borland SQL Link InterBase Driver Help
SQL_IB.CNF	BDE Configuration File for InterBase Driver
CONNECT.EXE	InterBase connection diagnostic tool
CONNECT.HLP	InterBase Windows connection diagnostic help file
GDS.DLL	InterBase API DLL
REMOTE.DLL	InterBase Networking interface DLL
INTERBAS.MSG	InterBase error message file

## TCP/IP Interface

The following files provide InterBase client applications their interface to Winsock 1.1 compliant TCP/IP products.

**Table 6.6** Winsock 1.1 client files

File name	Description
MVWASYNC.EXE	Asynchronous communication module
VSL.INI	TCP/IP transport initialization file
WINSOCK.DLL	Windows Socket DLL
MSOCKLIB.DLL	Maps Windows socket calls to VSL driver

For TCP/IP products that are not Winsock 1.1 compliant, InterBase client applications will require one of the following files. During installation, Delphi will prompt you to select the TCP/IP stack for which to install support. If the deployed application needs to support a different TCP/IP stack, you must copy the corresponding file from the installation disks.

**Table 6.7** Non-Winsock compliant TCP support files

File name	TCP/IP Product
M3OPEN.EXE	3Com 3+Open TCP Microsoft LAN Manager Digital Pathworks for DOS
M3OPEN.DLL	3Com 3+Open TCP Version 2.0
MBW.EXE	Beame & Whiteside TCP/IP
MFTP.EXE	FTP PC/TCP
MHPARPA.DLL	HP ARPA Service for DOS
MNETONE.EXE	Ungermann-Bass Net/One
MNOVLWP.DLL	Novell LAN WorkPlace for DOS
MPATHWAY.DLL	Wollongong Pathway Access for DOS
MPCNFS.EXE	Sun PC NFS
MPCNFS2.EXE	Sun PC NFS v3.5
MPCNFS4.DLL	Sun PC NFS v4.0
MWINTCP.EXE	Wollongong WIN TCP/IP for DOS

## Other communication protocols

The InterBase workgroup server for NetWare supports Novell SPX/IPX protocol. Two client files are required: NWIPXSPX.DLL and NWCALLS.DLL.

The InterBase Workgroup Server for Windows NT supports Microsoft Named Pipes protocol. No additional client files are required to support Named Pipes, but the client machine must have Microsoft LAN Manager or Windows for Workgroups 3.1.1 installed.



## Deploying ReportSmith support

---

To deploy an application that uses a *TReport*, you must include the .RPT files and ReportSmith Runtime in your deployment package. By default, Delphi installs the files required for ReportSmith Runtime in the RS\_RUN directory. These files require two to three megabytes of disk space. For more information about running reports, see *Creating Reports*.

**Note** For more information on deploying ReportSmith reports, refer to the file DEPLOY.TXT, installed to the DELPHI\DOC directory by default.



---

# A

## Using Database Desktop

This appendix describes Database Desktop and provides a synopsis of Database Desktop features. The complete Database Desktop *User's Guide* is available on the Delphi CD-ROM, and all features are described in Database Desktop Help.

### What is Database Desktop?

---

Database Desktop provides an easy way to create, restructure, and query tables to help you develop database applications with Delphi. You can use Database Desktop either as a standalone application on a single computer running Windows or as a multiuser application on a network.

This appendix contains information on using Database Desktop to work with data tables in a variety of formats.

### The Database Desktop window

---

This section discusses the Database Desktop window and its menus.

#### Starting Database Desktop

---



To start Database Desktop, double-click the Database Desktop icon in the Delphi program group or choose File | Run in the Program Manager and run DBD.EXE.

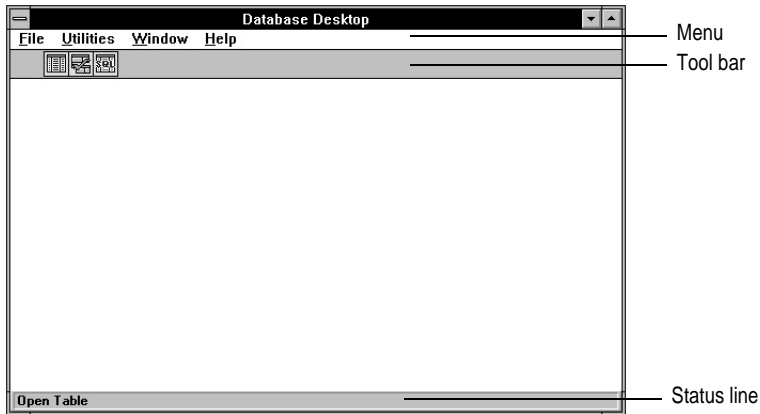
Database Desktop has several command-line options that let you control its configuration. For information on each option and its use, search for “command-line configuration” in the keyword list in Database Desktop Help.

## The Database Desktop window

---

The first time you start Database Desktop, the Database Desktop window opens. All Database Desktop windows are opened in and contained by this window.

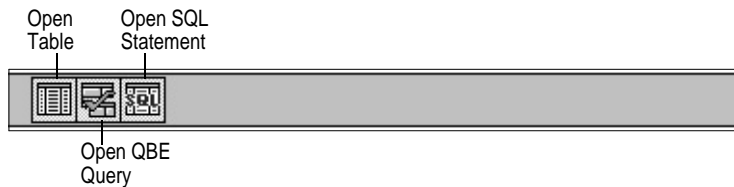
**Figure A.1** The Database Desktop application window



Files you open in Database Desktop appear in their own type of windows. Tables appear in Table windows, queries appear in Query windows, and SQL statements appear in the SQL Editor.

Below the menu is a tool bar. The tool bar changes when the active window changes. The following figure shows the application window tool bar.

**Figure A.2** Application window tool bar



## Managing files

---

In Database Desktop you work with three types of files: QBE queries, .SQL files, and tables. Other files are created automatically by Database Desktop.

For a list of file extensions used by Database Desktop, search for "file-name extensions" in the keyword list in Database Desktop Help, and choose the topic "Types of Files."

### Opening files

To open a QBE query, SQL statement, or table, follow these steps:

- 1 Choose File | Open.
- 2 Choose the type of file to open—QBE query, SQL statement, or table.

### 3 Specify the file to open.

The Select File dialog box appears. For detailed information on the Select File dialog box, search for “Select File dialog box” in the keyword list in Database Desktop Help.

**Note** To access tables stored on a network, you must specify the location of the network control file. You do this by running the BDE Configuration Utility; double-click the BDE Configuration Utility icon in the Delphi program group. See online Help in the BDE Configuration Utility for more details.

### Setting up a working directory

The *working directory* is where Database Desktop looks first for files. The Working Directory setting controls what files are listed in File | Open and File | Save dialog boxes. So, for example, if you want to open C:\DBD\SAMPLES\BOOKORD.DB, make C:\DBD\SAMPLES your working directory so that you see BOOKORD.DB when you choose File | Open | Table.

To specify a working directory, choose File | Working Directory, then type the path to the directory.

**SQL** You cannot set your working directory to an alias on a remote server.

### Setting up a private directory

You should store temporary tables, such as *Answer*, in a private directory so they do not get overwritten by other users or applications. Choose File | Private Directory to establish a private directory.

Files stored in your private directory are listed in File | Open and File | Save dialog boxes, preceded by :PRIV: . Private directory files are visible and available to you, but not to other network users.

### Aliases

You can assign an *alias* as a shorthand for a directory using the Alias Manager dialog box. For example, if you have a collection of tables and queries in one directory (called C:\DBD\PROJECTS\CUSTLIST), you can specify the alias :MYWORK: rather than type the entire path.

Using aliases, you can avoid typing long path names, and you can use the Path list in File | Open and File | Save dialog boxes to list files in any directory for which you have defined an alias.

To create an alias, choose File | Aliases. For information on creating, changing, or removing an alias, search for “aliases” in the keyword list in Database Desktop Help.

## Creating tables

---

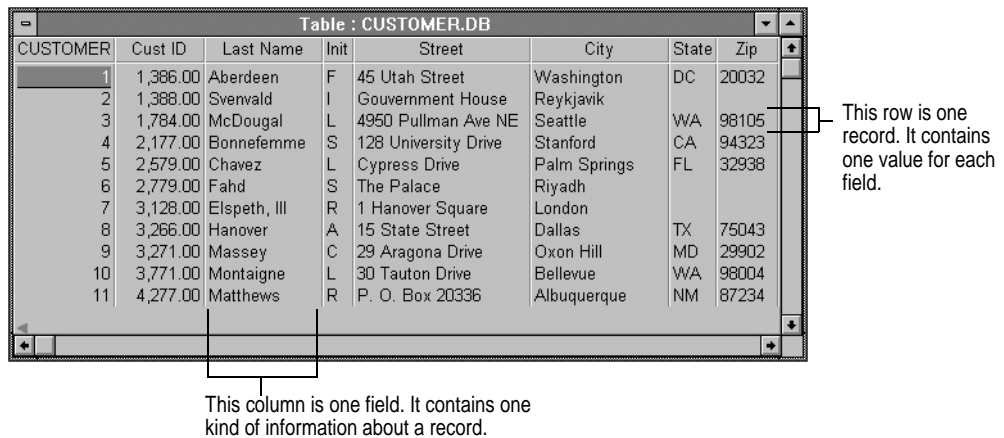
This section describes tables and discusses how to create and restructure Paradox, dBASE, and SQL tables in Database Desktop.

## Understanding tables

A database is an organized collection of information or data. An address book is an example of a database. It organizes data about people into specific categories: names, phone numbers, and addresses.

In a relational database, the data is organized into tables. Each row of a table contains information about a particular item; this is called a *record*. Each column contains one piece of the information that makes up a record; this is called a *field*.

**Figure A.3** A table



CUSTOMER	Cust ID	Last Name	Init	Street	City	State	Zip
1	1,386.00	Aberdeen	F	45 Utah Street	Washington	DC	20032
2	1,386.00	Svenvald	I	Gouvernement House	Reykjavik		
3	1,784.00	McDougal	L	4950 Pullman Ave NE	Seattle	WA	98105
4	2,177.00	Bonnefemme	S	128 University Drive	Stanford	CA	94323
5	2,579.00	Chavez	L	Cypress Drive	Palm Springs	FL	32938
6	2,779.00	Fahd	S	The Palace	Riyadh		
7	3,128.00	Elspeith, III	R	1 Hanover Square	London		
8	3,266.00	Hanover	A	15 State Street	Dallas	TX	75043
9	3,271.00	Massey	C	29 Aragona Drive	Oxon Hill	MD	29902
10	3,771.00	Montaigne	L	30 Tauton Drive	Bellevue	WA	98004
11	4,277.00	Matthews	R	P. O. Box 20336	Albuquerque	NM	87234

## Relational tables

Relational database applications such as dBASE and Paradox give you a way to link tables by comparing values stored in comparable fields in separate tables. The advantage of a relational database is that you can easily extract or combine data from several tables to get exactly the information you need, without changing the structure of the database. Also, a few small and discrete tables are more convenient to use and maintain than one large table.

The sample database files CUSTOMER.DB and BOOKORD.DB are examples of relational tables. These tables can be linked through the fields containing customers' ID numbers (*Cust ID* in the *Customer* table, *Cust* in *Bookord*). When the tables are linked, you can extract information from both tables into one table. For example, you can search for and extract a list of quantities ordered (from the *Bookord* table) and the respective last names (from *Customer*). The results are returned in an *Answer* table (see page 170).

## Planning tables

Planning is the first step in creating a table. Decide what you want the table to contain and how you want to lay it out. For tips on planning tables, search for "planning" in the keyword list in Database Desktop Help, and choose the topic "Planning Tables."

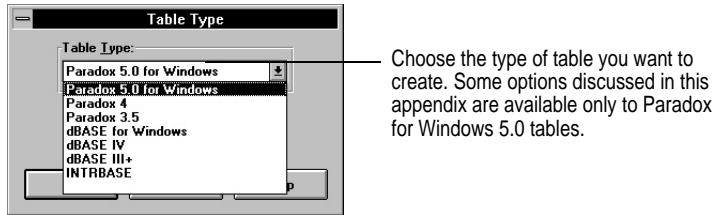
# Creating a new table

To create a new table,

- 1 Choose File | New | Table. Or right-click the Open Table tool bar button, and choose New.

The Table Type dialog box appears.

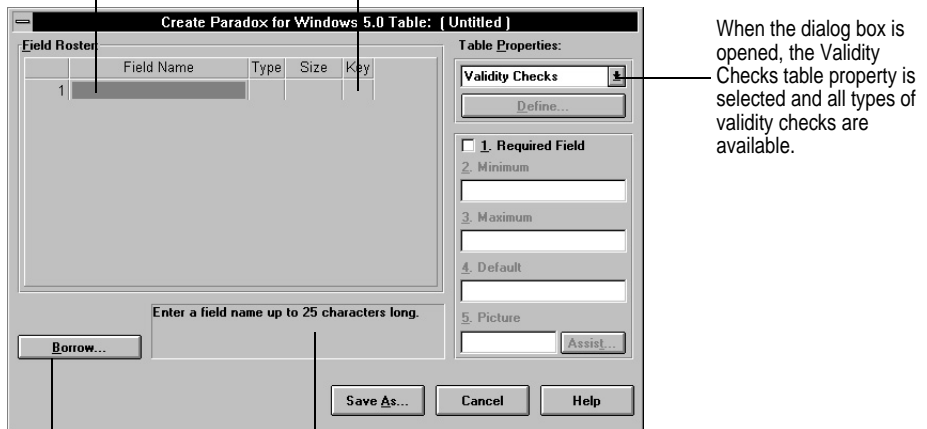
**Figure A.4** Table Type dialog box



- 2 If you want a table type other than Paradox for Windows, click the arrow next to the list box and select from the drop-down list.
- 3 Choose OK. The Create Table dialog box appears. This dialog box may have a slightly different appearance for different table types, but it will function the same.

**Figure A.5** The Create Table dialog box

Enter the field name, type, and size in the Field Roster. Press any key or double-click to key the table.



Choose this to borrow the structure of another table.

The status box gives you guidelines as you create the table.

For a step-by-step description of creating a table, search for “creating tables” in the keyword list in Database Desktop Help, and choose the topic “Creating a New Table.”

## Defining fields

---

Use the Field Roster in the Create Table dialog box (page 151) to define the fields of the new table. You can use the mouse, arrow keys, *Enter*, *Tab*, or *Shift+Tab* to move among the columns. (*Shift+Tab* moves backwards.)

### Field names

Type field names in the Field Name column of the Field Roster.

For information on rules governing field names, search for “field names” in the keyword list in Database Desktop Help, and choose the topic for the type of table you are using (Paradox, dBASE, or SQL).

### Adding, deleting, and rearranging fields

You can add, delete, and rearrange fields in the Field Roster.

For detailed information on changing fields, search for “fields” in the keyword list in Database Desktop Help.”

### Specifying field type

To specify the field type in the Create Table dialog box,

- 1 Select the Type column of the field you want.
- 2 Type the symbol (or name, for SQL tables) for the field type or select from the drop-down list. You can use the list in two ways:
  - Right-click the Type column again and click to select the field type.
  - Press *Spacebar* to see the list, then choose the field type.

For information on field types and sizes, search for “field types” in the keyword list in Database Desktop Help, and choose the topic for the type of table you are using (Paradox, dBASE, or SQL).

## Using indexes

---

The BDE uses indexes to keep track of the location of records in tables. This makes it easy to maintain a sorted order of a table and view like values together.

When you create an index for a Paradox or dBASE table, Database Desktop creates a file that contains the indexed field’s values and their locations. Database Desktop uses the index file to locate and sort the records in a table.

Indexes work differently for Paradox, dBASE, and SQL tables.

### Keys in Paradox tables

In Paradox tables, the primary index is called the *key*. A Paradox table’s key establishes the primary index and sort order for the table. A key also requires each value in the field(s) that defines the key to be unique. For example, if the Cust ID field is identified as the key of the *Customer table*, each value in the Cust ID field must be unique. Likewise, if the Cust, Date, and Item # fields are identified as the key of the *Bookord table*, the field



values (taken as an ordered group) must be unique. This guards against duplication of data within the table. Keys are required for linking tables and for using the data integrity features of Paradox tables.

To create a key, display the Create Table (page 151) or the Restructure Table (page 160) dialog box. Then move to the Key column in the Field Roster and double-click (or press any key). The key field indicator (\*) appears. Database Desktop keys the table on the selected field.

Follow these rules when defining keys:

- A table can have only one primary key. This key can be made up of one or more fields.
- If a key is defined as a single field, that field must be the first field in the Field Roster.
- If you identify more than one field as keyed, you create a *composite* key. These fields, taken as a group, must be unique for each record of the table. The composite key must be the first fields in the Field Roster.

## A dBASE table's index

When working with dBASE tables, Database Desktop uses an index to organize the records in a table according to the values in one or more fields.

When you create an index on a dBASE table, a file is created that contains the indexed field's values and their corresponding record numbers. Database Desktop refers to the index file when locating and displaying the records in a table.

Although Database Desktop supports both .MDX files and .NDX files, it is recommended that you use a dBASE production index (the .MDX file which uses the table's name as its file name) whenever possible. Although you can create nonproduction .MDX files as well as .NDX files, Database Desktop automatically maintains the production index.

For more information on dBASE indexes, search for "dBASE indexes" in the keyword list in Database Desktop Help.

**Note** Delphi does not support all dBASE index types. For more information, see Chapter 3, "Using data access components and tools."

## An SQL table's index

SQL tables use unique and non-unique indexes, but they do not use the primary keys that Paradox tables use. You can create multiple indexes for an SQL table; for each index, you specify whether it is unique or non-unique. Depending on the server, you may also be able to specify whether the index is case-sensitive and whether it is ascending or descending. SQL indexes, unlike Paradox and dBASE indexes, are always maintained.

You can use Database Desktop to create and modify indexes on SQL tables, but you cannot specify which index to use in Database Desktop.

When you use an SQL table in Database Desktop, the table should have a unique index. If it does not have a unique index you may not be able to view new records that you

insert, depending on the server. To add a unique index to an existing table, choose Utilities | Restructure.

For more information on SQL indexes, see “Creating indexes on SQL tables” on page 155.

## Defining secondary indexes

---

A *secondary index* is a field or group of fields other than the key field that can be used to sort the table or to link the table to other tables. Database Desktop enables you to create secondary indexes for Paradox tables.

You can use a secondary index to see an alternate view order for a Paradox table. For example, to view the *Customer* table by City, while keeping the table’s key order intact, you can use a secondary index on City to temporarily change the view order of the records.

For information on creating a secondary index, search for “secondary indexes” in the keyword list in Database Desktop Help, and choose the topic “Defining a Secondary Index.”

## Specifying validity checks

---

Validity checks govern the values you can enter in a field. The five types of validity checks are listed in Table A.1.

**Note** Validity checks work only on Paradox tables, not on dBASE tables. For SQL tables, the only validity check you can specify in Database Desktop is whether a field is required (not Null).

**Table A.1** Paradox validity checks

Validity check	Meaning
Required field	Every record in the table must have a value in this field. SQL tables can also use this validity check (equivalent to NOT NULL).
Minimum	The values entered in this field must be equal to or greater than the minimum you specify here.
Maximum	The values entered in this field must be less than or equal to the maximum you specify here.
Default	The value you specify here is automatically entered in this field. You can replace it with another value.
Picture	You specify a character string that acts as a template for the values that can be entered in this field. The values entered in this field are automatically formatted according to this picture.  For information on pictures, search for “picture strings” in the keyword list in Database Desktop Help, and choose the topic “Picture String Characters.”

---

For detailed information on validity checks, search for “validity checks” in the keyword list in Database Desktop Help.

**Note** A “Required field” validity check will set a Delphi *TField*’s *Required* property to True when the table is accessed from a Delphi application.

## Borrowing a table structure

When creating a table similar to one you already have, you can borrow its structure. Then you can either use it as is or change it. You must begin from a blank table structure to borrow another table's structure.

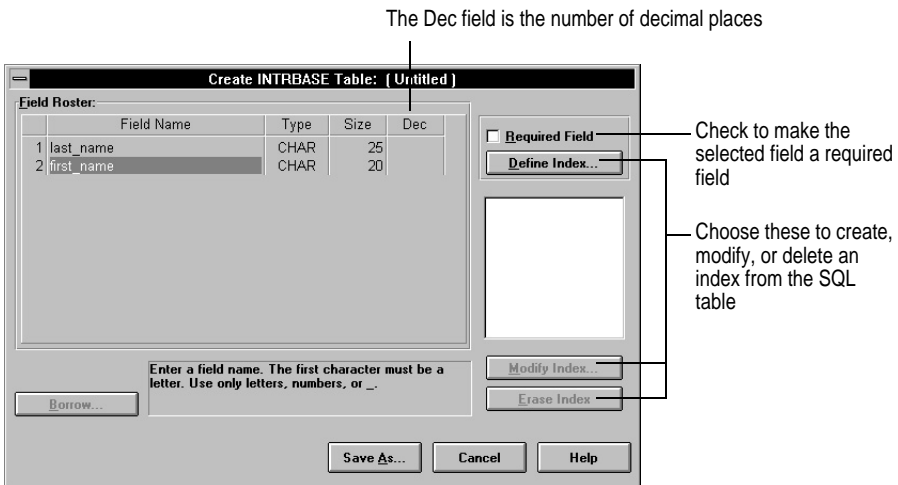
To borrow a table structure, choose Borrow from the Create Table dialog box.

For detailed information on borrowing, search for "borrow a table's structure" in the keyword list in Database Desktop Help.

## Creating an SQL table

When you create an SQL table, you can define the table structure (fields & types), specify required fields, and define indexes. The Create Table dialog box for SQL tables looks as shown in Figure A.6.

**Figure A.6** The Create Table dialog box for SQL tables



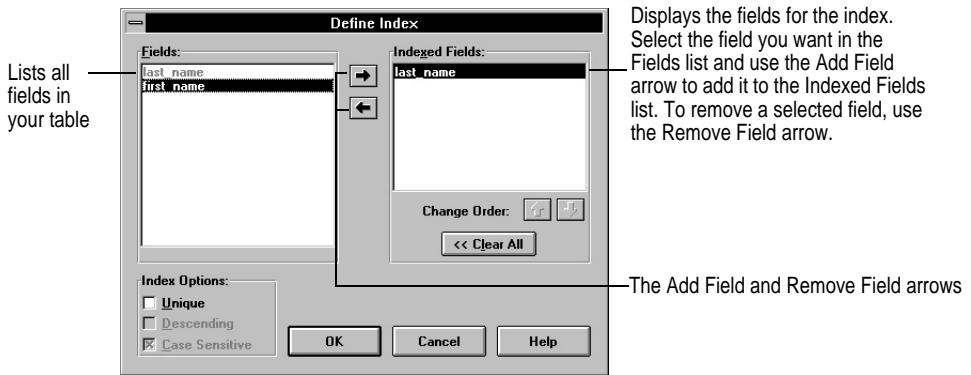
For information on valid field types for your SQL server, search for "field types" in the keyword list in Database Desktop Help, and choose the topic for your server.

## Creating indexes on SQL tables

You can use Database Desktop to create and modify indexes on SQL tables.

To create an index for an SQL table, display the Create Table (page 155) or the Restructure Table (page 160) dialog box. Then, choose Define Index. Database Desktop displays the Define Index dialog box, shown in Figure A.7.

**Figure A.7** The Define Index dialog box for SQL indexes



When you use an SQL table in Database Desktop, the table should have a unique index. If it does not have a unique index, you may not be able to view new records you insert.

For detailed information on using the Define Index dialog box, search for “Define Index Dialog Box” in the keyword list in Database Desktop Help, and choose the topic “Define Index Dialog Box (SQL Tables).”

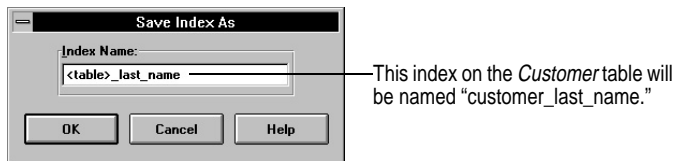
## Naming SQL indexes

For most database servers, index names must be unique within the database (or in some other predefined workspace). When you create an index on an SQL table, Database Desktop prompts you to prefix the index name with the table name to ensure that the index name is unique.

**Sybase note** Sybase index names need only be unique within a table, not within the entire database, so Database Desktop does not prefix Sybase index names with table names.

When you create an SQL index and choose OK from the Define Index dialog box, Database Desktop supplies the prefix “<table>\_” for the index name as follows:

**Figure A.8** Save Index As dialog box



You can include the table name with the index name or omit it:

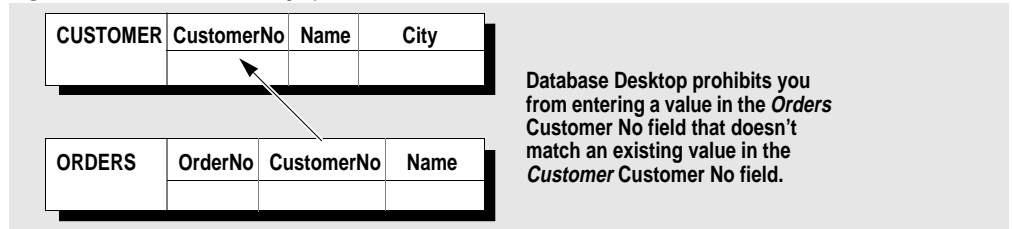
- If you type the index name following “<table>\_”, Database Desktop prefixes the index name with the table name and an underscore.
- If you delete “<table>\_”, Database Desktop omits the table name from the index name. If the index name is not unique, an error will occur when Database Desktop saves the table.

This index naming scheme also affects restructuring, as described in “Restructuring an SQL table” on page 160.

## Defining referential integrity for Paradox tables

Referential integrity means that a field or group of fields in one table (the “child” table) must refer to the key of another table (the “parent” table). Database Desktop enables you to define referential integrity rules for Paradox tables.

**Figure A.9** Referential integrity



Database Desktop accepts only those values that exist in the parent table's key as valid values for the specified field(s) of the child table. You can establish referential integrity only between like fields that contain matching values. For example, you can establish referential integrity between *Customer* and *Orders* on their *CustomerNo* fields. In both cases, the values contained in the specified fields are the same. The field names don't matter as long as the field types and sizes are identical.

**Note** You can establish referential integrity only between tables in the same directory.

Using referential integrity, Database Desktop checks the validity of a value before accepting it in the table. If you establish referential integrity between *Customer* and *Orders* on their *CustomerNo* fields, then enter a value in the *CustomerNo* field of *Orders*, Database Desktop searches the *CustomerNo* field of *Customer* and

- Accepts the value in *Orders* if it exists in *Customer*
- Rejects the value in *Orders* if it doesn't exist in *Customer*

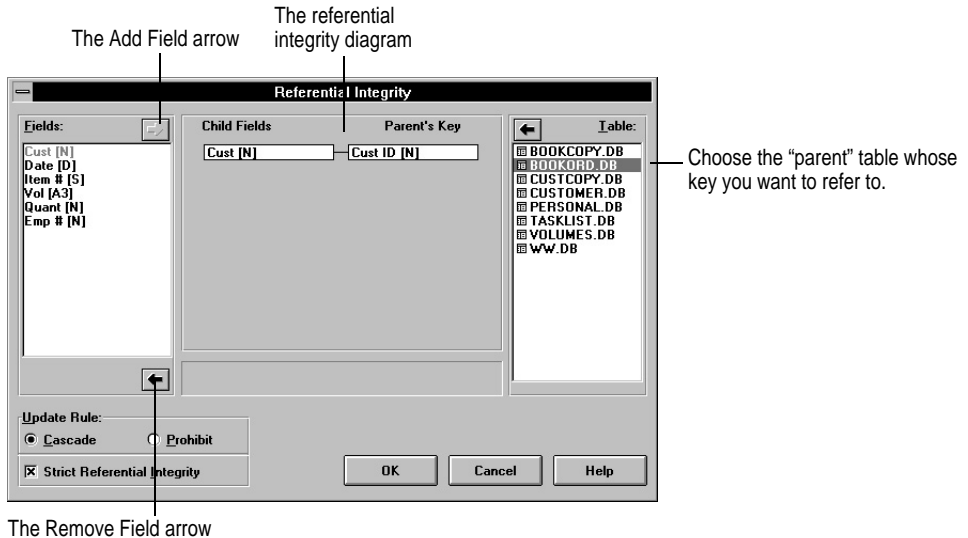
## Procedure

To define a referential integrity relationship,

- 1 In the Create Table (page 151) or Restructure Table (page 160) dialog box, choose Referential Integrity from the Table Properties list. The Define button becomes available.

## 2 Choose Define to open the Referential Integrity dialog box.

**Figure A.10** Referential Integrity dialog box



- 3 Choose the parent table from the Table list. The table's key field appears in the Parent's Key area of the referential integrity diagram.
- 4 Double-click the child table's field in the Fields list (or *Tab* to it and click the Add Field arrow or press *Alt+A*). The field name appears in the Child Fields area of the referential integrity diagram.
- 5 Choose the update rule you want.
- 6 Choose whether you want to enforce strict referential integrity.
- 7 Choose OK to name and save the referential integrity relationship.

For detailed information on defining referential integrity, search for "Referential Integrity" or "Referential Integrity Dialog Box" in the keyword list in Database Desktop Help.

## Creating table lookup

Table lookup helps you enter data in one Paradox table that already exists in the first field of another Paradox table—the *lookup* table. Table lookup lets you

- Require that the values you enter into a field exist in the first field of another table
- Refer to another table to look up the acceptable values for a field
- Copy values in the lookup table to the table you're editing

Table lookup is primarily a data entry tool. Unlike referential integrity, it doesn't track or control changes you make to the lookup table. Table lookup ensures that data is copied accurately from one table to another; referential integrity ensures that the ties between data in separate tables cannot be broken.

For detailed information on table lookup, search for “table lookup” in the keyword list in Database Desktop Help, and choose the topic “Looking up Table Values.”

**Note** Table lookup rules have no effect on Delphi applications accessing a table.

## Establishing passwords for Paradox tables

---

Sometimes it’s important to ensure that the Paradox table you create is protected from access by unauthorized users. Not only can you establish a password for a Paradox table as a whole, but you can also assign specific rights to the table or individual fields.

For detailed information on creating and using passwords, search for “passwords” in the keyword list in Database Desktop Help.

## Restructuring tables

---

Database Desktop enables you to restructure Paradox and dBASE tables to:

- Add or rename fields
- Change field types or sizes
- Modify indexes
- Modify table language drivers

Restructuring a table is very much like creating it for the first time. You will not be able to restructure a table if Delphi or any other application has the table open. For detailed information on restructuring tables, see the Database Desktop Help; choose Help | Contents | Tasks | Creating and Restructuring | Restructuring a Table.

**Note** Restructuring a table may require corresponding modifications in Delphi applications that access the table. For example, removing a column or changing its data type will raise an exception if an application has a persistent *TField* component for the modified column.

To restructure a table, choose Utilities | Restructure, then choose the table you want. If the table you want to restructure is already open in the active window, use Table | Restructure. The Restructure Table dialog box opens. The following figure shows an example of a Paradox table in the Restructure Table dialog box:

**Figure A.11** The Restructure Table dialog box for Paradox tables



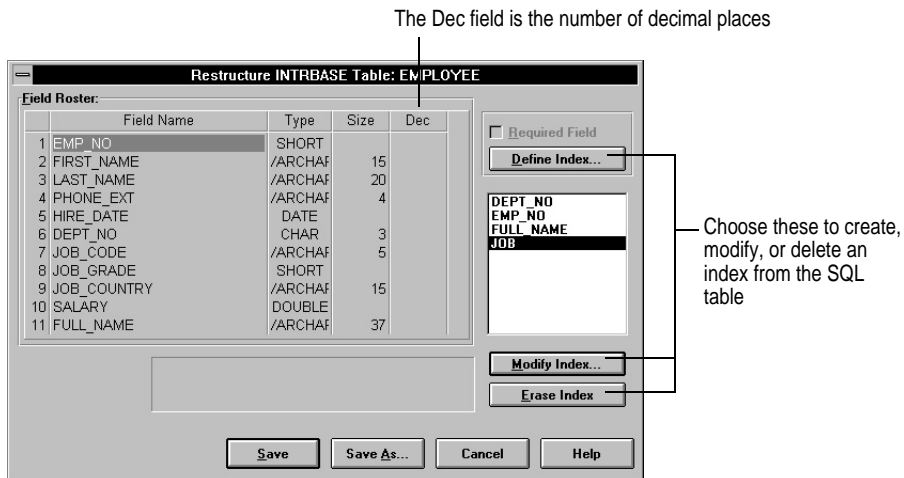
Work in the Restructure Table dialog box the same way you work in the Create Table dialog box.

Pack a table to reuse disk space left over from deleting records. Some restructure operations automatically pack your table. You can check Pack Table and choose OK when you want to be sure Paradox packs the table.

## Restructuring an SQL table

When you restructure an SQL table, you can add, modify, and drop indexes. You cannot otherwise change the structure of a table on a server. The Restructure Table dialog box for SQL tables looks as shown in Figure A.12.

**Figure A.12** The Restructure Table dialog box for SQL tables



## Prefixing the index name with the table name

Database Desktop prefixes some index names with the table name, as described in “Creating indexes on SQL tables” on page 155. These index names are also affected when you restructure an SQL table as follows:



- If you create a new index during a restructure, Database Desktop prompts you to prefix the index name with the table name.
- If you modify an index during a restructure, Database Desktop does not modify the index name, unless you rename the index as part of your modification.
- If you choose Save As during a restructure, Database Desktop prefixes all index names with the new table name, if you have not explicitly entered an index name. For example, suppose the EMPLOYEE table contains the following indexes:

```
EMPLOYEE_DEPT_NO
EMPLOYEE_EMP_NO
FULL_NAME
JOB
```

If you restructure the table and save it as MY\_DEPT, Database Desktop renames the indexes as follows:

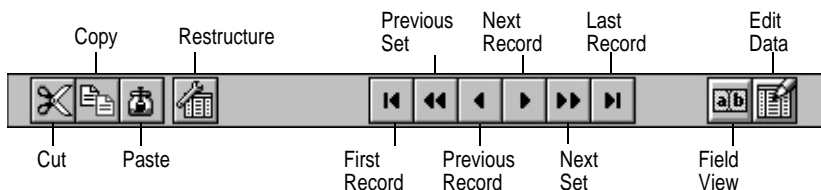
```
MY_DEPT_DEPT_NO
MY_DEPT_EMP_NO
MY_DEPT_FULL_NAME
MY_DEPT_JOB
```

## Viewing tables



To open a table, choose File | Open | Table. Or, if the application window is empty, click the Open Table button in the tool bar. In the Open Table dialog box, choose the table to open. The table you chose opens in a Table window, and the tool bar appears as shown in the following figure:

**Figure A.13** The Table window tool bar



When you first open a table, its data appears in a Table window in View mode. Each Table window contains an independent view of a table, so different views of a single table can be open at the same time. Up to 24 tables can be open at one time.

To be able to simultaneously access tables stored on a network, you must provide Database Desktop the location of the *network control file*. You do this by running the BDE Configuration Utility; double-click the BDE Configuration Utility icon in the Delphi program group. See online Help in the BDE Configuration Utility for more details.

A Delphi application can specify the network control file in the *NetFileDir* property of *TSession*. This enables a Delphi application access to tables stored on a network.


## Using scroll lock

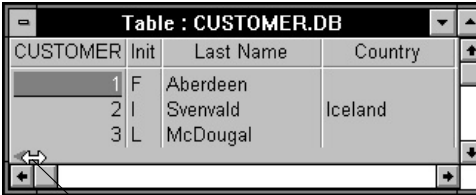
To lock one or more columns in place as you move horizontally through the table's columns, use a *scroll lock*. All columns to the left of the lock remain stationary as you move through the table's columns.



The scroll lock is a triangle in the lower left corner of the Table window. To place a lock, drag the triangle to the right side of the column(s) you want to lock. An active scroll lock appears as two triangles when you release the mouse button, as shown in Figure A.14.

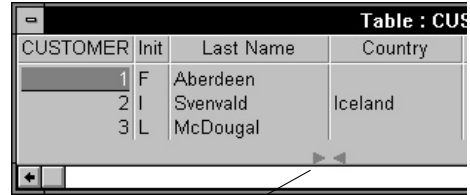
**Figure A.14** A scroll lock in the Table window

 The scroll lock triangle in the lower left corner.




CUSTOMER	Init	Last Name	Country
1	F	Aberdeen	
2	I	Svenvald	Iceland
3	L	McDougal	

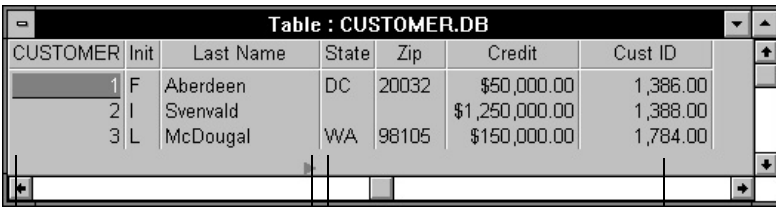
When you position the pointer over the scroll lock triangle, it changes to a double-headed arrow.



CUSTOMER	Init	Last Name	Country
1	F	Aberdeen	
2	I	Svenvald	Iceland
3	L	McDougal	

As you drag, the pointer changes to two arrows.

 After you release the mouse button, an active scroll lock appears.



CUSTOMER	Init	Last Name	State	Zip	Credit	Cust ID
1	F	Aberdeen	DC	20032	\$50,000.00	1,386.00
2	I	Svenvald			\$1,250,000.00	1,388.00
3	L	McDougal	WA	98105	\$150,000.00	1,784.00

As you scroll, these columns remain stationary.


As you scroll, these columns change.

## Customizing a table view

The *view* of a table is how it appears onscreen; you can modify and save a custom view of a table. Changing the view makes it easier to see specific fields; the actual *structure* of the table (its definition of field order and size) remains the same. To customize a view, you can rearrange, resize, and lock columns, and resize rows or table headings.

The following figure shows the *hot zones* on an open table view. Hot zones indicate areas on a table where you can drag to modify the view of the table. As the pointer passes over a hot zone, the pointer changes shape.

**Figure A.15** Hot zone pointers in the Table window

 — The pointer when changing the heading or row height




VENDORS	Vendor No	Vendor Name	Street
1	2014	Cacor Corporation	161 Southfield Rd
2	2641	Underwater	50 N 3rd Street
3	2674	J.W. Luscher Mfg.	65 Addams Street
4	3511	Scuba Professionals	3105 East Brace
5	3819	Divers' Supply Shop	5208 University Dr
6	3620	Techniques	52 Dolphin Drive

To change the heading height, drag the table name up or down.



VENDORS	Vendor No	Vendor Name	Street
1	2014	Cacor Corporation	161 Southfield Rd
2	2641	Underwater	50 N 3rd Street
3	2674	J.W. Luscher Mfg.	65 Addams Street
4	3511	Scuba Professionals	3105 East Brace
5	3819	Divers' Supply Shop	5208 University Dr
6	3620	Techniques	52 Dolphin Drive


To change the row height, drag this line up or down.

 — The pointer when changing the column width



VENDORS	Vendor No	Vendor Name	Street
1	2014	Cacor Corporation	161 Southfield Rd
2	2641	Underwater	50 N 3rd Street
3	2674	J.W. Luscher Mfg.	65 Addams Street
4	3511	Scuba Professionals	3105 East Brace
5	3819	Divers' Supply Shop	5208 University Dr
6	3620	Techniques	52 Dolphin Drive

To resize a column, drag its right grid line in its top row.

 — The pointer when moving a column



VENDORS	Vendor No	Vendor Name	Street
1	2014	Cacor Corporation	161 Southfield Rd
2	2641	Underwater	50 N 3rd Street
3	2674	J.W. Luscher Mfg.	65 Addams Street
4	3511	Scuba Professionals	3105 East Brace
5	3819	Divers' Supply Shop	5208 University Dr
6	3620	Techniques	52 Dolphin Drive

To move a column, drag its heading to the left or right.

## Rearranging and resizing columns



To move a column, position the pointer on a column's heading. When the pointer changes shape (shown at left), drag the column to its new position.



To resize a column, position the pointer on its right boundary line (either the heading area or the top row of data). When the pointer changes shape (shown at left), drag the boundary line to increase or decrease the width of the column.

## Resizing rows



To resize the height of all of the rows in a table, drag the line under the first record number. Move the line up to decrease the row height, or down to increase the row height.

## Resizing column headings



To resize the height of all the column headings, drag the line under the table name. The table name is located above the left-most column (which contains the record numbers).

## Saving a custom view

You can save a custom view, undo changes to a view, and restore the default view by using the commands on the Properties menu.

For detailed information on the Properties menu commands, search for “Properties menu” in the keyword list in Database Desktop Help.

## Editing data

This section introduces Database Desktop’s editing features. For detailed information on editing data, search for “editing data” or “Edit mode” in the keyword list in Database Desktop Help.

### Using Edit mode

To change data in a table, you must be in Edit mode. To enter Edit mode, do one of the following:



- Click the Edit Data button in the tool bar.
- Choose View | Edit Data.
- Press *F9*.

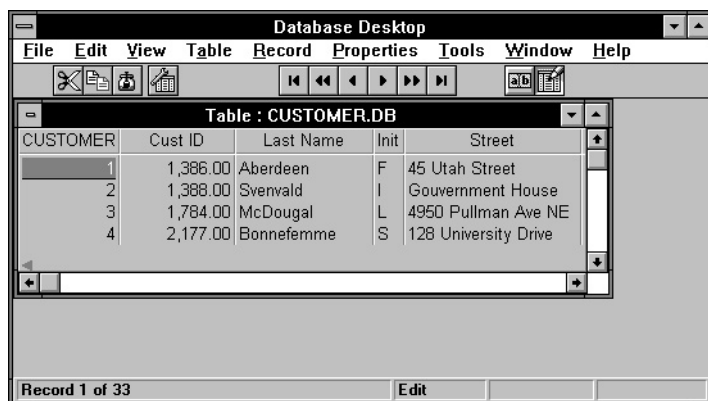
F9

In Edit mode, you can select any field and begin typing to replace its existing entry. When you enter Edit mode, the Edit Data button remains pressed in and the status line tells you Edit mode is active.

**Note** In Database Desktop you cannot edit data in the following field types:

- **Paradox:** Memo, Formatted Memo, Graphic, OLE, Autoincrement, Binary, or Bytes
- **dBASE:** Memo, OLE, or Binary
- **SQL:** any BLOB (binary large object) field or a text field that allows more than 255 characters

**Figure A.16** The *Customer* table in Edit mode



## Selecting fields and records

---

When you move to a field or click it, the field is highlighted. This indicates that the field is *selected*. In Edit mode, if you type anything into a selected field, you'll replace the existing entry with the value you type. The cut, copy, and paste operations affect the entire field entry when it's selected.

You can select more than one field at a time, or select a portion of a single field entry.

For detailed information on selecting fields and records, search for "selecting data" in the keyword list in Database Desktop Help.

## Field view

---

In Edit mode, you can change a field's entry in one of two ways:

- Select the field and type a new value. When you begin typing, the new value replaces the old entry.
- Select the field and edit the existing entry using *field view*.

For information on field view, search for "Field View" in the keyword list in Database Desktop Help.

## Adding, subtracting, and emptying records

---

Use the commands on the Utilities menu to add, subtract, or empty a table's records.

- **Add:** You can add the records from one table to another table. You can use Add on all table types. Choose Utilities | Add. Database Desktop opens the Add dialog box.

For detailed information on adding records, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Adding Records.

- **Subtract:** You can remove records that exist in one table from a different table by using the Subtract utility. You can subtract records only from a keyed Paradox table. Choose Utilities | Subtract. Database Desktop opens the Subtract dialog box.

For detailed information on subtracting records, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Subtracting Records.

**SQL** You cannot use an SQL table as the source of a Subtract operation.

- **Empty:** Use the Empty utility to remove all records from a table, leaving the table's structure (including all keys, indexes, validity checks, and so on) intact. You can use Empty on Paradox, dBASE, and SQL tables. Choose Utilities | Empty. Database Desktop opens the Empty dialog box.

For detailed information on emptying tables, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Emptying Tables.

# Sorting, copying, renaming, and deleting objects

---

You can use Database Desktop to sort tables and to copy, rename, and delete objects.

## Sorting tables

---

When you sort a table, you tell Database Desktop to rearrange the order of the records in the table and display them in the order you specify.

- If a table is keyed, Database Desktop creates a new, unkeyed table containing the sorted data. The original table remains unchanged.
- If a table is not keyed, the sort changes the actual location of the records in the table.

**SQL** You cannot sort SQL tables.

To sort a table, choose Utilities | Sort, then choose the table you want to sort from the Select File dialog box. Database Desktop opens the Sort Table dialog box.

For detailed information on sorting tables, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Sorting Tables.

## Copying objects

---

You can copy Paradox and dBASE tables, queries, SQL tables, and .SQL files from within Database Desktop. To copy an object, choose Utilities | Copy. Database Desktop opens the Copy dialog box.

When you copy a table, Database Desktop copies both the structure of the table and the data contained in it. You can copy tables from one table type to another. For example, you can copy a Paradox table to a dBASE or SQL table. To copy to an SQL table type, you must have an SQL database server and the appropriate SQL Link.

For detailed information on copying objects, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Copying Tables.

## Renaming objects

---

You can rename tables, queries, and .SQL files from within Database Desktop. You cannot rename SQL tables.

To rename an object, choose Utilities | Rename. Database Desktop opens the Rename dialog box. For detailed information on renaming objects, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Renaming Tables.

## Deleting objects

---

You can delete tables, queries, SQL tables, and .SQL files from within Database Desktop.

To delete an object, choose Utilities | Delete. Database Desktop opens the Delete dialog box. For detailed information on deleting objects, see Database Desktop Help; choose Help | Contents | Tasks | Using Table Utilities | Deleting Tables.

## Executing SQL statements

---

This section describes how to use the SQL Editor to enter and execute SQL statements.

### What is the SQL Editor?

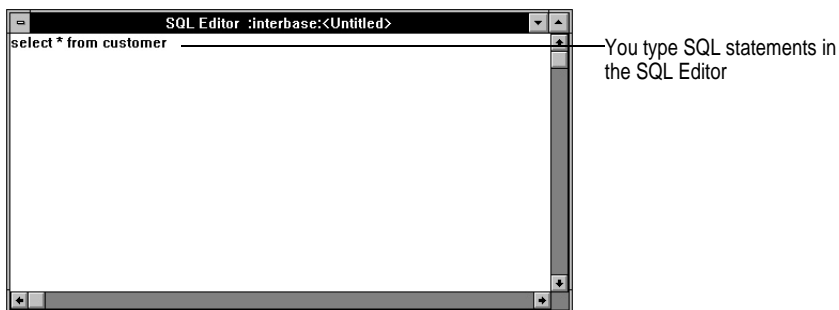
---

Programmers familiar with SQL can use the SQL Editor window to directly enter, execute, or save an SQL statement. You can save the SQL statement to a file, and then later load, modify, or execute it.

In the SQL Editor, you can enter SQL statements to be executed by a SQL database server. The syntax must conform to your server's dialect. This is referred to as *passthrough SQL*. The SQL server performs all error or syntax checking and executes the statement without any involvement by Database Desktop.

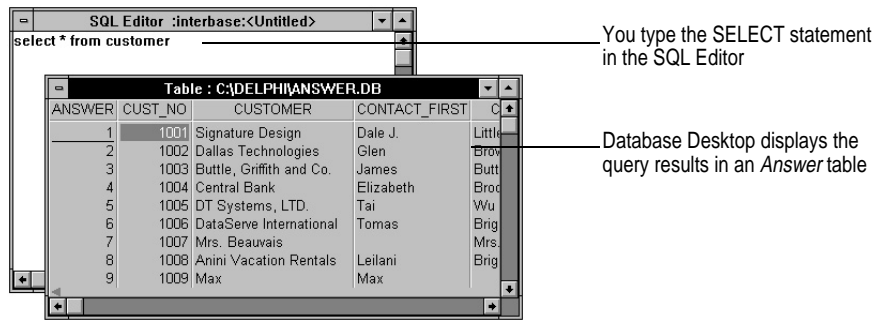
You can also use the SQL Editor to run SQL statements against Paradox or dBASE tables. This is known as local SQL, and the allowable syntax is a subset of ANSI standard SQL. For more information, see Appendix C, "Using local SQL."

**Figure A.17** The SQL Editor



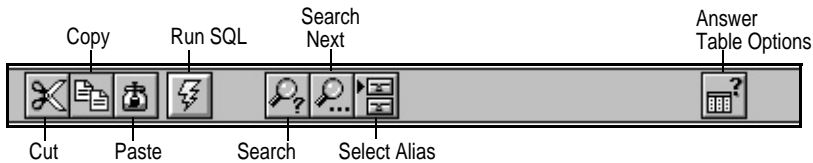
If you execute a SELECT statement in the SQL Editor, Database Desktop displays the resulting data in an *Answer* table, as shown in Figure A.18.

**Figure A.18** The SQL Editor and an *Answer* table



The SQL Editor has the tool bar shown in Figure A.19.

**Figure A.19** SQL Editor Toolbar



## Opening the SQL Editor

To open the SQL Editor, do one of the following:

To do this	Do this
Enter (and execute) a new SQL statement	Choose File   New   SQL Statement Or right-click the Open SQL Script tool bar button and choose New
Open (and edit or execute) an existing .SQL file	Choose File   Open   SQL Statement Or click the Open SQL Script tool bar button Or right-click the Open SQL Script tool bar button and choose Open
View the SQL equivalent of an open QBE query	Choose Query   Show SQL Or click the Open SQL Script tool bar button

## Specifying an alias

Before running an SQL statement, you must specify the alias that the statement will run against. To specify an alias, do one of the following:



- Choose SQL | Select Alias.
- Click the Select Alias tool bar button.

Database Desktop opens the Select Alias dialog box, where you can choose one of the aliases you created in the Alias Manager dialog box. You cannot include an alias in the text of the SQL statement.



## Running an SQL statement

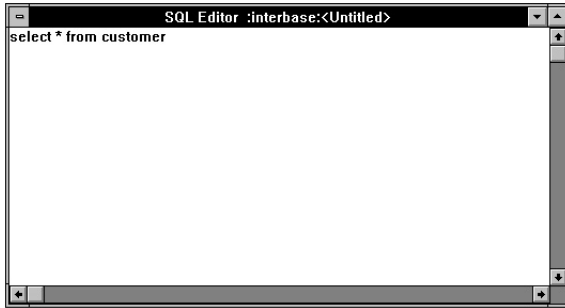
---

You can enter multiple SQL statements if your server allows it and you include only one SELECT statement.



To run an SQL statement that you have typed in the SQL Editor window, click the Run SQL tool bar button or choose SQL | Run SQL.

**Figure A.20** SQL statement in the SQL Editor



If your SQL statement is a query, the query results are displayed in an *Answer* table, as shown in Figure A.18 on page 168.

## Saving an SQL statement

---

To save the SQL statement in the active window, choose File | Save or File | Save As. When you save an SQL statement to your local hard disk, Database Desktop places it in an unformatted text file with an .SQL extension.

## Querying table data with QBE

---

A *query* is a question you ask about information in one or more tables. In addition to standard SQL queries, Database Desktop enables you to use a technique called query by example (QBE) to extract and manipulate data in tables. With QBE, you make the query image look like an example of the records you want to search for.

The following figure shows a query that gives examples of the fields you want to see (and a range of values within one of those fields), and the answer Database Desktop gives.

**Figure A.21** A query and its results

This example searches for book prices greater than \$28 and less than \$65 (the checkmarks specify which fields appear in the *Answer* table).

The *Answer* table displays the checked fields for records that match the example.

ANSWER	Title	Price
1	Austrian Arms and Armor	\$35.95
2	French Cooking	\$35.90
3	Hollywood's Golden Years	\$29.95
4	Organic Composting	\$31.00
5	Soccer Stars	\$31.95
6	Structural Steel	\$45.00
7	Tibetan Adventures	\$35.75
8	Vegetarian Lifestyle	\$29.90
9	Victorian Garden, The	\$35.95
10	Vinyl Fashions	\$34.00
11	Zen Influences in Japan	\$29.95

The result of a query is a temporary table called *Answer*. The *Answer* table is overwritten each time a query is run.

**SQL** If you have Borland SQL Links, you can use QBE to view and query tables on SQL servers. For more information, see the Database Desktop Help Contents.

For details on using QBE, see Database Desktop Help; choose Help | Contents | Tasks | Using Query-By-Example.

## B

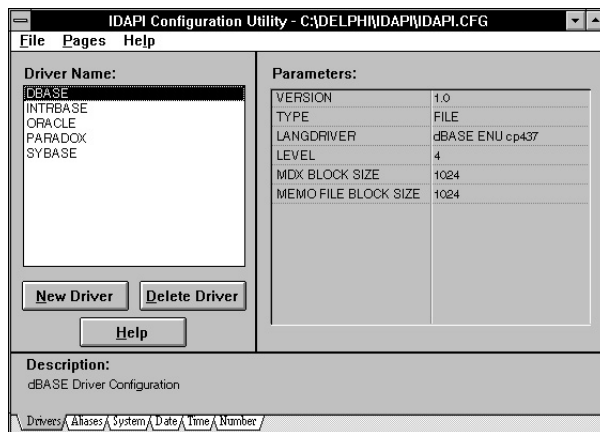
## Using the BDE configuration utility



The Borland Database Engine configuration utility (BDECFG.EXE) enables you to configure BDE aliases and change the settings reflecting your specific environment in the BDE configuration file, IDAPI.CFG.

To run the BDE Configuration Utility, double-click the BDE configuration utility icon in the Delphi program group. The BDE Configuration Utility opens:

**Figure B.1** BDE Configuration Utility main window



## Creating and managing aliases

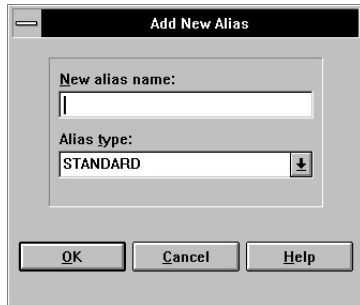
It is usually best to use an alias in your application instead of a hard-coded file or directory name. Setting up a standard alias consists of assigning a name to, and specifying the path name for a directory containing Paradox or dBASE files or the directory path and database name on a SQL server.

## Adding a new alias

To add a new alias,

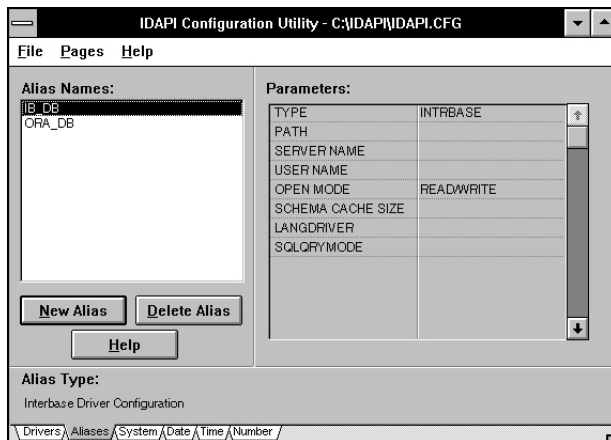
- 1 Select the Alias Manager (Aliases page) and choose the New Alias button. The Add New Alias dialog box appears. The type can be STANDARD or SQL-specific.

**Figure B.2** Sample Add New Alias dialog box



- 2 Enter the new alias name and select the SQL-specific alias type. Then choose OK to begin the setup process. The Alias Manager displays all the configuration parameters you can change to customize the new alias.

**Figure B.3** Customizing the new alias



- 3 If desired, edit the settings for the category you selected. If you leave any categories blank, the Alias Manager assumes you want to use the default for driver type.
- 4 When you are finished, select File | Save to save the new alias in the default configuration file; select File | Save as to save the new alias in a configuration file with a different name.

**Note** The other pages contain settings that can also be customized. See online help for specifics.

Page	Settings modified
Driver Manager	Those BDE uses to determine how an application creates, sorts, and handles tables.
System Manager	Those BDE uses to start an application.
Date Manager	Those used to convert string values into date values.
Time Manager	Those used to convert string values into time values.
Number Manager	Those used to convert string values to number values.

If you save the new alias in a configuration file with a different name, the BDE Configuration Utility displays:

**Figure B.4** BDE non-system configuration dialog box



Choose Yes if you want to activate this configuration file next time you start your application. Choose No if you want to keep using the current default configuration file. Your changes take effect the next time you start your application.

## Modifying an existing alias

To modify an existing alias,

- 1 Scan the list of Alias Names available through the current configuration file. If the alias you want to modify was stored in a different configuration file, use File | Open to load that configuration file.
- 2 Highlight the name of the alias you want to modify. The configuration for that alias appears in the Parameters section of the Alias Manager page.
- 3 Highlight the configuration parameter you want to change, and enter the desired value. If you leave any categories blank, the Alias Manager assumes you want to use the driver's default value.
- 4 When you are finished, select File | Save to save the new alias in the default configuration file; select File | Save As to save the new alias in a configuration file with a different name.

When you modify a driver parameter, all aliases that use the default setting for that parameter inherit the new setting.

Your changes take effect the next time you start your application.

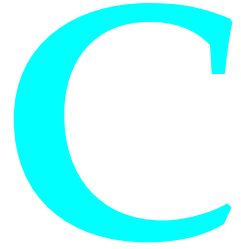
## Deleting an alias

---

To delete an alias,

- 1 Scan the list of Alias Names available through the current configuration file. If the alias you want to delete was stored in a different configuration file, use File | Open to load that configuration file.
- 2 Highlight the name of the alias you want to modify, and select the Delete Alias button.
- 3 Select File | Save to save your changes in the default configuration file; select File | Save As to save your changes in a different configuration file.

**Note** If an application attempts to use an alias that has been deleted, Delphi will raise an exception. If a Delphi form was saved with a table or query open, Delphi will attempt to open the dataset when the form is loaded and the exception will occur at that time. In many cases, modifying an alias can also cause an exception in Delphi forms if the changes to the alias require changes to the Delphi application.



## Using local SQL

The BDE enables limited access to database tables through local SQL (also called “client-based SQL”). Local SQL is a subset of ANSI-standard SQL enhanced to support Paradox and dBASE naming conventions for tables and fields (called “columns” in SQL). Two categories of SQL statements are supported:

- Data Manipulation Language (DML) for selecting, inserting, updating, and deleting table data.
- Data Definition Language (DDL) for creating, altering, and dropping tables, and for creating and dropping indexes.

This appendix describes naming conventions, syntax enhancements, and syntax limitations for local SQL. For a complete introduction to ANSI-standard SQL, see one of the many third-party books available at your local computer book store.

### Naming conventions for tables

---

ANSI-standard SQL confines each table name to a single word comprised of alphanumeric characters and the underscore symbol (`_`). Local SQL is enhanced to support full file and path specifications for table names. Table names with path or file-name extensions must be enclosed in single or double quotes. For example,

```
SELECT * FROM 'PARTS.DBF'  
SELECT * FROM "C:\SAMPLE\PARTS.DBF"
```

Local SQL also supports BDE aliases for table names. For example,

```
SELECT * FROM :PDOX:TABLE1
```

Finally, local SQL permits table names to duplicate SQL keywords as long as those table names are enclosed in single or double quotes. For example,

```
SELECT PASSID FROM "PASSWORD"
```

# Naming conventions for columns

---

ANSI-standard SQL confines each column name to a single word of alphanumeric characters and the underscore symbol (\_). Local SQL is enhanced to support Paradox and dBASE multi-word column names and column names that duplicate SQL keywords as long as those column name are

- Enclosed in single or double quotes.
- Prefaced with an SQL table name or table correlation name.

For example, the following column name is two words:

```
SELECT E."Emp Id" FROM EMPLOYEE E
```

In the next example, the column name duplicates the SQL DATE keyword:

```
SELECT DATELOG."DATE" FROM DATELOG
```

# Data manipulation

---

With some restrictions, local SQL supports the following statements for data manipulation:

SELECT, for retrieving existing data

INSERT, for adding new data to a table

UPDATE, for modifying existing data

DELETE, for removing existing data from a table

The following sections describe parameter substitution, aggregate, string, and date functions, and operators available to DML statements in local SQL.

## Parameter substitutions in DML statements

---

Variables or parameter markers (?) can be used in DML statements in place of values. Variables must always be preceded by a colon (:). For example,

```
SELECT LAST_NAME, FIRST_NAME  
FROM "CUSTOMER.DB"  
WHERE LAST_NAME > :var1 AND FIRST_NAME < :var2
```

## Supported set (aggregate) functions

---

The following ANSI-standard SQL set (or “aggregate”) functions are available to local SQL for use with data retrieval:

- SUM(), for totaling all numeric values in a column
- AVG(), for averaging all non-NULL numeric values in a column
- MIN(), for determining the minimum value in a column



- MAX(), for determining the maximum value in a column
- COUNT(), for counting the number of values in a column that match specified criteria

**Note** Expressions are not allowed in set functions.

## Supported string functions

---

Local SQL supports the following ANSI-standard SQL string manipulation functions for retrieval, insertion, and updating:

- UPPER(), to force a string to uppercase
- LOWER(), to force a string to lowercase
- SUBSTRING(), to return a specified portion of a string
- TRIM(), to remove repetitions of a specified character from the left, right, or both sides of a string

## Supported date function

---

Local SQL supports the EXTRACT() function for isolating a single numeric field from a date/time field on retrieval using the following syntax:

```
EXTRACT (extract_field FROM field_name)
```

For example, the following statement extracts the year value from a DATE field:

```
SELECT EXTRACT(YEAR FROM HIRE_DATE)
FROM EMPLOYEE
```

You can also extract MONTH, DAY, HOUR, MINUTE, and SECOND using this function.

**Note** EXTRACT does not support the TIMEZONE\_HOUR or TIMEZONE\_MINUTE clauses.

## Supported operators

---

Local SQL supports the following arithmetic operators:

```
+, -, *, /
```

Local SQL supports the following comparison operators:

```
<, >, =, <>, IS NULL
```

Local SQL supports the following logical operators:

```
AND, OR, NOT
```

Local SQL supports the following string concatenation operator:

```
||
```

## Using SELECT

---

The SELECT statement is used to retrieve data from one or more tables. A SELECT that retrieves data from multiple tables is called a “join.” Local SQL supports the following form of the SELECT statement:

```
SELECT [DISTINCT] column_list
FROM table_reference
[WHERE search_condition]
[ORDER BY order_list]
[GROUP BY group_list]
[HAVING having_condition]
```

Except as noted below, all clauses are handled as in ANSI-standard SQL. Clauses in square brackets are optional.

The *column\_list* indicates the columns from which to retrieve data. For example, the following statement retrieves data from two columns:

```
SELECT PART_NO, PART_NAME
FROM PARTS
```

### Using the FROM clause

The FROM clause specifies the table or tables from which to retrieve data. *table\_reference* can be a single table, a comma-delimited list of tables, or can be an inner or outer join as specified in the SQL-92 standard. For example, the following statement specifies a single table:

```
SELECT PART_NO
FROM "PARTS.DBF"
```

The next statement specifies a left outer join for *table\_reference*:

```
SELECT * FROM PARTS LEFT OUTER JOIN INVENTORY
ON PARTS.PART_NO = INVENTORY.PART_NO
```

### Using the WHERE clause

The optional WHERE clause reduces the number of rows returned by a query to those that match the criteria specified in *search\_condition*. For example, the following statement retrieves only those rows with PART\_NO greater than 543:

```
SELECT * FROM PARTS
WHERE PART_NO > 543
```

The WHERE clause can now include the IN predicate, followed by a parenthesized list of values. For example, the next statement retrieves only those rows where a part number matches an item in the IN predicate list:

```
SELECT * FROM PARTS
WHERE PART_NO IN (543, 544, 546, 547)
```

**Important** A *search\_condition* cannot include subqueries.

## Using the ORDER BY clause

The ORDER BY clause specifies the order of retrieved rows. For example, the following query retrieves a list of all parts listed in alphabetical order by part name:

```
SELECT * FROM PARTS
ORDER BY PART_NAME ASC
```

The next query retrieves all part information ordered in descending numeric order by part number:

```
SELECT * FROM PARTS
ORDER BY PART_NO DESC
```

Calculated fields can be ordered by correlation name or ordinal position. For example, the following query orders rows by FULL\_NAME, a calculated field:

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME, PHONE
FROM CUSTOMERS
ORDER BY FULL_NAME
```

## Using the GROUP BY clause

The GROUP BY clause specifies how retrieved rows are grouped for aggregate functions. In local SQL, any column names that appear in the GROUP BY clause must also appear in the SELECT clause.

## Heterogeneous joins

Local SQL supports joins of tables in different database formats; such a join is called a “heterogeneous join.” For example, it is possible to retrieve data from a Paradox table and a dBASE table as follows:

```
SELECT DISTINCT C.CUST_NO, C.STATE, O.ORDER_NO
FROM "CUSTOMER.DB" C, "ORDER.DBF" O
WHERE C.CUST_NO = O.CUST_NO
```

You can also use BDE aliases in place of table names.

## Using INSERT

---

In local SQL, INSERT is restricted to a list of values:

```
INSERT INTO CUSTOMER (FIRST_NAME, LAST_NAME, PHONE)
VALUES (:fname, :lname, :phone_no)
```

Insertion from one table to another through a subquery is not allowed.

## Using UPDATE

---

There are no restrictions on or extensions to the ANSI-standard UPDATE statement.

## Using DELETE

---

There are no restrictions on or extensions to the ANSI-standard DELETE statement.

# Data definition

---

Local SQL supports data definition language (DDL) for creating, altering, and dropping tables, and for creating and dropping indexes. All other ANSI-standard SQL DDL statements are not supported. In particular, views are not supported.

Local SQL does not permit the substitution of variables for values in DDL statements.

## Using CREATE TABLE

---

CREATE TABLE is supported with the following limitations:

- Column definitions based on domains are not supported.
- Constraints are limited to PRIMARY KEY for Paradox. Constraints are unsupported in dBASE.

For example, the following statement creates a Paradox table with a PRIMARY KEY constraint on the LAST\_NAME and FIRST\_NAME columns:

```
CREATE TABLE "employee.db"
(
  LAST_NAME CHAR(20),
  FIRST_NAME CHAR(15),
  SALARY NUMERIC(10,2)
  DEPT_NO SMALLINT,
  PRIMARY KEY(LAST_NAME, FIRST_NAME)
)
```

The same statement for a dBASE table should omit the PRIMARY KEY definition:

```
CREATE TABLE "employee.db"
(
  LAST_NAME CHAR(20),
  FIRST_NAME CHAR(15),
  SALARY NUMERIC(10,2)
  DEPT_NO SMALLINT
)
```

The following table lists SQL syntax for data types used with CREATE TABLE, and describes how those types are mapped to Paradox and dBASE types by the BDE:

**Table C.1** Data type mappings

SQL Syntax	BDE Logical	Paradox	dBASE
SMALLINT	fldINT16	fldPDXSHORT	fldDBNUM
INTEGER	fldINT32	fldPDXLONG	fldDBNUM
DECIMAL(x,y)	fldBCD	fldPDXBCD	N/A
NUMERIC(x,y)	fldFLOAT	fldPDXNUM	fldDBNUM(x,y)
FLOAT(x,y)	fldFLOAT	fldPDXNUM	fldDBFLOAT(x,y)
CHARACTER(n)	fldZSTRING	fldPDXALPHA	fldDBCHAR

**x = precision (default: specific to driver); y = scale (default: 0);  
n = length in bytes (default: 0); s = BLOB subtype (default: 1)**

**Table C.1** Data type mappings

SQL Syntax	BDE Logical	Paradox	dBASE
VARCHAR(n)	fldZSTRING	fldPDXALPHA	fldDBCHAR
DATE	fldDATE	fldPDXDATE	fldDBDATE
BOOLEAN	fldBOOL	fldPDXBOOL	fldDBBOOL
BLOB(n,s)	See Subtypes below	See Subtypes below	See subtypes below
TIME	fldTIME	fldPDXTIME	N/A
TIMESTAMP	fldTIMESTAMP	fldPDXTIMESTAMP	N/A
MONEY	fldFLOAT, fldstMONEY	fldPDXMONEY	fldDBFLOAT(20,4)
AUTOINC	fldINT32, fldstAUTOINC	fldPDXAUTOINC	N/A
BYTES(n)	fldBYTES(n)	fldPDXBYTES	fldDBBYTES (in-memory tables only)

**x = precision (default: specific to driver); y = scale (default: 0);  
n = length in bytes (default: 0); s = BLOB subtype (default: 1)**

The following table specifies how BLOB subtypes translate from SQL to Paradox and dBASE through the BDE:

**Table C.2** BLOB subtype mappings

SQL Subtype	BDE Logical	Paradox	dBASE
1	fldstMEMO	fldPDXMEMO	fldDBMEMO
2	fldstBINARY	fldPDXBINARY	fldDBBINARY
3	fldstFMTMEMO	fldPDXFMTMEMO	N/A
4	fldstOLEOBJ	fldPDXOLEBLOB	fldDBOLEBLOB
5	fldstGRAPHIC	fldPDXGRAPHIC	N/A

## Using ALTER TABLE

Local SQL supports the following subset of the ANSI-standard ALTER TABLE statement. You can add new columns to an existing table using this ALTER TABLE syntax:

```
ALTER TABLE table ADD column_name data_type [, ADD column_name data_type . . .]
```

For example, the following statement adds a column to a dBASE table:

```
ALTER TABLE "employee.dbf" ADD BUILDING_NO SMALLINT
```

You can delete existing columns from a table using the following ALTER TABLE syntax:

```
ALTER TABLE table DROP column_name [, DROP column_name . . .]
```

For example, the next statement drops two columns from a Paradox table:

```
ALTER TABLE "employee.db" DROP LAST_NAME, DROP FIRST_NAME
```

ADD and DROP operations can be combined in a single statement. For example, the following statement drops two columns and adds one:

```
ALTER TABLE "employee.dbf" DROP LAST_NAME, DROP FIRST_NAME, ADD FULL_NAME CHAR[30]
```

## Using DROP TABLE

---

DROP TABLE deletes a Paradox or dBASE table. For example, the following statement drops a Paradox table:

```
DROP TABLE "employee.db"
```

## Using CREATE INDEX

---

CREATE INDEX enables users to create indexes on tables using the following syntax:

```
CREATE INDEX index_name ON table_name (column [, column . . .])
```

Using CREATE INDEX is the only way to create indexes for dBASE tables. For example, the following statement creates an index on a dBASE table:

```
CREATE INDEX NAMEX ON "employee.dbf" (LAST_NAME)
```

Paradox users can only create secondary indexes with CREATE INDEX. Primary Paradox indexes can only be created by specifying a PRIMARY KEY constraint when creating a new table with CREATE TABLE.

## Using DROP INDEX

---

Local SQL provides the following variation of the ANSI-standard DROP INDEX statement for deleting an index. It is modified to support dBASE and Paradox file names.

```
DROP INDEX table_name.index_name | PRIMARY
```

The PRIMARY keyword is used to delete a primary Paradox index. For example, the following statement drops the primary index on EMPLOYEE.DB:

```
DROP INDEX "employee.db".PRIMARY
```

To drop any dBASE index, or to drop secondary Paradox indexes, provide the index name. For example, the next statement drops a secondary index on a Paradox table:

```
DROP INDEX "employee.db".NAMEX
```

## D

## The MAST database

This appendix describes the MAST sample database provided with Delphi and used for the MASTAPP sample application. The basic MAST database is in Paradox format. A SQL version that uses the Local InterBase Server is also provided. See README.TXT for a complete list of sample databases and applications.

MAST is the fictional Marine Adventures and Sunken Treasures company. MAST's customers are dive shops around the world. They sell products and supplies to these shops; the shops place orders for equipment. The following tables are used to track MAST's sales:

**Table D.1** MAST tables

Table name	Information in table
CUSTOMER.DB	Customer dive shop data, including customer number, name, etc.
EMPLOYEE.DB	MAST employee information
ITEMS.DB	Specific items that makes up customer orders
NEXTORD.DB	Table that maintains the next unique order number
ORDERS.DB	Orders placed by customer dive shops
PARTS.DB	Inventory information about items on hand at MAST
VENDOR.DB	MAST suppliers that sell goods to MAST

Each table contains a primary key. To link tables, some fields and data must be duplicated among tables. A table must have a primary key or secondary index assigned to the duplicate field before it can be linked to another table. Fields that are duplicated between tables use referential integrity to make sure their values match in all tables.

The following tables describe each of the MAST database tables by showing the fields contained in the table, the type of each field, the size of the alphanumeric fields, and the fields that are key fields.

The CUSTOMER table is structured as follows:

Field	Type	Size	Key
CustNo	Numeric		*
Company	Alphanumeric	30	
Addr1	Alphanumeric	30	
Addr2	Alphanumeric	30	
City	Alphanumeric	15	
State	Alphanumeric	20	
Zip	Alphanumeric	10	
Country	Alphanumeric	20	
Phone	Alphanumeric	15	
FAX	Alphanumeric	15	
TaxRate	Numeric		
Contact	Alphanumeric	20	
LastInvoiceDate	Timestamp		

In CUSTOMER, the CustNo field is a primary key because orders in the ORDERS table must be linked to customers. The secondary index on Company is named "ByCompany". ORDERS itself contains only information about each order placed by a customer.

The structure of the EMPLOYEE table is as follows:

Field	Type	Size	Key
EmpNo	Long integer		*
LastName	Alphanumeric	20	
FirstName	Alphanumeric	15	
PhoneExt	Alphanumeric	4	
HireDate	Timestamp		
Salary	Numeric		

The ITEMS table is structured as follows:

Field	Type	Size	Key
ItemNo	Numeric		*
OrderNo	Numeric		
PartNo	Numeric		
Qty	Long integer		
Discount	Numeric		

OrderNo is a secondary index and is used in the master-detail link in the Orders form.

The NEXTORD tables is a single-column table used to generate unique, sequential order numbers. In a multi-user environment using Paradox tables, the only way to guarantee



that an order number is unique is to store the last-used number in a table, and increment it each time a new number is fetched. The table is structured as follows:

Field	Type	Size	Key
NewKey	Numeric		

The ORDERS table is structured as follows:

Field	Type	Size	Key
OrderNo	Numeric		*
CustNo	Numeric		
SaleDate	Timestamp		
ShipDate	Timestamp		
EmpNo	Long integer		
ShipToContact	Alphanumeric	20	
ShipToAddr1	Alphanumeric	30	
ShipToAddr2	Alphanumeric	30	
ShipToCity	Alphanumeric	15	
ShipToState	Alphanumeric	20	
ShipToZip	Alphanumeric	10	
ShipToCountry	Alphanumeric	20	
ShipToPhone	Alphanumeric	15	
ShipVIA	Alphanumeric	7	
PO	Alphanumeric	15	
Terms	Alphanumeric	6	
PaymentMethod	Alphanumeric	7	
ItemsTotal	Money		
TaxRate	Numeric		
Freight	Money		
AmountPaid	Money		

In ORDERS, the link to CUSTOMER is through the CustNo field. A secondary index is defined on CustNo to ensure that ORDERS and CUSTOMERS can be sorted and linked in the same order. To be sure that values entered as CustNo in ORDERS match exactly one record in CUSTOMER, referential integrity is used to constrain values entered in the CustNo field to valid customer numbers already in the CUSTOMER table.

The primary key in ITEMS is a composite of the OrderNo and ItemNo fields; these fields also have secondary indexes. This enables ITEMS to link to ORDERS (using OrderNo) in the MASTAPP Orders form. Referential integrity is used to validate the information in PARTS.

The PARTS table tracks the inventory of products. PARTS is structured as follows so that it can be linked to VENDORS:

Field	Type	Size	Key
PartNo	Numeric		*
VendorNo	Numeric		
Description	Alphanumeric	30	
OnHand	Numeric	20	
OnOrder	Numeric		
Cost	Money		
ListPrice	Money		

PartNo is the primary key for this table. Because PartNo is a secondary index, the two tables can be linked. VendorNo is the primary key of VENDORS, so it is a secondary index in PARTS and is defined to use referential integrity to make sure values entered in VendorNo match a single VendorNo entry in the VENDORS table.

The VENDORS table has the following structure:

Field	Type	Size	Key
VendorNo	Numeric		*
VendorName	Alphanumeric	30	
Address1	Alphanumeric	30	
Address2	Alphanumeric	30	
City	Alphanumeric	20	
State	Alphanumeric	20	
Zip	Alphanumeric	10	
Country	Alphanumeric	15	
Phone	Alphanumeric	15	
FAX	Alphanumeric	15	
Preferred	Logical		

Using these tables, a customer's information is entered only once, and then referred to in other tables. Likewise, order, item, inventory, and vendor information is entered only once, reducing storage requirements, and the chance for data entry error.

# Index

## Symbols

---

- " (quotation mark), naming conventions 175, 176
- \* operator 177
- + (plus) sign in property lists 103
- + operator 177
- / operator 177
- : (colon) in SQL statements 117, 176
- < operator 177
- <> operator 177
- = operator 177
- > operator 177
- ? (question mark) in SQL statements 176
- | | operator 177
- (minus) sign in property lists 104
- operator 177
- ' (apostrophe), naming conventions 175, 176

## A

---

- abandoning changes 64, 101, 107
- Abort procedure 68
- AbortOnKeyViol property 92
- AbortOnProblem property 92
- accessing
  - data 79, 89, 100, 115, 116
    - at run time 87–88
    - field values 30, 32
    - optimizing access 119
    - remote servers 127, 141
  - data sources 5, 99
  - networks and 5
  - property values 40, 86
- Active property
  - datasets 59, 101
  - queries 50, 119
- Add command (Database Desktop) 165
- Add Fields window 81
- Add method 118
- Add New Alias dialog box 172
- ADD operations 181
- adding *See* creating
- AfterCancel event 68
- AfterClose event 68
- AfterDelete event 68

- AfterEdit event 68
  - AfterInsert event 68
  - AfterOpen event 68
  - AfterPost event 68
  - aggregate functions 141, 176–177, 179
  - Alias Manager 172
  - Alias Manager dialog box (Database Desktop) 149
  - aliases 5, 149, 171–174
    - changing 173
    - defined 149
    - deleting 174
    - local SQL and 175
    - predefined 20
    - queries 168
      - heterogenous 124
    - retrieving defaults 135
    - retrieving information 57
    - saving 172, 173
    - specifying 134, 172
  - Aliases command (Files) 149
  - Aliases page (BDE Configuration Utility) 172
  - AliasName property 135
  - alignment, text 108
  - Alignment property 33, 83, 108
  - ALTER TABLE keyword 181
  - altering *See* changing
  - ancestor objects 55
  - AND operator 177
  - ANSI/OEM conversion 93
  - ANSI-standard SQL naming conventions 175–176
  - Answer tables 170
  - apostrophe ('), naming conventions 175, 176
  - Append method 64
  - AppendRecord method 65
  - applications 79, 93
    - building 19
    - client/server 14, 118
    - deploying 16–18
    - remote servers 142–145
    - developing 12–16
      - client/server 127–129
      - fundamentals 13
    - displaying multiple views 95–97
    - multiuser 5, 135, 147
    - updating data 65
  - optimizing 128
  - sample
    - LINKQRY 123
    - MASTAPP 20, 183
    - TWOFORMS 95
  - starting 173
  - upsizing 140–142
- ApplyRange method 51, 73
- arithmetic operators 177
  - arrays 65, 72, 73
    - indexes 74
    - queries 122
    - stored procedures 139
  - AsBoolean property 35
  - ASCII files 6, 70
  - AsDateTime property 35
  - AsFloat property 35
  - AsInteger property 35
  - assigning values 37, 154
    - at run time 87, 88
    - calculated fields 68–69, 82–83
    - default 109, 154
    - names as 122
    - required 154
    - run-time 35
    - to buttons 114
    - to combo boxes 109
  - assignment statements 86
  - AsString property 35
  - asynchronous communication module 144
  - attaching to servers 5, 8, 56
    - remote databases 130–133
      - Connected property and 134
      - connection
        - parameters 131–132
        - login parameters 133
  - attributes *See* display attributes; properties
  - AUTO ODBC setting 132
  - AutoCalcFields property
    - OnCalcFields event and 68
  - AutoDisplay property 108, 109
  - AutoEdit property 22, 77, 101
  - automatically translating data 12
  - AutoSize property 102
  - AutoUnload property 88
  - averages 176
  - AVG function 176

## B

batAppend constant 89  
batAppendUpdate constant 89  
batch move operations 8, 75, 89–92  
    error handling 92  
BatchMove method 75  
batCopy constant 89  
batDelete constant 89  
batUpdate constant 89  
BDE (Borland Database Engine) 2, 16–18, 92  
    retrieving information 57  
BDE Configuration Utility 5, 171–174  
    starting 171  
BDE Configuration Utility Window 171  
BeforeCancel event 68  
BeforeClose event 63, 68  
BeforeDelete event 68  
BeforeEdit event 68  
BeforeInsert event 68  
BeforeOpen event 68  
BeforePost event 68  
binary collation sequence 142  
binary large objects *See* BLOBs  
bitmaps 99, 108, 110  
BLOB data 107  
    type compatibility 181  
BLOBs 99, 108  
BLROM800.LD 142  
BOF property 61  
bookmarks 66–67  
Boolean properties 84  
Boolean types 86  
Borland Database Engine *See* DBE  
Borland SQL Links 142, 170  
borrowing table structures 155  
Browse mode (datasets) 58  
BRPARTS.DFM 22  
Btrieve databases 132  
buffers 60  
    clearing 107  
    flushing 67  
building applications 19  
building databases 12–16, 147  
building forms 19–30  
    multi-table 24–30  
    single-table 21–24  
buttons 78  
    *See also* specific type  
    adding 99, 100  
    assigning values to 114

    associating strings with 114  
    navigator 106–107  
    enabling/disabling 107

## C

CalcFields mode (datasets) 58, 69  
calculated fields 9, 37–39, 58  
    assigning values 68–69, 82–83  
    defining 81, 82, 83  
    ordering 179  
    updating 38  
Calculated property 33, 83  
Cancel method 64  
Cancel navigator button 107  
canceling current operation 64, 101, 107  
canceling method calls 68  
CancelRange method 51, 73  
CanModify property 63, 75, 120  
Caption property  
    check boxes 113  
case  
    conversions 177  
    sensitivity 75  
centering data 83  
ChangedTableName property 92  
changing  
    *See also* editing  
    aliases 173  
    component names 23  
    data 63–66, 100, 101  
        abandoning changes 64, 101, 107  
        committing changes 135, 136, 137  
        edit boxes and 36  
        rolling back changes 136  
        single-table queries 48  
    datasets 78  
    events 82, 83, 85  
    property values 82, 83–84  
    table structures 159–161  
    tables 181  
    values *See* updating values  
character sets 89  
character strings *See* string lists; strings  
check boxes 100, 113  
checking for null values 73  
child tables 157  
Clear method 118  
clearing data buffers 107  
client platforms 16  
client/server applications 14, 118, 127–145  
    deploying 142–145  
    developing 127–129  
    handling security 132–133  
    transaction control 135–138  
client-based SQL 175  
Clipboard 108  
Close method  
    datasets 60  
    queries 118, 122  
code 23, 87  
    changing data 36  
    optimizing 61  
collation sequence 142  
colon (:) in SQL statements 117, 176  
column headings *See* field names  
columns  
    *See also* field names; fields  
    combo boxes and multiple 111  
    outlining 104  
combo boxes 99, 109–112  
    assigning values 109  
    multiple columns 111  
    sorting items 110  
    user input and 44–46  
Commit method 136  
COMMIT statements 135  
committing changes 135, 136, 137  
communication protocols 128, 131  
comparison operators 177  
Component palette 6  
components 2  
    data-aware 10, 87, 100  
        disabling/enabling 67  
    database 6, 8, 11, 55–56, 133  
    dataset 100  
    field 30–32  
    referencing 34  
    renaming 23  
composite keys 153  
concatenation operator 177  
configuration files 171  
configuration options 173  
configuring drivers 131  
confirming deletions 68, 104  
Connect method 89  
CONNECT.EXE 143  
CONNECT.HLP 143  
Connected property 134

- connections (server) 5, 56
  - inactive 56
  - remote databases 8, 130–133
    - Connected property and 134
    - connection
      - parameters 131–132
      - login parameters 133
- controls
  - See also* components
  - data 99–114
  - read-only 101, 108
    - graphics 109
- conversions
  - batch move operations and 91
  - case 177
  - numbers to strings 35
  - strings to date/time 173
  - strings to numbers 36, 173
- Copy command (Database Desktop) 166
- Copy dialog box (Database Desktop) 166
- copying
  - data 75, 166
  - table structures 8, 75
  - text 108
- CopyToClipboard method 108
- COREWIN.DLL 142
- COUNT function 177
- counting function 177
- CREATE INDEX keyword 182
- Create Table dialog box (Database Desktop) 151–152
  - SQL tables 155
- CREATE TABLE keyword 180
- creating
  - forms 19–30
    - multi-table 24–30, 93–95
    - single-table 21–24
  - indexes 182
  - new tables 89, 151, 155
  - Paradox tables 151, 158
  - SQL tables 155
- csDropDown constant 110
- csDropDownList constant 110
- csOwnerDrawFixed
  - constant 110
- csOwnerDrawVariable
  - constant 110
- csSimple constant 110
- Currency property 83
- current date and time 46
- current field 4, 165
- current operation, canceling 101, 107

- current record 4, 60, 104, 165
  - indicator 104
  - setting 106
- current values 34, 102
- cursors 60
  - moving 60–62, 72, 78
    - flickering screens and 67
- CUSTOMER.DB 24, 183
- customizing tables 162–164
- CUSTORD.DFM 24, 31
- cutting graphics 108
- cutting text 108
- CutToClipboard method 108

## D

- data
  - accessing 79, 89, 100, 115, 116
    - at run time 87–88
    - field values 30, 32
    - optimizing access 119
    - remote servers 127, 141
  - adding 100–101, 154
    - automatically 158
    - predefined values 109
    - to grids 103
  - automatically translating 12
  - changing 63–66, 100, 101
    - abandoning changes 64, 101, 107
    - committing changes 135, 136, 137
    - edit boxes and 36
    - rolling back changes 136
    - single-table queries 48
  - confirming deletions 68, 104
  - copying 75, 166
  - defined 5
  - deleting *See* deleting
  - displaying 9, 83, 161–164
    - at run time 87
    - current values 34, 102
    - in grids 83, 103, 105
    - multiple views 95–97
  - display-only 82, 84, 102, 107
    - truncated 102
  - editing 63–64, 83, 99, 102
    - canceling current operation 64, 101, 107
    - Database Desktop and 164–165
    - in grids 104
    - restrictions 164
  - filtering 50–51
  - formatting 39–42, 83, 85
    - at run time 40–42
  - invalid, restricting 43
  - porting 127–129
  - retrieving 8, 72–74
    - from multiple tables 178, 179
    - specific values 179
    - specifying conditions 178
  - searching for 42–43, 70–72
    - in unkeyed fields 47
    - nearest match 71, 72
  - sorting 74, 152, 166, 173
  - typing to replace 164
  - updating 63, 65–66, 101, 107
    - batch move operations 89
    - calculated fields 38
    - handing events 78
    - multiuser applications 65
    - read-only results 120
    - Refresh method and 67
  - validating 43–47
- Data Access page (Component palette) 7–10
- data buffers 60
  - clearing 107
  - flushing 67
- data controls 99–114
  - inactive 51, 77
  - linking to datasets 67, 100, 101
- Data Controls page (Component palette) 10–11, 99–100
- data definition 13
- Data Definition Language 180–182
- data entry forms 102, 158, 183
- data grids 99, 103–106
  - adding 103
  - displaying data 83
  - editing data 104
  - inserting records 104
  - linking to datasets 103
  - options 103–104
- Data Manipulation Language (DML) 176–179
- data masks 84
- data sources 2, 77–78
  - accessing 5, 99
    - networks and 5
  - adding 100
  - changing datasets 78
  - handling events 78
  - lists 111
  - supported 6
- data templates 83, 84, 154
- data types *See* types
- data-aware components 10, 87, 100
  - disabling/enabling 67

- database components 6, 8, 11, 55–56, 133
- Database Desktop 147–170
  - adding tables 149–161
  - copying data 166
  - defining fields 152
  - deleting records 165
  - editing data 164–165
  - executing SQL
    - statements 167–169
  - inserting records 165
  - managing files 148–149
  - opening tables 161
  - overview 12
  - password protection 159
  - saving SQL statements 169
  - sorting data 152, 166
  - specifying field types 152
  - starting 147
  - viewing data 161–164
- Database Desktop window 148
- database engines 2, 16, 115, 129
- Database Form Expert 21
  - building multi-table forms 24
  - Fields Editor vs. 32
  - opening tables at run time 23
  - renaming components 23
  - running forms 21
  - starting 21
- Database Form Expert command (Help) 21
- Database Management System (DBMS) 1
- database navigator 99, 106–107
  - buttons 106–107
  - enabling/disabling 107
- Database Properties Editor 134
- database servers 5, 6, 127
  - aliases and 149
  - connecting to *See* connections
  - developing applications *See* client/server applications
- database tools 1
- DatabaseLogin event 133
- DatabaseName property 29, 69, 116, 135
  - heterogenous queries 124
  - stored procedures 138
- databases 4–6, 89
  - accessing data 100
  - architecture 6–12
  - building 12–16, 147
  - defined 4
  - overview 2–4
  - printing reports 8, 52–53
  - relational 150
    - retrieving information 57
    - sample 183–186
- Databases property 56
- DataField property 45, 100
  - check boxes 113
  - combo boxes 110
- dataset components 100
- DataSet property 23, 77
- datasets 57–69, 113
  - changing 78
  - closing 59
  - current state 59
  - data sources and 77
  - defining 32
  - handling events 68–69
  - linking to data controls 67, 100, 101
  - linking to data grids 103
  - modes 58–59
  - moving through 60–62
  - opening 59
  - rights and privileges 75
- DataSource property 100, 123
- date functions 177
- Date Manager page (BDE Configuration Utility) 173
- dates 40, 85, 173
  - conversion functions 86
  - getting current 46
  - returning 177
- DAY constant 177
- .DB files 6, 70
- DB2 databases 132
- dBASE tables 5, 6, 70, 175
  - batch move operations and 90
  - deleting 182
  - indexes 74, 153, 182
  - memo fields 108
  - naming 175
  - retrieving data 73
  - searching for data 70, 72
  - setting ranges 50
  - specifying directories 69, 149
  - type compatibility 180–181
- DBD.EXE 147
- DBDEMOS alias 20
- .DBF files 6, 70
- DBHandle function 93
- DBIPROCS.PAS 93
- DBITYPES.PAS 93
- DBLocale function 93
- DBMS 1
- DBNMP3.DLL 143
- DDL *See* Data Definition Language
  - default settings 85, 173
- default values 109, 154
- DefaultDrawing property 105
- Define Field dialog box 38
- Define Index dialog box (Database Desktop) 155, 156
- Define Parameters dialog box 49
- Delete buttons 23
- Delete command (Database Desktop) 167
- Delete dialog box (Database Desktop) 167
- DELETE keyword 179
- Delete method 64
- Delete navigator button 107
- DeleteTable method 75
- deleting
  - aliases 174
  - characters from strings 177
  - fields 81, 181
  - graphics 108
  - indexes 182
  - queries 166
  - records 64, 75, 104, 165
    - batch move operations 89
  - tables 75, 166, 182
  - text 108
- DEPLOY.TXT 142
- deploying applications 16–18
  - remote servers 142–145
- designing forms 100
- desktop databases 12
  - upsizing 130, 140–142
- destination tables 89–92
- detail tables 24, 76–77, 93–95
- developing applications 12–16
  - client/server 127–129
  - fundamentals 13
- dgAlwaysShowEditor
  - constant 104
- dgAlwaysShowSelection
  - constant 104
- dgColLines
  - constant 104
- dgColumnResize
  - constant 104
- dgConfirmDelete
  - constant 104
- dgEditing
  - constant 104
- dgIndicator
  - constant 104
- dgRowLines
  - constant 104
- dgRowSelect
  - constant 104
- dgTabs
  - constant 104
- dgTitles
  - constant 104
- .DIB files 108
- directories
  - private (:PRIV:) 149
  - specifying paths 57, 69
    - aliases and 149
  - working (:WORK:) 149

DisableControls method 50, 51, 61, 67  
disabling data-aware components 67  
disabling navigator buttons 107  
display attributes 30, 83–84  
DisplayFormat property 33, 39, 41, 83, 103  
displaying data 9, 83, 161–164  
    at run time 87  
    current values 34, 102  
    in grids 83, 103, 105  
    multiple views 95–97  
displaying large images 109  
displaying property values 32, 103  
DisplayLabel property 33, 83  
display-only data 82, 84, 102, 107  
    truncated 102  
DisplayText property 40  
DisplayWidth property 33, 83  
distributing applications 16–18  
distribution rights 16  
dmAutomatic constant 105  
DML statements *See* Data Manipulation Language  
dmManual constant 105  
domains 138  
double quote ("), naming conventions 175, 176  
DragMode property 105  
drawing routines 105  
Driver Manager page (BDE Configuration Utility) 173  
DriverName property 135  
drivers  
    configuring 131  
    language 142, 143  
    ODBC 132, 138  
    retrieving information 57  
DROP INDEX keyword 182  
DROP operations 181  
DROP TABLE keyword 182  
DropConnections method 57  
drop-down lists *See* combo boxes; lists boxes  
DropDownCount constant 110  
dropping *See* deleting  
dsBrowse constant 59  
dsCalcFields constant 59  
dsEdit constant 59  
dsInactive constant 59  
dsInsert constant 59  
dsSetKey constant 59  
DSQL 121–124  
    defined 117

dynamic queries 48  
dynamic SQL statements 121–124  
    defined 117

## E

---

edit boxes 99, 102  
    *See also* combo boxes  
    changing data 36  
Edit Data command (Database Desktop) 164  
Edit method 63  
Edit mode 107  
    Database Desktop 164, 165  
    datasets 58, 63, 77  
    CanModify property and 63  
Edit navigator button 107  
EditFormat property 83, 103  
editing  
    *See also* changing data 63–64, 83, 99, 102  
    canceling current operation 64, 101, 107  
    Database Desktop and 164–165  
    in grids 104  
    restrictions 164  
    graphics 108  
    text 108  
editing methods 108  
EditMask property 33, 41, 83, 84  
editors  
    Database Properties 134  
    Fields *See* Fields Editor  
    Input Mask 84  
    Parameters 121, 122, 138  
    SQL 167–169  
    String List 109, 117  
EditRangeEnd method 73  
EditRangeStart method 73  
EDORDERS.DFM 38  
EMPLOYEE.DB 183  
Empty command (Database Desktop) 165  
EmptyTable method 75  
EnableControls method 50, 51, 61, 67  
Enabled property 77, 101  
entering SQL statements 117–119, 120  
    at run time 118, 122  
EOF property 61  
errors 92  
Esc key 101

events 83, 101  
    changing 82, 83, 85  
    data grids 105–106  
    data sources 78  
    datasets 68–69  
Events tab (Object Inspector window) 85  
example applications *See* sample applications  
exceptions 51, 101  
exclusive locks 75  
Exclusive property 75  
ExecProc method 139  
ExecSQL method 119  
Execute method 92  
executing SQL statements 116, 117, 119, 169  
    at run time 118  
experts *See* Database Form Expert  
expressions  
    aggregate functions and 177  
EXTRACT function 177

---

## F

fetching data *See* retrieving data  
field components 30–32  
Field Link Designer 76–77  
Field Link Designer dialog box 76  
field names 83  
    ANSI-standards 176  
    assigning as values 122  
    disabling 104  
    resizing columns 163  
field properties 32–34  
    changing 83  
field types *See* types  
field values *See* values  
field view 165  
FieldByName method 87–88  
FieldName property 33, 83  
fields 4, 30–43  
    adding 31, 79, 181  
    Fields Editor 81  
    calculated 9, 37–39, 58  
    assigning values 68–69, 82–83  
    defining 81, 82, 83  
    ordering 179  
    updating 38  
    current 4, 165  
    default settings 85  
    defining 152  
    deleting 81, 181  
    display-only 82, 84

- entering data 154
  - automatically 158
- handling events 85
- key 83, 152
  - partial keys and 73
  - specifying 50
- moving 163
  - at run time 105
- numeric 83, 154, 176
  - displaying values 83
  - in edit boxes 35, 36
- placing scroll locks 162
- reordering 31, 103, 163
  - at run time 105
  - Fields Editor 80
- resizing 104, 163
- selecting 104, 165
  - Tab key and 101, 104
- setting display attributes 83–84
- updating *See* updating data
- Fields Editor 30–32, 80–83, 100
  - activating 9
  - adding fields 31, 81
  - changing property values 82
  - closing 31
  - Database Form Expert vs. 32
  - defining calculated fields 37
  - deleting fields 81
  - editing events 82
  - overview 10
  - reordering fields 80
  - setting property values 32
  - starting 80
- Fields Editor window 80
- Fields property 87
- file servers 5, 127
  - aliases and 149
  - connecting to *See* connections
  - developing applications *See* client/server applications
- files
  - See also* applications;
  - databases
  - configuration 171
  - distributing 16, 17
  - opening 148
  - script 118
  - temporary 57, 92, 149, 170
  - text 6, 70
- filters 50–51
- finding field values 42, 70–72
  - nearest match 71, 72
- FindKey method 72
- FindNearest method 72
- First method 60
- First navigator button 106

- flickering screens 67
- floating-point numbers 86
- flushing data buffers 67
- focus, setting 85
- FocusControl method 85
- Form Expert *See* Database Form Expert
- FormatDateTime routine 85
- FormatFloat routine 85
- FormatInteger routine 85
- formatting data 39–42, 83, 85
  - at run time 40–42
- formatting text 40
- forms 19–30
  - data entry 102, 158, 183
  - designing 100
  - multi-table 24–30, 76
    - creating 93–95
  - order entry 93, 112
  - printing 53
  - running 21
  - single-table 21–24
- FreeBookmark method 67
- freeing memory 67
- functions
  - See also* methods; routines
  - aggregate 141, 176–177, 179
  - date 177
  - mathematical 141
  - search 70
  - string manipulation 177

## G

- .GDB files 6
- GDS.DLL 143
- GetAliasNames method 57
- GetAliasParams method 57
- GetBookmark method 67
- GetDatabaseNames method 57
- GetDriverNames method 57
- GetDriverParams method 57
- GetIndexNames method 74
- GetTableNames method 57
- GoToBookmark method 67
- GoToCurrent method 76
- GoToKey method 70
- GoToNearest method 70, 71, 72
- graphics 108
  - displaying large 109
  - editing 108
- grids *See* data grids

## H

- Handle function 93
- handles 93

- headings *See* field names
- Help Hints 107
- heterogeneous joins 179
- heterogenous queries 124
- hierarchy 7, 55
- Hints property 107
- horizontal dividing lines,
  - enabling/disabling 104
- HOUR constant 177

## I

- icons
  - Data Access page 7
  - Data Controls page 10
- IDAPI.CFG 171
- IDAPI.CNF 17
- IDAPI01.DLL 17
- IDAPICFG.EXE 17, 171
- IDAPICFG.HLP 17
- IDASCI01.DLL 17
- IDBAT01.DLL 17
- IDDBAS01.DLL 17
- IDODBC01.DLL 17
- IDPDX01.DLL 17
- IDQRY01.DLL 17
- IDR10009.DLL 17
- ILD01.DLL 17
- image boxes 99
- images 108
  - scrolling through 109
- inactive data controls 51, 77
- inactive database connections 56
- Inactive mode (datasets) 58
- Index property 33, 83
- indexes 74–75, 152–154, 180
  - See also* key fields
  - creating 182
  - deleting 182
  - naming 156
  - primary 74
  - secondary 71, 72, 74, 154, 182
  - SQL tables 153–154, 155–157, 160
- IndexFieldCount property 74
- IndexFieldNames property 50, 71
- IndexFields property 74
- IndexName property 71, 74
- Informix servers 143
- Informix tables 6, 90, 91
  - transactions 138
- inheritance 55
- InitialValues property 88
- inner joins 178
- input focus 85



Input Mask Editor 84  
Input Mask Editor window 84  
INSERT keyword 179  
Insert method 64  
Insert mode (datasets) 58, 64,  
106  
    CanModify property and 63  
Insert navigator button 106  
InsertRecord method 65  
instantiating TField 103  
integers 86  
IntegralHeight property 110  
integrity violations 92  
interactive SQL 129  
INTERBAS.MSG 143  
InterBase Server (LIBS) 127,  
129–130  
InterBase tables 5, 6, 130  
    accessing 143  
    batch move operations  
    and 91  
    transactions 138  
InterBase Workgroup Server for  
Windows NT 144  
International settings 85  
invalid data, restricting 43  
invoices 44, 113  
IS NULL operator 177  
ISAM.IEM 143  
isolation levels  
    (transactions) 137–138  
ISQL 129  
ItemHeight property  
    combo boxes 110  
    list boxes 111  
Items property 45  
    combo boxes 109  
    list boxes 110, 113  
    radio buttons 114  
ITEMS.DB 183  
iteration 61, 62, 87

## J

joins 178  
    heterogeneous 179

## K

KeepConnections property 56,  
134  
key fields 83, 152  
    *See also* indexes  
    partial keys and 73  
    specifying 50  
key violations 92

keyboard events 101, 106  
KeyExclusive property 72  
    ranges and 74  
KeyFieldCount method 71  
KeyViolTableName property 92  
keywords  
    *See also* SQL keywords  
    naming conventions  
    and 175, 176

## L

labels 101–102  
language drivers 142, 143  
large images 109  
Last method 60  
Last navigator button 106  
LDLLSQLW.DLL 143  
left justifying data 83  
LIBS *See* InterBase Server  
linking data controls to  
    datasets 67, 100, 101  
linking data grids to datasets 103  
linking queries 123  
linking tables 94, 150  
    one-many-many links 27  
    one-to-many links 24, 76  
LINKQRY sample  
    application 123  
list boxes 99, 100, 109, 110  
    *See also* combo boxes  
lists 111  
    scrolling through 110, 112  
    sorting items 110  
    string *See* string lists  
    user input and 44–45  
live results 48, 120  
    *See also* SQL queries  
LoadFromFile method 118  
local buffers 60  
    clearing 107  
    flushing 67  
Local Interbase Server (LIBS) *See*  
InterBase Server  
Locale function 93  
locating field values 42, 70–72  
    nearest match 71, 72  
locking tables 75  
loColLines constant 111  
logical operators 177  
Login dialog box 132  
login parameters 133  
LoginPrompt property 133  
lookup lists 111  
lookup tables 45–46, 158  
LookupDisplay property 45, 111

LookupField property 45, 111  
LookupSource property 45, 111  
loops 61, 62  
loRowLines constant 111  
loTitles constant 111  
LOWER function 177  
lowercase conversions 177

## M

M3OPEN.DLL 144  
M3OPEN.EXE 144  
maintaining database  
    connections 134  
Mappings property 91  
masks 84  
MAST sample database 183–186  
MASTAPP sample  
    application 20, 183  
master tables 24, 76–77, 93–95  
MasterFields property 76  
MasterSource property 76, 94  
mathematical functions 141  
MAX function 177  
maximum values 51, 83, 154, 177  
MaxLength property 108  
MaxValue property 83  
MBW.EXE 144  
memo boxes 99  
memo fields 108  
memory, freeing 67  
memos 107–108  
    scrolling through 108  
menu items 78  
messages 71  
metadata 138, 141  
methods  
    *See also* functions; procedures  
    bookmarks 67  
    canceling calls 68  
    editing 108  
MFTPEX.EXE 144  
MHPARPA.DLL 144  
Microsoft Access databases 132  
Microsoft Named Pipes  
    protocol 144  
Microsoft SQL servers 6, 142  
    transactions 138  
MIN function 176  
minimum values 51, 83, 154, 176  
minus (–) sign in property  
    lists 104  
MINUTE constant 177  
MinValue property 84  
MNETONE.EXE 144  
MNOVLWP.DLL 144

Mode property 89  
modifying *See* changing  
monetary values 83, 85  
MONTH constant 177  
mouse events 101, 105  
MoveBy method 62  
MoveCount property 92  
moving  
    cursors 60–62, 72, 78  
    fields 163  
        at run time 105  
    text 108  
moving among fields 104, 165  
    Tab key and 101, 104  
moving through images 109  
moving through lists 110, 112  
moving through memos 108  
moving through tables 60–62,  
    104  
    Database Desktop and 162,  
    165  
MPATHWAY.DLL 144  
MPCNFS.EXE 144  
MPCNFS2.EXE 144  
MPCNFS4.DLL 144  
MSOCKLIB.DLL 144  
multi-line text 107  
multiple query statements 118  
multi-table forms 24–30, 76  
    creating 93–95  
    Paradox tables 157, 158  
multi-table queries 124, 178  
    heterogeneous joins 179  
multiuser applications 5, 135,  
    147  
    *See also* client/server  
    applications; networks  
    updating data 65  
MVWASYNC.EXE 144  
MWINTCP.EXE 144

---

**N**

---

Name property 32, 33, 84  
naming  
    directories 149  
    fields 176  
    indexes 156  
    tables 175  
navigator *See* database navigator  
nbDelete constant 23  
nbNext constant 23  
NetFileDir property 57  
networks 127, 144, 149  
    accessing 5  
    temporary files and 149

Next method 60  
Next navigator button 106  
Next Record buttons 23  
NEXTORD.DB 183  
nil values 65  
NOT operator 177  
NOT SHARED setting 137  
Novell SPX/IPX interface 131,  
    144  
Now function 46  
null values 65, 121  
    checking for 73  
    stored procedures 139  
Number Manager page (BDE  
    Configuration Utility) 173  
numbers  
    conversion functions 86  
    converting  
        strings to 36, 173  
        to strings 35  
numeric fields 83, 154, 176  
    displaying values 83  
    in edit boxes 35, 36  
NWCALLS.DLL 144  
NWIPXSPX.DLL 144

---

**O**

---

object hierarchy 7, 55  
Object Inspector  
    displaying property  
    values 32, 33, 103  
ODBC data sources 6  
ODBC driver 132, 138  
ODBC.NEW 17  
ODBCINST.NEW 17  
OEM conversion 93  
OnCalcFields event 38, 58,  
    68–69, 82  
OnColEnter event 105  
OnColExit event 105  
OnDataChange event 78  
OnDbClick event 105  
OnDragDrop event 105  
OnDragOver event 105  
OnDrawDataCell event 105, 106  
one-many-many links 27  
OnEndDrag event 106  
OnEnter event 106  
one-to-many links 24, 76  
OnExit event 106  
OnGetText event 85  
OnKeyDown event 106  
OnKeyPress event 106  
OnKeyUp event 106  
OnLogin event 133

OnNewRecord event 68  
OnSetText event 85  
OnStateChange event 59, 78  
OnUpdateData event 78  
OnValidate event 43, 46  
Open command (Database  
    Desktop) 148, 161  
Open method  
    datasets 59  
    queries 119  
Open Table dialog box (Database  
    Desktop) 161  
opening  
    datasets 59  
    files 148  
    queries 168  
    tables 161  
        at run time 23  
operators (local SQL  
    statements) 177  
optimizing applications 128  
optimizing code 61  
optimizing queries 121  
Options property  
    combo boxes 111  
    data grids 103  
OR operator 177  
ORA6WIN.DLL 142  
ORA7WIN.DLL 142  
Oracle servers 142  
Oracle tables 6, 90, 91  
    stored procedures 140  
    transactions 138  
ORAW850.LD 142  
order entry forms 93, 112  
ORDERS.DB 24, 183  
OS.IEM 143  
outer joins 178  
outlining records and fields 104  
output 53  
Overload property 140  
overloading stored  
    procedures 140

---

**P**

---

Paradox tables 5, 6, 70, 175  
    batch move operations  
    and 90  
    creating 151, 158  
    deleting 182  
    indexes 74, 152, 154, 182  
    key fields 152  
    memo fields 108  
    multi-table forms 157  
        lookup tables and 158  
    naming 175

- password protection 159
- referential integrity 157–158
- restructuring 159
- retrieving data 73
- searching for data 70, 72
- setting ranges 50
- specifying directories 69, 149
- type compatibility 180–181
- validity checks 154
- ParamByName method 122
- ParamByName property 138
- parameterized statements 117
- parameters
  - connection 131–132
  - login 133
  - retrieving information 57
  - SQL statements 117, 121, 176
    - arrays and 122
    - assigning names 122
    - null values 121
    - setting at run time 121
  - stored procedures 138–139
- Parameters Editor 121, 122, 138
- Params property 50, 122, 131
- parent tables 157
- parsing query statements 121
- partial keys 73
- PARTS.DB 183
- passthrough SQL 120, 136, 167
  - transactions 136, 137
  - isolation levels 137
- passwords 159
- PasteFromClipboard
  - method 108
- pasting graphics 108
- pasting text 108
- picture strings 83, 84, 154
- placing bookmarks 66–67
- placing locks 75
- platforms 128
- plus (+) sign in property lists 103
- pointers 60
- porting data 127–129
- Post method 63, 64
  - Edit method and 64
  - OnCalcFields event and 69
- Post navigator button 107
- posting records 63, 64, 102
  - automatically 107
  - data grids 105
- predefined aliases 20
- predefined values 109
- Prepare method 116, 121–122, 139
- Preview property 52
- primary indexes 74, 152, 182
- Print method 53
- printing forms 53
- printing reports 8, 52–53
- Prior method 60
- Prior navigator button 106
- private directories (:PRIV:) 149
- Private Directory command (Database Desktop) 149
- PrivateDir property 57
- privileges 75
- ProblemTableName property 92
- procedures
  - See also* methods; routines
  - stored 8, 57, 138–140
    - executing 139
    - overloading 140
- project files 16
- projects *See* applications
- properties 6, 83
  - accessing 40, 86
  - Boolean 84
  - changing 82, 83–84
  - field 32–34, 83
  - values, displaying 32, 103
- Properties menu (Database Desktop) 163
- Properties tab (Object Inspector window) 83
- protocol (communications) 128, 131

## Q

- QBE (defined) 169
- queries 9, 47–50, 169–170
  - See also* SQL queries
  - deleting 166
  - dynamic 48
  - multi-table 124, 178
    - heterogeneous joins 179
  - opening 168
  - renaming 166
  - single-table 48
  - specifying table names 9
  - static 47
- query by example (defined) 169
- Query component 115, 117
- question mark (?) in SQL statements 176
- quotation mark ("), naming conventions 175, 176

---

- R
- radio buttons 100, 114
- ranges 50–51, 73
  - canceling 51
  - maximum values 51
  - minimum values 51
- RDS.IEM 143
- read-only controls 101, 108
  - graphics 109
- read-only privileges 75
- ReadOnly property 33, 75, 84
  - CanModify property and 63
- read-only results 48, 120
  - See also* SQL queries
- ReCalcReport method 89
- records
  - adding 64, 65, 106, 165
    - batch move operations 89
    - data grids and 104
  - current 4, 60, 104, 165
    - indicator, setting 104
    - setting 106
  - data grids and 103, 105
  - deleting 64, 75, 104, 165
    - batch move operations 89
  - moving through 60–62
  - ordering 152, 179
  - outlining 104
  - posting 63, 64, 102, 105
    - automatically 107
  - resizing 163
  - selecting 104, 106, 165
    - specific sets 47, 50
  - updating *See* updating data
- Redistributable BDE 16
- redrawing screens 105, 108, 109
- referencing components 34
- referential integrity 157–158
- Referential Integrity dialog box (Database Desktop) 158
- Refresh method 67
- Refresh navigator button 107
- relational databases 150
  - See also* databases
- releasing memory 67
- remote servers 5, 127
  - aliases and 149
  - connecting to 8, 130–133, 134
  - deploying applications 142–145
  - developing applications 12, 15
- REMOTE.DLL 143
- removing *See* deleting
- Rename command (Database Desktop) 166

- Rename dialog box (Database Desktop) 166
  - renaming
    - components 23
    - queries 166
    - tables 166
  - reordering fields 31, 103, 163
    - at run time 105
    - Fields Editor 80
  - repeat. . . until loops 62
  - ReportDir property 52, 88
  - ReportName property 52, 88
  - reports 8, 88–89
    - printing 52–53
  - ReportSmith 8, 52, 88
    - deploying applications 145
    - printing reports 52–53
  - ReportSmith Runtime 145
  - RequestLive property 48, 120
  - Required property 43
  - required values 154
  - reserved words
    - See also* SQL keywords
    - naming conventions and 175, 176
  - resizing fields 104, 163
  - resizing records 163
  - restoring previous values 64, 101, 107
  - Restructure command (Database Desktop) 159
  - Restructure Table dialog box (Database Desktop) 159, 160
  - restructuring tables 159–161
  - retrieving data 8, 72–74
    - from multiple tables 178, 179
    - specific values 179
    - specifying conditions 178
  - right justifying data 83
  - Rollback method 136
  - ROLLBACK statements 135
  - rolling back changes 136
  - rounding 86
  - routines
    - See also* functions; methods; procedures
    - drawing 105
  - rows *See* records
  - RS\_RUN directory 52
  - Run method 52, 88
  - Run SQL command (Database Desktop) 169
  - Run Visual Query Builder
    - command 118
  - RunMacro method 88
  - running SQL statements 116, 117, 119, 169
    - at run time 118
- ## S
- 
- sample applications
    - LINKQRY 123
    - MASTAPP 20, 183
    - TWOFORMS 95
  - sample database 183–186
  - saving
    - aliases 172, 173
    - SQL statements 118, 169
  - screens
    - flickering 67
    - redrawing 105, 108, 109
  - script files 118
  - scroll bars 108
  - scroll locks 162
  - ScrollBars property 108
  - scrolling through images 109
  - scrolling through lists 110, 112
  - scrolling through text 108
  - search conditions (SQL queries) 178
  - search functions 70
  - search mode 70
  - searching for data 42–43, 70–72
    - in unkeyed fields 47
    - nearest match 71, 72
  - SECOND constant 177
  - secondary indexes 154
    - deleting 182
    - searching on 71, 72, 74
  - security 132–133
  - SECURITY.IEM 143
  - Select Alias command (Database Desktop) 168
  - Select Alias dialog box (Database Desktop) 168
  - SELECT keyword 178
  - SELECT statements 118, 120, 178–179
  - selecting
    - See also* searching for data
    - fields 104, 165
      - Tab key and 101, 104
    - records 104, 106, 165
      - specific sets 47, 50
  - selection bars (grids) 104
  - servers 5, 6, 127
    - aliases and 149
    - connecting to *See* connections
    - developing applications *See* client/server applications
  - Session component 56
  - set functions 176–177, 179
  - SetFields method 65
  - SetKey method 70, 71
  - SetKey mode (datasets) 58, 70
  - SetRange method 73
  - SetRangeEnd method 51, 73
  - SetRangeStart method 51, 73
  - setting locks 75
  - setting property values 32
  - SetVariable method 89
  - SHARED AUTOCOMMIT
    - setting 137
  - SHARED NOAUTOCOMMIT
    - setting 137
  - Show SQL command (Database Desktop) 168
  - ShowHint property 107
  - ShowMessage function 71
  - single quote ('), naming conventions 175, 176
  - single-table forms 21–24
  - single-table queries 48
  - Size property 84
  - Sort command (Database Desktop) 166
  - sort order 74
  - Sort Table dialog box (Database Desktop) 166
  - Sorted constant 110
  - sorting data 74, 152, 166
    - configuration options 173
  - sorting items in lists 110
  - source tables 89–92
  - specifying aliases 134, 172
  - spreadsheets 99
  - SQL database servers 6
  - SQL Editor (Database Desktop) 167–169
    - saving statements 169
    - starting 168
  - SQL Editor window (Database Desktop) 167
  - SQL keywords
    - ALTER TABLE 181
    - COMMIT 135
    - CREATE INDEX 182
    - CREATE TABLE 180
    - DELETE 179
    - DROP INDEX 182
    - DROP TABLE 182
    - INSERT 179
    - naming conventions 175, 176
    - ROLLBACK 135
    - SELECT 178
    - UPDATE 179

- SQL Links 142
  - SQL property 116, 117, 118
  - SQL queries 9, 115–125
    - adding new fields 181
    - creating indexes 182
    - deleting fields 181
    - entering statements 117–119, 120
      - at run time 118, 122
    - heterogenous 124
    - linking 123
    - live vs. read-only results 120
    - local (client-based) 175–182
      - data definition 180–182
      - data manipulation 176–179
      - naming conventions 175–176
      - operators 177
    - multi-table 124, 178
      - heterogeneous joins 179
    - optimizing 121
    - preparing 121–122
    - retrieving specific values 179
    - running statements 116, 117, 119, 167–169
      - at run time 118
    - saving statements 118, 169
    - specifying search conditions 178
    - specifying table names 9
    - tutorial 47–50
    - type compatibility 180–181
    - updating 50
  - SQL script files 118
  - SQL Statement command (Database Desktop) 168
  - SQL tables
    - creating 155
    - indexes 74, 153–154, 155–157, 160
      - naming 156
    - naming 175
    - restructuring 160
    - retrieving data 73
    - searching for data 70, 72
    - setting ranges 50
    - sorting data 166
    - specifying directories 69
    - updating data 65
    - validity checks 154
  - SQL.IEM 143
  - SQL\_IB.CNF 143
  - SQL\_INF.CNF 143
  - SQL\_ORA.CNF 142
  - SQL\_SS.CNF 142
  - .SQL files 169
  - SQL13WIN.DLL 142
  - SQLD\_IB.DLL 143
  - SQLD\_IB.HLP 143
  - SQLD\_INF.DLL 143
  - SQLD\_INF.HLP 143
  - SQLD\_ORA.DLL 142
  - SQLD\_ORA.HLP 142
  - SQLD\_SS.DLL 142
  - SQLD\_SS.HLP 142
  - SQLPASSTHRUODE
    - setting 136, 136–137
  - SQLWIN.DLL 142
  - starting
    - applications 173
    - BDE Configuration Utility 171
    - Database Form Expert 21
    - Fields Editor 80
    - Input Mask Editor 84
    - Parameters Editor 138
  - StartTrans method 136
  - StartTransaction method 136
  - State property 59
  - static queries 47
  - static SQL statement (defined) 117
  - stored procedures 8, 57, 138–140
    - executing 139
    - overloading 140
    - parameters 138–139
  - StoredProcName property 138
  - String List Editor 109
    - entering SQL statements 117
  - string lists 117
    - associating with buttons 114
    - sorting items 110
  - string manipulation functions 177
  - strings 73
    - concatenating 177
    - conversion functions 86
    - converting
      - case 177
      - to date/time 173
      - to numbers 36, 173
    - picture 83, 84, 154
    - removing repeating characters 177
    - returning parts of 177
  - Structured Query Language *See* SQL
  - Style property
    - combo boxes 110
  - subqueries 178, 179
  - SUBSTRING function 177
  - substrings, returning 177
  - Subtract command (Database Desktop) 165
  - SUM function 176
  - sums 176
  - supported data sources 6
  - Sybase servers 121, 142
  - Sybase tables 6, 90, 91, 156
    - transactions 138
  - SYDC437.LD 143
  - SYDC850.LD 143
  - System Manager page (BDE Configuration Utility) 173
- ## T
- 
- Tab key 101
  - tabbing through fields 101, 104
  - Table Type dialog box (Database Desktop) 151
  - Table window (Database Desktop) 161
  - TableName property 22, 69
  - tables 4, 180
    - adding 8
      - Database Desktop 149–161
    - Answer 170
    - changing 181
    - child 157
    - configuration options 173
    - creating new 89, 151, 155
      - Paradox 151, 158
      - SQL 155
    - customizing views 162–164
    - deleting 75, 166, 182
    - destination 89–92
    - detail 24, 76–77, 93–95
    - linking 94, 150
      - one-many-many links 27
      - one-to-many links 24, 76
    - locking 75
    - locking columns 162
    - lookup 45–46, 158
    - master 24, 76–77, 93–95
    - moving through 60–62, 104
      - Database Desktop and 162, 165
    - multiple views 95–97, 180
    - naming, ANSI standards 175
    - opening 161
      - at run time 23
    - parent 157
    - placing bookmarks 66–67
    - renaming 166
    - retrieving information 57
    - source 89–92

- specifying for queries 9
  - structure
    - borrowing 155
    - changing 159–161
    - copying 8, 75
    - types 69–70
  - TableType property 69
  - taCenter constant 33
  - Tag property 84
  - taRightJustify constant 33
  - TBatchMove component 8, 89–92
  - TBCDField type 86
  - TBlobField type 86
  - TBookmark type 67
  - TBooleanField type 86
  - TBytesField type 86
  - TCP/IP interface 16, 131, 144
  - TCurrencyField type 85, 86
  - TDatabase component 8, 133–135
  - TDataSet class 8
  - TDataSource component 6, 8, 10, 77–78, 100
    - multiple views and 95
  - TDateField type 85, 86
  - TDateTimeField type 85, 86
  - TDBCheckBox component 100, 113
  - TDBComboBox component 99, 109–110
  - TDBEdit component 99, 102
  - TDBGrid component 99, 103–106
    - adding 101, 103
  - TDBImage component 99, 108
  - TDBListBox component 99, 109, 110
  - TDBLookupCombo component 100, 109, 111–112
  - TDBLookupList component 100, 109, 112
  - TDBMemo component 99, 107–108
  - TDBNavigator component 99, 106–107
    - adding 101
    - enabling/disabling buttons 107
  - TDBRadioGroup component 100, 114
  - TDBText component 99, 101–102
  - telephone numbers 40, 41
  - templates
    - data entry 83, 84, 154
  - temporary files 57, 92, 149, 170
  - .TXT files 6
  - text 108
    - See also* data
    - adding 107
    - aligning 108
    - editing 108
    - formatting 40
    - scrolling through 108
    - wordwrapping 108
  - text files 6, 70
  - text strings *See* string lists; strings
  - TField component 30, 33, 40, 79–87, 100
    - disabling 103
    - instantiating 103
    - overview 8
    - referencing 34
  - TFloatField type 84, 85, 86
  - TGraphicField type 86
  - tiDirtyRead constant 137
  - time 85, 173
    - conversion functions 86
    - returning 177
  - Time Manager page (BDE Configuration Utility) 173
  - timer events 101
  - timestamps 46
  - TIMEZONE\_HOUR clause 177
  - TIMEZONE\_MINUTE clause 177
  - TIntegerField type 85, 86
  - tiReadCommitted constant 137
  - tiRepeatableRead constant 137
  - TMemoField type 86
  - tool bars 22
    - Database Desktop windows 148, 161, 168
  - totaling numeric values 176
  - TQuery component 57, 100, 115–117
    - overview 6, 8, 9–10
    - TDataSource and 10
    - TTable vs. 116, 120
  - transactions 100, 128, 135–138
    - implicit vs. explicit control 136
    - isolation levels 137–138
  - TransIsolation property 137
  - translating data 12
  - Transliterate property 89
  - TReport component 8, 52, 88–89
  - TRIM function 177
  - truncated data 102
  - try...except blocks 119
  - try...finally blocks 51, 61, 67
  - TSession component 56–57, 134
  - TSmallIntField type 85, 86
  - TStoredProc component 8, 57, 138
  - TStringField type 84, 86
  - TStrings type 117
  - TTable component 6, 8–9, 57, 69–77, 100
    - multiple views and 96
    - TDataSource and 10
    - TQuery vs. 116, 120
  - TTimeField type 85, 86
  - TUTILITY.DLL 17
  - tutorial 19–53
  - TVarBytesField type 86
  - TWOFORMS.DPR 95
  - TWordField type 85, 86
  - .TXT files 70
  - types 31
    - batch move operations and 90–91
    - converting 86
    - editing 164
    - specifying 152
    - SQL-compatible 180–181
  - typing to replace entries 164
- 
- ## U
- 
- UniDirectional property 119
  - unique indexes 153, 156
  - UnPrepare method 122
  - UPDATE keyword 179
  - UPDATE statements 120
  - UpdateMode property 65–66
  - updating data 63, 65–66, 101, 107
    - batch move operations 89
    - calculated fields 38
    - handling events 78
    - multiuser applications 65
    - read-only results 120
    - Refresh method and 67
  - updating queries 50
  - UPPER function 177
  - uppercase conversions 177
  - upsizing desktop databases 130, 140–142
  - Utilities menu (Database Desktop) 165
- 
- ## V
- 
- validity checks 43–47, 154
    - types 43
  - Value property 86
  - ValueChecked property 113

- values
  - accessing 30, 32
  - assigning 37, 154
    - at run time 87, 88
    - calculated fields 68–69, 82–83
    - default 109, 154
    - names as 122
    - required 154
    - run-time 35
    - to buttons 114
    - to combo boxes 109
  - averaging 176
  - changing
    - edit boxes and 36
  - changing *See* updating data
  - converting
    - numbers to strings 35
    - string to date/time 173
    - string to numbers 36, 173
  - counting 177
  - currency 83, 85
  - display formats 85
  - displaying current 34, 102
  - finding 42, 70–72
    - nearest match 71, 72
  - minimum/maximum 83, 154
    - aggregate functions 176–177
    - specifying 51
  - nil 65
  - null 65, 121
    - checking for 73
    - stored procedures 139
  - predefined 109
  - property
    - accessing 40, 86
    - changing 82, 83–84
    - displaying 32, 103
    - setting 32
  - restoring previous 64, 101, 107
  - retrieving specific 179
  - rounding 86
  - totaling 176
  - typing to replace entries 164
  - updating *See* updating data
- Values property 114
- ValueUnchecked property 113
- variables
  - local SQL statements 176, 180
- VBX controls 16
- VENDOR.DB 183
- vertical dividing lines, enabling/disabling 104
- viewing *See* displaying
- views 95–97, 180
  - Visible property 33, 84, 103
  - VisibleButtons property 23, 107
  - Visual Basic controls 16
  - Visual Query Builder 10, 118–119
  - Visual Query Builder
    - window 118
  - VSL.INI 144

## W

---

- W3DBLIB.DLL 143
- WhereAll constant 66
- WhereChanged constant 66
- WhereKeyOnly constant 66
- Windows for Workgroups 144
- Windows ISQL 129
- Winsock 1.1 compliant TCP/IP products 144
- WINSOCK.DLL 144
- WordWrap property 108
- wordwrapping text 108
- working directories
  - (:WORK:) 149
- Working Directory command (Database Desktop) 149
- write privileges 75



# Database Application Developer's Guide

---



**Delphi™**

Borland International, Inc., 100 Borland Way  
P.O. Box 660001, Scotts Valley, CA 95067-0001



Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1995 Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

1E0R195

9596979899-987654321

W1

# Contents

<b>Chapter 1</b>	
<b>Introduction</b>	<b>1</b>
What you should know first . . . . .	1
Overview of Delphi's database features and capabilities . . . . .	2
What is a database? . . . . .	4
What is data? . . . . .	5
What is data access? . . . . .	5
Data sources . . . . .	6
Understanding Delphi database architecture . . . . .	6
Overview of the Data Access page . . . . .	7
Understanding TTable . . . . .	8
Understanding TQuery . . . . .	9
Understanding TDataSource . . . . .	10
Overview of the Data Controls page . . . . .	10
Overview of the Database Forms Expert . . . . .	11
Overview of the Database Desktop . . . . .	12
Developing applications for desktop and remote servers . . . . .	12
Database application development methodology . . . . .	13
Development scenarios . . . . .	13
Database application development cycle . . . . .	13
Design phase . . . . .	14
Implementation phase . . . . .	15
Deployment phase . . . . .	15
Deploying an application . . . . .	16
Deploying BDE support . . . . .	16
Language drivers . . . . .	17
ODBC Socket . . . . .	18
<b>Chapter 2</b>	
<b>Building a sample database application: MASTAPP</b>	<b>19</b>
Building forms . . . . .	19
MASTAPP aliases . . . . .	20
Building a single-table form . . . . .	21
Building a master-detail form . . . . .	24
Building a one-many-many form . . . . .	27
Working with fields . . . . .	30
Creating Tfield components . . . . .	30
Setting Tfield properties at design time . . . . .	32
Reading field values . . . . .	34
Assigning values to fields . . . . .	35
Defining a calculated field . . . . .	37
Formatting field values at design time . . . . .	39
Formatting field values at run time . . . . .	40
Searching for field values . . . . .	42
Validating data entry . . . . .	43
Using lists and lookups . . . . .	44
Writing code to check field values . . . . .	46
Using queries and ranges . . . . .	47
Setting a range . . . . .	50
Printing reports and forms . . . . .	52
<b>Chapter 3</b>	
<b>Using data access components and tools</b>	<b>55</b>
Database components hierarchy . . . . .	55
Using the TSession component . . . . .	56
Controlling database connections . . . . .	56
Getting database information . . . . .	57
Using datasets . . . . .	57
Dataset states . . . . .	58
Opening and closing datasets . . . . .	59
Navigating datasets . . . . .	60
The Next and Prior methods . . . . .	60
The First and Last methods . . . . .	60
The BOF and EOF properties . . . . .	61
The MoveBy function . . . . .	62
Modifying data in datasets . . . . .	63
The CanModify property . . . . .	63
Posting data to the database . . . . .	63
Editing records . . . . .	63
Adding new records . . . . .	64
Deleting records . . . . .	64
Canceling changes . . . . .	64
Working with entire records . . . . .	64
Setting the update mode . . . . .	65
Bookmarking data . . . . .	66
Disabling, enabling, and refreshing data-aware controls . . . . .	67
Using dataset events . . . . .	68
Abort a method . . . . .	68
Using OnCalcFields . . . . .	68
Using TTable . . . . .	69
Specifying the database table . . . . .	69
The TableType property . . . . .	69
Searching a table . . . . .	70
Using Goto functions . . . . .	70
Using Find functions . . . . .	72
The KeyExclusive property . . . . .	72
Limiting records retrieved by an application . . . . .	72
Using partial keys . . . . .	73
The KeyExclusive property . . . . .	74

Indexes . . . . .	74	Displaying data as labels with TDBText. . . . .	101
The Exclusive property . . . . .	75	Displaying and editing fields with TDBEdit. . . . .	102
Other properties and methods . . . . .	75	Editing a field. . . . .	102
The ReadOnly and CanModify properties . . . . .	75	Viewing and editing data with TDBGrid . . . . .	103
The GoToCurrent method . . . . .	76	Setting grid options . . . . .	103
Creating master-detail forms . . . . .	76	Editing in the grid . . . . .	104
The Field Link Designer. . . . .	76	Rearranging column order at run time. . . . .	105
Using TDataSource . . . . .	77	Controlling grid drawing. . . . .	105
Using TDataSource properties . . . . .	77	Using events to control grid behavior . . . . .	105
The DataSet property . . . . .	77	Navigating and manipulating records with TDBNavigator. . . . .	106
The Enabled property . . . . .	77	Hiding and showing navigator buttons . . . . .	107
The AutoEdit property . . . . .	77	Displaying fly-by help . . . . .	107
Using TDataSource events . . . . .	78	Displaying and editing BLOB text with TDBMemo . . . . .	107
The onDataChange event . . . . .	78	Displaying BLOB graphics with TDBImage . . . . .	108
The OnUpdateData event. . . . .	78	Using list and combo boxes . . . . .	109
The OnStateChange event . . . . .	78	TDBComboBox. . . . .	109
Using TFields and the Fields Editor . . . . .	79	TDBListBox . . . . .	110
What are TField components? . . . . .	79	TDBLookupCombo . . . . .	111
Using the Fields Editor. . . . .	80	TDBLookupList . . . . .	112
Starting the Fields Editor . . . . .	80	TDBCheckBox . . . . .	113
Adding a TField component . . . . .	81	TDBRadioGroup . . . . .	114
Deleting a TField component . . . . .	81	<b>Chapter 5</b>	
Defining a new TField component . . . . .	82	<b>Using SQL in applications</b>	<b>115</b>
Defining a calculated field . . . . .	82	Using TQuery . . . . .	115
Programming a calculated field . . . . .	82	When to use TQuery . . . . .	116
Editing a TField component. . . . .	83	How to use TQuery . . . . .	116
Editing Display properties . . . . .	83	The SQL property . . . . .	117
Using the Input Mask Editor . . . . .	84	Creating the query text . . . . .	117
Formatting fields. . . . .	85	Using the Visual Query Builder . . . . .	118
Handling TField events . . . . .	85	Executing a query . . . . .	119
Using TField conversion functions . . . . .	86	The UniDirectional property . . . . .	119
Accessing TField properties programmatically. . . . .	86	Getting a live result set . . . . .	120
Displaying data with standard controls. . . . .	87	Syntax requirements for live result sets . . . . .	120
Using the Fields property . . . . .	87	Dynamic SQL statements . . . . .	121
Using the FieldByName method. . . . .	87	Supplying values to parameters. . . . .	121
Incorporating reports in an application . . . . .	88	Preparing a query . . . . .	121
Using TBatchMove . . . . .	89	Using the Params property. . . . .	122
Batch move modes . . . . .	89	Using the ParamByName method . . . . .	122
Data type mappings . . . . .	90	Using the DataSource property . . . . .	123
Executing a batch move . . . . .	92	Dynamic SQL example . . . . .	124
Handling batch move errors . . . . .	92	Creating heterogenous queries . . . . .	124
Accessing the BDE directly . . . . .	92	<b>Chapter 6</b>	
Application examples . . . . .	93	<b>Building a client/server application</b>	<b>127</b>
Creating a master-detail form. . . . .	93	Portability versus optimization . . . . .	127
Displaying multiple views of a table. . . . .	95	Server portability . . . . .	128
<b>Chapter 4</b>			
<b>Using Data Controls</b>	<b>99</b>		
Data Control component basics . . . . .	100		
Updating fields . . . . .	101		

Client/server communication		Relational tables . . . . .	150
portability . . . . .	128	Planning tables . . . . .	150
Using the Local InterBase Server . . . . .	129	Creating a new table. . . . .	151
Building an application to access any		Defining fields . . . . .	152
server . . . . .	129	Field names . . . . .	152
Building an application to access		Adding, deleting, and rearranging	
InterBase . . . . .	130	fields . . . . .	152
Using InterBase in upsizing. . . . .	130	Specifying field type . . . . .	152
Connecting to a database server . . . . .	130	Using indexes. . . . .	152
Connectivity. . . . .	131	Keys in Paradox tables. . . . .	152
Using TCP/IP . . . . .	131	A dBASE table's index. . . . .	153
Connection parameters . . . . .	131	An SQL table's index. . . . .	153
Using ODBC. . . . .	132	Defining secondary indexes . . . . .	154
Handling server security . . . . .	132	Specifying validity checks . . . . .	154
Using the TDatabase component. . . . .	133	Borrowing a table structure . . . . .	155
The Connected property. . . . .	134	Creating an SQL table. . . . .	155
Creating application-specific aliases. . . . .	134	Creating indexes on SQL tables . . . . .	155
Understanding transaction control . . . . .	135	Naming SQL indexes . . . . .	156
Handling transactions in applications. . . . .	135	Defining referential integrity for Paradox	
Implicit transaction control . . . . .	136	tables . . . . .	157
Explicit transaction control . . . . .	136	Procedure . . . . .	157
Setting the SQL passthrough mode . . . . .	136	Creating table lookup . . . . .	158
Transaction isolation levels . . . . .	137	Establishing passwords for Paradox	
Using stored procedures. . . . .	138	tables . . . . .	159
Input and output parameters. . . . .	138	Restructuring tables. . . . .	159
Executing a stored procedure. . . . .	139	Restructuring an SQL table. . . . .	160
Oracle overloaded stored procedures . . . . .	140	Prefixing the index name with the	
Upsizing . . . . .	140	table name . . . . .	160
Upsizing the database . . . . .	140	Viewing tables . . . . .	161
Upsizing the application. . . . .	141	Using scroll lock . . . . .	162
Deploying support for remote server access .	142	Customizing a table view. . . . .	162
Oracle. . . . .	142	Rearranging and resizing columns . . . . .	163
Sybase and Microsoft SQL servers. . . . .	142	Resizing rows . . . . .	163
Informix . . . . .	143	Resizing column headings . . . . .	163
InterBase . . . . .	143	Saving a custom view . . . . .	163
TCP/IP Interface . . . . .	144	Editing data . . . . .	164
Other communication protocols . . . . .	144	Using Edit mode . . . . .	164
Deploying ReportSmith support . . . . .	145	Selecting fields and records . . . . .	165
		Field view . . . . .	165
		Adding, subtracting, and emptying	
		records. . . . .	165
<b>Appendix A</b>		Sorting, copying, renaming, and	
<b>Using Database Desktop</b>	<b>147</b>	deleting objects. . . . .	166
What is Database Desktop? . . . . .	147	Sorting tables . . . . .	166
The Database Desktop window . . . . .	147	Copying objects . . . . .	166
Starting Database Desktop . . . . .	147	Renaming objects . . . . .	166
The Database Desktop window . . . . .	148	Deleting objects . . . . .	166
Managing files. . . . .	148	Executing SQL statements . . . . .	167
Opening files . . . . .	148	What is the SQL Editor? . . . . .	167
Setting up a working directory. . . . .	149	Opening the SQL Editor . . . . .	168
Setting up a private directory . . . . .	149	Specifying an alias . . . . .	168
Aliases . . . . .	149	Running an SQL statement. . . . .	169
Creating tables. . . . .	149	Saving an SQL statement . . . . .	169
Understanding tables . . . . .	150		

Querying table data with QBE . . . . .	169	Using SELECT . . . . .	178
<b>Appendix B</b>		Using the FROM clause . . . . .	178
<b>Using the BDE configuration utility</b>	<b>171</b>	Using the WHERE clause . . . . .	178
Creating and managing aliases . . . . .	171	Using the ORDER BY clause . . . . .	179
Adding a new alias . . . . .	172	Using the GROUP BY clause . . . . .	179
Modifying an existing alias . . . . .	173	Heterogeneous joins . . . . .	179
Deleting an alias . . . . .	174	Using INSERT . . . . .	179
<b>Appendix C</b>		Using UPDATE . . . . .	179
<b>Using local SQL</b>	<b>175</b>	Using DELETE . . . . .	179
Naming conventions for tables . . . . .	175	Data definition. . . . .	180
Naming conventions for columns . . . . .	176	Using CREATE TABLE . . . . .	180
Data manipulation . . . . .	176	Using ALTER TABLE . . . . .	181
Parameter substitutions in DML		Using DROP TABLE . . . . .	182
statements . . . . .	176	Using CREATE INDEX . . . . .	182
Supported set (aggregate) functions . . . . .	176	Using DROP INDEX . . . . .	182
Supported string functions . . . . .	177	<b>Appendix D</b>	
Supported date function. . . . .	177	<b>The MAST database</b>	<b>183</b>
Supported operators . . . . .	177	<b>Index</b>	<b>187</b>

# Tables

1.1	Database features summary . . . . .	2	3.8	TField formatting routines . . . . .	.85
1.2	Additional Delphi Client/Server database features . . . . .	4	3.9	Published TField events . . . . .	.85
1.3	Delphi data sources. . . . .	6	3.10	TField conversion functions . . . . .	.86
1.4	Data Access components . . . . .	8	3.11	Important TReport methods. . . . .	.88
1.5	Data Controls components . . . . .	11	3.12	Batch move modes . . . . .	.89
1.6	Redistributable Borland Database Engine files . . . . .	17	3.13	Physical data type translations from Paradox tables to tables of other driver types . . . . .	.90
2.1	Important TTable properties for a single-table form . . . . .	22	3.14	Physical data type translations from dBASE tables to tables of other driver types . . . . .	.90
2.2	Important TDataSource properties for a single-table form . . . . .	22	3.15	Physical data type translations from InterBase tables to tables of other driver types . . . . .	.91
2.3	Important TDBGrid properties for a single-table form . . . . .	23	4.1	Data controls . . . . .	.99
2.4	Important TDBNavigator properties for a single-table form . . . . .	23	4.2	Expanded TDBGrid Options properties. . .	104
2.5	Important detail table properties. . . . .	26	4.3	TDBGrid events . . . . .	105
2.6	Important component properties for a master-detail form . . . . .	26	4.4	TDBNavigator buttons . . . . .	106
2.7	Important component properties for a one-many-many form . . . . .	29	4.5	Data-aware list box and combo box controls . . . . .	109
2.8	Important TField design-time properties. . .	33	5.1	Types of query result sets . . . . .	120
2.9	Important TField design-time properties. . .	40	6.1	Server transaction isolation levels. . . . .	138
2.10	Important TDBComboBox properties. . . . .	44	6.2	Oracle SQL Link files . . . . .	142
2.11	Important TDBComboBox properties. . . . .	45	6.3	Sybase SQL Link files . . . . .	142
2.12	Important TReport properties . . . . .	52	6.4	Informix SQL Link files. . . . .	143
3.1	TSession methods. . . . .	57	6.5	InterBase SQL Link files . . . . .	143
3.2	Dataset states . . . . .	58	6.6	Winsock 1.1 client files . . . . .	144
3.3	Navigational methods and properties . . . .	60	6.7	Non-Winsock compliant TCP support files . . . . .	144
3.4	Methods to insert, update and delete data in datasets . . . . .	63	A.1	Paradox validity checks. . . . .	154
3.5	Methods used to work with entire records. .	64	C.1	Data type mappings. . . . .	180
3.6	Dataset events . . . . .	68	C.2	BLOB subtype mappings. . . . .	181
3.7	TField properties . . . . .	83	D.1	MAST tables . . . . .	183

# Figures

1.1	Delphi database architecture . . . . .	2	4.13	A TDBRadioGroup component. . . . .	114
1.2	Structure of a table . . . . .	4	5.1	TQuery methods and flow. . . . .	117
1.3	Database components architecture . . . . .	7	5.2	Editing SQL statements in the String List Editor. . . . .	118
1.4	Data Access page of the Component palette . . . . .	7	5.3	Working in the Visual Query Builder . . . . .	119
1.5	The Data Controls page of the Component palette. . . . .	10	5.4	Parameters Editor . . . . .	121
1.6	Development cycle . . . . .	14	5.5	Form with linked queries. . . . .	123
2.1	Database forms described in the tutorial . . . . .	20	6.1	InterBase parameters in the String List Editor . . . . .	132
2.2	Building a single-table form using the Database Form Expert . . . . .	21	6.2	Database Login dialog box. . . . .	133
2.3	A single-table form. . . . .	22	6.3	Database Properties Editor. . . . .	134
2.4	Linking fields in a master-detail form. . . . .	25	6.4	TStoredProc Parameters Editor . . . . .	139
2.5	A master-detail form. . . . .	26	A.1	The Database Desktop application window . . . . .	148
2.6	One-many-many form. . . . .	29	A.2	Application window tool bar . . . . .	148
2.7	The Fields Editor . . . . .	31	A.3	A table . . . . .	150
2.8	Adding fields to a data set. . . . .	31	A.4	Table Type dialog box . . . . .	151
2.9	TField component properties. . . . .	33	A.5	The Create Table dialog box . . . . .	151
2.10	Defining a calculated field . . . . .	38	A.6	The Create Table dialog box for SQL tables. . . . .	155
2.11	Lists and lookups. . . . .	44	A.7	The Define Index dialog box for SQL indexes. . . . .	156
2.12	Setting a TQuery's SQL property . . . . .	48	A.8	Save Index As dialog box. . . . .	156
2.13	Defining a query parameter . . . . .	49	A.9	Referential integrity . . . . .	157
3.1	Delphi Data Access components hierarchy . . . . .	56	A.10	Referential Integrity dialog box . . . . .	158
3.2	Dataset state diagram . . . . .	59	A.11	The Restructure Table dialog box for Paradox tables . . . . .	160
3.3	Field Link designer . . . . .	76	A.12	The Restructure Table dialog box for SQL tables. . . . .	160
3.4	Fields Editor . . . . .	80	A.13	The Table window tool bar . . . . .	161
3.5	Fields Editor Add Fields dialog box. . . . .	81	A.14	A scroll lock in the Table window . . . . .	162
3.6	Define Field dialog box . . . . .	82	A.15	Hot zone pointers in the Table window . . . . .	163
3.7	Input Mask Editor . . . . .	84	A.16	The <i>Customer</i> table in Edit mode . . . . .	164
3.8	Sample form . . . . .	94	A.17	The SQL Editor. . . . .	167
3.9	Two forms . . . . .	95	A.18	The SQL Editor and an <i>Answer</i> table. . . . .	168
4.1	Data Controls Component palette . . . . .	99	A.19	SQL Editor Toolbar . . . . .	168
4.2	TDBText component . . . . .	102	A.20	SQL statement in the SQL Editor . . . . .	169
4.3	TDBEdit component at design time . . . . .	102	A.21	A query and its results . . . . .	170
4.4	TDBGrid component. . . . .	103	B.1	BDE Configuration Utility main window . . . . .	171
4.5	TDBNavigator component . . . . .	106	B.2	Sample Add New Alias dialog box. . . . .	172
4.6	TDBMemo component. . . . .	108	B.3	Customizing the new alias. . . . .	172
4.7	DBImage component. . . . .	108	B.4	BDE non-system configuration dialog box. . . . .	173
4.8	DBComboBox component . . . . .	109			
4.9	TDBListBox component . . . . .	110			
4.10	TDBLookupCombo component . . . . .	111			
4.11	TDBLookupList component . . . . .	113			
4.12	TDBCheckBox component . . . . .	113			



**IT KNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Introduction](#)

[Foreword](#)

[About the Author](#)

[CHAPTER 1—The Lure of Game Programming](#)

[The Dark Side of the Game Programming Industry](#)

[Difficult Learning Curve](#)

[Long Hours](#)

[Corporate Instability](#)

[The Light Side of the Game Programming Industry](#)

[Fun and Rewarding](#)

[Fame and Fortune](#)

[The People](#)

[Game Programming, In Delphi?](#)

[Delphi as a Games Development Platform](#)

[Opportunities](#)

[Untapped Markets](#)

[Types of Games](#)

[Real-Time Strategy](#)

[Turn-Based Strategy](#)

[First-Person Shooters](#)

[Adventure Games](#)

[Action Games](#)

[Puzzle Games](#)

[Sports Games](#)

[Board and Card Games](#)

[Simulations](#)

[Windows, Delphi, and Games](#)

[Summary](#)

[CHAPTER 2—The Anatomy of a Game](#)

[Game Dissection: The Vital Organs](#)



Graphics

Sound

User Input

Real-Time Event-Driven Programming

Basic Game Architecture

Initialization

Introduction

Game Startup

The Game Loop

Game Ending

Shutdown and Exit

The Game Loop

Retrieve User Input

Perform Antagonist AI and Other Game Logic

Update Sprite Positions

Check for Collisions

Start Sound Effects

Start Music

Render the Next Frame

Check for Game End

Speed Delay

Game States

The Demo State

The Playing State

The Intermission State

The Paused State

The Game Over State

Case Study

Summary

## CHAPTER 3—Basic Graphics Programming

The Graphical Device Interface

Graphical Elements

Pixels

Lines

Polygons

Bitmaps

Graphical Element Classifications

Video Mode Resolutions and Color Depth

Resolution

Color Depth

Video Board Memory

The Components of a Color

Palettized Modes

Non-palettized Modes

Graphics Output Techniques

The Device Context

Painting on the Canvas

Bitmap Essentials

Bitmap Types: DDB and DIB

The TBitmap Object

Raster Operations

Simulating Transparency

Palettes

Logical vs. System Palettes

Using a Palette

Further Adventures in Windows Programming

Summary

CHAPTER 4—An Introduction to DirectX

What is DirectX?

The Promise of DirectX

Benefits and Drawbacks of DirectX Programming

The DirectX Architecture

DirectX Components

DirectX and Delphi

The DirectX SDK

Project JEDI

Other Components and Headers

A DirectDraw Tutorial

Features

Functional Overview

COM Fundamentals

Surfaces

Page Flipping

Resolution, Color Depth, and Video Memory

Windowed Versus Full-Screen

Initializing DirectDraw

Rendering with GDI

Flipping the Surfaces

Displaying Bitmaps

Surface Memory Location

Loading and Displaying the Bitmap

Blitting Surfaces

Lost Surfaces

Retrieving DirectDraw Capabilities

The DirectX Baseline Application

DirectX Result Codes

Retrieving the DirectX Version

Summary

CHAPTER 5—Palettes

What is a Palette?

The IDirectDrawPalette Object

Creating the Palette Object

Defining Palette Colors

Using the Palette Object

Disconnecting Palettes

Palettes and GDI Functions

Palette Animation

Animating the Palette

Screen Transitions

Summary

CHAPTER 6—Sprite Techniques

The Illusion of Animation

Constructing the Animation Frames

Producing the Animation

The Illusion of Motion

Sprite Attributes

The Animation Cycle

Transparency Issues

Color Keying

Pixel Checking

Collision Detection

Bounding Boxes

Pixel Overlap Comparison

Animation Issues

Timing

Z-Order

Summary

## CHAPTER 7—Input Techniques

Human/Computer Interaction

Input Devices

Input Considerations

DirectInput

Features

Functional Overview

Initializing DirectInput

Retrieving Input Data

Keyboard Input

Mouse Input

Game Controller Input

Summary

## CHAPTER 8—Force Feedback

Features

Functional Overview

Basic Concepts

Kinetic Force

Condition

Magnitude

Coefficient

Duration

Sample Period

Gain

Axes

Direction

Envelope

Force Feedback Effects

Base Effect Structure

Creating Effects

Create and Acquire the DirectInput Device

Query for Force Feedback Support

Initialize a TDIEffect Structure and the Type-Specific Parameters Structure

Create the Effect Object

Download the Effect

Start the Effect

Stop the Effect

Case Study

Additional Effect Considerations

Retrieving Effect Status

Retrieving the Force Feedback System State

Sending Commands to the Force Feedback System

Modifying an Effect at Run Time

Envelopes

Summary

## CHAPTER 9—Sound and Music

Sound Effects

Simplistic Sound Effects

DirectSound

Music

MCISendCommand

MIDI

CD Audio

Summary

## CHAPTER 10—Optimization Techniques

Optimization Theory

Timing Code

Examining the Algorithm

Effective Optimization

Delphi-Specific Optimization Techniques

Automatic Compiler Optimizations

Additional Compiler Optimizations

Other Delphi-Specific Optimization Techniques

General Optimization Techniques

Loop Unrolling

Lookup Tables

Binary Bit Shifting

Additional Optimization Considerations

Summary

## CHAPTER 11—Special Effects

Bitmap Manipulation

Bitmap Transformations

Scaling

Rotation

Lighting Tricks

Transition Effects

Simple Texture Mapping

Dynamic Images

Star Fields

Fire

Summary

## CHAPTER 12—Artificial Intelligence Techniques

Movement

Random Movement

Aggression/Evasion

[Patterned Movement](#)  
[Finite State Machines](#)  
[Probability Machines](#)  
[Path Finding](#)  
[Algorithms](#)  
[Basic Implementation](#)  
[Back Stepping](#)  
[Case Study](#)  
[Enhancements](#)  
[Summary](#)

## [CHAPTER 13—Putting It All Together](#)

[Delphi Blocks](#)  
[General Game Architecture](#)  
[Player Advancement](#)  
[Game States](#)  
[User Input](#)  
[Sound and Music](#)  
[Graphics](#)  
[Enhancements](#)  
[The Case Study Code](#)  
[Summary](#)

[Appendix A](#)  
[Glossary](#)  
[Bibliography](#)  
[Index](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Introduction

In the beginning, there was darkness. In this Age of Innocence, the unwashed masses practiced the art of entertainment with small pieces of paper bearing symbolic images of kings, queens, and jesters. Sometimes, they used minute pieces of metal and plastic, shaped like various real-world objects, moving them about on large, colored cardboard sheets in accordance with the runes inscribed upon small white cubes. To the casual observer, all was content, but in the minds of the newest generation born of this age, there was a yearning for more. Unbeknownst to the world, this yearning grew, and soon these young pioneers began working in the dark recesses of colleges and universities to bring about a change that would alter the course of history in ways no learned scholar could have ever imagined. Brandishing vacuum tube sabers and using arcane words of power such as RAM, ROM, Bit, and Byte, these acolytes of technology brought forth a new creation known as the Computer. But, their new creation was naive, and lacked intelligence or sophistication; to control it, they had to use a mystical, magical language known as Code. Thus, the Art was born, and a new era had begun.

The First Age was marked by the genesis of immense, hulking beasts that required enormous resources to sustain life. These new life forms were rare, but typically stood several stories tall and required incredible life support systems to maintain an agreeable environment. These giants were moody and possessed little intelligence, but through great effort, they could be trained to perform incredible feats, for their day. The digital landscape was dotted with few of these behemoths, and their trainers were monk-like in their dedication. Unable to break free of their confines due to the immense size of their charges, these monks were forced to toil maddeningly long hours. Their torturous task of preparing the vegetarian meal of holy leaves consumed by these monsters had them working late into the night, and the Art required to create these holy leaves was arcane indeed.

These monks practiced the great Art in the same dark recesses from whence it was conceived, and the Art grew. The once young pioneers had become pedantic, wizened old masters. These old wizards were still rebellious and, yearning for a new form of entertainment, they used the Art in ways that the Powers That Be felt were unproductive. Away from the watchful eyes of their overseers, they shaped the Art in ways that previously existed only in their dreams.

In the dark hours of the morning, these masters used the Art to coax the new creations into providing their

long-sought-after entertainment, and they played games such as Tic-Tac-Toe and Pong. Several of these wizards used the Art to create much more sophisticated forms of entertainment, and played games such as Checkers and Chess with their silicon automatons. In the wake of the First Age, walking in the shadow of the evolving Art, a new form of the Art began to grow.

Like dinosaurs, the reign of these behemoths was relatively short-lived. A new catalyst known as the Corporation had used the knowledge learned in the First Age to effect an evolutionary step known as the Home PC. Thus, the Second Age began.

In the Second Age, the pantheon of overlord gods began to form, controlling the evolution of this new creation from on high. Deities known as Gates and Jobs worked their mystical creation power behind the curtains of reality, but their new creations were protozoan in nature. A step back from their massive predecessors, these new single-cell machines were much less sophisticated and powerful, and teetered on the edge of extinction almost as soon as they had appeared. However, these new creatures had become visible to the unwashed masses, and began to elicit their attention. At first, few of the Untrained could afford to adopt one as a pet, but their popularity grew, and soon many of these creations made their homes on desktops in homes and businesses. The Art, still arcane in nature, had begun to emerge from its dark recesses, and several of the Untrained began to delve into its mystical powers.

The Net, the once-dark netherworld populated solely by the wizards and masters of the Art, has now become a tourist attraction, replete with amusement parks, rest stops, and gift shops. Rumors of new CPUs with speeds in excess of 1,000 MHz are being whispered in the dark corners of cyber cafés and coffee shops. Thus, we stand at the dawn of yet another New Age. As the sun breaks over the distant horizon, its golden, shimmering rays cascading over the silicon juggernaut that stands before us, the prospects of the future are at once both exciting and frightening.

Thus, our journey begins.

## The Purpose of This Book

The primary goal of this book is to teach readers graphics and game programming using 8-bit color images under DirectX in the Delphi environment.



The majority of the book is dedicated to the mechanics of creating a game in Delphi, and will therefore be concerned mainly with exactly how to do something, such as fast flicker-free animation. Some parts address more theoretical issues, such as game design, and are very light in their coverage. Other parts are at a more intermediate level, such as artificial intelligence and optimization techniques. The techniques discussed in this book are very practical and can be put to use right away, as opposed to being academic theory. The goal is to give the Delphi games programmer the tools he/she needs to start competing in the shareware games market right away. This book will not cover marketing, package design, distribution, or other topics that do not deal directly with creating the game.



Due to the popularity of DirectX in the game programming industry, this book does not cover anything related to OpenGL. However, Wordware publishes an excellent OpenGL book aimed at Delphi programmers titled *Delphi Developer's Guide to OpenGL* (ISBN 1-55622-657-8). Although this book is DirectX-specific, some chapters may include code examples that run directly under Windows in order to illustrate a complex topic more clearly. This book is concerned with high-performance graphics programming under DirectX, and as such will concentrate only on full-screen, exclusive mode DirectX programming techniques.



This book will not, as others have, create a “framework” from which examples are drawn, nor will it create wrappers or attempt to hide the low-level DirectX API. This is so that readers become familiar with DirectX, not some artificial abstraction layer that might not work for their purposes. The closest to a “framework” that we’ll get in this book is the creation of a baseline application that simply gets a page flipping, full-screen DirectDraw application up and running, including mouse support. This was done in order to simplify the creation of all the examples included in the book, but it certainly doesn’t shield the developer from any of the nuances or pitfalls of DirectX programming. If you want to do game programming, you have to get in bed with the operating system, the API, and the hardware.

## Conventions

Descriptive text that details steps for implementing or using a specific technique, or narrative text that explains an algorithm or theory, uses a font style similar to the one used for this sentence. Text that contains program code uses a monospaced font style similar to the code snippet below:

```
{draw the paddle graphic to the back buffer}

SrcRect := Rect(252, 500, 339, 506);

FBackBuffer.BltFast(XPos, YPos, FGraphics, SrcRect,
                    DDBLTFAST_NOCOLORKEY OR DDBLTFAST_WAIT);
```

Additionally, the text contains several callouts that point out a specific detail or area of note that the developer should be aware of. These include notes, tips, cautions, and glossaries.

## Notes

These are general points of interest that detail specific items the developer should keep in mind when using the described techniques.

## Tips

Tips are general suggestions or hints for improving a specific technique. Tips may also point out an alternative to some specific detail of a technique or algorithm.

## Cautions

A caution callout alerts the developer to specific hazards with a technique or algorithm that could potentially crash the application or the system. These are specific details of which the developer should be keenly aware.

## Glossary

Being somewhat of an introductory text for game programming, and as the vast majority of readers of this book will have little to no game programming knowledge, glossary callouts identify specific words or phrases that are commonly used in the game programming industry. Game programming, like many specialty areas, tends to have a language all its own, and these glossary items will help the novice user get up to speed on the various jargon and technical terms used when describing game programming techniques.

## What This Book Is

This book is a tutorial that explains DirectX programming within the Delphi environment. Unfortunately, there are not very many professional Delphi programmers working with DirectX, and there are fewer still



who have full-time jobs making games using Delphi. The vast majority of Delphi programmers who will likely pick up this book are application and database programmers professionally, but dabble in game programming at night as a hobby or just to satisfy their curiosity. Thus, this book is aimed at those programmers who are curious and interested in game programming, but have little to no experience. As such, the chapters are arranged in a “how-to” style so that you can easily learn about one aspect of DirectX or another. Additionally, those chapters that deal with DirectX-specific components have a subtle reference layout, so that you can go back and easily find a specific detail about a particular DirectX component or function.



This book could be considered an introductory text for game programmers. To this end, it concentrates on the most common DirectX core components, such as DirectDraw, DirectSound, and DirectInput. Although it covers some advanced techniques, such as force feedback, it is aimed primarily at 2D sprite-based graphics techniques, and does not cover 3D techniques or Direct3D. The techniques herein can be used in scientific or other types of visualization applications, but it is slanted toward game programming.

## What This Book is Not

By necessity, this book is not a tutorial on the use of Delphi or the lexical constructions of the Object Pascal language. This book is also not a DirectX reference. The reference documentation that ships with the DirectX SDK is pretty good, and definitely expansive. Any decent DirectX reference would span many volumes. While many of the more common DirectX function calls are examined in detail, you would be wise to download the DirectX SDK so you can have the full reference documentation at your disposal.

Additionally, since we are concerned only with creating high-performance game and graphics applications, this book does not cover windowed DirectX programming techniques or overlays. Plus, so we can squeeze the maximum amount of performance out of the widest range of hardware, we focus only on 8-bit color video modes, and do not discuss non-palettized video modes.

## Reader Requirements

This book assumes that the reader is familiar with the Delphi IDE and with Object Pascal programming in general. It also helps to have a working knowledge of Windows API programming. While some graphics programming experience is helpful, it is not necessarily a requirement.

## Software Requirements

The applications in this book were tested using Delphi 3, 4, and 5 running under Windows 95 and 98. DirectX is not fully supported under Windows NT, so while some of the examples may work under Windows NT, most will not, and none were tested.

The examples in this book make use of the excellent DirectX header files written by Erik Unger for the JEDI project. You will find these files off of the root of the CD-ROM in a directory labeled DXSDK. This directory should be copied to your hard drive, and you will need to add the path to this directory to Delphi’s library path. This can be accomplished by clicking the Tools | Environment Options menu item in the Delphi IDE, selecting the Library tab, and clicking on the button next to the Library Path edit box. Adding the path to the DXSDK in this dialog box and clicking OK should allow Delphi to find the necessary files when compiling the examples.

---

### Note:

You must add the path to the DXSDK directory to Delphi’s Library Path before any of the examples will compile.

---

## Hardware Requirements

All examples were tested on a Pentium 450 with 32 MB of ram and an 8 MB video card. While this is, at the time of this writing, a respectably advanced machine, the examples should run on anything from Pentium 166 with 16 MB of RAM and a 2 MB video card or higher. Additionally, you will need a sound card and speakers for the examples using sound or music output, as well as a joystick or similar device for examples that retrieve input from game devices. Obviously, you will need a force feedback joystick in order to use any of the examples from the force feedback chapter.

## Chapter Organization

Due to the “how-to” nature of this book, the chapters have been arranged in a way to facilitate an interesting read from cover to cover. However, this book could also be used as a limited reference manual, so the chapters have been arranged in a somewhat logical order to keep similar topics together.

The first few chapters serve as a basic introduction to game programming. Several common concepts and terms are explained early on so as to facilitate an easier understanding of the topics to come. Basic game application architecture is also discussed, including some entry-level graphics programming techniques using regular Windows GDI function calls. DirectX is also introduced at this point, specifically covering DirectDraw.

Many intermediate techniques are then covered. These include techniques for use of the system palette as well as many sprite drawing and manipulation techniques. User input is then covered in some detail, including reading input from keyboard, mouse, and game controllers, such as joysticks, using DirectInput. Force feedback is covered to some extent, although this is a more advanced and specialized topic. Sound and music programming is discussed, examining various ways to output sound and music through both DirectSound and Windows multimedia functions.

Advanced topics include various ways to optimize game applications, as well as artificial intelligence techniques. Several more specific algorithms are covered that detail how to manipulate bitmaps and create graphical special effects. Most of these techniques are wrapped up into a case study application that highlights how these pieces come together to form a complete game. Installation of DirectX is also covered.

## Chapter Summaries

**Chapter 1: The Lure of Game Programming** looks at game programming in general, and examines why the game programming industry is so special in the world of computer science.

**Chapter 2: The Anatomy of a Game** looks at what goes into a game from a high-level perspective. It divides game applications into three constituent parts and examines each in detail.

**Chapter 3: Basic Graphics Programming** introduces the reader to basic concepts in graphics programming. Several Windows GDI graphics programming techniques are discussed in order to prepare the reader for more advanced topics.

**Chapter 4: An Introduction to DirectX** introduces the reader to the world of DirectX programming. DirectX is examined in a general manner, and then DirectDraw is discussed in detail.

**Chapter 5: Palettes** looks at palette control and manipulation from a DirectX application.

**Chapter 6: Sprite Techniques** examines how 2-D sprites work. Several different techniques are discussed, including sprite animation and collision detection.

**Chapter 7: Input Techniques** discusses the various methods for reading input from external devices. The keyboard, joystick, and mouse peripherals are discussed in detail, examining how input can be received using both Win32 API functions as well as DirectInput.

**Chapter 8: Force Feedback** examines the force feedback API and how it can be used to create tactile

feedback on hardware devices.

**Chapter 9: Sound and Music** looks at what it takes to output sound and music from a game application. Using both Windows multimedia functions as well as DirectSound, this chapter examines digital audio output, MIDI music output, and CD audio output.

**Chapter 10: Optimization Techniques** discusses various methods to optimize and enhance the performance of applications. These include Delphi compiler settings as well as programmatic techniques.

**Chapter 11: Special Effects** discusses how the Lock method of a DirectDraw surface gives developers incredible control over visual images. Several techniques are discussed for bitmap manipulation as well as dynamic image creation.

**Chapter 12: Artificial Intelligence Techniques** looks at various methods for implementing simplistic reasoning and logic functions for game antagonists. Such topics include finite state machines and pathing techniques.

**Chapter 13: Putting It All Together** takes many of the techniques discussed throughout the book and puts them together to form a full, working game called Delphi Blocks.

**Appendix: Installing DirectX** contains a short but concise explanation of how the DirectSetup API simplifies the process of installing DirectX to a user's machine.

**Glossary** contains an alphabetical listing of all of the glossary entries introduced throughout the book.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Foreword

Welcome to *Delphi Graphics and Game Programming Exposed!y with DirectX*. This excellent book is a joy to read. It takes a difficult subject, game programming, breaks it down into its constituent parts, and renders each facet of the whole in clear, engaging language. The text sparkles like a well-cut stone, and the code lights the way for programmers who want to explore new territory in the world of game programming.

This book introduces programmers to the key facts they need to know to create games with Borland Delphi. It carefully describes graphics programming, DirectX programming, gathering user input, playing sounds and music, programming game hardware, and discovering the right architecture for a robust game.

Many technical books are bought simply because they cover an important subject. For those who love games and graphics, certainly this book meets that requirement. However, the text is given a certain additional luster because its author crafted it so carefully. Over the last few years John Ayres has been slowly developing a reputation as one of the major voices in Delphi programming. The high quality of the books that he works on and the talks that he gives have made him a figure of significant stature to Delphi developers.

When John explains a subject, it stays explained. He writes in well-formed sentences and creates logically structured paragraphs that combine to form the framework of a text that fully elucidates a subject. Some programmers have the skills necessary to write great code, but lack the knowledge of how to craft a useful text that comprehensibly encapsulates their knowledge. Other programmers are gifted with the prerequisite writing skills, but they lack a deep knowledge of their subject. John is gifted in both areas, and as a result his work is interesting, well thought out, and easy to understand.

Because of my love for the Delphi programming environment, I'm particularly pleased that this text does such a fine job of showing how to take advantage of what's best in the Object Pascal language, the Delphi VCL, and Borland's many high-performance tools. Delphi is one of the great, if not the greatest, programming environments ever created. John shows you the right way to take advantage of all that raw power to build robust games that will stand the test of time.

Delphi is indeed one of the best tools available for creating computer games, and John Ayres rises to the challenge of his subject by creating a lucid, easy to understand text that covers the full range of relevant

technology. A number of excellent programmers have done much to advance the Delphi programming world. These people include the translator of the DirectX headers: Erik Unger; Delphi R&D members Chuck Jazdzewski and Danny Thorpe; Delphi Games Creator programmers Paul Bearne, John Pullen, and Jeff Kurtz; and DelphiX creator Hiroyuki Hori. To that list of names we can now add John Ayres, who has created an excellent resource of use to all Delphi programmers who aspire to write high-quality game and graphics code.

Charlie Calvert  
Borland Developer Relations

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Acknowledgments

While this book may bear only one name on the cover, it is the result of input from many people who made its production possible. In an effort to recognize the contributions of those who helped bring this book to life, I would like to acknowledge and give thanks to the following:

First and foremost, I must thank Erik Unger for his excellent translation of the DirectX headers. Without his dedication to the furthering of Delphi games programming, this book simply would not have been possible. He constantly strives to stay abreast of new DirectX developments, and makes new DirectX headers available almost as soon as Microsoft releases an update. He also has many other headers available for both OpenGL and the Glide API. I hope to shake your hand someday at an Inprise convention somewhere.

I must also mention Ari Feldman, who was generous enough to create the freeware sprite library SpriteLib. SpriteLib is included on the CD, and many of the images from this library are used in the examples and illustrations throughout the book. Ari Feldman's creation allowed me to include some quality artwork whereas little stick men and solid polygons would've been used in its absence.

Additionally, John Marshall provided some of the graphic artwork gracing the examples in the chapter on bitmap manipulation and special effects. John is an awesome artist, and it's an honor and a pleasure to have his work in this book.

Thanks to the folks at Beachware who were generous enough to donate 50 excellent sound effects from their *A Zillion Sounds 2.0* CD. This CD contains thousands of sound effects in WAV format that are perfect for use in many games, from button clicks to explosions and laser blasts. Beachware also offers many other CDs filled with textures and musical interludes that you will find useful in your gaming endeavors. Take a look at <http://www.beachware.com>.

I must also thank David Strahan. An extraordinary musician, he provided several MIDI music compositions used in some of the examples.

I also owe a debt of gratitude to Eric Best, Mark Harwell, Jeff Amano, and Dr. Jim Beckett for putting up with this extracurricular activity of mine while in the middle of many, many time-critical projects, and for



providing me with a nice, quiet place to write on my off hours.

Wordware Publishing deserves a big salute. Jim Hill, Russ Stultz, Beth Kohler, and all the fine people there do a wonderful job. Without Wordware Publishing, many of the more specialized Delphi books would never be published. These more targeted titles help the Delphi community to grow and foster a sense of maturity; may other publishers take note.

Of course, no acknowledgment would be complete without mentioning the people who brought us Delphi: Inprise. Inprise is starting to recognize the existence of the many game programmers out there interested in Delphi. Indeed, both Danny Thorpe and Charlie Calvert (native Borlanders) are big proponents of DirectX programming with Delphi. At last year's Inprise convention, I gave a very successful presentation on Delphi games programming, I believe the first one ever, and I've been invited to repeat it this year. I hope that this increased awareness of Delphi game programmers results in more direct support for DirectX and other core gaming APIs.

I would also like to mention Kenneth Harrison, Mike Tobin, and Burleigh Wood, three great friends of mine who served as sounding boards for my ideas and tolerated my absences from many multiplayer gaming sessions while I wrote. Burleigh even loaned me his force feedback joystick so I could write the force feedback chapter. They took time out of their schedules to look over several of the examples herein, and I appreciate their feedback and criticisms.

Many of you may have heard of the JEDI Project by now. This is an international organization of Delphi developers who are dedicated to converting API headers so that Delphi programmers worldwide have access to a greater range of Windows functionality. The members of this organization, through their many e-mails and discussion threads, provided many of the answers to tough dilemmas experienced while researching many of DirectX's nuances. For more information on the JEDI Project, see the page at the back of the book.

The Delphi Developers of Dallas user group was also very helpful in times of need. I gave many lectures and presentations based on the subject material for this book, and the questions and feedback of the membership helped to solidify and fine-tune many of the topics contained herein.

And were it not for the efforts of the growing community of Delphi game programmers, this book would have had no inspiration. Their efforts are slowly fostering acceptance of Delphi within the gaming industry. Indeed, there are several games being published that are written in Delphi, many of which use either freeware Delphi DirectX components or Erik's DirectX headers included with this book. Keep it up; the next game of the year just might have a "Powered by Delphi" logo somewhere in the About box.

Finally, I must give thanks to my family for putting up with the late nights and long hours it took to finish this project. I owe my wife special thanks and gratitude; she was pregnant with our first child during the course of this activity. Even though she denies it, I know that I've neglected her, and yet she still gives me her unconditional love. Without her constant support and encouragement, I would not have had the patience or desire to continue when the going got tough. Thanks, baby.

## **Dedication**

To my first child, Cassandra Marie Ayres. She and my wife were still part of each other during most of the production of this book, and as I write this, her "birth day" draws alarmingly near. I've never known such panic and such joy as children bring to their parents. It will be good to experience the world again through her eyes, as I intend to vicariously relive my childhood through her as best I can. Only a parent knows how the emotions of panic, alarm, joy, and jubilation can intermingle and become one; may your children bring you the same emotional gratification. Hero, mentor, role model, playmate, dance partner, confidant, keeper of secrets, and healer of wounds—to me, all these things add up to Fatherhood. I hope to live up to my daughter's expectations of a father, and may she grow to be more than the sum of her parents.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 1 The Lure of Game Programming

This chapter covers the following topics:

- The good and bad side of the game programming industry in general
- The requirements of a game development language
- The advantages and disadvantages of using Delphi as a game development platform
- Opportunities within the game programming industry
- The types of games currently under production

What pushes the advance of computing technology more than any other influence? Computer games. What provides a never-ending stream of content for dozens of colorful, flashy magazines? Computer games. What do hundreds of thousands, perhaps millions, of people play in their spare time for entertainment and competition? Computer games. What do the technologically adept talk about in newsgroups, in person, and in e-mail? Computer games. What inspires producers to create major motion pictures that flop at the box office and embarrass the people that both worked on them and created the original concept from which they were spawned? Computer games.

Well, perhaps the computer gaming buzz isn't quite that bad, but the computer gaming industry is a multibillion dollar industry that grows every year, and is filled with enough strife, antagonism, competition, and tactical warfare to fill several novels. Tens of thousands of web sites are dedicated to both the production and play of computer games. Dozens of magazines exist that also cover both sides of the gaming coin. Dozens of books have been written on game programming, such as this one, and even more books cover strategies and tactics for playing specific games. The gaming industry is very competitive and dynamic, and by this very nature, a little hostile. Gaming companies are formed and disbanded at an alarming rate, with new ones being born as others, both old and new, die miserable deaths.

As is true with just about anything in this world, the gaming industry comes with a light side and a dark side. There are many things about the gaming industry that would make most programmers cower in fear, yet there are just as many alluring attributes that serve as a powerful seduction towards pursuing a career in it. Although this book is not geared toward dwelling on the negative aspect of such things, let's start by taking a

reality check and examining the dark side of the game programming industry. Then we'll look at why the light side is so attractive to intrepid and enthusiastic programmers.

## The Dark Side of the Game Programming Industry

Game programming and the gaming industry in general have several negative aspects that may dissuade the casual programmer from doing much more than dabbling in it as a hobby. The items we're about to examine should not be considered an exhaustive list of condemning attributes, nor should they be considered a list of excuses for not pursuing a dream. As a purchaser of this book, you're at least a little curious about game programming, and perhaps considering a career change. Don't let these things convince you otherwise. Use the discussion of the following topics merely as food for thought to keep your hopes and plans in perspective.

### Difficult Learning Curve

Computer games push manufacturers to produce better, faster hardware because it is very difficult to produce some gaming effects that perform adequately enough to be believable. Although the PC is now capable of supporting some very intense graphics engines, it was never designed to do so. Without the assistance of dedicated hardware, some games would not be possible without very clever programming solutions.

They say knowledge is power, and that isn't more true than in the game programming industry. A new algorithm for shading or texture mapping more polygons in a shorter amount of time can mean the difference between a best-selling game (and thus a profitable company) or a flop (and thus layoffs, bankruptcy, and failed businesses). Because of this, game programmers and gaming companies tend to guard game programming techniques as if it were a matter of national security. Application programmers benefit from the assistance that Delphi (or MFC) gives them, and they can almost always find some example of how to accomplish their goal somewhere on the web or in a book. Game programmers, on the other hand, are often left to their own devices when attempting to solve a problem. Sure, one can usually find a basic example of how to accomplish a desired effect, but only if it has been done before and the technology is a few months old.



**Microsoft Foundation Classes (MFC):** Microsoft's object-oriented component hierarchy, similar to Delphi's VCL.

By its very nature, in order to create a commercial quality game that can compete with other games on a retail shelf, the programmers must employ advanced mathematical equations, professional quality music and sound, and highly detailed and artistic graphical images. Application programmers, on the other hand, usually have demands no greater than knowledge of client/server programming techniques, middleware, and database methodologies. These demands are by no means easy, but the knowledge for accomplishing these feats is usually more attainable than what a game programmer is called upon to know.

### Long Hours

Everyone logs extended time at the keyboard when under crunch mode. However, game programmers tend to be in crunch mode longer and more often than application programmers. This is often due to promises that the marketing department makes that the engineering department is required to keep. Because the gaming industry is so volatile, game applications need to be completed and out the door before a competitor releases a similar title. This requires game programmers to put in long hours; 16-hour days as a norm are not unheard of. Crunch mode for application programmers typically means 10- to 12-hour days. For the game programmer, it typically means that they sleep in their cube for an hour or two each night, shower at work (if they're lucky enough to have one), and get to go home after the title passes quality control.



**Tip:** If you're touring a company's offices during an interview, look for cots or blankets in people's cubes. That's a good indication of the type of hard work and dedication you'll be expected to demonstrate.

---

## Corporate Instability

The gaming industry is dynamic and cutthroat. Because there are so many game companies out there, with more appearing daily, competition is fierce. As technology marches on, generating better, more powerful machines, the demands on gamers rise at an ever increasing rate. The incredible burden on game developers to keep up with the demands of consumers, as well as to keep ahead of competitors, causes the production costs of mainstream game titles to approach that of Hollywood movies. The result is an ever narrowing profit margin.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Unfortunately, this tends to translate into a need for game companies to continually crank out best-selling titles. Extremely large game companies can afford a bad title or two, but smaller companies may be dramatically affected by bad sales of even a single title. This is especially true for startups; if that first title doesn't at least break even, it may be impossible to even meet employee payroll requirements, let alone to find enough capital or investors to fund further projects. Even companies that have been around for years could suddenly find themselves in financial stress after only a few bad games.

With the exception of only a few large gaming companies, this volatility seems to be the norm rather than the exception. This is all the more obvious by the number of computer gaming businesses that come and go daily. The end result is that gaming companies tend to be unstable, and are not a good place to go if you want a job that you can work at until retirement.

## **The Light Side of the Game Programming Industry**

In stark contrast to its negative aspects, the game programming industry exhibits a number of positive aspects that fuel its continued growth. The game programming industry in general is very exciting, in part due to its dynamic nature, but mostly because of the people you can meet and the technology you will work with.

### **Fun and Rewarding**

What sounds more exciting: creating a client/server application for tracking customer purchases and transactions (complete with reports and custom querying capabilities), or creating a first-person adventure game that lets you run through castles and destroy evil demons? If you chose the client/server application, close this book right now. Perhaps the coolest and most seductive trait of game programming is that you can use your imagination in creative ways not possible with application programming.

As opposed to conforming to Windows user interface guidelines, the game programmer can create highly specialized and flashy user interface elements that are both fun to code and exciting to behold. Creating buildings, cities, continents, and worlds to explore is tremendously entertaining for both the programmer and the user. While creating a tool that allows someone to accomplish a job quickly and efficiently is rewarding, creating a diversion that immerses one into a world of exploration and wonder for hours on end is even more so.

## Fame and Fortune

A good gaming engine is the holiest of holy grails in the programming industry. Pretty much anyone could make a client/server database if they wanted to (given the right training and education), but few can create a game the likes of Quake, Tomb Raider, or Wing Commander. Those that do, however, are elevated to the status of rock star, and are worshiped by thousands of adoring fans. It's always nice to be complimented for your work, perhaps even developing a reputation for quality products and performance amongst your peers. However, actually having groupies and fans takes the meaning of "good job" a bit further.

Another intriguing aspect of game programming is that it is one of the last remaining industries where individuals have the potential to make a lot of money in a short amount of time. With the continued growth of the Internet, it is feasible for a developer to create a game and make tons of money by selling it online. It may also be possible to sell a completed game to a gaming company, netting a nice large chunk or a share of royalties over time, but this practice is in a decline. Success stories of this nature are more the exception than the rule, but the potential does exist for a hard-working, dedicated game programmer to transform his or her labor of love into a cash cow.



---

**Tip:** If you're lucky, your company will agree to give you a cut of the royalties of any game you work on. Unfortunately, this is a rare and vanishing benefit, but could potentially make you a lot of money if the game sells well.

---

## The People

A psychology major could probably write a doctoral thesis on how the personalities of people in their chosen industry seem to be very similar. This at least seems true of those in the programming profession. What this means is that you'll most likely be working with people of similar interests, usually resulting in friendships and group social activities. Many gaming companies double as sites for late-night Dungeons and Dragons sessions, impromptu multiplayer Quake tournaments, and any number of other fun recreational activities. People in the game programming industry tend to be intelligent and have a good sense of humor, and in general are very enjoyable to work with.

## Game Programming, In Delphi?

When most people hear about Delphi, they think of rapid database application development. They think of its uses in the corporate environment as a wonderful tool for making myriad database accessing applications, a task for which it is indeed well suited. They compare it to other RAD tools like Visual Basic and PowerBuilder, contrasting their relative strengths and weaknesses in data access, speed, scalability, etc. This attitude has been prevalent since Delphi 1, and is indeed one of the reasons Delphi has done so well. Unfortunately, most people do not consider Delphi to be a viable choice for any other type of application development. Considering the fact that Delphi is built on a long line of well-used Pascal compilers, has full access to the Windows API, creates actual executable binary code, and shares the same back-end linker with Inprise's excellent C++ product, Delphi is indeed capable of doing much more than just enterprise-wide database applications.

In the game programming industry, the dominant language in the U.S. is C/C++. Let's think for a moment why that may be the case. Specifically, let's consider what could arguably be the most important attributes that a development language must offer in order for it to be considered a viable platform for games development.

1. It must compile fast, natively executable code. C/C++ offers this, and indeed you can find benchmarks that supposedly prove which compiler offers the fastest code. However, Delphi uses the same back-end linker that C++ Builder uses, and its compiled code is as fast as that of C++ Builder. Besides, the most computationally intensive parts of gaming applications are usually written in hand-optimized assembly in order to gain the maximum speed advantage. C/C++ may have a few lexical tricks that allow one to squeeze an extra clock cycle or two out of a few instructions, but you can

accomplish the same task in Delphi with creative programming efforts. In the speed department, some C/C++ compilers may create faster compiled code than Delphi, but you can always make up for it where it counts using hand-optimized assembly (just like the C/C++ boys do). Remember, a Pentium 166 with 16 MB of RAM is considered obsolete by today's standards, so these marginal speed advantages offered by some C/C++ compilers are much less important than they used to be. The bottom line is that Delphi is as fast as most C/C++ compilers when it comes to the natively compiled form of raw code, and when speed is absolutely essential, Delphi can compile hand-optimized assembly language as well as any C/C++ compiler.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

2. It must offer true pointer access to memory. As with C/C++, it's quite easy in Delphi to create a pointer, point it to some location in memory, and commence reading or writing as necessary. Like C/C++, Delphi also allows you to perform pointer arithmetic, moving the location to which the pointer points. This is a very powerful technique which, when used incorrectly, can cause spectacular system-wide crashes. C/C++ does not have any advantages over Delphi when it comes to using pointers.
3. It must offer the ability to manage memory allocation. This goes hand in hand with the ability to use pointers. Game applications typically push the limit on the hardware, and some must specifically manage the memory used by the application in order to realize maximum performance. Like C/C++, Delphi offers many ways to allocate memory, from the automatic memory management offered by the VCL, all the way to Pascal commands for memory management and Windows API functions for allocating and deallocating memory.
4. It must offer full access to operating system resources. Again, just like C/C++, Delphi has full, open access to the entire Windows API. Indeed, Delphi can use any DLL or type library to which it has an import unit. This is one area where C/C++ does have an advantage over Delphi. Since Windows is written in C/C++, DLLs and other resources created by Microsoft and other third-party developers typically have header units for C/C++ access before someone translates it into a Delphi unit. However, thanks to the efforts of the JEDI Project and other individuals, Delphi import units for hundreds of APIs are becoming available to the general public. Typically, Delphi will probably always be one step behind on access to the latest technologies. However, it's not too difficult a task to convert C/C++-headers into Delphi units, and you can usually find the converted unit on the Web or buy one from a Delphi vendor.

Most people will argue that there are several more attributes by which a development language must be judged before determining if it is a viable games development platform. As we've discussed, Delphi meets or exceeds all expectations for the above requirements, and could probably do the same for any other requirement that one could think of. Pascal is an old language; it's been around for quite a while, and is mature, stable, and proven.

It's unfortunate that many people still think of Pascal as a beginner's language, fit for little more than hobby applications. This is probably due to the irrational Western mentality that if something looks complex, or if it is hard to understand, it must be powerful. Bear in mind that a high-level language like C/C++ or Pascal is just an interface to assembly language that is easier for humans to understand. Each language has a few small advantages over the other, but ultimately an application written in either language is transformed into a final

assembly language form. The only differences lay in how this transformation is accomplished by the compiler, and Delphi's compiler can create assembly language that competes with the best of them.

## Delphi as a Games Development Platform

So, Delphi is indeed a viable game development platform. What, then, can we expect to encounter when developing a game using Delphi? Delphi's incredible debugging features will be almost unusable if you are creating a DirectX game, primarily because of how DirectX monkeys with the display hardware. However, certain aspects of how the VCL encapsulates the Windows API will take a lot of the drudgery out of DirectX programming, and will indeed make some tasks extremely easy. Most notably, the TBitmap object will be very useful for loading graphics into DirectX memory blocks, and the TCanvas object can be quite handy when performing standard GDI functions, such as outputting text to the screen. Of course, if you will be using Delphi to create non-DirectX games, the full power of the debugging features as well as the GDI encapsulation will make the task of programming a game just as easy as that of developing a database application. In this book, we will focus on creating DirectX games, and we will examine how the VCL can make some aspects of DirectX programming almost trivial.

As a whole, Delphi offers many advantages over other game development platforms. Every language has advantages and disadvantages over every other, which is why there are so many languages to choose from. While the following discussion is by no means an exhaustive look at Delphi versus C/C++, Delphi does have some specific advantages and disadvantages when compared to C/C++ that merit further investigation.

### Advantages

This is a truly opinionated and arguable statement, but Delphi's Object Pascal is a much better language than C/C++ for two reasons: It is not as arcane, and it is much stronger typed. While it is possible to write easily understandable code in C/C++, it is just as easy to write code that is completely unintelligible and unmanageable. C/C++ has a precompiler that allows one to redefine function names (among other things). While this can be quite useful in some instances, if used indiscriminately it can lead to a major headache when trying to decipher what a piece of code accomplishes. C/C++ has many other language constructs that allow one to write very archaic instructions decipherable only by its author or a C/C++ guru. The primary reason this is important is because, in today's development shops, a section of code may be written by one programmer but modified or updated by several others. It is an incredible waste of time and energy if a programmer tasked with the responsibility of updating a section of code has to spend several hours just determining what it does. The archaic quality of C/C++ code, which can be worsened by arrogant or apathetic programmers, tends to make the arduous task of modifying existing code very difficult, if not impossible. Alternatively, Pascal syntax is closer to regular English, and although it is possible to write some very arcane Pascal code, it is typically much easier to understand from a glance than C/C++. Some will argue that C/C++ was written by programmers for programmers, resulting in the greatly abbreviated lexical tokens perpetuating the language. While it is true that this may save a few keystrokes, in the opinion of this writer, Pascal syntax is much easier to understand from a brief glance and its format and construction make it much easier to decipher an algorithm from its implementation.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**

[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

This goes along with the strong typing imposed by Pascal compilers. Pascal is used as a beginner's language in universities and colleges because it forces programmers to write in a very structured, easily understood manner. The result is that programmers are forced to develop good programming practices. C/C++'s language structure is much more open and unrestricted. While this does allow one to perform some lexical tricks that may be difficult to reproduce in Delphi, it can result in code containing bugs that are very difficult to find. Delphi's strongly typed compiler helps find logical errors and mistakes that may have compiled without so much as a warning in a C/C++ compiler. Writing Delphi code is never bug free, but the strong typing imposed by its compiler helps the programmer find some bugs faster than would be possible with the same code in a C/C++ compiler.

Another great advantage to using Delphi is the overall attitude of the Delphi community at large. Overall, the Delphi community has a very helpful attitude. There are literally thousands of Delphi sites featuring thousands and thousands of free objects, code tips, and applications with source code. There are dozens of sites available that deal specifically with Delphi game programming, and there are several freely available game programming examples and objects, complete with source code. The JEDI Project, the Delphi Games Creator, and DelphiX are just a few examples of the type of free source code that one can obtain when searching the Web for Delphi game programming knowledge (many of these are contained on the accompanying CD). While you can find just as many (if not more) resources for C/C++ game programming, the Delphi community has an overall helpful attitude, indicative of the type of Delphi programmers that exist in the world.

### **Disadvantages**

The biggest disadvantage that one may face when pursuing Delphi game programming is in the lack of knowledge available in the industry. Almost every book you see discussing game programming will contain examples written in C/C++. Fortunately, the algorithms discussed in these books can be implemented in just about any language, but the code examples will be almost worthless unless you have a working knowledge of C/C++. This may require the aspiring Delphi game programmer to take a course or two in C/C++ in order to gain the knowledge to put the examples in the myriad of C/C++ game programming books to work in Object Pascal. Unfortunately, the result is an increased learning curve for future Delphi game programmers. It is interesting to note that when Delphi appeared several prominent figures in the game programming industry took note of its abilities and even heralded it as the next great game development platform. The editor of a

major gaming industry periodical even predicted that the best games of the coming year would be written in Delphi. This same periodical, and others, have run several articles illustrating game programming techniques using Delphi, written by some of the most visible programmers in the Delphi industry (most notably Charlie Calvert). Hopefully in the future we will see more books written about game programming in Delphi, spreading the knowledge of game application algorithms and generating more jobs for the Delphi games programmer.

Another major disadvantage already touched upon is the fact that Delphi programmers will probably have slower access to the newest programming technologies than C/C++ programmers. We can use DirectX technology as the perfect example. Microsoft has seen fit not to include a type library for the DirectX COM objects. Had a type library been included, Delphi could directly import the library and offer developers immediate access to all of the functions and procedures offered by DirectX. Instead, Delphi developers must translate the C/C++ headers that define the function calls, constants, and data structures by hand. This is a very onerous, time-consuming task, prone to errors that may be difficult to fix. Unfortunately, this means that most Delphi programmers must either write an import unit themselves or wait for one to be made available (for free or for purchase) before using the latest technologies. Thanks to the efforts of the JEDI Project and the individuals participating in it, many technologies previously out of reach of Delphi programmers are now becoming available. Indeed, Erik Unger, a JEDI Project participant, wrote the DirectX import units around which this book is based. Hopefully in the future we will see more cooperation between Microsoft and Inprise, which will result in immediate access to the latest technology by Delphi programmers.

## Opportunities

The game programming industry is as competitive and hostile as the movie production industry. Venture capitalists and investors are quick to throw millions of dollars—more than \$2.5 billion a year— at promising startups in hopes of grabbing a piece of the pie. Production costs of games typically average in the millions, and game companies rise and fall almost as frequently as most people change socks. In such a volatile environment, how can one expect to make any progress without being trampled?

Fortunately, the gaming industry is one of the few industries in which someone can compete from their garage or home office. For example, take the game Solitaire, a standard game shipping with Windows. Solitaire is arguably the most popular game in existence today. Probably everyone who has ever taken a look at Windows has played it at least once, and there are many who play it daily. This is a simple game that by no means pushes the envelope of technology, yet it offers an addictive, entertaining pastime that many people enjoy. A similar phenomenon, Tetris, has been copied and cloned to the point where you can find freeware Delphi Tetris components ready to be dropped on a form for instant gaming satisfaction. Theoretically, if Solitaire and Tetris can be such runaway hits, anyone with a good idea that translates into a simplistic game can do just as well.

For example, take a look at Deer Hunter. It's a relatively simplistic game using technology that is well documented, yet it is one of the best-selling games in recent history. It took the developers three months to complete, and it is certainly nothing that would be beyond the reach of a Delphi developer. Its concept is simple, it's aimed at the mass market, and the price point is right (it runs for about \$20).

With the global distribution available in the form of the Internet, and third-party companies offering the ability to take credit card orders for your software (for a nominal fee or percentage), it is possible for even the most remote programmer to compete with the big boys. The market for game software is much more open than the vertical markets (such as exist for word processors and spread sheets), and has a wider audience than that available for niche markets. Although you may not get rich from selling a \$5 shareware game, you could become quite wealthy by selling several \$5 to \$10 games.

As a reality check, bear in mind that the process of making games is no longer a one-man operation. Most games consist of good programming, good graphics, and good sound, and the combination of programmer, artist, and musician would be the equivalent of a modern-day Renaissance man. Indeed, to compete with such games as Quake, Wing Commander, or StarCraft, a team of several programmers, artists, content writers, game designers, actors, musicians, and sound engineers would be required to make a serious effort. Even for the most simplistic game, the talents of an artist should be sought out for all but the most elementary of graphical resources. Although it is possible to compete from one's garage, employing the talents of several

individuals may be required to meet the demands of your users.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

### Untapped Markets

The most actively pursued market for games are males from age 16 to 25 who buy several games a year. These are what you would call "hard-core gamers," and are the type of individuals who buy games like Quake, Tomb Raider, and Command & Conquer. This market is the most competitive, results in the biggest profit margin, and is pretty tied up by all of the big names in the game programming industry. It is these types of people who demand games that would require the aforementioned team of people to make any serious effort to create.

Are males ages 16 to 25 the only demographic to which games should be marketed? Certainly not. There are many untapped age groups and individuals that potentially offer an even greater market share of the gaming industry.

### Children

Children's software can be big business, with companies like Microsoft and others offering software that interacts with animatronic dolls or other devices. However, given the wide range of age groups and the commensurate knowledge to be learned at these ages, a plethora of software can be aimed at this group that combines education with entertainment. Several big-name companies are starting to dominate in this market, but there is still a lack of software for certain age groups that could be filled by the enthusiastic Delphi game programmer.

### Females

Women and girls use computers just as much as men and boys do, yet there is an incredible void in available software aimed specifically at this group. In a very generalized manner of speaking, most gaming software aimed at males is violent, bloody, and very fast paced. Taking into account that there are exceptions for every rule, this type of software typically does not interest most females. A huge untapped market exists for game software aimed specifically at women and younger girls, and any research into recent articles in gaming magazines can turn up scores of discussions and analyses of the types of games that would do well with this group.

## Casual Gamers

This group is starting to receive a lot of press coverage. A seemingly elusive and hard to reach group, the casual gamer is one who uses his or her computer for entertainment, but does not play the games aimed at hard-core gamers. These people like Solitaire and Tetris, and include everyone from housewives/househusbands taking a break while the baby sleeps to business people looking for something to do during lunch. Typically, this group doesn't like games that require long commitments, as you would see in games like Shogo or Final Fantasy. They like games that you can play quickly, something you can drop into and out of during lunch or a quick break. Many people are now trying to market to this group, and some are having better luck than others. This group presents the largest potential, as there are many more housewives/househusbands and business people out there than there are hard-core gamers.



---

*Note:* For a look at some professional games written in Delphi targeted for this market, visit <http://www.gametropolis.com>.

---

## Types of Games

There are almost as many game type categories as there are games. Limited only by the imagination of the programmer, a game can take literally any form, as long as it is entertaining and interactive. There are probably many more game genres left to be discovered by inventive programmers than what is listed here. When Id came out with Wolfenstein 3D, and later Doom, an entirely new category was launched that spawned many successful imitations. Games such as Dune 2 and WarCraft heralded the real-time strategy genre, perhaps even more popular than first-person shooters. Below is an arbitrary categorization of different game genres, discussing the features typically implemented in a game of that category. Delphi, like C/C++, is adequately suited for producing games that fit into any category.

### Real-Time Strategy

This genre, made popular by games such as Dune 2 and Command & Conquer, is one of the most prevalent in the industry. Real-time strategy games are typically war games that require the user to manage various units and other resources, attacking opponents and gathering supplies while the computer (or other human opponents) does the same. This action takes place in real time, meaning that everyone's activity occurs simultaneously and continuously, and while the player is gathering resources in one area, other units owned by that player may be under attack somewhere else. Games such as StarCraft, Command & Conquer, and Myth all fall under this category.

### Turn-Based Strategy

This category is similar to real-time strategy, in that these games are typically war games where the user manages various units and resources. However, unlike real-time strategy games, the action in a turn-based strategy game does not happen simultaneously. One player takes a turn ordering units to attack, defend, or gather supplies, while all other players wait. When the current player is finished, the next player in line takes a turn. Games such as X-Com and Civilization fall under this category.

### First-Person Shooters

By far the most popular of all games, first-person shooters place the user in a virtual world, offering a first-person perspective of the surroundings and featuring freedom of movement and interaction with the virtual environment. These games typically have a multiplayer aspect, and offer a feature known as deathmatch. In a deathmatch, players usually hunt each other down in a virtual game of tag. Highly addictive and incredibly entertaining, games in this category are usually the most complex and competitive of all games in the entire industry. Games such as Quake, Halflife, and X-Wing vs. Tie Fighter fall under this category.

### Adventure Games

While some first-person shooters could arguably be categorized under this heading, adventure games typically portray a virtual world with the player depicted in a third-person environment, usually in an isometric perspective. Interaction with the environment is usually very detailed and extensive, and a complex plot line gets the player involved in the story around which the game is based. Games such as the Ultima series and Fallout are in this category.

### Action Games

This broad category involves games that have more action than substance, such as pinball games and arcade style games. Quick action is the focus of these games, and while there may be a little bit of a plot line or “purpose,” it usually serves as little more than a thin justification for the action taking place. These “twitch” games are fun, highly addictive, and do not require long-term commitment. Games such as Asteroids and Missile Command fall under this category.

### Puzzle Games

Another broad category, puzzle games are games of logic that challenge players to exercise their cerebral muscles. Typically, they are abstract games that involve various fundamental abilities such as pattern matching or color grouping. This category includes games such as Tetris or Lose Your Marbles.

### Sports Games

Arguably a category all its own, sports games are computerized simulations of various real-world sports. Several big development houses have created some very realistic simulations of various sports, going so far as to include current rosters of professional athletes, their likenesses, and an estimation of their talents. Games such as NHL 99, Madden 99, and BFL fall under this category.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

### Board and Card Games

Perhaps the most common category for casual game programmers, this category is composed of games that could be played with cards, dice, or similar real- world gaming instruments. Typically, they simulate well-known games such as chess, checkers, or poker. These games are typically the easiest for first- time game programmers to create, as they tend to have well-defined rules that translate into well-structured algorithms, and their graphical requirements are not as severe as those of other games. Games such as FreeCell and Monopoly are in this category.

### Simulations

More a software toy than a game, these are applications that attempt to simulate some real-world process as precisely and accurately as possible. They tend to be very detailed, and while fun, typically require a long-term commitment from the player in order to obtain maximum gaming pleasure. This commitment is required in order to see the simulation grow from a beginning or newborn stage into a mature, elder stage. Games such as Sim City and Sim Tower fall under this category.

### Windows, Delphi, and Games

It goes without saying that Windows 95/98 is the most popular platform for computer games. Delphi, as we've discussed, is as powerful and capable as C/C++ when it comes to making games (or any type of Windows application, for that matter). Given the fact that there are many open markets as yet untapped by the big names in the gaming industry, there exists a huge potential for the enthusiastic and dedicated Delphi game programmer to compete and make a profit. Although this book won't show you how to design or market your game, it will show you the basics of game and graphics programming that will serve as a starting point for a future in game programming.

### Summary

In this chapter, we discussed several aspects about game programming in general and Delphi in particular as it relates to game programming. When contemplating a game application concept or analyzing Delphi as a game development platform, it is important to keep these points in mind:

- As is true with just about anything in this world, the gaming industry comes with a light side and a dark side. There are many things about the gaming industry that would make most programmers cower in fear, yet there are just as many alluring attributes that serve as a powerful seduction towards pursuing a career in it. In general, the game programming industry is plagued by difficult learning curves, long hours, and an unstable business environment. However, programming games can be a lot of fun and very rewarding, and can earn the successful individual a substantial amount of notoriety and wealth.
- When most people hear about Delphi, they think of rapid database application development. They think of its uses in the corporate environment as a wonderful tool for making myriad database accessing applications, a task for which it is indeed well suited. However, in the game programming industry, the dominant language in the U.S. is C/C++. If the most important attributes for a language to be considered a viable game development platform are fast, natively executable code, true pointer access to memory, memory allocation, and access to operating system resources, Delphi easily meets or exceeds all of these requirements. Ultimately, Delphi is as capable a game development platform as any other language, including C/C++.
- As a whole, Delphi offers many advantages over other game development platforms. Every language has advantages and disadvantages over every other, which is why there are so many languages to choose from. In general, the Object Pascal language is much more readable than C/C++, and its strong typing typically results in decreased debug time. There are also many free resources available to the Delphi programmer, and several enthusiastic mailing lists, web sites, and organizations dedicated to helping Delphi programmers at all experience levels. Unfortunately, most of the current game programming knowledge exists in the form of C/C++ examples, requiring the potential Delphi game programmer to have at least a passing knowledge of C/C++ in order to grow and learn new game programming algorithms. Delphi programmers also typically do not have immediate access to new technologies introduced by Microsoft due to lack of type libraries and Object Pascal import units.
- The game programming industry is as competitive and hostile as the movie production industry. Venture capitalists and investors are quick to throw millions of dollars—more than \$2.5 billion a year—at promising startups in hopes of grabbing a piece of the pie. Fortunately, the gaming industry is one of the few industries in which someone can compete from their garage or home office. Games such as Solitaire, Deer Hunter, and Tetris are very popular and successful, even though they don't push the envelope of current technology. With the global distribution available in the form of the Internet, and third-party companies offering the ability to take credit card orders for your software (for a nominal fee or percentage), it is possible for even the most remote programmer to compete with the big boys. Untapped markets such as the homebound, children, women, and casual gamers represent a potentially huge opportunity for enterprising Delphi game programmers.
- There are almost as many game type categories as there are games. Limited only by the imagination of the programmer, a game can take literally any form, as long as it is entertaining and interactive. In general, games can be segregated into the following categories: real-time strategy, turn-based strategy, first-person shooters, adventure, action, puzzle, sports, board and card games, and simulations. Delphi is suited for creating games that fit into any of these categories.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 2 The Anatomy of a Game

This chapter covers the following topics:

- The three vital elements of a game
- Basic game architecture
- Game loops
- Game states

Application design is an art unto itself. Several books have been written on the topic of effectively designing an application. Indeed, in the computer industry one can even specialize in such a discipline (if you're a programmer, you probably know someone with the title of "Chief Architect" who only designs applications and no longer writes any code). Application design can be childishly simple or diabolically complex, depending on the task the final application must perform.

However, from a purely academic perspective, all computer applications pretty much do the same thing: receive input, process it, and create output. In typical business applications, the input comes from either a device (commonly a mouse or keyboard) or a database connection, the processing typically involves mathematical calculations, and the output generally results in either a spreadsheet style of data display or (more excitingly) a colorful graph. By contrast, game applications must accept input from several different devices, processing the input by moving onscreen objects around, performing collision detection, playing sound and music as the situation demands, and deciding if missiles are fired or something has died. A game application's output involves drawing the results of these frantic calculations to the screen.

Game application architecture can vary as wildly as business application architecture. However, all games must perform similar actions, usually in a common order. In this chapter, we examine the various components that make up a typical game application. We will study an arbitrary application architecture that can be used for games, and we will discuss the individual subsystems of this architecture, focusing on the functionality they provide to the overall game application. The focus on this chapter is on the theory of game application construction, and as such presents very little actual code examples. However, several pseudo-code listings will be illustrated that encapsulate the techniques discussed in this chapter. Additionally, a

simplistic game example is included on the CD for this chapter that demonstrates this architectural theory in a more real-world implementation.

## **Game Dissection: The Vital Organs**

Games can be incredibly complex, with several different subsystems performing any number of functions that control the overall application. Many developers may feel a little overwhelmed by all the details that go into creating a game. However, like most things in life, game programming is not nearly as complex as one might at first imagine. It's certainly not an easy discipline; one must be very skilled in their talents and have the ability to apply them in creative ways in order to produce new and exciting effects and gaming environments. Fortunately, dozens of books have been written that discuss various game programming techniques in many genres, from 2-D turn-based board games to 3-D fast action, first-person shooters. Anything that one person can learn, another person can learn as well; game programming is a constant quest for new knowledge and new techniques, which gives the field one of its many enticements.

The many game programming books on the market (this one included) typically deal with very specific game programming techniques that demonstrate how some particular game environment or effect can be produced. This allows one to learn specific talents in specific areas, possibly specializing if so desired. While game programming is as varied and diversified as business applications programming, when viewed from a high-level standpoint all games can be reduced to three necessary parts: graphics, sound, and input.

### **Graphics**

With the exception of user interactivity, graphics are probably the most important part of a game in today's market. Game developers are now using software that was once only used by movie and television production houses to develop graphics for games, and the results are nothing short of amazing. Tools such as TrueSpace, LightWave, and SoftImage, which have all been used to create outstanding special effects for both movies and television, are a necessity for serious game graphics production. Almost any major game includes several animated sequences (usually produced by one of these tools), some of which are on par with action sequences seen in a major motion picture. Game element graphics have also seen amazing improvement over the last several years, with many interactive elements starting their lives as high-resolution 3-D models. However, as important as good graphics are to today's games, it is equally important to realize that just because a game has good graphics, it does not mean it is fun to play. Too many games are made with the theory that if you put in good graphics, users will buy it. Graphics are important for that first impression when a potential consumer picks up the box to have a look, and indeed for the continued enjoyment of the game, but graphics should functionally act as only a complement to the overall game design and playability.

Game graphics can further be broken down into two elements: background graphics and interactive images, or sprites.

### **Background Graphics**

Background graphics set the mood for a game. Usually, they define the environment in which the game world is set, and they provide a reference point for the user. Typically, background graphics are static, and represent objects that the user either cannot or rarely interacts with. For example, background graphics include the vehicle fuselage of a cockpit in a flight simulator, the radar control console casing in a real-time strategy game, or the surface over which puzzle pieces slide in a pattern matching game. Sometimes, background images may animate in some way, providing more realism to the environment and enhancing the overall gameplay experience (for example, moving the heads of spectators in a tennis game).

### **Interactive Images**

The most important graphical element of any game are those graphics with which the user interacts. Either by directly controlling the movement and actions of a specific graphical element, or by indirectly affecting the movement and actions of graphical elements through the direct manipulation of other graphical elements, it is these interactive images that represent the actual game play. Typically called sprites, these images generally move about the screen, animating as they change position from place to place.



---

**sprite:** An image that animates and moves around the screen in some fashion. Typically associated with images that the user can interact with, either directly or through interaction with other sprites.

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Sprites take many forms, and serve as a powerful element for increasing realism and improving a user's overall enjoyment of the game. For example, sprites include asteroids in a space game, attacking orcs in a war game, and even your brave soldiers who are about to get butchered by the attacking orcish hordes. Sprites can even be used solely for improved background animation, such as birds flying overhead or vultures picking at the remains of dead orcs and human soldiers.



*Note:* Creating art for games can be difficult if you're not an artist and do not have access to one. However, there are several freeware art libraries available on the web that contain sprite images. Check out the SpriteLib sprite graphics library by Ari Feldman on the accompanying CD.

## Sound

Sounds in a game application can add depth and richness that is simply impossible to achieve with graphics alone. Sound immerses a player in the world represented by the game, increasing both the realism and the enjoyment of the experience. Technically speaking, it is possible to create a game without sounds of any kind. However, such a game, while possibly enjoyable, will have a feeling of incompleteness and will leave the user with the impression that something is missing.

Just as graphics have progressed from their once-primitive lines and flat polygons, game sound creation has become a specialty field in its own right. Indeed, it is now common for well-known rock musicians to provide music for today's hottest games. Larger game companies are hiring Hollywood sound designers to create realistic sound effects. Even some movie and television production houses are now selling their sound effect libraries for use in game applications.

Game sounds can be broken into two component parts: sound effects and music.

## Sound Effects

Good sound effects can add great depth to any game. An Asteroids clone might be fun enough to play

without sound effects, but when you add the whine of laser blasts, the explosions of destroyed asteroids, the thrust of acceleration, and some blips and whirs to go along with normal UI elements such as buttons and menus, the game takes on a whole new dimension. Hearing the scrape of metal as your hero unsheathes his sword and a solid thud as the orcish leader's head falls to the ground after being severed adds both realism and drama to a game. Such a sequence would not be nearly as effective without the accompanying sound effects.

More importantly, sound effects can give the user audible clues as to changing game states. The typical game screen is busy enough, with the user trying desperately to outrun marauding pirates while avoiding cannon fire and shallow water and rocks. Sound effects can relate important information to the player, freeing her up to attend to the situation at hand instead of scanning the game interface to make sure that some critical system isn't on the verge of destruction. For example, a space flight simulator may feature a ship with a shield rating. As the user flies around trying to avoid destruction while accumulating the maximum number of kills, the shield readout must be continually checked to make sure that the ship is still safe from harm. Implementing a sound effect such as a voice saying "Shields Low" will both free the user from a repetitive task as well as induce a sense of urgency when the user hears the sound effect. Another example of this type of audible clue is the typical "We're under attack" heard in real-time strategy games. This frees the user from constantly scanning the map, which can quickly become a tiring chore, if not impossible. Even simple clicks or beeps on buttons and menus can greatly add to the realism and enjoyment of a game.



*Note:* Magazines and books that cover the movie industry, particularly the special effects field, sometimes contain information on how to make sound effects. These same techniques can be used to create awesome sound effects for games.

---

## Music

Like sound effects, music can add a tremendous amount of depth to a game. Music helps to define the mood for a game, and can work wonders for building tension during climactic moments, a sense of euphoria after a victory, or a sense of melancholy after a defeat. The best games use music that changes with the context of the game, building a sense of anticipation during slow moments just before an ambush, and changing to a heart-pounding rhythm during the attack. The best example of the effective use of music to evoke emotions is in movies. Pay attention to how ballads can produce a feeling of romance or loss, and how anthems can just as effectively make you feel king of the world. A game that can use music as effectively as a movie will generally provide a greater level of enjoyment for its users.

Unlike the other constituent elements of a game application, music may not always be an appropriate addition. While it may add depth to some games, it may be distracting or even a bit annoying in others. An upbeat, urgent rhythm played with electric guitars may be almost essential to a fast-paced aerial dogfight game, but the user may prefer total silence when playing a deeply tactical game such as chess. Thus, it is usually a good practice to include an option to turn off music (or even sound effects).

## User Input

User input is by far the most important part of any game. It is the reason that a game is played in the first place. Without a means of providing some sort of interface so that the user can interact with the game elements, the game is little more than an elaborate animation. The entertainment value of a game comes from this user/game element interaction. If the user wanted to watch a non-interactive animation, she would have watched TV or gone to the movies.

Providing a natural means of user interaction is a very important part in the overall design of a game. While a game may be pretty to look at and pleasant to listen to, if it is difficult to use, it won't be used long. The most appropriate means of user interaction will depend on the type of game. Obviously, flight simulators would best be controlled through a joystick of some sort, while games with lots of moving pieces that can be repositioned freely should probably use the mouse. For other games, the keyboard may be the most natural means of user interaction. Regardless of the most appropriate input device, it is good practice to support input from a wide variety of input devices. This gives users more control over what feels intuitive to them, as some

may feel more comfortable using the keyboard over the mouse, and vice versa.

User input can be broken down into two parts: standard input devices and peripheral input devices.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Standard Input Devices

Since games must provide some method for the user to interact with the game world, it would be nice to know if the end user's machine had a specific input device attached. Fortunately, it would be a rare occurrence indeed to find a machine that did not come with both a mouse and a keyboard as standard attachments. For many games, this combination of input devices works best, and most games may not need any other type of input device. Therefore, it is pretty safe to assume that the user will have at least the mouse and the keyboard available as input devices. Most, if not all, games now support both of these devices, in addition to others.

## Peripheral Input Devices

Peripheral input devices consist of game controllers that are not standard equipment with new computers. Such devices include joysticks, game pads, rudder pedals, and a plethora of other types of input devices of every shape and size. These devices are generally suited for use with "twitch" style games that require good hand-eye coordination, such as flight simulators or first-person shooters. However, several of the more esoteric devices may provide a much more intuitive feel for the user, as opposed to using just a plain keyboard or mouse or joystick. Fortunately, most of these devices are accessed programmatically in a very standard way. This allows the developer to support several devices using the same code base, although some specialty devices may require slightly different coding methods to support their use. As stated above, it is a good practice to support a wide variety of peripheral input devices so that the user will have a choice as to which device feels the most comfortable.

## Real-Time Event-Driven Programming

Anyone who has done any amount of serious work in Delphi is now intimately familiar with the event-driven model of application programming. Delphi developers routinely write event handlers, either by creating them manually through the IDE or by hooking them up programmatically in code. It's an inescapable practice, and it makes application coding and debugging both faster and easier.

Game programming, under Delphi or any language, is similar to application programming in many ways, but it is fundamentally different in a few areas. While object-oriented game programming shares the practice of

hooking up events, there's a slight twist to this model that is required for supporting the types of games that are commonly created today.

The model most programmers, especially Delphi programmers, are now very familiar and comfortable with is known as event-driven programming. The application is written in such a way that its functionality is triggered by external events. These events can take the form of programmatic signals from other applications or devices, but for the most part, these events are triggered from user input. Until some event is triggered, it simply sits there waiting for an event to fire. For example, consider a word processor. It might have a few events that are triggered from a programmatic source, such as an autosave function fired off of a timer event. However, if you open a word processor and do nothing, it will simply sit there until the user either presses a keyboard button or activates something with the mouse. Event-driven programming is a very effective and useful method by which very robust applications can be effectively written.

Game applications, on the other hand, are in continuous motion. Events are constantly being fired based on the current game world situation, and the application continues to process with or without user input. This is the most fundamental difference between game programming and application programming, and is known as the real-time event-driven model. The game application cannot sit there and wait until the user moves the joystick before continuing. The user expects enemy ships and objects to be in constant motion, and would probably be surprised if they were not killed in some spectacular way if they just sat there. Sure, some games are written with the same event-driven model as regular applications, such as a Solitaire game, a chess game, or some other turn-based game. However, arcade-style action games that exhibit constant motion must be written using the real-time event-driven model.



**Note:** It is possible to use Delphi's input events such as OnKeyDown to provide event-driven input to a game. However, while this may work for some games, this is generally not the best approach. We'll look at more desirable techniques in Chapter 7, Input Techniques.

## Basic Game Architecture

There are probably as many different ways to design and architect game application code as there are game applications themselves. Each game will have unique and individual requirements based on the style of game and what it does. However, most games must perform similar tasks, usually in a similar order. With this in mind, let's examine an arbitrary game architecture that will support the real-time event-driven model as discussed above.



**Figur 2-1:** Basic game architecture

Basically, every game can be divided into six different modules:

- Initialization
- Game Introduction
- Game Startup
- The Game Loop
- Game Ending
- Shutdown and Exit

Typically, the Initialization and Game Introduction modules are performed at the start of program execution, and usually only run once. The user is then presented with the Game Startup module, which generally contains code for letting the user customize the game in some manner. The Game Loop module is then entered, which contains the primary logic for the game. When the game is over, the Game Ending module presents the user with information on their game performance, and then jumps back into the Game Startup



module. The Shutdown and Exit module is finally called when the user wishes to discontinue the game and shut it down.

## Initialization

The initialization module is responsible for setting up the beginning game state. It allocates needed memory, initializes global variables, creates lookup tables, and loads resources such as graphics, sound, and music files. Typically, the initialization module will display a splash screen at the beginning of the initialization code so that the user has something to look at. This is an especially good practice if game initialization takes a while; otherwise the user may suspect that the application has frozen or stopped executing.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

In Delphi applications, the developer has a number of areas where initialization code can be located. The main form's OnCreate event is the most obvious location, but such code can also be placed in the OnActivate event or even in the initialization section of the unit. The initialization module typically contains code that is only run once at the beginning of application startup. Any variables that will be reset when a new game is started, such as the player's score or starting level, should be modified in the game startup module.



**Caution:** Pay attention to any dynamically created objects or dynamically allocated memory, and make sure to free these upon termination. Games that leak resources and cause the user to reboot the machine after play are not looked upon favorably.

**Introduction**

The introduction module is responsible for introducing the player to the game and welcoming them into the game world. Typically, this takes the form of some sort of animated introduction. Most games, especially those constructed by professional game production houses, will display an elaborate animated sequence that gives a little background to the game and sets the overall mood. Some opening sequences are as good as action movie trailers, and may even feature live actors. The goal with these introductory animations is to immediately pull the player into the game world by eliciting the appropriate emotions and generally getting them into the right mood. Even a simple text display of a short story can be effective if written correctly. Usually, the game will provide a method by which the user can view the introduction again if they so desire, to review the overall game goals and objectives or to simply experience some cool animation again.

While introductory animations are important for setting the mood and the story, it is equally important to remember that all introductions, no matter how cool, will become boring at some point. Some games will only show the introduction when the game is executed for the very first time. Introductory sequences will slow down users who play the game often, so it is important to remember to provide some method to skip the introduction, such as a keypress.



*Tip:* Always allow users to bypass any animation sequence by pressing a hot key.

---

## Game Startup

The game startup module is responsible for two things. First, it should reset any variables that need to be initialized when a new game is started, such as a player's score, their level, the number of lives left, etc. Second, game startup should allow the user to customize the game in any way supported. For example, game startup should include the code that allows the user to select which input device will be used, what side he'll be playing, or other game-specific options.

## The Game Loop

The game loop module is the focal point of the game application. It contains the primary game logic, and executes the code that creates the game environment and controls all game objects. Its primary responsibility is processing user input and rendering the game output.

The game loop itself is a set of actions that are performed in rapid succession. These actions are performed over and over in a loop, updating the screen to reflect the current state of the game world. These actions are performed with or without any input from the user. Each pass through the game loop updates the screen one time, and represents one frame of animation. The code located in the game loop usually receives the most attention throughout the development cycle of a game, as this code must be highly optimized to run fast enough to achieve an acceptable game speed.

What should be in the loop? That depends on the specific game, and varies on an individual basis. However, most game loops perform similar functions. The game loop is examined more closely below.

## Game Ending

The game ending module is responsible for reporting to the player their overall game performance. This can take many forms, from a simple display of high scores to a summary of several different game statistics, or even an ending animation displaying a different scenario dependent upon whether the user won or lost. Whatever the method, the game ending module should let the user know that the game is over, and provide them with a choice to either start a new game or exit. If the user wishes to play another game, it should give control to the game startup module; otherwise it should execute the shutdown and exit module.

## Shutdown and Exit

The shutdown and exit module is responsible for de-initializing variables and terminating the game application. It should free any dynamically allocated resources that were created in the initialization module, delete any temporary files, and generally ensure that the application will terminate without memory leaks or other artifacts. If the game offers the user a method by which it can be saved, it is usually a good idea to ask the user if they wish to save the current game before exiting. This will avoid frustrated and angry users who have spent several hours playing the game just to have their efforts destroyed by accidentally choosing the wrong option.



*Tip:* Always ask the user if they wish to save the game before exiting (if game saving is supported).

---

This module should be reachable from several different points in the game. At any time, a user may want to end the game and exit the application, and this should be obtainable from the game startup, game loop, and game ending modules. Typically, several different methods are employed that allow the user to exit the

game. For example, there is usually a button or menu item in the game startup and game ending modules that will exit the game. Hot keys are usually available from within the game loop that either give control to the shutdown and exit module or return the user to the game ending or game startup modules so that they can exit from there.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ Search Tips
- ▶ Advanced Search

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

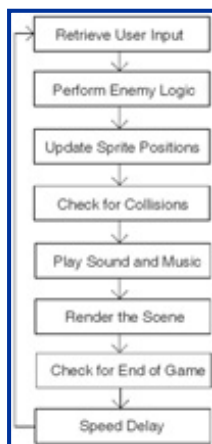
Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**The Game Loop**

As mentioned above, the actual game loop is usually the focal point of a game application. The game loop is where all of the action takes place. It processes a series of instructions over and over as quickly as possible to produce the on- screen images that reflect the current state of the game. This processing continues with or without user interaction, and is precisely the method by which the real-time event-driven model discussed previously is implemented.

Technically speaking, a game loop is responsible for three things: retrieving input, updating game objects, and rendering a new frame of animation. Of course, there's a lot more to a game loop than this oversimplified description suggests. What actually occurs in a game loop is very arbitrary and will depend on the needs of the individual game. However, all games perform similar tasks, usually in a similar order. Expanding upon the three tasks listed above, Figure 2-2 illustrates an example game loop that might be found in a simple action arcade game.



**Figure 2-2:** A simplistic game loop

There are several different techniques by which this game loop architecture could be implemented. In the examples in this book, we will be implementing the primary game loop inside the Application.OnIdle event.

This technique is used for several reasons. First, the main game loop will automatically be entered whenever the application becomes idle (after initializing everything in the OnCreate or OnActivate form event handlers or the initialization section of the unit). Second, the application object controls when the OnIdle event is called, meaning that any standard Windows events will be processed and the application never needs to call Application.ProcessMessages. Third, it's simply an easy, Delphi-centric technique for implementing the game loop that should be easy to understand and reproduce.

Alternatively, a while..do or repeat..until loop could be used to implement the game loop. The advantages of this technique are that the developer will have more control over the application, and it will run a little faster than the OnIdle technique. The application could even exit the game loop for some reason, perhaps to display a simple options screen that didn't need the complex animation capabilities afforded by the game loop. The disadvantages to this method of game loop implementation are that these types of loops are generally a little harder to work with when trying to support general Windows events. Application.ProcessMessages must certainly be called at some point within the loop so that standard Windows messages can be handled. Also, the variable controlling the loop's execution must be set so that the loop can exit before the application can terminate; simply calling Close or Application.Terminate will not work. Using this technique can result in little issues such as these that interfere with the normal operation of a Windows application. They can usually be handled with minor workarounds, however, so using a loop such as this may be advantageous depending on the type of game being produced. The following listing demonstrates an implementation of the arbitrary game loop using a while..do loop.

***Listing 2-1: Arbitrary game loop example***

---

```
procedure GameLoop;
var
    TimeStart: Longint; // used to time the length of one loop iteration
begin
    {this while..do loop will continue processing the same commands in
    rapid succession until instructed to exit from the loop}
    while GameRunning do
    begin
        {record the current time upon starting the loop}
        TimeStart := GetCurrentTime;

        {retrieve input from the user by accessing whichever input device
        the user has chosen}
        RetrieveInput;
```

```
{perform any enemy artificial intelligence and other logic  
associated with game elements not controlled by the user}
```

```
PerformEnemyAI;
```

```
{iterate through all sprites, updating their positions accordingly,  
taking into account user input and enemy AI calculations}
```

```
UpdateSpritePositions;
```

```
{check for any collisions between objects that would result in a  
change in the current game state, such as missiles colliding with a  
target or the player's sprite impacting a wall or other object}
```

```
CheckForCollisions;
```

```
{start any sound effects needed, based on collision detection or  
other game states}
```

```
StartSounds;
```

```
{start any music needed, again based on collision detection or other  
game states}
```

```
StartMusic;
```

```
{all processing has been completed, so draw the current state of the  
game to the screen}
```

```
DrawFrame;
```

```
{determine if the game is still running, based on whether the player  
has died or some other deciding threshold has been reached}
```

```
GameRunning := IsGameOver;
```

```
{determine how long this cycle has taken; if it has been too short,  
    pause for the difference so that the game does not run too fast}  
If GetCurrentTime-TimeStart<TimingValue then  
    Pause(GetCurrentTime-TimeStart);  
  
    {process any Windows messages that are waiting}  
Application.ProcessMessages;  
  
end; // repeat this process over and over until the game has ended  
end;
```

---

As a matter of practice, several of these tasks may be combined or performed simultaneously. For example, since an application must iterate through every game object to update its position, it would be practical to draw each object as its position is updated. Also, collisions usually result in an object being destroyed or its course being altered, so appropriate sound effects could be started at the time of the collision detection. The point is that these steps represent an arbitrary game loop, and may be rearranged or combined (or even split apart) as the individual circumstances warrant.

Even with this expanded view of a game loop implementation, many of the details attended to within the loop are not fully exposed. Throughout this book, we will examine many game examples that contain highly varied game loops, containing various functions as required by the individual game type. Later chapters will discuss in depth the tasks performed by an individual game example's game loop, but for now, let's examine an overview of what each segment is responsible for within our arbitrary game loop.

## Retrieve User Input

Retrieving user input involves reading the input values from whatever device is supported or that the user has chosen, and translating that information into a form relevant to the control of the game. Most Delphi applications typically handle user input through event handlers, such as `OnKeyDown` or `OnMouseMove`. For business applications and even most strategy or turn-based games, this method is fine. However, fast action arcade games typically require more control over the process, and handling user input through events may impede or interfere with the proper execution of the game.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

- ▶ Search Tips
- ▶ Advanced Search

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

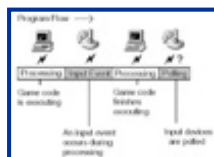
[Previous](#) [Table of Contents](#) [Next](#)

Thus, most games use a polling technique for retrieving user input. Polling involves directly accessing the device at a specific point within the game loop and reading its data. Theoretically, this polling could take place at any time, but it usually occurs at the beginning of the game loop so that the most recent movement can be reflected in the next frame of animation. Direct polling gives the application developer precise control over the processing of input, and can even allow the detection of multiple input events simultaneously (i.e., check if two keys are pressed at the same time).



**polling:** A method for retrieving input or checking some data value manually (as opposed to letting a separate thread check it automatically), usually repeated often at regular intervals.

The primary drawback to the direct polling technique is that user input events can get lost if the polling happens infrequently. This is generally a symptom of a slow game loop, where a large amount of processing takes place before the input device is again polled.



**Figure 2-3:** Missing a user input event

One method to remedy this situation is to conduct device polling at several different places throughout the game loop, and then process the sum of the movement data when it is time to update game world objects. However, another more accurate technique is to buffer user input events. To perform this technique efficiently, a separate thread must be utilized to perform the device polling and event queuing. Fortunately, DirectX provides several methods that make this technique viable, and we'll examine a demonstration of this in the chapter on input techniques.

Once the input has been retrieved, it is translated into events that are meaningful in the context of the game. For example, if the joystick button was pressed, the function for starting a missile or a bullet volley would be

called. How the input is translated is, of course, dependent on the type of game itself. The result of such user actions will be apparent after the sprite updating segment has been called and the next frame of animation has been drawn to the screen.

## Perform Antagonist AI and Other Game Logic

In this segment, algorithms that control the behavior of enemy units or other non-player units are performed, with the results being used to update the appropriate sprite positions or to execute any other appropriate actions. This may include deciding if an enemy spacecraft pursues the player or flees, but it also includes other, perhaps more mundane, decision making processes that affect other parts of the game. For example, background sprites that are never interacted with during the game may contain a little logic for changing their animation so that it does not appear repetitive.

This segment may run some very complex code depending on the type of game and its objectives. Some games, such as chess, may have a very advanced thinking engine that examines the current configuration of the board and makes a movement decision based upon the prediction of the next move that the opponent may make. Other games, such as real-time strategy war games, may have several smaller decision engines that determine the most direct route a unit may take to its destination. In general, this segment executes the artificial intelligence code that determines how the game objects will react to the actions of the player.



**artificial intelligence (AI):** The concept of cognitive software that thinks and reacts to situations in a sentient manner. In relation to game programming, this is the section of code responsible for determining the actions and reactions of game protagonists in relation to the actions and reactions of the player.

---

Artificial intelligence is a specialty area in its own right, and some game companies hire programmers specifically for their AI expertise. Some AI techniques are well documented and are easy to implement, such as chasing or evading the player or converging on a specific position. Others, such as natural speech generation or pattern recognition, are very difficult to implement and are either closely guarded trade secrets or the province of advanced academic research.

## Update Sprite Positions

The positions of all game objects are updated in this segment. Their velocities are added to their current positions, which are then usually checked to make sure that they are still within the boundaries of the game world. This segment is where the results of user input and artificial intelligence logic are realized. A ship is accelerated or turned according to how the user moved the joystick, and enemy soldiers and cannons are advanced forward or animated per the decisions of the artificial intelligence segment.

Since all objects in the game world need to be updated, the game loop typically iterates through each object, one at a time, applying the position translations as required. The game loop itself needs to run as quickly as possible, so other actions that are performed for each game object are typically executed in this segment as well. The most common action performed at the same time as position updating is probably drawing the sprite image into an offscreen buffer. However, other actions may include collision detection, artificial intelligence processing, or other functions. The process of actually drawing the sprite, as well as other sprite specific procedures, will be examined in Chapter 6, Sprite Techniques.

## Check for Collisions

The collision detection segment is necessary for determining if two sprites or game objects have collided, and the action that is a result thereof. Most games are driven off of events where two sprites come into contact. For example, when a bullet that was fired several frames ago finally touches the enemy ship at which it was targeted, something should happen.

There are hundreds of different methods by which to detect if two or more objects have collided. Some games may require that every object be checked against every other object; others may be interested in the

collision between only certain types of objects. Whatever the case, collision detection algorithms vary in complexity according to the task they must perform and the number of objects involved. Accuracy is also an area of great variation. Some may detect only a close collision, and may not be very accurate at all. Others may go as far as comparing the pixels of each bitmap at a certain frame of animation and at a certain position within the game world to see if any pixels actually overlap (a highly accurate method indeed). Speed also becomes a very important issue with the collision detection scheme put into production, as the more accurate the collision detection, typically the slower it will run. Collision detection algorithms will be covered in the chapter on sprite techniques.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

### Start Sound Effects

Games would tend to get boring without sound effects to accompany the actions of various game objects. The responsibility of this segment is to commence the output of various sound effects as warranted by the actions of both the user and the game controlled objects. Technically speaking, sound effect playback will probably be handled during other segments of the game loop. For example, when the user presses a button to fire a missile the user input retrieval segment will probably call a function that both starts the missile object as well as commences the playback of an appropriate sound effect. Other sound effects, such as idle background chatter or other appropriate noises that would be generated from non-interactive game objects, will probably be started when the artificial intelligence section makes one of these objects perform some action. Any other sound effects not handled by an appropriate segment, such as ambient sounds or user interface activation sounds, may be generated from this segment. We will see techniques for sound effect output in Chapter 9, Sound and Music.

### Start Music

As with sound effects, music enhances the gaming experience, and this segment is responsible for starting musical output or changing the tune of the existing output. Again, music playback may be handled by another, more appropriate segment. However, music is commonly used to reflect the overall mood or state of the game. As such, this segment would be responsible for examining the current state as it would reflect from the user's standpoint, and start or change the music as appropriate. For example, a neutral musical interlude may be output when the user is just flying along to the next waypoint. This could be changed, on the fly, to a more intense, urgent musical number when the user goes into combat. It could be changed to an even more intense score if the player or enemy is heavily damaged, and finally an appropriate tune could be played based on the user's victory or defeat. This segment would need to perform the necessary actions to determine the current state of the game, playing a specific musical piece as appropriate. The chapter on sound and music contains examples of musical score playback techniques.

### Render the Next Frame

This segment is responsible for the most important part of the game loop, that of displaying the current state of the game from the user's perspective in graphical form. Depending on the complexity of the game, this

may simply entail drawing two-dimensional bitmap images to the screen, or rendering a complex, realistic view based on the positions of three-dimensional models. Of all the segments in a game loop, this segment has the greatest variety of implementations. Like all game programming topics, some techniques are well documented, such as simple sprite output or even three-dimensional raycasting or rendering techniques. Other techniques, such as true three-dimensional space rendering, are trade secrets and are poorly documented, if at all.

Typically, rendering a frame of animation starts by drawing the background. This may be as simple as a game board or something more complex, like a textured, scaled wall. Sprite images are then drawn over this background. All of this drawing generally takes place in an offscreen graphical buffer. After all drawing and rendering has been completed, this buffer is then blasted to the screen, at which point it becomes visible as the next frame of animation. As sprite positions are modified, their images drawn to reflect these new positions, and the results displayed to the screen in rapid succession, the illusion of animation is achieved. DirectX provides several methods for performing graphical output, both simplistic and complex.

## Check for Game End

At some point, the current state of the game needs to be compared with a known state to see if a state change is in order. For example, if all of the player's lives have been used up, the game is over. Additionally, when all of the enemy creatures have been eliminated or the exit is found, the current level has ended and it is time to advance to the next level of difficulty.

Advanced games may potentially contain several states that affect how the overall game loop is processed. For example, when a level is completed, the game may go into a temporary intermission state where, instead of performing the normal gameplay animation, it displays a summary of the previous level's accomplishments as well as the name or number of the next level being entered. A discussion of expanding the game loop to support different game states is covered below.

## Speed Delay

The one constant in the computer industry is that machines will only get faster and faster as time marches on. Therefore, a game that plays at an acceptable speed today may be too fast to be humanly playable on a new machine six months from now. To ensure that the game runs at a specific speed on every machine, this segment determines if enough time has passed since the rendering of the last frame to constrain the game to a specific speed.

One technique for controlling the speed of a game is to time the duration of the processing of the current animation frame. A starting time is recorded at the beginning of the game loop. At the end of the loop, this starting time is subtracted from the current time. If the result is less than a specific amount of time that will constrain the game speed to a certain rate, the execution is paused for the difference.

This results in a game that should run at a constant speed on any machine equal to or faster than the minimum recommended system for the game. The sprite techniques chapter and other chapters throughout this book contain implementations of timing algorithms for controlling game speed.



**Note:** To accommodate slower systems, some games perform timing tests to see how fast animation can be performed. If it is slower than a certain threshold, the game may compensate by dropping animation frames or turning off some features. The result will be a scaled-down, probably less attractive version of the game, but it has the advantage of running on older systems, which may be a selling point for the customer.

---

## Game States

The game loop example detailed above provides the application with a framework that makes game animation and user input control easy and intuitive for the developer. However, it is not as robust as it needs to be for real-world applications. How many games jump right into the action immediately upon launching

them? Most games display some sort of mission briefing or demonstration while waiting for the user to actually start a game. Typically, action games will also display some sort of information when a user has completed a level. Still others allow the user to input their name when a game has ended.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

- ▶ Search Tips
- ▶ Advanced Search

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

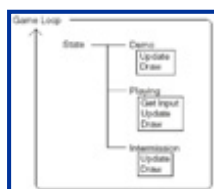
We have examined games in terms of the general tasks that are performed. Additionally, games can be examined from the standpoint of their current mode of execution. When a game is first launched, it is considered to be in one mode of execution, but when a new game is started and the user is actually playing, it is in a totally different mode of execution.

Our current game loop does not support this functionality. What is needed is a technique by which the game can be put into specific modes of execution, or states. For example, when the game is first launched, it should be initialized to a demo state. Other states, such as an intermission state or a game over state, will control what parts of the game loop are executed, and will display the appropriate output.

To extend our game loop, we will assume that it supports a game with five states:

- Demo
- Playing
- Intermission
- Paused
- Game Over

Each different state would involve executing code appropriate only for that state. Through each iteration of the game loop, the current state is checked, and an appropriate set of functions is executed that produces output (and retrieves input) commensurate with that state.



**Figure 2-4:** Controlling the game through states

This could be implemented by using a case statement. The game states could be represented as an enumerated type. Then, when the game loop is executed, the case statement would jump to the appropriate





```

                                {paused code}

                                end;

gsGameOver : begin

{game over code}

                                end;

                                end;

end;

```

This technique requires the case statement to be executed over and over. If this becomes a speed issue, this technique could be modified such that each case statement entry jumps to a function that contains its own internal loop. That internal loop would then continue until the game state changes. When this occurs, the internal loop exits, and the main game loop must be called again to switch execution into another function with its own internal loop. This modification is illustrated in Figure 2-5. Technically, an internal loop may simply display a static graphic and then pause for an appropriate period of time, such as for an intermission, instead of performing some sort of animation loop. However, static screens are usually pretty boring.



**Figure 2-5:** A modified game state looping architecture

Designing a game application in this manner allows it to be controlled in a very logical and ordered manner. The state of a game can flow from one state to another as appropriate, changing between logical states and ultimately creating a circuit or loop from game beginning through game play to game end and back again. For example, games may start in the demo state, switching into the playing state when a user starts a game. This state will move back and forth between the paused state and the intermission state as the user finishes levels and plays the game. Ultimately, the user will finish the game or will die, at which point the state will switch to the game over state, and finally back to the demo state, where the entire process can be restarted anew. Figure 2-6 illustrates the flow of game logic through a typical game loop constructed in this manner.



**Figure 2-6:** The flow of game states

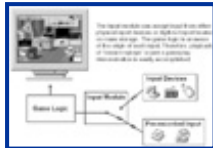
Each state's code is responsible for supplying specific output or retrieving specific input, as appropriate for the intention of that state. Some games may have more states than others; other games may not use this design at all, opting for another, more applicable control method. Even so, whether a game uses this technique or not, the execution of a game can be described in terms of states.

### The Demo State

This state typically provides some sort of animation while waiting for the user to start a new game. This may take the form of simple animations within user interface controls (such as animated images on buttons) or something more complex. This state could even be used to play the opening animation sequence introducing players to the game.

Some games provide a demonstration of game play as the background animation. This is certainly a popular technique with hot first-person shooters, and even with most action arcade-like games. Displaying a demonstration of actual game play can whet users' appetites for the game and help to familiarize them with the environment before actually diving in.

To appropriately show a demonstration of game play, this state must run some code elements of the state that is used for actual game play. Artificial intelligence routines could be used to simulate the input of a user and generate some interesting animation. Alternatively, input from an actual game session could be saved to an output file. If designed correctly, the functions that interpret user input could take their values from either a user input device or this digitized user input file. Thus, the demo state would simply open this file and begin streaming the data to the input routines, resulting in an exact duplication of the player's movements and actions.



**Figure 2-7:** Input coming from devices or files

This technique has many benefits. For one, the developer can control exactly what the user will see in the demonstration, which could avoid accidentally showing off hidden rooms or other surprises. Additionally, the developer could digitize walkthroughs that could then be used in separate game hint packages. Also, this method lends itself directly to instant replay functionality, highly desirable in sports and racing games.

[Previous](#) | [Table of Contents](#) | [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

### The Playing State

This state is responsible for the code that makes up the actual game play. All of the segments of the previous game loop presented above will be found here, such as user input processing, sprite updating, and screen drawing. The code run in this state will probably look similar to the example game loop implementation previously discussed.

### The Intermission State

When a user has completed a level (or a series of levels or a mission), games usually enter a type of intermission state. In this state, a report is given to the user that, depending on the type of game, details the user's accomplishments in the completed level. Such things as the number of kills, the number of secrets found or unnoticed, and the accuracy of shots are typically highlighted, as is the found or unnoticed, and the accuracy of shots are typically highlighted, as is the number of the next level or the name of the next mission. Sometimes, even an animation will be used to provide a transition from one level to the next, especially if a "chapter" of the game has been completed and an entirely new set of environments and enemies are about to be introduced. Intermission states only last a short while, setting the game state to playing once the intermission state is over.



**Tip:** It is a good practice to provide a hot key or some other mechanism whereby a user can skip the intermission state and proceed directly back to the playing state.

### The Paused State

The paused state is simply a suspension of game play. All game object logic and processing should be halted, including the output of sound effects and possibly even the output of music. Typically, pausing a game should simply toggle between the paused state and the playing (or intermission) state. Pausing a game should usually have no effect during a demo or game over state. Paused states are generally entered through either a hot key activation or by interacting with a user interface control (such as a button or menu item).

## The Game Over State

In the game over state, an expanded report of the user's accomplishments throughout the course of the game could be displayed. This includes the information displayed during the intermission state as it applies to the overall game achievements. End game animations could also be played, perhaps varying the exact animation dependent upon whether the player won or lost. The game over state usually only lasts for a short amount of time, perhaps only a little longer than the intermission state, before proceeding to the demo state.

## Case Study

In order to further demonstrate the concepts discussed in this chapter, we should examine the case study provided on the CD in this chapter's directory. It is a game titled Shoot 'em, where the player simply fires at oncoming enemy spaceships. It is a simple game using simple graphics, but it should serve as an example of implementing several of the topics we just covered. This is not a complete game, and many of the tasks required by a commercial quality game have been left out (such as sound and music output) so as not to complicate the code. The architecture used in this example game will be the basis for examples to come.

Of particular interest to this chapter is the game loop itself. The example implements a simplistic game loop similar to the first loop architecture we've discussed. It is state driven, and throughout the code you can see where this state is modified and how it affects the overall game processing. This game loop is embodied in the DrawSurfaces procedure. Take a look at this procedure to get a taste of how to implement an actual game loop.

This case study example is presented in its entirety to get you familiar with the types of examples demonstrated throughout the book. There are several programming techniques implemented in this example that we have yet to discuss, such as sprite manipulation. This example also contains some DirectX initialization code that we haven't yet discussed. Peruse the first part of the example to get familiar with the function calls that we'll be exploring in subsequent chapters. Beginning with the DrawSurfaces procedure, we see some techniques that we've outlined in this chapter. This example also demonstrates how to logically control such entities as sprites and background animation elements with arrays. Later in the book, we'll extend this to a more general approach for controlling game entities.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

7.0

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 2-3: The Shoot 'em example game*

```
unit ShootEmU;
```

```
{*****
```

```
    Shoot Example Application
```

```
Author: John Ayres
```

Like most examples in the book, this one takes advantage of the baseline DirectX application code. It implements a very simplistic game in order to demonstrate basic game architecture. The architecture of a game is arbitrary and will in part be determined by the type of game being created. However, most games share similar tasks in order to generate a frame of animation, and this example should serve to familiarize the user with some of the more common gaming application tasks.

```
*****
```

Copyright © 1999 by John Ayres, All Rights Reserved

This example utilizes sprites from the freeware sprite library  
SpriteLib by Ari Feldman.

Sprite lib sprites Copyright (c) 1998 by Ari Feldman. Check out  
<http://www.chromewave.com>

This code is freeware and can be used for any personal or commercial  
purposes without royalty or license. Use at your own risk.

```
*****}
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
DDraw;
```

```
{these constants are used to modify specific attributes of the DirectX  
application, such as the color of the main form, the requested resolution,  
etc.}
```

```
const
```

```
DXWIDTH      = 640;
```

```
DXHEIGHT     = 480;
```

```
DXCOLORDEPTH = 8;
```

```
BUFFERCOUNT = 1;
```

```
COOPERATIVELEVEL = DDSCD_FULLSCREEN or DDSCD_ALLOWREBOOT or  
                  DDSCD_ALLOWMODEX or DDSCD_EXCLUSIVE;
```

```
SURFACETYPE    = DDSCAPS_COMPLEX or DDSCAPS_FLIP or DDSCAPS_PRIMARYSURFACE;
```

```
const
```

```
{this user-defined message is used to start the flipping loop}
```

```
WM_DIRECTXACTIVATE = WM_USER + 200;
```

```
{the player's ship index}
```

```
PLAYERSHIPIDX = 0;
```

```
{the total number of ships}
```

```
NUMSHIPS = 5;
```

```
{the total number of bullets}
```

```
NUMBULLETS = 19;
```

```
{the total number of explosions}
```

```
NUMEXPLOSIONS = 5;
```

```
{the total number of stars}
```

```
NUMSTARS = 100;
```

```
type
```

```
{tracks the overall game state}
```

```
TGameState = (gsDemo, gsPlaying, gsIntermission);
```

```
{our base sprite class}
```

```
TSprite = class
```

```
public
```

```
    XPos, YPos,
```

```
    XVel, YVel,
```

```
    NumFrames,
```

```
    CurFrame,
```

```
    FrameWidth, FrameHeight,
```

```
    XFrameStartOffset, YFrameStartOffset: Integer;
```

```
    BoundBox,
```

```
    CollisionBox: TRect;
```

```
Living: Boolean;

CurAnimTime,

AnimThreshold: LongWord;

procedure Move; virtual;

procedure Draw; virtual;

end;

{our star class, descended from sprites}

TStar = class(TSprite)

public

    Color: Byte;

    procedure Move; override;

    procedure Draw; override;

end;

{the explosion class, also descended from sprites}

TExplosion = class(TSprite)

public

    procedure Move; override;

end;

{the form class}

TfrmDXAppMain = class(TForm)

    procedure FormCreate(Sender: TObject);

    procedure FormDestroy(Sender: TObject);

    procedure FormKeyDown(Sender: TObject; var Key: Word;

        Shift: TShiftState);

    procedure FormActivate(Sender: TObject);

private

    { Private declarations }
```



```
{flips back to the GDI surface to display the exception error message}
procedure ExceptionHandler(Sender: TObject; ExceptionObj: Exception);

{the main rendering loop}
procedure AppIdle(Sender: TObject; var Done: Boolean);

{intercepts certain messages to provide appropriate functionality}
procedure AppMessage(var Msg: TMsg; var Handled: Boolean);

{flips the DirectX surfaces}
procedure FlipSurfaces;

{restores any lost surfaces}
procedure RestoreSurfaces;

{draws the contents of surfaces}
procedure DrawSurfaces;
public
  {tracks the overall game state}
  GameState: TGameState;

  {tracks the level and score, number of ships left to destroy before
   advancing to the next level, and the number of lives the player has left}
  Level,
  Score,
  NumShipsToNextLevel,
  LivesLeft: Integer;

  {timing variable for pausing in the intermission mode}
  TimeAccum: Longint;

  {indicates if the player can shoot or not}
```

```
CanShoot: Boolean;
```

```
{tracks all of the living ships}
```

```
Ships: array[0..NUMSHIPS] of TSprite;
```

```
{tracks all of the living player bullets}
```

```
PlayerBullets: array[0..NUMBULLETS] of TSprite;
```

```
{tracks all of the living enemy bullets}
```

```
EnemyBullets: array[0..NUMBULLETS] of TSprite;
```

```
{tracks all of the living explosions}
```

```
Explosions: array[0..NUMEXPLOSIONS] of TExplosion;
```

```
{tracks all of the living stars}
```

```
Stars: array[0..NUMSTARS] of TStar;
```

```
{these are various procedures that initialize certain variables and  
generalize control of certain elements of the game}
```

```
procedure StartEnemy;
```

```
procedure StartExplosion(XPos, YPos: Integer);
```

```
procedure StartBullet(XPos, YPos, XVel, YVel: Integer; PlayerBullet: Boolean);
```

```
procedure ResetPlayer;
```

```
procedure ResetAll;
```

```
procedure PlayerShipKilled;
```

```
procedure NewGame;
```

```
end;
```

```
var
```

```
frmDXAppMain: TfrmDXAppMain;
```

```

{the main DirectDraw interface}

FDirectDraw: IDirectDraw4;

{the interfaces for the primary and back buffer surfaces and bitmap image}

FPrimarySurface,

FBackBuffer,

FGraphicsImage: IDirectDrawSurface4;

{the palette interface}

FPalette: IDirectDrawPalette;

implementation

uses

    DXTools, DDUtil, MMSystem, gensuppt;

{$R *.DFM}

{ *-->  BASELINE APPLICATION CODE <---* }

{ - the callback function used to ensure that the selected graphics mode -
  - is supported by DirectX - }

function EnumModesCallback(const EnumSurfaceDesc: TDDSurfaceDesc2;
                           Information: Pointer): HRESULT; stdcall;

begin
    {if the height, width, and color depth match those specified in the
     constants, then indicate that the desired graphics mode is supported}

    if (EnumSurfaceDesc.dwHeight = DXHEIGHT) and

        (EnumSurfaceDesc.dwWidth = DXWIDTH) and

```

```

        (EnumSurfaceDesc.ddpfPixelFormat.dwRGBBitCount = DXCOLORDEPTH) then

        Boolean(Information^) := TRUE;

Result := DDENUMRET_OK;

end;

{ -> Events Hooked to the Application <- }

{ - this event is called when an exception occurs, and simply flips back -
  - to the GDI surface so that the exception dialog box can be read - }

procedure TfrmDXAppMain.ExceptionHandler(Sender: TObject;
                                         ExceptionObj: Exception);
begin
    {disconnect the OnIdle event to shut off the rendering loop}
    Application.OnIdle := nil;

    {if the DirectDraw object was successfully created, flip to the GDI surface}
    if Assigned(FDirectDraw) then
        FDirectDraw.FlipToGDISurface;

    {display the exception message}
    MessageDlg(ExceptionObj.Message, mtError, [mbOK], 0);

    {reconnect the OnIdle event to reenter the rendering loop}
    Application.OnIdle := AppIdle;
end;

{ - handles certain messages that are required to make DirectX function -
  - properly within Delphi - }

procedure TfrmDXAppMain.AppMessage(var Msg: TMsg; var Handled: Boolean);
begin

```

```

case Msg.Message of

  WM_ACTIVATEAPP:

    {unhook the OnIdle event when the application is being deactivated.
     this will stop all rendering}

    if not Boolean(Msg.wParam) then

      Application.OnIdle := nil

    else

      {upon activating the application, send ourselves the user-defined
       message}

      PostMessage(Application.Handle, WM_DIRECTXACTIVATE, 0, 0);

  WM_DIRECTXACTIVATE:

    begin

      {upon activating, restore all surfaces (reloading their memory as
       necessary), hook up the OnIdle event, and redraw the contents of
       all surfaces}

      RestoreSurfaces;

      Application.OnIdle := AppIdle;

      DrawSurfaces;

    end;

  WM_SYSCOMMAND:

    begin

      {do not allow a screen saver to kick in}

      Handled := (Msg.wParam = SC_SCREENSAVE);

    end;

end;

end;

{ -> Form Events <- }

{ - initialize essential form properties - }

procedure TfrmDXAppMain.FormCreate(Sender: TObject);

var

```

```
iCount: Integer;

begin

    Randomize;

    {set up the application exception handler}
    Application.OnException := ExceptionHandler;

    {initialize form properties; note that the FormStyle property must be
    set to fsStayOnTop}
    BorderStyle := bsNone;
    BorderIcons := [];
    FormStyle := fsStayOnTop;
    Color := clBlack;
    Cursor := crNone;

    {initialize the game state to demo}
    GameState := gsDemo;

    {create all of the star objects}
    for iCount := 0 to NUMSTARS do
    begin
        {create the star}
        Stars[iCount] := TStar.Create;

        {set its position}
        Stars[iCount].XPos := Random(DXWIDTH);
        Stars[iCount].YPos := Random(DXHEIGHT);

        {set its velocity. stars are only going to move from the top of the screen
        to the bottom, so we're only concerned with the horizontal velocity}
        Stars[iCount].XVel := 0;
        Stars[iCount].YVel := Random(5)+1;
    end;
end;
```

```

        {set the color of the star based on its velocity}
    case Stars[iCount].YVel of
        1..2 : Stars[iCount].Color := 252;
        3..4 : Stars[iCount].Color := 251;
        5    : Stars[iCount].Color := 255;
    end;
end;

end;

end;

{ - provides essential cleanup functionality - }
procedure TfrmDXAppMain.FormDestroy(Sender: TObject);
var
    iCount: Integer;
begin
    {disengage our custom exception handler}
    Application.OnException := nil;

    {remember, we do not have to explicitly free the DirectDraw objects, as they
    will free themselves when they go out of context (such as when the
    application is closed)}

    for iCount := 0 to NUMSTARS do
        Stars[iCount].Free;
    end;

    { - this method initializes DirectX and creates all necessary objects - }
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
var
    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
    need a temporary interface}
    TempDirectDraw: IDirectDraw;

```

```

{structures required for various methods}

DDSurface: TDDSurfaceDesc2;

DDSCaps: TDDSCaps2;

{flag used to determine if the desired graphics mode is supported}

SupportedMode: Boolean;

begin

  {if DirectDraw has already been initialized, exit}

  if Assigned(FDirectDraw) then exit;

  {create a temporary DirectDraw object. this is used to create the
desired DirectDraw4 object}

  DXCheck( DirectDrawCreate(nil, TempDirectDraw, nil) );

  try

    {we can only get a DirectDraw4 interface through the QueryInterface
    method of the DirectDraw object}

    DXCheck( TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw) );

  finally

    {now that we have the DirectDraw4 object, the temporary DirectDraw
    object is no longer needed}

    TempDirectDraw := nil;

  end;

  {set the cooperative level to that defined in the constants}

  DXCheck( FDirectDraw.SetCooperativeLevel(Handle, COOPERATIVELEVEL) );

  {hook up the application message handler}

  Application.OnMessage := AppMessage;

  {call EnumDisplayModes and verify that the desired graphics mode is
  indeed supported}

```



```

FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

DDSurface.dwFlags := DDSD_HEIGHT or DDSD_WIDTH or DDSD_PIXELFORMAT;

DDSurface.dwHeight := DXHEIGHT;

DDSurface.dwWidth := DXWIDTH;

DDSurface.ddpfPixelFormat.dwSize := SizeOf(TDDPixelFormat_DX6);

DDSurface.ddpfPixelFormat.dwRGBBitCount := DXCOLORDEPTH;

SupportedMode := FALSE;

DXCheck( FDirectDraw.EnumDisplayModes(0, @DDSurface, @SupportedMode,
                                         EnumModesCallback) );

{if the desired graphics mode is not supported by the DirectX drivers,
 display an error message and shut down the application}

if not SupportedMode then
begin
    MessageBox(Handle, PChar(The installed DirectX drivers do not support a +
                               display mode of: +IntToStr(DXWIDTH)+' X +
                               IntToStr(DXHEIGHT)+' , +IntToStr(DXCOLORDEPTH)+
                               bit color'), Unsupported Display Mode Error',
               MB_ICONERROR or MB_OK);

    Close;

    Exit;
end;

{set the display resolution and color depth to that defined in the constants}
DXCheck( FDirectDraw.SetDisplayMode(DXWIDTH, DXHEIGHT, DXCOLORDEPTH, 0, 0) );

{initialize the DDSurface structure to indicate that we will be creating a
 complex flipping surface with one backbuffer}

FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

DDSurface.dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;

DDSurface.ddsCaps.dwCaps := SURFACETYPE;

```

```
DDSurface.dwBackBufferCount := BUFFERCOUNT;

{create the primary surface object}
DXCheck( FDirectDraw.CreateSurface(DDSurface, FPrimarySurface, nil) );

{indicate that we want to retrieve a pointer to the backbuffer (the surface
immediately behind the primary surface in the flipping chain) }
FillChar(DDSCaps, SizeOf(TDDSCaps2), 0);
DDSCaps.dwCaps := DDSCAPS_BACKBUFFER;

{retrieve the surface}
DXCheck( FPrimarySurface.GetAttachedSurface(DDSCaps, FBackBuffer) );

{load the palette from the bitmap to be used and attach it to the
primary surface}
FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+
SpriteImgs.bmp');
DXCheck( FPrimarySurface.SetPalette(FPalette) );

{load the bitmap image containing the background and animation frames}
FGraphicsImage := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+
SpriteImgs.bmp');

{indicate which color in the bitmap is transparent}
DXCheck( DDSSetColorKey(FGraphicsImage, $00000000) );

{initialize the surfaces with the images}
DrawSurfaces;

{post a message that will hook up the OnIdle event and start the main
rendering loop}
PostMessage(Handle, WM_ACTIVATEAPP, 1, 0);
```

```
end;
```

```
{ -> Form Methods <- }
```

```
{ - this method is called in order to flip the surfaces - }
```

```
procedure TfrmDXAppMain.FlipSurfaces;
```

```
var
```

```
    DXResult: HRESULT;
```

```
begin
```

```
    {perform the page flip. note that the DDFLIP_WAIT flag has been used,  
    indicating that the function will not return until the page flip has been  
    performed. this could be removed, allowing the application to perform other  
    processing until the page flip occurs. however, the application will need to  
    continuously call the Flip method to ensure that the page flip takes place}
```

```
    elseDXResult := FPrimarySurface.Flip(nil, DDFLIP_WAIT);
```

```
    {if the surfaces were lost, restore them. on any other error, raise an  
    exception}
```

```
    if DXResult = DDERR_SURFACELOST then
```

```
        RestoreSurfaces
```

```
    else if DXResult <> DD_OK then
```

```
        DXCheck(DXResult);
```

```
end;
```

```
{ - this method is called when the surface memory is lost -  
- and must be restored. surfaces in video memory that -  
- contain bitmaps must be reinitialized in this function - }
```

```
procedure TfrmDXAppMain.RestoreSurfaces;
```

```
begin
```

```
    {restore the primary surface, which in turn restores any implicit surfaces}
```

```
    FPrimarySurface._Restore;
```

```

    {restore the bitmap image's surface}

    FGraphicsImage._Restore;

    {reload the bitmap into the surface}

    DXCheck( DDReloadBitmap(FGraphicsImage, ExtractFilePath(ParamStr(0))+
        SpriteImgs.bmp') );

end;

{ - this method is continuously called by the application, and provides -
  - the main rendering loop.  this could be replaced by a custom      -
  - while..do loop                                                    - }

procedure TfrmDXAppMain.AppIdle(Sender: TObject; var Done: Boolean);
begin
    {indicates that the application should continuously call this method}

    Done := FALSE;

    {if DirectDraw has not been initialized, exit}

    if not Assigned(FDirectDraw) then Exit;

    {draw surface content and flip the surfaces}

    DrawSurfaces;

    FlipSurfaces;

end;

{ - this method is called when the contents of the surfaces need to be -
  - drawn. it will be continuously called by the AppIdle method, so any -
  - rendering or animation could be done within this method          - }

procedure TfrmDXAppMain.DrawSurfaces;
var
    SourceRect,
    ISectRect: TRect;

```

```

iCount, iCount2: Integer;

SurfaceDC: HDC;

WorkCanvas: TCanvas;

begin
    {erase the last frame of animation}

    ColorFill(FBackBuffer, $00000000, nil);

    {draw all of the background stars. the stars are a background animation
     that are unaffected by any game state, so we always draw them}

    for iCount := 0 to NUMSTARS do
        begin
            Stars[iCount].Move;

            Stars[iCount].Draw;
        end;

    {perform various actions based on the current game state}

    case GameState of

        gsDemo : begin

            {in the demo state, we simply want to draw some text on the screen}

            WorkCanvas := TCanvas.Create;

            FBackBuffer.GetDC(SurfaceDC);

            {draw the game's title and instructions}

            try

                WorkCanvas.Handle := SurfaceDC;

                WorkCanvas.Font.Size := 52;

                WorkCanvas.Font.Name := 'Arial';

                WorkCanvas.Font.Style := [fsBold, fsItalic];

                WorkCanvas.Font.Color := clRed;

                WorkCanvas.Brush.Style := bsClear;

                WorkCanvas.TextOut(DXWIDTH div 2 -

                    (WorkCanvas.TextWidth(Shoot 'em') div 2),

```

```

        DXHEIGHT div 2 -
        (WorkCanvas.TextHeight(Shoot 'em') div 2),
        Shoot 'em');

WorkCanvas.Font.Size := 14;

WorkCanvas.Font.Name := Arial';

WorkCanvas.Font.Style := [fsBold];

WorkCanvas.Font.Color := clWhite;

WorkCanvas.Brush.Style := bsClear;

WorkCanvas.TextOut(DXWIDTH div 2 - (WorkCanvas.
        TextWidth(Press Enter to begin') div 2),
        DXHEIGHT - (WorkCanvas.
        TextHeight(Press Enter to begin')) - 15,
        Press Enter to begin');

finally

    WorkCanvas.Handle := 0;

    FBackBuffer.ReleaseDC(SurfaceDC);

    WorkCanvas.Free;

end;

{when the user presses the Enter key, initialize a new
game, and change the game state}

if (GetAsyncKeyState(VK_RETURN) and $8000) = $8000 then
    NewGame;

end;

gsIntermission : begin

    {in the intermission state, we want to draw the current
    level on the screen for only a few seconds before
    changing the game state}

    {record the current time}

    if TimeAccum = -1 then

```

```

    TimeAccum := timeGetTime;

    {draw the current level on the screen}
    WorkCanvas := TCanvas.Create;
    FBackBuffer.GetDC(SurfaceDC);

    try
        WorkCanvas.Handle := SurfaceDC;

        WorkCanvas.Font.Size := 52;
        WorkCanvas.Font.Name := Arial';
        WorkCanvas.Font.Style := [fsBold, fsItalic];
        WorkCanvas.Font.Color := clRed;
        WorkCanvas.Brush.Style := bsClear;
        WorkCanvas.TextOut(DXWIDTH div 2 - (WorkCanvas.
            TextWidth(Level +IntToStr(Level)) div 2),
            DXHEIGHT div 2 - (WorkCanvas.
            TextHeight(Level +IntToStr(Level)) div 2),
            Level +IntToStr(Level));
    finally
        WorkCanvas.Handle := 0;
        FBackBuffer.ReleaseDC(SurfaceDC);
        WorkCanvas.Free;
    end;

    {if we have been displaying this for more than 3 seconds,
    reset certain variables and change the game state}
    if timeGetTime-TimeAccum>3000 then
    begin
        ResetPlayer;
        TimeAccum := -1;
        GameState := gsPlaying;
    end;

```

```

        end;

gsPlaying : begin
    {in the playing state, the user is actively playing the
    game. this section will contain the game logic for
    moving the player and performing enemy AI}
{retrieve user input}

    if Ships[PLAYERSHIPIDX] <> nil then
        begin
            {move left or right when cursor keys are pressed}

            if (GetAsyncKeyState(VK_RIGHT) and $8000) = $8000 then
                Ships[PLAYERSHIPIDX].XPos := Ships[PLAYERSHIPIDX].XPos+4;
            if (GetAsyncKeyState(VK_LEFT) and $8000) = $8000 then
                Ships[PLAYERSHIPIDX].XPos := Ships[PLAYERSHIPIDX].XPos-4;

            {make sure the player does not go beyond the
            screen boundaries}

            if Ships[PLAYERSHIPIDX].XPos < 0 then
                Ships[PLAYERSHIPIDX].XPos := 0;
            if Ships[PLAYERSHIPIDX].XPos>DXWIDTH-Ships[PLAYERSHIPIDX].
                FrameWidth then
                Ships[PLAYERSHIPIDX].XPos:=DXWIDTH-Ships[PLAYERSHIPIDX].
                FrameWidth;

            {fire a shot when the spacebar is pressed. the CanShoot
            variable is used here to force the player to release the
            spacebar before another shot can be fired}

            if ((GetAsyncKeyState(32) and $8000) = $8000) and
                (CanShoot) then
                begin
                    StartBullet(Ships[PLAYERSHIPIDX].XPos+
                                (Ships[PLAYERSHIPIDX].FrameWidth div 2)-5,
                                Ships[PLAYERSHIPIDX].YPos, 0, -8, TRUE);

                    CanShoot := FALSE;
                end
            end
        end
    end
end

```



```

end

else

if (GetAsyncKeyState(32) and $8000) = 0 then

    CanShoot := TRUE;

end;

{some enemy AI. this is incredibly simplistic, but it
simply creates an enemy ship at a random interval. the
frequency of enemy ship generation increases as the
level increases, making the game harder}

if Random(1000) < 5 * Level then

    StartEnemy;

{move all of the bullets}

for iCount := 0 to NUMBULLETS do

begin

    {move player bullets}

    if PlayerBullets[iCount] <> nil then

begin

    {update the bullet based on velocity}

    PlayerBullets[iCount].Move;

    {collision detection. we must determine if this
bullet has collided with an enemy ship}

    for iCount2 := PLAYERSHIPIDX+1 to NUMSHIPS do

        if (Ships[iCount2] <> nil) and

            (IntersectRect(ISectRect,

                PlayerBullets[iCount].CollisionBox,

                Ships[iCount2].CollisionBox)) then

begin

            {indeed, the bullet did strike an enemy ship.

we should create an explosion, increment the

```

```
player's score, and destroy both the bullet and  
the enemy ship}
```

```
StartExplosion(Ships[iCount2].XPos,  
              Ships[iCount2].YPos);
```

```
Inc(Score, Level * 5);
```

```
PlayerBullets[iCount].Living := FALSE;
```

```
Ships[iCount2].Living := FALSE;
```

```
{the level increases when the player has destroyed  
a certain number of enemy ships, so we must  
record this kill}
```

```
Dec(NumShipsToNextLevel);
```

```
end;
```

```
{if the bullet is still alive, draw it; otherwise  
destroy it}
```

```
if PlayerBullets[iCount].Living then
```

```
    PlayerBullets[iCount].Draw
```

```
else
```

```
begin
```

```
    PlayerBullets[iCount].Free;
```

```
    PlayerBullets[iCount] := nil;
```

```
end;
```

```
end;
```

```
{now, we move enemy bullets}
```

```
if EnemyBullets[iCount] <> nil then
```

```
begin
```

```
    {update the bullet based on velocity}
```

```
    EnemyBullets[iCount].Move;
```

```
{as with the player bullets, we must determine if an  
enemy bullet has collided with the player's ship}
```

```

if (Ships[PLAYERSHIPIDX] <> nil) and
    (IntersectRect(ISectRect,
        EnemyBullets[iCount].CollisionBox,
        Ships[PLAYERSHIPIDX].CollisionBox))then
begin
    {indeed, the enemy bullet has collided with the
    player's ship, destroying it. we should create an
    explosion and destroy both the enemy bullet and the
    player's ship}
StartExplosion(Ships[PLAYERSHIPIDX].XPos,
                Ships[PLAYERSHIPIDX].YPos);
    EnemyBullets[iCount].Living := FALSE;
    Ships[PLAYERSHIPIDX].Living := FALSE;

    {we want the player to see the explosion, so we must
    wait an appropriate amount of time before changing
    the game state}
    TimeAccum := timeGetTime;
end;

    {if the enemy bullet is still alive, draw it; otherwise
    destroy it}
if EnemyBullets[iCount].Living then
    EnemyBullets[iCount].Draw
else
begin
    EnemyBullets[iCount].Free;
    EnemyBullets[iCount] := nil;
end;
end;

```

{game state AI. again, this is very simplistic, but if the

```
user has destroyed a certain amount of enemy ships
(based on the level), the level increases. as the level
increases, the number of enemy ships that the user is
required to destroy also increases. this results in the
game getting harder and harder as the player progresses}
if NumShipsToNextLevel <= 0 then
begin
    Inc(Level);
    NumShipsToNextLevel := 5 * Level;
    ResetAll;

    {since we've increased the game level, we should go to
    intermission to inform the player of what has happened}
    GameState := gsIntermission;
    Break;
end;
end;

{update and draw any explosions}
for icount := 0 to NUMEXPLOSIONS do
    if Explosions[iCount] <> nil then
        begin
            {although explosions don't really move, they
            animate, and in the interest of similarity,
            they have a Move method like other sprites}
            Explosions[iCount].Move;

            {explosions are short lived. if the explosion is
            still living (and thus animating), draw it; otherwise
            destroy it}
            if Explosions[iCount].Living then
                Explosions[iCount].Draw
            else
```



```

{as before, we need to determine if the player has
increased to the next level. however, the player
has just died, so we don't want to immediately
change the game state}

if NumShipsToNextLevel <= 0 then

begin

    Inc(Level);

    NumShipsToNextLevel := 5 * Level;

end;

{we want the player to see the explosion, so we must
wait an appropriate amount of time before changing
the game state}

TimeAccum := timeGetTime;

end

else

begin

{there was no collision with the player's ship,
so perform some very simplistic AI. this will
randomly change the horizontal velocity of the
ship, and randomly fire a bullet at the player}

if Random(1000) < 10 then

    Ships[iCount].XVel := 0-Ships[iCount].XVel;

if Random(1000) < 50 then

    StartBullet(Ships[iCount].XPos+
                (Ships[iCount].FrameWidth div 2)-5,
                Ships[iCount].YPos, 0, 8, FALSE);

end;

end;

{update the ship's position based on velocity}

Ships[iCount].Move;

```

```

    {if the ship is still living, draw it, otherwise
      destroy it}

    if Ships[iCount].Living then
        Ships[iCount].Draw
    else
        begin
            Ships[iCount].Free;
            Ships[iCount] := nil;
        end;
    end;

end;

{display the number of ships left}
for iCount := 0 to LivesLeft-1 do
begin
    SourceRect := Rect(392, 48, 422, 70);

    FBackBuffer.BltFast(500+(iCount*30)+5, 455,
                        FGraphicsImage, SourceRect,
                        DDBLTFAST_SRCOLORKEY or
                        DDBLTFAST_WAIT);

end;

{display the score}
WorkCanvas := TCanvas.Create;
FBackBuffer.GetDC(SurfaceDC);

try
    WorkCanvas.Handle := SurfaceDC;

    WorkCanvas.Font.Size := 10;
    WorkCanvas.Font.Name := 'Arial';
    WorkCanvas.Font.Color := clWhite;

```

```

        WorkCanvas.Brush.Style := bsClear;

        WorkCanvas.TextOut(5, 465, Score: +IntToStr(Score));

    finally

        WorkCanvas.Handle := 0;

        FBackBuffer.ReleaseDC(SurfaceDC);

        WorkCanvas.Free;

    end;

    {if our timing variable has been initialized, the player
    has been killed. thus, we check to see if an appropriate
    amount of time has passed before resetting certain
    variables and reinitializing the player. this allows
    players to see the ship exploding when it collides with
    a missile or an enemy ship}

    if (TimeAccum > -1) and (timeGetTime-TimeAccum>2000) then

    begin

        PlayerShipKilled;

    end;

    end;

end;

end;

{ -> Deletable Events <- }

///

{ - as a matter of convenience this framework will terminate when the -
- Escape key is pressed, but this should probably be deleted and -
- replaced with your own terminate methods - }

procedure TfrmDXAppMain.FormKeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);

begin

```



```

    if Key = VK_ESCAPE then
        Close;
    end;
    ///

{ *-->>  END BASELINE APPLICATION CODE <<--* }

{ TSprite }

procedure TSprite.Draw;
var
    SourceRect: TRect;

    DestY: Integer;
begin
    {if it is time to increment the current frame...}
    if timeGetTime - CurAnimTime > AnimThreshold then
        begin
            {increment the frame of animation}
            Inc(CurFrame);

            {roll it over if necessary}
            if CurFrame >= NumFrames then
                CurFrame := 0;

            {record the time}
            CurAnimTime := timeGetTime;
        end;

    {draw the next animation frame into the backbuffer}

```

```

SourceRect := Rect(XFrameStartOffset+(FrameWidth*CurFrame),
                  YFrameStartOffset, XFrameStartOffset+
                  (FrameWidth*CurFrame)+FrameWidth,
                  YFrameStartOffset+FrameHeight);

{if the sprite is fully on screen...}
if (YPos >= 0) and (YPos <= DXHEIGHT-FrameHeight) then
    {draw it to the screen}
    FBackBuffer.BltFast(XPos, YPos, FGraphicsImage, SourceRect,
                        DBLTFAST_SRCOLORKEY OR DBLTFAST_WAIT)
else
begin
    {...otherwise, it is partially hidden at the top or bottom, so determine
     the rectangular portion of the sprite that is visible}
    if YPos < 0 then
begin
    DestY := 0;
    SourceRect := Rect(SourceRect.Left, SourceRect.Top-YPos,
                      SourceRect.Right, SourceRect.Bottom);
end
else
begin
    DestY := YPos;
    SourceRect := Rect(SourceRect.Left, SourceRect.Top, SourceRect.Right,
                      SourceRect.Top+(DXHEIGHT-YPos));
end;

    {copy the image portion to the screen}
    FBackBuffer.BltFast(XPos, DestY, FGraphicsImage, SourceRect,
                        DBLTFAST_SRCOLORKEY OR DBLTFAST_WAIT);

end;

end;

```

```

procedure TSprite.Move;

begin
    {move the sprite according to its horizontal velocity}

    XPos := XPos + XVel;

    YPos := YPos + YVel;

    {clip the sprite to the boundaries of the screen}

    if XPos > DXWIDTH-FrameWidth then

        begin

            XPos := DXWIDTH-FrameWidth;

            XVel := 0-XVel;

        end;

    if XPos < 0 then

        begin

            XPos := 0;

            XVel := 0-XVel;

        end;

    {update its bounding box}

    CollisionBox := Rect(XPos+BoundingBox.Left, YPos+BoundingBox.Top,
                        XPos+BoundingBox.Right, YPos+BoundingBox.Bottom);

    {if the sprite has gone off the top or bottom of the screen, it is dead}

    if (YPos < 0-FrameHeight) or (YPos > DXHEIGHT) then

        Living := FALSE;

    end;

    { TExplosion }

procedure TExplosion.Move;

begin

    {explosions don't actually move, but an explosion's lifespan is over when

```

```

    it has played all of its frames}

    if CurFrame+1 >= NumFrames then

        Living := FALSE;

    end;

{ TStar }

procedure TStar.Move;

begin

    {stars only move vertically, so update their position}

    YPos := YPos + YVel;

    {clip them to the boundaries of the screen, wrapping around if necessary}

    if YPos < 0 then

        YPos := DXHEIGHT;

    if YPos >= DXHEIGHT then

        YPos := 0;

    end;

procedure TStar.Draw;

var

    SurfaceDesc: TDDSurfaceDesc2;

begin

    {lock the backbuffer}

    SurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc2);

    FBackBuffer.Lock(nil, SurfaceDesc, DDLOCK_SURFACEMEMORYPTR or DDLOCK_WAIT,0);

    {color the appropriate pixel in the backbuffer using the star's color. this
    is a very fast way to plot individual pixels}

    try

        Byte(Pointer(Longword(SurfaceDesc.lpSurface)+

            (YPos*SurfaceDesc.lPitch+XPos))^ ) := Color;

    finally

```

```

    {the buffer must be unlocked before it can be flipped}

    FBackBuffer.Unlock(nil);

end;

end;

procedure TfrmDXAppMain.NewGame;

begin
    {initialize game variables}

    Level := 1;

    Score := 0;

    LivesLeft := 3;

    NumShipsToNextLevel := 5;

    {reset all sprites}

    ResetAll;

    {go to intermission}

    GameState := gsIntermission;

end;

procedure TfrmDXAppMain.StartBullet(XPos, YPos, XVel, YVel: Integer;
    PlayerBullet: Boolean);

var
    iCount: Integer;

begin
    {if we're starting a player bullet}

    if PlayerBullet then

    begin
        {find an open slot in the bullets array}

        for iCount := 0 to NUMBULLETS do

            if PlayerBullets[iCount] = nil then

                begin

```

```

        {create a new bullet}

        PlayerBullets[iCount] := TSprite.Create;

        {initialize the bullet sprite's properties}

        PlayerBullets[iCount].NumFrames := 1;

        PlayerBullets[iCount].CurFrame := 0;

        PlayerBullets[iCount].FrameWidth := 10;

        PlayerBullets[iCount].FrameHeight := 17;

        PlayerBullets[iCount].XFrameStartOffset := 372;

        PlayerBullets[iCount].YFrameStartOffset := 48;

        PlayerBullets[iCount].BoundingBox := Rect(0, 0, 10, 17);

        PlayerBullets[iCount].Living := TRUE;

        PlayerBullets[iCount].XPos := XPos;

        PlayerBullets[iCount].YPos := YPos;

        PlayerBullets[iCount].XVel := XVel;

        PlayerBullets[iCount].YVel := YVel;

        PlayerBullets[iCount].CurAnimTime := timeGetTime;

        PlayerBullets[iCount].AnimThreshold := 10000;

        {this bullet has been created, we can exit}

        Break;

    end;

end

else

begin

    {otherwise, search for an open slot in the enemy bullets array}

    for iCount := 0 to NUMBULLETS do

        if EnemyBullets[iCount] = nil then

            begin

                {create the bullet}

                EnemyBullets[iCount] := TSprite.Create;

                {initialize the bullet sprite's properties}

```

```

    EnemyBullets[iCount].NumFrames := 1;

    EnemyBullets[iCount].CurFrame := 0;

    EnemyBullets[iCount].FrameWidth := 10;

    EnemyBullets[iCount].FrameHeight := 10;

    EnemyBullets[iCount].XFrameStartOffset := 382;

    EnemyBullets[iCount].YFrameStartOffset := 51;

    EnemyBullets[iCount].BoundingBox := Rect(0, 0, 10, 10);

    EnemyBullets[iCount].Living := TRUE;

    EnemyBullets[iCount].XPos := XPos;

    EnemyBullets[iCount].YPos := YPos;

    EnemyBullets[iCount].XVel := XVel;

    EnemyBullets[iCount].YVel := YVel;

    EnemyBullets[iCount].CurAnimTime := timeGetTime;

    EnemyBullets[iCount].AnimThreshold := 10000;

    {this bullet has been created, we can exit}

    Break;

end;

end;

end;

procedure TfrmDXAppMain.StartEnemy;
var
    iCount: Integer;
begin
    {search for an open slot in the ships array. notice that the player's ship
    is always located at the first index, so we skip it}
    for iCount := PLAYERSHIPIDX+1 to NUMSHIPS do
        if Ships[iCount] = nil then
            begin
                {create the sprite}

                Ships[iCount] := TSprite.Create;
            end;
        end;
    end;
end;

```

```

        {initialize the sprite properties}

        Ships[iCount].NumFrames := 3;

        Ships[iCount].CurFrame := 0;

        Ships[iCount].FrameWidth := 62;

        Ships[iCount].FrameHeight := 36;

        Ships[iCount].XFrameStartOffset := 186;

        Ships[iCount].YFrameStartOffset := 0;

        Ships[iCount].BoundingBox := Rect(3, 3, 59, 33);

        Ships[iCount].Living := TRUE;

        Ships[iCount].XPos := Random(DXWIDTH - 61)+1;

        Ships[iCount].YPos := -36;

        Ships[iCount].XVel := Random(6)-3;

        Ships[iCount].YVel := Random(4)+1;

        Ships[iCount].CurAnimTime := timeGetTime;

        Ships[iCount].AnimThreshold := 100;

        {the sprite was created, we can exit}

        Break;

    end;

end;

procedure TfrmDXAppMain.StartExplosion(XPos, YPos: Integer);
var
    iCount: Integer;
begin
    {search for an open slot in the explosions array}
    for iCount := 0 to NUMEXPLOSIONS do
        if Explosions[iCount] = nil then
            begin
                {create the explosion sprite}

                Explosions[iCount] := TExplosion.Create;
            end;
        end;
    end;
end;

```



```

        {initialize the sprite properties}

        Explosions[iCount].NumFrames := 6;

        Explosions[iCount].CurFrame := 0;

        Explosions[iCount].FrameWidth := 62;

        Explosions[iCount].FrameHeight := 47;

        Explosions[iCount].XFrameStartOffset := 0;

        Explosions[iCount].YFrameStartOffset := 48;

        Explosions[iCount].Living := TRUE;

        Explosions[iCount].XPos := XPos;

        Explosions[iCount].YPos := YPos;

        Explosions[iCount].CurAnimTime := timeGetTime;

        Explosions[iCount].AnimThreshold := 100;

    {the sprite was created, we can exit}

        Break;

    end;

end;

procedure TfrmDXAppMain.ResetAll;

var

    iCount: Integer;

begin

    {free all of the ships}

    for iCount := 0 to NUMSHIPS do

        begin

            Ships[iCount].Free;

            Ships[iCount] := nil;

        end;

    {free all of the bullets}

    for iCount := 0 to NUMBULLETS do

        begin

            PlayerBullets[iCount].Free;

```

```

    PlayerBullets[iCount] := nil;

    EnemyBullets[iCount].Free;

    EnemyBullets[iCount] := nil;

end;

{free all of the explosions}

for iCount := 0 to NUMEXPLOSIONS do
begin
    Explosions[iCount].Free;

    Explosions[iCount] := nil;

end;

{reset the pause timer}

TimeAccum := -1;

end;

procedure TfrmDXAppMain.ResetPlayer;

begin
    {make sure the player ship is created}

    if not Assigned(Ships[PLAYERSHIPIDX]) then
        Ships[PLAYERSHIPIDX] := TSprite.Create;

    {initialize the player ship sprite properties}

    Ships[PLAYERSHIPIDX].NumFrames := 3;

    Ships[PLAYERSHIPIDX].CurFrame := 0;

    Ships[PLAYERSHIPIDX].FrameWidth := 62;

    Ships[PLAYERSHIPIDX].FrameHeight := 48;

    Ships[PLAYERSHIPIDX].XFrameStartOffset := 0;

    Ships[PLAYERSHIPIDX].YFrameStartOffset := 0;

    Ships[PLAYERSHIPIDX].BoundingBox := Rect(3, 3, 59, 45);

    Ships[PLAYERSHIPIDX].Living := TRUE;

    Ships[PLAYERSHIPIDX].XPos := DXWIDTH div 2;

```

```

Ships[PLAYERSHIPIDX].YPos := DXHEIGHT-Ships[PLAYERSHIPIDX].FrameHeight-20;

Ships[PLAYERSHIPIDX].XVel := 0;

Ships[PLAYERSHIPIDX].YVel := 0;

Ships[PLAYERSHIPIDX].CurAnimTime := timeGetTime;

Ships[PLAYERSHIPIDX].AnimThreshold := 100;

{indicate that the player can now shoot}

CanShoot := TRUE;

end;

procedure TfrmDXAppMain.PlayerShipKilled;

begin
    {decrement the number of lives}

    Dec(LivesLeft);

    {reset all of the sprites}

    ResetAll;

    {if the player has no more lives left, go to demo; otherwise go to
    intermission}

    if LivesLeft < 0 then
        GameState := gsDemo
    else
        GameState := gsIntermission;
    end;

end.

```

---

## Summary

In this chapter, we discussed the most important and most basic issues involved in designing game application software. When designing the architecture of game application software, it is important to keep these points in mind:

- Every game, no matter how complex, can be broken down into three constituent parts: graphics, sound, and user

input.

- The specific architecture of a game will vary widely depending on the type of game and its inherent functionality. Even so, most games usually perform similar tasks in a similar order. From a high-level standpoint, game software can be thought of in terms of six elements that perform necessary tasks: Initialization, Game Introduction, Game Startup, the Game Loop, Game Ending, and Shutdown and Exit.
- The game loop is usually the focal point of a game application, and is where all of the action takes place. It processes a series of instructions over and over as quickly as possible to produce the on-screen images that reflect the current state of the game. This processing continues with or without user interaction. It can be implemented in a variety of ways, and must perform several different tasks as rapidly as possible. Technically speaking, a game loop is responsible for three things: retrieving input, updating game objects, and rendering a new frame of animation. Additionally, it must perform collision detection, enemy AI, and other tasks commensurate with the specific type of game.
- Games can be examined from the standpoint of their current mode of execution, or state. A state based gaming architecture allows the game to be controlled in a more logical and ordered manner. The state of a game can flow from one state to another as appropriate, changing between logical states and ultimately creating a circuit from game beginning through game play to game end and back again. In general, games can be considered to execute in five different states: Demo, Playing, Intermission, Paused, and Game Over.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 3 Basic Graphics Programming

This chapter covers the following topics:

- Graphical elements, such as pixels, lines, and polygons
- Video mode resolutions and color depth, and the difference between palettized and non-palettized video modes
- Device contexts and their use
- Using the TCanvas object for graphical output
- Essential bitmap techniques
- Color palettes

Like other industries, the computer industry has matured to the point that it can be thought of in terms of specialties. There are specialists in hardware installation, technical support, client/server architecture, etc. Some fields, such as programming, are further divided. Programming offers many areas of specialization, including database programming, speech recognition, or even communications. Of these, graphics programming is one of the most fun, rewarding, and sought-after specialties. All client-side applications that interact directly with the user make some use of graphics programming techniques, be they simple GDI commands for drawing basic Windows interface elements or full-fledged, 3-D rendered landscapes in games. At some point, any programmer working on an application to be used by end users will need to have some knowledge of graphical programming techniques.

This book's primary focus is on high-performance graphics and games programming using DirectX. Although the techniques discussed throughout the book are advanced by their very nature, the neophyte to graphics programming should not be ignored. Many people have used Delphi solely for client/server database applications or business related programs, and are just now experimenting with its broader range of functionality. There have been several shareware games written in Delphi, and retail games created with Delphi, while rare, are starting to emerge. Thus, many readers of this book are probably new to graphics programming, and may be a little fearful of the advanced topics presented in the coming chapters.

To this end, this chapter has been dedicated to acclimating the Delphi programmer to terms and concepts

common in graphics programming. Although no one concept is discussed in any great depth, after reading this chapter, the casual Delphi programmer will be familiar with many of the terms and concepts that will be expounded upon in later chapters. This should give Delphi programmers a good base of knowledge to prepare them for the graphics programming techniques to come. Readers who are familiar with basic graphics programming topics, such as pixels, lines, polygons, bitmaps, and using the TCanvas object, may want to skip ahead to the next chapter, which introduces DirectX. However, if you're not familiar with these topics, read on.

## The Graphical Device Interface

One of the fundamental pieces of Windows is the Graphical Device Interface, or the GDI. This important subsystem is responsible for all of the graphical output displayed to the screen during a Windows session. The GDI subsystem is accessed through a wide variety of functions and procedures. Buttons, lines, scroll bars, application windows themselves, even text, are all drawn by these GDI functions and procedures.



**Graphical Device Interface (GDI):** The Windows subsystem responsible for all graphical output during a Windows session.

One of Windows's strongest attributes is its device independence, or its ability to be run on a wide variety of hardware combinations. Due to this requirement, the GDI's primary purpose is to provide a mechanism by which an application can output graphical content without the programmer being concerned about what type of hardware upon which the application will be running. It accomplishes this goal superbly; it is incredibly easy to draw lines, buttons, bitmaps, etc., without being concerned with hardware.

Unfortunately, as a direct consequence of this device independence, the GDI is notorious for its slow performance. Even with this reputation, the GDI is very useful when creating tools for games. While most games today are written in DirectX, it is very common for the game programmers to write tools under regular Windows using the GDI. Strictly speaking, while games run faster under DirectX, it's usually quicker to write the tools used to build the game using good ol' Windows and the GDI. The tools themselves rarely need the performance achievable with DirectX, and the incredible richness of the GDI API makes it simple and quick to produce the tools. It is now common for games to include some of the tools used in its creation, such as level editors. However, today's incredibly fast processors and Windows accelerated video cards are making this speed issue less of a problem. Although many would argue that a commercial quality game could not be accomplished using standard GDI techniques, several commercial and shareware games have been successfully created using nothing more than an intelligent application of GDI programming techniques.



**Figure 3-1:** Games using GDI techniques

*Earthworm Jim for Windows 95 by Activision*



*Pitfall: The Mayan Adventure by Activision*



*Mission: Moon by Beckett Interactive*

Later in the chapter, we'll explore how Delphi and the TCanvas class unlock the power of the GDI and make it incredibly easy for the application developer to take advantage of some very powerful graphics programming techniques. First, however, we need to cover a number of concepts that are critical to understanding graphics programming.

## Graphical Elements

To begin exploring the topic of graphics programming, one must be familiar with a number of terms that describe various graphical elements. Like any specialty, graphics programming has its share of jargon, much of which will be discovered throughout the pages of this book. However, there are several terms that describe concepts in just about every application of graphics programming. These are: the pixel, the line, polygons, and bitmaps.

### Pixels

When the GDI displays an image to the screen, whether it's an application window, a button, text, etc., it draws the graphic in a series of different colored dots. As these dots are drawn to the screen, they eventually form an image that is recognizable to the user. This dot is called a pixel, and it is the smallest graphic that can be drawn to the screen. This action of "drawing" individual colored dots to the computer screen happens so fast (usually) that it appears instantaneous to the viewer.



---

**pixel:** The smallest, single element in a graphical image or on the display surface.

The entire computer screen can be thought of as a grid of dots, or pixels. The size of this grid determines the video mode resolution, and the number of colors in which an individual pixel can be drawn determines the color depth (more on this later).



**Figure 3-2:** The screen is composed of dots, or pixels

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ Search Tips
- ▶ Advanced Search

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Lines

Next to pixels, lines are the most common graphical element seen in user interfaces. The GDI uses lines of different colors to simulate the beveled surfaces common in Windows user interfaces. Of course, lines have several thousand other practical uses, from representing scales and rulers to graphs and other more complex graphical content. Strictly speaking, a line is simply a collection of pixels arranged in a linear fashion.



**line:** A collection of pixels, usually arranged in a linear or connecting configuration.

Lines have a mathematical discipline all their own, known as geometry. The study of lines and the mathematics that manipulate them is almost a fundamental requirement for any serious graphics programming. Even if the application will not be strictly output lines, it is sometimes useful to utilize geometry (and trigonometry, see below) to produce incredibly realistic scene generation. We'll see more of this when we cover ray casting engines later in the book.



**Figure 3-3:** A line is a collection of pixels

## Polygons

Polygons are quite prevalent in the world of 2-D graphics programming. Buttons are usually square or rectangular in shape, and it's not uncommon to see controls or graphics that are triangular, pentagonal, or configured into some other shape. Simply put, a polygon is a collection of lines, connected at the end to form some sort of closed shape.





**polygon:** A collection of lines, arranged in such a way that the ends are connected to form a continuous, closed shape.

Circles, ovals, and curves can be lumped into this category. The concept of polygons is absolutely critical to 3-D programming. Like lines, polygons are governed by a mathematical discipline known as trigonometry (actually, trigonometry and geometry are almost joined at the hip, and I use the relationship to lines and polygons loosely). Trigonometry is the study of angles, and this coupled with polygonal shapes allows the creation of incredibly realistic 3-D environments. We'll see some examples of this later in the book.



**Figure 3-4:** A polygon is a collection of lines

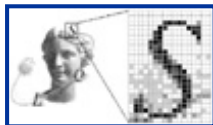
## Bitmaps

Computer graphics would quickly get boring if only lines and polygons could be used to construct a scene in a game or an interface for a business application. The human brain has incredible pattern recognition skills, and utilizing this in a game or user interface can bring to life exotic locales or make a business application easy and intuitive to use. Take a look at the Windows desktop or any business application, and the prevalent use of icons and other graphical metaphors to indicate a function. Generally speaking, these can be considered bitmaps. Like lines, a bitmap is a collection of pixels arranged in such a way that they represent a picture.



**bitmap:** A collection of pixels arranged in a rectangular configuration in such a way that they form a recognizable shape, such as a picture or texture.

Digitized photos, the little images of floppy disks, printers, and pieces of paper on buttons in a business application, and the texture on the walls of the latest 3-D first-person shooter are all considered bitmaps. To coin a cliché, a picture is indeed worth a thousand words, and their liberal use in games and applications can result in several hundred, or perhaps even thousands, of bitmap images in use in any one program. By their very nature, bitmaps consume lots of memory, but few serious applications can succeed without employing them.



**Figure 3-5:** A bitmap is a collection of pixels

## Graphical Element Classifications

Apart from the basic graphical elements described above, conversations in graphical programming sometimes indicate a broader subdivision of graphic types. Graphical output can generally be divided into two classifications: raster and vector. It is rare to hear about game applications being described as one or the other, but it is very common to discuss graphically based business applications or tools in terms of a raster-based or vector-based application.

## Raster Graphics

Raster graphics are images that are composed of individual pixels statically arranged in such a way as to represent a picture or image. Realistically, this refers to images such as icons and bitmaps.



**raster graphics:** Graphics that are composed of a collection of pixels that together form a picture or image. Generally refers to bitmaps, icons, and other such graphical types.

In general, raster graphics are very resolution dependent, and tend to lose resolution when scaled to a different size. The algorithms necessary for scaling or rotating raster graphics are generally very complex, and the quality of the results varies wildly depending upon the implementation.

The most common raster graphic file format is the humble Windows bitmap. Bitmaps can be created using the Paint application that comes with Windows. Delphi contains an Images directory that includes several bitmap files to be used as button glyphs or backgrounds within applications.

## Vector Graphics

Vector graphics are images that are described as a series of connected points in a virtual 2-D or 3-D grid. This type of graphic includes lines, polygons, circles, and curves. TrueType fonts, being described by a collection of connected points, are also considered to be vector graphics.



**vector graphics:** Graphics that are represented by a series of connected points in a virtual 2-D or 3-D Cartesian (or similar) coordinate system. Generally refers to lines, polygons, curves, and other such graphical types.

Unlike raster graphics, vector graphics are resolution independent, and can be scaled to any size without degradation of image quality. Although somewhat complex, the algorithms for scaling and rotating vector graphics are well documented and easily implemented, and the quality of the results is identical from implementation to implementation.

A common vector graphic file format is the Windows metafile. Its most prevalent use is with clip art type graphics that come with word processors and presentation programs. Because of its resolution independence, clip art stored in this format can be enlarged or shrunk as the user requires.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Video Mode Resolutions and Color Depth

Discussions of graphics programming usually start from a consideration of what resolution the video driver is currently in, and how many colors are available in which to draw the graphics. These two concepts are directly related, and are generally referred to in a combined sense as the video mode.



**video mode:** The video hardware state which dictates the display resolution and number of available colors for graphical output.

These two items will have a profound effect on the graphics programming techniques involved in drawing the graphical elements previously discussed. Both the resolution and the color depth of the current video mode will affect the perceived quality of the displayed graphical images. The greater the resolution and color depth, the more realistic and detailed the displayed graphics will become. However, on the downside, higher resolution and deeper color depth mean greater memory usage and slower speed.

### Resolution

The resolution of the current video mode controls how many pixels can be displayed on the screen.



**resolution:** A measurement of the number of pixels displayed on the screen.

As the resolution increases, more and more pixels are available for displaying graphics. This will directly affect the perceived size of graphical images. This is especially true for raster graphics, as vector graphics can be adjusted accordingly. For example, if a bitmap image is 50 pixels by 50 pixels in size, it will appear larger when displayed at its original resolution when the video mode is set to 800 x 600 pixels than it will when the video mode is set to 1024 x 768 pixels. As illustrated in Figure 3-6, a fixed size bitmap image will appear smaller in higher resolution video modes because individual pixels must be smaller in order to fit more of them onto the screen.



**Figure 3-6:** Two different disEplay resolutions

Supported resolutions are dependent upon the video card installed in the machine, the video driver, and the monitor upon which the output is displayed. The video driver and video card dictate the video modes available for use. However, occasionally the monitor connected to the machine will be unable to handle the new video mode.



**Caution:** Switching to a video mode that is unsupported by the monitor can potentially damage it, and should be avoided.

Perhaps the most common display resolution employed by desktop machines today is 800 pixels wide and 600 pixels high. However, it is usually wise to develop for the smallest common denominator in order to take advantage of a broader market (and subsequent revenue potential). Thus, many developers create games based on a 640 x 480 resolution. Generally speaking, it is harder to write code that will support a wide variety of resolution sizes. Nevertheless, the extra effort that allows users to take advantage of new equipment usually pays off, and makes your customers very happy.

### Color Depth

The color depth of the current video mode controls how many colors in which an individual pixel can be drawn. Color depth is measured in bits per pixel, meaning that the more bits used to describe a color, the more colors there are available in which to draw a pixel.



**color depth:** A measurement of the number of bits per pixel required to describe the color of an individual pixel.

The color depth of the current video mode will affect the techniques used to draw graphics more so than the display resolution. Color depth determines if the video mode will be palettized or non-palettized (discussed below). As this affects how pixels are stored, different techniques must be employed when drawing graphics. As the bits per pixel increase, the number of colors available for each individual pixel increases, but so does the amount of memory required to hold a bitmap. The following table summarizes common color depths and the amount of colors available per pixel under each:

Bits Per Pixel	Number of Colors Available
4	16 colors
8	256 colors
16	65,536 colors (high color mode)
24	16,777,216 colors (true color mode)
32	16,777,216 colors plus 256 levels of transparency (true color)

Color depths of only 4 bits per pixel are no longer used for games. Perhaps the most common color depth used in games is 8 bits per pixel, as this makes it very easy to manipulate individual pixels in a bitmap, or to move large amounts of pixels around in memory. However, thanks to hardware acceleration, faster processors, and other advances, higher color depths are starting to be used in more games. 16-bit color games are becoming more prevalent, but 24-bit and 32-bit color depths are still too slow to be used for games that run on anything other than today's higher end PCs.

## Video Board Memory

Apart from the video modes supported by the video driver and the graphics card itself, available video modes are further restricted by the amount of video RAM available on the graphics card. Only video RAM can be used support various video modes, and as the resolution and color depth of a video mode increase, so does the amount of video RAM required to support it. For example, a video mode of 800 x 600 pixels at a color depth of 8 bits per pixel requires 480,000 bytes, almost half a MB. If a video mode requires more memory than is available on the video card, it cannot be supported. For example, a video mode of 1024 x 760 at 16 bits per pixel cannot be supported unless the video card has at least 1,572,864 bytes of RAM available. This will become much more important later when we discuss DirectX surfaces.

## The Components of a Color

When discussing pixel colors and color depth, it is important to understand how colors are represented internally. The color of every pixel drawn to the screen is determined by three components: a red intensity, a blue intensity, and a green intensity. As these intensities increase or decrease, the color of an individual pixel is affected accordingly. Each intensity can have a minimum value of 0, and a maximum value of 255. If the intensity of each component is 0, the resulting color is black, while if each component is at 255, the resulting color is white. This gives a maximum of 16,777,216 colors, well beyond the scope of what the human eye can accurately detect from shade to shade. This is referred to as RGB color, standing for red, green, and blue, respectively.



**RGB color:** Refers to the method by which colors are represented, usually in the form of a red, green, and blue component, respectively.

Most of the time, colors are used in this RGB format. However, occasionally colors will be used in a reverse format (BGR), depending on which GDI functions are being used.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- Search Tips
- Advanced Search

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Palettized Modes**

Four and eight bits-per-pixel color depths are the only palettized video modes. As 4 bits-per-pixel color depths are no longer used for games, we will discuss only 8 bits-per-pixel color depths. At 8 bits-per-pixel, each pixel can assume a value between 0 and 255. These values act as an index into what can be considered a 256-entry array known as the color palette. Now, each entry in this array contains an 8-bit value for the blue, green, and red intensity of a color. In effect, this gives 8 bits-per-pixel color depths access to 16,777,216 potential colors, but only 256 of them can be shown at one time.



**Figure 3-7:** An 8 bits-per-pixel bitmap's pixels act as indices into a color palette

There are many things to consider when working with 8 bits-per-pixel color modes. A major disadvantage to this mode is that all bitmaps to be displayed simultaneously must be drawn using the same palette. This requires a lot of careful planning on the part of art production, and can limit the creativity of artists and the variety of artwork that can be displayed on screen. Bitmaps that are drawn using one palette but displayed on a device in which a different palette is currently being used, will have unpredictable results.

A bitmap's pixels could be mapped to the closest approximation of the appropriate color in the current palette, but this process is slow, usually produces inadequate results, and is not appropriate for fast action games. Different palettes can be used between scenes, however, which can reduce this effect. However, 8 bits-per-pixel video modes give the greatest speed advantage, and a tremendous number of colors can potentially be used while taking up the least amount of RAM. Up to four pixels can be moved simultaneously by using 32-bit memory movement instructions, allowing bitmaps to be moved about at blindingly fast speeds.

Another major advantage comes from palette animation. If colors within the palette are moved about, or cycled, every pixel drawn using that color is affected simultaneously and automatically. This can be used for various special effects, such as flowing water or lighting. Palette animation is not available under non-palettized video modes. Since this book is concerned with high-performance games and graphics programming, all examples will use only 8 bits-per-pixel video modes in order to squeeze out as much speed



as possible.

## Non-palettized Modes

16, 24, and 32 bits-per-pixel video modes directly store the color value for each pixel. This color value is encoded according to the bits-per-pixel used, and can vary from video board to video board. In general, 16 bits-per-pixel color depths use five bits to describe each of the red, green, and blue intensities of a color. 24 bits-per-pixel uses a full eight bits to describe each intensity, and 32 bits-per-pixel uses the same format with eight extra bits used as a translucency value.

Non-palettized video modes have the advantage of total freedom to display any number of bitmaps simultaneously, without worrying about color problems or palette management. The downside to this is that more memory is required for both display and bitmap storage, and games tend to run slower as it requires longer to move a bitmap from one place in memory to another. This book will concentrate on using only 8 bits-per-pixel color depths, so any further discussion of non-palettized video modes is beyond our scope.

## Graphics Output Techniques

As stated above, the GDI provides the primary functions and procedures by which graphics can be drawn to the screen. There are literally hundreds of functions and procedures, and covering all of them is well beyond the scope of this book. However, we will discuss a number of the most common GDI graphical output functions and will look at several examples that demonstrate their use.

### The Device Context

A device context represents a surface upon which graphical output can be drawn. Specifically, a device context represents the surface of an output device such as a monitor, printer, or plotter, and gives the developer access to modify its appearance. This access is accomplished through the specific device's device driver.



**device context (DC):** Represents the surface of an output device such as a monitor, printer, or plotter, allowing the application to perform graphical output.

Under regular Windows programming, the device context provides the only way to render graphics to the screen. Device contexts can be used with Direct-Draw functions to make certain tasks easier, as we'll see in later chapters. Most, if not all, GDI API functions require a handle to the device context upon which the specified graphical operation will be performed. Device contexts represent certain areas on the screen, from the surface of a control (such as a button) to the entire surface of a window or even the screen itself. To retrieve a handle to a device context, an application calls the GetDC function, passing it the handle of the window whose surface is to be drawn upon. It is very important to call the ReleaseDC function when the application is finished drawing its output, as neglecting to release the device context can cause Windows to lock up. Fortunately, we will not need to be concerned much with device contexts, as the TCanvas object encapsulates the functionality of the device context and makes graphical output much easier.

### Painting on the Canvas

Objects derived from TGraphicControl and TCustomControl provide a TCanvas property for performing graphical output. This allows the developer to determine the appearance of the control. The TCanvas object, as an encapsulation of the device context, contains numerous methods for drawing graphics, making the use of most GDI function calls unnecessary. However, when using GDI API functions that are not encapsulated by the TCanvas object, the Handle property can be used in parameters that require a handle to a device context. Lines, polygons, even individual pixels can be drawn on the surface of a TCanvas object with relative ease. Again, like the device context, the TCanvas object represents the surface of a window, such as a control or the application window itself. Thus, when anything is drawn on the canvas, it will appear on screen in the position of the control or window which it represents.



---

**Note:** Graphics output is controlled by specifying a horizontal and vertical coordinate, with the horizontal coordinate increasing from left to right and the vertical coordinate increasing from top to bottom. Since the TCanvas object represents an individual window or control, the origin of its coordinates is relative to the window or control it represents. Thus, the coordinate 0, 0 will be in the upper left-hand corner of the window or control it represents, which may or may not be the upper left-hand corner of the physical screen.

---

Now let's examine some techniques for drawing graphics. These techniques will be concerned with drawing graphics primitives, or simple lines, dots, and polygons. These simplistic techniques demonstrate how easy it is to use the TCanvas encapsulation of a device context for performing graphical output. They all use the Canvas property of the main form for the drawing surface.

[Previous](#) | [Table of Contents](#) | [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Pens and Brushes**

TCanvas objects contain several properties, two of which are extremely important when performing graphical output: the Pen property and the Brush property. The Pen property determines the characteristics of lines drawn on the TCanvas, such as a line's thickness and color. The Brush property determines the appearance of the interiors of polygons, such as the color and pattern. Using the TPen and TBrush objects, several pens and brushes can be constructed by the application and assigned to the appropriate properties when necessary. However, most of the time, it is sufficient to simply change the properties of the TCanvas as needed.

**Drawing Pixels**

The Pixels property of the TCanvas object gives the application access to individual pixels of the surface. This is accessed like a two-dimensional array, with the first index acting as the horizontal coordinate and the second index acting as the vertical coordinate. The Pixels property reports or receives a color in the form of a TColor type (which is an RGB color value, regardless of the video mode). The following example demonstrates using the Pixels property of a TCanvas object to draw random pixels.

*Listing 3-1: Using the Pixels property*

---

```
procedure TfrmPixelDemo.TimerTimer(Sender: TObject);  
  
var  
    iCount: Integer;  
  
begin  
    {draw 100 random pixels in a random RGB color}  
    for iCount := 0 to 99 do  
        Canvas.Pixels[Random(ClientWidth),Random(ClientHeight)] := RGB(Random(256),
```

```
Random(256),  
Random(256));
```

```
end;
```

---

Unfortunately, this method of accessing individual pixels of an image is incredibly slow, and should never be used for fast graphical output.



**Figure 3-8:** Drawing individual pixels

## Drawing Lines

Lines are drawn using the attributes set by the Pen property. The PenPos property determines the current position of the pen or the starting point of any drawing operation that uses the pen (such as drawing lines). This position can be modified by directly setting the PenPos property or by calling the MoveTo method. The LineTo method is called to actually draw the line, from the position indicated by the PenPos property to the position passed in the LineTo method. This pen position is updated to the coordinates passed to the LineTo position after the line is drawn. The following example demonstrates how to use these methods to draw various lines of random length, color, and width.

*Listing 3-2: Using the MoveTo and LineTo methods*

---

```
procedure TfrmLineDemo.TimerTimer(Sender: TObject);  
  
begin  
    {set a random pen color and width}  
  
    Canvas.Pen.Color := RGB(Random(256), Random(256), Random(256));  
  
    Canvas.Pen.Width := Random(5);  
  
    {draw a line between two random points}  
  
    Canvas.MoveTo(Random(ClientWidth), Random(ClientHeight));  
  
    Canvas.LineTo(Random(ClientWidth), Random(ClientHeight));  
  
end;
```

---



**Figure 3-9:** Drawing random lines

## Drawing Polygons

Polygons are drawn using the attributes set by both the Pen and Brush properties. The Pen property determines the outline of the polygon, and the Brush property determines how the polygon is filled. Several methods exist that draw polygons of different shapes, from rectangles to circles. The Polygon method can be used to draw polygons of any arbitrary shape simply by passing it an open array of TPoint structures. The following example demonstrates how to use several polygon drawing methods to draw polygons of various shapes and colors.

*Listing 3-3: Using various polygon drawing methods*

---

```
procedure TfrmPolygonDemo.TimerTimer(Sender: TObject);

type
    {enumerated type representing several standard polygon drawing routines}
    TPolyType = (ptChord, ptEllipse, ptPie, ptRectangle, ptRoundRect);

var
    X1, X2, X3, X4, Y1, Y2, Y3, Y4: Integer;

begin
    {initialize a set of random coordinates}

    X1 := Random(ClientWidth);
    X2 := Random(ClientWidth);
    X3 := Random(ClientWidth);
    X4 := Random(ClientWidth);

    Y1 := Random(ClientHeight);
    Y2 := Random(ClientHeight);
    Y3 := Random(ClientHeight);
    Y4 := Random(ClientHeight);

    {set the brush to a random RGB color}

    Canvas.Brush.Color := RGB(Random(256), Random(256), Random(256));
```

```

{draw a random polygon}

case TPolyType(Random(Ord(ptRoundRect)+1)) of

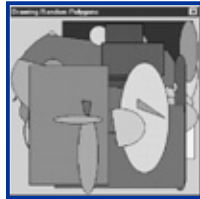
    ptChord      : Canvas.Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4);
    ptEllipse    : Canvas.Ellipse(X1, Y1, X2, Y2);
    ptPie        : Canvas.Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4);
    ptRectangle  : Canvas.Rectangle(X1, Y1, X2, Y2);
    ptRoundRect  : Canvas.RoundRect(X1, Y1, X2, Y2, X3, Y3);

end;

end;

```

---



**Figure 3-10:** Drawing random polygons

## Drawing Text

Perhaps the most common use of TCanvas graphical output techniques is for drawing text. The Font property of the TCanvas object determines the attributes of the text, including its typeface, size, and style. The TextOut method is used to draw the specified string at the indicated coordinates. These coordinates denote the top left corner of the text string. The following example demonstrates how to use the TextOut method to draw randomly colored text at random positions on the form.



**Tip:** By setting the Style property of the canvas's Brush property to bsClear, the background behind text will be transparent. This allows images underneath the text to show through, and is useful for drawing text on top of bitmap images.

---

### *Listing 3-4: Using the TextOut method*

---

```

procedure TfrmTextDemo.TimerTimer(Sender: TObject);

begin
    {set a random text color}

    Canvas.Font.Color := RGB(Random(256), Random(256), Random(256));

    {set a random text size}

```

```
Canvas.Font.Size := Random(10)+5;

{set the text typeface}

Canvas.Font.Name := Arial';

{set the brush style to clear so that the image behind the text
will be visible}

Canvas.Brush.Style := bsClear;

{draw text at a random coordinate}

Canvas.TextOut(Random(ClientWidth), Random(ClientHeight), Delphi Graphics');

end;
```



**Figure 3-11:** Drawing random text

## Bitmap Essentials

It is quite possible for one to create a really engaging and entertaining game using nothing but line and polygon drawing techniques. However, if such graphical output were used in all games, everything would look very similar, and there would be little visual variety. In order to achieve the desired visual quality and variety that would be necessary in almost any game, bitmaps must be used.

Bitmaps come in many varieties, from two color to true color, and in many different formats, such as the Windows BMP format, JPEG, TIFF, GIF, etc. Each format presents its own benefits and weaknesses, and each is useful under specific circumstances. Obviously, the only format native to Windows is the Windows BMP format. This format can easily be accessed by low-level Windows API functions as well as Delphi VCL objects and functions, so the BMP format is the one on which we will focus.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



# ITKNOWLEDGE.COM<sup>SM</sup>

Need IT. Find IT. Know IT.

Monday, January 12, 2004

## KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

## PUBLICATION LOOKUP

## JUMP TO TOPIC



## Desktop Library

[Click to access!](#)



### Delphi Graphics and Game Programming Exposed with DirectX 7.0

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

### Bitmap Types: DDB and DIB

Most bitmap formats have subformats that allow an image to be stored similarly to the original format but with slight modifications, and the Windows bitmap format is no exception. Typically, subformats come about as a result of updated technology. New technology may contain enhanced functionality that was not available when the original format was created. Thus, subformats are born containing attributes that allow the image to take advantage of the new technological achievements. Windows bitmaps come in two flavors: DDB and DIB.

### Device-Dependent Bitmaps

The device-dependent bitmap, or DDB, is an older bitmap format and is kept around solely for backward compatibility. This format is so named because it is very dependent on the device upon which it is displayed for certain information. The DDB format does not store any color palette information or resolution information, storing only its width, height, color depth, and array of pixels that make up the image. DDBs always use the system color palette when they are displayed. Coupled with the lack of original resolution information, it is very difficult to accurately display a DDB in its original form on anything other than the original device upon which it was created. Additionally, the image itself can only be modified through the use of GDI functions. The storage and display of a DDB is handled natively by the video driver and video hardware. Naturally, this makes the use and display of DDBs very fast, but they are not very flexible.



**device-dependent bitmap (DDB):** An older bitmap image format that stores only height, width, color depth, and image pixels.

### Device-Independent Bitmaps

The device-independent bitmap, or DIB, stores all the information required to accurately display the image on any device. Such information includes the color palette for the image, the resolution of the device upon which the bitmap was originally created, and a data compression flag. The pixels defining the DIB image are also directly accessible to the developer, making this format incredibly flexible. The DIB bitmap format describes a standardized pixel format, making the storage and display of such images a responsibility of generic GDI functions and global memory. As such, the use and display of DIBs is not quite as fast as DDBs, but their flexibility more than makes up for the loss in speed.



---

**device-independent bitmap (DIB):** A flexible bitmap image format that includes all necessary information to accurately reproduce the image on any device.

## The TBitmap Object

The Delphi TBitmap object encapsulates the functionality of both the DDB and DIB image formats. By default, when a TBitmap object is created, it holds an image in the DDB format. When a bitmap is loaded into it, the image will be stored in the DIB format. The TBitmap object contains many useful properties and methods that make the manipulation of bitmap images incredibly easy.

## Loading Images

Using regular GDI calls to simply load an image from a file may take half a dozen lines of code or more. With the TBitmap object, a simple call to its LoadFromFile method will load the image data and its palette, at which point the bitmap is ready for use. If the image is stored in a resource attached to the executable (or a DLL), the TBitmap's LoadFromResourceID or LoadFromResourceName methods give the developer flexibility in determining how the image is extracted from the resource. These methods alone more than justify the use of a TBitmap object over hard-core GDI function calls.

## Displaying Images

The TBitmap object itself does not contain any methods that deal directly with its display. This functionality lies in the domain of the TCanvas object. There are several methods by which a bitmap can be displayed, and they all have similar ancestors in the form of GDI function calls. The act of copying bitmap pixels to the screen for display is known as a bitblt, or blit.



---

**bitblt:** bit block transfer. Pronounced “blit,” this is the act of copying the pixels of a bitmap image onto the screen or other graphical output surface for display.

The easiest TCanvas method available for displaying an image is Draw. It simply takes three parameters: a horizontal and vertical position of the upper left-hand corner of the image and the image object itself. This draws the entire image with its upper left-hand corner located at the specified coordinate. Other methods include CopyRect, which allows only a portion of an image to be drawn (scaling as necessary), and StretchDraw, which allows the image to be drawn larger or smaller than its original size. Listing 3-5 illustrates loading a bitmap and displaying it on the surface of a form.

### *Listing 3-5: Loading and displaying a bitmap image*

---

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    {create the bitmap object}

    Bitmap := TBitmap.Create;

    {load in a bitmap image}

    Bitmap.LoadFromFile(ExtractFilePath(ParamStr(0))+'Athena.bmp');
end;
```

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    {free the bitmap object}
    Bitmap.Free;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    {display the bitmap using the Draw method}
    Canvas.Draw(8, 24, Bitmap);

    {display the bitmap using the CopyRect method}
    Canvas.CopyRect(Rect(192, 24, 344, 104),Bitmap.Canvas,Rect(62, 39, 111, 74));

    {display the bitmap using the StretchDraw method}
    Canvas.StretchDraw(Rect(8, 184, 352, 264), Bitmap);
end;

```



**Figure 3-12:** An image displayed using several different methods

### Direct Pixel Access

One of the biggest benefits of using a DIB over a DDB is the fact that individual pixels can be directly accessed. This makes feasible many advanced graphical rendering techniques, such as ray casting. Accessing the individual pixels of a DDB (and even using the Pixels property of a TCanvas) is incredibly slow and could never be used for such advanced graphical output techniques.

To access individual pixels using the TBitmap object, the ScanLine property must be used. The ScanLine property is an indexed property that requires a row number and returns a pointer to the block of memory that contains the specified rows of pixels. The actual format of the information stored in this block of memory is dependent upon the value of the TBitmap's PixelFormat property. The PixelFormat value reflects the number of bits required to describe one pixel in the image. Refer to the color depth topic discussed previously for more information on bits-per-pixel and its relation to the number of colors an image can contain. Thus, it is very important to know how the image information is stored before accessing it.

Since this book is primarily concerned with 256-color images and graphical modes, assume that a TBitmap object



has been loaded with a 256-color bitmap image. A variable of type PByteArray can be used to access the pixels pointed to by the ScanLine property. PByteArray is defined as a pointer to an array of bytes. The result of accessing the ScanLine property is assigned to our PByteArray variable, at which point the individual pixels can be accessed by a simple index into the array. Since a 256-color bitmap defines one pixel as 8 bits, or a byte, using a PByteArray variable allows the developer to access individual pixels correctly. If the image were of any other color depth, a different approach to accessing the pixels would need to be used. Listing 3-6 demonstrates accessing individual pixels through the ScanLine property.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**ITKNOWLEDGE.COM**<sup>SM</sup>  
Need IT. Find IT. Know IT.

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 3-6: Accessing individual pixels through the ScanLine property*

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Bitmap: TBitmap;
    Pixels: PByteArray;
    iBand, iRow, iCol: Integer;
    DrawColor: Byte;
begin
    {create the bitmap object}
    Bitmap := TBitmap.Create;

    {initialize the appropriate properties so we can access individual pixels
    in a known way}
    Bitmap.Width := 169;
    Bitmap.Height := 128;
    Bitmap.PixelFormat := pf8Bit;
```

```
{create 8 horizontal bands of alternating colors}

DrawColor := 19;

for iBand := 0 to 7 do

begin

    {flip-flop the color. this is an 8 bit image, so we specify the color
    as an index into the palette.}

    if DrawColor = 16 then

        DrawColor := 19

    else

        DrawColor := 16;

    {each band is comprised of 16 rows of color}

    for iRow := 0 to 15 do

    begin

        {retrieve a pointer to the row of pixels}

        Pixels := Bitmap.ScanLine[(iBand*16)+iRow];

        {set each pixel in the row to the specific color}

        for iCol := 0 to Bitmap.Width-1 do

            Pixels[iCol] := DrawColor;

        end;

    end;

    {draw the bitmap to the screen}

    Canvas.Draw(8, 8, Bitmap);

    {free the bitmap}

    Bitmap.Free;

end;
```



**Figure 3-13:** Individually painted pixels

## Raster Operations

In addition to simply copying bitmap pixels from place to place, certain functions can modify exactly how the pixels are drawn onto their destination according to a specified setting. This is known as a raster operation, or ROP, and it determines how the pixels from the source, the destination, and the destination device context's selected brush are combined. The pixels are combined using boolean operations.



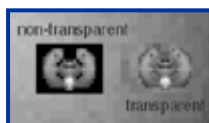
**raster operation (ROP):** A Boolean function performed on pixels when they are copied from bitmap to bitmap that determines how the pixels from the source, the destination, and the destination device context's selected brush are combined.

There are 256 total raster operations, each producing a different output based on the Boolean operator used and the order in which pixels are combined. Of these, only 16 are commonly used in Windows graphics programming. There are several GDI functions, such as `BitBlt` and `StretchBlt`, that allow a developer full access to all 256 raster operations. Using the Delphi VCL, the `CopyMode` property of the `TCanvas` object specifies the raster operation to be used, and can be set to one of 16 constants that represent the 16 most commonly used raster operations.

Raster operations are incredibly useful under certain circumstances. They can be used to easily implement specific special effects which, without raster operations, would be more difficult to implement otherwise. For example, using raster operations, it is very easy to simulate transparent pixels in a bitmap when it is displayed over a background. This effect and others could be implemented using any one of a hundred different techniques, but raster operations make their implementation fast and simple.

## Simulating Transparency

Perhaps the most common function performed by the graphical output code of a game is to display one bitmap over another, with the pixels of the background showing through the "transparent" pixels of the displayed bitmap or sprite. Without transparency, every sprite image would have to be rectangular, or the resulting animation would look horrible. Figure 3-14 graphically illustrates the importance of transparency.



**Figure 3-14:** The effect of transparency

The easiest method by which sprites can be transparently drawn to the background is known as masking. Masking uses two bitmaps for each sprite image, as each bitmap will be used with a different raster operation when the sprite is drawn.



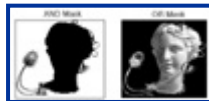
**masking:** The act of using two bitmaps, an AND bitmap and an OR bitmap, along with raster operations to draw one bitmap over another while simulating transparent pixels in the source bitmap.

This method is precisely what happens when an application sets a TBitmap object's Transparent property to True and then calls TCanvas.Draw. This is a very convenient encapsulation in Delphi. However, the only drawback is that the transparent color is either the color of the pixel in the bottom leftmost pixel of the bitmap or is specified in the TBitmap's TransparentColor property, depending on the value of its TransparentMode property. The result is that this specified transparent color cannot be used elsewhere in the image, and unless proper care is taken when constructing the image, the result may not be what is expected. Using the direct approach as explained below gives the developer more control over the appearance of the final image. Figure 3-15 displays the difference between using the TransparentColor property and using the technique described below.

The two sprite images consist of an AND mask and an OR mask. The OR mask is the actual sprite image as it should appear in the destination. The only requirement for the OR mask is that pixels that should be transparent must be totally black (this must be palette slot zero for 256 color images). The AND mask is a monochrome bitmap consisting of only black and white. The black areas should match those of the actual sprite image in the OR mask, and the white areas represent pixels that should be transparent (opaque, or sprite image pixels, must be palette slot 0, and transparent pixels must be palette slot 255). Figure 3-16 illustrates a sprite image (the OR mask) and its accompanying silhouette (the AND mask).



**Figure 3-15:** Using the TBitmap.Transparent property versus the direct approach



**Figure 3-16:** A sprite AND and OR mask

Drawing a sprite image with transparent pixels requires two steps. First, the AND mask is drawn to the destination bitmap using a boolean AND raster operation. The white pixels (representing the transparent areas) have a binary value of 11111111 for 256-color images (a decimal value of 255, the last palette slot). Higher color depths will result in a binary value of all ones. When this binary value is ANDed with the colors of the pixels in the destination image, the result is the color of the pixels in the destination image, thereby preserving the destination image data. The black pixels (representing opaque image data), on the other hand, have a binary value of 00000000. When ANDed with the destination image, the result is black. Effectively, this creates a “cutout” area in the destination image where the actual sprite image will appear.

The second step is to draw the OR mask to the destination bitmap using a boolean OR raster operation. The black pixels of the OR mask (representing the transparent areas), when ORed with the destination, will preserve the destination image data, just as the white pixels did in the first step. The actual sprite image data, when ORed with the destination, will be combined with the black “cutout” area produced from the first step. Thus, the sprite image data will be preserved. This results in a perfectly copied sprite with the background image showing through the transparent areas. Figure 3-17 demonstrates the results of each step in the process.



**Figure 3-17:** Using masks with raster operations

The result, while perhaps not blindingly fast, should be more than adequate under most circumstances, if there are not a large amount of sprites on the screen. This technique is easy to implement, but it does require double the storage space for sprite images. The following example demonstrates how this technique is implemented.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**

[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 3-7: Implementing masking techniques*

```
procedure TForm1.FormPaint(Sender: TObject);  
  
begin  
    {copy the background image to the destination}  
    Canvas.Draw(480, 48, imgBackground.Picture.Bitmap);  
  
    {combine the AND mask image with the background image in the destination  
    using a boolean AND operation. this carves out an area for the final  
    foreground image}  
    Canvas.CopyMode := cmSrcAnd;  
  
    Canvas.Draw(480+((imgBackground.Width div 2)-(imgAndMask.Width div 2)),  
                48+((imgBackground.Height div 2)-(imgAndMask.Height div 2)),  
                imgAndMask.Picture.Bitmap);  
  
    {copy the result of step one into the background' image used for step 2}  
    Canvas.CopyMode := cmSrcCopy;
```

```

Canvas.CopyRect(Rect(8, 288, 248, 468), Canvas, Rect(480, 48, 720, 228));
{copy the background' image resulting from step 1 into the destination}
Canvas.CopyRect(Rect(480, 288, 720, 468), Canvas, Rect(8, 288, 248, 468));

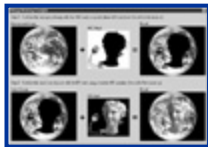
{combine the OR mask image with the result from step 1 in the destination
 using a boolean OR operation.  this copies the foreground image into the
 area carved out by step 1 while preserving the pixels around it, thereby
 creating the illusion of transparency.}

Canvas.CopyMode := cmSrcPaint;

Canvas.Draw(480+((imgBackground.Width div 2)-(imgAndMask.Width div 2)),
           288+((imgBackground.Height div 2)-(imgAndMask.Height div 2)),
           imgORMask.Picture.Bitmap);

end;

```



**Figure 3-18:** The transparently displayed image

## Palettes

As previously discussed, working under palettized video modes provides a number of advantages and disadvantages. Palettized modes can be a real pain at times, requiring careful planning on the part of art production and special considerations on the part of the programmer. However, the advantages of working under a 256-color video mode usually outweigh the disadvantages, primarily because it is a very speedy mode to work under, and palette animation can be used for special effects.

The Windows palette manager is a difficult beast to tame. There are many nuances to working in a palettized mode under Windows, making the creation of games using regular GDI functions difficult indeed. As this book's primary topic is game and graphics programming using DirectX, we will not cover palettes in the depth required for making non-DirectX games. However, it is useful to cover a few of the most basic palette topics in order to prepare the reader for DirectX palettes covered in a later chapter.

### Logical vs. System Palettes

There are two different kinds of palettes used by the Windows operating system: the system palette and logical palettes. The system palette is a Windows-maintained global palette containing all of the colors that can be displayed. The first and last ten palette entries are reserved for the system, and contain the 20 static colors that are used for drawing specific user interface elements, such as window title bars, borders, and 3-D user interface elements. The remaining 236 colors can be set in the application.

Logical palettes are created by the application and contain the colors that the application would like to use.



When the application specifies the palette it wishes to use, the Windows palette manager merges the logical palette with the current system palette. It accomplishes this by examining each entry in the logical palette and mapping it to an existing entry in the system palette, if a particular color exists in both palettes. If no matching entry is found, the color in the logical palette is placed into the first unused entry in the system palette. If an existing entry is not found and there are no more unused palette slots, the Windows palette manager maps the color to the closest approximation in the system palette. This continues until all colors in the logical palette have been merged with the colors in the system palette.

A logical palette must be selected into a device context before it can be used. If the specified device context identifies an active window, the logical palette is treated as a foreground palette and the Windows palette manager will mark all non-static entries in the system palette as unused. This allows the logical palette to potentially replace all non-static colors of the system palette. If the device context does not identify an active window, the logical palette is treated as a background palette, and can set unused palette slots only if any remain after the foreground palette has been merged.

The palette information stored in a DIB bitmap file can be considered a logical palette. Delphi takes care of most palette-specific concerns automatically, as the TBitmap object will load a bitmap's palette and put it into use. In the next chapter, we will learn how to load a palette from a bitmap and use it to initialize the DirectX palette.



**Note:** The TBitmap object has a property called Palette. This property is the handle to the bitmap's logical palette, and can be used in any GDI function that manipulates logical palettes.

---

## Using a Palette

Delphi encapsulates a large portion of the Windows API, making Windows programming easy and fun. Unfortunately, not everything in the Windows API has been conveniently simplified in this manner, and this includes palette functionality. This means that in order to use palettes, one must use low-level Windows API calls throughout most of the process.

In order to begin using a custom palette, a logical palette must first be made. First, variables of type PLogPalette and HPALETTE must be declared. The PLogPalette is a pointer to a logical palette structure, which contains a member for the palette version, a member for the number of palette entries, and then an array of type TPaletteEntry. The TPaletteEntry type is a structure that contains fields for the red, green, and blue components of a color, and a flag indicating how the entry is to be used. The PLogPalette variable is initialized to the appropriate values, including the desired colors, and is then passed to the CreatePalette function. This returns a handle to the newly created palette, which is then assigned to the HPALETTE variable. At this point, the palette can be selected into a device context using the SelectPalette function. If using a bitmap, the palette handle can be directly assigned to the TBitmap.Palette property. To actually merge the logical palette with the system palette so the new colors can be used, call the RealizePalette function. When the application no longer needs the logical palette, it should be deleted by calling the DeleteObject function. The following example demonstrates creating a logical palette to draw a red gradient fill.



**Figure 3-19:** The custom palette

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 3-8: Using custom palettes*

type

```
TForm1 = class(TForm)
```

```
    procedure FormCreate(Sender: TObject);
```

```
    procedure FormPaint(Sender: TObject);
```

```
    procedure FormDestroy(Sender: TObject);
```

```
private
```

```
    { Private declarations }
```

```
    FormPalette: HPALETTE;           // a handle to a logical palette
```

```
public
```

```
    { Public declarations }
```

```
    function GetPalette: HPALETTE; override;
```

```
end;
```

{Whoops! Delphi incorrectly imports the GetSystemPaletteEntries function.

Here's the correct declaration that gives all of the functionality available

```

with this API function}

function GetSystemPaletteEntries(DC: HDC; StartIndex, NumEntries: UINT;
                                PaletteEntries: Pointer): UINT; stdcall;

var
    Form1: TForm1;

implementation

{$R *.DFM}

function TForm1.GetPalette: HPALETTE;
begin
    {when something requests the palette of the form
     pass it back the new logical palette}
    Result := FormPalette;
end;

{link in the GetSystemPaletteEntries function}
function GetSystemPaletteEntries; external gdi32
    name GetSystemPaletteEntries';

procedure TForm1.FormCreate(Sender: TObject);
var
    ThePalette: PLogPalette;    // a logical palette definition structure
    iLoop: Integer;            // general loop counter
begin
    {get enough memory to hold the colors in the first 10 system
     palette slots, plus 32 of our own. this memory is temporary,
     and is no longer needed once the palette is created.}
    GetMem(ThePalette, SizeOf(TLogPalette)+42*SizeOf(TPaletteEntry));

```

```

{initialize the palette version number}
ThePalette^.palVersion := $300;

{we will have a total of 42 entries in our palette}
ThePalette^.palNumEntries := 42;

{get the first 10 system palette entries}
GetSystemPaletteEntries(Form1.Canvas.Handle, 0, 10,
                        @(ThePalette^.palPalEntry));

{we only want 32 new palette entries, and we want them to start
 immediately after the first 10 system palette entries. by retrieving
 the first 10 system palette entries, when we realize our new palette,
 the first 10 logical palette entries will be mapped to the first 10
 system palette entries, and our palette entries will follow}
for iLoop := 0 to 31 do
begin
    {create a gradient red palette}
    ThePalette^.palPalEntry[iLoop+10].peRed    := 255-((255 div 32)*iLoop);
    ThePalette^.palPalEntry[iLoop+10].peGreen := 0;
    ThePalette^.palPalEntry[iLoop+10].peBlue  := 0;
    {do not match this palette entry to any other palette entry}
    ThePalette^.palPalEntry[iLoop+10].peFlags := PC_NOCOLLAPSE;
end;

{create the palette}
FormPalette := CreatePalette(ThePalette^);

{free the temporary memory}

```

```
FreeMem(ThePalette, SizeOf(TLogPalette)+42*SizeOf(TPaletteEntry));  
end;  
  
procedure TForm1.FormPaint(Sender: TObject);  
var  
    OldPalette: HPALETTE;           // a handle to the previous palette  
    iLoop: Integer;                 // general loop control variable  
begin  
    {select our new logical palette into the device context}  
    OldPalette := SelectPalette(Canvas.Handle, FormPalette, FALSE);  
  
    {map our logical palette into the system palette}  
    RealizePalette(Canvas.Handle);  
  
    {display a red gradient}  
    for iLoop := 0 to 31 do  
        begin  
            Canvas.Brush.Color := $01000000 or iLoop+10;  
            Canvas.FillRect(Rect(10, iLoop*10+16, 260, (iLoop*10)+36));  
        end;  
  
        {select the previous palette back into the device context}  
        SelectPalette(Canvas.Handle, OldPalette, FALSE);  
    end;  
  
procedure TForm1.FormDestroy(Sender: TObject);  
begin  
    {we no longer need the logical palette, so delete it}  
    DeleteObject(FormPalette);  
end;
```

---

## Further Adventures in Windows Programming

The rest of this book could be dedicated to Windows GDI programming techniques without ever exhausting the subject. Indeed, several such books have been written that cover Windows graphics programming in much greater detail than that presented here. Our goal with this book is to study DirectX programming techniques under Delphi, so further discussion of GDI programming techniques is beyond our scope. However, if you would like more information concerning graphics programming under Windows, consult the following titles:

*Black Art of Windows Game Programming* (ISBN 1-87873-995-6, Waite Group Press)

*Delphi 2 Multimedia Adventure Set* (ISBN 1-88357-764-0, Coriolis Group)

*The Tomes of Delphi 3: Win32 Core API* (ISBN 1-55622-610-1, Wordware Publishing)

Several more books that cover graphics programming, both Windows-specific and general algorithms, are listed in the bibliography at the end of the book. The remainder of this book will be dedicated to creating high-speed games using DirectX techniques.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Summary**

In this chapter, we discussed several basic topics in graphics programming. Although all of the examples demonstrated these topics using normal Windows GDI function calls, the concepts and theories presented apply to graphics programming in general. When coding a graphically intensive application under Windows, it is important to keep these points in mind:

- The GDI subsystem is responsible for all of the graphical output displayed to the screen during a Windows session. One of Windows' strongest attributes is its device independence, or its ability to be run on a wide variety of hardware combinations. Due to this requirement, the GDI's primary purpose is to provide a mechanism by which an application can output graphical content without the programmer being concerned about what type of hardware upon which the application will be running. Unfortunately, as a direct consequence of this device independence, the GDI is notorious for its slow performance. Even so, the GDI is very useful when creating tools for games, and some commercial games have been written using little more than generic GDI function calls.
- One must be familiar with a number of terms that describe various common graphical elements. Such graphical elements include pixels, lines, polygons, and bitmaps. A pixel is defined as the smallest, single element in a graphical image or on the display surface. Lines are a collection of pixels, usually arranged in a linear configuration. A collection of lines, arranged in such a way that the ends are connected to form a continuous, closed shape, is known as a polygon. A bitmap is a collection of pixels arranged in a rectangular configuration in such a way that they form a recognizable shape, such as a picture or texture.
- Discussions of graphics programming usually start from a consideration of what resolution the video driver is currently in and how many colors are available in which to draw the graphics. These two concepts are directly related, and are generally referred to in a combined sense as the video mode. Both the resolution and the color depth of the current video mode will affect the perceived quality of the displayed graphical images. The greater the resolution and color depth, the more realistic and detailed the displayed graphics will become. However, on the downside, higher resolution and deeper color depth mean greater memory usage and slower speed.
- The color of every pixel drawn to the screen is determined by three components: a red intensity, a blue intensity, and a green intensity. As these intensities increase or decrease, the color of an individual pixel is affected accordingly. Each intensity can have a minimum value of 0 and a maximum value of 255. If the intensity of each component is 0, the resulting color is black, while if



each component is at 255, the resulting color is white. This gives a maximum of 16,777,216 colors. This is referred to as RGB color, denoting red, green, and blue, respectively.

- Four and eight bits-per-pixel color depths are the only palettized video modes. At 8 bits-per-pixel, each pixel can assume a value between 0 and 255. These values act as an index into what can be considered a 256-entry array known as the color palette. Each entry in this array contains an 8-bit value for the blue, green, and red intensity of a color. Palettized display modes require careful planning of art resources, but are very fast and allow for palette animation special effects. By contrast, 16, 24, and 32 bits-per-pixel video modes directly store the color value for each pixel. This color value is encoded according to the bits-per-pixel used, and can vary from video board to video board. Non-palettized video modes have the advantage of total freedom to display any number of bitmaps simultaneously, without worrying about color problems or palette management. The downside to this is that more memory is required for both display and bitmap storage, and games tend to run slower as it requires longer to move a bitmap from one place in memory to another.
- A device context represents a surface upon which graphical output can be drawn. Specifically, a device context represents the surface of an output device such as a monitor, printer, or plotter, and gives the developer access to modify its appearance. Fortunately, we will not need to be concerned much with device contexts, as the TCanvas object encapsulates the functionality of the device context and makes graphical output much easier.
- Bitmaps come in many varieties, from two color to true color, and in many different formats. The native Windows bitmap format is easy to use, and comes in two subformats: DDB and DIB. The device-dependent bitmap, or DDB, is an older bitmap format and is kept around solely for backward compatibility. The device-independent bitmap, or DIB, stores more information than a DDB and can accurately be reproduced on any output device. The use and display of DIBs is not quite as fast as DDBs, but their flexibility more than makes up for the loss in speed. Fortunately, the TBitmap object encapsulates the functionality of both the DDB and DIB image formats, making bitmap manipulation easy.
- There are many nuances to working in a palettized mode under Windows, making the creation of games using regular GDI functions difficult. There are two different kinds of palettes used by the Windows operating system: the system palette and logical palettes. The system palette is a Windows-maintained global palette containing all of the colors that can be displayed. Logical palettes are created by the application, and contain the colors that the application would like to use.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 4 An Introduction to DirectX

This chapter covers the following topics:

- The promise of DirectX
- The benefits and drawbacks of DirectX programming
- DirectX architecture
- DirectX components
- DirectX programming under Delphi
- COM fundamentals
- DirectDraw surfaces
- Page flipping
- Initializing DirectDraw
- GDI rendering with DirectDraw
- Displaying bitmaps
- Retrieving DirectDraw capabilities
- DirectX result codes
- Retrieving the DirectX version

Now that we've covered some basic graphics programming techniques, it's time to dive into what really makes game programming under Windows a reality: DirectX. When DirectX was initially introduced, it was not well received among game programmers. People feared that Microsoft was trying to force them into a game development architecture that would potentially render useless any proprietary graphics techniques. Fortunately, this was not Microsoft's aim, and after DirectX was looked over and evaluated by the gaming industry at large, it was embraced, and has now become the standard low-level programming interface around which games are now created. While it is possible to create some cool games using various GDI functions, to make games that will compete in today's market the savvy Delphi programmer must use the DirectX API.

### What is DirectX?

To put it technically, DirectX is really nothing more than a series of DLLs which give the developer access to low-level hardware while still maintaining some level of device independence. More specifically, DirectX is comprised of a number of COM objects which are implemented in the aforementioned DLLs. These COM objects provide the primary interface to system-level hardware in an object-oriented, language-independent fashion. The overall COM architecture allows the DirectX development team to expand and advance DirectX functionality without affecting applications written with earlier versions of DirectX. In essence, developers can now write games with all the power available under DOS while still maintaining the device independence of the Windows operating system.

In practice, DirectX programming is a lot like using Delphi VCL objects without the benefit of a visual environment. Various DirectX objects are instantiated and initialized, and sometimes linked with other DirectX objects through various method calls. Unlike the VCL, however, DirectX uses several very complex data structures as parameters in many methods. The DirectX API itself is huge, spread across numerous subsystems and various objects within those subsystems. We'll examine several of these subsystems closely throughout the remainder of this book.

## **The Promise of DirectX**

Developing games in a DOS environment can be a nightmare. The multitude of new hardware devices pouring out of hardware vendors had dramatically differing interfaces, making it difficult indeed to support more than a handful of devices in any one game. A high-level operating system like Windows, on the other hand, offers many benefits to game developers, such as consistent programming interfaces to new devices, simplified installation and configuration features, and extensive built-in functionality. However, before the advent of DirectX, this level of abstraction in the Windows environment posed a serious performance penalty that was unacceptable in any fast-action, high-performance game.

The DirectX development team's goal was to create a low-level architecture that would provide developers with the hardware control they need at the speed they demanded, while still retaining Windows' device-independent benefits. This was to be accomplished without imposing a restrictive high-level design architecture. First and foremost, DirectX had to be fast in order to allow game developers to create the high-performance, adrenaline-pumping action games demanded by hard-core game players. DirectX applications must also coexist peacefully with other Windows applications, such as spreadsheets and word processors. Therefore, DirectX promises three things to developers: fast execution, minimal style restraints, and a level of abstraction that will provide benefits from future hardware.

## **Speed**

One of the things that makes DirectX so fast is that it takes advantage of hardware acceleration whenever possible. If a hardware device directly supports a certain function within its electronics, such as copying an image from one area in memory to another, the DirectX drivers send the function call directly to the hardware (which is probably where the "Direct" moniker was derived). This results in blindingly fast performance.

Another benefit to hardware acceleration is true multitasking capabilities. Any DirectX function that is supported in hardware will return immediately after the function is called, even before the task of that function is carried out. This allows the CPU to continue processing application instructions while other hardware is carrying out the previous command. For example, if the display hardware supports moving an image from one area to another, the application can instruct the hardware to perform this task while asynchronously performing game logic calculations.

## **Minimal Style Restraints**

By its very nature, DirectX is a very low-level application programming interface. Game developers have spent a lot of time and money in creating innovative techniques for performing graphical output, techniques which can make one game stand apart from the others. If this competitive edge were to be threatened, many game developers would never have taken a second look at DirectX. Therefore, instead of creating a general gaming library that imposed high-level restrictions in how a game is put together, DirectX acts as a general enhancement to the Windows operating system. This results in many low-level function calls that provide

power and speed while not insisting that developers create games “the right way.” While this may not exactly automate certain tasks, it does allow for the greatest level of creativity and ingenuity when designing new games while affording the developer a certain level of device independence.

## Abstraction

The level of abstraction provided by the Windows operating system allows DirectX to take advantage of hardware advances as they become available. The DirectX interfaces were not designed merely to support the currently available hardware; they were designed in such a way that developers can take advantage of new hardware benefits when they are introduced. This shifts the burden of providing support for new hardware from the developers to the hardware manufacturers themselves. In many cases, when users purchase some new hardware device, all that is required for their existing DirectX games to take advantage of new benefits is to download the latest DirectX drivers. For example, when MMX technology was introduced, DirectX drivers were updated to support the new features. If a user downloaded and installed the new DirectX drivers, existing games would immediately and transparently take advantage of the new technology.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Benefits and Drawbacks of DirectX Programming**

For the most part, DirectX delivers exceptionally on all of its promises. It does indeed give developers a high-performance interface to system-level hardware while maintaining the benefits of the Windows operating system. DirectX offers the game programmer many benefits over using GDI functions, indeed many benefits that weren't even available under DOS. Even so, there are a few drawbacks to using DirectX over GDI, especially in the Delphi environment. While none of the drawbacks are truly serious, Delphi developers should be aware of both the advantages and disadvantages of using DirectX over traditional Windows programming techniques.

**Benefits**

**Speed** As we've already discussed, DirectX allows the game developer to squeeze the maximum performance out of applications. Performing just about any game application task is substantially faster than employing similar GDI functions. In fact, DirectX offers some functionality that would be very difficult and time consuming to emulate using the multimedia API.

**Greater Control** DirectX gives applications incredible control over some things that would be impossible, impractical, or difficult under GDI. Such tasks as direct memory access to bitmapped graphics or audio data (in either hardware or system memory) and full control over the color palette (without the annoying GDI remapping of colors) are now possible through DirectX.

**Feature Emulation** Many features for which hardware acceleration is unavailable are emulated in DirectX. This provides the developer with a certain minimal framework of functionality that is guaranteed to exist with or without hardware support. While this may result in some loss of performance, software emulation of hardware features is implemented using very low-level multimedia API functions and assembly language, and in many cases is almost as fast as the hardware implementation. Most of the standard features have a software emulation, such as copying image data from one memory area to another or mixing sound buffers on the fly.

**Drawbacks**



**Difficult Debugging** DirectX applications are simply harder to debug than traditional Windows applications. Most high-performance games are written in a full-screen video mode, and in a full-screen video mode, you can't set breakpoints and step through your code. There are other debugging techniques you can use to get around this limitation, but once an application initializes a full-screen video mode, most of Delphi's awesome debugging features are completely useless.

**Few Delphi Examples** DirectX has, until recently, been solely the domain of C/C++ programmers. As such, most examples are written in C/C++. All of the DirectX documentation is certainly written for C/C++. Unfortunately, Delphi programmers may be required to learn at least a nominal amount of C/C++ before taking advantage of the DirectX documentation and numerous DirectX books and references that are available. Fortunately, several companies and individuals have written numerous DirectX libraries for Delphi, and there are several Delphi-specific DirectX programming web sites now available. Hopefully, this will result in an increase of DirectX documentation and references for the ambitious Delphi game programmer.

**Steep Learning Curve** Even in its most recent incarnation, DirectX is still a fairly new API. As such, one must usually perform several steps in order to perform certain actions, especially when initializing many of the DirectX subsystems. The programmer is usually required to perform these steps in a very specific order, and unless the programmer is very familiar with how certain DirectX functionality is implemented, it may be very difficult to get things working initially. This, coupled with the fact that the DirectX API is huge and there are few Delphi examples, usually results in a rather steep learning curve for first-time game programmers.

## The DirectX Architecture

The DirectX API itself, in the form of various COM objects and their methods, acts as an interface into the DirectX drivers. On the back side, the DirectX drivers can be thought of as two separate systems: the hardware abstraction layer (HAL) and the hardware emulation layer (HEL).

---

**HAL:** hardware abstraction layer; the DirectX drivers responsible for direct interface to hardware capabilities.



**HEL:** hardware emulation layer; the DirectX drivers responsible for emulating certain functionality if not present in hardware.

---

The HAL represents the hardware itself. When a DirectX object is created, it interrogates the HAL about hardware capabilities. The HEL, on the other hand, represents the software emulation of certain features. If an application asks a DirectX object to perform a certain task, it will be performed in hardware if the HAL indicates that the capability exists. Otherwise, the task will be performed in software if the HEL supports the emulated feature. Some features would be very slow to emulate in software, and thus may not be available in the HEL. These features will simply be unavailable to the application unless the HAL indicates they are present in the hardware. The HAL provides developers with hardware acceleration when available, and the HEL fills in where it can to provide a consistent, base set of features.

It is important to note that, for the most part, this is completely transparent to developers. Unless a developer is using a feature that is specifically implemented in hardware and not emulated by the HEL (such as overlays), an application need not be concerned if hardware support is present or not. However, an application can query DirectX to see if specific features are implemented in hardware or are emulated. An application can then use this knowledge to optimize various game functions, taking advantage of those functions that are specifically supported by the hardware and using a different approach for those functions that are too slow when emulated. This allows game developers to write applications that take advantage of available hardware under various configurations while still functioning adequately under all configurations.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## DirectX Components

DirectX is comprised of many components. Indeed, it seems as if new DirectX components are added every day, and there are sure to be more and more in subsequent versions. While the inventive game developer could potentially make use of all of these DirectComponents, there are only a few that are essential for game development. In this book, we will specifically look at the DirectDraw, DirectSound, and DirectInput components.

### DirectDraw

The DirectDraw component is responsible for all graphical output to the screen. DirectDraw objects provide functionality for such tasks as tear-free animation using page flipping techniques, access to graphical coprocessors, display memory management, and color palette manipulation. It is the core component of DirectX, and serves as the foundation for other DirectX components, such as Direct3D.

### DirectSound

The DirectSound component is responsible for digitized audio playback. It provides stereo and 3-D sound functionality and sound card memory management. It allows the developer to asynchronously play any number of sounds, mixing them on the fly and automatically converting their format to that of the primary sound output stream.

### DirectInput

The DirectInput component is responsible for providing the application with data retrieved from any type of input device. This not only includes the mouse and keyboard, but joysticks, flightsticks, rudder pedals, steering wheels, and other devices that have yet to be invented. DirectInput also supports output devices, such as force feedback peripherals.

### Other DirectX Components



There are several other DirectX components that are of particular interest to game developers, but are not quite as essential as the previous three. Several of these components may be required depending on the type of game being produced, and certainly any game could be enhanced by taking advantage of the functionality represented by these components. This is not an exhaustive list by any means, but game developers may want to further investigate the following components:

**DirectPlay** – Provides transport-independent messaging services between multiple computers. This is an essential component for multiplayer games, and includes functionality for organizing and launching a multiplayer game session.

**DirectSetup** – Provides automated installation of DirectX drivers.

**Direct3D** – Provides three-dimensional graphics rendering functionality. This component is segregated into low-level and high-level interfaces, which provide varying levels of control and expedience.

**DirectMusic** – Provides playback of MIDI music compositions. This component allows developers to directly control the exact sound of each instrument, and includes the ability to mix separate MIDI compositions on the fly to achieve context-sensitive musical transitions.

## DirectX and Delphi

Now that we know what DirectX is, we need to know what is required in order to use DirectX under the Delphi environment. Unfortunately, Delphi programmers do not enjoy the automatic access to DirectX resources that C/C++ developers do. If Microsoft were to include a type library along with the DirectX DLLs, it would be a simple task indeed to import the type library and immediately begin using the DirectX COM objects. Alas, this is not the case, and accessing DirectX functionality is the first obstacle that enthusiastic Delphi game developers must face.

### The DirectX SDK

As a C/C++ developer, you would simply need to purchase the DirectX SDK from Microsoft and install its accompanying files to begin developing DirectX applications. As a Delphi developer, however, there is no need to purchase the DirectX SDK. All that is required is the DirectX redistributable files available for free download from Microsoft's DirectX web site, or from the many commercial games that include them.

However, it may be worth your while to purchase the DirectX SDK, as it includes many benefits. The most important thing you will gain by purchasing the SDK is the DirectX documentation. As this book is not a reference manual, it will not discuss many of the nuances of DirectX programming that are available in the hundreds of pages of DirectX documentation. Dozens of examples are also available in the SDK which, although written in C/C++, are valuable references for investigating some of DirectX's more powerful features. The SDK also includes a DirectX control panel applet that allows you to control numerous global settings that may be helpful during debugging.

Perhaps most importantly, the DirectX SDK includes information on the files that can be distributed with your applications. While you can always suggest that your users visit the Microsoft web site to download the DirectX files if they do not already have them installed, it is a wise practice to always include all required files and DLLs with your applications. Therefore, you will need to be aware of the many legal requirements for bundling the DirectX redistributables with your applications. For more information about the DirectX SDK, check out <http://www.microsoft.com/directx/resources/devdl.htm>. You can order the DirectX SDK for a nominal fee by visiting <http://www.microsoft.com/directx/sdkcdorder.asp>.

### Project JEDI

If you have the DirectX end user redistributable files installed (which you probably do if you've played any recent games), all you will need are the Delphi import files converted from the original DirectX C/C++ header files. All code in this book is based around the Project JEDI DirectX header files written by Erik Unger. Project JEDI (Joint Endeavor of Delphi Innovators) is a nonprofit organization composed of Delphi

programmers all over the world. Their goal is to convert C/C++ header files for various APIs into Delphi import units, thus giving Delphi developers everywhere access to the same programming resources as their C/C++ cousins.

Erik's header files are a raw conversion of the original C/C++ headers, supporting DirectX 6 under Delphi 4. This gives Delphi developers the exact same functionality as would be available under C/C++. This is not a library that automates certain tasks; it is a hard-core unit containing the same low-level objects and methods that the C/C++ programmers use. This book is aimed at teaching the reader how to program DirectX, not how to use some arbitrary DirectX library. Thus, these units were the perfect choice. Indeed, after reviewing several of the DirectX examples used throughout this book, you should be able to comprehend most C/C++ examples, as the DirectX method calls are the same. This book, and the companion CD, contain the latest version of Erik's header files at the time of publication. However, to get more information about the header files, some examples, and the latest version, check out <http://www.delphi-jedi.org/DelphiGraphics/jedi-index.htm>.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Erik Unger's headers give Delphi developers access to DirectDraw, DirectSound, DirectInput, DirectPlay, DirectSetup, and Direct3D. This is the full functionality available under all of these DirectX subsystems, including force feedback and 3-D sound. The DXTools unit accompanying the files also includes several utility functions, such as testing the return results of DirectX methods and performing matrix manipulations. Throughout the course of this book, we will develop a utility unit converted from a C/C++ utility unit shipping with the DirectX SDK that simplifies several menial tasks such as loading a bitmap, a WAV file, or a palette. Later in this chapter, we will begin to use our utility unit and Erik's header files to create a baseline DirectX application from which all other examples in this book are based.



**Tip:** While the latest headers at the time of the writing were used for all of the book's examples, it is quite possible that the headers will have gone through several more versions by the time this book goes to print. The examples in this book will compile with the headers included on the CD, but you should get the latest headers from the JEDI page to have access to all the newest features. Bear in mind that the book examples may require a little adjustment when used with the new headers, as method declarations may have changed.

## Other Components and Headers

Fortunately, the Delphi gaming community is starting to see a rise in the number of available DirectX headers and components. Some are freely available, others you must purchase, but all of them give the Delphi developer a little more functionality than the raw headers accompanying this book.

## DelphiX

A freeware component suite written by Hiroyuki Hori, DelphiX comes complete with source code and gives the developer access to DirectDraw, Direct3D, DirectSound, DirectInput (including force feedback support), and DirectPlay. There are several cool components in this suite which allow the developer to bundle graphics and sound into compressed libraries, and includes some awesome alpha blending functionality. The examples clearly demonstrate some of the advanced techniques capable with DelphiX, and there are several freeware tools available that make editing sound and graphics libraries easy. For more information, check out <http://www.ingjapan.or.jp/hori/indexe.html>.

## The Delphi Games Creator (DGC)

Another freeware component suite, the DGC gives developers access to DirectDraw, DirectSound, DirectPlay, and DirectInput. Written by John Pullen, Paul Bearne, and Jeff Kurtz, the DGC comes complete with source code. The DGC also comes with editors for bundling sound and graphics into libraries, and will map all images in a graphics library to a single palette. Palette cycling and cool transition effects are just some of the added functionality available with this component suite. It also includes a scrolling map component and editor for making tile-based map engines. One of the first and oldest freeware DirectX component suites, the DGC has a large following, and is suitable for private or commercial DirectX applications. For more information, check out <http://www.ex.ac.uk/~PMBearne/DGC.html>.

## RingZero GDK

Created by Systems Advisory Group Enterprises, Inc., the RingZero GDK and RingZero 3D are component suites that give the Delphi developer access to DirectDraw, DirectSound, DirectPlay, DirectInput, and Direct3D. Although these component suites must be purchased, you gain the benefit of technical support and continuing updates (which may or may not always be available with freeware components). Source code is also available. These component suites are very easy to use, and can expedite the creation of a DirectX game if you are impatient or have a pressing deadline. Unfortunately, as of this writing, these components only support DirectX 5, and can only be used under Delphi 2 or 3. However, by the time this book reaches store shelves, RingZero may be compatible with Delphi 4 and DirectX 6. For more information, check out <http://www.sage-inc.com>.

## A DirectDraw Tutorial

DirectX has hopefully been installed to your machine by now, and we have the units required to give us access to DirectX components and functionality. At this point, we can begin exploring the world of DirectX programming and all of its nuances, pitfalls, and rewards.

DirectDraw is the core, most fundamental DirectX component, and as such, is a good starting point for a discussion of DirectX programming. We will go over a good number of DirectDraw concepts, many of which are mirrored in other DirectX components. Several of the methods called and tasks performed by the DirectDraw objects are very similar, if not exact, to those of other DirectX components, such as DirectSound and DirectInput. The learning curve discussed earlier starts here, as there are quite a few concepts to explain.

### Features

DirectDraw offers many features above and beyond what is available through the Win32 API. As with all DirectX components, DirectDraw gives the developer more control over graphical output than the Windows API, while providing incredible performance not attainable using GDI functions. DirectDraw provides the following features and many more for powerful Windows game programming:

- Direct access to image memory, whether located in system or display memory
- The ability to specify the location of surface memory (i.e., system or display memory)
- Hardware acceleration for many graphical functions
- Screen resolution and color depth changing on the fly
- Direct control over the system palette (in full-screen mode)
- Powerful page-flipping methods for incredibly fast animation

In the following sections we will see how DirectDraw is used to change the display mode and color depth, how bitmaps are loaded into usable DirectDrawSurfaces, and how page-flipping animation is implemented.

### Functional Overview

DirectX programming primarily consists of creating parent objects, namely objects that represent specific DirectX components, and then using these objects to create child objects that represent a specific

functionality. The following illustration and discussion outline actions that will become familiar when we cover other DirectX components.

DirectDraw programming consists of creating a DirectDraw object and several DirectDrawSurface objects. The DirectDraw object itself acts as a liaison between the application and the display driver. Through the DirectDraw object, we can change the display resolution and color depth as well as create individual DirectDrawSurface and DirectDrawPalette objects.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

**7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Individual DirectDrawSurface objects represent display buffer memory. This can be either offscreen storage for image data or the visible display itself. The application uses these surface objects to communicate directly with the video hardware, either through direct memory access of image data or methods that access hardware acceleration. Surfaces provide methods to copy image data to and from each other, performing various operations on the image data such as scaling or ignoring transparent pixels. DirectDrawSurface objects encapsulate much of the power provided by DirectDraw.

DirectDrawPalette objects hold color data for images that are stored in a palettized format. They also represent the application's interface directly to the display hardware's palette. DirectDrawPalette objects will be covered in more detail in the next chapter.

### COM Fundamentals

DirectX objects are COM based, consequently involving all of the associated benefits and drawbacks of using COM objects. Microsoft's Component Object Model (COM) is a huge beast, not unlike that of the VCL. A full-fledged discussion of COM is beyond the scope of this book, and unnecessary for the purposes of DirectX programming. However, there are a few COM theories that we must understand before delving into the heart of DirectX programming.

Working with DirectX objects has some similarities to working with other COM objects, but there are a few differences. The primary difference that most experienced COM programmers will discover is the fact that DirectX objects do not need to be created by using the CoCreateInstance function. Other COM objects frequently used in a traditional COM application, such as IClassFactory, are also unnecessary. All DirectX objects are created through custom function calls that return an instantiated COM object. For example, the DirectDrawCreate function is used to create an instantiated IDirectDraw object (more on this later). If you've never worked with COM objects before, this takes some of the confusion out of the loop, and is similar to creating traditional VCL objects in code.

A COM object can almost be thought of as a VCL object with no properties, only methods. The methods of a COM object are known as an interface. A COM object can have more than one interface, and it can be extended by added new interfaces, similar to deriving a new component from an ancestor. However, the COM standard dictates that an interface can never change. The result of this is that older code is guaranteed



to work with newer COM objects, as the original interfaces will still be implemented and available. For this reason, you will see several different types of interface declarations in the DirectX header files. As each new version of DirectX comes out, offering new functionality, an entirely new interface must be created to support this functionality. However, you can rest assured that your DirectX game will still work when DirectX 7, 8, 9, etc., are introduced in the future.

An interesting feature implemented by COM allows applications to query a COM object to see if it implements a particular interface. This allows your application to be extremely flexible when implementing certain features. For example, force feedback is a newer addition to DirectInput functionality. Your application, upon startup, could query the DirectInput objects to see if they support the interface required to implement force feedback output. If they do not, indicating that the game is running on a machine with an older version of DirectX, you could simply bypass the part of your code that implemented force feedback. However, if the force feedback interface is available, your application can make full use of its features. In this way, game applications can be very robust, taking advantage of special features only on those machines that have them available, while still running acceptably on older machines without those features. We will see an example of this later in this chapter when we discuss how to detect what version of DirectX the user has installed.

## Reference Counting

One fundamental concept in COM is the idea of lifetime management. COM was implemented with the idea that several users, or clients, could potentially use the exact same COM object, thus saving memory and resources. Therefore, the COM objects themselves implement a method of reference counting. Essentially, COM objects keep a count of how many times they are being used. When they are created, this count is initialized to 1. When a new client wishes to use the COM object, this reference count is incremented. When a client no longer needs the COM object and it goes out of scope, such as when an application terminates, the reference count is decremented. When the reference count reaches 0, the object knows it can free itself.

This reference count incrementing and decrementing is accomplished by calling the `AddRef` and `Release` methods implemented in all COM objects. However, fortunately for Delphi developers, Delphi takes care of this automatically and, for the most part, transparently. Reference counting does add some esoteric nuances to using COM objects that are not present when using VCL objects. As a Delphi DirectX developer, don't worry about explicitly freeing DirectX objects after they are created; they will free themselves when they go out of scope or when the application terminates. For more information on general COM programming concepts, look at *Delphi 4 Developer's Guide* by Teixeira and Pacheco (ISBN 0-672-31284-0) or *Delphi Developer's Handbook* by Cantu, Lam, and Gooch (ISBN 0-782-11987-5).

## GUID

COM objects are identified by a unique number known as a GUID, pronounced "goo-id" or "gwid" (like "squid"), which stands for globally unique identifier. Without going into too much detail, Microsoft provides an API call that generates a GUID that is completely unique; it is guaranteed to be unique if the machine on which the GUID was created has a network card installed. The concept of what a GUID is or how it is used is only important to the DirectX developer in that GUIDs are used to identify many individual objects and devices, such as specific DirectX objects (such as memory surfaces or sound buffers) and specific drivers (such as a particular video driver or device). We will be using GUIDs throughout every example in the book.

## Surfaces

Perhaps the single, most important concept in DirectDraw programming is that of the surface. A surface is directly analogous to Delphi's `TBitmap` or `TCanvas` object. It represents a virtual "picture" upon which graphical output can be drawn. In reality, a surface is a linear chunk of memory that is managed as if it were rectangular. Surfaces have width and height, among many other properties. The memory for a surface can be located in a number of places, but primarily it will be found in system memory or in the memory on the display card itself.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**IT KNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

There are many, many different types of surfaces, some of which are very specialized for a particular purpose. Surfaces can be used specifically for 3-D rendering, or for storing textures used in 3-D rendering; they can be set up to receive live video feeds, or perhaps to store only transparency information (known as the alpha channel). For the purposes of this book, we will be concerned with only three different types: the primary surface, the backbuffer surface, and off-screen surfaces.



**alpha channel:** A component of color that records levels of transparency/opacity as opposed to actual color data.

## The Primary and Backbuffer Surfaces

Technically speaking, these are really the same type of surface. The primary surface represents the image that the user sees on the display screen. Anything drawn onto the primary surface is immediately displayed. The backbuffer surface has the same capabilities and format of the primary surface, and is used in an animation technique known as page flipping (more on this below).

## Off-screen Surfaces

An off-screen surface, as it will be used in this book, is simply a surface in memory that is never directly displayed on the screen. Typically the same format as the primary and backbuffer surfaces, off-screen surfaces are generally used for storing background graphics and sprite animation frames. They can also be used as work surfaces for composing a graphical image. The image in these surfaces is then copied onto the primary or backbuffer surfaces to be displayed. We'll see an example of this later in the chapter.

## Page Flipping

Page flipping is an incredibly powerful technique for producing flicker-free animation. The term "page flipping" comes from the inspiration for this animation technique—that of traditional cartoon animation. Cartoons (and movies, for that matter) create the illusion of animation by displaying to the viewer a series of

images, one after the other. Someone can create a crude animation by taking a stack of paper, drawing an image on each page, and then flipping through it rapidly. This is the architecture of page flipping animation. The idea is to create two surfaces, the primary surface and the backbuffer surface. The next frame of animation is composed on the backbuffer surface, and then it is “flipped” to the display screen. This updates the entire display screen in one fell swoop, resulting in a flickerless animation.



**Figure 4-1:** Page flipping animation

This page flipping animation architecture is incredibly easy to implement under DirectX. Just about every video card available supports this in hardware, making it incredibly fast. Therefore, we will use this page flipping technique for nearly every animation and game example throughout the book. We will see how to initialize and implement page flipping in the examples to follow.

## Resolution, Color Depth, and Video Memory

We discussed resolution, color depth, and video memory quite a bit in Chapter 3, Basic Graphics Programming. That discussion is just as relevant to DirectDraw programming as it was to GDI graphics programming. However, under DirectDraw, the developer has much more control over the resolution and color depth. This also affects how the memory represented by a surface is directly accessed. To simplify matters, we will be concerned only with 256-color palettized display modes. A number of different display modes may be available under DirectX, and we’ll demonstrate how to retrieve a list of the available display modes shortly.

## Windowed Versus Full-Screen

Perhaps the most powerful aspect of DirectDraw programming is that the application can have control over the entire display screen, known as a full-screen application. However, DirectDraw also allows a developer to take advantage of DirectDraw capabilities for drawing graphics in a regular window while sharing desktop space with other windows. This is known as windowed mode. Generally speaking, it is more difficult to support a windowed DirectDraw application than it is to support a full-screen mode. A windowed DirectDraw application requires a DirectDrawClipper object (not covered in this book), and must handle a number of other requirements typical of a traditional GDI graphical application, such as dealing with a shared palette. We are concerned only with high-performance game programming, so this book will only demonstrate full-screen DirectDraw applications. This will greatly simplify the code, and will provide us with the greatest amount of control and power. The distinction between windowed and full-screen is known as the cooperative level, and will be covered in more detail below.

## Initializing DirectDraw

We’ve covered several COM and DirectDraw programming concepts, so now it’s time to see them in action. There are several tasks the application is required to perform before DirectDraw can be instantiated and the display mode modified. Some of these tasks are optional, some are not. DirectDraw, and DirectDraw surfaces, are instantiated by performing the following:

- Enumerating display devices
- Creating the DirectDraw object
- Enumerating the supported display modes
- Setting the cooperative level
- Setting the display mode
- Creating surfaces

## Enumerating Display Devices

While it is safe to assume that most machines only have one display device attached, the latest version of Windows does allow multiple monitors and other display devices (such as virtual reality headsets) to be installed simultaneously. Although it is not strictly necessary, it may be good practice to allow the user to pick the display device on which to display the DirectDraw application. However, for the examples in the book, we will only be concerned with the primary display device.

The function `DirectDrawEnumerateEx` is used to list all available display devices. It is a callback function that will call the specified function once for each installed display HAL it finds. The first device enumerated will be the primary display device; any others attached to the system will follow. The `DirectDrawEnumerateEx` function is declared as:

```
function DirectDrawEnumerateEx(  
  
    lpCallback: TDDEnumCallbackEx; // the address of the callback function  
  
    lpContext: Pointer;           // an application-specific value  
  
    dwFlags: DWORD                // scope flags  
  
): HRESULT; stdcall;           // returns a DirectX error code
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

The first parameter is the address of the function that is called for each enumerated device. The second parameter is a pointer to an application-defined value, such as a pointer to a TList or perhaps just a variable, and can be set to nil if not needed. The last parameter is a series of flags that control the scope of the enumeration. The callback function itself is defined as:

```
function TDDEnumCallbackEx(  
  
    lpGUID: PGUID; // a pointer to the device GUID  
  
    lpDriverDescription: PAnsiChar; // the device description  
  
    lpDriverName: PAnsiChar; // the device name  
  
    lpContext: Pointer; // the application-specific value  
  
    Monitor: HMonitor // a handle to the monitor  
  
): BOOL; stdcall; // returns TRUE or FALSE
```

The first three parameters of this function are the most interesting. The lpGUID parameter will contain the GUID for the specific device. This GUID must be stored, as it must be used later for creating the DirectDraw object. The lpDriverDescription and lpDriverName parameters contain the textual description and name of the enumerated device. These can be presented to the user in a list of devices, allowing them to choose the desired display object. The lpContext parameter simply contains the pointer passed to the initial DirectDrawEnumerateEx function. The last parameter will be zero in most cases, and can be ignored. The callback function must return a value of TRUE to continue the enumeration; returning FALSE signals the function to stop enumerating devices.

The following example demonstrates enumerating all available display devices.

*Listing 4-1: Enumerating display devices*

implementation

uses

DDraw;

```
function EnumDevices(lpGUID: PGUID; lpDriverDescription: PAnsiChar;
```

```
    lpDriverName: PAnsiChar; lpContext: Pointer;
```

```
    Monitor: HMonitor): BOOL; stdcall;
```

begin

```
    {add the name and description of the enumerated device. here, we would  
    need to store the GUID for the device so it would be available when  
    the user selected the desired display device}
```

```
TStringList(lpContext).Add(lpDriverName+' - '+lpDriverDescription);
```

```
    {continue enumeration}
```

```
    Result := TRUE;
```

end;

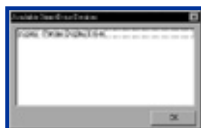
```
procedure TForm1.FormCreate(Sender: TObject);
```

begin

```
    {enumerate all attached display devices}
```

```
    DirectDrawEnumerateEx(EnumDevices, lbxDevices.Items, 0);
```

end;



**Figure 4-2:** The list of attached display devices

## Creating the DirectDraw Object

The first truly required step in initializing a DirectDraw application is to create the DirectDraw object itself. To create the DirectDraw object, call the DirectDrawCreate function, which is defined as:

```

function DirectDrawCreate(
lpGUID: PGUID; // the GUID of the DirectDraw device
out lpLpDD: IDirectDraw; // the IDirectDraw object to be instantiated
pUnkOuter: IUnknown // unused, set to nil
): HRESULT; stdcall; // returns a DirectX error code

```

The first parameter takes the GUID of the selected device. It can be set to nil to create a DirectDraw object associated with the primary display device, or it can be set to a GUID as retrieved from device enumeration above. Additionally, it can be set to DDCREATE\_HARDWAREONLY or DDCREATE\_EMULATIONONLY to only use the HAL or HEL, respectively. This allows the developer to perform testing and debugging against emulated or hardware accelerated features. The second parameter takes a pointer to the IDirectDraw object. When this function returns, this object will be instantiated. The last parameter is used for COM aggregation, and must be set to nil as it is currently unsupported.



**Caution:** It is possible to create multiple DirectDraw objects, but this is inadvisable. Each DirectDraw object is independent of the others, and surfaces and other child objects cannot be shared between them.

Now, this function will only return an instantiated IDirectDraw object, but since we are concerned with DirectX 6 functionality, we need to get an instantiated IDirectDraw4 object. Thus, we should create a temporary IDirectDraw object, and once this is instantiated, we can use the QueryInterface method of the object to retrieve an instantiated IDirectDraw4 object. The temporary object can then be released by setting it to nil. We'll illustrate the creation of an IDirectDraw4 object below.



**Note:** Delphi's AS operator can be used to accomplish the same thing. However, use of the QueryInterface method is a little less abstract.

## Enumerating the Supported Display Modes

It is typically much harder to create an application that supports multiple display modes. As discussed in the previous chapter, changing display modes modifies the size of graphics as they are displayed on screen, and changes the amount of screen area that must be filled with graphics. This puts an additional requirement on the programmer to determine how graphics will be displayed in different modes (if they are to be scaled, stretched, etc.), and may put additional requirements on artists if they must develop additional graphics to support different resolutions and color depths. However, allowing the user to choose which display mode works best on their system is a good practice, as the user can then tune the game to achieve acceptable speeds. Even if your game will support only one graphic mode, it is useful to determine if the machine supports this mode. Our examples will be concerned with only one specific display mode in order to simplify the code, but we still need to enumerate the available display modes to ensure that the machine can handle the application.

Enumerating display modes is similar to enumerating display devices. After the IDirectDraw4 object is created, we can call its EnumDisplayModes method to enumerate all display resolutions and color depths supported by the display driver. The EnumDisplayModes method is defined as:

```

function EnumDisplayModes(
dwFlags: DWORD; // scope flags
lpDDSurfaceDesc: PDDSurfaceDesc2; // a surface description structure

```

```
lpContext: Pointer; // an application-defined value
lpEnumModesCallback: TDDEnumModesCallback2 // the callback function
): HRESULT; // returns a DirectX error code
```

The first parameter is a series of flags that determine how the display modes are enumerated, and can be set to zero. The second parameter is a pointer to a TDDSurfaceDesc2 structure that contains information defining the types of display modes to be enumerated. We can use this to enumerate only specific types of display modes, such as all 800 x 600 modes or all palettized display modes. Setting this parameter to nil will enumerate all supported display modes. The third parameter is a pointer to an application-defined value, and the last parameter is a pointer to the callback function itself. The callback function is defined as:

```
function TDDEnumModesCallback2(
const lpDDSurfaceDesc: TDDSurfaceDesc2; // a surface description record
lpContext: Pointer // the application-defined value
): HRESULT; // returns a DirectX error code
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

The first parameter contains a TDDSurfaceDesc2 structure initialized with information about the enumerated display device. This includes information such as the width, height, and color depth. The second parameter contains a pointer to the application-defined value. This function must return DDENUMRET\_OK to continue the enumeration; DDENUMRET\_CANCEL will stop the enumeration.

The TDDSurfaceDesc2 object is complex, and is used in a number of different IDirectDrawSurface4 methods. The TDDSurfaceDesc2 structure is defined as:

TDDSurfaceDesc2 = packed record

```
    dwSize: DWORD; // size of the TDDSurfaceDesc structure

    dwFlags: DWORD; // flags indicating valid members

    dwHeight: DWORD; // surface height

    dwWidth: DWORD; // surface width

    case Integer of

    0: (

        lpitch : LongInt; // distance to start of next row

    );

    1: (

        dwLinearSize : DWORD; // optimized surface size

        dwBackBufferCount: DWORD; // number of backbuffers

    case Integer of

    0: (
```



```

    dwMipMapCount: DWORD;           // number of mip-map levels
    dwAlphaBitDepth: DWORD;         // depth of alpha buffer
    dwReserved: DWORD;             // reserved
    lpSurface: Pointer;            // pointer to surface memory
    ddckCKDestOverlay: TDDColorKey; // color key for destination overlays
    ddckCKDestBlt: TDDColorKey;     // color key for destination blt
    ddckCKSrcOverlay: TDDColorKey;  // color key for source overlays
    ddckCKSrcBlt: TDDColorKey;     // color key for source blt
    ddpfPixelFormat: TDDPixelFormat; // pixel format description
    ddsCaps: TDDSCaps2;            // surface capabilities
    dwTextureStage: DWORD;         // multitexture cascade stage
);

1: (
    dwRefreshRate: DWORD;          // refresh rate
);
);
end;

```

The following listing illustrates creating an `IDirectDraw4` object and enumerating all supported display modes.

***Listing 4-2: Creating `IDirectDraw` and enumerating display modes***

---

```

var
    Form1: TForm1;
    FDirectDraw: IDirectDraw4;

implementation

function EnumModesCallback(const lpDDSurfaceDesc: TDDSurfaceDesc2;
                           lpContext: Pointer): HRESULT; stdcall;
begin
    {list the height, width, and color depth of the enumerated display mode}
    TStringList(lpContext).Add(IntToStr(lpDDSurfaceDesc.dwWidth)+' X '+

```

```

        IntToStr(lpDDSurfaceDesc.dwHeight)+' , '+
        IntToStr(lpDDSurfaceDesc.ddpfPixelFormat.dwRGBBitCount)+
        bits/pixel');

    {continue enumeration}

    Result := DDENUMRET_OK;

end;

procedure TForm1.FormCreate(Sender: TObject);

var

    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
    need a temporary interface}

    TempDirectDraw: IDirectDraw;

begin

    {create a temporary DirectDraw object. this is used to create the
    desired DirectDraw4 object}

    DirectDrawCreate(nil, TempDirectDraw, nil);

    try

        {we can only get a DirectDraw4 interface through the QueryInterface
        method of the DirectDraw object}

        TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw);

    finally

        {now that we have the desired DirectDraw object, the temporary DirectDraw
        object is no longer needed}

        TempDirectDraw := nil;

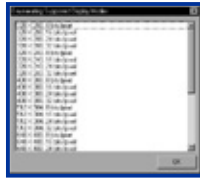
    end;

    {begin enumerating supported display modes}

    FDirectDraw.EnumDisplayModes(0, nil, lbxDisplayModes.Items, EnumModesCallback);

end;

```



**Figure 4-3:** Supported display modes

## Setting the Cooperative Level

As discussed earlier, the cooperative level determines if the application runs in full-screen mode or if it will share the desktop with other windows. Basically, it determines how the application will work with other applications. The concept of cooperative level is also found in other DirectX components.

Setting the cooperative level requires a simple call to the `IDirectDraw4`'s `SetCooperativeLevel` method, which is defined as:

```
function SetCooperativeLevel  
  
    (hWnd: HWND;                // a handle to the main window  
  
     dwFlags: DWORD             // flags controlling the cooperative level  
  
    ): HRESULT;                // returns a DirectX error code
```

The first parameter is a handle to the application's main window. The second is a series of flags that specify the actual cooperative level. For our discussion, we should pass `DDSCL_EXCLUSIVE` and `DDSCL_FULLSCREEN`, which indicates that our application will be responsible for maintaining the entire display. We'll see an example of its use below.

## Setting the Display Mode

Here's where the real action takes place. We've created the `IDirectDraw4` object, allowed the user to select a display mode or determined that our predefined display mode is supported, and we've set the cooperative level. All that's left is to actually change the display mode itself. This is accomplished through `IDirectDraw4`'s `SetDisplayMode` method. The `SetDisplayMode` method is defined as:

```
function SetDisplayMode(  
  
    dwWidth: DWORD;            // the desired width, in pixels  
  
    dwHeight: DWORD;          // the desired height, in pixels  
  
    dwBPP: DWORD;             // the color depth, in bits  
  
    dwRefreshRate: DWORD;     // the refresh rate  
  
    dwFlags: DWORD            // control flags  
  
    ): HRESULT;               // returns a DirectX error code
```

The first and second parameters specify the desired width and height of the display, in pixels, and it must match a supported display mode as returned by the `EnumDisplayModes` function. The third parameter indicates the number of bits that describe an individual pixel's color, and again this must match a supported display mode. The refresh rate and flags parameters can both be set to zero. When this function is called, the display mode is immediately switched to the mode specified. The screen will likely go blank, you'll hear some clicking, and then the display hardware will reset to

the specified mode.

Now, before we can set the display mode, DirectDraw makes some demands of the main window that must be met before either the SetCooperativeLevel or the SetDisplayMode methods will work. According to the DirectX documentation, the window whose handle is sent to the SetCooperativeLevel method must not have any border, and must be a stay-on-top window. This can be accomplished in Delphi by setting the form's BorderStyle to bsNone and the FormStyle to fsStayOnTop in the form's OnCreate event. Only then will the SetDisplayMode method work correctly.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

The following example demonstrates setting the cooperative level and switching the display mode.

*Listing 4-3: Setting the display mode to full screen*

```
var  
  
    Form1: TForm1;  
  
    FDirectDraw: IDirectDraw4;  
  
implementation  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    {initialize form properties.  note that the FormStyle property must be  
    set to fsStayOnTop}  
  
    BorderStyle := bsNone;  
  
    BorderIcons := [];  
  
    FormStyle := fsStayOnTop;  
  
end;
```

```

procedure TForm1.FormActivate(Sender: TObject);
var
    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
    need a temporary interface}
    TempDirectDraw: IDirectDraw;
begin
    {create a temporary DirectDraw object. this is used to create the
    desired DirectDraw4 object}
    DirectDrawCreate(nil, TempDirectDraw, nil);

    try
        {we can only get a DirectDraw4 interface through the QueryInterface
        method of the DirectDraw object}
        TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw);
    finally
        {now that we have the desired DirectDraw object, the temporary DirectDraw
        object is no longer needed}
        TempDirectDraw := nil;
    end;

    {set the cooperative level}
    FDirectDraw.SetCooperativeLevel(Handle, DDSCL_FULLSCREEN or DDSCL_EXCLUSIVE);

    {set the display resolution and color depth}
    FDirectDraw.SetDisplayMode(640, 480, 8, 0, 0);
end;

```



**Note:** You'll notice that the DirectDraw initialization code has been moved to the form's OnActivate event. We've found this is a more appropriate place to put initialization code, and the examples throughout the rest of the book will expound upon the code placed in this method.

---

The result of this example is a bit surprising. The form itself was still visible, albeit in a full-screen mode, and the mouse cursor was still able to manipulate objects on the form. This is possible because the GDI is still outputting graphical images, and by default the memory area to which the GDI is rendering is still visible when the DirectDraw application sets its display mode. This can be useful when a DirectDraw application needs to start up and display an interface to the user that is easily put together using regular Delphi controls.

## Page Flipping

At this point, we can instantiate DirectDraw and adjust the display hardware's display mode, but not much else. We still can't draw anything to the screen, making any sort of animation very difficult. We need to modify our DirectDraw code so it supports the page flipping architecture discussed earlier. In order to accomplish this, we must first create a primary surface and an attached backbuffer surface.

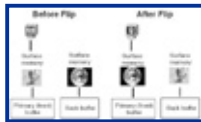
## Creating Surfaces

We will create the primary and backbuffer surfaces simultaneously, creating what is known as a complex surface. A complex surface is one surface with a series of separate surfaces that are implicitly attached to it. These surfaces are arranged in a linear fashion. Typically, page flipping applications create only two surfaces (the primary and one backbuffer). However, some applications may need three or more.



**Note:** The primary surface and all backbuffers must reside in video memory. Thus, the number of backbuffers that can be created is dependent upon the size of the surfaces, the color depth, and the available display memory.

The primary surface is always visible to the user. Animation is accomplished by drawing any graphical output to the backbuffer surface, which is offscreen. When the next frame of animation needs to be displayed, the backbuffer surface memory and the primary surface memory change places (or flip). Only the memory referenced by these two objects change; no other attributes or other attached objects are affected. Your BackBuffer object will always point to the backbuffer surface, and your PrimarySurface object will always point to the primary surface. This is illustrated below.



**Figure 4-4:** Page flipping the primary and backbuffer surfaces

We can create the primary and backbuffer surfaces by calling the IDirectDraw4 method CreateSurface. The CreateSurface method is defined as:

```
function CreateSurface(  
  
var lpDDSurfaceDesc: TDDSurfaceDesc2; // a surface description structure  
  
var lpIddsSurface: IDirectDrawSurface4; // the IDirectDrawSurface4 object  
  
pUnkOuter: IUnknown // unused, set to nil  
  
): HRESULT; // returns a DirectX error code
```

The first parameter is a data structure that defines the type of surface we wish to create. The second parameter is a pointer to the IDirectDrawSurface4 object that is instantiated by the function. The last parameter is used for COM aggregation and should be set to nil.

The TDDSurfaceDesc2 structure is a very complex data structure as we examined above, but for the purposes of this function, we are concerned only with a few members. To begin, we should initialize all members of the surface by calling FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0). We need to indicate that we are interested in the

capabilities and buffer count members of this structure, so we need to set its dwFlags member to DDSD\_CAPS or DDSD\_BACKBUFFERCOUNT (by ORing the values, we combine them). We then must set its ddsCaps.dwCaps member to DDSCAPS\_COMPLEX or DDSCAPS\_FLIP or DDSCAPS\_PRIMARYSURFACE. The DDSCAPS\_COMPLEX flag indicates we are creating a complex surface with implicitly attached surfaces, DDSCAPS\_FLIP indicates this will be a flipping structure, and DDSCAPS\_PRIMARYSURFACE indicates that the root surface in the flipping structure will be the primary surface. We must also set the dwBackBufferCount to indicate how many backbuffers we will need. In our case, we can set this to one. Finally, we must set the dwSize member to SizeOf(TDDSurfaceDesc2).



**Caution:** DirectX methods use lots and lots of very complex data structures. These data structures may vary from version to version, and some methods may use only specific members of any particular structure. It is imperative that any data structure with a member denoting its size must have this member initialized by calling SizeOf. Failing to perform these two requirements can lead to failed method calls.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

When this method returns, the IDirectDrawSurface4 object will point to the primary surface. However, we really want to perform all of our animation on the backbuffer surface, so we must instantiate an object to point to this surface. This is accomplished through the IDirectDrawSurface4's GetAttachedSurface method. When this method is called on a complex surface with implicitly attached surfaces, it instantiates an IDirectDrawSurface4 object to point to the specified surface. The GetAttachedSurface method is defined as:

```
function GetAttachedSurface(
var lpDDSCaps: TDDSCaps2;           // surface capabilities
var lpIAttachedSurface: IDirectDrawSurface4 // a surface object
): HRESULT;                          // returns an error code
```

The first parameter is a TDDSCaps2 structure whose dwCaps member must be set to DDSCAPS\_BACKBUFFER. The second parameter is an IDirectDrawSurface4 object that will be set to point to the backbuffer surface when the function returns. This function returns the first attached surface whose capabilities match those defined in the function call. In our case, we receive a pointer to our only backbuffer surface. The TDDSCaps2 structure is defined as:

```
TDDSCaps2 = packed record
    dwCaps: DWORD;           // capabilities flags
    dwCaps2 : DWORD;        // more capability flags
    dwCaps3 : DWORD;        // yet more capability flags
    dwCaps4 : DWORD;        // still more capability flags
end;
```

The use of the CreateSurface and GetAttachedSurface functions will be demonstrated below.

## Rendering with GDI

We've created a page flipping surface structure that will demonstrate DirectDraw's animation capabilities, but unless we actually draw something to the screen, we won't be able to tell if anything is happening when we actually flip the surfaces. Let's cover a cool DirectDraw and Delphi trick that will make some tasks incredibly easy.

The IDirectDrawSurface4 object contains a method called GetDC which returns a handle to a GDI device context. This allows us to use any GDI function call to render graphics, or text, directly onto the surface. If we create a TCanvas object and set its handle to the value returned by GetDC, we can take full advantage of all TCanvas methods. This is a very powerful Delphi technique that can make graphics programming incredibly simple. The GetDC method is defined as:

```
function GetDC(  
  
var lphDC: HDC // a variable that receives the DC  
  
): HRESULT; // returns a DirectX error code
```

The only parameter to this method is a variable that will receive the device context handle for the surface.

After we are through using the device context, we must release it before performing any DirectDraw methods that work on the surface, such as flipping or copying surface memory to or from the surface in question. We accomplish this by calling the IDirectDrawSurface4 method ReleaseDC, which is defined as:

```
function ReleaseDC(  
  
hDC: Windows.HDC // a variable containing the DC to release  
  
): HRESULT; // returns a DirectX error code
```

This method's only parameter must be set to the device context handle returned from the previous call to GetDC.



**Caution:** It is absolutely imperative that every call to GetDC be matched with a call to ReleaseDC. When a device context is retrieved, the normal execution of Windows is put on hold. Therefore, any code between GetDC and ReleaseDC must execute as quickly as possible, as messages and other system events could get backed up. Also, any other DirectDraw methods that work directly on the surface must not be called until the device context has been released. Therefore, it is good practice to use a try..finally block, putting the ReleaseDC call in the Finally part to ensure that the device context is released. Otherwise, your system will crash violently.

---

Any GDI function or TCanvas method can be used for rendering graphics to a surface. Bear in mind that you will be suffering from GDI's sluggishness, but you may be able to reuse proprietary GDI graphics rendering methods and leverage your former development investments. Specifically, we'll be using these techniques to perform text output in a number of examples. We will also examine how this makes the process of loading and displaying bitmaps incredibly simple.

## Flipping the Surfaces

OK, we've created our complex flipping structure, we've drawn some text on the front and backbuffer in order to differentiate them, now we're ready for action. To actually flip the contents of the surfaces, call the Flip method of the primary surface object. The Flip method is defined as:

```
function Flip(  
lpDDSurfaceTargetOverride: IDirectDrawSurface4; // surface to flip  
dwFlags: DWORD // wait flags  
): HRESULT; // returns an error code
```

Typically, the first parameter is set to nil as the default behavior is to flip to the next buffer in the chain. However, if there is more than one backbuffer, this parameter can be set to the particular buffer in order to flip it to the screen. The last parameter is set to either zero or DDFLIP\_WAIT. If it is set to zero, the function immediately returns, regardless if the flip operation could be set up. If the DDFLIP\_WAIT flag is specified, the function does not return until the flip operation occurs. Setting this parameter to zero allows the application to perform extra processing while waiting for the display hardware to set up the flip operation. However, the Flip method must continuously be called in order to set up the flip operation. If the flip operation could not be set up, this function will return DDERR\_WASSTILLDRAWING. In most cases, it is easiest to set this parameter to DDFLIP\_WAIT, which is the flag we will use in the examples throughout this book.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

The following example demonstrates the creation of the primary and backbuffer surfaces, rendering using GDI functions, and page flipping.

*Listing 4-4: A simple page flipping application*

```
var  
  
    Form1: TForm1;  
  
    FDirectDraw: IDirectDraw4;  
  
    FPrimarySurface,  
  
    FBackBuffer: IDirectDrawSurface4;  
  
implementation  
  
procedure TForm1.FormCreate(Sender: TObject);  
  
begin  
  
    {initialize form properties. note that the FormStyle property must be  
    set to fsStayOnTop}  
  
    BorderStyle := bsNone;  
  
    BorderIcons := [];
```

```

FormStyle := fsStayOnTop;

Color := clBlack;

{turn off the cursor}

ShowCursor(FALSE);

end;

procedure TForm1.FormActivate(Sender: TObject);

var

    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
    need a temporary interface}

    TempDirectDraw: IDirectDraw;

    {structures required for various methods}

    DDSurface: TDDSurfaceDesc2;

    DDSCaps: TDDSCaps2;

    {these variables are used in GDI rendering}

    TempCanvas: TCanvas;

    SrfcDC: HDC;

begin

    {create a temporary DirectDraw object. this is used to create the
    desired DirectDraw4 object}

    DirectDrawCreate(nil, TempDirectDraw, nil);

    try

        {we can only get a DirectDraw4 interface through the QueryInterface
        method of the DirectDraw object}

        TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw);

    finally

        {now that we have the desired DirectDraw object, the temporary DirectDraw

```

```

    object is no longer needed}

    TempDirectDraw := nil;

end;

{set the cooperative level}
FDirectDraw.SetCooperativeLevel(Handle, DDSCL_FULLSCREEN or DDSCL_EXCLUSIVE);

{set the display resolution and color depth}
FDirectDraw.SetDisplayMode(640, 480, 8, 0, 0);

{initialize the DDSurface structure to indicate that we will be creating a
    complex flipping surface with one backbuffer}
FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);
DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);
DDSurface.dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;
DDSurface.ddsCaps.dwCaps := DDSCAPS_COMPLEX or DDSCAPS_FLIP or
                            DDSCAPS_PRIMARYSURFACE;

DDSurface.dwBackBufferCount := 1;

{create the primary surface object}
FDirectDraw.CreateSurface(DDSurface, FPrimarySurface, nil);

{indicate that we want to retrieve a pointer to the backbuffer (the surface
    immediately behind the primary surface in the flipping chain) }
FillChar(DDSCaps, SizeOf(TDDSCaps2), 0);
DDSCaps.dwCaps := DDSCAPS_BACKBUFFER;

{retrieve the surface}
FPrimarySurface.GetAttachedSurface(DDSCaps, FBackBuffer);

{create a temporary canvas object and retrieve an HDC for the primary surface}
TempCanvas := TCanvas.Create;

```

```

FPrimarySurface.GetDC(SrfcDC);

try

    {set the canvas's handle to the surface's DC}

    TempCanvas.Handle := SrfcDC;

    {clear the surface to black}

    TempCanvas.Brush.Color := clBlack;

    TempCanvas.FillRect(Rect(0, 0, 640, 480));

    {draw some lime colored text}

    TempCanvas.Font.Color := clLime;

    TempCanvas.TextOut(100, 100, 'Primary Surface. Press Escape to close');

finally

    {don't forget to release the DC}

    TempCanvas.Handle := 0;

    FPrimarySurface.ReleaseDC(SrfcDC);

end;

{get the backbuffer's DC}

FBackBuffer.GetDC(SrfcDC);

try

    {set up the canvas, and clear the backbuffer to blue}

    TempCanvas.Handle := SrfcDC;

    TempCanvas.Brush.Color := clBlue;

    TempCanvas.FillRect(Rect(0, 0, 640, 480));

    {draw some yellow colored text}

    TempCanvas.Font.Color := clYellow;

    TempCanvas.TextOut(200, 200, 'Backbuffer Surface. Press Escape to close');

finally

    {release the DC and destroy the temporary canvas object}

```

```

TempCanvas.Handle := 0;

FBackBuffer.ReleaseDC(SrvcDC);

TempCanvas.Free;

end;

{commence flipping}

FlipClock.Enabled := TRUE;

end;

procedure TForm1.FlipClockTimer(Sender: TObject);
begin
    {flip the contents of the primary and backbuffer surfaces}

    FPrimarySurface.Flip(nil, DDFLIP_WAIT);

end;

```

---

## Displaying Bitmaps

The ability to draw text, or even lines and polygons, into a surface is cool, but to really create anything visually exciting, we need to be able to display bitmaps. DirectDraw, unfortunately, does not have any built-in functionality for loading a bitmap, but using GDI techniques and Delphi's TBitmap object, we can easily remedy the situation.

The most common application for loading a bitmap is to store background or sprite images off screen, which will later be copied into the primary or, most likely, the backbuffer surface. Therefore, the first order of business is to create the off-screen surface, which will hold the bitmap image. Again, we will use the CreateSurface method. However, this time we will initialize the dwFlags member of the TDDSurfaceDesc2 structure to DDSD\_CAPS or DDSD\_HEIGHT or DDSD\_WIDTH. We must then set the dwWidth and dwHeight members to the desired width and height of the surface, typically reflecting the width and height of the bitmap image. Finally, we set the ddsCaps.dwCaps member to DDSCAPS\_OFFSCREENPLAIN, indicating that this is a plain, off-screen surface. The color depth of the surface will match that of the primary surface.



**Caution:** You cannot create an off-screen surface wider than the primary surface in video memory unless the hardware specifically supports it. We cover retrieving the hardware capabilities below. However, you can create off-screen surfaces wider than the primary surface in system memory.

---

## Surface Memory Location

When creating a surface, the application can request that the memory for a surface be allocated from a specific area. This allows an application to create surfaces, and thus load bitmaps, into memory on the display hardware, or to system memory. Video memory has the advantage of being incredibly fast when copied into another surface also located on the video memory; display memory has the advantage of being directly accessed by the CPU faster than video memory. When creating a surface, the following four flags can be combined with other flags in the ddsCaps.dwCaps member of the TDDSurfaceDesc2 structure, thereby controlling where the memory for the surface is located.

**Table 4-1:** Memory location flags



Flag	Memory Location
DDSCAPS_LOCALVIDMEM	Conventional video memory. Must be combined with DDSCAPS_VIDEOMEMORY.
DDSCAPS_NONLOCALVIDMEM	Accelerated Graphics Port (AGP) display memory. Must be combined with DDSCAPS_VIDEOMEMORY.
DDSCAPS_VIDEOMEMORY	Video memory.
DDSCAPS_SYSTEMMEMORY	System memory.



**Note:** DDSCAPS\_NONLOCALVIDMEM is applicable only to AGP hardware, and only if the DirectDraw driver for that hardware supports it. You can determine if AGP is supported by DirectDraw by retrieving the hardware capabilities as discussed below.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

You do not have to indicate a memory location for surfaces. By default, surfaces will be created in video memory. When video memory is depleted, or a surface is too big to fit into what's left, it is automatically created in system memory. However, if a location is specified in the CreateSurface call and there is not enough memory, it will fail.

### **Loading and Displaying the Bitmap**

Actually loading the bitmap is quite easy. We simply create a TBitmap object and use its LoadFromFile method to pull the bitmap into memory. Of course, we must still somehow copy the bitmap into our off-screen surface. This is where the previous information about GDI comes in. Using GDI, we can retrieve a device context for our off-screen surface and use the GDI BitBlt function to copy the bitmap into the surface itself. Alternatively, we can create another TCanvas object, assign the surface's device context to the TCanvas's handle, and use the TCanvas.Draw method to copy the bitmap into the surface. With that accomplished, we can release the surface's device context and destroy the TBitmap object. This is illustrated in Listing 4-5.

### **Blitting Surfaces**

OK, now that the bitmap has been loaded into our off-screen surface, which should be located in display memory if there was enough room, we need to copy it from the off-screen surface onto our primary surface for display. Usually we would copy it into the backbuffer surface as part of our animation rendering engine, but for this example, we will simply copy it directly to the primary surface.

To copy one portion of a surface into another surface, we use the IDirectDrawSurface4's BltFast method. The BltFast method is defined as:

```
function BltFast(  
  
    dwX: DWORD; // the destination horizontal coordinate  
  
    dwY: DWORD; // the destination vertical coordinate  
  
    lpDDSrcSurface: IDirectDrawSurface4; // the source surface  
  
    var lpSrcRect: TRect; // the rectangular area on the source  
  
    dwTrans: DWORD // transfer flags
```

```
) : HRESULT; // returns a DirectX error code
```

The first two parameters determine the horizontal and vertical coordinates where the source surface area will be copied onto the destination surface. The third parameter is a pointer to the source surface, and the fourth parameter defines the rectangular area in the source surface that is to be copied onto the destination surface. The last parameter contains a series of flags that control certain aspects about the transfer. In particular, the `DDBLTFAST_WAIT` flag can be specified to indicate that the function should not return until the copying has been accomplished. Other flags indicate transparency options, but these will be covered in the chapter on sprite techniques.

If the resulting bitmap looks somewhat corrupted, don't worry. Our current example makes no use of palettes, and thus the resulting appearance of the bitmap will be at the mercy of the current contents of the system palette. We will examine palettes at length in the next chapter.

*Listing 4-5: Loading and displaying a bitmap*

---

```
var
```

```
    Form1: TForm1;
```

```
    FDirectDraw: IDirectDraw4;
```

```
    FPrimarySurface,
```

```
    FBitmap: IDirectDrawSurface4;
```

```
Implementation
```

```
procedure TForm1.FormActivate(Sender: TObject);
```

```
var
```

```
    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we  
    need a temporary interface}
```

```
    TempDirectDraw: IDirectDraw;
```

```
    {structures required for various methods}
```

```
    DDSurface: TDDSurfaceDesc2;
```

```
    {these variables are used in GDI rendering}
```

```
    TempCanvas: TCanvas;
```

```
    SrfcDC: HDC;
```

```

{the bitmap object}

TempBitmap: TBitmap;

{defines the area to be copied}

SrcRect: TRect;

begin

    {create a temporary DirectDraw object. this is used to create the
    desired DirectDraw4 object}
    DirectDrawCreate(nil, TempDirectDraw, nil);

    try

        {we can only get a DirectDraw4 interface through the QueryInterface
        method of the DirectDraw object}
        TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw);

    finally

        {now that we have the desired DirectDraw object, the temporary DirectDraw
        object is no longer needed}
        TempDirectDraw := nil;

    end;

    {set the cooperative level}
    FDirectDraw.SetCooperativeLevel(Handle, DDSCL_FULLSCREEN or DDSCL_EXCLUSIVE);

    {set the display resolution and color depth}
    FDirectDraw.SetDisplayMode(640, 480, 8, 0, 0);

    {initialize the DDSurface structure to indicate that we will be creating a
    primary surface}
    FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);
    DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);
    DDSurface.dwFlags := DDSD_CAPS;

```

```

DDSsurface.ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE;

{create the primary surface object}
FDDirectDraw.CreateSurface(DDSsurface, FPrimarySurface, nil);

{create a temporary canvas object and retrieve an HDC for the primary
surface}
TempCanvas := TCanvas.Create;
FPrimarySurface.GetDC(SrfcDC);
try
    {set the canvas's handle to the surface's DC}
    TempCanvas.Handle := SrfcDC;

    {clear the surface to black}
    TempCanvas.Brush.Color := clBlack;
    TempCanvas.FillRect(Rect(0, 0, 640, 480));
finally
    {don't forget to release the DC}
    TempCanvas.Handle := 0;
    FPrimarySurface.ReleaseDC(SrfcDC);
end;

{create the temporary bitmap}
TempBitmap := TBitmap.Create;

{load the bitmap}
TempBitmap.LoadFromFile(ExtractFilePath(ParamStr(0))+'Athena.bmp');

{initialize the attributes for the surface}
FillChar(DDSsurface, SizeOf(TDDSurfaceDesc2), 0);
DDSsurface.dwSize := SizeOf(TDDSurfaceDesc2);

```

```
DDSurface.dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
DDSurface.dwWidth := TempBitmap.Width;
DDSurface.dwHeight := TempBitmap.Height;
DDSurface.ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
```

```
{create the desired DirectDraw surface object}
```

```
FDirectDraw.CreateSurface(DDSurface, FBitmap, nil);
```

```
{retrieve a DC for the bitmap surface}
```

```
FBitmap.GetDC(SrfcDC);
```

```
try
```

```
    TempCanvas.Handle := SrfcDC;
```

```
    {draw the bitmap onto the surface using GDI functions}
```

```
    TempCanvas.Draw(0, 0, TempBitmap);
```

```
finally
```

```
    {cleanup}
```

```
    TempCanvas.Handle := 0;
```

```
    FBitmap.ReleaseDC(SrfcDC);
```

```
    TempCanvas.Free;
```

```
end;
```

```
{record the size of the surface (same as the bitmap), as this will be needed  
when we copy the surface into the primary surface}
```

```
SrcRect := Rect(0, 0, TempBitmap.Width, TempBitmap.Height);
```

```
{destroy the bitmap object}
```

```
TempBitmap.Free;
```

```
{blit the off-screen surface onto the primary surface, thus displaying  
the bitmap}
```

```
FPrimarySurface.BltFast(50, 50, FBitmap, SrcRect, DDBLTFAST_WAIT);
```

```
end;
```

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
```

```
    Shift: TShiftState);
```

```
begin
```

```
    if Key = VK_ESCAPE then
```

```
        Close;
```

```
end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Lost Surfaces**

DirectX applications, even under exclusive, full-screen mode, must still coexist gracefully with other Windows applications. This means that users are free to Alt+Tab into and out of a DirectX application into other Windows applications. When a full-screen DirectX application is Alt+Tabbed out of, it is automatically minimized by DirectDraw, and the GDI resumes normal output in the original display mode. This has serious implications for surfaces located in display memory. When the GDI takes over in a situation like this, all display memory is reallocated to the GDI itself, thus wiping out anything stored in a surface located in display memory. When the DirectDraw application is returned to, any methods that manipulate display memory surfaces are likely to return a DDERR\_SURFACELOST error.

When this error is detected, it is a simple matter to restore the surface's memory. Simply call the Restore method of the IDirectDrawSurface4 object in question. The Restore method is defined as:

```
function Restore: HRESULT; // returns a DirectX error code
```

This will restore the memory allocated for that surface, and further methods that manipulate that surface will no longer fail until the surface's memory is lost again. However, the contents of that surface may be undefined, and must be restored. Thus, your application should contain a function that initializes the contents of all surfaces that can be called again and again in the event that surface memory is lost.



**Note:** This only applies to surfaces stored in video memory; system memory surfaces and their content will never be lost. You will need to restore the primary surface memory, which in turn implicitly restores the memory of all attached surfaces (such as the backbuffer).

**Retrieving DirectDraw Capabilities**

DirectX was written so that a developer can take advantage of certain hardware features if they are available, such as overlays or the ability to create off-screen surfaces wider than the primary surface. We can determine what features are available by calling the IDirectDraw4's GetCaps method. The GetCaps method is defined as:



```

function GetCaps(
var lpDDDDriverCaps: TDDCaps;           // hardware capabilities
var lpDDHELCaps: TDDCaps                // emulated capabilities
): HRESULT;                             // returns a DirectX error code

```

The first parameter returns a TDDCaps structure outlining the capabilities supported by the hardware itself, and the second parameter returns a similar structure detailing capabilities that are emulated. The hardware capabilities will vary depending on the available hardware, but the emulated capabilities will remain the same regardless of the hardware (but dependent on the version of DirectDraw installed). These structures are huge, and contain just about anything the developer or application would need to know about available features. The following example illustrates how the members of the individual structures are queried to determine the existence of a particular feature.

***Listing 4-6: Retrieving DirectDraw capabilities***

---

```

procedure TForm1.FormCreate(Sender: TObject);
var
    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
    need a temporary interface}
    TempDirectDraw: IDirectDraw;

    {hardware and emulated capability structures}
    HALCaps, HELCaps: TDDCaps;
begin
    {create a temporary DirectDraw object. this is used to create the
    desired DirectDraw4 object}
    DirectDrawCreate(nil, TempDirectDraw, nil);
    try
        {we can only get a DirectDraw2 interface through the QueryInterface
        method of the DirectDraw object}
        TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw);
    finally
        {now that we have the desired DirectDraw object, the temporary DirectDraw
        object is no longer needed}
        TempDirectDraw := nil;
    end;
end;

```

```

{initialize the capabilities structures}

FillChar(HALCaps, SizeOf(TDDCaps), 0);
FillChar(HELCaps, SizeOf(TDDCaps), 0);
HALCaps.dwSize := SizeOf(TDDCaps);
HELCaps.dwSize := SizeOf(TDDCaps);

{retrieve the hardware and software capabilities}

FDirectDraw.GetCaps(HALCaps, HELCaps);

{display several pieces of information}

SysInfo.Items.Add('Total Video Memory: '+IntToStr(HALCaps.dwVidMemTotal
                div 1000)+' KB');

SysInfo.Items.Add('');

{hardware caps}

SysInfo.Items.Add('Hardware Capabilities');
SysInfo.Items.Add('-----');

if (HALCaps.dwCaps and DDCAPS_BLT) = DDCAPS_BLT then
    SysInfo.Items.Add('Hardware accelerated blits');

if (HALCaps.dwCaps and DDCAPS_BLTCOLORFILL) = DDCAPS_BLTCOLORFILL then
    SysInfo.Items.Add('Hardware accelerated color fills');

if (HALCaps.dwCaps and DDCAPS_BLTQUEUE) = DDCAPS_BLTQUEUE then
    SysInfo.Items.Add('Asynchronous hardware blits');

if (HALCaps.dwCaps and DDCAPS_BLTSTRETCH) = DDCAPS_BLTSTRETCH then
    SysInfo.Items.Add('Hardware stretch blits');

if (HALCaps.dwCaps2 and DDCAPS2_CERTIFIED ) = DDCAPS2_CERTIFIED then
    SysInfo.Items.Add('Microsoft certified driver');

```

```
if (HALCaps.dwCaps2 and DDCAPS2_NONLOCALVIDMEM ) = DDCAPS2_NONLOCALVIDMEM then
    SysInfo.Items.Add('AGP video card');

if (HALCaps.dwCKeyCaps and DDCKEYCAPS_SRCBLT) = DDCKEYCAPS_SRCBLT then
    SysInfo.Items.Add('Hardware accelerated source color key blits');

SysInfo.Items.Add('');

SysInfo.Items.Add('');

{emulated caps}

SysInfo.Items.Add('Emulated Capabilities');

SysInfo.Items.Add('-----');

if (HELCaps.dwCaps and DDCAPS_BLT) = DDCAPS_BLT then
    SysInfo.Items.Add('Emulated blits');

if (HELCaps.dwCaps and DDCAPS_BLTCOLORFILL) = DDCAPS_BLTCOLORFILL then
    SysInfo.Items.Add('Emulated color fills');

if (HELCaps.dwCaps and DDCAPS_BLTQUEUE) = DDCAPS_BLTQUEUE then
    SysInfo.Items.Add('Emulated asynchronous blits');

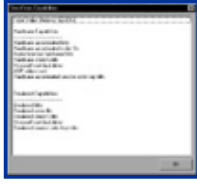
if (HELCaps.dwCaps and DDCAPS_BLTSTRETCH) = DDCAPS_BLTSTRETCH then
    SysInfo.Items.Add('Emulated stretch blits');

if (HELCaps.dwCaps2 and DDCAPS2_CERTIFIED ) = DDCAPS2_CERTIFIED then
    SysInfo.Items.Add('Microsoft certified driver');

if (HELCaps.dwCaps2 and DDCAPS2_NONLOCALVIDMEM ) = DDCAPS2_NONLOCALVIDMEM then
    SysInfo.Items.Add('Emulated AGP video card');

if (HELCaps.dwCKeyCaps and DDCKEYCAPS_SRCBLT) = DDCKEYCAPS_SRCBLT then
    SysInfo.Items.Add('Emulated source color key blits');
```

end ;



**Figure 4-5:** Some supported DirectDraw features

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## The DirectX Baseline Application

In an effort to expedite the creation of DirectX applications, this book contains a baseline DirectX application that instantiates DirectDraw, sets up a page flipping architecture, and implements a high-speed rendering loop that can attain the highest animation speed possible. This is implemented by providing an event handler for the TApplication's OnIdle event. The OnIdle event is called whenever the application becomes idle; if the Done parameter of this method is set to False, this event will continuously be called by the application, providing a perfect spot for a rendering loop. There are stubbed out methods for drawing surface content and restoring surface content in the event that it is lost. A number of constants are defined at the top of the main unit that control the display resolution and color depth, as well as the cooperative level and the flags for the complex flipping surface. The indicated display resolution and color depth is checked to see if it is supported before setting the display mode. It will also shut off the rendering loop automatically when the application is Alt+Tabbed out of, and restart it when Alt+Tabbed back into. Theoretically, all one must do is load off-screen surfaces in the area indicated in the Activate method, flesh out the RestoreSurfaces method to restore any off-screen surface memory, and complete the DrawSurfaces method to render animation to the backbuffer surface. The act of flipping the surfaces is carried out by the rest of the application. The Escape key will exit the application and return to Windows.

Bear in mind that this is a general architecture meant to assist the reader in quickly getting a DirectX application up and running. It also provides a standardized framework from which every other example throughout the rest of this book will be based, cutting down on the amount of code that must be illustrated in example listings. This is not the best architecture for every game. Indeed, you may need to alter it dramatically depending on the requirements of your application. However, the baseline application tries to be as unimposing as possible, taking care of standard functionality while providing plenty of room for application-specific logic. What follows is a complete listing of the baseline application. Future examples will be based on this application, and thus will not contain any of the code for initializing DirectDraw; code listings will only contain code pertinent to the current discussion. This baseline application will run as is, but the output will be a black screen. However, it is a simple task to implement code in the DrawSurfaces method that will output animation to the screen, and we will see many examples of this in the coming chapters.

*Listing 4-7: The DirectX baseline application*

---

```
unit DDrawSkelU;
```

```
{*****
```

Skeleton DirectDraw Application

Author: John Ayres

This is a bare bones DirectDraw application that creates a primary surface with one backbuffer and implements a continuous flip between them. It contains some minimal error checking code as well as an application exception handling routine that flips back to the GDI surface so that exception messages can be seen. The Application.OnIdle event is used to implement the loop that flips the surfaces. This uses a large chunk of CPU time (which is ok in a game), but could potentially be replaced with a While..Do loop or some other looping architecture.

This baseline code is used throughout the book as a starting point for examples. It was designed to be easily extensible, and should provide all of the initialization and startup code that you will need for your own DirectX applications.

Code that is marked by '///

```
*****
```

Copyright © 1999 by John Ayres, All Rights Reserved

This code is freeware and can be used for any personal or commercial purposes without royalty or license. Use at your own risk.

```
*****}
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

```
DDraw;
```

```
{these constants are used to modify specific attributes of the DirectX  
application, such as the resolution and color depth}
```

```
const
```

```
DXWIDTH      = 640;
```

```
DXHEIGHT     = 480;
```

```
DXCOLORDEPTH = 8;
```

```
BUFFERCOUNT = 1;
```

```
COOPERATIVELEVEL = DDSCL_FULLSCREEN or DDSCL_ALLOWREBOOT or DDSCL_ALLOWMODEX  
                  or DDSCL_EXCLUSIVE;
```

```
SURFACETYPE   = DDSCAPS_COMPLEX or DDSCAPS_FLIP or DDSCAPS_PRIMARYSURFACE;
```

```
const
```

```
{this user-defined message is used to start the flipping loop}
```

```
WM_DIRECTXACTIVATE = WM_USER + 200;
```

```
type
```

```
TfrmDXAppMain = class(TForm)
```

```
  procedure FormCreate(Sender: TObject);
```

```
  procedure FormDestroy(Sender: TObject);
```

```
  procedure FormKeyDown(Sender: TObject; var Key: Word;
```

```
    Shift: TShiftState);
```

```
  procedure FormActivate(Sender: TObject);
```

```
private
```

```
  { Private declarations }
```

```
{flips back to the GDI surface to display the exception error message}
```

```
  procedure ExceptionHandler(Sender: TObject; ExceptionObj: Exception);
```

```
{the main rendering loop}
```

```
  procedure AppIdle(Sender: TObject; var Done: Boolean);
```

```
{intercepts certain messages to provide appropriate functionality}
procedure AppMessage(var Msg: TMsg; var Handled: Boolean);

{flips the DirectDraw surfaces}
procedure FlipSurfaces;

{restores any lost surfaces}
procedure RestoreSurfaces;

{draws the contents of surfaces}
procedure DrawSurfaces;

public
  { Public declarations }
end;

var
  frmDXAppMain: TfrmDXAppMain;

  {the main DirectDraw interface}
  FDirectDraw: IDirectDraw4;

  {the interfaces for the primary and backbuffer surfaces}
  FPrimarySurface,
  FBackBuffer: IDirectDrawSurface4;

implementation

uses
  DXTools;

{$R *.DFM}
```



```

{ *-->>  BASELINE APPLICATION CODE <---* }

{ - the callback function used to ensure that the selected graphics -
  - mode is supported by DirectX - }

function EnumModesCallback(const EnumSurfaceDesc: TDDSurfaceDesc2;
                           Information: Pointer): HRESULT; stdcall;

begin
  {if the height, width, and color depth match those specified in the
   constants, then indicate that the desired graphics mode is supported}
  if (EnumSurfaceDesc.dwHeight = DXHEIGHT) and
     (EnumSurfaceDesc.dwWidth = DXWIDTH) and
     (EnumSurfaceDesc.ddpfPixelFormat.dwRGBBitCount = DXCOLORDEPTH) then
    Boolean(Information^) := TRUE;

  Result := DDENUMRET_OK;
end;

{ -> Events hooked to the application <- }

{ - this event is called when an exception occurs, and simply flips back -
  - to the GDI surface so that the exception dialog box can be read - }
procedure TfrmDXAppMain.ExceptionHandler(Sender: TObject;
                                          ExceptionObj: Exception);

begin
  {disconnect the OnIdle event to shut off the rendering loop}
  Application.OnIdle := nil;

  {if the DirectDraw object was successfully created, flip to the GDI surface}
  if Assigned(FDirectDraw) then
    FDirectDraw.FlipToGDISurface;

```

```

{display the exception message}

MessageDlg(ExceptionObj.Message, mtError, [mbOK], 0);

{reconnect the OnIdle event to reenter the rendering loop}

Application.OnIdle := AppIdle;

end;

{ - this method is continuously called by the application, and provides -
  - the main rendering loop. this could be replaced by a custom -
  - while..do loop - }

procedure TfrmDXAppMain.AppIdle(Sender: TObject; var Done: Boolean);begin
  {indicates that the application should continuously call this method}

  Done := FALSE;

  {if DirectDraw has not been initialized, exit}

  if not Assigned(FDirectDraw) then Exit;

  {note that at this point game logic could be inserted that controls things
   like sprite movement and collision detection}

  {draw surface content and flip the surfaces}

  DrawSurfaces;

  FlipSurfaces;

end;

{ - handles certain messages that are required to make DirectX function -
  - properly within Delphi - }

procedure TfrmDXAppMain.AppMessage(var Msg: TMsg; var Handled: Boolean);
begin
  case Msg.Message of

    WM_ACTIVATEAPP:

      {unhook the OnIdle event when the application is being deactivated.

```

```

        this will stop all rendering}

if not Boolean(Msg.wParam) then
    Application.OnIdle := nil
else
    {upon activating the application, send ourselves the user-
    defined message}

    PostMessage(Application.Handle, WM_DIRECTXACTIVATE, 0, 0);
WM_DIRECTXACTIVATE:
begin
    {upon activating, restore all surfaces (reloading their memory as
    necessary), hook up the OnIdle event, and redraw the contents of
    all surfaces}

    RestoreSurfaces;

    Application.OnIdle := AppIdle;

    DrawSurfaces;

end;
WM_SYSCOMMAND:
begin
    {do not allow a screen saver to kick in}

    Handled := (Msg.wParam = SC_SCREENSAVE);

end;

end;

end;

{ -> Form Events <- }

{ - initialize essential form properties - }
procedure TfrmDXAppMain.FormCreate(Sender: TObject);
begin
    {set up the application exception handler}

    Application.OnException := ExceptionHandler;

```

```

{initialize form properties. note that the FormStyle property must be
  set to fsStayOnTop}

BorderStyle := bsNone;

BorderIcons := [];

FormStyle   := fsStayOnTop;

Color       := clBlack;

Cursor      := crNone;

{turn off the mouse cursor}

ShowCursor(FALSE);

end;

{ - provides essential cleanup functionality - }
procedure TfrmDXAppMain.FormDestroy(Sender: TObject);
begin
  {disengage our custom exception handler}

  Application.OnException := nil;

  {display the mouse cursor}

  ShowCursor(TRUE);

  {remember, we do not have to explicitly free the DirectX objects,
   as they will free themselves when they go out of context (such as
   when the application is closed)}
end;

{ - this method initializes DirectX and creates all necessary objects - }
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
var
  {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
   need a temporary interface}

  TempDirectDraw: IDirectDraw;

```

```

{structures required for various methods}

DDSurface: TDDSurfaceDesc2;

DDSCaps: TDDSCaps2;

{flag used to determine if the desired graphics mode is supported}

SupportedMode: Boolean;

begin

  {if DirectDraw has already been initialized, exit}

  if Assigned(FDirectDraw) then exit;

  {create a temporary DirectDraw object. this is used to create the
  desired DirectDraw4 object}

  DXCheck( DirectDrawCreate(nil, TempDirectDraw, nil) );

  try

    {we can only get a DirectDraw4 interface through the QueryInterface
    method of the DirectDraw object}

    DXCheck( TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw) );

  finally

    {now that we have the DirectDraw4 object, the temporary DirectDraw
    object is no longer needed}

    TempDirectDraw := nil;

  end;

  {hook up the application message handler}

  Application.OnMessage := AppMessage;

  {call EnumDisplayModes and verify that the desired graphics mode
  is indeed supported}

  FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

  DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

```

```

DDSurface.dwFlags := DDSD_HEIGHT or DDSD_WIDTH or DDSD_PIXELFORMAT;

DDSurface.dwHeight := DXHEIGHT;

DDSurface.dwWidth := DXWIDTH;

DDSurface.ddpfPixelFormat.dwSize := SizeOf(TDDPixelFormat_DX6);

DDSurface.ddpfPixelFormat.dwRGBBitCount := DXCOLORDEPTH;

SupportedMode := FALSE;

DXCheck( FDirectDraw.EnumDisplayModes(0, @DDSurface, @SupportedMode,
                                     EnumModesCallback) );

{if the desired graphics mode is not supported by the DirectX drivers,
 display an error message and shut down the application}

if not SupportedMode then
begin
    MessageBox(Handle, PChar(The installed DirectX drivers do not support a '+'
                             'display mode of: '+IntToStr(DXWIDTH)+' X '+
                             IntToStr(DXHEIGHT)+'', '+IntToStr(DXCOLORDEPTH)+' bit color'),
               Unsupported Display Mode Error', MB_ICONERROR or MB_OK);

    Close;

    Exit;
end;

{set the cooperative level to that defined in the constants}
DXCheck( FDirectDraw.SetCooperativeLevel(Handle, COOPERATIVELEVEL) );

{set the display resolution and color depth to that defined in the constants}
DXCheck( FDirectDraw.SetDisplayMode(DXWIDTH, DXHEIGHT, DXCOLORDEPTH, 0, 0) );

{initialize the DDSurface structure to indicate that we will be creating a
 complex flipping surface with one backbuffer}
FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

DDSurface.dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;

```

```

DDSsurface.ddsCaps.dwCaps := SURFACETYPE;

DDSsurface.dwBackBufferCount := BUFFERCOUNT;

{create the primary surface object}

DXCheck( FDirectDraw.CreateSurface(DDSsurface, FPrimarySurface, nil) );

{indicate that we want to retrieve a pointer to the backbuffer (the surface
immediately behind the primary surface in the flipping chain) }

FillChar(DDSCaps, SizeOf(TDDSCaps2), 0);

DDSCaps.dwCaps := DDSCAPS_BACKBUFFER;

{retrieve the surface}

DXCheck( FPrimarySurface.GetAttachedSurface(DDSCaps, FBackBuffer) );

{at this point, offscreen buffers and other surfaces should be created. other
DirectDraw objects should be created and initialized as well, such as
palettes. the contents of all offscreen surfaces should also be initialized
at this point}

{post a message that will hook up the OnIdle event and start the main
rendering loop}

PostMessage(Handle, WM_ACTIVATEAPP, 1, 0);

end;

{ -> Form Methods <- }

{ - this method is called in order to flip the surfaces - }

procedure TfrmDXAppMain.FlipSurfaces;

var

    DXResult: HRESULT;

begin

    {perform the page flip. note that the DDFLIP_WAIT flag has been used,
indicating that the function will not return until the page flip has been

```

performed. this could be removed, allowing the application to perform other processing until the page flip occurs. however, the application will need to continuously call the Flip method to ensure that the page flip takes place}

```
DXResult := FPrimarySurface.Flip(nil, DDFLIP_WAIT);
```

```
{if the surfaces were lost, restore them. on any other error,  
raise an exception}
```

```
if DXResult = DDERR_SURFACELOST then
```

```
    RestoreSurfaces
```

```
else if DXResult <> DD_OK then
```

```
    DXCheck(DXResult);
```

```
end;
```

```
{ - this method is called when the surface memory is lost -  
- and must be restored. surfaces in video memory that -  
- contain bitmaps must be reinitialized in this function - }
```

```
procedure TfrmDXAppMain.RestoreSurfaces;
```

```
begin
```

```
    {restore the primary surface, which in turn restores any implicit surfaces}
```

```
    FPrimarySurface._Restore; {here you should reload any bitmaps stored in video  
memory surfaces}
```

```
end;
```

```
{ - this method is called when the contents of the surfaces need to be -  
- drawn. it will be continuously called by the AppIdle method, so any -  
- rendering or animation could be done within this method - }
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
begin
```

```
end;
```

```
{ -> Deletable Events <- }
```



```
///  
{ - as a matter of convenience this framework will terminate when the -  
  - Escape key is pressed, but this should probably be deleted and -  
  - replaced with your own terminate methods - }  
  
procedure TfrmDXAppMain.FormKeyDown(Sender: TObject; var Key: Word;  
  Shift: TShiftState);  
  
begin  
  if Key = VK_ESCAPE then  
    Close;  
  
end;  
  
///  
  
{ *-->> END BASELINE APPLICATION CODE <<--* }  
  
end.
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## DirectX Result Codes

Just about every DirectX function and method returns an HRESULT, a 32-bit number that maps to a DirectX error code. There are hundreds of error codes, all of which are located in the DDRAW unit. These codes can be translated into a textual form by calling the DXErrorString function located in the DXTools unit. The DXTools unit also contains a procedure called DXCheck that will raise an exception if a DirectX function returns any error code other than DD\_OK.

In theory, every call to any DirectX method should be wrapped with this DXCheck function. In practice, this may be impractical, but you should always check the results of any extremely critical methods, such as blits, flips, and initialization code. Methods such as these can cause an application to waltz off into limbo if they fail and are not gracefully handled, so checking the result code is very important for a robust application.

The DXErrorString function can be incredibly useful when debugging a DirectX application. The result of this function could be written out to a disk-based text file, and later examined if the DirectX application crashes. This tends to happen often when developing DirectX applications, and due to the fact that you cannot step through your code or use any of Delphi's awesome debugging features, this is a useful, if sloppy, debugging technique.

## Retrieving the DirectX Version

Most commercial DirectX applications ship with a version of the DirectX run-time redistributables. The developer knows that the application will be using either the version installed with the application or a more recent version installed by other applications or the operating system. However, sometimes it is necessary to determine which version of DirectX is installed before initializing objects, particularly if the application is to be distributed without the DirectX run-time redistributables, such as on floppy or over the Internet.

Specifically, you should not be checking for an overall DirectX version, but instead you should check for the availability of specific interfaces. As we've seen, the QueryInterface method implemented in every COM object can determine if a specific interface is supported. Using this method, you can determine if certain features are supported on the target machine. For example, if the IDirectDraw4 interface is not available, you know that you're working with a version of DirectDraw older than 6. By querying for other interfaces, you

can determine if 3-D sound or force feedback devices are supported. Using the QueryInterface method is much more reliable than checking the drive for the existence and time stamp of particular files.

Additionally, DirectSetup features a method for retrieving the current version of DirectX that is installed on the machine. See Appendix A for complete information about DirectSetup and how to retrieve the current version of DirectX.

## Summary

In this chapter, we discussed several DirectX and DirectDraw topics. The examples built from initializing DirectDraw all the way to implementing a general page flipping architecture and displaying bitmaps. DirectX programming in general is a very complex topic, filled with many nuances with which the developer must become familiar. We covered DirectDraw programming to a minor degree in order to illustrate this point, and as a foundation upon which the rest of the book's knowledge will be built. When coding a DirectX application, and using DirectDraw in particular, it is important to keep these points in mind:

- The DirectX development team's goal was to create a low-level architecture that would provide developers with the hardware control they need at the speed they demanded, while still retaining Windows' device-independent benefits. This was to be accomplished without imposing a restrictive high-level design architecture. Therefore, DirectX promises three things to developers: fast execution, minimal style restraints, and a level of abstraction that will provide benefits from future hardware.
- DirectX delivers exceptionally on all of its promises, and offers many benefits, such as speed, control, and feature emulation. However, DirectX programming has a number of drawbacks, such as difficult debugging, few Delphi-specific examples, and a steep learning curve. The developer must determine if the benefits and advantages offered by DirectX outweigh the drawbacks and disadvantages for the specific functionality required by the application.
- The DirectX API itself, in the form of various COM objects and their methods, acts as an interface into the DirectX drivers. On the back side, the DirectX drivers can be thought of as two separate systems: the hardware abstraction layer (HAL) and the hardware emulation layer (HEL). The HAL represents the hardware itself. The HEL, on the other hand, represents the software emulation of certain features. It is important to note that, for the most part, this is completely transparent to developers.
- DirectX is comprised of many components. While the inventive game developer could potentially make use of all of these components, there are only a few which are essential for game development: DirectDraw, DirectSound, and DirectInput. Other components that may be useful in a particular game include DirectPlay, DirectSetup, Direct3D, and DirectMusic.
- Under Delphi, you do not necessarily need the DirectX SDK; you only need the DirectX redistributables that are installed with the DirectX SDK or almost any commercially available game. However, the DirectX SDK includes a number of debugging and control panel utilities that can be useful, and is the only source for the DirectX API documentation. In this book, we use the DirectX headers from the JEDI project, translated by Erik Unger. However, there are several other commercial and shareware/freeware DirectX components and headers that can dramatically decrease the production time for a DirectX project under Delphi.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

- DirectX objects are COM based, consequently involving all of the associated benefits and drawbacks of using COM objects. Working with DirectX objects has some similarities to working with other COM objects, but there are a few differences. The primary difference is the fact that DirectX objects do not need to be created by using the CoCreateInstance function. All DirectX objects are created through custom function calls that return an instantiated COM object. A COM object can almost be thought of as a VCL object with no properties, only methods. The methods of a COM object are known as an interface. A COM object can have more than one interface, and it can be extended by added new interfaces, similar to deriving a new component from an ancestor. However, the COM standard dictates that an interface can never change. COM objects determine when they can destroy themselves, and thus do not need to be freed by the creating application. A unique identifier called a GUID is used to distinguish individual COM objects.
- Perhaps the single most important concept in DirectDraw programming is that of the surface. A surface is directly analogous to Delphi's TBitmap or TCanvas object. It represents a virtual "picture" upon which graphical output can be drawn. In reality, a surface is a linear chunk of memory that is managed as if it were rectangular. Surfaces have width and height, among many other properties. The memory for a surface can be located in a number of places, but primarily it will be found in system memory or in the memory on the display card itself. There are many different types of surfaces, some of which are very specialized for a particular purpose. For the purposes of this book, we will be concerned with only three different types: the primary surface, the backbuffer surface, and off-screen surfaces.
- Page flipping is an incredibly powerful technique for producing flicker-free animation. The term "page flipping" comes from the inspiration for this animation technique, that of traditional cartoon animation. Cartoons (and movies, for that matter) create the illusion of animation by displaying to the viewer a series of images, one after the other. Someone can create a crude animation by taking a stack of paper, drawing an image on each page, and then flipping through it rapidly. This is the architecture of page flipping animation. The idea is to create two surfaces, the primary surface and the backbuffer surface. The next frame of animation is composed on the backbuffer surface, and then it is "flipped" to the display screen. This updates the entire display screen in one fell swoop, resulting in a flickerless animation.
- There are several tasks an application is required to perform before DirectDraw can be instantiated and the display mode modified. Some of these tasks are optional, some are not. The application may want to enumerate all attached display devices, although this is not specifically required. After the user picks an enumerated display device, or if the application simply decides to go with the primary

display device, the DirectDraw object can be created. Supported display modes can then be enumerated; while not specifically required, the display mode into which the display hardware is switched must be a supported mode, which can be verified by enumerating supported display modes. The cooperative level can then be set, determining if the application will run full screen or windowed. Finally, setting the display mode causes the hardware to switch over to the indicated resolution and color depth. Bear in mind that certain requirements are imposed on the main window before the display mode can be set properly.

- A device context handle can be retrieved from a surface, allowing the application to use GDI rendering functions. When combined with a TCanvas object, the full power of Delphi's built-in graphical rendering functionality can be brought to bear. However, the sluggishness of the GDI will also be present. It is very important that any device contexts retrieved be released before performing any DirectDraw functions that manipulate the surface.
- Bitmaps can be easily loaded and stored into a surface by using the TBitmap object combined with the drawing capabilities of a TCanvas object and a device context retrieved from a surface.
- The memory associated with surfaces located in display memory can be lost if the user Alt+Tabs out of the application. When the DirectDraw application receives focus again, it must call the Restore method of any display memory surfaces, including the primary surface. Their contents will be undefined, so they must be reinitialized by reloading bitmaps or redrawing graphics.
- DirectX was written so that a developer can take advantage of certain hardware features if they are available, such as overlays or the ability to create off-screen surfaces wider than the primary surface. We can determine what features are available by calling the IDirectDraw4's GetCaps method.
- Just about every DirectX function and method returns an HRESULT, a 32-bit number that maps to a DirectX error code. There are hundreds of error codes, all of which are located in the DDraw unit. These codes can be translated into a textual form by calling the DXErrorString function located in the DXTools unit. You should always check the results of any extremely critical methods, such as blits, flips, and initialization code. Methods such as these can cause an application to waltz off into limbo if they fail and are not gracefully handled, so checking the result code is very important for a robust application.
- Sometimes it is necessary to determine which version of DirectX is installed before initializing objects, particularly if the application is to be distributed without the DirectX run-time redistributables, such as on floppy or over the Internet. Specifically, you should not be checking for an overall DirectX version, but instead you should check for the availability of specific interfaces. The QueryInterface method implemented in every COM object can determine if a specific interface is supported.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





[KEYWORD SEARCH](#)

[Search Tips](#)

[Advanced Search](#)

[PUBLICATION LOOKUP](#)

[JUMP TO TOPIC](#)



**Desktop Library**

[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 5 Palettes

This chapter covers the following topics:

- The definition of a color palette
- The creation and use of the IDirectDrawPalette object
- Palette animation

Game programmers tend to have mixed feelings about palettized display modes. On one hand, you've got special animation effects and superior performance benefiting you from using a palettized display mode. On the other hand, you've got graphic resource management nightmares, an extra layer of indirection and complexity, and a limited resource in the form of color. With the increasing speed of today's hardware, someday the performance boost available with palettized display modes will not outweigh the benefits of using 16-bit or 32-bit color modes. However, at this point, creating games in a 256-color palettized display mode still has many advantages over the higher color depth modes. Since this book discusses techniques for game and graphics programming under 256 video modes, we should examine how DirectX handles color palettes in a little more depth.

In this chapter, we will examine how palettes are supported under DirectDraw in full-screen exclusive mode. We covered color depth and palettes back in Chapter 3, Basic Graphics Programming, and palettes still work very much the same way under DirectX. For the sake of completeness (in case you skipped Chapter 3), we'll briefly examine what a color palette is and how it works. Then we'll dive right into creating DirectDraw palettes, how they are used, and some special effects you can accomplish with them. If you read the basic graphics programming chapter, skip ahead to the section on the IDirectDrawPalette object.

### What is a Palette?

A color palette is a mechanism by which a potentially large range of colors can be displayed without using enormous amounts of memory. Four and eight bits-per-pixel color depths are the only palettized video modes. As 4 bits-per-pixel color depths are no longer used for games, we will discuss only 8 bits-per-pixel color depths. At 8 bits-per-pixel, each pixel can assume a value between 0 and 255. These values act as an index into what can be considered a 256-entry array known as the color palette. Now, each entry in this array contains an 8-bit value for the blue, green, and red intensity of a color. In effect, this gives 8 bits-per-pixel color depths access to 16,777,216 potential colors, but only 256 of them can be shown at one time. When a bitmap is displayed, the video hardware

looks at the 8-bit pixel value, and using that value as an index into the 256-entry color table, it draws a pixel using the red, green, and blue intensities stored in the indicated color table entry.



**Figure 5-1:** An 8 bits-per-pixel bitmap's pixels act as indices into a color palette

There are many things to consider when working with 8 bits-per-pixel color modes. A major disadvantage to this mode is that all bitmaps to be displayed simultaneously must be drawn using the same palette. This requires a lot of careful planning on the part of art production, and can limit the creativity of artists and the variety of artwork that can be displayed on screen. Bitmaps that are drawn using one palette but displayed on a device in which a different palette is currently being used will have unpredictable results. A bitmap's pixels could be mapped to the closest approximation of the appropriate color in the current palette, but this process is slow, usually produces inadequate results, and is not appropriate for fast action games. Different palettes can be used between scenes, however, which can reduce this effect.

However, 8 bits-per-pixel video modes give the greatest speed advantage, and a tremendous number of colors can potentially be used while taking up the least amount of RAM. Up to four pixels can be moved simultaneously by using 32-bit memory movement instructions, allowing bitmaps to be moved about at blindingly fast speeds. Another major advantage comes from palette animation. If colors within the palette are moved about, or cycled, every pixel drawn using that color is affected simultaneously and automatically. This can be used for various special effects, such as flowing water or lighting. Palette animation is not available under non-paletted video modes. Since this book is concerned with high-performance games and graphics programming, all examples will use only 8 bits-per-pixel video modes in order to squeeze out as much speed as possible.

## The IDirectDrawPalette Object

Using DirectDraw, an application has direct access to the display card's hardware palette registers. In order to set the palette entries of the display hardware, the developer must use the IDirectDrawPalette object. Of all the DirectX objects, the palette object is perhaps the easiest to manipulate and use. It has only a few methods, and using the object is very straightforward.

### Creating the Palette Object

The CreatePalette method of the IDirectDraw object is called to actually create the IDirectDrawPalette object. This simple function takes only a few parameters, and is declared as:

```
function CreatePalette (  
  
    dwFlags: DWORD;                // control flags  
  
    lpColorTable: pointer;         // a pointer to an array of  
                                   // TPaletteEntry structures  
  
    var lplpDDPalette: IDirectDrawPalette; // a pointer to the new  
                                           // IDirectDrawPalette object  
  
    pUnkOuter: IUnknown           // COM aggregation  
  
): HRESULT;                       // returns an error code
```

The first parameter takes a series of constants that can be combined using the OR boolean operator to indicate the type of palette being created and some of its capabilities. In particular, 2-, 4-, 16-, or 256-entry palettes can be created by specifying the appropriate flags. Since we're primarily interested in 256-color display modes, we'll use the DDPCAPS\_8BIT flag to indicate that we want 256 entries in the palette. The third parameter takes a pointer to a variable of type IDirectDrawPalette that will hold the instantiated palette object when this function

returns. The final parameter will be used in the future for COM aggregation. For now, we'll always set this parameter to nil.

The second parameter is really the heart of this function. It is a pointer to an array of palette entry structures that contain the colors used to initialize the palette. In our case, this will be an array of 256 entries. These structures are of type TTrueColor, and are defined as:

```
TTrueColor = record  
  
    case integer of  
  
        1 : (Data : DWORD);           // the color as a 32-bit value  
  
        2 : (R,G,B,A : byte);        // red, green, blue, and alpha intensities  
  
    end;
```

This structure contains members for the red, green, and blue components of a color, defining a 24-bit RGB color value. The final member is useful only under windowed DirectX and with Direct3D, so we will not be concerned with it.



*Note* Incidentally, the TPaletteEntry data structure type could also be used. It is defined somewhat differently, but it is the same size as this structure, and its members map to the same members of the TTrueColor structure.

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).





**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN: 1556226373 Pub Date: 12/01/99**

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Defining Palette Colors**

It would be easy enough for an application to programmatically define all of the colors in the palette. We could simply iterate through all entries in the color table array, setting the red, green, and blue values as necessary to create the desired palette. While this is certainly a workable approach, and is desirable in the case where a custom color gradient effect needs to be achieved, most often an application's palette needs to be initialized with the colors used in a bitmap.

Fortunately for us, we're using Delphi, which makes this whole problem incredibly easy to solve. To start, we'll need to create a TBitmap object, and use it to load in the bitmap whose palette we wish to use. Then, we'll use the GDI function GetPaletteEntries to retrieve all of the bitmap's palette entries. The GetPaletteEntries function is defined as:

```
GetPaletteEntries(  
  
    Palette: HPALETTE;           // a handle to a logical palette  
  
    StartIndex: UINT;           // the starting palette index  
  
    NumEntries: UINT;           // the number of palette entries to retrieve  
  
    PaletteEntries: Pointer      // a pointer to an array of TPaletteEntry structures  
  
): UINT;                       // returns the number of palette entries retrieved
```

The first parameter takes a handle to a logical palette, which we can simply set to the Palette property of the TBitmap object we've just created. The second parameter indicates the index of the first palette slot we would like to retrieve. The third parameter indicates the number of palette slots to retrieve. The final parameter is a pointer to an array of TPaletteEntry structures which will receive the palette data.

The cool thing here is that we can pass this function the same array of TTrueColor structures that we'll use in the call to CreatePalette. The entire procedure only takes a few lines of code, and looks something like this:

```
var
```

```

Palette: array[0..255] of TTrueColor;

DDPalette: IDirectDrawPalette;

TempBitmap: TBitmap;

begin

    the temporary bitmap

    TempBitmap := TBitmap.Create;

    try

        the bitmap, and extract its palette

        TempBitmap.LoadFromFile(c:\somebitmap.bmp');

        GetPaletteEntries(TempBitmap.Palette, 0, 256, Palette);

        the palette object

        DDraw.CreatePalette(DDPCAPS_8BIT, @Palette, DDPalette, nil);

    finally

        TempBitmap.Free;

    end;

end;

```

This creates an `IDirectDrawPalette` object that is initialized with the palette stored in the specified bitmap. We've encapsulated this functionality in the `DDUtil` unit on the companion CD in the function `DDLloadPalette`. You can simply call this function, passing it a pointer to the `DirectDraw` object and the path and filename of the bitmap whose palette you wish to use, and it returns an instantiated `IDirectDrawPalette` object. This makes palette creation and initialization a one-line function call.




---

**Caution:** By default, only 254 palette entries in an 8-bit palette can be used. `DirectDraw` predefines palette slot 0 as black and palette slot 255 as white. You can use all 256 entries by specifying the `DDPCAPS_ALLOW256` flag in the first parameter of the call to `CreatePalette`. However, some video cards use palette slot 0 to define the overscan color, the color of the area around the usable display area. Some versions of Windows NT won't allow this slot to be changed at all. Changing palette slot 0 may cause some unpredictable effects on some video cards, so don't use the `DDPCAPS_ALLOW256` flag unless absolutely necessary.

---



**overscan:** The area on a monitor that borders the usable display area.

## Using the Palette Object

Actually using a palette object is almost as simple as creating one. We use the `SetPalette` method of the `IDirectDrawSurface4` object to associate a palette with a palettized surface. When a palette is associated with a surface, its color table is inserted into the hardware color registers, instantly changing the color of any displayed pixels. The only parameter this function takes is a pointer to the `IDirectDrawPalette` object that was created earlier. The `SetPalette` method is defined as:

```
function SetPalette (  
  
    lpDDPalette: IDirectDrawPalette      // a pointer to an IDirectDrawPalette object  
  
): HRESULT;                             // returns a DirectX error code
```

The only parameter to this method is a pointer to the IDirectDrawPalette object we've created. Only the palette associated with the primary surface will affect the actual hardware palette registers. Thus, you must attach the newly created palette to the primary surface before the new colors take effect. Palettes can actually be attached to any palettized surface, and the same palette can be attached to more than one surface, but only the palette attached to the primary surface will actually affect the display.



---

**Note:** If a surface becomes lost and must be restored, any attached palettes are unaffected, and do not need to be reconnected after a surface is restored. Additionally, palettes do not move when surfaces are flipped, so it is unnecessary to attach palettes to both a primary surface and backbuffer.

---

If an IDirectDrawPalette object is not attached to the primary surface, the current entries in the hardware palette (left over from the last application) will be used. This usually results in bitmaps and graphics that appear garbled or corrupted. Therefore, unless your graphics will only be using the standard 20 Windows colors (left in the palette if the last application ran was a regular Windows application), you should always instantiate and attach a new palette object to the primary surface.

The following examples illustrate what happens when a bitmap is displayed with and without associating a palette with the primary surface. When an IDirectDrawPalette object is created, initialized with the bitmap's palette, and associated with the primary surface, the displayed bitmap appears as it did when it was created. In the second example, the bitmap is shown without associating any sort of palette with the primary surface. The results will be unpredictable, and will depend largely upon whatever application last modified the palette registers.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 5-1: Displaying a bitmap with a palette*

```
var
```

```
.
```

```
.
```

```
.
```

```
FPalette: IDirectDrawPalette;
```

```
.
```

```
.
```

```
.
```

```
procedure TFormDXAppMain.FormActivate(Sender: TObject);
```

```
var
```

```
.
```

```

.
.

{the array of palette entries}
    Palette: array[0..255] of TTrueColor;

    {our temporary bitmap}
    TempBitmap: TBitmap;
begin
    .
    .
    .

    {retrieve the surface}
    DXCheck( FPrimarySurface.GetAttachedSurface(DDSCaps, FBackBuffer) );

    {create the temporary bitmap}
    TempBitmap := TBitmap.Create;

    try
        {load the bitmap, and extract its palette}
        TempBitmap.LoadFromFile(ExtractFilePath(ParamStr(0))+ 'Background.bmp' );
        GetPaletteEntries(TempBitmap.Palette, 0, 256, Palette);

        {create the palette object}
        DXCheck( FDirectDraw.CreatePalette(DDPCAPS_8BIT or DDPCAPS_ALLOW256,
                                           @Palette, FPalette, nil) );

    finally
        {free our temporary bitmap object}
        TempBitmap.Free;

    end;

```

```
{attach the palette to the primary surface so that it takes affect}
```

```
DXCheck( FPrimarySurface.SetPalette(FPalette) );
```

```
{at this point, offscreen buffers and other surfaces should be created. other
```

```
DirectDraw objects should be created and initialized as well, such as
```

```
palettes. the contents of all surfaces should also be initialized at this
```

```
point}
```

```
FBackground := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+
```

```
Background.bmp' );
```

```
{copy the bitmap to both the primary surface and the backbuffer}
```

```
Source := Rect(0, 0, DXWIDTH, DXHEIGHT);
```

```
FPrimarySurface.BltFast(0, 0, FBackground, Source, DDBLTFAST_NOCOLORKEY or
```

```
DDBLTFAST_WAIT);
```

```
FBackBuffer.BltFast(0, 0, FBackground, Source, DDBLTFAST_NOCOLORKEY or
```

```
DDBLTFAST_WAIT);
```

```
.
```

```
.
```

```
.
```

```
end;
```

---

**Listing 5-2:** *Displaying a bitmap without a palette*

---

```
procedure TFormDXAppMain.FormActivate(Sender: TObject);
```

```
var
```

```
.
```

```
.
```

.  
begin

.  
.  
.  
{retrieve the surface}

DXCheck( FPrimarySurface.GetAttachedSurface(DDSCaps, FBackBuffer) );

{at this point, offscreen buffers and other surfaces should be created. other  
DirectDraw objects should be created and initialized as well, such as  
palettes. the contents of all surfaces should also be initialized at this  
point}

FBackground := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+  
Background.bmp');

{copy the bitmap to both the primary surface and the backbuffer}

Source := Rect(0, 0, DXWIDTH, DXHEIGHT);

FPrimarySurface.BltFast(0, 0, FBackground, Source, DDBLTFAST\_NOCOLORKEY or  
DDBLTFAST\_WAIT);

FBackBuffer.BltFast(0, 0, FBackground, Source, DDBLTFAST\_NOCOLORKEY or  
DDBLTFAST\_WAIT);

.  
.  
.  
end;

---

## Disconnecting Palettes

You can disconnect any palette from a surface by simply calling the `SetPalette` method again with a parameter of `nil`. However, you can directly attach a new palette to a surface without disconnecting any current palette first. When a palette is connected to a surface, its reference count is incremented by one. Attaching the same palette to other surfaces increases the reference count, but attaching the same palette to the same surface again does not increase it.

## Palettes and GDI Functions

When mixing GDI functions with DirectX, GDI functions will use whatever palette is attached to the primary surface at the time the function is called. We must be cautious, since the functions we have examined that load a bitmap into a surface take advantage of the `TBitmap` object, which ultimately uses GDI functions for drawing the bitmap. If there is not currently a palette attached to the primary surface, the current hardware palette will be used. By default, GDI will remap bitmap bits to the closest matching colors available when the bitmap is copied to a surface's DC. Sometimes, the results are acceptable. However, if you subsequently attempt to attach a palette to the primary surface, the bitmap bits will have been translated by GDI and will not match up with the palette entries, resulting in visual garbage. When loading bitmaps using surface DCs and the `TBitmap` object, always load and attach the bitmap's palette to the primary surface first.



---

**Note:** Unlike graphics programming under GDI, there is no color conversion performed when blitting a bitmap from one surface to another. All pixels will be interpreted in terms of the destination surface's palette. Therefore, you need only attach a palette object to the primary surface.

---

## Palette Animation

Apart from the blitting speed benefits realized from having only 8 bits per pixel, palette animation is another great benefit of palettized display modes. Palette animation is a cheap yet powerful method for achieving effects that would be very slow under higher video modes. Animations such as running water and fades are easily implemented using palette animation; the same trick would require constant blitting under non-palettized video modes. We'll examine a cool text effect achievable through palette animation, as well as a generic screen transition where a bitmap fades in and out.

### Animating the Palette

Palette animation works by replacing the color value of a specific color table entry with another value. This instantly changes the color of any pixel drawn using that color table entry to the new color. There are two methods of accomplishing this. The first method requires the creating of multiple palette objects. Each palette object contains different color values in the specific color table entries to be animated. To achieve the animation, you simply call the `SetPalette` method of the primary surface in a continuous loop, changing the palette object used each time. While this will work, it essentially changes all 256 color registers each time, thus making it somewhat slow. It is uncommon to change all 256 color registers during each iteration of a palette animation, so this method is not recommended.

The second method is a more practical solution in that it is generally faster and doesn't waste as many resources as multiple palette objects would. It takes advantage of two methods of the `IDirectDrawPalette` object: the `GetEntries` and the `SetEntries` methods. The `GetEntries` method is defined as:

```
function GetEntries (  
  
    dwFlags: DWORD;           // not currently used, set to 0  
  
    dwBase: DWORD;           // the starting entry  
  
    dwNumEntries: DWORD;     // the number of entries to retrieve  
  
    lpEntries: pointer       // a pointer to an array of TPaletteEntry structures  
  
): HRESULT;
```



This function fills the array of TPaletteEntry structures specified in the fourth parameter with the color values of the palette's entries, starting from the entry specified in the second parameter. We can modify these color values as necessary, and then place them back into the palette using the SetEntries method, which is defined as:

```
function SetEntries (  
  
dwFlags: DWORD;           // not currently used, set to 0  
  
dwStartingEntry: DWORD;   // the starting entry  
  
dwCount: DWORD;          // the number of entries to set  
  
lpEntries: pointer        // a pointer to an array of TPaletteEntry structures  
  
): HRESULT;              // returns an error code
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

This function simply sets the color values in the palette to those specified in the array of TPaletteEntry structures specified in the fourth parameter, starting from the entry specified in the second parameter. When this action is performed on a palette attached to the primary surface, the colors of pixels drawn using these palette entries are immediately changed, and palette animation has occurred.



---

**Note:** When using this method to change a range of color entries, it is faster to change several entries that are adjacent (i.e., entries 5, 6, 7, 8, 9), as opposed to changing non-contiguous color entries (i.e., 13, 57, 73, 109).

---

It would be easy to demonstrate this technique by filling the screen with a color, say palette slot 5, and then using GetEntries and SetEntries in a loop to continuously change the color in entry 5. Similarly, if displaying a bitmap with a lot of pixels drawn using palette slot 5, this technique could change all of those pixels instantly. This could be useful for creating several variations of a monster using only one bitmap. In level one, the monster could be blue, but in level two, you can change the color table entry to red, and using the same bitmap, display a red version of the monster, perhaps indicating that it is a little tougher.

To demonstrate the technique of palette animation, we're going to do something a little more ambitious. Since the GetEntries and SetEntries methods allow us to change a range of color entries, it would be cool if we could draw a bitmap using a color gradient, and then rotate those color entries so the gradient appears to move. Using this technique is also a good way to implement moving water effects.

We start by loading our bitmap in and displaying it. Then, in our main animation loop, we use GetEntries to retrieve the range of color values to be modified. We then save the value in the last entry, slide all the other entries up one, and place the saved entry in the first spot. This rotates the color entries as illustrated in Figure 5-2.



**Figure 5-2:** Rotating color entries

We then place the modified entries back into the palette by calling `SetEntries`. The bitmap used in this example is simply the words “Delphi Rocks” drawn using a red gradient. Rotating the palette entries using this technique makes the gradient appear to move, and results in a very cool text animation effect. Listing 5-3 shows the heart of the example code that performs the actual palette animation.

*Listing 5-3: Animating a palette*

---

```
procedure TfrmDXAppMain.DrawSurfaces;
var
    {our palette slots to be modified}
    Entries: array[0..29] of TTrueColor;

    {a temporary palette slot holder}
    Temp: TTrueColor;

    {loop variable}
    iCount: Integer;
begin
    {retrieve the range of palette entries to be animated}
    FPalette.GetEntries(0, 9, 30, @Entries);

    {rotate the color entries up one, moving the last palette
    slot into the first slot}
    Temp := Entries[29];
    for iCount := 29 downto 1 do
        Entries[iCount] := Entries[iCount-1];
    Entries[0] := Temp;

    {place the rotated entries into the palette. this will cause
    the colors in the displayed image to change, creating the illusion
    of movement}
```

```
FPalette.SetEntries(0, 9, 30, @Entries);
```

```
end;
```

---

## Screen Transitions

As another use for palette animation, let's examine a very clean, professional screen transition effect known as the fade. A fade is a cinematic effect where an image slowly appears from a solid color, such as black or white.



**fade:** A transition effect where an image slowly appears out of a solid color, such as black or white.

Unlike the previous technique, we'll be performing palette animation on all 256 entries in the palette so that the entire image is affected (however, you could use only a subset of the entries in the palette to fade only portions of the image, but that's beyond the scope of this exercise). We must start by retrieving the entire color table into a permanent array that will hold all of the color's original values. The trick to this technique is in setting the values of a "work" array to a percentage of the values stored in the permanent, or "original colors" array, and then setting the palette's colors to the work array's values. We could simply change the red, green, and blue color components by a constant value, but that would make some colors fade in and out faster than others, causing the image to appear to change as it fades in or out. By setting the individual color elements as a percentage of the original value for that entry, we achieve a smooth transition effect. To fade in from black, the work array values are started at zero, and then gradually increased until they reach the original values. Fading out just reverses the procedure. Listing 5-4 demonstrates the heart of this technique.

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

7.0

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)



**Tip:** Most games require a standardized palette such that all graphics are based on the same palette. If you are creating a game that features many different screens, you could create all backgrounds and artwork with individual palettes for each screen. Then, you could use the fade in/out screen transition and change the palette based on the new screen. This would allow you to use multiple palettes in a game, providing for a richer depth of color.

**Listing 5-4: Performing screen transitions**

```
procedure TfrmDXAppMain.FadeEffect(FadeIn: Boolean);  
  
const  
  
    STEPS = 100;    // we want 100 fade steps  
  
var  
  
    iCount, iCount2: Integer;  
  
begin  
  
    {if we are fading in from black...}  
  
    if FadeIn then  
  
    begin  
  
        for iCount := 0 to STEPS do  
  
        begin  
  
            {modify the work' palette to a percentage of each original palette's
```

```

        color value, based on our current step}

for iCount2 := 0 to 255 do
begin
    AnimPalette[iCount2].R := Trunc(OrgPalette[iCount2].R*(iCount/STEPS));
    AnimPalette[iCount2].G := Trunc(OrgPalette[iCount2].G*(iCount/STEPS));
    AnimPalette[iCount2].B := Trunc(OrgPalette[iCount2].B*(iCount/STEPS));
    AnimPalette[iCount2].A := 0;
end;

{plug the palette into the hardware, causing the colors to take effect}
FPalette.SetEntries(0, 0, 256, @AnimPalette);

{pause for a fraction of a second}
Sleep(10);
end;
end
else
{if we are fading out to black...}
begin
    for iCount := 0 to STEPS do
    begin
        {begin the work' palette to a percentage of each original palette's
        color value, based on our current step. notice this is opposite from
        the previous block}
        for iCount2 := 0 to 255 do
        begin
            AnimPalette[iCount2].R := OrgPalette[iCount2].R-
                Trunc(OrgPalette[iCount2].R*(iCount/STEPS));
            AnimPalette[iCount2].G := OrgPalette[iCount2].G-
                Trunc(OrgPalette[iCount2].G*(iCount/STEPS));
            AnimPalette[iCount2].B := OrgPalette[iCount2].B-

```

```

Trunc(OrgPalette[iCount2].B*(iCount/STEPS));

AnimPalette[iCount2].A := 0;

end;

{plug the palette into the hardware, causing the colors to take effect}

FPalette.SetEntries(0, 0, 256, @AnimPalette);

{pause for a fraction of a second}

Sleep(10);

end;

end;

end;

```

---

## Summary

In this chapter, we discussed how the `IDirectDrawPalette` object provides an application with hardware palette management functionality within an exclusive, full-screen `DirectDraw` environment. Keep these points in mind when programmatically manipulating the hardware palette:

- The hardware palette is only used under palettized display modes. Thus, you will only be concerned with the palette when creating 256-color games.
- Palettes give us access to a large range of colors while keeping memory requirements to a minimum. This also improves application performance as moving pixels between surfaces is very efficient. Under palettized display modes, we also have access to palette animation, which makes certain special effects easy to implement.
- The colors in the palette could be defined programmatically, but typically they are initialized from the palette of a bitmap containing graphics used in the game.
- By default, only 254 palette entries can be used. `DirectDraw` predefines palette slot 0 as black and palette slot 255 as white. You can use all 256 entries by specifying the `DDPCAPS_ALLOW256` flag in the first parameter of the call to `CreatePalette`. However, bear in mind that some video cards use palette slot 0 to define the overscan color, the color of the area around the usable display area.
- If a surface becomes lost and must be restored, any attached palettes are unaffected, and do not need to be reconnected after a surface is restored.
- Palettes do not move when surfaces are flipped, so it is unnecessary to attach palettes to both a primary surface and backbuffer.
- Only the palette associated with the primary surface will affect the actual hardware palette registers. Thus, you must attach the newly created palette to the primary surface before the new colors take effect. Palettes can actually be attached to any palettized surface, and the same palette can be attached to more than one surface, but only the palette attached to the primary surface will actually affect the display. If a palette is not attached to the primary surface, the current entries in the hardware palette (left over from the last application) will be used.
- When mixing GDI functions with `DirectX`, GDI functions will use whatever palette is attached to the primary surface at the time the function is called.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).





**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

7.0

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 6 Sprite Techniques

This chapter covers the following topics:

- The illusion of animation
- The illusion of motion
- Transparency issues
- Collision detection
- Animation issues

As we discussed in Chapter 2, The Anatomy of a Game, the most important graphical elements of any game are those graphics with which the user interacts, known as sprites. Sprites are such an integral part of any game that knowledge of their implementation and utilization is paramount. Sprites provide the animated elements of a game that make the gaming environment come alive. These can take the form of images that the player directly interacts with, such as a spaceship, or background elements that are non-interactive yet provide animation that makes the environment richer, deeper, and more believable. Without sprites, all you would have is a static bitmap, which would be quite boring.

The natural discussion of sprite implementation also leads to such sprite related issues as collision detection. With the exception of the most simplistic, most games need to detect when one sprite comes into contact with another. This would be important when, say, a missile launched from your ship comes into contact with a space-borne hunk of rock. Most players would expect the game to replace the rock with several smaller rocks and increment their score. Thus, it is important to understand the techniques by which sprite collision can be detected.

In this chapter, we examine several techniques for implementing sprites, discussing animation and movement as well as collision detection and timing issues.

### The Illusion of Animation

The method by which a game displays an animated image on the video screen has actually been around for many decades, even before computers were a household item. Indeed, if you have any knowledge of how movies or cartoons work, you already know the fundamentals of sprite animation.

The basic premise is simple. A series of images is required, where each image shows the character or object to be animated in a slightly different position than the image before it, like a filmstrip. Each image is known as a frame.



**frame:** A single image in a series of images to be animated.

When each image is rapidly displayed one after another, the human brain expects to see smooth motion, and any gaps in movement from one frame to another (provided it is not too large) will be automatically filled in. In essence, it's really nothing more than a complex visual illusion.



**Figure 6-1:** Movie frames

Creating the illusion of animated images using the computer is almost identical to this technique. Instead of a filmstrip, a series of bitmaps is constructed in which each bitmap contains an image of the character or object in a slightly different position. If each bitmap is copied to the video screen in rapid succession, the illusion of animation will be achieved. This is known as frame animation.



**frame animation:** The process of producing animation by rapidly displaying a series of images depicting an object or character in slightly different positions each frame.

Unlike a filmstrip, which contains images of a scene in its entirety, the series of bitmaps to be animated usually contains only an image of the sprite itself. In fact, these images usually only contain images of the sprite in different positions without regard to its actual location in the scene. Location is controlled by other methods, which we'll see shortly. Of course, a series of bitmaps that does depict an entire scene could be used under special circumstances, such as showing an opening animation or perhaps an animation when the user is moving from one level to another. However, for the purposes of this chapter, we will examine only sprite animation techniques.

## Constructing the Animation Frames

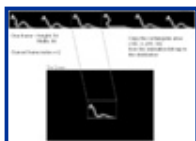
The first step that is required is to create the series of bitmaps that will depict the animated object or character. As explained above, each bitmap should display the object or character in a slightly different position. You could store these as individual bitmap files, but the most common practice is to store all of these images in one bitmap, with each frame of the animation in the correct sequence, one after another. Another common practice is to make all frames of the animation equal in size. This simplifies the code when copying the image from the bitmap to the screen, as demonstrated in the example to follow.



**Figure 6-2:** An animated image bitmap

## Producing the Animation

The actual process of producing animation is simple. We start by defining a variable that will hold the current frame number, starting at 0. Through each iteration of the game loop, we increment the frame number by 1. If it goes beyond the actual number of frames, reset it to 0. Then, we copy the appropriate frame out of the bitmap onto the screen, using this frame number as an index.



**Figure 6-3:** Producing animation

Since each frame of the animation is exactly the same size, it is very easy to calculate the appropriate place in the bitmap from which to copy the image. Assuming that we've arranged our frames horizontally in the bitmap, we only need to compute the X offset within the bitmap at which to start and end the copy. These are calculated using the following formulas:

```
XStart := FrameNum * WidthOfOneFrame;
```

```
XEnd := XStart + WidthOfOneFrame;
```

These offsets can easily be used with the Blt or BltFast methods of the IDirectDrawSurface object to copy the correct frame to the backbuffer. Incrementing the frame number each iteration through the game loop causes a new frame of the animation to be displayed each time. If this occurs in rapid succession, and the bitmaps are displayed at the same spot, we achieve the illusion of animation, as the example below illustrates.

***Listing 6-1: Simple frame animation***

---

```
procedure TfrmDXAppMain.DrawSurfaces;
var
    SourceRect: TRect;           // determines the area to copy from in the source bitmap
    SrfcDC: HDC;                // these two are used to make text drawing easier
    TempCanvas: TCanvas;

begin
    {increment the overall frame number}
    Inc(FFrameNum);

    {start the frame number from the beginning if it goes beyond the
    number of frames in the animation}
    if FFrameNum > 5 then
        FFrameNum := 0;

    {blit the entire snake frame bitmap to illustrate the animation frames}
    SourceRect := Rect(0, 0, 540, 50);
    FBackBuffer.BltFast(50, 75, FImages, SourceRect,
                        DDBLTFAST_NOCOLORKEY OR DDBLTFAST_WAIT);

    {blit the current frame of the animation}
    SourceRect := Rect(FFrameNum*90, 0, (FFrameNum*90)+90, 50);
    FBackBuffer.BltFast(275, 215, FImages, SourceRect,
                        DDBLTFAST_NOCOLORKEY OR DDBLTFAST_WAIT);
```

- .
- .
- .

end ;

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## The Illusion of Motion

Creating the illusion of animation was simple enough. However, even the most complex and beautifully rendered animation would quickly get boring if it stayed in the same spot. So, how do we go about moving the animated sprite across the screen to produce a more interesting and believable animation?

Again, the process is quite simple. On the most primitive level, when the application prepares to draw the next frame of animation, simply change the horizontal or vertical coordinate (or both) at which the image will be drawn into the destination. If these coordinates keep changing, the animated image will be drawn at different positions, creating the illusion of motion.

## Sprite Attributes

Before we begin, a few items of information need to be tracked for each sprite. In particular, its current X (horizontal) and Y (vertical) coordinate must be tracked so the application knows where to draw the sprite. A horizontal and vertical velocity could also be stored. These velocities would indicate the number of pixels to move the sprite, and would be added to the X and Y coordinates on each iteration through the game loop. If we also tracked the current frame of animation, the number of frames in the animation, and a pointer to the bitmap or DirectDraw surface containing the sprite images, we could create a basic sprite object that might look like this:

```
TSprite = class

    FXPos, FYPos,                // tracks the current position

    FXVel, FYVel,                // tracks the velocity

    Height, Width,              // sprite frame height and width

    FCurFrame,                 // tracks the current frame number
```

```

FFrameNum: Integer;           // tracks the number of frames

FImages: IDirectDrawSurface4; // a pointer to the sprite images

end;

```

## The Animation Cycle

The effect we want to produce is the image of an object moving across the screen in a believable manner. To achieve this illusion of motion, we will tackle the process of animation by breaking it into three distinct steps: erasing the sprite image, updating the sprite position, and drawing the sprite image.

If we simply move the sprite and continue to draw it on the screen without ever erasing it, the previous sprite images will still be visible, ruining the animation effect. We want our animation to look like a single object moving about as one would expect to see in a cartoon or movie. Thus, the first thing we must do is erase the previous sprite image. For this example, we will simply clear the entire surface by performing a color fill using the Blt method. The IDirectDrawSurface4's Blt method is defined as:

```

function Blt(

lpDestRect: PRect;           // the destination rectangle

lpDDSrcSurface: IDirectDrawSurface4; // the source surface

lpSrcRect: PRect;           // the source rectangle

dwFlags: DWORD;             // control flags

lpDDBltFx: PDDDBltFX        // a pointer to a TDDDBltFX structure

): HRESULT;                  // returns a DirectX error code

```

The first parameter describes the rectangular area in the destination surface to which the image data is copied. The second parameter identifies the surface whose image data is being copied. The third parameter describes the rectangular area in the source surface from which the image data is copied. Note that if these rectangular areas are not the same size, the resulting image will be scaled as required. The fourth parameter is a series of flags indicating which members of the final parameter are valid and generally control the behavior of the blit. The final parameter is a pointer to a TDDDBltFX structure. This rather large and complex structure contains information for various special graphical effects, such as color fills and raster operations. We'll see an example of this function's use to perform a color fill in Listing 6-2.



**Note:** In the DDUtil unit, we've created a function called ColorFill that uses the Blt method to fill a region with a color. You specify a surface, the color in the form of a TColor, and a rectangular region to be filled (or nil to fill the entire surface). This allows the developer to fill an area with a solid color using only one line of code.



**Figure 6-4:** The animation cycle

Next, we simply add the horizontal and vertical velocities to the horizontal and vertical coordinates of the sprite. At this point, we may do some simple calculations to see if the sprite has moved off screen or out of bounds, and adjust its velocities accordingly. We then increment the frame number and draw the appropriate frame of the sprite image at the new coordinates. Figure 6-4 illustrates this process.

The value of the sprite velocities should take into account the type of animation being performed. If the animation is a top-down view of a ship flying in space, the horizontal and vertical velocities could be adjusted to make the sprite move all around the screen.

However, if the animation is a side view of a walking character, we would probably only want to adjust the horizontal velocity (unless, of course, the character is jumping). The velocity value must also be adjusted to make the animation look more believable (i.e., a character would move faster when running than when walking).

Implementing this animation method is little more than an extension of the previous frame animation example, as the following listing demonstrates.

*Listing 6-2: Simple sprite movement*

---

```
procedure TfrmDXAppMain.DrawSurfaces;
var
    SourceRect: TRect
    SrfcDC: HDC;
    TempCanvas: TCanvas;
    BltFx: TDDBltFx;
begin
    {erase the last frame of animation by using a simple color fill to
    'black out' the entire backbuffer surface}
    FillChar(BltFx, SizeOf(TDDBltFx), 0);
    BltFx.dwSize := SizeOf(TDDBltFx);
    BltFx.dwFillColor := 0;
    FBackBuffer.Blt(nil, nil, nil, DBLT_COLORFILL or DBLT_WAIT, @BltFx);

    {increment the frame number and update the sprite's position}
    with FSprite do
    begin
        Inc(FCurFrame);
```

```
{make sure we roll over the frame number as necessary}
if FCurFrame > FFrameNum then
    FCurFrame := 0;

{add the velocities to the positions. our sprite will only move
 horizontally, so we do not need to bother with the vertical values}
FXPos := FXPos + FXVel;

{see if the sprite is moving offscreen, and adjust
 its position accordingly}
if FXPos < 0 then
    FXPos := DXWIDTH - 90;

{blit the current frame}
SourceRect := Rect(FCurFrame*90, 0, (FCurFrame*90)+90, 49);
FBackBuffer.BltFast(FXPos, FYPos, FImages, SourceRect,
                    DDBLTFAST_SRCCOLORKEY OR DDBLTFAST_WAIT);

end;

.
.
.

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)



Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

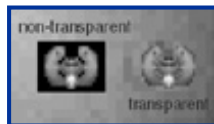
**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Transparency Issues

So far, the examples we've seen all demonstrate sprite animation against a solid black background. This may work for some games, but what if we wanted to have some sort of cool background graphic behind our sprites, like a starfield or perhaps a forest, mountains, or other type of setting? This not only makes the overall animation more visually stimulating, but it improves its believability and makes the overall gaming experience more realistic. To do this, we need to draw the frames of the sprite image using some technique for displaying the background through the parts of the sprite image that are supposed to be transparent. Figure 6-5 illustrates the difference between a sprite image being displayed with and without transparent pixels.



**Figure 6-5:** Sprite transparency

Implementing transparency such as this is another broad topic in graphics programming that's been widely studied. There are probably hundreds, if not thousands, of methods to display a graphic with transparent areas. The trick is to do it as quickly as possible so it won't negatively affect the performance of the game. We will examine a few of the most common techniques used to display graphics with transparent pixels. These will include DirectDraw color keying and direct pixel checking.

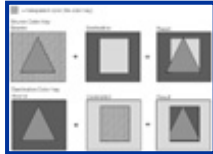


**Tip:** You can potentially optimize your application by including several different methods for blitting graphics in the game engine. It is perfectly normal to have one blitting function for graphics with no transparency (i.e., tiled graphics such as floors, or textures), a separate general blitting function for sprites with transparent pixels, and then perhaps a highly optimized assembly blitter that's used for displaying graphics when optimum speed is required. The point is that you do not have to use only one blitting technique for all graphics in your game engine; use many blitting techniques for optimization and greater application performance.

## Color Keying

Color keying is the DirectX way of implementing transparency. With color keying, you specify a color that will be transparent. When DirectDraw blits this graphic onto the background, any pixels whose color matches that of the color key will not be drawn to the destination. Thus, the background will “show through” those pixels, as if they were transparent. The effect is similar to a technique that film makers use called blue screen. In a blue screen effect, an actor stands in front of a solid blue background. Later, the film is combined with a background image, which shows through all of the blue areas. Blue, in this case, is considered the color key.

Color keying is very easy to implement in DirectX. It’s supported by the HEL, so you know it will always work, and if supported by the hardware, it’s incredibly fast. The only drawback to color keying is in the fact that every pixel in your image that is the color of the color key will be transparent. Thus, that color cannot be used for anything opaque in the image. However, the loss of one useful color out of 256 is easy to live with.



**Figure 6-6:** Color keying

Under DirectX, there are actually two types of color keys: source and destination. In source color keying, the source image (the image to be copied) is the one containing the transparent pixels. When the source image is copied onto the destination image, the color keyed pixels in the source image are replaced by the pixels in the destination. With destination color keys, the destination image is the one containing the transparent pixels. When the source image is copied onto the destination image, the color keyed pixels in the destination are replaced by the pixels in the source.

In our examples, we will cover only source color keying, as destination color keying is not supported by the HEL and requires hardware support.

### Setting a Color Key

We specify a color key by calling the IDirectDrawSurface4’s SetColorKey method. SetColorKey is defined as:

```
function SetColorKey (  
  
dwFlags: DWORD; // indicates the type of color key  
  
lpDDColorKey: PDDColorKey // the transparent color(s)  
  
) : HRESULT; // returns a DirectX error code
```

The first parameter takes a flag that indicates the behavior of the color key. For our purposes, we’ll set this to DDCKEY\_SRCBLT, indicating a source color key. The second parameter is a pointer to a TDDColorKey structure that defines the color or colors to be interpreted as transparent. The TDDColorKey record is defined as:

```
TDDColorKey = packed record  
  
dwColorSpaceLowValue: DWORD; // low boundary of color space  
  
dwColorSpaceHighValue: DWORD; // high boundary of color space
```

```
end;
```

The values of these fields are specified in the color format of the surface. Since we're using 256-color (or palettized) surfaces, we simply indicate the index of the color that we wish to be transparent. This record structure allows us to define a range of colors that will be transparent. This is primarily useful when working with video data, as shadows and other slight variations in color due to digitizing the video result in a background that will be a range of colors as opposed to one single color. Game artwork should usually be drawn with only one transparent color, and for our purposes, we will define only one color as transparent by setting both record elements to the same value.

In our examples, we'll only set the color key of surfaces that contain source images to be blitted over a background, such as sprite images. For example, to set the color key of an arbitrary surface to palette index 0, we only need a few lines of code:

```
var  
  
    ColorKey: TDDColorKey;  
  
begin  
  
    ColorKey.dwColorSpaceLowValue   := 0  
  
    ColorKey.dwColorSpaceHighValue  := 0  
  
  
    DirectDrawSurface.SetColorKey(DDCKEY_SRCBLT, @ColorKey);  
  
end;
```



*Note:* We have implemented a function in the DDUtil unit called `DDSetColorKey` that allows the developer to set the color key for a surface in one line of code. This function was translated from a similar function in the DirectX SDK. You simply pass in the surface and a color in the form of a `TColor`. The function uses a complex technique for matching the specified color to a format appropriate for the pixel format of the surface, so the function can be used for surfaces of any color depth.

---

## Using Color Keys

Now that we have our color key set up, we can extend the previous sprite example to display the sprite image over a background. After we load the sprite images, we set their color key as described above. Then, when we actually draw the sprite using the `BltFast` method, we specify the `DDBLTFAST_SRCCOLORKEY` in its `dwTrans` parameter to indicate that the source surface has a color key that should be taken into account during the blit.

Additionally, we must also load the image to be used as our sprite background. Now, as discussed previously, during the animation cycle we must erase the last frame of the sprite animation. Instead of simply filling the entire surface with black as we did previously, this time we'll draw the background image into the backbuffer. The background image will fill the entire backbuffer, and thus will erase the last frame of animation.

*Listing 6-3: Sprite transparency through color keying*

---

```

procedure TfrmDXAppMain.FormActivate(Sender: TObject);
var
    .
    .
    .

    {the color key record}
    ColorKey: TDDColorKey;
begin
    .
    .
    .

    {create a palette based on the sprite bitmap}
    FPalette := DDLoadPalette(FDirectDraw,
ExtractFilePath(ParamStr(0))+`SnakeFrames.bmp`);

    {attach the palette to the primary surface so that it takes effect}
    DXCheck( FPrimarySurface.SetPalette(FPalette) );

    {load in the bitmap containing the background and sprite images}
    FImages := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+
        `SnakeFrames.bmp`);

    {load the background bitmap}
    FBackground := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+
        `Background.bmp`);

```

```

    {initialize the color key record...}
    ColorKey.dwColorSpaceLowValue := 0;
    ColorKey.dwColorSpaceHighValue := 0;

    {...and set the source color key}
    FImages.SetColorKey(DDCKEY_SRCBLT, @ColorKey);

    {initialize the sprite object}
    FSprite := TSprite.Create;
    FSprite.FXPos := DXWIDTH - 90;
    FSprite.FYPos := 215;
    FSprite.FXVel := -1;
    FSprite.FYVel := 0;
    FSprite.FCurFrame := 0;
    FSprite.FFrameNum := 5;
    FSprite.FImages := FImages;

    .
    .
    .

end;

procedure TfrmDXAppMain.DrawSurfaces;
var
    SourceRect: TRect
begin
    {erase the last frame of animation}

```

```

SourceRect := Rect(0, 0, DXWIDTH, DXHEIGHT);

FBackBuffer.BltFast(0, 0, FBackground, SourceRect,

                    DBBLTFAST_NOCOLORKEY OR DBBLTFAST_WAIT);

{increment the frame number and update the sprite's position}

with FSprite do

begin

    Inc(FCurFrame);

    {rollover the frame number as necessary}

    if FCurFrame > FFrameNum then

        FCurFrame := 0;

    {add the velocities to the positions. our sprite will only move

    horizontally, so we do not need to bother with the vertical values}

    FXPos := FXPos + FXVel;

    {see if the sprite is moving offscreen, and adjust

    its position accordingly}

    if FXPos < 0 then

        FXPos := DXWIDTH - 90;

    {blit the current frame, taking into account a source color key}

    SourceRect := Rect(FCurFrame*90, 0, (FCurFrame*90)+90, 49);

    FBackBuffer.BltFast(FXPos, FYPos, FImages, SourceRect,

                        DBBLTFAST_SRCOLORKEY OR DBBLTFAST_WAIT);

end;

end;

```

---

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Pixel Checking**

A somewhat more direct way of implementing transparency, this method draws the sprite image one pixel at a time. As it is drawing individual pixels, it checks the pixel it will be drawing to see if its color matches the specified transparent color. If so, it skips drawing that pixel entirely; otherwise it draws the pixel into the destination.

Implementing this method is a bit more involved than color keying. By its very nature, it is pointer intensive and requires some creative pointer manipulation. It also requires more intimate knowledge of the pixel format of both the source and destination surfaces. It is slower than color keying, but through clever optimization can be as fast as non-hardware assisted color keying. However, the biggest benefit that a developer will see from this method is that it allows the creation of incredible special effects, such as lighting and translucency.

**Locking a Surface**

This method requires direct access to the memory that contains both the source bitmap and the destination, which is usually the backbuffer surface. Using normal GDI functions, this would be impossible. This is one of the things that makes DirectDraw so attractive to game developers.

To retrieve a pointer to surface memory, we must use the Lock method of the IDirectDrawSurface4 object. Using this method, we can access the entire surface memory or provide the coordinates for accessing only a specific rectangular area of the memory. The Lock method is defined as:

```
function Lock (  
    lpDestRect: PRect; // the surface region to lock  
    out lpDDSurfaceDesc: TDDSurfaceDesc2; // a TDDSurfaceDesc2 structure  
    dwFlags: DWORD; // indicates locking behavior  
    hEvent: THandle // unused  
): HRESULT; // returns a DirectX error code
```

The first parameter indicates the rectangular area of the surface to lock; specifying nil in this parameter locks the entire surface. The second parameter is a TDDSurfaceDesc2 structure that will be initialized with relevant details about the surface. In particular, the lpSurface member of this structure will contain a pointer to the upper left pixel of the surface memory. Other members of this structure will contain additional information about the surface, such as its height, width, and stride. The third parameter is a series of flags controlling access to the locked memory, such as read-only or write-only. The final parameter is unused and should be set to zero.

As an example, the code for retrieving a pointer to our backbuffer memory would look something like:

```
var
    BackSurfaceDesc: TDDSurfaceDesc2;
begin
    {initialize the size member}
    BackSurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc2);

    {lock the backbuffer memory}
    FBackBuffer.Lock(nil, BackSurfaceDesc, DDLOCK_WAIT or DDLOCK_SURFACEMEMORYPTR, 0)
```



*Note:* It is very important to always set the size member of any record structure that has one before using it in any function call.

---

As with all DirectX functions, it is usually a good practice to check the return code from a call to the Lock method. The DDLOCK\_WAIT flag will cause the Lock method to wait until the surface is available and no longer being accessed by the hardware blitter. However, if the surface is locked by another thread or process, this function will return immediately with a DDERR\_SURFACEBUSY error regardless of the DDLOCK\_WAIT flag.

## Accessing the Memory

When the Lock method returns, the lpSurface member of the TDDSurfaceDesc2 structure will contain a pointer to the surface memory. Exactly how this is used depends entirely on the pixel format of the surface. Since all of our examples deal with 256-color palettized display modes, pixels will be stored as bytes. Therefore, if we typecast the lpSurface member as a pointer to a byte, writing a value at the location reference by this pointer will change one byte on the surface, and thus one pixel in the image the surface contains.

In practice, it is useful to access this pointer as a pointer to an array of bytes. This allows us to access horizontal bytes as if they were stored in a one-dimensional array. Although perhaps not immediately obvious, the process is simple. We'll need to create a new data type that is declared as an array of byte with only one element, and another data type that is a pointer to this previous type, like so:

```
TBytePtr = array[0..0] of Byte;
PBytePtr = ^TBytePtr;
```

Then, if we set a variable of type PBytePtr to the value of the lpSurface member, we can easily reference the surface memory as an array of bytes:

```

var

    BackMem: PBytePtr;

begin

    BackMem := PBytePtr(BackSurfaceDesc.lpSurface);

```

## Drawing the Sprite

We must lock both the backbuffer surface and the surface containing the sprite images. Once this is done, we iterate through each row of the sprite, and each pixel in the row. For each pixel in the current row being drawn, we check to see if it is equal to the specified transparent color. If it is not, we go ahead and set the appropriate pixel in the destination memory to the value of the pixel being checked. If it is indeed equal to the transparent color, we simply move on to the next pixel in the row. The following example demonstrates an implementation of this technique.

### *Listing 6-4: Drawing a sprite with transparent pixels*

---

```

procedure TfrmDXAppMain.DrawSurfaces;

var

    SourceRect: TRect

    BackSurfaceDesc,

    SpriteSurfaceDesc: TDDSurfaceDesc2;

    BackMem, SpriteMem: PBytePtr;

    Row, Col: Integer;

begin

    {erase the last frame of animation}

    SourceRect := Rect(0, 0, DXWIDTH, DXHEIGHT);

    FBackBuffer.BltFast(0, 0, FBackground, SourceRect,

                        DDBLTFAST_NOCOLORKEY OR DDBLTFAST_WAIT);

    {increment the frame number and update the sprite's position}

    with FSprite do

    begin

        Inc(FCurFrame);

        {rollover the frame number as necessary}

        if FCurFrame > FFrameNum then

            FCurFrame := 0;

```

```

{add the velocities to the positions. our sprite will only move
  horizontally, so we do not need to bother with the vertical values}
FXPos := FXPos + FXVel;

{see if the sprite is moving offscreen, and adjust
  its position accordingly}
if FXPos < 0 then
  FXPos := DXWIDTH - 90;

{-begin blitting the current frame-}
{we need to lock both the backbuffer surface and the sprite image
  surface, so begin by locking the backbuffer}
BackSurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc2);

{we 're going to lock the entire surface}
DXCheck(FBackBuffer.Lock(nil, BackSurfaceDesc,
  DDLOCK_WAIT or DDLOCK_SURFACEMEMORYPTR, 0));

{initialize the byte array pointer to point at the locked memory}
BackMem := PBytePtr(BackSurfaceDesc.lpSurface);

{increment this pointer to the position in the buffer at which the
  sprite should be displayed (based on the sprite's position)}
BackMem := Pointer(Longint(BackMem)+FXPos+(FYPos*BackSurfaceDesc.lPitch));

{now, lock the sprite surface}
SpriteSurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc2);

{again, we'll lock the entire surface}
DXCheck(FImages.Lock(nil, SpriteSurfaceDesc,

```

```
DDLOCK_WAIT or DDLOCK_SURFACEMEMORYPTR, 0));
```

```
{initialize the byte array pointer to point at the locked memory}
```

```
SpriteMem := PBytePtr(SpriteSurfaceDesc.lpSurface);
```

```
{increment this pointer to the position of the current sprite frame}
```

```
SpriteMem := Pointer(Longint(SpriteMem)+(FCurFrame*90));
```

```
{iterate through each row of pixels in the sprite image (our sprite image is  
90 X 50)}
```

```
for Row := 0 to 49 do
```

```
begin
```

```
{iterate through each pixel in the current row}
```

```
for Col := 0 to 89 do
```

```
{if the pixel in the sprite image is not 0 (the transparent color),  
copy it into the destination surface}
```

```
if SpriteMem^[Col] <> 0 then
```

```
BackMem^[Col] := SpriteMem^[Col];
```

```
{move to the next row in the sprite and backbuffer surface memory}
```

```
Inc(BackMem, BackSurfaceDesc.lPitch);
```

```
Inc(SpriteMem, SpriteSurfaceDesc.lPitch);
```

```
end;
```

```
*****
```

```
{we have now copied the sprite, so unlock the surface memory}
```

```
FBackBuffer.Unlock(nil);
```

```
FImages.Unlock(nil);
```

```
end;
```

```
end;
```

---





▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Of special consideration is the part in the rendering loop (the DrawSurfaces method) where the pointers to the backbuffer and sprite surface memory are incremented to point to the next row:

```
{move to the next row in the sprite and backbuffer surface memory}
```

```
Inc(BackMem, BackSurfaceDesc.lPitch);
```

```
Inc(SpriteMem, SpriteSurfaceDesc.lPitch);
```

Among the values returned by the call to the Lock method is the surface's pitch, or stride. This value is the actual distance between two pixels that are vertically adjacent. It is a result of storing memory that is logically arranged in rectangular fashion, but is actually accessed linearly. Often, the pitch is the same as the width of the surface, as one might expect. However, never assume that the pitch is constant, even for different surfaces in the same video memory. Always use the lPitch value to move vertically between rows.



**Caution:** Some display cards can organize memory differently than others, and the resolution and color depth can have dramatic effects on how memory is accessed directly. Therefore, always increment the row by using the lPitch value.

### Unlocking a Surface

With every call to the Lock method, a call to the Unlock method must be made. This allows the locked surface to be accessed again by other threads and processes, and most importantly by the hardware. A surface cannot be blitted or otherwise manipulated by the hardware unless it has been unlocked. After calling Unlock, any pointers to the surface memory will no longer be valid. Therefore, it is necessary to call Lock again during the next rendering cycle to retrieve a valid pointer to surface memory. The Unlock method of IDirectDrawSurface4 is defined as:

```
function Unlock (
```

```
lpRect: PRect
```

```
// the previously locked rectangular area
```

```
: HRESULT; // returns a DirectX error code
```

The only parameter defines the rectangular area that was passed to the initial Lock method. It is possible to lock multiple rectangular, overlapping areas simultaneously. Thus, DirectDraw needs to know which rectangular area to unlock, and the values for this parameter must match those passed to the original Lock method. Pass nil here if you used nil in the Lock method to lock the entire surface.



**Caution:** When accessing surface memory, it is very important to complete rendering quickly and unlock the surface as soon as possible. This is because Windows is effectively stopped between the calls to Lock and Unlock. This can cause things to get backed up rather quickly, causing all sorts of nasty side effects, like system hangs. This won't occur when locking surfaces explicitly created in surface memory, but always try to use tight, efficient code when locking a surface and always remember to call the Unlock method when you no longer require direct memory access.

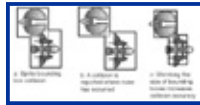
## Collision Detection

All games, at some point, must deal with the situation in which one sprite touches another. For some games, this may be when a bullet strikes an alien or when an animated mouth collides with a glowing white pellet to eat it. A game application must determine when these situations happen, and this is where collision detection comes in.

There exist as many different methods to perform collision detection as there do methods to draw a sprite. Many articles have been written on this subject, exploring the nuances of checking large amounts of sprites against each other and the optimizations required to perform such techniques in a timely fashion. In most cases, however, sprite collision can be detected using one of two techniques: bounding boxes and pixel overlap comparisons.

### Bounding Boxes

The bounding box method of collision detection works by defining a rectangular region around the sprite, its "bounding box." To determine if a collision with another sprite has occurred, the application simply determines if their bounding box rectangles intersect. If indeed the rectangles intersect, a collision has occurred and the game application can respond accordingly, as shown in the following illustration.



**Figure 6-7:** Sprite bounding boxes

Notice that in Figure 6-7b, the intersected rectangles would indicate that a collision has occurred, although clearly the sprites are not touching. This is the inherent drawback to using bounding boxes, in that many times during close calls a collision may be flagged when in fact the sprites never touched. Shrinking the size of the bounding boxes to where they do not encapsulate the entire sprite diminishes this error, as shown in Figure 6-7c. However, the reverse effect can occur, where two sprites have collided although no collision was detected. For the most part, it is better to err on the side of the player and allow the infrequent true collisions to go unresolved. However, for some games where precise collision detection is required, it is necessary to actually check the pixels of one image with the pixels of another to see if they overlap, as we examine shortly.



**Tip:** It is possible to use two or more bounding boxes for a sprite, thereby providing a more accurate approximation of the sprite's surface. The GDI also contains many functions to create regions of different shapes and sizes, including circular and polygonal, which can then be used to detect collisions more accurately.

Windows has highly optimized routines that make manipulating rectangles a breeze. In particular, the `IntersectRect` function determines if two rectangles overlap and returns a rectangle that defines the area of intersection. The `IntersectRect` function is defined as:

```
IntersectRect (
```





```

        Sprite.FYPos+Sprite.Height);

{draw the sprite transparently}

Area := Rect (0, 0,  Sprite.Width, Sprite.Heigh);

FBackBuffer.BltFast(Sprite.FXPos, Sprite.FYPos, Sprite.FImages,
                    Area, DBLTFFAST_SRCCOLORKEY or DBLTFFAST_WAIT);

{draw a red border around the sprite to display its bounding box}

    .
    .
    .

end;

begin

    {erase the previous animation frame}

    Area := Rect (0, 0, DXWIDTH, DXHEIGHT);

    FBackBuffer.BltFast(0, 0, FBackground, Area,
                        DBLTFFAST_NOCOLORKEY or DBLTFFAST_WAIT);

    {move and draw the sprites}

    for iCount := 0 to 1 do

        MoveSprite(Sprites[iCount]);

    {if any of the sprite's bounding boxes intersect, signal a collision. real
     world examples would generally require checking many rectangles against
     each other}

    if IntersectRect(Intersection, Sprites[0].Boundary, Sprites[1].Boundary) then

        Messagebeep(0);

end;

```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**ITKNOWLEDGE.COM**<sup>SM</sup>  
Need IT. Find IT. Know IT.

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Pixel Overlap Comparison**

The bounding box method of collision detection can easily be extended to provide very accurate pixel level collision detection. Basically, a standard bounding box collision detection is performed. It is necessary that the bounding boxes of the sprites be the same dimensions as the sprite images (actually, they could be different, but the resulting offset math makes the pixel offset determinations cumbersome). Once a collision has occurred, it is necessary to determine the exact rectangular intersection area, which is conveniently reported by the IntersectRect function. The coordinates of the upper left corner of the rectangle are subtracted from the coordinates of the upper left corner of each of the sprite image positions. This will give a starting offset of the intersection rectangle relative to the sprites. From this starting offset within the sprite images, a simple embedded loop is used to walk through the pixels in each sprite image, comparing the colors of the pixels in the relative position of each sprite as determined by the intersection rectangle. If any non-transparent pixels match up, then the sprites overlap and a collision has occurred, as illustrated below.



**Figure 6-8:** Pixel level collision detection

This algorithm provides very accurate collision detection. Due to its nature, we must directly access the surface memory as we covered above. This is a slow collision detection technique, but it could possibly be optimized using assembly language. However, for small numbers of sprites, the performance penalty is minimal. The following example demonstrates pixel-accurate collision detection.

*Listing 6-6: Collision detection at the pixel level*

---

```
procedure TfrmDXAppMain.DrawSurfaces;

var

    Intersection, Area: TRect;

    iCount: Integer;
```

```
Sprite1SurfaceDesc,  
Sprite2SurfaceDesc: TDDSurfaceDesc2;  
Sprite1Mem, Sprite2Mem: PBytePtr;  
iRow, iCol: Integer;  
SrfcDC: HDC;  
Hit: Boolean;
```

```
{this function moves and draws a sprite}  
procedure MoveSprite(var Sprite: TSprite);  
var  
    SrfcDC: HDC;  
    TempCanvas: TCanvas;  
begin  
    .  
    .  
    .  
  
end;
```

```
begin  
    {erase the previous animation frame}  
    FBackBuffer.BltFast(0, 0, FBackground, Rect(0, 0, DXWIDTH, DXHEIGHT),  
        DDBLTFAST_NOCOLORKEY or DDBLTFAST_WAIT);  
  
    {move and draw the sprites}  
    for iCount := 0 to 1 do  
        MoveSprite(Sprites[iCount]);  
  
    {retrieve the intersection of the sprite's bounding boxes. if there is
```

indeed an intersection, compare the pixels in intersecting areas of each sprite. if the AND of any two pixels is greater than 0 (our transparent pixel value is 0, so two overlapped image pixels will return a value greater than 0), then there was indeed a pixel level collision.}

if IntersectRect(Intersection, Sprites[0].Boundary, Sprites[1].Boundary) then begin

{at this point, the bounding boxes have collided, so we need to prepare to access the sprite surfaces. we will lock the entire surface of each sprite}

Sprite1SurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc2);

DXCheck(FShip1.Lock(nil, Sprite1SurfaceDesc,  
DDLOCK\_WAIT or DDLOCK\_SURFACEMEMORYPTR, 0));

{initialize the byte array to the intersected rectangular area of the sprite}

Sprite1Mem := PBytePtr(Sprite1SurfaceDesc.lpSurface);

Sprite1Mem := Pointer(Longint(Sprite1Mem)+  
((Intersection.Top-Sprites[0].FYPos)\*  
Sprite1SurfaceDesc.lPitch+  
(Intersection.Left-Sprites[0].FXPos)));

{lock the other sprite's surface}

Sprite2SurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc2);

DXCheck(FShip2.Lock(nil, Sprite2SurfaceDesc,  
DDLOCK\_WAIT or DDLOCK\_SURFACEMEMORYPTR, 0));

{and initialize the other byte array to the intersected rectangular area of the sprite}

Sprite2Mem := PBytePtr(Sprite2SurfaceDesc.lpSurface);

Sprite2Mem := Pointer(Longint(Sprite2Mem)+  
((Intersection.Top-Sprites[1].FYPos)\*  
Sprite2SurfaceDesc.lPitch+

```

        (Intersection.Left-Sprites[1].FXPos)));

{begin determining if a collision has taken place.}
Hit := FALSE;
for iRow := 0 to (Intersection.Bottom-Intersection.Top)-1 do
    for iCol := 0 to (Intersection.Right-Intersection.Left)-1 do
        {if the pixels in the intersected rectangular areas of
        both sprites is something other than a transparent
        pixel, a collision has occurred}
        if Sprite1Mem^[iRow*Sprite1SurfaceDesc.lPitch+iCol] and
            Sprite2Mem^[iRow*Sprite2SurfaceDesc.lPitch+iCol] > 0 then
            Hit := TRUE;

{visually indicate any collisions}
FBackBuffer.GetDC(SrfcDC);
try
    if Hit then
        TextOut(SrfcDC, 10, 460, 'Collision!', Length('Collision!'));
finally
    FBackBuffer.ReleaseDC(SrfcDC);
end;

{we no longer need access to surfaces, so unlock them}
FShip1.Unlock(nil);
FShip2.Unlock(nil);
end;
end;

```

---

## Animation Issues

Apart from getting sprites to move and animate, game developers are confronted with dozens of sprite-based issues, depending on the type of game. Complex sprites may have hundreds of animation frames, segregated into different

actions, all of which must be triggered at the appropriate time by the game state. Collision detection may be a big issue, made even worse by large amounts of sprites that can move around in a virtual world several times the size of the screen. These and other animation issues are sure to appear as you develop your Delphi game masterpiece.

There are a few animation issues that are fairly common among different games, and generally have straightforward resolutions. You've probably noticed that our previous sprite examples demonstrating animation ran incredibly fast, so fast that the animation was unbelievable. This is an issue of timing. Implementing a method to slow down the animation of a sprite is very elementary, and is little more than an extension of our previous sprite movement algorithm.

Another topic to consider is the concept of z-order, or depth. Some games may exhibit many sprites in which collision is infrequent or unimportant. In this instance, it is important to consider which sprites should be drawn first, as the order in which sprites are drawn will connote depth to the viewer. Those sprites that are farthest away should naturally be drawn under those sprites that are closer. Again, implementing such functionality is relatively straightforward.



**z-order:** The position along the z, or third-dimensional, axis. This is an imaginary axis that runs perpendicular to the screen, and gives the illusion of depth.

---

## Timing

In order to slow down sprite animation, the current frame number should not be incremented each time through the game loop. Instead, it should be incremented only at specific intervals. To implement this technique, we need to add two members to our sprite object: one for the current interval measurement and one for a threshold amount. When this current interval measurement exceeds the threshold, we can increment the frame counter.

Technically, we could set the threshold amount to an integer value, and then increment the current interval measurement each time through the game loop. However, this would make the speed of the sprite animation contingent on the speed of the game. A more efficient method would be to base the speed of animation on an elapsed time amount. This would constrain the animation to a specific rate that would be constant on all machines.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

We can do this by setting our threshold amount to a specific time interval, measured in milliseconds. Our current interval measurement would then be set to the system time of the last increment of the frame number. In the game loop, we then check to see if the amount of time indicated in the threshold has elapsed since the last increment of the frame number. If so, we increment the frame number and reset the current interval measurement. The following example demonstrates this technique.

*Listing 6-7: Accurate sprite animation timing*

type

```
sprite class TSprite = class
    FXPos, FYPos,           // tracks the current position
    FXVel, FYVel,          // tracks the velocity
    FCurFrame,           // tracks the current frame number
    FFrameNum: Integer;    // tracks the number of frames
    FImages: IDirectDrawSurface4; // a pointer to the sprite images
    FCurTime,             // tracks a timing value
    FTimeThreshold: LongWord; // holds the animation timing threshold
end;
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
var
```

```
.  
.   
.
```

```
begin
```

```
.  
.   
.
```

```
{initialize the sprite object}
```

```
FSprite := TSprite.Create;
```

```
FSprite.FXPos := DXWIDTH - 90;
```

```
FSprite.FYPos := 215;
```

```
FSprite.FXVel := -1;
```

```
FSprite.FYVel := 0;
```

```
FSprite.FCurFrame := 0;
```

```
FSprite.FFrameNum := 5;
```

```
FSprite.FImages := FImages;
```

```
{retrieve the current system time to initialize the timing algorithm}
```

```
FSprite.FCurTime := timeGetTime;
```

```
{we want the sprite to animate every 100 milliseconds}
```

```
FSprite.FTimeThreshold := 100;
```

```
.  
.
```

```

    .
end;

procedure TfrmDXAppMain.DrawSurfaces;

var
    .
    .
    .

begin
    {erase the last frame of animation by using a simple color fill to
    'black out' the entire backbuffer surface}
    FillChar(BltFx, SizeOf(TDDBltFx), 0);
    BltFx.dwSize := SizeOf(TDDBltFx);
    BltFx.dwFillColor := 0;
    FBackBuffer.Blt(nil, nil, nil, DBLT_COLORFILL or DBLT_WAIT, @BltFx);

    {increment the frame number and update the sprite's position}
    with FSprite do
    begin
        {if the elapsed time since the last frame number increment
        is greater than our threshold amount...}
        if timeGetTime - FCurTime > FTimeThreshold then
        begin
            {animate the sprite by incrementing the frame number}
            Inc(FCurFrame);

            {make sure we roll over the frame number as necessary}
            if FCurFrame > FFrameNum then
                FCurFrame := 0;
        end;
    end;
end;

```

```

        {store the current system time}

        FCurTime := timeGetTime;

    end;

    .
    .
    .

end;

```

---

## Z-Order

As stated above, the z-order of a sprite is its position along an imaginary third- dimensional axis. Some games may feature foreground sprites and background sprites, while others may have sprites that move closer to and farther from the user, scaling as appropriate in a pseudo three-dimensional viewpoint. In games like these, drawing sprites in the correct order becomes an important issue.

However, actually accomplishing this illusion of depth is quite simple. Essentially, the application needs to draw the sprites starting with the farthest from the user and ending with the closest. As you draw the nearer sprites, their images will obscure those farther away, giving the illusion of depth. This is known as the painter's algorithm; the technique is illustrated below.



**Figure 6-9:** The painter's algorithm

We've already implemented the painter's algorithm in a very simplistic way by drawing the background behind the sprites before drawing them. In a more complex, real-world application, the z-order of the sprites themselves must be taken into account. We can accomplish this by simply adding another member to our sprite object for tracking this z-order.

In practice, we'll need to move the sprites and then sort them by z-order before drawing. The sorting step is necessary in order to draw the sprites in the correct order. Any well-documented, high-performance sorting algorithm will do. In the following example, we don't employ any sort of real sorting algorithm for the sake of code simplicity. However, the example does demonstrate a simple z-ordering mechanism. Sprites that are moving downward have a higher, or closer, z-order than those moving upward. Implementing scaling to draw the sprites larger when they are closer and smaller when they are farther away would improve the illusion.

**Listing 6-8:** *Drawing sprites according to z-order*

---

```

type

```

```

.
.
.

{the sprite class}

  TSprite = record
    FXPos, FYPos,           // tracks the current position
    FXVel, FYVel,          // tracks the velocity
    Height, Width,         // sprite frame height and width
    FCurFrame,            // tracks the current frame number
    FFrameNum: Integer;    // tracks the number of frames
    FImages: IDirectDrawSurface4; // a pointer to the sprite images
    FCurTime,              // tracks a timing value
    FTimeThreshold: LongWord; // holds the animation timing threshold
    ZOrder: Integer;       // the sprite depth
  end;

var

.
.
.

  Sprites: array[0..1] of TSprite;
  ZOrderIndex: array[0..1] of Integer;

implementation

.

```

```
.  
.  
  
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
.  
.  
.
```

```
{initialize the first spaceship}
```

```
with Sprites[0] do
```

```
begin
```

```
    Width := 61;
```

```
    Height := 56;
```

```
    FXPos := DXWIDTH div 2;
```

```
    FYPos := DXHEIGHT;
```

```
    FYVel := 3;
```

```
    FImages := FShip1;
```

```
    ZOrder := 0;
```

```
end;
```

```
{initialize the second spaceship}
```

```
with Sprites[1] do
```

```
begin
```

```
    Width := 60;
```

```
Height := 74;
```

```
FXPos := DXWIDTH div 2;
```

```
FYPos := 0;
```

```
FYVel := -3;
```

```
FImages := FShip2;
```

```
ZOrder := 1;
```

```
end;
```

```
.
```

```
.
```

```
.
```

```
end;
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
  iCount: Integer;
```

```
  Area: TRect;
```

```
  {this function moves a sprite}
```

```
  procedure MoveSprite(var Sprite: TSprite);
```

```
  begin
```

```

Sprite.FYPos := Sprite.FYPos + Sprite.FYVel;
if Sprite.FYPos<0 then
begin
    Sprite.FYPos := 0;
    Sprite.FYVel := 0-Sprite.FYVel;

    {we've only got 2 z-order values, so flip it when the sprite
    hits a boundary}
    Sprite.ZOrder := 1-Sprite.ZOrder;
end
else
if Sprite.FYPos>DXHEIGHT-Sprite.Height then
begin
    Sprite.FYPos := DXHEIGHT-Sprite.Height;
    Sprite.FYVel := 0-Sprite.FYVel;

    Sprite.ZOrder := 1-Sprite.ZOrder;
end;
end;

begin
    {erase the previous animation frame}
    Area := Rect(0, 0, DXWIDTH, DXHEIGHT);
    FBackBuffer.BltFast(0, 0, FBackground, Area,
                        DDBLTFAST_NOCOLORKEY or DDBLTFAST_WAIT);

    {move the sprites}
    for iCount := 0 to 1 do
        MoveSprite(Sprites[iCount]);
    {perform some really simplistic z-order sorting}

```



```
if Sprites[0].ZOrder > Sprites[1].ZOrder then
begin
    ZOrderIndex[0] := 0;
    ZOrderIndex[1] := 1;
end
else
begin
    ZOrderIndex[0] := 1;
    ZOrderIndex[1] := 0;
end;
```

{now that the sprites are sorted per their z-order, draw them starting with the farthest sprite. this will result in a layered graphic with distant sprites being obscured by closer sprites}

```
for iCount := 0 to 1 do
    FBackBuffer.BltFast(Sprites[ZOrderIndex[iCount]].FXPos,
                        Sprites[ZOrderIndex[iCount]].FYPos,
                        Sprites[ZOrderIndex[iCount]].FImages,
                        Rect(0, 0, Sprites[ZOrderIndex[iCount]].Width,
                            Sprites[ZOrderIndex[iCount]].Height),
                        DDBLTFAST_SRCCOLORKEY or DDBLTFAST_WAIT);
end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

## Summary

In this chapter, we discussed several techniques for implementing sprites and sprite animation. We looked at various DirectDraw methods for implementing transparency, and we also took a look at what is required for detecting collisions between two sprites. We also lightly covered some of the more common animation issues facing developers. When writing sprite manipulation code, it is important to keep these points in mind:

- The most important graphical elements of any game are those graphics with which the user interacts. Sprites provide the animated elements of a game that make the gaming environment come alive. These can take the form of images that the player directly interacts with, such as a spaceship, or background elements that are non-interactive yet provide animation that makes the environment richer, deeper, and more believable
- To produce the illusion of animation, a series of images is required, where each image shows the character or object to be animated in a slightly different position than the image before it, like a filmstrip. When each image is rapidly displayed one after another, the human brain expects to see smooth motion, and any gaps in movement from one frame to another will be automatically filled in.
- To produce the illusion of motion, when the application prepares to draw the next frame of animation, simply change the horizontal or vertical coordinate at which the image will be drawn into the destination. If these coordinates keep changing, the animated image will be drawn at different positions.
- A few items of information need to be tracked for each sprite. In particular, its current x and y coordinate must be tracked so the application knows where to draw the sprite. A horizontal and vertical velocity could also be stored. These velocities would indicate the number of pixels to move the sprite, and would be added to the x and y coordinates on each iteration through the game loop. It should also track the current frame of animation, the number of frames in the animation, and a pointer to the bitmap or DirectDraw surface containing the sprite images.
- The process of standard sprite animation is accomplished by following three distinct steps: erasing the sprite image, updating the sprite position, and drawing the sprite image.
- In order to make sprite images more believable, they must contain pixels that are rendered transparently so that the background shows through. This can be accomplished through color keying, or through a more direct approach by drawing each pixel individually and skipping those pixels that are transparent.
- When using the Lock method to access surface memory directly, remember to always call the

Unlock method before using the surface in other method calls.

- All games, at some point, must deal with the situation when one sprite touches another. For some games, this may be when a bullet strikes an alien or when an animated mouth collides with a glowing white pellet to eat it. In most cases, sprite collision can be detected using one of two techniques: bounding boxes and pixel overlap comparisons.
- Bounding box collision detection works by defining a rectangular region around the sprite, its “bounding box.” To determine if a collision with another sprite has occurred, the application simply determines if their bounding box rectangles intersect
- After determining the rectangular intersection area between two sprites, we can access the sprite image surfaces and compare the pixels that are within this intersection rectangle. If any non-transparent pixels overlap, we have a pixel overlap collision. This method is very accurate, but tends to be slow.
- Sprite animation should be regulated to make it look more believable. We can accomplish this by incrementing the current frame number only at specific intervals.
- Games with multiple layers of sprites must take into account the sprites’ z-order. Sprites should be sorted by z-order and drawn starting with the farthest sprites. This will result in closer sprites obscuring sprites that are farther from the viewer, giving an illusion of depth.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 7 Input Techniques

This chapter covers the following topics:

- Human/computer interaction
- DirectInput fundamentals
- Retrieving keyboard input
- Retrieving mouse input
- Retrieving game controller input

An excellent graphics engine is important to the popularity and enjoyment of a game. The sound effects and music that are synchronized to the action on screen are also imperative. However, think of the most enjoyable game you've ever played, and consider what it would be like if you had no way of interacting with the game elements. It would be like watching a movie, at best. How about if you could interact with the game through only a single button on the keyboard, or perhaps with only the movement of the mouse cursor? Your actions would be limited indeed, as would your enjoyment of the game. As with graphics and sound, user input is a vital piece of overall game development that merits serious consideration.

User input functionality is something most developers, and game players, take for granted. Even so, the method by which a game player interacts with the game can have a profound effect on how skillfully the player navigates the game's environment, and consequently the overall enjoyment of the game. Some game types naturally lean towards certain types of input devices; flight simulators tend to "feel right" when played with a joystick, whereas real-time strategy games are usually best played with the mouse. The supported devices allowing the user to provide game input, and how those devices are implemented in terms of how they manipulate game elements, can make or break a game.

In this chapter, we examine various techniques for retrieving user input through a variety of devices. Regular Windows API functions will be briefly touched upon, as our discussion will focus on how DirectInput provides developers with a responsive and powerful interface to any input device available.

## Human/Computer Interaction

Human/computer interaction is the concept of how humans and machines, typically computers, interact with one another in a method that is meaningful and intelligible to both. Entire volumes have been written on techniques to make human/computer interaction more intuitive, including both input techniques themselves as well as the presentation of the interface. While the majority of this concept is beyond the scope of this book, certain elements are relative to our discussion of user input techniques.



---

**human/computer interaction:** The concept of techniques and methods by which humans and machines, primarily computers, exchange information.

Exactly how user input will be translated into game element manipulation is dependent on the game itself. Flight simulators need input that affects the direction of flight, or perhaps even the selection and use of a weapon. Strategy games, by contrast, need input that determines which token is selected and its direction of movement or combat. These concepts will be accounted for when the game itself is designed. Perhaps the most important consideration for user input is the device by which the user will interact with the machine and, more specifically, the game.

### Input Devices

Technically speaking, today's machines have a wide variety of devices by which they can receive input from users. Some of the more exotic input devices, such as speech recognition, are becoming available to the average computer user, and may one day replace many, or all, of the standard input devices currently in use. While it may be a good marketing hook to provide support for some of these more esoteric input devices, all games should provide support for standard input devices found on almost every machine. These standard input devices consist of the keyboard, the mouse, and game controllers.

### Keyboard

It would be hard to imagine any computer in today's market that doesn't come with a keyboard. The keyboard has been the standard input device since the dawn of the home PC, and it is safe to assume that it will be available for input when your game is executed. Retrieving keyboard input is easy and very straightforward, and can provide a very intuitive interface for almost any game.

### Mouse

The mouse is another standard input device available on just about every machine. While not quite as old as the keyboard, the mouse is just as necessary and probably more important for user input, thanks to Windows. Since our games will be using DirectX, and will thus be restricted to the Windows platform, we can assume that the mouse will be available for input when the game is executed. Retrieving mouse input is almost as easy and straightforward as retrieving keyboard input, and like the keyboard, it provides a very intuitive interface for certain types of games.

### Game Controllers

About five years ago, this category would probably have been called simply Joysticks. However, in the recent past, a multitude of input devices has been created that are aimed specifically at providing intuitive input for various games. Rudder pedals, steering wheels, gamepads, throttles, and all manner of sundry and arcane input devices are now available, making some games much more enjoyable to play. Even the standard joystick has become incredibly complex, with many sporting impressive arrays of buttons, switches, sliders, and even tactile responses known as force feedback.



---

**force feedback:** A method by which an input device can output tactile feedback to the user through various actuator motors.

Strictly speaking, a game cannot automatically assume that any type of game controller is hooked up to the machine, as game controllers have not yet become a standard input device shipped with new machines. It is good programming practice to provide an alternative form of input in the absence of a more appropriate input device, even if this input interface is somewhat cumbersome or even awkward. Retrieving input from various game controllers is necessarily more difficult than retrieving input from the mouse or keyboard, primarily because of the sheer number and variety of available game controllers. However, if a game lends itself to input from a specific type of game controller, such as a steering wheel for driving games, then the game should provide support for the most appropriately intuitive input device.

## Input Considerations

Even though the developer has determined which input devices will be supported and how they interact with game elements, individual users may find this structure somewhat restrictive based on their particular preferences. To accommodate those who feel your perfect interface could use some improvement, it is important to allow the user to change the configuration of input methods to their desired taste, within the structure of the input interface.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Perhaps the easiest to implement, and the most effective in terms of user configuration preferences, is a method by which users can redefine which keys or buttons perform which actions. This could be accomplished by presenting the user with a dialog box containing all available actions and the keys or buttons to which those actions are mapped. This should include keys on the keyboard and buttons on both the mouse and joystick. For example, the trigger button on a joystick may be used to fire a round of bullets in a flight simulator game, by default. However, the user may be more comfortable performing this action with another button. Allowing for the remapping of input methods in this way lets the individual user customize the game to his or her preferences, thus making it more enjoyable to a wider audience.



**Tip:** For an excellent example of an implementation of this technique, take a look at games like Quake and Halflife.

## DirectInput

DirectInput is the DirectX component responsible for providing user input from all manner of input devices, such as keyboards, mice, and game controllers. As with other DirectX components, the DirectInput API is large, and has its own set of requirements and nuances.

For some devices, DirectInput programming is almost trivial. For others, it is incredibly complex. DirectInput's complexity comes from the fact that it was designed to provide input from future game controllers that have not yet been invented. This forward thinking allows developers to provide support for new game controllers without having to rely on a new version of DirectInput. This support comes at the price of obscurity and complexity, but the marketing hook of supporting the latest input device may well be worth the effort. Fortunately, standard input devices such as the mouse and keyboard are relatively easy to interface with. Even joysticks and similar game controllers, while more complex, are not absolutely arcane in their implementation.

## Features



DirectInput offers many features above and beyond what is available through the Win32 API. As with all DirectX components, DirectInput gives the developer more control over user input than the Windows API, while also providing an improvement in performance. DirectInput provides the following features of specific interest to game developers:

- Responsive input retrieval directly from input device hardware
- Support for almost any input device, including mice, keyboards, joysticks, and other game controllers
- Access to force feedback devices for providing tactile output
- Support for multiple input devices attached to the machine, such as two or more keyboards or mice

Later in this chapter we will see how DirectInput is used to retrieve user input from the keyboard, the mouse, and standard game controllers such as joysticks.

## Functional Overview

Similar to DirectDraw, DirectInput programming consists of creating a DirectInput object and several DirectInputDevice objects. The DirectInput object itself acts as a liaison between the application and input device drivers. Individual DirectInputDevice objects represent the application's interface directly to input hardware or, more specifically, that hardware's device driver. The DirectInput object is used to create individual DirectInputDevice objects, which can then be queried to determine the specific capabilities of the device.

The DirectInputDevice object is used to communicate with all input devices. In other words, there is not a separate object type for communicating with keyboards, mice, or joysticks; they are all interfaced through a DirectInputDevice object. This is one of the reasons why DirectInput programming can be complex. The individual methods of a DirectInputDevice object all work with input devices in similar ways according to the context of the type of device. For example, you set device properties through a single function call, yet the properties themselves are dependent upon the device. You can set the range of motion returned from a joystick as a property, but this wouldn't make any sense if you tried to do the same with a keyboard.

However, at the same time that this type of interface introduces a level of complexity, communicating with different devices is very similar in form from device to device. After you've grasped a particular concept with one device, it is very easy to apply that knowledge to other devices. This may result in what appears to be some redundancy when we discuss individual devices later in the chapter. However, the information reported by these methods is dependent on the device we are communicating with, making these concepts uniquely different depending on the context in which they are used.

## Buttons and Axes

Where DirectInput is concerned, a button is considered to be any switch that has an on and off position. There is no distinction made between buttons on a mouse, keyboard, or joystick. Bear in mind that just because a device is referred to as a button, it does not necessarily have to be physically represented as a button. A device on a game controller that has a specific on and off position may take the form of a physical toggle switch, but will be treated by DirectInput as a button.

Conversely, an axis has a range of positions where its value is affected by the physical movement of a device in some particular direction. Like buttons, axes do not necessarily have a specific physical representation. Joysticks and mice are objects that, generally speaking, represent two axes. However, sliders and dials on joysticks or game controllers and the wheel on a Microsoft IntelliMouse are also axes. Rudder pedals, steering wheels, and flight yokes are other examples of controllers that represent axes.

In similar fashion, an axis does not necessarily have an association with any particular direction in space. Joysticks and mice report movement along the x and y axes, which may translate into the movement of a pointer or a change in the direction of flight. However, in the context of a specific game, this information may be used to indicate the depth at which an object is placed or the circular rotation of a dial. This points out that just because an axis may have a specific name, its value is meaningful only in the context in which it is used and does not have to map to any physical direction or movement.

While many devices report movement along various axes, the measurement of this movement is dependent on the type of device reporting it. Some devices report axis movement as a relative measurement, which others report it as an absolute measurement.

**Relative Measurement** Some axes can move an infinite distance in a particular direction, and lack a default or “rest” position. An example of this type of axis measurement would be that reported by a mouse. Therefore, the values returned by this type of device are meaningful only in relation to the last measurement taken from the device.

**Absolute Measurement** Other axes can physically only move a specific distance. This would include the movement of the joystick or the movement of dials and sliders on a game controller. These values are typically measured from a fixed point and have a specific range.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)



*Note:* DirectInput does allow the developer to redefine a relative axis to report absolute data and vice versa. The employment and implications of this technique are dependent on the context of the application itself.

## Immediate and Buffered Data

Since DirectInput communicates directly with device hardware, it does not return the state of axis movement or button presses with Windows messages. Instead, there are two flexible methods by which the application can retrieve the state of an input device: immediate and buffered.

Buffered data retrieval simulates the mechanism by which Windows reports input. Individual input events are stored in a buffer, which takes the form of an array of data structures containing the various axis movement values or button clicks. When you create a DirectInputDevice object for a particular input device, there is an additional step required in order to initialize the device to store information in this buffer.

Immediate data retrieval does not report a stream of singular events, but returns the current state of the device at the time of the function call. This will be reported in an initialized data structure dependent upon the specific device type, and can include information on the entire device or only for a specific object on that device, such as a button. There is no additional setup required to retrieve immediate data from a device.

Either of these data retrieval methods are perfectly acceptable depending on the requirements of the individual application. Indeed, both could be used simultaneously. However, it is generally most useful to utilize buffered data retrieval for relative data measurements and immediate data retrieval for absolute data measurements. We will see examples of both later in the chapter.

## Polling and Event Notification

The “when” of retrieving input data may be as important as the “how.” In what seems to be a recurring theme throughout this book, DirectInput provides two choices: polling and event notification.

Polling involves checking the state of the input device at the time that the input is needed. Typically, this is performed in the game loop. The state of the device may be checked only once, or perhaps multiple times, and is usually returned in the form of immediate data. Since the device is only checked at specific intervals, it is possible for the user to manipulate the device between checks, thus resulting in lost input. Polling may seem like a throwback to the days of DOS game programming. Indeed, it is similar, but polling for new input data is incredibly easy to implement, and in most cases is adequate for precise input measurement.

Event notification, on the other hand, is very similar to event-driven programming. This is a mechanism by which DirectInput will notify the application when new input data is available, and is best paired with buffered data retrieval. While this method of retrieving input can be very powerful, it is much more difficult to implement and involves the use of threads, Win32 event objects, and critical sections. We will see examples of both later in the chapter.

## Cooperative Levels

As discussed in the DirectDraw tutorial, a cooperative level determines how the application works with other applications. As with other DirectX components, DirectInput gives the developer a choice as to how the application will share input devices with other applications, in terms of who can gain access to the device and if input can be retrieved based on application focus.

The first measurement of cooperative level determines when your application has access to data from the device, known as foreground or background access. Foreground access means that your application will have access to input data only when its main window has focus. Background access allows your application to retrieve input data whether it has focus or not; input data is available at all times, even if the application is minimized. Foreground mode is the most common for games and other applications. Nevertheless, you may need to use background mode if the application requires access to the device when a dialog box is open.

The second measurement of cooperative level is based on exclusivity, which determines if only a single application has access to a device. Applications that acquire a device in non-exclusive mode can retrieve input from that device unless another application acquires it exclusively. When a device is acquired exclusively, only the acquiring application can receive input from that device. In non-exclusive mode, other applications can acquire the device in non-exclusive or exclusive mode. In exclusive mode, other applications can only retrieve the device in non-exclusive mode. Thus, only one application at a time can have exclusive access to a device. For some devices, such as force feedback, this is a necessity, as two applications sending force feedback effects simultaneously could confuse the user and possibly damage the hardware.

When specifying a cooperative level, the application must indicate either foreground or background in addition to either exclusive or non-exclusive access. The following table summarizes the two cooperative level measurements and their implications for the current application and others in the system.

Measurement	Implications	Validity
Non-Exclusive/Background	Other applications can acquire the device exclusively or non-exclusively; input data is available at all times	All devices except force feedback
Non-Exclusive/Foreground	Other applications can acquire the device exclusively or non-exclusively; input data is available only when application has focus	All devices except force feedback
Exclusive/Background	Other applications can acquire the device non-exclusively; input data is available at all times	All devices except keyboards
Exclusive/Foreground	Other applications can acquire the device non-exclusively; input data is available only when application has	All devices except keyboards

Bear in mind that Windows requires exclusive access to the keyboard at all times, so that functions such as Alt+Tab and Ctrl+Alt+Delete are always available. Thus, an application can only obtain non-exclusive access to the keyboard. Conversely, Windows can only use the mouse in an exclusive mode, and thus loses access to the mouse when another application acquires it exclusively. This means that when an application acquires the mouse in an exclusive mode, the mouse cursor will disappear and Windows will no longer generate mouse messages.

There is a small performance gain when acquiring the mouse in exclusive mode, since mouse messages are not being generated. However, there is no particular performance advantage to exclusive over non-exclusive cooperative levels for other devices.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)



**Tip:** It is standard practice to inform users to close all other applications before starting your game. It is doubtful that users would try to play two or more games at once, so you should generally set the cooperative level to exclusive/foreground.

## Device Acquisition

Once a device is set up, the application cannot receive input data until it instructs DirectInput that it is ready to do so. This process is known as acquiring the device. Once a device is acquired, input data can be retrieved in whatever manner desired as discussed above. Successfully acquiring the device is dependent on the cooperative level specified by the application and by those of other applications using DirectInput, as we just illustrated.

However, bear in mind that a device can be forcibly unacquired by the application. This may happen when another application receives focus or acquires the device in exclusive mode (again dependent on the cooperative level of both applications and who acquired the device first). In this instance, any method that returns input data will return a value of DIERR\_INPUTLOST or DIERR\_NOTACQUIRED. You will need to check for this error value in your application, possibly attempting to reacquire the device until successful. You may also deliberately unacquire the device, such as when Windows needs the mouse in order to give the user access to menus.

## Initializing DirectInput

As with other DirectX components, several steps are required in order to set up DirectInput and begin interfacing with user input devices. In general, most of these steps are required to read data from keyboards, mice, and game controllers. We will cover these steps in a broad sense here in order to familiarize you with them. They will be covered again more precisely when we look at retrieving data from specific input devices. DirectInput, and DirectInput devices, are instantiated by performing the following:

- Creating the DirectInput object
- Enumerating input devices
- Creating the input device
- Setting the data format
- Setting the cooperative level
- Setting device properties
- Acquiring the device



## Creating the DirectInput Object

The first step in retrieving data from an input device is to create the DirectInput object itself. Creating the DirectInput object involves calling the DirectInputCreate function, which is defined as:

```
function DirectInputCreate(  
  
    hinst: THandle;                // the application instance handle  
  
    dwVersion: DWORD;             // DirectInput version number  
  
    out ppDI: IDirectInput;       // the IDirectInput object to be instantiated  
  
    punkOuter: IUnknown           // unused, set to nil  
  
): HRESULT; stdcall;             // returns a DirectX error code
```

The first parameter is a handle to the calling application's instance. The second parameter indicates the DirectInput version number. Most applications should simply set this parameter to `DIRECTINPUT_VERSION` (a constant defined in the `DInput.pas` file). However, if an earlier version number is indicated, DirectInput will emulate that version. This is useful if you had no need for newer DirectInput functionality, such as force feedback. The third parameter takes a pointer to the IDirectInput object. When this function returns, this object will be instantiated. The last parameter is used for COM aggregation, and must be set to nil as it is currently unsupported.



---

**Caution:** If indicating an earlier version in the call to DirectInputCreate, the application must use the data structures compatible with this earlier version. Note that this will also affect the value to which the `dwSize` member of these structures must be initialized.

---

## Enumerating Input Devices

Technically speaking, this step is not required if your application will only support keyboard and mouse input devices. It may be useful if the target machine has more than one keyboard or mouse attached, but you can still tell DirectInput to create objects that represent the primary keyboard and mouse. However, game controllers are another story. Since there are no primary or default game controllers, applications that wish to support joysticks, steering wheels, etc., must obtain individual GUIDs for these objects through enumeration.

The `EnumDevices` method of the IDirectInput object is used to list all available input devices. It is a callback function that will call the specified function once for each type of device attached to the system as indicated by the scope flags. The `EnumDevices` method is defined as:

```
function EnumDevices(  
  
    dwDevType: DWORD;             // indicates the type to enumerate  
  
    lpCallback: TDIEnumDevicesCallbackA; // the callback function  
  
    pvRef: Pointer;              // an application-defined value  
  
    dwFlags: DWORD               // flags controlling the scope  
  
): HRESULT;                     // returns a DirectX error code
```

The first parameter is a flag indicating the type of devices to enumerate. This can be set to zero to enumerate all devices, `DIDEVTYPE_MOUSE` to enumerate mice, `DIDEVTYPE_KEYBOARD` to enumerate keyboards, or `DIDEVTYPE_JOYSTICK` to enumerate game controllers. The second parameter is a pointer to the callback function

itself. The third parameter is a pointer to an application-defined value. The final parameter is a series of flags that further restrict the enumeration. This can be set to `DIEDFL_ATTACHEDONLY` to restrict enumeration to attached devices and/or `DIEDFL_FORCEFEEDBACK` to enumerate only force feedback devices. The callback function is defined as:

```
function TDIEnumDevicesCallbackA(  
  
var lpddi: TDIDeviceInstance;           // a device description record  
  
pvRef: Pointer                          // the application-defined value  
  
): BOOL; stdcall;                       // returns TRUE or FALSE
```

The first parameter contains a `TDIDeviceInstance` structure initialized with information about the enumerated input device. The second parameter contains a pointer to the application-defined value. This function must return `TRUE` to continue the enumeration; `FALSE` will stop the enumeration.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
**ISBN: 1556226373 Pub Date: 12/01/99**

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

The TDIDeviceInstance structure passed into this callback function contains everything necessary for presenting the user with a list of choices, as well as creating an input object of the desired type. The TDIDeviceInstance structure is defined as:

TDIDeviceInstance = packed record

```

    dwSize: DWORD; // size of structure

    guidInstance: TGUID; // object instance GUID

    guidProduct: TGUID; // object product GUID

    dwDevType: DWORD; // device type flag

    tszInstanceName: Array [0..MAX_PATH-1] of WideChar; // instance name string

    tszProductName: Array [0..MAX_PATH-1] of WideChar; // product name string

    guidFFDriver: TGUID; // force feedback driver

    wUsagePage: WORD; // HID usage page code

    wUsage: WORD; // HID usage page

```

end;

An application could display a list of available input devices based on the instance name and product name available from the TDIDeviceInstance structure. If the instance GUID was stored along with this information, the application could create an input device using the GUID of the associated device that the user selected. The following example demonstrates creating an IDirectInput object and enumerating all attached game controllers.

*Listing 7-1: Creating an IDirectInput object and enumerating game controllers*

```

var

    Form1: TForm1;

    FDirectInput: IDirectInput;

implementation

function EnumInputDevs(var lpddi: TDIDeviceInstance; pvRef: Pointer): Integer;
    stdcall;

begin
    {display the instance name and product name of the enumerated game device}
    TStringList(pvRef).Add(lpddi.tszInstanceName+' - '+lpddi.tszProductName);

    {continue enumeration}

    Result := 1;

end;

procedure TForm1.FormActivate(Sender: TObject);

begin
    {create the DirectInput object}

    DirectInputCreate(hInstance, DIRECTINPUT_VERSION, FDirectInput, nil);

    {enumerate all attached game devices}

    FDirectInput.EnumDevices(DIDEVTYPE_JOYSTICK, EnumInputDevs,
                               lbxJoysticks.Items, DIEDFL_ATTACHEDONLY);

end;

```

---

At this point, the method calls for creating and initializing input device objects are the same, but the arguments passed into these methods are different depending on the input device itself. We will cover them briefly here, but we will wait to demonstrate any examples until we get to the sections where we demonstrate reading input data from various devices.



**Figure 7-1:** Attached game devices

## Creating the Input Device

The `DirectInput` object has been created, and we've presented the user with a list of input devices from which one device has been selected. Given this device's instance GUID, we can now create an `IDirectInputDevice` object representing the device. We create this object by calling `IDirectInput`'s `CreateDevice` method. The `CreateDevice` method is defined as:

```
function CreateDevice(  
    rguid: PGUID; // the device instance GUID  
    var lpDirectInputDevice: IDirectInputDevice; // the IDirectInputDevice object  
    pUnkOuter: IUnknown // unused, set to nil  
): HRESULT; // returns a DirectX error code
```

The first parameter is the instance GUID for the desired input device. For game controllers, this should be a GUID retrieved from device enumeration. For the primary keyboard, you can use `GUID_SysKeyboard`, and for the primary mouse, you can use `GUID_SysMouse`. The second parameter is a pointer to the `IDirectInputDevice` object. When this function returns, this object will be instantiated. The final parameter is used for COM aggregation and should be set to `nil`.

This function will only return an instantiated `IDirectInputDevice` object, but if we wish to use force feedback devices, we need to get an instantiated `IDirectInputDevice2` object. We should probably retrieve an `IDirectInputDevice2` object anyway, as some game controllers have a special requirement for retrieving data, as explained below. Thus, we should create a temporary `IDirectInputDevice` object, and once this is instantiated, we can use the `QueryInterface` method of the object to retrieve an instantiated `IDirectInputDevice2` object. The temporary object can then be released by setting it to `nil`.

## Setting the Data Format

Once the device has been created, you must instruct `DirectInput` as to the format in which input data should be reported. This step is required, as input data from a mouse, for example, would be completely different than that retrieved from the keyboard or a joystick. You may be asking, why should we do this if `DirectInput` knows what kind of device we just created? Again, this is part of `DirectInput`'s open architecture. Setting the data format in this manner allows the developer to declare a custom data format for new devices not yet created. We'll see an example of this later when we cover game controllers. Actually, setting the data format is quite easy, and requires calling the `IDirectInputDevice`'s `SetDataFormat` method. The `SetDataFormat` method is defined as:

```
function SetDataFormat(  
    var lpdf: TDIDataFormat // a data format structure  
): HRESULT; // returns a DirectX error code
```

The only parameter to this method is a variable of type `TDIDataFormat`. The `TDIDataFormat` structure is a complicated beast due to this open architecture. Fortunately, there are several standard, predefined data formats that we can use for communicating with just about any input device. For the keyboard, we can set this parameter to `c_dfDIKeyboard`; the mouse uses `c_dfDIMouse`. Joysticks supposedly could use either `c_dfDIJoystick` or `c_dfDIJoystick2` (for advanced game controllers and force feedback). However, in practice, we have found that neither

of these data structures seem to work correctly. Game controllers come in all shapes and sizes and can have a wide variety of buttons, axes, sliders, POV hats, etc. Therefore, creating a custom data structure on the fly based on the type of objects available on the device usually provides better results. We'll see an example of this later in the chapter when we discuss game controllers.



**POV hat:** The point-of-view (POV) hat is a device that typically rests on the top of a joystick, and can be used to indicate a viewing direction separate from movement direction.

The value passed to this function ultimately determines the data structure used when retrieving immediate data from any particular device. The parameters of the `GetDeviceState` method (covered below in the individual sections for each different device) are directly affected by the value passed to `SetDataFormat`. Each of the predefined values for `SetDataFormat` correspond to the following data structures that must be subsequently used in the call to `GetDeviceState` when retrieving immediate data.

*Table 7-2: Data formats and their data structures*

Format	Structure
c_dfDIKeyboard	TDIKeyboardState
c_dfDIMouse	TDIMouseState
c_dfDIJoystick *	TDIJoyState
c_dfDIJoystick2 *	TDIJoyState2

\* Don't use these; use the technique for building a custom data format later in the chapter.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Setting the Cooperative Level**

We are almost ready to begin using the device, so now it is time to set the cooperative level. The cooperative levels and their ramifications were explained above. Setting the cooperative level requires a call to the IDirectInputDevice's SetCooperativeLevel method. The SetCooperativeLevel method is defined as:

```
function SetCooperativeLevel(
    hwnd: HWND; // a handle to the application window
    dwFlags: DWORD // cooperative level flags
): HRESULT; // returns a DirectX error code
```

The first parameter is the handle to the application's main window. This lets Windows and DirectInput know the application to which the input data should be sent. The second parameter is a combination of flags indicating the desired cooperative level. This must be set to one of each from DISCL\_BACKGROUND/ DISCL\_FOREGROUND and DISCL\_EXCLUSIVE/DISCL\_NONEXCLUSIVE.

**Setting Device Properties**

This step may be optional depending on the type of input device and the method by which input data is retrieved (i.e., immediate or buffered). However, some devices have certain properties that must be initialized before data can be retrieved from them. For example, joysticks must have their axes ranges set before they can return meaningful input data. Also, if you intend on using buffered data retrieval, you must set the size of the buffer. Setting device properties is accomplished by calling the IDirectInputDevice's SetProperty method. The SetProperty method is defined as:

```
function SetProperty(
    rguidProp: TRefGUID; // property identifier
    var pdiph: TDIPropHeader // the property data structure
```

```
): HRESULT; // returns a DirectX error code
```

This is another method where DirectInput's open architecture results in complexity. The first parameter identifies which property of the device to set. The value of this parameter is very dependent on the type of the device. For instance, it would be meaningless to set the range of an axis for a keyboard device. This parameter could be a GUID provided by the manufacturer of the device that identifies some specific property. Fortunately, however, there are several predefined values that we can use for setting specific properties. Perhaps the most common value passed to this parameter is `DIPROP_BUFFERSIZE`, which indicates we wish to set the size of the buffer for buffered data retrieval.

The second parameter is a pointer to a `TDIPropHeader` data structure. Here's where the complexity comes in. This parameter is a little misleading, because the method actually takes a pointer to whatever data structure is defined for the particular property being set. This could be some huge, custom data structure as defined by the product manufacturer. The only requirement is that the data structure must begin with a `TDIPropHeader` structure, which lets DirectInput know just how big the main structure is and the object on the device (such as an axis) whose properties are being set. Fortunately, most properties require only a single `DWORD` of information (such as buffer size) or perhaps two `LongWords` (in the case of ranges). The `TDIPropHeader` structure itself is defined as:

```
TDIPropHeader = packed record

    dwSize: DWORD; // the size of the main structure

    dwHeaderSize: DWORD; // the size of the TDIPropHeader structure

    dwObj: DWORD; // the device object identifier

    dwHow: DWORD; // indicates how to identify the device object

end;
```

The first member of this structure identifies the size of the overall structure. The second member should be set to the size of the `TDIPropHeader` structure. The third member identifies the device object (such as an axis) whose property is being set. There are a number of predefined identifiers that we will use to set various properties, and we'll see examples of this later in the sections on mouse and joystick programming. This member could be set to zero if a property that affects the entire device is being set, such as the buffer size. The last member tells DirectInput how the `dwObj` member is interpreted. This should be set to `DIPH_DEVICE` in the case of properties that affect the entire device. For individual device objects, there are two methods by which these objects can be identified. In this book, we will be concerned only with the offset method, so we should set this last member to `DIPH_BYOFFSET`.

DirectInput defines two standard structures that are most commonly used with `SetProperty`. The `TDIPropDWord` structure is used for setting properties that take only a single `DWORD` of information (such as buffer size), and the `TDIPropRange` structure is used for setting properties requiring two `LongWords` (such as ranges). The `TDIPropDWord` structure is defined as:

```
TDIPropDWord = packed record

    diph: TDIPropHeader; // the TDIPropHeader structure

    dwData: DWORD; // the value for the DWORD property

end;
```

The `TDIPropRange` structure is defined as:

```
TDIPropRange = packed record

    diph: TDIPropHeader; // the TDIPropHeader structure
```

```
lMin: Longint;           // the minimum range value

lMax: Longint;          // the maximum range value

end;
```

We'll see examples using the SetProperty method with these two data structures later in the chapter.



---

*Note:* Device properties can be retrieved by using the GetProperty method, which takes the exact same parameters.

---

## Acquiring the Device

The input device has been set up, and we are now ready to begin retrieving data. As explained above, we must acquire the device, which is accomplished by calling IDirectInputDevice's Acquire method. This simple method is defined as:

```
function Acquire: HRESULT;           // returns a DirectX error code
```

Subsequent calls to Acquire, once the device has been acquired, have no effect. A device can similarly be unacquired by calling the Unacquire method, which also takes no parameters.

## Retrieving Input Data

While the steps required for initializing and setting up DirectInput and DirectInputDevices are similar for every device, exactly how the input data is retrieved and extracted varies from device to device. This will also depend on whether you will be using immediate data retrieval or buffered data retrieval. This leads to subtle nuances in which initialization steps are required, and how the data returned from certain methods is interpreted.

In this section, we will take a look at examples for reading input data from three standard devices: the keyboard, the mouse, and the joystick. Each is similar in implementation, but there are differences that warrant a closer inspection. At the same time, we will also demonstrate both immediate data retrieval and buffered data retrieval. While we will only implement one or the other method of data retrieval in each example to simplify the code, immediate or buffered data retrieval can be used for any input device.

One thing to consider when developing your game application is if the complexity of DirectInput is even warranted. The Windows API contains numerous powerful functions that allow the user to read input from the keyboard, the mouse, or even game controllers. There is no support for force feedback, but if your application only requires a very simple interface to input devices, you may want to stick with standard Windows messages, or perhaps the Win32 API. In the following examples, we will demonstrate simplistic methods for reading input from devices using Win32 API functions before delving into the more complex DirectInput methods. Which you choose will be totally dependent on the type of game and the performance it will require, but either or both techniques are perfectly acceptable.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Keyboard Input**

Of all input devices, the keyboard is perhaps the least complex. Having been the standard for input since almost the dawn of computers themselves, reading keyboard input is very straightforward and easy.

DirectInput approaches the keyboard not as a text input device, but as a gamepad with many buttons (according to the DirectInput documentation). As such, you gain the benefit of being able to differentiate between the left and right versions of identical keyboard buttons, such as Shift, Alt, and Ctrl. However, DirectInput completely ignores such control panel settings as repeat rate and other keyboard properties. Thus, pressing a key registers as a single key press, regardless of how long it has been held down. Similarly, releasing a key also registers as a singular event. Therefore, you must specifically write your code to respond to a key being held down, starting when a key press is noted and ending when the key has been detected as released (if necessary).

**Windows Functions**

Windows provides developers with the `GetAsyncKeyState` function, which is very similar to retrieving immediate keyboard data using DirectInput. `GetAsyncKeyState` simply indicates if a specific key is pressed or released at the time of the function call. `GetAsyncKeyState` is defined as:

```
function GetAsyncKeyState(  
    vKey: Integer           // a virtual key code  
): SHORT;                 // returns a key press state code
```

This function's only parameter takes a virtual key code, such as `VK_LEFT` for the left cursor key. It returns a 16-bit value that indicates, if the most significant bit is set, the specified key is currently pressed. For applications that only require a modicum of keyboard input, this function may be more than adequate. The following example demonstrates the use of this function.

*Listing 7-2: Retrieving keyboard input through `GetAsyncKeyState`*



```

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    {poll the keyboard to check the state of various keys. the program can
    respond to multiple keys changing states simultaneously}
    if (GetAsyncKeyState(VK_LEFT) and $8000) = $8000 then
        shpLeft.Brush.Color := clRed
    else
        shpLeft.Brush.Color := clMaroon;

    if (GetAsyncKeyState(VK_RIGHT) and $8000) = $8000 then
        shpRight.Brush.Color := clRed
    else
        shpRight.Brush.Color := clMaroon;

    if (GetAsyncKeyState(VK_UP) and $8000) = $8000 then
        shpUp.Brush.Color := clRed
    else
        shpUp.Brush.Color := clMaroon;

    if (GetAsyncKeyState(VK_DOWN) and $8000) = $8000 then
        shpDown.Brush.Color := clRed
    else
        shpDown.Brush.Color := clMaroon;
end;

```

---

Unfortunately, this method cannot distinguish between left and right versions of certain keyboard keys, such as Shift, Alt, etc. Although easy to implement, it must still go through the Windows operating system to retrieve input data, which for many games may be too slow. For the best response times and total keyboard input control, we must use DirectInput.

### **DirectInput Initialization**

We must follow most of the steps outlined above to initialize DirectInput to receive keyboard data. In most cases, we only need to determine if a particular button is being pressed at a specific time, so immediate data retrieval is the perfect solution. This being the case, initialization of DirectInput is very straightforward.

The first order of business is creating the `DirectInput` object. We can then immediately create an `IDirectInputDevice` object for the primary keyboard using the `GUID_SysKeyboard` value. Since it's not likely that the keyboard will have force feedback capabilities, we'll skip querying for an `IDirectInputDevice2` object. We must then set the data format to `c_dfDIKeyboard`, the standard keyboard data format. We'll set the cooperative level for foreground, non-exclusive, and finally acquire the device so we can retrieve data. That's all there is to initializing `DirectInput` for immediate keyboard data retrieval. These steps are illustrated in the example below.



*Note:* Remember, the keyboard cannot be acquired in an exclusive cooperative level.

---

**Listing 7-3:** *Initializing DirectInput for keyboard data retrieval*

---

```
{create the DirectInput object}

DirectInputCreate(hInstance, DIRECTINPUT_VERSION, FDirectInput, nil);

{create the DirectInputDevice object for the primary keyboard}

FDirectInput.CreateDevice(@GUID_SysKeyboard, FDirectInputDevice, nil);

{set the data format for keyboard data}

FDirectInputDevice.SetDataFormat(c_dfDIKeyboard);

{set the appropriate cooperative level}

FDirectInputDevice.SetCooperativeLevel(Handle, DISCL_FOREGROUND or
                                        DISCL_NONEXCLUSIVE) );

{acquire the keyboard}

FDirectInputDevice.Acquire;
```

---

## Reading Input Data

Retrieving immediate data from any device involves a single call to the `IDirectInputDevice`'s `GetDeviceState` method. This method returns a snapshot of the state of the device at the time the function was called, including all pressed keys, axis movements, etc. The `GetDeviceState` function is defined as:

```
function GetDeviceState(
    cbData: DWORD; // the size of the data structure
```

```

lpvData: Pointer // a pointer to the appropriate data structure
): HRESULT; // returns a DirectX error code

```

Remember the call to the `SetDataFormat` method? This indicated to `DirectInput` the structure in which to report device data. The value passed to `SetDataFormat` indicates the data structure to be passed into `GetDeviceState` in the `lpvData` parameter. For keyboards, we use the `TDIKeyboardState` structure. The first parameter must be set to the size of this data structure.

The `TDIKeyboardState` structure itself is simply a 256-entry array of bytes. Each entry corresponds to a specific key on the keyboard. The array can be indexed using the `DIK_*` constants defined in the `DInput.pas` file for each specific key. If the hit bit of this byte is set, the key is currently down; otherwise it is up.

Therefore, to retrieve the current state of all keyboard keys, we simply call the `GetDeviceState` method of the `IDirectInputDevice` object we just created and check the high bit of the keys with which we are concerned. We must check for input data each time we need it, a technique known as polling as we discussed when we looked at polling versus event notification. The following example demonstrates this technique.

***Listing 7-4: Retrieving immediate keyboard input data***

---

```

procedure TfrmDXAppMain.DrawSurfaces;

var
    KeyboardState: TDIKeyboardState;
    SourceRect: TRect;

begin
    .
    .
    .

    {retrieve the current state of the keyboard as immediate data}
    FDirectInputDevice.GetDeviceState(SizeOf(KeyboardState), @KeyboardState);

    {if any of the cursor keys are held down, display an appropriate graphic}
    if (KeyboardState[DIK_UP] and $80) = $80 then
    begin
        SourceRect := Rect(0, 110, 110, 110+110);

        FBackBuffer.BltFast(265, 75, FImage, SourceRect, DDBLTFAST_NOCOLORKEY or
            DDBLTFAST_WAIT);
    end;
end;

```

```
if (KeyboardState[DIK_DOWN] and $80) = $80 then
begin
    SourceRect := Rect(2*110, 110, (2*110)+110, 110+110);
    FBackBuffer.BltFast(265, 295, FImage, SourceRect, DDBLTFAST_NOCOLORKEY or
        DDBLTFAST_WAIT);
end;

if (KeyboardState[DIK_LEFT] and $80) = $80 then
begin
    SourceRect := Rect(3*110, 110, (3*110)+110, 110+110);
    FBackBuffer.BltFast(155, 185, FImage, SourceRect, DDBLTFAST_NOCOLORKEY or
        DDBLTFAST_WAIT);
end;

if (KeyboardState[DIK_RIGHT] and $80) = $80 then
begin
    SourceRect := Rect(110, 110, 110+110, 110+110);
    FBackBuffer.BltFast(375, 185, FImage, SourceRect, DDBLTFAST_NOCOLORKEY or
        DDBLTFAST_WAIT);
end;

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Mouse Input**

The other standard input device, the mouse, is probably used in just about every type of game for allowing the user to choose options and to set up game parameters. While it is almost as easy to read mouse data as it is the keyboard, there are certain nuances that must be considered.

The mouse returns data for up to three buttons and up to three axes. Since DirectInput works directly with the mouse driver, any control panel settings for the mouse are ignored, such as button swapping. However, the application could check on the status of button swapping by calling `GetSystemMetrics(SW_SWAPBUTTON)`. The axis data returned by the mouse is relative by nature, as the mouse can move infinitely in any direction. These values are arbitrary and not based on screen coordinates, control panel settings, etc. The x-axis and y-axis values returned by the mouse correspond to the horizontal and vertical movements of the physical device; mouse wheel values will be reported as a z-axis. As in the windowed world, mouse travel to the left or away from the user is negative, and right or toward the user is positive.

DirectInput allows you to retrieve axis movement, but it does not provide a mouse cursor (under exclusive mode). This must be implemented in the application itself. This is certainly not a difficult task, as you know where to draw the mouse cursor by the x-axis and y-axis values you've retrieved (as they pertain to your application). This allows you to implement elaborate animated mouse cursors using techniques as discussed in the chapter on sprites. The animation could even be context sensitive based on the position of the cursor or the current game state. The only thing to keep in mind is that you will want to draw the mouse cursor as the last step of your rendering process so it is not obscured by other graphics.



*Note:* Setting a non-exclusive cooperative level will display a mouse cursor, but it will flicker horribly in a page flipping architecture such as that used throughout this book. It is usually best to acquire the mouse exclusively and implement your own mouse cursor drawing functionality.

One drawback to drawing the mouse cursor in this manner is that its responsiveness will be very dependent on the frame rate of the application itself. For example, say the user starts the game on a very slow machine, and it has a frame rate of about five frames per second. If you draw your mouse cursor at the end of the rendering loop, it will be very jerky and unresponsive, and could make leaving the game very difficult indeed. In order to combat this dependence on the game's overall frame rate, we will be implementing a multithreaded solution for drawing the

mouse cursor.

This multithreaded architecture will also be useful for detecting available mouse input through event notification. This technique involves the use of Windows event objects as well as a critical section for thread synchronization. This complex architecture will be covered in more detail below.

## Windows Functions

Windows provides developers with the `GetCursorPos` function, which is very similar to retrieving immediate mouse data for the cursor position using `DirectInput`. The `GetAsyncKeyState` function can be used to retrieve the current state of mouse buttons. The `GetCursorPos` function is defined as:

```
function GetCursorPos(  
    var lpPoint: TPoint // receives coordinates of cursor  
): BOOL; // returns TRUE or FALSE
```

This function's only parameter is a `TPoint` variable that receives the horizontal and vertical coordinate of the mouse cursor's current position. For applications that only require a modicum of mouse input, this function may be more than adequate. The following example demonstrates the use of this function.

### *Listing 7-5: Retrieving mouse input through `GetCursorPos`*

---

```
procedure TForm1.Timer1Timer(Sender: TObject);  
var  
    CursorPos: TPoint;  
begin  
    {retrieve the position of the mouse cursor}  
    GetCursorPos(CursorPos);  
  
    {the retrieved position is in screen coordinates, so we must convert them  
    to client coordinates}  
    CursorPos := ScreenToClient(CursorPos);  
  
    {draw the background into the offscreen buffer, erasing the last frame'  
    Offscreen.Canvas.Draw(0, 0, Background);  
  
    {now, transparently draw our custom mouse cursor into the offscreen buffer.  
    note that we've moved the hotspot' so that it will be in the center of the
```

```
    cursor}

    BitBlt(Offscreen.Canvas.Handle, CursorPos.X-16, CursorPos.Y-16,
          CursorAND.Width, CursorAND.Height, CursorAND.Canvas.Handle, 0, 0,
          SRCAND);

    BitBlt(Offscreen.Canvas.Handle, CursorPos.X-16, CursorPos.Y-16,
          CursorOR.Width, CursorOR.Height, CursorOR.Canvas.Handle, 0, 0,
          SRCPAINT);

    {finally, draw the new frame of animation onto the screen}

    Canvas.Draw(0, 0, Offscreen);

end;
```

---

One drawback with `GetCursorPos` is that it reports the position of the mouse in relation to the screen, and thus is an absolute measurement as it reports values in the range of 0, 0 to screen width, screen height. This may be fine for some games, but others may require the infinite resolution and quicker response offered by `DirectInput`.

## DirectInput Initialization

We must follow most of the steps outlined above to initialize `DirectInput` to receive mouse data. Since we are dealing with a stream of relative axis values, we will use the buffered method of input data retrieval. Therefore, unlike the keyboard, we must set certain properties of the device to initialize this buffer. Due to our multithreaded architecture, we have a few more tasks to complete, some of which are not directly related to `DirectInput`.



*Note:* Retrieving immediate mouse data is similar to retrieving immediate keyboard data, and will return the cumulative sum of all axes movements since the last call to `GetDeviceState`.

---

The first order of business is creating the `DirectInput` object. We can then immediately create an `IDirectInputDevice` object for the primary mouse using the `GUID_SysMouse` value. Since it's not likely that the mouse will have force feedback capabilities, we'll skip querying for an `IDirectInputDevice2` object. We must then set the data format to `c_dfDIMouse`, the standard mouse data format. We'll then set the cooperative level for foreground, exclusive. These steps are illustrated below.

[Previous](#) [Table of Contents](#) [Next](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).





**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 7-6: Initializing DirectInput for mouse data retrieval*

```
{create the DirectInput object}

DirectInputCreate(hInstance, DIRECTINPUT_VERSION, FDirectInput, nil);

{create the DirectInputDevice object for the primary mouse}

FDirectInput.CreateDevice(@GUID_SysMouse, FMouseDevice, nil);

{set the data format for mouse data}

FMouseDevice.SetDataFormat(c_dfDIMouse);

{set the appropriate cooperative level}

FMouseDevice.SetCooperativeLevel(Handle, DISCL_FOREGROUND or DISCL_EXCLUSIVE);
```

With this accomplished, we should now set the size of the buffer that will hold our mouse input data. This property takes only a simple DWORD of information, so we can use a TDIPropDWord structure in our call to SetProperty. After initializing the appropriate size members, we should set the diph.dwObj member to zero and the diph.dwHow member to DIPH\_DEVICE, indicating that this property affects the entire device. Setting the dwData member to 64 indicates we want to set the buffer size to hold 64 “data packets” (this number is arbitrary, and can be higher or lower as dictated by the application requirements). Thus, our buffer will be capable of holding 64 individual instances of mouse input, be that axis movement, button presses, etc. If we do not retrieve and flush this buffer before this limit is

reached, any subsequent input data will be lost. This process is illustrated below.

**Listing 7-7: Setting the size of the buffer**

---

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
var
    .
    .
    .

    {the mouse property structure}
    MouseProp: TDIPropDWord;
begin
    .
    .
    .

    {initialize our property structure}
    MouseProp.diph.dwSize := SizeOf(TDIPropDWord);
    MouseProp.diph.dwHeaderSize := SizeOf(TDIPropHeader);
    MouseProp.diph.dwObj := 0;
    MouseProp.diph.dwHow := DIPH_DEVICE;
    MouseProp.dwData := 64;           // we want 64 data packets'

    {set the size of the buffer property}
    DXCheck( FMouseDevice.SetProperty(DIPROP_BUFFERSIZE, MouseProp.diph) );
```

---

**Event Notification** As we've previously discussed, DirectInput has a method known as event notification where it can signal the application that an input event has occurred on some device. This releases the application from the requirement of polling the device through every iteration of the game loop, but it introduces some serious implications.

This technique works by using a Windows event object. Basically, an event is little more than a flag that exists in a

“signaled” or “non-signaled” state, and is used for thread synchronization. Using the `IDirectInputDevice`’s `SetEventNotification` method you can pass the device an event, which will be set to a signaled state when new input is available.

The only way this is useful is in a call to one of the Win32 API wait functions. These functions halt execution of a thread until the event object passed into the function becomes signaled. This is useless in a single-threaded application, as it would halt the execution of the primary thread unless the mouse is moving, making for a boring game indeed. This is why we must use this technique in a multithreaded application. We’ll see one of these wait functions in action below.

We will actually want to create two events, one which is passed to `DirectInput` to signal the availability of new mouse input and one to signal the thread to terminate. We can create these events using the Win32 API function `CreateEvent`, which is defined as:

```
function CreateEvent(  
  
    lpEventAttributes: PSecurityAttributes; // pointer to security attributes  
  
    bManualReset: BOOL; // flag for manual reset event  
  
    bInitialState: BOOL; // flag for initial state  
  
    lpName: PChar // name of the event object  
  
): THandle; // returns a handle of the event object
```

For our purposes, we can set the first parameter to `nil`, and the second and third parameters to `FALSE`. We can set the final parameter to an appropriate name for the event, but it is unimportant to our application. These events will be stored in an array for easy management. We will also create a `TCriticalSection` object, which we will use to synchronize the main thread and our secondary input thread. Using the critical section object will be covered below when we read the mouse data from the buffer.

Now that we’ve created our events, we need to tell `DirectInput` which event to signal when mouse input becomes available. This is accomplished by calling the `IDirectInputDevice`’s `SetEventNotification` method, defined as:

```
function SetEventNotification(  
  
    hEvent: THandle // a handle to the event to be signaled  
  
): HRESULT; // returns a DirectX error code
```

The only parameter to this method is the handle of the event to be signaled, which we just created above. The following example demonstrates the setup required for initializing event notification.

#### ***Listing 7-8: Initializing event notification***

---

```
{create our events, one for indicating new mouse input data, one for  
    indicating that the application is terminating}  
  
FMouseEvents[MOUSEEVENT] := CreateEvent(nil, FALSE, FALSE, MouseEvent');  
  
FMouseEvents[QUITEVENT] := CreateEvent(nil, FALSE, FALSE, QuitEvent');
```

```
{create our critical section synchronization object}

FCritSec := TCriticalSection.Create;

{set the event notification}

DXCheck( FMouseDevice.SetEventNotification(FMouseEvents[MOUSEEVENT]) );
```

---

**Secondary Thread Creation** Now that the critical section and events have been set up, we need to create our secondary thread. This second thread is where we will perform all input data retrieval. We used the File | New menu to create a new TThread object, using the Execute method as the location for our code that retrieves mouse input. We'll examine the details of this code below when we cover reading input data. To actually create the thread, we call our TThread object's Create method, passing its only parameter a value of TRUE to indicate we want the thread to be suspended upon creation. The thread should be suspended as we have not yet acquired the mouse. We then set its Priority property to tpTimeCritical, indicating we want it to be immediately responsive, and its FreeOnTerminate property to TRUE, indicating it should free itself when it is terminated. These steps are illustrated in the following listing.

**Listing 7-9: Creating the secondary input thread**

---

```
{create our secondary input thread}

FMouseThread := TMouseThread.Create(TRUE);

{we want it to be supremely responsive}

FMouseThread.Priority := tpTimeCritical;

{indicate it should free itself when it terminates}

FMouseThread.FreeOnTerminate := TRUE;
```

---

**Device Acquisition** Now that the events have been created and set up and the secondary input thread has been initialized, we can acquire the device by calling the Acquire method. As a final step, we must begin executing the input thread by calling the thread's Resume method. We are now finally ready to start retrieving buffered, event-driven mouse input data.

[Previous](#) [Table of Contents](#) [Next](#)





**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Reading Input Data**

As we previously mentioned, our secondary thread will handle the task of retrieving the input data from the buffer. Thus, all of the code we are about to review is located in the Execute method of our secondary TThread object.

The first item of note in the Execute method concerns the notification events we created. After checking for termination, the very first task we perform in the thread is to call the WaitForMultipleObjects Win32 API function. This function halts execution of the thread until one or more event objects become signaled (known as blocking the thread). The WaitForMultipleObjects function is defined as:

```
function WaitForMultipleObjects(  
  
    nCount: DWORD; // the number of objects in the array  
  
    lpHandles: PWOHandleArray; // a pointer to an array of object handles  
  
    bWaitAll: BOOL; // wait flag  
  
  
    dwMilliseconds: DWORD // timeout interval  
  
): DWORD; // returns an array index
```

We pass in our array of object handles and the number of objects in the array in the first two parameters. We only want the thread to be blocked until either of the events becomes signaled, so we set the third parameter to FALSE. We want the thread to wait indefinitely until one of the events is signaled, so we set the final parameter to INFINITE. This function call will not return until either of the events becomes signaled.

When the function does return, we need to check its return value. We subtract the constant WAIT\_OBJECT\_0 from this return value to get the index in the array of the object that became signaled. If the Quit event was signaled (indicating we are terminating the application), we should immediately terminate the thread and exit the Execute method. Otherwise, we know it was the mouse event that became signaled, and we are ready to start retrieving mouse input data.



---

**Note:** We set the Quit event to signaled using the SetEvent Win32 API function in the main form's OnDestroy event. We must also use the CloseHandle Win32 API function to dispose of the event handles we created. See the full example on the CD for details.

---

Once past this, we make use of our critical section object. A critical section is a device that is used to synchronize access to variables and objects from multiple threads. This is accomplished by calling the TCriticalSection's Enter method. If one thread calls the Enter method of a critical section after another thread has already called it, this second thread is blocked until the first thread calls the TCriticalSection's Leave method. At this point, the other thread becomes unblocked, and any other thread will be blocked in a similar fashion until the current thread calls the Leave method. By using critical sections, we can allow only one thread at a time access to specific variables and objects. This is important, as this secondary input thread will be drawing to the primary DirectDraw surface, and surfaces can only be accessed by one thread at a time. It will also be updating global variables that track the mouse cursor, as well as a list of mouse click events that are used by the primary thread to determine if the user clicked on something interactive.

Now we are ready to being retrieving input data from our buffer. This is accomplished by calling the IDirectInputDevice's GetDeviceData method. The GetDeviceData method is defined as:

```
function GetDeviceData(  
  
    cbObjectData: DWORD;           // the size of the device data structure  
  
    rgdod: PDIDeviceObjectData;   // a pointer to the device data structure  
  
    var pdwInOut: DWORD;          // a variable indicating the number of elements  
  
    dwFlags: DWORD                // data removal flags  
  
): HRESULT;                       // returns a DirectX error code
```

The first parameter is set to the size of a TDIDeviceObjectData structure. The second parameter is a pointer to an array of TDIDeviceObjectData structures which, upon return, will contain the buffered device data. The third parameter indicates how many elements to retrieve, and should be equal to or less than the number of elements in the array. When this method returns, it will contain the actual number of elements retrieved. The last parameter indicates if the data should be removed from the buffer. Setting this to zero removes the data from the buffer; a value of DIGDD\_PEEK leaves the buffer intact.



---

**Note:** You do not have to remove the entire contents of the buffer at once. You may remove them one at a time or in similar small increments, if desired.

---

This method will return one TDIDeviceObjectData structure for each individual input event that has occurred on the device since the last time data was retrieved. This means there is one event for a button press, one for a button release (even if it was the same button), one for every individual movement along *each* axis, etc. This is why it is important to set the buffer size to an appropriate number, as it is easy to overflow the buffer if a lot of activity is occurring.

The TDIDeviceObjectData structure contains all the information necessary to report any input event on the specific device. The TDIDeviceObjectData structure is defined as:

```
TDIDeviceObjectData = packed record  
  
    dwOfs: DWORD;                 // the device object identifier  
  
    dwData: DWORD;                // the data
```

```
dwTimeStamp: DWORD; // when the event occurred

dwSequence: DWORD; // the event sequence number

end;
```

The first member, dwOfs, indicates what actually caused the input. This value is very specific to the type of device being used. In the case of buffered keyboard data, this value will be one of the DIK\_\* key values. For mice, this could be DIMOFS\_X, DIMOFS\_Y, or DIMOFS\_Z to indicate movement along one of the axes, or DIMOFS\_BUTTON0, DIMOFS\_BUTTON1, or DIMOFS\_BUTTON0 to indicate a change of state in one of the mouse buttons. There are similar DIJOFS\_\* values defined for various buttons and axes on joysticks. The dwData member contains the data for the input event, and again is very specific to both the device and the object on that device that signaled the input event. In the case of one of the axes, dwData will contain either relative or absolute axis movement, in arbitrary units. In the case of buttons, if the high bit is set, the button was pressed; if it is clear, the button was released.

If we enter a loop to retrieve one input event at a time, we can examine all of these members and update various global variables accordingly. In our example on the CD, we add any axis movement to global mouse cursor variables, and then ultimately clip them to the boundaries of our DirectDraw screen resolution. While our example is concerned only with button presses, we do store each mouse button press in a list of button press objects, storing the location of the press and the button that was pressed. We can use this later in our application logic to determine if the user clicked on an interactive game element.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)



**Tip:** The dwTimeStamp member of this structure indicates when the input event actually occurred. The time stamps from subsequent events could potentially be used to determine if the user double-clicked on some item. The dwSequence number, on the other hand, indicates the chronological sequence in which input events were obtained. This is useful for determining the order of input from two separate devices, as DirectInput never assigns the same sequence number for events from different devices, even if they have the same time stamp. This might come in handy if developing a multiplayer game where two people can play from the same machine.

All of this new functionality has been added to our baseline application so we can easily create DirectX applications with mouse support. While we are not going to cover it here, this new baseline application contains some additional code for drawing the mouse cursor on the primary surface from the secondary input thread. This solves the problem we noted earlier about the mouse cursor dependency on the animation frame rate. The following listing illustrates using the event notification and buffered data retrieval methods for obtaining mouse input data.

**Listing 7-10: Retrieving buffered mouse input data**

```
interface
```

```
type
```

```
.  
. .  
.
```

```
{this class is used to track click events}
```

```

TMouseData = class

public
    XPos, YPos: Integer;

    Button: TMouseButton;

    constructor Create(X, Y: Integer; MButton: TMouseButton);

end;

var
    CurX, CurY, OldX, OldY: Integer;           // the mouse position variables
    MouseClickList: TList;                    // the list of click events
    FCritSec: TCriticalSection;               // synchronization object
    FMouseEvents: TWOHandleArray;            // events array

const
    {we're using a 32 X 32 pixel mouse cursor image}
    CURSORWIDTH = 32;
    CURSORHEIGHT = 32;

    {our animated mouse cursor has 8 frames}
    NUMFRAMES = 8;

implementation

constructor TMouseData.Create(X, Y: Integer; MButton: TMouseButton);
begin
    {create and initialize the mouse data object for a click event}
    XPos := X;
    YPos := Y;
    Button := MButton;

end;

```

```

procedure TMouseEvent.Execute;

var
    EventNum: Integer;

    NumElements: DWORD;

    DeviceData: TDIDeviceObjectData;

    DataResult: HRESULT;

    SourceRect,

    DirtyRect: TRect;

begin
    {continuously perform this code until terminated}

    while not Terminated do

        begin

            {wait until one of the events becomes signaled}

            EventNum := WaitForMultipleObjects(2, @FMouseEvent, FALSE, INFINITE);

            {was the quit event signaled? if so, terminate}

            if EventNum-WAIT_OBJECT_0 = QUITEVENT then

                begin

                    Terminate;

                    Break;

                end;

            {we are about to use global variables, so enter the critical section}

            FCritSec.Enter;

            {store the current mouse cursor coordinates}

            OldX := CurX;

            OldY := CurY;

            {begin retrieving buffered data}

            while TRUE do

                begin

```

```

{we only want to retrieve one element at a time}

NumElements := 1;

{retrieve the buffered data}

DataResult := FMouseDevice.GetDeviceData(SizeOf(TDIDeviceObjectData),
                                           @DeviceData, NumElements, 0);

{if we retrieved data, update global variables}
if (DataResult = DI_OK) and (NumElements = 1) then
begin
    case DeviceData.dwOfs of
        {update mouse cursor positions}
        DIMOFS_X      : CurX := CurX + DeviceData.dwData;
        DIMOFS_Y      : CurY := CurY + DeviceData.dwData;
        {update our list of click events}
        DIMOFS_BUTTON0 : MouseClickList.Add(TMouseData.Create(CurX, CurY,
                                                                mbLeft));
        DIMOFS_BUTTON1 : MouseClickList.Add(TMouseData.Create(CurX, CurY,
                                                                mbRight));

    end;
end
else
    {if there were no move input events to retrieve, continue}
    Break;
end;

{clip the global mouse cursor positions to the extents of our
screen resolution}
if CurX < 0 then
    CurX := 0;
if CurX > DXWIDTH-CURSORWIDTH then

```

```

CurX := DXWIDTH-CURSORWIDTH-1;

if CurY < 0 then

    CurY := 0;

if CurY > DXHEIGHT-CURSORHEIGHT then

    CurY := DXHEIGHT-CURSORHEIGHT-1;

{if the mouse position has changed...}

if not ((CurX = OldX) and (CurY = OldY)) then

begin

    {draw the mouse cursor if the old and new positions do not overlap...}

    if (ABS(CurX - OldX)>= CURSORWIDTH) or

        (ABS(CurY - OldY)>= CURSORHEIGHT) then

begin

    {erase the old cursor}

SourceRect := Rect(NUMFRAMES*CURSORWIDTH, 0, NUMFRAMES*CURSORWIDTH+
                    CURSORWIDTH, CURSORHEIGHT);

FPrimarySurface.BltFast(OldX, OldY, FMouseCursors, SourceRect,
                        DBLTFAST_NOCOLORKEY or DBLTFAST_WAIT);

    {save area under new cursor location}

SourceRect := Rect(CurX, CurY, CurX+CURSORWIDTH, CurY+CURSORHEIGHT);

FMouseCursors.BltFast(NUMFRAMES*CURSORWIDTH, 0, FPrimarySurface,
                      SourceRect, DBLTFAST_NOCOLORKEY or
                      DBLTFAST_WAIT);

    {draw new cursor}

SourceRect := Rect(FCursorFrame*CURSORWIDTH, 0, (FCursorFrame*
                    CURSORWIDTH)+CURSORWIDTH, CURSORHEIGHT);

FPrimarySurface.BltFast(CurX, CurY, FMouseCursors, SourceRect,
                        DBLTFAST_NOCOLORKEY or DBLTFAST_WAIT);

end

```

```

else

{...or draw the mouse cursor if the old and new positions do overlap}

begin

  {copy the dirty rectangle to the scratch buffer}

  UnionRect(DirtyRect, Rect(OldX, OldY, OldX+CORSORWIDTH,

                           OldY+CORSORHEIGHT),

            Rect(CurX, CurY, CurX+CORSORWIDTH, CurY+CORSORHEIGHT));

  FMouseScratch.BltFast(0, 0, FPrimarySurface, DirtyRect,

                        DBLTFAST_NOCOLORKEY or DBLTFAST_WAIT);

  {erase the old cursor from the scratch buffer}

  SourceRect := Rect(NUMFRAMES*CORSORWIDTH, 0, NUMFRAMES*CORSORWIDTH+

                    CORSORWIDTH, CORSORHEIGHT);

  FMouseScratch.BltFast(OldX-DirtyRect.Left, OldY-DirtyRect.Top,

                        FMouseCursors, SourceRect, DBLTFAST_NOCOLORKEY or

                        DBLTFAST_WAIT);

  {save the area under the new cursor}

  SourceRect := Rect(CurX-DirtyRect.Left, CurY-DirtyRect.Top, (CurX-

                    DirtyRect.Left)+CORSORWIDTH, (CurY-

                    DirtyRect.Top)+CORSORHEIGHT);

  FMouseCursors.BltFast(NUMFRAMES*CORSORWIDTH, 0, FMouseScratch,

                        SourceRect, DBLTFAST_NOCOLORKEY or

                        DBLTFAST_WAIT);

  {draw new cursor into scratch buffer}

  SourceRect := Rect(FCursorFrame*CORSORWIDTH, 0, (FCursorFrame*

                    CORSORWIDTH)+CORSORWIDTH, CORSORHEIGHT);

  FMouseScratch.BltFast(CurX-DirtyRect.Left, CurY-DirtyRect.Top,

                        FMouseCursors, SourceRect, DBLTFAST_NOCOLORKEY or

                        DBLTFAST_WAIT);

```

```
{draw scratch buffer back to primary buffer}

SourceRect := Rect(0, 0, DirtyRect.Right-DirtyRect.Left,
                  DirtyRect.Bottom-DirtyRect.Top);

FPrimarySurface.BltFast(DirtyRect.Left, DirtyRect.Top, FMouseScratch,
                        SourceRect, DDBLTFAST_NOCOLORKEY or
                        DDBLTFAST_WAIT);

end;

end;

{we are finished using global variables, so we can now leave
 the critical section}

FCritSec.Leave;

end;

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Game Controller Input**

Game controller input programming can be both fun and complex. The fun comes from the fact that game controllers can be of almost any shape and size, and tend to resemble non-computer input devices such as steering wheels, rudder pedals, flight yokes, etc. This incredible variety is also the cause of the complexity. Since game controllers can have almost any number of axes, buttons, sliders, etc., the developer never knows the exact configuration of an attached controller without deducing what is available programmatically. This leads to a level of complexity not found in similar code for interfacing with keyboards and mice.

Game controllers can return a wide range of information. Data for up to three axes can be returned, including their rotation, velocity, acceleration, and force (with the existence of force feedback). Additionally, the state of two sliders, four POV hats, and 128 buttons can also be retrieved. By default, all of the axis information returned from game controllers is absolute, as most axes can move only so far in either direction. However, these values are arbitrary and not based on screen coordinates, etc. Like the mouse, axis data corresponds to the physical movement of the device. However, since game controllers can be almost any shape, the sign of these values and their relation to the direction of physical movement on the device is dependent upon the device itself (such as rotation of an axis). Also unlike the mouse, game controllers do take into account control panel settings, such as calibration. Therefore, you do not have to worry about writing specific code for calibrating a game controller before retrieving its state.



**Tip:** To run the control panel application for game controllers to begin calibration (or any control panel associated with any IDirectInputDevice, for that matter), simply call IDirectInputDevice's RunControlPanel method, passing zero in its two parameters. This allows users to easily calibrate game controllers from within your game.

Game controller input programming has a number of additional nuances that one must be aware of. As stated earlier, the predefined data format structure for joysticks and game controllers does not seem to work appropriately. Thus, in the examples below, we will enumerate all objects on an input device (such as axes, buttons, etc.) and create a custom data format. This will provide support for a wider range of game controllers (theoretically, all game controllers) and will result in more robust and stable code.



## Additional Game Controller Considerations

Above and beyond what is required to set up buffered data retrieval, game controllers require that the developer set up additional properties for each axis before input data can be retrieved. Specifically, the range of values for each axis must be set, as well as their deadzone and saturation levels.

**Range** A minimum and maximum range must be set for each axis. These values are completely arbitrary; DirectInput will scale the data returned from the driver to fit within this range, but these values are meaningful only within the context in which they are used. The center position of the axis will be the center of this specified range.

**Deadzone** The deadzone indicates an area around the center of an axis in which the axis can move, yet the position will be reported as the lowest value within the range. This allows a joystick to wander around a little in the center without reporting movement. This can be useful for setting sensitivity levels. This deadzone range is not subtracted from the overall range; DirectInput compresses the range values into a smaller part of the physical range of the axis. The deadzone value is always between 0 and 10,000, indicating a percentage of the overall range of axis travel in hundredths of degrees. For example, a value of 2,000 indicates we wish the first 20 percent of the physical range of axis travel to fall within the deadzone. This range is based on the center position of the axis, with one-half of the range on either side.

**Saturation** Saturation is similar to the deadzone in that it defines an area at the extent of axis travel in which the axis can move but the position will be reported as the highest value within the range. Again, this can be useful for setting sensitivity levels. However, some joysticks may only report the highest range values when the stick is moved to a diagonal position. Like the deadzone, saturation values are always between 0 and 10,000 (hundredths of degrees), and indicate the percentage of physical axis travel at which the highest range values will be reported. For example, a value of 9,000 indicates that the saturation zone begins at 90 percent of the physical range of axis travel (the last 10 percent of axis travel will fall within the saturation zone). Again, this range is based on the center position of the axis, and the range values themselves are compressed into the remaining part of the physical axis range.



**Caution:** It is possible to set a saturation and a deadzone that completely obscure discrete axis measurements (i.e., saturation and deadzone can be set such that axis values will only be reported as the minimum or maximum value in the specified range).

---



**Note:** Axes on a game pad do not report discrete values; they only report values at the central and extreme ends of the specified range.

---

## Game Controller Objects

Game controllers come with a variety of objects with which the user can provide input data. These objects include axes, POVs, and buttons. Although most game controllers exhibit input objects that are not available on the keyboard and mouse, data retrieved from these objects is not unlike that retrieved from more conventional input devices.

**Lateral and Rotational Axes** Game controller axes come in two flavors: lateral and rotational. A lateral axis moves in a straight line, such as left and right or forward and backward. Conversely, a rotational axis moves in a circular fashion, rotating about the x-, y-, or z-axis. The twisting motion on some joysticks is an example of a rotational axis. The values returned from each axis type is similar, however, and will be within the range specified by the developer.

**POV Hats** A point-of-view hat is neither a button nor an axis (it has more than two positions but reports values in a fixed range). The data reported by a POV is in polar coordinates, with 0 to the north (or top) and subsequent values increasing in a clockwise direction. These values are reported in hundredths of degrees (i.

e., 9000 = 90 degrees). When the POV is centered, the low word of the reported value will be \$FFFF. Most POVs have a granularity of 45 or 90 degrees.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)



**Caution:** While the GetProperty method of IDirectInputDevice can be used to retrieve the granularity, some game controller devices report this information incorrectly.

**Buttons** Game controller button state is reported exactly as keyboard or mouse buttons: If the high bit of the low word is set, the button is down; otherwise it is up.

### Windows Functions

Windows provides several functions to developers for determining game controller capabilities and retrieving input data. For the most part, these functions are incredibly powerful and can report almost as much data as their DirectInput counterparts. For most applications, these functions may supply all the required power.



**Note:** MMSystem must be added to the Uses clause of any unit employing these functions.

Game controller capabilities can be retrieved by using the joyGetDevCaps function. joyGetDevCaps is defined as:

```
joyGetDevCaps(  
  
  uJoyID: UINT;           // the game controller identifier  
  
  lpCaps: PJoyCaps;      // a TJoyCaps structure  
  
  uSize: UINT            // the size of the TJoyCaps structure  
  
): MMRESULT;           // returns an error condition
```



```

CrossHairY := JoyToRelative(0, pbCrossHairSurface.Width, JoyCaps.wYmin,
                            JoyCaps.wYmax, JoyPos.wYpos);

{draw the X, Y axis crosshair}
DrawCrossHair(CrossHairX, CrossHairY);

{draw the Point Of View direction}
if (JoyCaps.wCaps and JOYCAPS_HASPOV) = JOYCAPS_HASPOV then
    DrawPOV(JoyPos.dwPOV);

{draw the position of additional axes}
if (JoyCaps.wCaps and JOYCAPS_HASZ) = JOYCAPS_HASZ then
    shpZBarPos.Width := JoyToRelative(0, pnlZBar.Width, JoyCaps.wZmin,
                                       JoyCaps.wZmax, JoyPos.wZpos);

if (JoyCaps.wCaps and JOYCAPS_HASR) = JOYCAPS_HASR then
    shpRBarPos.Width := JoyToRelative(0, pnlRBar.Width, JoyCaps.wRmin,
                                       JoyCaps.wRmax, JoyPos.dwRpos);

if (JoyCaps.wCaps and JOYCAPS_HASU) = JOYCAPS_HASU then
    shpUBarPos.Width := JoyToRelative(0, pnlUBar.Width, JoyCaps.wUmin,
                                       JoyCaps.wUmax, JoyPos.dwUpos);

if (JoyCaps.wCaps and JOYCAPS_HASV) = JOYCAPS_HASV then
    shpVBarPos.Width := JoyToRelative(0, pnlVBar.Width, JoyCaps.wVmin,
                                       JoyCaps.wVmax, JoyPos.dwVpos);

{test the joystick buttons}
for iCount := 0 to 31 do
    if (ButtonLights[iCount] <> nil) then
        if Longbool(JoyPos.wButtons and Trunc(Power(2, iCount))) then

```

```

        ButtonLights[iCount].Brush.Color := clRed

    else

        ButtonLights[iCount].Brush.Color := clMaroon;

end;

```

---

Unfortunately, some game controllers have objects that provide input that joyGetPosEx simply cannot detect. Additionally, there is no Win32 API support for force feedback capabilities. These functions can allow a developer to rapidly implement game controller support, and may be useful for prototyping.

However, for the best response times, support for any game controller, and force feedback functionality, developers must use DirectInput.



**Figure 7-2:** Win32 joystick input

## DirectInput Initialization

We must follow most of the steps outlined above to initialize DirectInput to receive game controller data. However, since there is no system game controller and we are required to set up our own data format, there are several additional steps that must be taken.

As always, we start by creating the DirectInput object. We must know the exact instance GUID of the game controller device before a DirectInputDevice object can be created, so we must enumerate the attached game controllers. Therefore, we call the EnumDevices method, passing it a type of DIDEVTYPE\_JOYSTICK to indicate that we wish to enumerate game controllers. Usually there is only one game controller connected to the system. However, you may wish to create a list of all attached devices and present it to the user, as we explained earlier in the chapter. Once the instance GUID of the game controller is known, we can create the input device by calling the CreateDevice method. We need to use the Poll method, which is only available in the IDirectInputDevice2 interface, so we'll need to use QueryInterface to retrieve the appropriate object.

Now that we've created the device, we can retrieve its capabilities. We need to know how many buttons, axes, etc., this controller has; this and other information can be retrieved by calling the GetCapabilities method of the IDirectInputDevice object. GetCapabilities is defined as:

```

function GetCapabilities(

var lpDIDevCaps: TDIDevCaps           // a TDIDevCaps structure

): HRESULT;                           // returns a DirectX error code

```

The only parameter to this method is a pointer to a TDIDevCaps structure. This structure describes the capabilities of the queried device, and is defined as:

```

TDIDevCaps = packed record

    dwSize: DWORD;                     // the size of the structure

    dwFlags: DWORD;                    // device flags

```

```
dwDevType: DWORD; // device type identifier
dwAxes: DWORD; // number of axes
dwButtons: DWORD; // number of buttons
dwPOVs: DWORD; // number of POVs
dwFFSamplePeriod: DWORD; // force feedback sample period
dwFFMinTimeResolution: DWORD; // force feedback time resolution
dwFirmwareRevision: DWORD; // firmware version
dwHardwareRevision: DWORD; // hardware version
dwFFDriverVersion: DWORD; // device driver version
end;
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN: 1556226373 Pub Date: 12/01/99**

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Apart from the various members that denote the number of available objects, the dwFlags member may be of particular interest. This member contains a combination of flags that denote various attributes of the overall device. You can check to see if this member contains DIDC\_ATTACHED, indicating that a game controller is physically attached to the machine. Checking it for DIDC\_FORCEFEEDBACK indicates that the device supports force feedback.

In addition to retrieving the capabilities of the overall input device, we may also want to retrieve information on a specific object on that device. This is accomplished by calling the IDirectInputDevice's GetObjectInfo method. GetObjectInfo is defined as:

```
function GetObjectInfo(
var pdidoi: TDIDeviceObjectInstance; // a TDIDeviceObjectInstance structure
dwObj: DWORD; // object identifier
dwHow: DWORD // object identification method
): HRESULT; // returns a DirectX error code
```

The first parameter takes a pointer to a TDIDeviceObjectInstance structure that will contain information about the desired object when the function returns. The second parameter is the device object identifier whose information we wish to retrieve. The final parameter should be set to DIPH\_BYOFFSET. The TDIDeviceObjectInstance structure is another large structure that we will use again later when we construct our custom data format. The TDIDeviceObjectInstance structure is defined as:

```
TDIDeviceObjectInstance = packed record
    dwSize: DWORD; // size of the structure
    guidType: TGUID; // object type GUID
    dwOfs: DWORD; // offset (also object identifier)
    dwType: DWORD; // object type
```



```

dwFlags: DWORD; // attribute flags
tsizName: Array [0..MAX_PATH-1] of CHAR; // object name
dwFFMaxForce: DWORD; // maximum force magnitude
dwFFFForceResolution: DWORD; // force resolution
wCollectionNumber: WORD; // link collection number
wDesignatorIndex: WORD; // designator index
wUsagePage: WORD; // usage page
wUsage: WORD; // usage
dwDimension: DWORD; // dimensional units
wExponent: WORD; // dimension exponent
wReserved: WORD; // reserved

end;

```

Incidentally, this method can be used to simply detect the existence of a specific object. If the return value is not `DI_OK` for a specific object, then we know it doesn't exist on the input device. The following example demonstrates creating the object, retrieving device capabilities, and using the aforementioned technique to determine if any rotational axes are available on the device.

***Listing 7-12: Creating the input device and querying capabilities***

---

```

procedure TestForRotationalAxis(Offset: Integer; AxisName: string;
                                Panel: TPanel; Shape: TShape);

var
    ObjectInst: TDIDeviceObjectInstance;

begin
    {we will be retrieving information on a device object, so prepare a
    data structure}

    FillChar(ObjectInst, SizeOf(TDIDeviceObjectInstance), 0);

    ObjectInst.dwSize := SizeOf(TDIDeviceObjectInstance);

    {this simply determines if the specified object exists. we are using this
    here to specifically test for rotational axes (see where this function
    is called), but we could also use it to determine the existence of
    other axes or even buttons}

```

```
if FDirectInputDevice.GetObjectInfo(
    ObjectInst, Offset, DIPH_BYOFFSET) = DI_OK then
with frmJoyTestBed.memControllerInfo.Lines do
begin
    Add(    +AxisName+' rotational axis');
    Panel.Color := clMaroon;
end
else
begin
    Panel.Color := clBtnFace;
    Shape.Width := 0;
end;
end;

procedure TfrmJoyTestBed.FormActivate(Sender: TObject);
var
    FJoyList: TStringList;
    iCount: Integer;

    DevCaps: TDIDevCaps;

    TempDevice: IDirectInputDevice;

    .
    .
    .

begin
    {create the DirectInput object}
    DirectInputCreate(hInstance, DIRECTINPUT_VERSION, FDirectInput, nil);
```

```

{create a list to hold information on each available game controller}

FJoyList := TStringList.Create;

{enumerate attached game controllers}

FDirectInput.EnumDevices(DIDEVTYPE_JOYSTICK, EnumInputDevs, FJoyList,
                        DIEDFL_ATTACHEDONLY);

{we'll add this information to a list box in a dialog box, and display this
to the user so they can choose their desired input controller}

frmDevices := TfrmDevices.Create(Self);

for iCount := 0 to FJoyList.Count-1 do
    frmDevices.lbxJoyDevices.Items.Add(FJoyList[iCount]);

frmDevices.ShowModal;

{create the input device based on the input controller specified by the user}
DXCheck( FDirectInput.CreateDevice(PGUID(FJoyList.Objects[0])^, TempDevice,
                                    nil) );

{close this dialog box, and delete the list of GUIDs}

frmDevices.Release;

for iCount := 0 to FJoyList.Count-1 do
    Dispose(PGUID(FJoyList.Objects[0]));

FJoyList.Free;

{we need the new version of the input device interface, so retrieve it}
try
    DXCheck(TempDevice.QueryInterface(IDirectInputDevice2, FDirectInputDevice));
finally
    TempDevice := nil
end;

{retrieve the capabilities of this device}

```

```

DevCaps.dwSize := SizeOf(DevCaps);

FDirectInputDevice.GetCapabilities(DevCaps);

.
.
.

{determine if the controller is unplugged}
if (DevCaps.dwFlags and DIDC_ATTACHED) <> DIDC_ATTACHED then
    ShowMessage(Controller is unplugged');

{create button indicators}
for iCount := 0 to DevCaps.dwButtons-1 do
begin
    ButtonLights[iCount]           := TShape.Create(GroupBox2);
    ButtonLights[iCount].Left       := 96+((iCount mod 5)*(49+5));
    ButtonLights[iCount].Top        := 48+((iCount div 5)*(17+5));
    ButtonLights[iCount].Width      := 49;
    ButtonLights[iCount].Height     := 17;
    ButtonLights[iCount].Brush.Color := clMaroon;
    ButtonLights[iCount].Parent     := GroupBox2;
end;

{indicate if a POV is available}
if DevCaps.dwPOVs > 0 then
begin
    Label9.Enabled := TRUE;
    Panel2.Color := clWhite;
end
else
begin
    Label9.Enabled := FALSE;

```

```

    Panel2.Color := clBtnFace;

end;

{display controller capabilities}

with DevCaps, memControllerInfo.Lines do

begin

    Add(Number of buttons: +IntToStr(dwButtons));

    Add(Number of axes: +IntToStr(dwAxes));

    Add(Number of POVs: +IntToStr(dwPOVs));

    Add(' ');

    Add(Rotational Axes:');

    TestForRotationalAxis(DIJOFS_RX, RX', pnlRXBar, shpRXBarPos);

    TestForRotationalAxis(DIJOFS_RY, RY', pnlRYBar, shpRYBarPos);

    TestForRotationalAxis(DIJOFS_RZ, RZ', pnlRZBar, shpRZBarPos);

end;

.

.

.

end;

```

---

**Setting the Data Format** Now that the device has been created and its capabilities have been determined, we need to set the data format. As mentioned earlier, we must generate our own data format programmatically to ensure that the greatest number of input devices are supported. We begin by declaring a variable of type `TDIDataFormat`. `TDIDataFormat` is defined as:

```

TDIDataFormat = packed record

    dwSize: DWORD;           // size of this structure

    dwObjSize: DWORD;       // size of a TDIOBJECTDATAFORMAT structure

    dwFlags: DWORD;        // attribute flags

    dwDataSize: DWORD;     // size of data packet

```

```
dwNumObjs: DWORD; // number of objects in the array  
rgodf: PDIOBJECTDATAFORMAT; // array of TDIOBJECTDATAFORMAT structures  
end;
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

This structure carries information that describes a device's data format. The first member should be set to the size of the TDIDataFormat structure. The second member is set to the size of a TDIOBJECTDataFormat structure. The third member is a series of flags that describe certain attributes of the data format. For our purposes, we can set this member to DIDF\_ABSAXIS to indicate that axes measurements are absolute. The fourth member indicates the size of the data packet returned from the driver. This value must be a multiple of 4, and must exceed the largest offset value for an object's data in the packet. For our purposes, we'll set this to the size of the TDIJoyState2 structure. The fifth member indicates the number of objects that are in the sixth member, which is a pointer to an array of TDIOBJECTDataFormat structures. The TDIOBJECTDataFormat structure is defined as:

```
TDIOBJECTDataFormat = packed record

    pguid: ^TGUID;           // object GUID

    dwOfs: DWORD;           // object offset

    dwType: DWORD;          // object type

    dwFlags: DWORD;         // attribute flags

end;
```

The TDIOBJECTDataFormat structure contains information about a specific object on the input device, such as an individual axis or button. The first member is the object's unique identifying GUID. The second member is the object's offset within the data packet. This is also the object identifier that we have used in other methods. The third member indicates the object type (such as a button or axis), and the final member contains attribute flags for the object.

After we initialize the first four members of the TDIDataFormat structure, we must begin building the array of TDIOBJECTDataFormat structures for each object on the device. To do this, we must employ the EnumObjects method. EnumObjects works the same as all other enumeration functions in that it calls a callback function once for each object on the input device. The EnumObjects method is defined as:

```
function EnumObjects(

lpCallback: TDIEnumDeviceObjectsCallbackA;    // the callback function
```

```

pvRef: Pointer; // application-defined value
dwFlags: DWORD // enumeration flags
): HRESULT; // returns a DirectX error code

```

The first parameter is a pointer to the callback function, the second parameter is a pointer to an application-defined value, and the final parameter is a series of flags that control the scope of enumeration. For our purposes, we'll set this final parameter to DIDFT\_ALL to indicate we wish to enumerate all devices. The callback function itself must be defined as:

```

function TDIEnumDeviceObjectsCallbackA(
var lpddoi: TDIDeviceObjectInstance; // a TDIDeviceObjectInstance structure
pvRef: Pointer // the application-defined data
): BOOL; stdcall; // returns TRUE or FALSE

```

The first parameter is an initialized TDIDeviceObjectInstance structure containing all the information we need about this specific object. The only other parameter is a pointer to the application-defined value. This callback function should return TRUE to continue enumeration or FALSE otherwise.

As we are enumerating, for each object, we can save the GUID, offset, and type as reported from the TDIDeviceObjectInstance structure into our array of TDIOBJECTDATAFORMAT structures. We also must keep a count of the number of objects found.

When enumeration completes, we will have a correctly initialized TDIDataFormat structure, complete with an array of correctly initialized TDIOBJECTDATAFORMAT structures. This entire data structure is then sent to the SetDataFormat method to finally set the correct data format for the game controller. This is illustrated in the following example.

***Listing 7-13: Setting the data format***

---

```

procedure SetDataFormat;

function DIEnumDeviceObjectsProc(const peff: TDIDeviceObjectInstance;
                                pvRef: Pointer): HRESULT; stdcall;

begin
    {if the offset is not beyond the indicated data size... (useful if
     we were to use the TDIJoyState data structure instead) }
    if peff.dwOfs < FDataFormat.dwDataSize then
    begin
        {save the type GUID for this object}
        FDataFormatGUIDs[FDataFormat.dwNumObjs] := peff.guidType;
    end
end

```



```

    {add the object information to our custom made data format}
    with FDataFormatObjects[FDataFormat.dwNumObjs] do
    begin
        pguid := @FDataFormatGUIDs[FDataFormat.dwNumObjs];
        dwOfs := peff.dwOfs;
        dwType := peff.dwType;
        dwFlags := 0;
    end;

    {if this object is an axis, store it in our axis list}
    if (peff.dwType and DIDFT_AXIS) > 0 then
    begin
        FAxisList[NumAxes] := peff.dwOfs;
        Inc(NumAxes);
    end;

    {increment the number of objects found}
    Inc(FDataFormat.dwNumObjs);
end;

{continue enumeration}
Result := DIENUM_CONTINUE;
end;

begin
    {enumerate all objects on this input device. this will build our custom data
    structure programmatically with the correct information for each object}
    FDirectInputDevice.EnumObjects(@DIEnumDeviceObjectsProc, nil, DIDFT_ALL);

    {set the data format for the input device}
    DXCheck( FDirectInputDevice.SetDataFormat(FDataFormat) );

```

```

end;

procedure TfrmJoyTestBed.FormActivate(Sender: TObject);
var
    .
    .
    .

begin
    .
    .
    .

    {initialize our axis tracking variable}
    NumAxes := 0;

    {now we need to begin creating the data format structure and setting the
    data format, so we must initialize the overall data format structure}
    with FDataFormat do
    begin
        dwFlags := DIDF_ABSAXIS;
        dwDataSize := SizeOf(TDIJoyState2);
        dwSize := SizeOf(FDataFormat);
        dwObjSize := SizeOf(TDIObjectDataFormat);
        dwNumObjs := 0;
        rgodf := @FDataFormatObjects;
    end;

    {create and set a data format specifically made for this device}

```

```
SetDataFormat ;
```

```
.  
. .  
. . .
```

```
end ;
```

---

***Setting Axis Properties*** With the data format set, we are now ready to start setting various properties for all of the axes.

As discussed above, we must set the range, deadzone, and saturation for each axis. To accomplish this task, we'll need variables of type TDIPropRange for the range and TDIPropDWord for the deadzone and saturation. As we illustrated in the mouse example, we must initialize the various size members of this structure. Unlike the mouse example, however, we will set the dwObj members of both structures to the object identifier for each individual axis. We could set this to DIPH\_DEVICE, setting the properties for all axes on the device, but we may want to set different ranges, deadzones, and saturations for each device. For the TDIPropRange structure, we'll set the lMin and lMax members to 1,000 and 1,000, respectively. For the TDIPropDWord structure, we'll set the dwData member to 1,000 for deadzone properties (indicating 10 percent deadzone), and 9,000 for saturation properties (indicating that the last 10 percent is saturation). We can then call the SetProperty method once for each axis and each property. Once this is complete, we can finally set our cooperative level and acquire the device so we can start retrieving input. This is illustrated in the following listing.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**ITKNOWLEDGE.COM**<sup>SM</sup>  
Need IT. Find IT. Know IT.

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 7-14: Setting axis properties*

---

```
procedure TfrmJoyTestBed.FormActivate(Sender: TObject);
```

```
var
```

```
    iCount: Integer;
```

```
    RangeProp: TDIPropRange;
```

```
    DataProp: TDIPropDWord;
```

```
    .
```

```
    .
```

```
    .
```

```
begin
```

```
    .
```

```
    .
```

```
    .
```

```
{we must now set the range for each axis, so initialize a range  
property structure}
```

```
RangeProp.diph.dwSize := SizeOf(TDIPropRange);
```

```
RangeProp.diph.dwHeaderSize := SizeOf(TDIPropHeader);
```

```
RangeProp.diph.dwHow := DIPH_BYOFFSET;
```

```
RangeProp.lMin := -1000;
```

```
RangeProp.lMax := 1000;
```

```
{we must also set deadzones and saturations for each axis, so  
initialize a DWord property structure}
```

```
DataProp.diph.dwSize := SizeOf(TDIPropDWord);
```

```
DataProp.diph.dwHeaderSize := SizeOf(TDIPropHeader);
```

```
DataProp.diph.dwHow := DIPH_BYOFFSET;
```

```
{for each axis...}
```

```
for iCount := 0 to NumAxes-1 do
```

```
begin
```

```
    {set its range...}
```

```
    RangeProp.diph.dwObj := FAxisList[iCount];
```

```
    DXCheck( FDirectInputDevice.SetProperty(DIPROP_RANGE, RangeProp.diph) );
```

```
    {its deadzone... (this value is in 100ths of a percent)}
```

```
    DataProp.dwData := 1000;
```

```
    DataProp.diph.dwObj := FAxisList[iCount];
```

```
    DXCheck( FDirectInputDevice.SetProperty(DIPROP_DEADZONE, DataProp.diph) );
```

```
    {and its saturation (this value is also in 100ths of a percent)}
```

```
    DataProp.dwData := 9000;
```

```
    DXCheck( FDirectInputDevice.SetProperty(DIPROP_SATURATION, DataProp.diph));
```

```
end;
```

```

.
.
.

{set the cooperative level}
DXCheck( FDirectInputDevice.SetCooperativeLevel(Handle, DISCL_EXCLUSIVE or
                                                DISCL_FOREGROUND) );

{finally, acquire the device so we can start retrieving information}
DXCheck( FDirectInputDevice.Acquire );

{enable the timer that retrieves the controller state}
tmrJoyPoll.Enabled := TRUE;

end;

```

---

## Reading Input Data

Compared to setting up DirectInput for game controllers, actually reading their input data is rather simple.

Game controllers usually do not generate hardware interrupts. As such, Windows (and the device driver) is not informed that an input event has occurred on the device. Thus, for almost all game controllers, the Poll method must be called to instruct the driver to retrieve device information before the application can retrieve the device's state. The Poll method takes no parameters. If a game controller does not require polling, this method has no effect. Therefore, it's usually a good practice to always call the Poll method regardless of the game controller or its driver.

Once this is done, the application can call the GetDeviceState method, passing in a pointer to a TDIJoyState2 structure. Although you can use buffered data techniques, most applications can get by with retrieving immediate game controller data. The TDIJoyState2 structure is defined as:

```

TDIJoyState2 = packed record

    lX: Longint;           // x-axis position
    lY: Longint;           // y-axis position
    lZ: Longint;           // z-axis position
    lRx: Longint;          // x-axis rotation
    lRy: Longint;          // y-axis rotation
    lRz: Longint;          // z-axis rotation
    rglSlider: Array [0..1] of Longint; // extra axes positions
    rgdwPOV: Array [0..3] of DWORD;    // POV directions

```

```

rgbButtons: Array [0..127] of BYTE;      // 128 buttons

lVX: Longint;                            // x-axis velocity
lVY: Longint;                            // y-axis velocity
lVZ: Longint;                            // z-axis velocity
lVRx: Longint;                           // x-axis angular velocity
lVRy: Longint;                           // y-axis angular velocity
lVRz: Longint;                           // z-axis angular velocity
rglVSlider: Array [0..1] of Longint;     // extra axes velocities
lAX: Longint;                            // x-axis acceleration
lAY: Longint;                            // y-axis acceleration
lAZ: Longint;                            // z-axis acceleration
lARx: Longint;                           // x-axis angular acceleration
lARy: Longint;                           // y-axis angular acceleration
lARz: Longint;                           // z-axis angular acceleration
rglASlider: Array [0..1] of Longint;     // extra axes accelerations
lFX: Longint;                            // x-axis force
lFY: Longint;                            // y-axis force
lFZ: Longint;                            // z-axis force
lFRx: Longint;                           // x-axis torque
lFRy: Longint;                           // y-axis torque
lFRz: Longint;                           // z-axis torque
rglFSlider: Array [0..1] of Longint;     // extra axes forces

end;
```

This rather large structure defines the state of every object on the game controller. The position and rotation of various axes is easily retrievable, as is the state of buttons, POVs, etc. The following listing illustrates how to retrieve the current state of a game controller.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) | 
 [Contact Us](#) | 
 [About Us](#) | 
 [Privacy](#) | 
 [Ad Info](#) | 
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 7-15: Retrieving immediate game controller input data*

```
procedure TfrmJoyTestBed.tmrJoyPollTimer(Sender: TObject);
var
    DevState: TDIJoystate2;
    iCount: Integer;
begin
    {more often than not, game controllers do not generate hardware
    interrupts. thus, we have to poll the device driver in order to
    get the current state of the device.}
    FDirectInputDevice.Poll;

    {retrieve the state of all objects on the device}
    DXCheck( FDirectInputDevice.GetDeviceState(SizeOf(DevState), @DevState) );

    {we're assuming here that the game controller has an X and Y axis
    (as joysticks and gamepads do), so draw a crosshair}
```



```

DrawCrossHair((DevState.lX + 1000) div 20, (DevState.lY + 1000) div 20);

{draw slider bars for any other axes}
for iCount := 2 to NumAxes-1 do
    case FAxisList[iCount] of
        DIJOFS_Z   : shpZBarPos.Width := (DevState.lZ + 1000) div 20;
        DIJOFS_RX  : shpRXBarPos.Width := (DevState.lRx + 1000) div 20;
        DIJOFS_RY  : shpRYBarPos.Width := (DevState.lRy + 1000) div 20;
        DIJOFS_RZ  : shpRZBarPos.Width := (DevState.lRz + 1000) div 20;
    end;

{if a POV is available (availability of which should probably be placed
in a global variable as opposed to checking the enabled state of a
label), then draw a crosshair}
if Label9.Enabled then
    DrawPov(DevState.rgdwPOV[0]);

{test the joystick buttons}
for iCount := 0 to 127 do
    if ButtonLights[iCount] <> nil then
        if (DevState.rgbButtons[iCount] and $80) = $80 then
            ButtonLights[iCount].Brush.Color := clRed
        else
            ButtonLights[iCount].Brush.Color := clMaroon;
        end;
    end;
end;

```

---

## Summary

In this chapter, we discussed several techniques for retrieving user input data from various input devices. We examined both common Win32 API functions and their DirectInput counterparts for reading the state of keyboard, mice, and game controller devices. We also covered various techniques for retrieving this data, including immediate and buffered data as well as polling and event notification. When coding user input data

routines using DirectInput or Win32 API functions, it is important to keep these points in mind:

- Input is one of the three fundamental parts of a game, and as such is important to overall game enjoyment. Exactly how user input will be translated into game element manipulation is dependent on the game itself. Flight simulators need input that affects the direction of flight, or perhaps even the selection and use of a weapon. Strategy games, by contrast, need input that determines which token is selected and its direction of movement or combat.
- Input devices go beyond the standard mouse and keyboard. There are now a wide variety of peripheral game controllers in all shapes and sizes. Rudder pedals, steering wheels, gamepads, throttles, and all manner of sundry and arcane input devices are now available, making some games much more enjoyable to play. Even the standard joystick has become incredibly complex, with many sporting impressive arrays of buttons, switches, sliders, and even tactile responses known as force feedback.
- An important user input consideration is support for user configurable controls. Even though the developer has determined which input devices will be supported and how they interact with game elements, individual users may find this structure somewhat restrictive based on their particular preferences.
- DirectInput is the DirectX component responsible for providing user input from all manner of input devices, such as keyboards, mice, and game controllers. DirectInput offers many features above and beyond what is available through the Win32 API. As with all DirectX components, DirectInput gives the developer more control over user input than the Windows API, while also providing an improvement in performance.
- Similar to DirectDraw, DirectInput programming consists of creating a DirectInput object and several DirectInputDevice objects. The DirectInput object itself acts as a liaison between the application and input device drivers. Individual DirectInputDevice objects represent the application's interface directly to input hardware, or more specifically, that hardware's device driver.
- Where DirectInput is concerned, a button is considered to be any switch that has an on and off position. There is no distinction made between buttons on a mouse, keyboard, or joystick. An axis has a range of positions where its value is affected by the physical movement of a device in some particular direction. Some axes can move an infinite distance in a particular direction, and are measured in relative values. Other axes can physically move only a specific distance, and are measured in absolute values.
- Buffered data retrieval simulates the mechanism by which Windows reports input. Individual input events are stored in a buffer, which takes the form of an array of data structures containing the various axis movement values or button clicks. Immediate data retrieval does not report a stream of singular events, but returns the current state of the device at the time of the function call.
- Polling involves checking the state of the input device at the time that the input is needed. Typically, this is performed in the game loop. Event notification, on the other hand, is very similar to event-driven programming. This is a mechanism by which DirectInput will notify the application when new input data is available, and is best paired with buffered data retrieval.
- As discussed in the DirectDraw tutorial, a cooperative level determines how the application works with other applications. As with other DirectX components, DirectInput gives the developer a choice as to how the application will share input devices with other applications, in terms of who can gain access to the device and if input can be retrieved based on application focus. Bear in mind that Windows requires exclusive access to the keyboard at all times, so that functions such as Alt+Tab and Ctrl+Alt+Delete are always available. Thus, an application can only obtain non-exclusive access to the keyboard.
- Once a device is set up, the application cannot receive input data until it instructs DirectInput that it is ready to do so. This process is known as acquiring the device. Bear in mind that a device can be forcibly unacquired by the application. This may happen when another application receives focus or acquires the device in exclusive mode.
- Several steps are required in order to set up DirectInput and begin interfacing with user input devices. In general you must create the DirectInput object, enumerate input devices, create the input device, set the data format, set the cooperative level, set device properties, and then acquire the device.
- While the steps required for initializing and setting up DirectInput and DirectInputDevices are similar for every device, exactly how the input data is retrieved and extracted varies from device to device. This will also depend on whether you will be using immediate data retrieval or buffered data retrieval. This leads to subtle nuances in which initialization steps are required, and how the data returned from certain methods is interpreted.
- Retrieving keyboard data is the easiest. The GetAsyncKeyState Win32 API function could potentially be used if the full power of DirectInput was not needed. However, you would not be able to differentiate between left and right versions of certain keys, such as Shift or Ctrl.
- Mouse data is best retrieved using a buffer coupled with event notification. This requires setting up a secondary thread and using critical sections to synchronize variable access. The Win32 API function

GetCursorPos function could potentially be used instead, but this function reports absolute values relative to the screen.

- Game controller data input retrieval is very complex. We must set up a custom data format, but this complexity results in support for a wider range of devices. We must also set up the range, deadzone, and saturation properties for each axis. We could use the joyGetPosEx function, but this function only supports up to six axes and 32 buttons, and has no support for force feedback.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

- ▶ Search Tips
- ▶ Advanced Search

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 8 Force Feedback

This chapter covers the following topics:

- Creating and using standard force feedback effects
- Creating and using custom force feedback effects
- Modifying a force feedback effect on the fly
- Shaping force feedback effects with envelopes

Cutting-edge computer games offer an entertainment experience very similar to Hollywood movies. Such games feature almost photo-realistic environments, complete with appropriate lighting, 3-D sound effects, and music to enhance the emotional impact. However, consider how much more realistic and immersive a gaming experience would be if you could feel the blow of your axe against an enemy's armor, or the resistance of gravity and air as you wheel your jet fighter through a tight turn. This is what force feedback technology is all about.

Force feedback allows the game developer to add the illusion of mass and momentum to game elements previously only represented as dots of colored light. Like sound and music, it adds an intangible realistic element to a game that could not be accomplished otherwise. Force feedback support appeared in DirectX 5, and since then, game manufacturers have slowly implemented force feedback support into their products. Like sound and music, no game absolutely requires force feedback in order to make it entertaining. On the other hand, unless you're making a game like Solitaire, it would be crazy to ignore the exciting functionality this new technology offers; it is certainly a requirement for action games to actively compete in the retail market.

The force feedback API is part of DirectInput. At the time of this writing, force feedback functionality is available primarily on joysticks and a few wheels, but the technology is designed to work with devices of any shape and size. The result is a very flexible yet complex system. For our purposes, we will focus our discussion of force feedback as it applies to joysticks, but keep in mind that the concepts to follow can be applied to any type of force feedback device.



---

**Tip:** These examples were written and tested specifically on the Microsoft SideWinder® Force Feedback Pro joystick, but they should work on any force feedback device with an x- and y-axis.

---

## Features

Force feedback has no equivalent in the Win32 API world, and as such cannot be compared to any existing functionality. As we've discussed, the purpose of force feedback is to provide the user with tactile output based on conditions within the game world. This is accomplished through either an active push on one or more of the axes supported by the device, or through some form of resistance along those axes when the user manipulates the device. Considering this, DirectInput's force feedback functionality provides the following features of interest to game developers:

- Several different standard effects representing a wide range of tactile responses
- Device-specific effects implemented by the device driver (when available)
- Modification of effects on the fly (when available)
- Creation of customized effects
- Triggered effects controlled by the device

In the following sections we will discuss how the Delphi game developer can take advantage of each of these features to create tactile feedback that provides a new level of realism to game environments.

## Functional Overview

In order to begin using force feedback, the application must already have an initialized `IDirectInputDevice2` object. An elaborate data structure is then initialized depending on the type of effect desired. `IDirectInputDevice2` uses this initialized data structure to create an `IDirectInputEffect` object.

The `IDirectInputEffect` object represents the developer's direct interface to the force feedback hardware. This is not to be confused with the functionality provided by the `IDirectInputDevice2` object; `IDirectInputDevice2` is responsible for communicating user input from the device, whereas `IDirectInputEffect` is responsible for outputting tactile feedback responses to the device. Through the `IDirectInputEffect` interface, the application can query the force feedback hardware's capabilities, start and stop the effect, modify the effect on the fly, and control other aspects of the force feedback hardware itself.

Once an effect is created, it must be downloaded to the device. This typically happens automatically, as we'll discuss below. The act of downloading an effect tells the driver to prepare the effect for playback. How this is accomplished is determined by the driver, but typically the effect's parameters are placed into hardware memory so that the device will not need to communicate with the system in response to axis movement and button presses. However, a device can only handle a finite number of effects, and may return a `DIERR_DEVICEFULL` error. The workaround is to simply unload another, hopefully unused, effect and then try again.

Once an effect is created, it must be explicitly started. Effects can play infinitely or for a specified amount of time, and can be stopped if necessary. Alternatively, when the effect is being created, it can be associated with a button on the device. The device driver will then start the effect when the button is pressed. Triggered effects such as this will stop when the effect duration has been reached or when the button is released, whichever comes first. This functionality is totally independent of the application. Some devices even support repeating an effect while the button is held down, perfect for re-creating the kickback of a machine gun or other automatic weapon.

## Basic Concepts

When discussing force feedback effects, it is necessary to understand several general concepts as they apply to the effect itself. These concepts define the action of the effect, such as its strength and direction. Such

concepts include kinetic forces, conditions, magnitude, coefficient, gain, duration, sample period, axes, direction, and envelope.

## Kinetic Force

A kinetic force is an effect that actively pushes on one or more axes. This push can be in a single direction or in alternating directions such as occurs with periodic effects. This active push is applied regardless of the position or movement of the axis (or axes). An example of this is a push on the joystick in the direction of the user to simulate recoil from firing a rocket.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)



**kinetic force:** An active push on one or more axes on a force feedback device.



**Note:** Force feedback devices are required to have a safety feature installed that prevents the playback of effects unless the user is actively gripping the device. This helps avoid potential injury that might occur if the user is trying to pick up the device while a kinetic force effect is playing.

### Condition

Unlike a kinetic force, a condition is a resistance or force that is applied to one or more axes in response to their movement or position. For example, the self-centering effect that a force feedback joystick automatically exhibits is a condition that tries to move the stick back to its center position. Another example would be to apply a dampening effect to the stick in order to re-create the resistance of moving around in an underwater environment. There are currently four different types of conditions, which we'll cover in more detail shortly.



**condition:** A resistant force applied to one or more axes on a force feedback device in response to the movement or position of those axes.

### Magnitude

Magnitude is a measurement of the strength of a kinetic force, relative to the total maximum force that can be output by the device. The magnitude of a force can be expressed as an integer value between -10,000 and 10,000, where positive values represent a force in one direction and negative values represent the same force in the opposite direction. The DI\_FF NOMINALMAX constant defined by the DInput unit, initialized to 10,000, represents the total maximum force that can be exerted by the device on any given axis. This constant



allows the developer to define the magnitude of a force as a percentage of the total maximum force available on the device (i.e., `DI_FFNO`



**magnitude:** A measurement of the strength of a kinetic force output by a force feedback device, relative to the total maximum force available on the device itself.

## Coefficient

Coefficient is conceptually similar to magnitude, except that it is a measurement of the strength of resistance to movement along an axis. Like magnitude, this value is expressed as an integer between 10,000 and -10,000, where positive values represent resistance and negative values represent acceleration (i.e., it has the effect of pulling along the direction of motion). Again, it is best to use the `DI_FFNO` constant to define a coefficient as a percentage of the total maximum force.



**coefficient:** A measurement of the strength of resistance output by a force feedback device in response to the movement of an axis, relative to the total maximum force available on the device itself.



**Note:** In the case of a spring condition (discussed later in this chapter), a negative coefficient has the effect of pushing the axis away from the defined neutral point.

## Duration

Duration defines how long the effect will play, measured in microseconds. The constant `DI_SECONDS`, found in the `DInput` unit, can be used to define the duration in terms of one-second intervals (or fractions thereof). The constant `INFINITE` can also be used to indicate an effect with no duration. This type of effect will play until explicitly stopped or the `IDirectInputEffect` object is freed. In the case of triggered effects, playback is stopped when the duration expires or the button is released.



**Tip:** An infinite duration is typically used for condition effects or periodic forces; it typically doesn't make sense to specify an infinite duration for a constant or ramp force.

## Sample Period

Sample period defines the interval between actual changes in the strength of a force, measured in microseconds. Setting this to a large value will cause the strength of the effect to be adjusted fewer times over the course of its duration, which may cause a ratcheting sensation. Setting this value to zero allows for the smoothest change in relative force strength.

## Gain

Gain acts as a scaling factor for magnitudes and coefficients. It is expressed as a value between 0 and 10,000, indicating the percentage by which to scale the magnitude or coefficient. For example, 10,000 indicates no scaling, whereas 5,000 indicates 50 percent of the specified magnitude or coefficient. Gain can be applied to the entire device or only to individual effects. The `DI_FFNO` constant can again be used here to easily define a scaling factor as a percentage of the nominal value.





**gain:** A scaling factor for magnitudes and coefficients, measured in hundredths of a percent.



**Note:** Use the SetProperty method of the IDirectInputDevice interface in order to set the gain for the entire device.

## Axes

Axes determine what axis (or axes) of movement on the force feedback device will be affected by the force feedback effect. This also indirectly determines the available directions for an effect, depending on how many axes will be affected by the force feedback output. Of course, the axes available are totally dependent on the type of device. For example, most force feedback joysticks allow output along the x and y axes, as shown below. Other devices, such as steering wheels, may define these axes differently. Axes can be identified by using the DIJOFS\_\* constants.



**Figure 8-1:** Axes of force feedback output

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

**7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Direction**

Direction can be a complicated concept when it comes to force feedback. Bear in mind that the force feedback API was designed to work not only with such devices as joysticks, but with other devices like steering wheels, throttle controls, and devices that can move in three dimensions. In a nutshell, direction determines the bearing of an effect along an axis (or axes) when its magnitude or coefficient values are positive. Negative values act in the opposite of the defined direction.

Single axis directions are easily handled, as you can simply control the sign of the magnitude or coefficient in order to define the direction. In the case of multiple axes, however, there are three available coordinate systems: spherical, polar, and Cartesian. The Cartesian coordinate system may be useful if force feedback is being used in conjunction with Direct3D. The spherical coordinate system is necessary for defining the direction of an effect on three or more axes of movement. Since our discussion is focused on joystick programming, however, we'll use the polar coordinate system. The polar coordinate system is perhaps the easiest to use, as it requires only the compass direction of the force, measured in hundredths of a degree. The DI\_DEGREES constant is defined in the DInput unit, and can be used to easily specify a direction in terms of degrees.

Bear in mind that the direction of a force is in terms of the direction from which the force originates. For example, a force from the north would push the stick toward the user, and would thus have a direction of zero. A force from the west, pushing the stick to the right, would have a direction of  $90 * DI\_DEGREES$ . The following illustration graphically explains the concept of direction as it applies to a joystick.



**Figure 8-2:** Forcefeedback direction

**Envelope**

An envelope is a set of values that modify the magnitude of a force at the beginning and end of playback, and can be applied to all forces except conditions. This allows a developer to define the attack and fade of a force, like a musical note. We'll examine envelopes in greater detail later in the chapter.



**envelope:** A series of values that shape a force at the beginning and end of playback.

## Force Feedback Effects

Force feedback effects are divided into six different categories: conditions, constant forces, periodic forces, ramp forces, device-specific effects, and custom forces. Each one of these is defined by an overall basic effect structure, which describes certain global attributes for the effect such as duration and direction, and a more type-specific structure, which describes very specific effect attributes depending on its category. Some of these categories even have subtypes. This makes for a rather complex system, but it should be more than adequate for just about any tactile feedback desired.

### Base Effect Structure

All force feedback effects are defined by a primary structure called `TDIEffect`. This structure is used when creating the effect as well as modifying the effect after it has been created. This complex structure defines the overall parameters of the effect and includes members that hold other structures that further define the effect, and are dependent on the type of effect desired. The `TDIEffect` structure is defined as:

```
TDIEffect = packed record

    dwSize : DWORD;                // the size of the structure, in bytes

    dwFlags : DWORD;               // indicates the directional coordinate
                                   system and axes/button identification

    dwDuration : DWORD;            // duration of the effect in microseconds

    dwSamplePeriod : DWORD;        // the sample period, in microseconds

    dwGain : DWORD;                // the gain for the specific effect

    dwTriggerButton : DWORD;       // trigger button identifier

    dwTriggerRepeatInterval : DWORD; // repetition interval, in microseconds

    cAxes : DWORD;                 // number of axes involved in the effect

    rgdwAxes : ^DWORD;             // array of axis identifiers

    rglDirection : ^longint;       // array of direction bearings

    lpEnvelope : PDIEnvelope;      // pointer to a TDIEnvelope structure

    cbTypeSpecificParams : DWORD;  // the size of the structure in the
                                   lpvTypeSpecificParams member

    lpvTypeSpecificParams : pointer; // pointer to type-specific data structure

end;
```

As usual with all DirectX data structures, the `dwSize` member should be set to `SizeOf(TDIEffect)`. The `dwFlags` member indicates both the directional coordinate system as well as the method by which axes and buttons are to be identified. For our purposes, we'll use the values `DIEFF_POLAR` to indicate a polar coordinate system and `DIEFF_OBJECTOFFSETS` to indicate button and axis data format offsets (using the OR operator to combine them). Most of the other members, such as duration, sample period, and gain, correspond directly to the concepts

covered earlier, and should be set accordingly. The `dwTriggerButton` member indicates the button that will start the effect. This member can be set to a button offset (i.e., `DIJOFS_BUTTON1`) or `DIEB_NOTRIGGER`, indicating that the effect is not triggered. Assigning a value to `dwTriggerRepeatInterval` will cause the effect to be repeated as long as the button is held down (if supported). The `cAxes` member should be set to the number of axes involved in the effect, and the `rgdwAxes` member should point to an array of `DWORDs`, with each member holding a `DIJOFS_*` constant for each individual axis. The `rglDirection` member is a pointer to an array of `longints` that define the direction of the effect. There must be `cAxes` members in this array, but for the polar coordinate system, the bearing is stored in the first member, and the last member (remember, we're only using two axes) is set to zero. The `lpEnvelope` member points to a `TDIEnvelope` structure, which we'll discuss in more detail shortly. Finally, the `cbTypeSpecificParams` should be set to the size, in bytes, of the data structure pointed to by the `lpvTypeSpecificParams` member. This data structure is dependent on the type of desired effect; we'll take a closer look at each effect type below, and examine their individual type-specific parameter data structures.



*Note:* If two or more effects are assigned to a button, the last one downloaded to the device will be the one triggered.

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Supported Effect Parameters** Unfortunately, some members of the TDIEffect structure may be unsupported by some drivers. For example, a specific driver may not allow an effect to be associated with a trigger, or it may not support repeated playback of triggered effects. However, the application can use the GetEffectInfo method of the IDirectInputDevice2 interface to determine which TDIEffect parameters are supported. The GetEffectInfo method is defined as:

```
function GetEffectInfo(  
  
var pdei: TDIEffectInfoA; // effect information structure  
  
const rguid: TGUID // effect GUID  
  
) : HRESULT; // returns a DirectX error code
```

The rguid parameter should be set to the GUID of the effect for which information is requested (these GUIDs are listed below when we get to creating the effect). The pdei parameter should be set to a variable of type TDIEffectInfoA. When this method returns, the TDIEffectInfoA structure will be initialized with all of the relevant information concerning the specified effect. The TDIEffectInfoA structure is defined as:

```
TDIEffectInfoA = packed record  
  
    dwSize : DWORD; // size of the structure in bytes  
  
    guid : TGUID; // the effect GUID  
  
    dwEffType : DWORD; // effect type flags  
  
    dwStaticParams : DWORD; // supported parameter flags  
  
    dwDynamicParams : DWORD; // dynamically modifiable parameters  
  
    tszName : array [0..MAX_PATH-1] of CHAR; // effect name
```

end ;

The dwStaticParams member contains a series of flags that indicate which members of the TDIEffect structure are supported. These flags are listed in Table 8-1. If you combine each flag with the dwStaticParams member using a Boolean AND operation, you can easily determine which TDIEffect members are supported by the driver. However, it is not always necessary to determine if a specific member is supported or not. If a member of this structure is not supported, it will simply be ignored by the driver. Therefore, you could go ahead and assign values to specific members and never worry about getting an error when you create the effect. On the other hand, you'll probably need to know if an effect can be assigned to a trigger button and if this effect can be repeated if that is integral to game play, and find an appropriate alternative if this functionality is not supported.

**Table 8-1: Supported TDIEffect member flags**

Flag	Supported TDIEffect Member
DIEP_DURATION	dwDuration
DIEP_SAMPLEPERIOD	dwSamplePeriod
DIEP_GAIN	dwGain
DIEP_TRIGGERBUTTON	dwTriggerButton
DIEP_TRIGGERREPEATINTERVAL	dwTriggerRepeatInterval
DIEP_AXES	cAxes, rgdwAxes
DIEP_DIRECTION	rglDirection
DIEP_ENVELOPE	lpEnvelope

The dwSize member, of course, must be set to `SizeOf(TDIEffectInfoA)` before calling the `GetEffectInfo` method. The `guid` member contains the GUID of the effect, and the `tszName` member contains the name of the effect type. The `dwEffType` member will contain a series of `DIEFT_*` flags indicating the effect type. You may want to check this member, as if it contains the `DIEFT_DEADBAND`, `DIEFT_FFATTACH`, `DIEFT_FFFADE`, `DIEFT_POSNEGCOEFFICIENTS`, `DIEFT_POSNEGSATURATION`, or `DIEFT_SATURATION` flags, the effect supports the `lDeadBand` parameter (covered below under condition effects), the attack and fade envelope parameters, positive and negative coefficient values (covered below under condition effects), positive and negative saturation values (also covered below under conditions), and saturation of condition effects, respectively.

Incidentally, `dwDynamicParams` also contains a series of `DIEP_*` flags that indicate which members of the TDIEffect structure can be changed dynamically while the effect is playing. This allows an application to modify an effect on the fly to reflect changes in the game world or to create more complex effects. We'll look at an example of modifying an effect on the fly later in this chapter. The following listing demonstrates how the `GetEffectInfo` method can be used to determine which members of the TDIEffect structure are supported.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 8-1: Determining supported effect parameters*

```
var  
  
    frmSupFxParams: TfrmSupFxParams;  
  
    FDirectInput: IDirectInput;  
  
    FDirectInputDevice: IDirectInputDevice2;  
  
    {these are used to create our on-the-fly device data format}  
  
    FDataFormat: TDIDataFormat;  
  
    FDataFormatObjects: array[0..255] of TDIOBJECTDATAFORMAT;  
  
    FDataFormatGUIDs: array[0..255] of TGUID;  
  
    {these just track present axes}  
  
    FAxisList: array[0..255] of Integer;  
  
    NumAxes: Integer;  
  
implementation
```

```

uses DXTools, frmDevicesU;

function EnumInputDevs(var lpddi: TDIDeviceInstance; pvRef: Pointer): Integer;
                                                                    stdcall;

var
    TempGuid: PGuid;

begin
    {we must save this guid in a separate list for use later when we
    create the device}
    New(TempGuid);
    TempGuid^ := lpddi.guidInstance;

    {display device information}
    TStringList(pvRef).AddObject(lpddi.tszInstanceName+'-'+lpddi.tszProductName,
                                TObject(TempGuid));

    {continue enumeration}
    Result := 1;
end;

procedure SetDataFormat;

function DIEnumDeviceObjectsProc(const peff: TDIDeviceObjectInstance;
    pvRef: Pointer): HRESULT; stdcall;

begin
    {if the offset is not beyond the indicated data size... (useful if
    we were to use the TDIJoyState data structure instead) }
    if peff.dwOfs < FDataFormat.dwDataSize then

```



```

begin

    {save the type GUID for this object}

    FDataFormatGUIDs[FDataFormat.dwNumObjs] := peff.guidType;

    {add the object information to our custom made
    data format}

    with FDataFormatObjects[FDataFormat.dwNumObjs] do
    begin

        pguid := @FDataFormatGUIDs[FDataFormat.dwNumObjs];

        dwOfs := peff.dwOfs;

        dwType := peff.dwType;

        dwFlags := 0;

    end;

    {if this object is an axis, store it in our axis list}

    if (peff.dwType and DIDFT_AXIS) > 0 then

    begin

        FAxisList[NumAxes] := peff.dwOfs;

        Inc(NumAxes);

    end;

    {increment the number of objects found}

    Inc(FDataFormat.dwNumObjs);

end;

    {continue enumeration}

    Result := DIENUM_CONTINUE;

end;

```

```
begin
```

```

    {enumerate all objects on this input device. this will build our custom data
    structure programmatically with the correct information for each object}

```

```

FDirectInputDevice.EnumObjects(@DIEnumDeviceObjectsProc, nil, DIDFT_ALL);

{set the data format for the input device}
DXCheck( FDirectInputDevice.SetDataFormat(FDataFormat) );
end;

procedure TfrmSupFxParams.FormActivate(Sender: TObject);
var
    FJoyList: TStringList;
    iCount: Integer;

    TempDevice: IDirectInputDevice;

    RangeProp: TDIPropRange;
    DataProp: TDIPropDWord;

    EffectInfo: TDIEffectInfoA;
begin
    {create the DirectInput object}
    DirectInputCreate(hInstance, DIRECTINPUT_VERSION, FDirectInput, nil);

    {create a list to hold information on each available game controller}
    FJoyList := TStringList.Create;

    {enumerate attached game controllers}

    /*** may want to cull out only those with force feedback here
    FDirectInput.EnumDevices(DIDEVTYPE_JOYSTICK, EnumInputDevs, FJoyList,
        DIEDFL_ATTACHEDONLY);

    {we'll add this information to a list box in a dialog box, and display this
    to the user so they can choose their desired input controller}

```

```

frmDevices := TfrmDevices.Create(Self);

for iCount := 0 to FJoyList.Count-1 do
    frmDevices.lbxJoyDevices.Items.Add(FJoyList[iCount]);

frmDevices.ShowModal;

{create the input device based on the input controller specified by the user}
DXCheck( FDirectInput.CreateDevice(PGUID(FJoyList.Objects[0])^,
                                     TempDevice, nil) );

{close this dialog box, and delete the list of GUIDs}
frmDevices.Release;

for iCount := 0 to FJoyList.Count-1 do
    Dispose(PGUID(FJoyList.Objects[0]));

FJoyList.Free;

{we need the new version of the input device interface, so retrieve it}
try
    DXCheck(TempDevice.QueryInterface(IDirectInputDevice2, FDirectInputDevice));
finally
    TempDevice := nil;
end;

{now we need to begin creating the data format structure and setting the
 data format, so we must initialize the overall data format structure}
with FDataFormat do
begin
    dwFlags := DIDF_ABSAXIS;

    dwDataSize := SizeOf(TDIJoyState2);

    dwSize := SizeOf(FDataFormat);

    dwObjSize := SizeOf(TDIObjectDataFormat);

    dwNumObjs := 0;

    rgodf := @FDataFormatObjects;

```

```

end;

{create and set a data format specifically made for this device}
SetDataFormat;

{we must now set the range for each axis, so initialize a range
property structure}
RangeProp.diph.dwSize := SizeOf(TDIPropRange);
RangeProp.diph.dwHeaderSize := SizeOf(TDIPropHeader);
RangeProp.diph.dwHow := DIPH_BYOFFSET;
RangeProp.lMin := -1000;
RangeProp.lMax := 1000;

{we must also set deadzones and saturations for each axis, so
initialize a DWord property structure}
DataProp.diph.dwSize := SizeOf(TDIPropDWord);
DataProp.diph.dwHeaderSize := SizeOf(TDIPropHeader);
DataProp.diph.dwHow := DIPH_BYOFFSET;

{for each axis...}
for iCount := 0 to NumAxes-1 do
begin
    {set its range...}
    RangeProp.diph.dwObj := FAxisList[iCount];
    DXCheck( FDirectInputDevice.SetProperty(DIPROP_RANGE, RangeProp.diph) );

    {its deadzone... (this value is in 100ths of a percent)}
    DataProp.dwData := 1000;
    DataProp.diph.dwObj := FAxisList[iCount];
    DXCheck( FDirectInputDevice.SetProperty(DIPROP_DEADZONE, DataProp.diph) );

```

```

    {and its saturation (this value is also in 100ths of a percent)}

    DataProp.dwData := 9000;

    DXCheck( FDirectInputDevice.SetProperty(DIPROP_SATURATION,DataProp.diph));
end;

{set the cooperative level.}
DXCheck( FDirectInputDevice.SetCooperativeLevel(Handle, DISCL_EXCLUSIVE or
                                                DISCL_FOREGROUND) );

{finally, acquire the device so we can start retrieving information}
DXCheck( FDirectInputDevice.Acquire );

{initialize the effect information structure}
FillChar(EffectInfo, SizeOf(TDIEffectInfoA), 0);
EffectInfo.dwSize := SizeOf(TDIEffectInfoA);

{retrieve effect information for constant forces}
FDirectInputDevice.GetEffectInfo(EffectInfo, GUID_ConstantForce);

{begin displaying supported TDIEffect parameters}
lbxEffectInfo.Items.Add(Effect Name: +EffectInfo.tszName);
lbxEffectInfo.Items.Add(');
lbxEffectInfo.Items.Add(Supported TDIEffect Members:');

if (EffectInfo.dwStaticParams and DIEP_DURATION) = DIEP_DURATION then
    lbxEffectInfo.Items.Add(    dwDuration');
if (EffectInfo.dwStaticParams and DIEP_SAMPLEPERIOD) =
    DIEP_SAMPLEPERIOD then
    lbxEffectInfo.Items.Add(    dwSamplePeriod');
if (EffectInfo.dwStaticParams and DIEP_GAIN) = DIEP_GAIN then
    lbxEffectInfo.Items.Add(    dwGain');
if (EffectInfo.dwStaticParams and DIEP_TRIGGERBUTTON) =

```

```
    DIEP_TRIGGERBUTTON then

    lbxEffectInfo.Items.Add(      dwTriggerButton' );

    if (EffectInfo.dwStaticParams and DIEP_TRIGGERREPEATINTERVAL) =
        DIEP_TRIGGERREPEATINTERVAL then

        lbxEffectInfo.Items.Add(      dwTriggerRepeatInterval' );

    if (EffectInfo.dwStaticParams and DIEP_AXES) = DIEP_AXES then

        lbxEffectInfo.Items.Add(      cAxes' );

    if (EffectInfo.dwStaticParams and DIEP_DIRECTION) = DIEP_DIRECTION then

        lbxEffectInfo.Items.Add(      rglDirection' );

    if (EffectInfo.dwStaticParams and DIEP_ENVELOPE) = DIEP_ENVELOPE then

        lbxEffectInfo.Items.Add(      lpEnvelope' );

end;
```



**Figure 8-3:** Microsoft SideWinder Force Feedback Pro supported parameters

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Conditions

As we've discussed, conditions are force feedback effects that respond to the position or motion of an axis. Condition effects are divided into four subtypes: spring, friction, damper, and inertia, all of which produce similar but subtly different tactile feedback sensations.

All condition effects use a TDICCondition structure for their type-specific parameters. The TDICCondition structure is defined as:

```
TDICCondition = packed record
```

```
    lOffset : longint;           // effect offset

    lPositiveCoefficient : longint; // positive offset coefficient

    lNegativeCoefficient : longint; // negative offset coefficient

    dwPositiveSaturation : DWORD; // positive offset max force output

    dwNegativeSaturation : DWORD; // negative offset max force output

    lDeadBand : longint;       // inactive region
```

```
end;
```

Each condition subtype makes use of specific members of this structure, as we'll examine below. Unlike other effects, conditions can accept an array of TDICCondition structures in the lpvTypeSpecificParams member of the TDIEffect structure. Each TDICCondition structure defines condition parameters for each axis, with each TDICCondition structure in the array matching up to the axis specified in the rgdwAxes member at the same index. When using an array of TDICCondition structures in this manner, the cbTypeSpecificParams member of the TDIEffect structure must be set to `SizeOf(TDICCondition)` multiplied by the number of members in the

array. We'll see an example of this later in the chapter.

**Spring** A spring condition causes the device to exert a force on an axis when it is moved away from a central location. This central location is defined by the `IOffset` member. A value of 0 puts this point at the center of the axis. The `IOffset` member can be set to values ranging from 10,000 to 10,000 (or you can use the `DI_FFMINOMINALMAX` constant again), with negative values to the left and positive values to the right, relative to the axis.

The `IPositiveCoefficient` indicates the strength of the force exerted on the axis when it is moved to the farthest position from the defined central location. This force is increased from 0 to `IPositiveCoefficient` as the axis is moved, giving a sensation of increased resistance as the axis is moved farther and farther away from the neutral point. Again, the `IPositiveCoefficient` can be set to values in the range 10,000 to 10,000, with positive values pushing the axis toward the central location and negative values pulling the axis away from the central location. If the driver supports negative coefficients, the `INegativeCoefficient` member can be set to values within this range, which can be used to indicate a different amount of force to be applied when the axis is moved to the negative side of the central location. If negative coefficients are not supported, the positive coefficient is used.

The `IDeadBand` member defines an area around the central location in which the axis can move before any amount of force is applied. The saturation values define a similar area at the farthest distance of the axis, with the negative saturation value indicating an area on the negative side of the central location. As with coefficients, if negative saturation values are not supported, the positive saturation value is used. `dwPositiveSaturation`, `dwNegativeSaturation`, and `IDeadBand` members can be set in the range of 0 to 10,000, indicating a percentage of the range of travel in hundredths of a percent.



**Note:** The `IDeadBand` member may not be supported by some drivers.

---

**Friction** The friction condition causes the device to exert a constant force on an axis as the axis is moved, regardless of its position. The `IPositiveCoefficient` member indicates the strength of this force, with values ranging from 10,000 to 10,000. Positive values indicate a push on the axis in the opposite direction of its movement, giving a resistant sensation, and negative values indicate a pull on the axis in the direction of its movement, giving an accelerating sensation. The `IOffset`, `INegativeCoefficient`, `dwPositiveSaturation`, `dwNegativeSaturation`, and `IDeadBand` members are ignored.

**Damper** The damper condition causes the device to exert a force on an axis in proportion to its velocity of movement. In other words, the faster the axis is moving, the more force is exerted. The `IPositiveCoefficient` member indicates the maximum force applied, with values in the range of 10,000 to 10,000. Like friction, positive values indicate a push on the axis in the opposite direction of its movement, and negative values indicate a pull on the axis in the direction of its movement.

The `IOffset` member is used to define a central point very similar to the spring condition. However, this central point is in relation to the axis' velocity, with a value of 0 indicating no velocity. A value other than 0 will cause a force to be applied to the axis even when it is not moving. The saturation and deadband parameters act in a similar manner in relation to velocity, and can be used to indicate a minimum and maximum velocity range in which force would be exerted on the axis.

**Inertia** The inertia condition is very similar to the damper condition, except that force is exerted in proportion to the acceleration of an axis. In other words, no force will be applied if the axis is moving at a steady velocity. Like the damper condition, the `IPositiveCoefficient` member is used to indicate the maximum force, with positive values causing resistance and negative values causing acceleration. Also like the damper condition, the `IOffset` can be used to define a central point relative to the acceleration of the axis, with non-zero values generating acceleration. Saturation and deadband work in a similar manner.

## Constant Forces



A constant force is simply a push on an axis at a given magnitude. It is the simplest of all force feedback effects, and as such, it has a very simple type-specific data structure. Constant forces use a TDIConstantForce structure for their type-specific parameters, which is defined as:

```
TDIConstantForce = packed record  
  
    lMagnitude : longint;           // the strength of the force  
  
end;
```

The only member of this structure, lMagnitude, indicates the strength of the force in the range 10,000 to 10,000. Positive values push the axis in the direction indicated by the rglDirection member of the TDIEffect structure; negative values push the axis in the opposite direction.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Ramp Forces**

A ramp force creates a steadily increasing or decreasing push on an axis over the duration specified by the `dwDuration` member of the `TDIEffect` structure. Ramp forces use a `TDIRampForce` structure for their type-specific parameters, which is defined as:

`TDIRampForce = packed record`

```
    lStart : longint;           // the starting force magnitude
    lEnd   : longint;          // the ending force magnitude
end;
```

As with constant forces, the `lStart` and `lEnd` members indicate the strength of the force in the range 10,000 to 10,000, with positive values pushing the axis in the indicated direction and negative values pushing in the opposite direction.



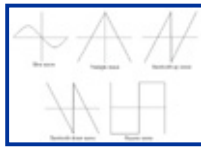
---

*Note:* The `dwDuration` member of the `TDIEffect` structure cannot be set to `INFINITE` for ramp forces.

---

**Periodic Forces**

A periodic effect causes a force to be applied to an axis in an oscillating manner. This is also known as a wave effect, as the magnitude and direction of a force changes over time. Similar to conditions, periodic forces are divided into subtypes: sine, triangle, sawtooth up, sawtooth down, and square. These produce alternating forces on an axis in a manner suggested by their name, as indicated by the following illustration.



**Figure 8-4:** Periodic forces

All periodic effects use a TDIPeriodic structure for their type-specific parameters. The TDIPeriodic structure is defined as:

```
TDIPeriodic = packed record
```

```
    dwMagnitude : DWORD;           //the absolute force magnitude
    lOffset : longint;             //neutral wave position
    dwPhase : DWORD;               //force start position
    dwPeriod : DWORD;              // wave frequency
```

```
end;
```

The dwMagnitude member indicates the absolute magnitude of the force to be exerted on the axis, in a range of 0 to 10,000. While the force is playing, this magnitude will steadily cycle between this value through the negative of this value and back again, creating a push on the axis in alternating directions. The manner in which this cycling takes effect is determined by the force type, as indicated in Figure 8-4. For example, a sine force would cycle the magnitude, and thus direction, of the force in a steady, smooth manner, where a square force would push the axis in alternating directions at the greatest magnitude in a very jarring, immediate manner.

The lOffset member indicates the center of the waveform in relation to the amount of force exerted, and will raise or lower the absolute magnitude of the force appropriately. For example, a value of zero indicates that the waveform will oscillate around zero magnitude. A value of 1,000, however, indicates that the waveform will oscillate around 1,000 magnitude, modifying the positive and negative magnitudes of the waveform accordingly. This is visually illustrated in the figure below.



**Figure 8-5:** Modifying the waveform offset



**Note:** The sum of dwMagnitude and the absolute value of lOffset cannot exceed 10,000 or DI\_FFNO MINALMAX.

The dwPhase member indicates the point at which the waveform should start, and is measured in hundredths of degrees. For example, setting this member to 9,000 for a sine periodic effect shifts the starting point by 90 degrees, and creates a cosine effect. This member is not always supported by all drivers. Finally, the dwPeriod member indicates the frequency of the waveform, measured in microseconds. For most drivers, this value cannot be set above one second, and it will have a minimum practical value based on how fast the motors in the device can reverse their direction.

## Device-Specific Effects

Many force feedback manufacturers provide several effects built in to the device driver. Many of these may be standard effect types, such as ramp forces, constant forces, or conditions, but some may be custom forces. Additionally, some of these forces may have adjustable type-specific parameters. If they have adjustable parameters

and are not a custom force, you can use the type-specific parameters structure appropriate for the force type to modify the force during playback. Otherwise, you'll need to get the type-specific parameters structure from the driver documentation if you wish to change an adjustable device-specific custom force on the fly.

Of course, device-specific forces are going to change from device to device, but non-adjustable device-specific forces are perhaps the easiest to use as the application needs to know nothing about them. In order to use a device-specific effect, we must first retrieve a list of available effects; this is accomplished by using the EnumEffects method of the IDirectInputDevice2 interface. The EnumEffects method is defined as:

```
function EnumEffects(  
  
    lpCallback: TDIEnumEffectsCallbackA;        // a pointer to the callback function  
  
    pvRef: pointer;                             // application-specific data  
  
    dwEffType: DWORD                            // flag affecting enumeration  
  
    ) : HRESULT;                               // returns a DirectX error code
```

The pvRef parameter is simply a pointer to application-specific data, which will be passed to the callback function. The dwEffType parameter contains a DIEFT\_\* flag which determines what type of effects are to be enumerated. In our case, we'll use the DIEFT\_HARDWARE flag to indicate that we want only hardware-specific effects. The lpCallback parameter is a pointer to the callback function. The callback function itself must be defined as:

```
function TDIEnumEffectsCallbackA(  
  
    pdei: TDIEffectInfoA;                       // a TDIEffectInfoA structure  
  
    pvRef: pointer                             // a pointer to application-specific data  
  
    ) : BOOL; stdcall;                         // returns TRUE or FALSE
```

The pdei parameter contains an initialized TDIEffectInfoA structure that has all of the pertinent information about the enumerated effect. In particular, the GUID member of this structure contains the GUID for the enumerated effect, which will need to be recorded for subsequent use when the effect is created. The pvRef parameter simply contains a pointer to the application-defined data specified when the EnumEffects method was called. This callback function should return TRUE to continue enumeration or FALSE to cancel it. Listing 8-2 demonstrates how to use the EnumEffects method to enumerate all device-specific effects that do not have any adjustable type-specific parameters.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 8-2: Enumerating device-specific effects*

```
var
```

```
FDirectInput: IDirectInput;
```

```
FDirectInputDevice: IDirectInputDevice2;
```

```
.
```

```
.
```

```
.
```

```
implementation
```

```
uses DXTools, frmDevicesU;
```

```
.
```

```
.
```

```
.
```

```
function EnumEffects(var pdei: TDIEffectInfoA; pvRef: pointer) : Integer; stdcall;
begin
    {filter out any effects that have adjustable type-specific parameters}
    if (pdei.dwStaticParams and DIEP_TYPESPECIFICPARAMS) =
        DIEP_TYPESPECIFICPARAMS then
    begin
        Result := 1;
        Exit;
    end;

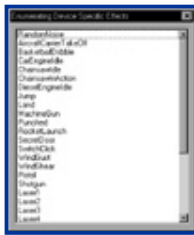
    {add the name of the effect to the list box}
    frmJoyTestBed.lbxDevEffects.Items.Add(pdei.tszName);

    {continue enumeration}
    Result := 1;
end;

procedure TfrmEnumDevEffects.FormActivate(Sender: TObject);
begin
    .
    .
    .

    {enumerate all hardware-specific effects}
    FDirectInputDevice.EnumEffects(EnumEffects, nil, DIEFT_HARDWARE);
end;
```

---



**Figure 8-6:** Microsoft SideWinder Force Feedback Pro device effects

Like all effects, you'll use a TDIEffect structure when creating device-specific effects. However, some effects may have default values that will be used by the driver whenever certain specific members of the TDIEffect structure are set to a particular value. For example, the Microsoft SideWinder Force Feedback Pro joystick will use default values for the sample period and duration if the dwSamplePeriod and dwDuration members of the TDIEffect structure are set to 1000. Other devices will probably use different values, and may require delving into the driver documentation.

## Custom Forces

If none of the standard effect types provide the tactile feedback sensation desired, you can attempt to create it using a custom force. A custom force is little more than an array of magnitudes. It replays this array of magnitudes as constant forces; each element is played in sequence at a constant interval and in the same direction (which can be reversed by negative magnitudes). Custom forces use the TDICustomForce structure for their type-specific parameters, which is defined as:

```
TDICustomForce = packed record

    cChannels : DWORD;           // number of channels

    dwSamplePeriod : DWORD;      // sample period

    cSamples : DWORD;           // number of elements in sample array

    rgfForceData : ^longint;     // pointer to the array of samples

end;
```

The cChannels member indicates how many axes are involved in the force, and results in each sample in the samples array being rotated amongst the axes (i.e., in the case of two axes and a cChannels setting of 2, element 1 applies to the first axis, element 2 to the second, element 3 to the first, etc.). This functionality may not be supported by all drivers.

The dwSamplePeriod member is very similar to the dwSamplePeriod member of the TDIEffect structure. The dwSamplePeriod member of the TDICustomForce structure indicates how long each element in the samples array will play, whereas the dwSamplePeriod in the TDIEffect structure indicates how long DirectInput waits before recalculating the force output by the device. Most devices do not support the dwSamplePeriod member of the TDICustomForce structure, and instead use the dwSamplePeriod member of the TDIEffect structure to determine how long to play each sample. In most cases, both members should be set to the same value to avoid any confusing behavior.

The cSamples member should be set to the number of elements in the array, which is pointed to by the rgfForceData member. This array contains a series of longints that define the magnitudes in the range 10,000 to 10,000.



---

**Note:** Applying an envelope to a custom force may not be supported by all drivers.

---

## Creating Effects

As usual, a specific series of steps must be taken by the application in order to create a force feedback effect and prepare it for playback. Some of these steps may be optional depending on certain properties of the effect. In general, a force feedback effect is initialized and used by performing the following steps:

- Create and acquire the DirectInput device
- Query for force feedback support
- Initialize a TDIEffect structure and the appropriate type-specific parameters structure
- Create the effect object
- Download the effect (sometimes optional)
- Start the effect (sometimes optional)
- Stop the effect (sometimes optional)

## Create and Acquire the DirectInput Device

Force feedback is a subsystem of DirectInput. As such, an initialized IDirectInputDevice2 interface must be created before any force feedback functionality can be utilized. Please refer to Chapter 7, Input Techniques, for a detailed explanation of creating an IDirectInputDevice2 object. Specifically, we'll be using the code from that chapter for communicating with the joystick. Other game controllers will function similarly.

## Query for Force Feedback Support

While you could certainly require the user to have a force feedback joystick installed before using your application, it would be much wiser to allow input from any game controller attached to the system, and use force feedback only when it was available on the device. In order to determine if a device supports force feedback, we must call the GetCapabilities method of the IDirectInputDevice2 interface. This method is described in the Input Techniques chapter, but for the sake of completeness, its definition is reprinted here:

```
function GetCapabilities(
    var lpDIDevCaps: TDIDevCaps           // a TDIDevCaps structure
): HRESULT;                             // returns a DirectX error code
```

The only parameter to this method is a pointer to a TDIDevCaps structure. Again, for the sake of completeness, it is defined as:

```
TDIDevCaps = packed record

    dwSize: DWORD;                       // the size of the structure

    dwFlags: DWORD;                      // device flags

    dwDevType: DWORD;                   // device type identifier

    dwAxes: DWORD;                      // number of axes

    dwButtons: DWORD;                   // number of buttons

    dwPOVs: DWORD;                      // number of POVs

    dwFFSamplePeriod: DWORD;            // force feedback sample period

    dwFFMinTimeResolution: DWORD;      // force feedback time resolution

    dwFirmwareRevision: DWORD;         // firmware version

    dwHardwareRevision: DWORD;         // hardware version

    dwFFDriverVersion: DWORD;          // device driver version

end;
```

The only member we're concerned with at this point is the dwFlags member. If this member contains the flag



DIDC\_FORCEFEEDBACK, then we know that this device supports force feedback functionality. Incidentally, we can check for other DIDC\_\* flags to determine if the device supports such force feedback parameters as attack and fade (which refer to envelopes), deadband, negative coefficients and saturations, etc.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**

[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Initialize a TDIEffect Structure and the Type-Specific Parameters Structure**

Now that we have a device and we know it supports force feedback, it is time to define the type of effect we wish to create. This is accomplished by initializing a TDIEffect structure and a type-specific parameters structure appropriate for the desired effect.

The first thing we should do is initialize the type-specific parameters structure. For example, if we were creating a periodic effect, we would need a TDIPeriodic structure, initialized in the following manner:

```
PeriodicInfo.dwMagnitude := DI_FFNOMINALMAX;
```

```
PeriodicInfo.lOffset := 0;
```

```
PeriodicInfo.dwPhase := 0;
```

```
PeriodicInfo.dwPeriod := DI_SECONDS div 20;
```

This indicates that we want a periodic effect that uses the maximum magnitude available on the device with no offset or phase shift and a relatively quick frequency. This will result in a vibrating effect, which we could potentially use to portray the feel of a powerful weapon such as a chainsaw or machine gun.



**Tip:** Remember that the overall strength of any effect will be scaled by the gain set for the device.

Next, we need to define the axes on which this effect will be played. We're using the joystick, so we'll indicate that the effect should use both the x- and y-axis. We'll use a small array of DWORDs, indicating the axes themselves using DIJOFS\_\* flags. We'll also indicate the direction of the effect. In this case, we'll be using the polar coordinate system, so we need a small array of longints (with as many elements as there are in the axis array). We simply need to indicate only the compass bearing of the desired direction in the first element, and set the last

element to zero. This is accomplished like so:

```
AxisArray[0] := DIJOFS_X;  
AxisArray[1] := DIJOFS_Y;  
  
DirArray[0] := 180*DI_DEGREES;  
DirArray[1] := 0;
```

With this completed, we need to initialize the TDIEffect structure. For example:

```
EffectInfo.dwSize := SizeOf(TDIEffect);  
EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;  
EffectInfo.dwDuration := INFINITE;  
EffectInfo.dwSamplePeriod := 0;  
EffectInfo.dwGain := DI_FFNOINALMAX;  
EffectInfo.dwTriggerButton := DIJOFS_BUTTON0;  
EffectInfo.dwTriggerRepeatInterval := INFINITE;  
EffectInfo.cAxes := 2;  
EffectInfo.rgdwAxes := @AxisArray;  
EffectInfo.rglDirection := @DirArray;  
EffectInfo.lpEnvelope := nil;  
EffectInfo.cbTypeSpecificParams := SizeOf(TDIPeriodic);  
EffectInfo.lpvTypeSpecificParams := @PeriodicInfo;
```

Here, we've indicated that we have an effect of infinite duration, and that we'll be using a polar coordinate system with device objects indicated by offsets. We've set the gain to maximum, so our effect will play with the strongest possible force available on the device (scaled by any gain set for the device itself). We've hooked the effect up to button 0 on the device, so it will automatically play when the user presses the button. A trigger repeat interval of infinite will really have no effect, as the effect will automatically continue to play until the button is released because of the infinite duration. We've also indicated the axes and direction, and we set up the cbTypeSpecificParams and lpvTypeSpecificParams members to indicate the size of and point to our type-specific parameters structure, respectively.

### Create the Effect Object

Now that the effect is completely defined, it needs to be created. The CreateEffect method of the IDirectInputDevice2 interface is used to generate an initialized IDirectInputEffect object. The CreateEffect method is defined as:

```

function CreateEffect(

const rguid: TGUID;           // the effect type guid

lpeff: PDIEffect;           // pointer to a TDIEffect structure

var ppdeff: IDirectInputEffect; // returns an IDirectInputEffect object

punkOuter: IUnknown         // COM aggregation

) : HRESULT;                // returns a DirectX error code

```

The first parameter takes a GUID for the type of effect to be created. This can be either a predefined GUID for standard effects (listed below) or a GUID obtained from calling the EnumEffects method. The second parameter takes a pointer to the TDIEffect structure we initialized above. The third parameter takes a pointer to the IDirectInputEffect object, which will be instantiated when the method returns. The final parameter is used for COM aggregation and should be set to nil.

**Table 8-2: Predefined effect GUIDs**

GUID	Description
GUID_ConstantForce	Constant forces
GUID_RampForce	Ramp forces
GUID_Square	Square wave periodic forces
GUID_Sine	Sine wave periodic forces
GUID_Triangle	Triangle wave periodic forces
GUID_SawtoothUp	Sawtooth up wave periodic forces
GUID_SawtoothDown	Sawtooth down wave periodic forces
GUID_Spring	Spring conditions
GUID_Damper	Damper conditions
GUID_Inertia	Inertia conditions
GUID_Friction	Friction conditions
GUID_CustomForce	Custom forces

## Download the Effect

Once an effect is created, it must be downloaded into the device before it can be played. Downloading an effect is accomplished by calling the Download method of the IDirectInputEffect interface, which is defined as:

```

function Download : HRESULT;           // returns a DirectX error code

```

Now, it is not always necessary to download an effect. If the input device is in an acquired state when the effect is created, it is downloaded automatically. It is also downloaded automatically whenever the Start method is called (see below). For some effects, such as a triggered effect that was created before the device was acquired, the Download method must be used (as the Start method will not be called for triggered effects). Also, there is a little bit of a lag when an effect is initially downloaded, so explicitly downloading an effect before gameplay starts may reduce any detectable pause the first time an effect is to be played.

As discussed earlier, a DIERR\_DEVICEFULL error may be returned when using a force feedback effect. This indicates that the device has reached its maximum number of downloadable force feedback effects, and room must be created before a new effect can be used. An existing effect can be unloaded from a device by calling the Unload method of the IDirectInputEffect interface, which is defined as:

```

function Unload : HRESULT; // returns a DirectX error code

```

It is possible to use the DIPROP\_FFLOAD flag with the GetProperty method of the IDirectInputDevice2 interface to retrieve the memory load for the device. You'll need to use a TDIPropDWORD structure, and upon returning, the dwData member of this structure will contain a value in the range of 0 to 100, indicating the percentage of device memory in use. Checking this value before downloading a new effect may give you some indication as to whether or not the device will be able to hold the new effect without unloading an existing one.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ KEYWORD SEARCH

- ▶ Search Tips
- ▶ [Advanced Search](#)

▶ PUBLICATION LOOKUP

▶ JUMP TO TOPIC



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

7.0

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)



*Note:* If a device is unacquired (as would happen when another application gained focus), all effects are automatically unloaded. Effects should be downloaded again when the device is reacquired, especially triggered effects for which the Start method will never be called.

**Start the Effect**

Once the effect is downloaded, it can be played. Triggered effects are started whenever the specified button is pressed, but non-triggered effects must be explicitly told to begin playing. This is accomplished by calling the Start method of the IDirectInputEffect interface. Note that the input device must be acquired at the exclusive cooperative level before the Start method will succeed. The Start method is defined as:

```
function Start(  
  
    dwIterations: DWORD;           // number of times to repeat the effect  
  
    dwFlags: DWORD                 // playback flags  
  
    ) : HRESULT;
```

The dwIterations parameter indicates how many times to play the effect. This parameter can be set to a positive integer value or to INFINITE, indicating it will continue playing until explicitly stopped. Of course, for effects that have a defined duration of INFINITE, the dwIterations parameter is effectively useless, and some devices do not even support multiple iterations.

The dwFlags parameter indicates how the effect should be played on the device. By default, an effect will be mixed with other effects currently playing on the device when it is started. Setting this flag to DIES\_SOLO will stop all other currently playing effects before playing the new effect. The DIES\_NODOWNLOAD flag can also be used to indicate that the effect should not automatically be downloaded. This parameter can be set to zero or any combination of these two flags.



---

*Note:* If the effect is already playing when Start is called, it will stop and be restarted from the beginning.

---

## Stop the Effect

Effects with a finite duration or those associated with a button will automatically stop when the duration has been reached or the button is released. However, some effects, such as those explicitly started with an infinite duration, will continue to play until the device is unacquired. Sometimes, it may be necessary to stop an effect, and this is where the Stop method of the IDirectInputEffect interface comes in. It is defined as:

```
function Stop : HRESULT; // returns a DirectX error code
```

## Case Study

Creating force feedback effects, while somewhat confusing, isn't quite as complicated as other DirectX subsystems. The following listing is a case study example which demonstrates how to create each type of effect. While not particularly robust, this example should serve to demonstrate how the above steps are followed for each effect type.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 8-3: Creating force feedback effects*

```
var  
  
.  
.  
.  
  
{these hold all of the different effects}  
  
PeriodicEffect,  
  
RampEffect,  
  
ConstantEffect,  
  
ConditionEffect,  
  
DeviceEffect,  
  
CustomEffect: IDirectInputEffect;  
  
.  
.  
.
```



```

{global effect info structures}

AxisSize: array[0..1] of DWORD;

DirArray: array[0..1] of longint;

EffectInfo: TDIEffect;

implementation

uses DXTools, frmDevicesU;

.
.
.

function EnumEffects(var pdei: TDIEffectInfoA; pvRef: pointer) : Integer; stdcall;
var
    TempGuid: PGuid;
begin
    {filter out any effects that have adjustable type-specific parameters}
    if (pdei.dwStaticParams and DIEP_TYPESPECIFICPARAMS) =
        DIEP_TYPESPECIFICPARAMS then
    begin
        Result := 1;
        Exit;
    end;

    {create a new guid and save the guid info}
    New(TempGuid);

    TempGuid^ := pdei.guid;

    {add the name of the effect and its guid to the list box}

```

```
frmEffectTest.lbxDevEffects.Items.AddObject(pdei.tszName, TObject(TempGuid));

{continue enumeration}

Result := 1;

end;

.
.
.

procedure TfrmEffectTest.FormActivate(Sender: TObject);

begin

.
.
.

{set the cooperative level.}
DXCheck( FDirectInputDevice.SetCooperativeLevel(Handle, DISCL_EXCLUSIVE
                                                or DISCL_FOREGROUND) );

{finally, acquire the device so we can start retrieving information}
DXCheck( FDirectInputDevice.Acquire );

{initialize the global axis array. we'll set it up to affect the
 X and Y axis of a joystick}
AxisArray[0] := DIJOFS_X;
AxisArray[1] := DIJOFS_Y;

{indicate the direction of the effects. we'll be using polar coordinates
 so we'll indicate a 180 degree direction}
DirArray[0] := 180*DI_DEGREES;
DirArray[1] := 0;
```

```

    {enumerate all hardware-specific effects}

    FDirectInputDevice.EnumEffects(EnumEffects, nil, DIEFT_HARDWARE);

end;

procedure TfrmEffectTest.btnPeriodicClick(Sender: TObject);

var
    PeriodicInfo: TDIPeriodic;

begin
    {turn off the device effects list box and reset the force feedback device}

    lbxDevEffects.Visible := FALSE;

    FDirectInputDevice.SendForceFeedbackCommand(DISFFC_RESET);

    {set up our periodic effect parameters}

    PeriodicInfo.dwMagnitude := DI_FFNOMINALMAX;

    PeriodicInfo.lOffset := 0;

    PeriodicInfo.dwPhase := 0;

    PeriodicInfo.dwPeriod := DI_SECONDS div 20;

    {set up general parameters for the periodic effect}

    EffectInfo.dwSize := SizeOf(TDIEffect);

    EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

    EffectInfo.dwDuration := INFINITE;           // repeat indefinitely

    EffectInfo.dwSamplePeriod := 0;

    EffectInfo.dwGain := DI_FFNOMINALMAX;

    EffectInfo.dwTriggerButton := DIJOFS_BUTTON0; // tie it to button 0

    EffectInfo.dwTriggerRepeatInterval := INFINITE;

    EffectInfo.cAxes := 2;

    EffectInfo.rgdwAxes := @AxisArray;

    EffectInfo.rglDirection := @DirArray;

    EffectInfo.lpEnvelope := nil;

    EffectInfo.cbTypeSpecificParams := SizeOf(TDIPeriodic);

```

```

EffectInfo.lpvTypeSpecificParams := @PeriodicInfo;

{create the periodic effect}

FDirectInputDevice.CreateEffect(GUID_sine, @EffectInfo, PeriodicEffect, nil);

end;

procedure TfrmEffectTest.btnRampClick(Sender: TObject);

var
    RampInfo: TDIRampForce;

begin
    {turn off the device effects list box and reset the force feedback device}

    lbxDevEffects.Visible := FALSE;

    FDirectInputDevice.SendForceFeedbackCommand(DISFFC_RESET);

    {set up our ramp effect parameters}

    RampInfo.lStart := -DI_FF NOMINALMAX;

    RampInfo.lEnd := DI_FF NOMINALMAX;

    {set up general parameters for the ramp effect}

    EffectInfo.dwSize := SizeOf(TDIEffect);

    EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

    EffectInfo.dwDuration := DI_SECONDS div 2;           // half second duration

    EffectInfo.dwSamplePeriod := 0;

    EffectInfo.dwGain := DI_FF NOMINALMAX;

    EffectInfo.dwTriggerButton := DIJOFS_BUTTON0;       // tie it to button 0

    EffectInfo.dwTriggerRepeatInterval := 0;

    EffectInfo.cAxes := 2;

    EffectInfo.rgdwAxes := @AxisArray;

    EffectInfo.rglDirection := @DirArray;

    EffectInfo.lpvEnvelope := nil;

    EffectInfo.cbTypeSpecificParams := SizeOf(TDIRampForce);

    EffectInfo.lpvTypeSpecificParams := @RampInfo;

```

```

    {create the ramp effect}

    FDirectInputDevice.CreateEffect(GUID_RampForce, @EffectInfo, RampEffect, nil);

end;

procedure TfrmEffectTest.btnConstantClick(Sender: TObject);

var
    ConstantInfo: TDIConstantForce;

begin
    {turn off the device effects list box and reset the force feedback device}

    lbxDevEffects.Visible := FALSE;

    FDirectInputDevice.SendForceFeedbackCommand(DISFFC_RESET);

    {set up our constant effect parameters}

    ConstantInfo.lMagnitude := DI_FFNOMINALMAX;

    {set up general parameters for the constant effect}

    EffectInfo.dwSize := SizeOf(TDIEffect);

    EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

    EffectInfo.dwDuration := DI_SECONDS div 2;           // half second duration

    EffectInfo.dwSamplePeriod := 0;

    EffectInfo.dwGain := DI_FFNOMINALMAX;

    EffectInfo.dwTriggerButton := DIJOFS_BUTTON0;      // tie it to button 0

    EffectInfo.dwTriggerRepeatInterval := 0;

    EffectInfo.cAxes := 2;

    EffectInfo.rgdwAxes := @AxisArray;

    EffectInfo.rglDirection := @DirArray;

    EffectInfo.lpEnvelope := nil;

    EffectInfo.cbTypeSpecificParams := SizeOf(TDIConstantForce);

    EffectInfo.lpvTypeSpecificParams := @ConstantInfo;

    {create the constant effect}

    FDirectInputDevice.CreateEffect(GUID_ConstantForce, @EffectInfo,

```

```

ConstantEffect, nil);

end;

procedure TfrmEffectTest.btnConditionClick(Sender: TObject);

var
    ConditionInfo: array[0..1] of TDICCondition;
    DirArray: array[0..1] of longint;

begin
    {turn off the device effects list box and reset the force feedback device}
    lbxDevEffects.Visible := FALSE;
    FDirectInputDevice.SendForceFeedbackCommand(DISFFC_RESET);

    {for a condition, we don't really want a direction, so empty
    the direction array}
    DirArray[0] := 0;
    DirArray[1] := 0;

    {set up our condition effect parameters for the first axis}
    ConditionInfo[0].lOffset := 0;
    ConditionInfo[0].lPositiveCoefficient := DI_FFNOINALMAX;
    ConditionInfo[0].lNegativeCoefficient := DI_FFNOINALMAX;
    ConditionInfo[0].dwPositiveSaturation := 0;
    ConditionInfo[0].dwNegativeSaturation := 0;
    ConditionInfo[0].lDeadBand := 0;

    {set up our condition effect parameters for the second axis}
    ConditionInfo[1].lOffset := 0;
    ConditionInfo[1].lPositiveCoefficient := DI_FFNOINALMAX;
    ConditionInfo[1].lNegativeCoefficient := DI_FFNOINALMAX;
    ConditionInfo[1].dwPositiveSaturation := 0;
    ConditionInfo[1].dwNegativeSaturation := 0;
    ConditionInfo[1].lDeadBand := 0;

```

```

{set up general parameters for the condition effect}

EffectInfo.dwSize := SizeOf(TDIEffect);

EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

EffectInfo.dwDuration := INFINITE;           // infinite duration

EffectInfo.dwSamplePeriod := 0;

EffectInfo.dwGain := DI_FFNOMINALMAX;

EffectInfo.dwTriggerButton := DIEB_NOTRIGGER; // not tied to a button

EffectInfo.dwTriggerRepeatInterval := 0;

EffectInfo.cAxes := 2;

EffectInfo.rgdwAxes := @AxisArray;

EffectInfo.rglDirection := @DirArray;

EffectInfo.lpEnvelope := nil;

{notice that here we've got to indicate the size of the entire array}

EffectInfo.cbTypeSpecificParams := 2*SizeOf(TDICondition);

EffectInfo.lpvTypeSpecificParams := @ConditionInfo;

{create the condition effect}

FDirectInputDevice.CreateEffect(GUID_Friction, @EffectInfo, ConditionEffect,
                                nil);

{start the condition effect, killing all other effects currently playing}

ConditionEffect.Start(1, DIES_SOLO);

end;

procedure TfrmEffectTest.btnDeviceClick(Sender: TObject);
begin
    {simply make the device effects list box visible}

    lbxDevEffects.Visible := TRUE;

end;

procedure TfrmEffectTest.lbxDevEffectsClick(Sender: TObject);
begin

```

```

if lbxDevEffects.ItemIndex > -1 then

begin

    {reset the force feedback device}

    FDirectInputDevice.SendForceFeedbackCommand(DISFFC_RESET);

    {set up general parameters for the device-specific effect}

    EffectInfo.dwSize := SizeOf(TDIEffect);

    EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

    EffectInfo.dwDuration := DI_SECONDS*2;           // half second duration

    EffectInfo.dwSamplePeriod := 0;

    EffectInfo.dwGain := DI_FF NOMINALMAX;

    EffectInfo.dwTriggerButton := DIJOFS_BUTTON0;   // tied to button 0

    EffectInfo.dwTriggerRepeatInterval := 0;

    EffectInfo.cAxes := 2;

    EffectInfo.rgdwAxes := @AxisArray;

    EffectInfo.rglDirection := @DirArray;

    EffectInfo.lpEnvelope := nil;

    {we only play effects that do not have adjustable type-specific parameters,
     so we indicate such in the TDIEffect structure}

    EffectInfo.cbTypeSpecificParams := 0;

    EffectInfo.lpvTypeSpecificParams := nil;

    {create the device-specific effect}

    FDirectInputDevice.CreateEffect(PGuid(lbxDevEffects.Items.Objects
                                         [lbxDevEffects.ItemIndex])^, @EffectInfo, DeviceEffect, nil);

end;

end;

procedure TfrmEffectTest.btnCustomClick(Sender: TObject);

var

    CustomInfo: TDICustomForce;

    MagnitudeArray: array[0..3] of longint;

begin

```



```

{turn off the device effects list box and reset the force feedback device}

lboxDevEffects.Visible := FALSE;

FDirectInputDevice.SendForceFeedbackCommand(DISFFC_RESET);

{set up our array of magnitudes for the custom effect. this will create
 an increasing oscillation}

MagnitudeArray[0] := -DI_FFNOINALMAX div 4;
MagnitudeArray[1] := DI_FFNOINALMAX div 2;
MagnitudeArray[2] := 0-(DI_FFNOINALMAX div 4) * 3;
MagnitudeArray[3] := DI_FFNOINALMAX;

{set up our custom effect parameters}

CustomInfo.cChannels := 1;

CustomInfo.dwSamplePeriod := 100000;

CustomInfo.cSamples := 4;

CustomInfo.rglForceData := @MagnitudeArray;

{set up general parameters for the custom effect}

EffectInfo.dwSize := SizeOf(TDIEffect);

EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

EffectInfo.dwDuration := INFINITE; // infinite duration

EffectInfo.dwSamplePeriod := 100000; // sample periods the same

EffectInfo.dwGain := DI_FFNOINALMAX;

EffectInfo.dwTriggerButton := DIJOFS_BUTTON0; // tied to button 0

EffectInfo.dwTriggerRepeatInterval := 0;

EffectInfo.cAxes := 2;

EffectInfo.rgdwAxes := @AxisArray;

EffectInfo.rglDirection := @DirArray;

EffectInfo.lpEnvelope := nil;

EffectInfo.cbTypeSpecificParams := SizeOf(TDICustomForce);

EffectInfo.lpvTypeSpecificParams := @CustomInfo;

```

```
{create the custom effect}

dxcheck(FDirectInputDevice.CreateEffect(GUID_CustomForce, @EffectInfo,
CustomEffect, nil));

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**IT KNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Additional Effect Considerations

While this is not meant to be a complete reference to DirectX, there are several more force feedback specific methods that have not been discussed, but may be very useful to applications that incorporate force feedback functionality. In particular, applications may need to determine the status of an effect or the state of the overall force feedback system, send a command to the force feedback system, or modify effect parameters on the fly.

## Retrieving Effect Status

There may be times when the application needs to determine if an effect is currently playing or not. This is easily accomplished by calling the `GetEffectStatus` method of the `IDirectInputEffect` interface. The `GetEffectStatus` method is defined as:

```

function GetEffectStatus(

var pdwFlags : DWORD                // flags indicating the effect status

) : HRESULT;                        // returns a DirectX error code

```

The only parameter to this method is a variable that will be initialized to one or more flags. The effect is currently playing if the DIEGES\_PLAYING flag is present in the pdwFlags variable.

### Retrieving the Force Feedback System State

Another useful method of the IDirectInputDevice2 interface is the GetForceFeedbackState method. This method returns a series of flags which indicate the overall state of the force feedback device. The GetForceFeedbackState method is defined as:

```

function GetForceFeedbackState(

var pdwOut: DWORD                // flags indicating overall device state

) : HRESULT;                    // returns a DirectX error code

```

The pdwOut parameter is simply a variable that, upon return, will be initialized with the flags that indicate the overall force feedback system status. This can be a combination of one or more flags from the following table.

**Table 8-3: Force feedback state flags**

Flag	Description

DIGFFS_EMPTY	The device has no downloaded effects.
DIGFFS_STOPPED	No effects are playing.
DIGFFS_PAUSED	All active effects are paused.
DIGFFS_ACTUATORSON	Force feedback actuators are enabled.
DIGFFS_ACTUATORSOFF	Force feedback actuators are disabled.
DIGFFS_POWERON	Power to the force feedback device is available.
DIGFFS_POWEROFF	Power to the force feedback device is not available.
DIGFFS_SAFETYSWITCHON	The safety switch is on.
DIGFFS_SAFETYSWITCHOFF	The safety switch is off, and the device cannot operate.
DIGFFS_USERFFSWITCHON	The force feedback switch is on.
DIGFFS_USERFFSWITCHOFF	The force feedback switch is off, and force feedback effects cannot be played.
DIGFFS_DEVICELOST	The device has been unacquired and must be reacquired.

## Sending Commands to the Force Feedback System

There are times when it is necessary to send global commands to the force feedback system. For example, if the user pauses the game, it would be nice to pause all playing effects without having to enumerate them all and call Stop for each one, and then restart them when the user resumes play. This can be accomplished through the SendForceFeedbackCommand method of the IDirectInputDevice2 interface. The SendForceFeedbackCommand method is defined as:

```
function SendForceFeedbackCommand(
    dwFlags: DWORD // the command value
) : HRESULT; // returns a DirectX error code
```

The single parameter of this function takes one value from the following table.

<i>Table 8-4: Force feedback system commands</i>	
Command	Description

DISFFC_RESET	Resets the entire device. All effects are destroyed and must be re-created.
DISFFC_STOPALL	All actively playing effects are stopped.
DISFFC_PAUSE	All actively playing effects are paused. When restarted, they continue playing to the remainder of their duration. New effects cannot be started.
DISFFC_CONTINUE	Continues playing from a paused state.
DISFFC_SETACTUATORSON	Turns force feedback actuators on.
DISFFC_SETACTUATORSOFF	Turns force feedback actuators off. Effects can still be played but will be ignored by the device.

## Modifying an Effect at Run Time

In order to further customize an effect to reflect changes in the game world, an effect can be modified while it is playing, provided that this is supported by the driver. As we examined earlier, we can use the `GetEffectInfo` method to determine which parameters can be dynamically changed during effect playback. In practice, we can change any of the parameters at any time; the parameters listed in `dwDynamicParams` indicate those parameters that can be changed without restarting the effect. Changing a parameter that is not included in `dwDynamicParams` will cause the force feedback system to stop and then restart the effect.

To change an effect's parameters during playback, call the `SetParameters` method of the `IDirectInputEffect` interface. The `SetParameters` method is defined as:

```
function SetParameters(
    var peff: TDIEffect;           // a TDIEffect structure with new parameters
    dwFlags: DWORD                // control flags
) : HRESULT;                    // returns a DirectX error code
```

The first parameter contains a `TDIEffect` structure that contains the new values for the effect parameters to be changed. It is not necessary to initialize the entire structure, only those members containing the new effect parameters. The second parameter contains a series of flags that indicate which members of the `TDIEffect` structure contain the new information. These are the same flags as returned by the `GetEffectInfo` method, and are listed in Table 8-1. Some additional flags available with this method control the behavior of the force feedback system when the effect is modified. For example, calling the `SetParameters` method automatically downloads the effect. This can be disabled by including the `DIEP_NODOWNLOAD` flag in the second parameter. These additional flags are listed below.

**Table 8-5: Parameter flags**

Flag	Description
DIEP_NODOWNLOAD	Effect parameters are updated but not downloaded to the device.
DIEP_NORESTART	Suppresses restart of an effect. If the effect must be restarted, an error code of DIERR_EFFECTPLAYING is returned, and the effect continues to play with the original parameters.
DIEP_START	Starts playing the effect if it is not playing already.
DIEP_TYPESPECIFICPARAMS	Indicates that the lpvTypeSpecificParams and cbTypeSpecificParams members of the TDIEffect structure contain new data.

The following listing demonstrates changing an effect's direction at run time.

[Previous](#) | [Table of Contents](#) | [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)  
▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 8-4: Changing effect parameters at run time*

---

var

.

.

.

{tracks the direction in degrees}

EffectDir: Integer;

implementation

.

.

.

procedure TfrmJoyTestBed.FormActivate(Sender: TObject);



```

var

    .
    .
    .

    AxisArray: array[0..1] of DWORD;
    DirArray: array[0..1] of longint;
    EffectInfo: TDIEffect;
    PeriodicInfo: TDIPeriodic;

begin

    .
    .
    .

    {this effect plays on the X and Y axis}
    AxisArray[0] := DIJOFS_X;
    AxisArray[1] := DIJOFS_Y;

    {initialize the direction}
    EffectDir := 1;
    DirArray[0] := EffectDir*DI_DEGREES;
    DirArray[1] := 0;

    {initialize a periodic effect}
    PeriodicInfo.dwMagnitude := DI_FFMINOMINALMAX;
    PeriodicInfo.lOffset := 0;
    PeriodicInfo.dwPhase := 0;
    PeriodicInfo.dwPeriod := DI_SECONDS;;

    {initialize the global effect parameters}

```

```

EffectInfo.dwSize := SizeOf(TDIEffect);

EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;

EffectInfo.dwDuration := INFINITE;           // infinite duration

EffectInfo.dwSamplePeriod := 0;

EffectInfo.dwGain := DI_FFNOINALMAX;

EffectInfo.dwTriggerButton := DIEB_NOTRIGGER; // no trigger

EffectInfo.dwTriggerRepeatInterval := 0;

EffectInfo.cAxes := 2;

EffectInfo.rgdwAxes := @AxisArray;

EffectInfo.rglDirection := @DirArray;

EffectInfo.lpEnvelope := nil;

EffectInfo.cbTypeSpecificParams := SizeOf(TDIPeriodic);

EffectInfo.lpvTypeSpecificParams := @PeriodicInfo;

{create a sine wave periodic effect}

FDirectInputDevice.CreateEffect(GUID_sine, @EffectInfo, PeriodicEffect, nil);

{start the effect, suppressing all other effects on the device}

PeriodicEffect.Start(1, DIES_SOLO);

{enable the timer that changes the effect direction}

tmrDirChanger.Enabled := TRUE;

end;

procedure TfrmJoyTestBed.tmrDirChangerTimer(Sender: TObject);

var

    DirArray: array[0..1] of longint;

    EffectInfo: TDIEffect;

begin

    {increase the effect direction angle}

    Inc(EffectDir, 2);

```

```

{roll the angle over nicely}
if EffectDir > 360 then

    EffectDir := 1;

{display the new angle}
lblAngle.Caption := Effect Angle: +IntToStr(EffectDir);

{indicate the new direction}
DirArray[0] := EffectDir*DI_DEGREES;
DirArray[1] := 0;

{set up the effect parameters}
FillChar(EffectInfo, SizeOf(TDIEffect), 0);
EffectInfo.dwFlags := DIEFF_POLAR;
EffectInfo.dwSize := SizeOf(TDIEffect);
EffectInfo.rglDirection := @DirArray;
EffectInfo.cAxes := 2;

{dynamically change the parameters, indicating that the effect
should not restart. if the device doesn't support this, the change
will fail, but the effect will continue to play}
PeriodicEffect.SetParameters(EffectInfo, DIEP_DIRECTION or DIEP_NORESTART);
end;

```

---

## Envelopes

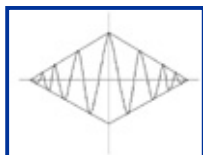
The basic effect types give the developer a wide range of possibilities for creating tactile feedback. Indeed, using custom forces or modifying a force at run time gives us even greater control over the “feel” of the feedback. We can further customize an effect by using an envelope.

An envelope is a method by which a force can be shaped, and is very similar in concept to envelopes in musical terms. Specifically, an envelope describes the starting magnitude for an effect and the amount of time it takes for the magnitude to go from this initial starting point to the magnitude defined by the type-specific parameters of the specific effect. This is known as the attack. Similarly, the envelope also describes the final magnitude of the effect

and the amount of time it takes for the magnitude to go from the effect-defined magnitude to this final level. This is known as the fade. These times take place during the entire duration of the effect, so the sum of both times should not exceed this duration. Any remaining time indicates the sustain, and determines how much time the effect is played at the defined magnitude. A visual illustration of an envelope is shown in Figure 8-7.



**envelope:** A series of values that shape a force at the beginning and end of playback.



**Figure 8-7:** An envelope shaping an effect



**Note:** Envelopes do not apply to condition effects.

---

Envelopes are defined by a `TDIEnvelope` structure. A pointer to this structure is placed in the `lpEnvelope` member of the `TDIEffect` structure when an effect is defined. The `TDIEnvelope` structure is defined as:

```
TDIEnvelope = packed record

    dwSize : DWORD;           // the size of the structure in bytes

    dwAttackLevel : DWORD;    // the initial magnitude

    dwAttackTime : DWORD;     // the attack duration

    dwFadeLevel : DWORD;     // the ending magnitude

    dwFadeTime : DWORD;      // the fade duration

end;
```

As usual with so many DirectX structures, the `dwSize` member must be set to `SizeOf(TDIEnvelope)`. The `dwAttackLevel` member defines the starting magnitude for the effect. Actually, this defines a percentage, in hundredths of a percent, of the maximum available force in the direction of the effect, and thus can be a value between 0 and 10,000. The `dwFadeLevel` member is very similar, except that it defines the percentage of maximum available force at the end of effect playback. Both `dwAttackTime` and `dwFadeTime` define the duration of the attack and fade in microseconds.



**Note:** Neither the attack nor fade levels need to be less than the sustain level. If they are defined greater than the sustain level, the effect will start at a greater level than the sustain and will fall down to it (or vice versa in the case of fade level).

---

The following example demonstrates applying an envelope to a sine wave periodic force.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 8-5: Using an envelope with periodic forces*

```
procedure TfrmJoyTestBed.FormActivate(Sender: TObject);
```

```
var
```

```
·
```

```
·
```

```
·
```

```
AxisArray: array[0..1] of DWORD;
```

```
DirArray: array[0..1] of longint;
```

```
EffectInfo: TDIEffect;
```

```
PeriodicInfo: TDIPeriodic;
```

```
Envelope: TDIEnvelope;
```

```
begin
```

```
·
```

```
·
```

```

.

{this effect plays on the X and Y axis}
AxisArray[0] := DIJOFS_X;
AxisArray[1] := DIJOFS_Y;

{initialize the direction}
DirArray[0] := 180*DI_DEGREES;
DirArray[1] := 0;

{initialize a periodic effect}
PeriodicInfo.dwMagnitude := DI_FFMINOMINALMAX;
PeriodicInfo.lOffset := 0;
PeriodicInfo.dwPhase := 0;
PeriodicInfo.dwPeriod := DI_SECONDS div 10;

{initialize the envelope. this will cause the effect to rise to the
sustain level and then immediately fall back to 0 magnitude, as the
entire duration of the effect is only 2 seconds, and both the
attack and fade time each last a full second}
Envelope.dwSize := SizeOf(TDIEnvelope);
Envelope.dwAttackLevel := 0;
Envelope.dwAttackTime := DI_SECONDS;
Envelope.dwFadeLevel := 0;
Envelope.dwFadeTime := DI_SECONDS;

{initialize the global effect parameters}
EffectInfo.dwSize := SizeOf(TDIEffect);
EffectInfo.dwFlags := DIEFF_OBJECTOFFSETS or DIEFF_POLAR;
EffectInfo.dwDuration := DI_SECONDS * 2;           // infinite duration
EffectInfo.dwSamplePeriod := 0;

```

```

EffectInfo.dwGain := DI_FFNOINALMAX;

EffectInfo.dwTriggerButton := DIJOFS_BUTTON0; // tied to button 0

EffectInfo.dwTriggerRepeatInterval := 0;

EffectInfo.cAxes := 2;

EffectInfo.rgdwAxes := @AxisArray;

EffectInfo.rglDirection := @DirArray;

EffectInfo.lpEnvelope := @Envelope;

EffectInfo.cbTypeSpecificParams := SizeOf(TDIPeriodic);

EffectInfo.lpvTypeSpecificParams := @PeriodicInfo;

{create a sine wave periodic effect}

FDirectInputDevice.CreateEffect(GUID_sine, @EffectInfo, PeriodicEffect, nil);

end;

```

---

## Summary

In this chapter, we discussed techniques for utilizing force feedback technology. We examined how to use both standard and custom effects, as well as how to modify effects on the fly. We also looked at shaping effects using envelopes. When using force feedback technology in your games, it is important to keep these points in mind:

- Force feedback is part of the DirectInput subsystem, and was designed to work not only with joysticks but with any type of input device.
- When discussing force feedback effects, it is necessary to understand several general concepts as they apply to the effect itself. Such concepts include kinetic forces, conditions, magnitude, coefficient, gain, duration, sample period, axes, direction, and envelope.
- A kinetic force is an effect that actively pushes on one or more axes. This push can be in a single direction or in alternating directions such as occurs with periodic effects.
- A condition is a resistance or force that is applied to one or more axes in response to their movement or position.
- Magnitude is a measurement of the strength of a kinetic force, relative to the total maximum force that can be output by the device. The magnitude of a force can be expressed as an integer value between 10,000 and 10,000, where positive values represent a force in one direction and negative values represent the same force in the opposite direction.
- The `DI_FFNOINALMAX` constant is set to 10,000 and represents the total maximum force that can be exerted by the device on any given axis. This constant allows the developer to define the magnitude of a force as a percentage of the total maximum force available on the device.
- Coefficient is a measurement of the strength of resistance to movement along an axis. Like magnitude, this value is expressed as an integer between 10,000 and 10,000.
- Duration defines how long the effect will play, measured in microseconds.
- The constant `DI_SECONDS` can be used to define the duration in terms of one-second intervals (or fractions thereof).
- Sample period defines the interval between actual changes in the strength of a force, measured in microseconds.
- Gain acts as a scaling factor for magnitudes and coefficients. It is expressed as a value between 0 and 10,000, indicating the percentage by which to scale the magnitude or coefficient.
- Axes determine what axis (or axes) of movement on the force feedback device will be affected by the force feedback effect.



- Direction determines the bearing of an effect along an axis (or axes) when its magnitude or coefficient values are positive. Negative values act in the opposite of the defined direction.
- An envelope is a set of values that modify the magnitude of a force at the beginning and end of playback, and can be applied to all forces except conditions.
- Force feedback effects are divided into six different categories: conditions, constant forces, periodic forces, ramp forces, device-specific effects, and custom forces.
- All force feedback effects are defined by a primary structure called TDIEffect. This structure is used when creating the effect as well as modifying the effect after it has been created.
- Each type of effect is defined by different parameters. These parameters are stored in a type-specific parameter data structure which differs from effect to effect.
- Some members of the TDIEffect structure may be unsupported by some drivers. However, the application can use the GetEffectInfo method of the IDirectInputDevice2 interface to determine which TDIEffect parameters are supported.
- Some effects must be downloaded to the device after they are created before they can be played. However, if the input device is in an acquired state when the effect is created, it is downloaded automatically. It is also downloaded automatically whenever the Start method is called. For some effects, such as a triggered effect that was created before the device was acquired, the Download method must be used.
- A DIERR\_DEVICEFULL error may be returned when using a force feedback effect. This indicates that the device has reached its maximum number of downloadable force feedback effects, and an existing effect must be unloaded from a device by calling the Unload method.
- Triggered effects are automatically started when the trigger is pushed, and stopped when the trigger is released. Otherwise, effects must be explicitly started using the Start method. For effects with infinite duration, the Stop method must be called to halt playback.
- Use the GetEffectStatus method to determine if an effect is currently playing.
- Use the GetForceFeedbackState method to determine the overall state of the force feedback device.
- Use the SendForceFeedbackCommand method to send global commands to the force feedback system.
- The SetParameters method allows a force feedback effect to be modified on the fly.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 9 Sound and Music

This chapter covers the following topics:

- Playing sound effects using multimedia functions and DirectSound
- Playing MIDI music using MCI commands
- Playing Red Book (CD) audio using MCI commands

Most people discuss games in terms of the technology used for the graphics engine. Hot first-person shooters are discussed in terms of colored lighting, interactive environments, the polygon count of characters, etc. Real-time strategy game discussions feature such terms as computer AI, realism of surface features (such as hills), tactical advantages of weapons, etc. Although rarely the topic of discussion, the importance of the third major piece of a game, sound and music, is not lost on game developers.

Regardless of the technological accomplishment represented by the game's graphics engine, no game is truly complete without some sort of sound effects to bring the environment to life, or a driving soundtrack to set the mood. Would Quake have been as good if you couldn't hear a heavy-breathing mutant around the corner? Would X-Wing have given you the same thrill without the musical fanfare following a completed mission? Indeed, music and sound effects round out a game and add the polish that can make a good game great.

In this chapter, we examine various techniques for incorporating sound effects and music into Delphi games. We will examine using DirectSound for manipulation and playback of sound effects, and we will take a look at regular Windows MCI commands for playing music in both MIDI and Red Book Audio formats.

DirectX 6.1 and above features a DirectX component known as DirectMusic. DirectMusic could potentially revitalize the use of MIDI through its incredible features. It provides an interface for supporting downloadable instrument sounds as well as methods to change the playback of MIDI files on the fly. This can result in context-sensitive musical playback that, supposedly, may never be the same twice. Unfortunately, we were unable to cover DirectMusic in this volume, but look for several articles about its use appearing in trade magazines in the very near future.

## Sound Effects

Sound effects are very important in a game, as they help to immerse the player in the environment depicted by the game's playing area. Viewing an animation of a ship exploding is much more exciting when accompanied by a loud roar. Walking down a dark passageway is much more realistic when the hollow sounds of boot heels on damp stone is heard. Using sound effects in this manner can draw the user into the environment in ways that even the most advanced 3-D engine could never hope to accomplish.

In addition to immersion and suspension of belief, sound effects can give audible clues to the user that indicate a change in the environment. For example, the hum of an engine could be changed to reflect the amount of damage it has taken, providing a correlation to the engine's maximum speed. A low growl or heavy breathing sound could signal the presence of an unknown assailant, heightening the user's tension level and excitement. Even the subtle sounds of clicks and whirs emitted when the game's buttons and sliders are manipulated help make the environment all the more real to the user.

The Windows Multimedia API contains a number of sound output functions. Some are very easy to implement but lack the control that might be required for some games. Most games will need to play several sound effects at once, such as the whine of an engine, laser bursts, and explosions. Playing several sound effects simultaneously is known as mixing, and unfortunately such functionality is not built in to the Windows Multimedia API.



**mixing:** Combining several sounds in real time in such a way that they can be played simultaneously.

---

Low-level multimedia sound output functions could be used to implement mixing or even manipulate the sounds in real time to create special effects such as echo or reverb. However, such code is difficult to write at best, and just as difficult to debug. Let's examine the easiest method to implement sound output under Windows, and then we'll take a look at how DirectSound provides all the sound output power that a game will ever need.

### Simplistic Sound Effects

Not every game requires the power of full-blown DirectSound implementation. Indeed, if your game uses few sound effects, or does not require the ability to play several sounds simultaneously, it is much easier to implement the sound output functionality built into the Windows API. By far, the easiest yet most flexible function available for playing sound effects is the PlaySound function. The PlaySound function can play audio files in the WAV format, from either a file on disk, a resource, or even an image of the sound in memory. It can play the sound asynchronously, allowing the application to continue processing, and it can play sounds in a continuous loop. This function would be perfect for almost any gaming need except for one major drawback: It cannot play more than one sound at a time. As stated above, low-level audio functions could be used to implement mixing, but this is difficult, and now obsolete thanks to DirectSound. The PlaySound function is defined as:

```
function PlaySound(  
  
    pszSound: PChar;           // the sound to be played  
  
    hmod: HMODULE;            // a module handle  
  
    fdwSound: DWORD           // flags controlling sound playback  
  
): BOOL;                     // returns TRUE or FALSE
```

The first parameter indicates the sound to be played, and can be a filename, an alias, or a resource identifier as specified by the flags in the third parameter. If the first parameter is set to nil, any currently playing WAV is stopped. The second parameter contains the handle of the executable or DLL if the sound to be played is a resource; otherwise it should be set to zero. The final parameter is a series of flags that indicate how to interpret the first parameter as well as how the sound file should be played, such as in a loop.



*Note:* You must include MMSystem in the Uses clause of any unit that will use the PlaySound function.

---

The following example demonstrates how to use the PlaySound function to play sound effects from various locations.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 9-1: Playing sound effects with PlaySound*

---

implementation

```
uses MMSystem;
```

```
var
```

```
    WaveFile: TMemoryStream;
```

```
{ $R *.DFM }
```

```
{ $R WaveRes.res }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    {create the image of a WAV file in memory}
```

```
WaveFile := TMemoryStream.Create;

WaveFile.LoadFromFile('Memory.wav');

end;

procedure TForm1.FormDestroy(Sender: TObject);

begin

    {delete the in memory WAV file}

    WaveFile.Free;

end;

procedure TForm1.btnPlayFileClick(Sender: TObject);

begin

    {play a sound from a WAV file}

    PlaySound('File.wav', 0, SND_FILENAME or SND_ASYNC);

end;

procedure TForm1.btnPlayResClick(Sender: TObject);

begin

    {play a sound from a resource (see the WaveRes.rc file for the
    resource script)}

    PlaySound('WaveResource', hInstance, SND_RESOURCE or SND_ASYNC);

end;

procedure TForm1.btnPlayMemClick(Sender: TObject);

begin

    {play a sound from a WAV file image in memory}

    PlaySound(WaveFile.Memory, 0, SND_MEMORY or SND_ASYNC);

end;

procedure TForm1.btnPlayLoopClick(Sender: TObject);
```

```

begin
    {play a continuously looping sound from a file}
    PlaySound('Loop.wav', 0, SND_FILENAME or SND_ASYNC or SND_LOOP);

    btnPlayStop.Enabled := TRUE;
end;

procedure TForm1.btnPlayStopClick(Sender: TObject);
begin
    {stop all sound output}
    PlaySound(nil, 0, SND_PURGE);

    btnPlayStop.Enabled := FALSE;
end;

```

---

As illustrated, it is relatively simple to play a sound file by simply passing the PlaySound function the name of the WAV file. Playing a sound from a resource is almost as simple. The WAV files must be compiled into a resource by using the BRCC32.exe resource compiler included with Delphi (see the WaveRes.rc file in the directory containing the above example on the companion CD). Once this is complete, the resource file is linked into the executable using the \$R compiler directive. Then, it's a simple matter of passing the PlaySound function the name of the WAV resource (as opposed to the name of a WAV file), the handle of the module containing the resources (useful for putting WAV resources into DLLs), and the SND\_RESOURCE flag.



**Note:** WAV files must be compiled using a resource type of "WAVE" in the resource script file before PlaySound can find them.

---

Playing sounds from memory is easy enough, and improves the performance of your application as PlaySound does not have to reload the WAV file or resource each time it is called. By using the TMemoryStream object, a WAV file can easily be loaded into memory and passed to the PlaySound function via its Memory property.

Together with the ability to continuously play a sound (by using the SND\_LOOP flag) or stop all sound output (by using the SND\_PURGE flag), the PlaySound function is relatively powerful. If your game or application will not require the ability to play multiple sounds simultaneously, the PlaySound function will probably provide all the functionality required. However, to optimize sound output performance, play a virtually unlimited number of sounds simultaneously, or produce special effects, DirectSound is the way to go.



## DirectSound

DirectSound is that portion of DirectX that is responsible for the manipulation and playback of digitized sound (music is handled via other API functions described later in the chapter). While not nearly as simplistic as the PlaySound function, the methods in the DirectSound API provide the developer with plenty of audio playback control while not being nearly as difficult as writing your own mixing and manipulation code with low-level audio API calls.

### Features

DirectSound offers many features above and beyond what is available through the Win32 API. As with all DirectX components, DirectSound gives the developer more control over sound output than the Windows API, while also providing an improvement in performance. DirectSound provides the following features of specific interest to game developers:

- Automatic use of hardware acceleration (when available)
- Automatic conversion of WAV formats
- Simultaneous output of a virtually unlimited number of sounds
- Special effects such as panning, volume control, frequency adjustment, and 3-D placement

In the following sections we will discuss how the Delphi game developer can take advantage of each of these features to create rich sound output for applications or games.

### Functional Overview

Similar to DirectDraw, DirectSound programming consists of creating a DirectSound object and several sound buffers. The DirectSound object itself represents the developer's direct interface to the audio hardware. Through the IDirectSound interface, the application can query the audio hardware's capabilities, create sound buffers, and control other aspects of the hardware itself.

Of primary interest to DirectSound programmers is the secondary sound buffer object (or specifically the IDirectSoundBuffer object). A secondary sound buffer object represents a sound, or WAV file, in memory. It is a similar relationship as surfaces are to bitmaps in DirectDraw programming. This memory is typically located in system RAM. However, some sound cards contain hardware RAM for the purpose of accelerating sound output. DirectSound will automatically take advantage of this hardware memory by placing secondary sound buffers in the sound card RAM. All of this is transparent to the developer. The majority of time spent in programming DirectSound will be in creation and manipulation of these secondary sound buffers.

When a secondary sound buffer is played, its sound information is mixed with other secondary sound buffers and placed in the primary buffer. The primary buffer represents the conglomeration of all the sounds that will be played simultaneously, and is constantly streamed out to the sound output hardware. As secondary buffers are mixed into the primary buffer, DirectSound automatically converts their format into that of the primary buffer. This makes it easy for the developer to use sounds in varying formats, such as 8- and 16-bit sound, or sounds recorded at different sampling rates. As this mixing occurs, any special effects are applied to the sound, such as panning, volume, or 3-D positioning. The number of buffers that DirectSound can mix is limited only by processing time.

DirectSound provides a software mixer for combining system memory buffers and hardware buffers into the primary buffer. System memory buffers are always mixed in software, but if a hardware mixer is available, the mixed system buffers are streamed to the hardware mixer, along with the hardware buffers. This elaborate process is illustrated in Figure 9-1.



**Figure 9-1:** DirectSound functional overview



Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Sound Buffers** As stated above, the majority of time spent when programming DirectSound is in creation and manipulation of secondary sound buffers. The primary sound buffer can also be accessed, allowing the developer to change the primary sound buffer format or even implement custom mixing algorithms. There are a number of attributes and concepts associated with sound buffers that must be explained and understood before they can be put to good use.

**Audio Format** As of this writing, all sound buffers must contain sound data in the pulse code modulation (PCM) format.



**pulse code modulation (PCM):** An uncompressed audio format; the most common format for Windows WAV files.

Fortunately, this is the most common format for Windows WAV files. Unfortunately, you cannot assume that any file with the .wav extension is in the PCM format. WAV files are also stored in the Resource Interchange File Format (RIFF), which is a complex, variable length file format consisting of header "chunks" followed by data "chunks" of variable length.



**Resource Interchange File Format (RIFF):** A complex, variable length file consisting of variable length "chunks." Each chunk can describe data or a header (which contains a description of the data following it). Commonly used for multimedia file formats, such as WAV files.

The data chunks in WAV files can contain audio data in any number of different formats (you can examine this by opening the Sound Recorder under Accessories and choosing Save As). WAV RIFF files contain a header "chunk" in the form of a TWaveFormatEx structure, which allows the developer to check the format of the WAV file before copying its data into a secondary sound buffer. The TWaveFormatEx structure contains a number of useful pieces of information describing the WAV file, such as the number of channels, the sampling rate, and the number of bits per sample. The TWaveFormatEx structure is defined as:

TWaveFormatEx = packed record

```

wFormatTag: Word;          // format type, must be WAVE_FORMAT_PCM for
                             DirectSound compatibility

nChannels: Word;          // number of channels (1 = mono, 2 = stereo)}

nSamplesPerSec: DWORD;    // the sample rate, in hertz (samples per second),
                             typically 11025 (telephone quality), 22050 (radio
                             quality), or 44100 (CD quality)

nAvgBytesPerSec: DWORD;   // average transfer rate (nSamplesPerSec*nBlockAlign)

nBlockAlign: Word;        // bytes per sample (nChannels*nBitsPerSample/8)

wBitsPerSample: Word;     // number of bits per sample (8 or 16)

cbSize: Word;             // number of bytes of extra information, can be
                             ignored

end;

```



**hertz (Hz):** A measurement of samples per second.

---

The audio format of secondary buffers will be dictated by the format of the WAV file they contain. However, the primary buffer has a default audio format of 22,050 Hz, 2 channels, 8 bits per sample. DirectSound gives the developer the abilities to change this format, as we'll see shortly.

**Buffer Memory Locations** When DirectSound creates a secondary sound buffer, it automatically tries to locate the buffer in RAM on the sound card, if available. If there are no hardware buffers available, the sound is stored in system memory. Sound buffers located in hardware have the shortest path to the primary sound buffer, and thus are appropriately suited for sounds that are needed quickly and will be repeated often (such as gunshots), or looped continually (such as an engine hum). Although sound buffers are created in hardware memory by default, the developer can specify the desired memory location of a sound buffer by using specific flags, as described later in this chapter. The number of sound buffers that DirectSound can make are restricted only by available free memory (system memory and sound card hardware memory)

The existence of hardware buffers usually indicates the existence of a hardware mixer. Each hardware buffer takes up one hardware mixing channel (even if the buffer isn't playing), and the number of available hardware buffers is limited by the number of hardware mixing channels. This affects the number of sounds that can be mixed by the hardware mixer. DirectSound will take up the slack by performing any necessary mixing in the software mixer. This will all be transparent to the developer but is useful knowledge when optimizing performance.

**Buffer Types** Secondary sound buffers come in two flavors: static and streaming. Static buffers are used for short sounds that can be placed into memory in their entirety. These are typically short, often repeated or looping sounds. Streaming buffers are used for large sounds that cannot fit into a reasonably sized block of memory, or for sounds that change often and must be copied into the buffer in pieces as it is playing.

By default, DirectSound tries to create a streaming buffer. However, like the buffer memory location, the developer can indicate a desired buffer type through the use of various flags in the buffer creation function.

**Optimum Buffer Configuration** Since DirectSound tries to locate secondary sound buffers in hardware memory first, sound buffers should be created in order of importance. This will ensure that the sounds used most often will enjoy the greatest performance. However, to optimize performance, the developer should dictate the type of buffer when one is created, and where it will be placed. In general, static buffers should be used for short, often repeated or

looping sounds, and should be placed in hardware memory. Streaming buffers, on the other hand, should be placed in system memory (on ISA cards you have no choice, as hardware streaming buffers are not supported). Locating a streaming buffer in hardware does not eliminate any processing overhead as new audio data is continually moving between hardware and software. Thus, hardware buffers should be reserved only for buffers that are not continually updated.

The exception to this rule is when a sound buffer will be duplicated. DirectSound allows copies of a sound buffer to be created that contain their own play and write pointers but do not copy the actual buffer memory. This is useful when a certain sound will be played multiple times, but you want the user to hear the end of the previous sound while the beginning of the next sound is being played (i.e., the dissipation at the end of a laser burst while the next laser burst is charging). Duplicating a hardware buffer requires using hardware resources that may be needed elsewhere. Indeed, the call to duplicate a hardware buffer will fail if there are an insufficient number of hardware resources available. For this reason, it is better to place static buffers that will be duplicated in system memory. An example of duplicating a sound buffer for this purpose will be demonstrated later in the chapter.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

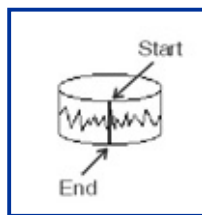
by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Buffer Memory Format** The memory in a DirectSound sound buffer is conceptually circular. Basically, this means that when DirectSound comes to the end of a buffer, it automatically goes back to the beginning. Think of sound buffer memory like the surface of a cylinder, as illustrated by Figure 9-2.

Each buffer contains a play and write pointer into the memory containing the audio data. The current play position indicates the offset of the next byte that will be sent to the mixer. The current write position is located beyond the current play position and indicates the first byte in the buffer at which it is safe to write new data. The data located between the play and write positions is queued for playback and should never be altered. The write position maintains a distance of approximately 15 milliseconds of play time ahead of the play position. Nevertheless, applications should be writing data to the buffer well ahead of the current write position to allow for any delays in moving the data into the buffer. This information is typically useful only when dealing with streaming buffers, as the application must track the point at which it is safe to stream more audio data into the buffer.



**Figure 9-2** Sound buffer memory is conceptually circular

**Audible Focus** The final important concept when dealing with sound buffers is that of audible focus. When a sound buffer is created, the developer can determine if the sound will continue to be heard when another application receives focus. By default, sounds will not be heard when the application loses focus (although they continue to play silently). A sound buffer can be given Global focus upon creation, indicating that the sound will always be audible unless another application receives focus and obtains exclusive access to the audio hardware (it will continue to play when the application is in the background). Alternatively, a sound buffer can have Sticky focus, indicating that it will be audible unless another application using DirectSound receives focus.

### Initialization

As with DirectDraw, there are several steps the developer must follow to set up DirectSound before loading and playing sound buffers. Some of these steps are somewhat optional, but in general, DirectSound is instantiated by performing the following:

- Enumerating audio devices
- Creating the DirectSound object
- Setting the cooperative level
- Retrieving audio device capabilities
- Setting the format of the primary buffer

**Enumerating Audio Devices** DirectSound provides a function that returns information for every audio output device with an installed hardware driver detected in the system. As a general rule, it is unnecessary for most applications to enumerate all audio output devices. The user can select a preferred audio output device through the Multimedia control panel applet, and DirectSound should be created using this selection unless there is good reason to do otherwise. However, if the developer wants the application to check all available audio drivers for certain capabilities, or just wishes to provide the user with a list of available audio drivers to choose from, the DirectSoundEnumerate function is used. The DirectSoundEnumerate function is defined as:

```
function DirectSoundEnumerate(
    lpDSEnumCallback: TDSEnumCallback;           // a pointer to the callback function
    lpContext: Pointer                             // an application-defined value
): HRESULT;                                       // returns a DirectX error code
```

The first parameter is a pointer to a user-defined callback function. DirectSoundEnumerate prepares a list of every installed audio driver and sends information about each one, one at a time, to this user-defined callback function. This information includes the GUID for the device, the description of the device, and the module name of the DirectSound driver for the device. The other parameter is a 32-bit user-defined value (this is defined as a pointer but is intended for application-specific use and is ignored by the function). The TDSEnumCallback callback function itself is defined as:

```
function TDSEnumCallback(
    lpGuid: PGUID;                                // the audio device GUID
    lpstrDescription: PAnsiChar;                  // a description of the device
    lpstrModule: PAnsiChar;                       // the module name
    lpContext: Pointer                             // the user-defined value
): BOOL;                                         // returns TRUE or FALSE
```

The first parameter is the GUID for the enumerated audio device. The second and third parameters contain textual information describing the audio device. The final parameter contains the user-defined value passed to the original DirectSoundEnumerate function. Every time the callback function is called, it should return a value of TRUE to continue the enumeration or FALSE if no other driver information is required.

The first device enumerated by this function is always the preferred audio output device selected by the user, as described above. This device is known as the primary sound driver.



**primary sound driver:** The audio output device selected by the user on the Audio tab of the Multimedia control panel applet.

---

Actually, this is not a true device enumeration, as no GUID will be returned for this device and the callback function will only receive a description string of “Primary Sound Driver” for the purpose of displaying it in a list. Once the primary sound driver has been enumerated, the callback function is called once for every available sound card.



---

**Note:** On any system with a sound card, the callback function will be called at least twice: once for the primary sound driver and once for the actual sound card hardware. A GUID will be received on the second and subsequent calls.

---

The following example demonstrates the use of the DirectSoundEnumerate function to retrieve the names of all available audio drivers installed on the system.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 9-2: Enumerating audio output devices*

---

{the user-defined enumeration function. note that the stdcall keyword must be used before Windows can call this function correctly}

```
function EnumSoundDevices(lpGuid: PGUID; // the device GUID
                           lpstrDescription: PAnsiChar; // the description
                           lpstrModule: PAnsiChar; // the module name
                           lpContext: Pointer // the user-defined info
                           ): BOOL; stdcall; // stdcall must be used
```

begin

```
{add the description and the module name of the enumerated driver to the
list box. the first one will be the primary sound driver, and will not
contain a module name}
```

```
Form1.lbxSoundDevices.Items.Add('lpstrDescription' - +lpstrModule);
```

```
{indicate that we wish to continue enumeration}
```

```
Result := TRUE;
```



```

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    {begin enumerating all available audio drivers}

    DirectSoundEnumerate(EnumSoundDevices, nil);
end;

```

---



**Figure 9-3:** The list of drivers retrieved from DirectSoundEnumerate

**Creating the DirectSound Object** To begin actually using DirectSound functionality, a DirectSound object must be created. This is accomplished by calling the DirectSoundCreate function. The DirectSoundCreate function is defined as:

```

function DirectSoundCreate(

lpGuid: PGUID;           // the sound device GUID

out ppDS: IDirectSound; // the IDirectSound object to be instantiated

pUnkOuter: IUnknown     // unused

): HRESULT;             // returns a DirectX error code

```

The first parameter is a pointer to the GUID of the desired audio device (retrieved through enumeration as explained above, or set to nil to indicate the primary sound driver). The second parameter is an IDirectSound interface which will be instantiated when the function returns. The final parameter is used for COM aggregation and should be set to nil.



**Caution:** It is possible to create multiple DirectSound objects, but this is inadvisable. When an additional DirectSound object is created, its cooperative level (explained below) overrides that of any previously created objects. A single DirectSound object should be created for the duration of the application.

---

**Setting the Cooperative Level** Once the DirectSound object is created, the application should immediately set its cooperative level. The cooperative level determines how the audio device will be shared between the current application and others, as well as access to certain functionality. The cooperative level is set by calling the SetCooperativeLevel method of the IDirectSound interface. The SetCooperativeLevel method is defined as:

```

function SetCooperativeLevel(

hwnd: HWND;           // a window handle

dwLevel: DWORD       // the cooperative level flags

```

```

): HRESULT; // returns a DirectX error code

```

The first parameter is a handle to the application's main window. The second parameter contains a series of flags indicating the desired cooperative level. The flags indicating each cooperative level are described in the table below. Most applications will use the Normal cooperative level, as this allows the most compatible operation with other applications that use the sound card. However, games will normally use the Priority or Exclusive levels, as these allow for the most efficient and optimized use of audio hardware. The cooperative levels below are listed in ascending order, and each level has the abilities of all levels below it.

**Table 9-1: DirectSound cooperative levels**

Level	Flag	Abilities	Restrictions
Normal	DSSCL_NORMAL	Can play secondary sound buffers.	Cannot set the format of the primary sound buffer, cannot write to the primary sound buffer, and cannot call the Compact method.
Priority	DSSCL_PRIORITY	Can change the format of the primary buffer, has first rights to hardware resources, and can call the Compact method.	
Exclusive	DSSCL_EXCLUSIVE	Mutes all sound output from background applications.	
Write Primary	DSSCL_WRITEPRIMARY	Can write directly to the primary sound buffer.	Cannot play secondary sound buffers.



**Note:** When using DirectDraw in conjunction with DirectSound, the window handle passed to the SetCooperativeLevel method of the IDirectSound object must also be sent to the SetCooperativeLevel of the IDirectDraw4 object.

**Retrieving Audio Device Capabilities** Most applications will be concerned with simply creating a DirectSound object and several secondary buffers, and outputting sound, letting DirectSound take care of the details. However, when creating high-performance, CPU-intensive games, it may be necessary to take a more active role in defining how the system will use available hardware resources. This is where the GetCaps method of the IDirectSound object comes in handy. The GetCaps method is defined as:

```

function GetCaps(
var lpDSCaps: TDSCaps // a TDSCaps structure
): HRESULT; // returns a DirectX error code

```

The only parameter for this function is a TDSCaps structure that is initialized with the capabilities of the audio device when the function returns. Several members of this structure, such as the dwMaxHwMixingAllBuffers, dwMaxHwMixingStaticBuffers, and dwTotalHwMemBytes members, can be used to optimize performance by determining what sounds can be placed in hardware buffers in the most efficient manner. Unfortunately, some members of this structure may not be universally supported by all drivers, and the values returned by some members may just be estimates. The best way to use the GetCaps method is to call it after creating each hardware sound buffer.



**Note:** The dwSize member of the TDSCaps structure must be initialized to SizeOf(TDSCaps) before calling the GetCaps method.

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

Of particular interest to the game programmer is the value of the dwFlags member. This member contains a combination of one or more flags that indicate device capabilities. Specifically, if the DSCAPS\_EMULDRIVER flag is included in this value, a DirectSound driver is not actually present in the system, and all DirectSound functionality is being emulated. This means that no hardware acceleration is available, secondary sound buffers can only be created in system memory, and playback latency will be higher. See the example program in Listing 9-3 for a list of available flags.

The TDSCaps structure is defined as:

TDSCaps = packed record

```
dwSize: DWORD; // the size of the structure, in bytes
dwFlags: DWORD; // capability flags
dwMinSecondarySampleRate: DWORD; // minimum sample rate for secondary
// buffers in hardware
dwMaxSecondarySampleRate: DWORD; // maximum sample rate for secondary
// buffers in hardware
dwPrimaryBuffers: DWORD; // number of primary buffers (always 1)
dwMaxHwMixingAllBuffers: DWORD; // maximum number of hardware buffers
dwMaxHwMixingStaticBuffers: DWORD; // maximum number of static hardware
// buffers
dwMaxHwMixingStreamingBuffers: DWORD; // maximum number of streaming hardware
// buffers
dwFreeHwMixingAllBuffers: DWORD; // number of free hardware buffers
dwFreeHwMixingStaticBuffers: DWORD; // number of free static hardware
```

```

        buffers
dwFreeHwMixingStreamingBuffers: DWORD; // number of free streaming hardware
        buffers
dwMaxHw3DAllBuffers: DWORD; // maximum number of 3-D hardware
        buffers
dwMaxHw3DStaticBuffers: DWORD; // maximum number of 3-D static hardware
        buffers
dwMaxHw3DStreamingBuffers: DWORD; // maximum number of 3-D streaming
        hardware buffers
dwFreeHw3DAllBuffers: DWORD; // number of free 3-D hardware buffers
dwFreeHw3DStaticBuffers: DWORD; // number of free 3-D static hardware
        buffers
dwFreeHw3DStreamingBuffers: DWORD; // number of free 3-D streaming hardware
        buffers
dwTotalHwMemBytes: DWORD; // total hardware memory, in bytes
dwFreeHwMemBytes: DWORD; // free hardware memory, in bytes
dwMaxContigFreeHwMemBytes: DWORD; // largest contiguous block of free
        hardware memory, in bytes
dwUnlockTransferRateHwBuffers: DWORD; // rate of data transfer to hardware
        buffers, in kilobytes per second
dwPlayCpuOverheadSwBuffers: DWORD; // percentage of CPU cycles used to mix
        software buffers
dwReserved1: DWORD; // reserved, not used
dwReserved2: DWORD; // reserved, not used
end;

```

The following example demonstrates how to create the DirectSound object, set its cooperative level, and retrieve audio hardware capabilities.

***Listing 9-3: Creating DirectSound and retrieving hardware capabilities***

---

```

procedure TForm1.FormCreate(Sender: TObject);
var

```

```

DXSound: IDirectSound;

DevCaps: TDSCaps;

begin
    {create the DirectSound object}
    DirectSoundCreate(nil, DXSound, nil);

    {set its cooperative level to the lowest setting, as we only need to
    retrieve device capabilities}
    DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

    {initialize the TDSCaps structure}
    FillChar(DevCaps, SizeOf(TDSCaps), 0);
    DevCaps.dwSize := SizeOf(TDSCaps);
    {retrieve the capabilities of the audio hardware}
    DXSound.GetCaps(DevCaps);

    {display general device capabilities}
    with lbxDevCaps.Items do
    begin
        Add('General driver capabilities -');
        Add('')
        if (DevCaps.dwFlags and DSCAPS_CERTIFIED) = DSCAPS_CERTIFIED then
            Add('Driver tested and certified by Microsoft');
        if (DevCaps.dwFlags and DSCAPS_CONTINUOUSRATE) = DSCAPS_CONTINUOUSRATE then
            Add('Supports all sample rates between min and max');
        if (DevCaps.dwFlags and DSCAPS_EMULDRIVER) = DSCAPS_EMULDRIVER then
            Add('No DirectSound driver detected');
        if (DevCaps.dwFlags and DSCAPS_PRIMARY16BIT) = DSCAPS_PRIMARY16BIT then
            Add('Supports 16-bit primary buffer');
        if (DevCaps.dwFlags and DSCAPS_PRIMARY8BIT) = DSCAPS_PRIMARY8BIT then
            Add('Supports 8-bit primary buffer');
        if (DevCaps.dwFlags and DSCAPS_PRIMARYMONO) = DSCAPS_PRIMARYMONO then

```

```

    Add('Supports a 1-channel primary buffer');

if (DevCaps.dwFlags and DSCAPS_PRIMARYSTEREO) = DSCAPS_PRIMARYSTEREO then
    Add('Supports a 2-channel primary buffer');

if (DevCaps.dwFlags and DSCAPS_SECONDARY16BIT) = DSCAPS_SECONDARY16BIT then
    Add('Supports 16-bit hardware mixed secondary buffers');

if (DevCaps.dwFlags and DSCAPS_SECONDARY8BIT) = DSCAPS_SECONDARY8BIT then
    Add('Supports 8-bit hardware mixed secondary buffers');

if (DevCaps.dwFlags and DSCAPS_SECONDARYMONO) = DSCAPS_SECONDARYMONO then
    Add('Supports 1 channel hardware mixed secondary buffers');

if (DevCaps.dwFlags and DSCAPS_SECONDARYSTEREO)=DSCAPS_SECONDARYSTEREO then
    Add('Supports 2 channel hardware mixed secondary buffers');

Add('Number of primary buffers: ' +IntToStr(DevCaps.dwPrimaryBuffers));

Add('Software mixing CPU overhead:' +
    IntToStr('DevCaps.dwPlayCpuOverheadSwBuffers')+'%');

{if the driver is a true DirectSound driver, it can take advantage of any
 hardware acceleration, so display hardware-specific attributes}

if (DevCaps.dwFlags and DSCAPS_EMULDRIVER) DSCAPS_EMULDRIVER then
begin
    Add('');

    Add('');

    Add('Hardware specific capabilities -');

    Add('');

    Add('Minimum secondary hardware buffer sample rate: +
        IntToStr(DevCaps.dwMinSecondarySampleRate));

    Add('Maximum secondary hardware buffer sample rate: +
        IntToStr(DevCaps.dwMaxSecondarySampleRate));

    Add('Maximum hardware buffers: +
        IntToStr(DevCaps.dwMaxHwMixingAllBuffers));

    Add('Maximum static hardware buffers: +
        IntToStr(DevCaps.dwMaxHwMixingStaticBuffers));

```

```
Add(Maximum streaming hardware buffers: +
    IntToStr(DevCaps.dwMaxHwMixingStreamingBuffers));
Add('Free hardware buffers:' + IntToStr(DevCaps.dwFreeHwMixingAllBuffers));
Add(Free static hardware buffers: +
    IntToStr(DevCaps.dwFreeHwMixingStaticBuffers));
Add(Free streaming hardware buffers: +
    IntToStr(DevCaps.dwFreeHwMixingStreamingBuffers));
Add(Maximum 3-D hardware buffers: +
    IntToStr(DevCaps.dwMaxHw3DAllBuffers));
Add(Maximum 3-D static hardware buffers: +
    IntToStr(DevCaps.dwMaxHw3DStaticBuffers));
Add(Maximum 3-D streaming hardware buffers: +
    IntToStr(DevCaps.dwMaxHw3DStreamingBuffers));
Add(Free 3-D hardware buffers: + IntToStr(DevCaps.dwFreeHw3DAllBuffers));
Add(Free 3-D static hardware buffers: +
    IntToStr(DevCaps.dwFreeHw3DStaticBuffers));
Add(Free 3-D streaming hardware buffers: +
    IntToStr(DevCaps.dwFreeHw3DStreamingBuffers));
Add(Total hardware memory: + IntToStr(DevCaps.dwTotalHwMemBytes)+ 'bytes');
Add('Free hardware memory: ' + IntToStr(DevCaps.dwFreeHwMemBytes)+ 'bytes');
Add(Maximum contiguous free hardware memory: +
    IntToStr(DevCaps.dwMaxContigFreeHwMemBytes)+ ' bytes');
Add('Data transfer rate:' +
    IntToStr(DevCaps.dwUnlockTransferRateHwBuffers)+ 'kb per second');
end;
end;
end;
```

---





**Figure 9-4:** The audio hardware capabilities

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



At this point, the pointer to the primary sound buffer object can be released, as the new format settings are persistent until the format of the primary sound buffer is again modified.



*Note:* The cooperative level of the DirectSound object must be set to a minimum of DSSCL\_PRIORITY before the format of the primary sound buffer can be changed.

---

In the following example, the sample sound is known to have a format of 44,100 Hz, 2 channels, 16 bits per sample. The format of the primary sound buffer is changed to that of the sample sound to facilitate the most optimum audio playback environment. Several functions not yet discussed are used in the example, but will be explained later in the chapter.

**Listing 9-4:** *Setting the format of the primary sound buffer*

---

```
var
    DXSound: IDirectSound;
    Sound: IDirectSoundBuffer;

procedure TForm1.FormCreate(Sender: TObject);
var
    BufDesc: TDSBufferDesc;
    Format: TWaveFormatEx;
    PrimeBuf: IDirectSoundBuffer;
begin
    {create the DirectSound object}
    DirectSoundCreate(nil, DXSound, nil);

    {set its cooperative level high enough to allow the application to
    change the format of the primary buffer}
    DXSound.SetCooperativeLevel(Handle, DSSCL_PRIORITY);

    {initialize the buffer description structure to indicate we wish to
    retrieve the primary buffer}
```

```

FillChar(BufDesc, SizeOf(TDSBufferDesc), 0);

BufDesc.dwSize      := SizeOf(TDSBufferDesc);
BufDesc.dwFlags     := DSBCAPS_PRIMARYBUFFER;
BufDesc.dwBufferBytes := 0;
BufDesc.lpwfxFormat := nil;

{retrieve the primary buffer}
DXSound.CreateSoundBuffer(BufDesc, PrimeBuf, nil);

{initialize the wave format structure to the desired primary buffer
format settings. this will be the same format used by the example
sound, which should have lower playback latency since it no longer
needs to be converted}
FillChar(Format, SizeOf(TWaveFormatEx), 0);
Format.wFormatTag      := WAVE_FORMAT_PCM;
Format.nChannels       := 2;
Format.nSamplesPerSec  := 44100;
Format.wBitsPerSample  := 16;
Format.nBlockAlign     := (Format.nChannels*Format.wBitsPerSample) div 8;
Format.nAvgBytesPerSec := Format.nSamplesPerSec*Format.nBlockAlign;

{set the format of the primary buffer}
PrimeBuf.SetFormat(Format);

{release the primary buffer (it retains the settings just defined)}
PrimeBuf := nil;

{load in the example sound}
Sound := DSLoadSound(DXSound, ExtractFilePath(ParamStr(0))+`Sample.wav`,
FALSE);

```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    {play the example sound (the primary buffer is now in the same format  
    as this sound buffer)}
```

```
    Sound.SetCurrentPosition(0);
```

```
    Sound.Play(0, 0, 0);
```

```
end;
```

---

## Creating Buffers

Now that DirectSound has been properly instantiated, the application has checked that the sound card provides the desired capabilities, and the format of the primary sound buffer has been changed to match that of the sounds to be played, it is time to start loading and playing sounds.

As with initialization, there are several steps the developer must follow in order to properly create a secondary sound buffer and load it with sound data. Unlike initialization, however, none of these steps are optional, and some are repeated regularly when creating a streaming buffer. In order to load and play a WAV file, the application must perform the following:

- Describe the format of the WAV file to be played
- Describe various secondary sound buffer attributes, such as the size of the buffer and any buffer capabilities
- Create the sound buffer (an IDirectSoundBuffer object) using the above information
- Copy the sound data from the WAV file into the buffer
- Play the sound buffer
- Repeat the last two steps for streaming buffers

By far, the most complex step in initializing a secondary sound buffer is copying the data from a WAV file into the sound buffer itself. Unfortunately, DirectSound does not come with any built-in methods for retrieving audio data from a file or a resource. However, the DirectX SDK comes with several units that contain functions for performing this necessary step, specifically the Wave.C and DSUtil.C units. These units have been converted to Pascal, and live in the form of the DSUtil.pas, DSWaveFiles.pas, and DSWaveResources.pas units. The DSUtil unit contains a function (DSLoadSound, used in the previous example) that will load a sound from a WAV file or a resource, passing back an instantiated IDirectSoundBuffer interface (static sounds only). The DSUtil unit uses functions located in the DSWaveFiles and DSWaveResources units to provide this functionality, and the examples that follow will discuss how to use these functions to initialize sound buffers with sound data.

[Previous](#) [Table of Contents](#) [Next](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**

[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

7.0

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

For the most part, the steps required to load and play a sound are identical for both static and streaming buffers. The primary difference is in how a streaming buffer is continually loaded with new information as it is played. The most common situation, and easiest to follow, is loading and playing static buffers, which is discussed first, followed by streaming buffers.

**Static Buffers** The first step in creating any secondary sound buffer is to initialize a `TWaveFormatEx` structure with information concerning the format of the sound buffer. Of course, this information comes from the desired sound itself. As described earlier in the chapter, WAV files are stored in the RIFF file format, which includes the information we are seeking conveniently configured in the `TWaveFormatEx` format. Parsing WAV file information out of a RIFF file is not too terribly complicated, but is involved enough that it would take several pages just to describe the process. Instead of wasting space describing such an esoteric task, we will use several functions from the `DSWaveFiles.pas` unit (converted from the original `Wave.C` unit found in the DirectX SDK) that perform this function. These functions use several of the low-level multimedia input/output functions, and perusing their code may provide useful insight for providing your own wave parsing routines.

To retrieve the desired information, we call the `WaveOpenFile` function. This function opens the WAV file and initializes a `TWaveFormatEx` structure (passed in as a variable parameter) with the WAV file's format information. Then we must call the `WaveStartDataRead` function, which positions a file pointer at the start of the actual sound data within the WAV file, and retrieves the total size of the sound data.

At this point, we are just about ready to create the actual sound buffer object, so we must initialize a `TDSBufferDesc` structure with information describing what type of sound buffer to create. The `TDSBufferDesc` structure is defined as:

```
TDSBufferDesc = packed record

dwSize: DWORD;           // the size of the structure, in bytes

dwFlags: DWORD;         // buffer description flags

dwBufferBytes: DWORD;   // the size of the sound buffer, in bytes

dwReserved: DWORD;     // reserved, must be 0
```

```
lpwfxFormat: PWaveFormatEx; // a pointer to the TWaveFormatEx structure  
end;
```



**Note:** The dwSize member of the TDSBufferDesc structure must be set to SizeOf(TDSBufferDesc) before calling the CreateSoundBuffer method.

For static buffers, the dwBufferBytes member should be set to the total size of the sound data as retrieved by the WaveStartDataRead function. The lpwfxFormat member should point to the TWaveFormatEx structure initialized by the WaveOpenFile function. The most important member of this structure, dwFlags, can be set to a combination of values that determine several things about the sound buffer, including its location in memory and its type. By default, DirectSound tries to create a streaming buffer, so this member must be set to DSBCAPS\_STATIC in order to create a static buffer. Also by default, the sound buffer will be located in hardware memory if any is available. The DSBCAPS\_LOCHARDWARE flag can be used to force the sound buffer into hardware memory (failing if there is not enough free hardware memory for the buffer), or the DSBCAPS\_LOCSOFTWARE flag can be used to force the sound buffer into system memory. Several other flags are available which provide other controls over the buffer, such as panning and volume, which will be explained below.

Now that the buffer description has been initialized, the CreateSoundBuffer method of the IDirectSound object is called. The CreateSoundBuffer method is defined as:

```
function CreateSoundBuffer(  
  
    const lpDSBufferDesc: TDSBufferDesc; // the buffer description  
  
    var lpIDirectSoundBuffer: IDirectSoundBuffer; // the sound buffer object  
  
    pUnkOuter: IUnknown // unused  
  
): HRESULT; // returns a DirectX error code
```

The first parameter is the TDSBufferDesc structure that has been initialized as discussed above. The second parameter is an IDirectSoundBuffer object that will be instantiated when the function returns. The final parameter is used for COM aggregation and should be set to nil. This creates a secondary sound buffer that is now ready to receive actual sound data.

In order to start copying audio data from the sound file into the buffer, we must retrieve a pointer directly to the sound buffer memory itself. This is accomplished by calling the Lock method of the IDirectSoundBuffer object (this is similar to calling the Lock method of a DirectDraw surface). The Lock method is defined as:

```
function Lock(  
  
    dwWriteCursor, // offset of start of lock  
  
    dwWriteBytes: DWORD; // number of bytes to lock  
  
    var lpLpvAudioPtr1: Pointer; // first memory pointer  
  
    var lpdwAudioBytes1: DWORD; // number of bytes locked  
  
    var lpLpvAudioPtr2: Pointer; // second memory pointer  
  
    var lpdwAudioBytes2: DWORD; // number of bytes locked
```



```
dwFlags: DWORD // lock control flags
): HRESULT; // returns a DirectX error code
```

The first parameter is the offset, in bytes, of the beginning of the locked memory from the start of the sound buffer. The second parameter indicates how many bytes should be locked. The next four parameters indicate where the buffer is actually locked and how many bytes were locked. These parameters are explained in detail below under streaming buffers. The final parameter can indicate that the entire buffer should be locked, or locking should begin from the position of the Write cursor.

The information returned by this function is, in turn, used in a call to the WaveReadFile function, which actually copies the audio data from the WAV file into the sound buffer. Once the data is copied, the Unlock method of the IDirectSoundBuffer is called to unlock the sound buffer memory, and the WaveCloseReadFile function is called to close the actual WAV file. The Unlock method is defined as:

```
function Unlock(
lpvAudioPtr1: Pointer; // offset of first lock
dwAudioBytes1: DWORD; // number of bytes to be unlocked
lpvAudioPtr2: Pointer; // offset of second lock
dwAudioBytes2: DWORD // number of bytes to be unlocked
): HRESULT; // returns a DirectX error code
```

The four parameters passed to this function correspond to the pointers and byte lock counts retrieved from the initial Lock method.

The sound buffer has been created and loaded with audio data, and is now ready to be output through the audio hardware. The SetCurrentPosition method of the IDirectSoundBuffer object can be called to set the actual position within the sound data to start the audio playback. The SetCurrentPosition method is defined as:

```
function SetCurrentPosition(
dwPosition: DWORD // the current position offset
): HRESULT; // returns a DirectX error code
```

The only parameter in this function is the offset, in bytes, from the beginning of the buffer to which the current playing position should be set.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Finally, the Play method is called to actually begin audio playback. The Play method is defined as:

```
function Play(  
dwReserved1, // reserved  
dwReserved2, // reserved  
dwFlags: DWORD // play control flags  
): HRESULT; // returns a DirectX error code
```

The first two parameters are unused and should be set to zero. The third parameter can be set to zero to simply play the sound once, or it can be set to DSBPLAY\_LOOPING to continuously play the sound. The Stop method can be called to discontinue any sound playback from the buffer (the Stop method has no parameters).



**Note:** Like DirectDraw surfaces, the memory allocated to secondary sound buffers can be lost when another application takes over the sound hardware. In this case, any calls to the Lock or Play methods will return DSERR\_BUFFERLOST. The Restore method can be called to reallocate the buffer memory. However, the contents of this memory will be invalid, and the audio data must be recopied into the sound buffer.

The following example demonstrates how to create a static sound buffer, load it with audio data from an external WAV file, and play it.

*Listing 9-5: Creating and playing static wave files*

var

```

DXSound: IDirectSound;

Sounds: array[0..5] of IDirectSoundBuffer; // holds all of our sound objects

const
  {the names of the WAV files to be used}
  FileNames: array[0..5] of string = ('bigz05.wav', 'bigzap.wav', 'bloop.wav',
                                       'boingg.wav', 'bwik.wav', 'elslide.wav');

procedure TForm1.FormCreate(Sender: TObject);

var
  iCount: Integer;

begin
  {create the DirectSound object}
  DirectSoundCreate(nil, DXSound, nil);

  {set its cooperative level (we won't change the primary buffer format, so
   a cooperative level of Normal is all that is needed)}
  DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

  {load in the six example sounds}
  for iCount := 0 to 5 do
    LoadSound(FileNames[iCount], Sounds[iCount]);
end;

.
.
.

{this function is used to load a WAV file and initialize an
 IDirectSoundBuffer object with the audio data}
procedure TForm1.LoadSound(FileName: string; var Buffer: IDirectSoundBuffer);

var
  Audio: PByte;

```

```

Junk: Pointer;

NumBytesLocked: DWORD;

JunkBytes: DWORD;

WaveFormat: PWaveFormatEx;

mmioHndle: HMMIO;

ChunkInfo,

ChunkInfoParent: TMMCKInfo;

BufDesc: TDSBufferDesc;

begin

    {open the specified WAV file and read its format information}

    WaveOpenFile(PChar(ExtractFilePath(ParamStr(0))+FileName), mmioHndle,

                WaveFormat, @ChunkInfoParent);

    {advance the file pointer to the beginning of audio data, and retrieve the

     size of the buffer required to hold all of the audio data}

    WaveStartDataRead(mmioHndle, @ChunkInfo, @ChunkInfoParent);

    {initialize a buffer description record, including the size of the audio

     data and its format retrieved from the previous two function calls}

    FillMemory(@BufDesc, SizeOf(TDSBufferDesc), 0);

    BufDesc.dwSize := SizeOf(TDSBufferDesc);

    BufDesc.dwFlags := DSBCAPS_STATIC; // could also use DSBCAPS_LOCHARDWARE to

                                     // force the buffer into hardware memory

    BufDesc.dwBufferBytes := ChunkInfo.cksize;

    BufDesc.lpwfxFormat := WaveFormat;

    {create the sound buffer object}

    DXSound.CreateSoundBuffer(BufDesc, Buffer, nil);

    {lock the entire buffer}

    Buffer.Lock(0, 0, Pointer(Audio), NumBytesLocked, Junk, JunkBytes,

```

```
DSBLOCK_ENTIREBUFFER);
```

```
{copy the audio data into the sound buffer object}
```

```
WaveReadFile(mmioHndle, NumBytesLocked, Audio, @ChunkInfo, JunkBytes);
```

```
{unlock the buffer}
```

```
Buffer.Unlock(Audio, NumBytesLocked, nil, 0);
```

```
{close the WAV file. at this point, the IDirectSound buffer has been  
initialized and is ready to be played}
```

```
WaveCloseReadFile(mmioHndle, WaveFormat);
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
{the Tag property of each button is set to the appropriate index into  
the array of IDirectSoundBuffer objects. when a button is pressed, reset  
the current position of the appropriate buffer, and start playing it}
```

```
Sounds[TButton(Sender).Tag].SetCurrentPosition(0);
```

```
Sounds[TButton(Sender).Tag].Play(0, 0, 0);
```

```
end;
```

---

The DSWaveResources.pas unit contains functions similar to those found in the DSWaveFiles.pas unit for loading WAV file information from resources. Internally, dramatically different API functions are called, mainly those used for manipulating resources on a binary level as opposed to multimedia file input/output functions. Specifically, the DSGetWaveResource function is called to initialize both the TDSBufferDesc and the TWaveFormatEx structures. The CreateSoundBuffer method is then called as usual, and the DSFillSoundBuffer function is called to actually copy the audio data into the buffer. The following example demonstrates how to use these functions to create and play static sound buffers from audio data in a resource. This example also demonstrates how to check for a lost sound buffer and restore its contents.

**Listing 9-6:** *Creating and playing static wave files from resource information*

---

```
var
```

```
DXSound: IDirectSound;
```

```

Sounds: array[0..5] of IDirectSoundBuffer; // holds all of our sound objects

const
    {the resource names of the waves to be used}
    ResNames: array[0..5] of PChar = ('bigz05', 'bigzap', 'bloop',
                                       'boingg', 'bwik', 'elslide');

    {this function is used to load a wave resource and initialize an
    IDirectSoundBuffer object with the audio data}
procedure LoadSound(ResName: PChar; var Buffer: IDirectSoundBuffer);
var
    BufDesc: TDSBufferDesc;
    pbWaveData: PByte;
begin
    {empty the buffer description record}
    FillMemory(@BufDesc, SizeOf(TDSBufferDesc), 0);

    {load the wave resource, initializing the format and required buffer size}
    if DSGetWaveResource(0, ResName, BufDesc.lpwfxFormat, pbWaveData,
                        BufDesc.dwBufferBytes) then
    begin
        {initialize the remaining buffer description members}
        BufDesc.dwSize := SizeOf(TDSBufferDesc);
        BufDesc.dwFlags := DSBCAPS_STATIC; // could also use DSBCAPS_LOCHARDWARE to
                                           // force the buffer into hardware memory

        {create the sound buffer object}
        DXSound.CreateSoundBuffer(BufDesc, Buffer, nil);

        {fill the sound buffer with the audio data from the resource}
        DSFillSoundBuffer(Buffer, pbWaveData, BufDesc.dwBufferBytes);
    end;
end;

```

```

end;

{this function is used to restore the contents of an IDirectSoundBuffer object
 when its memory has been lost. it is very similar to the LoadSound function}
procedure ReloadSound(ResName: PChar; var Buffer: IDirectSoundBuffer);
var
    BufDesc: TDSBufferDesc;
    pbWaveData: PByte;
begin
    {empty the buffer description record}
    FillMemory(@BufDesc, SizeOf(TDSBufferDesc), 0);

    {load the wave resource, initializing a pointer to the wave data
     and the size of the data}
    if DSGetWaveResource(0, ResName, BufDesc.lpwfxFormat, pbWaveData,
        BufDesc.dwBufferBytes) then
    begin
        {restore the buffer's memory}
        Buffer.Restore;

        {fill the sound buffer with the audio data from the resource}
        DSFillSoundBuffer(Buffer, pbWaveData, BufDesc.dwBufferBytes);
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    {the Tag property of each button is set to the appropriate index into
     the array of IDirectSoundBuffer objects. when a button is pressed, reset
     the current position of the appropriate buffer, and start playing it}
    Sounds[TButton(Sender).Tag].SetCurrentPosition(0);

```

```

{attempt to play the sound. if the buffer memory has been lost...}
if Sounds[TButton(Sender).Tag].Play(0, 0, 0) = DSERR_BUFFERLOST then
begin
    {...restore the sound buffer}

    ReloadSound(ResNames[TButton(Sender).Tag], Sounds[TButton(Sender).Tag]);

    {play the restored sound buffer}

    Sounds[TButton(Sender).Tag].SetCurrentPosition(0);

    Sounds[TButton(Sender).Tag].Play(0, 0, 0);

end;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
    iCount: Integer;
begin
    {create the DirectSound object}

    DirectSoundCreate(nil, DXSound, nil);

    {set its cooperative level (we won't change the primary buffer format, so
    a cooperative level of Normal is all that is needed)}

    DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

    {load in the six example sounds}

    for iCount := 0 to 5 do

        LoadSound(ResNames[iCount], Sounds[iCount]);

end;

```



Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Streaming Buffers** As mentioned previously, streaming buffers are used for large sounds that will not fit into a reasonably sized chunk of memory. A good use for streaming buffers would be for playback of digitized dialog, or perhaps a digitized music track. Creating streaming buffers is very similar to creating static buffers. The application must first read in a WAV file (or resource) and retrieve its format. Then, it must set up a TDSBufferDesc structure with the appropriate attributes.

There are two major differences between creating streaming buffers and creating static buffers: the flags used in the buffer description structure and the size of the buffer itself. As you would expect, the dwFlags member of the TDSBufferDesc structure will not contain the DSBCAPS\_STATIC flag. However, this member should contain the DSBCAPS\_GETCURRENTPOSITION2 flag to ensure the most accurate play position tracking. As for the size of the buffer, instead of setting it to the total size required to hold all of the audio data as in static buffer creation, it is set to a size large enough to hold approximately two seconds worth of audio data. Two seconds is an arbitrary number and depends on the application's specific needs. The actual size of the buffer is dependent on the wave's sample rate and bits per sample (the nAvgBytesPerSec member of the wave's TWaveFormatEx structure should be used).

To actually begin loading data into the buffer, we must first know where the current play and write positions are within the buffer (described earlier under Buffer Memory Format). The GetCurrentPosition method of the IDirectSoundBuffer object retrieves the offset from the beginning of the buffer for both the play and write positions. The GetCurrentPosition method is defined as:

```
function GetCurrentPosition(  
  
    lpdwCapturePosition,           // the current play position  
  
    lpdwReadPosition: PDWORD      // the current write position  
  
): HRESULT;                       // returns a DirectX error code
```

The first parameter is set to the current position of the play cursor, as an offset from the beginning of the buffer in bytes. The second parameter is set to the current position of the write cursor, again as an offset in

bytes from the beginning of the buffer.

How the data is loaded into the buffer is up to the application. For example, data could be written into the buffer right at the start of the write position (not recommended), or it could be written into the buffer at some point ahead of the current play position. In our examples below, we'll write data into the buffer based on the current play position.

You may have noticed how the Lock method returns two lock start offsets and two bytes locked counts. This is again due to the circular nature of buffer memory. For example, say the buffer is 5,000 bytes long, and the Lock method is called with a starting offset of 3,000 and a request to lock 3,000 bytes. The last 2,000 bytes in the buffer can be locked, but the final 1,000 requested bytes must be locked from the beginning of the buffer. This is why two offsets and two byte counts are returned. Again, depending on how your application copies data into the buffer, these values can be used to continuously copy data into the buffer in a circular format.

All that remains is to play the buffer and determine exactly when to write data into the buffer. We've already seen how a buffer can be played. In the case of streaming buffers, it might be useful to fill the buffer with silence and start playing it immediately, especially if the sound will be looped. Filling the buffer with silence involves simply zeroing out the buffer's memory using the techniques described above for writing data into a buffer.



*Note:* We have a function in the DSUtil unit called FillBufferWithSilence that performs this task.

---

As for when to write data into the buffer, there are two methods available to the application: polling and notification. Polling involves using the GetCurrentPosition method as described above. GetCurrentPosition is called at regular intervals (like every pass through the game's main loop), writing data into the buffer based on the Play or Write position returned at that moment. The advantages to this method are that it's simple and easy to implement. The disadvantages are that unless polling is performed at regular intervals, the Play position can easily race past the point where new data should have been copied into the buffer, resulting in old data being played back and a generally messed up audio output. The other method, notification, involves using the IDirectSoundNotify interface together with a number of low-level API functions such as CreateEvent and WaitForMultipleObjects to create event objects that are signaled when the Play position reaches certain points within the buffer. The advantage to this method is that it is more efficient than continually calling the GetCurrentPosition method. The disadvantages are that it is much more difficult to set up, requires multiple threads (in some cases), and is consequently more difficult to debug. For our purposes, the polling method will be more than adequate.

The following example demonstrates creating a streaming buffer and writing data into the buffer by polling the current play position.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 9-7: Creating a streaming buffer*

```
var  
  
    DXSound: IDirectSound;  
  
    Sound: IDirectSoundBuffer;  
  
  
    WaveFormat: PWaveFormatEx;  
  
    mmioHndle: HMMIO;  
  
    ChunkInfo,  
  
    ChunkInfoParent: TMMCKInfo;  
  
  
procedure TForm1.FormCreate(Sender: TObject);  
  
var  
  
    Audio: PByte;  
  
    Junk: Pointer;  
  
    NumBytesLocked: DWORD;  
  
    JunkBytes: DWORD;  
  
    BufDesc: TDSBufferDesc;
```

```

begin
    {the OnIdle event serves as our game loop as it would in
    DirectDraw examples}
    Application.OnIdle := AppIdle;

    {create the DirectSound object}
    DirectSoundCreate(nil, DXSound, nil);

    {set its cooperative level (we won't change the primary buffer format, so
    a cooperative level of Normal is all that is needed)}
    DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

    {open the specified WAV file and read its format information}
    WaveOpenFile(PChar(ExtractFilePath(ParamStr(0))+`longwave.wav`), mmioHndle,
        WaveFormat, @ChunkInfoParent);

    {advance the file pointer to the beginning of audio data, and retrieve the
    size of the buffer required to hold all of the audio data}
    WaveStartDataRead(mmioHndle, @ChunkInfo, @ChunkInfoParent);

    {initialize a buffer description record, including the size of the audio
    data and its format retrieved from the previous two function calls}
    FillMemory(@BufDesc, SizeOf(TDSBufferDesc), 0);
    BufDesc.dwSize := SizeOf(TDSBufferDesc);
    BufDesc.dwFlags := DSBCAPS_GETCURRENTPOSITION2 or DSBCAPS_LOCSOFTWARE;
    BufDesc.dwBufferBytes := WaveFormat.nAvgBytesPerSec*2; // 2 seconds worth
    BufDesc.lpwfxFormat := WaveFormat;

    {create the sound buffer object}

    DXSound.CreateSoundBuffer(BufDesc, Sound, nil);

```

```
{initialize it with silence}

FillBufferWithSilence(Sound);

{begin playing the buffer}

Sound.Play(0, 0, DSBPLAY_LOOPING);

end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    {disconnect our loop}

    Application.OnIdle := nil;

    {we no longer need to access the WAV file, so close it}

    WaveCloseReadFile(mmioHndle, WaveFormat);

end;

procedure TForm1.AppIdle(Sender: TObject; var Done: Boolean);
const
    LastPlayPos: Integer = 0;
var
    CurPlayPos,
    CurWritePos: Integer;
    StartOffset: Integer;
    Audio,
    Junk: Pointer;
    NumBytesLocked,
    JunkBytes,
    BytesRead: DWORD;
begin
    {setting Done to FALSE indicates that we want the application to
    continuously call this function}
```

```

Done := FALSE;

{if the DirectSound object has been created...}
if Assigned(Sound) then
begin
    {retrieve the play and write positions}
    Sound.GetCurrentPosition(@CurPlayPos, @CurWritePos);

    {since our buffer is only 2 seconds long, WaveFormat.nAvgBytesPerSec
    represents the midpoint of the buffer (1 second)}
    if (CurPlayPos<LastPlayPos) or ((CurPlayPos>=WaveFormat.nAvgBytesPerSec)
    and (LastPlayPos<WaveFormat.nAvgBytesPerSec)) then
    begin
        {wrap around the starting offset as necessary}
        If CurPlayPos>=WaveFormat.nAvgBytesPerSec then
            StartOffset := 0
        else
            StartOffset := WaveFormat.nAvgBytesPerSec;

        {lock the buffer, preparing it to write out 1 second of data}
        Sound.Lock(StartOffset, WaveFormat.nAvgBytesPerSec, Pointer(Audio),
            NumBytesLocked, Junk, JunkBytes, 0);

        {copy the audio data into the sound buffer object}
        WaveReadFile(mmioHndle, NumBytesLocked, Audio, @ChunkInfo, BytesRead);

        {copy any extra bytes as necessary}
        if BytesRead<NumBytesLocked then
        begin
            WaveStartDataRead(mmioHndle, @ChunkInfo, @ChunkInfoParent);

            WaveReadFile(mmioHndle, NumBytesLocked-BytesRead,
                Pointer(Longint(Audio)+BytesRead), @ChunkInfo, BytesRead);

```

```

end;

{unlock the buffer}

Sound.Unlock(Audio, NumBytesLocked, nil, 0);

{store the last known playing position}

LastPlayPos := CurPlayPos;

end;

end;

end;

```

---

## Duplicating Buffers

It is quite common in action games to have a short sound that is repeated often, such as gunshots, laser bursts, or even explosions. To play a regular static buffer in its entirety, the `SetCurrentPosition` method is called to “rewind” the sound, and then the `Play` method is called to begin audio output. If this sound is to be repeated rapidly, such as when the user is frantically pressing the fire button, the sound may seem to stutter, as each time the sound is replayed the previously playing sound is started from the beginning. The solution would be some method by which this sound buffer can be copied, sharing the sound data with the original buffer but with its own playing position. Fortunately, `DirectSound` provides a method by which a static buffer can be duplicated.

By calling the `DuplicateSoundBuffer` method, another `IDirectSoundBuffer` object is created. This new object has its own play and write pointers, but it shares the audio data with the original sound (as opposed to creating its own sound buffer). It can be played or stopped regardless of the state of the original sound, but since it shares the original buffer, a duplicated buffer should not be locked nor should its data be modified. The `DuplicateSoundBuffer` method is defined as:

```

function DuplicateSoundBuffer(

lpDsbOriginal: IDirectSoundBuffer;           // the original sound buffer

var lpDsbDuplicate: IDirectSoundBuffer      // the duplicated sound buffer

): HRESULT;                                 // returns a DirectX error code

```

The first parameter is a pointer to the sound buffer that is to be duplicated. The second parameter is an `IDirectSoundBuffer` object that is instantiated as a copy of the original when the function returns.

Bear in mind that although the sound buffer is not reproduced, duplicating a sound buffer does require some resources. Thus, it is better to locate static buffers to be duplicated in system memory, as duplicating a hardware buffer may require additional hardware resources that are not available or should be used for more efficient purposes.



**Note:** Calling `DuplicateSoundBuffer` on a hardware buffer will fail when no more (or insufficient) hardware resources are available.

---



The DirectX SDK comes with an object that encapsulates this buffer duplication, making it easy to manage sounds that must be played multiple times concurrently. While the following example makes no use of any such object, it is easy to see how one could potentially be created to encapsulate the functionality as illustrated. The example demonstrates how to play a single sound multiple times by using the DuplicateSoundBuffer method to create several copies of the sound. For comparison purposes, the original sound can be played as well, illustrating how using the DuplicateSoundBuffer method can help eliminate stutter from sounds played in rapid succession.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**

[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 9-8: Using the DuplicateSoundBuffer method*

```
var  
  
    DXSound: IDirectSound;  
  
    Sound: IDirectSoundBuffer;  
  
    {the array of duplicated sound buffers}  
    SoundArray: array[0..5] of IDirectSoundBuffer;  
  
    {stores the current index into the array}  
    CurSound: Integer;  
  
procedure TForm1.FormCreate(Sender: TObject);  
var  
    iCount: Integer;  
begin  
    {create the DirectSound object}
```

```

DirectSoundCreate(nil, DXSound, nil);

{set its cooperative level}
DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

{load in the example sound}
Sound := DSLoadSound(DXSound, ExtractFilePath(ParamStr(0))+`Sample.wav`,
                    FALSE);

{create several duplicates of the original sound}
for iCount := 0 to 5 do
    DXSound.DuplicateSoundBuffer(Sound, SoundArray[iCount]);

{set the index to the first element}
CurSound := 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    {simply play the original buffer from the beginning. results in a stutter
    if played rapidly}
    Sound.SetCurrentPosition(0);
    Sound.Play(0, 0, 0);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    {increment the index and wrap it around if necessary}
    Inc(CurSound);

    if CurSound>5 then
        CurSound := 0;
end;

```

```

    {play a duplicated buffer from the beginning}

    SoundArray[CurSound].SetCurrentPosition(0);

    SoundArray[CurSound].Play(0, 0, 0);

end;

```

## Buffer Controls

The `IDirectSoundBuffer` interface contains several methods by which the application can control certain aspects of the sound. Specifically, an application can control pan, frequency, and volume. This allows a developer to implement several special sound effects that can greatly enhance a game. For example, the pan control can be used to move the sound of an engine whine from side to side as a spaceship crosses the screen. Volume could be used to reduce and finally eliminate an engine whine as a ship moves farther away. The frequency could even be adjusted up or down to indicate the speed of an engine, or perhaps even damage.

Specifically, pan is controlled through the `GetPan` and `SetPan` methods, frequency is controlled through the `GetFrequency` and `SetFrequency` methods, and volume is controlled through the `GetVolume` and `SetVolume` methods. All methods take but a single parameter. Pan and volume are expressed in relative terms, but frequency is expressed in hertz. All attributes have an acceptable range, and pan and volume have a neutral value, which is illustrated in Table 9-2.

The desired sound buffer controls are specified when the `IDirectSoundBuffer` object is created. As mentioned previously, the `dwFlags` member of the `TDSBufferDesc` structure can be set to a combination of values that indicate the memory location of the sound buffer and other attributes. The three flags used to give control of a buffer's pan, frequency, and volume are illustrated in Table 9-2. Additionally, the `DBSCAPS_CTRLALL` or the `DBSCAPS_CTRLDEFAULT` flags can be used to indicate that the buffer has access to all controls or the pan, frequency, and volume controls, respectively.



**Caution:** As an issue of performance, buffers with additional controls require additional processing time for real-time modification of the sound data. Therefore, it is suggested that only the specific control flags truly needed for a buffer be used, and the `DBSCAPS_CTRLALL` or `DBSCAPS_CTRLDEFAULT` flags should only be used for buffers that actually need all controls.

**Table 9-2:** Pan, frequency, and volume ranges and neutral values

Control	Flag	Neutral Value	Range
Frequency	<code>DBSCAPS_CTRLFREQUENCY</code>	None	Software buffers: <code>DSBFREQUENCY_MIN</code> to <code>DSBFREQUENCY_MAX</code> .  Hardware buffers: dwMinSecondarySample rate to dwMaxSecondarySampleRate (see the <code>GetCaps</code> method)
Pan	<code>DBSCAPS_CTRLPAN</code>	<code>DSBPAN_CENTER</code>	<code>DSBPAN_LEFT</code> to <code>DSBPAN_RIGHT</code>
Volume	<code>DBSCAPS_CTRLVOLUME</code>	<code>DSBVOLUME_MAX</code>	<code>DSBVOLUME_MIN</code> to <code>DSBVOLUME_MAX</code>

The following example demonstrates how to create a sound buffer using the pan, volume, and frequency

controls, retrieve their settings, and modify their values in real time.

**Listing 9-9:** *Using buffer controls for special effects*

---

```
procedure TForm1.FormCreate(Sender: TObject);
var
    Audio: PByte;
    Junk: Pointer;
    NumBytesLocked: DWORD;
    JunkBytes: DWORD;
    WaveFormat: PWaveFormatEx;
    mmioHndle: HMMIO;
    ChunkInfo,
    ChunkInfoParent: TMMCKInfo;
    BufDesc: TDSBufferDesc;
    Pos: LongInt;
    FreqPos: DWORD;
begin
    {create the DirectSound object}
    DirectSoundCreate(nil, DXSound, nil);

    {set its cooperative level high enough to allow the application to
    change the format of the primary buffer}
    DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

    {open the WAV file}
    WaveOpenFile(PChar(ExtractFilePath(ParamStr(0))+ 'Example.wav'), mmioHndle,
                WaveFormat, @ChunkInfoParent);

    {begin reading WAV file data}
    WaveStartDataRead(mmioHndle, @ChunkInfo, @ChunkInfoParent);
```

```

{initialize the sound buffer description structure, indicating the
  controls we wish to use}
FillMemory(@BufDesc, SizeOf(TDSBufferDesc), 0);

BufDesc.dwSize := SizeOf(TDSBufferDesc);

BufDesc.dwFlags := DSBCAPS_STATIC or DSBCAPS_CTRLFREQUENCY or
                  DSBCAPS_CTRLPAN or DSBCAPS_CTRLVOLUME;

BufDesc.dwBufferBytes := ChunkInfo.cksize;

BufDesc.lpwfxFormat := WaveFormat;

{create the sound buffer}
DXCheck( DXSound.CreateSoundBuffer(BufDesc, Sound, nil) );

{copy the sound data into the buffer}
DXCheck( Sound.Lock(0, 0, Pointer(Audio), NumBytesLocked, Junk, JunkBytes,
                  DSBLOCK_ENTIREBUFFER) );

WaveReadFile(mmioHndle, NumBytesLocked, Audio, @ChunkInfo, JunkBytes);

DXCheck( Sound.Unlock(Audio, NumBytesLocked, nil, 0) );

{close the WAV file}

WaveCloseReadFile(mmioHndle, WaveFormat);

{initialize the Pan trackbar}

tbrPan.Min := DSBPAN_LEFT;

tbrPan.Max := DSBPAN_RIGHT;

Sound.GetPan(Pos);

tbrPan.Position := Pos;

{initialize the Volume trackbar}

tbrVolume.Min := DSBVOLUME_MIN;

```

```
tbrVolume.Max := DSBVOLUME_MAX;

Sound.GetVolume(Pos);

tbrVolume.Position := Pos;

{initialize the Frequency trackbar}

tbrFrequency.Max := DSBFREQUENCY_MAX;

tbrFrequency.Min := DSBFREQUENCY_MIN;

Sound.GetFrequency(FreqPos);

tbrFrequency.Position := FreqPos;

{begin playing the sound}

Sound.Play(0, 0, DSBPLAY_LOOPING);

end;

procedure TForm1.tbrPanChange(Sender: TObject);

begin

    {set the Pan position}

    Sound.SetPan(tbrPan.Position);

end;

procedure TForm1.tbrVolumeChange(Sender: TObject);

begin

    {set the Volume position}

    Sound.SetVolume(tbrVolume.Position);

end;

procedure TForm1.tbrFrequencyChange(Sender: TObject);

begin

    {set the Frequency}

    Sound.SetFrequency(tbrFrequency.Position);

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).





▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Music

Music, like sound effects, adds depth to a game, giving it a feeling of completeness. Even more than sound effects, music adds mood to a game, and when used effectively can adjust the user's frame of mind as effectively as a movie sound track.

Most games feature music that changes in response to changing conditions within the game. An effective use of music would be to play something eerie and mysterious when exploring catacombs, change to a heart-pounding rock-and-roll piece during the heat of combat, and end with a euphoric anthem to signal victory or a melancholic hymn to signal defeat. The difficulty comes in writing such musical ballads; programmatically outputting the result is easy by comparison.

The topic of musical output, while poorly documented, has been explored by a number of adventurous programmers, and as such there are several techniques for its implementation. One simple technique may be to simply digitize a musical piece and use the methods described above for its playback. Several other techniques involve digitized music compressed in various formats and then uncompressed on the fly as it is output to the audio device. However, for our purposes, we will examine only two common musical output techniques: MIDI and CD audio. Both of these techniques use Win32 MCI API commands, and as such do not make use of DirectX or DirectSound functionality.



**Tip:** As stated at the beginning of the chapter, the DirectX 6.1 and above SDK contains the new DirectMusic component that brings the power of DirectX to MIDI output. The header files were not available during the production of this book, and thus we were unable to cover them in this chapter. However, the DirectMusic header files are available on the CD accompanying this book.

## MCISendCommand

All of the examples to follow make use of a very powerful Win32 API function called MCISendCommand. This is a high-level command in the Media Control Interface (MCI) subsystem of Windows. The MCISendCommand function is simple in its declaration but very complex in terms of what it can do. You



**Desktop Library**  
[Click to access!](#)

can use the `MCISendCommand` function to take advantage of any native Windows multimedia functionality, including playback of WAV files, video files, and MIDI and CD music (as we'll see below), and interaction with video recording and playback equipment. Our uses of this function will be limited solely to MIDI and CD audio output, but you should be aware that this is a very powerful and complex function that unfortunately is poorly documented. The `MCISendCommand` function is defined as:

```
function mciSendCommand(  
  
mciId: MCIDEVICEID;           // the MCI device identifier  
  
uMessage: UINT;               // the command message  
  
dwParam1,                     // command message flags  
  
dwParam2: DWORD               // address of parameter structure  
  
) : MCIERROR;                 // returns an MCI error code
```

The first parameter to this function is the identifier of the MCI device to which the command should be sent. This is retrieved when the device is first opened. The second parameter is the actual command to be sent to the device. Each command instructs the device to perform a different task, and we'll see several commands in the following examples. The third parameter is a series of flags that are dependent on the type of command sent, and indicate which members of the structure identified by the last parameter contain valid data. In particular, the `MCI_WAIT` flag is used often to indicate that the function should not return until the command has been carried out. The final parameter takes a pointer to a data structure, again dependent upon the type of command being sent. Some data structures must be initialized, while others are initialized by the function and contain pertinent data when it returns. For some reason, due to the way this function was declared, you must pass the structure's address by typecasting it as a longint. This will be illustrated in the examples to follow.

## Notification

`MCISendCommand` is asynchronous by nature, and will return immediately unless the `MCI_WAIT` flag was specified. In this event, you can usually specify the `MCI_NOTIFY` flag to indicate that the function should notify the application when it has completed. This is accomplished by sending the application an `MM_MCINOTIFY` message. It is a simple matter to define a handler for this message, and this technique is used in Listing 9-10.

## MIDI

MIDI, or Musical Instrument Digital Interface, is a format for describing musical sound, much the same as the WAV format describes digitized sound. It is an effective means of providing music within a small footprint. Both software for creating MIDI files and freeware or shareware MIDI songs are readily available.



**MIDI (Musical Instrument Digital Interface):** A digital format for describing music.

---

In order to implement MIDI music playback, several steps must be followed. With the exception of detecting available hardware, none of these steps are optional. Outputting MIDI music is implemented by performing these tasks:

- Check for MIDI output hardware existence
- Open the MIDI output device

- Set the time format
- Commence playback
- Retrieve playback status
- Close the device

## Check for MIDI Output Hardware Existence

The target machine must have a device capable of outputting MIDI music. It would be very difficult these days to find a sound card that did not also include the ability to output MIDI music. Thus, it is safe to assume that any machine with a sound card is capable of outputting MIDI. However, the developer can use the function `midiOutGetNumDevs` to determine if there are indeed any devices that can output MIDI music. The `midiOutGetNumDevs` function takes no parameters and returns a simple integer. If this integer value is greater than zero, at least one hardware device has been found that is capable of MIDI music output.

## Open the MIDI Output Device

The first real step in playing MIDI music is to open the “sequencer” device (the hardware device capable of outputting MIDI music) and the MIDI file that will be played. This requires a variable of the type `TMCI_Open_Parms`. This data structure describes the device being opened and the file that will be played on the device. The `TMCI_Open_Parms` data structure is defined as:

```
TMCI_Open_Parms = record

    dwCallback: DWORD;           // a window handle

    wDeviceID: MCIDEVICEID;     // the device identifier

    lpstrDeviceType: PAnsiChar; // the name of the device

    lpstrElementName: PAnsiChar; // the name of the element

    lpstrAlias: PAnsiChar;      // a device alias

end;
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

**7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

The first member of this structure can be set to the handle of a window which will receive the MM\_MCINOTIFY message (if so desired) when the function completes. The second member will be set to the MCI device identifier when the function returns, and should be stored for later use. The third parameter is set to the name of the device; for MIDI file playback, this should be set to 'sequencer.' The fourth parameter is set to the name of the MIDI file, which is opened and prepared for playback when this function returns. The final member contains a device alias and for our purposes can be ignored.

For our purposes, we can also ignore the dwCallback member. Thus, to open the MIDI device and a MIDI file, we must initialize lpstrDeviceType to 'sequencer,' set lpstrElementName to the filename of our MIDI file, and pass the structure to the MCISendCommand function, specifying MCI\_OPEN as our command message. For example:

```
{open the MIDI output device, loading the Canyon MIDI song}

OpenParms.lpstrDeviceType := 'sequencer';

OpenParms.lpstrElementName := 'C:\Windows\Media\Canyon.mid';

MCISendCommand(0, MCI_OPEN, MCI_OPEN_TYPE or MCI_OPEN_ELEMENT or MCI_WAIT,

               LongInt(@OpenParms));

{save a handle to the device}

MCIDeviceID := OpenParms.wDeviceID;
```

When this command returns, the wDeviceID member of TMCI\_Open\_Parms will contain a device identifier for the opened MIDI sequencer. The application should store this value, as it will be used in all subsequent MCISendCommand function calls.

**Set the Time Format**

The MCI subsystem needs to know the time format for playback of various media so it will not come out garbled or trashed. MIDI music is timed by the millisecond, so we must inform MCI to set the time format for playback to milliseconds. This is accomplished by initializing a variable of type `TMCI_Set_Parms`. The `TMCI_Set_Parms` structure is defined as:

```
TMCI_Set_Parms = record

    dwCallback: DWORD;           // a window handle

    dwTimeFormat: DWORD;        // the time format

    dwAudio: DWORD;             // audio output channel

end;
```

The first member of this structure can be set to the handle of a window which will receive the `MM_MCINOTIFY` message (if so desired) when the function completes. The second parameter is a flag that indicates the time format. The final member indicates the audio output channel.

For our purposes, we are only interested in the `dwTimeFormat` member, which should be set to `MCI_FORMAT_MILLISECONDS` to indicate a millisecond time format. This initialized structure is then sent to the `MCISendCommand` function with a command message of `MCI_SET`. We must also pass in the device ID that we recorded from the previous call to indicate the device whose time format is being set. For example:

```
{set the time format to milliseconds}

SetParms.dwTimeFormat := MCI_FORMAT_MILLISECONDS;

MCISendCommand(MCIDeviceID, MCI_SET, MCI_WAIT or MCI_SET_TIME_FORMAT,

               LongInt(@SetParms));
```

## Commence Playback

The final step simply informs the multimedia system that we wish to play the file. This requires a variable of the type `TMCI_Play_Parms`. The `TMCI_Play_Parms` data structure is defined as:

```
TMCI_Play_Parms = record

    dwCallback: DWORD;           // a window handle

    dwFrom: DWORD;              // play start position

    dwTo: DWORD;                // play end position

end;
```

The first member of this structure can be set to the handle of a window which will receive the `MM_MCINOTIFY` message (if so desired) when the function completes. The second member indicates the position at which to start playback, and the final member indicates the position at which to end playback.

This data structure allows the application to specify the exact position from which to begin playback and an exact position at which to end the playback. Even though the `MCISendCommand` function requires a variable of this type, there is no need to initialize any of the members to simply play the MIDI file; it will play it from start to

finish by default. The MCI\_PLAY command message instructs Windows to begin playback. For example:

```
{commence song playback, indicating that we wish to receive an MM_MCINOTIFY
message when the playback has completed}

PlayParms.dwCallback := Handle;

MCISendCommand(MCIDeviceID, MCI_PLAY, MCI_NOTIFY, LongInt(@PlayParms));
```

Notice that we set the dwCallback member to the handle of our main window, and indicated that we wished to be notified when the function finishes. If we had used the MCI\_WAIT flag, our application would hang until the MIDI file had finished playing. Obviously, this is not acceptable, so we indicated that we would instead like to receive the MM\_MCINOTIFY message when the music stops. In our example below, we use this to implement a looping feature.

## Retrieve Playback Status

While certainly not a required step, most applications at some point may need to know the current position of playback. This could be used to change the music according to the context, waiting for playback to reach a specific point so the music can be changed at an appropriate time. We can retrieve the current playback status by using a variable of type TMCI\_Status\_Parms. TMCI\_Status\_Parms is defined as:

```
TMCI_Status_Parms = record

    dwCallback: DWORD;           // a window handle

    dwReturn: DWORD;            // the status value

    dwItem: DWORD;              // the status item

    dwTrack: DWORD;             // length

end;
```

The first member of this structure can be set to the handle of a window which will receive the MM\_MCINOTIFY message (if so desired) when the function completes. The second member contains the status value when the function returns. The third member contains a flag indicating the item whose status is to be retrieved. For our purposes, this can be set to MCI\_STATUS\_POSITION to indicate we wish to retrieve the status of the playback position. The final member indicates the length or number of tracks. The MCI\_STATUS command message indicates we wish to retrieve the device's status. For example:

```
{retrieve the current position of the song playback}

StatusParms.dwItem := MCI_STATUS_POSITION;

MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,

                LongInt(@StatusParms));
```

## Close the Device

When the application terminates or MIDI music playback is no longer required, the MCI device must be closed and released by calling the MCISendCommand with a command message of MCI\_CLOSE. The MCI device ID

must be passed in to the first parameter, but the last two parameters are unused and should be set to zero.

The following example implements the discussed steps in order to play back a MIDI music file.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).





**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 9-10: Playing MIDI music*

```
procedure TfrmMidiExample.FormCreate(Sender: TObject);
begin
    {determine if there are any devices capable of outputting MIDI music}
    if midiOutGetNumDevs<1 then
        begin
            ShowMessage('There are no devices attached to the system capable of' +
                'outputting MIDI music.');
            Application.Terminate;
        end;
    end;

procedure TfrmMidiExample.FormClose(Sender: TObject);
    var Action: TCloseAction);
begin
    {make sure that the MIDI device is closed before exiting the program}
    MCICommand(MCIDeviceID, MCI_CLOSE, 0, 0);
```



```

end;

procedure TfrmMidiExample.btnPlayClick(Sender: TObject);
var
    OpenParms: TMCI_Open_Parms;
    PlayParms: TMCI_Play_Parms;
    StatusParms: TMCI_Status_Parms;
    SetParms: TMCI_Set_Parms;
begin
    {open the MIDI output device, loading the Canyon MIDI song}
    OpenParms.lpstrDeviceType := 'sequencer';
    OpenParms.lpstrElementName := 'C:\Windows\Media\Canyon.mid';
    MCISendCommand(0, MCI_OPEN, MCI_OPEN_TYPE or MCI_OPEN_ELEMENT or MCI_WAIT,
        LongInt(@OpenParms));

    {save a handle to the device}
    MCIDeviceID := OpenParms.wDeviceID;

    {set the time format to milliseconds}
    SetParms.dwTimeFormat := MCI_FORMAT_MILLISECONDS;
    MCISendCommand(MCIDeviceID, MCI_SET, MCI_WAIT or MCI_SET_TIME_FORMAT,
        LongInt(@SetParms));

    {retrieve the length of the song}
    StatusParms.dwItem := MCI_STATUS_LENGTH;
    MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,
        LongInt(@StatusParms));

    {display this length in terms of minutes and seconds}
    Labell1.Caption := 'Format(%d:%.2d', [StatusParms.dwReturn div 1000 div 60,
        Trunc(Frac(StatusParms.dwReturn/1000/60)*60)]);

```

```

{commence song playback, indicating that we wish to receive an MM_MCINOTIFY
  message when the playback has completed}

PlayParms.dwCallback := Handle;

MCISendCommand(MCIDeviceID, MCI_PLAY, MCI_NOTIFY, LongInt(@PlayParms));

{enable the playback position polling timer}

tmrPosition.Enabled := TRUE;

end;

.
.
.

procedure TfrmMidiExample.tmrPositionTimer(Sender: TObject);

var
  StatusParms: TMCI_Status_Parms;

begin
  {retrieve the current position of the song playback}

  StatusParms.dwItem := MCI_STATUS_POSITION;

  MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,

    LongInt(@StatusParms));

  {indicate this position in terms of minutes and seconds}

  Label2.Caption := 'Format(%d:%.2d', [StatusParms.dwReturn div 1000 div 60,

    Trunc(Frac(StatusParms.dwReturn/1000/60)*60)]);

end;

```

---

## CD Audio

MIDI offers the advantage of providing large musical scores with minimal space requirements. However, unless the user has a dedicated MIDI sound card (unlikely unless they are a musician), musical sound quality will always be less than perfect. Many games are circumventing this limitation by playing their musical scores directly off of the CD on which the game is distributed. This offers the highest possible sound quality, although the amount of music

available to the game will be limited by the number of scores and their length, as CD audio requires much more storage space than MIDI files.

The method by which data and music is stored on the CD varies. In a mixed-mode CD, all executable data must be placed in what a regular CD player would recognize as the first track. Multisession CDs do not have this requirement.



**Caution:** While the application is accessing data from the CD, it cannot play a musical track, and vice versa.

---

The music data itself is stored on the CD in a format known as Red Book Audio. This format provides for stereo music at the highest quality, and is the format used by the professional music industry to record music onto conventional audio CDs. An interesting side effect to this arrangement is that a CD game taking advantage of Red Book Audio music can be placed into a conventional CD player and played as if it were any regular music CD. The data track will be nothing but white noise, but the musical tracks can be replayed as desired.



**Red Book Audio:** The digital format used to store professional music on conventional audio CDs.

---

It is important to note that the end user's machine must be set up correctly before it can play CD audio music. This involves installing a small wire that runs directly from the CD to the audio output hardware. While most new machines have this installed automatically, it may not always be available. Also, as an added benefit, this direct connection between the hardware results in true multitasking when playing CD audio. The audio signals are sent directly to the sound output hardware, bypassing the CPU completely. Thus, playing CD audio involves almost no CPU utilization, resulting in almost no performance penalties.

Like MIDI music playback, several steps must be followed in order to initialize and commence the playback of a CD audio track. CD audio playback is implemented by performing these tasks:

- Open the CD device
- Check for the existence of the CD
- Set the time format
- Retrieve the number of tracks and their length
- Commence playback
- Stop playback
- Close the device

## Open the CD Device

If the music for a game is stored on a CD along with the executable code, the target machine must contain a CD-ROM drive in order to use the application. Thus, we do not need to check for the existence of a CD-ROM drive. However, it is good practice to test for the existence of a CD in the drive before attempting to play a track from it. This will be discussed in a moment.

Like MIDI, we must first open the CD device. However, unlike MIDI, we do not need to specify a file. Again, we will need a variable of the type `TMCI_Open_Parms`, but this time its `lpstrDeviceType` member must be set to the string `'cdaudio'`. For example:

```
{open the cd audio playback device}

OpenParms.lpstrDeviceType := 'cdaudio';

MCISendCommand(0, MCI_OPEN, MCI_OPEN_TYPE or MCI_WAIT, Longint(@OpenParms));
```

```
{save a handle to the cd audio device}
```

```
MCIDeviceID := OpenParms.wDeviceID;
```

## Check for the Existence of the CD

Once the CD audio device is opened, we can poll it to determine if a CD is actually inserted. To accomplish this, we need a variable of the type `TMCI_Status_Parms`. The `dwItem` member of this data structure must be set to `MCI_STATUS_MEDIA_PRESENT` to indicate that we wish to test for the presence of a CD in the drive. We then call the `MCI SendCommand` function, passing `MCI_STATUS` in its second parameter. For example:

```
{determine if a CD is present in the drive}
```

```
StatusParms.dwItem := MCI_STATUS_MEDIA_PRESENT ;
```

```
MCI SendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,
```

```
LongInt(@StatusParms));
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

When this function returns, if the dwReturn member is set to zero, there is no CD present in the drive, and the application should display an error message asking the user to insert the CD, repeating the check as necessary. If the dwReturn member is greater than zero, there is a CD in the drive, although it may not be the game's CD. Each CD contains a unique identifying number that we can use to determine if it is the required CD. The example in Listing 9-11 demonstrates how to extract this identifying number.

### Set the Time Format

Like MIDI, we must inform the MCI subsystem as to the timing format used for playback. Red Book Audio is timed by tracks, minutes, seconds, and frames, where there are 75 frames per second. We will use a variable of type TMCI\_Set\_Parms, setting its dwTimeFormat member to MCI\_FORMAT\_TMSF to indicate this unusual timing format. For example:

```
{set the time format for the CD audio device}

SetParms.dwTimeFormat := MCI_FORMAT_TMSF;

MCISendCommand(MCIDeviceID, MCI_SET, MCI_WAIT or MCI_SET_TIME_FORMAT,

                Longint(@SetParms));
```

### Retrieve the Number of Tracks and Their Length

While it's not absolutely necessary to retrieve the number of tracks on a CD, you do need to record the length of each track. More specifically, you must record the length of the very last track for reasons that are explained below. For the purposes of our example, however, we will record the number of tracks on the CD and the length of each so they may be displayed by our CD player example.

We make use of another TMCI\_Status\_Parms variable, setting its dwItem member to MCI\_STATUS\_NUMBER\_OF\_TRACKS to indicate we wish to retrieve the number of tracks. Once this is accomplished, we can then set dwTrack to a specific track number (we'll iterate through each track) and the dwItem member to MCI\_STATUS\_LENGTH, indicating we wish to retrieve its length. For example:

```

{retrieve the number of tracks on the CD}

StatusParms.dwItem := MCI_STATUS_NUMBER_OF_TRACKS ;

MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,

               LongInt(@StatusParms));

NumTracks := StatusParms.dwReturn;

{for each track, indicate its length}

for iCount := 1 to NumTracks do

begin

    StatusParms.dwTrack := iCount;

    StatusParms.dwItem := MCI_STATUS_LENGTH;

    MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM or

                  MCI_TRACK, LongInt(@StatusParms));

    .

    .

    .

end;

```

## Commence Playback

Once the CD device is opened, and we have determined that a CD does in fact exist, we can play a track. For this, we will need a variable of the type `TMCI_Play_Parms`. However, we must make use of the `dwFrom` and `dwTo` members of this structure in order to indicate the exact desired track to play.

Because of the unusual timing format, we must use the `MCI_MAKE_TMSF` function to encode an appropriate starting and ending play position. Playing a single track is relatively simple, as we can specify a starting track with an ending track one greater than the starting track. For example:

```

PlayParms.dwFrom := MCI_MAKE_TMSF(1, 0, 0, 0);

PlayParms.dwTo   := MCI_MAKE_TMSF(2, 0, 0, 0);

MCISendCommand(MCIDeviceID, MCI_PLAY, MCI_FROM or MCI_TO, Longint(@PlayParms));

```

This indicates that we wish to play track one in its entirety. However, this technique will not work on the last track, since there is not another track beyond it (and setting the from member to a value one greater than the number of tracks on a CD results in an error). Therefore, you must specify the exact length of the track when playing the very last one. This is why we recorded all of the track lengths earlier. We'll see an implementation of this in Listing 9-11.

## Stop Playback

Before another CD track can be played, the currently playing track must be stopped. This is accomplished by simply calling the MCICommand function with a command message of MCI\_STOP. For example:

```
MCICommand(MCIDeviceID, MCI_STOP, MCI_WAIT, 0);
```

## Close the Device

As with MIDI, the MCI device must be closed and released by calling the MCICommand with a command message of MCI\_CLOSE when the application terminates. The MCI device ID must be passed in to the first parameter, but the last two parameters are unused and should be set to zero.

The following example implements the discussed steps in order to play back Red Book Audio data from a CD.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 9-11: Playing CD Audio tracks*

```
procedure TFormCDExample.FormCreate(Sender: TObject);  
  
var  
  
    OpenParms: TMCI_Open_Parms;  
  
    StatusParms: TMCI_Status_Parms;  
  
    SetParms: TMCI_Set_Parms;  
  
    InfoParms: TMCI_Info_Parms;  
  
  
    UniqueID: array[0..255] of char;  
  
    NumTracks,  
  
    iCount: Integer;  
  
begin  
  
    {open the cd audio playback device}  
  
    OpenParms.lpstrDeviceType := 'cdaudio';  
  
    MCISendCommand(0, MCI_OPEN, MCI_OPEN_TYPE or MCI_WAIT, Longint(@OpenParms));
```



```

{save a handle to the cd audio device}

MCIDeviceID := OpenParms.wDeviceID;

{determine if a CD is present in the drive}
StatusParms.dwItem := MCI_STATUS_MEDIA_PRESENT ;

MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,

                LongInt(@StatusParms));

if not LongBool(StatusParms.dwReturn) then

begin

    ShowMessage('No CD is inserted in the CD-ROM drive. Please insert a CD' +

                'and try again.');
```

Application.Terminate;

```
end;
```

```
{retrieve and display the unique CD identification number. this is for
information only, and is not required for playing a track from the CD}
InfoParms.dwRetSize := 255;
InfoParms.lpstrReturn := UniqueID;
MCISendCommand(MCIDeviceID, MCI_INFO, MCI_INFO_MEDIA_IDENTITY or MCI_WAIT,

                Longint(@InfoParms));

lblUniqueID.Caption := UniqueID;
```

```
{set the time format for the CD audio device}
SetParms.dwTimeFormat := MCI_FORMAT_TMSF;
MCISendCommand(MCIDeviceID, MCI_SET, MCI_WAIT or MCI_SET_TIME_FORMAT,

                Longint(@SetParms));
```

```
{retrieve the number of tracks on the CD}
StatusParms.dwItem := MCI_STATUS_NUMBER_OF_TRACKS ;
MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,

                LongInt(@StatusParms));

NumTracks := StatusParms.dwReturn;
```

```

{for each track, indicate its length}

for iCount := 1 to NumTracks do

begin

    StatusParms.dwTrack := iCount;

    StatusParms.dwItem := MCI_STATUS_LENGTH;

    MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM or

                    MCI_TRACK, LongInt(@StatusParms));

{when polling for length on a specific track, the value returned in the TMSF

    format is not exactly what you would expect. the Track macro returns

    minutes, the Minute macro returns seconds, the Second macro returns frames,

    and the Frame macro returns nothing. notice also that we are saving the

    exact length of each track in the Objects property}

lbxCDTracks.Items.AddObject('Track +IntToStr(iCount)+' - +

                            'Format(%d:%.2d',

                            [MCI_TMSF_TRACK(StatusParms.dwReturn),

                            MCI_TMSF_MINUTE(StatusParms.dwReturn)]),

                            TObject(MCI_MAKE_TMSF(

                            iCount, MCI_TMSF_TRACK(StatusParms.dwReturn),

                            MCI_TMSF_MINUTE(StatusParms.dwReturn),

                            MCI_TMSF_SECOND(StatusParms.dwReturn))));

    end;

end;

procedure TfrmCDEExample.lbxCDTracksDblClick(Sender: TObject);

var

    PlayParms: TMCI_Play_Parms;

begin

    {the user has indicated that another track should be played. therefore, stop

    any CD audio currently being output}

```

```
MCISendCommand(MCIDeviceID, MCI_STOP, MCI_WAIT, 0);
```

```
{the index of the selected item matches the index of the requested CD audio  
track. however, CD audio track indexes are one-based, and list box indexes  
are zero-based, so we must add a one to the selected item index. this  
indicates the starting track from which to begin playback}
```

```
PlayParms.dwFrom := MCI_MAKE_TMSF(lbxCDTracks.ItemIndex+1, 0, 0, 0);
```

```
{remember that we stored the actual length of each track in the Objects  
property along with the track name? since there is no track following the  
last track on the CD, we cannot tell the device to play the current track  
to the next track. to play the last track on the CD, we must indicate the  
exact length of that track. therefore, if we are playing the last track,  
we extract the track's exact length from the Objects property. otherwise  
we can simply tell it to play to the next track}
```

```
if lbxCDTracks.ItemIndex = lbxCDTracks.Items.Count-1 then
```

```
    PlayParms.dwTo := Longint(lbxCDTracks.Items.Objects[lbxCDTracks.ItemIndex])
```

```
else
```

```
    PlayParms.dwTo := MCI_MAKE_TMSF(lbxCDTracks.ItemIndex+2, 0, 0, 0);
```

```
{instruct the CD audio device to play the indicated selection}
```

```
MCISendCommand(MCIDeviceID, MCI_PLAY, MCI_FROM or MCI_TO,
```

```
    Longint(@PlayParms));
```

```
end;
```

```
procedure TfrmCDEExample.TimerTimer(Sender: TObject);
```

```
var
```

```
    StatusParms: TMCI_Status_Parms;
```

```
begin
```

```
    {poll the CD to determine the position of the current playback}
```

```
    StatusParms.dwItem := MCI_STATUS_POSITION;
```

```
    MCISendCommand(MCIDeviceID, MCI_STATUS, MCI_WAIT or MCI_STATUS_ITEM,
```

```
LongInt(@StatusParms));
```

```
{the returned position value is in TMSF format. thus, we must extract the  
appropriate minutes and seconds values}
```

```
lblPosition.Caption := 'Format(%d:%.2d',  
                        [MCI_TMSF_MINUTE(StatusParms.dwReturn),  
                        MCI_TMSF_SECOND(StatusParms.dwReturn)]);
```

```
end;
```

```
procedure TfrmCDEExample.FormClose(Sender: TObject; var Action: TCloseAction);
```

```
begin
```

```
{the application is terminating, so stop any CD audio playback, and  
close the CD audio device}
```

```
MCISendCommand(MCIDeviceID, MCI_STOP, MCI_WAIT, 0);
```

```
MCISendCommand(MCIDeviceID, MCI_CLOSE, 0, 0);
```

```
end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Summary**

In this chapter, we discussed several techniques for playing both sound effects and music. We examined various DirectSound methods for performing digitized audio playback, including changing the format of the primary buffer as well as duplicating sound buffers. We also covered the steps required to output MIDI music as well as more superior techniques for playing audio tracks from a CD. When coding for sound or music output using DirectSound or Win32 multimedia API functions, it is important to keep these points in mind:

- Sound effects are very important for game enjoyment, as they not only enhance the believability of the gaming environment but also provide the player with audible clues that indicate changes in the game state.
- The PlaySound function is incredibly powerful for what it does. It can be used to play sounds from a file, from a resource, or from memory. While it may be adequate under some circumstances, it cannot play multiple sounds simultaneously.
- DirectSound offers many features above and beyond what is available with Win32 API functions. Hardware acceleration is automatically utilized when available. A virtually unlimited number of sounds can be mixed and played simultaneously, and several special effects can be applied to the output, such as panning, volume control, and frequency adjustment.
- Similar to DirectDraw, DirectSound programming consists of creating a DirectSound object and several sound buffers. The DirectSound object itself represents the developer's direct interface to the audio hardware. Through the IDirectSound interface, the application can query the audio hardware's capabilities, create sound buffers, and control other aspects of the hardware itself.
- DirectSound buffers are circular in nature. This is important when dealing with streaming buffers, as accessing buffer memory may require extra steps when starting near the end of the buffer.
- When DirectSound creates a secondary sound buffer, it automatically tries to locate the buffer in RAM on the sound card, if available. If there are no hardware buffers available, the sound is stored in system memory. Sound buffers located in hardware have the shortest path to the primary sound buffer, and thus are appropriately suited for short sounds that are needed quickly and will be repeated often.
- Secondary sound buffers come in two flavors: static and streaming. Static buffers are used for short sounds that can be placed into memory in their entirety. These are typically short, often repeated or looping sounds. Streaming buffers are used for large sounds that cannot fit into a reasonably sized

block of memory or for sounds that change often, and must be copied into the buffer in pieces as it is playing. By default, DirectSound tries to create a streaming buffer.

- Sound buffers should be created in order of importance. This will ensure that the sounds used most often will enjoy the greatest performance. However, to optimize performance, the developer should dictate the type of buffer when one is created, and where it will be placed. In general, static buffers should be used for short, often repeated or looping sounds, and should be placed in hardware memory. Streaming buffers should be placed in system memory. The exception to this rule is when a sound buffer will be duplicated. Duplicating a hardware buffer requires using hardware resources that may be needed elsewhere. Thus, it is better to place static buffers that will be duplicated in system memory.
- When a sound buffer is created, the developer can determine if the sound will continue to be heard when another application receives focus. By default, sounds will not be heard when the application loses focus (although they continue to play silently).
- DirectSound is optimized for 16-bits-per-sample sounds. Therefore, changing the primary sound buffer format for 16-bit sounds can further reduce playback latency and reduce the load on the CPU when processing sound data.
- The most complex step in initializing a secondary sound buffer is copying the data from a WAV file into the sound buffer itself. Unfortunately, DirectSound does not come with any built-in methods for retrieving audio data from a file or a resource. However, the DSUtil.pas, DSWaveFiles.pas, and DSWaveResources.pas units contain translations from DirectX SDK sound utility functions to copy audio data from a WAV file or a resource into a DirectSound buffer.
- Music, like sound effects, adds depth to a game, giving it a feeling of completeness. Even more than sound effects, music adds mood to a game, and when used effectively can adjust the user's frame of mind as effectively as a movie sound track.
- The Windows Media Control Interface features the MCISendCommand function that allows us to easily play both MIDI music and Red Book Audio from CDs. However, DirectX 6.1 and above feature a new DirectX component called DirectMusic that may revolutionize the use of MIDI.
- The MCISendCommand function is asynchronous by nature, and will return immediately unless the MCI\_WAIT flag was specified. In this event, you can usually specify the MCI\_NOTIFY flag to indicate that the function should notify the application when it has completed. This is accomplished by sending the application an MM\_MCINOTIFY message. It is a simple matter to define a handler for this message.
- MIDI, or Musical Instrument Digital Interface, is a format for describing musical sound, much the same as the WAV format describes digitized sound. It is an effective means of providing music within a small footprint, and both software for creating MIDI files as well as freeware or shareware MIDI songs are readily available.
- MIDI musical sound quality is less than perfect. Many games are circumventing this limitation by playing their musical scores directly off of the CD on which the game is distributed. This offers the highest possible sound quality, although the amount of music available to the game will be limited by the number of scores and their length, as CD audio requires much more storage space than MIDI files.
- Music data is stored on the CD in a format known as Red Book Audio. This format provides for stereo music at the highest quality, and is the format used by the professional music industry to record music onto conventional audio CDs. The end user's machine must be set up correctly before it can play CD audio music. This involves installing a small wire that runs directly from the CD to the audio output hardware, and while most new machines have this installed automatically, it may not always be available.

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITMAP

CONTACT US

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 10 Optimization Techniques

This chapter covers the following topics:

- Optimization theory
- Delphi-specific optimization techniques
- General optimization techniques
- Additional optimization considerations

The technique of optimizing code has been elevated to an art form. With today's more advanced hardware, operating systems, and compilers, typically only the more experienced programmers will be relegated the duty of optimization. One must be intimately familiar with the workings of these three elements to truly be a master optimizer. However, there are several general techniques that are well publicized that can dramatically increase the speed at which an application runs, and you don't have to eat, sleep, and breath assembly code to use them. Fortunately, Delphi helps a lot in the optimization department. There are also a number of Delphi-specific code traps that can dramatically slow down your applications. Optimization is a very broad topic, and there are several books that examine it exhaustively. In this chapter, we will examine a number of both Delphi-related optimization techniques and some general techniques that can be applied in almost any programming language.



**Note:** You should run the examples in this chapter with optimization turned off (under Project | Options | Compiler). The example code is somewhat simplistic, and Delphi's internal optimizations are usually so good that they can further optimize the code. Running the examples with optimization turned off gives you a better idea as to how the illustrated optimization will benefit application performance in a real-world situation. Also note that because the examples are simplistic, the loops within them are generally repeated more times than what would normally be encountered.

### Optimization Theory



While optimization can be considered more an art than a science, there are a few general optimization rules that everyone should follow. The most important step in optimization is knowing where to optimize. After you've determined where the bottlenecks are in application execution, the next step is to evaluate the problem and determine how to optimize the code.

You could take a guess at which part of your code is the slowest, but this method will most likely result in wasted time and increased frustration. The best course of action would be to employ a profiler to time program execution. Profilers generally report the number of times a particular function was called and the average length of execution. Using a profiler can quickly highlight the bottlenecks in an application, indicating a general location where optimization would be most effective.

The freeware Delphi profiler GpProfile is an excellent profiling tool for all 32-bit versions of Delphi, and it comes with source code. At the time of this writing, it is available at <http://members.xoom.com/primozg/gpprofile/gpprofile.htm>. Additionally, several Delphi tool vendors are working on retail versions of Delphi profilers, which should be available by the time this book reaches store shelves.



**profiler:** A profiler is an application that typically measures application execution time. The level at which an application is timed usually varies, from timing the execution of individual lines to the execution time of functions, procedures, or the overall application.

---

In general, it is best to ignore optimization until the application is almost complete. Trying to write optimized code from the outset can be very distracting. Concentrate on actually getting the application to run first. Chances are that a developer could spend several hours optimizing a piece of code that is ultimately replaced midway through development. Optimizing code as a last step in the development process ensures that the time will be well spent.

### Timing Code

In the absence of a profiler, there are several techniques a Delphi programmer can employ to time the execution of functions and methods. The one employed by the examples in this book involves the use of the `timeGetTime` function. The `timeGetTime` function retrieves the time that has elapsed since Windows was started, in milliseconds. Under Windows 95, this has a precision of 1 millisecond. Under Windows NT, the default precision is 5 or more milliseconds, but this can be adjusted by calling the `timeBeginPeriod` and `timeEndPeriod` functions.



**Note:** You must include `MMSYSTEM` in the `Uses` clause of any unit using the `timeGetTime` function.

---

Timing a function involves only three steps: recording the current value of `timeGetTime` in a variable, running the function code, and then again recording the current value of `timeGetTime`. The difference of these two values equals the total function execution time, as the following example illustrates.

#### *Listing 10-1: Using `timeGetTime` to time function execution*

---

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var
```

```

StartTime,

EndTime: LongInt;

begin

    {start tracking the time}

    StartTime := timeGetTime;

    {code to be timed goes here}

    {record the ending time}

    EndTime := timeGetTime;

    {the difference between the starting and ending times gives

    us the overall time required to execute the timed code}

    Label1.Caption := 'Elapse Time: '+IntToStr(EndTime - StartTime);

end;

```

---

The `timeGetTime` function should be as accurate as required for just about any profiling needs. However, if more accuracy is required, the `QueryPerformanceCounter` and `QueryPerformanceFrequency` functions can be used. These functions give the developer access to the high-resolution performance counter that is present in just about every computer manufactured since the 486 was cutting-edge technology. `QueryPerformanceFrequency` returns the number of counts per second that the high-resolution timer records, and `QueryPerformanceCounter` returns the current count. Bear in mind, however, that these functions are useful only for timing very short periods, and cannot be used to time anything that executes for more than one second.

### Examining the Algorithm

Once the bottleneck has been found, the first step in optimizing the code should be to reevaluate the algorithm. Examining the algorithm embodied by the code and finding a better algorithm will typically result in improved performance over that gained by any one particular optimization technique described below.

The classic example is the bubble sort versus the binary sort. Both algorithms achieve the same result—a sorted list. However, the binary sort method is much faster than the bubble sort. Had a developer spent time employing various specific optimization techniques such as using assembly language or lookup tables, the performance of the original code may have been improved, but most likely would have resulted in a lot of wasted time and effort.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Effective Optimization**

Knowing where to begin is probably the hardest part about optimizing code. In general, don't worry about optimizing short pieces of code that are not called very often in the application. For example, if a few lines of code (or even one line) takes two seconds to process but is only called a few times during the entire life of the application, it is not worth examining. If possible, this type of code should be placed where a user will not mind the wait. In the typical game application, this usually involves the loading of game resources such as graphics and sound. This type of code should be executed when an application is starting and a splash screen is being displayed. Code of this nature can also be executed when changing levels or starting a level.

Typically, the best place to begin optimization is within the innermost part of loops. Take, for example, the following pseudo-code:

```
for iCount := 0 to 10 do  
  
begin  
  
    {outer loop code}  
  
    for iCount2 := 0 to 100 do  
  
begin  
  
    {inner loop code}  
  
end;  
  
end;
```

If the outer loop code took a total of 100 milliseconds to execute for one iteration, excluding the inner loop, it would take a total of 1000 milliseconds, or one full second, for the entire loop to process (iterating 10 times).

If the inner loop code took a total of 10 milliseconds to execute one iteration, it would take a total of 1000 milliseconds to process (iterating 100 times). Since the outer loop code takes longer to execute than the inner loop code (excluding the fact that the inner loop is executed with the outer loop code), one might be tempted to try optimizing the outer loop. The grand total execution time required to process this example is approximately 11 seconds (1 full second for the inner loop, executed 10 times (1 \* 10), plus 1 more second for the outer loop). Even though the outer loop takes longer to process than the inner loop, if the total execution time of the outer loop was reduced to even 1 millisecond, the entire process will still take more than 10 seconds to execute. Now, if the inner loop execution time was cut in half (from 10 milliseconds to 5), the total execution time would be reduced from 11 seconds to approximately 6 seconds. That's a big savings. As this example illustrates, the biggest performance enhancements will usually come from optimizing code performed in the innermost loops.

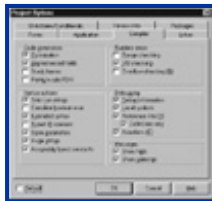


**Tip:** Typical culprits, and places to begin looking for optimization opportunities, usually include loops that redraw the entire screen a pixel at a time as well as manager loops that perform specific actions on each item in a list.

Finally, knowing when to quit is almost as important as knowing where to start. Optimization gurus state that any code, however optimized, could always be optimized further. However, there is a diminishing margin of returns encountered at a certain level of optimization. For example, say a developer spends 10 hours optimizing code, decreasing its execution time by 100 milliseconds. The time spent was probably worth it. Now, if it takes another 5 or 10 hours to reduce the execution time by another 10 milliseconds, the developer's time could probably be better spent elsewhere. At some point, execution time will only be dropping by a few milliseconds, and a decision will need to be made about whether or not the time spent performing the optimization is worth the performance increases.

## Delphi-Specific Optimization Techniques

Delphi itself offers a number of very useful optimization options. The compiler performs quite a few automatic optimizations. There are several other options available that also affect how Delphi generates machine code. All of these options are available on the Compiler tab of the Project Options dialog, accessed using Project|Options.



**Figure 10-1:** Project options



**Caution:** If you are not familiar with these settings or have never used them before, Delphi may act in a manner that might be quite unfamiliar. Get to know these settings and how they affect Delphi's overall performance and behavior.

## Automatic Compiler Optimizations

The very first check box under Code Generation is the Optimization option. Clicking this box on activates a number of automatic compiler optimization techniques. Unfortunately, with this setting, it's an all-or-nothing affair. Turning this option off for debugging purposes is useful, but always turn this option on before performing a final build. When this option is on, it allows the compiler to perform a number of heavy-duty optimizations that can dramatically affect the performance of code. Some of these optimizations include register optimizations, call stack elimination, common subexpression removal, loop induction variables, and dead code elimination.

## Register Optimization

If Delphi determines that a particular variable is used heavily, it will store that variable in a hardware register. This eliminates the need to access the value of the variable by loading it from memory, resulting in much faster code. Delphi takes this one step further by determining if certain variables are used extensively in different parts of code. If so, a single register can be used for each variable.

## Call Stack Elimination

Delphi offers a number of different calling conventions that affect how parameters are passed to a function. Different calling conventions are required based on the task being accomplished, such as calling functions stored in C or C++ DLLs, or using Windows API functions. However, by default, Delphi uses a calling convention known as register. The register calling convention will pass up to three parameters to a function by using hardware registers. This makes it unnecessary to create a stack frame, eliminating quite a bit of initialization and cleanup code and improving application performance. However, if more than three parameters are passed, registers cannot be used and a stack frame will be created. Bear in mind that while a normal function can pass up to three parameters and still use this optimization, object methods can only pass two parameters, as Self is automatically passed as the third parameter. If a function requires more than three parameters, one way to still benefit from this optimization is to create a record structure defining all of the parameters. A single pointer to this record can then be passed to the function, allowing the function to access all of the required information while passing only one parameter in a register.



**Tip:** A stack is also generated when a function returns more than a 32-bit number, such as returning a string. Try to write functions that don't return anything more than a single 32-bit number.

---

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

7.0

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Common Subexpression Removal

When compiling complex mathematical equations, Delphi will examine the equation to determine if any computations are performed more than once. If so, these common subexpressions are removed. Instead, Delphi will perform this calculation only once, and the resulting value will be used wherever the original calculation was located in the equation. The result is a mathematical computation that is reduced to a very compact, efficient form. This allows the developer to write computationally intensive code in a clear, concise manner without obscuring it by hand-optimizing the equation.

## Loop Induction Variables

If a variable is used only as an index into an array or a string, Delphi will instead replace the variable with a pointer that is incremented to directly access the string or array values. This eliminates multiplication operations and increases the speed at which array or string values are accessed. Additionally, using an index that is of a data type native to the hardware (such as a 32-bit integer type) will improve performance.

## Dead Code Elimination

While processing code, if Delphi detects that there is a block of code that is never called from the application, or code that ultimately has no effect, it will not be compiled into the final application. This is useful for reducing the size of the final build. However, during debugging, the compiler may remove code that would be useful to examine after a breakpoint.

## Additional Compiler Optimizations

While these automatic compiler optimizations are quite handy, there are several other options available under the Project Options dialog box that can affect the behavior of an application.

## Aligned Record Fields

When this option is turned on, all elements of record structures are adjusted so that they align on DWORD (32-bit) boundaries. The Intel architecture natively uses a 32-bit information format, and aligning the elements of a record so that each element begins at an address evenly divisible by 32 bits improves performance when accessing each element. The packed keyword can be used when declaring a record structure that will force alignment of the



elements in that record structure to 32-bit boundaries.



**Caution:** Bear in mind that this can change the size of a record structure, which is a problem when reading in information that was previously saved as non-aligned records. This may also have an effect on function calls to Windows or other third-party DLLs that take a record structure as a parameter.

---

## Stack Frames

When this option is checked on, the compiler will always pass parameters to functions on the stack, creating stack frames for every function and defeating the stack frame elimination optimization. This can be turned on for debugging purposes, but should always be turned off for the final build.

## Range Checking

When this option is turned on, all array and string indexing expressions are verified as being within their defined bounds, and all assignments to scalar and subrange variables are checked to be within range. This generates additional code when accessing array and string elements, ultimately degrading application performance. Turn this option on only while debugging, and make sure it is off for the final build.

## Overflow Checking

Similar to range checking, overflow checking controls the generation of overflow checking code for certain integer arithmetic operations. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. Thus, when this option is turned on, additional code is generated for every arithmetic operation performed, degrading performance. Again, turn this option on for debugging purposes, but turn it off for the final build.



**Tip:** You can force any of these options on or off in your source code by including compiler directives. These directives, however, will override any settings made in this dialog box, which may cause confusion.

---

## Other Delphi-Specific Optimization Techniques

While these compiler optimizations help to improve overall application performance, there are several other tricks specific to Delphi that can dramatically increase application speed. Armed with a little knowledge about specific details about Delphi and the Object Pascal language, a developer can avoid common pitfalls that can completely negate the performance benefits realized by the compiler optimizations. Let's examine some very specific techniques that can improve performance in a number of ways.

## Exceptions

Exceptions are intended to inform both the user and the developer when something has gone wrong. Although they are very useful when employed correctly, they are incredibly slow. The developer should avoid trapping for exceptions at all costs, especially within a loop. The classic example is trapping for the `EZeroDivide` exception when performing division. If a number is divided by zero, it will raise an `EZeroDivide` exception. This could be put into a `Try..Except` block, simply swallowing the exception if it occurs. However, this is incredibly slow, and a simple example of this iterated as few as 10,000 times will take several seconds to execute on a Pentium 333 (on which the example was tested). If this `Try..Except` block is replaced by a simple `If..Then` to check if the divisor is zero before performing the math, 10,000 iterations take place in less than a millisecond (again, tested on a Pentium 333). Therefore, do not rely on exceptions for trapping errors when writing high-performance code. The following example demonstrates this technique.

---





*Note:* You will need to turn off Stop on Delphi Exception for this example, located in Tools | Debugger Options | Language Exceptions.

---

**Listing 10-2:** *Exception handling in a loop*

---

```
procedure TForm1.Button1Click(Sender: TObject);
var
    iCount: Integer;
    Junk: Real;
    Divisor: Integer;

    StartTime,
    EndTime: LongInt;
begin
    {make sure that optimization is off}
    {$O-}

    {begin recording the time}
    StartTime := timeGetTime;

    {perform a task that may or may not generate an exception (in this example
    it will always generate an exception). we don't want the user to see this
    exception, so we programmatically handle it}
    Divisor := 0;
    for iCount := 0 to 10000 do
    begin
        try
            Junk := 10 / Divisor;
        except
            on EZeroDivide do;
        end;
    end;
end;
```

```

{recording the ending time...}
EndTime := timeGetTime;

{...and display the elapsed time}
Label1.Caption := 'Elapse Time (exception): '+IntToStr(EndTime - StartTime);

{begin recording the time}
StartTime := timeGetTime;

{perform the same task as before, but this time specifically check for
instances that may create an exception}
Divisor := 0;
for iCount := 0 to 10000 do
begin
    if Divisor <> 0 then
        Junk := 10 / Divisor;
end;

{record the ending time...}
EndTime := timeGetTime;

{...and display the elapsed time}
Label2.Caption := 'Elapse Time (if..then): '+IntToStr(EndTime - StartTime);
end;

```



**Tip:** Try..Finally blocks incur little to no overhead, so it is safe to wrap high-performance code within a Try..Finally block.

---

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Object Method Types

When declaring an object method, it can be defined as one of three types: static, virtual, or dynamic. Each one has its strengths and weaknesses. Static methods do not gain the benefits of inheritance or polymorphism, one of the cornerstones of object-oriented programming. Virtual and dynamic methods both allow for inheritance and polymorphism, but the way in which applications access these methods is implemented differently. Without expounding on the inner workings of Delphi too greatly, virtual methods create an entry in a Virtual Method Table that is stored with each object. Dynamic methods, on the other hand, use a different method call-dispatching algorithm. Where dynamic methods are optimized for code size, virtual methods are optimized for speed. However, static methods are the fastest of all, since they are called directly with no need for method dispatching or VMT lookups. In general, try to use static methods as often as possible, but definitely use virtual instead of dynamic when using object-oriented programming techniques.

## Variants

The variant data type is another new Delphi addition that makes certain programming practices incredibly easy. Using the variant type is almost essential when performing OLE or COM programming. However, because of its abilities to store just about any type of data, inexperienced programmers may be tempted to use only variables of type variant so as to avoid any data type incompatibilities. Unfortunately, because of its flexibility, the variant data type is very slow and should be avoided in high-performance code. With the variant being a 16-byte structure, it also wastes space. Unless you are doing some sort of OLE programming, avoid variants and use native data types.

*Listing 10-3: Variants versus native types*

---

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
    iCount: Integer;
```

```
Junk: Variant;

Junk2: Integer;

StartTime,

EndTime: LongInt;

begin

    {turn optimization off}

    {$O-}

    {begin recording the time}

    StartTime := timeGetTime;

    {perform some standard task using a variant data type}

    Junk := 0;

    for iCount := 0 to 100000 do

        begin

            Junk := Junk + 1;

        end;

    {record the ending time...}

    EndTime := timeGetTime;

    {...and display the elapsed time}

    Label1.Caption := 'Elapse Time (variant): '+IntToStr(EndTime - StartTime);

    {begin recording the time}

    StartTime := timeGetTime;

    {perform the same function using a native data type}

    Junk2 := 0;

    for iCount := 0 to 100000 do
```

```
begin
    Junk2 := Junk2 + 1;
end;

{record the ending time...}

EndTime := timeGetTime;

{...and display the elapsed time}

Label2.Caption := 'Elapse Time (integer): '+IntToStr(EndTime - StartTime);
end;
```

---

## String Parameters

An entire chapter could be written on the internal operations performed when Delphi works with strings. Both string and variant local variables within a function or method require prologue and epilogue code to be generated around the function to initialize and deinitialize the variable. Temporary storage variables may be generated when passing string results from one function as the parameter to another (such as passing the result of an `IntToStr` directly as a parameter), further increasing the generated code. Variant and string local variables automatically generate a stack frame within a function, defeating register optimizations. In general, avoid creating small utility routines that work on or use strings. If necessary, string or variant data should be processed in fewer, large routines that do as much as possible. Never use string variables or parameters to process information where an integer variable or parameter could accomplish the same task.

## Data Types

The data types used by your application will impact its overall performance. Certain mathematical functions perform faster using certain data types; indeed, some data types are incredibly slow by comparison. In general, you'll achieve the greatest performance when using native data types, such as integer. The `MathTest` example on the CD in the directory for this chapter can be used as a benchmark for testing execution speed of different operations on different data types. All of the floating-point and integer data types are tested, and though the exact execution times will vary from machine to machine, it is easy to see which data types perform better under certain circumstances than others.

## General Optimization Techniques

We gain a lot of benefit from the automatic optimization that Delphi provides. We can also improve performance by being aware of the inner workings of certain Delphi code constructs. However, we should also be aware of general coding practices that can be used to improve performance in any language.

### Loop Unrolling

It is very common for the point of execution within an application to jump from one point to another during the course of execution. For example, an `If..Then` statement causes the point of execution to skip some instructions based on the expression value. The time required for the execution point to change a single time is minimal. However, when the execution point jumps around repeatedly, these minimal time increments can accumulate.

The biggest offenders in this category are loops, such as For..Next, While..Do, and Repeat..Until. Applications typically revolve around looping constructs such as these, performing the same functions on large amounts of data. This results in the execution point changing repeatedly. The amount of code and the task it performs within loops are prime targets for optimization. However, one technique you can use to improve loop performance is to unroll it.

The concept of unrolling a loop is simple. Instead of performing one block of operations several times, perform several blocks of the same operations a fewer number of times. For example, if you have a loop that performs one operation 10,000 times, try performing 10 of those operations within the loop, and reduce the loop to run only 1,000 times. This will reduce the number of execution point jumps by an order of magnitude, and can dramatically improve overall performance. The following example illustrates this technique.

***Listing 10-4: Unrolling a loop***

---

```
procedure TForm1.Button1Click(Sender: TObject);
var
    StartTime,
    EndTime,
    iCount: Integer;
    Accum: Real;
begin
    {turn optimization off}
    {$O-}

    {initialize our variable and start tracking the time}
    Accum := 0;
    StartTime := timeGetTime;

    {perform some task many, many times}
    for iCount := 0 to 999999 do
        Accum := Accum + 0.5;

    {record the ending time...}
    EndTime := timeGetTime-StartTime;

    {...and display the elapsed time}
```

```
Label1.Caption := 'Regular loop time: '+IntToStr(EndTime);

{rest the variable and start tracking time}

Accum := 0;

StartTime := timeGetTime;

{perform the same task as before, but this time repeat the task several
times within the loop, and adjust the number of loops accordingly. note
that the task is performed the same number of times as before, but now
we've got an order of magnitude less jumps}

for iCount := 0 to 99999 do

begin

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

    Accum := Accum + 0.5;

end;

{record the ending time...}

EndTime := timeGetTime-StartTime;

{...and display the elapsed time}

Label2.Caption := 'Unrolled loop time: '+IntToStr(EndTime);

end;
```



---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Lookup Tables**

Most game applications require the use of complicated or sophisticated mathematical functions. Functions such as Sin and Cos are common, and some games may require even more arcane trigonometry or calculus equations. Unfortunately, even with math coprocessors, these functions are quite slow, and if they are required in a loop, the application can slow to a crawl. However, if the total range of all inputs to a mathematical function or equation is known, you may be able to use a lookup table.

Simply put, a lookup table is an array that contains the results of a mathematical function or equation for all known input values, where the input value acts as an index into the table. For example, the Sin and Cos functions both take an argument in the form of a degree (actually they take radians, but the argument can be converted to radians and it's easier to think of it as a degree). The total range of inputs that return unique outputs is 0 to 359 degrees. Thus, you can create a 360-entry array and put the Sin or Cos value for each input in the range in its appropriate entry. Then, instead of calling the Sin or Cos function, you simply use the argument as an index into this array. Retrieving predetermined values in this manner is incredibly fast, usually much more so than actually running the equation or function. The table is generally initialized when the application starts. This technique can be applied to any area where a limited range of input values is known. The following listing illustrates this technique.

*Listing 10-5: Using lookup tables*

---

```
procedure TForm1.Button1Click(Sender: TObject);  
  
var  
  
    StartTime, EndTime, iCount: Integer;  
  
    Lookup: array[0..359] of Real;  
  
    SinValue: Real;  
  
begin
```

```
{turn optimization off}

{$O-}

{generate the lookup table}
for iCount := 0 to 359 do
    Lookup[iCount] := Sin((iCount)*(PI / 180));

{begin tracking the time}
StartTime := timeGetTime;

{perform a slow mathematical calculation}
for iCount := 0 to 99999 do
    SinValue := Sin(50*(PI / 180));

{record the ending time...}
EndTime := timeGetTime-StartTime;

{...and display the elapsed time}
Label1.Caption := 'Sine Calculation: '+IntToStr(EndTime);

{start tracking the time}
StartTime := timeGetTime;

{perform the calculation by finding the answer in the lookup table}
for iCount := 0 to 99999 do
    SinValue := Lookup[50];

{record the ending time...}
EndTime := timeGetTime-StartTime;
```

```
{...and display the elapsed time}

Label2.Caption := 'Sine Lookup: '+IntToStr(EndTime);

end;
```

---

## Binary Bit Shifting

Every PC processor has built-in hardware to shift the bits in a register left or right. This has the result of effectively multiplying or dividing the number in this register by a power of 2. Applications can take advantage of this hardware through the use of the shl and shr functions. Shl shifts bits to the left, multiplying the number by two; shr shifts them to the right, dividing by two. This can be used for incredibly fast integer multiplication and division, as it bypasses the math coprocessor. The only drawback to this is that the multiplication or division has to be a multiple of two. However, if an equation can be broken down into components that use simple addition or subtraction with multiplication or division by a power of 2, performance can be dramatically improved. For example, the equation  $Y * 320 + X$  would require one multiplication and one addition by the math coprocessor. However, the equivalent function  $((Y \text{ shl } 8) + (Y \text{ shl } 6)) + X$  requires two shifts and two additions. Although the number of functions performed on this equation has increased, additions take less time than multiplication, and the shifts are almost instantaneous, resulting in faster performance. The following example demonstrates how to use bit shifting for multiplication.

### *Listing 10-6: Bit Shifting*

---

```
procedure TForm1.Button1Click(Sender: TObject);

var

    iCount: Integer;

    Result: Longword;

    StartTime,

    EndTime: LongInt;

begin

    {we must run this example with optimization turned on to see the

    performance gains}

    {$O+}

    {set up a dummy variable and begin tracking the time}

    Result := 1;

    StartTime := timeGetTime;
```

```
{perform some function or task}

for iCount := 0 to 10000000 do
    Result := Result * 2;

{record the ending time...}
EndTime := timeGetTime;

{..and display the elapsed time}
Label1.Caption := 'Elapse Time (multiply): '+IntToStr(EndTime - StartTime);

{set up a dummy variable and begin tracking the time}
Result := 1;
StartTime := timeGetTime;

{perform some function or task using a faster method}
for iCount := 0 to 10000000 do
    Result := Result shl 1;

{record the ending time...}
EndTime := timeGetTime;

{...and display the elapsed time}
Label2.Caption := 'Elapse Time (shl): '+IntToStr(EndTime - StartTime);

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

## Additional Optimization Considerations

Developers are always discovering new tricks and techniques to squeeze a little more performance out of an application. The topics we've covered have only scratched the surface of the myriad of arcane programming constructs that programmers use to improve performance. Beyond the specific optimization techniques we've discussed in this chapter, developers should be aware of general coding practices that should help improve overall performance.

Memory allocation is one of those key areas that can dramatically affect the performance of an application. The task of dynamically allocating memory is an expensive one in terms of time. If possible, you should always dynamically allocate all memory up front, when the application starts. Of course, this is only possible if the total amount of required memory can be determined when it is allocated. If it is impossible to determine the total amount of required memory, try allocating the memory in larger blocks, using up the allocated memory until more is needed. For example, if you're using dynamic arrays, instead of allocating one entry at a time, try allocating 10 entries at a time and using those 10 until more are required. This results in fewer dynamic memory allocations, which in turn results in better performance.

As an alternative to dynamic memory allocation, you may want to try using only global variables. Global variables cause the system to allocate their memory when the application starts; by using only these global variables, you never have to dynamically allocate memory, and thus never incur the cost of such a task. Of course, this may only be applicable if the number of variables required is not too great. Otherwise, the hassle of managing such constructions in terms of coding and code maintenance may outweigh the speed benefits.

Another advantage to using global variables is that they can be used within local functions without creating a stack. In this manner, you could potentially pass hundreds of parameters to a procedure or function without creating and initializing a data structure as discussed above. You have all of the disadvantages associated with using global variables in such a manner, but you may be able to utilize this method in a limited fashion where you need both speed and a large parameter list.

Constant arrays are another area that one should look at. Constant arrays are easy to maintain, but their use incurs an array dereference. If possible, simply use several constants instead. This may be harder to maintain in the long run, but you avoid the array dereference, and ultimately it will improve performance where the

constants are used.

However, where optimization is paramount and speed is critical, one may have to drop down to assembly language. It's hard to beat the speed of hand-optimized assembly. Unfortunately, assembly language, even verbosely documented, is both hard to maintain and hard to modify. It also takes a developer experienced in assembly language to do a decent job. Assembly language can be written right in the Delphi IDE; you simply have to enclose it within an `asm..end` block. An assembly language tutorial is beyond the scope of this book. However, several general Delphi books contain discussions of using assembly language within Delphi, and many good 32-bit assembly language references are available. Fortunately, with the speed of today's machines and Delphi's optimizing compiler, the use of hand-optimized assembly is not quite as necessary as it was just a few years ago.

## Summary

In this chapter, we discussed several techniques for improving the performance of games and applications. We looked at the optimizations that Delphi performs, as well as some Delphi-specific optimization techniques. We also examined some general optimization techniques that are language independent and discussed further optimization considerations. When optimizing your Delphi applications and games, it is important to keep these points in mind:

- The most important step in optimization is knowing where to optimize. After you've determined where the bottlenecks are in application execution, the next step is to evaluate the problem and determine how to optimize the code. Never guess at where your bottlenecks might occur; either use a profiler or some technique for timing code execution.
- Reevaluating an algorithm can have a greater impact on performance than any collection of singular optimization tricks or techniques.
- The task of optimization has a diminishing margin of returns. Spending 10 hours to improve performance by 10% may be worthwhile; spending another 10 hours to improve performance by only 1% is probably a waste of time.
- Optimization should be the last task in application development. Trying to optimize code halfway through the development cycle can be a waste of time, especially if the optimized code gets cut out in favor of another method.
- Never spend time optimizing code that is only called once or twice throughout the entire execution of an application. The most common place to begin optimization is within loops, specifically inner loops that may be performed thousands of times.
- Delphi itself offers a number of very useful optimization options. The compiler itself performs quite a few automatic optimizations, such as register optimizations, call stack elimination, common subexpression removal, the use of loop induction variables, and dead code elimination. Other optional compiler optimizations include aligned record fields, elimination of stack frames, range checking, and overflow checking.
- Exceptions should only be used when an error occurs that must be responded to. Never use exceptions to account for errors (such as dividing by zero), as they are incredibly slow.
- Virtual methods are optimized for speed, and dynamic methods are optimized for size. However, static methods are the fastest as they do not incur a dereference.
- Never use a variant when a standard data type will do. Bear in mind that some standard data types are faster than others for certain operations.
- Avoid using strings as parameters or as return values from functions, as this implicitly creates extra code that slows down the application.
- Loops should be unrolled so as to reduce the number of jumps the application performs.
- Lookup tables should be used for mathematical functions and equations when a finite range of input values is known.
- When multiplying or dividing by a power of 2, using binary bit shifting, as this takes advantage of hardware available in every PC.



Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 11 Special Effects

This chapter covers the following topics:

- Bitmap transformations
- Bitmap scaling
- Bitmap rotation
- Lighting tricks
- Transition effects
- Simple texture mapping
- Star fields
- Fire plasma effects

Although interactive computer games are slowly gaining ground, it is probably safe to say that movies and television are the most popular forms of entertainment among the masses. A good film or movie with Oscar-winning actors and exotic, beautiful sets and locations can be very enjoyable. Indeed, classic films such as *Ben Hur* and *Gone with the Wind* are timeless in their ability to enthrall and entertain.

Everyone enjoys a realistic, stirring performance by a good actor, and of course a good plot can make or break a movie (the same rule applies to games). Even so, when people watch movies, they want to be transported to another place and time, and it would be very difficult to achieve an effective illusion without the use of special effects. What would the *Star Wars* movies have been like without the massive space battles? Or *Star Trek* without the transporter? Special effects provide visualization of events or actions that could not take place in reality, and, in an ironic way, can add an air of “realism” to fantastic settings.

The same holds true for computer games. It is easy to portray exotic locations and creatures using computer graphics. Even so, you can accomplish only so much by just relying on the blitting functions to copy graphics from one surface to another. As we’ve already examined, the Lock method gives us the ability to directly manipulate the contents of a surface. This gives us incredible power, limiting game world visualization only to our imaginations.

This chapter examines how the Lock method can be used to produce effects that would be difficult to reproduce by using Blt or BltFast alone. While a comprehensive discussion of computer graphic special effects could probably fill several volumes, we'll examine a few selected case studies to demonstrate the power of directly manipulating surface contents and to spark our imagination for further applications of this technique.



**Caution:** Always remember to call the Unlock method on a surface before it is used with other DirectDraw methods, such as Flip, or your application will crash.

---

## Bitmap Manipulation

Simply manipulating bitmaps in one way or another can produce many cool effects. The texture mapping that is performed in 3-D games could be considered a bitmap manipulation. Basically, by taking a plain, flat bitmap and drawing it in some way other than just blitting it to the screen in a plain, flat manner can give the bitmap an appearance of depth or texture. This can also stretch the art budget dollar, since producing visual effects programmatically will reduce the amount of artwork required.

There are thousands of ways that bitmaps can be manipulated to produce all kinds of interesting effects. In the interest of space, we'll examine only a small handful. Indeed, even the implementation of these examples can vary widely from programmer to programmer, but we'll examine some simplistic ways to produce some interesting effects by manipulating bitmaps. Specifically, we'll look at bitmap transformations, scaling, rotation, a cheap lighting effect, transition effects, and a very simple texture mapping routine to produce the illusion of a bitmap wrapped around a sphere.

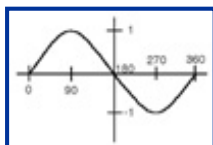
### Bitmap Transformations

This rather broad title addresses techniques for transforming bitmaps and drawing them in a non-linear or non-rectangular manner. You could classify every effect in this section as a bitmap transformation. However, our discussion deals with techniques for using mathematical equations to modify the location of pixels as they are drawn to a destination. Specifically, we're going to look at using the Sin function to apply a sine wave to a bitmap of the Texas state flag in order to produce an animation of a flag blowing in the wind.

### Basic Theory

At the heart of this technique is the Sin function. If you've ever taken a trigonometry class, you should be very familiar with this function. Without going into a lot of detail, the Sin function takes a number, more specifically a degree on a circle (i.e.,  $0^\circ$  to  $360^\circ$ ), and returns a value between -1 and 1. If you graphed out the result of the Sin function, it would look something like Figure 11-1.

This produces what is known as a sine wave. It is a cyclic function that will return a result in the range of -1 to 1 for any value. If we scaled the resulting values from this function and used these as vertical offsets, we could easily produce a simple but effective wave animation with our bitmap.



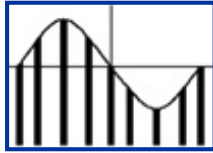
**Figure 11-1:** A graph of the Sin function results



**Tip:** Remember that the trigonometry functions are notoriously slow, so you should probably use an optimization like a lookup table for trig function results in real-world applications.

---

We'll need to draw our bitmap as vertical strips of pixels in order to make this work. Beginning with a starting value of 0, we use the Sin function with this value to determine the vertical offset for the first strip of pixels. We then draw the first vertical strip of pixels within the bitmap at some starting point within our destination, with the top of this strip offset vertically by the return value of the Sin function. Iterating horizontally through our bitmap, we'll add 1 to the value passed to the Sin function as we draw each vertical strip of pixels, using this offset to modify the starting point of the top of the strip. This will produce an oscillating offset for the starting point of the top of each vertical line of pixels, as demonstrated in Figure 11-2.



**Figure 11-2:** Using Sin to modify vertical starting points

If we increment the starting point each time we draw the bitmap, it will make the wave appear to move, and we'll produce an effective animation. Of course, we'll need to scale the values returned from the Sin function by some constant in order to produce a more pronounced wave effect. We'll also need to scale the values given to the Sin function in order to increase the frequency of the waves. The following listing demonstrates this concept.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 [EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 11-1: Producing sine wave patterns with a bitmap*

---

interface

const

.

.

.

{controls wave magnitude}

MAGNITUDE = 10;

{controls wave frequency}

FREQUENCY = 720;

.

.

.

var

.

.

.

{the starting angle}

AngleStart: Integer;

implementation

.

.

.

procedure TfrmDXAppMain.FormCreate(Sender: TObject);

begin

.

.

.

{initialize our starting angle}

AngleStart := 0;

end;

.

.

.

```

{ - this method initializes DirectX and creates all necessary objects - }
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
begin
    .
    .
    .

    {create a palette based on this bitmap}
    FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+
        'texasflag.bmp');

    {attach the palette to the primary surface so that it takes effect}
    DXCheck( FPrimarySurface.SetPalette(FPalette) );

    {load in the bitmap containing the flag image}
    FFlag := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+
        'texasflag.bmp');

    .
    .
    .

end;

    .
    .
    .

procedure TfrmDXAppMain.DrawSurfaces;
var
    DestSrfcInfo, FlagSrfcInfo: TDDSurfaceDesc2;

```

```

DestPtr, FlagPtr: PByteArray;

DestX, DestY, StartY, AngleBegin: Integer;

begin
    {erase the last frame of animation}
    ColorFill(FBackBuffer, 0, nil);

    try
        {lock the flag surface}
        FlagSrfcInfo.dwSize := SizeOf(TDDSurfaceDesc2);
        DXCheck( FFlag.Lock(nil, FlagSrfcInfo, DDLOCK_SURFACEMEMORYPTR or
                    DDLOCK_WAIT, 0) );
        FlagPtr := PByteArray(FlagSrfcInfo.lpSurface);

        {lock the destination surface}

        DestSrfcInfo.dwSize := SizeOf(TDDSurfaceDesc2);
        DXCheck( FBackBuffer.Lock(nil, DestSrfcInfo, DDLOCK_SURFACEMEMORYPTR or
                    DDLOCK_WAIT, 0) );

        {this positions the initial starting point of destination drawing
         so that the flag will be centered horizontally and vertically}
        DestPtr := PByteArray(integer(DestSrfcInfo.lpSurface)+
                                (DestSrfcInfo.lPitch*((DXHEIGHT div 2)-
                                (FlagSrfcInfo.dwHeight div 2)))+
                                ((DXWIDTH div 2)-(FlagSrfcInfo.dwWidth div 2)));

        {increment our starting angle. this produces the motion of the wave,
         and a larger value will make the wave move faster. a negative value
         makes the wave move from left to right; positive values reverse
         this direction. incidentally, we never have to 'roll over' this
         value as Sin will return a value in the range -1 to 1 for any

```



```

argument}

Inc(AngleStart, -5);

{begin iterating through each vertical strip}
for DestX := 0 to FlagSrfcInfo.dwWidth do
begin
    {we want to add 1 to the starting angle for each column in the bitmap}
    AngleBegin := DestX+AngleStart;

    {the FREQUENCY value increases the frequency of the wave across
    the bitmap. with a value of 360, the waves will peak and valley only
    once across the entire length of the bitmap. our frequency value is
    720, which causes 2 peaks and valleys, and makes a nice effect.
    increasing or decreasing this value accordingly may simulate wind
    strength. this value is divided by the width of the bitmap to produce
    an 'angles-per-pixel' ratio so that the effect will spread evenly
    across the horizontal length of the bitmap. this value must be in
    terms of radians instead of degrees, so we must multiply it by
    (PI/180). finally, the MAGNITUDE value scales the results and produces
    offsets, in our example, between -10 and 10. this makes the peaks appear
    higher and the valleys appear lower.}
    StartY:=Trunc((Sin(((FREQUENCY/FlagSrfcInfo.dwWidth)*AngleBegin)*
        (PI/180))*MAGNITUDE));

    {begin drawing the vertical strip of pixels}
    for DestY := 0 to FlagSrfcInfo.dwHeight do
        {the vertical offset is added to the vertical destination value, which
        may move this column of pixels up or down}
        DestPtr[DestSrfcInfo.lPitch*(DestY+StartY)+DestX] :=
            FlagPtr[FlagSrfcInfo.lPitch*DestY+DestX];
    end;
end;

```

```
finally
    {release all locks}
    FFlag.Unlock(nil);
    FBackBuffer.Unlock(nil);
end;
end;
end.
```

---

## Alternatives/Enhancements

The resulting animation from our bitmap transformation, while effective, is a little cheesy. We could produce a more realistic effect by applying another sine wave along the x-axis, perhaps at a 45-degree angle from horizontal. If we scaled the horizontal sine wave offset so that pixels moved left or right very little at the left side of the image, but they moved a large distance at the right side of the image, we could produce a more realistic wind effect. The result would make the flag look like it folded back on itself, and that it was attached to a flagpole.

## Scaling

Sometimes it may be necessary to draw a bitmap image at a size other than its original dimensions. For example, we could create the illusion of a creature getting closer to the user by drawing its image larger and larger as it draws nearer. By scaling an image in this manner, we can produce the illusion of depth, or even provide zooming functionality for things like sniper scopes.

## Basic Theory

Scaling a bitmap involves drawing the pixels of the source bitmap into the destination according to a specific ratio. This ratio determines the final size of the bitmap as it appears in the destination. The ratio is expressed as a fraction, or a floating-point number; the larger the number, the larger the bitmap, and vice versa. For example, a ratio of 2 indicates that the final bitmap will be twice as large as its original size, whereas a ratio of 0.5 indicates that the final bitmap will be half of its original size.

Specifically, the ratio indicates how each pixel is mapped into the destination. If we iterate through the pixels in the destination and divide their coordinates by the scaling factor, we can determine which pixel to select from the source. This is accomplished by the following equations:

```
Source_X := Trunc(Destination_X / Scale);
Source_Y := Trunc(Destination_Y / Scale);
```

A scaling factor less than 1 causes pixels in the source image to be skipped, thus resulting in a smaller final image. Similarly, a scaling factor greater than 1 causes pixels in the source image to be replicated, resulting in a larger final image. The following listing demonstrates this technique.

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**ITKNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 11-2: Scaling a bitmap image*

---

interface

.

.

.

var

.

.

.

{this is used as an argument to the Sin function to  
create an oscillating scale factor}

Angle: Integer;

```
implementation
```

```
.  
.br/>.
```

```
procedure TfrmDXAppMain.FormCreate(Sender: TObject);
```

```
begin
```

```
.  
.br/>.
```

```
{initialize the angle variable}
```

```
Angle := 0;
```

```
end;
```

```
.  
.br/>.
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
.  
.br/>.
```

```
{create a palette based on this bitmap}
```

```
FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+
```

```
  'Athena.bmp');
```

```
{attach the palette to the primary surface so that it takes effect}
```

```
DXCheck( FPrimarySurface.SetPalette(FPalette) );
```

```
{load in the bitmap containing the image}
```

```
FImages := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+  
                        'Athena.bmp');
```

```
.
```

```
.
```

```
.
```

```
end;
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
    SrcfInfo, ImageInfo: TDDSurfaceDesc2;
```

```
    SrcPtr, DestPtr: PByteArray;
```

```
    DestX, DestY, SrcX, SrcY: Integer;
```

```
    Scale: Double;
```

```
begin
```

```
    {erase the last frame of animation}
```

```
    ColorFill(FBackBuffer, 0, nil);
```

```
    {increment our 'angle' variable}
```

```
    Inc(Angle);
```

```

{using the angle variable with the Sin function to create a
  cyclic scaling value, compute a scale that oscillates between 0.5 and 1.5}
Scale := 1+(Sin((Angle)*(PI/180))*0.5);

try
  {lock the image surface}
  ImageInfo.dwSize := SizeOf(TDDSurfaceDesc2);
  DXCheck( FImages.Lock(nil, ImageInfo, DDLOCK_SURFACEMEMORYPTR or
                    DDLOCK_WAIT, 0) );
  SrcPtr := PByteArray(ImageInfo.lpSurface);

  {lock the destination surface}
  SrfcInfo.dwSize := SizeOf(TDDSurfaceDesc2);
  DXCheck( FBackBuffer.Lock(nil, SrfcInfo, DDLOCK_SURFACEMEMORYPTR or
                    DDLOCK_WAIT, 0) );
  DestPtr := PByteArray(Integer(SrfcInfo.lpSurface)+100*SrfcInfo.lPitch+100);

  {iterate through the destination pixels, with a maximum scale of 1.5}
  for DestY := 0 to Trunc(ImageInfo.dwHeight * 1.5) do
  begin
    for DestX := 0 to Trunc(ImageInfo.dwWidth * 1.5) do
    begin
      {compute the source pixels based on the scale and our current
        destination pixel}
      SrcY := Trunc(DestY / Scale);
      SrcX := Trunc(DestX / Scale);

      {if the source pixel coordinate is within the source image...}
      if (SrcX > 0) and (SrcX < ImageInfo.dwWidth) and
        (SrcY > 0) and (SrcY < ImageInfo.dwHeight) then

```

```

        {...copy this source pixel into the destination}

        DestPtr[DestX] := SrcPtr[(ImageInfo.lPitch*SrcY)+SrcX];

    end;

    {increment to the next line in the destination surface}

    DestPtr := PByteArray(Longint(DestPtr)+SrcInfo.lPitch);

    end;

finally

    {unlock both buffers}

    FImages.Unlock(nil);

    FBackBuffer.Unlock(nil);

    end;

end;

```

---

## Alternatives/Enhancements

Scaling is implemented by DirectDraw; it is emulated by the HEL, and some video cards provide hardware support for scaling. Thus, you do not really need this algorithm to scale a bitmap. However, when you use DirectDraw to perform scaling, you are relying on the implementation present in the current driver or hardware of the end user's machine. This could potentially lead to differing results from system to system. Using our own techniques, we can control the scaling, ensuring a consistent result on every machine. We could also employ interpolation techniques to determine the color of a scaled pixel from those around it. The resulting image could be of a much higher quality than that capable from DirectDraw, as it will not have the jagged edges produced by the DirectDraw scaling algorithms.



**Tip:** To let DirectDraw perform scaling, use the Blt function and specify the desired size for the destination rectangle.

---

As with most of these bitmap manipulation effects, scaling a bitmap can be very computationally intensive. We could pre-compute the scaled pixel positions relative to an arbitrary starting point for the entire range of scaling values when the application starts. The resulting lookup table would be very fast, but it would probably take up loads and loads of memory, depending on the range of scaling values. It might be possible to provide several bitmaps at specific sizes and scale between them as needed. This might allow us to optimize the scaling routine, but it would require more storage space for bitmaps. However, this would allow us to improve the quality of the images as they get larger.

## Rotation

Most top-down, 2-D, sprite-based games feature sprite images that seem to be able to rotate in any direction. This could be accomplished by drawing the sprite image in all possible rotations, but this would be required for all frames of animation and would result in a large number of images for even one sprite. While this is a perfectly acceptable way of implementing rotation, we could use bitmap rotation equations to allow us to rotate images to any arbitrary angle while saving on memory space.



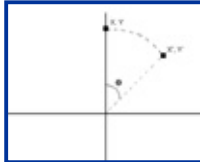
## Basic Theory

You may remember from trigonometry that it is mathematically possible to rotate a point about the origin by an arbitrary angle. Given an original x and y coordinate and an arbitrary angle, a point can be rotated using the following equations:

```
New_X := Original_X * Cos(Angle) - Original_Y * Sin(Angle);
```

```
New_Y := Original_X * Sin(Angle) + Original_Y * Cos(Angle);
```

This new point is rotated about the origin by the specified angle, as illustrated below.



**Figure 11-3:** Rotating a point



**Caution:** A common mistake made when implementing these equations is to rotate the x coordinate, and then use this newly rotated value in the rotation equations for the y coordinate. The result will be a highly skewed image which will eventually shrink into nothing if rotations continue. This is why you must store the rotated values into separate variables.

If we treat each pixel in a bitmap as a point, we can use these equations to rotate the pixels in the bitmap. Now, most bitmap rotation algorithms map the pixels from the source bitmap to the destination bitmap, but most of the time, this results in a rotated image that contains holes. Therefore, we should iterate through the pixels in our destination image, determining the pixels we should draw from the source image. If we determine the center of the source image, like so:

```
CenterX := (SourceImageWidth / 2);
```

```
CenterY := (SourceImageHeight / 2);
```

then we can determine which pixels to pull from the source image based on the pixel being drawn in the destination image using these equations:

```
SrcX := Trunc(CenterX+(DestX-CenterX)*Cos(Angle)-(DestY-CenterY)*Sin(Angle));
```

```
SrcY := Trunc(CenterY+(DestX-CenterX)*Sin(Angle)+(DestY-CenterY)*Cos(Angle));
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**IT KNOWLEDGE.COM**<sup>SM</sup>  
Need IT. Find IT. Know IT.

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

This will result in a perfectly rotated image with no holes.

Incidentally, we should load our source image into system memory as opposed to video memory. This will speed up the rotation, as it is much faster to access a surface in system memory than video memory. Also, we'll further optimize this by creating a sine and cosine lookup table instead of calling the Sin and Cos functions, using the angle as the index within these tables. The following listing demonstrates this technique.

*Listing 11-3: Rotating a bitmap*

---

```
interface
```

```
    .
```

```
    .
```

```
    .
```

```
var
```

```
    .
```

```
    .
```

```
    .
```

```
{the arbitrary angle}
```

```
Theta: Integer;
```

```
{these are used to create sine and cosine function lookup tables  
as an optimization}
```

```
iAngle: Integer;
```

```
SinTable: array[0..359] of single;
```

```
CosTable: array[0..359] of single;
```

```
implementation
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.FormCreate(Sender: TObject);
```

```
begin
```

```
.
```

```
.
```

```
.
```

```
{initialize our starting angle}
```

```
Theta := 0;
```

```
end;
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
begin
    .
    .
    .

    {load the palette from this bitmap}
    FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+
        'athena.bmp');

    {set the palette}
    FPrimarySurface.SetPalette(FPalette);

    {load the bitmap to be rotated}
    FImage := DDLoadBitmapSysMem(FDirectDraw, ExtractFilePath(ParamStr(0))+
        'athena.bmp');

    .
    .
    .

end;

    .
    .
    .

procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
    SrcInfo, DestInfo: TDDSurfaceDesc2;

    SrcPtr, DestPtr: PByteArray;

    DestX, DestY,
    SrcX, SrcY: Integer;

    SinTheta, CosTheta: Single;

    Area: TRect;

    CenterX, CenterY: Single;
begin
    {erase the last frame of animation}
    ColorFill(FBackBuffer, 0, nil);

    {increment our angle, making sure to roll it over if necessary}
    Inc(Theta);
    if Theta>359 then
        Theta := 0;

    {determine the sine and cosine of this angle by using our
    lookup tables}
    SinTheta := SinTable[Theta];
    CosTheta := CosTable[Theta];

    try
        {lock the source image}
        SrcInfo.dwSize := SizeOf(TDDSurfaceDesc2);
```

```

DXCheck( FImage.Lock(nil, SrcInfo, DDLOCK_SURFACEMEMORYPTR or
                DDLOCK_WAIT, 0) );

SrcPtr := PByteArray(SrcInfo.lpSurface);

{specify the rectangular area in the destination image to lock by
determining coordinates that will center the image}
Area := Rect((DXWIDTH div 2)-(SrcInfo.dwWidth div 2),
            (DXHEIGHT div 2)-(SrcInfo.dwHeight div 2),
            (DXWIDTH div 2)+(SrcInfo.dwWidth div 2),
            (DXHEIGHT div 2)+(SrcInfo.dwHeight div 2));

{lock the destination surface}
DestInfo.dwSize := SizeOf(TDDSurfaceDesc2);
DXCheck( FBackBuffer.Lock(@Area, DestInfo, DDLOCK_SURFACEMEMORYPTR or
                DDLOCK_WAIT, 0) );

DestPtr := PByteArray(DestInfo.lpSurface);

{determine the center of the source image}
CenterX := (SrcInfo.dwWidth / 2);
CenterY := (SrcInfo.dwHeight / 2);

{begin iterating through the pixels of the destination image}
for DestY := 0 to SrcInfo.dwHeight do
begin
    for DestX := 0 to SrcInfo.dwWidth do
    begin
        {determine the source pixel to use for this destination pixel}
        SrcX := Trunc(CenterX + (DestX - CenterX)*CosTheta -
                    (DestY - CenterY)*SinTheta);
        SrcY := Trunc(CenterY + (DestX - CenterX)*SinTheta +

```

```
(DestY - CenterY)*CosTheta);
```

```
{if this pixel is within the source image...}
```

```
if (SrcX > 0) and (SrcX < SrcInfo.dwWidth) and
```

```
(SrcY > 0) and (SrcY < SrcInfo.dwHeight) then
```

```
{...copy it to the destination}
```

```
DestPtr[DestX] := SrcPtr[SrcY*SrcInfo.lPitch+SrcX];
```

```
end;
```

```
{move to the next line in the destination image}
```

```
DestPtr := PByteArray(Longint(DestPtr)+DestInfo.lPitch);
```

```
end;
```

```
finally
```

```
{clean up}
```

```
FImage.Unlock(nil);
```

```
FBackBuffer.Unlock(@Area);
```

```
end;
```

```
end;
```

```
.
```

```
.
```

```
.
```

```
initialization
```

```
{we'll create lookup tables for the sine and cosine functions, using
```

```
the angle as an index into these tables}
```

```
for iAngle := 0 to 359 do
```

```
begin
```

```
SinTable[iAngle] := Sin(iAngle*(PI/180));
```

```
CosTable[iAngle] := Cos(iAngle*(PI/180));
```

```
end;
```

---

## Alternatives/Enhancements

Unlike bitmap scaling, rotation is not something that is emulated by DirectDraw. Some cards provide hardware support for bitmap rotation, but even these could be restricted to rotations by only 90-degree angles. Therefore, you cannot rely on DirectDraw for bitmap rotation on every machine; you must implement this functionality yourself.

Bitmap rotation can be very slow due to its mathematically intense nature. However, rotation could be optimized by the use of assembly language. A less dramatic approach may be to use lookup tables that contained the rotated pixel coordinates for all angles. The angle could then be used as an index into this lookup table. This would result in incredibly fast bitmap rotation, but it would be a memory hog.

## Lighting Tricks

In nature, the color of an object can vary according to the intensity of light to which it is subjected. When the lights go down, things appear darker; when an object moves into the shadow of another object, its color darkens. Modeling this in our graphics engines can make our game worlds appear much more realistic and dynamic. There are many ways to model realistic lighting, but one method in particular is easy to implement and produces acceptable results.

## Basic Theory

This technique works by imposing restrictions on the palette of an image. For example, say we want to display an image that will have four different levels of light intensity. We need to segregate our palette into four different segments, so  $256 / 4 = 64$  palette slots per segment.

Now, our artists need to draw the image using only the first 64 palette slots. These slots should contain the colors of the image at their brightest intensity. The other three segments of 64 palette slots contain exactly the same colors as those in the first 64 slots, but at darker and darker intensities. In other words, the second segment of 64 slots contains the colors of the first 64 slots at 75% of their original intensity, the third segment of 64 slots contains these colors at 50% intensity, and the final segment contains the colors at 25% of their original intensity.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

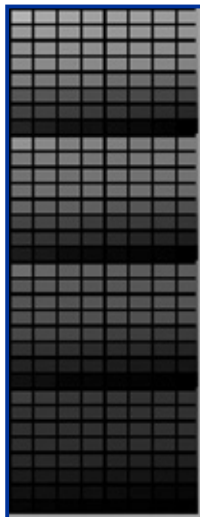
Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

When the application draws the image, it needs to draw it one pixel at a time. Then, depending on the light level, we simply add 0, 64, 128, or 192 to the color of the original pixel. This will draw the pixel using the colors lower and lower in the palette, resulting in an image with darker and darker pixels. Using this technique, we can create a spotlight effect as demonstrated in Listing 11-4.



**Figure 11-4:** A segregated palette

*Listing 11-4: Using lighting techniques to create a spotlight*

---

```
procedure TfrmDXAppMain.FormCreate(Sender: TObject);  
  
begin
```

```
.  
.   
  
{position the mouse cursor to appear in the center of the screen}
```

```
CurX := 320;
```

```
CurY := 240;
```

```
OldX := 320;
```

```
OldY := 240;
```

```
.  
.   
.   

```

```
end;
```

```
.  
.   
.   

```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
.  
.   
.   

```

```
{load the palette for the image}
```

```
FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+  
                        'image.bmp');
```

```
{set the palette}
```

```
FPrimarySurface.SetPalette(FPalette);
```

```

{finally, load the image}

Image := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+'image.bmp');

{erase the background}

ColorFill(FBackBuffer, 255, nil);

ColorFill(FPrimarySurface, 255, nil);

.

.

.

end;

.

.

.

procedure TfrmDXAppMain.DrawSurfaces;

var

    DestSrfcInfo, ImageSrfcInfo: TDDSurfaceDesc2;

    DestPtr, ImagePtr: PByteArray;

    ImgStartX, ImgStartY,

    PointX, PointY,

    Distance: Integer;

    iCount, iCount2: Integer;

begin
    try
        {lock the image surface}

```

```

ImageSrcfInfo.dwSize := SizeOf(TDDSurfaceDesc2);

DXCheck( Image.Lock(nil, ImageSrcfInfo, DDLOCK_SURFACEMEMORYPTR or
          DDLOCK_WAIT, 0) );

ImagePtr := PByteArray(ImageSrcfInfo.lpSurface);

{lock the destination surface}
DestSrcfInfo.dwSize := SizeOf(TDDSurfaceDesc2);
DXCheck( FBackBuffer.Lock(nil, DestSrcfInfo, DDLOCK_SURFACEMEMORYPTR or
          DDLOCK_WAIT, 0) );

{this positions the initial starting point of destination drawing
 so that the image will be centered horizontally and vertically}
DestPtr := PByteArray(integer(DestSrcfInfo.lpSurface)+
                      (DestSrcfInfo.lPitch*((DXHEIGHT div 2)-
                      (ImageSrcfInfo.dwHeight div 2)))+
                      ((DXWIDTH div 2)-(ImageSrcfInfo.dwWidth div 2)));

{store the starting offset of the image}
ImgStartX := (DXWIDTH div 2)-(ImageSrcfInfo.dwWidth div 2);
ImgStartY := (DXHEIGHT div 2)-(ImageSrcfInfo.dwHeight div 2);

for iCount := 0 to ImageSrcfInfo.dwHeight-1 do
begin
  for iCount2 := 0 to ImageSrcfInfo.dwWidth-1 do
  begin
    {retrieve the current pixel coordinates and translate them such
     that the mouse cursor acts as the origin}
    PointX := (ImgStartX+iCount2) - CurX;
    PointY := (ImgStartY+iCount) - CurY;

    {determine the relative distance of this pixel from the mouse
     cursor, and restrain it to a range of 0-4}

```

```

Distance := ((PointX*PointX)+(PointY*PointY)) div 600;

if Distance > 3 then

    Distance := 3;

    {draw the destination pixel. the distance is used to determine
    which 64-slot segment of the palette is used to draw the
    destination pixel color}

    DestPtr[iCount2] := ImagePtr[iCount2]+(64*Distance);

end;

{increment to the next line in the image}

ImagePtr := PByteArray(Integer(ImagePtr)+ImageSrcInfo.lPitch);

{increment to the next line in the destination buffer}

DestPtr := PByteArray(Integer(DestPtr)+DestSrcInfo.lPitch);

end;

finally

    {release all locks}

    Image.Unlock(nil);

    FBackBuffer.Unlock(nil);

end;

end;

```

---

## Alternatives/Enhancements

This is a really cheap trick that is easy to implement and does not put too great a strain on your artists. However, another method for implementing lighting involves creating a lookup table for each level of light intensity desired. Specifically, the application creates a 256-entry array of integers, and begins iterating through each entry in the original palette. For each color, the application determines the color components at a darker intensity, and then searches the original palette for an entry that is the closest match for this darker color. The number of this matching palette entry is then placed into the array at the index of the original color. The pixels in the bitmap now act as an index into the new array, which in turn then act as an index into the original palette. It is possible to create many more light intensity levels using this technique as opposed to the one described above. It also allows your artists to use more colors, but they must be mindful to use a palette containing several shades of each individual color so that the color-matching step finds an appropriate match.

Alternatively, you could use a non-palettized video mode. Using a non- palettized color mode frees your artists to use whatever colors they wish, and the results of this trick can be much more spectacular than that shown here. The effect will take a little longer, depending on the chosen video mode, as the application must move double to quadruple the information. However, using this technique, the banding side effect of the technique shown above can be eliminated, and the lighting effect can be much more realistic.



**Tip:** It is possible to lessen the banding effect using the above illustrated technique by using a mask for the lighted area instead of programmatically calculating the distance from the mouse cursor. The pixels in each band could be dithered somewhat around the edges of each band to help lessen the banding effect.

---

## Transition Effects

While it is perfectly acceptable to simply move from one scene to another with no fanfare, providing some sort of animation during scene transitions makes the software appear to have a smoother flow. We can take another lesson from the motion picture industry here. Typically, movies use a number of different wipes, fades, and other effects to smoothly transition from one scene to another. The next time you watch any of the *Star Wars* films, pay attention to how these transition effects create a smooth flow throughout the movie.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Basic Theory**

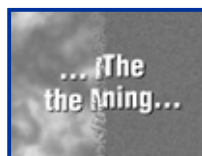
Each transition effect may be implemented in wildly differing ways or with only subtle, minor differences, depending on the desired effect. In this example, let's use three simple effects that are relatively easy to implement yet provide cool animation effects. Specifically, these transition effects will be melt, crawl, and dissolve.

**Melt** This is the classic melting screen effect made popular by id software's Doom. It works by first drawing the next scene image to the back buffer, and then drawing the closing scene one column at a time into the back buffer. An array of integers, one entry per column of pixels, is used to track the vertical offset of the start of the pixel column. By starting some columns before others, we can create the illusion of the foreground screen melting away to reveal the next scene.



**Figure 11-5:** The melt effect

**Crawl** This is a variation of the dissolve effect below. Starting from one side of the screen, we use two variables to track a beginning and ending column (or row), spaced some arbitrary distance apart. At each frame of the animation, we increment both the beginning and ending columns (or rows). These define an area into which several pixels from the next scene are drawn. Their coordinates are chosen at random and copied into the back buffer at the same position. Past the ending column (or row), the "exposed" portion of the next scene is drawn. The effect appears to eat away at the foreground image, revealing the image of the next scene underneath.



**Figure 11-6:** The crawl effect

**Dissolve** This is a common effect that makes one image appear to dissolve, revealing the image beneath it. It works by randomly choosing a pixel from the next image and copying it to the back buffer (over the previous image) at the same position. However, it is rather slow to use the entire screen area for the range of random pixel selection. Thus,

we constrain the random pixel selection to an  $8 \times 8$  pixel area, and then replicate that throughout the entire image. In other words, the screen is divided into  $8 \times 8$  blocks. The next pixel is chosen from the first  $8 \times 8$  area, and we use this coordinate as an offset for each  $8 \times 8$  block throughout the screen. This creates a fast, effective dissolve effect.



**Figure 11-7:** The dissolve effect

The following listing demonstrates a simple implementation of these three effects. Press the spacebar to move from one image to the next.

*Listing 11-5: Transition effects*

---

```
interface
    .
    .
    .

type
    TTransitionType = (transMelt, transCrawl, transDissolve);

    .
    .
    .

var

    .
    .
    .

    {holds the scene images}

    ScreenBackgrounds: array[0..2] of IDirectDrawSurface4;
```



```
{tracks which background image is used next}
```

```
NextBackground: Integer;
```

```
{tracks the next transition effect to be used}
```

```
NextTransition: TTransitionType;
```

```
implementation
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.FormCreate(Sender: TObject);
```

```
begin
```

```
.
```

```
.
```

```
.
```

```
{initialize the next background}
```

```
NextBackground := 0;
```

```
{initialize the next transition effect}
```

```
NextTransition := transMelt;
```

```
NextTransition := transMelt;
```

```
end;
```

```
.
```

```
.
```

```
.
```

```

procedure TfrmDXAppMain.FormActivate(Sender: TObject);

begin

    .
    .
    .

{load the palette}

    FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+
                                'begin.bmp');

{set the palette}

    FPrimarySurface.SetPalette(FPalette);

{load the three images to be used in the transition effect animations}

    ScreenBackgrounds[0] := DDLoadBitmap(FDirectDraw, ExtractFilePath(
                                        ParamStr(0))+ 'begin.bmp');
    ScreenBackgrounds[1] := DDLoadBitmap(FDirectDraw, ExtractFilePath(
                                        ParamStr(0))+ 'middle.bmp');
    ScreenBackgrounds[2] := DDLoadBitmap(FDirectDraw, ExtractFilePath(
                                        ParamStr(0))+ 'end.bmp');

{copy the first image to the screen}

    DestRect := Rect(0, 0, DXWIDTH, DXHEIGHT);

    FPrimarySurface.BltFast(0, 0, ScreenBackgrounds[0], DestRect,
                            DDBLTFAST_NOCOLORKEY OR DDBLTFAST_WAIT);

    FBackBuffer.BltFast(0, 0, ScreenBackgrounds[0], DestRect,
                        DDBLTFAST_NOCOLORKEY OR DDBLTFAST_WAIT);

    .
    .
    .

```

```
end;
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.FormKeyDown(Sender: TObject; var Key: Word;
```

```
    Shift: TShiftState);
```

```
begin
```

```
    if Key = VK_ESCAPE then
```

```
        Close;
```

```
    {if the spacebar was pressed...}
```

```
    if Key = Ord(' ') then
```

```
    begin
```

```
        {move to the next transition}
```

```
        NextTransition := Succ(NextTransition);
```

```
        {rollover the transition effect if necessary}
```

```
        if Ord(NextTransition) > Ord(transDissolve) then
```

```
            NextTransition := transMelt;
```

```
        {perform the transition effect}
```

```
        CoolScreenTransition(NextTransition);
```

```
    end;
```

```
end;
```

```
procedure TfrmDXAppMain.CoolScreenTransition(TransitionType: TTransitionType);
```

```
var
```

```
    HeadCol, TailCol: Integer;
```

```

iCount, iCount2, iCount3: Integer;

Block, DestRect: TRect;

RandBlock, RandBlock2: Integer;

Worms: array[0..DXWIDTH-1] of Integer;

StillGoing: Boolean;

begin

  {move to the next image}

  Inc(NextBackground);

  {rollover the image index, if necessary}

  if NextBackground > 2 then

    NextBackground := 0;

  {copy the foreground image into the back buffer}

  DestRect := Rect(0, 0, DXWIDTH, DXHEIGHT);

  FBackbuffer.BltFast(0, 0, FPrimarySurface, DestRect, DDBLTFAST_NOCOLORKEY or

    DDBLTFAST_WAIT);

  case TransitionType of

    transMelt   : begin

      {initialize the starting offsets for each column of

        pixels}

      for iCount := 0 to DXWIDTH-1 do

        Worms[iCount] := 0;

      {initialize the loop variable}

      StillGoing := TRUE;

      {begin the effect}

      while StillGoing do

        begin

```

```

{assume that the effect will stop}

StillGoing := FALSE;

{for each column of pixels...}
for iCount := 0 to DXWIDTH-1 do
begin
    {if this column has not yet started to move...}
    if Worms[iCount] = 0 then
        {...give the column a 25% chance of starting}
        if Random(4) = 1 then
            Worms[iCount] := 1;

    {if this column has not yet reached to bottom,
    but is in motion...}
    if (Worms[iCount]<DXHEIGHT-1)and(Worms[iCount]>0) then
begin
    {...there is still at least one column still
    'alive', so the effect has not yet stopped}
    StillGoing := TRUE;

    {copy this column of pixels into its new position}
    Block := Rect(iCount, Worms[iCount], iCount+1,
                DXHEIGHT-3);
    FBackBuffer.BltFast(iCount, Worms[iCount]+3,
                        FBackBuffer, Block,
                        DDBLTFAST_NOCOLORKEY or
                        DDBLTFAST_WAIT);

    {copy the same column from the next scene above
    the currently moving column}
    Block := Rect(iCount, 0, iCount+1, Worms[iCount]+3);

```

```

        FBackBuffer.BltFast(iCount, 0,
                               ScreenBackgrounds[NextBackground],
                               Block, DDBLTFAST_NOCOLORKEY or
                               DDBLTFAST_WAIT);

        {move the column down a bit}
        Worms[iCount] := Worms[iCount]+3;
    end;
end;

{the back buffer now contains the next frame of the
 animation, so copy it to the screen}
DestRect := Rect(0, 0, DXWIDTH, DXHEIGHT);
FPrimarySurface.BltFast(0, 0, FBackbuffer, DestRect,
                        DDBLTFAST_NOCOLORKEY or
                        DDBLTFAST_WAIT);

end;

{the animation is over, so copy the final image to both
 the front and back buffers to erase the last remains of
 the animation sequence}
Block := Rect(0, 0, DXWIDTH, DXHEIGHT);
FBackbuffer.BltFast(0, 0,
                    ScreenBackgrounds[NextBackground],
                    Block, DDBLTFAST_NOCOLORKEY or
                    DDBLTFAST_WAIT);

FPrimarySurface.BltFast(0, 0,
                        ScreenBackgrounds[NextBackground],
                        Block, DDBLTFAST_NOCOLORKEY or
                        DDBLTFAST_WAIT);

end;

transCrawl : begin

```

```

{begin tracking the beginning and ending columns}

HeadCol := 0;

TailCol := -10;

{while the ending column is still on the screen...}
while TailCol < (DXWIDTH div 5)-1 do
begin
    {iterate through the columns of pixels between the
    beginning and ending column}
    for iCount := TailCol to HeadCol do
        {if this column is on the screen...}
        if iCount > -1 then
            for iCount2 := 0 to 20 do
                begin
                    {choose a random block of pixels}
                    RandBlock := Random(DXHEIGHT div 5);

                    {copy this block of pixels from the next image
                    into the destination at the same coordinates}
                    Block := Rect(5*iCount, 5*RandBlock, 5*iCount+5,
                                5*RandBlock+5);

                    FBackBuffer.BltFast(Block.Left, Block.Top,
                                        ScreenBackgrounds[NextBackground],
                                        Block, DDBLTFAST_NOCOLORKEY or
                                        DDBLTFAST_WAIT);

                end;

            {if the ending column is on the screen...}
            if TailCol > -1 then
                begin
                    {make sure we copy the entire column of pixels from

```

```

        the next image into the destination so that the
        final image will be complete}

Block := Rect(5*TailCol, 0, 5*TailCol+5, DXHEIGHT);

FBackBuffer.BltFast(Block.Left, Block.Top,

                    ScreenBackgrounds[NextBackground],

                    Block, DDBLTFAST_NOCOLORKEY or
                    DDBLTFAST_WAIT);

end;

{increment the beginning and ending columns}

Inc(HeadCol);

Inc(TailCol);

{copy this frame of animation to the screen}

Block := Rect(0, 0, DXWIDTH, DXHEIGHT);

FPrimarySurface.BltFast(0, 0, FBackbuffer, Block,

                        DDBLTFAST_NOCOLORKEY or
                        DDBLTFAST_WAIT);

{pause for a short period}

Sleep(20);

end;

{the animation is over, so copy the final image to both
the front and back buffers to erase the last remains of
the animation sequence}

Block := Rect(0, 0, DXWIDTH, DXHEIGHT);

FBackbuffer.BltFast(0, 0,

                    ScreenBackgrounds[NextBackground],

                    Block, DDBLTFAST_NOCOLORKEY or
                    DDBLTFAST_WAIT);

FPrimarySurface.BltFast(0, 0,

```



```

        ScreenBackgrounds[NextBackground],
        Block, DDBLTFAST_NOCOLORKEY or
        DDBLTFAST_WAIT);

    end;

transDissolve : begin

    {we will copy 100 blocks of pixels for this effect}
    for iCount := 0 to 100 do
    begin
        {choose a random block of pixels from our 8x8 area}
        RandBlock := Random(8);
        RandBlock2 := Random(8);

        {this will be mirrored throughout the screen relative
        to each segmented 8x8 block of pixels}
        for iCount2 := 0 to 15 do
            for iCount3 := 0 to 12 do
                begin
                    {copy the relative block of pixels from the
                    next image into the back buffer}
                    Block := Rect((5*RandBlock)+(iCount2*8*5),
                                (5*RandBlock2)+(iCount3*8*5),
                                (5*RandBlock)+(iCount2*8*5)+5,
                                (5*RandBlock2)+(iCount3*8*5)+5);

                    FBackBuffer.BltFast(Block.Left, Block.Top,
                                        ScreenBackgrounds[NextBackground],
                                        Block, DDBLTFAST_NOCOLORKEY or
                                        DDBLTFAST_WAIT);

                end;

                {copy this frame of animation to the screen}
                Block := Rect(0, 0, DXWIDTH, DXHEIGHT);
            end;
        end;
    end;

```

```
FPrimarySurface.BltFast(0, 0, FBackbuffer, Block,  
                        DDBLTFAST_NOCOLORKEY or  
                        DDBLTFAST_WAIT);
```

```
{pause for a short period}
```

```
Sleep(20);
```

```
end;
```

```
{the animation is over, so copy the final image to both  
the front and back buffers to erase the last remains of  
the animation sequence}
```

```
Block := Rect(0, 0, DXWIDTH, DXHEIGHT);
```

```
FBackbuffer.BltFast(0, 0,  
                    ScreenBackgrounds[NextBackground],  
                    Block, DDBLTFAST_NOCOLORKEY or  
                    DDBLTFAST_WAIT);
```

```
FPrimarySurface.BltFast(0, 0,  
                        ScreenBackgrounds[NextBackground],  
                        Block, DDBLTFAST_NOCOLORKEY or  
                        DDBLTFAST_WAIT);
```

```
end;
```

```
end;
```

```
end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)

[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX**

**7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Alternatives/Enhancements**

The only thing limiting transition effects from one scene to another is the imagination of the programmer. While the three covered here are easy to implement, you can probably think of several dozen more that could be implemented just as easily. Many transition effects available from third-party products are little more than extensions of the effects implemented in the above example.

The effects demonstrated above rely on the fact that the images involved in the transition have the same palette. Using some palette manipulation, it may be possible to use images with two different palettes. This would require a new palette to be created for each frame of the transition animation, but this would result in a more flexible scene transitioning system.



**Tip:** Using a non-palettized display mode would, of course, eliminate this drawback.

**Simple Texture Mapping**

We've seen how easy it is to blit rectangular sections of bitmaps onto the screen or into offscreen buffers. If you've done any basic Windows graphics programming, you also know how easy it is to draw geometric shapes. It would be cool if we could somehow draw geometric shapes but use a bitmap as the contents of that shape, as opposed to simply filling it with color. The technique of mapping a bitmap image onto a geometric shape is known as texture mapping.



**texture mapping:** A general term for algorithms that map a bitmap image onto a geometric shape.

Texture mapping techniques can be applied in many exciting ways. Texture mapping techniques allow us to produce very convincing graphical images of real-world objects, which would be very difficult or impossible to

achieve otherwise. Algorithms that implement texture mapping range from the childishly simple to the absurdly complex. Let's look at a relatively simple texture mapping method that will allow us to wrap a bitmap around a circular shape to produce the illusion of a sphere.

## Basic Theory

Perhaps the first thing we will need is the texture to be mapped onto the shape. In this example, we'll create an animation of a planet spinning in space. Thus, our texture map should look like a planet, and will be rectangular in shape.

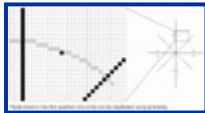


**Figure 11-8:** The texture map

Next, we need to define the circular shape itself. Specifically, we'll be using some of the scaling techniques described previously, so we need a series of points that define the perimeter of the circle. We can generate these points by using the equation for a circle:

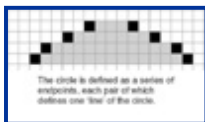
```
MinorAxis := Sqrt((Radius * Radius)-(MajorAxis * MajorAxis))
```

The circle is generated at application startup and therefore does not have to be particularly fast. However, we'll make use of the symmetrical property of circles to generate points only for the first quadrant. These points can then be mirrored around the horizontal, vertical, and diagonal axes to form an entire circle, as illustrated in Figure 11-9.



**Figure 11-9:** Using symmetry to mirror points around a circle

Since our circular shape will be drawn one line of pixels at a time, the generated points serve as endpoints for each line of the circle. The actual texture mapping occurs by using the scaling techniques described above in conjunction with these endpoints to stretch each line of the texture bitmap between the endpoints of each line in the circle.



**Figure 11-10:** Defining a circle using endpoints

By defining a rectangular area smaller than the texture map, and then using scaling techniques to stretch the defined rectangular area not only vertically across the circular shape but also horizontally between each pair of endpoints, the resulting image seems to warp and magnify at the center of the circle. This produces a convincing illusion of an image wrapped around a sphere. The following listing demonstrates this technique.

**Listing 11-6:** *Wrapping a texture around a sphere*

---

```
interface
```

```
const
```

```
.  
.br/>.
```

```
{the position of the center of the screen}
```

```
XPos = 320;
```

```
YPos = 240;
```

```
.br/>.br/>.
```

```
var
```

```
.br/>.br/>.
```

```
{tracks our endpoints}
```

```
FEndptList: TList;
```

```
{tracks the width of the portion of the texture to be wrapped  
around the sphere}
```

```
FTextureSwatchWidth: Integer;
```

```
{tracks the texture offset}
```

```
TextureOffset: Integer;
```

```
.
```

```
.  
.br/>  
@code =  
  
procedure TfrmDXAppMain.FormDestroy(Sender: TObject);  
  
begin  
  
.  
.  
.  
  
    {free the endpoint list}  
  
    FEndptList.Free;  
  
end;  
  
  
  
procedure TfrmDXAppMain.FormActivate(Sender: TObject);  
  
begin  
  
.  
.  
.  
  
    {load the palette}  
  
    FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+  
                                'Texture.bmp');  
  
    {set the palette}  
  
    FPrimarySurface.SetPalette(FPalette);  
  
    {create the circle}  
  
    GenerateCircle(100, ExtractFilePath(ParamStr(0))+'Texture.bmp');
```

```

{initialize the texture offset}
TextureOffset := 0;

{erase the screen}
ColorFill(FPrimarySurface, 255, nil);
ColorFill(FBackBuffer, 255, nil);

.
.
.

end;

procedure TfrmDXAppMain.DrawSurfaces;
var
    iCount, iCount2: Integer;
    SrcYRatio, SrcXRatio: Double;
    TextureX: Integer;

    SrcRect: TRect;
    SourceSrfcDesc, DestSrfcDesc: TDDSurfaceDesc2;
    iCol, iRow: Integer;
    SourceSrfcPtr, DestSrfcPtr: PBytePtr;
begin
    try
        {begin by locking the source buffer}
        FillChar(SourceSrfcDesc, SizeOf(TDDSurfaceDesc2), 0);
        SourceSrfcDesc.dwSize := SizeOf(TDDSurfaceDesc2);
        FTexture.Lock(nil, SourceSrfcDesc, DDLOCK_WAIT, 0);

```

```

{now lock the destination buffer}

FillChar(DestSrfcDesc, SizeOf(TDDSurfaceDesc2), 0);

DestSrfcDesc.dwSize := SizeOf(TDDSurfaceDesc2);

FBackBuffer.Lock(nil, DestSrfcDesc, DDLOCK_WAIT, 0);

DestSrfcPtr := PBytePtr(Longint(DestSrfcDesc.lpSurface)+XPos+
                        ((YPos-(FEndptList.Count div 2))*DestSrfcDesc.lPitch));

{determine the ratio for stretching the texture map vertically}

SrcYRatio := SourceSrfcDesc.dwHeight / FEndptList.Count;

{increment the texture offset}

Inc(TextureOffset);

{rollover the texture offset if necessary}

if TextureOffset > SourceSrfcDesc.dwWidth then
    TextureOffset := 0;

{for each pair of endpoints in the endpoint list...}

for iCount := 0 to FEndptList.Count-1 do
begin
    {initialize the source pointer according to the next vertical pixel}

    SourceSrfcPtr := PBytePtr(Longint(SourceSrfcDesc.lpSurface)+
                                (SourceSrfcDesc.lPitch*Trunc(SrcYRatio * iCount)));

    {determine the ratio for stretching the texture map horizontally.
     the endpoints are stored as a single value relative to the center of
     the circle, so this value must be doubled to get the correct ratio}

    SrcXRatio := FTextureSwatchWidth / (Integer(FEndptList[iCount])*2);

    {for each pixel in this horizontal row...}

    for iCount2 := 0 to Integer(FEndptList[iCount])*2 do

```



```

begin
    {retrieve the next texture pixel}
    TextureX := Trunc(SrcXRatio * iCount2)+TextureOffset;

    {roll this over if necessary}
    if TextureX >= SourceSrfcDesc.dwWidth then
        TextureX := TextureX-SourceSrfcDesc.dwWidth;

        {since the endpoint is a value relative to the center axis of
        the circle, we simply subtract the endpoint value from our loop
        counter to get the correct horizontal pixel coordinate}
        DestSrfcPtr^[iCount2-Integer(FEndptList[iCount])] :=
            SourceSrfcPtr^[TextureX];
    end;

    {increment to the next line of pixels in the destination}
    Inc(DestSrfcPtr, DestSrfcDesc.lPitch);
end;

finally
    {unlock both buffers}
    FTexture.Unlock(nil);
    FBackBuffer.Unlock(nil);
end;
end;

.
.
.

procedure TfrmDXAppMain.GenerateCircle(Radius: Integer; Texture: string);

```

```

var
    MajorAxis, MinorAxis: integer;
    YOffset: Integer;

    XOfsQuad1, XOfsQuad2: TList;

    iCount: Integer;
begin
    {create the endpoint list}
    FEndptList := TList.Create;

    {create the quadrant lists}
    XOfsQuad1 := TList.Create;
    XOfsQuad2 := TList.Create;

    {initialize the major and minor axes and the y offset}
    MajorAxis := 0;
    YOffset := Radius;
    MinorAxis := Radius;

    {using the symmetry of a circle, we will store endpoints as only a single
    value relative to the Y axis, or center, of the circle. when drawing
    the texture, we will simply use 0-EndPointValue as the first point in
    the pair of endpoints}

    {loop until the major and minor axes meet}
    while MajorAxis <= MinorAxis do
    begin
        {increment the major axis}
        Inc(MajorAxis);
    end
end

```

```

{find the minor axis according to the circle equation, adding a small
  offset to make a more rounded circle}
MinorAxis := Trunc(Sqrt((Radius*Radius) - (MajorAxis*MajorAxis))+0.5);

{add this value to the second quadrant}
XOfsQuad2.Insert(0, TObject(MinorAxis));

{in the first 8th of the circle, the Y value will not be incrementing
  each time, so only store this endpoint when the Y value increases}
if MinorAxis < YOffset then
begin
  {add this endpoint to the first quadrant}
  XOfsQuad1.Add(TObject(MajorAxis));
  YOffset := MinorAxis;
end;
end;

{define the width of the texture to be mapped as the major axis value. this
  will cause the texture to be shrunk near the top of the circle, and
  stretched near the center of the circle}
FTextureSwatchWidth := MajorAxis;

{finally, add the endpoints in our quadrant lists to the final endpoint
  list in the correct order (from the top of the circle to the bottom)}
for iCount := 0 to XOfsQuad1.Count-1 do
  FEndptList.Add(XOfsQuad1[iCount]);
for iCount := 1 to XOfsQuad2.Count-1 do
  FEndptList.Add(XOfsQuad2[iCount]);

{these coordinates just list the top half of the circle, so we must

```

```
mirror these and add them to the list to define the bottom half}

for iCount := FEndptList.Count-1 downto 0 do
    FEndptList.Add(FEndptList[iCount]);

{free the quadrant lists}

XOfsQuad1.Free;

XOfsQuad2.Free;

{load the circle's texture map}

FTexture := DDLoadBitmap(FDirectDraw, Texture)

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

### Alternatives/Enhancements

There are hundreds, and probably thousands, of ways to map a texture onto a geometric shape. Texture mapping is an integral part of 3-D programming, unless you want your models to look like flat colors. Full 3-D texture mapping is beyond the scope of this book, but the idea behind it remains the same.

Using the lighting techniques described earlier, the spinning planet could be made more realistic by modifying the palette and drawing part of the texture using darker colors, creating a “night” effect. Additionally, you could use a second texture for the clouds, and use transparency techniques to make the clouds appear to move around while the planet stays still, or at least moves at a different rate. Alternatively, a bitmap with text could be used to create an interesting marquee.

### Dynamic Images

The ability to mold and shape a bitmap allows us to produce some pretty spectacular special effects. Creating an image on the fly can produce even more dramatic effects. Blitting sprite images on top of a background certainly creates a new image on the fly, but what we’re concerned with here is creating an image based on some sort of algorithm or mathematical function as opposed to just creating a composited layer of bitmaps.

You could create dynamic images such as this in any manner imaginable. Fractal imaging comes to mind, which uses various mathematical equations to produce stunning graphics of everything from abstract forms to images that model natural objects, such as leaves or mountains, in an incredibly lifelike way. Full 3-D games use collections of points and polygons to produce realistic gaming environments. While this list could go on and on, we’ll look at two methods for creating some pretty cool special effects: 3-D star fields and fire.

### Star Fields

Everyone has seen the Windows star field screen saver. At a fast enough velocity, it resembles something that you would see on the bridge of the *Enterprise* going at warp speed. Science fiction-based games are quite popular, and may benefit from the ability to produce a cool warp-speed star field animation that could be used when traveling from system to system.

## Basic Theory

This technique makes use of some very basic 3-D concepts. Specifically, each star is tracked as a point in 3-D space. Before we discuss how the stars are moved and how they are subsequently drawn to the screen, we must first understand some very basic 3-D graphics principles.

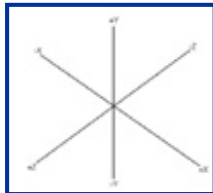
In a 2-D world, for which most of the examples in this book have been written, coordinates are typically based off of a flat grid. In mathematical terms, this can be considered a Cartesian grid, where the y-axis represents vertical placement and the x-axis represents horizontal placement. The origin, or where these two axes meet, is where the x-axis and y-axis are both at 0. The y-axis increases as you move upward and decreases as you move downward. Similarly, the x-axis increases to the right and decreases to the left. (For the moment, we'll ignore the fact that on a computer screen, the origin is in the upper left-hand corner and y increases as you move downward.) The figure at the top of the following page demonstrates a typical Cartesian coordinate system.

Using this system, simple 2-D sprites can be tracked by a simple x and y coordinate (as we've been doing throughout this book). They can move left and right or up and down, but this system has no method for tracking depth, or which sprites are on top of each other.



**Figure 11-11:** A Cartesian coordinate system

To track depth, we need to add another axis. This is known as the z-axis, and in concept, it is perpendicular to the x and y axes. You can think of it as another axis that runs straight into the computer screen. The z-axis increases as you go “into” the computer screen, and decreases as you come “out of” the computer screen. The following figure demonstrates this concept.



**Figure 11-12:** A 3-D coordinate system

A star can be effectively tracked as a single point, so one x, y, z coordinate will represent a single star. If we now initialize a star at a specific coordinate, and then modify its z coordinate, it will appear to move toward or away from the user as desired.

The values for each axis in a system such as this are somewhat arbitrary. In our example, each axis can range from -1000 to 1000. Stars can be positioned anywhere within this square area. Now, even though we've got an acceptable method of tracking star positions, we still can't draw them to the screen. How do we convert these arbitrary axis positions into actual screen coordinates that we can draw?

This is where perspective projection comes in. The equations are very simple, and will produce a correct 2-D coordinate for a 3-D point that will accurately reflect the perspective in relation to its depth on the z-axis. Skipping the lengthy derivation and proofs for producing these equations, they are:

```
Screen_X := XPos / ZPos * VIEWDISTANCE;
```

```
Screen_Y := YPos / ZPos * VIEWDISTANCE;
```

Dividing the horizontal and vertical positions by the depth is what gives us the correct perspective. The

viewing distance constant is required because without it, the origin of the z-axis is effectively the glass of the computer screen. By multiplying by a constant viewing distance, we effectively push the origin of the z-axis into the computer screen. By modifying this, we can make things appear closer or farther away, depending on the size of this value.

To further enhance this illusion, we've dynamically created a grayscale palette and based the color of the pixel drawn to the screen on its z-axis position. This causes the stars to fade from black to white as they move toward the user, which further enhances the 3-D effect. The following listing demonstrates this effect.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**ITKNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 11-7: A 3-D star field*

---

```
const
```

```
.  
. .  
. . .
```

```
{the total number of stars on the screen at once}
```

```
NUMSTARS = 1000;
```

```
{the viewing distance}
```

```
VIEWDISTANCE = 350;
```

```
.  
. .  
. . .
```



```
type
  {tracks single point within our 3D world}

  TStar = record
    X, Y, Z: Single;
  end;

  .
  .
  .

var
  .
  .
  .

  {the palette interface}
  FPalette: IDirectDrawPalette;

implementation

  .
  .
  .

procedure TfrmDXAppMain.FormActivate(Sender: TObject);

var
  .
  .
```

```

.

{holds palette colors}
PalColors: TColorTable;

iCount: Integer;
begin

.
.
.

{define the colors for the palette. this produces a ranged palette,
fading from black to white in the lower half, and all white in
the upper half. this is used to make the stars appear to fade in as
they move closer to the user}
for iCount := 0 to 127 do
begin
    PalColors[iCount].R := 255;

    PalColors[iCount].G := 255;
    PalColors[iCount].B := 255;
    PalColors[iCount].A := 0;
end;

for iCount := 128 to 255 do
begin
    PalColors[iCount].R := 255-(2*(iCount-128));
    PalColors[iCount].G := 255-(2*(iCount-128));
    PalColors[iCount].B := 255-(2*(iCount-128));
    PalColors[iCount].A := 0;

```

```
end;
```

```
{create the actual palette}
```

```
FDirectDraw.CreatePalette(DDPCAPS_8BIT or DDPCAPS_ALLOW256,  
                           @PalColors, FPalette, nil);
```

```
{attach the palette to the primary surface}
```

```
FPrimarySurface.SetPalette(FPalette);
```

```
{finally, initialize all of the stars}
```

```
for iCount := 0 to NUMSTARS do
```

```
    InitStar(iCount);
```

```
    .
```

```
    .
```

```
    .
```

```
end;
```

```
    .
```

```
    .
```

```
    .
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
    SrfcDesc: TDDSurfaceDesc2;
```

```
    BytePtr: PByte;
```

```
    iCount: Integer;
```

```

    XPos, YPos: Integer;
begin
    {clear the back buffer}
    ColorFill(FBackBuffer, 255, nil);

    {lock the back buffer. all animation will take place on this surface}
    FillChar(SrfcDesc, SizeOf(TDDSurfaceDesc2), 0);
    SrfcDesc.dwSize := SizeOf(TDDSurfaceDesc2);
    FBackBuffer.Lock(nil, SrfcDesc, DDLOCK_WAIT, 0);
    BytePtr := SrfcDesc.lpSurface;

    {move and draw all of the stars}
    for iCount := 0 to NUMSTARS do
    begin
        {a negative value here moves the stars closer to the screen. a positive
        value would move them away, giving the sensation of moving backwards}
        Stars[iCount].Z := Stars[iCount].Z - 15;

        {if the star's Z value is greater than zero...}
        if Stars[iCount].Z > 0 then
        begin
            {project the star onto our 2D computer screen in a perspective
            correct manner. note that since the origin of the computer
            screen is in the upper left-hand corner, we're moving the stars
            into the center of the screen to produce a more desirable effect.
            note that we're also reversing the sign of the Y coordinate so
            that it accurately reflects the movement in a Cartesian coordinate
            system}
            XPos := Round((DXWIDTH div 2)+
                (Stars[iCount].X/Stars[iCount].Z*VIEWDISTANCE));

```

```

YPos := Round((DXHEIGHT div 2)-
              (Stars[iCount].Y/Stars[iCount].Z*VIEWDISTANCE));

{if our stars are off of the screen...}
if (XPos<0) or (XPos>DXWIDTH) or (YPos<0) or (YPos>DXHEIGHT) then
  {...re-create it}
  InitStar(iCount)
else
  {otherwise, draw it at the correct position, using a color
   proportional to its Z axis position, making it appear to fade in
   from black}
  PByte(Longint(BytePtr)+XPos+(YPos*SrfcDesc.lPitch))^ :=
    Trunc(255*(Stars[iCount].Z/1000));
end
else
  {...otherwise, reinitialize the star}
  InitStar(iCount);
end;

{finally, unlock the buffer}
FBackBuffer.Unlock(nil);
end;

.
.
.

procedure TfrmDXAppMain.InitStar(Index: Integer);
begin
  {place the star at a random position}

```

```
Stars[Index].X := random(2000)-1000;  
  
Stars[Index].Y:=random(2000)-1000;  
  
{start the star as far back as necessary}  
  
Stars[Index].Z:=1000;  
  
end;  
  
end.
```

---

## Alternatives/Enhancements

Allowing the user to specify the velocity of the stars through some sort of throttle control would create the perception of speed. By modifying the starting position of the stars, it would even be possible to create the sensation of turning. Coupled with a fast bitmap scaling algorithm, this effect could be used to create a pretty cool arcade-type space shooter.



*Tip:* While this animation is pretty fast in its current implementation, there's really no need to use a three-dimensional coordinate system with perspective projection unless we're going to be creating a full 3-D game. This same effect can be reproduced using only x and y velocities and integer math, which could dramatically speed up the effect and leave more CPU time for more critical tasks. Remember, graphical effects do not have to be totally accurate; as long as it looks right, it is right.

---

Alternatively, if this was going to be used simply as a static animation or a transition from scene to scene, it may be faster just to create an actual animation using bitmaps drawn by an artist. However, the method by which stars are implemented could be considered a primitive particle system. By starting several "stars" at the exact same coordinate but giving each one a different heading and velocity and implementing a lifespan counter for each one so it disappears after a certain amount of time, a very realistic particle explosion system could be created. Adding gravity effects to the particles could produce an even more remarkable effect.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Fire**

The fire effect is a really cool trick that, when performed properly, can create a very convincing illusion of fire. This effect is well documented both on the Web and in various books on graphics programming, and has been around for quite some time. It is a very computationally intensive effect, but it can be highly optimized and runs quite fast when implemented in assembly code. There are numerous ways of implementing this effect, but we'll examine one that produces a nice effect at decent speeds.

**Basic Theory**

The first thing we need to accomplish is creating an appropriate palette. For an effective fire, we'll create a palette that ranges from white in the upper slots (255-192), through yellow and orange, to red, and finally to black in the lowest slots (63-0). We could adjust the palette range somewhat and put some blues in, to simulate a cooler flame, or perhaps add some grays at the end, which would effectively simulate smoke.

This effect involves moving pixels that already exist in a surface. Therefore, we need to create a new surface that will contain the image of the fire. Since we'll be reading values from this surface, we'll place it in system memory. This algorithm is a CPU hog, so even though we'll create a surface that's the same size as the screen, we'll only use a small part of it for the effect. In our example, we've defined a starting and ending vertical position as well as a starting and ending horizontal position, which will confine the effect to the rectangle defined by these coordinates.

Now for the real work. Starting at the first row of pixels (defined by our starting vertical position) in the fire effect surface, we begin iterating through each column (as defined by our starting and ending horizontal position). We retrieve the current value of this pixel, subtract a random amount from this number, and then place the new value in the row of pixels above our current row. In effect, this produces an animation of pixels moving upwards on the screen. Additionally, we add a random number between -1 and 1 to move the pixels left or right in a chaotic manner.



**Figure 11-13:** Flaming pixels

Relative to the colors we defined in our palette, the pixel colors will slowly move from white, through yellow and red, and finally to black as they move upward due to the subtraction of a random value from their current color. This produces a very effective illusion of flames. At each frame of the animation, we'll need to place some random pixel values on the bottom row of the effect to give the algorithm some starting values. Implementing this effect is very simple, as illustrated by the following example.

*Listing 11-8: The flame effect*

---

```
const

    .

    .

    .

    {define the rectangular area for the effect}

    XSTART = 200;

    XEND   = 440;

    YSTART = 100;

    YEND   = 400;

    {define the random decay value}

    DECAY  = 6;

type

    .

    .

    .

    TBytePtr = array[0..0] of Byte;

    PBytePtr = ^TBytePtr;

var

    .

    .

    .
```



```
{the interfaces for the primary and backbuffer surfaces}
```

```
FPrimarySurface,
```

```
FBackBuffer,
```

```
FireSurface: IDirectDrawSurface4;
```

```
{the palette interface}
```

```
FPalette: IDirectDrawPalette;
```

```
implementation
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
var
```

```
.
```

```
.
```

```
.
```

```
{hold the palette colors}
```

```
PalColors: TColorTable;
```

```
iCount: Integer;
```

```
{this procedure stuffs the RGB values into the appropriate color table entry}
```

```
procedure SetPalEntry(Idx, Red, Green, Blue: Integer);
```

```
begin
```

```
    PalColors[Idx].R := Red;
```

```

    PalColors[Idx].G := Green;

    PalColors[Idx].B := Blue;

    PalColors[Idx].A := 0;

end;

begin

    .

    .

    .

    {define the colors for the palette. this will produce a spread from white
    to yellow/orange to red, fading to black}

    for iCount := 0 to 63 do

    begin

        SetPalEntry(iCount, iCount shl 2, 0, 0);

        SetPalEntry(64+iCount, 255, iCount shl 2, 0);

        SetPalEntry(128+iCount, 255, 255, iCount shl 2);

        SetPalEntry(192+iCount, 255, 255, 255);

    end;

    {create the actual palette}

    FDirectDraw.CreatePalette(DDPCAPS_8BIT or DDPCAPS_ALLOW256, @PalColors,

        FPalette, nil);

    {attach the palette to the primary surface}

    FPrimarySurface.SetPalette(FPalette);

    {create the offscreen buffer for the fire. we're going to be modifying this
    constantly, so we'll place it in system memory}

    FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

    DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

```

```
DDSurface.dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
```

```
DDSurface.dwWidth := DXWIDTH;
```

```
DDSurface.dwHeight := DXHEIGHT;
```

```
DDSurface.ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN or
```

```
DDSCAPS_SYSTEMMEMORY;
```

```
{initialize the surface}
```

```
DXCheck( FDirectDraw.CreateSurface(DDSurface, FireSurface, nil) );
```

```
{initialize the offscreen buffer to all black}
```

```
ColorFill(FireSurface, 0, nil);
```

```
.
```

```
.
```

```
.
```

```
end;
```

```
.
```

```
.
```

```
.
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
SrcRect: TRect;
```

```
SrfcDesc: TDDSurfaceDesc2;
```

```
iCol, iRow: Integer;
```

```
FireSrfcPtr,
```

```
BottomLine: PBytePtr;
```

```
CurPixel: Byte;
```

```
begin
```

```

{begin by locking the fire surface buffer}

FillChar(SrfcDesc, SizeOf(TDDSurfaceDesc2), 0);

SrfcDesc.dwSize := SizeOf(TDDSurfaceDesc2);

FireSurface.Lock(nil, SrfcDesc, DDLOCK_WAIT, 0);

FireSrfcPtr := PBytePtr(Longint(SrfcDesc.lpSurface)+
                        (YSTART*SrfcDesc.lPitch));

{we'll also need a pointer to the bottom line of the rectangular
 area for the effect}

BottomLine := PBytePtr(Longint(SrfcDesc.lpSurface)+(YEND*SrfcDesc.lPitch));

{initialize some new pixels in the fire}

for iCol := XSTART to XEND do
    BottomLine^[iCol] := 255-Random(32);

{perform the actual flame calculations}

for iRow := YSTART to YEND do
begin
    for iCol := XSTART to XEND do
        begin
            {retrieve the pixel from this column}
            CurPixel := FireSrfcPtr^[iCol];

            {if the pixel is less than the decay value or is at the far ends of
             the effect area, blacken it out...}
            If (CurPixel < Decay) or (iCol = XSTART) or (iCol = XEND) then
                {draw a black pixel on the row above the current one}
                PBytePtr(Longint(FireSrfcPtr)-SrfcDesc.lPitch)^[iCol] := 0
            else
                {...otherwise, draw a new pixel on the row above the current one,
                 either directly above or to either side of the current pixel, but
                 subtract a random value from its color. this causes the color of the

```

```
        pixel to slowly move from white, through yellow and red, to black}
PBytePtr(Longint(FireSrfcPtr)-SrfcDesc.lPitch)^[iCol+Random(3)-1] :=
        CurPixel - Random(DECAY);

end;

{move to the next row in the fire surface}
Inc(FireSrfcPtr, SrfcDesc.lPitch);
end;

{unlock the buffer}
FireSurface.Unlock(nil);

{finally, blit the fire surface to the back buffer to make it visible}
SrcRect := Rect(0, 0, DXWIDTH, DXHEIGHT);
FBackBuffer.BltFast(0, 0, FireSurface, SrcRect, DDBLTFAST_WAIT);
end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**IT KNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

### Alternatives/Enhancements

The strength of the blaze could easily be controlled by writing smaller starting pixel values on the bottom line, or perhaps writing values on only a few pixels and initializing the rest to 0. Because of the way this algorithm works, any pixels drawn into the fire canvas will become part of the blaze. By drawing text into the surface, the letters would appear to burn, creating a very cool effect.

Perhaps the easiest modification to this effect is changing the colors used in the palette. By defining a different range, we could make green fire, purple fire, etc. Indeed, by coupling this with some palette cycling, we could make the fire change colors on the fly. Employing a different method for moving the pixels could also dramatically change the effect. Various other plasma effects demonstrated in examples found throughout the Web use both of these techniques.

### Summary

In this chapter, we discussed several techniques for providing visual special effects that can enhance and complement a game application. We examined various bitmap manipulation methods, including lighting and simple texture mapping, as well as methods for dynamically creating images. When creating visual effects for your games, it is important to keep these points in mind:

- Computer graphic special effect techniques can be used to produce visual images that would be difficult or impossible to reproduce using blitting techniques alone.
- By drawing bitmaps using mathematical equations to modify the location of pixels as they are drawn to a destination, you can produce some very cool effects. Manipulating and transforming bitmaps in this way can give images a sense of texture or depth, and can stretch the art budget dollar.
- Bitmaps can be easily scaled to a larger or smaller size. DirectDraw supports this natively, but by using our own methods, we can control certain aspects and produce images with much better resolutions.
- Rotating an image is only supported by DirectDraw if it is available in the hardware. We can use our own techniques for rotating images, but most simple implementations are slow due to the mathematical intensity of the algorithm.
- Lighting techniques can be used to create game worlds that are more realistic. A cheap, simple

method is to segregate the palette into four segments of 64 slots, with the first segment containing the full intensity colors, and the other segments containing the same colors at darker and darker intensities. This results in banding, but the effect is quick and produces adequate results.

- Transition effects give a game a better sense of flow. Pay attention to how modern movies transition between scenes for inspiration. Many transition effects are relatively easy to implement.
- Texture mapping is a complex subject with intense math requirements. However, we can use scaling techniques in order to produce some simple texture mapping effects.
- Using some basic 3-D mathematical algorithms, it is fairly easy to produce a realistic star field. The equations used in this effect are at the core of real-time 3-D graphics programming techniques.
- By moving the pixels in an image, we can create a very realistic fire effect. It is possible to modify how these pixels move as well as perform a little palette rotation to produce some very interesting plasma effects.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 12 Artificial Intelligence Techniques

This chapter covers the following topics:

- Antagonist movement techniques
- Finite state machines
- Probability machines
- Path finding techniques

There is no more dangerous opponent than a living, breathing human being. Humans can be incredibly unpredictable, and good game players can change tactics in a fraction of a nanosecond to take advantage of some new predicament within the game world. This is what makes multiplayer games so attractive; when facing real opponents, the replayability factor of just about any game is dramatically extended.

However, you can't always hook up with another human player. Some people may not even have the necessary hardware, and some may just want to play a game by themselves without worrying about the competition level introduced by battling other humans. In order for a single-player game to grab and hold the player's interest, it needs to be challenging, as good graphics and sound will entertain a user for only so long. This is where artificial intelligence techniques come in. In essence, artificial intelligence is the process of simulating intelligent thought and behavior through algorithmic methods.

As one would surmise from such a broad topic, a discussion of artificial intelligence could cover volumes upon volumes. Indeed, artificial intelligence is an area of specialty in the realm of computer science. Several large colleges and corporations spend millions of dollars (or more) yearly on furthering the research of artificial intelligence. Everything from simulating the thought process of an ant to accurately modeling the interaction of neurons in the human brain is being researched, with varying degrees of success.

Of course, most of these scholarly experiments employ technology that is either beyond the reach of game programmers or is simply "too much" for a game application. What we're interested in accomplishing is not an accurate model of living, thinking organisms, but a close facsimile thereof. We need game entities that appear to behave in an intelligent manner, but they do not have to truly be intelligent. As long as they challenge the player and provide a few surprises every now and then, we've done our job.

In this chapter, we examine various techniques for implementing simplistic artificial intelligence that will give our



game's enemy units a sense of personality and behavior. Since we are concentrating on artificial intelligence programming, the examples in this chapter will use very simple graphics so we don't unnecessarily complicate the code.

## Movement

The most noticeable display of artificial intelligence is in the movement of enemy units. Indeed, in most action games, the user is going to be primarily concerned with the movement of enemy units, such as whether they are aggressively chasing the user's ship or running like cowards. Let's look at several techniques that can be used to breathe life into the movement of game world antagonists.

### Random Movement

Perhaps the simplest simulation of artificial intelligence is to introduce an element of randomness. When an enemy unit moves in a fashion that is unpredictable to the player, that unit can appear to exhibit a rudimentary intelligence. Consider the apparently random flight path of a common housefly. It darts about here and there, yet it always appears to have a mind of its own and to know where it's going.

Random movement is simple to implement. For example, say we have a very basic sprite structure defined as:

```
TSprite = record  
  
    XPos, YPos: Integer;           // horizontal and vertical sprite position  
  
end;
```

To implement random movement, we can simply use the Object Pascal function `Random` to return a number indicating the number of units to move in a particular direction. The `Random` function takes a single integer argument as the extent of the range of numbers to be returned, and returns an integer in the range  $0 \leq X < \text{Argument}$ . If we specify a range of 3, which would randomly return the numbers 0, 1, or 2, and then subtract 1, our modified range of returned numbers would be -1, 0, or 1. By adding this translated number to the horizontal and vertical positions of our sprite, it would appear to move about randomly. The following listing demonstrates this technique.

#### *Listing 12-1: Random movement*

---

```
type  
  
    .  
  
    .  
  
    .  
  
    TSprite = record  
  
        XPos, YPos: Integer;  
  
    end;
```

```
var
```

```
Enemy: TSprite;
```

```
implementation
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
  .
```

```
  .
```

```
  .
```

```
  {initialize the enemy's starting position}
```

```
  Enemy.XPos := 320;
```

```
  Enemy.YPos := 240;
```

```
  .
```

```
  .
```

```
  .
```

```
end;
```

```
  .
```

```
  .
```

```
  .
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
  SrfcDc: HDC;
```

```
  TempCanvas: TCanvas;
```

```
begin
```

```
  {erase the last frame of animation}
```

```
ColorFill(FBackBuffer, clBlack, nil);
```

```
{move the enemy randomly. this could be either no movement  
or movement by one pixel in a positive or negative direction}
```

```
Enemy.XPos := Enemy.XPos + Random(3)-1;
```

```
Enemy.YPos := Enemy.YPos + Random(3)-1;
```

```
{check for collisions with the side of the screen}
```

```
if Enemy.XPos < 0 then
```

```
    Enemy.XPos := 0;
```

```
if Enemy.XPos > DXWIDTH - 1 then
```

```
    Enemy.XPos := DXWIDTH - 1;
```

```
if Enemy.YPos < 0 then
```

```
    Enemy.YPos := 0;
```

```
if Enemy.YPos > DXHEIGHT - 1 then
```

```
    Enemy.YPos := DXHEIGHT - 1;
```

```
{retrieve a device context for use with GDI output techniques}
```

```
FBackBuffer.GetDC(SrvcDc);
```

```
TempCanvas := TCanvas.Create;
```

```
try
```

```
    {initialize the canvas properties}
```

```
    TempCanvas.Handle := SrvcDc;
```

```
    TempCanvas.Font.Color := clWhite;
```

```
    TempCanvas.Pen.Style := psClear;
```

```
    TempCanvas.Brush.Style := bsSolid;
```

```
    TempCanvas.Brush.Color := clRed;
```

```
{draw a small red circle at the current sprite position}
```

```
TempCanvas.Ellipse(Enemy.XPos, Enemy.YPos, Enemy.XPos + 3, Enemy.YPos + 3);
```

```
{draw some text}
```

```
TempCanvas.Brush.Style := bsClear;

TempCanvas.TextOut(0, 0, 'Random Movement')

finally
    {clean up the temporary canvas and device context}

TempCanvas.Handle := 0;

FBackBuffer.ReleaseDC(Srfdc);

TempCanvas.Free;

end;

end;
```

This does indeed move our sprite around the screen randomly, but it merely jitters around the middle of the screen and doesn't really project an air of intelligence at all. We could improve this by waiting a random amount of time before changing directions, which would give us a more housefly-like behavior, but there is no real intelligence underneath, and this enemy would pose no true threat to the player.

### Aggression/Evasion

In order to portray a sense of intelligence, enemy units should react in a manner dictated somewhat by the actions of the player. This could be anything from the player's movements (or lack thereof) to the number of rockets or bullets the player launched to the status of a player's shields or hit points. When an enemy reacts to the player's actions or current status, its intelligence can be convincingly real.

This can also give an enemy a sense of personality. An enemy that actively hunts down the player can be perceived as confident, powerful, and aggressive. By contrast, an enemy that constantly eludes the player or turns tail can be perceived as cowardly, injured, or perhaps even cunning, like an assassin stalking its target.



**Tip:** Varying the velocity, perhaps according to the proximity of the player, will enhance this technique and improve the perceived intelligence.

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)  
▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



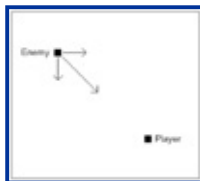
**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

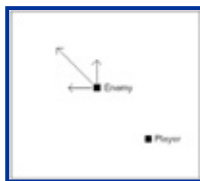
[Previous](#) [Table of Contents](#) [Next](#)

Modeling aggression is simple enough. The application simply examines the horizontal and vertical position of the enemy unit and the player unit, and adjusts the enemy unit's horizontal and vertical positions to bring it close to the player. For example, if the player unit's horizontal position is less than the enemy's, subtracting 1 from the enemy's horizontal position brings it closer to the player. If the player's horizontal position is greater than the enemy's, adding 1 to the enemy's horizontal position brings it closer to the player. The same holds true for the vertical position. This is illustrated below.



**Figure 12-1:** Chasing the player

Modeling evasion is just as simple. Indeed, if we simply reverse the signs of the values being added to the horizontal and vertical position of the enemy, it will run from the player instead of chasing after it. This is illustrated below.



**Figure 12-2:** Evading the player

Of course, most games would need to take into account obstacles and line of sight, but the basic principle holds true. The following example illustrates how to implement chase and evasion algorithms. The player's dot is controlled by the cursor keys, and the enemy dot will either chase or flee from the user based on its current status, which is toggled by the Enter key.

*Listing 12-2: Aggression and evasion in action*

```
type
```

```
.  
.   
.
```

```
TSprite = class
```

```
    XPos, YPos: Integer;
```

```
    procedure MoveSprite(XOffset, YOffset: Integer);
```

```
end;
```

```
var
```

```
.  
.   
.
```

```
    Enemy, Player: TSprite;
```

```
    Chase: Boolean;
```

```
implementation
```

```
.  
.   
.
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
.  
.   
.
```

```
{create the two sprites}

Enemy := TSprite.Create;
Player := TSprite.Create;

{initialize the enemy and player sprite positions}

Enemy.XPos := 320;
Enemy.YPos := 0;

Player.XPos := 320;
Player.YPos := 477;

{start out in aggression/chase mode}

Chase := TRUE;
```

```
.
.
.
```

```
end;
```

```
.
.
.
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
    SrfcDc: HDC;
```

```
    TempCanvas: TCanvas;
```

```
EnemyOffset: Integer;
```

```
begin
```

```
  {erase the last frame of animation}
```

```
  ColorFill(FBackBuffer, clBlack, nil);
```

```
  {if the Enter key was pressed, toggle aggression/evasion}
```

```
  if (GetAsyncKeyState(VK_RETURN) and $8000) = $8000 then
```

```
    Chase := not Chase;
```

```
  {move the player sprite based on the cursor keys}
```

```
  if (GetAsyncKeyState(VK_LEFT) and $8000) = $8000 then
```

```
    Player.MoveSprite(-2, 0);
```

```
  if (GetAsyncKeyState(VK_RIGHT) and $8000) = $8000 then
```

```
    Player.MoveSprite(2, 0);
```

```
  if (GetAsyncKeyState(VK_UP) and $8000) = $8000 then
```

```
    Player.MoveSprite(0, - 2);
```

```
  if (GetAsyncKeyState(VK_DOWN) and $8000) = $8000 then
```

```
    Player.MoveSprite(0, 2);
```

```
  {determine the appropriate movement for chasing or evading}
```

```
  if Chase then
```

```
    EnemyOffset := 1
```

```
  else
```

```
    EnemyOffset := -1;
```

```
  {move the enemy accordingly}
```

```
  if Player.XPos > Enemy.XPos then
```

```
    Enemy.MoveSprite(EnemyOffset, 0);
```

```
  if Player.XPos < Enemy.XPos then
```

```
    Enemy.MoveSprite(-EnemyOffset, 0);
```

```
  if Player.YPos > Enemy.YPos then
```



```

    Enemy.MoveSprite(0, EnemyOffset);

if Player.YPos < Enemy.YPos then

    Enemy.MoveSprite(0, -EnemyOffset);

{retrieve a device context and create a canvas}
FBackBuffer.GetDC(SrfcDc);
TempCanvas := TCanvas.Create;
try
    {set up the canvas}
    TempCanvas.Handle := SrfcDc;
    TempCanvas.Font.Color := clWhite;

    {draw the two sprites as small dots}
    TempCanvas.Pen.Style := psClear;
    TempCanvas.Brush.Style := bsSolid;
    TempCanvas.Brush.Color := clBlue;
    TempCanvas.Ellipse(Player.XPos, Player.YPos, Player.XPos+3, Player.YPos+3);
    TempCanvas.Brush.Color := clRed;
    TempCanvas.Ellipse(Enemy.XPos, Enemy.YPos, Enemy.XPos+3, Enemy.YPos+3);

    {indicate which state the enemy sprite is in}
    TempCanvas.Brush.Style := bsClear;
    if Chase then
        TempCanvas.TextOut(0, 0, 'Chasing')
    else
        TempCanvas.TextOut(0, 0, 'Evading');
        TempCanvas.TextOut(0, 440, 'Press Enter to toggle state');
finally
    {clean up}
    TempCanvas.Handle := 0;
    FBackBuffer.ReleaseDC(SrfcDc);

```

```

    TempCanvas.Free;

end;

end;

procedure TSprite.MoveSprite(XOffset, YOffset: Integer);
begin
    {update the sprite's position}

    XPos := XPos + XOffset;

    YPos := YPos + YOffset;

    {clip to the screen boundaries}

    if XPos < 0 then
        XPos := 0;

    if XPos > DXWIDTH - 1 then
        XPos := DXWIDTH - 1;

    if YPos < 0 then
        YPos := 0;

    if YPos > DXHEIGHT - 1 then
        YPos := DXHEIGHT - 1;

end;

```

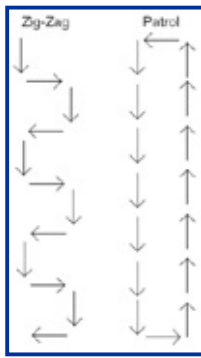
---

We could have just as easily made the chase/evade decision based upon whether or not the player was actively moving. If the player's sprite position has changed, we could have switched to evade mode; if the player hadn't moved, we could go right to chase mode. This would certainly keep players on their toes, and in the event our enemy units could kill a player, this would certainly discourage anyone from waiting around somewhere for very long!

### **Patterned Movement**

Chasing the player, running from the player, and moving around randomly can get pretty boring after a while. Typically, enemy units need to exhibit more complex movement, and one way to simulate this is through movement patterns.

For example, take the old arcade game Galaxian. When the enemies appeared in each round, they flew onto the screen in a complex flight path. Although their movements were a bit predictable, it made their appearance more menacing and deliberate. We can use patterned movement in our games to model similar complex, deliberate movements and make our enemies appear to move about more intentionally. Indeed, we could create several patterns and choose one randomly (or in a more context-sensitive manner), as illustrated in Figure 12-3.



**Figure 12-3:** Movement patterns

There are several methods by which patterned movement can be implemented based on the type of game and the sprite engine. If we had a simple, 2-D shooter, our patterns could simply be based off of a series of horizontal and vertical offsets that move the unit in the desired manner. These offsets would be relative to the current position of the sprite, so the patterns could be used anywhere within the game world.



**Tip:** It is relatively easy to flip the movement pattern horizontally or vertically, thus increasing the flexibility of this technique.

To implement this method of patterned movement, we could make an array of TPoints, filling each entry with the horizontal or vertical offset for that position within the pattern. As we move the sprite, we keep track of an index into this array of offsets, incrementing the offset at a preset time to advance to the next movement within the pattern. Once the pattern is complete, we can choose another array (or another row within a multidimensional array) and reset the index to 0, thus starting a new pattern. This technique is illustrated below.

**Listing 12-3:** *Implementing patterned movement*

```
TSprite = record
    XPos, YPos: Integer;
    CurPattern,
    CurPatternIdx: Integer;
    PatternIdxChange,
    CurTimer: Longint;
end;

TPatterns = array[0..5, 0..31] of TPoint;

const
    {in this example, the patterns are stored as direct X and Y offsets, but
    any method of specifying a direction and/or offset will do}
```

```
Patterns: TPatterns = (  
  ((x: 1; y: 0), (x: 1; y: 0), (x: 1; y: 0), (x: 0; y: 1), (x: 0; y: 1), ...  
  ((x: -1; y: -1), (x: -1; y: -1), (x: -1; y: -1), (x: -1; y: -1), ...  
  ((x: 1; y: 0), (x: 1; y: 0), (x: 1; y: 1), (x: 1; y: 1), (x: 0; y: 1), ...  
  ((x: 1; y: 0), (x: 1; y: 0), (x: 1; y: 0), (x: 1; y: 0), (x: 0; y: -1), ...  
  ((x: 1; y: 0), (x: 1; y: 0), (x: 1; y: 0), (x: 1; y: -1), (x: 0; y: -1), ...  
  ((x: 0; y: -1), (x: 0; y: -1), (x: 0; y: -1), (x: 0; y: -1), ...
```

```
var
```

```
.  
. .  
. .
```

```
Enemy: TSprite;
```

```
implementation
```

```
.  
. .  
. .
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
.  
. .  
. .
```

```
{initialize the enemy sprite}
```

```
Randomize;
```

```
Enemy.XPos := 320;
```

```
Enemy.YPos := 240;

Enemy.CurPattern := Random(6);    // start at a random pattern

Enemy.CurPatternIdx := 0;

Enemy.PatternIdxChange := 100;   // change the pattern index often

Enemy.CurTimer := timeGetTime;
```

```
.
.
.
```

```
end;
```

```
.
.
.
```

```
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
    SrfcDc: HDC;
```

```
    TempCanvas: TCanvas;
```

```
begin
```

```
    {erase the last frame of animation}
```

```
    ColorFill(FBackBuffer, clBlack, nil);
```

```
    {if it is time to change the pattern index...}
```

```
    if timeGetTime - Enemy.CurTimer > Enemy.PatternIdxChange then
```

```
    begin
```

```
        {...move to the next step in the pattern}
```

```
        Enemy.CurPatternIdx := Enemy.CurPatternIdx + 1;
```

```
    {if we have moved the entire pattern...}
```

```

if Enemy.CurPatternIdx > 31 then
begin
    {...choose a new pattern}
    Enemy.CurPattern := Random(6);
    Enemy.CurPatternIdx := 0;
end;

{track for the next pattern index change}
Enemy.CurTimer := timeGetTime;
end;

{update the enemy sprite's position based on the current step with
the pattern}
Enemy.XPos := Enemy.XPos + Patterns[Enemy.CurPattern, Enemy.CurPatternIdx].X;
Enemy.YPos := Enemy.YPos + Patterns[Enemy.CurPattern, Enemy.CurPatternIdx].Y;

{clip the enemy to the screen boundaries}
if Enemy.XPos < 0 then
    Enemy.XPos := 0;
if Enemy.XPos > DXWIDTH - 1 then
    Enemy.XPos := DXWIDTH - 1;
if Enemy.YPos < 0 then
    Enemy.YPos := 0;
if Enemy.YPos > DXHEIGHT - 1 then
    Enemy.YPos := DXHEIGHT - 1;

{retrieve a device context and create a canvas}
FBackBuffer.GetDC(SrfcDc);
TempCanvas := TCanvas.Create;
try
    {set up the canvas}
    TempCanvas.Handle := SrfcDc;

```

```
TempCanvas.Font.Color := clWhite;

{draw the enemy sprite}

TempCanvas.Pen.Style := psClear;

TempCanvas.Brush.Style := bsSolid;

TempCanvas.Brush.Color := clRed;

TempCanvas.Ellipse(Enemy.XPos, Enemy.YPos, Enemy.XPos + 3, Enemy.YPos + 3);

{indicate the pattern number}

TempCanvas.Brush.Style := bsClear;

TempCanvas.TextOut(0, 0, 'Pattern: '+IntToStr(Enemy.CurPattern));

finally

    {clean up}

    TempCanvas.Handle := 0;

    FBackBuffer.ReleaseDC(SrfcDc);

    TempCanvas.Free;

end;

end;
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

This makes our enemy sprite appear to move in a more deliberate manner. We can adjust the offsets accordingly to make some patterns move the sprite faster or slower, and with some minor modifications we could make some patterns longer than others. By selecting different patterns, the movement appears more deliberate while at the same time reducing its predictability.



*Note:* In real-world situations, we should make sure that the sprite is not positioned such that a movement pattern will run it into a movement boundary immediately. In our example, we simply clip to the edges of the screen, but this can make a movement pattern almost useless. In a real-world situation, we should reverse the pattern when it detects a movement boundary, or even exit the pattern immediately and choose another form of movement.

## Finite State Machines

We've been examining methods by which intelligent movement can be modeled, and so far we've come up with some acceptable algorithms that can provide convincing, if somewhat limited, intelligent movement. However, we haven't discussed methods by which enemy units determine which movement technique to use.

We could just randomly pick something, but this would not model any sort of intelligent thought process. We need an algorithm by which an enemy unit can make a series of rational decisions based on various inputs from the game world, such as the current environment as well as the actions of the player. This can be accomplished through a method known as a finite state machine. A finite state machine is a logical construction that consists of several "states," each of which defines a specific behavior, and "state transitions," which define specific rules for moving from state to state.



**finite state machine:** A logical construction that contains a series of states defining specific behaviors and a series of state transitions defining rules for moving from state to state.

Using finite state machines, enemy units can have very complex behavioral models. These can include not only movements but also different tactics or other actions based on the actions of the player, the state of other game objects, and the overall situation. State machines also allow for the easy extension of behaviors, as all that would be required is the logic for the behavior under the new state and logic for moving into and out of this state. An arbitrary state machine could be visualized as illustrated below.





**Figure 12-4:** An abstracted finite state machine

Each circle in the illustration represents a specific state, or behavior, and the lines connecting them represent the logic determining how one state leads to another.

One way to model this in software would be to define each state as an integer value or perhaps an enumerated type. A case statement could then be utilized to execute the appropriate code for the current state. The transitions from state to state could be controlled in a number of ways, from randomly choosing a different state at a specific interval to complex logic that considers the player's attributes versus the computer controlled adversary's. In the following example, we've used a case statement to control the logic for each state, and we simply choose a random state after a specific amount of time has elapsed. While viewing the animation, you can definitely tell a difference in the sprite's movement behavior among the various states.

**Listing 12-4:** Movement behavior through a finite state machine

---

```
type
    .
    .
    .

TSpriteStates = (ssChasing, ssEvading, ssRandomMoves, ssPatternMoves, ssWait,
                 ssPlayer);

TStateNames = array[ssChasing..ssWait] of string;

TSprite = class
    XPos, YPos: Integer;

    Direction: Integer;

    CurPattern,

    CurPatternIdx: Integer;

    PatternIdxChange,

    CurPatternTimer,

    StateChange,

    StateTimer: Longint;

    State: TSpriteStates;
```

```

    procedure ChangeState;

    procedure Move;

    procedure Draw(Canvas: TCanvas);

end;

TPatterns = array[0..7, 0..31] of Integer;

const

    {this array allows us to display the name of the state}
    StateNames: TStateNames = ('Chasing', 'Evading', 'RandomMoves',
                                'PatternMoves', 'Wait');

    {in this example, pattern movement is implemented as a direction, with 0
     indicating north, or upward movement. the directions increment in a
     clockwise manner}

    Patterns: TPatterns = (
        (0, 0, 0, 1, 1, 1, 7, 7, 7, 7, 7, 7, 1, 1, 1, 1, 1, 1, 7, 7, 7, 7, 7, 7, 7,
        7,
        1, 1, 1, 0, 0, 0),
        (1, 1, 1, 2, 2, 2, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0,
        0,
        2, 2, 2, 1, 1, 1),
        (2, 2, 2, 3, 3, 3, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1,
        1,
        3, 3, 3, 2, 2, 2),
        (3, 3, 3, 4, 4, 4, 2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2,
        2,
        4, 4, 4, 3, 3, 3),
        (4, 4, 4, 5, 5, 5, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 3, 3, 3, 3, 3, 3, 3,
        3,
        5, 5, 5, 4, 4, 4),
        (5, 5, 5, 6, 6, 6, 4, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4,
        4,
        6, 6, 6, 5, 5, 5),

```

```
(6, 6, 6, 7, 7, 7, 5, 5, 5, 5, 5, 5, 7, 7, 7, 7, 7, 5, 5, 5, 5, 5, 5, 5,
5,
7, 7, 7, 6, 6, 6),
(7, 7, 7, 0, 0, 0, 6, 6, 6, 6, 6, 6, 0, 0, 0, 0, 0, 6, 6, 6, 6, 6, 6, 6,
6,
0, 0, 0, 7, 7, 7)
);
```

```
{these indicate sprite positions in the array of sprites}
```

```
ENEMY1IDX = 0;
```

```
ENEMY2IDX = 1;
```

```
PLAYERIDX = 2;
```

```
var
```

```
.  
.
.
```

```
{tracks all of the sprites}
```

```
Sprites: array[ENEMY1IDX..PLAYERIDX] of TSprite;
```

```
implementation
```

```
.  
.
.
```

```
procedure TfrmDXAppMain.FormActivate(Sender: TObject);
```

```
begin
```

```
.  
.
```

```

{seed the random number generator}

Randomize;

{initialize the enemy sprites}

Sprites[ENEMY1IDX] := TSprite.Create;
Sprites[ENEMY1IDX].XPos := 15;
Sprites[ENEMY1IDX].YPos := 240;
Sprites[ENEMY1IDX].PatternIdxChange := 100;
Sprites[ENEMY1IDX].State := ssRandomMoves;
Sprites[ENEMY1IDX].StateChange := 500;
Sprites[ENEMY1IDX].StateTimer := timeGetTime;
Sprites[ENEMY1IDX].Direction := 2;

Sprites[ENEMY2IDX] := TSprite.Create;
Sprites[ENEMY2IDX].XPos := 625;
Sprites[ENEMY2IDX].YPos := 240;
Sprites[ENEMY2IDX].PatternIdxChange := 50; // this sprite will change pattern
                                           // steps faster than the other...

Sprites[ENEMY2IDX].State := ssRandomMoves;
Sprites[ENEMY2IDX].StateChange := 250; // ...as well as states
Sprites[ENEMY2IDX].StateTimer := timeGetTime;
Sprites[ENEMY2IDX].Direction := 6;

{initialize the player sprite}

Sprites[PLAYERIDX] := TSprite.Create;
Sprites[PLAYERIDX].XPos := 320;
Sprites[PLAYERIDX].YPos := 240;
Sprites[PLAYERIDX].State := ssPlayer;

```

```
.  
.br/>.br/>  
end;
```

```
.  
.br/>.br/>  
procedure TfrmDXAppMain.DrawSurfaces;
```

```
var
```

```
    SrfcDc: HDC;
```

```
    TempCanvas: TCanvas;
```

```
    iCount: Integer;
```

```
begin
```

```
    {erase the last frame of animation}
```

```
    ColorFill(FBackBuffer, clBlack, nil);
```

```
    {move the player sprite}
```

```
    if (GetAsyncKeyState(VK_LEFT) and $8000) = $8000 then
```

```
        Sprites[PLAYERIDX].XPos := Sprites[PLAYERIDX].XPos - 2;
```

```
    if (GetAsyncKeyState(VK_RIGHT) and $8000) = $8000 then
```

```
        Sprites[PLAYERIDX].XPos := Sprites[PLAYERIDX].XPos + 2;
```

```
    if (GetAsyncKeyState(VK_UP) and $8000) = $8000 then
```

```
        Sprites[PLAYERIDX].YPos := Sprites[PLAYERIDX].YPos - 2;
```

```
    if (GetAsyncKeyState(VK_DOWN) and $8000) = $8000 then
```

```
        Sprites[PLAYERIDX].YPos := Sprites[PLAYERIDX].YPos + 2;
```

```
    {move all sprites. note that this simply performs screen boundary clipping
```

```
        for the player sprite, but we could have any number of enemy sprites}
```

```
    for iCount := ENEMY1IDX to PLAYERIDX do
```

```

    Sprites[iCount].Move;

{retrieve a device context and create a canvas}
FBackBuffer.GetDC(SrfcDc);
TempCanvas := TCanvas.Create;
try
    {set up the canvas}
    TempCanvas.Handle := SrfcDc;
    TempCanvas.Font.Color := clWhite;

    {draw the sprites}
    for iCount := ENEMY1IDX to PLAYERIDX do
        Sprites[iCount].Draw(TempCanvas);

    {indicate the current state for each enemy sprite}
    TempCanvas.Brush.Style := bsClear;
    TempCanvas.TextOut(0, 0, 'Enemy1 State:' +
                        StateNames[Sprites[ENEMY1IDX].State]);
    TempCanvas.TextOut(500, 0, 'Enemy2 State: ' +
                        StateNames[Sprites[ENEMY2IDX].State]);

finally
    {cleanup}
    TempCanvas.Handle := 0;
    FBackBuffer.ReleaseDC(SrfcDc);
    TempCanvas.Free;

end;
end;

.
.
.

```

```

{ TSprite }

procedure TSprite.ChangeState;
begin
    {set the state randomly}
    State := TSpriteStates(Random(5));

    {if this is the pattern move state...}
    if (State = ssPatternMoves) then
    begin
        {choose a random pattern}
        CurPattern := Random(8);

        {initialize to the start of the pattern}
        CurPatternIdx := 0;

        {retrieve the current time}
        CurPatternTimer := timeGetTime;

        {32 steps in a pattern multiplied by the time per each step should
        give enough time to complete the entire pattern}
        StateChange := PatternIdxChange * 32;
    end
    else
        {...otherwise, change state again in 1 to 3 seconds}
        StateChange := Random(2000)+1000;

        {begin timing the state}
        StateTimer := timeGetTime;
    end;

```

```

procedure TSprite.Draw(Canvas: TCanvas);
begin
    {draw enemy sprites in blue, player sprites in red}
    if State <> ssPlayer then
        Canvas.Brush.Color := clBlue
    else
        Canvas.Brush.Color := clRed;

    {draw a small dot for the sprite}
    Canvas.Pen.Style := psClear;
    Canvas.Brush.Style := bsSolid;
    Canvas.Ellipse(XPos, YPos, XPos + 3, YPos + 3);
end;

procedure TSprite.Move;
const
    {initialize a horizontal and vertical velocity}
    XVel: Integer = 1;
    YVel: Integer = 1;
begin
    {if this sprite is an enemy sprite...}
    if State <> ssPlayer then
        begin
            {if it is time to change the state, then change it}
            if timeGetTime - StateTimer > StateChange then
                ChangeState;

            {for each state...}
            case State of
                ssChasing      : begin
                    {we're chasing the player, so move the sprite

```



```

        closer to the player sprite's position}
    if Sprites[PLAYERIDX].XPos > XPos then
        XPos := XPos + 1;
    if Sprites[PLAYERIDX].XPos < XPos then
        XPos := XPos - 1;
    if Sprites[PLAYERIDX].YPos > YPos then
        YPos := YPos + 1;
    if Sprites[PLAYERIDX].YPos < YPos then
        YPos := YPos - 1;
    end;
ssEvading      : begin
    {we're evading the player, so move the sprite
    farther away from the player sprite's position}
    if Sprites[PLAYERIDX].XPos > XPos then
        XPos := XPos - 1;
    if Sprites[PLAYERIDX].XPos < XPos then
        XPos := XPos + 1;
    if Sprites[PLAYERIDX].YPos > YPos then
        YPos := YPos - 1;
    if Sprites[PLAYERIDX].YPos < YPos then
        YPos := YPos + 1;
    end;
ssRandomMoves  : begin
    {we're moving randomly, but we don't want to move
    each frame of animation or the sprite will look
    like it's having a seizure}
    if timeGetTime-CurPatternTimer>PatternIdxChange*3 then
    begin
        {modify the horizontal and vertical velocity}
        XVel := Random(3) - 1;
        YVel := Random(3) - 1;

```

```

        {reset the timer}

        CurPatternTimer := timeGetTime;

    end;

    {add the current velocity to the horizontal and
    vertical positions}

    XPos := XPos + XVel;

    YPos := YPos + YVel;

    end;

ssPatternMoves : begin

    {if it is time to change the current pattern step
    index...}

    if timeGetTime-CurPatternTimer > PatternIdxChange then
    begin

        {increment to the next step}

        CurPatternIdx := CurPatternIdx + 1;

        {if we've reached the end of this pattern...}

        if CurPatternIdx > 31 then
        begin

            {choose a new pattern}

            CurPattern := Random(8);

            {initialize to the first step of the pattern}

            CurPatternIdx := 0;

        end;

        {reset the pattern timer}

        CurPatternTimer := timeGetTime;

    end;

```

{as stated above, pattern movement is implemented as a direction, so modify the sprite's position according to the direction at this pattern's movement step}

case Patterns[CurPattern, CurPatternIdx] of

0 : begin

    XPos := XPos + 1;

end;

1 : begin

    XPos := XPos + 1;

    YPos := YPos + 1;

end;

2 : begin

    YPos := YPos + 1;

end;

3 : begin

    XPos := XPos - 1;

    YPos := YPos + 1;

end;

4 : begin

    XPos := XPos - 1;

end;

5 : begin

    XPos := XPos - 1;

    YPos := YPos - 1;

end;

6 : begin

    YPos := YPos - 1;

end;

7 : begin

    XPos := XPos + 1;

    YPos := YPos - 1;

```
                end;
            end;
        end;

        end;

end;

{clip to the boundaries of the screen}

if XPos < 0 then

    XPos := 0;

if XPos > DXWIDTH - 3 then

    XPos := DXWIDTH - 3;

{we'll keep sprites from moving up where the status is displayed}

if YPos < 20 then

    YPos := 20;

if YPos > DXHEIGHT - 3 then

    YPos := DXHEIGHT - 3;

end;
```

---

Implementing artificial intelligence as a finite state machine makes it easy to specify some very complex behavior and add new behavior if necessary. When coupled with some logic to control state changes based on the current game state, such as hit points, power, player hit points, etc., enemy sprites can appear to be very intelligent indeed.

However, the current implementation lacks personality. Enemy sprites may exhibit some fairly complex thought patterns, but they all think alike, and that can soon become boring to players. What is needed is a way to imbue enemy sprites with a little more character and individualism.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by John Ayres  
Wordware Publishing, Inc.  
ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

**Probability Machines**

That's where probability machines come in. A probability machine is really just an extension of a finite state machine, allowing the application to simulate personalities based on a certain distribution factor. Using this technique, we can make some sprites appear more passive, some more aggressive, some a little bit sneakier, etc.

It works by choosing the next state based on a weighted distribution as identified by the type of personality we're modeling. For example, let's say we have five states, and we want to model an aggressive behavior. Whenever it comes time to change states, we want to choose the next state based on a percentage chance as illustrated below.

*Table 12-1: Aggressive personality percentage chance for state selection*

State	Percentage
Chase	50%
Evade	10%
Wait	10%
Random Movement	10%
Patterned Movement	20%

So, in this example, an aggressive personality would choose the chase state 50% of the time. A more timid personality would likely choose the evade state more often. Other personality types would result in varying distributions.

One way to implement such a system would be to create an array of 10 elements, where each element holds a state (either as an integer index or an enumerated type). This array would then be populated by each state a number of times equal to its probability of selection divided by 10. Over time, each state would be selected according to its probability. For example, if the chase state had a 50% chance of selection, it would be in the array five times, and over time, it would indeed have a 50% chance of being selected when the state changed. This is the method by which personalities are implemented in the following example.

*Listing 12-5: Personalities through probability machines*

type

.  
. .  
.

{this defines the different states}

```
TSpriteStates = (ssChasing, ssEvading, ssRandomMoves, ssPatternMoves, ssWait,  
                ssPlayer);
```

```
TStateNames = array[ssChasing..ssWait] of string;
```

{this defines the different personalities}

```
TPersonalities = (perAggressive, perDefensive, perTactical);
```

```
TPersonalityNames = array[perAggressive..perTactical] of string;
```

{this defines the probabilities tables that implement the personalities}

```
TProbabilityTable = array[0..9] of TSpriteStates;
```

```
TPersonality = array[TPersonalities] of TProbabilityTable;
```

{our sprite class}

```
TSprite = class
```

```
    XPos, YPos: Integer;
```

```
    CurPattern,
```

```
    CurPatternIdx: Integer;
```

```
    PatternIdxChange,
```

```
    CurPatternTimer,
```

```
    StateChange,
```

```
    StateTimer,
```

```
    PersonalityChange,
```

```
    PersonalityTimer: Longint;
```

```
    State: TSpriteStates;
```

```
    Personality: TPersonalities;
```

```
procedure ChangeState;

procedure Move;

procedure Draw(Canvas: TCanvas);

end;
```

```
TPatterns = array[0..7, 0..31] of Integer;
```

```
const
```

```
{these probability tables implement the behavior of each personality.
each state is entered into each table a number of times equal to its
percentage chance of selection divided by 10 (i.e., 50% chance of selection =
5 times in the table)}
```

```
AggressivePersonality: TProbabilityTable = (ssChasing, ssChasing, ssChasing,
                                             ssChasing, ssChasing, ssChasing,
                                             ssPatternMoves, ssPatternMoves,
                                             ssRandomMoves, ssWait);
```

```
DefensivePersonality: TProbabilityTable = (ssEvading, ssEvading, ssEvading,
                                             ssEvading, ssEvading, ssEvading,
                                             ssRandomMoves, ssRandomMoves,
                                             ssPatternMoves, ssWait);
```

```
TacticalPersonality: TProbabilityTable = (ssEvading, ssEvading, ssRandomMoves,
                                             ssRandomMoves, ssWait, ssWait,
                                             ssPatternMoves, ssPatternMoves,
                                             ssChasing, ssChasing);
```

```
{this allows us to easily display the name of the current
state and personality}
```

```
StateNames: TStateNames = ('Chasing', 'Evading', 'RandomMoves',
                             'PatternMoves', 'Wait');
```

```
PersonalityNames: TPersonalityNames = ('Aggressive', 'Defensive', 'Tactical');
```

{this table implements patterned movement. moves are stored as directions  
in this example, with 0 equal to north (or straight up movement) incrementing  
in a clockwise direction}

```
Patterns: TPatterns = (  
  (0, 0, 0, 1, 1, 1, 7, 7, 7, 7, 7, 7, 1, 1, 1, 1, 1, 1, 7, 7, 7, 7, 7, 7, 7,  
  7,  
    1, 1, 1, 0, 0, 0),  
  (1, 1, 1, 2, 2, 2, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0,  
  0,  
    2, 2, 2, 1, 1, 1),  
  (2, 2, 2, 3, 3, 3, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1,  
  1,  
    3, 3, 3, 2, 2, 2),  
  (3, 3, 3, 4, 4, 4, 2, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2,  
  2,  
    4, 4, 4, 3, 3, 3),  
  (4, 4, 4, 5, 5, 5, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 3, 3, 3, 3, 3, 3, 3,  
  3,  
    5, 5, 5, 4, 4, 4),  
  (5, 5, 5, 6, 6, 6, 4, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4,  
  4,  
    6, 6, 6, 5, 5, 5),  
  (6, 6, 6, 7, 7, 7, 5, 5, 5, 5, 5, 5, 7, 7, 7, 7, 7, 7, 5, 5, 5, 5, 5, 5, 5,  
  5,  
    7, 7, 7, 6, 6, 6),  
  (7, 7, 7, 0, 0, 0, 6, 6, 6, 6, 6, 6, 0, 0, 0, 0, 0, 0, 6, 6, 6, 6, 6, 6, 6,  
  6,  
    0, 0, 0, 7, 7, 7)  
);
```

{these indicate sprite positions in the array of sprites}

```
ENEMYIDX = 0;
```

```
PLAYERIDX = 1;
```

```
var
```



.  
. .  
.

{track the sprites}

Sprites: array[ENEMYIDX..PLAYERIDX] of TSprite;

{our personality table}

Personalities: TPersonality;

implementation

.  
. .  
.

procedure TfrmDXAppMain.FormDestroy(Sender: TObject);

begin

.  
. .  
.

{free the sprite objects}

Sprites[ENEMYIDX].Free;

Sprites[PLAYERIDX].Free

end;

procedure TfrmDXAppMain.FormActivate(Sender: TObject);

begin

```
.  
.br/>.
```

```
{seed the random number generator}
```

```
Randomize;
```

```
{initialize the enemy sprite}
```

```
Sprites[ENEMYIDX] := TSprite.Create;
```

```
Sprites[ENEMYIDX].XPos := 320;
```

```
Sprites[ENEMYIDX].YPos := 240;
```

```
Sprites[ENEMYIDX].State := ssWait;
```

```
Sprites[ENEMYIDX].StateChange := 10;
```

```
Sprites[ENEMYIDX].StateTimer := timeGetTime;
```

```
Sprites[ENEMYIDX].PatternIdxChange := 100;
```

```
Sprites[ENEMYIDX].Personality := perAggressive;
```

```
Sprites[ENEMYIDX].PersonalityChange := 25000;
```

```
Sprites[ENEMYIDX].PersonalityTimer := timeGetTime;
```

```
{initialize the player sprite}
```

```
Sprites[PLAYERIDX] := TSprite.Create;
```

```
Sprites[PLAYERIDX].XPos := 320;
```

```
Sprites[PLAYERIDX].YPos := 478;
```

```
Sprites[PLAYERIDX].State := ssPlayer;
```

```
.  
.br/>.
```

```
end;
```

```

.
.
.

procedure TfrmDXAppMain.DrawSurfaces;

var
    SrfcDc: HDC;
    TempCanvas: TCanvas;
    iCount: Integer;

begin
    {erase the last frame of animation}
    ColorFill(FBackBuffer, clBlack, nil);

    {move the player sprite based on which cursor keys are depressed}
    if (GetAsyncKeyState(VK_LEFT) and $8000) = $8000 then
        Sprites[PLAYERIDX].XPos := Sprites[PLAYERIDX].XPos - 2;
    if (GetAsyncKeyState(VK_RIGHT) and $8000) = $8000 then
        Sprites[PLAYERIDX].XPos := Sprites[PLAYERIDX].XPos + 2;
    if (GetAsyncKeyState(VK_UP) and $8000) = $8000 then
        Sprites[PLAYERIDX].YPos := Sprites[PLAYERIDX].YPos - 2;
    if (GetAsyncKeyState(VK_DOWN) and $8000) = $8000 then
        Sprites[PLAYERIDX].YPos := Sprites[PLAYERIDX].YPos + 2;

    {move both sprites (note that this simply clips the player sprite to
    the screen boundaries, as the player sprite's position has already
    been updated)}
    for iCount := ENEMYIDX to PLAYERIDX do
        Sprites[iCount].Move;

    {retrieve a device context and create a canvas}
    FBackBuffer.GetDC(SrfcDc);

```

```

TempCanvas := TCanvas.Create;

try
    {use the canvas for easy drawing functionality}

    TempCanvas.Handle := SrfcDc;

    TempCanvas.Font.Color := clWhite;

    {draw the sprites}

    for iCount := ENEMYIDX to PLAYERIDX do

        Sprites[iCount].Draw(TempCanvas);

    {display the current state and personality of the enemy sprite}

    TempCanvas.Brush.Style := bsClear;

    TempCanvas.TextOut(0, 0, 'Personality: '+
                        PersonalityNames[Sprites[ENEMYIDX].Personality]);

    TempCanvas.TextOut(0, 15, 'State: '+StateNames[Sprites[ENEMYIDX].State]);

finally
    {clean up}

    TempCanvas.Handle := 0;

    FBackBuffer.ReleaseDC(SrfcDc);

    TempCanvas.Free;

end;

end;

.
.
.

procedure TSprite.ChangeState;

var

    Distance: Integer;

begin

    {set the state randomly based on the personality}

```

```

State := Personalities[Personality][Random(10)];

{if this is the tactical state...}
if Personality = perTactical then
begin
    {determine the distance to the player}
    Distance := Abs(XPos - Sprites[PLAYERIDX].XPos)+
                Abs(YPos - Sprites[PLAYERIDX].YPos);

    {if the player is too close, we'll evade the player}
    if Distance < 100 then
        State := ssEvading
    else
        {if the player is too far away, we'll move toward the player}
        if Distance > 300 then
            State := ssChasing
        end;
end;

{if this is the pattern move state...}
if (State = ssPatternMoves) then
begin
    {choose a random pattern}
    CurPattern := Random(8);

    {initialize to the start of the pattern}
    CurPatternIdx := 0;

    {retrieve the current time}
    CurPatternTimer := timeGetTime;

    {32 steps in a pattern multiplied by the time per each step should

```

```

        give enough time to complete the entire pattern}

    StateChange := PatternIdxChange * 32;

end

else

    {...otherwise, change state again in 1 to 3 seconds}

    StateChange := Random(2000)+1000;

    {begin timing the state}

    StateTimer := timeGetTime;

end;

procedure TSprite.Draw(Canvas: TCanvas);

begin

    {draw enemy sprites in blue, player sprites in red}

    if State <> ssPlayer then

        Canvas.Brush.Color := clBlue

    else

        Canvas.Brush.Color := clRed;

    {draw a small dot for the sprite}

    Canvas.Pen.Style := psClear;

    Canvas.Brush.Style := bsSolid;

    Canvas.Ellipse(XPos, YPos, XPos + 3, YPos + 3);

end;

procedure TSprite.Move;

const

    {initialize a horizontal and vertical velocity}

    XVel: Integer = 1;

    YVel: Integer = 1;

begin

    {if this sprite is an enemy sprite...}

```

```

if State <> ssPlayer then
begin
  {if it is time to change the personality, then change it}
  if timeGetTime - PersonalityTimer > PersonalityChange then
  begin
    {change to the next personality}
    if Personality = perTactical then
      Personality := perAggressive
    else
      Personality := Succ(Personality);

    {reset the personality change timer}
    PersonalityTimer := timeGetTime;
  end;

  {if it is time to change the state, then change it}
  if timeGetTime - StateTimer > StateChange then
    ChangeState;

  {for each state...}
  case State of
    ssChasing      : begin
                        {we're chasing the player, so move the sprite
                        closer to the player sprite's position}
                        if Sprites[PLAYERIDX].XPos > XPos then
                          XPos := XPos + 1;
                        if Sprites[PLAYERIDX].XPos < XPos then
                          XPos := XPos - 1;
                        if Sprites[PLAYERIDX].YPos > YPos then
                          YPos := YPos + 1;
                        if Sprites[PLAYERIDX].YPos < YPos then

```

```

        YPos := YPos - 1;

    end;

ssEvading      : begin

    {we're evading the player, so move the sprite
     farther away from the player sprite's position}

    if Sprites[PLAYERIDX].XPos > XPos then

        XPos := XPos - 1;

    if Sprites[PLAYERIDX].XPos < XPos then

        XPos := XPos + 1;

    if Sprites[PLAYERIDX].YPos > YPos then

        YPos := YPos - 1;

    if Sprites[PLAYERIDX].YPos < YPos then

        YPos := YPos + 1;

    end;

ssRandomMoves  : begin

    {we're moving randomly, but we don't want to move
     each frame of animation or the sprite will look
     like it's having a seizure}

    if timeGetTime - CurPatternTimer > PatternIdxChange*3

        then

            begin

                {modify the horizontal and vertical velocity}

                XVel := Random(3) - 1;

                YVel := Random(3) - 1;

                {reset the timer}

                CurPatternTimer := timeGetTime;

            end;

    {add the current velocity to the horizontal and
     vertical positions}

    XPos := XPos + XVel;

```



```

        YPos := YPos + YVel;

    end;

ssPatternMoves : begin

    {if it is time to change the current pattern step
    index...}

    if timeGetTime-CurPatternTimer > PatternIdxChange then
    begin

        {increment to the next step}

        CurPatternIdx := CurPatternIdx + 1;

        {if we've reached the end of this pattern...}

        if CurPatternIdx > 31 then
        begin

            {choose a new pattern}

            CurPattern := Random(8);

            {initialize to the first step of the pattern}

            CurPatternIdx := 0;

        end;

        {reset the pattern timer}

        CurPatternTimer := timeGetTime;

    end;

    {as stated above, pattern movement is implemented
    as a direction, so modify the sprite's position
    according to the direction at this pattern's
    movement step}

    case Patterns[CurPattern, CurPatternIdx] of

        0 : begin

            XPos := XPos + 1;

```

```
        end;
1 : begin
        XPos := XPos + 1;
        YPos := YPos + 1;
        end;
2 : begin
        YPos := YPos +1;
        end;
3 : begin
        XPos := XPos - 1;
        YPos := YPos + 1;
        end;
4 : begin
        XPos := XPos - 1;
        end;
5 : begin
        XPos := XPos - 1;
        YPos := YPos - 1;
        end;
6 : begin
        YPos := YPos - 1;
        end;
7 : begin
        XPos := XPos + 1;
        YPos := YPos - 1;
        end;
        end;
end;

end;

{clip to the boundaries of the screen}
```

```
if XPos < 0 then
    XPos := 0;
if XPos > DXWIDTH - 21 then
    XPos := DXWIDTH - 21;

{we'll keep sprites from moving up where the status is displayed}
if YPos < 30 then
    YPos := 30;
if YPos > DXHEIGHT - 3 then
    YPos := DXHEIGHT - 3;
end;
```

```
initialization
```

```
{set up the personality tables}
Personalities[perAggressive] := AggressivePersonality;
Personalities[perDefensive] := DefensivePersonality;
Personalities[perTactical] := TacticalPersonality;
end.
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

As you can see from this simplistic example, the change in behavior of the enemy sprite is subtle, but you should be able to tell a definite personality, especially between the Aggressive and Defensive postures. Indeed, if we changed a sprite's personality based on game state (i.e., going from Aggressive to Defensive when hit points went below 30%), the sprite would seem a little more dynamic.

We could even change the distribution based on the success of a particular state within a personality during the game. For example, if the game timed how long it took to kill the player during each state and found that a particular state was more effective, it could modify the distribution so that that state was chosen more often. This would make the game grow with the player; the sprites would appear to "learn," and the game would become more challenging.

## Path Finding

Most games feature some form of entity or sprite that must move from one position in the game world to another. Usually, this is pretty simple, as we can determine an appropriate x and y velocity that will move the sprite in the direction of its goal and sit back until it gets there. This will work fine for game worlds that are flat and don't feature any obstacles that would hinder movement. Unfortunately, most game worlds tend to be a bit more complex, with several immobile or impassible objects. The presence of these objects negates the possibility of game entities always moving in straight lines to get from their origin to their destination, posing a problem that has been researched heavily over the years by both the academic world and the gaming industry alike.

When a game entity encounters an object that hinders its movement, it must find another direction of movement leading around the obstacle and toward the final destination. This is known as path finding. Technically, path finding describes algorithms for determining the shortest, most direct path from a starting position to an ending position, avoiding all impassible obstacles.



**path finding:** Algorithmic techniques for determining the shortest, most direct path from a starting position to an ending position based on terrain and obstacles in the game world.

There are numerous applications for such techniques. Real-time strategy games such as Age of Empires and WarCraft make heavy use of path finding algorithms. Indeed, any game that features entities moving through a world filled with impassible obstacles needs to implement some form of path finding. Path finding algorithms can be applied to real-world problems as well. We'll focus our discussion on path finding as it applies to 2-D game worlds, and we'll examine one well-known algorithm that produces acceptable results.

## Algorithms

The subject of finding a particular path over a specific terrain and algorithms for accomplishing this task have been researched almost as extensively as methods for producing graphical output. Several books have been published that cover this subject to varying degrees of depth. Many of these algorithms, while accurate, tend to be too slow for implementation in a game. However, three algorithms are well documented and well known in the game industry, and tend to serve as the basis from which most game implementations of path finding algorithms are drawn. These are known as breadth first search, depth first search, and A\* (pronounced "A-Star").

### Breadth First Search

The breadth first search algorithm is very exhaustive, and while it can eventually find a very accurate, short path, it typically takes way too long for most games. It searches for the shortest path by choosing the next node in a particular direction and determining its distance from the destination. It performs this for all available directions, then it moves to the node closest to the destination and performs these calculations again. However, if two or more nodes are the same distance from the destination, it recursively performs these calculations for each of these nodes. Essentially, this algorithm explores all possible paths from the starting point to the destination, as long as those paths lead toward the destination. This can result in an exponentially increasing number of calculations depending on the size of the map and the position of obstacles. Calculation times for this algorithm are typically measured in seconds. While this may be completely appropriate for real-world problems, it is way too slow for a game application, which needs to calculate paths in milliseconds, if not faster.

### Depth First Search

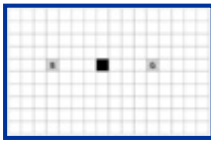
By contrast, the depth first search simply chooses a direction toward the destination and begins moving along nodes in that direction until it reaches the destination. If it encounters an obstacle, it backs up one node and tries again in a different direction. While this algorithm can eventually find some path to the destination, it typically will not be the shortest or the most direct route. However, also unlike breadth first search techniques, path calculation times are much, much shorter. While this will find some type of path in a short amount of time, it will usually result in a path that makes game entities wander about almost aimlessly, and certainly in no logical or intelligent manner.

### A\*

Perhaps the most widely documented solution for path finding problems is the A\* algorithm. A\* combines the best of both breadth first search and depth first search to determine the shortest path in a relatively minimal amount of time. Like the depth first search, it chooses a direction toward the destination, and begins moving along nodes in that direction. Like the breadth first search algorithm, it checks the distance of the next nodes in several directions to determine which is closest to the destination. It then moves to this next node and starts the calculations again. The result is a single path for which several directions were considered at each node. This typically produces a fairly intelligent looking path within an acceptable amount of time.

### Basic Implementation

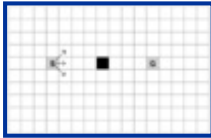
In our example, we'll take a fairly typical approach and define a two-dimensional game world composed of square tiles. We'll need to define a starting point and an ending point, and populate this map with several impassible obstacles, like so:



**Figure 12-5:** The game world

Additionally, we'll need to set up a list that will hold information for the nodes we have selected on our path. We'll also need an array the size of the game world to mark nodes that have already been explored. We'll examine how these two requirements are utilized below.

We begin by determining the direction of the goal relative to our starting point. We'll use this to determine which nodes we should look at next. Based on this direction, we'll examine three nodes—the next node in the direction of the goal and the two nodes to either side of it. In this example, our goal is directly east of the starting point, so we'll examine the northeast, east, and southeast nodes relative to our starting point:



**Figure 12-6:** Examining nodes

Now, for each of these nodes being examined, we need to determine their distance from the destination. If you remember your algebra, the distance formula is:

```
Distance := Sqrt(Sqr(X2 - X1) + Sqr(Y2 - Y1));
```



**Tip:** The Sqrt and Sqr functions are very slow. We could simply use multiplication in the case of the Sqr function, and perhaps utilize a lookup table for the Sqrt function in order to optimize this equation.

Using this equation, we find that the node directly to the east is the closest to the destination. We'll add this new node to our list of nodes along the path, as well as mark this new node as visited in our array of visited nodes (we also need to add the starting node to the list as well as mark it as visited before we start searching for a path). We then move to this new node, and begin the process again.



**Note:** The phrase “moving to the new node” indicates that we are simply tracking the current node we are processing in our algorithm. The actual movement of the game entity does not take place until the entire path is determined; it does not move in real time.

[Previous](#) [Table of Contents](#) [Next](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**KEYWORD SEARCH**

- [Search Tips](#)
- [Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



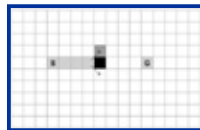
**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

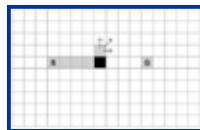
[Previous](#) [Table of Contents](#) [Next](#)

We'll continue this process until we encounter an obstacle. Even though the next eastward node would be closest to the destination, it is an impassible obstacle and cannot be selected. Therefore, we need to select one of the other two nodes that are closest to the destination and are not blocked. In this example, either of these nodes will work just fine, so we'll pick one at random.



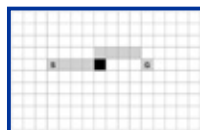
**Figure 12-7:** Moving past an obstacle

Notice that we have now changed directions from east to northeast, which affects which nodes we'll examine on the next pass. Specifically, the next nodes we will examine will now be north, northeast, and east of the current node.



**Figure 12-8:** Examining nodes in a new direction

Once again, we'll see that the eastward node is the closest to the goal. This entire process continues until the goal is finally reached.

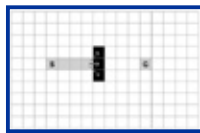


**Figure 12-9:** A path from start to finish

### Back Stepping

This basic algorithm will get us there in simple game worlds, but what about situations where we run into larger obstacles, such as the one below.



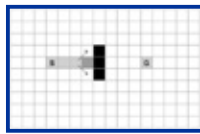


**Figure 12-10:** A large wall

As we can see here, all three examined nodes are blocked, and thus the path cannot proceed any further. This is where the array that tracks which nodes have been visited comes in. We need to delete the current node we are on from our list of path nodes, and back up to the previous node. When we reexamine the next closest nodes, we'll treat the node we just visited as an impassible obstacle. This keeps us from choosing a node that we know will not work.

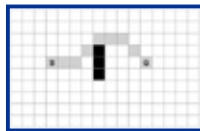


**Note:** When we back up to a previous node, we also need to reset our direction to the direction we were facing in this previous node.



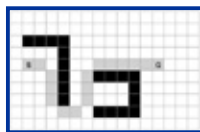
**Figure 12-11:** Backing up and reexamining nodes

Continuing with this process, we see that our algorithm finds a relatively direct path to the destination.



**Figure 12-12:** Pathing around a large wall

Backing up and reexamining nodes in the previous direction really boosts this algorithm's intelligence. Indeed, while our implementation of this algorithm isn't totally infallible, it is pretty smart, and can get us around some fairly complex obstacles as illustrated below.



**Figure 12-13:** Navigating a complex game world

## Case Study

Unlike most examples throughout this book, the path finding example is implemented using standard Delphi controls for the visual feedback. This was done to highlight the path finding algorithm and to make it easier to examine the algorithm at run time. You can't really set breakpoints and step through code once DirectDraw has exclusive access and has set the video mode to full screen. Implementing an A\* path finding algorithm using standard Delphi controls allows you to step through it line by line to examine what's happening as it runs. This will also make it easier to improve the algorithm's performance and accuracy if you wish to play with it. In the output from this test application, the blue square is the origin, the red square is the destination, black squares indicate impassible obstacles, and purple squares indicate the final path. Green squares indicate nodes that were visited but did not belong on the final path.

The following listing demonstrates a simple, straightforward implementation of the A\* algorithm.

**Listing 12-6:** *The A\* test bed*

---

interface

.  
. .  
.

{defines the directions}

```
TCompassDirection = (cdNorth, cdNorthEast, cdEast, cdSouthEast, cdSouth,  
                    cdSouthWest, cdWest, cdNorthWest);
```

{this defines node offsets relative to the current node for  
a particular direction}

```
TDirectionOffset = array[TCompassDirection] of TPoint;
```

{our node record, recording its coordinates and compass direction}

```
TNode = record
```

```
    Direction: TCompassDirection;
```

```
    GridPt: TPoint;
```

```
end;
```

```
PNode = ^TNode;
```

```
procedure ClearPathQueue;
```

```
var
```

```
    frmPathFinder: TfrmPathFinder;
```

{tracks the origin and destination}

```
StartPt, EndPt: TPoint;
```

```
SetStart: Boolean;
```

{tracks the list of nodes on the final path}

```
PathQueue: TList;
```

```

{defines the game world. nodes defined as 0 are clear,
 1 are obstacles}

MapGrid: array[0..14, 0..14] of Byte;

{tracks which nodes we've already visited}

VisitedNodes: array[0..14, 0..14] of Boolean;

const

  {this defines the actual node offsets at a specific direction
   relative to the current node}

  DirectionOffset: TDirectionOffset = ((X: 0; Y: -1),
                                         (X: 1; Y: -1),
                                         (X: 1; Y: 0),
                                         (X: 1; Y: 1),
                                         (X: 0; Y: 1),
                                         (X: -1; Y: 1),
                                         (X: -1; Y: 0),
                                         (X: -1; Y: -1) );

  {define each of our node types}

  NODECLEAR      = '';
  NODEOBSTACLE   = '1';
  NODESTART      = '2';
  NODEEND        = '3';
  NODEPATH       = '4';
  NODEVISITED    = '5';

implementation

procedure TfrmPathFinder.FormCreate(Sender: TObject);
begin

```

```
{initialize the starting and ending node}

StartPt := Point(-1, -1);

EndPt := Point(-1, -1);

{indicate that the left mouse button sets the start node by default}

SetStart := TRUE;

{create our list of path nodes}

PathQueue := TList.Create;

end;

procedure TfrmPathFinder.FormDestroy(Sender: TObject);
begin
    {free all existing nodes, and then free the path list}

    ClearPathQueue;

    PathQueue.Free;

end;

procedure ClearPathQueue;
var
    iCount: Integer;
begin
    {free each node in the list}
    for iCount := 0 to PathQueue.Count-1 do
        FreeMem(PathQueue[iCount], SizeOf(TNode));

    {empty the list}

    PathQueue.Clear;

end;
```

```

procedure TfrmPathFinder.grdPathMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
var
    GrdRow, GrdCol: Integer;
begin
    {map the mouse cursor coordinates to grid cell coordinates}
    grdPath.MouseToCell(X, Y, GrdCol, GrdRow);

    {if the right mouse button was pressed...}
    if Button = mbRight then
    begin
        {...set or clear an obstacle node}
        if grdPath.Cells[GrdCol, GrdRow] = NODEOBSTACLE then
            grdPath.Cells[GrdCol, GrdRow] := NODECLEAR
        else
            grdPath.Cells[GrdCol, GrdRow] := NODEOBSTACLE;
        end
    else
        {...otherwise, we're setting a starting or ending node}
        if SetStart then
        begin
            {if a start node has already been placed, clear the previous node}
            if StartPt.X <> -1 then
                grdPath.Cells[StartPt.X, StartPt.Y] := NODECLEAR;

                {set the new start node}
                grdPath.Cells[GrdCol, GrdRow] := NODESTART;
                StartPt := Point(GrdCol, GrdRow);
            end
        else
        begin
            {if an end node has already been placed, clear the previous node}

```

```

    if EndPt.X <> -1 then
        grdPath.Cells[EndPt.X, EndPt.Y] := NODECLEAR;

        {set the new end node}
        grdPath.Cells[GrdCol, GrdRow] := NODEEND;
        EndPt := Point(GrdCol, GrdRow);
    end;

end;

procedure TfrmPathFinder.grdPathDrawCell(Sender: TObject; ACol, ARow: Integer;
    Rect: TRect; State: TGridDrawState);
begin
    {most squares should be colored white}
    grdPath.Canvas.Brush.Color := clWhite;

    {determine the color of the square based on its type}
    if grdPath.Cells[ACol, ARow] = NODEOBSTACLE then
        grdPath.Canvas.Brush.Color := clBlack;
    if grdPath.Cells[ACol, ARow] = NODESTART then
        grdPath.Canvas.Brush.Color := clBlue;
    if grdPath.Cells[ACol, ARow] = NODEEND then
        grdPath.Canvas.Brush.Color := clRed;
    if grdPath.Cells[ACol, ARow] = NODEPATH then
        grdPath.Canvas.Brush.Color := clPurple;
    if grdPath.Cells[ACol, ARow] = NODEVISITED then
        grdPath.Canvas.Brush.Color := clGreen;

    {fill this square with the appropriate color}
    grdPath.Canvas.FillRect(Rect);
end;

```

```

procedure TfrmPathFinder.shpStartPointMouseDown(Sender: TObject;
    Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    {indicate we are setting a start node}
    SetStart := TRUE;
end;

procedure TfrmPathFinder.shpDestPointMouseDown(Sender: TObject;
    Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    {indicate we are setting an end node}
    SetStart := FALSE;
end;

procedure TfrmPathFinder.btnClearMapClick(Sender: TObject);
var
    iCount, iCount2: Integer;
begin
    {clear out the map}
    for iCount := 0 to 14 do
        for iCount2 := 0 to 14 do
            grdPath.Cells[iCount, iCount2] := '';
        end;
    end;
end;

procedure TfrmPathFinder.btnFindPathClick(Sender: TObject);
var
    iCount, iCount2: Integer;

    CurPt, EvalPt, NewPt: TPoint;

```

```

TempNode: PNode;

Dist, EvalDist: Longword;

Dir, NewDir: TCompassDirection;

SearchDirs: array[0..2] of TCompassDirection;

begin
  {don't do anything until a starting and ending point have been set}
  if (StartPt.X = -1) or (EndPt.X = -1) then
    Exit;

  {clear out the visited nodes array}
  FillChar(VisitedNodes, SizeOf(VisitedNodes), 0);

  {populate the map grid with the specified impassible tiles}
  for iCount := 0 to 14 do
    for iCount2 := 0 to 14 do
      if grdPath.Cells[iCount, iCount2] = NODEOBSTACLE then
        MapGrid[iCount, iCount2] := 1
      else
        MapGrid[iCount, iCount2] := 0;

  {delete the current path}
  ClearPathQueue;

  {initialize the tracking variables}
  CurPt := StartPt;
  VisitedNodes[CurPt.X, CurPt.Y] := TRUE;

  {determine the initial direction}

  {destination left of origin}

```



```
if EndPt.X < StartPt.X then
begin
    if EndPt.Y > StartPt.Y then
        Dir := cdSouthWest
    else
        if EndPt.Y < StartPt.Y then
            Dir := cdNorthWest
        else
            Dir := cdWest;
    end
else
    {destination right of origin}
    if EndPt.X > StartPt.X then
    begin
        if EndPt.Y > StartPt.Y then
            Dir := cdSouthEast
        else
            if EndPt.Y < StartPt.Y then
                Dir := cdNorthEast
            else
                Dir := cdEast;
        end
    end
else
    {destination directly above or below origin}
    if EndPt.Y > StartPt.Y then
        Dir := cdSouth
    else
        Dir := cdNorth;

    {create a node object}
    GetMem(TempNode, SizeOf(TNode));
```

```

{initialize the node object with our current (starting) node information}
TempNode^.Direction := Dir;
TempNode^.GridPt.X := CurPt.X;
TempNode^.GridPt.Y := CurPt.Y;

{add this starting node to the path list}
PathQueue.Add(TempNode);

{begin searching the path until we reach the destination node}
while (CurPt.X <> EndPt.X) or (CurPt.Y <> EndPt.Y) do
begin
    {reset the new coordinates to indicate nothing has been found}
    NewPt := Point(-1, -1);

    {reset our distance to the largest value available (new nodes should
    be well under this distance to the destination)}
    Dist := $FFFFFFFF;

    {determine the 3 search directions}
    SearchDirs[0] := Pred(Dir);
    if Ord(SearchDirs[0]) < Ord(cdNorth) then
        SearchDirs[0] := cdNorthWest;
    SearchDirs[1] := Dir;
    SearchDirs[2] := Succ(Dir);
    if Ord(SearchDirs[2]) > Ord(cdNorthWest) then
        SearchDirs[2] := cdNorth;

    {evaluate grid locations in 3 directions}
    for iCount := 0 to 2 do
    begin
        {get the coordinates of the next node to examine relative to the

```

```

current node, based on the direction we are facing }

EvalPt.X := CurPt.X + DirectionOffset[SearchDirs[iCount]].X;
EvalPt.Y := CurPt.Y + DirectionOffset[SearchDirs[iCount]].Y;

{make sure this node is on the map}

if (EvalPt.X > -1) and (EvalPt.X < 15) and
    (EvalPt.Y > -1) and (EvalPt.Y < 15) then
    {make sure we've never visited this node}
    if not VisitedNodes[EvalPt.X, EvalPt.Y] then
        {make sure this isn't an impassible node}
        if MapGrid[EvalPt.X, EvalPt.Y] = 0 then
            begin
                {this is a clear node that we could move to. calculate
                the distance from this node to the destination.

                NOTE: since we're interested in just relative distances as
                opposed to exact distances, we don't need to perform a square
                root. this will dramatically speed up this calculation}
                EvalDist := ((EndPt.X - EvalPt.X)*(EndPt.X - EvalPt.X)) +
                    ((EndPt.Y - EvalPt.Y)*(EndPt.Y - EvalPt.Y));

                {if we have found a node closer to the destination, make this
                the current node}
                if EvalDist < Dist then
                    begin
                        {record the new distance and node}
                        Dist := EvalDist;
                        NewPt := EvalPt;

                        {record the direction of this new node}
                        NewDir := SearchDirs[iCount];
                    end
            end
        end
    end
end

```

```

        end;

end;

{at this point, if NewPt is still (-1, -1), we've run into a wall and
cannot move any further. thus, we have to back up one and try again.
otherwise, we can add this new node to our list of nodes}

{we've got a valid node}
if NewPt.X <> -1 then
begin
    {make this new node our current node}
    CurPt := NewPt;

    {point us in the direction of this new node}
    Dir := NewDir;

    {mark this node as visited}
    VisitedNodes[CurPt.X, CurPt.Y] := TRUE;

    {create a node object}
    GetMem(TempNode, SizeOf(TNode));

    {initialize this node object with the new node information}
    TempNode^.Direction := Dir;
    TempNode^.GridPt.X := CurPt.X;
    TempNode^.GridPt.Y := CurPt.Y;

    {put this new node in the path list}
    PathQueue.Add(TempNode);

    {if we've recorded 50 nodes, break out of this loop}

```

```

    if PathQueue.Count > 50 then
        Break;
    end
else
begin
    {we've backed up to the point where we can no longer back up. thus, a
    path could not be found. we could improve this algorithm by
    recalculating the starting direction and trying again until we've
    searched in all possible directions}

    if PathQueue.Count = 1 then
        Break;

        {point us in the direction of the previous node}
        Dir := TNode(PathQueue[PathQueue.Count-2]^).Direction;

        {retrieve the coordinates of the previous node and make that
        our current node}
        CurPt := Point(TNode(PathQueue[PathQueue.Count-2]^).GridPt.X,
            TNode(PathQueue[PathQueue.Count-2]^).GridPt.Y);

        {free the last node in the list and delete it}
        FreeMem(PathQueue[PathQueue.Count-1], SizeOf(TNode));
        PathQueue.Delete(PathQueue.Count-1);
    end;
end;

{at this point, we've found a path from the starting node to the destination
node, so we should populate the grid and allow it to display the final path}

{specify visited nodes}
for iCount := 0 to 14 do
    for iCount2 := 0 to 14 do

```

```

    if VisitedNodes[iCount, iCount2] then

        grdPath.Cells[iCount, iCount2] := NODEVISITED;

{specify nodes on the path}

for iCount := 0 to PathQueue.Count-1 do

begin

    grdPath.Cells[TNode(PathQueue[iCount]^).GridPt.X,

                    TNode(PathQueue[iCount]^).GridPt.Y] := NODEPATH;

end;

{specify the starting and ending nodes}

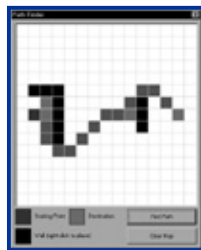
grdPath.Cells[StartPt.X, StartPt.Y] := NODESTART;

grdPath.Cells[EndPt.X, EndPt.Y] := NODEEND;

end;

```

---



**Figure 12-14:** The A\* algorithm in action

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



# IT KNOWLEDGE.COM<sup>SM</sup>

Need IT. Find IT. Know IT.

Monday, January 12, 2004

## KEYWORD SEARCH

- Search Tips
- Advanced Search

## PUBLICATION LOOKUP

## JUMP TO TOPIC



**Desktop Library**  
[Click to access!](#)



### Delphi Graphics and Game Programming Exposed with DirectX 7.0

by John Ayres

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

### Enhancements

Although this algorithm is relatively powerful, it is easy to bog it down and send it on a wild goose chase. In particular, large U-shaped sections of obstacles can make the algorithm calculate dozens or even hundreds of nodes that will not be part of the final path. This could potentially overflow the list of path nodes, or at a minimum cause a noticeable pause when a unit calculates a path over long distances. Therefore, your game world should avoid such constructions.



**Caution:** This implementation could potentially get the sprite caught in an infinite loop on some maps, resulting in an overflow situation which would most likely crash the machine.

One way to avoid an overflow on the path list would be to restrict the number of nodes in this list to some arbitrary number. Our example constrains the path to a total number of 50 nodes. This may need to be adjusted for large maps, but it will ensure that the path node list never overflows, and it will restrict pathing time to a maximum amount. If a unit tries to path to a destination, but the path node list reaches the limit before the destination is reached, you could just set a flag for that entity, and repath when the unit reaches the last node in its list. Allowing the user to set waypoints for a unit will further improve pathing, as the paths between each waypoint tend to be shorter and encounter fewer obstacles.

This algorithm is somewhat intelligent, but it can be easily fooled by certain obstacle constructions. Looking at nodes in the three directions relative to the direction of the destination is somewhat of an optimization, and cuts down on the number of calculations required at each node. However, this sometimes leads to paths that look less than intelligent, even downright stupid. The algorithm could be changed so that it checks all eight directions at each node. While this will dramatically increase the number of calculations, the overall path will tend to be much shorter and look more natural.

Additionally, this algorithm does not take into account nodes with different movement costs. For example, say that the game world was populated by terrain consisting of hard rock, grasslands, mud, swamp, and water. While water terrain might be impassible, the others can be moved through but at different degrees of

difficulty. Hard rock or roads would certainly be the easiest to move over, but grasslands, mud, and swamp would all impose penalties to movement while not being totally impassible. Elevation could also be a factor. This algorithm would need to be modified somewhat if your game world features terrain with varying movement costs.

## Summary

In this chapter, we discussed techniques for implementing simplistic artificial intelligence routines that give our sprites the ability to display cunning and strategy when moving around the game world. We looked at different movement techniques, including chasing or evading the player and patterned movement. We also looked at controlling the movement decisions based on a finite state machine, as well as techniques for weighting the selected states as a method for implementing rudimentary personalities. When using artificial intelligence techniques in your games, it is important to keep these points in mind:

- No matter how good your artificial intelligence algorithms may be, there is no greater challenge than playing another human being. However, some people may be unable to hook up with someone else to play a multiplayer game, or they may just enjoy playing by themselves. Therefore, it is important to include artificial intelligence algorithms in your games that make the antagonists at least moderately challenging.
- The most noticeable display of artificial intelligence is in the movement of enemy units. In most action games, the user is going to be primarily concerned with the movement of enemy units, such as whether they are aggressively chasing the user's ship or running like cowards.
- Perhaps the simplest simulation of artificial intelligence is to introduce an element of randomness. When an enemy unit moves in a fashion that is unpredictable to the player, that unit can appear to exhibit a rudimentary intelligence. Consider the apparently random flight path of a common housefly. It darts about here and there, yet it always appears to have a mind of its own and to know where it's going.
- An enemy that actively hunts down the player can be perceived as confident, powerful, and aggressive. By contrast, an enemy that constantly eludes the player or turns tail can be perceived as cowardly, injured, or perhaps even cunning, like an assassin stalking its target.
- We can use algorithms that make enemy units move about in a specific pattern. This can make antagonists appear to move in a very deliberate manner. Very complex movements can be modeled this way, where some patterns could be useful for evading the player, such as a zigzag pattern, and some patterns could be useful for chasing the player, such as a circular pattern to move around the player's flank.
- A finite state machine is a logical construction that consists of several "states," each of which defines a specific behavior, and "state transitions," which define specific rules for moving from state to state. Finite state machines are a method by which an enemy unit can make a series of rational decisions based on various inputs from the game world, such as the current environment as well as the actions of the player.
- A probability machine is an extension of a finite state machine, allowing the application to simulate personalities based on a certain distribution factor. Using this technique, we can make some sprites appear more passive, some more aggressive, some a little bit sneakier, etc. It works by choosing the next state based on a weighted distribution as identified by the type of personality we're modeling.
- Most games feature a game world filled with moving sprites which must navigate around immovable or impassible obstacles. Path finding artificial intelligence routines must be utilized in order to make the sprite appear to move in an intelligent manner, selecting the most direct, shortest path available.
- Breadth first search and depth first search techniques could be used, but these are typically too slow or too inaccurate for game applications. The A\* (pronounced "A-star") algorithm combines the best of breadth first search and depth first search to find a fairly direct, short path in a short amount of time.

[Previous](#) [Table of Contents](#) [Next](#)



Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## CHAPTER 13 Putting It All Together

This chapter covers the following topics:

- The Delphi Blocks game architecture
- User input implementation
- Sound and music implementation
- Graphics implementation
- Potential enhancements

Each individual piece of a game is interesting to study and can be downright fun to program. Sound and music programming is always interesting, as it's cool to programmatically control a sound's properties such as volume or panning, and perhaps even more interesting to output music from various sources. Input programming is also a lot of fun, especially when reading from devices other than the standard mouse or keyboard. Force feedback is especially cool, as it's unusual to control an external peripheral using code instead of the other way around. And, of course, graphics programming is never boring, and is limited to only what your imagination can stir up. However, while each piece is interesting individually, orchestrating them together into a digital masterpiece is incredibly rewarding and entertaining.

We've covered many useful game programming techniques throughout this book, and now it's time to put them together and make something useful. In this chapter, we will examine a full-fledged game written using the techniques and examples that we've developed throughout the course of the text. We'll break down several important aspects of the game, and see how all of these parts come together to make a working, playable game application.

### Delphi Blocks

Our case study game is called Delphi Blocks. If you remember the old Breakout game, this one will be very familiar. While we're not breaking any new technological ground with this application, it does serve as a working example of bringing graphics, sound, and input programming together to make a working, playable,

and even fun game.

The basic idea is to destroy all of the blocks on the screen using a ball. This ball bounces around the screen, off of blocks and the screen sides. The user can control a small paddle at the bottom of the screen to deflect the ball toward the blocks. If the ball goes off of the bottom of the screen, it is lost and another ball is put into play. The user has three balls; when all of them are lost, the game is over. The game play is simple, but it is easy to learn without ever consulting a manual and can be addictive.

## General Game Architecture

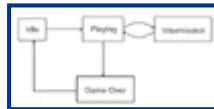
This is a simplistic game with an equally simplistic architecture. The code is encapsulated into one unit, although we could have easily segregated several portions of the code into individual units in order to increase its readability and adaptability. In general, we decided to make the overall game level based to facilitate a more intuitive player advancement system, using a state-driven architecture to ease implementation and simplify game control.

## Player Advancement

As stated above, the player's goal is to destroy every block on the screen. We'll use a level based system of advancement, so that when all of the blocks have been destroyed, the level is increased and a new grid of blocks is generated. The score for each block is based on the level, so the higher the level, the more each block is worth. However, the speed of the ball is also based on the level, so at higher levels the ball will be moving at more and more rapid velocities, making the game harder and justifying the score increase for each block. This type of advancement is important, as it challenges the players and forces them to get better in order to play the game for longer periods of time.

## Game States

The game is state driven, using the basic state-driven architecture we covered back in Chapter 2. There are four specific states: Idle, Playing, Intermission, and Game Over. Game execution flows through the states in the following manner:



**Figure 13-1:** Delphi Blocks state flow

**Idle** This state is basically a demo state. All it does is sit and display a static image with a full grid of blocks and the game title while waiting for the user to start a new game. Unlike the other states, though, in this state we display two buttons, one for starting the game and one for quitting. We'll discuss how the buttons work when we cover user input.

**Playing** This is the main game state, and includes the logic for moving the sprites and performing the graphics rendering. Additionally, it also checks to see if the current level has been cleared, and if so, it initializes a new level and puts the game in intermission.

**Intermission** This state simply provides the player with a short pause between levels. This is important, as a twitch game like Delphi Blocks can get rather hectic at higher levels, and you really need to provide players with an opportunity to catch their breath every now and then. All this state does is display a full grid of blocks with an indicator of the next level.

**Game Over** The game goes into this state when the player has lost all of the available balls. It serves to indicate to the user that the game has been completed, and a new one must be started. Like the intermission state, it simply displays a full grid of blocks with the words "Game Over" superimposed over them.

## User Input

During the idle state, the user has the choice of selecting two buttons, one for starting a game and one for quitting the application. While actually playing the game, the user has control over a paddle that can move left and right. At higher levels, the user needs to be able to move the paddle rapidly and intuitively. Thus, the most obvious form of control for this type of game would be the mouse. We've used a modification of the baseline application with mouse support in order to accomplish this task. Using this baseline, it was very easy to get mouse support up and running with little effort.

## Buttons

The buttons were actually very easy to implement. They are displayed only during the idle state, and it is in this idle state that we check for click events. If a click is detected, we simply check to see if the mouse cursor was located over one of the buttons, and take the appropriate action:

```
{determine if the mouse was clicked in one of the buttons}

if MouseClickList.Count>0 then

begin

    {while there are any events in the mouse click list, retrieve
    them all and empty the list}

    while MouseClickList.Count > 0 do

    begin

        {free the current mouse data}

        if MouseData <> nil then

        begin

            MouseData.Free;

            MouseData := nil;

        end;

        {retrieve and delete the click event}

        MouseData := TMouseData(MouseClickList[0]);

        MouseClickList.Delete(0);

    end;

    {if the quit button was pushed...}

    if PtInRect(Rect(416, 350, 544, 414),

                Point(MouseData.XPos, MouseData.YPos)) then

        Close;
```

```
{if the start button was pushed...}  
  
if PtInRect(Rect(96, 350, 224, 414),  
  
    Point(MouseData.XPos, MouseData.YPos)) then  
  
    StartGame;
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved.  
Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read  
EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 Pub Date: 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Baseline Differences

While the baseline application with mouse support was functional, we needed to make some minor modifications. Specifically, we want to show a mouse cursor during the idle state, but we want to suppress the display of the mouse cursor during any other state. Therefore, we ripped all of the mouse cursor display code out of the mouse input thread and added a variable to control mouse cursor display. If the mouse cursor is to be hidden, only the horizontal position of the mouse cursor is updated, and it is clipped according to the dimensions of the paddle graphic.

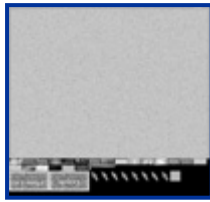
## Sound and Music

Any game can be improved by using sound effects, and this game is certainly no different. We decided to use a few simple sounds to indicate when a ball hits a block, hits the side of the screen, hits the paddle, or falls off of the bottom and dies. There's not much opportunity for sounds to overlap, so we could have used regular Windows multimedia function calls to output the sound. However, we went with DirectSound in case we wanted to expand the sound functionality at a later date. DirectSound programming is relatively easy anyway, and this architecture certainly would support advanced sound functionality if we needed it.

We also wanted to have some music playing in the background, so the application uses MCI commands to continuously play a MIDI file while the application is active. We could have easily used DirectSound to stream a large WAV file to the sound card, which would have given us better sound quality, but this example shows how Windows multimedia functions and DirectSound can be used simultaneously.

## Graphics

The graphical requirements for this application are not very stringent, and were thus very easy to implement. This is a 2-D sprite game that doesn't feature very much animation, so all of the graphics easily fit into one file. We load this bitmap into an offscreen surface, letting DirectDraw determine its location; if there is enough video memory available, the surface will be located on the video card itself. Since we use nothing but simple blits, available hardware acceleration will dramatically improve the performance of the game. The game images are arranged in the single bitmap like so:



**Figure 13-2:** Delphi Blocks game graphics

The largest part of the graphic file is the background screen. The images for the block sprites are located below the background, including the image of the ball and paddle. We also see the button images as well as the cursor sprite image.

## Sprites

This game features a large amount of sprites in the form of the blocks and the ball. We could easily extend the block grid to be larger or smaller, and the game is already set up to support more than one active ball. In order to encapsulate this functionality, we will use an object to manage all of the block sprites, and another object to manage all of the ball sprites.

The block manager itself is pretty simplistic. It contains an array property that holds a 10×15 grid of TBlock objects, with each TBlock object representing one block sprite on the screen. The manager contains functions for initializing the entire block grid as well as drawing all of the blocks or deleting one specific block from the grid. This encapsulation allows us to deal with the large quantity of block sprites using one simple class. It is defined as:

```
{defines the manager that will track all of the blocks}

TBlockMgr = class
    {this static grid can be modified to increase or decrease the overall
    block area and size}

    BlockGrid: array[0..14, 0..9] of TBlock;

    NumBlocks: Integer;

    destructor Destroy; override;

    procedure Draw;

    procedure InitializeBlocks;

    procedure KillBlock(Row, Col: Integer);

end;
```

The ball manager object is very similar in functionality and design. It contains a list object that tracks several TBall objects. This allows us to add any number of active balls to the screen and track them all simultaneously. Additional methods move and draw all of the active balls, clear the entire list of balls, or delete a specific ball sprite. It is defined as:

```
{defines the manager that will track all of the ball sprites. note that
```

```

this gives us the ability to add and track several active balls at once}
TBallMgr = class
    BallList: TList;

    VelCoefficient: Integer;

    FTotalBalls: Integer;

    function GetActiveBalls: Integer;

    constructor Create;
    destructor Destroy; override;
    procedure AddBall(XPos, YPos, XVel, YVel: Real; DecBalls: Boolean);
    procedure KillBall(Idx: Integer);
    procedure MoveBalls;
    procedure DrawBalls;
    procedure Clear;

    property TotalBalls: Integer read FTotalBalls write FTotalBalls;
    property ActiveBalls: Integer read GetActiveBalls;
end;

```

## Collision Detection

Collision detection really is the heart of this game. The entire idea is to make a ball collide with a block, and the balls must bounce off the walls and the paddle in a predictable manner.

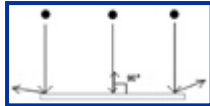
Bouncing off the walls is a relatively simple behavior to implement. We simply check the horizontal and vertical coordinates of a ball after it has been moved, and if it has collided with the sides or top of the screen, its horizontal or vertical velocity is reversed as appropriate. A collision with the bottom of the screen results in a lost ball.

Collision with the paddle is a little more involved. To simplify matters, we use the bounding box collision detection technique covered in Chapter 6. To review, each ball has a bounding rectangle, as does the paddle. When these two rectangles intersect, a collision has occurred. The paddle is rectangular, and the ball is almost rectangular, so this collision detection technique is accurate enough for our purposes.

However, we would like to give the user some control over the ball so he or she can influence its direction.

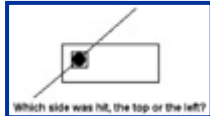


Therefore, the angle of deflection that the ball takes is dependent on the position at which the ball collides with the paddle. Towards the center of the paddle, the ball tends to deflect at a 90-degree angle to the paddle itself, or straight upward. As the ball collides nearer and nearer to the edge of the paddle, the deflection angle becomes more and more acute, as the following illustration shows:



**Figure 13-3:** Ball deflection angle

Collisions with the blocks are where things get kind of hairy. We first perform a bounding box collision test to see if they collided. However, we need to know if the ball hit the top, sides, or bottom of the block in order to determine which velocity to reverse. Unfortunately, it is impossible to determine this from a simple bounding box collision, as illustrated below:



**Figure 13-4:** Which side was hit—the top or the left?

Therefore, we need a more accurate method of collision detection that can determine which side of the box was hit. This can be accomplished by using a general line segment intersection algorithm. The specific algorithm used in the example is covered on page 199 of *Graphics Gems III* by David Kirk. The algorithm was deduced by Franklin Antonio. The math behind this algorithm is complex, and although it is documented thoroughly in the example code, please consult *Graphics Gems III* for a detailed explanation of this algorithm.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN: 1556226373 Pub Date: 12/01/99**

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

**Enhancements**

We've created a complete game application that utilizes the vast majority of the techniques covered throughout the book. However, while it serves as a good case study, it certainly is not ready to be shrink-wrapped and put on store shelves. This is by design, as it leaves many opportunities for you, the reader, to exercise some creativity and take this basic game to the next level.

The first thing that should be enhanced is the collision detection code. While it is very accurate, it is incredibly math intensive, and a better, more optimized method could probably be implemented. We are also redrawing the entire screen each frame, which could be optimized by only drawing the paddle and any active balls in their new positions using dirty rectangle animation techniques.

Some general game enhancements might include the ability to track high scores, or perhaps show some skill statistics during the game over state. Additional balls can easily be activated in response to certain events, as the game already supports such functionality. It would be easy to tie such events to special blocks, where some blocks add another ball or perhaps increase/decrease the size of the paddle. You could even take this one step further and perhaps give the paddle a laser or other such weapon, or even make blocks that are indestructible or take multiple hits to destroy.

Perhaps the easiest modification would be to provide better graphics, or even different graphic sets depending on the level. These could even be themed; you could have an ice level, or a lava level, or even a water level. Techniques from the special effects chapter or the palettes chapter could be used to make some blocks glow, rotate, fade in and out, etc.

From a usability standpoint, it might be good to provide users with a configuration screen when the application starts so they can specify things like sound volume or skill level. This could include such things as a sound theme, utilizing better sound effects or different sound effects based on the user's preference or level. Context-sensitive music would also improve the feel of the game. You might also want to support input devices other than the mouse, such as the keyboard or joystick. Providing force feedback support would really add a new dimension to this game. If you want to provide this type of configuration while in full screen video mode, a more general architecture should probably be created to support user interface objects like buttons and menus.

A very fun enhancement for this game would be to include multiplayer support. One player could be at the bottom, and the other player could be at the top. This would also allow you to use some artificial intelligence techniques if you allowed the user to play against the computer. Using some of the incredible Internet technology now available in Delphi, it wouldn't be too hard to make this game playable over the Internet.

**The Case Study Code**

The case study itself is presented here in its entirety so that you can see an entire game application laid out in all its glory. The code is heavily commented so as to make it easy to understand. This demonstrates how sound, user input, and graphics come together to form a complete, workable, usable game application.

*Listing 13-1: Delphi Blocks*

---

```
unit DelphiBlocksU;
```

```
{*****
```

```
Delphi Blocks
```

```
Author: John Ayres
```

```
Based on the baseline architecture with mouse support, this project is a 'Breakout' style of game. The object is to hit the moving ball with the paddle and destroy all of the blocks on the screen. If the ball goes off of the bottom of the screen, it is lost and a new one is created. You have three balls before the game is over. This is a simple implementation of a game that shows how to use the mouse input functionality of DirectInput in a real-world situation.
```

```
*****
```

```
Copyright (c) 1999 by John Ayres, All Rights Reserved
```

```
This code is freeware and can be used for any personal or commercial purposes without royalty or license. Use at your own risk.
```

```
*****}
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
DDraw, DInput, MouseThreadU, MMSystem, DSound;
```

```
{these constants are used to modify specific attributes of the DirectX  
application, such as the color of the main form, the requested resolution,  
etc.}
```

```
const
```

```
    FORMCOLOR      = clBlack;
```

```
    FORMBORDER     = bsNone;
```

```
    FORMCURSOR     = crNone;
```

```
    DXWIDTH        = 640;
```

```
    DXHEIGHT       = 480;
```

```
    DXCOLORDEPTH   = 8;
```

```
    BUFFERCOUNT   = 1;
```

```
    COOPERATIVELEVEL = DDSCL_FULLSCREEN or DDSCL_ALLOWREBOOT or DDSCL_ALLOWMODEX  
                      or DDSCL_EXCLUSIVE;
```

```
    SURFACETYPE     = DDSCAPS_COMPLEX or DDSCAPS_FLIP or DDSCAPS_PRIMARYSURFACE;
```

```
const
```

```
    {this user-defined message is used to start the flipping loop}
```

```
    WM_DIRECTXACTIVATE = WM_USER + 200;
```

```
type
```

```
    {controls the overall game state}
```

```
    TGameState = (gsIdle, gsPlaying, gsIntermission, gsGameOver);
```

```
    TBallMgr = class;
```

```
    {defines a ball sprite}
```

```
    TBall = class
```

```
        XPrev, YPrev,
```

```
        XPos, YPos,
```

```
XVel, YVel: Real;

BoundingBox: TRect;

Index: Integer;

Manager: TBallMgr;

procedure Move;
procedure Draw;
end;

{defines the manager that will track all of the ball sprites. note that
this gives us the ability to add and track several active balls at once}
TBallMgr = class
    BallList: TList;

    VelCoefficient: Integer;

    FTotalBalls: Integer;

    function GetActiveBalls: Integer;

    constructor Create;
    destructor Destroy; override;
    procedure AddBall(XPos, YPos, XVel, YVel: Real; DecBalls: Boolean);
    procedure KillBall(Idx: Integer);
    procedure MoveBalls;
    procedure DrawBalls;
    procedure Clear;
```

```
property TotalBalls: Integer read FTotalBalls write FTotalBalls;

property ActiveBalls: Integer read GetActiveBalls;

end;

{defines the paddle sprite}

TPaddle = class

    XPos, YPos: Integer;

    BoundBox: TRect;

    procedure Move;

    procedure Draw;

end;

TBlockMgr = class;

{defines a single block}

TBlock = class

    GraphicIdx: Integer;

    BoundBox: TRect;

    procedure Draw;

end;

{defines the manager that will track all of the blocks}

TBlockMgr = class

    {this static grid can be modified to increase or decrease the overall
    block area and size}

    BlockGrid: array[0..14, 0..9] of TBlock;

    NumBlocks: Integer;
```

```

    destructor Destroy; override;

    procedure Draw;

    procedure InitializeBlocks;

    procedure KillBlock(Row, Col: Integer);

end;

{the primary form class}

TfrmDXAppMain = class(TForm)

    procedure FormCreate(Sender: TObject);

    procedure FormDestroy(Sender: TObject);

    procedure FormKeyDown(Sender: TObject; var Key: Word;

        Shift: TShiftState);

    procedure FormActivate(Sender: TObject);

private

    { Private declarations }

    {necessary for our mouse input}

    FMouseThread: TMouseThread;

    {the block manager}

    FBlockMgr: TBlockMgr;

    {the player's paddle sprite}

    FPaddle: TPaddle;

    {the ball manager}

    FBallMgr: TBallMgr;

    {tracks the current level}

    FLevel: integer;

```

```
{tracks the score}
```

```
FScore: integer;
```

```
{we will need to add an active ball both when the current ball dies as  
well as when the user completes a level and starts a new one. thus,  
sometimes we want to add a ball without decreasing the amount of balls  
available. this variable will track this state}
```

```
FUseBall: Boolean;
```

```
{the current game state}
```

```
FGameState: TGameState;
```

```
{the MCI device ID used for MIDI playback}
```

```
MCIDeviceID: UINT;
```

```
{the MIDI notify message handler}
```

```
procedure MCINotify(var Msg: TMessage); message MM_MCINOTIFY;
```

```
{flips back to the GDI surface to display the exception error message}
```

```
procedure ExceptionHandler(Sender: TObject; ExceptionObj: Exception);
```

```
{the main rendering loop}
```

```
procedure AppIdle(Sender: TObject; var Done: Boolean);
```

```
{intercepts certain messages to provide appropriate functionality}
```

```
procedure AppMessage(var Msg: TMsg; var Handled: Boolean);
```

```
{flips the DirectX surfaces}
```

```
procedure FlipSurfaces;
```

```
{restores any lost surfaces}
```

```
procedure RestoreSurfaces;
```



```
    {draws the contents of surfaces}

    procedure DrawSurfaces;

public
    { Public declarations }

    procedure InitLevel;

    procedure StartGame;

end;

procedure PlayGameSound(SoundNum: Integer);

var

    frmDXAppMain: TfrmDXAppMain;

    {the main DirectDraw interface}
    FDirectDraw: IDirectDraw4;

    {the interfaces for the primary and backbuffer surfaces}
    FPrimarySurface,
    FBackBuffer,
    FMouseCursors,           // holds mouse cursor graphics
    FGraphics: IDirectDrawSurface4; // holds the game graphics

    {the overall palette}
    FPalette: IDirectDrawPalette;

    {mouse cursor animation variables}
    FCursorFrame, FCursorFrameCount: Integer;

    {DirectInput objects for mouse functionality}
    FDirectInput: IDirectInput;
    FMouseDevice: IDirectInputDevice;
```

```
{DirectSound object}
```

```
DXSound: IDirectSound;
```

```
{an array of sound buffers}
```

```
Sounds: array[0..3] of IDirectSoundBuffer;
```

```
const
```

```
{these are used in defining events for the mouse input functionality}
```

```
MOUSEEVENT = 0;
```

```
QUITEVENT = 1;
```

```
{these identify the sounds in our sound array}
```

```
BLOCKHIT = 0;
```

```
LOSTBALL = 1;
```

```
SIDEHIT = 2;
```

```
PADDLEHIT = 3;
```

```
{the names of the wave files to be used}
```

```
FileNames: array[BLOCKHIT..PADDLEHIT] of string = ('bhump.wav',  
                                                    'bigboom.wav',  
                                                    'hitside.wav',  
                                                    'bwik.wav');
```

```
implementation
```

```
uses
```

```
DXTools, DDUtil, SyncObjs, Math, DSUtil{***}, gensuppt;
```

```
{$R *.DFM}
```

```

{ *-->>  BASELINE APPLICATION CODE <<--* }

{ - the callback function used to ensure that the selected graphics mode -
  - is supported by DirectX - }

function EnumModesCallback(const EnumSurfaceDesc: TDDSsurfaceDesc2;
                           Information: Pointer): HRESULT; stdcall;

begin
  {if the height, width, and color depth match those specified in the
    constants, then indicate that the desired graphics mode is supported}
  if (EnumSurfaceDesc.dwHeight = DXHEIGHT) and
      (EnumSurfaceDesc.dwWidth = DXWIDTH) and
      (EnumSurfaceDesc.ddpfPixelFormat.dwRGBBitCount = DXCOLORDEPTH) then
    Boolean(Information^) := TRUE;

  Result := DDENUMRET_OK;
end;

{ -> Events Hooked to the Application <- }

{ - this event is called when an exception occurs, and simply flips back -
  - to the GDI surface so that the exception dialog box can be read - }

procedure TfrmDXAppMain.ExceptionHandler(Sender: TObject;
                                         ExceptionObj: Exception);

begin
  {disconnect the OnIdle event to shut off the rendering loop}
  Application.OnIdle := nil;

  {if the DirectDraw object was successfully created, flip to the GDI surface}

```

```

if Assigned(FDirectDraw) then

    FDirectDraw.FlipToGDISurface;

{display the exception message}
MessageDlg(ExceptionObj.Message, mtError, [mbOK], 0);

{reconnect the OnIdle event to reenter the rendering loop}
Application.OnIdle := AppIdle;
end;

{ - handles certain messages that are required to make DirectX function -
  - properly within Delphi - }
procedure TfrmDXAppMain.AppMessage(var Msg: TMsg; var Handled: Boolean);
begin
    case Msg.Message of

        WM_ACTIVATEAPP:

            {unhook the OnIdle event when the application is being deactivated.
              this will stop all rendering}

            if not Boolean(Msg.wParam) then
                begin
                    FMouseDevice.Unacquire;

                    Application.OnIdle := nil;

                end
            else

                {upon activating the application, send ourselves the
                  user-defined message}

                PostMessage(Application.Handle, WM_DIRECTXACTIVATE, 0, 0);

        WM_DIRECTXACTIVATE:

            begin

                {upon activating, restore all surfaces (reloading their memory
                  as necessary), hook up the OnIdle event, and redraw the contents
                  of all surfaces}

```

```

        FMouseDevice.Acquire;

        RestoreSurfaces;

        Application.OnIdle := AppIdle;

        DrawSurfaces;

    end;

WM_SYSCOMMAND:

    begin

        {do not allow a screen saver to kick in}

        Handled := (Msg.wParam = SC_SCREENSAVE);

    end;

end;

end;

{ -> Form Events <- }

{ - initialize essential form properties - }

procedure TfrmDXAppMain.FormCreate(Sender: TObject);

begin

    {set up the application exception handler}

    Application.OnException := ExceptionHandler;

    {initialize form properties.  note that the FormStyle property must be
    set to fsStayOnTop}

    BorderStyle := bsNone;

    BorderIcons := [];

    FormStyle := fsStayOnTop;

    Color := clBlack;

    Cursor := crNone;

    {initialize the mouse cursor to the top left of the screen}

```

```
CurX := 0;

CurY := 0;

{hide the actual Windows mouse cursor}

ShowCursor(FALSE);

end;

{ - provides essential cleanup functionality - }

procedure TFormDXAppMain.FormDestroy(Sender: TObject);

var

    iCount: Integer;

begin

    {disengage our custom exception handler}

    Application.OnException := nil;

    {remember, we do not have to explicitly free the DirectDraw objects,
    as they will free themselves when they go out of context (such as
    when the application is closed)}

    {unacquire the mouse device}

    FMouseDevice.Unacquire;

    {fire the quit event so our mouse input thread will exit}

    SetEvent(FMouseEvents[QUITEVENT]);

    Sleep(100);

    {free any remaining data in the mouse click list}

    for iCount := 0 to MouseClickList.Count-1 do

        TMouseData(MouseClickList[iCount]).Free;

    {free the click list}

    MouseClickList.Free;
```

```

    {free our manager objects}

    FBallMgr.Free;

    FBlockMgr.Free;

    FPaddle.Free;

    {make sure that the MIDI device is closed before exiting the program}

    MCISendCommand(MCIDeviceID, MCI_CLOSE, 0, 0);

end;

{ - this method initializes DirectX and creates all necessary objects - }

procedure TfrmDXAppMain.FormActivate(Sender: TObject);

var

    {we can only get a DirectDraw4 interface from the DirectDraw interface, so we
    need a temporary interface}

    TempDirectDraw: IDirectDraw;

    {structures required for various methods}

    DDSurface: TDDSurfaceDesc2;

    DDSCaps: TDDSCaps2;

    {flag used to determine if the desired graphics mode is supported}

    SupportedMode: Boolean;

    {mouse property structure}

    MouseProp: TDIPropDWord;

    SrcRect: TRect;

    iCount: Integer;

    {necessary structures for initializing MIDI output}

```

```

OpenParms: TMCI_Open_Parms;

StatusParms: TMCI_Status_Parms;

SetParms: TMCI_Set_Parms;

PlayParms: TMCI_Play_Parms;

begin

    {if DirectDraw has already been initialized, exit}

    if Assigned(FDirectDraw) then exit;

    {create a temporary DirectDraw object. this is used to create the
     desired DirectDraw4 object}

    DXCheck( DirectDrawCreate(nil, TempDirectDraw, nil) );

    try

        {we can only get a DirectDraw4 interface through the QueryInterface
         method of the DirectDraw object}

        DXCheck( TempDirectDraw.QueryInterface(IID_IDirectDraw4, FDirectDraw) );

    finally

        {now that we have the DirectDraw4 object, the temporary DirectDraw
         object is no longer needed}

        TempDirectDraw := nil;

    end;

    {set the cooperative level to that defined in the constants}

    DXCheck( FDirectDraw.SetCooperativeLevel(Handle, COOPERATIVELEVEL) );

    {hook up the application message handler}

    Application.OnMessage := AppMessage;

    {call EnumDisplayModes and verify that the desired graphics mode is
     indeed supported}

    FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

    DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

```



```

DDSurface.dwFlags := DDSD_HEIGHT or DDSD_WIDTH or DDSD_PIXELFORMAT;

DDSurface.dwHeight := DXHEIGHT;

DDSurface.dwWidth := DXWIDTH;

DDSurface.ddpfPixelFormat.dwSize := SizeOf(TDDPixelFormat_DX6);

DDSurface.ddpfPixelFormat.dwRGBBitCount := DXCOLORDEPTH;

SupportedMode := FALSE;

DXCheck( FDirectDraw.EnumDisplayModes(0, @DDSurface, @SupportedMode,
                                     EnumModesCallback) );

{if the desired graphics mode is not supported by the DirectX drivers,
 display an error message and shut down the application}

if not SupportedMode then
begin
    MessageBox(Handle, PChar('The installed DirectX drivers do not support a '+
                             'display mode of: '+IntToStr(DXWIDTH)+' X '+
                             IntToStr(DXHEIGHT)+' ', '+IntToStr(DXCOLORDEPTH)+
                             ' bit color'), 'Unsupported Display Mode Error',
               MB_ICONERROR or MB_OK);

    Close;

    Exit;
end;

{set the display resolution and color depth to that defined in the constants}
DXCheck( FDirectDraw.SetDisplayMode(DXWIDTH, DXHEIGHT, DXCOLORDEPTH, 0, 0) );

{initialize the DDSurface structure to indicate that we will be creating a
 complex flipping surface with one back buffer}

FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);

DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

DDSurface.dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;

DDSurface.ddsCaps.dwCaps := SURFACETYPE;

```

```
DDSurface.dwBackBufferCount := BUFFERCOUNT;
```

```
{create the primary surface object}
```

```
DXCheck( FDirectDraw.CreateSurface(DDSurface, FPrimarySurface, nil) );
```

```
{indicate that we want to retrieve a pointer to the back buffer (the surface  
immediately behind the primary surface in the flipping chain) }
```

```
FillChar(DDSCaps, SizeOf(TDDSCaps2), 0);
```

```
DDSCaps.dwCaps := DDSCAPS_BACKBUFFER;
```

```
{retrieve the surface}
```

```
DXCheck( FPrimarySurface.GetAttachedSurface(DDSCaps, FBackBuffer) );
```

```
{at this point, offscreen buffers and other surfaces should be created. other  
DirectDraw objects should be created and initialized as well, such as  
palettes. the contents of all surfaces should also be initialized at this  
point}
```

```
{load the palette for the game's images}
```

```
FPalette := DDLoadPalette(FDirectDraw, ExtractFilePath(ParamStr(0))+  
                          'Images.bmp');
```

```
{attach this palette to the primary surface}
```

```
FPrimarySurface.SetPalette(FPalette);
```

```
{load the actual graphics used in the game}
```

```
FGraphics := DDLoadBitmap(FDirectDraw, ExtractFilePath(ParamStr(0))+  
                          'Images.bmp');
```

```
{initialize the surface description structure to create the mouse  
cursor surface}
```

```
FillChar(DDSurface, SizeOf(TDDSurfaceDesc2), 0);
```

```

DDSurface.dwSize := SizeOf(TDDSurfaceDesc2);

DDSurface.dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;

DDSurface.dwWidth := 224;

DDSurface.dwHeight := 32;

DDSurface.ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;

{create the mouse cursor surface}

DXCheck( FDirectDraw.CreateSurface(DDSurface, FMouseCursors, nil) );

{copy the mouse cursor graphics into the mouse cursor surface}

SrcRect := Rect(256, 520, 256+(6*32)+32, 552);

FMouseCursors.BltFast(0, 0, FGraphics, SrcRect, DDBLTFAST_NOCOLORKEY or
                    DDBLTFAST_WAIT);

{set the transparent color in both the mouse cursor graphics as well as
the game graphics}

DDSetColorKey(FMouseCursors, $00000000);

DDSetColorKey(FGraphics, $00000000);

{now that all of DirectDraw is initialized, begin initializing DirectInput}

{create our mouse click list object}

MouseClickedList := TList.Create;

{create the DirectInput object}

DXCheck(DirectInputCreate(hInstance, DIRECTINPUT_VERSION, FDirectInput, nil));

{create a DirectInput device for the mouse}

DXCheck( FDirectInput.CreateDevice(GUID_SysMouse, FMouseDevice, nil) );

{set the appropriate data format for mouse data}

```

```
DXCheck( FMouseDevice.SetDataFormat(c_dfDIMouse) );

{set the cooperative level. note that we must set a nonexclusive cooperative
 level for mice as the mouse must be shared with other Windows applications}
DXCheck( FMouseDevice.SetCooperativeLevel(Handle,DISCL_FOREGROUND or
                                             DISCL_NONEXCLUSIVE) );

{initialize the structure to instruct DirectInput that we want an input
 buffer big enough to hold 64 pieces of input information}
MouseProp.diph.dwSize := SizeOf(TDIPropDWord);
MouseProp.diph.dwHeaderSize := SizeOf(TDIPropHeader);
MouseProp.diph.dwObj := 0;
MouseProp.diph.dwHow := DIPH_DEVICE;
MouseProp.dwData := 64;

{set the actual buffer size property}
DXCheck( FMouseDevice.SetProperty(DIPROP_BUFFERSIZE^, MouseProp.diph) );

{create our events, one for indicating new mouse input data, one for
 indicating that the application is terminating}
FMouseEvents[MOUSEEVENT] := CreateEvent(nil, FALSE, FALSE, 'MouseEvent');
FMouseEvents[QUITEVENT] := CreateEvent(nil, FALSE, FALSE, 'QuitEvent');

{create our critical section synchronization object}
FCritSec := TCriticalSection.Create;

{set the event notification}
DXCheck( FMouseDevice.SetEventNotification(FMouseEvents[MOUSEEVENT]) );

{create our secondary input thread}
FMouseThread := TMouseThread.Create(TRUE);
```

```
{we want it to be supremely responsive}

FMouseThread.Priority := tpTimeCritical;

{indicate it should free itself when it terminates}

FMouseThread.FreeOnTerminate := TRUE;

{post a message that will hook up the OnIdle event and start the
  main rendering loop}

PostMessage(Handle, WM_ACTIVATEAPP, 1, 0);

{now, initialize game management objects}

{create the ball manager}

FBallMgr := TBallMgr.Create;

{create and initialize the paddle}

FPaddle := TPaddle.Create;

FPaddle.YPos := 420;

{create and initialize the block manager}

FBlockMgr := TBlockMgr.Create;

FBlockMgr.InitializeBlocks;

{open the MIDI output device, loading the Canyon MIDI song}

OpenParms.lpstrDeviceType := 'sequencer';

OpenParms.lpstrElementName := PChar(ExtractFilePath(ParamStr(0))+
                                     'custom16.mid');

MCI SendCommand(0, MCI_OPEN, MCI_OPEN_TYPE or MCI_OPEN_ELEMENT or MCI_WAIT,
               LongInt(@OpenParms));

{save a handle to the device}

MCIDeviceID := OpenParms.wDeviceID;
```

```

{set the time format to milliseconds}

SetParms.dwTimeFormat := MCI_FORMAT_MILLISECONDS;

MCISendCommand(MCIDeviceID, MCI_SET, MCI_WAIT or MCI_SET_TIME_FORMAT,

               LongInt(@SetParms));

{commence song playback, indicating that we wish to receive an MM_MCINOTIFY
 message when the playback has completed}

PlayParms.dwCallback := Handle;

MCISendCommand(MCIDeviceID, MCI_PLAY, MCI_NOTIFY, LongInt(@PlayParms));

{create the DirectSound object}

DirectSoundCreate(nil, DXSound, nil);

{set its cooperative level (we won't change the primary buffer format, so
 a cooperative level of Normal is all that is needed)}

DXSound.SetCooperativeLevel(Handle, DSSCL_NORMAL);

{load in the sounds}

for iCount := BLOCKHIT to PADDLEHIT do

    Sounds[iCount] := DSLoadSound(DXSound, ExtractFilePath(ParamStr(0))+

                                   FileNames[iCount], FALSE);

{finally, acquire the mouse and fire off the mouse input thread}

FMouseDevice.Acquire;

FMouseThread.Resume;

end;

{ -> Form Methods <- }

```

```

{ - the message handler for the MM_MCINOTIFY message - }

procedure TfrmDXAppMain.MCINotify(var Msg: TMessage);

var

    PlayParms: TMCI_Play_Parms;

begin

    {replay the song from the beginning}

    PlayParms.dwCallback := Handle;

    PlayParms.dwFrom      := 0;

    MCISendCommand(MCIDeviceID, MCI_PLAY, MCI_NOTIFY or MCI_FROM,

                    LongInt(@PlayParms));

end;

{ - this method is called in order to flip the surfaces - }

procedure TfrmDXAppMain.FlipSurfaces;

var

    DXResult: HRESULT;

begin

    {perform the page flip. note that the DDFLIP_WAIT flag has been used,

    indicating that the function will not return until the page flip has been

    performed. this could be removed, allowing the application to perform other

    processing until the page flip occurs. however, the application will need

    to continuously call the Flip method to ensure that the page flip takes

    place}

    DXResult := FPrimarySurface.Flip(nil, DDFLIP_WAIT);

    {pause until the flip has been performed}

    while FPrimarySurface.GetFlipStatus(DDGFS_ISFLIPDONE) <> DD_OK do;

    {if the surfaces were lost, restore them. on any other error,

    raise an exception}

    if DXResult = DDERR_SURFACELOST then

```

```

    RestoreSurfaces

else if DXResult <> DD_OK then

    DXCheck(DXResult);

end;

{ - this method is called when the surface memory is lost -
  - and must be restored.  surfaces in video memory that -
  - contain bitmaps must be reinitialized in this function - }

procedure TfrmDXAppMain.RestoreSurfaces;

var

    SrcRect: TRect;

begin

    {restore the primary surface, which in turn restores any implicit surfaces}
    FPrimarySurface._Restore;

    {reload the game images}
    DDReloadBitmap(FGraphics, ExtractFilePath(ParamStr(0))+`Images.bmp`);

    {restore the mouse cursor surface}
    FMouseCursors._Restore;

    {copy the mouse cursor graphics into the mouse cursor surface}
    SrcRect := Rect(256, 520, 256+(6*32)+32, 552);
    FMouseCursors.BltFast(0, 0, FGraphics, SrcRect, DDBLTFAST_NOCOLORKEY or
        DDBLTFAST_WAIT);

end;

{ - this method is continuously called by the application, and provides -
  - the main rendering loop.  this could be replaced by a custom -
  - while..do loop - }

procedure TfrmDXAppMain.AppIdle(Sender: TObject; var Done: Boolean);

begin

```



```

{indicates that the application should continuously call this method}

Done := FALSE;

{if DirectDraw has not been initialized, exit}
if not Assigned(FDirectDraw) then Exit;

{draw surface content and flip the surfaces. notice that we must use a
critical section here, as we may need to access the mouse cursor coordinate
variables, and we can't do that if the mouse input thread is accessing
them at the same time}

FCritSec.Enter;

DrawSurfaces;

FlipSurfaces;

FCritSec.Leave;

end;

{ - this method is called when the contents of the surfaces need to be -
- drawn. it will be continuously called by the AppIdle method, so any -
- rendering or animation could be done within this method - }

procedure TfrmDXAppMain.DrawSurfaces;

var

    SrcRect: TRect;

    SrfcDC: HDC;

    WorkCanvas: TCanvas;

    MouseData: TMouseData;

    iCount: Integer;

begin

    {copy the background into the back buffer, erasing the last
    frame of animation}

    SrcRect := Rect(0, 0, DXWIDTH, DXHEIGHT);

```

```

FBackBuffer.BltFast(0, 0, FGraphics, SrcRect, DDBLTFAST_NOCOLORKEY or
                    DDBLTFAST_WAIT);

{draw all of the blocks that are still alive}

FBlockMgr.Draw;

{based on the game state...}

case FGameState of

  gsIdle : begin

    {if we are idle, we want to show our animated mouse cursor}

    ShowMouse := TRUE;

    {draw the Start and Quit buttons}

    SrcRect := Rect(0, 520, 128, 584);

    FBackBuffer.BltFast(96, 350, FGraphics, SrcRect,
                        DDBLTFAST_NOCOLORKEY or DDBLTFAST_WAIT);

    SrcRect := Rect(128, 520, 256, 584);

    FBackBuffer.BltFast(416, 350, FGraphics, SrcRect,
                        DDBLTFAST_NOCOLORKEY or DDBLTFAST_WAIT);

    {draw the game title}

    WorkCanvas := TCanvas.Create;

    FBackBuffer.GetDC(SrfdcDC);

    with WorkCanvas do
    begin

      {set up the canvas to use the surface}

      Handle := SrfdcDC;

      {initialize the canvas font attributes}

      Font.Name := 'Arial';

      Font.Color := clRed;

      Font.Style := [fsBold];

```

```
Font.Size := 64;

Brush.Style := bsClear;

{draw the actual title}
TextOut(320-(TextWidth('Delphi Blocks!') div 2), 125,
        'Delphi Blocks!');

{unhook the canvas from the surface}
Handle := 0;
end;

{release the surface device context}
FBackBuffer.ReleaseDC(SrvcDC);

{free the canvas}
WorkCanvas.Free;

{determine if the mouse was clicked in one of the buttons}
if MouseClickList.Count>0 then
begin
    {while there are any events in the mouse click list, retrieve
    them all and empty the list}
    while MouseClickList.Count > 0 do
    begin
        {free the current mouse data}
        if MouseData <> nil then
        begin
            MouseData.Free;
            MouseData := nil;
        end;
    end;
end;
```

```

        {retrieve and delete the click event}

        MouseData := TMouseData(MouseClickList[0]);

        MouseClickList.Delete(0);

    end;

    {if the quit button was pushed...}
    if PtInRect(Rect(416, 350, 544, 414),
        Point(MouseData.XPos, MouseData.YPos)) then
        Close;

    {if the start button was pushed...}
    if PtInRect(Rect(96, 350, 224, 414),
        Point(MouseData.XPos, MouseData.YPos)) then
        StartGame;

    {free the current mouse data object}
    MouseData.Free;

    MouseData := nil;

    end;

end;

gsPlaying : begin

    {move and draw all of the active balls}
    FBallMgr.MoveBalls;

    FBallMgr.DrawBalls;

    {display a ball for each one the player has left}
    SrcRect := Rect(252, 507, 261, 516);

    for iCount := 0 to FBallMgr.TotalBalls-1 do
        FBackBuffer.BltFast(10+iCount*16, 440, FGraphics, SrcRect,
            DDBLTFAST_SRC or DDBLTFAST_COLORKEY or
            DDBLTFAST_WAIT);
    end;
end;

```

```
{create a canvas and retrieve a surface device context}

WorkCanvas := TCanvas.Create;

FBackBuffer.GetDC(SrfcDC);

with WorkCanvas do

begin

    {hook the canvas up to the surface}

    Handle := SrfcDC;

    {initialize the surface font attributes}

    Font.Name := 'Arial';

    Font.Color := clRed;

    Font.Style := [fsBold];

    Font.Size := 12;

    Brush.Style := bsClear;

    {draw the number of balls remaining}

    TextOut(10, 450, 'Score: '+IntToStr(FScore));

    {unhook the canvas from the surface}

    Handle := 0;

end;

{release the surface device context}

FBackBuffer.ReleaseDC(SrfcDC);

{free the canvas}

WorkCanvas.Free;

{if there are no more blocks alive...}

if FBlockMgr.NumBlocks < 1 then

begin
```

```

        {increment the current level}

        Inc(FLevel);

        {initialize a new level}

        InitLevel;

        {and go to intermission}

        FGameState := gsIntermission;

    end;

end;

gsIntermission : begin

    {we are starting to play, so hide the animated
    mouse cursor}

    ShowMouse := FALSE;

    {create a canvas and retrieve a surface device context}

    WorkCanvas := TCanvas.Create;

    FBackBuffer.GetDC(SrfcDC);

    with WorkCanvas do

    begin

        {hook the canvas up to the surface}

        Handle := SrfcDC;

        {initialize the canvas font attributes}

        Font.Name := 'Arial';

        Font.Color := clRed;

        Font.Style := [fsBold];

        Font.Size := 64;

        Brush.Style := bsClear;

        {draw the level}

        TextOut(320-(TextWidth('Level '+IntToStr(FLevel))
            div 2), 125, 'Level '+IntToStr(FLevel));

```

```
        {unhook the canvas from the surface}

        Handle := 0;

    end;

    {release the surface device context}

    FBackBuffer.ReleaseDC(SrfcDC);

    {release the canvas}

    WorkCanvas.Free;

    {flip the surfaces so that this new graphic
     is displayed}

    FlipSurfaces;

    {pause for 2 seconds}

    Sleep(2000);

    {switch back to playing the game}

    FGameState := gsPlaying;

    {add an active ball}

    FBallMgr.AddBall(DXWIDTH div 2, 300, -1, 1, FUseBall);

    end;

gsGameOver : begin

    {hide the animated mouse cursor}

    ShowMouse := FALSE;

    {create a canvas and retrieve a surface device context}

    WorkCanvas := TCanvas.Create;

    FBackBuffer.GetDC(SrfcDC);
```

```
with WorkCanvas do
begin
    {hook the canvas up to the surface}
    Handle := SrfcDC;

    {initialize the canvas font attributes}
    Font.Name := 'Arial';
    Font.Color := clRed;
    Font.Style := [fsBold];
    Font.Size := 56;
    Brush.Style := bsClear;

    {draw the Game Over text}
    TextOut(320-(TextWidth('G A M E   O V E R') div 2),
           125, 'G A M E   O V E R');

    {unhook the canvas from the surface}
    Handle := 0;
end;

{release the surface device context}
FBackBuffer.ReleaseDC(SrfcDC);

{release the canvas}
WorkCanvas.Free;

{flip the surfaces to display the new graphic}
FlipSurfaces;

{pause for 2 seconds}
Sleep(2000);
```



```

        {and go to idle}

        FGameState := gsIdle;

    end;

end;

{if we wish to display the mouse cursor...}

if ShowMouse then

begin

    {increment the frame count}

    Inc(FCursorFrameCount);

    {when we need to increment the frame number...}

    if FCursorFrameCount mod 7 = 0 then

    begin

        {increment the current mouse cursor frame}

        Inc(FCursorFrame);

        {roll it over if necessary}

        if FCursorFrame > 6 then

            FCursorFrame := 0;

        end;

        {draw cursor to back buffer}

        SrcRect := Rect(FCursorFrame*CURSORWIDTH, 0,

            (FCursorFrame*CURSORWIDTH)+CURSORWIDTH, CURSORHEIGHT);

        FBackBuffer.BltFast(CurX, CurY, FMouseCursors, SrcRect,

            DDBLTFAST_SRCOLORKEY OR DDBLTFAST_WAIT);

    end

else

begin

```

```

    {move the paddle and draw it to the back buffer}

    FPaddle.Move;

    FPaddle.Draw;

    end;

end;

{ -> Deletable Events <- }

///

{ - as a matter of convenience this framework will terminate when the -
  - Escape key is pressed, but this should probably be deleted and -
  - replaced with your own terminate methods - }

procedure TfrmDXAppMain.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
    if Key = VK_ESCAPE then
        Close;
    end;
///

{ *-->>  END BASELINE APPLICATION CODE <<--* }

{initializes a level}
procedure TfrmDXAppMain.InitLevel;
begin
    {reinitialize all of the blocks}

    FBlockMgr.InitializeBlocks;

    {clear any active balls}

    FBallMgr.Clear;

```

```
{set the ball manager's velocity coefficient based on the current level.  
  this makes the balls move faster at higher levels}  
FBallMgr.VelCoefficient := FLevel div 2;  
if FBallMgr.VelCoefficient < 2 then  
  FBallMgr.VelCoefficient := 2;  
  
{we'll be creating a new ball, but since we're at the start of a level,  
  we don't want to decrement the available ball count}  
FUseBall := FALSE;  
end;  
  
{starts a game}  
procedure TfrmDXAppMain.StartGame;  
begin  
  {initialize the level count}  
  FLevel := 1;  
  
  {initialize the score}  
  FScore := 0;  
  
  {initialize the level}  
  InitLevel;  
  
  {give the user 3 balls}  
  FBallMgr.TotalBalls := 3;  
  
  {upon starting a game, we want to decrement the overall ball count when  
  we create a new one}  
  FUseBall := TRUE;
```

```

    {go to intermission}

    FGameState := gsIntermission;

end;

{plays a sound}

procedure PlayGameSound(SoundNum: Integer);

begin
    {set the position of the sound to the start}

    Sounds[SoundNum].SetCurrentPosition(0);

    {commence sound playback}

    Sounds[SoundNum].Play(0, 0, 0);

end;

{ TBall }

procedure TBall.Draw;

var

    SrcRect: TRect;

begin
    {draw the ball to the back buffer}

    SrcRect := Rect(252, 507, 261, 516);

    FBackBuffer.BltFast(Trunc(XPos), Trunc(YPos), FGraphics, SrcRect,

                        DDBLTFAST_SRCOLORKEY OR DDBLTFAST_WAIT);

end;

procedure TBall.Move;

var

    ISectRect: TRect;

    iRow, iCol: Integer;

```

```
BlockCollision: boolean;
```

```
DeflectAngle: Real;
```

```
{assumes p1-p2, p3-p4 are the lines; returns true if the lines intersect}
```

```
function CheckLineIntersect(P1, P2, P3, P4: TPoint): boolean;
```

```
var
```

```
  A, B, C: TPoint;
```

```
  Denom: integer;
```

```
  Alpha, Beta: double;
```

```
begin
```

```
  {assume the line segments do not intersect}
```

```
  Result := FALSE;
```

```
  {this uses an algorithm found in Graphics Gems III (ISBN 0-12-409673-5) by  
  David Kirk (Academic Press) on page 199. the algorithm is by Franklin  
  Antonio. basically, it is a quick way to determine if two line segments  
  intersect}
```

```
  A := Point(P2.X-P1.X, P2.Y-P1.Y);
```

```
  B := Point(P3.X-P4.X, P3.Y-P4.Y);
```

```
  C := Point(P1.X-P3.X, P1.Y-P3.Y);
```

```
  Denom := (A.Y*B.X)-(A.X*B.Y);
```

```
  {must check to see if denom is 0 here}
```

```
  if Denom <> 0 then
```

```
  begin
```

```
    {determine the alpha and beta values}
```

```
    Alpha := ((B.Y*C.X)-(B.X*C.Y))/Denom;
```

```
    Beta := ((A.X*C.Y)-(A.Y*C.X))/Denom;
```

```

        {if both alpha and beta are in the range  $0 \leq X \leq 1$ , then the lines
        intersect}

        if ((Alpha >= 0) and (Alpha <= 1)) and ((Beta >= 0) and (Beta <= 1)) then
            Result := TRUE;
        end;
    end;

begin
    {if the game state is in game over, exit immediately}
    if frmDXAppMain.FGameState = gsGameOver then
        Exit;

    {record the previous position of the ball}
    XPrev := XPos;
    YPrev := YPos;

    {update the ball's position according to its velocity}
    XPos := XPos + XVel;
    YPos := YPos + YVel;

    {determine collision with sides}

    {left}
    if XPos < 5 then
        begin
            XPos := 5;
            XVel := 0 - XVel;

            {indicate we hit the side}
            PlayGameSound(SIDEHIT);
        end;

    {right}

```

```
if XPos > DXWIDTH - 5 - 9 then
begin
    XPos := DXWIDTH - 5 - 9;
    XVel := 0 - XVel;

    {indicate we hit the side}
    PlayGameSound(SIDEHIT);
end;

{top}
if YPos < 5 then
begin
    YPos := 5;
    YVel := 0 - YVel;

    {indicate we hit the side}
    PlayGameSound(SIDEHIT);
end;

{a collision with the bottom indicates that the ball has died}
if YPos > DXHEIGHT - 5 - 9 then

begin
    {delete this ball from the managers list}
    Manager.BallList.Delete(Index);

    {play the lost ball sound}
    PlayGameSound(LOSTBALL);

    {if there are no more active balls...}
    if Manager.ActiveBalls < 1 then
```

```

{...but there are balls left...}

if Manager.TotalBalls > 0 then
begin
    {indicate we will decrement the available balls when we create
    a new one}

    frmDXAppMain.FUseBall := TRUE;

    {go to intermission}

    frmDXAppMain.FGameState := gsIntermission;
end
else
    {...otherwise, the game is over}

    frmDXAppMain.FGameState := gsGameOver;

{free this ball}

Free;

{the ball has died, so exit}

Exit;

end;

{keep up with the bounding box}

BoundingBox := Rect(Trunc(XPos), Trunc(YPos), Trunc(XPos+9), Trunc(YPos+9));

{check for collision with paddle}

if IntersectRect(ISectRect, BoundingBox, frmDXAppMain.FPaddle.BoundingBox) then
begin
    {the deflection angle is based on the point where the ball hits the paddle.
    the ball will deflect at a greater angle near the edges, but it will
    deflect almost straight up at the center. this gives the user the
    ability to control the direction of the ball. the deflection angle
    increases from 0 at the very tip to 90 degrees at the center}

```



```

DeflectAngle := (180/92)*(ISectRect.Left-
                                frmDXAppMain.FPaddle.BoundingBox.Left);

if DeflectAngle > 90 then
    DeflectAngle := 180 - DeflectAngle;

{reverse the vertical velocity}
YVel := 0 - Sin(DegToRad(DeflectAngle))*Manager.VelCoefficient;

{initialize the horizontal velocity based on the deflection angle and
 its current direction}
if XVel >=0 then
    XVel := Cos(DegToRad(DeflectAngle))*Manager.VelCoefficient
else
    XVel := 0 - Cos(DegToRad(DeflectAngle))*Manager.VelCoefficient;
{indicate that we hit the paddle}
PlayGameSound(PADDLEHIT);
end;

{now, check for collisions with blocks}
for iRow := 0 to 9 do
    for iCol := 0 to 14 do
        {if there is an active block at this position...}
        if (frmDXAppMain.FBlockMgr.BlockGrid[iCol, iRow] <> nil) then
            {perform a quick bounding box collision test}
            if IntersectRect(ISectRect, BoundingBox,
                            frmDXAppMain.FBlockMgr.BlockGrid[iCol, iRow].BoundingBox)
                then
                    begin
                        {the bounding boxes have collided. now, since we want to support
                        balls that move at any velocity, we must check the line segment
                        formed by the ball's current and previous location with each line

```

segment of the current block. since a ball can strike a block from any direction, we must determine which side of the block was struck in order to determine which velocity value is to be reversed}

```
BlockCollision := FALSE;
```

```
{check bottom lines}
```

```
if CheckLineIntersect(Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
                               iRow].BoundingBox.Left, frmDXAppMain.FBlockMgr.
                               BlockGrid[iCol, iRow].BoundingBox.Bottom),
                       Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
                               iRow].BoundingBox.Right, frmDXAppMain.FBlockMgr.
                               BlockGrid[iCol, iRow].BoundingBox.Bottom),
                       Point(Trunc(XPrev+5), Trunc(YPrev+5)),
                       Point(Trunc(XPos+5), Trunc(YPos+5))) then
```

```
begin
```

```
    {the bottom of the block was struck, so reverse the vertical
    velocity and signal a collision}
```

```
    YVel := 0 - YVel;
```

```
    BlockCollision := TRUE;
```

```
end
```

```
else
```

```
{check top lines}
```

```
if CheckLineIntersect(Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
                               iRow].BoundingBox.Left, frmDXAppMain.FBlockMgr.
                               BlockGrid[iCol, iRow].BoundingBox.Top),
                       Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
                               iRow].BoundingBox.Right, frmDXAppMain.FBlockMgr.
                               BlockGrid[iCol, iRow].BoundingBox.Top),
                       Point(Trunc(XPrev+5), Trunc(YPrev+5)),
                       Point(Trunc(XPos+5), Trunc(YPos+5))) then
```

```

begin
    {the top of the block was struck, so reverse the vertical
    velocity and signal a collision}
    YVel := 0 - YVel;
    BlockCollision := TRUE;
end
else

{check right lines}
if CheckLineIntersect(Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
    iRow].BoundingBox.Right, frmDXAppMain.FBlockMgr.
    BlockGrid[iCol, iRow].BoundingBox.Bottom),
    Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
    iRow].BoundingBox.Right, frmDXAppMain.FBlockMgr.
    BlockGrid[iCol, iRow].BoundingBox.Top),
    Point(Trunc(XPrev+5), Trunc(YPrev+5)),
    Point(Trunc(XPos+5), Trunc(YPos+5))) then
begin
    {the right side of the block was struck, so reverse the horizontal
    velocity and signal a collision}
    XVel := 0 - XVel;
    BlockCollision := TRUE;
end
else

{check left lines}
if CheckLineIntersect(Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
    iRow].BoundingBox.Left, frmDXAppMain.FBlockMgr.
    BlockGrid[iCol, iRow].BoundingBox.Bottom),
    Point(frmDXAppMain.FBlockMgr.BlockGrid[iCol,
    iRow].BoundingBox.Left, frmDXAppMain.FBlockMgr.

```

```

        BlockGrid[iCol, iRow].BoundingBox.Top),
        Point(Trunc(XPrev+5), Trunc(YPrev+5)),
        Point(Trunc(XPos+5), Trunc(YPos+5))) then
begin
    {the left side of the block was struck, so reverse the horizontal
    velocity and signal a collision}
    XVel := 0 - XVel;
    BlockCollision := TRUE;
end;

{if the ball collided with a block, kill the block}
if BlockCollision then
begin
    frmDXAppMain.FBlockMgr.KillBlock(iRow, iCol);

    {indicate that we hit a block}
    PlayGameSound(BLOCKHIT);

    {increase the score}
    frmDXAppMain.FScore := frmDXAppMain.FScore+5*frmDXAppMain.FLevel;
end;
end;

end;

{ TBallMgr }

procedure TBallMgr.AddBall(XPos, YPos, XVel, YVel: Real; DecBalls: Boolean);
var
    TempBall: TBall;
begin
    {if we are to decrement the total ball count, then do so}
    if DecBalls then

```

```
    Dec(FTotalBalls);

    {create a ball sprite}
    TempBall := TBall.Create;

    {initialize its position}
    TempBall.XPos := XPos;
    TempBall.YPos := YPos;
    TempBall.XPrev := XPos;
    TempBall.YPrev := YPos;

    {initialize its velocity}
    TempBall.XVel := XVel;
    TempBall.YVel := YVel;

    {initialize its bounding box}
    TempBall.BoundingBox := Rect(Trunc(XPos), Trunc(YPos), Trunc(XPos+9),
                                Trunc(YPos+9));

    {hook it to the manager}
    TempBall.Manager := Self;

    {add the ball to the list}
    BallList.Add(TempBall);

    {set its index in the list}
    TempBall.Index := BallList.Count-1;
end;

constructor TBallMgr.Create;

begin
```

```
{create the ball list}

BallList := TList.Create;

{initialize the velocity coefficient. increasing or decreasing this value
 will have a similar effect on the speed of the balls}

VelCoefficient := 2;

end;

destructor TBallMgr.Destroy;

var
    iCount: Integer;

begin
    {destroy any balls in the list}
    for iCount := 0 to BallList.Count-1 do
        TBall(BallList[iCount]).Free;

    {free the ball list}
    BallList.Free;

    {finish the destruction}
    inherited Destroy;
end;

procedure TBallMgr.DrawBalls;

var
    iCount: Integer;

begin
    {draw each ball in the list}
    for iCount := 0 to BallList.Count-1 do
        TBall(BallList[iCount]).Draw;
end;
```

```
procedure TBallMgr.KillBall(Idx: Integer);
begin
    {if the index is not greater than the number of balls and there is a
    ball at the indicated index...}
    if (Idx < BallList.Count) and (BallList[Idx] <> nil) then
    begin
        {free and delete this ball}
        TBall(BallList[Idx]).Free;
        BallList.Delete(Idx);
    end;
end;

procedure TBallMgr.MoveBalls;
var
    iCount: Integer;
begin
    {move all of the balls in the list}
    for iCount := 0 to BallList.Count-1 do
        TBall(BallList[iCount]).Move;
    end;

procedure TBallMgr.Clear;
var
    iCount: Integer;
begin
    {clear all of the balls}
    for iCount := 0 to BallList.Count-1 do
        KillBall(0);

        BallList.Clear;
    end;
```

```

function TBallMgr.GetActiveBalls: Integer;
begin
    {return the number of balls in the list}
    Result := BallList.Count;
end;

{ TPaddle }

procedure TPaddle.Draw;
var
    SrcRect: TRect;
begin
    {draw the paddle graphic to the back buffer}
    SrcRect := Rect(252, 500, 339, 506);
    FBackBuffer.BltFast(XPos, YPos, FGraphics, SrcRect,
                        DBLTFast_NOCOLORKEY OR DBLTFast_WAIT);
end;

procedure TPaddle.Move;
begin
    {to move the paddle, we really just need to set its horizontal position
    based on that of the mouse}
    XPos := CurX;

    {update its bounding box}
    BoundBox := Rect(Trunc(XPos), Trunc(YPos), Trunc(XPos+92), Trunc(YPos+15));
end;

{ TBlock }

```



```

procedure TBlock.Draw;
var
    SrcRect: TRect;
begin
    {draw the indicated block graphic to the screen}
    SrcRect := Rect((GraphicIdx mod 15)*42, (GraphicIdx div 15)*20+480,
                    (GraphicIdx mod 15)*42+42, (GraphicIdx div 15)*20+20+480);
    FBackBuffer.BltFast(BoundBox.Left, BoundBox.Top, FGraphics, SrcRect,
                        DDBLTFAST_NOCOLORKEY or DDBLTFAST_WAIT);
end;

{ TBlockMgr }

destructor TBlockMgr.Destroy;
var
    iCol, iRow: Integer;
begin
    {destroy all of the blocks}
    for iRow := 0 to 9 do
        for iCol := 0 to 14 do
            if BlockGrid[iCol, iRow] <> nil then
                BlockGrid[iCol, iRow].Free;

    {finish the destruction}
    inherited Destroy;
end;

procedure TBlockMgr.Draw;
var
    iCol, iRow: Integer;
begin

```

```

{draw all of the blocks}

for iRow := 0 to 9 do
    for iCol := 0 to 14 do
        if BlockGrid[iCol, iRow] <> nil then
            BlockGrid[iCol, iRow].Draw;
        end;
    end;

procedure TBlockMgr.InitializeBlocks;
var
    iCol, iRow: Integer;
begin
    {seed the random number generator}
    Randomize;

    {create a block for each position in the block grid}
    for iRow := 0 to 9 do
        for iCol := 0 to 14 do
            begin
                {create the actual block}
                BlockGrid[iCol, iRow] := TBlock.Create;

                {randomly set it to one of 20 block graphics}
                BlockGrid[iCol, iRow].GraphicIdx := Random(20);

                {initialize its bounding box}
                BlockGrid[iCol, iRow].BoundingBox := Rect(iCol*42+5, iRow*20+75,
                                                            (iCol*42+5)+42, (iRow*20+75)+20);
            end;
        end;

    {initialize the number of blocks}
    NumBlocks := 150;
end;

```

```
procedure TBlockMgr.KillBlock(Row, Col: Integer);
begin
    {free the specified block}
    BlockGrid[Col, Row].Free;
    BlockGrid[Col, Row] := nil;

    {decrement the number of blocks}
    NumBlocks := NumBlocks - 1;
end;

end.
```

---

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



**IT KNOWLEDGE.COM**<sup>SM</sup>  
Need IT. Find IT. Know IT.

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

ISBN: 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

*Listing 13-2: The Delphi Blocks mouse thread unit*

```
unit MouseThreadU;
```

```
interface
```

```
uses
```

```
Classes, Controls, Windows, SyncObjs;
```

```
type
```

```
TMouseThread = class(TThread)
```

```
private
```

```
{ Private declarations }
```

```
protected
```

```
procedure Execute; override;
```

```
end;
```

```
{this will hold mouse data for click events}
```

```

TMouseData = class

public
    XPos, YPos: Integer;
    Button: TMouseButton;

    constructor Create(X, Y: Integer; MButton: TMouseButton);

end;

var
    CurX, CurY: Integer;
    MouseClickList: TList;
    FCritSec: TCriticalSection;
    FMouseEvents: TWOHandleArray;
    ShowMouse: Boolean;

const
    CURSORWIDTH = 32;           // we have a 32 X 32 pixel cursor
    CURSORHEIGHT = 32;
    NUMFRAMES = 8;             // with 8 frames of animation

implementation

uses DInput, DelphiBlocksU, DDraw;

{this creates and initializes one mouse data object for recording
click events}

constructor TMouseData.Create(X, Y: Integer; MButton: TMouseButton);
begin
    XPos := X;
    YPos := Y;
    Button := MButton;

```

```

end;

{ TMouseEvent }

procedure TMouseEvent.Execute;

var
    EventNum: Integer;
    BufferEmpty: Boolean;
    NumElements: DWORD;
    DeviceData: TDIDeviceObjectData;
    DataResult: HRESULT;

begin
    {while the thread has not terminated...}
    while not Terminated do
        begin
            {pause this thread until either a mouse event or a quit event has fired}
            EventNum := WaitForMultipleObjects(2, @FMouseEvent, FALSE, INFINITE);

            {if the quit event fired...}
            if EventNum-WAIT_OBJECT_0 = QUITEVENT then
                begin
                    {terminate the thread and exit}
                    Terminate;
                    Break;
                end;
        end;

        {enter the critical section. this prevents the main thread from accessing
        variables used within this block of code until the critical section
        is exited}
        FCritSec.Enter;

```



```

else
    {otherwise, if nothing was retrieved, we've emptied the buffer}
    BufferEmpty := TRUE;
end;

{if the mouse cursor is to be shown...}
if ShowMouse then
begin
    {clip the mouse cursor to the screen}
    if CurX < 0 then
        CurX := 0;
    if CurX > DXWIDTH-CURSORWIDTH then
        CurX := DXWIDTH-CURSORWIDTH-1;
    if CurY < 0 then
        CurY := 0;
    if CurY > DXHEIGHT-CURSORHEIGHT then
        CurY := DXHEIGHT-CURSORHEIGHT-1;
end
else
begin
    {...otherwise, we're only showing the paddle, so we only need to
    check the horizontal position of the mouse cursor}
    if CurX < 5 then
        CurX := 5;
    if CurX > DXWIDTH - 92 then
        CurX := DXWIDTH - 92;

end;

{finally, we must leave the critical section so that the main thread
can have access to the mouse cursor location variables}

```



```
    FCritSec.Leave;  
  
    end;  
  
end;  
  
initialization  
  
    {initially, we want to show the mouse cursor}  
  
    ShowMouse := TRUE;  
  
end.
```

---

## Summary

In this chapter, we discussed how graphics, user input, and sound and music programming techniques come together to create a complete game application. We examined how each part of a game is implemented, including techniques for tracking several sprites as well as using DirectSound and Windows multimedia functions simultaneously. We also looked at various ways to enhance and improve the Delphi Blocks game. When using these techniques to create your own games, it is important to keep these points in mind:

- The state-driven architecture is useful for many styles of games and is easy to implement. More complex games may require an equally more complex architecture, but the concepts are similar.
- Always implement the most natural form of user input for the type of game being developed. For Delphi Blocks, this was the mouse, but for other games, it might be the joystick or keyboard. While not absolutely necessary, it is good programming practice to give the user a choice of input devices.
- Sound and music always enhance a game. Sound for Delphi Blocks was implemented using DirectSound in order to allow for easier modification. However, music was implemented using Windows multimedia functions in order to show how the two can coexist gracefully. This should demonstrate how you can use whatever method is appropriate for your gaming application.
- The simple 2-D graphics in this game use straightforward techniques for implementing the game animation. These techniques are acceptably fast, but could be optimized. However, bear in mind that optimization has a diminishing margin of returns, so only optimize those parts that truly need it.
- While not a commercial quality game, this case study could easily be extended and molded into one within a few weeks. Remember, Delphi can do anything that C can do; you just have to have the knowledge and patience.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy statement](#).



**KEYWORD SEARCH**

[Search Tips](#)  
[Advanced Search](#)

**PUBLICATION LOOKUP**

**JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.  
**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## APPENDIX

### Installing DirectX

This appendix covers the following topics:

- Retrieving the version number of the currently installed DirectX
- Installing DirectX components, either in full or in part
- Overriding the default behavior of installation to provide custom status messages and feedback

Before a game written using the DirectX API can be run on a user's system, that system must have DirectX installed. Fortunately, Microsoft allows game developers to redistribute the DirectX run-time components and drivers at no charge. Additionally, they've included methods that make the installation of DirectX largely automated. In keeping with their naming convention for this technology, setup of DirectX components and drivers is accomplished through the DirectSetup API.

In this appendix, we'll examine how the DirectSetup API allows us to install DirectX with a minimum of hassle. We'll also examine ways to determine the current version of DirectX already installed, as well as how to override the default behavior of the installation in order to provide our own interface.

### DirectX Redistributables

DirectX is available from a number of sources. Many computer manufacturers are shipping machines with DirectX already installed. Indeed, future versions of Windows will include DirectX automatically. The DirectX run-time components are also available both online (at Microsoft's web site) as well as part of the installation for hundreds of commercial games and applications.



**Tip:** You may want to suggest that users download the latest DirectX drivers straight from the Microsoft web site as part of your standard technical support. Users can retrieve the end user DirectX files from <http://www.microsoft.com/directx/homeuser/downloads/default.asp> at the time of this writing. Microsoft technical support should also be able to help users if they are having trouble installing DirectX drivers.

If DirectX is already installed on a user's system, it is not necessary to install it again when the user installs your application (unless, of course, the application uses a more current version of DirectX than that currently installed).

Thus, if you are certain that your end users already have DirectX installed, there is no need to worry about the techniques described here. Unfortunately, although Microsoft is taking steps to make it simple for users to upgrade their Windows installation, we cannot always assume that users have DirectX installed on their machines.

Microsoft allows developers to redistribute the run-time portion of DirectX royalty free with our applications. This distribution is subject to the terms of their end user license agreement, also known as an EULA, which accompanies the DirectX SDK. Therefore, you'll need to obtain the DirectX SDK in order to get all of the information and rights necessary for redistributing DirectX. For more information on obtaining the latest DirectX SDK and other DirectX run-time components, please visit <http://www.microsoft.com/directx/developer/downloads/default.asp>.



**EULA:** End user license agreement; a legal document that details specific licensing issues concerning software.

The redistributable files themselves are located in the Redist directory of the DirectX SDK. The license agreement for redistributing DirectX run-time components instructs us that we must not modify DirectX in any way, nor change the files that must be redistributed, with one exception. The DirectX redistributable files come in several localized flavors, and we can leave out any specific language for which we will not support. Thus, if we only want to ship the English language version of DirectX, we need only include the 'eng' directory under the DirectX\Drivers directory of the redistributable files. This may be desirable, as the size of the entire redistributable file set is over 50 MB. Please see the DirectX SDK EULA.txt file that ships with the DirectX SDK for more details concerning redistribution of DirectX with applications.



**Note:** At the time of this writing, the DirectX redistributables cannot be installed under Windows NT. DirectX can only be installed on NT through NT service packs.

---

## DirectSetup

As mentioned previously, DirectSetup is responsible for installing DirectX components on an end user's machine. Unlike the other DirectX subsystems, the DirectSetup API is rather small. There are only a few functions included in this API, and full installation of all DirectX run-time components and drivers can be accomplished by calling one simple function. Alternatively, the default behavior of the entire installation process can be overridden and monitored. The DirectSetup API can make installation of DirectX as simple or as complicated as necessary, giving developers plenty of flexibility in their installation programs.

### Features

DirectSetup provides several techniques by which an application can have a varying degree of control over the installation of DirectX. The following features are of interest to any developer implementing DirectX installation functionality:

- Quick, painless installation of the entire DirectX system, run-time components only, or drivers only through one simple function call
- Ability to override the default behavior of DirectX installation to provide custom status reports and installation query messages
- Testing of the DirectX installation without copying any files into system directories
- Determination of the version of the currently installed DirectX system

In the following sections we will see how DirectSetup is used to install DirectX components in an automated, default manner as well as techniques to override this default behavior to provide additional feedback.

### Functional Overview

The purpose of DirectSetup is to install DirectX in a manner such that it does not inadvertently affect the system. Therefore, it takes a very conservative approach, and by default, it will not replace any driver that is questionable or may impact the system in any negative way. In addition, it also backs up any driver that is replaced, so that the user may reverse the installation and restore the original drivers if necessary.

In contrast to other DirectX subsystems, the DirectSetup API is not a collection of COM components, but rather a small handful of functions. Indeed, our discussion of DirectSetup will encompass only three of these functions: DirectXSetupGetVersion, DirectXSetup, and DirectXSetupSetCallback.

The first function, DirectXSetupGetVersion, is rather simple. It merely retrieves the version and revision numbers of the DirectX components currently installed. This function can be used to determine if the current DirectX version is older than the one included with the application, or if DirectX is installed at all.



---

*Tip:* You can also determine the version of DirectX currently installed on the user's machine using techniques from Chapter 4, An Introduction to DirectX.

---

The second function, DirectXSetup, is perhaps the most important. This function performs the actual installation of all of the DirectX components. Using this function, we can indicate if the entire DirectX system should be installed or only certain parts. Specifically, we can instruct the function to install only run-time components, audio drivers, or display drivers.

The final function, DirectXSetupSetCallback, is used to provide a callback function used by the installation procedure that notifies the application when an installation event occurs. This allows the application to customize the behavior of the installation process and display status messages and message boxes different from the default. This function must be used to install a callback function before the DirectXSetup function is called.

## Using DirectSetup

In what should be a very familiar theme by now, using DirectSetup involves some very specific steps performed in a specific order. Not surprisingly, some of these steps are optional, some are not. In order to use DirectSetup to install DirectX, the application must perform the following:

- Locate the DirectX redistributables in the same directory as the setup application
- Retrieve available drive space
- Retrieve the current version of DirectX (optional)
- Set a callback function (optional)
- Call the DirectXSetup function

### Locate the DirectX Redistributables

More a requirement than a step, this is a task the developer must perform when packaging the application files for distribution. In order to use DirectSetup functions, the DirectX redistributable files must be placed somewhere on the distribution media (presumably a CD). Specifically, the dsetup.dll, dsetup16.dll, and dsetup32.dll must be in the same directory as the setup application so they'll be loaded when the application starts. The other DirectX redistributable files must be located in a directory named DirectX (not case sensitive) which may be located elsewhere, but typically it is more convenient to simply place them in the same directory as the setup application (see below for more details). In the example for this chapter, we placed the redistributables in the same directory as the example code, so refer to the accompanying CD for an example of directory structure.

### Retrieve Available Drive Space

Technically speaking, this step is not an absolute requirement, as the DirectX installation procedure will fail gracefully if it runs out of disk space. DirectX requires a minimum of 15 MB of free space (if you only use one language) on the drive in which Windows is installed. Out of courtesy to your customers, who may be computer novices that could get confused by insufficient disk space error messages during an install, you should always check and inform users if they do not have enough space to install DirectX.

### Retrieve the Current Version of DirectX

Since many machines now have DirectX already installed, either by another application or by the manufacturer, you may want to check to see if the currently installed DirectX version is older than that shipped with your application. If the installed version is more current than the one you are shipping, you can skip DirectX installation altogether.

In order to retrieve the version number, the application must call the `DirectXSetupGetVersion` function. The `DirectXSetupGetVersion` function is defined as:

```
function DirectXSetupGetVersion(  
  
var lpdwVersion: DWORD;           // the version number  
  
var lpdwRevision: DWORD           // the revision number  
  
    : Integer;                     // returns non-zero if successful
```

The first parameter is a `DWORD` variable that will contain the major version number in the high-order word and the minor version number in the low-order word when the function returns. The second parameter is also a `DWORD` variable that will contain the release number in the high-order word and the build number in the low-order word. The minor version number reflects the “commercial” version of DirectX, and will be 6 for DirectX 6 or higher, 5 for DirectX 5, etc.



*Note:* There was no version 4 of DirectX.

---

## Set a Callback Function

If the developer wishes the DirectX installation to proceed using its standard, default behavior, this step can be skipped. However, in order to customize the behavior of the installation process, the application needs to inform `DirectSetup` that it should call a callback function during the installation process. This is accomplished by calling the `DirectXSetupSetCallback` function, which is defined as:

```
function DirectXSetupSetCallback(  
  
Callback: TDSetup_Callback         // pointer to the callback function  
  
    ) : Integer;                   // returns zero
```

Currently, this function only returns zero, whether it succeeded or failed. The only parameter for this function is a pointer to the callback function. This function will be called by the installation process at various times when certain events happen or the user needs to supply additional information. The callback function itself must be defined as:

```
function TDSetup_Callback(  
  
Reason: DWORD;                     // reason flags  
  
MsgType: DWORD;                    // message type flags  
  
szMessage: PChar;                  // message string  
  
szName: PChar;                      // driver name string  
  
pInfo: Pointer                      // additional driver information  
  
    : DWORD; stdcall;              // returns MessageBox return values
```

We’ll examine this callback function in more detail below when we cover techniques for customizing the installation

process.

## Call the DirectXSetup Function

To begin the process of actually copying DirectX system files onto the hard drive and configuring Windows with the new components, we must call the DirectXSetup function. The DirectXSetup function is defined as:

```
function DirectXSetup(  
  
    hWnd: HWND; // parent window handle  
  
    lpzRootPath: PAnsiChar; // DirectX redistributables path  
  
    dwFlags: DWORD // installation control flags  
  
    : Integer; // returns a DirectX error code
```

The first parameter is a handle to a window that will be used as the parent window for any dialog boxes that need to be displayed during the installation. The second parameter is the path for the location of the DirectX redistributable files. Setting this to nil causes DirectSetup to look for the files in the current directory, as described above.

The third parameter is set to a series of flags that indicate how DirectX is to be installed. This controls whether the entire DirectX system will be installed, or only certain parts. The available flags for this parameter are shown in Table A-1. The DirectSetup documentation recommends that applications perform a full install using the DSETUP\_DIRECTX flag. However, an application could provide the user with a method to install only the run-time components or perhaps just audio and/or video drivers in the case of a malfunction or for troubleshooting purposes. Another very useful flag is DSETUP\_TESTINSTALL, which runs through a test installation of DirectX but doesn't actually replace any drivers or copy any run-time components to the system. Using this in conjunction with a callback function allows the developer to make a detailed list of drivers that will be replaced during the installation, which could then be presented to the user in a summary format before the actual installation process began.

**Table A-1:** DirectXSetup control flags

Flag	Description
DSETUP_DDRAWDREV	Installs display drivers only
DSETUP_DIRECTX	Performs a full installation of DirectX
DSETUP_DSOUNDREV	Installs audio drivers only
DSETUP_DXCORE	Installs run-time components only
DSETUP_TESTINSTALL	Performs a test installation without installing files

This function can return a number of error codes that indicate various problems such as insufficient disk space or missing source files. However, the application should specifically watch out for the DSETUPERR\_SUCCESS\_RESTART return value. This indicates that the machine must be rebooted before DirectX functionality will be available. DirectSetup does not automatically inform the user of this, so your installation application will need to do the job. A return value of DSETUPERR\_SUCCESS indicates that installation was successful and no reboot is necessary. See the DSetup.pas file (part of the JEDI DirectX headers) for a complete listing of error codes returnable by DirectXSetup.

## Overriding Default Behavior

If the application did not specify a callback function, calling the DirectXSetup function will launch the installation of DirectX, and it will proceed to install DirectX drivers and components without any intervention from the user or the application. DirectSetup will provide a simple status window to the user, but this is the only feedback it outputs. The following illustration shows the default DirectSetup installation status box.





### **Figure A-1:** Default DirectSetup status message

If any sort of error occurs, such as insufficient drive space, it will not display any message to the user, relying on the application to interpret the return value and display the appropriate message. It also will not ask the user for any sort of confirmation. If it encounters an unrecognized driver or other potential driver incompatibility problem, it simply skips the driver and returns success, without ever bothering the user. As mentioned before, this approach is taken so that the system is not left in an unusable state due to driver conflicts. However, this can lead to an unusable DirectX installation.

Therefore, it might be wise to provide a little more customized feedback to the user, especially if your application is to be widely distributed. As mentioned previously, we should set up a callback function using the `DirectXSetupSetCallback` function. For the sake of completeness (and so you don't have to flip back and forth in this section), the callback function itself is defined as:

```
function TDSetup_Callback(  
  
Reason: DWORD; // reason flags  
  
MsgType: DWORD; // message type flags  
  
szMessage: PChar; // message string  
  
szName: PChar; // driver name string  
  
pInfo: Pointer // additional driver information  
  
: DWORD; stdcall; // returns MessageBox return values
```

The first parameter contains one of several flags that indicate why the callback function was called. These flags are listed in Table A-2 below. As you can see, this callback function is called on several occasions, notably when an error has occurred or when DirectSetup is about to install DirectX drivers or components. Of particular interest is the `DSETUP_CB_MSG_NOMESSAGE` flag. This flag is sent to the callback function merely to provide a method for updating a status bar. When this flag is received, none of the other parameters contain any useful information, and the callback function should return immediately.

<b>Flag</b>	<b>Description</b>
<code>DSETUP_CB_MSG_NOMESSAGE</code>	Update pulse
<code>DSETUP_CB_MSG_CANTINSTALL_UNKNOWNOS</code>	The operating system cannot be determined, and DirectX can't be installed
<code>DSETUP_CB_MSG_CANTINSTALL_NT</code>	Cannot install under Windows NT
<code>DSETUP_CB_MSG_CANTINSTALL_BETA</code>	Pre-release beta version of Windows 95 detected; cannot install DirectX
<code>DSETUP_CB_MSG_CANTINSTALL_NOTWIN32</code>	16-bit version of Windows detected; cannot install DirectX
<code>DSETUP_CB_MSG_CANTINSTALL_WRONGLANGUAGE</code>	The DirectX driver or component is not localized to the installed Windows language
<code>DSETUP_CB_MSG_CANTINSTALL_WRONGPLATFORM</code>	The DirectX driver or component is for another type of computer
<code>DSETUP_CB_MSG_PREINSTALL_NT</code>	DirectX is already installed on this version of NT
<code>DSETUP_CB_MSG_NOTPREINSTALLEDONNT</code>	DirectX cannot be installed on this version of NT
<code>DSETUP_CB_MSG_SETUP_INIT_FAILED</code>	General setup failure
<code>DSETUP_CB_MSG_INTERNAL_ERROR</code>	An internal error has occurred
<code>DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE</code>	A driver is being considered for upgrade

DSETUP_CB_MSG_OUTOFDISKSPACE	Insufficient disk space to install all DirectX components
DSETUP_CB_MSG_BEGIN_INSTALL	Installation has begun
DSETUP_CB_MSG_BEGIN_INSTALL_RUNTIME	Installation of DirectX run-time components has begun
DSETUP_CB_MSG_BEGIN_INSTALL_DRIVERS	Installation of device drivers has begun
DSETUP_CB_MSG_FILECOPYERROR	An error occurred copying a file

The second and third parameters are used when a message should be presented to the user. These values are intended to be used with the MessageBox function, with the MessageType parameter indicating the flags to be used with MessageBox, and the szMessage parameter containing the text of the message to be presented. In any case when MessageType is not zero, the callback function should return a value commensurate with the values returned by the MessageBox function for the flags indicated in the MessageType parameter. Ideally, the callback function will just pass all of this to the MessageBox function, in turn passing its return value back as the return value for the callback. However, the application can display any form of dialog box desired with any type of feedback desired, as long as the callback return value matches that which would be returned by the MessageBox function for the given flags in the MessageType parameter. The sole exception to this rule is that the function can always return zero, in which case DirectSetup will act in a default manner as if no callback function had been specified.

In the case where MessageType is zero, the DirectSetup documentation states that we should always return IDOK. The IDOK return value, in this case, will suppress the default status window displayed by DirectSetup. However, if we return zero when MessageType is zero, the default status dialog box will be displayed. This allows us to display our own status indicator or use DirectSetup's default status indicator as desired.



**Caution:** It is possible through the use of the callback function to provide default responses and install DirectX without any intervention from the user, or even without the user being aware of the installation. This is very bad programming practice, and can potentially lead to an incorrect installation. Always inform the user that DirectX is being installed, and especially give the user the appropriate feedback and options when drivers are being upgraded.

## Verification of Driver Upgrade

The most interesting of all of the reason flags is DSETUP\_CB\_MSG\_CHECK\_DRIVER\_UPGRADE. This indicates that DirectX is considering upgrading a driver, and verification from the user is recommended. The MessageType and szMessage parameters will most certainly contain values indicating the type of message box to display as well as a text description of the problem and recommended user action. The fourth parameter, szName, will contain the name of the driver being considered for upgrade.

The final parameter, pInfo, is a pointer to a TDSetup\_CB\_UpgradeInfo structure that contains additional information about the driver in question. This simple structure is defined as:

```
TDSetup_CB_UpgradeInfo = record
    UpgradeFlags: DWORD;           // driver upgrade information flags
end;
```

The only member of this structure, UpgradeFlags, can contain one or more flags from Table A-3 which indicate the status of the upgrade. Of particular interest are the DSETUP\_CB\_UPGRADE\_FORCE and DSETUP\_CB\_UPGRADE\_KEEP flags. DSETUP\_CB\_UPGRADE\_FORCE indicates that Windows may not function correctly unless the driver is upgraded. In this case, DirectSetup will upgrade the driver regardless of the value returned by the callback function. The DSETUP\_CB\_UPGRADE\_KEEP flag, on the other hand, indicates that Windows may not function correctly if the driver is upgraded. In this case, Direct Setup will not upgrade the driver, again regardless of the return value. Other flags indicate the type of driver being upgraded or other warnings that the user should take into account when deciding to allow or disallow the upgrade.

**Table A-3:** Driver upgrade information flags



Flag	Description
DSETUP_CB_UPGRADE_KEEP	The system may fail if the driver is upgraded
DSETUP_CB_UPGRADE_SAFE	This driver is safe to upgrade; upgrading recommended
DSETUP_CB_UPGRADE_FORCE	The system may fail if the driver is not upgraded
DSETUP_CB_UPGRADE_UNKNOWN	Existing device driver unrecognized; upgrading is not recommended
DSETUP_CB_UPGRADE_HASWARNINGS	Upgrading the driver may affect one or more programs; upgrading is not recommended
DSETUP_CB_UPGRADE_CANTBACKUP	Existing system components cannot be backed up, and therefore cannot subsequently be restored
DSETUP_CB_UPGRADE_DEVICE_ACTIVE	The device driver is currently in use
DSETUP_CB_UPGRADE_DEVICE_DISPLAY	This is a display device driver
DSETUP_CB_UPGRADE_DEVICE_MEDIA	This is a media device driver (typically an audio device)

The following example demonstrates how to use the `DirectXSetup` function in conjunction with a callback function to suppress the default status dialog box as well as provide custom feedback information to the user. It also illustrates how to perform both a test installation as well as a full installation of all DirectX components.

***Listing A-1: Installing DirectX***

---

```

interface

    .

    .

    .

    {declare the callback function}

    function DSetupCallback(Reason: DWORD; MsgType: DWORD;

        szMessage: PChar; szName: PChar; pInfo: Pointer) : DWORD; stdcall;

    .

    .

    .

implementation

    {this function is called numerous times throughout the DirectX

        setup process}

```

```

function DSetupCallback(Reason: DWORD; MsgType: DWORD;
    szMessage: PChar; szName: PChar; pInfo: Pointer) : DWORD; stdcall;

begin
    {if the MsgType parameter is 0, return IDOK, otherwise let's
    display a message box and return the appropriate value}
    if MsgType = 0 then
        Result := IDOK    // setting this to 0 displays the standard status box
    else
        Result := MessageBox(frmDXSetup.Handle, szMessage, 'DirectX Setup',MsgType);
    {display a message in our list box indicating the reason
    why the callback function was called}
    case Reason of
        DSETUP_CB_MSG_NOMESSAGE :
            frmDXSetup.lbxStatus.Items.Add('NoMessage Pulse (could display +
                a progression indicator on this message)');

        DSETUP_CB_MSG_CANTINSTALL_UNKNOWNOS :
            frmDXSetup.lbxStatus.Items.Add('Can't install, unknown OS');

        DSETUP_CB_MSG_CANTINSTALL_NT :
            frmDXSetup.lbxStatus.Items.Add('Can't install to versions of Windows +
                NT prior to 4.0');

        DSETUP_CB_MSG_CANTINSTALL_BETA :
            frmDXSetup.lbxStatus.Items.Add('Pre-release beta version of Windows 95 +
                detected, cannot install.');
```

```

        DSETUP_CB_MSG_CANTINSTALL_NOTWIN32 :
            frmDXSetup.lbxStatus.Items.Add('DirectX is not compatible with Windows +
                3.x');
```

```

        DSETUP_CB_MSG_CANTINSTALL_WRONGLANGUAGE :
```

```
frmDXSetup.lbxStatus.Items.Add('This version of DirectX not localized to+  
the current language configured for Windows.');
```

```
DSETUP_CB_MSG_CANTINSTALL_WRONGPLATFORM :
```

```
frmDXSetup.lbxStatus.Items.Add('DirectX component or driver incompatible+  
with current hardware.');
```

```
DSETUP_CB_MSG_PREINSTALL_NT :
```

```
frmDXSetup.lbxStatus.Items.Add('DirectX already installed on this +  
version of Windows NT.');
```

```
DSETUP_CB_MSG_NOTPREINSTALLEDONNT :
```

```
frmDXSetup.lbxStatus.Items.Add('Cannot install DirectX on this version +  
of Windows NT.');
```

```
DSETUP_CB_MSG_SETUP_INIT_FAILED :
```

```
frmDXSetup.lbxStatus.Items.Add('Setup of the DirectX component or +  
device driver has failed.');
```

```
DSETUP_CB_MSG_INTERNAL_ERROR :
```

```
frmDXSetup.lbxStatus.Items.Add('An internal error has occurred and +  
DirectX setup has failed.');
```

```
{here, DirectSetup is considering upgrading a driver. the user will have  
already been asked for confirmation above, but here we can see how  
we can query the pInfo parameter for more information about the  
upgrade}
```

```
DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE :
```

```
begin
```

```
frmDXSetup.lbxStatus.Items.Add('Upgrading driver: '+szName);
```

```

frmDXSetup.lbxStatus.Items.Add(szMessage);

{determine and display upgrade information}

if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_KEEP) = DSETUP_CB_UPGRADE_KEEP then
    frmDXSetup.lbxStatus.Items.Add('Current driver cannot be upgraded');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_SAFE) = DSETUP_CB_UPGRADE_SAFE then
    frmDXSetup.lbxStatus.Items.Add('Driver recognized, upgrade is safe');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_FORCE) = DSETUP_CB_UPGRADE_FORCE then
    frmDXSetup.lbxStatus.Items.Add('Upgrade must be performed');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_UNKNOWN) = DSETUP_CB_UPGRADE_UNKNOWN then
    frmDXSetup.lbxStatus.Items.Add('Unknown driver');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_HASWARNINGS) = DSETUP_CB_UPGRADE_HASWARNINGS then
    frmDXSetup.lbxStatus.Items.Add('Upgrade is not recommended');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_CANTBACKUP) = DSETUP_CB_UPGRADE_CANTBACKUP then
    frmDXSetup.lbxStatus.Items.Add('Driver cannot be backed up, +
        restoration impossible');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_DEVICE_ACTIVE) =
    DSETUP_CB_UPGRADE_DEVICE_ACTIVE then
    frmDXSetup.lbxStatus.Items.Add('Driver being upgraded is currently +
        in use');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and
    DSETUP_CB_UPGRADE_DEVICE_DISPLAY) =
    DSETUP_CB_UPGRADE_DEVICE_DISPLAY then
    frmDXSetup.lbxStatus.Items.Add('Driver is a display driver');
if (TDSetup_CB_UpgradeInfo(pInfo^).UpgradeFlags and

```

```

        DSETUP_CB_UPGRADE_DEVICE_MEDIA) =
        DSETUP_CB_UPGRADE_DEVICE_MEDIA then
        frmDXSetup.lbxStatus.Items.Add('Driver is a sound driver');
    end;

DSETUP_CB_MSG_OUTOFDISKSPACE :
    frmDXSetup.lbxStatus.Items.Add('Out of disk space');

DSETUP_CB_MSG_BEGIN_INSTALL :
    frmDXSetup.lbxStatus.Items.Add('DirectX installation commencing');

DSETUP_CB_MSG_BEGIN_INSTALL_RUNTIME :
    frmDXSetup.lbxStatus.Items.Add('Installing DirectX run-time components');

DSETUP_CB_MSG_BEGIN_INSTALL_DRIVERS :
    frmDXSetup.lbxStatus.Items.Add('Installing DirectX device drivers');

DSETUP_CB_MSG_FILECOPYERROR :
    frmDXSetup.lbxStatus.Items.Add('An error occurred trying to copy a file');
end;
end;

procedure TfrmDXSetup.FormCreate(Sender: TObject);
var
    Version, Revision: DWORD;
begin
    {retrieve the version and revision number for the currently
    installed DirectX}
    DirectXSetupGetVersion(Version, Revision);

```

```

{display the version and revision numbers}

lblVersion.Caption := Format('%d.%d.%d.%d (Commercial Version: %d)',
                             [HiWord(Version), LoWord(Version),
                              HiWord(Revision), LoWord(Revision),
                              LoWord(Version)]);

{set up our callback function}

DirectXSetupSetCallback(@DSetupCallback);

end;

procedure TfrmDXSetup.btnTestInstallClick(Sender: TObject);
begin
    {clear the feedback list box}

    lbxStatus.Items.Clear;

    { - we'll use the ErrorString function located in the DSetup.pas unit -
      - to retrieve a text string describing the success code returned by -
      - DirectXSetup. we should specifically check to see if the system -
      - must be rebooted. - }

    {perform a test installation}

    lbxStatus.Items.Add(ErrorString(DirectXSetup(Handle, nil, DSETUP_TESTINSTALL)));

    {indicate that installation has finished}

    lbxStatus.Items.Add('DirectSetup finished');

end;

procedure TfrmDXSetup.btnRealInstallClick(Sender: TObject);
begin
    {clear the feedback list box}

    lbxStatus.Items.Clear;

```

```

{this time, we'll perform a full installation of DirectX. we're still
using the ErrorString function as described above}

lbxStatus.Items.Add(ErrorString(DirectXSetup(Handle, nil, DSETUP_DIRECTX)));

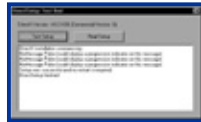
{indicate that installation has finished}

lbxStatus.Items.Add('DirectSetup finished');

end;

end.

```



**Figure A-2:** The customized DirectXSetup feedback

## Summary

In this appendix, we discussed how DirectXSetup can automate the task of installing DirectX onto an end user's machine. We examined various installation methods, including the standard, no-frills installation as well as a highly customized installation that overrides the default behaviors of DirectXSetup. We also discussed retrieving the version of the currently installed DirectX, as well as some issues with redistribution of DirectX files. When writing an installation program that uses DirectXSetup to install DirectX components, it is important to keep these points in mind:

- If the version of DirectX installed on the target machine is equal to or more current than the version shipping with your application, there is no need to install DirectX.
- Only those files located in the Redist directory of the DirectX SDK are redistributable with applications. Refer to the DirectX SDK EULA.txt file for more information, or visit <http://www.microsoft.com/directx/developer/downloads/default.asp>.
- The only change to the redistributable files that can be made is that languages unsupported by your application can be left out. This can dramatically reduce the storage requirements for the DirectX redistributable file set.
- Redistributables, and in particular the dsetup.dll, dsetup16.dll, and dsetup32.dll, must be located in the same directory as the installation program.
- The minimum required disk space for DirectX is 15 MB, if you are installing only one language version.
- The minor version number returned by DirectXSetupGetVersion reflects the commercial version of DirectX.
- The DirectXSetup function can perform a full installation of all DirectX components, an installation of specific components (such as run-time components or drivers), or a test installation. The test installation acts like a real installation, except it makes no changes to the system.
- By default, DirectXSetup does not provide any feedback to the user other than a status dialog box, nor does it retrieve confirmation from the user concerning driver replacement. If any driver upgrade would cause any type of problem, it will simply be skipped and the DirectXSetup function will return success, with no indication to the user of any problems encountered.
- A callback function can be specified, which will be called by DirectXSetup at various times throughout the installation process. This allows the application to override default behavior and provide customized feedback to the user.
- If the MsgType parameter of the callback function is anything other than zero, it should return a value commensurate with that returned by the MessageBox function.

- If the MsgType parameter contains zero, the callback function should return a value of IDOK. However, returning a value of zero causes DirectSetup to perform the default action.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.





**IT KNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX 7.0**

by *John Ayres*  
Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Glossary

### A

**artificial intelligence (AI):**

The concept of cognitive software that thinks and reacts to situations in a sentient manner. In relation to game programming, this is the section of code responsible for determining the actions and reactions of game antagonists in relation to the actions and reactions of the player.

**alpha channel:**

A component of color that records levels of transparency/opacity as opposed to actual color data.

### B

**bitmap:**

A collection of pixels arranged in a rectangular configuration in such a way that they form a recognizable shape, such as a picture or texture.

**BLT:**

bit block transfer. Pronounced "blit," this is the act of copying the pixels of a bitmap image onto the screen or other graphical output surface for display.

### C

**coefficient:**

A measurement of the strength of resistance output by a force feedback device in response to the movement of an axis, relative to the total maximum force available on the device itself.

**color depth:**

A measurement of the number of bits per pixel required to describe the color of an individual pixel.

**condition:**

A resistant force applied to one or more axes on a force feedback device in response to the movement or position of those axes.

## *D*

### **device context (DC):**

Represents the surface of an output device such as a monitor, printer, or plotter, allowing the application to perform graphical output.

### **device-dependent bitmap (DDB):**

An older bitmap image format that stores only height, width, color depth, and image pixels.

### **device-independent bitmap (DIB):**

A flexible bitmap image format that includes all necessary information to accurately reproduce the image on any device.

## *E*

### **end user license agreement (EULA):**

A legal document that details specific licensing issues concerning software.

### **envelope:**

A series of values that shape a force at the beginning and end of playback.

## *F*

### **fade:**

A transition effect where an image slowly appears out of a solid color, such as black or white.

### **finite state machine:**

A logical construction that contains a series of states defining specific behaviors and a series of state transitions defining rules for moving from state to state.

### **frame:**

A single image in a series of images to be animated.

### **Frame animation:**

The process of producing animation by rapidly displaying a series of images depicting an object or character in slightly different positions each frame.

### **force feedback:**

A method by which an input device can output tactile feedback to the user through various actuator motors.

## *G*

### **gain:**

A scaling factor for magnitudes and coefficients, measured in hundredths of a percent.

### **Graphical Device Interface (GDI):**

The Windows subsystem responsible for all graphical output during a Windows session.

## *H*

### **HAL:**

Hardware abstraction layer; the DirectX drivers responsible for direct interface to hardware capabilities.

### **HEL:**

Hardware emulation layer; the DirectX drivers responsible for emulating certain functionality if not present in hardware.

### **hertz (Hz):**

A measurement of samples per second.

### **human/computer interaction:**

The concept of techniques and methods by which humans and machines, primarily computers, exchange information.

## *K*

**kinetic force:**

An active push on one or more axes on a force feedback device.

**L****line:**

A collection of pixels, usually arranged in a linear or connecting configuration.

**M****magnitude:**

A measurement of the strength of a kinetic force output by a force feedback device, relative to the total maximum force available on the device itself.

**masking:**

The act of using two bitmaps, an AND bitmap and an OR bitmap, along with raster operations to draw one bitmap over another while simulating transparent pixels in the source bitmap.

**Microsoft Foundation Classes (MFC):**

Microsoft's object-oriented component hierarchy, similar to Delphi's VCL.

**mixing:**

Combining several sounds in real time in such a way that they can be played simultaneously.

**MIDI (Musical Instrument Digital Interface):**

A digital format for describing music.

**O****overscan:**

The area on a monitor that borders the usable display area.

**P****path finding:**

Algorithmic techniques for determining the shortest, most direct path from a starting position to an ending position based on terrain and obstacles in the game world.

**pixel:**

The smallest, single element in a graphical image or on the display surface.

**polling:**

A method for retrieving input or checking some data value manually (as opposed to letting a separate thread check it automatically), usually repeated often at regular intervals.

**polygon:**

A collection of lines, arranged in such a way that the ends are connected to form a continuous, closed shape.

**POV hat:**

The point-of-view (POV) hat is a device that typically rests on the top of a joystick, and can be used to indicate a viewing direction separate from movement direction.

**primary sound driver:**

The audio output device selected by the user on the Audio tab of the Multimedia control panel applet.

**profiler:**

A profiler is an application that typically measures application execution time. The level at which an application is timed usually varies, from timing the execution of individual lines to the execution time of functions, procedures, or the overall application.

**pulse code modulation (PCM):**

An uncompressed audio format; the most common format for Windows WAV files.

**R****raster graphics:**

Graphics that are composed of a collection of pixels that together form a picture or image. Generally

refers to bitmaps, icons, and other such graphical types.

**raster operation (ROP):**

A Boolean function performed on pixels when they are copied from bitmap to bitmap that determines how the pixels from the source, the destination, and the destination device context's selected brush are combined.

**Red Book Audio:**

The digital format used to store professional music on conventional audio CDs.

**resolution:**

A measurement of the number of pixels displayed on the screen.

**Resource Interchange File Format (RIFF):**

A complex, variable length file consisting of variable length "chunks." Each chunk can describe data or a header (which contains a description of the data following it). Commonly used for multimedia file formats, such as WAV files.

**RGB color:**

Refers to the method by which colors are represented, usually in the form of a red, green, and blue component, respectively.

**S**

**sprite:**

An image that animates and moves around the screen in some fashion. Typically associated with images that the user can interact with, either directly or through interaction with other sprites.

**T**

**texture mapping:**

A general term for algorithms that map a bitmap image onto a geometric shape.

**V**

**vector graphics:**

Graphics that are represented by a series of connected points in a virtual 2-D or 3-D Cartesian (or similar) coordinate system. Generally refers to lines, polygons, curves, and other such graphical types.

**video mode:**

The video hardware state that dictates the display resolution and number of available colors for graphical output.

**Z**

**z-order:**

The position along the z, or third-dimensional, axis. This is an imaginary axis that runs perpendicular to the screen, and gives the illusion of depth.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

**Search this book:**

[Previous](#) [Table of Contents](#) [Next](#)

## Bibliography

There are a lot of great books out there on Windows graphics programming, game programming, and DirectX programming. The information contained in this publication is based in part on research and knowledge gleaned from the following books:

Abrash, Michael. *Michael Abrash's Zen of Graphics Programming*. Coriolis Group, 1996.

Ayres, et. al. *The Tomes of Delphi 3: Win32 Graphical API*. Wordware Publishing, 1998.

Bargen and Donnelly. *Inside DirectX*. Microsoft Press, 1998.

Kawick, Mickey. *Real-Time Strategy Game Programming*. Wordware Publishing, 1999.

Kirk, David. *Graphics Gems III*. Academic Press, 1992.

LaMothe, Andre. *Black Art of 3D Game Programming*. Waite Group Press, 1995.

LaMothe, Andre. *Teach Yourself Game Programming in 21 Days*. SAMS Publishing, 1994.

LaMothe, Ratcliff, et. al. *Tricks of the Game Programming Gurus*. SAMS Publishing, 1994.

Lyons, Eric R. *Black Art of Windows Game Programming*. Waite Group Press, 1995.

Swokowski, Earl W. *Fundamentals of Trigonometry*. PWS-KENT Publishing Company, 1989.

Trujillo, Stan. *High Performance Windows Graphics Programming*. Coriolis Group, 1998.

[Previous](#) [Table of Contents](#) [Next](#)



**IT KNOWLEDGE.COM**<sup>SM</sup>  
*Need IT. Find IT. Know IT.*

Monday, January 12, 2004

▶ **KEYWORD SEARCH**

- ▶ [Search Tips](#)
- ▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

▶ **JUMP TO TOPIC**



**Desktop Library**  
[Click to access!](#)



**Delphi Graphics and Game Programming Exposed with DirectX  
7.0**

by *John Ayres*

Wordware Publishing, Inc.

**ISBN:** 1556226373 **Pub Date:** 12/01/99

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

## Index

### A

- A\*, 457
- absolute measurement, 208
- accessing memory, 184-185
- Acquire, 219
- action games, 13
- adventure games, 13
- AI, *see* artificial intelligence
- algorithms
  - fire, 418
  - painter's, 197
  - path finding, 456-457
  - rotation, 387-388
  - scaling, 383
  - star fields, 412
  - texture mapping, 405
- aligned record fields, 363-364
- alpha channel, 116
- animation, 172-174, *see also* frame animation
  - cycle, 175-177
  - issues, 194-195, 197
  - palette, 164-167
- architecture, 17, 24
  - DirectX, 107-108
- artificial intelligence, 32
- artificial intelligence techniques
  - aggression/evasion, 429-430
  - finite state machines, 437-438
  - movement, 426
  - path finding, 455-456

- patterned movement, 433-434
- probability machines, 446
- random movement, 426
- audible focus, 311-312
- audio buffers, 309, 322-323
  - audible focus, 311-312
  - controls, 336-337
  - creating, 322-326
  - duplicating, 334-335
  - format, 309
  - frequency, 336
  - locking, 324-325
  - memory format, 311
  - memory locations, 310
  - optimum configuration, 310-311
  - pan, 336
  - playing, 325-326
  - static, 323-326
  - streaming, 330-331
  - types, 310
  - unlocking, 325
  - volume, 336
- audio format, 309-310
- automatic compiler optimizations, 362-363
- axes, 207-208, 238, 262

## **B**

- back stepping, 460-461
- background graphics, 19
- baseline application, 142-150
- BGR, 78
- binary bit shifting, 372
- BitBlt, 87
- bitmap manipulation, 378
  - bitmap transformations, 378
  - dynamic images, 412
  - lighting tricks, 392
  - rotation, 387
  - scaling, 383
  - texture mapping, 405
  - transition effects, 396-397
- bitmaps, 74, 86-87
  - device-dependent, 86
  - device-independent, 87
  - direct pixel access, 89
  - displaying images, 87
  - displaying under DirectDraw, 133-135
  - loading, 87, 135
  - masking, 92
  - raster operations, 91
- blitting surfaces, 135
- Blt, 176
- BltFast, 135
- board games, 14
- bounding boxes, 188-189
- breadth first search, 456
- brushes, 82
- buffered data, 208-209
- buffers, *see* audio buffers



buttons, 207-208, 238

## C

C/C++, 5-7, 8-9

calibration, 236

call stack elimination, 362-363

canvas, 81

    drawing on, 82-85

card games, 14

casual gamers, 12

CD audio, 346

    implementing, 347-350

    Red Book Audio, 347

children, 11

coefficient, 260

collision detection, 33, 188

    bounding boxes, 188-189

    pixel overlap comparison, 191-192

color

    components, 78

    depth, 77

    keying, 179-181

    palette, *see* palettes

common subexpression removal, 363

compiler optimizations, 363-364

    automatic, 362-263

Component Object Model (COM), 114-115

    GUID, 115

    reference counting, 115

components,

    color, 78

    DirectX, 108-109

conditions, 260, 271-272

    damper, 273

    friction, 272

    inertia, 273

    spring, 272

constant arrays, 374

constant forces, 273

cooperative levels, 209-210, 217

    DirectDraw, 124

    DirectInput, 209-211, 217

    DirectSound, 314-315

coordinate systems,

    3-D, 413

    Cartesian, 412

coordinates, 81

CopyMode property, 91

crawl, 397

CreateDevice, 215

CreateEffect, 282

CreateEvent, 228

CreatePalette, 157

CreateSoundBuffer, 324

CreateSurface, 128

creating

    DirectDraw objects, 120-121

    DirectInput objects, 212

    DirectSound objects, 314



- effect objects, 282-283
- force feedback effects, 279-285
- input devices, 215
- surfaces, 127-129
- custom forces, 278-279

## D

- damper, 273
- data types, 368-369
- DC, *see* device context
- DDB, *see* device-dependent bitmaps
- dead code elimination, 363
- deadzone, 237
- Delphi and DirectX, 109-112
- Delphi Blocks, 473-474
  - code, 481-514
  - enhancements, 480-481
  - game states, 474-475
  - general game architecture, 474
  - graphics, 477-480
  - music, 476
  - player advancement, 474
  - sound, 476
  - user input, 475-476
- Delphi Games Creator (DGC), 112
- Delphi games development, 7
- Delphi games programming, 5-10
- DelphiX, 111-112
- demo state, 39-40
- depth, 194, *see also* z-order
- depth first search, 457
- device context, 80-81
- device-dependent bitmaps, 86
- device-independent bitmaps, 87
- device-specific effects, 276
- devices, *see also* game controller input, keyboard input, mouse input
  - acquiring, 211, 219
  - audio, 312-313
  - CD audio, 346-347
  - enumerating, 212-214
  - game controllers, 205, 236
  - keyboard, 204, 220
  - MIDI, 341
  - mouse, 205, 224
  - peripheral input, 22
  - polling, 31, 209
  - standard input, 22
- DIB, *see* device-independent bitmaps
- direct pixel access, 89
- Direct3D, 109
- DirectDraw, 108
  - displaying bitmaps, 133
  - features, 113
  - functional overview, 113-114
  - initializing, 118-129
  - rendering with GDI, 129-130
  - retrieving capabilities, 139
  - surfaces, 116
- DirectDrawCreate, 120

- DirectDrawEnumerateEx, 118
- DirectInput, 109, 206
  - buffered data, 208-209
  - buttons and axes, 207-208
  - cooperative levels, 209-211
  - device acquisition, 211
  - features, 206-207
  - functional overview, 207
  - game controller input, 236-237
  - immediate data, 208-209
  - initializing, 211-219, 221, 226
  - keyboard input, 220-223
  - mouse input, 224-225
  - polling and event notification, 209
- DirectInputCreate, 212
- direction, 262-263
- DirectMusic, 109
- DirectPlay, 109
- DirectSetup, 109, 519
  - callback function, 521-523
  - features, 519
  - functional overview, 519-520
  - overriding default behavior, 523-525
  - using, 520-523
  - verification of driver upgrade, 526-527
- DirectSound, 108-109, 307
  - audible focus, 311-312
  - audio format, 309-310
  - buffer memory format, 311
  - buffer memory locations, 310
  - buffer types, 310
  - creating buffers, 322-326
  - features, 307
  - functional overview, 308
  - initialization, 312-316, 320
  - optimum buffer configuration, 310-311
  - sound buffers, 309
- DirectSoundCreate, 314
- DirectSoundEnumerate, 312
- DirectX, 104-106
  - and Delphi, 109-112
  - architecture, 107-108
  - baseline application, 142-150
  - benefits, 106
  - components, 108-109
  - components and headers, 111-112
  - drawbacks, 107
  - HAL, 107
  - HEL, 107
  - redistributables, 517-518, 520
  - result codes, 150
  - SDK, 110
- DirectXSetup, 522
- DirectXSetupGetVersion, 521
- DirectXSetupSetCallback, 522
- display mode, setting, 124-125
- dissolve, 397
- Download, 283
- drawing

- lines, 83
- pixels, 82
- polygons, 83-84
- sprites, 185
- text, 85
- DuplicateSoundBuffer, 334
- duration, 261
- DXErrorString, 150
- dynamic images, 412
  - fire, 418
  - star fields, 412

## **E**

- effects, *see* bitmap manipulation, force feedback, sound effects
- EnumDevices, 213
- EnumDisplayModes, 121
- EnumEffects, 276
- enumerating
  - audio devices, 312-313
  - display devices, 118-119
  - input devices, 212-214
  - supported display modes, 121-123
- EnumObjects, 247
- envelope, 263, 296-297
- EULA, 518
- event notification, 209, 227-228
- exceptions, 365

## **F**

- fade, 166
- females, 11-12
- finite state machines, 437-438
- fire, 418
- first-person shooters, 13
- Flip, 130
- flipping, *see* page flipping
- force feedback, 205, 257
  - axes, 262
  - coefficient, 260
  - condition, 260
  - creating effects, 279-285
  - direction, 262-263
  - duration, 261
  - envelope, 263, 296
  - features, 258
  - functional overview, 258-259
  - gain, 261
  - kinetic force, 259
  - magnitude, 260
  - modifying at run time, 293-294
  - sample period, 261
- force feedback effects, 263-266
  - conditions, 271-273
  - constant forces, 273
  - custom, 278-279
  - device-specific effects, 276
  - periodic forces, 274-275
  - ramp forces, 273-274
- forces,

- constant, 273
- custom, 278-279
- kinetic, 259
- periodic, 274-275
- ramp, 273-274
- frame, 172
- frame animation, 172-173
- frames, rendering, 34
- frequency, 336
- friction, 272
- full-screen mode, 117-118

## G

- gain, 261
- game
  - architecture, 24-27
  - logic, 32
  - markets, 11-12
  - modules, 24-27
  - states, 35-41
  - types, 12-14
- game controller input, 236-237
- game controller objects, 238
- game controllers, 205
  - axes, 238
  - buttons, 238
  - calibration, 236
  - deadzone, 237
  - DirectInput initialization, 241-242
  - POV hats, 238
  - range, 237
  - reading input data, 251
  - saturation, 237
  - setting axis properties, 249
  - setting the data format, 246-247
  - Windows functions, 238-239
- game development opportunities, 10
- game development platform attributes, 6-7
- game ending, 26, 34
- game loop, 26-28
  - segments, 30-35
- game over state, 41
- game programming industry, 1-5, 10
- GDI, *see* Graphical Device Interface
- GetAsyncKeyState, 220
- GetAttachedSurface, 128
- GetCapabilities, 241-242, 280
- GetCaps, 139, 316
- GetCurrentPosition, 330
- GetCursorPos, 225
- GetDC, 129
- GetDeviceData, 230-231
- GetDeviceState, 222
- GetEffectInfo, 265
- GetEffectStatus, 291
- GetEntries, 164
- GetForceFeedbackState, 291
- GetObjectInfo, 242
- GetPaletteEntries, 158

- global variables, 374
- Graphical Device Interface, 70
  - rendering with, 129-130
- graphical elements, 72-76
  - bitmaps, 74
  - classification of, 75
  - lines, 73
  - pixels, 72
  - polygons, 73-74
- graphics, 18-19
- GUID, 115

## H

- HAL, 107
- hardware palette, 160
- hat, *see* POV hat
- HEL, 107
- hertz (Hz), 310
- HPALETTE, 97
- human/computer interaction, 204

## I

- IDirectDraw, 120
- IDirectDrawPalette, 157
- IDirectDrawSurface, 128
- IDirectInput, 212
- IDirectSound, 314
- images,
  - displaying, 87-88
  - loading, 87
- immediate data, 208-209
- inertia, 273
- initialization, 25
- input considerations, 205-206
- input data,
  - reading, 251
  - retrieving, 219-220
- input devices, 204-205, *see also* devices
  - enumerating, 212-214
  - peripheral, 22
  - standard, 22
- interactive images, 19
- intermission state, 40
- internal loop, 38
- IntersectRect, 189
- introduction, 25

## J

- joyGetDevCaps, 239
- joyGetPosEx, 239

## K

- keyboard, 204
  - input, 220-223
- kinetic force, 259

## L

- lighting tricks, 392

- lines, 73
- Lock, 184, 324-325
- locking surfaces, 183-184
- lookup tables, 370-371
- loop induction variables, 363
- loop unrolling, 369
- loops, *see also* game loop
  - internal, 38
- lost surfaces, 138-139

## M

- magnitude, 260
- masking, 92
- MCISendCommand, 340-341
- measurements, 208
- melt, 396
- memory,
  - accessing, 184-185
  - allocation, 373
  - buffer format, 311
  - video board, 78
- Microsoft Foundation Classes (MFC), 2
- MIDI, 341
  - implementing, 341-345
- mixing, 304
- modes,
  - non-palettized, 80
  - palettized, 79-80
- motion, 175-177
- mouse, 205
- mouse input, 224
  - device acquisition, 229
  - DirectInput initialization, 226-227
  - event notification, 227-228
  - mouse cursor responsiveness, 224
  - reading input data, 229-232
  - secondary thread creation, 229
  - Windows functions, 225
- movement, 426
  - aggression/evasion, 429-430
  - patterned, 433-434
  - random, 426
- music, 21, 339-340
  - CD audio, 346-350
  - MIDI, 341-345
  - Red Book Audio, 347

## N

- non-palettized modes, 80

## O

- object method types, 366-367
- Object Pascal, 8
- objects
  - IDirectDraw, 120
  - IDirectDrawPalette, 157
  - IDirectDrawSurface, 128
  - IDirectInput, 212
  - IDirectSound, 314

- TBitmap, *see* TBitmap
- TCanvas, *see* TCanvas
- TDIEffect, 264
- TDIEnvelope, 297
- optimization, 357, *see also* profilers
  - automatic compiler optimizations, 362-363
  - compiler optimizations, 363-364
  - considerations, 373-374
  - Delphi-specific, 361-369
  - general, 369-372
  - theory, 358
- optimum buffer configuration, 310
- overflow checking, 364
- overscan, 159

## P

- page flipping, 116-117, 126-127
- painter's algorithm, 197
- palettes, 79, 95, 156-157
  - and GDI functions, 163
  - animation, 164-167
  - DirectDraw, 157-160
  - disconnecting, 163
  - hardware, 160
  - HPALETTE, 97
  - logical vs. system, 96
  - primary surface, 160
  - TPaletteEntry, 97
  - using, 97
- palettized modes, 79-80
- pan, 336
- Pascal, *see* Object Pascal
- path finding, 455-456
  - algorithms, 456-457
  - back stepping, 460-461
  - basic implementation, 457-460
  - enhancements, 470-471
  - example, 462-470
- paused state, 40
- pens, 82
- periodic forces, 274-275
- peripheral input devices, 22
- pixel checking, 183
- pixel overlap comparison, 191-192
- pixels, 72
- Play, 325
- playing state, 40
- PlaySound, 305
- polling, 30-31, 209
- polygons, 73-74
- POV hat, 216, 238
- primary buffer format, 320-321
- primary sound driver, 313
- probability machines, 446
- profilers, 358
  - effective optimization, 360
  - examining the algorithm, 359
  - timing code, 358
- Project JEDI, 110-111

pulse code modulation (PCM), 309  
puzzle games, 14

## R

ramp forces, 273-274  
range checking, 364  
range, 237  
raster graphics, 75  
raster operations, 91  
real-time event-driven programming, 23  
real-time strategy games, 12-13  
rectangles, manipulating, 189  
Red Book Audio, 347  
reference counting, 115  
register optimization, 362  
relative measurement, 208  
ReleaseDC, 129  
rendering with GDI, 129-130  
resolution, 76-77  
Resource Interchange File Format (RIFF), 309  
Restore, 138  
restoring lost surfaces, 138-139  
result codes, 150  
retrieving

- audio device capabilities, 316-317
- available drive space, 521
- DirectDraw capabilities, 139
- effect status, 291
- force feedback system state, 291-292
- input data, 219-220
- the DirectX version, 150-151
- user input, 30-31

RGB, 78  
RingZero GDK, 112  
rotation, 387

## S

sample period, 261  
saturation, 237  
scaling, 383  
ScanLine property, 89-90  
screen transitions, 166-167  
SendForceFeedbackCommand, 292  
SetColorKey, 180  
SetCooperativeLevel, 124, 217, 315  
SetCurrentPosition, 325  
SetDataFormat, 216  
SetDisplayMode, 125  
SetEntries, 164-165  
SetEventNotification, 228  
SetFormat, 320  
SetPalette, 160  
SetParameters, 293  
SetProperty, 217  
setting

- the cooperative level, 124, 314-315
- the display mode, 124-125
- the primary buffer format, 320-321

Shoot 'em case study, 41-66



- shutdown and exit, 27
- simulations, 14
- sound, 20, *see also* DirectSound
- sound buffers, 309, *see also* audio buffers
- sound effects, 20-21, 33, 304
  - mixing, 304
  - simplistic, 304-305
- special effects, *see* bitmap manipulation
- speed delay, 35
- sports games, 14
- spring, 272
- SpriteLib, 20
- sprites, 19, 171, *see also* collision detection
  - attributes, 175
  - drawing, 185
  - transparency issues, 178-181, 183
  - updating positions, 32-33
- stack frames, 364
- standard input devices, 22
- star fields, 412
- Start, 284
- startup, 26
- states, *see* game states
- static buffers, 323-326
- Stop, 284
- streaming buffers, 330-331
- string parameters, 368
- surfaces, *see also* palettes
  - backbuffer, 116
  - blitting, 135
  - creating, 127-129
  - flipping, 130-131
  - locking, 183-184
  - lost, 138-139
  - memory location, 134
  - off-screen, 116
  - primary, 116
  - rendering with GDI, 129
  - unlocking, 187-188

## T

- TBitmap, 87
- TCanvas, 81
- TDDColorKey, 180
- TDDEnumCallbackEx, 119
- TDDEnumModesCallback2, 122
- TDDSCaps2, 129
- TDDSurfaceDesc2, 122
- TDICondition, 271
- TDIConstantForce, 273
- TDICustomForce, 278
- TDIDataFormat, 246
- TDIDevCaps, 242, 280
- TDIDeviceInstance, 213-214
- TDIDeviceObjectData, 231
- TDIDeviceObjectInstance, 242-243
- TDIEffect, 264
- TDIEffectInfoA, 265
- TDIEnumDeviceObjectsCallbackA, 247

- TDIEnumDevicesCallbackA, 213
- TDIEnumEffectsCallbackA, 276
- TDIEnvelope, 297
- TDIJoyState2, 251
- TDIKeyboardState, 222
- TDIObjectDataFormat, 246
- TDIPeriodic, 274-275
- TDIPropDWord, 218
- TDIPropHeader, 218
- TDIPropRange, 219
- TDIRampForce, 274
- TDSBufferDesc, 323
- TDSCaps, 316-317
- TDSEnumCallback, 312-313
- TDSetup\_Callback, 522, 524
- TDSetup\_CB\_UpgradeInfo, 526
- texture mapping, 405
- timing, 194-195, 358-359
- TMCI\_Open\_Parms, 342
- TMCI\_Play\_Parms, 343
- TMCI\_Set\_Parms, 343
- TMCI\_Status\_Parms, 344
- TPaletteEntry, 97, 158
- transition effects, 396-397
- transparency
  - issues, 178-181, 183
  - simulating, 92-95
- TransparentMode, 92
- TTrueColor, 158
- turn-based strategy games, 13
- TWaveFormatEx, 309

## U

- Unload, 283
- Unlock, 188, 325
- unlocking a surface, 187-188
- user input, 21-22
  - retrieving, 30-31

## V

- variants, 367
- vector graphics, 75
- video board memory, 78
- video mode, 76
  - color depth, 77-78
  - resolution, 76-77
- video palette, *see* palettes
- volume, 336

## W

- WaitForMultipleObjects, 229-230
- windowed mode, 117-118
- Windows palette manager, *see* palettes

## Z

- z-order, 194, 197

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.

## Delphi Frequently Asked Questions

- [10291: Problems with printing, preview and paper size in Quick Reports.](#) 19-OCT-99
- [10306: Unidirectional property not supported for all Databases](#) 01-OCT-99
- [10307: BDE Error 000F](#) 01-OCT-99
- [10308: Help | About for BDE Admin version 5.1 says version 5.1](#) 01-OCT-99
- [10309: Blobs are not supported as params for a Stored Proc](#) 01-OCT-99
- [10310: How do I make a TQuery not use memory as scrolls forward](#) 01-OCT-99
- [10312: Opening TQuery results in Operation Not Applicable Error](#) 01-OCT-99
- [10314: Oracle 8.0.5 client is not supported](#) 01-OCT-99
- [10315: Procedures and Functions in Oracle Packages not listed](#) 01-OCT-99
- [10317: Different types of dBASE Unique style indexes](#) 01-OCT-99
- [10318: Table is Full with Paradox Tables](#) 01-OCT-99
- [10319: TIBQuery.RecordCount doesn't return count of all records](#) 01-OCT-99
- [10320: Use GetResults to get output params from MSSQL and Sybase](#) 01-OCT-99
- [10350: Executing Oracle functions](#) 14-OCT-99
- [10357: How to query dates using Day-Month-Year using local SQL](#) 06-OCT-99
- [10359: MS SQL Server 7: "Could not find object" or blank fields.](#) 06-OCT-99
- [10385: Non-blob column in table required to perform operation with Oracle](#) 14-OCT-99
- [10432: How do I add a system menu item for the "right click" of the application's task bar icon?](#) 15-OCT-99
- [10452: How to handle exceptions in TThread objects](#) 18-OCT-99
- [10476: Delphi doesn't run after installing using a Read-Write CD drive](#) 21-OCT-99
- [15002: 'General SQL Error -2147221164 Table: Alias:<Alias Name>'](#) 15-JUL-98
- [15004: Application Server Forms](#) 16-JUL-98
- [15012: 'Syntax error in query. Incomplete query clause'](#) 15-JUL-98
- [15014: Re-executing export From Table](#) 16-JUL-98
- [15021: 'Can't delete or change record. Since related record exists in table ='xxx''](#) 15-JUL-98
- [15023: Files necessary to run Application Server](#) 02-APR-99
- [15030: Alternate security databases](#) 15-JUL-98
- [15032: Single and multiple application server instances](#) 16-JUL-98
- [15039: Other non-BDE file needed to access Microsoft Access databases](#) 15-JUL-98
- [15041: Locating the servers available in the registry](#) 16-JUL-98
- [15048: MSACCESS driver and multi-threaded apps](#) 15-JUL-98
- [15050: Logging in to the remote datamodule](#) 16-JUL-98
- [15057: Current MSACCESS driver limitations](#) 15-JUL-98
- [15059: Database login for multiple clients](#) 16-JUL-98
- [15066: BLObs as SQL input parameters](#) 15-JUL-98

**15068:** Comparison of Sockets, DCOM, OLE Enterprise 16-JUL-98  
**15075:** files that need to be included with applications using MSACCESS 15-JUL-98  
**15077:** Minimizing Connections 16-JUL-98  
**15084:** 'Cannot Load IDAPI Service Library' error 15-JUL-98  
**15085:** Accessing InterBase thru an AT service 02-JUL-98  
**15086:** Multiple Transports 16-JUL-98  
**15093:** 'Translate error, value out of bounds' error 15-JUL-98  
**15095:** Moving the application server to another machine 16-JUL-98  
**15102:** Paradox slow on some computers and not on others 28-JUL-99  
**15103:** Using Date as a Field Name 02-JUL-98  
**15104:** Using Multiple Servers 16-JUL-98  
**15111:** Paradox table version level 15-JUL-98  
**15113:** Accessing Server Methods from Client 16-JUL-98  
**15120:** BDEAdmin:Invalid Filename for Oracle Server Name 05-OCT-99  
**15122:** Server Termination 16-JUL-98  
**15131:** Un-registering a server 16-JUL-98  
**15138:** BDE network installation 26-MAR-99  
**15140:** Dataset event handlers not fired 16-JUL-98  
**15147:** BDE configuration utility 24-APR-98  
**15149:** Dataset event handlers 16-JUL-98  
**15156:** FoxPro support: capabilities 15-JUL-98  
**15158:** DBCLIENT.DLL 16-JUL-98  
**15165:** Microsoft Access; capabilities 15-JUL-98  
**15167:** Modifying application server SQL from the client. 02-APR-99  
**15174:** SQL Parameter binding 24-APR-98  
**15176:** Displaying a bitmap on the client area of an MDI parent form. 02-APR-99  
**15183:** New functions 15-JUL-98  
**15185:** ServerName property (TMidasConnection or TRemoteServer) 16-JUL-98  
**15192:** ODBC Support 15-JUL-98  
**15202:** Sorting on Calculated fields 16-JUL-98  
**15211:** TClientDataSet, auto-increment fields 16-JUL-98  
**15218:** MS SQL error '17824 Unable to write to ListenOn connection ... 06-OCT-99  
**15220:** TClientDataSet.Data property 16-JUL-98  
**15227:** Performance with two or more queries 06-OCT-99  
**15236:** 'Memo too large' error with MSSQL 06-OCT-99  
**15238:** TCP/IP: TMidasConnection, Socket Server 16-JUL-98  
**15246:** TCP/IP: DCOM Dependency? 16-JUL-98  
**15253:** 'Vendor initialization failure: ORANT71.DLL' error. 15-JUL-98  
**15255:** TCP/IP: Load balancing, OLE Enterprise 16-JUL-98  
**15262:** 'External Exception C0000008' or 'Access Violation' 15-JUL-98

**15264:** TCP/IP: Retrieving Client IP and Host Name 02-APR-99

**15271:** error: SYB-12403 ctsend(): user api layer: external error 06-OCT-99

**15273:** Error: "Application is not licensed to run this feature" 16-JUL-98

**15280:** 'ct\_cursor(CLOSE); user api layer: external error 06-OCT-99

**15282:** Error: 'Interface not supported' 16-JUL-98

**15289:** 'External Exception C0000008' error on application exit 06-OCT-99

**15291:** Error: "Could not find IDataBroker in type library" 16-JUL-98

**15299:** Error: "Cannot load IDAPI service library" 16-JUL-98

**15305:** Hetergenious joins 15-JUL-98

**15307:** Error: "DAX Error - Name not unique in this context" 16-JUL-98

**15313:** File Extensions for dBASE and Paradox tables 05-OCT-99

**15315:** Error: "DAX Error class not registered." 16-JUL-98

**15321:** Unsupported Expression Indexes 15-MAY-98

**15323:** Error: "EOleSysError : Error loading type library/DLL" 16-JUL-98

**15329:** 'Insufficient memory for this operation' error 15-JUL-98

**15331:** Error: 'Missing Data Provider or Data Packet' 16-JUL-98

**15337:** Sybase SQL Anywhere update mode 13-JUL-99

**15339:** Error: 'Name Conflicting' 16-JUL-98

**15344:** Error connecting to an Oracle 7.x server using the Oracle 8 client and BDE 5. 01-SEP-99

**15346:** Error: 'RPC Server is unavailable' 16-JUL-98

**15351:** ALTER TABLE statment on Paradox tables removes passwords 15-JUL-98

**15353:** Error: 'Trying to append data to a non-partial dataset' 16-JUL-98

**15359:** 'Interface mismatch. Engine version different' 15-JUL-98

**15366:** Access violation when calling a non-delphi VCL created DLL 15-JUL-98

**15372:** 'Insufficient memory to complete operation' error 06-OCT-99

**15374:** Error: 'Windows Socket Error: (10060), on API 'connect?' 16-JUL-98

**15379:** Access to PDOXUSRS.NET slow with Novell client 05-OCT-99

**15381:** Connection Checklist 16-JUL-98

**15387:** 'Invalid BLOb handle' error 05-OCT-99

**15389:** Displaying a list of constraint error messages 16-JUL-98

**15394:** Checking for NULL in OnUpdateData handler 16-JUL-98

**15401:** Benefits of the ConstraintBroker 16-JUL-98

**15408:** Getting the Win95 Desktop ListView Handle 16-JUL-98

**15414:** Deploying Midas 16-JUL-98

**15420:** Table data subset through IProvider 16-JUL-98

**15426:** TProvider Component vs. Provider Property 16-JUL-98

**15432:** How do I make changes to the VCL code? 16-JUL-98

**15438:** Providing lookup capability 16-JUL-98

**15443:** Reading the autoincrement value of Paradox table 05-OCT-99

**15445:** MasterSource/MasterFields behavior 16-JUL-98

**15450:** [The ordinal 237 could not be located ..](#) 05-OCT-99

**15452:** [ODBC Constraints](#) 02-APR-99

**15457:** [Problem using Oracle 8 driver\(sqlora8.dll\) with Oracle 7.x servers](#) 18-AUG-98

**15459:** [Resource Usage](#) 16-JUL-98

**15464:** [How much does MTS improve connect times?](#) 18-AUG-98

**15466:** [Sending a file via DCOM](#) 16-JUL-98

**15471:** [Phanton records appear on post](#) 05-OCT-99

**15473:** [Error: 'DAX Error - Name not unique in this context'](#) 16-JUL-98

**15479:** [Error: 'DAX Error class not registered.'](#) 16-JUL-98

**15484:** [Vendor Init with Oracle](#) 18-AUG-98

**15486:** [Error: 'EOleSysError : Error loading type library/DLL'](#) 16-JUL-98

**15492:** [Error: 'General SQL Error : Connection does not exist' in threaded application](#) 16-JUL-98

**15497:** [General SQL Error, Can't load MSJTER32.DLL](#) 05-OCT-99

**15499:** [Error: 'Interface not supported'](#) 16-JUL-98

**15504:** [Single Tier Applications](#) 16-JUL-98

**15509:** [Preventing windows task switching](#) 16-JUL-98

**15514:** [Using Stored Procedures](#) 16-JUL-98

**15519:** [Testing IDispatch Descendants](#) 16-JUL-98

**15524:** [Where to get TUpdateSQLProvider component](#) 16-JUL-98

**15529:** [Drag and Drop with Files](#) 16-JUL-98

**15534:** [Getting a handle on your files](#) 16-JUL-98

**15539:** [TSocketDispatcher](#) 16-JUL-98

**15544:** [How to use the TUpdateSQLProvider.OnUpdateRecord method](#) 16-JUL-98

**15549:** [About Deploying DBClient in a Web Application](#) 16-JUL-98

**15554:** [Retrieving a long file name from the command line.](#) 16-JUL-98

**15559:** [Getting the local internet machine name and IP address.](#) 02-APR-99

**15564:** [Using global values in your application.](#) 16-JUL-98

**15569:** [What is Delphi/400 Client Server Suite?](#) 30-MAR-99

**15574:** [Hiding your application from the Windows Taskbar.](#) 16-JUL-98

**15580:** [Using an animated cursor.](#) 16-JUL-98

**15585:** [More Info on Delphi/400 Client Server Suite](#) 30-MAR-99

**15597:** [Retrieving a users login name.](#) 06-JAN-00

**15603:** [Changing the papersize of a print job.](#) 02-APR-99

**15609:** [Purchasing Delphi/400 Client Server Suite](#) 01-APR-99

**15615:** [Creating a rotated font.](#) 16-JUL-98

**15621:** [Converting from Long to Short file names and paths](#) 16-JUL-98

**15627:** [Making text background transparent](#) 16-JUL-98

**15633:** [Getting user name and company info from Windows.](#) 16-JUL-98

**15639:** [Getting the serial number of a disk.](#) 16-JUL-98

**15645:** [Executing a program from my application](#) 16-JUL-98

**15651:** Writing a raw string of data to the printer. 28-JUN-99

**15657:** Multi-Threaded ISAPI DLLs 02-APR-99

**15663:** InterBase access from a web server application 02-APR-99

**15669:** Where can I find evaluation copies of Delphi/400 Client Server Suite? 01-APR-99

**15675:** Upgrade to Delphi/400 for Delphi Developer users 30-MAR-99

**15681:** Technical Support for Delphi/400 01-APR-99

**15687:** Benefits to AS/400 developers and users 30-MAR-99

**15693:** How does Delphi/400 complement and add to Borland's strategy? 30-MAR-99

**15705:** Minimum Hardware and Software Requirements 01-APR-99

**15711:** Displaying the selected row of a DBGrid in a different color 16-JUL-98

**15717:** inserting contents of a file at current position in TMemo 16-JUL-98

**15729:** Using the LzExpand functions in your applications 16-JUL-98

**15735:** Changing the display mode settings in code. 16-JUL-98

**15741:** How many colors will my display mode support 16-JUL-98

**15747:** Generating a Windows Console Application 16-JUL-98

**15753:** Opening an explorer window to a given folder 16-JUL-98

**15759:** Prevent the TEdit component from beeping when invalid 16-JUL-98

**15765:** Parameter value not in allowed range 16-JUL-98

**15771:** Mapping a DLL into another process 16-JUL-98

**15777:** Deleting a directory and all the directories files 16-JUL-98

**15783:** Changing the default width/height of the editor 16-JUL-98

**15789:** Getting a list of the available drives on a system 16-JUL-98

**15795:** Adding a document to the Windows 95 Start Button's documents 16-JUL-98

**15801:** Retrieving the program that is associated with a given extension 16-JUL-98

**15807:** Using Ampersands inside Delphi. 21-MAR-00

**15812:** "Error reading symbol file." error 16-JUL-98

**15818:** Executing a '.LNK' file 16-JUL-98

**15824:** How can I change the drive that MediaPlayer uses to play Audio 16-JUL-98

**15830:** Creating a delay without a timer. 16-JUL-98

**15836:** Converting a color to its VCL string value 16-JUL-98

**15842:** Using escape codes with the format() function 16-JUL-98

**15848:** Aligning maximized forms 16-JUL-98

**15854:** Working with Floating Point, Double, Single, Real, Extended types 16-JUL-98

**15860:** 'Tried to search marked block but is invalid' error 16-JUL-98

**15866:** Displaying the first frame of an AVI File 16-JUL-98

**15872:** How can I retrieve a list of assigned properties 16-JUL-98

**15878:** Save and load metafiles in a BLOB field without using DBImage 16-JUL-98

**15884:** How can I test if a given string contains a valid number? 16-JUL-98

**15890:** Creating a palette with more than one entry 16-JUL-98

**15896:** How can I easily move a Window control at runtime? 16-JUL-98



**15902:** Detecting what direction a printer will rotate for a portrait 02-APR-99

**15908:** Removing a forms caption bar? 16-JUL-98

**15914:** Determining if a given printer is capable of printing a PostScript file 02-APR-99

**15919:** How can I create a dynamic array of TPoints to draw a polygon? 16-JUL-98

**15924:** How can I prevent the user from moving or sizing my form? 16-JUL-98

**15930:** Detecting if the PrintScreen key has been pressed 16-JUL-98

**15936:** Setting the MaxPage property of the PrintDialog component 16-JUL-98

**15942:** Trap scrolling messages for the ScrollBars of a TScrollBox 16-JUL-98

**15948:** Embedding a page break in the Rich Edit control 16-JUL-98

**15954:** How can I disable the Delphi / Borland C++ Builder splash screen 16-JUL-98

**15960:** adjusting the tab location in a TMemo component 16-JUL-98

**15966:** The TDateTime component commonly ignores key input 16-JUL-98

**15972:** Color reduction techniques for graphics. 16-JUL-98

**15978:** Changing the font style of a dbgrid row 16-JUL-98

**15984:** Get a count and list all the controls on a TNoteBook 16-JUL-98

**15990:** Changing the background color of Text 16-JUL-98

**15996:** Displaying a 24 bit True Color bitmap image on a 256 color display 16-JUL-98

**16002:** Data segment too large error. 16-JUL-98

**16008:** Turning an low power compliant monitor off? 16-JUL-98

**16019:** Intercepting the Ctrl-V key in a TMemo 16-JUL-98

**16025:** Subclass the window procedure for a TForm 16-JUL-98

**16031:** Does Delphi have an equivalent for the VB function DoEvents? 16-JUL-98

**16037:** How can I capture an image from a video source? 16-JUL-98

**16043:** Control Panel Applets 16-JUL-98

**16049:** Auto increment Build Number 16-JUL-98

**16055:** Avoiding an Application Error when using EM\_POSFROMCHAR 16-JUL-98

**16061:** Can I use the AssignPrn method to write directly to a port? 02-APR-99

**16067:** CopyCursor 16-JUL-98

**16073:** Selection rectangle 16-JUL-98

**16079:** Draw on frame 16-JUL-98

**16085:** Object Inspector and Enter Key 16-JUL-98

**16091:** EnumFontFamilies() 02-APR-99

**16097:** How can I assign a file variable in a Record? 16-JUL-98

**16103:** Creating a flashing icon 16-JUL-98

**16109:** Getting the Printer driver and port name from TPrinter's 02-APR-99

**16115:** Getting the printing margin of the page 02-APR-99

**16121:** Retrieving the line number that a memo's cursor is on. 16-JUL-98

**16127:** How can I avoid an access violation when using the move procedure 16-JUL-98

**16133:** Using Resident Font 16-JUL-98

**16139:** Printing in Delphi without using the TPrinter Unit 02-APR-99

**16145:** Arrow and Function Keys 16-JUL-98  
**16151:** How can I convert a .dfm file to text and vice versa. 16-JUL-98  
**16157:** Getting the time and date in Universal Time 16-JUL-98  
**16163:** Adjusting form size to accommodate a required client size 16-JUL-98  
**16169:** Querying the Audio Cd Autorun feature of Win32 16-JUL-98  
**16175:** Testing/setting the CD autorun option under Win32 16-JUL-98  
**16181:** Storing/Playing an .AVI file in a database 16-JUL-98  
**16193:** Setting comm port configuration programmatically under Win95 16-JUL-98  
**16199:** Changing com port config via Win95 CommConfig Dialog 16-JUL-98  
**16211:** Sending an image to the printer 16-JUL-98  
**16217:** Using TClientDataSet with local data 16-JUL-98  
**16223:** 'Duplicate resource' error 16-JUL-98  
**16229:** Adding hint windows to applications 16-JUL-98  
**16235:** Using InvalidateRect() to repaint the entire form 16-JUL-98  
**16241:** Using mouse\_event() to simulate mouse events 16-JUL-98  
**16247:** Painting a rainbow color spectrum onto a form 16-JUL-98  
**16253:** Notifying applications that the registry has changed 16-JUL-98  
**16259:** Turning the screen saver off? 16-JUL-98  
**16265:** Stopping TScrollbar button from blinking 16-JUL-98  
**16271:** Where is SYNCOBJS.pas unit? 16-JUL-98  
**16277:** Creating temporary canvas 16-JUL-98  
**16283:** TTimer not firing at correct interval 16-JUL-98  
**16289:** Error Setting Exception Debug Hook 16-JUL-98  
**16295:** Working with objects and functions in other units 16-JUL-98  
**16301:** Programmatically closing another application 16-JUL-98  
**16307:** Formatting a drive under Win32 16-JUL-98  
**16313:** Disabling the Win95 start button 16-JUL-98  
**16319:** Determining minor version number 16-JUL-98  
**16325:** Inserting a file at the cursor 16-JUL-98  
**16331:** Indenting a block of code 16-JUL-98  
**16337:** Controlling shortcut keystrokes in a string grid 16-JUL-98  
**16343:** Disabling ALT-F4 to prevent form closing 16-JUL-98  
**16349:** Prevent window update 16-JUL-98  
**16355:** Preventing multiple application instances 16-JUL-98  
**16361:** Programatically installing printer driver 16-JUL-98  
**16367:** Simultaneous printing to multiple printers 02-APR-99  
**16373:** Stack Overflow, Runtime error 202; 16-JUL-98  
**16379:** Closing popup menu in system tray application 16-JUL-98  
**16385:** Keeping a glyph's transparent area transparent. 16-JUL-98  
**16390:** Transparent property to work for a CoolBar background bitmap 16-JUL-98

**16395:** How can I determine if a given machine has a math coprocessor. 16-JUL-98

**16400:** How can I dial out through the modem under Win32? 16-JUL-98

**16405:** Retrieving BDE system information 16-JUL-98

**16409:** What does the error 'Token not found' 02-JUL-98

**16410:** How can I use a true DWORD value passed to my Delphi application? 16-JUL-98

**16415:** Going to the next compiler error 16-JUL-98

**16420:** Changing the input caret 16-JUL-98

**16425:** Is there any way to record a keyboard macro in the IDE? 16-JUL-98

**16430:** How can I tell in code if my application is running on Windows NT 16-JUL-98

**16435:** How can I detect if a given key was pressed during application startup? 16-JUL-98

**16440:** Duplicate resource errors 16-JUL-98

**16445:** How can I detect if a non-menu key has been pressed? 16-JUL-98

**16450:** Changing the default project directory 16-JUL-98

**16455:** Getting bin names from your printer 02-APR-99

**16460:** Changing the port of a printer 02-APR-99

**16465:** Getting the actual size of a TComboBox 16-JUL-98

**16470:** Converting a number from one base to another 16-JUL-98

**16475:** Getting printer capabilities 02-APR-99

**16480:** Changing the location of the object repository 16-JUL-98

**16485:** Getting property information 16-JUL-98

**16490:** Moving controls at runtime 16-JUL-98

**16495:** re-mark a Delphi 16-Bit executable to be a Windows 4.0 application 02-APR-99

**16500:** Does Delphi have an equivalent to the Visual Basic SendKeys function? 16-JUL-98

**16505:** Using Tapi 16-JUL-98

**16510:** Setting custom tabstops in a ListBox. 16-JUL-98

**16515:** Compile Error: Too many files open 16-JUL-98

**16520:** How can I create a transparent form? 16-JUL-98

**16525:** Using WM\_COPYDATA. 16-JUL-98

**16527:** Borland Assist for Delphi/400 30-JAN-02

**16530:** Selecting a cell in a DBGrid 16-JUL-98

**16535:** Trapping non client areas 16-JUL-98

**16545:** Changing the printer setup between pages 02-APR-99

**16550:** Text in Windows metafiles 16-JUL-98

**16555:** How can I get the serial number of an Audio CD? 16-JUL-98

**16560:** Error: 'Mismatch in datapacket' 16-JUL-98

**16565:** Accessing new interfaces in a RemoteDataModule 16-JUL-98

**16575:** Reassigning events at runtime 16-JUL-98

**16580:** Showing two forms allowing the first to retain focus 16-JUL-98

**16585:** Disabling close button of a window 16-JUL-98

**16595:** Creating a Resource only DLL 16-JUL-98

**16600:** Deleting an executable while it is running 16-JUL-98

**16605:** Windows Dialer dials out when I am already connected to the internet 02-APR-99

**16610:** displaying an application associated icons 16-JUL-98

**16615:** getting DLL function parameter listings 16-JUL-98

**16620:** Loading a locale file into the THTML component 02-APR-99

**16625:** Printing a web page using the HTML control 02-APR-99

**16630:** Illegal Operation, Windows Lockup, IDE 16-JUL-98

**16635:** doing work while application is Idle 16-JUL-98

**16640:** passing functions/procedure to other functions/procedures 16-JUL-98

**16645:** printing a TStringGrid or a TDBGrid 02-APR-99

**16650:** printing in color 02-APR-99

**16655:** rotating a Metafile 16-JUL-98

**16660:** saving TFont information 16-JUL-98

**16664:** Colored text on a TStatusBar 16-JUL-98

**16668:** Typecasting 16-JUL-98

**16672:** IDE highlighting the incorrect line 16-JUL-98

**16676:** Determining the version/release of Delphi 3 01-APR-99

**16680:** Component Palette Corruption Problems 16-JUL-98

**16683:** Runtime Error 255 02-APR-99

**16687:** Doing a complete reinstallation of Delphi 3 02-APR-99

**16691:** Calls to the abort method fail to compile 16-JUL-98

**16695:** Changing the main form of an app. at runtime 16-JUL-98

**16699:** Showing dbgrid is disabled 16-JUL-98

**16703:** Detecting the full path/filename of a DLL 16-JUL-98

**16707:** Programmatically dropping down a menu 16-JUL-98

**16711:** Executing the explorer find file dialog 16-JUL-98

**16715:** Radiogroups and focus 16-JUL-98

**16719:** Trapping the Windows system key 'PrintScreen' 16-JUL-98

**16723:** Accessing HKEY\_LOCAL\_MACHINE under NT without Admin. rights 16-JUL-98

**16727:** Detecting how many jobs are in the print spooler 16-JUL-98

**16731:** Bitwise AND evaluates different under Delphi 3 16-JUL-98

**16735:** Dialogs displaying 'No to all' button instead of mbHelp 16-JUL-98

**16739:** '\$2109 Cannot initialize the BDE' error when deploying InstallShield application 16-JUL-98

**16743:** Raise components: 'Can't Load Package' error 16-JUL-98

**16747:** Local InterBase: 'Internal Error; Near: query shellmgr' error 02-APR-99

**16751:** Cannot write EXE or DLL to disk. 16-JUL-98

**16754:** How can I have a TBitBtn component that has a word wrapped caption? 16-JUL-98

**16758:** Bitmaps and captions on TBitBtn 16-JUL-98

**16762:** Detecting print errors 02-APR-99

**16766:** Disabling system menu in code 16-JUL-98

**16770:** [Avoiding flicker on DrawCell](#) 16-JUL-98

**16774:** [How can I determine the names of the installed comm ports?](#) 16-JUL-98

**16778:** [Extracting Icons](#) 16-JUL-98

**16782:** [Adjusting number and with of columns in a TFileListBox](#) 16-JUL-98

**16786:** [Notifying forms of global changes](#) 16-JUL-98

**16790:** [How can I activate different hints for each cell in a grid](#) 16-JUL-98

**16794:** [Dynamically drawing a transparent image](#) 16-JUL-98

**16797:** [which rows have been selected in a TDBGrid](#) 02-JUL-98

**16798:** [Porting strings from 16 to 32-bit](#) 16-JUL-98

**16801:** [Distributing IDAPI with your EXE](#) 02-JUL-98

**16802:** [Overwrite in TMemo and TEdit](#) 16-JUL-98

**16806:** [Small bitmaps in a TPopUpMenu](#) 16-JUL-98

**16810:** [Refreshing the windows desktop](#) 16-JUL-98

**16814:** [Explanation of rounding](#) 16-JUL-98

**16818:** [StretchDraw on an icon](#) 16-JUL-98

**16822:** [How can I determine the week number of a given day in the year?](#) 16-JUL-98

**16825:** [How do I change a specific IDAPI configuration setting?](#) 02-JUL-98

**16826:** [Sending a message to all controls on a form](#) 16-JUL-98

**16830:** [Resizing an array](#) 16-JUL-98

**16834:** [How can I place a string that is padded with spaces into a](#) 16-JUL-98

**16838:** [Timed beeps from the PC speaker](#) 16-JUL-98

**16841:** [TQuery, error 'Creating Cursor handle'](#) 13-MAY-98

**16842:** [How can I place a bitmap in a Metafile?](#) 16-JUL-98

**16845:** [What are IDAPI and SQL links](#) 02-JUL-98

**16846:** [Images conflicts between different versions of IE](#) 16-JUL-98

**16849:** [Informix Support](#) 02-JUL-98

**16850:** [Debugging into the VCL source code](#) 16-JUL-98

**16854:** [Big-endian and little-endian formatted integers](#) 16-JUL-98

**16858:** [Displaying documents in the default browser](#) 02-APR-99

**16861:** [Mask edit for TDBEdit controls](#) 02-JUL-98

**16862:** [Disabling Full Window Drag](#) 16-JUL-98

**16865:** [Packing dBase tables](#) 02-JUL-98

**16866:** [MaxLength property of a DBEdit is not automatically set](#) 16-JUL-98

**16869:** [How can I assign a Null value to a numeric field in a table?](#) 02-JUL-98

**16870:** [How can I get the default Windows folders](#) 16-JUL-98

**16873:** [Getting current record from a TDBGrid](#) 02-JUL-98

**16874:** [Changing the font in a hint window](#) 16-JUL-98

**16878:** [Converting a TColor value to a HTML Color string](#) 02-APR-99

**16881:** [What is the equivalent in BCB to a Pdox TCursor](#) 02-JUL-98

**16882:** [Using JPEGs from resource files](#) 16-JUL-98

**16885:** error Index out of range with TTable FindNearest or FindKey 02-JUL-98

**16886:** System Modal Windows on WinNT 16-JUL-98

**16889:** error initializing Database engine 02-JUL-98

**16890:** Playing MPEG files in Delphi apps 16-JUL-98

**16894:** Running apps on Windows startup 16-JUL-98

**16898:** Alloting CPU time 16-JUL-98

**16902:** Yielding cycles to other applications 16-JUL-98

**16905:** creating indexes with the TTable create function 02-JUL-98

**16906:** Reducing the resource requirements of TNotebook and TTabbedNotebook 16-JUL-98

**16909:** Accessing MS Access data with BCB 02-JUL-98

**16910:** Changing the root key 16-JUL-98

**16913:** Is there a programmatic way to add an alias to the IDAPI.CFG 02-JUL-98

**16914:** Show a AVI clip in full screen zoom mode 16-JUL-98

**16917:** Max Fields for dBase & Paradox Tables 02-JUL-98

**16918:** Play an AVI file that has sound, silently 16-JUL-98

**16922:** Error warnings too long 16-JUL-98

**16925:** TTable AddIndex() with ixUnique and Paradox tables 02-JUL-98

**16926:** Flush binary file from disk cache to disk 16-JUL-98

**16929:** How do I do a locate on a non-indexed field? 02-JUL-98

**16930:** Determine the last access time of a given file 28-JUN-99

**16933:** using return instead of tab or mouse to move off a control 02-JUL-98

**16934:** Formatting and displaying integers with commas 16-JUL-98

**16938:** Create hints for Menu items 16-JUL-98

**16942:** Native Access driver and ISAPI/NSAPI 02-APR-99

**16945:** Problems with MS Access 02-JUL-98

**16949:** Clearing the window in a console application 16-JUL-98

**16953:** Changing the owner of a component at runtime 16-JUL-98

**16957:** Accessing a grid's column/row by name 16-JUL-98

**16960:** Converting ASCII files to Paradox tables 02-JUL-98

**16961:** Reducing flicker on component redraw 16-JUL-98

**16965:** Assigning a comp type to an integer 16-JUL-98

**16969:** Testing if you are connected to the internet 02-APR-99

**16972:** Rebuilding Paradox Index files 02-JUL-98

**16973:** Seting the number of fixed columns in a TDbGrid 16-JUL-98

**16977:** Changing the default form size 16-JUL-98

**16981:** Getting the handle of a Window containing a DOS box 16-JUL-98

**16985:** Refreshing the TDriveComboBox drive list 16-JUL-98

**16989:** Removing drives listed in a TDriveComboBox 16-JUL-98

**16993:** Determining if you have Delphi v3.02 installed 01-APR-99

**16996:** Defining PRIMARYLANGID(), SUBLANGID(), and MAKELANGID() macros 16-JUL-98

**17000:** Forcing edit mode in a TListView via a keypress 16-JUL-98

**17004:** Differences between Create(Self) and Create(Application) 16-JUL-98

**17008:** Using the Shell API function SHBrowseForFolder() 16-JUL-98

**17012:** Adding accelerator characters to TTab sheets 16-JUL-98

**17016:** Limiting the amount of text entered in a TEdit 16-JUL-98

**17020:** Detecting if the system time has been changed 16-JUL-98

**17024:** Trapping for when a user is done resizing a window 16-JUL-98

**17028:** Using the WIN API high resolution performance counter 16-JUL-98

**17032:** Keyboard shortcut to a component that has no caption 16-JUL-98

**17036:** Getting modem status under Win32 16-JUL-98

**17039:** BDE 4.51 causes Application Error 02-JUL-98

**17040:** Disabling a radio button in a TRadioGroup 16-JUL-98

**17044:** Click a speed button in code 16-JUL-98

**17048:** What is the tag property useful for? 16-JUL-98

**17052:** Using an instance of TPrinter 16-JUL-98

**17056:** Customizing TTrackBar 16-JUL-98

**17060:** Detecting tab key press 16-JUL-98

**17064:** adding system menu items to a form 16-JUL-98

**17068:** Resizing StringGrid Columns 16-JUL-98

**17071:** Endless looping with the TMediaPlayer 16-JUL-98

**17075:** Frame count of an AVI 16-JUL-98

**17079:** drawing transparent text onto a TBitmap 16-JUL-98

**17083:** displaying seconds, minutes, tracks of an AVI 16-JUL-98

**17087:** Clearing the contents of a Canvas 16-JUL-98

**17091:** Shift, Ctrl, Alt key checking 16-JUL-98

**17095:** making a Delphi DLL with as little in the uses clause as possible 16-JUL-98

**17099:** Moving a form by clicking in its client area 16-JUL-98

**17103:** Keeping an application minimized 16-JUL-98

**17107:** Clearing the recent Documents from the Start Menu 16-JUL-98

**17110:** Changing RichEdit Font styles with HotKey combinations 16-JUL-98

**17114:** using the Lookup method to return values from multiple columns 16-JUL-98

**17118:** Copying files using the Standard Windows Copy file dialog box 16-JUL-98

**17122:** Using an Icon as the Glyph in a TSpeedButton 16-JUL-98

**17126:** Getting stronger type checking 16-JUL-98

**17130:** Getting a string representation of a Windows error code 16-JUL-98

**17134:** Getting the path to the directory where Windows was installed from 16-JUL-98

**17138:** responding when the user minimizes or maximizes a Form 16-JUL-98

**17142:** Creating a custom word break procedure 16-JUL-98

**17146:** Creating a Lookup field at runtime 16-JUL-98

**17150:** 'Object not found' with STANDARD database 16-JUL-98

**17154:** Sharing data dictionaries 16-JUL-98  
**17158:** 'Your application is not enabled for use with this driver' error 16-JUL-98  
**17162:** open or create a project using the TReport component 16-JUL-98  
**17166:** Specify a DefaultExpression for a TField object 16-JUL-98  
**17170:** Getting values in a TDBGrid before they are posted 16-JUL-98  
**17174:** Invalid class string' error during connect 16-JUL-98  
**17178:** 'Cannot load IDAPI service library' error during connect 16-JUL-98  
**17182:** Remote datasets for multiple clients 16-JUL-98  
**17186:** Error reading Database1.SessionName 02-APR-99  
**17190:** Error reading Session1.SessionName 02-APR-99  
**17194:** Client application hang 16-JUL-98  
**17198:** 'Provider name was not recognized by the server' error 16-JUL-98  
**17202:** 'Could not find interface IDataBroker in type library' error 16-JUL-98  
**17206:** Blank form at design or run time 16-JUL-98  
**17210:** Saving a QuickReport custom preview to text 16-JUL-98  
**17214:** TQRExpr does not recognize DisplyFormat settings 16-JUL-98  
**17218:** Automating Word 7 16-JUL-98  
**17221:** How to use Access97 drivers with BDE 4.51 02-JUL-98  
**17222:** New methods not showing in code insight window 16-JUL-98  
**17226:** 'EOleSys..operation unavailable' error 16-JUL-98  
**17230:** Access violations with VC++ created OCX controls 16-JUL-98  
**17234:** 'Error saving I(Interface)' 16-JUL-98  
**17238:** 'Error 0: RLINK32 error opening file (TypeLibrary).tlb' error 16-JUL-98  
**17242:** ActiveForms not displaying in Internet Explorer 16-JUL-98  
**17245:** Keeping the Local IBServer from starting up automatically 13-MAY-98  
**17246:** How can I get serial number of my drive 16-JUL-98  
**17250:** 'TActiveFormX declaration missing or incorrect' error 16-JUL-98  
**17254:** Missing ProviderName in TClientDataSet 02-APR-99  
**17258:** Propagating constraints from middle-tier to client 02-APR-99  
**17262:** 'DAX Error: Unexpected failure.' error 16-JUL-98  
**17266:** Determining Drive Type 16-JUL-98  
**17270:** Client launching 02-APR-99  
**17274:** 'DAX Error: Server execution failed' error 16-JUL-98  
**17278:** 'DAX Error: Access is denied' error 16-JUL-98  
**17282:** Old ActiveForm information displayed 02-APR-99  
**17286:** What directory is the ActiveForm in 02-APR-99  
**17290:** How do I launch the Control Panel (and its applets) 16-JUL-98  
**17294:** 'Invalid filename' error 28-JUN-99  
**17298:** 'Invalid configuration parameter for alias 28-JUN-99  
**17302:** Conditional defines for all Delphi versions 02-APR-99



**17306:** Launching a web browser 02-APR-99

**17310:** TClientSocket and TServerSocket 02-APR-99

**17314:** Installing old Delphi 2 components 02-APR-99

**17318:** Rebuilding components 16-JUL-98

**17322:** Using Visual Basic arrays in Delphi 02-APR-99

**17326:** Specify an overlay image for a node in a treeview 16-JUL-98

**17330:** Opening an application's help file 16-JUL-98

**17334:** TToolBar and TSpeedButton misaligned 02-APR-99

**17338:** Intel's Lan Desk management Suite causes lockup 16-JUL-98

**17341:** Where is CERTINST.DOC ? 02-JUL-98

**17342:** Declaring a SafeArray in the type library editor 16-JUL-98

**17346:** Setting a transparent background color for TImageList 16-JUL-98

**17350:** DBGrid fixed leftmost column 16-JUL-98

**17354:** Component palette shown incorrectly - Matrox Millenium card 02-APR-99

**17358:** Using FindFirst to search for files. 16-JUL-98

**17362:** Getting an handle to a window in another application. 16-JUL-98

**17366:** Using an AVI file with TAnimate 02-APR-99

**17370:** Migrating a project from Delphi 1 or 2 to Delphi 3 16-JUL-98

**17378:** Checking drive ready status. 16-JUL-98

**17382:** External function failure when passing boolean parms. 16-JUL-98

**17386:** No default printer selected error message. 02-APR-99

**17390:** Creating a non-VCL console project. 16-JUL-98

**17394:** Printing to file. 02-APR-99

**17398:** Getting the longfilename from a file. 16-JUL-98

**17406:** Getting the filename and path of a local table. 16-JUL-98

**17410:** Temporarily turning off range checking. 16-JUL-98

**17414:** Getting the taskbar handle. 16-JUL-98

**17418:** Why am I having trouble displaying and/or printing a very large bitmap? 21-MAR-00

**17422:** Copying a information from one memo field to another using TBlobStream 16-JUL-98

**17426:** Broadcasting a message. 16-JUL-98

**17430:** Registering a file extension and context menu operations 16-JUL-98

**17434:** Custom InPlace Editor in StringGrid 16-JUL-98

**17438:** Delphi starup failure. 16-JUL-98

**17442:** Creating Non-Visual Components Without Icons 16-JUL-98

**17446:** Compiler options. 16-JUL-98

**17450:** Turning on Screen saver with code. 16-JUL-98

**17454:** Returning application's exit code. 16-JUL-98

**17458:** Extracting an icon from a .exe or .dll. 16-JUL-98

**17462:** Determining if Delphi Running. 16-JUL-98

**17466:** Checking if TrueType fonts are available. 16-JUL-98

**17470:** Why Delphi drawing functions draw one pixel short. 16-JUL-98  
**17474:** Recompiling complib. 16-JUL-98  
**17478:** KeyStrokes: Opening files in Delphi editor. 16-JUL-98  
**17482:** Getting the Windows version in code? 16-JUL-98  
**17486:** How do I create a power function? 16-JUL-98  
**17490:** Local Interbase Startup Configuration. 02-APR-99  
**17494:** Preventing Components from Resizing 16-JUL-98  
**17498:** Embedding a quite in a string. 16-JUL-98  
**17502:** How do I restart Windows in code. 16-JUL-98  
**17506:** Implementing the VB TwipsPerPixel in Delphi 03-AUG-99  
**17510:** Converting 16bit resource to 32bit. 16-JUL-98  
**17514:** Creating resource files. 16-JUL-98  
**17518:** Using the WinApi GetObject to get a bitmap. 16-JUL-98  
**17522:** calling windows Beep() function 16-JUL-98  
**17526:** Performing undo on an edit control. 16-JUL-98  
**17533:** Embedding a copyright symbol in a string? 16-JUL-98  
**17537:** Adding TrueType fonts to windows in code. 16-JUL-98  
**17544:** Converting resources from 16bit to 32bit. 16-JUL-98  
**17547:** TDateTime does not set the field out of 1900 13-MAY-98  
**17548:** Copying contents of a control to the clipboard. 16-JUL-98  
**17552:** Checking available disk space on large drives. 16-JUL-98  
**17555:** Dropping a combobox's list in code. 16-JUL-98  
**17559:** Setting a form's maxwidth in code. 16-JUL-98  
**17563:** Getting the windows directory. 16-JUL-98  
**17567:** Getting the size of the desktop. 16-JUL-98  
**17571:** Hiding the windows taskbar. 16-JUL-98  
**17575:** Keeping an application in a minimized state? 16-JUL-98  
**17582:** Getting the current line number of a TMemo? 16-JUL-98  
**17591:** Calling DeleteFile function 16-JUL-98  
**17595:** Using far pointers. 16-JUL-98  
**17599:** String Trim functions. 16-JUL-98  
**17603:** Virtual key constants. 16-JUL-98  
**17607:** Changing the Windows wallpaper in code? 02-APR-99  
**17610:** Passing TPrinter handle to GetDeviceCaps() 16-JUL-98  
**17614:** Memo Input and line breaks. 16-JUL-98  
**17618:** Changing windows default printer in code. 02-APR-99  
**17622:** Raster operations. 16-JUL-98  
**17626:** How do I create a Resource only DLL? 16-JUL-98  
**17630:** Using GetTimeZoneInformation() function? 16-JUL-98  
**17634:** Trying to delete a running .exe? 16-JUL-98

**17638:** Determining is a canvas is a Metafile Canvas? 16-JUL-98

**17641:** Access97 ODBC with BDE 4.51 02-JUL-98

**17642:** Disabling autodial when using internet controls. 02-APR-99

**17646:** Displaying a file's associated icon. 16-JUL-98

**17649:** Can I get function params from a dll. 16-JUL-98

**17653:** Loading a local html file in the HTML control 02-APR-99

**17657:** Printing a WEB page with the HTML control? 02-APR-99

**17661:** Preventing system lockup in Win95. 16-JUL-98

**17665:** Using the Application's OnIdle Event. 16-JUL-98

**17669:** EHRNTRW.DLL not found 01-APR-99

**17673:** 'Index List' error 01-APR-99

**17677:** Passing a function to a procedure or vice versa. 16-JUL-98

**17681:** How do I Print a TStringGrid / TDBGGrid? 16-JUL-98

**17685:** How do I print in color? 16-JUL-98

**17692:** Saving a TFont Object. 16-JUL-98

**17696:** Listbox selected property 16-JUL-98

**17700:** Getting a list of valid time zones from windows. 16-JUL-98

**17704:** What types of Routers are able to be used? 01-APR-99

**17707:** Deploying the BDE: InstallShield can't find BDE drivers 08-JUN-98

**17708:** BCD Number Truncate to only 4 Decimal Places 02-APR-99

**17712:** Requirements for running CO400INS 01-APR-99

**17716:** Changing font color on standard win controls. 16-JUL-98

**17720:** Typecasting a PChar to a longint. 16-JUL-98

**17724:** How do I use Paths and Geometric Pens? 16-JUL-98

**17728:** How do I convert a color to a shade of gray? 16-JUL-98

**17732:** Overcomming Stream read error. 16-JUL-98

**17736:** Drawing in scale without using Windows Mapping Modes? 16-JUL-98

**17740:** How can I eject a CD-ROM in code? 16-JUL-98

**17744:** Extracting RGB components of a color. 16-JUL-98

**17748:** How do I create an icon from a bitmap? 21-MAR-00

**17752:** How do I create a bitmap from an icon? 16-JUL-98

**17756:** Does my CD-ROM Drive contain an Audio CD? 16-JUL-98

**17759:** How can I tell if the mouse is over my form? 16-JUL-98

**17763:** Is my machine connected to the network. 16-JUL-98

**17767:** Storing icons in your application for use at runtime? 16-JUL-98

**17771:** How do I draw using LoEnglish Mapping Modes? 16-JUL-98

**17774:** Non VCL Bitmap buffering 16-JUL-98

**17777:** Positioning the caret in an edit control. 16-JUL-98

**17780:** Color problems in 256 color mode. 16-JUL-98

**17784:** How do I recurse sub directories? 16-JUL-98

**17787:** Sending a file to the Windows Recycle Bin? 16-JUL-98  
**17791:** Justifying edit text. 16-JUL-98  
**17798:** Read and BlockRead file function failure in 32bit. 16-JUL-98  
**17800:** Record alignment in 32bit pascal. 16-JUL-98  
**17804:** Calling windows RegisterClass function. 16-JUL-98  
**17807:** Sending keystroke #255 to a windows control. 16-JUL-98  
**17810:** TQuery RequestLive results in error "Table/View does not exist." 04-AUG-98  
**17811:** Freeing an object stored in a list. 16-JUL-98  
**17814:** StrPCopy and long strings 16-JUL-98  
**17818:** How can I draw directly on the Windows Desktop? 16-JUL-98  
**17822:** Copying a 400 table to a different type of table. 01-APR-99  
**17825:** CO400 and AnyNet 01-APR-99  
**17828:** Making ini file changes permanent. 16-JUL-98  
**17831:** Nested fields in DBGrid 16-JUL-98  
**17834:** Covert string of (zeros and ones) to binary. 16-JUL-98  
**17836:** Bad PC First port on line1 01-APR-99  
**17838:** Finding the version info for OS/400 01-APR-99  
**17841:** Linking an animated cursor to a resource file. 21-MAR-00  
**17845:** DataSetFields don't display in Nested Table 16-JUL-98  
**17848:** External Exception EEFACE when connecting 18-AUG-98  
**17849:** Adding install to Win95 Add/Remove programs. 16-JUL-98  
**17852:** Query with a string does not work properly 01-APR-99  
**17855:** CO400INS library created, not CO400 01-APR-99  
**17857:** PtInRegion pixel function 16-JUL-98  
**17859:** F004 Error 01-APR-99  
**17861:** How can I close the CD Tray in code? 16-JUL-98  
**17863:** Setting break on exception in Delphi 4 01-APR-99  
**17865:** Where is "View | Project Source" 01-APR-99  
**17867:** Creating a file association for my application? 16-JUL-98  
**17869:** How to move Client Objects to a different AS/400 01-APR-99  
**17871:** Simulating a mouse move in code. 16-JUL-98  
**17873:** Viewing currently loaded DLLs/Modules from within Delphi 4 01-APR-99  
**17875:** Delphi .dcu format 16-JUL-98  
**17877:** 'Setup cannot find any system', using SNA Server 01-APR-99  
**17879:** Detecting display changes. 16-JUL-98  
**17881:** Executing and application by extention. 16-JUL-98  
**17883:** Handling command line params with spaces 16-JUL-98  
**17885:** Transfer initialization (1) =error 55 01-APR-99  
**17887:** Creating new constructors 16-JUL-98  
**17889:** Reading dos environment variables 16-JUL-98

**17891:** Not authorized to use RSTLIB command 01-APR-99

**17893:** Protection Error 01-APR-99

**17895:** Getting the Lowest common denominator. 16-JUL-98

**17897:** Delphi/400 requires QSECOFR login 01-APR-99

**17899:** Placing multiple copies of a component on a form. 16-JUL-98

**17901:** How do I launch a control panel applet? 16-JUL-98

**17903:** APPC Allocate Error No retry 01-APR-99

**17905:** Integer to 'Binary String' 16-JUL-98

**17907:** Detecting PlugNPlay hardware changes? 16-JUL-98

**17909:** Launching your default browser on a URL. 16-JUL-98

**17911:** Creating a Win95 Shell icon application. 16-JUL-98

**17913:** Making sure changes survive a power failure 16-JUL-98

**17915:** Calculating a point using angles and distance 16-JUL-98

**17917:** Calculating an angle from two points 16-JUL-98

**17919:** Changing delphi default directory 16-JUL-98

**17921:** Changing the printer port in code. 16-JUL-98

**17923:** TCombobox drop down size. 16-JUL-98

**17928:** Finding distance between two points. 16-JUL-98

**17930:** Determining if an object supports a property. 16-JUL-98

**17932:** Determining if a combobox is in dropped state. 16-JUL-98

**17934:** How do I plot a line? 16-JUL-98

**17936:** Moving a control at runtime with the mouse. 16-JUL-98

**17938:** Debugging CORBA clients and servers 16-JUL-98

**17940:** Placing a grid in edit mode programatically. 16-JUL-98

**17942:** APPC LU NAME ARE INVALID OR NOT FOUND 01-APR-99

**17944:** Trapping messages in non client area of a form. 16-JUL-98

**17946:** Creating a PolyPolygon using a point array. 16-JUL-98

**17948:** Converting a RGB color to a CMYK color 02-APR-99

**17952:** Metafile fonts and the VCL 16-JUL-98

**17954:** "APPC MEMORY ALLOCATION ERROR" with Delphi 400 01-APR-99

**17956:** Connecting to a remote CORBA server 16-JUL-98

**17958:** Error 'There are no fonts installed'. 16-JUL-98

**17960:** Can't create local description file 01-APR-99

**17962:** Making the drop-down list appear faster 16-JUL-98

**17964:** Why isn't a network install of Delphi 3 supported? 16-JUL-98

**17966:** Getting the summary function to work 16-JUL-98

**17968:** Adding sums of fields 16-JUL-98

**17970:** Is there a TQRDBMemo? 16-JUL-98

**17972:** Switching between different data sources 16-JUL-98

**17974:** Improving performance 16-JUL-98

**17976:** Printing Delphi controls using QuickReports 02-APR-99  
**17978:** Using additional data sets in expressions 16-JUL-98  
**17980:** Stretching memos 16-JUL-98  
**17982:** Setting the caption of the preview window 16-JUL-98  
**17984:** Printing to another printer than the default one 16-JUL-98  
**17986:** Printing calculated text 16-JUL-98  
**17988:** Cannot find A2E table 01-APR-99  
**17990:** A sum(if(.....,0)) always returns 0 16-JUL-98  
**17992:** Font problems on NT 4.0 16-JUL-98  
**17994:** Printing a footer on only the last page 16-JUL-98  
**17996:** Getting AddPrintable to work 16-JUL-98  
**17998:** No default printer selected when using Quickreports 02-APR-99  
**18000:** The Count function 16-JUL-98  
**18002:** Using a TQRExp component at run-time 16-JUL-98  
**18004:** Detecting if there's a disk in a drive 16-JUL-98  
**18006:** Converting strings to dates 16-JUL-98  
**18008:** Where is the 40COMUPD file 16-JUL-98  
**18010:** Cannot find 'CO40032.DLL' 01-APR-99  
**18012:** Cannot create local description file 01-APR-99  
**18014:** Object Activation 16-JUL-98  
**18016:** Protection Error, Runtime version 01-APR-99  
**18018:** 'Record not found' or 'End Of File' 01-APR-99  
**18020:** Signal IO error 01-APR-99  
**18022:** SQL open error 02-APR-99  
**18024:** Unknown Field Name 01-APR-99  
**18026:** Finding the RepositoryId of a CORBA Server 16-JUL-98  
**18028:** IDAPI Service Library not Found 01-APR-99  
**18030:** Delphi/400 Crash in Fields Editor 01-APR-99  
**18032:** Preventing changing pages in a page control 16-JUL-98  
**18034:** Crash in lookup field or dblookuplist 01-APR-99  
**18036:** Delphi/400 : Queries against \*LIBL 01-APR-99  
**18038:** Composite key with blank field inside 01-APR-99  
**18040:** Opening a file with many indexes is slow 01-APR-99  
**18042:** Only one connection is available via Sna/Server router 01-APR-99  
**18044:** Delphi 3/400: Record blocking doesn't work 01-APR-99  
**18046:** If the last value of key is blank, findKey doesn't find any record 01-APR-99  
**18048:** Client DataSet doesn't work with Delphi3/400 01-APR-99  
**18050:** Update, AppendUpdate Modes for BatchMove don't work 01-APR-99  
**18052:** Low Speed AS/400 with TCP/IP desynchronises 01-APR-99  
**18054:** Delphi 3/400: 2 TQueries on the Same Form 01-APR-99

- 18056:** TQuery with NULL values causes hang 01-APR-99
- 18058:** Delphi 3/400: OnFilterRecord causes hang 01-APR-99
- 18060:** Cancel does not unlock locked record 02-APR-99
- 18062:** Cannot write more than 223 characters in a dataarea 01-APR-99
- 18064:** Not all fields allowed with Decision Cube 01-APR-99
- 18066:** Delphi 3/400: Invalid Parameter with BDE Admin 01-APR-99
- 18068:** Delphi 3/400: Delphi pads out strings with nulls 01-APR-99
- 18070:** Delphi 3/400:4 Digit integers are not SmallInt 01-APR-99
- 18072:** Delphi 3/400: Modifying Memo fields 01-APR-99
- 18074:** Delph 3/400: Float fields are rounded 01-APR-99
- 18076:** Delphi 3/300:Closing TQuery results in a GPF 01-APR-99
- 18078:** Delphi 3/400:Table list displays only 499 tables 01-APR-99
- 18080:** Delphi 3/400: ApplyUpdates doesn't work 01-APR-99
- 18082:** Getting a cool menu in your app 16-JUL-98
- 18084:** Running 16-bit Delphi on Windows NT4 26-MAY-98
- 18086:** Installing Delphi from a hard drive. 14-MAR-00
- 18088:** "PPC PARTNER LU NAME NAME NOT FOUND OR INVALID" error 01-APR-99
- 18090:** Creating Delphi 4 Style Toolbars/Menus 01-SEP-99
- 18727:** Merging the configuration file has failed 13-JUL-99
- 18737:** Oracle 8 and returning the REF CURSOR from a stored procedure 01-SEP-99
- 18749:** Unknown Internal Operating System Error 01-SEP-99
- 18752:** Creating NonMaintained or .NDX indexes on dBASE Tables 01-SEP-99
- 18758:** Registering service applications under NT 01-APR-99
- 18761:** Automating Word 8 (word97) 10-MAR-99
- 18763:** Error: Too many fields in table create 01-SEP-99
- 18765:** Detecting if a mouse has a wheel 02-APR-99
- 18767:** Oracle's Oracle 7.x ODBC drivers, VARCHAR2 >255 01-SEP-99
- 18770:** IBM CAE 5 PTF 3 and 4 for DB2 Universal server and client 01-SEP-99
- 18772:** Informix SE, errors while working with Borland Informix SQL Links 01-SEP-99
- 18774:** Microsoft SQL Server "Unexpected EOF from SQL Server" 01-SEP-99
- 18776:** Microsoft SQL Server error 17824 Unable to write to ListenOn connection 01-SEP-99
- 18778:** Sybase DBLIB, Access Violation in KERNEL32.DLL 01-SEP-99
- 18780:** Sybase CTLIB, multiple stored procedure result sets 01-SEP-99
- 18782:** DB2 Win3.1 and Win95/WinNT combined installation. 01-SEP-99
- 18784:** IBM DB2 CLI driver errors SQL0551N and SQL0805N 01-SEP-99
- 18786:** BDE >=4.0, Intersolve ODBC 3.0, TQuery CachedUpdates, TUpdateSql, ApplyUpdates, null/  
blank string 01-SEP-99
- 18788:** Oracle Integer to BDE or Delphi float, ENABLE INTEGER 01-SEP-99
- 18789:** SQL LINKS >=4.0 BLOB SIZE parameter 01-SEP-99
- 18798:** SQL Explorer reports Oracle 8 database as Oracle 7 database 01-SEP-99

- 18804:** Access Violation when opening BDE Object in DataModule of DLL 01-SEP-99
- 18806:** Sybase/"record has been modified by another user" error with datetime fields 01-SEP-99
- 18808:** Delphi4, Oracle 7.x stored procedures with multiple parameters with one being a REF Cursor 01-SEP-99
- 18823:** Sybase 11.5 server causes problems for SQL Explorer 26-MAR-99
- 18825:** Preventing changing to a page in a page control 01-SEP-99
- 18827:** Installing Delphi 4 From a Network Location 01-SEP-99
- 18829:** Vendor Init when using Oracle 7.2 with Delphi4/BDE5 26-MAR-99
- 19235:** Oracle 8 and writing Data to a LOB (blob/clob) field. 26-MAR-99
- 19238:** Date Field Precision For Borland Database Formats 01-SEP-99
- 19240:** SQL against Access tables with special characters in field names, such as # 26-MAR-99
- 19242:** Vendor INIT error when connecting to DB2 Version 5 26-MAR-99
- 19244:** SQL Error returns error when trying to get TableList for DB2 26-MAR-99
- 19254:** Interbase, cached updates and Blob not found error 26-MAR-99
- 19255:** MS Access and trying to detect Null blobs using TBlobField.IsNull method 26-MAR-99
- 19257:** Installation error -113 01-SEP-99
- 19258:** Installation error -117 01-SEP-99
- 19259:** Missing DATAPUMP.EXE error when deploying BDE with ISEXPRESS 29-JUL-99
- 19260:** Missing component palette under NT 01-SEP-99
- 19261:** Random access violations in the IDE 01-SEP-99
- 19262:** Installation error " ins5176" at end of installation 10-MAR-99
- 19263:** Multiple splash screens keep popping up 01-SEP-99
- 19264:** "ICON IMAGE IS NOT VALID" error when installing 01-SEP-99
- 19271:** Blob have limit of 1000000 bytes when connecting via ODBC 26-MAR-99
- 19272:** IDCO400.DLL and IDCO400.CFG not marked as shared DLLs. 01-APR-99
- 19274:** Non-Blob column in table required to perform operation with MSSQL 16-NOV-99
- 19279:** Cannot Load IDAPI Service library when connecting to Access 26-MAR-99
- 19280:** Vendor INIT failed error when connecting via Delphi/400 01-APR-99
- 19285:** How to prevent closing of an undocked ToolBar 10-MAR-99
- 19288:** Error: "BORDBK40.DLL is missing or not registered" 10-MAR-99
- 19291:** Paradox errors when using SQL servers and cached updates 10-MAR-99
- 19294:** #\$(\*(# is not a valid identifier. 10-MAR-99
- 19296:** In Memory Tables use System Language Driver 26-MAR-99
- 19297:** Deleting all records from dBASE table may work differently between 16 bit and 32 bit. 26-MAR-99
- 19312:** AS/400 DataBasename setting in BDE alias. 01-APR-99
- 19379:** The beta version of the NetManage InterNet Controls has expired 01-APR-99
- 19382:** Oracle fields of type MLSLABEL are not supported 26-MAR-99
- 19384:** BatchMove of Time field from Paradox to Oracle results in timestamp in the year 3000. 26-MAR-99
- 19387:** How can I generate an error for SELECT statements that divide by zero? 26-MAR-99



- 19389:** What is a good BDE Alternative? 22-FEB-99
- 19395:** Delphi is not brought to the front when breakpoints are hit in Windows 98 10-MAR-99
- 19396:** GPF in module MACXW4.DRV 10-MAR-99
- 19414:** "Socket subsystem error" during install of InterBase 01-MAR-99
- 19417:** Updating your AS/400 OS release does not effect your Delphi/400 install 01-APR-99
- 19428:** Error: APPC Allocate Error Security Not Valid 01-APR-99
- 19430:** Error: Nested Transactions Not supported 26-MAR-99
- 19432:** DbiOpenTableList does not list Informix views 26-MAR-99
- 19436:** Access CrossTab queries queries 26-MAR-99
- 19438:** Incorrect Application Icon 01-APR-99
- 19439:** Creating a custom TInplaceEdit for TDBGrid 01-APR-99
- 19440:** Delphi 3/400 and Delphi 4/400 installed on the same machine 01-APR-99
- 19443:** Generating auxiliary tables for Local SQL 26-MAR-99
- 19444:** Live Queries do not support TO \_Date function 01-APR-99
- 19447:** curNATIVEHNDL, dbNATIVEHNDL, stmtNATIVEHNDL nor supported for Access 26-MAR-99
- 19452:** 0 Dates and Informix 26-MAR-99
- 19455:** Live query against ADT's are not supported 26-MAR-99
- 19457:** Filters are not supported against ADTs. 26-MAR-99
- 19459:** BLOBs and CLOBs not supported in REF child table 26-MAR-99
- 19461:** IN Not supported for Live Queries against SQL Databases 26-MAR-99
- 19462:** Creating Floats in Access Table results in integers being created 26-MAR-99
- 19464:** Fieldtypes that get translated to BCD 26-MAR-99
- 19465:** Default values not imported to Data Dictionary for Access 26-MAR-99
- 19467:** Can't Create Data Dictionaries for ODBC connections to Access and Oracle 26-MAR-99
- 19469:** Oracle 8 RAW fields are limited to 255 bytes 26-MAR-99
- 19470:** Delphi/400 deployment fails if BDE partial install is deployed 01-APR-99
- 19490:** ODBC and DB2 don't do Prepare's 05-OCT-99
- 19495:** Row Level locking not supported in drivers other than Paradox, dBASE and FoxPro 26-MAR-99
- 19497:** DbiGetSysVersion does not report minor releases 26-MAR-99
- 19504:** Special SQL syntax required to INSERT into BLOBs/CLOBs when parameters are >= 4K. 13-JUL-99
- 19505:** Sybase / MSSQL ARITHABORT ON and "Divide by zero occurred." error. 13-JUL-99
- 19506:** Informix Online and ROW LEVEL LOCKING (LOCK MODE ROW) 13-JUL-99
- 19507:** Oracle database link, cursor, ORA-03106: fatal two-task communication protocol error 13-JUL-99
- 19508:** Raising MSSQL and Sybase errors with a severity <=10 do not get raised by SQL Links. 26-JUL-99
- 19557:** New Interbase SQL Links support in BDE 5.01 (SQLINT32.DLL v5.01.24) 13-JUL-99
- 19558:** MSSQL SQL Links driver and the LANGDRIVER setting 26-JUL-99
- 19559:** BDE 12550 Invalid configuration parameter error 13-JUL-99
- 19560:** "lock timeout" error, TIMEOUT and MAX QUERY TIME parameters. 13-JUL-99

**19561:** Oracle user functions, ORA-06553 expression of of wrong type 13-JUL-99

**19565:** MSSQL 7 SQL Links (v5.01.23) feature support, MS DBLIB (NTWDBLIB.DLL) 13-JUL-99

**19566:** SQL Explorer left pane cannot be resized. 13-JUL-99

**19602:** Computer freezes at splash screen when Delphi is started. 30-MAR-00

**19603:** Computer freezes when Delphi is started under Windows 98 01-APR-99

**19614:** Safe Mode and VGA mode. 21-JUN-99

**19615:** Starting in Safe Mode or VGA mode. 21-JUN-99

**19616:** Delphi 4 running slowly. 21-JUN-99

**19618:** Error "Can't find PVCS" when installing Delphi Professional. 21-JUN-99

**19652:** Creating a lookup field in a TDBGrid for Builder 04-AUG-99

**19655:** SET NOCOUNT ON causes "Couldn't perform the edit because another user changed the record" error 28-JUL-99

**19657:** FoxPro IDX files 13-JUL-99

**19658:** How do I get the version information for a file? 21-JUN-99

**19660:** Access and QuickReports 13-JUL-99

**19661:** How do I set the Wallpaper on Windows98/2000 when ActiveDesktop is turned on? 21-JUN-99

**19662:** "Invalid Row ID" when migrating from Oracle 7 to Oracle 8 21-JUN-99

**19663:** MSACCESS Tables with System Database on a CDROM 13-JUL-99

**19664:** General SQL error # 2147221164 13-JUL-99

**19666:** SQL Anywhere and Blobs>32K when RequestLive=False 13-JUL-99

**19667:** Invalid Field Type when creating ODBC DataSource 13-JUL-99

**19669:** Oracle 7 faster than Oracle 8 13-JUL-99

**19674:** Shared Memory Conflict error 13-JUL-99

**19676:** Unable to resolve records. Table name not found 21-JUN-99

**19694:** "Invalid field type" database error with MSSQL server 7 28-AUG-00

**19698:** Oracle wrapped stored procedure and the Oracle SQL Links driver. 13-JUL-99

**19699:** Database Desktop and the number of displayed digits after the decimal point. 13-JUL-99

**19700:** Obtaining a list of available BDE language drivers 13-JUL-99

**19701:** Unrecognized Database Format error when opening Access table 13-JUL-99

**19727:** \$2B05 "Permission denied." MS Terminal Server, application using the Borland Database Engine 13-JUL-99

**19729:** Maximum number of DBPROCESSES already allocated 13-JUL-99

**19731:** NT Authentication/Integrated security 13-JUL-99

**19741:** No such method error on BOA\_init() 04-AUG-99

**19742:** ODBC Connections via Microsoft ODBC Text driver 13-JUL-99

**19744:** Unable to use IDE tools to access and/or update tables in Informix database 04-AUG-99

**19773:** External Exception EEFACE under Winnt 4.0 04-AUG-99

**19778:** Editing record generates BDE Error 10259 (\$2813) 28-JUL-99

**19779:** Character fields from Access ODBC connection are reported as being double their true size 28-JUL-99

**19781:** BatchMove from TEXT doesn't create Time or TimeStamp fields. 28-JUL-99

**19784:** Oracle MSLABEL fields 28-JUL-99

**19785:** Dates with single digit years operate differently with a leading zero 28-JUL-99

**20073:** How to avoid "Random" Exceptions being raised when using TJPEGImage or the JPEG unit. 12-NOV-99

**20090:** System Resource Capacities in Windows 95 17-NOV-99

**20100:** "Expected type..." error using ADO to access numbers in Oracle Table. 18-NOV-99

**20158:** Hidden or missing IDE 29-NOV-99

**20179:** How do I execute a stored procedure with TIBSQL? 30-NOV-99

**20180:** Garbage added to end of Memo field with ODBC connection 30-NOV-99

**20195:** "Cannot load an IDAPI service library" error with a message of "SQLORA8.DLL" 01-DEC-99

**20234:** TeamSource Error(4): Invalid extension, in file '..\Archives\root.tsdata' 28-MAR-00

**20243:** Conflict between InteliMouse driver v2.2d and TDBLookupComboBox 03-DEC-99

**20330:** BDE Administrator not responding. 14-DEC-99

**20362:** "Could not perform corba dispatch, no interface repository found" error. 16-DEC-99

**20386:** The integrated debugger and NT 20-DEC-99

**20393:** Parameters to Access Long Text fields limited to 255 characters 21-DEC-99

**20394:** Versions of Oracle Supported 21-DEC-99

**20395:** Versions of DB2 supported by BDE 5.10 21-DEC-99

**20420:** Parameter "ParamName" Not Found error when using an ADOQuery and Parameters 23-DEC-99

**20421:** Why are my Data Access components not showing? 23-DEC-99

**20433:** Interpreting Cryptic Jpeg Error Number's 29-DEC-99

**20468:** How to enable the cut and copy operations for the TWebBrowser component in Delphi 5. 04-JAN-00

**20553:** How do I Debug an In-Proc COM Server? 17-JAN-00

**20559:** General SQL Error when accessing a memo field in MS SQL 6.5. 18-JAN-00

**20566:** Master/Detail relationships with data-aware controls. 19-JAN-00

**20580:** How SQL Links determines if an MSSQL table can refreshed. 19-JAN-00

**20590:** Stored Procedures not working in MS-SQL 7 19-JAN-00

**20632:** What version of PVCS does TeamSource™ require? 25-JAN-00

**20641:** How do I change an ISAPI DLL project to a CGI project, or vice-versa? 26-JAN-00

**20644:** Loading an ISAPI DLL with Informix results in Error. 27-JAN-00

**20646:** ISAPI DLL when connecting to Informix gives sql.iem error 27-JAN-00

**20659:** Delphi 5 trial edition premature expiration 28-JAN-00

**20661:** "Operation not applicable" when opening ClientDataSet. 28-JAN-00

**20689:** How do I set the desktop icon color in code, or make the text behind them transparent? 01-FEB-00

**20716:** Customizing the time/date display of the TDateTimePicker 21-MAR-00

**20755:** Does the BDE use Client side or server side cursors for MSSQL? 07-FEB-00

**20782:** How to obtain IDAPI32.LIB and IDAPI.H 09-FEB-00

**20791:** Versions of the BDE that support MSSQL 7 10-FEB-00

**20802:** Access Violation when opening any Access Table 11-FEB-00

**20903:** Missing icons for the office components on the Server tab. 29-FEB-00

**21480:** Why can't I connect to an Access database using the BDE and native MSACCESS driver? 28-MAR-00

**21484:** How do I prevent my application's main window from showing when my application loads? 29-MAR-00

**21516:** How to capture the URL and title of the current webpage being viewed. 28-MAR-00

**21519:** Items disappearing from the Uses tab in the Type Library editor. 28-MAR-00

**21543:** How do I get a list of CDROM drives on a computer? 29-MAR-00

**21548:** New Interbase driver configuration parameter COMMIT RETAIN. 29-MAR-00

**21549:** Code sample to detect if there is a sound device present on a computer. 29-MAR-00

**21550:** Converting an integer into a binary string 30-MAR-00

**21552:** RichEdit in a COM server is not being correctly created 29-MAR-00

**21553:** Getting "unmapped SQL error code 10038" when connecting to MS SQL. 29-MAR-00

**21564:** Initializing an array or record to null. 30-MAR-00

**21573:** Why is my system freezing when I start Delphi, just after closing the Just-in-time debugging dialog box? 30-MAR-00

**21592:** A final prompt, asking user if they really want to exit. 31-MAR-00

**21701:** Clearing a console window screen 11-APR-00

**21703:** Oracle 8 problems introduced in BDE 5.1.1. 11-APR-00

**21721:** Loading Params for some Stored Procs in Oracle Packages causes AV 12-APR-00

**21725:** Closing and reopening a prepared query with Oracle 8 causes memory leak. 12-APR-00

**21726:** Logging in with an incorrect login results in different errors with Oracle 7 and 8 12-APR-00

**21727:** General SQL Error when connecting to MSSQL 12-APR-00

**21728:** Application Taskbar Button 12-APR-00

**21742:** TTable.Edit always executes SELECT statement 13-APR-00

**21773:** Delphi 3 Midas apps not usable with BDE 5.1.1. 19-APR-00

**21774:** Read only access to FoxPro is recommended. 02-AUG-00

**21821:** BDE uses small packet sizes 27-APR-00

**22086:** How to tell if a form has been created (or any object)? 30-MAY-00

**22090:** Internal Error 2140 with NT Service Application and ADO 30-MAY-00

**22100:** What version of Delphi 5 does the Code Explorer come with? 31-MAY-00

**22101:** How to embed Adobe Acrobat into your application. 31-MAY-00

**22113:** Indices Vs. Keys 31-MAY-00

**22114:** How to use sets in code, such as adding to TDBGrid.Options, at run time 31-MAY-00

**22115:** Setting default value for decimal places in the BDE. 31-MAY-00

**22213:** Non-TThread Threading in Delphi, Access Violation, Invalid Pointer Operation. 09-JUN-00

**22269:** Delphi 5 on Windows 2000 and Windows ME 20-JUN-00

**22390:** Auto-Increment fields in InterBase 18-JUL-00

**22391:** Row-Level Locking in MSSQL 7 18-JUL-00

**22506:** Delphi 1 no longer available. 21-JUL-00

**22584:** With an inproc server that uses Sharemem, the caller AV's on exit. 31-JUL-00

**22586:** Activating the clientdataset closes the query. 31-JUL-00

**22597:** When Quick Report printer settings are changed during a preview at run time, they will have no affect. 02-AUG-00

**22609:** Changing Quick Report BandType at runtime. 31-JUL-00

**22610:** How to assign the Delphi global printer object, to the Quick Reports global printer object. 02-AUG-00

**22696:** Error: "Type mismatch" 09-AUG-00

**22866:** How to send a response early in a web application action. 16-AUG-00

**22867:** How to turn off ISAPI DLL caching on Windows 2000 and IIS5 16-AUG-00

**22940:** Can you drag and drop existing SQL structures directly onto a form? 24-AUG-00

**22985:** Error "Unable to write to registry" during Interbase 5.5 install. 29-AUG-00

**23006:** Passing nil to a variable parameter. 31-AUG-00

**23007:** How to access more than one table with the QuickRep component. 31-AUG-00

**23008:** Variable alignment problems in record type. 31-AUG-00

**23009:** Auto Increment Fields in a ClientDataSet 31-AUG-00

**23011:** Using the requery method in place of the refresh method. 31-AUG-00

**23012:** Checking the BDE version progammatically. 31-AUG-00

**23175:** Finding all files (or files of a certain type) in a given directory 18-SEP-00

**25128:** Creating a BDE alias programmatically from Delphi 24-JUL-00

**25183:** Handling the "Capability not supported" error 24-JUL-00

**25187:** An EDatabase Exception is generated by a Delphi or C++ Builder application 24-JUL-00

**25188:** Handling BDE Engine Errors 24-JUL-00

**25207:** Automatically Generating Data to Create Large Tables Using Delphi 24-JUL-00

**25213:** Handling "General SQL error: violation of FOREIGN KEY constraint . . ." error 24-JUL-00

**25223:** Converting Data in Paradox Tables to InterBase Using Delphi 24-JUL-00

**25238:** How to retrieve an InterBase Blob in Delphi using SavetoFile? 24-JUL-00

**25249:** Calling an IB stored procedure using TStoredProc 24-JUL-00

**25276:** Vendor Initialization Failed error when trying to start InterBase application 24-JUL-00

**25287:** SQL Monitor 24-JUL-00

**25294:** Insufficient memory to complete operation 24-JUL-00

**25337:** When installing BDE based products they overwrite IB v5 gds32.dll with v4.2.1 dl 24-JUL-00

**25360:** Setting the BDE DRIVER FLAGS parameter for individual aliases 24-JUL-00

**25364:** How to insert an InterBase BLOb in Delphi using LoadFromFile? 24-JUL-00

**25382:** Error installing from InterBase: String variable is not large enough for strin 24-JUL-00

**25392:** Getting 'Error creating cursor handle' when using TStoredProc 24-JUL-00

**25433:** Using BLOBs in Stored Procedures through the BDE 24-JUL-00

**25435:** Can a Delphi Application Connect to an InterBase Database on Any Platform? 24-JUL-00

**25493:** How to connect to an InterBase database using the BDE API 24-JUL-00

**25594:** Is it possible to connect to InterBase from a Delphi app without using the BDE? 24-JUL-00

**25617:** Performance techniques when using BDE and Delphi TTables with InterBase 24-JUL-00

**25619:** Do I need Local InterBase on the client machines along with my BDE application? 24-JUL-00

**25620:** An Introduction to the TBatchMove Component in Delphi and C++ Builder 24-JUL-00

**25621:** How to create an About Box in Delphi or C++ Builder 24-JUL-00

**25625:** How to Set Up a C++ Builder or Delphi Form to View InterBase Data 24-JUL-00

**25636:** Using Form Wizard to Quickly Create a Delphi Form 24-JUL-00

**25639:** Connecting via ODBC using Delphi or C++ Builder 24-JUL-00

**25644:** What are Delphi and C++ Builder Action Lists and how do I use them? 24-JUL-00

**25645:** A possible explanation for 'Name not Unique in this context' errors 24-JUL-00

**25647:** Using Variables & For Loops in Delphi 24-JUL-00

**25648:** Peep Show Picture Viewer 24-JUL-00

**25665:** Handling exceptions in Delphi 24-JUL-00

**25669:** Can I create stored procedures via Delphi or C++ Builder? 24-JUL-00

**25670:** How to Use Delphi or C++ Builder to Run a Query in a Text file 24-JUL-00

**25671:** Where are the Delphi 4 InterBase Certificate Keys and IDs located? 24-JUL-00

**25679:** How to create SQL queries with SQL Builder 24-JUL-00

**25682:** Delphi Training: Accepting User Input Using An Edit Box 24-JUL-00

**25683:** Using Control Characters With Delphi 24-JUL-00

**25704:** InterBase array datatype and the BDE 24-JUL-00

**25707:** Creating a Worker Thread for Background Processing 24-JUL-00

**25745:** Why can't I display data in descending order in Delphi? 24-JUL-00

**25774:** I am getting permission denied errors when I run a Delphi or C++ Builder data-aware application. 24-JUL-00

**25807:** Purpose of the interbase Client32 files 24-JUL-00

**25831:** How do I use the InterBase API security functions to add, modify and delete users 24-JUL-00

**25849:** Displaying 4 digit years in Delphi applications 24-JUL-00

**25870:** How do I use the InterBase Express (IBX) SQL monitor? 24-JUL-00

**25915:** BDE Merge Module now available. 25-SEP-00

**25954:** Where are my components? 29-SEP-00

**26068:** Resolving DFM corruption issues with CVS. 17-OCT-00

**26072:** Delphi 5 is unable to start due to problems with a missing export in a .BPL file. 18-OCT-00

**26107:** Where is the file dsgnintf.pas? 24-OCT-00

**26111:** Manually uninstalling Delphi 24-OCT-00

**26151:** Unable to install Delphi 5 after having the Trial edition fully uninstalled 27-OCT-00

**26402:** How to print an HTML document from inside of your application 08-DEC-00

**26407:** Dynamically changing the MainForm of an Application when it starts. 11-DEC-00

**26408:** Help pages are coming up blank in Delphi. 11-DEC-00

**26416:** How to move a string into a TMemoryStream or TFileStream. 11-DEC-00

**26523:** Is Turbo Pascal still available? 02-JAN-01

**26563:** How do I split the editor? 05-JAN-01

**26609:** BORDBK50.DLL is missing or not registered. 11-JAN-01

**26634:** What is "Error 113" during installation? 15-JAN-01

**26667:** Saving your project's breakpoint settings. 19-JAN-01

**26668:** How to set the default form font in Delphi. 19-JAN-01

**26731:** How to display the current user name. 29-JAN-01

**26735:** How to use the Find Error entry under the search heading of the menu bar. 30-JAN-01

**26752:** How to detect if the current user is logged in administrator. 31-JAN-01

**26754:** How to detect the computer name. 31-JAN-01

**26889:** Why does my Application run slowly under Linux? 14-FEB-01

**26890:** Error -10 while installing 14-FEB-01

**26891:** What are the Differences in the install paths for Kylix? 14-FEB-01

**26937:** Why am I getting the error "Operation must use an updateable query"? 21-FEB-01

**27022:** How do I get a TRichEdit to scroll in code? 06-MAR-01

**27031:** Error "Application is not licensed to use this feature" when running a Delphi application. 07-MAR-01

**27039:** Setting a Kylix TListView component's subitem images to nothing 09-MAR-01

**27041:** TIBLicensingService component not found on Component Pallet. 09-MAR-01

**27043:** Trying to revoke a drop target that has not been registered with TWebBrowser. 09-MAR-01

**27067:** Trouble Shooting Errors Received After Adding a Unit to the Uses Section of Your Unit 15-MAR-01

**27128:** How to custom draw menu items 26-MAR-01

**27129:** Selecting an item in a TListView when the checkbox is checked 26-MAR-01

**27138:** How to create Data Link File with Windows 2000 27-MAR-01

**27166:** Assigning an Oracle BLOB greater than 32K must be done using AsMemo 30-MAR-01

**27227:** WHERE clause with greater than 256 ANDs or ORs generates size of optimizer block exceeded error 17-APR-01

**27255:** Finding the column and row of the cursor in a TRichEdit 25-APR-01

**27265:** Corrupted help files. 27-APR-01

**27266:** BDE on multiprocessor machines. 27-APR-01

**27267:** Setting process affinity for your application. 27-APR-01

**27298:** Finding product information from Borland's website for previous versions of Delphi 08-MAY-01

**27308:** Does Delphi 5 include Resource Workshop and InstallShield Express? 09-MAY-01

**27312:** What databases can be used with dbExpress? 10-MAY-01

**27321:** How to get UpdateMode to work for TQuery on a SQL Server 7.0 table. 15-MAY-01

**27329:** How to retrieve the position of the cursor. 17-MAY-01

**27349:** I installed my application with InstallShield and there are files missing. 25-MAY-01

**27369:** Why won't my server socket reconnect? 31-MAY-01

**27370:** Why won't the virtual key codes work in my program? 31-MAY-01

**27373:** VisiBroker applications created with Delphi 5 Enterprise on Windows 2000 31-MAY-01

**27375:** Can you use the Naming Service with the version of VisiBroker that comes with Delphi 5 Enterprise? 31-MAY-01

**27379:** Install option is inactive for the Trial Version of Delphi 5 31-MAY-01

**27394:** Borland Kylix Install Frequently Asked Questions 08-JUN-01

**27405:** Borland Delphi Presales Frequently Asked Questions 13-JUN-01

**27406:** What do I do if I am getting missing or corrupted file messages? 13-JUN-01

**27407:** How do I determine if I have a bad CD? 13-JUN-01

**27411:** Borland Delphi Install Frequently Asked Questions 13-JUN-01

**27428:** Error "database unavailable" when connecting to Interbase via IIS. 15-JUN-01

**27442:** SIGSEGV error when running MySQL on Kylix. 20-JUN-01

**27459:** What is the message "Generating font matrix" when starting Kylix? 26-JUN-01

**27463:** My console application generates a Runtime Error when it starts for no apparent reason. 28-JUN-01

**27478:** What version of TeeChart is supplied with Delphi 6? 05-JUL-01

**27486:** Where can I obtain the help file for the VtChart component? 06-JUL-01

**27506:** How to change file attributes programmatically. 10-JUL-01

**27511:** Insufficient disk space on C: drive while installing Delphi to D: drive. 13-JUL-01

**27522:** Why is there no "Save" button in the save dialog box? 17-JUL-01

**27541:** Problem generating font matrix in Kylix. 30-JUL-01

**27543:** How do I set a custom CVS PServer port? 30-JUL-01

**27544:** Some links and/or executables are missing after running Kylix install. 30-JUL-01

**27545:** Error "permission denied" when trying to run Kylix 30-JUL-01

**27547:** The Buttons are missing or grayed out in Database Desktop. 30-JUL-01

**27549:** How do I create an executable file using the command line directive in Linux with Kylix? 31-JUL-01

**27550:** Error message "Failed to create process" when trying to use the Corba Server and Client Application wizards. 31-JUL-01

**27551:** Getting Fatal Error "File not found: 'OrbPas.dcu'" when using a Corba Object or Corba Data Module in Delphi 6. 31-JUL-01

**27553:** What can I use in Kylix, as a possible replacement for ShellExecute in Windows? 31-JUL-01

**27554:** How to create a console application in Kylix. 31-JUL-01

**27555:** How to change the shape of your forms. 31-JUL-01

**27561:** How to capture the current cursor icon and draw it. 31-JUL-01

**27572:** How to stop IIS when you are writing web applications as ISAPI dll's 02-AUG-01

**27578:** Oracle Stored Procedures Generate Access Violations in ORA805.dll 03-AUG-01

**27579:** How to use TADOTable Seek procedure. 03-AUG-01

**27588:** Lost trial key. 06-AUG-01

**27593:** Error "Unsupported calling convention: REGISTER" in SOAP Application. 06-AUG-01

**27594:** Does Kylix support ODBC? 06-AUG-01

**27595:** Does Kylix support Postgres? 06-AUG-01

**27600:** What is DBGo? 06-AUG-01



**27602:** Do the ADO components come with the professional version of Delphi 6? 06-AUG-01

**27613:** I can't find ADO Express for Delphi on shop.borland.com. Where can I buy ADO Express for Delphi 5? 07-AUG-01

**27647:** What features come with the Delphi 6 free trial? 13-AUG-01

**27656:** Not enough disk space to install Delphi 6 on Windows 98. 14-AUG-01

**27699:** Heterogeneous queries with MSSQL 7 27-AUG-01

**27702:** Why reusing or inheriting from a Remote Data Module is not a good idea. 28-AUG-01

**27703:** How to find what the user path is for their Environment Variables. 28-AUG-01

**27704:** How to download a file from the Internet. 28-AUG-01

**27706:** How to invoke the Windows ShortCut Dialog wizard. 28-AUG-01

**27707:** How to close your application using the Escape key. 28-AUG-01

**27708:** Error 10 while installing Kylix 29-AUG-01

**27718:** How can I make the background color different for alternating lines of text in a TListBox in Delphi? 30-AUG-01

**27719:** Action Manager Component referenced in Text Editor Tutorial is Missing 31-AUG-01

**27726:** WebBrowser1.GoBack and WebBrowser1.GoForward creating 'Unspecified error'. 31-AUG-01

**27753:** How to make sure the function DLLPROC is called when your DLL is unloaded. 11-SEP-01

**27757:** How to create Interbase stored procedures that return parameters with TIBQuery. 13-SEP-01

**27779:** Where is proxies.dcu? 19-SEP-01

**27780:** Missing components in Delphi 6. 19-SEP-01

**27784:** How can I create an identical binary executable in Delphi? 20-SEP-01

**27799:** How to convert numbers into roman numerals. 26-SEP-01

**27805:** How to get the date of the last day in the current month. 27-SEP-01

**27827:** Making changes to my dBase tables causes McAfee to scan tables. 02-OCT-01

**27845:** How to debug an Apache Shared Module 09-OCT-01

**27846:** Problems installing Delphi 6 Professional or Enterprise editions after uninstalling the free personal edition. 09-OCT-01

**27890:** Working around time zone issues with Web Broker 16-OCT-01

**27908:** Where is TReport in Delphi 6? 03-FEB-03

**27914:** Borland license information is either invalid or missing 23-OCT-01

**27924:** Internal Error 2381, "Directory Does not Exist" 25-OCT-01

**27949:** Unofficial WebSnap updates and bug fixes 30-OCT-01

**27958:** Uninstalling Delphi 6 prompts for the CD even after saving the installation information to the hard drive 31-OCT-01

**27966:** Steps to uninstall and reinstall Delphi 6 31-OCT-01

**27980:** Problems registering Kylix 01-NOV-01

**27987:** Error "InstallShield for Windows is unable to start..." when installing Delphi. 06-NOV-01

**28032:** Missing components in Kylix. 16-NOV-01

**28034:** Manually unistalling Kylix. 16-NOV-01

**28057:** How to run executables created in Kylix 29-NOV-01

**28061:** Getting error -145 when trying to apply the Delphi 6 Update 30-NOV-01

**28063:** Is Visibroker for C++ 4.5 supported with Delphi 6? 30-NOV-01

**28070:** How can I avoid expanding a tree node when double-clicking on it? 30-NOV-01

**28072:** How can I tell if I have downloaded the correct Delphi 6 Update? 30-NOV-01

**28084:** Installing the EasySoft ODBC driver from InterBase 6 from Delphi 6 04-DEC-01

**28118:** How to change Kylix directory permissions for all users when installed as root. 13-DEC-01

**28176:** Cannot access deployed EJB in BES 5.0 with a Delphi Client 30-DEC-01

**28178:** Possible solutions for IDE Internal Error in Kylix 2. 31-DEC-01

**28182:** Timeout problems with SOAP services in Delphi 6. 31-DEC-01

**28185:** How to keep seconds from being included in the display when using TDateTimePicker 31-DEC-01

**28186:** How can one prevent a TForm from being moved or resized? 02-JAN-02

**28187:** Delphi 6 on Windows XP 02-JAN-02

**28192:** Kylix 2 Not Registered 03-JAN-02

**28214:** How to make the TDateTimePicker display blank. 09-JAN-02

**28236:** Component properties do not appear in the Translation Manager 14-JAN-02

**28270:** ISAPI DLL used to work with Delphi 5 but doesn't work in Delphi 6 23-JAN-02

**28276:** Access violation in dcc60.dll and Internal Error LA33 in Delphi 6. 24-JAN-02

**28277:** ItemIndex property of TStaticActionList and TVirtualActionList does not represent which item is selected. 24-JAN-02

**28298:** Loading a TListView component that contains SubItems. 31-JAN-02

**28301:** How to make your main form fill up the Desk Top space. 31-JAN-02

**28321:** How to create an example of your problem for Borland Developer Support 31-JAN-02

**28322:** How to generate steps to reproduce your issue for Borland Developer Support 31-JAN-02

**28323:** What to have ready before you call Borland Developer Support 31-JAN-02

**28327:** How to detect the CPU type of the computer the application is running on. 31-JAN-02

**28328:** Getting the date of a file using Delphi. 31-JAN-02

**28329:** How does one change the default directory that Delphi uses to save projects? 31-JAN-02

**28330:** How to clear the document listing found in the Windows Start Menu. 31-JAN-02

**28366:** Unofficial WebSnap Fixes Page 2 06-FEB-02

**28400:** Problems using ADO in a WebSnap application. 13-FEB-02

**28411:** Why can't I run my Kylix application outside of the IDE? 19-FEB-02

**28413:** Where is the browser pane in the web page editor in Kylix? 19-FEB-02

**28414:** Where is the web page preview pane in the code editor? 19-FEB-02

**28417:** Why can't I get breakpoints to work when debugging a COM+ or MTS app in Delphi? 19-FEB-02

**28419:** Web App Debugger incorrectly formats dates based on the current regional settings 20-FEB-02

**28431:** Problems with IBExpress when installing the Delphi 6 Update Pack 2 21-FEB-02

**28445:** Endless Registration Loop when Trying to Register Delphi 6 26-FEB-02

**28447:** How to disable mouse clicking in a TWebBrowser 26-FEB-02

**28451:** How can I use a dialog window to have a user select a folder without a file name? 27-FEB-02

**28457:** How to disable a Menu item with out greying it out. 28-FEB-02

**28467:** How do you display the current line number of a selected record in a DBGrid? 28-FEB-02

**28473:** How can I log off the current user from within my application? 28-FEB-02

**28475:** How to ensure that a path name ends with a trailing path delimiter ('\'). 28-FEB-02

**28478:** How to remove the backslash from the end of a path. 28-FEB-02

**28494:** Operation Not Supported error when calling TSQLQuery.RecordCount 05-MAR-02

**28514:** Unofficial SOAP Bug Fixes 15-MAR-02

**28520:** How do I search backwards in the C++Builder/Delphi editor? 18-MAR-02

**28603:** Unable to locate PDOXUSRS.NET when using Novell client 4.83 08-APR-02

**28605:** Project options window and other IDE windows appear too small. 08-APR-02

**28615:** What causes internal error T2261? 12-APR-02

**28616:** Getting a result set from an Oracle stored procedure in Delphi or C++Builder. 12-APR-02

**28620:** Problem with IBClientDataSet component in Delphi 6. 16-APR-02

**28662:** How to get WebSnap applications to display new lines in a MEMO. 09-MAY-02

**28666:** Where is 2AllBooks.pdf? 10-MAY-02

**28675:** Sending an early response in a WebSnap application loses session values. 10-MAY-02

**28692:** Can't load package dclnet60.bpl 17-MAY-02

**28694:** Missing database login prompt. 17-MAY-02

**28699:** Oracle 9i BDE certification. 28-MAY-02

**28701:** Problems installing / registering free (trial) edition of Kylix on CD. 28-MAY-02

**28718:** Keep getting error: "Maximum number of DBPROCESSES already allocated". 31-MAY-02

**28734:** Setting the Just-in time debugger default in Delphi 6. 06-JUN-02

**28735:** Missing package libpng.so.2 when installing Kylix 2 on Mandrake Linux 8.2. 06-JUN-02

**28740:** Trial and open editions of Kylix and Delphi do not include all files. 10-JUN-02

**28746:** Internal error 2349 while installing Delphi. 12-JUN-02

**28773:** Error "No invocable class registered..." in Midas/Soap app after installing Delphi 6 update pack 2. 17-JUN-02

**28798:** How to catch a 'Tab Character' key stroke inside your TEdit control. 26-JUN-02

**28812:** Kylix freezes after clicking the Register Now button. 27-JUN-02

**28841:** The {Program Name} is linked to the missing export OLEAUT32.DLL:VarNot 01-JUL-02

**28849:** SQL Explorer Form isn't Displaying Controls Properly 08-JUL-02

**28863:** Using TTable generates errors when using Filters 12-JUL-02

**28878:** Access Violation at address ### in module vcl60.bpl or coreide60.bpl when running a project in Delphi 16-JUL-02

**28882:** IDE Internal Error Solutions 17-JUL-02

**28888:** Network initialization failed. Lock violation with file pdoxusrs.net. 22-JUL-02

**28889:** Applying Delphi updates to the evaluation version 23-JUL-02

**28890:** Removed FastNet while installing the Indy components. 23-JUL-02

**28912:** MySQL queries yeild only 10 columns. 25-JUL-02

**28920:** Delphi Generates "Can't Create Delphi32.\$\$\$" Errors when Launching. 26-JUL-02

**28927:** How to Get the Filename of the Process that is Currently Executing. 30-JUL-02

**28944:** [Runtime error 230 when running Kylix application outside of the IDE.](#) 31-JUL-02

**28963:** [How to Convert a String of Integers into an Array of Byte.](#) 05-AUG-02

**28964:** [Converting Double Types from Little Endian to Big Endian Using Variant Records.](#) 05-AUG-02

**28965:** [ISAPI and CGI Applications get Errors Initializing the BDE.](#) 05-AUG-02

**28968:** [My WebSnap Application Executes the Wrong Action, or Skips the Action All Together.](#) 05-AUG-02

**28999:** [Error "Can't load package dclbdbdesign.so.6.9" when starting Kylix 3 for C++.](#) 15-AUG-02

**29013:** [My ActiveX Controls are not Responding to 'Tab' and 'Cursor' Keys.](#) 23-AUG-02

**29027:** [How to Get the Page Name of a WebModule in a WebSnap Application.](#) 28-AUG-02

**29033:** [Kylix 3 freezes when opening or saving a file.](#) 29-AUG-02

**29034:** [Kylix 3 trial - compiler errors state "message catalog not found".](#) 29-AUG-02

**29035:** [Delphi 7 and Visibroker 3.3 support](#) 29-AUG-02

**29037:** [Changing the Desktop wallpaper using Delphi](#) 29-AUG-02

**29045:** [Evaluating Corba using the Delphi 7 Trial](#) 30-AUG-02

**29048:** [Corrupted .pdf files in the Corba EJB demo folder](#) 30-AUG-02

**29049:** [Help files for Idl2Pas in Delphi 7 Enterprise](#) 30-AUG-02

**29050:** [Mystery errors compiling Delphi project.](#) 30-AUG-02

**29054:** [Kylix 3 C++ compiler issue with locales where decimal separator is not a period.](#) 31-AUG-02

**29059:** [Error uninstalling Delphi 6: Error 1926. Could not set file security for file Filename.msi](#) 03-SEP-02

**29064:** [A Better Way to End the Endless Registration Loop when Trying to Register Delphi 6](#) 05-SEP-02

**29078:** [Adding QuickReports to the Delphi 7 Component Palette](#) 10-SEP-02

**29082:** [Where is the Rave PDF File Documentation for Delphi 7?](#) 10-SEP-02

**29083:** [Where is TCorbaConnection in Delphi 7?](#) 11-SEP-02

**29091:** [Problem installing Delphi: Internal Error 2735.](#) 13-SEP-02

**29105:** [Delphi 7 registration wizard is missing text.](#) 18-SEP-02

**29142:** [E2257 in QRvDefine.hpp when compiling TRvProject](#) 29-SEP-02

**29143:** [Cannot register Kylix 3 due to endless registration loop](#) 29-SEP-02

**29156:** [Kylix 3 freezes on start up.](#) 30-SEP-02

**29178:** [Kylix 3 does not install properly on Redhat 8.0](#) 10-OCT-02

**29179:** [Error: "d7reg.exe has encountered a problem and needs to close."](#) 11-OCT-02

**29200:** [During installation, Serial Number and Authorization are invalid](#) 18-OCT-02

**29212:** [When attempting to compile, the following error comes up: 'System.pas' not found](#) 23-OCT-02

**29226:** [Cannot create new connection because in manual or distributed transaction mode](#) 30-OCT-02

**29229:** [How can I generate a log file of the Delphi installation process?](#) 30-OCT-02

**29230:** [How to Cut and Paste WebSnap Script from the Web Page Designer into the Code Editor.](#) 30-OCT-02

**29231:** [Converting a Delphi VisiBroker 3.3 app to Delphi 7 VisiBroker 4.5](#) 30-OCT-02

**29232:** [My Kylix application won't run outside the IDE.](#) 30-OCT-02

**29239:** [How to enable .NET Compiler Warnings when using dcc32 compiler](#) 31-OCT-02

**29240:** ["CoInitialize has not been called" when using TXMLDocument](#) 31-OCT-02

- 29241:** Strange access violations when using TXMLDocument at runtime 31-OCT-02
- 29304:** Limit reached for installs using Kylix serial number 25-NOV-02
- 29306:** Kylix 3 encounters declaration syntax errors in TIME.H 26-NOV-02
- 29313:** ModelMaker demo has expired. Please check the ModelMakers tools website for more info. 30-NOV-02
- 29314:** Upgrading from Delphi 7 Enterprise to Delphi 7 Architect 30-NOV-02
- 29385:** Kylix 3 C++ IDE hangs at the splash screen under Mandrake 9.0 23-DEC-02
- 29390:** Kylix 3 registration fails and indicates the Serial Number is for an upgrade version only 23-DEC-02
- 29395:** Whom should we contact to renew our support or maintenance contract with Borland? 26-DEC-02
- 29396:** How to obtain a Serial Number and Authorization key for Delphi 7 Personal Promotion 27-DEC-02
- 29535:** The procedure entry point RtlUnhandledExceptionFilter could not be located in the dynamic link library ntdll.dll 15-JAN-03
- 29540:** Adding InterBase Server Manager to the Tools Menu of Delphi or C++ Builder 15-JAN-03
- 29541:** Converting a Delphi VisiBroker 3.3 app to Delphi 7 VisiBroker 4.5 16-JAN-03
- 29558:** Accessing InterBase via dbExpress with Kylix produces error: Unable to load libgds.so 17-JAN-03
- 29562:** Can't find ibx.hlp error 20-JAN-03
- 29573:** Unable to register Delphi 7 Trial 27-JAN-03
- 29575:** Checking to see if the latest RTL Update has been applied to Delphi 6 27-JAN-03
- 29586:** Getting error 1629 when attempting to install InstallShield Express 28-JAN-03
- 29587:** Using a DSO on Apache 2.0.43, created with Kylix 3 28-JAN-03
- 29588:** There is not enough space on c:\ to extract this package 29-JAN-03
- 29651:** Following the directions in the websnap readme for compiling Apache for DSO support doesn't work 31-JAN-03
- 29707:** Exception encountered when installing the Rave BE 5.08 update 04-FEB-03
- 29754:** Procedure entry point cannot be found in Rave50VCLBE70.bpl 06-FEB-03
- 29755:** Why can't I use SaveChartToFile and LoadChartFromFile with my TChart component? 06-FEB-03
- 29766:** Delphi 7 and FastNet Components 11-FEB-03
- 29768:** TServerSocket and TClientSocket are Missing from Delphi 7. 11-FEB-03
- 29789:** Problems running WebSnapSurvey IntraWeb demo with Delphi 7. 24-FEB-03
- 29790:** Migrating from Delphi 4: What happened to the routines FMTBCDToCurr and CurrToFMTBCD? 24-FEB-03
- 29802:** Having trouble changing the path that Delphi 6 is installed to 27-FEB-03
- 29804:** Installing Kylix 3 Open Edition on Mandrake 9 27-FEB-03
- 29805:** Class not registered error in Delphi 7 when dropping any component installed with dclisp70.bpl onto a form 27-FEB-03
- 29806:** Class TReport not found 27-FEB-03
- 29821:** Installation incomplete: Unable to locate ModelMaker 05-MAR-03
- 29826:** Where are the Office '97 components in Delphi 7? 11-MAR-03
- 29833:** Intraweb Context Sensitive Help not working 13-MAR-03
- 29839:** Where can I download the updated Registration Wizard for Delphi 7? 18-MAR-03

- 29840:** CLX components not showing up under the System tab on the component palette 18-MAR-03
- 29853:** Icon Image not valid error 24-MAR-03
- 29862:** Error: "Projects macro in project group file is missing or incorrect" 26-MAR-03
- 29865:** problems using valuetype in idl file: idl2pas: fatal error 26-MAR-03
- 29866:** problems with dbExpress connecting to DB2. Get error: SQL0204N "tablename" is an undefined name. SQLSTATE-42704 26-MAR-03
- 29872:** Cannot install Delphi because a previous version is already installed 28-MAR-03
- 29900:** Installing Delphi in Silent Mode 01-APR-03
- 29994:** 'Drive C: does not exist. Please verify the correct drive was given' when trying to import slip activation file 30-APR-03
- 29995:** Rave 4.5 CE version for Linux doesn't print 30-APR-03
- 29996:** Enabling automatic scrolling in a TTreeView 30-APR-03
- 29997:** Numerous users on one Windows 2000 / NT / XP machine 30-APR-03
- 30002:** Failed to create process error when trying to use the CORBA Server or Client Wizards 30-APR-03
- 30003:** When launching Delphi 7 on a Windows 95 machine you get this error: 'The COREIDE70.bpl is linked to missing export USER32.DLL: TrackMouseEvent' 30-APR-03
- 30006:** Create Interface for SOAP server 30-APR-03
- 30007:** Using the Rave Reporting tool, how do I make a Header repeat on each page? 30-APR-03
- 30008:** How do I insert today's date into my Rave Report? 30-APR-03
- 30023:** Unable to resolve error, 'Table is read only.' 09-MAY-03
- 30058:** On Windows XP you get "Error 1606. Could not access network location \." when installing Delphi 29-MAY-03

**Article #10291: Problems with printing, preview and paper size in Quick Reports.**

QUESTION:

I'm having problems with PrintPreview and Paper Size in QuickReports, Is there a fix for this?

ANSWER:

Yes. For Delphi 4 users a QuickReports upgrade to version 3.0.5 for Delphi 4 users is now available from the QuSoft web page at: <http://www.qusoft.com/scripts/download.dll> .

The file is called QR305SD4.EXE. This is the version which shipped with Delphi 5. This version includes some significant bug fixes, including issues with custom page sizes, qr2demo, DIV function, title length, expression wizard and pop-up menu items (right mouse click).

For Delphi 5 users there is now an update to version 5.0.7 available from the same site. This file is called QR307SD5.EXE.

Fixes here include several printer issues, particularly with HP printers which we've heard several reports about. For more detailed information see <http://www.qusoft.com/scripts/download.dll>

Last Modified: 16-DEC-00

## Article #10306: Unidirectional property not supported for all Databases

Question:

What databases support the unidirectional property of a TQuery?

Answer:

Databases that are supported:

Interbase

Oracle

Sybase

MSSQL

Informix

Database that are not supported:

ODBC

DB2

Paradox

Access

dBASE

FoxPro

In databases where the unidirectional property is not supported setting the unidirectional to true does nothing. No error message is generated.

Last Modified: 19-OCT-99



## Article #10307: BDE Error 000F

**Question:** I'm getting BDE Error 000F. What does this mean?

**Answer:** BDE Error 000F usually indicates that the buffer manager ran out of memory, though there may be other causes for this error. Increasing the BDE SHAREDMEMSIZE parameter in BDE Admin may solve the problem, or just delay the error from occurring.

Last Modified: 26-OCT-00

**Article #10308: Help | About for BDE Admin version 5.1 says version 5.1**

Q. How come Help | About for BDE Admin with BDE 5.10 says version 5.01? Is it a bug?

A. For version BDE 5.10 BDE Admin was not recompiled only the version resource was changed. Therefore, Help | About remains unchanged by design.

Last Modified: 27-OCT-99

## **Article #10309: Blobs are not supported as params for a Stored Proc**

Q. Are blobs supported as params (either input or output) to a Stored Proc?

A. No. Trying to do so results in the error "Capability not supported".

Last Modified: 13-FEB-01

## **Article #10310: How do I make a TQuery not use memory as scrolls forward**

Q. I notice that as I scroll my TQuery forward more and more memory is used. How do I prevent this?

A. If your TQuery is forward only cursor, then setting unidirectional to true will resolve the problem.

Note that any memory used by the TQuery is freed when the TQuery is closed.

Last Modified: 01-OCT-99

## **Article #10312: Opening TQuery results in Operation Not Applicable Error**

### Question:

When I open my TQuery, why am I getting an "Operation Not Applicable Error" when my TQuery returns no records?

### Answer:

If you have Unidirectional set to True for your TQuery this error will occur. If you think you may be in this condition, run a select count(\*) query to determine the number of records returned from your Query before running the TQuery.

Note: in Delphi 2 and 3 this condition does not occur.

Last Modified: 19-OCT-99

## **Article #10314: Oracle 8.0.5 client is not supported**

Q. Is Oracle client 8.0.5 supported by the BDE Native driver?

A. No. Due to blob inconsistencies that QA and R&D encountered with this client it is not certified. Version 8.0.4 and 8.1.5 are supported.

"Data Truncated" is a common error that occurs when using the 8.0.5 Client.

Last Modified: 01-OCT-99

## Article #10315: Procedures and Functions in Oracle Packages not listed

**Question:** I have a procedure (or a function) in my Oracle package, but when I get a list of stored procedure names for my TStoredProc it is not listed. Why is this?

**Answer:** Only procedure names listed in *sys.all\_objects* are listed, because returning these results is a fast query. To get a list of procedures in packages, each package would require parsing which would be a slow operation.

Last Modified: 26-OCT-00

## **Article #10317: Different types of dBASE Unique style indexes**

Unique Indexes in dBASE tables have historically operated differently than unique indexes in other databases. Most or all other databases use unique indexes to prevent duplicate entries from being placed in a table. The way dBASE tables have worked is not to prevent duplicate entries from being entered in a table, but to only display unique values in the index when the index is selected.

One side effect of this technique is dealing with deleted records. If you delete the record which displays when the unique index is chosen, no values for the record deleted will be displayed even though they exist in the database until the database is packed.

When Visual dBASE 7 was released it offered a new way to create unique indexes. It added the new option of creating unique indexes which are distinct. Such indexes operate in the manner that most other database operate in that duplicate values are prohibited.

Visual dBASE can create and use either style of index. Delphi and C++ Builder can use either style index, but can only create distinct unique indexes. So, if you need to create unique indexes which are not distinct you must use Visual dBASE to create them.

Visual dBASE is owned by dBASE2000. For additional information on Visual dBASE see [www.dbase200.com](http://www.dbase200.com)

Last Modified: 19-OCT-99



## Article #10318: Table is Full with Paradox Tables

### Question:

I am attempting to add data to my Paradox table but I'm getting the Error "Table is Full". What is the cause of this and how can I add more records?

### Answer:

Paradox tables have a block size which determines the maximum record size for a table and the table's maximum size. By default tables have a block size of 4K. Regardless of the block size, tables can have a maximum of 64K blocks. This means by default a table can grow to 256 MB.

You can increase the block size of a table in 1 of two ways.

1. Run BDE Admin and change the block size for the Paradox driver to a revised number. Legal block sizes are 1024, 2048, 4096, 16384 and 32768. If your block size is larger than 4096 then you must also change level to be at least 5. Finally, create a new table and batchmove your data from your old table to your new table
2. It is possible to change the block size of an existing table by calling dbiDoRestucture. For an example see [www.borland.com/devsupport/bde/files/pxrest.zip](http://www.borland.com/devsupport/bde/files/pxrest.zip)

Last Modified: 19-OCT-99

## Article #10319: TIBQuery.RecordCount doesn't return count of all records

### Question:

When I execute TIBQuery.RecordCount why doesn't it give me a count of all my records in my Query?

### Answer:

TIBQuery.RecordCount does the same thing as TIBSQL.RecordCount. Here is what Help says for TIBSQL.RecordCount:

Use the RecordCount property to see how many records are returned by a query. If the result set is to return 100 rows, RecordCount will only be 100 after all the records have been visited. That is, after looking at the first record, RecordCount is 1, and so forth.

The reason RecordCount operates in this fashion is calculating all the records in a dataset is potentially time consuming operation as it would require scanning through the entire dataset. The most efficient way to calculate the number of records in a dataset is to execute a `SELECT COUNT(*)` statement.

Last Modified: 18-OCT-99

## Article #10320: Use GetResults to get output params from MSSQL and Sybase

### Question:

I'm calling a StoredProc on MSSQL or Sybase that returns a result set and other other output parameters. Why do I never get any values in my output parameters?

### Answer:

For MSSQL and Sybase, parameters are not returned until you go to the end of the result set. You must execute code similar to the following:

```
with StoredProc1 do
begin
  Open;
  Last;
  GetResults;
  //Retrieve output parameters here
end;
```

For more information on GetResults see TStoredProc.GetResults in Delphi or C++ Builder Help.

Last Modified: 19-OCT-99

## Article #10350: Executing Oracle functions

### QUESTION:

Can I execute an Oracle User Function via a Stored Proc?

### ANSWER:

Yes. There are a number of things to remember about this process:

1. Functions cannot be dragged and dropped from SQL Explorer like procedures can.
2. The first parameter in the parameter list (parameter 0) is the return value of the function. The DataType should match the return data type of your function. The name must be Result. The ParamType must be ptResult.
3. If you are creating a TStoredProc dynamically Prepare does not automatically load function parameters with BDE 5.10 like it did with previously with older versions of the BDE. You'll have to use TStoredProc.Params.CreateParam to add all parameters to your TStoredProc.

Last Modified: 18-OCT-99

## **Article #10357: How to query dates using Day-Month-Year using local SQL**

### Question:

How do I query a Paradox or dBASE table using Day-Month-Year instead of Month-Day-Year?

### Answer:

The local SQL engine uses the separator to decide the format of a date. A / is interpreted as M/D/Y. A . is interpreted as D.M.Y.

### Examples:

11/1/99 is November 1, 1999.

11.1.99 is January 11, 1999.

Last Modified: 19-OCT-99

**Article #10359: MS SQL Server 7: "Could not find object" or blank fields.**

**QUESTION:**

What causes BDE Error 8712 "Could not find object" when accessing MS SQL Server 7 table? Other Symptoms could be columns not appearing or blank.

**ANSWER:**

MS SQL Server 7 introduces new field types called NVARCHAR, NCHAR and NTEXT. These types represent Unicode data and are not supported by the BDE.

The workaround is to change these field types to VARCHAR or TEXT.

Please note there are no ODBC drivers which have been certified by Borland which support MSSQL Server 7, only up to MSSQL Server 6.5 (see the Readme.txt file).

We recommend using the ADO components available with Delphi 5 to connect to MS SQL Server 7.

Last Modified: 10-OCT-00

## Article #10385: Non-blob column in table required to perform operation with Oracle

### Question:

Why am I getting the error "Non-blob column in table required to perform operation" when trying to access a BLOB or CLOB with Oracle 8?

### Answer:

TTables and Live TQueries may not allow access to Oracle 8 CLOB or BLOB fields when the user has a role of DBA, EXP\_FULL\_DATABASE, IMP\_FULL\_DATABASE or SELECT\_CATALOG\_ROLE especially if the table was created by a user without one of these roles. Users with these roles return LOB indexes in the list of index definitions because the BDE queries a different set of system views compared to users without these roles.

These LOB indexes do not have corresponding column definitions. The error will occur when the primary key has a name that causes it to be sorted after the LOB indexes in the result set.

The error does not occur for users with roles other than DBA, EXP\_FULL\_DATABASE, IMP\_FULL\_DATABASE or SELECT\_CATALOG\_ROLE.

To determine what roles a the current user has query SYS.USER\_ROLE\_PRIVS.

To determine what roles all users have been granted query SYS.DBA\_ROLE\_PRIVS when logged in as a user with a GRANTED role of DBA, EXP\_FULL\_DATABASE, IMP\_FULL\_DATABASE or SELECT\_CATALOG\_ROLE.

Last Modified: 19-OCT-99

## Article #10432: How do I add a system menu item for the "right click" of the application's task bar icon?

**Question:** How do I add a system menu item for the "right click" of the application's task bar icon?

**Answer:** [FAQ 2054D](#) describes how to add a custom menu item to the system menu of a form, but what if you want to add one to the task bar's right click?

The reason that FAQ 2054D doesn't add anything to the task bar's right click menu is because Delphi creates a hidden window. You have to set the system menu for this hidden window to make it work.

Here is an example of how to do that, and then respond to the proper message:

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    procedure OnAppMessage(var Msg: TMsg; var Handled: Boolean);
  end;

var
  Form1: TForm1;

implementation

const
  SC_MyMenuItem = WM_USER + 1;

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  // Assign the application's OnMessage event to my own
  // procedure so I can check for the hidden window's WM_SYSCOMMAND
  // message.
  Application.OnMessage := OnAppMessage;
  AppendMenu(GetSystemMenu(Application.Handle, FALSE), MF_SEPARATOR, 0, '');
  AppendMenu(GetSystemMenu(Application.Handle, FALSE),
    MF_STRING,
    SC_MyMenuItem,
    'My Menu Item');
end;

procedure TForm1.OnAppMessage(var Msg: TMsg; var Handled: Boolean);
begin
  // Check to see if the message is a system command
```



```
// and the message's wParam is SC_MenuItem
if (Msg.message = WM_SYSCOMMAND) and (Msg.wParam = SC_MenuItem) then
begin
    ShowMessage('Got the message');
    Handled := True;
end;
end;
```

Last Modified: 03-NOV-99

## Article #10452: How to handle exceptions in TThread objects

### Question:

I have a TThread object which may raise an exception in the Execute procedure. When an exception is raised, I want to be able to show that exception to the end user. How do I go about doing this in the most efficient way?

### Answer:

With a TThread object, if you don't catch an exception in the Execute procedure of a TThread, you may get access violations. The Delphi IDE may break fine on the exception, but often when the application is run outside of the IDE you get an "Application error has occurred" exception and your application stops running.

If you don't care about showing the end user that an exception occurred, you can simply wrap your Execute procedure with a try..finally block such as:

```
procedure TMyThread.Execute;  
begin  
  try  
    // Do your thread stuff here  
  except // Eat all exceptions  
  end;  
end;
```

Quite often, this isn't the best solution and you will want to show the message to the end user, or allow your application to further process the message.

The easiest way to do this, is to add an Exception object to your TThread class, and call the appropriate handler based on the type of exception. Here is an example of how to do this. The project consists of one form with a Button placed on it:

```
unit Unit1;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Classes,  
  Graphics, Controls, Forms, Dialogs, StdCtrls;  
  
type  
  TForm1 = class(TForm)  
    Button1: TButton;
```

```

    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

TMyThread = class(TThread)
private
    FException: Exception;
    procedure DoHandleException;
protected
    procedure Execute; override;
    procedure HandleException; virtual;
public
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TMyThread.DoHandleException;
begin
    // Cancel the mouse capture
    if GetCapture <> 0 then SendMessage(GetCapture, WM_CANCELMODE, 0, 0);
    // Now actually show the exception
    if FException is Exception then
        Application.ShowException(FException)
    else
        SysUtils.ShowException(FException, nil);
end;

procedure TMyThread.Execute;
begin
    FException := nil;
    try
        // raise an Exception
        raise Exception.Create('I raised an exception');
    except
        HandleException;
    end;
end;
end;

```

```
procedure TMyThread.HandleException;
begin
  // This function is virtual so you can override it
  // and add your own functionality.
  FException := Exception(ExceptObject);
try
  // Don't show EAbort messages
  if not (FException is EAbort) then
    Synchronize(DoHandleException);
finally
  FException := nil;
end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Create an instance of the TMyThread
  with TMyThread.Create(True) do
  begin
    FreeOnTerminate := True;
    Resume;
  end;
end;

end.
```

Last Modified: 03-NOV-99

## **Article #10476: Delphi doesn't run after installing using a Read-Write CD drive**

### **Question:**

I just installed Delphi using a Read/Write CD drive. It took an unusually long time to install, and now it doesn't run - it just gives an error message. How can I fix this problem?

### **Answer:**

It appears that some Read-Write CD Drives don't work quite right when trying to install Delphi. They may take an unusually long time to install, with the CD-ROM 'clicking' as it does a reading-stopping-reading process. There is a known problem with HyVal IDE drive causing this problem.

The solution is to install a new driver for the RW-CD that hopefully doesn't have the problem. Or, you can try installing from a traditional CD-ROM drive instead of the RW-Drive. If you are on a network, another alternative is to 'Map' someone else's CD-ROM drive and install Delphi from this mapped drive.

Last Modified: 03-NOV-99

## Article #15002: 'General SQL Error -2147221164 Table: Alias:<Alias Name>'

Question and Answer Database

FAQ2B.txt      'General SQL Error -2147221164 Table: Alias:<Alias Name>'  
Category      :Database (MSACCESS)  
Platform      :Win95/NT  
Product       :

Question:

"General SQL Error -2147221164 Table: Alias: "  
error: I get this error when I try and open a MSACCESS  
database. What is the problem?

Answer:

This problem seems to be closely related to a DAO DLL versioning  
problem. The easiest way to resolve this is to un-install Office  
97 from Control Panel | Add/Remove Programs. There is information  
related to this problem in Microsoft's knowledge base  
<http://www.microsoft.com/kb/articles/q141/2/94.htm>

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15004: Application Server Forms

Question and Answer Database

FAQ4D.txt      Application Server Forms  
Category      :DCOM/MultiTier/ActiveForm  
Platform       :All  
Product        :Delphi 3.x

Question:

Do I need a form in the application server?

Answer:

Yes. It doesn't need to be visible, but you do need one.  
To make the form not visible, set  
Application.ShowMainForm := False in the project file.

Example project source:

```
begin
  ?Application.ShowMainForm := False;?
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15012: 'Syntax error in query. Incomplete query clause'

Question and Answer Database

FAQ12B.txt 'Syntax error in query. Incomplete query clause'  
Category :Database (MSACCESS)  
Platform :Win95/NT  
Product :

Question:

"Syntax error in query. Incomplete query clause" error:  
I get this error when I run a query against a table with a space in the table name. I have double quotes around the name but it still does not work. What am I doing wrong?

Answer:

You need to change the double quotes " " to brackets [ ]

Example:

```
SELECT * FROM "MY TABLE NAME" needs to change to  
SELECT * FROM [MY TABLE NAME]
```

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99



## Article #15014: Re-executing export From Table

Question and Answer Database

FAQ14D.txt Re-executing export From Table

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

After using the right-click to create the provider function, how can I re-execute the context menu option to "Export from Table"?

Answer:

Once you have exported a provider interface, the context menu option is no longer visible. To re-display the context menu option you must delete the associated property in the Type Library Editor and press the Type Library Editors Refresh button. You should also delete the "Get\_XXX" entry in the RemoteDataModule source.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15021: 'Can't delete or change record. Since related record exists in table ='xxx'

Question and Answer Database

FAQ21B.txt 'Can't delete or change record. Since related record exists in table ='xxx'

Category :Database (MSACCESS)

Platform :Win95/NT

Product :

Question:

"Can't delete or change record. Since related record exists in table ='xxx', referential integrity rules would be violate" error: What causes this error and how do I get around the problem?

Answer:

The problem is that referential integrity is currently not supported with the MSACCESS driver. Borland recognizes this as a problem and is working on a solution. Currently, the only workaround is to remove referential integrity from the MSACCESS tables. You can see this behavior with Access 95 by changing the RI value in the master table to the same value and trying to post. Access 97 has fixed this problem.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15023: Files necessary to run Application Server

Question and Answer Database

FAQ23D.txt - Files necessary to run Application Server

Category :DCOM/MultiTier/ActiveForm  
Platform :All-32Bit  
Product :Delphi3.x, Delphi4.x,

Question:

What files and DLLs are necessary to run a server application on the middle tier?

Answer:

See "Files and components for Borland MIDAS" at  
"<http://www.borland.com/midas/papers/components.html>" Files and Components Paper

4/2/99 11:06:03 AM

Last Modified: 01-SEP-99

## Article #15030: Alternate security databases

Question and Answer Database

FAQ30B.txt     Alternate security databases

Category     :Database (MSACCESS)

Platform     :Win95/NT

Product      :

Question:

Is it possible to use alternate security databases other than the one specified in the registry?

Answer:

With BDE 4.00, there is no setting for the security database; you must tuse the default database. With BDE 4.01, a new driver parameter "SYSTEM DATABASE" was added. It is located on the MSACCESS driver page in the BDE Administrator tool.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15032: Single and multiple application server instances

Question and Answer Database

FAQ32D.txt      Single and multiple application server instances  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

Question:

What is the difference between single and multiple instance application servers in a multi-tier application?

Answer:

This can best be thought of from the point of view of the application server and how many clients can it support. One multi-instance server can support multiple clients. A single instance server can only support one client. Multi-instance servers create multiple data modules in one instance of the server. Single instance servers are launched for each client.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15039: Other non-BDE file needed to access Microsoft Access databases

Question and Answer Database

FAQ39B.txt      Other non-BDE file needed to access Microsoft Access databases

Category        :Database (MSACCESS)

Platform        :Win95/NT

Product         :

Question:

What other software do I need to access Microsoft Access databases and tables?

Answer:

You must have Microsoft's DAO version 3.0 drivers installed. BDE 4.00 and 4.01 does not support DAO 3.5 (Access 97). If you upgrade Access 95 to Access 97, DAO 3.0 still resides on your machine, so the driver will function correctly. If you install Access 97 with no prior version of Access 95, the driver will not function.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15041: Locating the servers available in the registry

Question and Answer Database

FAQ41D.txt    Locating the servers available in the registry  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

How do I locate the application servers available in the registry of the machine?

Answer:

Read the registry key under HKEY\_CLASSES\_ROOT\CLSID\\* looking for keys that have a subkey "Borland DataBroker". These entries are application servers. Here's an example that loads application server names to a Listbox.

```
procedure TForm1.FormCreate(Sender: TObject);
var

uses Registry;
    I: integer;
    TempList: TStringList;
begin
    TempList := TStringList.Create;
    try
        with TRegistry.Create do
            try
                RootKey := HKEY_CLASSES_ROOT;
                if OpenKey('CLSID', False) then
                    GetKeyNames(TempList);
                CloseKey;
                for I := 1 to TempList.Count - 1 do
                    if KeyExists('CLSID\' + TempList[I]
                        + '\Borland DataBroker') then
                        begin
                            if OpenKey('CLSID\' + TempList[I] + '\ProgID', False)
                                then begin
                                    Listbox1.Items.Add(ReadString(''));
                                    CloseKey;
                                end;
                        end;
            end;
        end;
    end;
end;
```

```
        end;  
    end;  
finally  
    Free;  
end;  
finally  
    TempList.Free;  
end;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15048: MSACCESS driver and multi-threaded apps

Question and Answer Database

FAQ48B.txt MSACCESS driver and multi-threaded apps

Category :Database (MSACCESS)

Platform :Win95/NT

Product :

Question:

Can I use the MSACCESS driver to create multi-threaded applications?

Answer:

Yes and No. DAO 3.0 is not thread safe so all applications using DAO (including Borland's MSACCESS driver) must access the database through the main process' thread. The application can still use threads as long as the threads do not do any database operations with the MSACCESS driver.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15050: Logging in to the remote datamodule

Question and Answer Database

FAQ50D.txt    Logging in to the remote datamodule  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

How is the UserName and Password passed to the remote datamodule?

Answer:

On the RemoteDataModule, drop a TDatabase and add an automation procedure (use Edit | Add To Interface) for login. Make sure that the HandleShared property is True on the TDatabase.

```
procedure Login(UserName, Password: WideString);  
begin  
  { DB = TDatabase }  
  
  { Something unique between clients }  
  DB.DatabaseName := UserName + 'DB';  
  DB.Params.Values['USER NAME'] := UserName;  
  DB.Params.Values['PASSWORD'] := Password;  
  DB.Open;  
end;
```

After you have created the automation method on the RemoteDataModule (using Edit | Add To Interface), you can call that method using;

```
RemoteServer1.AppServer.Login('USERNAME', 'PASSWORD');
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15057: Current MSACCESS driver limitations

Question and Answer Database

FAQ57B.txt Current MSACCESS driver limitations

Category :Database (MSACCESS)

Platform :Win95/NT

Product :

Question:

What restrictions are there on the MSACCESS driver?

Answer:

BCD (binary coded decimal) support is not available.

Key violation tables cannot be created with DbAddIndex.

(All other DbAddIndex functionality is supported.)

QBE, heterogeneous queries, and queries using the local SQL engine are not supported.

In the SDK, DbDoRestructure and DbRegenIndex are not available.

Opening a table when TTable.Exclusive is set to True prevents the user from opening that table again, even in same session.

7/15/98 3:23:06 PM

Last Modified: 05-MAR-01

## Article #15059: Database login for multiple clients

Question and Answer Database

FAQ59D.txt Database login for multiple clients

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

### Question:

Since the middle tier is doing the database access on behalf of multiple thin clients, do we lose the ability to associate a particular database user ID with a particular human user? This would seem to be a problem if you want to grant different access rights to different groups of users at the database level.

### Answer:

Yes, if you want to use one connection to the database for all clients. If you want, you can have the clients login to the database. To do this, create an automation method on the RemoteDataModule that takes the user name and password and use that to login to the database. Also, drop a TSession on your RemoteDataModule and set it's AutoSessionName property to True.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15066: BLObs as SQL input parameters

Question and Answer Database

FAQ66B.txt      BLObs as SQL input parameters

Category        :Database (MSACCESS)

Platform        :Win95/NT

Product         :

Question:

Can BLObs be used as input parameters to queries and stored procedures?

Answer:

Yes. Before BDE 4.0, no BLOb could be used as an input parameter.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15068: Comparison of Sockets, DCOM, OLE Enterprise

Question and Answer Database

FAQ68D.txt Comparison of Sockets, DCOM, OLE Enterprise

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

What are some of the differences between sockets, DCOM, OLE Enterprise?

Answer:

	NT Security	No Client Setup	Callback Capability
Sockets		X	
DCOM	X		X
OLE Enterprise			X

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15075: files that need to be included with applications using MSACCESS

Question and Answer Database

FAQ75B.txt files that need to be included with applications using MSACCESS

Category :Database (MSACCESS)

Platform :Win95/NT

Product :

Question:

What other files need to be included with applications using the MSACCESS driver?

Answer:

Along with the BDE core DLLs, you need IDDAO32.DLL. This is BDE's MSACCESS driver file. You will also need to distribute DAO 3.0 on all machines using the MSACCESS driver. Before deploying an application, verify that you have distribution rights for DAO and the JET engine. The BDE MSACCESS driver can be freely distributed.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15077: Minimizing Connections

Question and Answer Database

FAQ77D.txt     Minimizing Connections  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

How can I demonstrate the use of fewer database connections than clients using our three-tier application? For example, I would like to show 10 clients using no more than 5 database connections.

Answer:

Actually, you can have all 10 clients use 1 database connection. Put a TDatabase on the RemoteDataModule and set HandleShared to True. The database will login once no matter how many clients are connected to your application.

**?Note: Be sure to check your licensing agreement with the database server vendor. The fact that it's possible doesn't unburden you of any licensing obligations you may have.?**

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15084: 'Cannot Load IDAPI Service Library' error

Question and Answer Database

FAQ84B.txt 'Cannot Load IDAPI Service Library' error  
Category :Database (MSACCESS)  
Platform :Win95/NT  
Product :

Question:

"Cannot Load IDAPI Service Library" error: What does this error mean and how can I resolve it?

Answer:

If Access 95 or Office 95 was not used to load the DAO 3.0 engine on the machine, the cause is a missing registry entry. To fix this, place the following entry in the registry:

key name: HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Shared Tools\DAO  
string: Path  
value: (path to your DAO3032.DLL)

Another cause of this error is if the registry is set to read only. The only workaround for this cause is to either make the registry read / write, or make that particular entry read / write.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15085: Accessing InterBase thru an AT service

Question and Answer Database

FAQ85C.txt      Accessing InterBase thru an AT service

Category      :Database Issues

Platform       :All

Product        :C++Builder 1.x

Question:

I have an app that, thru BDE, updates an InterBase table. From the command-line it runs fine. But if I try to run it thru an AT service I get:

```
Error connecting to db: Unknown user name or password.
unavailable database. Alias: myalias
```

How can I make this work?

Answer:

First, as you are probably already doing, you need to set the following properties on your Database object:

```
MyDatabase->AliasName = "myalias";
MyDatabase->LoginPrompt = false;
MyDatabase->Params->Add("USERNAME=SYSDBA");
MyDatabase->Params->Add("PASSWORD=masterkey");
```

Then, you will need to install BDE 4.0, the one that ships with Delphi 3.0.

And finally, you will need to set the AT service to be interactive, like:

```
AT 11:30 /INTERACTIVE c:\myexe
```

```
7/2/98 10:32:32 AM
```

Last Modified: 01-SEP-99

## Article #15086: Multiple Transports

Question and Answer Database

FAQ86D.txt      Multiple Transports  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

Question:

Can some of my clients use DCOM and others use Sockets?

Answer:

Yes.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15093: 'Translate error, value out of bounds' error

Question and Answer Database

FAQ93B.txt 'Translate error, value out of bounds' error  
Category :Database (FoxPro)  
Platform :Win95/NT  
Product :

Question:

"Translate error, value out of bounds" error with blank date fields: What causes this error and how do I get around the problem?

Answer:

Fox Pro stores one of three things in a date field:

An actual date (ex 19970727 is 7/27/1997)  
8 Spaces (ASCII 20)  
8 Zeros (ASCII 30)

All of these are acceptable to Fox Pro. The Fox Pro driver for the BDE only accepts 1 & 2 as valid dates and returns a translate error for the 3rd (8 ASCII zeros) as it doesn't consider this as a valid date. To fix the problem, restructure the date field to a char field and then restructure the table again changing the char field back to a date field. This translation writes back only valid values to the date field and fixes the problem.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15095: Moving the application server to another machine

Question and Answer Database

FAQ95D.txt     Moving the application server to another machine  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

### Question:

When server is moved to a different machine the client machine is not able to communicate or rather find the server. I know it needs to be registered as well with the server machine but how?

### Answer:

When you move the server to another machine you need to do 2 things.

- 1) Register the server.     SERVER.EXE /REGSERVER
- 2) Change the ComputerName property of the TRemoteServer to point to the name of the server.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15102: Paradox slow on some computers and not on others

Question and Answer Database

FAQ: FAQ102B - Paradox slow on some computers and not on others

Category: Database (Miscellaneous)

Platform: All

Product:

Question:

When accessing Paradox tables on a server machine that is different from the client, some machines are slow while others are fast. What is wrong with the slow machines?

Answer:

If you have McAfee VirusShield, make sure it is set to only check "Program Files Only" instead of "All Files".

7/28/99 12:36:28 PM

Last Modified: 01-SEP-99

## Article #15103: Using Date as a Field Name

Question and Answer Database

FAQ103C.txt Using Date as a Field Name

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

I have a field called DATE and when I do a Query on the table with the SQL Statement `SELECT DATE FROM "tablename.db"` I get an error "Invalid use of Keyword. Token: DATA line 1" What is the problem?

Answer:

Never name a Field "Date". Date is a Keyword in SQL and therefore SQL statements that use the word in the wrong way will throw this error.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #15104: Using Multiple Servers

Question and Answer Database

FAQ104D.txt Using Multiple Servers  
Category :DCOM/MultiTier/ActiveForm  
Platform :All  
Product :Delphi 3.x

### Question:

It appears that only one remote data module will work reliably in a project. Is this case?

### Answer:

No. You can have multiple Remote Data Modules in a project. The caveat is that multiple Remote Data Modules should not access one another because you can't count on the other Remote Data Modules being created or know what state they're in. Remote Data Modules are created on demand when a client requests one. You can use regular data modules to partition the application and access those from the RDM.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15111: Paradox table version level

Question and Answer Database

FAQ111B.txt Paradox table version level

Category :Database (Miscellaneous)

Platform :All Windows

Product :

Question:

When I create a Paradox Table in the Database Desktop and request a Paradox Version 7 table, the request is not honored. How can I work around this?

Answer:

The Database Desktop will create a table with the lowest version that will handle the tables needs and is declared in the BDE Administrator (IDAPI Configuration utility) under the Configuration>Drivers>Paradox>Level setting. To get a Version 7 table in the Database Desktop, simply use a version 7 feature. To permanently change the Paradox Table level for all tables made by the Database Desktop, go to the BDE Administrator and go to the Configuration>Drivers>Paradox>Level setting and change it to 7.

Example for changing the level of a Paradox Table in the Database Desktop:

Use the restructure option in Database Desktop to create or modify an index to use unique or descending attributes.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15113: Accessing Server Methods from Client

Question and Answer Database

FAQ113D.txt     Accessing Server Methods from Client

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

How do I access a method on the application server from  
TClientDataSet?

Answer:

RemoteServer.AppServer.MyMethod. AppServer is a read-only property that points to the interface returned to the remote server by an application server's provider component. Client applications can communicate directly with the application server through this interface.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15120: BDEAdmin:Invalid Filename for Oracle Server Name

Question and Answer Database

FAQ120B.txt    BDEAdmin:Invalid Filename for Oracle Server Name  
Category     :Database (Oracle)  
Platform     :All  
Product      :Applies to all

### Question:

I'm setting up my alias to an Oracle database.  
When I pick Server Name I get an error of invalid filename.  
How come?

### Answer:

If your protocol is set to TNS, this error will occur if  
HKEY\_LOCAL\_MACHINESOFTWAREORACLEORACLE\_HOME is missing,  
points to an invalid directory, or the TNSNAMES.ORA file cannot  
be found within where ORACLE\_HOME points.

8/18/98 11:11:27 AM

Last Modified: 05-OCT-99

## Article #15122: Server Termination

Question and Answer Database

FAQ122D.txt     Server Termination  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product     :Delphi 3.x

Question:

Does an application server auto-terminate when all connections to it are dropped?

Answer:

The application keeps track of how it was started. If an automation call started it, it will shut down when all objects have been freed, otherwise it won't...

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15131: Un-registering a server

Question and Answer Database

FAQ131D.txt    Un-registering a server  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

How do you get rid of a "registered" ServerName if I don't want to use it any longer?

Answer:

Run the executable with the /UNREGSERVER flag:

```
MYSERVER.EXE /UNREGSERVER
```

This is the standard way of unregistering a self-registering OLE automation server.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15138: BDE network installation

Question and Answer Database

FAQ138B.txt - BDE network installation

Category :Install Issues

Platform :Win95/NT

Product :

Question:

I want to put the entire BDE DLLs and configuration files on our LAN. Is this possible and is there an easy way to setup the client machines? I don't know what the registry settings are for the BDE.

Answer:

[www.borland.com/devsupport/bde/files/bdenet.zip](http://www.borland.com/devsupport/bde/files/bdenet.zip) has a BDE network installation example that allows a user to setup the BDE and SQL Links on a client machine without having to install the BDE and SQL Links to the client. Registry entries are placed into the client machines so that the client can access a network version of the BDE. Two programs are included: one which helps configure the .INI file that is necessary to create the registry entries, and another program that shows how the methods work to create the registry entries.

This is not officially supported by Borland. It was developed by Developer Support to help in BDE network installations.

Last Modified: 11-FEB-00

## Article #15140: Dataset event handlers not fired

Question and Answer Database

FAQ140D.txt Dataset event handlers not fired

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

When I ApplyUpdates on the ClientDataSet, the OnNewRecord for the Server dataset code is not fired. How can I make this work?

Answer:

Well, you can't really. The updates go through BDE and do not go through the dataset.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15147: BDE configuration utility

Question and Answer Database

FAQ147B.txt BDE configuration utility

Category :BDE Features

Platform :Win95/NT

Product :

Question:

Is there a new BDE configuration Utility?

Answer:

Yes. The BDE Configuration Utility (BDECFG32.EXE) has been updated and is now the BDE Administrator (BDEADMIN.EXE). The administrator now shares the same "look and feel" of Delphi's database explorer and should be used in place of the BDE Configuration Utility (BDECFG32.EXE).

4/24/98 11:13:37 AM

Last Modified: 01-SEP-99



## Article #15149: Dataset event handlers

Question and Answer Database

FAQ149D.txt     Dataset event handlers  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

How do I handle business rules that are tied to dataset event handlers (OnBeforePost, etc.) on the server?

Answer:

Install and use the TUpdateSQLProvider component, found in the \demos\Usqlprov directory). It has an OnUpdateRecord event that fires for every record getting updated.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15156: FoxPro support: capabilities

Question and Answer Database

FAQ156B.txt FoxPro support: capabilities

Category :Database (FoxPro)

Platform :Win95/NT

Product :

Question:

What are the capabilities of the new FoxPro driver?

Answer:

The dBASE (STANDARD) driver now includes support for FoxPro version 2.0, 2.5, and 2.6 compressed index (.CDX) and BLOB (.FPT) files, letting you open and create FoxPro 2.5 tables.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15158: DBCLIENT.DLL

Question and Answer Database

FAQ158D.txt      DBCLIENT.DLL  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

Question:  
What is the role of DBClient.DLL?

Answer:

DBCLIENT.DLL contains the low-level implementation of TClientDataSet. It manages all of the data and implements functionality like filters, sorting, etc. The relationship between TClientDataSet and DBCLIENT.DLL is very similar to the one between TTable and the BDE. The functionality is surfaced through two COM interfaces (DSBase & DSCursor) and requires OLE registration. If DBCLIENT.DLL is placed in any directory on the path or in the EXE directory on the client machine, it will automatically be registered by the VCL code in TClientDataSet. Otherwise it must be registered using REGSVR32.EXE or TREGSVR.EXE.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15165: Microsoft Access; capabilities

Question and Answer Database

FAQ165B.txt Microsoft Access; capabilities

Category :Database (MSACCESS)

Platform :Win95/NT

Product :

Question:

What are the capabilities of the new Access driver?

Answer:

If you have a version of the Microsoft JET engine (included with Microsoft Access and FoxPro) installed on your system you can now use the BDE to open or create Microsoft Access tables using the MSACCESS driver.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15167: Modifying application server SQL from the client.

Question and Answer Database

FAQ167D.txt - Modifying application server SQL from the client.

Category :DCOM/MultiTier/ActiveForm  
Platform :All-32Bit  
Product :

Question:

Is it possible to modify the TQuery.SQL property when it resides in a remote data module?

Answer:

Yes. There are three different avenues to accomplish this:

1. If it is a parameterized query, you can use the IProvider.SetParams call. Assuming you have 2 parameters (Param1 and Param2). The following code fragment demonstrates matching by index, Value1 goes to the first parameter, etc...

```
ClientDataSet1.Provider.SetParams(VarArrayOf(Value1, Value2));
```

Or matching by param name:

```
var  
  V: Variant;  
  ParamCount: Integer; //used for ease of reading  
begin  
  ClientDataSet1.Close;  
  ParamCount := 2;  
  V := VarArrayCreate([0, ParamCount - 1], varVariant);  
  V[0] := VarArrayOf(['Param1',Value1]);  
  V[1] := VarArrayOf(['Param2',Value2]);  
  ClientDataSet1.Provider.SetParams(V);  
  ClientDataSet1.Open;  
end;
```

For more example code see \demos\midas\setparam.

2. If you want to change the SQL, then you can use the `IProvider.DataRequest` method.

On the Client;

```
CDS.Data := CDS.Provider.DataRequest('select * from customer');
```

On the server, you must use a `TProvider` object, and assign the `OnDataRequest` event;

```
function TForm1.Provider1DataRequest(Sender: TObject;  
  Input: OleVariant): OleVariant;  
begin  
  //assumes DataSet is a TQuery.  
  Provider1.DataSet.Close;  
  TQuery(Provider1.DataSet).SQL.Text := Input;  
  Provider1.DataSet.Open;  
  Result := Provider1.Data;  
end;
```

For more example code see `\demos\midas\adhoc`.

3. Create you own interface function that executes the SQL statement.

In the `RemoteDataModule`:

a. Edit | Add To Interface "procedure `ExecSQL(SQL: WideString);`" This will add a procedure to your interface.

b. On the server, code the new procedure:

```
procedure RemoteData.ExecSQL(SQL: WideString);  
begin  
{ Query1 is kept in the RemoteDatamodule for this purpose}  
  Query1.SQL.Text := SQL;  
  Query1.ExecSQL;
```

end;

c. On the client, call the new procedure:

```
RemoteServer1.AppServer.ExecSQL('Delete from deal details  
  where dealnumber=1');
```

4/2/99 11:37:54 AM

Last Modified: 01-SEP-99

## Article #15174: SQL Parameter binding

Question and Answer Database

FAQ174B.txt SQL Parameter binding

Category :BDE Features

Platform :All

Product :BDE 4.0

Question:

What type of parameter binding is supported?

Answer:

USERHTML

BDE now supports parameter binding for BLObs and strings longer than 255 characters except for the MSACCESS driver.

4/24/98 11:13:57 AM

Last Modified: 01-SEP-99



## Article #15176: Displaying a bitmap on the client area of an MDI parent form.

Question and Answer Database

FAQ176D.txt - Displaying a bitmap on the client area of an MDI parent form.

Category :Windows API  
Platform :All-32Bit  
Product :

Question:

How do I display a bitmap to the client area of an MDI parent form?

Answer:

Here are the necessary steps to add wallpaper to a MDI parent form:

Create a new project  
Set the form's FormStyle to fsMDIForm  
Drop an image on the form and select a bitmap into it.  
Find the { Private Declarations } comment in the form's definition and add these lines right after it:

```
FClientInstance : TFarProc;  
FPrevClientProc : TFarProc;  
procedure ClientWndProc(var Message: TMessage);
```

Find the "implementation" line and the {\$R \*.DFM} line that follows it. After that line, enter this code:

```
procedure TMainForm.ClientWndProc(var Message: TMessage);  
var  
    Dc : hDC;  
    Row : Integer;  
    Col : Integer;  
begin  
    with Message do  
        case Msg of
```

```

WM_ERASEBKGDND:
begin
  Dc := TWMEraseBkGnd(Message).Dc;
  for Row := 0 to ClientHeight div Image1.Picture.Height do
    for Col := 0 to ClientWidth div Image1.Picture.Width do
      BitBlt(Dc,
        Col * Image1.Picture.Width,
        Row * Image1.Picture.Height,
        Image1.Picture.Width,
        Image1.Picture.Height,
        Image1.Picture.Bitmap.Canvas.Handle,
        0,
        0,
        SRCCOPY);
      Result := 1;
    end;
  else
    Result := CallWindowProc(FPrevClientProc,
      ClientHandle,
      Msg,
      wParam,
      lParam);
  end;
end;

```

In the OnCreate method for the form, type the following lines of code:

```

FClientInstance := MakeObjectInstance(ClientWndProc);
FPrevClientProc := Pointer(GetWindowLong(ClientHandle,
  GWL_WNDPROC));
SetWindowLong(ClientHandle,
  GWL_WNDPROC, LongInt(FClientInstance));

```

Add a new form to your project and set its FormStyle property to fsMDIChild.

Now you have a working MDI project with "wallpaper" where the image bitmap is tiled to cover the MDI form's client area.

4/2/99 11:38:30 AM

Last Modified: 01-SEP-99

## Article #15183: New functions

Question and Answer Database

FAQ183B.txt      New functions

Category        :BDE Features

Platform        :Win95/NT

Product         :

Question:

Are there any new BDE API functions?

Answer:

Yes. DbAddDriver and DbDeleteDriver let you add and remove ODBC drivers from the current BDE configuration or session. The BDE now automatically adds ODBC drivers and data sources as BDE aliases to the active session when they aren't currently stored in the configuration file. Also, please look at the [www.borland.com/devsupport/bde/bdeapiex/index.html](http://www.borland.com/devsupport/bde/bdeapiex/index.html) page for examples not found in the BDE32 help file.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15185: ServerName property (TMidasConnection or TRemoteServer)

Question and Answer Database

FAQ185D.txt ServerName property (TMidasConnection or TRemoteServer)

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

I get the impression that in order to register the server interfaces with the client I have to execute the server at each client. Is this true?

Answer:

No. On the client, fill in the ComputerName and ServerGUID properties of the TRemoteServer. This allows the GUID to be located in the remote computer's registry in the CLSID section. The GUID will have a ProgID section containing the server name that is used to populate the ServerName property of TRemoteServer drop down list.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15192: ODBC Support

Question and Answer Database

FAQ192B.txt ODBC Support

Category :BDE Features

Platform :Win95/NT

Product :

Question:

What version of ODBC is supported by the BDE?

Answer:

Versions 3.x and earlier are now supported by BDE 4.0.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15202: Sorting on Calculated fields

Question and Answer Database

FAQ202D.txt     Sorting on Calculated fields

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product     :Delphi 3.x

Question:

How do you make TClientDataSet sort on lookup or calculated fields?

Answer:

In the fields editor, when creating a new field to be sorted on, choose New Field from the context menu and choose InternalCalc as the field type. Or if creating TFields in code, make the TField.FieldKind fkInternalCalc instead of fkCalc. fkInternalCalc is only available in ClientDataSets.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15211: TClientDataSet, auto-increment fields

Question and Answer Database

FAQ211D.txt TClientDataSet, auto-increment fields

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

We want to add several records that contain auto-increment fields to the ClientDataSet and then apply it to the remote datamodule. The key field is an auto-increment field. How is this done?

Answer:

You should be able to just leave this field blank on the client application and it will be initialized when you call ApplyUpdates to insert the record. If you need to see the generated value back on the client, call Refresh.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15218: MS SQL error '17824 Unable to write to ListenOn connection ...

Question and Answer Database

FAQ218B.txt MS SQL error '17824 Unable to write to ListenOn connection ...

Category :Database (MSSQL)

Platform :Win95/NT

Product :

Question:

What would cause error:

MS SQL Server error "17824 Unable to write to ListenOn connection 'xxx', loginname 'xxx', hostname 'xxx'."

Answer:

PROBLEM:

After establishing a number of concurrent application connections to the MS SQL Server the following error is encountered:

MS SQL Server error "17824 Unable to write to ListenOn connection 'xxx', loginname 'xxx', hostname 'xxx'."

POSSIBLE CAUSE:

It is possible that the PROCEDURE CACHE percentage is set too low. Please see your MS SQL Server documentation and NT Event Viewer for more information on debugging network errors and tuning your MS SQL Server's PROCEDURE CACHE.

Relevant SQL Server options:

MAX PROCEDURE CACHE ...

MAX WORKER THREADS

PROCEDURE CACHE

The procedure cache holds entries for each procedure and other maintenance overhead.

Settings for the PROCEDURE CACHE include:

MAX PROCEDURE CACHE ACTIVE %

MAX PROCEDURE CACHE USED%



PROCEDURE CACHE USED%

MAX WORKER THREADS

The max worker threads option allows control over the number of threads allocated to the user pool.

7/15/98 3:23:06 PM

Last Modified: 06-OCT-99

## Article #15220: TClientDataSet.Data property

Question and Answer Database

FAQ220D.txt TClientDataSet.Data property

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

How is the Data property used typically in an application?

Answer:

Typically, it won't be used. It is how you can get the packet as a variant array of bytes. You can also assign to it if you receive a data packet from the server through some method outside the IProvider interface.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15227: Performance with two or more queries

Question and Answer Database

FAQ227B.txt Performance with two or more queries

Category :Database (MSSQL)

Platform :All

Product :Applies to all

Question:

Using MSSQL, or Sybase for that matter, why does a second simple select query run more slowly than a preceding query.

Answer:

MS SQL has a "pending result sets" behavior where the 1st query will only retrieve a small set of records (usually enough to fill a grid or any other data aware controls), and there will be a pending result set. The second query causes that pending result set to be retrieved. This can be tested by closing the first query before running the second or connecting each query to a TDatabase. The BDE uses a "special" connection type that is restricted to one per database connection. When a second query is run within the same database connection, the BDE must fetch and cache the previous query's records before starting the next query.

Suggestions:

Close the first query before opening the second

Use multiple databases, one per query

Use a more restrictive where clause to limit the number of records

Use a more restrictive domain result to retrieve only the fields you need

7/15/98 3:23:06 PM

Last Modified: 06-OCT-99

## Article #15236: 'Memo too large' error with MSSQL

Question and Answer Database

FAQ236B.txt 'Memo too large' error  
Category :Database (MSSQL)  
Platform :Win95/NT  
Product :

### Question:

Using BLOB I/O an error, 'Memo Too Large' is returned. The error is: General SQL Error: 7134, unmapped [text: text table and table reference by the text pointer disagree.]. This is not a problem if the table does not contain a unique index. In this case all BLOB information is retrieved through the execution of the query. What does this error mean and how can I resolve it?

### Answer:

A fix to MSSQL server is available from Microsoft:

<ftp://ftp.microsoft.com/bussys/sql/transfer/sql265i.exe> - Intel Patch  
or

<ftp://ftp.microsoft.com/bussys/sql/transfer/sql265a.exe> - Alpha Patch

The password to use when extracting the files is "6.50.265" (without the quotation marks). To extract with the password, the syntax is:  
`sql265i.exe -s6.50.265`

7/15/98 3:23:06 PM

Last Modified: 06-OCT-99

## Article #15238: TCP/IP: TMidasConnection, Socket Server

Question and Answer Database

FAQ238D.txt TCP/IP: TMidasConnection, Socket Server

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

How does the socket server (\bin\Scktsrvr.exe) interact with TMidasConnection?

Answer:

The socket server is something like a service that replaces DCOM, but without the security. When a client application sets Connected = True on a TMidasConnection, the socket server on the server machine creates the remote data module. Then it takes the IDispatch invoke and GetIDsOfNames methods and marshals them through the socket connection, using TCP/IP.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15246: TCP/IP: DCOM Dependency?

Question and Answer Database

FAQ246D.txt TCP/IP: DCOM Dependency?

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

To use TMidasConnection, do I need DCOM installed?

Answer:

No. TMidasConnection does not rely on DCOM.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

**Article #15253: 'Vendor initialization failure: ORANT71.DLL' error.**

Question and Answer Database

FAQ253B.txt 'Vendor initialization failure: ORANT71.DLL' error

Category :Database (Oracle)

Platform :Windows NT

Product : BDE versions prior to 5.0

Question:

When attempting a connection to Oracle using the BDE Oracle SQL Links driver, why am I getting the error, Vendor initialization failure: ORANT71.DLL, when the driver's Vendor Init parameter is set to ORA73.DLL [or ORA72.DLL]?

Answer:

The BDE always searches the path first for the DLL specified in the Vendor Init parameter and if that fails, the BDE searches, by default, for ORANT71.DLL. Thus the Vendor Init error means that the BDE failed to find and load the DLL you specified and ORANT71.DLL.

Last Modified: 23-FEB-00

## Article #15255: TCP/IP: Load balancing, OLE Enterprise

Question and Answer Database

FAQ255D.txt TCP/IP: Load balancing, OLE Enterprise

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

If I use TMidasConnection with the sockets (TCP/IP) transport, can I still take advantage of OLE Enterprise load balancing and fail over? Would I then have to install OLE Enterprise on the client machines?

Answer:

You can take advantage of OLE Enterprise without installing it on the clients machines. Just set up the server (the machine running SCKTSRVR.EXE) as a OLE Enterprise client. That way, when a request comes in from a real client, SCKTSRVR will attempt to create the object, and OLE Enterprise will intercept and provide the service.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15262: 'External Exception C0000008' or 'Access Violation'

Question and Answer Database

FAQ262B.txt 'External Exception C0000008' or 'Access Violation'  
Category :Database (Oracle)  
Platform :All  
Product :Applies to all

Question:

"External Exception C0000008" or "Access Violation" error with 2.3.3 SQL NET client: I get an Access Violation with Delphi 3 and an External Exception error with Delphi 2 when connecting to an Oracle server when in Delphi's IDE or Turbo Debugger with NT4. If I run the application alone, it executes correctly. How do I get around the problem?

Answer:

Borland has determined that this is a problem with SQL NET 2.3.3. Oracle also recognizes this as a SQL NET problem. There are a couple of workarounds:

Rename or remove the registry entry

HKEY\_LOCAL\_MACHINE\SOFTWARE\ORACLE\OTRACE73

This is the solution that Oracle support recommends!  
Start Oracle's Installer and remove the "Oracle Trace collection Servces 7.3.3.x.x"

Turn off Integrated Debugging in Delphi's IDE

Use an Oracle SQL NET that is older than 2.3.3

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15264: TCP/IP: Retrieving Client IP and Host Name

Question and Answer Database

FAQ264D.txt - TCP/IP: Retrieving Client IP and Host Name

Category :DCOM/MultiTier/ActiveForm  
Platform :All-32Bit  
Product :

Question:

At client, how can I get the IP address or host name of itself.

Answer:

The following example requires using the Winsock API encapsulated in WinSock.pas.

```
uses
  WinSock;

procedure GetHostInfo(var Name, Address: string);
var
  WSADATA: TWSADATA;
  HostEnt: PHostEnt;
begin
  { no error checking... }
  WSASStartup(2, WSADATA);
  SetLength(Name, 255);
  Gethostname(PChar(Name), 255);
  SetLength(Name, StrLen(PChar(Name)));
  HostEnt := gethostbyname(PChar(Name));
  with HostEnt^ do
    Address := Format('%d.%d.%d.%d', [
      Byte(h_addr^[0]),
      Byte(h_addr^[1]),
      Byte(h_addr^[2]),
      Byte(h_addr^[3])]);
  WSACleanup;
end;
```

4/2/99 11:39:52 AM

Last Modified: 01-SEP-99

**Article #15271: error: SYB-12403 ctsend(): user api layer: external error**

Question and Answer Database

FAQ271B.txt error: SYB-12403 ctsend(): user api layer: external error

Category :Database (Sybase)

Platform :Win95/NT

Product :

Question:

What would cause error "SYB-12403 ctsend(): user api layer: external error"

Answer:

This routine is Fixed in BDE/SQL Links 4.0

ctsend(): user api layer: external error: this routine cannot be called because another command structure has results pending

7/15/98 3:23:06 PM

Last Modified: 06-OCT-99

## Article #15273: Error: "Application is not licensed to run this feature"

Question and Answer Database

FAQ273D.txt Error: "Application is not licensed to run this feature"

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

"Application is not licensed to run this feature" error when running demo application from \demos\db\clientds.

Answer:

This error occurs when attempting to use Delphi 3 Client/Server features from Delphi 3 Professional or Standard editions. This could happen if you compiled the demos using a compiler other than the one that comes with Delphi 3 C/S. Check that the Delphi 3 Professional was not installed previously without first being un-installed. To correct this situation, uninstall Delphi, double check for leftover Delphi files, and finally re-install Delphi 3.0 C/S.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15280: 'ct\_cursor(CLOSE); user api layer: external error

Question and Answer Database

FAQ280B.txt     'ct\_cursor(CLOSE); user api layer: external error  
Category       :Database (Sybase)  
Platform       :Win95/NT  
Product        :

Question:

SELECT INTO statment returning error "ct\_cursor(CLOSE); user api layer: external error: A cursor must be opened before this command type can be intialized." What is wrong and how do I get around the problem?

Answer:

The Sybase CTLIB client will always parse a passthrough sql statement weather or not we (Delphi, BDE, SQL Links, etc.) issue an 'open' or 'exec'. If a SELECT is found to be the **FIRST** word, it (Sybase CTLIB) will try to create a cursor hence the error message. There is a workaround. Place a comment in front of the SQL text like so:

CHANGE

```
select * into #temptable from oldtable
```

TO

```
/**/select * into #temptable from oldtable
```

7/15/98 3:23:06 PM

Last Modified: 06-OCT-99

## Article #15282: Error: 'Interface not supported'

Question and Answer Database

FAQ282D.txt    Error:    'Interface not supported'

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

Why do I receive an "Interface is not supported" error when trying to use an interface?

Answer:

Verify that STDVCL32.DLL and your type library are registered. If casting an interface, you must be using DCOM and the type library must be registered on the client.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15289: 'External Exception C0000008' error on application exit

FAQ289B.txt 'External Exception C0000008' error on application exit

Category :Database (Sybase)

Platform :Windows NT

Product :Delphi

### Question:

I get an External Exception error in Delphi when connecting to Sybase server when in Delphi's IDE or Turbo Debugger with NT4 running the 10.0.4 Sybase client. If I run the application alone, it executes correctly. How do I get around the problem?

### Answer:

In NT4.0 anything that is being debugged with any debugger will cause the OS to report exceptions that would otherwise be ignored. "External Exception C0000008" one of those types of errors. Changing the Sybase client version will solve the problem. You can move back to the 10.0.3 client.

7/15/98 3:23:06 PM

Last Modified: 03-NOV-99



## Article #15291: Error: "Could not find IDataBroker in type library"

Question and Answer Database

FAQ291D.txt     Error: "Could not find IDataBroker in type library"

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

"Could not find IDataBroker in type library" error. How do I resolve this error?

Answer:

Stdvcl32.dll may have gotten deleted from your Window's system directory. If you need to, copy it from your CD and register it using TRegSvr.exe (found in the Delphi bin directory).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15299: Error: "Cannot load IDAPI service library"

Question and Answer Database

FAQ299D.txt      Error: "Cannot load IDAPI service library"

Category        :DCOM/MultiTier/ActiveForm

Platform        :All

Product         :Delphi 3.x

Question:

I receive a "Cannot load Idapi service library" error.

How do I resolve this?

Answer:

This general IDAPI startup error can occur for a number of reasons. In the context of a multi-tier application, you should be sure that STDVCL32.DLL and DBCLIENT.DLL are registered on the server. Also, make sure that you have IDPROV32.DLL in the BDE directory.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15305: Hetergenious joins

Question and Answer Database

FAQ305B.txt Hetergenious joins  
Category :Database (Miscellaneous)  
Platform :All  
Product :Applies to all

Question:

Hetergenious joins: How do I join files from differing alias (heterogeneous joins) using a SQL statment?

Answer:

For each server table, a connection must exist before the query is executed. A TDatabase can be used to establish the connection.

The SQL statement must use real alias names, as set up in the BDE configuration utility, not DatabaseName properties from a TDatabase.

The SQL syntax should be formatted as follows: a double quote, a colon, a real alias name, a colon, the table name, and a double quote. See example below.

Leave the DatabaseName property of the TQuery blank, unless the alias is local. For example, specifying DBDEMOS in the Query's DatabaseName property will work, but using IBLOCAL will cause a general SQL error.

Example:

```
SELECT A.ITEMNO, B.VENDORNO  
FROM ":DBDEMOS:ITEMS" A, ":IBLOCAL:VENDORS" B  
WHERE A.VENDORNO = B.VENDORNO
```

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15307: Error: "DAX Error - Name not unique in this context"

Question and Answer Database

FAQ307D.txt    Error:    "DAX Error - Name not unique in this context"  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

I receive error: "DAX Error - Name not unique in this context".  
How do I resolve this?

Answer:

Set the HandleShared property on your TDatabase component to  
True.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15313: File Extensions for dBASE and Paradox tables

Question and Answer Database

FAQ313B.txt    File Extensions  
Category     :Database (Miscellaneous)  
Platform     :All  
Product      :Applies to all

Question:

What are the file extensions for dBASE and Paradox tables?

Answer:

File Extensions for Paradox Tables.?

.DB        Paradox table?  
.FAM      listing of related files (like a table's .TV file)?  
.LCK      Lock file?  
.MB       Blob file?  
.PX       Primary index?  
.TV       Table view settings (not used by BDE)?  
.VAL      Validity checks and referential integrity?  
.Xnn      Secondary single-field index, numbered?  
.Ynn      Secondary single-field index, numbered?  
.XGn      Composite secondary index ?  
.YGn      Composite secondary index ?

File Extensions for dBASE Tables?

.DBF      dBASE table  
.DBT      Memos for a dBASE table  
.MDX      Maintained index of a dBASE table  
.NDX      Non-maintained index of a dBASE table  
.TVF      Table view settings for a dBASE table

4/24/98 12:59:04 PM

Last Modified: 05-OCT-99

## Article #15315: Error: "DAX Error class not registered."

Question and Answer Database

FAQ315D.txt     Error: "DAX Error class not registered."

Category        :DCOM/MultiTier/ActiveForm

Platform        :All

Product         :Delphi 3.x

Question:

The client machine displays error "DAX Error class not registered."

Answer:

You need to have DBCLIENT.DLL on the client, and it needs to be registered.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15321: Unsupported Expression Indexes

Question and Answer Database

FAQ321B.txt      Unsupported Expression Indexes

Category        :Database (FoxPro)

Platform        :Win95/NT

Product         :

### Question:

What types of FoxPro indexes are not supported by the BDE?  
When trying to open some tables, I get an "Invalid Index Descriptor" error.

### Answer:

This error occurs when the production index (.CDX) associated the table has an index tag which has an expression which the BDE cannot evaluate. The solution is to delete the tag using FoxPro create an equivalent index that the BDE can understand.

The following conditions are not supported by the BDE and will cause "Invalid Index Descriptor" error.

DTOC(, 1) format not supported; Use DTOC().

ALLTRIM function not supported; Use LTRIM(RTRIM(Field)).

5/15/98 1:37:20 PM

Last Modified: 01-SEP-99



## Article #15323: Error: "EOleSysError : Error loading type library/DLL"

Question and Answer Database

FAQ323D.txt      Error: "EOleSysError : Error loading type library/DLL"  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

### Question:

The client machine displays error "EOleSysError: Error loading type library/DLL". DBCLIENT.DLL and IDPROV32.DLL files are present in the BDE directory of Delphi.

### Answer:

Make sure that STDVCL32.DLL is on the machine in the WINDOWS\SYSTEM directory and registered.

```
REGSVR32 STDVCL32.DLL
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15329: 'Insufficient memory for this operation' error

Question and Answer Database

FAQ329B.txt 'Insufficient memory for this operation' error

Category :Database (Miscellaneous)

Platform :All

Product :Applies to all

### Question:

Whenever I attempt to open many query or table cursors, I finally reach a point when I get an error, "Insufficient memory for this operation". What do I do?

### Answer:

First, you should read the technical document at

[www.borland.com/devsupport/bde/ti\\_list/TI2751.html](http://www.borland.com/devsupport/bde/ti_list/TI2751.html)

Some current internal limits of BDE. This document will tell you what most of absolute BDE limits are.

IMPORTANT: If your application is not reaching one of the limits covered there and you are also getting the above error message, most likely the SHAREDMEMSIZE setting in the BDE will solve your problem. This setting will allow you to increase the BDE's internal object memory pool. A setting between 4096 and 8192 should be more than plenty.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15331: Error: 'Missing Data Provider or Data Packet'

Question and Answer Database

FAQ331D.txt     Error: 'Missing Data Provider or Data Packet'  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

Question:

Why does the error "Missing Data Provider or Data Packet" occur?

Answer:

The error occurs when the ClientDataSet can't resolve to a Provider reference on the application server or if the Provider doesn't return records. Check that the TClientDataSet ProviderName is filled in.

Background info: The error message originates in DBConsts.pas as a constant SNoDataProvider which in turn shows up in DBClient.pas (the unit that supports TClientDataSet). The error is triggered in two situations:

- 1) When the TClientDataSet.Provider can't obtain a reference to a provider through the RemoteServer component.
- 2) When the Provider doesn't return records to the TClientDataSet (during TClientDataSet.OpenCursor).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15337: Sybase SQL Anywhere update mode

Question and Answer Database

FAQ: FAQ337B - Sybase SQL Anywhere update mode

Category: Database (ODBC)

Platform: All-32Bit

Product:

Question:

I am using Sybase SQL Anywhere version 5.5 and I want to be able to set the UpdateMode for my dataset (Query or Table) to be upWhereKeyOnly. When I watch SQL Monitor to see the update logic used, it appears to act like UpdateMode is set to upWhereAll. Why can't I update by key only?

Answer:

This problem has to do with the fact that primary key indexes are not surfaced by SQL Anywhere in the BDE. If you run the stored procedure sp\_statistics with your table name as a parameter(ex. sp\_statistics customer), it should return all indexes on the table specified. Doing this you will see that the index on the primary key doesn't surface a index name.

The way to correct this is by configuring your DSN for SQL Anywhere to allow keys to be seen. This is done by checking the check box titled Microsoft Applications(Keys in SQL statistics) in the Additional Connection Options Section of the ODBC Configuration dialog for SQL Anywhere. After setting this, when you connect to your SQL anywhere database you will be able to see primary keys and UpdateMode will work as expected.

7/13/99 11:59:03 AM

Last Modified: 01-SEP-99

## Article #15339: Error: 'Name Conflicting'

Question and Answer Database

FAQ339D.txt    Error: 'Name Conflicting'  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

My DCOM Server is generated as Multiple Instance however, when the second client tries to connect the following error "Name Conflicting" appears.

Answer:

You need to set the TDatabase.HandleShared property to True.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15344: Error connecting to an Oracle 7.x server using the Oracle 8 client and BDE 5.

Question and Answer Database

FAQ344B.txt Error connecting to an Oracle 7.x server using the Oracle 8 client and BDE 5.

Category :Database (Oracle)

Platform :All

Product :BDE 4.0

### Question:

I am trying to use BDE 5.0 and the Oracle 8 client software to connect to my Oracle 7.x server. I created a new alias and when I attempt to retrieve data using it, I am getting the error ORA-12663 "Services requested by client not available on server." All of the Oracle supplied tools connect and let me see data. What is wrong?

### Answer:

First, be sure you are using at least v8.0.4 of the Oracle client. Next be sure that in your Oracle alias for your 7.x server, you set the 'Object Mode' setting to false. When you create new Oracle aliases in BDE administrator with BDE v5.0, the Object mode parameter is set to true regardless of what DLL32 (SQLORA8.dll or SQLORA32.dll) you have selected. For aliases that will reference Oracle 7.x servers, Object Mode must be set to false for correct operation.

3/26/99

Last Modified: 01-SEP-99

## Article #15346: Error: 'RPC Server is unavailable'

Question and Answer Database

FAQ346D.txt     Error: 'RPC Server is unavailable'

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

The client machine displays error "RPC Server is unavailable".

Answer:

The error is usually caused because the server can't be located. Make sure you can ping the server machine using the string you typed into the ComputerName property of the TRemoteServer. You might also want to try using an IP address instead in case there is some problem with your DNS configuration or the hosts file. Try setting the "Default Authentication Level" to (Run DCOMCNFG on the server to do this.) Lastly, check the Microsoft Developer's Network on the Web for the latest information on this Microsoft transport. The MSDN can be located at the time of the writing at [www.microsoft.com/msdn](http://www.microsoft.com/msdn).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15351: ALTER TABLE statment on Paradox tables removes passwords

Question and Answer Database

FAQ351B.txt ALTER TABLE statment on Paradox tables removes passwords

Category :Database (Miscellaneous)

Platform :All

Product :BDE 4.0

Question:

When adding or removing fields with the ALTER TABLE SQL statment, all passwords on the table are lost. What can I do about it?

Answer:

Get a version of the BDE the is greater then BDE 4.00.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99



## Article #15353: Error: 'Trying to append data to a non-partial dataset'

Question and Answer Database

FAQ353D.txt    Error: 'Trying to append data to a non-partial dataset'

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

I'm getting the error "Trying to append data to a non-partial dataset". What condition will cause this error ?

Answer:

This is usually caused by a call to AppendData when the TClientDataSet thinks it has all the data.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15359: 'Interface mismatch. Engine version different'

Question and Answer Database

FAQ359B.txt 'Interface mismatch. Engine version different'  
Category :Database (Miscellaneous)  
Platform :All  
Product :Applies to all

Question:

"Interface mismatch. Engine version different" error:  
This error occurs when I start my Delphi 3 application.  
What is wrong?

Answer:

All Delphi 3 database applications require that BDE 4.0 or greater is on the system. If you distribute this type of application, you must also distribute BDE version 4.0 or greater.

7/15/98 3:23:06 PM

Last Modified: 01-SEP-99

## Article #15366: Access violation when calling a non-delphi VCL created DLL

Question and Answer Database

FAQ366B.txt    Access violation when calling a non-delphi VCL created DLL

Category     :Database (Miscellaneous)  
Platform     :All  
Product      :BDE 4.0

Question:

Access violation in Delphi IDE when calling a C/C++/Delphi BDE API DLL: How do I resolve this access violation?

Answer:

There is a BDE API function called DbidllExit which can be prototyped the same as DbExit. Place the prototype in your idapi.h (C / C++) and call DbidllExit just before DbExit. For Delphi, simply call DbidllExit before DbExit.

7/15/98 3:23:06 PM

Last Modified: 06-OCT-99

## Article #15372: 'Insufficient memory to complete operation' error

Question and Answer Database

FAQ372B.txt 'Insufficient memory to complete operation' error  
Category :Database (Miscellaneous)  
Platform :All  
Product :BDE

### Question:

"Insufficient memory to complete operation" error when attempting various BDE / Database operations: What is wrong?

### Answer:

More than likely, you have hit one of the BDE fixed, but configurable, memory limitations. Use the BDE Administrator (BDEADMIN.EXE) to increase MEMSIZE and SHAREDMEMSIZE settings. You might also want to re-evaluate the application hitting this limitation because it is likely that the application is a serious resource hit on the machine. Not using auto-create forms and only opening tables and queries when needed can free up resources.

7/15/98 3:23:06 PM

Last Modified: 19-OCT-99

## Article #15374: Error: 'Windows Socket Error: (10060), on API 'connect?'

Question and Answer Database

FAQ374D.txt     Error: 'Windows Socket Error: (10060), on API 'connect?'  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

### Question:

I have a 3 tier app. using the TMidasConnection. I can connect to the server app when I use a client machine within the local net. I can also connect to it if I dial in directly into the local net. But, if I try to connect via the Internet (through an ISP) I get the error "Windows Socket Error: (10060), on API 'connect'". While connected to the ISP, I can ping my server machine, but the component does not see it.

### Answer:

You are probably running into the firewall. If the server is not outside the firewall sometimes you can use a Port # > 1024 to access it, but you should check with your IS department.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15379: Access to PDOXUSRS.NET slow with Novell client

Question and Answer Database

FAQ379B.txt    Access to PDOXUSRS.NET slow  
Category     :Database (Miscellaneous)  
Platform     :All  
Product      :Applies to all

### Question:

Accessing the PDOXUSRS.NET file on Novell drive takes 7-10 seconds with Windows 95 Client to Novell. Occurs with any BDE client.

### Answer:

Problem only occurs with Novell client. Microsoft is OK. So, the simplest solution is to switch to Netware client. Also, according to Novell the problem only occurs with Windows 95. Windows NT is fine.

Novell has a discussion of this problem on their web site. It may be found by going to [support.novell.com/search/kb\\_index.htm](http://support.novell.com/search/kb_index.htm), search on [pdoxusrs.net](http://pdoxusrs.net), then select the document called "slowness when launching Paradox on Windows 95"

Here is the solution this document states:

```
Cache writes = off
Close Behind Ticks=0
File Caching=0
Lock Delay=1
Lock Retries=1
Read Only Compatibility= on
True Commit=On
Opportunistic Locking=off (with the older client)
```

7/15/98 3:23:06 PM



## Article #15381: Connection Checklist

Question and Answer Database

FAQ381D.txt      Connection Checklist  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

Question:

What kinds of things can I check for every time I deploy a multi-tier application?

Answer:

Note: This list is not intended to be inclusive, but covers some of the most often missed items.

### ?On the Server?

- Export all dataset interfaces by right clicking the dataset components and selecting the "Export from .." context menu option.
- Rebuild the application.
- If you're running TCP/IP as the transport layer, run `\bin\Scktsrvr.exe` on the server.

### ?On the Client?

1. If possible, verify your connectivity setup. If you're using TCP/IP, you can ping the server from the client.
2. On the TRemoteServer or TMidasConnection:
  - Verify the ComputerName.
  - If the application is on another computer, verify the GUID.
  - If local, verify the ServerName.
  - Verify that you can set `Connected = true`.
  - If using TMidasConnection, check that your `ConnectType` matches the connectivity you have set up (i.e. `ctSockets` for TCP/IP, etc).
3. Verify that each TClientDataSet `ProviderName` and `RemoteServer` properties are set.



Also, check that you have needed files on both client and server and that they are registered.

See: "Files and components for Borland MIDAS"

[www.borland.com/midas/papers/components.html](http://www.borland.com/midas/papers/components.html).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15387: 'Invalid BLOB handle' error

Question and Answer Database

FAQ387B.txt 'Invalid BLOB handle' error

Category :Database (Miscellaneous)

Platform :All

Product :BDE 4.0

Question:

"Invalid BLOB handle" error: What is the cause and how do I fix this error?

Answer:

If the error occurs on a non-live (dead, canned) query or a table with no indexes, then increasing the new BDE 4.x configuration parameter 'BLOBS TO CACHE' will fix the problem. This setting determines how many BLOBs will be cached on the client. Applications that deal with fetching dead BLOBs using dead table opens or queries can set a limit on the number of BLOBs to cache depending on the resource available on the client. Setting a value 100 means the application can work with a maximum of 100 BLOB records cached. Fetching more than 100, then scrolling back 100 records results in an "Invalid BLOB handle in record buffer" error message.

NOTE:This parameter does not apply to live table opens.

Default Value: 64

Range: 64 to 65536

7/15/98 3:23:06 PM

Last Modified: 05-OCT-99

## Article #15389: Displaying a list of constraint error messages

Question and Answer Database

FAQ389D.txt     Displaying a list of constraint error messages

Category        :DCOM/MultiTier/ActiveForm

Platform        :All

Product         :Delphi 3.x

Question:

How can I show a list of constraint error messages to the user on the client, correct the errors and re-post at one time?

Answer:

You can only ever have one failed constraint at a time. For example, if you have 3 record level constraints and they are all going to fail when you post, then you will get an exception when you post that corresponds to the first constraint that failed. From there, you can't get the second constraint that failed until you fix the problem with the first one and attempt to re-post.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15394: Checking for NULL in OnUpdateData handler

Question and Answer Database

FAQ394D.txt      Checking for NULL in OnUpdateData handler

Category        :DCOM/MultiTier/ActiveForm

Platform        :All

Product         :Delphi 3.x

Question:

How can I tell if a user has changed a string field to null in the OnUpdateData event handler? How can I differentiate between a field that was changed to a null string and a field that was not changed at all.

Answer:

Use the NewValue property on the TField when reading the second record (the one which contains the edits). If the returned variant is empty (or Unassigned) then that field was not modified. Here is some code that demonstrates this:

```
var
  NewVal: Variant;
begin
  NewVal := DataSet.FieldByName('MyStrField').NewValue;
  if VarIsEmpty(NewVal) then
    ShowMessage('Field was not edited')
  else if VarIsNull(NewVal) then
    ShowMessage('Field was blanked out')
  else
    ShowMessage('New Field Value: ' + String(NewVal));
end;
```

If you look at the source for the RecError form (in the repository), you will see how it uses this information to display the '' string when showing the reconcile errors. On the server you add record level constraints using the constraints property of your TQuery/TTable or field level constraints using persistent TField objects (either CustomConstraint or ImportedConstraint). If you use field level constraints, they are enforced when data is posted to the field (i.e. when you tab out of a data aware

control associated with the field).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15401: Benefits of the ConstraintBroker

Question and Answer Database

FAQ401D.txt Benefits of the ConstraintBroker

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

What are the main benefits of the ConstraintBroker?

Answer:

The ConstraintBroker allows you to include check constraints with the data. That means that when you request the data, you get the rules. Those rules are automatically and codelessly enforced. Since they are codelessly enforced, you don't have to re-write or update your application if the rule changes. In fact, this is a simple way of updating the client application without shutting it down. Every application that uses the ConstraintBroker automatically has this feature.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15408: Getting the Win95 Desktop ListView Handle

Question and Answer Database

FAQ408D.txt Getting the Win95 Desktop ListView Handle

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get a handle to the Windows 95 Desktop to access the Desktop icons?

Answer:

The Windows 95 Desktop is overlaid with a ListView component. You simply need to get a handle to the ListView. Example:

```
function GetDesktopListViewHandle: THandle;
var
  S: String;
begin
  Result := FindWindow('ProgMan', nil);
  Result := GetWindow(Result, GW_CHILD);
  Result := GetWindow(Result, GW_CHILD);
  SetLength(S, 40);
  GetClassName(Result, PChar(S), 39);
  if PChar(S) <> 'SysListView32' then Result := 0;
end;
```

Once you have the handle, you can use the list view-related API functions in the CommCtrl unit to manipulate the desktop. See the LVM\_xxxx messages in the Win32 online help.

For example the following line of code:

```
SendMessage(GetDesktopListViewHandle, LVM_ALIGN, LVA_ALIGNLEFT, 0);
```

will align the desktop icons to the left side of the Windows 95 desktop.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15414: Deploying Midas

Question and Answer Database

FAQ414D.txt    Deploying Midas  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

Once I'm done developing an application server, how do I deploy to a dedicated PC?

Answer:

You need to purchase a MIDAS license for the application server. When you purchase a MIDAS license you receive a CD with the MIDAS suite for install. You need to install the EXE's and DLL's that you created for your app, the rest (which also includes the runtime packages, BDE, SQL Links, etc.) are installed by the MIDAS software.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15420: Table data subset through IProvider

Question and Answer Database

FAQ420D.txt Table data subset through IProvider

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

How can I position a TTable and get data from that starting position from the Provider?

Answer:

Using TQuery and Provider.SetParams is preferred. However, it is possible to do this with TTable. First have the client retrieve only the meta-data from the table by setting PacketRecords = 0, then calling Open. Next call an automation method on the server (you must define this method), which will cause the table to be positioned on the desired record. Finally, set PacketRecords to some value greater than zero, and call GetNextPacket.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15426: TProvider Component vs. Provider Property

Question and Answer Database

FAQ426D.txt TProvider Component vs. Provider Property

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

What is the purpose of the TProvider component vs. the Provider property of the dataset? When would the TProvider component be necessary?

Answer:

The Provider property of the dataset is an interface of type IProvider. A TProvider is not required, you can just use DataSet.Provider. Look at TDBDataSet.GetProvider for how it gets the IProvider. TProvider is useful, if you need any of the events.

This example will open a client dataset looking at a TTable on the same form.

?

```
uses BDEProv;  
...  
procedure Button1Click(Sender: TObject);  
begin  
    ClientDataSet1.Provider := Table1.Provider;  
    ClientDataSet1.Open;  
end;  
?
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15432: How do I make changes to the VCL code?

Question and Answer Database

FAQ432D.txt    How do I make changes to the VCL code?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I make changes to the VCL code?

Answer:

Note: Custom changes to the vcl are not supported by Borland or Borland Developer Support.

-But if you choose to do so...

VCL source code changes should never be made to the interface section of a unit. VCL source code changes should only be made to the implementation section.

The safest way to make changes to the VCL is to create a new directory called "patched". Copy the VCL file you want to change to the new directory. Make your changes (with comments) to the copied file. Next, add the path to your patched directory to the beginning of your library path. Restart Delphi/C++ Builder, and rebuild your project.

The library path setting can be found off the main menu of:

Delphi 1 : Options | Environment | Library

Delphi 2 : Tools | Options | Library

Delphi 3 : Tools | Environment Options | Library

C++ Builder : Options | Environment | Library

7/16/98 4:31:28 PM



## Article #15438: Providing lookup capability

Question and Answer Database

FAQ438D.txt Providing lookup capability

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

How do I provide lookup capability using ClientDataSets?

Answer:

Add an automation method to the server which can perform the lookup for you and return the result. You could do this in the OnGetText method of a TField object. Then you would just use a regular TDBEdit instead of a data aware component like the TDBLookupComboBox. Some master tables that change infrequently you may want to store locally. With the data stored locally you could use an automation method to check for changes and refresh the client if necessary.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15443: Reading the autoincrement value of Paradox table

Question and Answer Database

FAQ443B.txt Reading the autoincrement value of Paradox table

Category :Database (Miscellaneous)

Platform :All

Product :Applies to all

Question:

How is the next autoincrement value determined for a Paradox table?

Answer:

The current highest value is stored beginning at byte 73 decimal. The next value is determined by adding 1 to it.

Here is a simple Delphi function that returns the current autoincrement value.

```
function getAutoInc(filename : string) : LongInt;
var
    mystream : tfilestream;
    buffer : longint;
begin
    mystream := tfilestream.create(filename,
                                   fmOpenread + fmShareDenyNone);
    mystream.Seek(73, soFromBeginning);
    mystream.readbuffer(buffer, 4);
    mystream.Free;
    getAutoInc := buffer;
end;
```

8/18/98 11:10:49 AM

Last Modified: 05-OCT-99

## Article #15445: MasterSource/MasterFields behavior

Question and Answer Database

FAQ445D.txt MasterSource/MasterFields behavior

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

Are MasterSource/MasterFields like a filter in the sense that the ClientDataSet retrieves all records and filters for those records that match the linked fields?

Answer:

It really depends on how you have it set up. If you set PacketRecords to -1 then yes, that is the way it works. If you have PacketRecords to 0 and FetchOnDemand as True, then it will use Provider.SetParams and fetch just the records that match the master.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15450: The ordinal 237 could not be located ..

Question and Answer Database

FAQ450B.txt     The ordinal 237 could not be located ..

Category     :Database (Oracle)

Platform     :Win95/NT

Product      :

### Question:

I'm attempting to connect to Oracle and I'm getting the error "The ordinal value 237 could not be located in the dynamic link library ORA73.DLL"

### Answer:

This error occurs when you have a working install of Oracle other than version 7.3.2 and then install MTS and switch VENDOR INIT in BDE Admin to MTXOCI.DLL.

The error occurs because MTS installs version 7.3.2 of ORA73.DLL to WindowsSystem32.

To solve the problem rename ORA73.DLL in Windows/System32 and make sure ORA73.DLL is in your path.

8/18/98 12:49:44 PM

Last Modified: 04-NOV-99

## Article #15452: ODBC Constraints

Question and Answer Database

FAQ452D.txt - ODBC Constraints

Category :Database/VCL

Platform :All-32Bit

Product :

Question:

Can I import constraints from an ODBC database?

Answer:

No. ODBC does not define a constraint meta-data API.

4/2/99 11:40:25 AM

Last Modified: 01-SEP-99

## Article #15457: Problem using Oracle 8 driver(sqlora8.dll) with Oracle 7.x servers

Question and Answer Database

FAQ457B.txt      Problem using Oracle 8 driver(sqlora8.dll) with Oracle 7.x servers

Category        :Database(Oracle8)

Platform        :Win95/NT

Product         :

### Question:

I am trying to use BDE 5.0 and the Oracle 8 client software to connect to my Oracle 7.x server. I created a new alias and when I attempt to retrieve data using it, I am getting the error ORA-12663 "Services requested by client not available on server." All of the Oracle supplied tools connect and let me see data. What is wrong?

### Answer:

First, be sure you are using at least v8.0.4 of the Oracle client. Next be sure that in your Oracle alias for your 7.x server, you set the 'Object Mode' setting to false. When you create new Oracle aliases in BDE administrator with BDE v5.0, the Object mode parameter is set to true regardless of what DLL32(SQLORA8.dll or SQLORA32.dll) you have selected. For aliases that will reference Oracle 7.x servers Object Mode must be set to false for correct operation.

8/18/98 1:39:00 PM

Last Modified: 01-SEP-99

## Article #15459: Resource Usage

Question and Answer Database

FAQ459D.txt     Resource Usage  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

### Question:

How is memory used differently when a remote datamodule is created as a single vs. multiple instance?

### Answer:

A single instance server is going to require more memory for each additional client because the OS must create a whole new process and duplicate all of the data storage space. A multi-instance server will only require as much memory as used by the remote data module and DBCLIENT.DLL for data. The best way to figure out how much this is to use the task manager in NT4.0 and examine the memory used before and after client #2 connects.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15464: How much does MTS improve connect times?

Question and Answer Database

FAQ464B.txt      How much does MTS improve connect times?

Category        :Database(MTS)

Platform        :Win95/NT

Product         :

Question:

I've heard MTS Pooling can improve second and subsequent connect times to Oracle and MSSQL. Can you tell me more about the speed improvements?

Answer:

In small tests done in Developer Support, we have seen the following for second a subsequent connects:

Oracle 8, MTS ON, Average Connect Time: .13 seconds

MSSQL, MTS ON, Average Connect Time: .13 seconds

Oracle 8, MTS OFF, Average Connect Time: .45 seconds

MSSQL, MTS OFF, Average Connect Time: .65 seconds

8/18/98 12:51:05 PM

Last Modified: 05-OCT-99

## Article #15466: Sending a file via DCOM

Question and Answer Database

FAQ466D.txt    Sending a file via DCOM  
Category     :DCOM/MultiTier/ActiveForm  
Platform     :All  
Product      :Delphi 3.x

Question:

I would like to send a file through DCOM however I can only send standard types. How can I send bytes across to a client?

Answer:

You need to use a variant array (VarArrayLock should be useful as well).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15471: Phantom records appear on post

Question and Answer Database

FAQ471B.txt Phantom records appear on post

Category :Database (Oracle)

Platform :Win95/NT

Product :

Question:

I'm using an Oracle 8 database and I'm getting a duplicate copy of my record whenever it is posted. What is the cause?

Answer:

This occurs when using the Oracle 7 client to connect an Oracle 8 database. This is an unsupported configuration.

The Oracle 8 SQL Links 5.0 driver was tested using the Oracle 8.0.4 client (OCI).

We recommend using the Oracle 8 SQL Links driver and Oracle 8.0.4 client when connecting to an Oracle 8 server.

8/18/98 1:35:47 PM

Last Modified: 05-OCT-99

## Article #15473: Error: 'DAX Error - Name not unique in this context'

Question and Answer Database

FAQ473D.txt      Error: 'DAX Error - Name not unique in this context'

Category        :DCOM/MultiTier/ActiveForm

Platform        :All

Product         :Delphi 3.x

Question:

How do I eliminate "DAX Error - Name not unique in this context"?

Answer:

Set the HandleShared property on your TDatabase component to True.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15479: Error: 'DAX Error class not registered.'

Question and Answer Database

FAQ479D.txt     Error: 'DAX Error class not registered.'

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

The client machine displays error "DAX Error class not registered."

Answer:

You need to have DBCLIENT.DLL on the client, and it needs to be registered.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15484: Vendor Init with Oracle

Question and Answer Database

FAQ484B.txt Vendor Init with Oracle  
Category :Database (Oracle)  
Platform :Win95/NT  
Product :

Question:

Vendor Init error after upgrading to BDE 5.

Answer:

Change the Vendor Init in BDE Admin to 'ORA73.DLL'.  
Upgrading to BDE 5 automatically changes the Vendor Init to 'OCI.  
DLL', a Oracle 8 DLL.

8/18/98 1:36:27 PM

Last Modified: 01-SEP-99

## Article #15486: Error: 'EOleSysError : Error loading type library/DLL'

Question and Answer Database

FAQ486D.txt    Error: 'EOleSysError : Error loading type library/DLL'  
Category       :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

### Question:

The client machine displays error "EOleSysError: Error loading type library/DLL". DBCLIENT.DLL and IDPROV32.DLL files are present in the BDE directory of Delphi.

### Answer:

Make sure that STDVCL32.DLL is on the machine in the WINDOWS\SYSTEM directory and registered. See "Files necessary to run Application Server".

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15492: Error: 'General SQL Error : Connection does not exist' in threaded application

Question and Answer Database

FAQ492D.txt    Error: 'General SQL Error : Connection does not exist'  
in threaded application

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

I tried to do some DB updates in a thread. I ran into the following error : "General SQL Error : Connection does not exist". When I run the code outside the thread I have no problems at all. How can I fix this?

Answer:

You must have a Session in each thread.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15497: General SQL Error, Can't load MSJTER32.DLL

Question and Answer Database

FAQ497B.txt    General SQL Error, Can't load MSJTER32.DLL  
Category     :Database (MSACCESS)  
Platform     :Win95/NT  
Product      :

### Question:

How come I'm getting a General SQL Error about not being able to load MSJTER32.DLL when trying to connect to Access?

### Answer:

MSJTER32.DLL comes with Access 95. The most common cause of this error is when DLL32 in BDEADMIN is configured to connect to Access 95 DAO and the Access 97 DAO is installed. To configure BDEADMIN to use the Access 97 DAO, Select the Configuration Tab, Configuration: Drivers: Native: MSACCESS and set DLL32 to IDDA3532.DLL.

Another way to get this error is if BDEADMIN is configured to connect to the Access 95 DAO and the Access 95 DAO is not installed. Obtain the Access 95 DAO from Microsoft.

8/18/98 3:32:18 PM

Last Modified: 05-OCT-99

## Article #15499: Error: 'Interface not supported'

Question and Answer Database

FAQ499D.txt     Error: 'Interface not supported'

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

I receive an "Interface is not supported" error when trying to use an interface.

Answer:

Verify that STDVCL32.DLL and your type library are registered. If casting an interface, you must be using DCOM and the type library must be registered on the client.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15504: Single Tier Applications

Question and Answer Database

FAQ504D.txt Single Tier Applications

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

How do I set up a single tier application using Client Datasets?

Answer:

Currently, the only way to activate a client dataset at design time is to associate it with a provider, to load data from a file, or to assign data from another dataset. Typically, for a single tier application you would create the dataset at runtime using the CreateDataSet method.

Here is how you might code the FormCreate and FormDestroy methods:

?

### ?OnFormCreate:?

```
if FileExists('MYDATA.CDS') then
  ClientDataSet.LoadFromFile('MYDATA.CDS')
else
begin
  ClientDataSet.FieldDefs.Add('Field1', etc.);
  ClientDataSet.FieldDefs.Add('Field2', etc.);
  ClientDataSet.CreateDataset;
end;
```

### ?OnFormDestroy:?

```
if ClientDataSet.Active then
begin
  // Next line is optional,
  // if you don't want undo support you could
  // set LogChanges = False instead.
```

```
ClientDataSet.MergeChangeLog;  
ClientDataSet.SaveToFile('MYDATA.CDS');  
end;  
?
```

Once you've run the app once and created the data file, you can easily load it at design time (from the right-click component menu) if you need to have the dataset active at design time for some reason.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15509: Preventing windows task switching

Question and Answer Database

FAQ509D.txt Preventing windows task switching

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I prevent task switching (Alt-Tab, Ctrl-Esc, Ctrl-Alt-Del) in my application?

Answer:

Preventing a user from switching to another application is possible under Windows 95 and earlier versions of Windows by tricking Windows into thinking the screen saver is running. This method does not work under Windows NT, and is not guaranteed to be available in future versions of Windows. Many versions of Windows may also respond to a task switching trap installed by a CBT (Computer Based Training) application hook. To prevent task switching under Windows NT, you will need access to a third party keyboard device driver.

The following code will trick Windows into believing that the screen saver is running:

```
var
  OldValue : LongBool;
begin
  {turns the trap on}
  SystemParametersInfo(97, Word(True), @OldValue, 0);

  {turns the trap off}
  SystemParametersInfo(97, Word(False), @OldValue, 0);
end;
```

7/16/98 4:31:28 PM



## Article #15514: Using Stored Procedures

Question and Answer Database

FAQ514D.txt Using Stored Procedures  
Category :DCOM/MultiTier/ActiveForm  
Platform :All  
Product :Delphi 3.x

Question:

How do I use stored procedures in a multi-tier environment?

Answer:

You'll need to get the provider interface and set the parameters before trying to open the client dataset. Unfortunately there is no way to do this at design time, but it should work OK at run time. Here is some code that shows basically what you need to do:

?

```
procedure TDBClientTest.Button3Click(Sender: TObject);
var
  Params: Variant;
begin
  ClientData.Provider := RemoteServer1.GetProvider
('ProviderName');
  Params := VarArrayCreate([0, 1], varVariant);
  Params[0] := VarArrayOf(['@InParam1', 101]);
  Params[1] := VarArrayOf(['@InParam2', 'test_param']);
  ClientData.Provider.SetParams(Params);
  ClientData.Open;
end;
?
```

This example assumes the stored procedure takes 2 parameters, and binds them by name. If you need to update the data returned from a stored procedure, you will need to use a TUpdateSQLProvider.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15519: Testing IDispatch Descendants

Question and Answer Database

FAQ519D.txt    Testing IDispatch Descendants

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

Is there a minimal test I can perform to test the validity of the GUID, and that the interface descends from IDispatch (and therefore supports automation methods)?

Answer:

Call `CreateRemoteComObject` and use the name of the computer and the GUID you are trying to connect to. If it returns an error, this indicates a server problem, otherwise a client problem.

?

```
const
  MyGUID = '{444...111}'; //Whatever the guid is...
var
  Unk: IUnknown;
  Disp: IDispatch;
begin
  { Make sure this line works correctly }
  Unk := CreateRemoteComObject('server1',
    StringToGUID(MyGUID));
  { If it does, then cast it to a IDispatch }
  Disp := Unk as IDispatch;
end;
?
```

If this works and problems remain, then you'll need to step through your client code and see where it is failing. If not, then run the server in the debugger and connect with the client, then you should be able to debug near to where it fails.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15524: Where to get TUpdateSQLProvider component

Question and Answer Database

FAQ524D.txt Where to get TUpdateSQLProvider component

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

I keep seeing references to this TUpdateSQLProvider...  
where can I get it? I thought it was in the Delphi  
update, but I don't see it on my palette.

Answer:

It's not on the palette, it is a sample.

`\\delphi30\demos\midas\usqlprov`

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15529: Drag and Drop with Files

Question and Answer Database

FAQ529D.txt Drag and Drop with Files

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I accept files that are dropped on my application?

Answer:

You must interface with the Windows Shell API module to let Windows know that your application accepts dropped files (this can be done in your main form's create event), and then you must respond to the drag events as they happen by creating an event handler.

The following is an example of a Delphi form that accepts dropped files and adds the names of the files to a memo component:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Mem1: TMemo;
    procedure FormCreate(Sender: TObject);
  private
    procedure WMDROPPFILES(var Message: TWMDROPPFILES);
      message WM_DROPFILES;
    { Private declarations }
  public
    { Public declarations }
  end;
```



```

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses ShellApi;

procedure TForm1.FormCreate(Sender: TObject);
begin
  {Let Windows know we accept dropped files}
  DragAcceptFiles(Form1.Handle, True);
end;

procedure TForm1.WMDROPPFILES(var Message: TWMDROPPFILES);
var
  NumFiles : longint;
  i : longint;
  buffer : array[0..255] of char;
begin
  {How many files are being dropped}
  NumFiles := DragQueryFile(Message.Drop,
                           -1,
                           nil,
                           0);

  {Accept the dropped files}
  for i := 0 to (NumFiles - 1) do begin
    DragQueryFile(Message.Drop,
                  i,
                  @buffer,
                  sizeof(buffer));
    Form1.Memo1.Lines.Add(buffer);
  end;
end;

end.

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15534: Getting a handle on your files

Question and Answer Database

FAQ534D.txt Getting a handle on your files

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How do I get a file handle when using Delphi file variables?

Answer:

If you are using a TextFile, typecast your your textfile variable like this to expose the handle property:

```
TheHandle := TTextRec(MyFile).handle;
```

For binary files, typecast your file variable like this:

```
TheHandle := TFileRec(MyFile).handle;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15539: TSocketDispatcher

Question and Answer Database

FAQ539D.txt     TSocketDispatcher  
Category        :DCOM/MultiTier/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

### Question:

In the Help File is the following: Add a TSocketDispatcher object to a form or data module to create an application server that implements an IDispatch interface using TCP/IP. There is no TSocketDispatcher Component on the pallet. So how do you create it?.

### Answer:

You don't really need to do this. ScktSrvr.EXE is in the \delphi30\bin directory. Running it on your AppServer will allow sockets to connect to your application. If you want to use it anyway, look at the ScktSrvr.EXE file found in \delphi30\source\vcl directory. It has the code that instantiates a TSocketDispatcher.

The help file currently contains this note:

?

Application servers do not need to explicitly add a TSocketDispatcher so that they can connect to TMIDASConnection components in client applications. Delphi ships with an executable called ScktSrvr.exe. Run ScktSrvr on the server machine. If the application server does not want to use the default port number (211), pass the port number the application server should use for listening for client connections as a command line argument. If ScktSrvr.exe is running on the server machine, it will automatically launch the application server after it accepts client requests.

?

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15544: How to use the TUpdateSQLProvider.OnUpdateRecord method

Question and Answer Database

FAQ544D.txt    How to use the TUpdateSQLProvider.OnUpdateRecord method

Category     :DCOM/MultiTier/ActiveForm

Platform     :All

Product      :Delphi 3.x

Question:

How do I use TUpdateSQLProvider.OnUpdateRecord?

Answer:

In the UpdateRecord event, if UpdateKind is ukModify then there are 2 records that designate the important values. The first record contains old values, the second contains new values. In the event, the cursor will start on the second record.

The following example forces a field called Test to UpperCase.

```
procedure TForm1.UpdateSQLProvider1UpdateRecord(DataSet:
  TClientDataSet; UpdateKind: TUpdateKind;
  UpdateQuery: TQuery; var Applied: Boolean);
begin
  with DataSet do
  begin
    if (UpdateKind in [ukModify, ukInsert]) and
      (not VarIsEmpty(FieldByName('TEST').NewValue) then
      UpdateQuery.ParamByName('TEST').Value :=
      UpperCase(FieldByName('TEST').NewValue);
    end;
  end;
end;
```

I use VarIsEmpty on the NewValue to see if the value has changed, then I set the correct param for the query to the value I actually want it to be.



## Article #15549: About Deploying DBClient in a Web Application

Question and Answer Database

FAQ549D.txt About Deploying DBClient in a Web Application

Category :DCOM/MultiTier/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

I include dbclient.dll in the "additional files" section of the web deployment options but it never downloads to the client. How do I get the file down to the client, and once there, how does one register the Dll?

Answer:

Your INF file should have something like this;

```
[Add.Code]
```

```
dbclient.dll=dbclient.dll
```

```
[dbclient.dll]
```

```
file=http://yoursite.com/dbclient.cab
```

```
clsid={9E8D2F81-591C-11D0-BF52-0020AF32BD64}
```

```
RegisterServer=yes
```

```
FileVersion=4,0,0,36
```

Replace "yoursite" with your HTTP address where the cab file resides. FileVersion is the version of the file in your CAB file (check the version information of DBCLIENT to make sure the above matches).

Make sure FileVersion is the version of your DBCLIENT.DLL. You can put dbclient.dll in a cab file using CABARC found in the delphi\bin directory. A sample CABARC command line might look like the following example.

```
CABARC N DBCLIENT.CAB DBCLIENT.DLL
```





## Article #15554: Retrieving a long file name from the command line.

Question and Answer Database

FAQ554D.txt Retrieving a long file name from the command line.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I retrieve a long file name that contains spaces from the command line passed to my application?

Answer:

The Windows API function `GetCommandLine()` will provide you with the complete command line to your application. Note: you will need to make a copy of the returned string, and parse out the executable path and name from the beginning of the returned string.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s : string;
begin
  s:= GetCommandLine;
  Delete(s, 1, pos('exe"', s) + 4);
  Mem1.Lines.Add(s);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15559: Getting the local internet machine name and IP address.

Question and Answer Database

FAQ559D.txt - Getting the local internet machine name and IP address.

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

How do I get the local internet machine name and IP address?

Answer:

Getting the local machine name and IP address is a straight forward process that is most easily accomplished using the TCP component. Simply drop a TCP component from the internet page of the component palette on to a form, and access the following members of the TCP component:

```
Memo1.Lines.Add(TCP1.LocalHostName);  
Memo1.Lines.Add(TCP1.LocalIp);
```

If you prefer not to use the TCP component, here is an example that interfaces directly with your winsock layer:

```
uses Winsock;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
var  
    wVersionRequested : WORD;  
    wsaData : TWSaData;  
begin  
    {Start up WinSock}  
    wVersionRequested := MAKEWORD(1, 1);  
    WSASStartup(wVersionRequested, wsaData);  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var  
    p : PHostEnt;
```

```
s : array[0..128] of char;
p2 : pchar;
begin
  {Get the computer name}
  GetHostName(@s, 128);
  p := GetHostByName(@s);
  Mem1.Lines.Add(p^.h_Name);
  {Get the IpAddress}
  p2 := iNet_ntoa(PInAddr(p^.h_addr_list^^));
  Mem1.Lines.Add(p2);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  {Shut down WinSock}
  WSACleanup;
end;
```

4/2/99 11:43:39 AM

Last Modified: 01-SEP-99

## Article #15564: Using global values in your application.

Question and Answer Database

FAQ564D.txt Using global values in your application.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How do I create a global variable/constant/type in my application.

Answer:

Create a new unit and declare the variable, constant, type in interface section of the unit. When you want to use the variable in another unit, simply add the unit name to the uses clause of unit you wish to use it from.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15569: What is Delphi/400 Client Server Suite?

### Question:

What is the Delphi/400 Server Suite?

### Answer:

The Delphi/400 Client Server Suite is combination of Borland's Delphi Client Server Suite and two other software products acquired under license from TCIS of France. Those two products are ClientObjects/400 and ScreenDesigner/400.

Delphi/400 is now distributed by [System Objects](#)

Last Modified: 30-JAN-02

## Article #15574: Hiding your application from the Windows Taskbar.

### Question:

How do I hide my application from the Windows Taskbar?

### Answer:

It is possible to have your application running without an icon appearing in the task bar. Here's how to do this:

- In Delphi go to the menu option Project -> View Source.
- Add the Windows unit to the uses clause.
- Add `Application.ShowMainForm := False;` to the line after "Application.Initialize;"
- Add: `ShowWindow(Application.Handle, SW_HIDE);` to the line before "Application.Run;"
- If you still want your form to show then Add: `ShowWindow(Form1.Handle, SW_SHOWNORMAL);` to the line just before "Application.Run;"

Your main project source file should now look something like this:

```
program Project1;

uses
  Windows,
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.RES}

begin
  Application.Initialize;
  Application.ShowMainForm := False;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  ShowWindow(Application.Handle, SW_HIDE);
  ShowWindow(Form1.Handle, SW_SHOWNORMAL); //Only add this if you
want your form to show.
  Application.Run;
end.
```

- In the "initialization" section (at the very bottom) of each unit that uses a form, add:

```
begin
  ShowWindow(Application.Handle, SW_HIDE);
end.
```

Last Modified: 10-MAY-01

## Article #15580: Using an animated cursor.

Question and Answer Database

FAQ580D.txt Using an animated cursor.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How do use an animated cursor?

Answer:

You will first need to get a handle to a Windows cursor and assign it to the TScreen component's Cursors array.

Predefined cursors start at a negative index, and user defined custom cursors are assigned positive indexes.

The following is an example of a Delphi form that uses an animated cursor:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  h : THandle;
begin
  h := LoadImage(0,
                'C:\TheWall\Magic.ani',
                IMAGE_CURSOR,
                0,
                0,
                LR_DEFAULTSIZE or
                LR_LOADFROMFILE);
  if h = 0 then ShowMessage('Cursor not loaded') else begin
    Screen.Cursors[1] := h;
    Form1.Cursor := 1;
  end;
end;
```





## Article #15585: More Info on Delphi/400 Client Server Suite

Question and Answer Database

FAQ585D.txt - More Info on Delphi/400 Client Server Suite

Category :AS400

Platform :All

Product :

Question:

How do I get more information on Delphi/400 Client Server Suite?

Answer:

Marketing, Sales, Development and Support are now provided by [System Objects](#)

Last Modified: 17-FEB-00

## Article #15597: Retrieving a users login name.

Question and Answer Database

FAQ597D.txt Retrieving a users login name.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get the user's login name under 32-bit versions of Windows?

Answer:

The following function demonstrates how to use the GetUserName API call to accomplish this task in Delphi 3 or higher:

```
function GetCurrentUserName: string;
var
  Len: Cardinal; { This will have to be Integer, not cardinal, in
Delphi 3. }
begin
  Len := 255; { arbitrary length to allocate for username string,
plus one for null terminator }
  SetLength(Result, Len - 1); { set the length }
  if GetUserName(PChar(Result), Len) then { get the username }
    SetLength(Result, Len - 1) { set the exact length if it
succeeded }
  else
  begin
    RaiseLastWin32Error; { raise exception if it failed }
  end;
end;
```

In Delphi 2, use a function like the following:

```
function GetCurrentUserName2: string;
var
  Len: integer;
  buf: array [0..255] of char;
```

```
begin
  Len := SizeOf(buf);
  if GetUserName(@buf,Len) then
  begin
    Result := buf;
  end
  else
  begin
    Raise Exception.Create('Win32 Error: ' + IntToStr(GetLastError));
    { You could use FormatMessage to translate the error code into an
    error message }
  end;
end;
```

01/06/00

Last Modified: 06-JAN-00

## Article #15603: Changing the papersize of a print job.

Question and Answer Database

FAQ603D.txt - Changing the papersize of a print job.

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How can I change the papersize of my print job?

Answer:

One way to change printer settings at the start of a print job is to change the printer's devicemode structure.

See: TDEVMODE in the Delphi 1.02 help file or DEVMODE in the Delphi 2.01 help file for other settings you can change (providing the print driver supports the change).

The following example, contains code to change the papersize and the paper bin that is used:

```
procedure TForm1.Button1Click(Sender: TObject);
var
Device : array[0..255] of char;
Driver  : array[0..255] of char;
Port    : array[0..255] of char;
hDMode  : THandle;
PDMode  : PDEVMODE;
begin
  Printer.PrinterIndex := Printer.PrinterIndex;
  Printer.GetPrinter(Device, Driver, Port, hDMode);
  if hDMode <> 0 then begin
    pDMode := GlobalLock(hDMode);
    if pDMode <> nil then begin

      {Set to legal}
      pDMode^.dmFields := pDMode^.dmFields or dm_PaperSize;
```

```
pDMode^.dmPaperSize := DMPAPER_LEGAL;
```

```
{Set to custom size}
```

```
pDMode^.dmFields := pDMode^.dmFields or  
DM_PAPERSIZE or  
DM_PAPERWIDTH or  
DM_PAPERLENGTH;
```

```
pDMode^.dmPaperSize := DMPAPER_USER;
```

```
pDMode^.dmPaperWidth := 100 {SomeValueInTenthsOfAMillimeter};
```

```
pDMode^.dmPaperLength := 100 {SomeValueInTenthsOfAMillimeter};
```

```
{Set the bin to use}
```

```
pDMode^.dmFields := pDMode^.dmFields or DMBIN_MANUAL;
```

```
pDMode^.dmDefaultSource := DMBIN_MANUAL;
```

```
GlobalUnlock(hDMode);
```

```
end;
```

```
end;
```

```
Printer.PrinterIndex := Printer.PrinterIndex;
```

```
Printer.BeginDoc;
```

```
Printer.Canvas.TextOut(100,100, 'Test 1');
```

```
Printer.EndDoc;
```

```
end;
```

4/2/99 11:21:00 AM

Last Modified: 01-SEP-99

## Article #15609: Purchasing Delphi/400 Client Server Suite

Question and Answer Database

FAQ609D.txt - Purchasing Delphi/400 Client Server Suite

Category :AS400

Platform :All

Product :

Question:

How can I purchase the Delphi/400 Client Server Suite?

Answer:

Delphi/400 is solely developed, marketed, sold and supported now from System Objects.

Last Modified: 16-FEB-00

## Article #15615: Creating a rotated font.

Question and Answer Database

FAQ615D.txt Creating a rotated font.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I create a rotated font?

Answer:

Rotating fonts is a straight forward process, so long as the Windows font mapper can supply a rotated font based on the font you request.

Note: Using a TrueType font virturally guarantees success.

Here is an example of creating a font that is rotated 45 degrees:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  lf : TLogFont;
  tf : TFont;
begin
  with Form1.Canvas do begin
    Font.Name := 'Arial';
    Font.Size := 24;
    tf := TFont.Create;
    tf.Assign(Font);
    GetObject(tf.Handle, sizeof(lf), @lf);
    lf.lfEscapement := 450;
    lf.lfOrientation := 450;
    tf.Handle := CreateFontIndirect(lf);
    Font.Assign(tf);
    tf.Free;
    TextOut(20, Height div 2, 'Rotated Text!');
  end;
end;
```



Last Modified: 01-SEP-99

## Article #15621: Converting from Long to Short file names and paths

Question and Answer Database

FAQ621D.txt     Converting from Long to Short file names and paths

Category     :Windows API

Platform     :All

Product      :All 32 bit

Question:

How do I retrieve the short file name of a given file or path?

Answer:

Use the Windows API function GetShortPathName.

Here is an Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Buffer : array [0..255] of char;
begin
  GetShortPathName(
    'C:\Program Files\Borland\Common Files\Bde\Bde32.hlp',
    @Buffer,
    sizeof(Buffer));
  Mem1.Lines.Add(Buffer);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15627: Making text background transparent

Question and Answer Database

FAQ627D.txt Making text background transparent

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I make the text background transparent?

Answer:

Use the Windows API function SetBkMode().

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldBkMode : integer;
begin
  with Form1.Canvas do begin
    Brush.Color := clRed;
    FillRect(Rect(0, 0, 100, 100));
    Brush.Color := clBlue;
    TextOut(10, 20, 'Not Transparent!');
    OldBkMode := SetBkMode(Handle, TRANSPARENT);
    TextOut(10, 50, 'Transparent!');
    SetBkMode(Handle, OldBkMode);
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15633: Getting user name and company info from Windows.

Question and Answer Database

FAQ633D.txt Getting user name and company info from Windows.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I retrieve the user name and company information from Windows?

Answer:

This information is stored under the "HKEY\CURRENT USER" section of the Windows registry. The task of retrieving this entry is greatly simplified by using the TRegIniFile component.

Example:

```
uses Registry;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    reg: TRegIniFile;
```

```
begin
```

```
    reg := TRegIniFile.create('SOFTWARE\MICROSOFT\MS SETUP (ACME)\');
```

```
    Mem1.Lines.Add(reg.ReadString('USER INFO',
```

```
        'DefName',
```

```
        'Frank Borland'));
```

```
    Mem1.Lines.Add(reg.ReadString('USER INFO',
```

```
        'DefCompany',
```

```
        'A Valued Borland Customer'));
```

```
    reg.free;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15639: Getting the serial number of a disk.

Question and Answer Database

FAQ639D.txt Getting the serial number of a disk.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I retrieve the serial number of a disk?

Answer:

You can use the Windows API function `GetVolumeInformation()`

```
procedure TForm1.Button1Click(Sender: TObject);
var
  VolumeSerialNumber : DWORD;
  MaximumComponentLength : DWORD;
  FileSystemFlags : DWORD;
  SerialNumber : string;
begin
  GetVolumeInformation('C:\',
                      nil,
                      0,
                      @VolumeSerialNumber,
                      MaximumComponentLength,
                      FileSystemFlags,
                      nil,
                      0);
  SerialNumber := IntToHex(HiWord(VolumeSerialNumber), 4) +
                  '-' +
                  IntToHex(LoWord(VolumeSerialNumber), 4);
  Mem1.Lines.Add(SerialNumber);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15645: Executing a program from my application

Question and Answer Database

FAQ645D.txt Executing a program from my application

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I execute a program from my application?

Answer:

Using the WinExec function is the easiest way to execute an program from your application.

To execute a standard program, simply use:

```
WinExec('C:\WINDOWS\notepad.exe', SW_SHOWNORMAL);
```

To add a command line, use:

```
WinExec('C:\WINDOWS\notepad.exe C:\WINDOWS\win.ini', SW_SHOWNORMAL);
```

To pop up a DOS box, use:

```
WinExec('command.com', SW_SHOWNORMAL);
```

To execute a DOS command, use:

```
WinExec('command.com /c dir *.*', SW_SHOWNORMAL);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15651: Writing a raw string of data to the printer.

Question and Answer Database

FAQ: FAQ651D - Writing a raw string of data to the printer.

Category: Printing

Platform: All-32Bit

Product:

Question:

How do I write a raw string of a data to the printer?

Answer:

Under Win16, you can use the SpoolFile function, or the Passthrough escape if the printer supports it. This is detailed in TI3196 - Direct Commands to Printer - Passthrough/Escape available from the Borland web site. Under Win32, you should use WritePrinter.

The following is an example of opening a printer and writing a raw string of data to the printer. Note that you must send the correct printer name, such as "HP LaserJet 5MP" for the function to succeed. You are also responsible for embedding any necessary control codes that the printer may require.

```
uses WinSpool;
```

```
procedure WriteRawStringToPrinter(PrinterName:String; S:String);
var
  Handle: THandle;
  N: DWORD;
  DocInfol: TDocInfol;
begin
  if not OpenPrinter(PChar(PrinterName), Handle, nil) then
  begin
    ShowMessage('error ' + IntToStr(GetLastError));
    Exit;
  end;
  with DocInfol do begin
```

```
pDocName := PChar('test doc');  
pOutputFile := nil;  
pDataTyPe := 'RAW';
```

```
end;  
StartDocPrinter(Handle, 1, @DocInfo1);  
StartPagePrinter(Handle);  
WritePrinter(Handle, PChar(S), Length(S), N);  
EndPagePrinter(Handle);  
EndDocPrinter(Handle);  
ClosePrinter(Handle);  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  WriteRawStringToPrinter('HP', 'Test This');  
end;
```

6/28/99 10:01:10 AM

Last Modified: 01-SEP-99



## Article #15657: Multi-Threaded ISAPI DLLs

Question and Answer Database

FAQ657D.txt - Multi-Threaded ISAPI DLLs

Category :Internet/WEB  
Platform :All-32Bit  
Product :

Question:

Why can't my ISAPI DLL created in Delphi 3 handle multiple connections.

Answer:

Although the ISAPI DLL wizard in Delphi 3 creates a DLL that is multi-thread safe, there is an omission in the source code which fails to set a flag telling the application that the DLL is multi-thread safe. This shortcoming can be overcome by simply adding the line:

```
IsMultiThread := TRUE;
```

as the first line of your DPR's begin-end block.

4/2/99 11:43:51 AM

Last Modified: 01-SEP-99

## Article #15663: InterBase access from a web server application

Question and Answer Database

FAQ663D.txt - InterBase access from a web server application

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

How can I access Local Interbase from a web server application?

I'm using NT and getting the error message "Unknown user name or password." when attempting to access Local Interbase from a Web server application. How do I fix the problem?

Answer:

Make the following changes to your configuration:

?

?

1. Interbase must be running as a service (not as an app w/ tray icon). Which you can change by clicking on the Properties for the tray icon for IB.

?

2. Both the Interbase server and WWW Server must have the following right set: Control Panel | Services | InterBase Server | Startup make sure "Allow Service to Interact with Desktop is checked".

?

4/2/99 11:44:02 AM

Last Modified: 01-SEP-99

## **Article #15669: Where can I find evaluation copies of Delphi/400 Client Server Suite?**

Question and Answer Database

FAQ669D.txt - Evaluation Copies of Delphi/400 Client Server Suite

Category :AS400

Platform :All

Product :

Question:

Will Evaluation copies of Delphi/400 Client Server Suite be available?

Answer: Evaluation copies are available from System Objects.

Last Modified: 01-MAR-00

## Article #15675: Upgrade to Delphi/400 for Delphi Developer users

Question and Answer Database

FAQ675D.txt - Upgrade to Delphi/400 for Delphi Developer users

Category :AS400  
Platform :All-32Bit  
Product :

Question:

Can Delphi Developers users upgrade to Delphi/400?

Answer:

Yes.

Delphi Developers users would need to purchase and upgrade to Delphi/400 Client/Server. International customers please contact your local Borland office or Client/Server Business Partner Delphi Client/Server users can upgrade to Delphi/400 Client/Server. International customers please contact your local Borland office or Client/Server Business Partner.

3/30/99 4:14:34 PM

Last Modified: 17-FEB-00

## Article #15681: Technical Support for Delphi/400

Question and Answer Database

FAQ681D.txt - Technical Support for Delphi/400

Category :AS400

Platform :All

Product :

Question:

How and where do I get technical support?

Answer: Marketing, Sales, Development and Support are now provided by System Objects

Last Modified: 17-FEB-00

## Article #15687: Benefits to AS/400 developers and users

Question and Answer Database

FAQ687D.txt - Benefits to AS/400 developers and users

Category :AS400  
Platform :All  
Product :

Question:

What are the key benefits to AS/400 developers and users?

Answer:

The main benefit of Delphi/400 Client/Server Suite to the AS/400 community, is that this solution will extend and enhance the life of their current AS/400 investments. For example their existing applications can be accessed via Delphi, rather than being re-written from scratch. Delphi also provides an easy to use GUI interface allowing new applications to be developed in the shortest possible time. More and more users are wishing to access corporate information combined with rapid reporting. Delphi's Internet enablement, will also allow these companies to start developing both Internet and Intranet applications immediately.

3/30/99 4:18:28 PM

Last Modified: 01-SEP-99

**Article #15693: How does Delphi/400 complement and add to Borland's strategy?**

Question and Answer Database

FAQ693D.txt - How does Delphi/400 complement and add to Borland's strategy?

Category :AS400

Platform :All

Product :

Marketing, Sales, Development and Support are now provided by System Objects

Last Modified: 17-FEB-00

## Article #15705: Minimum Hardware and Software Requirements

Question and Answer Database

FAQ705D.txt - Minimum Hardware and Software Requirements

Category :AS400  
Platform :All  
Product :

Question:

What are the minimum hardware and software requirements?

Answer:

Delphi/400 Client/Server Suite:

Intel 486/25-based PC or higher  
Microsoft Windows 95 or Windows NT  
8Mb(12Mb recommended)  
50Mb hard disk space  
CD-ROM Drive  
Mouse or other Windows pointing device

ClientObjects/400:

All the AS/400s except those of the AS/Entry series.  
OS/400 Version 2.3 or later.

Supported Routers:

IBM, Client Access for Windows 95 Vn. 2.3  
Netsoft, NS Bundle Vn. 5.0  
Microsoft, SNA Server Vn. 2.11/3.0  
Wall Data, Rumba Access for Windows 95/Windows NT

4/1/99 2:44:12 PM

Last Modified: 01-SEP-99



## Article #15711: Displaying the selected row of a DBGrid in a different color

Question and Answer Database

FAQ711D.txt    Displaying the selected row of a DBGrid in a different color

Category       :Database/VCL

Platform       :All

Product        :Delphi 3.x

Question:

How can I show selected row in a dbgrid in a different color without using dgRowSelect (because I need to edit the data)?

Answer:

If you want to color the selected row in a DBGrid but you don't want to use the dgRowSelect option because you want to be able to edit the data you can use the following technique on the DBGrid.OnDrawColumnCell event:

```
type
  TCustomDBGridCracker = class(TCustomDBGrid);

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  with Cracker(Sender) do
    if DataLink.ActiveRecord = Row - 1 then
      Canvas.Brush.Color := clRed
    else
      Canvas.Brush.Color := clWhite;
  DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15717: inserting contents of a file at current position in TMemo

Question and Answer Database

FAQ717D.txt inserting contents of a file at current position in TMemo

Category :VCL  
Platform :All  
Product :All 32 bit

Question:

How can I insert the contents of a file at the current position in a TMemo component?

Answer:

Use a TMemoryStream to read the file, then use the TMemo's SetSelTextBuf() method to insert the text;

```
var
  TheMStream : TMemoryStream;
  Zero : char;
begin
  TheMStream := TMemoryStream.Create;
  TheMStream.LoadFromFile('C:\AUTOEXEC.BAT');
  TheMStream.Seek(0, soFromEnd);
  //Null terminate the buffer!
  Zero := #0;
  TheMStream.Write(Zero, 1);
  TheMStream.Seek(0, soFromBeginning);
  Mem1.SetSelTextBuf(TheMStream.Memory);
  TheMStream.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15729: Using the LzExpand functions in your applications

Question and Answer Database

FAQ729D.txt Using the LzExpand functions in your applications

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I use the LzExpand functions in my application?

Answer:

Here is an example that will decompress a file that was compressed using the Microsoft compress.exe utility utilizing the -r (rename) command line switch. Note: If you already know the expanded name of the file, or it was not compressed with the -r command line switch, then simply omit the code associated with the GetExpandedName function, and directly hard code the file name as needed.

```
uses LzExpand;
```

```
{ $IFDEF WIN32 }
```

```
const MAX_PATH = 144;
```

```
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var lpDstFileName : PChar;
```

```
    SrcFile : integer;
```

```
    DstFile : integer;
```

```
    SrcOs : TOFSTRUCT;
```

```
    DstOs : TOFSTRUCT;
```

```
begin
```

```
    GetMem(lpDstFileName, MAX_PATH);
```

```
    GetExpandedName('C:\TestFile.bi_', lpDstFileName);
```

```
    SrcFile := LZOpenFile('C:\TestFile.bi_', SrcOs, OF_READ);
```

```
    DstFile := LZOpenFile(lpDstFileName, DstOs, OF_CREATE);
```

```
    LZCopy(SrcFile, DstFile);
```

```
    LZClose(SrcFile);
```

```
    LZClose(DstFile);
```

```
    FreeMem(lpDstFileName, MAX_PATH);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15735: Changing the display mode settings in code.

Question and Answer Database

FAQ735D.txt Changing the display mode settings in code.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I change the display mode settings in code?

Answer:

You can use the Windows API function EnumDisplaySettings() to get a list of valid mode settings, and use the Windows API function ChangeDisplaySettings() to actually change the display setting. Note that many drivers will not perform the reset without rebooting the computer, even though they are capable of doing so.

Example:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, Spin;
```

```
type
```

```
PdmArray = ^TDmArray;  
TDmArray = array[0..0] of TDeviceMode;
```

```
type
```

```
TForm1 = class(TForm)  
    Memo1: TMemo;  
    SpinEdit1: TSpinEdit;  
    Button1: TButton;  
    procedure FormCreate(Sender: TObject);  
    procedure FormDestroy(Sender: TObject);
```

```

    procedure Button1Click(Sender: TObject);
private
    lpDmArray : PDmArray;
    NumModes : integer;
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
    i : integer; // auto-initialized to zero
    MoreModes : bool;
    dm : TDeviceMode;

begin
    Mem1.Lines.Clear;
    MoreModes := True;

    while MoreModes do begin
        MoreModes :=
            EnumDisplaySettings(nil,
                                i,
                                dm);
        Mem1.Lines.Add('Mode ' +
            IntToStr(i) + ': ' +
            IntToStr(dm.dmBitsPerPel) + ' Bits Per Pixel ' +
            IntToStr(dm.dmPelsWidth) + ' x ' +
            IntToStr(dm.dmPelsHeight));
        Inc(i);
    end;

    NumModes := i;
    SpinEdit1.MinValue := 0;
    SpinEdit1.MaxValue := NumModes;

    GetMem(lpDmArray, sizeof(TDeviceMode) * NumModes);
    FillChar(lpDmArray^, sizeof(TDeviceMode) * NumModes, #0);

```

```

{$IFOPT R+}
  {$DEFINE CKRANGE}
  {$R-}
{$ENDIF}
  for i := 0 to (NumModes - 1) do
    EnumDisplaySettings(nil,
                        i,
                        lpDmArray[i]);

{$IFDEF CKRANGE}
  {$UNDEF CKRANGE}
  {$R+}
{$ENDIF}
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  ReturnVal : LongInt;
begin
{$IFOPT R+}
  {$DEFINE CKRANGE}
  {$R-}
{$ENDIF}
  ReturnVal := ChangeDisplaySettings(lpDmArray[SpinEdit1.Value],
                                    CDS_UPDATEREGISTRY);

{$IFDEF CKRANGE}
  {$UNDEF CKRANGE}
  {$R+}
{$ENDIF}
  with Mem1.Lines do begin
    case ReturnVal of
      DISP_CHANGE_SUCCESSFUL: Add('DISP_CHANGE_SUCCESSFUL');
      DISP_CHANGE_RESTART   : Add('DISP_CHANGE_RESTART');
      DISP_CHANGE_BADFLAGS  : Add('DISP_CHANGE_BADFLAGS');
      DISP_CHANGE_FAILED    : Add('DISP_CHANGE_FAILED');
      DISP_CHANGE_BADMODE   : Add('DISP_CHANGE_BADMODE');
      DISP_CHANGE_NOTUPDATED: Add('DISP_CHANGE_NOTUPDATED');
    end;
  end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeMem(lpDmArray, sizeof(TDeviceMode) * NumModes);
end;

```

end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15741: How many colors will my display mode support

Question and Answer Database

FAQ741D.txt    How many colors will my display mode support

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I find the number of colors the current display mode supports?

Answer:

You can find the color depth of a display context in the following manner:

```
GetDeviceCaps(Form1.Canvas.Handle, BITSPIXEL) *
```

```
GetDeviceCaps(Form1.Canvas.Handle, PLANES)
```

Will give you the total number of bits used to color a pixel. possible  
Return values of:

1 = 2 colors bpp

4 = 16 colors bpp

8 = 256 colors bpp

15 = 32768 colors (will return 16 on most drivers) bpp

16 = 65535 colors bpp

24 = 16,777,216 colors bpp

32 = 16,777,216 colors (same as 24) bpp

You can use:

```
NumberOfColors := (1 shl  
    (GetDeviceCaps(Form1.Canvas.Handle, BITSPIXEL) *  
    GetDeviceCaps(Form1.Canvas.Handle, PLANES)));
```

to calculate the total number of colors.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15747: Generating a Windows Console Application

Question and Answer Database

FAQ747D.txt     Generating a Windows Console Application

Category     :Windows API

Platform     :All

Product      :All 32 bit

Question:

How do I generate a Windows Console Application?

Answer:

Create a new application, then, from the View... Menu option, start the "Project Manager", and remove all units from the project. When asked if you would like to save, Click on the 'NO' button. Then from the View... menu option, select "Project Source" and edit it so it looks like the below example. Note: You can add and use additional units, so long as you do not use any VCL components.

Example:

```
program Project1;

uses
  Windows;

{$R *.RES}

begin
  Writeln('A Console Applicaiton');
  SetConsoleTextAttribute(GetStdHandle(
    STD_OUTPUT_HANDLE),
    FOREGROUND_RED OR
    BACKGROUND_BLUE);
  Writeln('Red text on a blue background ');
  SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
    FOREGROUND_BLUE OR BACKGROUND_RED);
  writeln('Blue text on a red background ');
  {White text on black background}
  SetConsoleTextAttribute(      GetStdHandle(STD_OUTPUT_HANDLE),
```

```
BACKGROUND_GREEN OR  
BACKGROUND_RED OR  
BACKGROUND_BLUE);
```

```
Writeln('Press Enter To End');
```

```
Readln;
```

```
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15753: Opening an explorer window to a given folder

Question and Answer Database

FAQ753D.txt Opening an explorer window to a given folder

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I open an explorer window to a given folder?

Answer:

A) The following example opens a explorer window containing the contents of the Windows directory.

Example:

```
uses ShellApi;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShellExecute(0,
                'explore',
                'C:\WINDOWS',
                nil,
                nil,
                SW_SHOWNORMAL);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15759: Prevent the TEdit component from beeping when invalid

Question and Answer Database

FAQ759D.txt Prevent the TEdit component from beeping when invalid

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I prevent the TEdit component from beeping when invalid keystrokes are entered?

Answer:

Trap the KeyPress event for the keys you do not want, and set the key to #0 to prevent the beep.

Example:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if ((UpCase(Key) < 'A') or
      (UpCase(Key) > 'Z')) then
    Key := #0;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15765: Parameter value not in allowed range

Question and Answer Database

FAQ765D.txt Parameter value not in allowed range

Category :Miscellaneous

Platform :All

Product :All 32 bit

### Question:

When I run my console application or execute a DOS box, I sometimes get an undocumented message "Parameter value not in allowed range" that displays in the window. The message seems to be posted before the first line of code is run since it is there even when I use "trace into" to start the program. How can I get around this?

### Answer:

On some Win95 OSR2 systems, there is an invalid value contained in the "Initial Environment" setting in the \_DEFAULT.PIF, and CONAGENT.PIF file that Windows uses for dos box defaults. Opening the properties dialog for these files, and making a change to the "Initial Environment" setting (located on the Memory tab) seems to correct the problem.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15771: Mapping a DLL into another process

Question and Answer Database

FAQ771D.txt Mapping a DLL into another process

Category :Windows API

Platform :All

Product :All 32 bit

Question:

Is it possible to map a DLL into another process under Win32?

Answer:

Yes, the following details one method of forcing a DLL to be mapped into a different process under both Windows NT, and Windows 95.

?

?

1. Install a global WH\_GETMESSAGE hook which is implemented by the DLL. Note that you must store the HHOOK variable returned by the SetWindowsHookEx() function (and any other global variables that are used in the DLL) into a memory mapped file to be able to access it in the target process. Note that any global variables contained in the DLL will not be mapped into the other process, and will contain uninitialized values when referenced while the DLL is mapped into the other process(s).

?

2. PostMessage a custom message (WM\_USER + something) to a window in the target process.

?

3. When the hook traps the message the DLL will be mapped into the target process during the processing of the message.

?



7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15777: Deleting a directory and all the directories files

Question and Answer Database

FAQ777D.txt    Deleting a directory and all the directories files  
Category        :VCL  
Platform        :All  
Product         :All 32 bit

Question:

How can I delete a directory and all the directories files?

Answer:

The following example demonstrates deleting all the files in a directory and then the directory itself. Additional processing would be required to delete read only files and files that are in use.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DirInfo: TSearchRec;
  r : Integer;
begin
  r := FindFirst('C:\Download\Test\*..*', FaAnyfile, DirInfo);
  while r = 0 do begin
    if ((DirInfo.Attr and FaDirectory <> FaDirectory) and
        (DirInfo.Attr and FaVolumeId <> FaVolumeID)) then
      if DeleteFile(pChar('C:\Download\test\' + DirInfo.Name))
         = false then
        ShowMessage('Unable to delete : C:\Download\test\' +
                    DirInfo.Name);
    r := FindNext(DirInfo);
  end;
  SysUtils.FindClose(DirInfo);
  if RemoveDirectory('C:\Download\Test') = false then
    ShowMessage('Unable to delete direcotry : C:\Download\test');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15783: Changing the default width/height of the editor

Question and Answer Database

FAQ783D.txt Changing the default width/height of the editor

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I change the default width and height of the IDE's Editor window?

Answer:

The following steps will allow you to change the default size of the IDE's editors window:

- 1) Run the Windows registry editor program "RegEdit.exe".

Depending on the version of the compiler you are using, open one of the following keys:

HKEY\_CURRENT\_USER\Software\Borland\Delphi\2.0\Editor

HKEY\_CURRENT\_USER\Software\Borland\Delphi\3.0\Editor

HKEY\_CURRENT\_USER\Software\Borland\C++Builder\1.0\Editor

- 2) Right click on the Editor folder in the left window pane, and choose "New -> String Value" from the popup menu. A new item titled "New Value #1" will appear in the right window pane.

- 3) Right click on the new item that was added in the right window pane, select "Rename" from the popup menu, can change the name of the new item to "DefaultWidth".

- 4) Right click on the new item that you renamed to "DefaultWidth", and select "Modify" from the pop up window. Now type in the value in pixels that you wish the IDE's Editor Window default width to be.

- 5) Repeat the above steps starting with step #2, but this time add a second item called "DefaultHeight".

- 6) Close the "regedit.exe" program.

Note that the change will not be effective until you exit the IDE and restart it.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15789: Getting a list of the available drives on a system

Question and Answer Database

FAQ789D.txt    Getting a list of the available drives on a system  
Category     :Windows API  
Platform     :All  
Product      :All 32 bit

### Question:

Is there a fast way to get a list of the available drives on a system?

### Answer:

The following example demonstrates getting a list of the available logical drives available on the system.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ld : DWORD;
  i : integer;
begin
  ld := GetLogicalDrives;
  for i := 0 to 25 do begin
    if (ld and (1 shl i)) <> 0 then
      Mem1.Lines.Add(Char(Ord('A') + i) + ':\');
    end;
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15795: Adding a document to the Windows 95 Start Button's documents

Question and Answer Database

FAQ795D.txt Adding a document to the Windows 95 Start Button's documents

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I add a document to the Windows 95 Start Button's documents menu?

Answer:

Use the Shell function SHAddToRecentDocs.

Example:

```
uses ShlOBJ;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  s : string;
```

```
begin
```

```
  s := 'C:\Download\ntkfaq.html';
```

```
  SHAddToRecentDocs(SHARD_PATH, pChar(s));
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15801: Retrieving the program that is associated with a given extension

Question and Answer Database

FAQ801D.txt    Retrieving the program that is associated with a given extension

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I retrieve the program that is associated with a given extension?

Answer:

The following example demonstrates retrieving the file associated with a file extension.

Example:

```
uses
{$IFDEF WIN32}
    Registry; {We will get it from the registry}
{$ELSE}
    IniFiles; {We will get it from the win.ini file}
{$ENDIF}

{$IFDEF WIN32}
    const MAX_PATH = 144;
{$ENDIF}

function GetProgramAssociation (Ext : string) : string;
var
{$IFDEF WIN32}
    reg: TRegistry;
    s : string;
{$ELSE}
    WinIni : TIniFile;
    WinIniFileName : array[0..MAX_PATH] of char;
    s : string;
{$ENDIF}
```

```

begin
{$IFDEF WIN32}
  s := '';
  reg := TRegistry.Create;
  reg.RootKey := HKEY_CLASSES_ROOT;
  if reg.OpenKey('.') + ext + '\shell\open\command',
    false) <> false then begin
{The open command has been found}
  s := reg.ReadString('');
  reg.CloseKey;
end else begin
{perhaps thier is a system file pointer}
  if reg.OpenKey('.') + ext,
    false) <> false then begin
    s := reg.ReadString('');
    reg.CloseKey;
    if s <> '' then begin
{A system file pointer was found}
  if reg.OpenKey(s + '\shell\open\command',
    false) <> false then
{The open command has been found}
    s := reg.ReadString('');
    reg.CloseKey;
    end;
  end;
end;
end;
{Delete any command line, quotes and spaces}
if Pos('%', s) > 0 then
  Delete(s, Pos('%', s), length(s));
if ((length(s) > 0) and
  (s[1] = '"')) then
  Delete(s, 1, 1);
if ((length(s) > 0) and
  (s[length(s)] = '"')) then
  Delete(s, Length(s), 1);
while ((length(s) > 0) and
  ((s[length(s)] = #32) or
  (s[length(s)] = '"'))) do
  Delete(s, Length(s), 1);
{$ELSE}
  GetWindowsDirectory(WinIniFileName, sizeof(WinIniFileName));
  StrCat(WinIniFileName, '\win.ini');
  WinIni := TIniFile.Create(WinIniFileName);
  s := WinIni.ReadString('Extensions',
    ext,

```



```
'' );
```

```
WinIni.Free;
```

```
{Delete any command line}
```

```
if Pos(' ^', s) > 0 then
```

```
    Delete(s, Pos(' ^', s), length(s));
```

```
{$ENDIF}
```

```
result := s;
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    ShowMessage(GetProgramAssociation('gif'));
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15807: Using Ampersands inside Delphi.

Question and Answer Database

FAQ807D.txt Using Ampersands inside Delphi

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I make a Window control's text property contain an ampersand character?

Answer:

Use two ampersand characters. The Windows default text handling for controls interprets a single Ampersand character as meaning the next character is a hot key.

Example:

```
Button1.Caption := 'This && That';
```

7/16/98 4:31:28 PM

Last Modified: 21-MAR-00

## Article #15812: "Error reading symbol file." error

Question and Answer Database

FAQ812D.txt "Error reading symbol file." error

Category :IDE

Platform :All

Product :All 32 bit

Question:

Q) When I open a project, I get the message "Error reading symbol file." then the project file wont compile. How can I get around this?

Answer:

A) This usually occurs when you move a project to another drive or directory. Simply deleting the project's .DSM and .DSK files should remedy the situation.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15818: Executing a '.LNK' file

Question and Answer Database

FAQ818D.txt     Executing a '.LNK' file

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

Q) How can I execute a '.LNK' file?

Answer:

A) The following example demonstrates executing a .lnk file in the Windows start menu.

Example:

```
uses ShellApi;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShellExecute(0,
                nil,
                'C:\WINDOWS\START MENU\DELPHI\Delphi3.lnk',
                nil,
                nil,
                SW_SHOWNORMAL);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15824: How can I change the drive that MediaPlayer uses to play Audio

Question and Answer Database

FAQ824D.txt    How can I change the drive that MediaPlayer uses to play Audio

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I change the drive that MediaPlayer uses to play Audio CD's?

Answer:

Change the filename property to point to the root directory of the cd drive you wish to play the cd from.

Example:

```
MediaPlayer1.FileName := 'E:';
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15830: Creating a delay without a timer.

Question and Answer Database

FAQ830D.txt Creating a delay without a timer.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I create a delay without using a TTimer component, and without locking up the system?

Answer:

Here is an example of using a loop that checks the time and calls `Application.ProcessMessages` to allow Windows messages to be processed during the loops execution.

```
procedure Delay(ms : longint);
var
  TheTime : LongInt;
begin
  TheTime := GetTickCount + ms;

  while GetTickCount < TheTime do
    Application.ProcessMessages;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Start Test');
  Delay(2000);
  ShowMessage('End Test');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15836: Converting a color to its VCL string value

Question and Answer Database

FAQ836D.txt    Converting a color to its VCL string value

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I convert a color to its VCL string value?

Answer:

The graphics.pas file has a ColorToString() function that will convert a valid TColor value to it's string representation using a predefined constant (if possible) or a hexadecimal string. The StringToColor() function provides the reciprocal function, taking a predefined color constant string or hexadecimal string and returning the TColor value.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Mem1.Lines.Add(ColorToString(clRed));
  Mem1.Lines.Add(IntToStr(StringToColor('clRed')));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15842: Using escape codes with the format() function

Question and Answer Database

FAQ842D.txt Using escape codes with the format() function

Category :VCL

Platform :All

Product :All 32 bit

Question:

I want to insert escape codes when using the Format() function. I'm trying to create a tab-delimited string. In "C" I could do something like `sprintf(buffer, "%s\t%s", str);`. How might I accomplish this in Pascal?

Answer:

The Pascal Format function does not use escape codes. Instead, you should insert the actual characters into the string using their ASCII representations:

Note that embedded character codes must be placed outside of the quotes.

Examples:

```
Buffer := Format('%s'#9's', [Str1, Str2]);
```

```
ShowMessage(Format('%s'#9's', ['Column1', 'Column2']));
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15848: Aligning maximized forms

Question and Answer Database

FAQ848D.txt     Aligning maximized forms

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

When I start my form maximized, it overlaps the task bar, and does not align with the top of the screen. How can I get around this?

Answer:

This can happen when the form's position is set to `poScreenCenter`. To get the proper results, set the form's position to `poDefault` in the object inspector.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15854: Working with Floating Point, Double, Single, Real, Extended types

Question and Answer Database

FAQ854D.txt Working with Floating Point, Double, Single, Real, Extended types

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

I have two floating point values that are the same, yet when I compare them using " if d1 = d2 then .. " or " if d1 <> d2 then .. " the results are incorrect. Is this a bug?

Answer:

No. Unlike integers, IEEE floating point numbers are only approximates, not exact numbers. You should never use = or <> to compare two floating point numbers. Instead, subtract the two numbers and compare them against a very small number. Example:

```
if abs(d1-d2) < 0.00001 then  
ShowMessage('D1 and D2 are equal');
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15860: 'Tried to search marked block but is invalid' error

Question and Answer Database

FAQ860D.txt 'Tried to search marked block but is invalid' error

Category :IDE

Platform :All

Product :All 32 bit

### Question:

When I search for text using the find dialog in the IDE, I receive the error 'Tried to search marked block but is invalid'. How can I get around this?

### Answer:

You have the Search Selected Text option turned on, but do not have a text block selected. In the find dialog, select "Global", instead of "Selected text" in the "Scope" setting.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15866: Displaying the first frame of an AVI File

Question and Answer Database

FAQ866D.txt    Displaying the first frame of an AVI File

Category       :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I display the first frame of an AVI File?

Answer:

The following demonstrates a hard core technique to get the MediaPlayer to reliably show the first frame of an AVI file.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.ProcessMessages;
  MediaPlayer1.Open;
  Application.ProcessMessages;
  MediaPlayer1.Step;
  Application.ProcessMessages;
  MediaPlayer1.Previous;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15872: How can I retrieve a list of assigned properties

Question and Answer Database

FAQ872D.txt How can I retrieve a list of assigned properties

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I easily retrieve a list of assigned properties and events for a given component in text form?

Answer:

Select a component on the form and select copy from the Edit menu. The text representation of the component and its non-default properties can now be pasted into the code window in Delphi, or any text editor.

Example:

```
object MediaPlayer1: TMediaPlayer
  Left = 48
  Top = 32
  Width = 253
  Height = 30
  Cursor = crHelp
  ColoredButtons =
    [btPlay, btStop, btNext, btPrev, btStep, btBack, btRecord, btEject]
  ParentShowHint = False
  ShowHint = True
  TabOrder = 2
  OnClick = MediaPlayer1Click
end
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15878: Save and load metafiles in a BLOB field without using DBImage

Question and Answer Database

FAQ878D.txt     Save and load metafiles in a BLOB field without using DBImage

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I save and load metafiles in a BLOB field without using the DBImage component?

Answer:

The following example demonstrates saving metafile images to a database as they exist on the disk, preserving any Placeable metafile headers that may be present in the Metafile. The image is displayed from the database in a TImage component.

Example:

```
unit Unit1;
```

```
interface
```

```
{ $IFDEF WIN32 }
```

```
  uses
```

```
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
    Dialogs, StdCtrls, ExtCtrls, DBCtrls, Grids, DBGrids, Db,  
    DBTables;
```

```
{ $ELSE }
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs, DBTables, DB, Grids, DBGrids, ExtCtrls, StdCtrls;
```

```
{ $ENDIF }
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    Table1: TTable;
```

```
    DataSource1: TDataSource;
```

```

DBGrid1: TDBGrid;
Image1: TImage;
Button1: TButton;
Table1Name: TStringField;
Table1WMF: TBlobField;
OpenDialog1: TOpenDialog;
procedure FormCreate(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure DataSource1DataChange(Sender: TObject; Field: TField);
private
  { Private declarations }
  FileName : string; {Used to hold a temp file name}
  procedure LoadWMFFromDatabase; {loads a WMF from the database}
public
  { Public declarations }
end;

```

```

var
  Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  {Used for loading metafiles}
  OpenDialog1.Filter := 'Metafiles (*.wmf)|*.wmf';
  OpenDialog1.Options := [ofHideReadOnly, ofNoChangeDir];
  Image1.Stretch := true;
end;

```

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
  {Erase the temp file if it exists}
  if FileName <> '' then
    DeleteFile(FileName);
end;

```

```

{This function gets a temporary file name form the system}
function GetTemporaryFileName : string;
{$IFDEF WIN32}
  const MAX_PATH = 144;
{$ENDIF}

```



```

var
  {$IFDEF WIN32}
    lpPathBuffer : PChar;
  {$ENDIF}
  lpbuffer : PChar;
begin
  {Get the file name buffer}
  GetMem(lpBuffer, MAX_PATH);
  {$IFDEF WIN32}
  {Get the temp path buffer}
  GetMem(lpPathBuffer, MAX_PATH);
  {Get the temp path}
  GetTempPath(MAX_PATH, lpPathBuffer);
  {Get the temp file name}
  GetTempFileName(lpPathBuffer,
                  'tmp',
                  0,
                  lpBuffer);
  {Free the temp path buffer}
  FreeMem(lpPathBuffer, MAX_PATH);
  {$ELSE}
  {Get the temp file name}
  GetTempFileName(GetTempDrive('C'),
                  'tmp',
                  0,
                  lpBuffer);
  {$ENDIF}
  {Create a pascal string containing}
  {the temp file name and return it}
  result := StrPas(lpBuffer);
  {Free the file name buffer}
  FreeMem(lpBuffer, MAX_PATH);
end;

procedure TForm1.LoadWMFFromDatabase;
var
  FileStream: TFileStream; {a temp file}
  BlobStream: TBlobStream; {the WMF Blob}
begin
  Image1.Picture.Metafile.Assign(nil);
  {Create a blob stream for the WMF blob}
  BlobStream := TBlobStream.Create(Table1WMF, bmRead);
  if BlobStream.Size = 0 then begin
    BlobStream.Free;
    Exit;
  end;
end;

```

```

end;
{if we have a temp file then erase it}
if FileName <> '' then
  DeleteFile(FileName);
{Get a temp file name}
FileName := GetTemporaryFileName;
{Create a temp file stream}
FileStream := TFileStream.Create(FileName,
                                fmCreate or fmOpenWrite);
{Copy the blob to the temp file}
FileStream.CopyFrom(BlobStream, BlobStream.Size);
{Free the streams}
FileStream.Free;
BlobStream.Free;
{Display the image}
Image1.Picture.Metafile.LoadFromFile(FileName);
end;

{Save a wmf file to the database}
procedure TForm1.Button1Click(Sender: TObject);
var
  FileStream: TFileStream; {to load the wmf file}
  BlobStream: TBlobStream; {to save to the blob}
begin
  {Allow the button to repaint}
  Application.ProcessMessages;
  if OpenFileDialog1.Execute then begin
    {Turn off the button}
    Button1.Enabled := false;
    {Assign the avi file name to read}
    FileStream := TFileStream.Create(OpenDialog1.FileName,
                                    fmOpenRead);

    Table1.Edit;
    {Create a BlobStream for the field Table1WMF}
    BlobStream := TBlobStream.Create(Table1WMF, bmReadWrite);
    {Seek to the Beginning of the stream}
    BlobStream.Seek(0, soFromBeginning);
    {Delete any data that may be there}
    BlobStream.Truncate;
    {Copy from the FileStream to the BlobStream}
    BlobStream.CopyFrom(FileStream, FileStream.Size);
    {Free the streams}
    FileStream.Free;
    BlobStream.Free;
    {Post the record}

```

```
Table1.Post;  
{Load the metafile in to a TImage}  
LoadWMFFromDatabase;  
{Enable the button}  
Button1.Enabled := true;  
end;  
end;
```

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:  
TField);  
begin  
  if (Sender as TDataSource).State = dsBrowse then  
    LoadWMFFromDatabase;  
end;
```

end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15884: How can I test if a given string contains a valid number?

Question and Answer Database

FAQ884D.txt    How can I test if a given string contains a valid number?

Category       :Object Pascal

Platform        :All

Product         :All 32 bit

Question:

How can I test if a given string contains a valid number?

Answer:

Use one of the string to number function inside a try except block, and test for an exception.

Example:

```
function IsStrANumber(NumStr : string) : bool;
begin
    result := true;
    try
        StrToInt(NumStr);
    except
        result := false;
    end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if IsStrANumber(Edit1.Text) then
        ShowMessage(Edit1.Text) else
        ShowMessage('Not a Number');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15890: Creating a palette with more than one entry

Question and Answer Database

FAQ890D.txt     Creating a palette with more than one entry  
Category        :Windows API  
Platform        :All  
Product         :All 32 bit

### Question:

The RTL defines the TLogPalette as having only one palette entry. How can I create a palette with more than one entry?

### Answer:

The RTL defines the TLogPalette as having only one entry to allow you to dynamically create a palette with the number of palette entries you wish. To access the additional entries, range checking must be turned off (this can be done in code), and you must access the palette entry index using a variable.

The following example demonstrates dynamically creating a logical palette structure with 256 entries, turns off range checking (if it is on), accesses the palette entries by filling them with gray shades, turns range checking back on (if it was off), creates a handle to palette, and then cleans up the allocated resources.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  lpPalette : PLogPalette;
  hPal      : hPalette;
  i         : integer;
begin
  {Allocate the memory used by the palette}
  GetMem(lpPalette,
         sizeof(TLogPalette) + (255 * sizeof(TPaletteEntry)));

  {Fill out the palette header}
  lpPalette^.palVersion := $300;
```

```

lpPalette^.palNumEntries := 256;

{Turn range checking off if it is on and}
{remember the range checking state}
{$IFOPT R+}
  {$DEFINE CKRANGE}
  {$R-}
{$ENDIF}

{Fill in the palette structure color table with shades of gray}
for i := 0 to 255 do begin
  lpPalette^.PalPalEntry[i].peRed := i;
  lpPalette^.PalPalEntry[i].peGreen := i;
  lpPalette^.PalPalEntry[i].peBlue := i;
end;

{Turn range checking back on if it was on when we started}
{$IFDEF CKRANGE}
  {$UNDEF CKRANGE}
  {$R+}
{$ENDIF}

{Create a palette handle}
hPal := CreatePalette(lpPalette^);

{Free the memory use by the palette structure}
FreeMem(lpPalette,
        sizeof(TLogPalette) + (255 * sizeof(TPaletteEntry)));

{Do something with the palette here}

{Delete the palette handle after use}
DeleteObject(hPal);
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15896: How can I easily move a Window control at runtime?

Question and Answer Database

FAQ896D.txt How can I easily move a Window control at runtime?

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I easily move a Window control at runtime?

Answer:

The following example shows how to move a window control at runtime by holding down the control key and clicking the left mouse button. This method will not work with non-window controls such as TLabel components.

Example:

```
procedure TForm1.Button1MouseDown(Sender: TObject; Button:
  TMouseButton; Shift: TShiftState; X, Y: Integer);
{$IFDEF WIN32}
var
  pt : TPoint;
{$ENDIF}
begin
  if ssCtrl in Shift then begin
    ReleaseCapture;
    SendMessage(Button1.Handle, WM_SYSCOMMAND, 61458, 0);
{$IFDEF WIN32}
    GetCursorPos(pt);
    SendMessage(Button1.Handle,
      WM_LBUTTONUP,
      MK_CONTROL,
      Longint(pt));
    {$ENDIF}
  end;
end;
```

7/16/98 4:31:28 PM





## Article #15902: Detecting what direction a printer will rotate for a portrait

Question and Answer Database

FAQ902D.txt - Detecting what direction a printer will rotate for a portrait

Category :Printing  
Platform :All-32Bit  
Product :

### Question:

Some printers seems to rotate the portrait orientation to achieve a landscape page differently than others. This seems to affect the direction of a rotated font. How can I tell what direction a printer will rotate output on a landscape page, and if landscape mode is available on a given printer?

### Answer:

The following example demonstrates how to call the Windows API function DeviceCapabilities(), to find out how a portrait page is rotated to achieve a landscape page, enabling you to adjust your output accordingly. Generally, a print driver will rotate a landscape page either 90 degrees, or 270 degrees from its portrait orientation.

### Example:

```
uses Printers;

{$IFDEF WIN32}
function DeviceCapabilitiesA(pDevice : PAnsiChar;
    pPort : PAnsiChar;
    fwCapability : Word;
    pOutput : PAnsiChar;
    DevMode : PDeviceModeA): Integer stdcall;
    external 'winspool.drv' name 'DeviceCapabilitiesA';

function DeviceCapabilitiesW(pDevice : PWideChar;
    pPort : PWideChar;
    fwCapability: Word;
    pOutput: PWideChar;
```

```

                                DevMode: PDeviceModeW): Integer stdcall;
external 'winspool.drv' name 'DeviceCapabilitiesW';

function DeviceCapabilities(pDevice : PChar;
                            pPort : PChar;
                            fwCapability : Word;
                            pOutput : PChar;
                            DevMode: PDeviceMode): Integer stdcall
external 'winspool.drv' name 'DeviceCapabilitiesA';
{$ENDIF}

procedure TForm1.Button1Click(Sender: TObject);
var
  Device : array[0..255] of char;
  Driver : array[0..255] of char;
  Port   : array[0..255] of char;
  hDMode : THandle;
  Rotation : integer;
begin
  if PrintDialog1.Execute then begin
    {Reset the printer}
    Printer.PrinterIndex := Printer.PrinterIndex;
    Printer.GetPrinter(Device, Driver, Port, hDMode);
    Rotation :=
      DeviceCapabilities(Device,
                        Port,
                        DC_ORIENTATION,
                        nil,
                        nil);

    case Rotation of
      0 : ShowMessage('This device does not ' +
                     'support landscape');
      90 : ShowMessage('This device rotates the portrait ' +
                      'orientation 90 degrees ' +
                      'to support landscape');
      270 : ShowMessage('This device rotates the portrait ' +
                       'orientation 270 degrees ' +
                       'to support landscape');

    end;
  end;
end;

```

Last Modified: 01-SEP-99

## Article #15908: Removing a forms caption bar?

Question and Answer Database

FAQ908D.txt Removing a forms caption bar?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I get rid of a forms caption bar?

Answer:

Reset the form style, then reset the height of the form. Note: You will need to add a method to terminate the application, as there will no longer be title bar buttons or a system menu.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    SetWindowLong(Form1.Handle,
                  GWL_STYLE,
                  GetWindowLong(Handle,GWL_STYLE) and not WS_CAPTION);
    Height := ClientHeight;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



```

        nil,
        256,
        @a);
{$ELSE}
  if Escape(Dc,
            QUERYESCSUPPORT,
            sizeof(TestInt),
            @TestInt,
            Nil) > 0 then begin
    Err := Escape(Dc,
                  GetTechnology,
                  0,
                  nil,
                  @a);
  {$ENDIF}
    if Err > 0 then begin
      s := UpperCase(StrPas(a));
      if Pos('POSTSCRIPT', s) > 0 then
        result := true;
    end;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if PrintDialog1.Execute then
    if IsPostScriptPrinter(Printer.Handle) then
      ShowMessage('This is a Postscript Printer') else
      ShowMessage('This is not a Postscript Printer');
end;

```

4/2/99 11:21:29 AM

Last Modified: 01-SEP-99

## Article #15919: How can I create a dynamic array of TPoints to draw a polygon?

Question and Answer Database

FAQ919D.txt    How can I create a dynamic array of TPoints to draw a polygon?

Category       :Object Pascal

Platform       :All

Product        :All 32 bit

Question:

How can I create a dynamic array of TPoints to draw a polygon?

Answer:

The easist way is to declare a type of array that has only one element, turn range checking off, access the array through the use of a variable, and call the Windows API function Polygon().

Example:

```
type
  TPtArray = array[0..0] of TPoint;
  PPtArray = ^TPtArray;

procedure DrawDynamicPolyArray(NumPoints : integer;
                               Canvas : TCanvas);

var
  p : PPtArray;
  i : integer;
begin
  {$IFOPT R+}
  {$DEFINE CKRANGE}
  {$R-}
  {$ENDIF}
  GetMem(p, sizeof(TPoint) * NumPoints);
  Randomize;
  for i := 0 to (NumPoints -1) do begin
    p^[i].x := Random(Form1.Width);
    p^[i].y := Random(Form1.Height);
  end;
  Polygon(Canvas.Handle, p, NumPoints);
  FreeMem(p, sizeof(TPoint) * NumPoints);
```

```
{ $IFDEF CKRANGE }  
  { $UNDEF CKRANGE }  
  { $R+ }  
{ $ENDIF }  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Form1.Canvas.Brush.Color := clWhite;  
  Form1.Canvas.FillRect(Rect(0, 0, Form1.Width, Form1.Height));  
  Form1.Canvas.Brush.Color := clRed;  
  DrawDynamicPolyArray(SpinEdit1.Value,  
                        Form1.Canvas);  
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  SpinEdit1.MinValue := 2;  
  SpinEdit1.MaxValue := 1000;  
  SpinEdit1.Value := 100;  
end;
```

end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15924: How can I prevent the user from moving or sizing my form?

Question and Answer Database

FAQ924D.txt    How can I prevent the user from moving or sizing my form?

Category     : Object Pascal

Platform     : All

Product      : All 32 bit

Question:

How can I prevent the user from moving or sizing my form?

Answer:

Trap the Windows WM\_WINDOWPOSCHANGING message and "or" the flags of the WindowPos structure passed in the message's lParam parameter with the predefined constants SWP\_NOMOVE and SWP\_NOSIZE.

Example:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
    procedure WMPosChange(var Message: TWMWINDOWPOSCHANGING);
      message WM_WINDOWPOSCHANGING;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMPosChange(var Message: TWMWINDOWPOSCHANGING);
begin
  PWindowPos(TMessage(Message).lParam).Flags :=
    PWindowPos(TMessage(Message).lParam).Flags or
    SWP_NOMOVE or
```

SWP\_NOSIZE;

end;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15930: Detecting if the PrintScreen key has been pressed

Question and Answer Database

FAQ930D.txt Detecting if the PrintScreen key has been pressed

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I tell if the PrintScreen key has been pressed?

Answer:

The PrintScreen system key is not processed during the TForm keydown event. The following example tests if the PrintScreen key has been pressed by calling the Windows API function GetAsyncKeyState() during the Application.OnIdle event.

Example:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    procedure AppIdle(Sender: TObject; var Done: Boolean);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnIdle := AppIdle;
end;
```

```
procedure TForm1.AppIdle(Sender: TObject; var Done: Boolean);
begin
  if GetAsyncKeyState(VK_SNAPSHOT) <> 0 then
    Form1.Caption := 'SnapShot';
  Done := True;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15936: Setting the MaxPage property of the PrintDialog component

Question and Answer Database

FAQ936D.txt     Setting the MaxPage property of the PrintDialog component

Category     : Object Pascal

Platform     : All

Product      : All 32 bit

Question:

When I set the MaxPage property of the PrintDialog component, the VCL always adds one to the number of pages shown in the dialog, and after execution, the Max page property does not change. Is this a bug?

Answer:

The VCL simply wraps the Windows common print dialog. You should fill in all the fields before executing the dialog, and after execution. Check the PrintDialog's FromPage and ToPage properties for the pages to print.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  PrintDialog1.Options := [poPageNums];
  PrintDialog1.PrintRange := prPageNums;
  PrintDialog1.MinPage := 1;
  PrintDialog1.MaxPage := 5;
  PrintDialog1.FromPage := 1;
  PrintDialog1.ToPage := 5;
  if PrintDialog1.Execute then begin
    ShowMessage(IntToStr(PrintDialog1.FromPage));
    SdshowMessage(IntToStr(PrintDialog1.ToPage));
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15942: Trap scrolling messages for the ScrollBars of a TScrollBar

Question and Answer Database

FAQ942D.txt    Trap scrolling messages for the ScrollBars of a TScrollBar

Category       :VCL  
Platform       :All  
Product        :All 32 bit

Question:

How can I trap the scrolling messages for the ScrollBars of a TScrollBar component?

Answer:

The following example traps the scrolling messages for a TScrollBar component, keeping the two scroll bars in sync. If either of the scroll bars are scrolled, then the position of the other scroll bar is changed to the same value. The scroll messages are trapped by subclassing the scroll box's WinProc.

Example:

```
type
{$IFDEF WIN32}
    WParameter = LongInt;
{$ELSE}
    WParameter = Word;
{$ENDIF}
    LParameter = LongInt;

{Declare a variable to hold the window procedure we are replacing}
var
    OldWindowProc : Pointer;

function NewWindowProc(WindowHandle : hWnd;
                        TheMessage   : WParameter;
                        ParamW       : WParameter;
                        ParamL       : LParameter) : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
var
```

```

TheRangeMin : integer;
TheRangeMax : integer;
TheRange : integer;
begin

if TheMessage = WM_VSCROLL then begin
{Get the min and max range of the horizontal scroll box}
  GetScrollRange(WindowHandle,
                  SB_HORZ,
                  TheRangeMin,
                  TheRangeMax);
{Get the vertical scroll box position}
  TheRange := GetScrollPos(WindowHandle,
                           SB_VERT);
{Make sure we wont exceed the range}
  if TheRange < TheRangeMin then
    TheRange := TheRangeMin else
  if TheRange > TheRangeMax then
    TheRange := TheRangeMax;
{Set the horizontal scroll bar}
  SetScrollPos(WindowHandle,
               SB_HORZ,
               TheRange,
               true);

end;

if TheMessage = WM_HSCROLL then begin
{Get the min and max range of the horizontal scroll box}
  GetScrollRange(WindowHandle,
                  SB_VERT,
                  TheRangeMin,
                  TheRangeMax);
{Get the horizontal scroll box position}
  TheRange := GetScrollPos(WindowHandle,
                           SB_HORZ);
{Make sure we wont exceed the range}
  if TheRange < TheRangeMin then
    TheRange := TheRangeMin else
  if TheRange > TheRangeMax then
    TheRange := TheRangeMax;
{Set the vertical scroll bar}
  SetScrollPos(WindowHandle,
               SB_VERT,
               TheRange,
               true);

```

```

end;

{ Call the old Window procedure to }
{ allow processing of the message. }
NewWindowProc := CallWindowProc(OldWindowProc,
                                WindowHandle,
                                TheMessage,
                                ParamW,
                                ParamL);

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
{ Set the new window procedure for the control }
{ and remember the old window procedure.      }
OldWindowProc := Pointer(SetWindowLong(ScrollBar1.Handle,
                                       GWL_WNDPROC,
                                       LongInt(@NewWindowProc)));
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
{ Set the window procedure back }
{ to the old window procedure.  }
SetWindowLong(ScrollBar1.Handle,
              GWL_WNDPROC,
              LongInt(OldWindowProc));
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #15948: Embedding a page break in the Rich Edit control

Question and Answer Database

FAQ948D.txt Embedding a page break in the Rich Edit control

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I embed a page break in the Rich Edit control?

Answer:

You cannot. The /Page command is removed by the RichEdit control.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15954: How can I disable the Delphi / Borland C++ Builder splash screen

Question and Answer Database

FAQ954D.txt    How can I disable the Delphi / Borland C++ Builder splash screen

Category       :IDE

Platform        :All

Product         :All 32 bit

Question:

Q) How can I disable the Delphi / Borland C++ Builder splash screen?

Answer:

A) Start the main Delphi / Borland C++ Builder passing "-ns" (without the quotes) on the command line.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15960: adjusting the tab location in a TMemo component

Question and Answer Database

FAQ960D.txt adjusting the tab location in a TMemo component

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I adjust the tab location in a TMemo component?

Answer:

Send a EM\_SETTABSTOPS message to the Memo control. The following example sets the first 5 tab stops to 20 pixels.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  DialogUnitsX : LongInt;
  PixelsX : LongInt;
  i : integer;
  TabArray : array[0..4] of integer;
begin
  Mem1.WantTabs := true;
  DialogUnitsX := LoWord(GetDialogBaseUnits);
  PixelsX := 20;
  for i := 1 to 5 do begin
    TabArray[i - 1] :=
      ((PixelsX * i ) * 4) div DialogUnitsX;
  end;
  SendMessage(Mem1.Handle,
              EM_SETTABSTOPS,
              5,
              LongInt(@TabArray));
  Mem1.Refresh;
end;
```



## Article #15966: The TDateTime component commonly ignores key input

Question and Answer Database

FAQ966D.txt    The TDateTime component commonly ignores key input  
Category     :Object Pascal  
Platform     :All  
Product      :All 32 bit

### Question:

The TDateTime component commonly ignores key input. For example, when I key in "49" in the minute field, it inputs a "4" then a "9", when I clearly want "49". Is this a bug?

### Answer:

No. The VCL simply wraps the Windows DateTime control. To input a multiple digit number, quickly enter the first number twice, followed by the additional numbers you wish to input. In the case of "49", enter "449"

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15972: Color reduction techniques for graphics.

Question and Answer Database

FAQ972D.txt Color reduction techniques for graphics.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I accurately (and automatically) display HiColor and True Color (16 and 24 bit) images on a 256 color display?

Answer:

Generally, you cannot. This requires sophisticated color reduction techniques, and not all will work well on all images. Most all color reduction methods work by weighing the average of similar colored pixels in a given area of the image, and combining them. Consider an image of a person standing in a park with green grass, flowers, trees and a blue sky. You would need to remap potentially millions of colors to fit into the Windows palette (236 colors). While this can usually be done in a professional paint program with some trial and error, a single function would have much trouble remapping the colors to faithfully reproduce all images. This is the reason why paint programs that can accurately reduce colors give so many choices in the method used to reduce the color.

In the above example, you could combine many shades of blue and green, and dedicate more colors to faithfully reproduce the person's flesh tones. This method would probably produce an acceptable image, since the shades of the sky, grass, and trees are far less important than the shades necessary to reproduce flesh tones. On the other hand, if the purpose of the image was to show a spectacular panorama of a canyon, you might want to put less emphasis on the exact color of a house that might be in the picture, and instead concentrate on the colors of the canyon. As you can see, where one color reduction function may work well on one image, it would most likely have trouble with others.

Some paint programs use a "Color Wash" palette, that sometimes can display with fair accuracy a colorful 24 bit photograph. This is done by creating a palette that contains a wash of colors that are generally useful for displaying most images. Again, pixels are compared by area, and remapped to a color that is reasonably close to the original if possible.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15978: Changing the font style of a dbgrid row

Question and Answer Database

FAQ978D.txt Changing the font style of a dbgrid row

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How can I change the font style of one particular row in a dbgrid (one record)?

Answer:

Paste the code below into the OnDrawDataCell event of the dbgrid.

Note: The code below uses the DBDEMOS table "CUSTOMER.DB", a TDBGrid, a TDataSource and a TTable.

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect:
TRect;
  Field: TField; State: TGridDrawState);
begin
// If the record's CustNo is 1351 draw the entire row with a
// line through it. (set the font style to strike out)
  if (Sender as TDBGrid).DataSource.DataSet.FieldByName('CustNo').
AsString = '1351' then
    with (Sender as TDBGrid).Canvas do begin
      FillRect(Rect);
      // Set the font style to StrikeOut
      Font.Style := Font.Style + [fsStrikeOut];
      // Draw the cell right aligned for floats + offset
      if (Field.DataType = ftFloat) then
        TextOut(Rect.Right-TextWidth(Field.AsString)-3,
          Rect.Top+3, Field.AsString)
      // Otherwise draw the cell left aligned + offset
      else
        TextOut(Rect.Left+2,Rect.Top+3,Field.AsString);
    end;
end;
```



7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15984: Get a count and list all the controls on a TNoteBook

Question and Answer Database

FAQ984D.txt    Get a count and list all the controls on a TNoteBook  
Category       :VCL  
Platform        :All  
Product         :All 32 bit

Question:

How can I get a count and list all the controls on a TNoteBook component?

Answer:

The following example demonstrates getting a list of the pages in a TNotebook component, and listing the controls on each page of the Notebook component. The list is added into a Listbox component.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  n: integer;
  p: integer;
begin
  ListBox1.Clear;
  with Notebook1 do begin
    for n := 0 to ControlCount - 1 do begin
      with TPage(Controls[n]) do begin
        ListBox1.Items.Add('Notebook Page: ' +
                           TPage(Notebook1.Controls[n]).Caption);
        for p := 0 to ControlCount - 1 do
          ListBox1.Items.Add(Controls[p].Name);
        ListBox1.Items.Add(EmptyStr);
      end;
    end;
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15990: Changing the background color of Text

Question and Answer Database

FAQ990D.txt Changing the background color of Text

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I change the background color of Text?

Answer:

Use the Windows API functions SetBkColor() and TextOut().

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldTextColor : TColorRef;
  OldBkColor   : TColorRef;
  OldBkMode    : Integer;
begin
  OldTextColor := SetTextColor(Form1.Canvas.Handle, RGB(0, 0, 255));
  OldBkColor   := SetBkColor(Form1.Canvas.Handle, RGB(255, 0, 0));
  OldBkMode    := SetBkMode(Form1.Canvas.Handle, OPAQUE);
  TextOut(Form1.Canvas.Handle,
           100, 100,
           'Blue text on red Background',
           27);
  SetBkMode(Form1.Canvas.Handle, OldBkMode);
  SetBkColor(Form1.Canvas.Handle, OldBkColor);
  SetTextColor(Form1.Canvas.Handle, OldTextColor);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #15996: Displaying a 24 bit True Color bitmap image on a 256 color display

Question and Answer Database

FAQ996D.txt    Displaying a 24 bit True Color bitmap image on a 256 color display

Category       :VCL

Platform       :All

Product        :All 32 bit

Question:

How can I display a 24 bit True Color bitmap image on a 256 color Display under Delphi 3?

Answer:

You can take advantage of the new graphics capabilities of the TBitmap and TJpegImage components of Delphi 3. When Delphi 3 loads a bitmap image, it keeps a copy of the device independent bitmap image it loads from a file in the background. The TJPEGImage component is very good at color reducing a full color image down to 256 colors. By Loading the bitmap, then assigning the image to a Jpeg and saving it to a temporary ".JPG" file, you can then load the temporary file back into a TImage with much better results than simply loading the bitmap file unconverted. The following example demonstrates the necessary steps to achieve these results.

Example:

```
uses JPEG;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  JP : TJPEGImage;
```

```
  IM : TImage;
```

```
  TempFileName : string;
```

```
begin
```

```
  {Pop up a Open Dialog}
```

```
  OpenDialog1.Options := [ofNoChangeDir, ofFileMustExist];
```

```
  OpenDialog1.Filter := 'Bitmap Files (*.bmp)|*.bmp';
```

```
  if OpenDialog1.Execute then begin
```

```
    {Create a temporary TImage}
```

```
    IM := TImage.Create(nil);
```

```
{Load the bitmap file}
  IM.Picture.LoadFromFile(OpenDialog1.FileName);
{Create a temporary TJPEGImage}
  JP := TJPEGImage.Create;
{Priority on quality}
  JP.Performance := jpBestQuality;
{Assign the bitmap to the JPEG}
  JP.Assign(IM.Picture.Graphic);
{Free the temp image}
  IM.Free;
{Make a temp file name with the extension of .jpg}
  TempFileName := 'test.jpg';
{Save the JPEG to a temp file}
  JP.SaveToFile(TempFileName);
{Free the JPEG}
  JP.Free;
{Load the temp file to an image on the form}
  Image1.Picture.LoadFromFile(TempFileName);
{Delete the temp file}
  DeleteFile(TempFileName);
end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16002: Data segment too large error.

Question and Answer Database

FAQ1002D.txt Data segment too large error.

Category :Compiler

Platform :All

Product :All 32 bit

Question:

When I add a large number of typed constants to my application, my data segment grows too large. How can I get around this?

Answer:

Typed constants are added to the Program's data segment. Untyped constants are added to the code segment of the application only if they are used. Unless you are going to reassign the value of a typed constant, you should use an untyped constant instead. Typed constants should be replaced with initialized variables to be compatible with future versions of the compiler.

Example:

```
{This adds the constant to the data segment}
const SomeTypedString : string = 'Test';
const SomeTypedInt : integer = 5;
```

```
{This adds the constant to the code segment if used}
const SomeUnTypedString = 'Test';
const SomeUnTypedInt = 5;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16008: Turning an low power compliant monitor off?

Question and Answer Database

FAQ1008D.txt Turning an low power compliant monitor off?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I turn an low power compliant monitor off?

Answer:

You can send a `wm_SysCommand` message with the `WParam` parameter set to `SC_MonitorPower` and the `LParam` parameter set to 0 to turn the monitor off, and -1 to turn the monitor back on. The following example turns the monitor off for a 10 second interval.

Example:

type

```
TForm1 = class(TForm)
  Button1: TButton;
  Timer1: TTimer;
  procedure FormCreate(Sender: TObject);
  procedure Timer1Timer(Sender: TObject);
  procedure Button1Click(Sender: TObject);
private
  { Private declarations }
public
  MonitorOff : bool;
  { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```



```
begin
  Timer1.Enabled := false;
  Timer1.Interval := 10000;
  MonitorOff := false;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if MonitorOff then begin
    MonitorOff := false;
    SendMessage(Application.Handle,
      wm_SysCommand,
      SC_MonitorPower,
      -1);
    Timer1.Enabled := false;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  MonitorOff := true;
  Timer1.Enabled := true;
  SendMessage(Application.Handle,
    wm_SysCommand,
    SC_MonitorPower,
    0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16019: Intercepting the Ctrl-V key in a TMemo

Question and Answer Database

FAQ1019D.txt Intercepting the Ctrl-V key in a TMemo

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I intercept the Ctrl-V key in a TMemo component and paste custom text that is not currently on the clipboard?

Answer:

The following example demonstrates trapping the Ctrl-V key in a TMemo component.

Example:

```
uses ClipBrd;
```

```
procedure TForm1.Memo1KeyDown(Sender: TObject; var Key: Word;  
  Shift: TShiftState);
```

```
begin
```

```
  if ((Key = ord('V')) and (ssCtrl in Shift)) then begin
```

```
    if Clipboard.HasFormat(CF_TEXT) then
```

```
      Clipboard.Clear;
```

```
      Memo1.SelText := 'Delphi is RAD!';
```

```
      key := 0;
```

```
    end;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16025: Subclass the window procedure for a TForm

Question and Answer Database

FAQ1025D.txt      Subclass the window procedure for a TForm

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I subclass the window procedure for a TForm?

Answer:

Override the WinProc procedure for the form. The following example overrides the window procedure for the form, and responds to the WM\_CANCELMODE message, indicating that a messagebox or other dialog box has been executed.

Example:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure WndProc (var Message: TMessage); override;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WndProc (var Message: TMessage);
begin
  if Message.Msg = WM_CANCELMODE then begin
    Form1.Caption := 'A dialog or message box has popped up';
```

```
end else
  inherited
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Test Message');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16031: Does Delphi have an equivalent for the VB function DoEvents?

Question and Answer Database

FAQ1031D.txt Does Delphi have an equivalent for the VB function DoEvents?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

Does Delphi have an equivalent for the VB function DoEvents?

Answer:

Yes, use Application.ProcessMessages.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16037: How can I capture an image from a video source?

Question and Answer Database

FAQ1037D.txt     How can I capture an image from a video source?

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How can I capture an image from a video source?

Answer:

For a complete example, you need to get the Microsoft Video for Windows SDK. The following simple example shows how to open the default video capture device, grab a frame from the device, save a frame from the device to the disk as a .BMP (device independent bitmap) file, record a .AVI file (with sound, but without preview), and close the device. Note: You must have a video capture device installed for this example to work.

Example:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, ExtCtrls, StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)  
    Panell: TPanel;  
    OpenVideo: TButton;  
    CloseVideo: TButton;  
    GrabFrame: TButton;  
    SaveBMP: TButton;  
    StartAVI: TButton;  
    StopAVI: TButton;  
    SaveDialog1: TSaveDialog;  
    procedure FormCreate(Sender: TObject);
```

```
procedure OpenVideoClick(Sender: TObject);
procedure CloseVideoClick(Sender: TObject);
procedure GrabFrameClick(Sender: TObject);
procedure SaveBMPClick(Sender: TObject);
procedure StartAVIClick(Sender: TObject);
procedure StopAVIClick(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
hWndC : THandle;
```

```
CapturingAVI : bool;
```

```
public
```

```
{ Public declarations }
```

```
end;
```

```
var
```

```
Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
const WM_CAP_START           = WM_USER;
const WM_CAP_STOP           = WM_CAP_START + 68;
const WM_CAP_DRIVER_CONNECT = WM_CAP_START + 10;
const WM_CAP_DRIVER_DISCONNECT = WM_CAP_START + 11;
const WM_CAP_SAVEDIB        = WM_CAP_START + 25;
const WM_CAP_GRAB_FRAME     = WM_CAP_START + 60;
const WM_CAP_SEQUENCE       = WM_CAP_START + 62;
const WM_CAP_FILE_SET_CAPTURE_FILEA = WM_CAP_START + 20;
```

```
function capCreateCaptureWindowA(lpszWindowName : PCHAR;
                                dwStyle : longint;
                                x : integer;
                                y : integer;
                                nWidth : integer;
                                nHeight : integer;
                                ParentWin : HWND;
                                nId : integer): HWND;
    STDCALL EXTERNAL 'AVICAP32.DLL';
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    CapturingAVI := false;
```

```
    hWndC := 0;
```

```
    SaveDialog1.Options :=
```

```

    [ofHideReadOnly, ofNoChangeDir, ofPathMustExist]
end;

procedure TForm1.OpenVideoClick(Sender: TObject);
begin
    hWndC := capCreateCaptureWindowA('My Own Capture Window',
                                     WS_CHILD or WS_VISIBLE ,
                                     Panell.Left,
                                     Panell.Top,
                                     Panell.Width,
                                     Panell.Height,
                                     Form1.Handle,
                                     0);

    if hWndC <> 0 then
        SendMessage(hWndC, WM_CAP_DRIVER_CONNECT, 0, 0);
end;

procedure TForm1.CloseVideoClick(Sender: TObject);
begin
    if hWndC <> 0 then begin
        SendMessage(hWndC, WM_CAP_DRIVER_DISCONNECT, 0, 0);
        hWndC := 0;
    end;
end;

procedure TForm1.GrabFrameClick(Sender: TObject);
begin
    if hWndC <> 0 then
        SendMessage(hWndC, WM_CAP_GRAB_FRAME, 0, 0);
end;

procedure TForm1.SaveBMPClick(Sender: TObject);
begin
    if hWndC <> 0 then begin
        SaveDialog1.DefaultExt := 'bmp';
        SaveDialog1.Filter := 'Bitmap files (*.bmp)|*.bmp';
        if SaveDialog1.Execute then
            SendMessage(hWndC,
                        WM_CAP_SAVEDIB,
                        0,
                        longint(pchar(SaveDialog1.FileName)));
    end;
end;

procedure TForm1.StartAVIClick(Sender: TObject);

```



```
begin
  if hWndC <> 0 then begin
    SaveDialog1.DefaultExt := 'avi';
    SaveDialog1.Filter := 'AVI files (*.avi)|*.avi';
    if SaveDialog1.Execute then begin
      CapturingAVI := true;
      SendMessage(hWndC,
        WM_CAP_FILE_SET_CAPTURE_FILEA,
        0,
        Longint(pchar(SaveDialog1.FileName)));
      SendMessage(hWndC, WM_CAP_SEQUENCE, 0, 0);
    end;
  end;
end;
```

```
procedure TForm1.StopAVIClick(Sender: TObject);
```

```
begin
  if hWndC <> 0 then begin
    SendMessage(hWndC, WM_CAP_STOP, 0, 0);
    CapturingAVI := false;
  end;
end;
```

```
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16043: Control Panel Applets

Question and Answer Database

FAQ1043D.txt Control Panel Applets

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

How do I create a Control Panel Applet?

Answer:

Creating a Control Panel Applet is a straight forward process. Simply create a Dynamic Link Library with the extension of .cpl (Control Panel Library) and place it in the Windows system directory. Each cpl file can support multiple control panel applets. The cpl will have a single function entry point called CPLApplet() that must be exported by name. All the following control panel messages will come through this single entry point:

Message:

CPL\_INIT - Sent to indicate CPLApplet() was found. Return TRUE to continue the loading process.

CPL\_GETCOUNT - Return the number of applets supported by the cpl.

CPL\_INQUIRE - Sent for information about each applet supported by the cpl. lParam1 contains the zero based applet number for the inquiry. lParam2 points to a TCplInfo structure. The idIcon field of the TCplInfo structure should be initialized with the resource id for an icon to display, and the idName and idInfo fields should be initialized with the resource string id for the name and description string id. lData can contain applet defined data for use by the applet.

CPL\_SELECT - Sent when the applet's icon has been selected by the user. lParam1 contains the applet number that was selected. lParam2 contains the applet's user defined lData value.

CPL\_DBLCLK - Sent when the applet's icon has been double-clicked. lParam1 contains the applet number. lParam2 contains the applet's

user defined lData value. This message initiates the display of the applet's dialog box.

CPL\_STOP - Sent for each applet when the control panel is exiting. lParam1 contains the applet number. lParam2 contains the applet's user defined lData value. Any applet specific cleaning up should be performed during this call.

CPL\_EXIT - Sent prior to the control panel call to FreeLibrary. Non-applet specific cleaning up should be performed during this call.

CPL\_NEWINQUIRE - Same as CPL\_INQUIRE except lParam2 is a pointer to a NEWCPLINFO structure.

Your control panel library will need some additional resources to function. You will need to create a resource file containing a string table containing both the name and description of your applet(s) and icons for each applet in your cpl. You can create a res file from a .rc (resource script file) using the BRCC.EXE or BRCC32.EXE command line resource compiler, or a WYSIWYG resource editor like Borland's Resource Workshop.

Example .rc file containing a string table with two strings and a pointer to a icon file resource:

```
STRINGTABLE
{
  1, "TestApplet"
  2, "My Test Applet"
}

2 ICON C:\SOMEPATH\CHIP.ICO
```

To compile the .rc file to a .res file that can be linked with your application, simply type on the dos command line the full path to the resource compiler, and the full path to the name of the .rc file to compile. Here is an example:

```
c:\Delphi\Bin\brcc32.exe c:\Delphi\MyRes.rc
```

When the compiler is finished, you should have a new file with the same name as the .rc file you've compiled, only with an extension of ".res".

If you are developing for multiple platforms, you should create both a

16 and 32 bit res file for linkage with your application.

The following is an example of a control panel applet that executes a secondary executable in response to the CPL\_DBLCLK message. You can adapt the code to show a form or dialog box as well. The example is coded to compile for Win16 and Win32 environments.

To build the project, you will need to compile the above .rc file to a res file named either: TCPL32.RES or TCPL16.RES.

Example:

```
library TestCpl;
```

```
{ $IFDEF WIN32 }
```

```
uses
```

```
    SysUtils,
```

```
    Windows,
```

```
    Messages;
```

```
{ $ELSE }
```

```
uses
```

```
    SysUtils,
```

```
    WinTypes,
```

```
    WinProcs,
```

```
    Messages;
```

```
{ $ENDIF }
```

```
{ $IFDEF WIN32 }
```

```
    { $R TCPL32.RES }
```

```
{ $ELSE }
```

```
    { $R TCPL16.RES }
```

```
{ $ENDIF }
```

```
const NUM_APPLETS = 1;
```

```
{ $IFDEF WIN32 }
```

```
    const CPL_DYNAMIC_RES = 0;
```

```
{ $ENDIF }
```

```
const CPL_INIT = 1;
```

```
const CPL_GETCOUNT = 2;
```

```
const CPL_INQUIRE = 3;
```

```
const CPL_SELECT = 4;
```

```
const CPL_DBLCLK = 5;
```

```
const CPL_STOP = 6;
```

```
const CPL_EXIT = 7;
```

```
const CPL_NEWINQUIRE = 8;
{$IFDEF WIN32}
  const CPL_STARTWPARMS = 9;
{$ENDIF}
const CPL_SETUP = 200;
```

```
{$IFNDEF WIN32}
type DWORD = LongInt;
{$ENDIF}
```

```
type TCplInfo = record
  idIcon : integer;
  idName : integer;
  idInfo : integer;
  lData : LongInt;
end;
PCplInfo = ^TCplInfo;
```

```
type TNewCplInfoA = record
  dwSize : DWORD;
  dwFlags : DWORD;
  dwHelpContext : DWORD;
  lData : LongInt;
  IconH : HIcon;
  szName : array [0..31] of char;
  szInfo : array [0..63] of char;
  szHelpFile : array [0..127] of char;
end;
PNewCplInfoA = ^TNewCplInfoA;
```

```
{$IFDEF WIN32}
type TNewCplInfoW = record
  dwSize : DWORD;
  dwFlags : DWORD;
  dwHelpContext : DWORD;
  lData : LongInt;
  IconH : HIcon;
  szName : array [0..31] of WChar;
  szInfo : array [0..63] of WChar;
  szHelpFile : array [0..127] of WChar;
end;
PNewCplInfoW = ^TNewCplInfoW;
{$ENDIF}
```

```
type TNewCplInfo = TNewCplInfoA;
```

```

type PNewCplInfo = ^TNewCplInfoA;

function CPLApplet(hWndCPL : hWnd;
                  iMessage : integer;
                  lParam1 : longint;
                  lParam2 : longint) : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
  case iMessage of
    CPL_INIT : begin
      Result := 1;
      exit;
    end;
    CPL_GetCount : begin
      Result := NUM_APPLETS;
      exit;
    end;
    CPL_Inquire : begin
      PCplInfo(lParam2)^.idIcon := 2;
      PCplInfo(lParam2)^.idName := 1;
      PCplInfo(lParam2)^.idInfo := 2;
      PCplInfo(lParam2)^.lData := 0;
      Result := 1;
      exit;
    end;
    CPL_NewInquire : begin
      PNewCplInfo(lParam2)^.dwSize := sizeof(TNewCplInfo);
      PNewCplInfo(lParam2)^.dwHelpContext := 0;
      PNewCplInfo(lParam2)^.lData := 0;
      PNewCplInfo(lParam2)^.IconH := LoadIcon(hInstance,
                                                MakeIntResource(2));
      lStrCpy(@PNewCplInfo(lParam2)^.szName, 'TestCPL');
      lStrCpy(PNewCplInfo(lParam2)^.szInfo, 'My Test CPL');
      PNewCplInfo(lParam2)^.szHelpFile[0] := #0;
      Result := 1;
      exit;
    end;
    CPL_SELECT : begin
      Result := 0;
      exit;
    end;
    CPL_DBLCLK : begin
      WinExec('Notepad.exe', SW_SHOWNORMAL);
      Result := 1;
      exit;
    end;
  end;
end;

```

```
end;
CPL_STOP : begin
  Result := 0;
  exit;
end;
CPL_EXIT : begin
  Result := 0;
  exit;
end else begin
  Result := 0;
  exit;
end;
end;
end;
end;

exports CPLApplet name 'CPLApplet';

begin
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16049: Auto increment Build Number

Question and Answer Database

FAQ1049D.txt Auto increment Build Number

Category :IDE

Platform :All

Product :Delphi 3.x

Question:

How can I get the Build Number to Auto increment?

Answer:

You must select the Build All option. Simply re-compiling or running the application will not increment the build number.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16055: Avoiding an Application Error when using EM\_POSFROMCHAR

Question and Answer Database

FAQ1055D.txt      Avoiding an Application Error when using EM\_POSFROMCHAR

Category        :VCL

Platform        :All

Product         :Delphi 3.x

Question:

How can I avoid an Application Error when using the EM\_POSFROMCHAR and EM\_POSFROMCHAR in a RichEdit control?

Answer:

The definitions for EM\_CHARFROMPOS and EM\_POSFROMCHAR are different in the RichEdit and Messages units:

RichEdit unit:

```
EM_POSFROMCHAR                = WM_USER + 38;  
EM_CHARFROMPOS                = WM_USER + 39;
```

Messages unit:

```
EM_POSFROMCHAR                = 214;  
EM_CHARFROMPOS                = 215;
```

To avoid confusion, these constants have been removed from the RichEdit unit in Delphi 3.

When using the EM\_POSFROMCHAR message, WParam should contain the address of the point structure that will receive the coordinates, and LParam should contain the zero based offset of the character. The return value is not initialized. Note that the y coordinate can be zero or negative.

When using EM\_CHARFROMPOS, wparam is not used, and LParam should contain the address of the point structure that will receive the coordinates. The return value should contain the required character index.

Last Modified: 01-SEP-99

## Article #16061: Can I use the AssignPrn method to write directly to a port?

Question and Answer Database

FAQ1061D.txt - Can I use the AssignPrn method to write directly to a port?

Category :Printing  
Platform :All-32Bit  
Product :

Question:

Can I use the AssignPrn method to write directly to a port?

Answer:

No, the AssignPrn method of printing actually uses the Windows GDI function TextOut() to write to the print driver.

4/2/99 11:21:38 AM

Last Modified: 01-SEP-99

## Article #16067: CopyCursor

Question and Answer Database

FAQ1067D.txt CopyCursor

Category :Windows API

Platform :All

Product :Delphi 3.x

Question:

Where is the Windows API function CopyCursor defined in 32-Bit versions of Delphi and C++ Builder?

Answer:

While the call is documented in the Win32 help file, the function is simply mapped to the Windows API function CopyIcon(). Note that the 32 bit version of COpy icon does not take a hInstance parameter.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16073: Selection rectangle

Question and Answer Database

FAQ1073D.txt Selection rectangle

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I create a selection rectangle for use with selecting portions of an image for editing?

Answer:

The easiest way to do this is to use the Windows API function DrawFocusRect. The DrawFocusRect function uses the XOR function when drawing, so drawing the rectangle twice in the same location erases the rectangle, and the rectangle will always be visible when drawing over different colors.

Example:

type

```
TForm1 = class(TForm)
  procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
  procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
private
  { Private declarations }
  Capturing : bool;
  Captured : bool;
  StartPlace : TPoint;
  EndPlace : TPoint;
public
  { Public declarations }
end;
```

var

```
Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
function MakeRect(Pt1 : TPoint;  
                  Pt2 : TPoint) : TRect;
```

```
begin  
  if pt1.x < pt2.x then begin  
    Result.Left := pt1.x;  
    Result.Right := pt2.x;  
  end else begin  
    Result.Left := pt2.x;  
    Result.Right := pt1.x;  
  end;  
  if pt1.y < pt2.y then begin  
    Result.Top := pt1.y;  
    Result.Bottom := pt2.y;  
  end else begin  
    Result.Top := pt2.y;  
    Result.Bottom := pt1.y;  
  end;  
end;
```

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
  Shift: TShiftState; X, Y: Integer);
```

```
begin  
  if Captured then  
    DrawFocusRect(Form1.Canvas.Handle,  
                  MakeRect(StartPlace,  
                            EndPlace));  
  
  StartPlace.x := X;  
  StartPlace.y := Y;  
  EndPlace.x := X;  
  EndPlace.y := Y;  
  DrawFocusRect(Form1.Canvas.Handle,  
                MakeRect(StartPlace,  
                          EndPlace));  
  
  Capturing := true;  
  Captured := true;  
end;
```

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,  
  Y: Integer);
```

```
begin
  if Capturing then begin
    DrawFocusRect(Form1.Canvas.Handle,
                  MakeRect(StartPlace,
                          EndPlace));
    EndPlace.x := X;
    EndPlace.y := Y;
    DrawFocusRect(Form1.Canvas.Handle,
                  MakeRect(StartPlace,
                          EndPlace));
  end;

end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Capturing := false;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16079: Draw on frame

Question and Answer Database

FAQ1079D.txt Draw on frame

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to draw on the frame of a form?

Answer:

Create a message handler for the Windows Message WM\_NCPAINT message. The following example paints a 1 pixel red border around the frame of the form.

Example:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
    procedure WMNCPaint(var Msg : TWMNCPaint); message WM_NCPAINT;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMNCPaint(var Msg: TWMNCPaint);
var
  dc : hDc;
  Pen : hPen;
  OldPen : hPen;
  OldBrush : hBrush;
begin
```



```
inherited;  
dc := GetWindowDC(Handle);  
msg.Result := 1;  
Pen := CreatePen(PS_SOLID, 1, RGB(255, 0, 0));  
OldPen := SelectObject(dc, Pen);  
OldBrush := SelectObject(dc, GetStockObject(NULL_BRUSH));  
Rectangle(dc, 0,0, Form1.Width, Form1.Height);  
SelectObject(dc, OldBrush);  
SelectObject(dc, OldPen);  
DeleteObject(Pen);  
ReleaseDC(Handle, Canvas.Handle);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16085: Object Inspector and Enter Key

Question and Answer Database

FAQ1085D.txt Object Inspector and Enter Key

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I place a return character in a string at design time from the object inspector?

Answer:

You cannot, but there is a work around available in Delphi 3.

Enter the text you want without the enter key, then right click on the form, and choose the "View as text" option. Find the string and change it to something like this:

```
Hint = 'LineOne'+#13+LineTwo'
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16091: EnumFontFamilies()

Question and Answer Database

FAQ1091D.txt - EnumFontFamilies()

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How do I use the Windows API callback function EnumFontFamilies()?

Answer:

The following example adds the list of fonts and sizes for the current printer to a TMemo component.

Example:

```
uses Printers;
```

```
function EnumFontFamilyProc(var lf : TLogFont;  
                           var tm : TNewTextMetric;  
                           FontType : integer;  
                           var Memo : TMemo) : integer  
{ $IFDEF WIN32 } stdcall; { $ELSE } ; export; { $ENDIF }  
begin  
  Memo.Lines.Add(StrPas(@lf.lfFaceName) +  
                #32 + IntToStr(lf.lfHeight));  
  result := 1;  
end;
```

```
function EnumFontFamiliesProc(var lf : TLogFont;  
                              var tm : TNewTextMetric;  
                              FontType : integer;  
                              var Memo : TMemo) : integer  
{ $IFDEF WIN32 } stdcall; { $ELSE } ; export; { $ENDIF }  
begin  
  if FontType = TRUETYPE_FONTTYPE then begin  
    Memo.Lines.Add(StrPas(@lf.lfFaceName) + #32 + 'All Sizes');  
  end else
```

```
EnumFontFamilies(Printer.Handle,  
                  @lf.lfFaceName,  
                  @EnumFontFamilyProc,  
                  LongInt(@Memo));  
  
result := 1;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
var  
    tm : TTextMetric;  
    i : integer;  
begin  
    if PrintDialog1.Execute then begin  
        EnumFontFamilies(Printer.Handle,  
                          nil,  
                          @EnumFontFamiliesProc,  
                          LongInt(@Memo1));  
    end;  
end;
```

4/2/99 11:21:46 AM

Last Modified: 01-SEP-99

## Article #16097: How can I assign a file variable in a Record?

Question and Answer Database

FAQ1097D.txt How can I assign a file variable in a Record?

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I assign a file variable in a Record?

Answer:

While the Object Pascal language does not allow a direct assignment of a file variable, you can use the Move() procedure to move the contents of the variable to another variable. Note: If you save the record to a file for recall at a later date, you will need to re-initialize the file variable.

Example:

```
TMyRec = Record
  f : file;
  OtherField : integer;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f : file;
  MyRec : TMyRec;
begin
  AssignFile(f, 'SomeFile.txt');
  {This line will not compile}
  MyRec.f := f;
  {This line will!}
  Move(f, MyRec.f, sizeof(f));
  Rewrite(MyRec.f);
  CloseFile(MyRec.f);
end;
```

Last Modified: 01-SEP-99

## Article #16103: Creating a flashing icon

Question and Answer Database

FAQ1103D.txt Creating a flashing icon

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I create a flashing icon?

Answer:

You can call the Windows API function `FlashWindow()`:

Example:

```
var
    Flash : bool;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    FlashWindow(Form1.Handle, Flash);
    FlashWindow(Application.Handle, Flash);
    Flash := not Flash;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Flash := False;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16109: Getting the Printer driver and port name from TPrinter's

Question and Answer Database

FAQ1109D.txt - Getting the Printer driver and port name from TPrinter's

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How can I get the Printer driver and port name from TPrinter's GetPrinter() method?

Answer:

Sometimes, these parameters will return blank string. You can use the Windows API function GetProfileString() to retrieve the associated driver and port from the Windows.ini file.

Example:

```
uses Printers;
```

```
{ $IFDEF WIN32 }
```

```
const MAX_PATH = 144;
```

```
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  pDevice : pChar;
```

```
  pDriver  : pChar;
```

```
  pPort    : pChar;
```

```
  hDMode   : THandle;
```

```
begin
```

```
  if PrintDialog1.Execute then begin
```

```
    GetMem(pDevice, cchDeviceName);
```

```
    GetMem(pDriver, MAX_PATH);
```

```
    GetMem(pPort, MAX_PATH);
```

```
    Printer.GetPrinter(pDevice, pDriver, pPort, hDMode);
```

```
    if lstrlen(pDriver) = 0 then begin
```

```
      GetProfileString('Devices', pDevice, '', pDriver, MAX_PATH);
```



```
    pDriver[pos(',', pDriver) - 1] := #0;
end;
if lStrLen(pPort) = 0 then begin
    GetProfileString('Devices', pDevice, '', pPort, MAX_PATH);
    lStrCpy(pPort, @pPort[lStrLen(pPort)+2]);
end;
Mem1.Lines.Add('Device := ' + StrPas(pDevice));
Mem1.Lines.Add('Driver := ' + StrPas(pDriver));
Mem1.Lines.Add('Port := ' + StrPas(pPort));
FreeMem(pDevice, cchDeviceName);
FreeMem(pDriver, MAX_PATH);
FreeMem(pPort, MAX_PATH);
end;
end;
```

4/2/99 11:21:55 AM

Last Modified: 01-SEP-99

## Article #16115: Getting the printing margin of the page

Question and Answer Database

FAQ1115D.txt - Getting the printing margin of the page

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How can I get the printing margin of the page?

Answer:

Under Win16, call the Windows API function `Escape()`, passing the constant `GETPRINTINGOFFSET`. Under Win32, simply call the Windows API function `GetDeviceCaps()` passing the predefined constants `PHYSICALOFFSETX` and `PHYSICALOFFSETY`. Since there is no guarantee a given escape is supported, always call `Escape` with the `QUERYESCSUPPORT` constant to make sure a given escape is implemented. The following example returns the margin of a given printer under both WIN16 and WIN32, accounting for the fact that not all printers will support the escape code `GETPRINTINGOFFSET` (under WIN16). If this is the case, the printing offset is approximated by getting the page size, subtracting the physical resolution of the device, and then dividing by two.

Example:

```
uses Printers;

procedure TForm1.Button1Click(Sender: TObject);
var
  EscapeCode : integer;
  Margin : TPoint;
begin
  if PrintDialog1.Execute then begin
    {$IFDEF WIN32}
      Margin.x :=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETX);
      Margin.y :=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETY);
    {$ELSE}
      EscapeCode := GETPRINTINGOFFSET;
```

```

if Escape(Printer.Handle,
    QUERYESCSUPPORT,
    sizeof(EscapeCode),
    @EscapeCode,
    nil) <> 0 then
if Escape(Printer.Handle,
    GETPRINTINGOFFSET,
    0,
    nil,
    @Margin) < 1 then begin
EscapeCode := GETPHYSPAGESIZE;
if Escape(Printer.Handle,
    QUERYESCSUPPORT,
    sizeof(EscapeCode),
    @EscapeCode,
    nil) <> 0 then
if Escape(Printer.Handle,
    GETPHYSPAGESIZE,
    0,
    nil,
    @Margin) > 0 then begin
Margin.x := (Margin.x -
    GetDeviceCaps(Printer.Handle, HorzRes)) div 2;
Margin.y := (Margin.y -
    GetDeviceCaps(Printer.Handle, VertRes)) div 2;
end else begin
Margin.x := 0;
Margin.y := 0;
end;
end;
{$ENDIF}
Mem1.Lines.Add(IntToStr(Margin.x));
Mem1.Lines.Add(IntToStr(Margin.y));
end;
end;

```

4/2/99 11:22:04 AM

Last Modified: 01-SEP-99

## Article #16121: Retrieving the line number that a memo's cursor is on.

Question and Answer Database

FAQ1121D.txt Retrieving the line number that a memo's cursor is on.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I retrieve the line number that a memo's cursor is on?

Answer:

The zero based location of the cursor can be retrieved by sending an EM\_LINEFROMCHAR message. WParam contains the character number that you wish the line number for, or -1 for the current line (where the caret is located or the beginning of a text selection).

Example:

```
LineNumber := Mem1.Perform(EM_LINEFROMCHAR, -1, 0);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16127: How can I avoid an access violation when using the move procedure

Question and Answer Database

FAQ1127D.txt    How can I avoid an access violation when using the move procedure

Category        :Object Pascal

Platform        :All

Product         :All 32 bit

Question:

How can I avoid an access violation when using the move procedure with pointers?

Answer:

You must dereference the pointers, otherwise you are moving the address of the pointers and possibly corrupting other memory locations.

Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  p1 : pointer;
  p2 : pointer;
begin
  GetMem(p1, 128);
  GetMem(p2, 128);
  {This line may cause an access violation}
  Move(p1, p2, 128);
  {This line is correct}
  Move(p1^, p2^, 128);
  FreeMem(p1, 128);
  FreeMem(p2, 128);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16133: Using Resident Font

Question and Answer Database

FAQ1133D.txt Using Resident Font

Category :VCL

Platform :All

Product :All 32 bit

Question:

Instead of printing graphics, how can I get TPrinter to use the default resident font in the printer?

Answer:

Use the Windows API function GetStockObject() to retrieve the device default font handle, and assign it to Printer.Font.Handle.

Example:

```
uses Printers;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    tm : TTextMetric;
```

```
    i : integer;
```

```
begin
```

```
    if PrintDialog1.Execute then begin
```

```
        Printer.BeginDoc;
```

```
        Printer.Canvas.Font.Handle := GetStockObject(DEVICE_DEFAULT_FONT);
```

```
        GetTextMetrics(Printer.Canvas.Handle, tm);
```

```
        for i := 1 to 10 do begin
```

```
            Printer.Canvas.TextOut(100,
```

```
                i * tm.tmHeight +
```

```
                tm.tmExternalLeading,
```

```
                'Test');
```

```
        end;
```

```
        Printer.EndDoc;
```

```
    end;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16139: Printing in Delphi without using the TPrinter Unit

Question and Answer Database

FAQ1139D.txt - Printing in Delphi without using the TPrinter Unit

Category :Printing  
Platform :All Windows  
Product :

Question:

How can I print in Delphi without using the TPrinter Unit?

Answer:

The following example uses the Windows API function `PrintDlg()` to allow the user to choose a printer and prints two pages using the returned `dc`.

Example:

```
uses CommDlg;
```

```
{ $IFDEF WIN32 }
```

```
const MAX_PATH = 144;
```

```
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    Pd : TPrintDlg;
```

```
    DocInfo: TDocInfo;
```

```
begin
```

```
    FillChar(Pd, sizeof(Pd), #0);
```

```
    Pd.lStructSize := sizeof(Pd);
```

```
    Pd.hWndOwner := Form1.Handle;
```

```
    Pd.Flags := PD_RETURNDC;
```

```
    if PrintDlg(pd) then begin
```

```
        FillChar(DocInfo, sizeof(DocInfo), #0);
```

```
        DocInfo.cbSize := SizeOf(DocInfo);
```

```
        GetMem(DocInfo.lpszDocName, 32);
```

```
        GetMem(DocInfo.lpszOutput, MAX_PATH);
```

```
        lStrCpy(DocInfo.lpszDocName, 'My Document');
```

```
        {Add this line to print to a file }
```



```
lStrCpy(DocInfo.lpszOutput, 'C:\Download\Test.doc');
StartDoc(Pd.hDc, DocInfo);
StartPage(Pd.hDc);
TextOut(Pd.hDc, 100, 100, 'Page 1', 6);
EndPage(Pd.hDc);
StartPage(Pd.hDc);
TextOut(Pd.hDc, 100, 100, 'Page 2', 6);
EndPage(Pd.hDc);
EndDoc(Pd.hDc);
FreeMem(DocInfo.lpszDocName, 32);
FreeMem(DocInfo.lpszOutput, MAX_PATH);
end;
end;
```

4/2/99 11:23:09 AM

Last Modified: 01-SEP-99

## Article #16145: Arrow and Function Keys

Question and Answer Database

FAQ1145D.txt Arrow and Function Keys

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I trap the Arrow and Function Keys?

Answer:

Check for the correct virtual key in the KeyDown event for the form.

Example:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = VK_RIGHT then
    Form1.Caption := 'Right';
  if Key = VK_F1 then
    Form1.Caption := 'F1';
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16151: How can I convert a .dfm file to text and vice versa.

Question and Answer Database

FAQ1151D.txt How can I convert a .dfm file to text and vice versa.

Category :Miscellaneous

Platform :All

Product :All 32 bit

Question:

How can I convert a .dfm file to a text file, and then back to a .dfm file?

Answer:

Use the Convert.exe file found in the Delphi\C++ Builder bin directory.

Example:

```
C:\Delphi\convert enum.dfm enum.txt
```

```
C:\Delphi\convert enum.txt enum.dfm
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16157: Getting the time and date in Universal Time

Question and Answer Database

FAQ1157D.txt Getting the time and date in Universal Time

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I get the time and date in Universal Time?

Answer:

Use the Windows API function `GetSystemTime`.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  lt : TSYSTEMTIME;
  st : TSYSTEMTIME;
begin
  GetLocalTime(lt);
  GetSystemTime(st);
  Mem1.Lines.Add('LocalTime = ' +
    IntToStr(lt.wmonth) + '/' +
    IntToStr(lt.wDay) + '/' +
    IntToStr(lt.wYear) + ' ' +
    IntToStr(lt.wHour) + ':' +
    IntToStr(lt.wMinute) + ':' +
    IntToStr(lt.wSecond));
  Mem1.Lines.Add('UTCTime = ' +
    IntToStr(st.wmonth) + '/' +
    IntToStr(st.wDay) + '/' +
    IntToStr(st.wYear) + ' ' +
    IntToStr(st.wHour) + ':' +
    IntToStr(st.wMinute) + ':' +
    IntToStr(st.wSecond));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16163: Adjusting form size to accommodate a required client size

Question and Answer Database

FAQ1163D.txt     Adjusting form size to accommodate a required client size

Category     :Miscellaneous  
Platform     :All  
Product      :All 32 bit

Question:

How do I adjust my form size to accommodate a required client size?

Answer:

You can make a call to the Windows API function `AdjustWindowRectEx()`. This function takes in a rectangle of your requested client size in screen space, and returns a rectangle containing the required window size and position to accommodate the request.

Notes: 1) The returned rectangle may be larger than the screen. 2) The form's client area may be reduced by the automatic addition of scroll bars. 3) Windows containing title bars have a minimum size allowed by the system, and cannot be made smaller. 4) Menus may form multiple menu bars if the Window is sized too small for the menu to display on a single line. The `AdjustWindowRect()` function does not take this into account.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  pt : TPoint;
  r : TRect;
begin
  pt := ClientToScreen(Point(0, 0));
  r.Left := pt.x;
  r.Top := pt.y;
  pt := ClientToScreen(Point(128, 128));
  r.Right := pt.x;
  r.Bottom := pt.y;
```

```
AdjustWindowRectEx(r,  
                  GetWindowLong(Form1.Handle, GWL_STYLE),  
                  Form1.Menu = nil, {if Form1 has a menu}  
                  GetWindowLong(Form1.Handle, GWL_EXSTYLE));  
Form1.Top := r.Top;  
Form1.Left := r.Left;  
Form1.Width := r.Right - r.Left;  
Form1.Height := r.Bottom - r.Top;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16169: Querying the Audio Cd Autorun feature of Win32

Question and Answer Database

FAQ1169D.txt    Querying the Audio Cd Autorun feature of Win32

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How can I query the Audio Cd Autorun feature of Win32, and how can I set this value?

Answer:

This setting is stored under the two following Windows registry keys:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\AudioCD\Shell

HKEY\_CLASSES\_ROOT\AudioCD\Shell

The value will be the string "play" if the Autorun feature is enabled.

The following example demonstrates both reading and writing these key values.

Example:

```
function IsAudioCdAutoRunOn : bool;
var
  reg: TRegistry;
  Classes : string;
  ClassesRoot : string;
  i : integer;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.OpenKey('SOFTWARE\Classes\AudioCD\Shell',
             false);
  Classes := reg.ReadString('');
```



```

reg.CloseKey;
reg.RootKey := HKEY_CLASSES_ROOT;
reg.OpenKey('AudioCD\Shell',
           false);
ClassesRoot := reg.ReadString('');
reg.CloseKey;
reg.free;
for i := 1 to length(Classes) do
  Classes[i] := UpCase(Classes[i]);
for i := 1 to length(ClassesRoot) do
  ClassesRoot[i] := UpCase(ClassesRoot[i]);
result := ((Classes = 'PLAY') and
          (ClassesRoot = 'PLAY'));
end;

procedure SetAudioCdAutoRun(bOn : bool);
var
  reg: TRegistry;
  Classes : string;
  ClassesRoot : string;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.OpenKey('SOFTWARE\Classes\AudioCD\Shell',
            false);
  if bOn then
    reg.WriteString('', 'play') else
    reg.WriteString('', '');
  reg.CloseKey;
  reg.RootKey := HKEY_CLASSES_ROOT;
  reg.OpenKey('AudioCD\Shell',
            false);
  if bOn then
    reg.WriteString('', 'play') else
    reg.WriteString('', '');
  reg.CloseKey;
  reg.free;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if IsAudioCdAutoRunOn then
    ShowMessage('Autorun Audio Cd is On') else
    ShowMessage('Autorun Audio Cd is Off');
end;

```

end;

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    SetAudioCdAutoRun(true);  
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
    SetAudioCdAutoRun(false);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16175: Testing/setting the CD autorun option under Win32

Question and Answer Database

FAQ1175D.txt     Testing/setting the CD autorun option under Win32  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

### Question:

How can I test and set the CD autorun option under 32 bit versions of Windows?

### Answer:

You can check the 4 byte value in the Windows registry under:

```
Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\  
NoDriveTypeAutoRun
```

Setting the following bit numbers of the value of this key will cause Windows not to use any auto run capabilities that may be present for the device denoted by the bit number.

```
DRIVE_UNKNOWN = bit # 0  
DRIVE_NO_ROOT_DIR = bit #1  
DRIVE_REMOVABLE= Bit # 2  
DRIVE_FIXED = Bit #3  
DRIVE_REMOTE= Bit #4  
DRIVE_CDROM= Bit #5  
DRIVE_RAMDISK= Bit #6
```

You can also set this value to test the autorun capability of a program you are developing from a floppy or network drive, by setting the value to boot the autorun program from a floppy disk (DRIVE\_REMOVABLE) or network drive (DRIVE\_REMOTE). If the device does not have auto-insert detection capabilities, simply inserting the disk and pressing F-5 from the Windows explorer should cause the autorun program to boot.

The following example demonstrates the ability to check if the autorun detection of the CD is enabled, and also demonstrates the ability of turning the autorun detection both on and off

for the CD.

Example

```
uses
  Registry;

function IsCdAutoRunOn : bool;
var
  reg: TRegistry;
  AutoRunSetting : integer;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  reg.OpenKey(
    'Software\Microsoft\Windows\CurrentVersion\Policies\Explorer',
    false);
  reg.ReadBinaryData('NoDriveTypeAutoRun',
    AutoRunSetting,
    sizeof(AutoRunSetting));

  reg.CloseKey;
  reg.free;
  result := not ((AutoRunSetting and (1 shl 5)) <> 0);
end;

procedure SetCdAutoRun(bOn : bool);
var
  reg: TRegistry;
  AutoRunSetting : integer;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  reg.LazyWrite := false;
  reg.OpenKey(
    'Software\Microsoft\Windows\CurrentVersion\Policies\Explorer',
    false);
  reg.ReadBinaryData('NoDriveTypeAutoRun',
    AutoRunSetting,
    sizeof(AutoRunSetting));

  if bOn then
    AutoRunSetting := AutoRunSetting and not (1 shl 5)
  else
    AutoRunSetting := AutoRunSetting or (1 shl 5);
  reg.WriteBinaryData('NoDriveTypeAutoRun',
    AutoRunSetting,
```

```
                sizeof(AutoRunSetting));
    reg.CloseKey;
    reg.free;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if IsCdAutoRunOn then
        ShowMessage('CD AutoRun Is On') else
        ShowMessage('CD AutoRun Is Off');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    SetCdAutoRun(True);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    SetCdAutoRun(False);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16181: Storing/Playing an .AVI file in a database

Question and Answer Database

FAQ1181D.txt Storing/Playing an .AVI file in a database

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How can I store an AVI file in a database and then play AVI files?

Answer:

AVI files can be stored in BLOB (binary large object) fields. The easiest way to play an AVI file stored in a BLOB is to write the BLOB data to a temporary file, then let the multimedia player play the file. The following example demonstrates how to store an AVI file to a BLOB field, and also play the AVI file from the BLOB field.

Example:

```
var
  FileName : string;

{This function gets a temporary file name form the system}
function GetTemporaryFileName : string;
{$IFDEF WIN32}
  const MAX_PATH = 144;
{$ENDIF}
var
  {$IFDEF WIN32}
    lpPathBuffer : PChar;
  {$ENDIF}
  lpbuffer : PChar;
begin
  {Get the file name buffer}
  GetMem(lpBuffer, MAX_PATH);
  {$IFDEF WIN32}
  {Get the temp path buffer}
  GetMem(lpPathBuffer, MAX_PATH);
```

```

{Get the temp path}
  GetTempPath(MAX_PATH, lpPathBuffer);
{Get the temp file name}
  GetTempFileName(lpPathBuffer,
                  'tmp',
                  0,
                  lpBuffer);
{Free the temp path buffer}
  FreeMem(lpPathBuffer, MAX_PATH);
{$ELSE}
{Get the temp file name}
  GetTempFileName(GetTempDrive('C'),
                  'tmp',
                  0,
                  lpBuffer);
{$ENDIF}
{Create a pascal string containg}
{the temp file name and return it}
  result := StrPas(lpBuffer);
{Free the file name buffer}
  FreeMem(lpBuffer, MAX_PATH);
end;

{Read a AVI file into a blob field}
procedure TForm1.Button1Click(Sender: TObject);
var
  FileStream: TFileStream; {to load the avi file}
  BlobStream: TBlobStream; {to save to the blob}
begin
  {Allow the button to repaint}
  Application.ProcessMessages;
  {Turn off the buttons}
  Button1.Enabled := false;
  Button2.Enabled := false;
  {Assign the avi file name to read}
  FileStream := TFileStream.Create(
    'C:\PROGRA~1\BORLAND\DELPHI~1\DEMOS\COOLSTUF\COOL.AVI',
    fmOpenRead);
  Table1.Edit;
  {Create a BlobStream for the TField Table1AVI}
  BlobStream := TBlobStream.Create(Table1AVI, bmReadWrite);
  {Seek to the Beginning of the stream}
  BlobStream.Seek(0, soFromBeginning);
  {Delete any data that may be there}
  BlobStream.Truncate;

```

```

{Copy from the FileStream to the BlobStream}
  BlobStream.CopyFrom(FileStream, FileStream.Size);
{Free the streams}
  FileStream.Free;
  BlobStream.Free;
{Post the record}
  Table1.Post;
{Enable the buttons}
  Button1.Enabled := true;
  Button2.Enabled := true;
end;

```

```

{Read an avi stored in a blob, and play it}
procedure TForm1.Button2Click(Sender: TObject);

```

```

var
  FileStream: TFileStream; {a temp file}
  BlobStream: TBlobStream; {the AVI Blob}
begin
  {Create a blob stream for the AVI blob}
  BlobStream := TBlobStream.Create(Table1.AVI, bmRead);
  if BlobStream.Size = 0 then begin
    BlobStream.Free;
    Exit;
  end;
  {Close the media player}
  MediaPlayer1.Close;
  {Reset the file name}
  MediaPlayer1.FileName := '';
  {Refresh the play window}
  MediaPlayer1.Display := Panell1;
  Panell1.Refresh;
  {if we have a temp file then erase it}
  if FileName <> '' then
    DeleteFile(FileName);
  {Get a temp file name}
  FileName := GetTemporaryFileName;
  {Create a temp file stream}
  FileStream := TFileStream.Create(FileName,
                                  fmCreate or fmOpenWrite);
  {Copy the blob to the temp file}
  FileStream.CopyFrom(BlobStream, BlobStream.Size);
  {Free the streams}
  FileStream.Free;
  BlobStream.Free;
  {Setup the Media player to play the AVI file}

```



```
MediaPlayer1.FileName := filename;
MediaPlayer1.DeviceType := dtAviVideo;
MediaPlayer1.Open;
MediaPlayer1.Play;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  {Unassign the temp file from the media player}
  MediaPlayer1.Close;
  MediaPlayer1.FileName := '';
  {Erase the temp file}
  if FileName <> '' then
    DeleteFile(FileName);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16193: Setting comm port configuration programmatically under Win95

Question and Answer Database

FAQ1193D.txt    Setting comm port configuration programmatically under Win95

Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

Question:

How can I set the comm ports configuration programmatically under Windows 95?

Answer:

The following example demonstrates setting the comm port under Windows95.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  CommPort : string;
  hCommFile : THandle;
  Buffer : PCommConfig;
  size : DWORD;
begin
  CommPort := 'COM1';
  {Open the comm port}
  hCommFile := CreateFile(PChar(CommPort),
                        GENERIC_WRITE,
                        0,
                        nil,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        0);
  if hCommFile=INVALID_HANDLE_VALUE then
  begin
    ShowMessage('Unable to open '+ CommPort);
    exit;
  end;
  {Allocate a temporary buffer}
```

```
GetMem(Buffer, sizeof(TCommConfig));

{Get the size of the CommConfig structure}
{as it may be different than documented}
size := 0;
GetCommConfig(hCommFile, Buffer^, size);

{Free the temporary buffer}
FreeMem(Buffer, sizeof(TCommConfig));

{Allocate the CommConfig structure}
GetMem(Buffer, size);
GetCommConfig(hCommFile, Buffer^, size);

{Change the baud rate}
Buffer^.dcb.BaudRate := 1200;

{Set the comm port to the new configuration}
SetCommConfig(hCommFile, Buffer^, size);

{Free the buffer}
FreeMem(Buffer, size);

{Close the comm port}
CloseHandle(hCommFile);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16199: Changing com port config via Win95 CommConfig Dialog

Question and Answer Database

FAQ1199D.txt Changing com port config via Win95 CommConfig Dialog  
Category :Miscellaneous  
Platform :All  
Product :All 32 bit

### Question:

When I use the Windows 95 CommConfig Dialog box, the settings entered do not seem to change the com port's configuration. How can I set the comm ports configuration using the CommConfig Dialog box?

### Answer:

The following example demonstrates setting the comm port under Windows95 using the CommConfig Dialog box. Note: Any changes made to the dialog are not updated in the system. You are responsible for updating the com ports configuration.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  CommPort : string;
  hCommFile : THandle;
  Buffer : PCommConfig;
  size : DWORD;
begin
  CommPort := 'COM1';
  {Open the comm port}
  hCommFile := CreateFile(PChar(CommPort),
                        GENERIC_WRITE,
                        0,
                        nil,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        0);
  if hCommFile=INVALID_HANDLE_VALUE then
  begin
```

```

    ShowMessage('Unable to open '+ CommPort);
    exit;
end;
{Get a temporary buffer}
    GetMem(Buffer, sizeof(TCommConfig));
{Get the size of the CommConfig structure}
{as it may be different than documented}
    size := 0;
    GetCommConfig(hCommFile, Buffer^, size);
{Free the temporary buffer}
    FreeMem(Buffer, sizeof(TCommConfig));
{Get the CommConfig structure}
    GetMem(Buffer, size);
    GetCommConfig(hCommFile, Buffer^, size);

{Pop up the comm port config dialog}
    if CommConfigDialog(PChar(CommPort),
        Form1.Handle,
        Buffer^) = true then begin
        {Set the com port to the values entered}
        {in the dialog if the user pressed ok}
        SetCommConfig(hCommFile, Buffer^, size);
    end;

{Free the buffer}
    FreeMem(Buffer, size);

{Close the comm port}
    CloseHandle(hCommFile);
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16211: Sending an image to the printer

Question and Answer Database

FAQ1211D.txt      Sending an image to the printer

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How can I reliably print an image to the printer?

Answer:

Sending a bitmap based on the screen to the printer is an invalid operation that will usually fail, unless the print driver has been designed to detect this error condition and compensate for the error. This means you should use the VCL canvas methods `Draw`, `StretchDraw`, `CopyRect`, `BrushCopy`, and the like to transfer a bitmap to the printer, since the underlying bitmap is based on the screen, and is device dependent. The only way to reliably print an image is to use DIBs (Device Independent Bitmaps). Getting a valid DIB can be difficult, as there are many Windows API functions that must be used correctly. Further, many video drivers incorrectly fill in the DIB structure in regards to the color table in the DIB.

The following example demonstrates an attempt to overcome some of these problems and limitations. The example should compile successfully under all versions of Delphi/C++ Builder.

The core function in the example, `BlttBitmapAsDib()`, accepts a handle to a device to image to, the x and y coordinates you wish the bitmap to be imaged at, the width and height you wish the image to be (stretching and shrinking is acceptable), and the `TBitmap` you wish to image.

Example:

```
uses Printers;
```

```
type
```

```
  PPalEntriesArray = ^TPalEntriesArray; {for palette re-construction}
```

```
TPalEntriesArray = array[0..0] of TPaletteEntry;
```

```
procedure BltTBitmapAsDib(DestDc : hdc;      {Handle of where to blt}
                          x : word;        {Bit at x}
                          y : word;        {Blt at y}
                          Width : word;    {Width to stretch}
                          Height : word;   {Height to stretch}
                          bm : TBitmap);   {the TBitmap to Blt}

var
  OriginalWidth : LongInt;                {width of BM}
  dc : hdc;                                {screen dc}
  IsPaletteDevice : bool;                 {if the device uses palettes}
  IsDestPaletteDevice : bool;             {if the device uses palettes}
  BitmapInfoSize : integer;               {sizeof the bitmapinfoheader}
  lpBitmapInfo : PBitmapInfo;            {the bitmap info header}
  hBm : hBitmap;                           {handle to the bitmap}
  hPal : hPalette;                         {handle to the palette}
  OldPal : hPalette;                       {temp palette}
  hBits : THandle;                         {handle to the DIB bits}
  pBits : pointer;                         {pointer to the DIB bits}
  lPPalEntriesArray : PPalEntriesArray;  {palette entry array}
  NumPalEntries : integer;                 {number of palette entries}
  i : integer;                             {looping variable}

begin
  {If range checking is on - lets turn it off for now}
  {we will remember if range checking was on by defining}
  {a define called CKRANGE if range checking is on.}
  {We do this to access array members past the arrays}
  {defined index range without causing a range check}
  {error at runtime. To satisfy the compiler, we must}
  {also access the indexes with a variable. ie: if we}
  {have an array defined as a: array[0..0] of byte,}
  {and an integer i, we can now access a[3] by setting}
  {i := 3; and then accessing a[i] without error}
  {$IFOPT R+}
    {$DEFINE CKRANGE}
    {$R-}
  {$ENDIF}

  {Save the original width of the bitmap}
  OriginalWidth := bm.Width;

  {Get the screen's dc to use since memory dc's are not reliable}
  dc := GetDc(0);
  {Are we a palette device?}
```

```

IsPaletteDevice :=
  GetDeviceCaps(dc, RASTERCAPS) and RC_PALETTE = RC_PALETTE;
{Give back the screen dc}
dc := ReleaseDc(0, dc);

{Allocate the BitmapInfo structure}
if IsPaletteDevice then
  BitmapInfoSize := sizeof(TBitmapInfo) + (sizeof(TRGBQUAD) * 255)
else
  BitmapInfoSize := sizeof(TBitmapInfo);
GetMem(lpBitmapInfo, BitmapInfoSize);

{Zero out the BitmapInfo structure}
FillChar(lpBitmapInfo^, BitmapInfoSize, #0);

{Fill in the BitmapInfo structure}
lpBitmapInfo^.bmiHeader.biSize := sizeof(TBitmapInfoHeader);
lpBitmapInfo^.bmiHeader.biWidth := OriginalWidth;
lpBitmapInfo^.bmiHeader.biHeight := bm.Height;
lpBitmapInfo^.bmiHeader.biPlanes := 1;
if IsPaletteDevice then
  lpBitmapInfo^.bmiHeader.biBitCount := 8
else
  lpBitmapInfo^.bmiHeader.biBitCount := 24;
lpBitmapInfo^.bmiHeader.biCompression := BI_RGB;
lpBitmapInfo^.bmiHeader.biSizeImage :=
  ((lpBitmapInfo^.bmiHeader.biWidth *
    longint(lpBitmapInfo^.bmiHeader.biBitCount)) div 8) *
  lpBitmapInfo^.bmiHeader.biHeight;
lpBitmapInfo^.bmiHeader.biXPelsPerMeter := 0;
lpBitmapInfo^.bmiHeader.biYPelsPerMeter := 0;
if IsPaletteDevice then begin
  lpBitmapInfo^.bmiHeader.biClrUsed := 256;
  lpBitmapInfo^.bmiHeader.biClrImportant := 256;
end else begin
  lpBitmapInfo^.bmiHeader.biClrUsed := 0;
  lpBitmapInfo^.bmiHeader.biClrImportant := 0;
end;

{Take ownership of the bitmap handle and palette}
hBm := bm.ReleaseHandle;
hPal := bm.ReleasePalette;

{Get the screen's dc to use since memory dc's are not reliable}
dc := GetDc(0);

```





```

for i := 0 to (NumPalEntries - 1) do begin
    lpBitmapInfo^.bmiColors[i].rgbRed :=
        lPPalEntriesArray^[i].peRed;
    lpBitmapInfo^.bmiColors[i].rgbGreen :=
        lPPalEntriesArray^[i].peGreen;
    lpBitmapInfo^.bmiColors[i].rgbBlue :=
        lPPalEntriesArray^[i].peBlue;
end;
FreeMem(lPPalEntriesArray, sizeof(TPaletteEntry) * 256);
end;

if IsPaletteDevice then begin
    {Select the old palette back in}
    SelectPalette(dc, OldPal, TRUE);
    {Realize the old palette}
    RealizePalette(dc);
end;

{Give back the screen dc}
dc := ReleaseDc(0, dc);

{Is the Dest dc a palette device?}
IsDestPaletteDevice :=
    GetDeviceCaps(DestDc, RASTERCAPS) and RC_PALETTE = RC_PALETTE;

if IsPaletteDevice then begin
    {If we are using a palette, it must be}
    {selected into the dc during the conversion}
    OldPal := SelectPalette(DestDc, hPal, TRUE);
    {Realize the palette}
    RealizePalette(DestDc);
end;

{Do the blt}
StretchDibits(DestDc,
              x,
              y,
              Width,
              Height,
              0,
              0,
              OriginalWidth,
              lpBitmapInfo^.bmiHeader.biHeight,
              pBits,

```

```
    lpBitmapInfo^,  
    DIB_RGB_COLORS,  
    SrcCopy);
```

```
if IsDestPaletteDevice then begin  
  {Select the old palette back in}  
  SelectPalette(DestDc, OldPal, TRUE);  
  {Realize the old palette}  
  RealizePalette(DestDc);  
end;
```

```
{De-Allocate the Dib Bits}  
GlobalUnLock(hBits);  
GlobalFree(hBits);
```

```
{De-Allocate the BitmapInfo}  
FreeMem(lpBitmapInfo, BitmapInfoSize);
```

```
{Set the ownership of the bimap handles back to the bitmap}  
bm.Handle := hBm;  
bm.Palette := hPal;
```

```
  {Turn range checking back on if it was on when we started}  
{$IFDEF CKRANGE}  
  {$UNDEF CKRANGE}  
  {$R+}  
{$ENDIF}  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  if PrintDialog1.Execute then begin  
    Printer.BeginDoc;  
    BltTBitmapAsDib(Printer.Canvas.Handle,  
                    0,  
                    0,  
                    Image1.Picture.Bitmap.Width,  
                    Image1.Picture.Bitmap.Height,  
                    Image1.Picture.Bitmap);  
    Printer.EndDoc;  
  end;  
end;
```

Last Modified: 01-SEP-99

## Article #16217: Using TClientDataSet with local data

Question and Answer Database

FAQ1217D.txt Using TClientDataSet with local data  
Category :Multitier/Remote Datasets  
Platform :All  
Product :Delphi 3.x

Question:

Is it possible to use a TClientDataSet in a Local application with paradox tables without using a TProvider and a TRemoteServer?

Answer:

You can't get rid of the Provider, but you can use it in a single-tier application. In order to open the Client DataSet you assign the provider manually.

```
CDS.Provider := Table1.Provider;  
CDS.Open;
```

Also, make sure you include BDEProv in the uses clause.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16223: 'Duplicate resource' error

Question and Answer Database

FAQ1223D.txt 'Duplicate resource' error

Category :Compiler

Platform :All

Product :All 32 bit

Question:

When I try to add a resource to my project's .res file, I get a "Duplicate Resource" error when linking. The resource I have added is a unique resource. How can I successfully add the resource to my project?

Answer:

The projects resource file is generated by the IDE wizard, and is not intended to be modified. To add additional resources to your project, create a separate resource file with a unique name that does not conflict with either the project or any of the unit names. Then to add the resource file to Delphi, simply add the following line to any unit file in the project:

```
{ $R MyRes.Res }
```

(where MyRes.Res is the name of the resource file).

To add the resource to Borland's C++ Builder, add the resource from the project manager.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16229: Adding hint windows to applications

Question and Answer Database

FAQ1229D.txt Adding hint windows to applications

Category :Miscellaneous

Platform :All

Product :All 32 bit

Question:

How can I pop up a hint window?

Answer:

You can either programmatically move the mouse pointer over the control with the hint, or you can display a hint window of your own;

Example # 1 - programmatically moving the mouse.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  pt : TPoint;
begin
  {Let the button repaint}
  Application.ProcessMessages;
  {Get the screen coordinates of the center of the button}
  pt := ClientToScreen(Point(Button1.Left + Button1.Width div 2,
                             Button1.Top + Button1.Height div 2));
  {Position the cursor at the center of the button}
  SetCursorPos(Pt.x, Pt.y);
end;
```

Example #2 - Creating you own hint window. For this example, you will need a TPanel and a TTimer component. The Panel will be used as the hint window, and will be invisible untill called. The timer will hide the panel after a specified duration.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Timer1.Enabled := false;
  Panell.Visible := false;
  Panell.BevelInner := bvNone;
```

```
Panel1.BevelOuter := bvNone;
Panel1.BorderStyle := bsSingle;
Panel1.Color := clWhite;
Button1.Hint := 'Hint test';
end;
```

```
procedure TForm1.ShowAHint(x : integer;
                           y : integer;
                           Caption : string;
                           Duration : LongInt);
```

```
var
  dc : hdc;
  OldFont : hFont;
  pt : TSize;
  p : pChar;
begin
  if Timer1.Enabled <> false then
    Timer1.Enabled := false;
  Timer1.Enabled := false;
  if Panel1.Visible <> false then
    Panel1.Visible := false;
  if Caption = '' then exit;
  Panel1.Caption := caption;
  {Get the width of the caption string}
  GetMem(p, Length(Panel1.Caption) + 1);
  StrPCopy(p, Panel1.Caption);
  dc := GetDc(Panel1.Handle);
  OldFont := SelectObject(dc, Panel1.Font.Handle);
  GetTextExtentPoint32(dc, p, Length(Panel1.Caption), pt);
  SelectObject(dc, OldFont);
  ReleaseDc(Panel1.Handle, Dc);
  FreeMem(p, Length(Panel1.Caption) + 1);
  {Position and show the panel}
  Panel1.Left := x;
  Panel1.Top := y;
  Panel1.Width := pt.cx + 6;
  Panel1.Height := pt.cy + 2;
  Panel1.Visible := true;
  {Fire off the timer to hide the panel}
  Timer1.Interval := Duration;
  Timer1.Enabled := true;
end;
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
```



```
if Panell1.Visible <> false then
    Panell1.Visible := false;
    Timer1.Enabled := false;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    {Let the button repaint}
    Application.ProcessMessages;
    ShowAHint(Button1.Left,
              Button1.Top + Button1.Height + 6,
              Button1.Hint,
              2000);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16235: Using InvalidateRect()to repaint the entire form

Question and Answer Database

FAQ1235D.txt Using InvalidateRect()to repaint the entire form

Category :Windows API

Platform :All

Product :All 32 bit

Question:

When I resize my form, the paint area is clipped, but I need to repaint the whole area. The form's ClipRect is read only.

How can I get around this?

Answer:

During the resize event, make a call to the Windows API function InvalidateRect(). Passing nil as the second parameter will cause the clipping rect of the form to include the entire client area of the form. The third parameter denotes if the background of the form should be erased and repainted. Simply calling the invalidate method of the form may result in a clipping region and possibly the background getting erased.

Example:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  InvalidateRect(Form1.Handle, nil, false);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16241: Using mouse\_event() to simulate mouse events

Question and Answer Database

FAQ1241D.txt Using mouse\_event() to simulate mouse events

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I use the mouse\_event() procedure to simulate a mouse event?

Answer:

The following example demonstrates using the Windows API function mouse\_event() to simulate a mouse event. When Button2 is clicked, the example code places the mouse over Button1 and clicks it. Mouse Coordinates given are in "Mickey's", where their are 65535 "Mickey's" to a screen's width.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Button 1 clicked');
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Pt : TPoint;
begin
  {Allow Button2 to repaint it's self}
  Application.ProcessMessages;
  {Get the point in the center of button 1}
  Pt.x := Button1.Left + (Button1.Width div 2);
  Pt.y := Button1.Top + (Button1.Height div 2);
  {Convert Pt to screen coordinates}
  Pt := ClientToScreen(Pt);
  {Convert Pt to mickeys}
  Pt.x := Round(Pt.x * (65535 / Screen.Width));
  Pt.y := Round(Pt.y * (65535 / Screen.Height));
  {Move the mouse}
  Mouse_Event(MOUSEEVENTF_ABSOLUTE or
              MOUSEEVENTF_MOVE,
```

```
Pt.x,
```

```
Pt.y,
```

```
0,
```

```
0);
```

```
{Simulate the left mouse button down}
```

```
  Mouse_Event(MOUSEEVENTF_ABSOLUTE or
```

```
    MOUSEEVENTF_LEFTDOWN,
```

```
    Pt.x,
```

```
    Pt.y,
```

```
    0,
```

```
    0);;
```

```
{Simulate the left mouse button up}
```

```
  Mouse_Event(MOUSEEVENTF_ABSOLUTE or
```

```
    MOUSEEVENTF_LEFTUP,
```

```
    Pt.x,
```

```
    Pt.y,
```

```
    0,
```

```
    0);;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16247: Painting a rainbow color spectrum onto a form

Question and Answer Database

FAQ1247D.txt     Painting a rainbow color spectrum onto a form

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How do I paint the color spectrum of a rainbow, and if the spectrum is clicked on, how do I calculate what color was clicked on?

Answer:

The following example demonstrates painting a color spectrum, and calculating the color of a given point on the spectrum. Two procedures are presented: PaintRainbow() and ColorAtRainbowPoint(). The PaintRainbow() procedure paints a spectrum from red to magenta if the WrapToRed parameter is false, or paint red to red if the WrapToRed parameter is true. The rainbow can progress either in a horizontal or vertical progression. The ColorAtRainbowPoint() function returns a TColorRef containing the color at a given point in the rainbow.

Example:

```
procedure PaintRainbow(Dc : hDc; {Canvas to paint to}
                      x : integer; {Start position X}
                      y : integer; {Start position Y}
                      Width : integer; {Width of the rainbow}
                      Height : integer {Height of the rainbow};
                      bVertical : bool; {Paint vertically}
                      WrapToRed : bool); {Wrap spectrum back to red}
var
  i : integer;
  ColorChunk : integer;
  OldBrush : hBrush;
  OldPen : hPen;
  r : integer;
  g : integer;
```

```

b : integer;
Chunks : integer;
ChunksMinus1 : integer;
pt : TPoint;
begin
  OffsetViewportOrgEx(Dc,
                     x,
                     y,
                     pt);

  if WrapToRed = false then
    Chunks := 5 else
    Chunks := 6;
  ChunksMinus1 := Chunks - 1;

  if bVertical = false then
    ColorChunk := Width div Chunks else
    ColorChunk := Height div Chunks;

{Red To Yellow}
  r := 255;
  b := 0;
  for i := 0 to ColorChunk do begin
    g:= (255 div ColorChunk) * i;
    OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r, g, b)));
    if bVertical = false then
      PatBlt(Dc, i, 0, 1, Height, PatCopy) else
      PatBlt(Dc, 0, i, Width, 1, PatCopy);
    DeleteObject(SelectObject(Dc, OldBrush));
  end;

{Yellow To Green}
  g:=255;
  b:=0;
  for i := ColorChunk to (ColorChunk * 2) do begin
    r := 255 - (255 div ColorChunk) * (i - ColorChunk);
    OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r, g, b)));
    if bVertical = false then
      PatBlt(Dc, i, 0, 1, Height, PatCopy) else
      PatBlt(Dc, 0, i, Width, 1, PatCopy);
    DeleteObject(SelectObject(Dc, OldBrush));
  end;

{Green To Cyan}
  r:=0;

```

```

g:=255;
for i:= (ColorChunk * 2) to (ColorChunk * 3) do begin
  b := (255 div ColorChunk)*(i - ColorChunk * 2);
  OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r, g, b)));
  if bVertical = false then
    PatBlt(Dc, i, 0, 1, Height, PatCopy) else
    PatBlt(Dc, 0, i, Width, 1, PatCopy);
  DeleteObject(SelectObject(Dc,OldBrush));
end;

```

```
{Cyan To Blue}
```

```

r := 0;
b := 255;
for i:= (ColorChunk * 3) to (ColorChunk * 4) do begin
  g := 255 - ((255 div ColorChunk) * (i - ColorChunk * 3));
  OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r, g, b)));
  if bVertical = false then
    PatBlt(Dc, i, 0, 1, Height, PatCopy) else
    PatBlt(Dc, 0, i, Width, 1, PatCopy);
  DeleteObject(SelectObject(Dc, OldBrush));
end;

```

```
{Blue To Magenta}
```

```

g := 0;
b := 255;
for i:= (ColorChunk * 4) to (ColorChunk * 5) do begin
  r := (255 div ColorChunk) * (i - ColorChunk * 4);
  OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r, g, b)));
  if bVertical = false then
    PatBlt(Dc, i, 0, 1, Height, PatCopy) else
    PatBlt(Dc, 0, i, Width, 1, PatCopy);
  DeleteObject(SelectObject(Dc, OldBrush))
end;

```

```
if WrapToRed <> false then begin
```

```
{Magenta To Red}
```

```
r := 255;
```

```
g := 0;
```

```

for i := (ColorChunk * 5) to ((ColorChunk * 6) - 1) do begin
  b := 255 -((255 div ColorChunk) * (i - ColorChunk * 5));
  OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r,g,b)));
  if bVertical = false then
    PatBlt(Dc, i, 0, 1, Height, PatCopy) else
    PatBlt(Dc, 0, i, Width, 1, PatCopy);
  DeleteObject(SelectObject(Dc,OldBrush));

```

```

    end;
end;

{Fill Remainder}
if (Width - (ColorChunk * Chunks) - 1 ) > 0 then begin
    if WrapToRed <> false then begin
        r := 255;
        g := 0;
        b := 0;
    end else begin
        r := 255;
        g := 0;
        b := 255;
    end;
    OldBrush := SelectObject(Dc, CreateSolidBrush(Rgb(r, g, b)));
    if bVertical = false then
        PatBlt(Dc,
            ColorChunk * Chunks,
            0,
            Width - (ColorChunk * Chunks),
            Height,
            PatCopy) else
        PatBlt(Dc,
            0,
            ColorChunk * Chunks,
            Width,
            Height - (ColorChunk * Chunks),
            PatCopy);
    DeleteObject(SelectObject(Dc, OldBrush));
end;
OffsetViewportOrgEx(Dc,
                    Pt.x,
                    Pt.y,
                    pt);

end;

function ColorAtRainbowPoint(ColorPlace : integer;
                             RainbowWidth : integer;
                             WrapToRed : bool) : TColorRef;

var
    ColorChunk : integer;
    ColorChunkIndex : integer;
    ColorChunkStart : integer;
begin
    if ColorPlace = 0 then begin

```



```

    result := RGB(255, 0, 0);
    exit;
end;
{WhatChunk}
if WrapToRed <> false then
    ColorChunk := RainbowWidth div 6 else
    ColorChunk := RainbowWidth div 5;
    ColorChunkStart := ColorPlace div ColorChunk;
    ColorChunkIndex := ColorPlace mod ColorChunk;
case ColorChunkStart of
    0 : result := RGB(255,
                      (255 div ColorChunk) * ColorChunkIndex,
                      0);
    1 : result := RGB(255 - (255 div ColorChunk) * ColorChunkIndex,
                      255,
                      0);
    2 : result := RGB(0, 255, (255 div ColorChunk) * ColorChunkIndex);
    3 : result := RGB(0,
                      255 - (255 div ColorChunk) * ColorChunkIndex,
                      255);
    4 : result := RGB((255 div ColorChunk) * ColorChunkIndex,
                      0,
                      255);
    5 : result := RGB(255,
                      0,
                      255 - (255 div ColorChunk) * ColorChunkIndex);
else
    if WrapToRed <> false then
        result := RGB(255, 0, 0) else
        result := RGB(255, 0, 255);
    end;{Case}
end;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    PaintRainbow(Form1.Canvas.Handle,
                 0,
                 0,
                 Form1.ClientWidth,
                 Form1.ClientHeight,
                 false,
                 true);
end;

```

```
procedure TForm1.FormResize(Sender: TObject);
begin
  InvalidateRect(Form1.Handle, nil, false);
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Color : TColorRef;
begin
  Color := ColorAtRainbowPoint(y,
                                Form1.ClientWidth,
                                true);
  ShowMessage(IntToStr(GetRValue(Color)) + #32 +
              IntToStr(GetGValue(Color)) + #32 +
              IntToStr(GetBValue(Color)));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16253: Notifying applications that the registry has changed

Question and Answer Database

FAQ1253D.txt    Notifying applications that the registry has changed  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

### Question:

When I make a change to the registry, some applications do not seem to acknowledge the changes until they are restarted. How can I get the applications to respond to the changes?

### Answer:

Broadcast a WM\_WININICHANGE message to the system, sending a null terminated string detailing the registry section that changed. Most well written applications should respond to the WM\_WININICHANGE message.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    SendMessage(HWND_BROADCAST,
                WM_WININICHANGE,
                0,
                LongInt(PChar('RegistrySection')));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16259: Turning the screen saver off?

Question and Answer Database

FAQ1259D.txt Turning the screen saver off?

Category :Miscellaneous

Platform :All

Product :All 32 bit

Question:

How do I turn the screen saver off?

Answer:

The following example shows how to turn the screensaver off.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {Turn it off}
  SystemParametersInfo(SPI_SETSCRENSAVEACTIVE,
                        0,
                        nil,
                        0);
  {Turn it on}
  SystemParametersInfo(SPI_SETSCRENSAVEACTIVE,
                        1,
                        nil,
                        0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16265: Stopping TScrollbar button from blinking

Question and Answer Database

FAQ1265D.txt Stopping TScrollbar button from blinking

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I stop the button on a TScrollBar component from blinking?

Answer:

Set the scrollbar's TabStop property to False.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16271: Where is SYNCOBJS.pas unit?

Question and Answer Database

FAQ1271D.txt    Where is SYNCOBJS.pas unit?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Where is SYNCOBJS.pas unit?

Answer:

This is included with the Delphi 3 client server suite, in the directory:

C:\Program Files\Borland\Delphi 3\Source\INTERNET\syncobjs.pas

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16277: Creating temporary canvas

Question and Answer Database

FAQ1277D.txt Creating temporary canvas

Category :VCL

Platform :All

Product :All 32 bit

### Question:

I need a temporary canvas to draw to, but when I attempt to create one, I get several application errors. How do I do this?

### Answer:

Create a Bitmap, and use the TBitmap's canvas property to draw on. The following example creates a Bitmap, draw on its' canvas, draws the canvas to the form's background, and then free the bitmap.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bm : TBitmap;
begin
  bm := TBitmap.Create;
  bm.Width := 100;
  bm.Height := 100;
  bm.Canvas.Brush.Color := clRed;
  bm.Canvas.FillRect(Rect(0, 0, 100, 100));
  bm.Canvas.MoveTo(0, 0);
  bm.Canvas.LineTo(100, 100);
  Form1.Canvas.StretchDraw(Form1.ClientRect,
                           Bm);

  bm.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16283: TTimer not firing at correct interval

Question and Answer Database

FAQ1283D.txt TTimer not firing at correct interval

Category :VCL

Platform :All

Product :All 32 bit

### Question:

When I create the TTimer component, it does not fire at the correct specified time. How can I get more accurate results?

### Answer:

Windows timer event were not designed to accommodate hi-resolution timing. In other words, while you may set the timer to fire every 1000 milliseconds, it may fire sometimes at intervals longer than 1000 milliseconds. Values below 55 milliseconds will never fire on time under Windows, since this is the Timer components minimum resolution. You can get the time during the timer event and compare it to the last time the event fired to receive a better timing resolution.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16289: Error Setting Exception Debug Hook

Question and Answer Database

FAQ1289D.txt     Error Setting Exception Debug Hook

Category        :IDE

Platform        :All

Product         :All 32 bit

### Question:

Sometimes when I run my program from the IDE and have an exception, when I fix the problem and attempt to re-run the application, I get a message "Error Setting Exception Debug Hook", then Delphi refuses to run the application. I then have to restart Delphi. Is there any way around this?

### Answer:

Yes! If you see the message, just cancel the dialog and press Ctrl-F2 to reset the program. Then do a "build all" and re-run your application.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16295: Working with objects and functions in other units

Question and Answer Database

FAQ1295D.txt Working with objects and functions in other units

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I call a function or procedure in another unit, or retrieve and set the value of a variable in another unit?

Answer:

Anything defined in the interface section of a unit is visible and callable from another unit that uses that unit. This includes Types, Constants, Classes, Functions, and Procedures. You should simply add the unit's name you want to use to the uses clause of the unit you wish to use it from. Unless necessary, you should define the uses clause in the implementation section, to help avoid circular reference errors. This is where two units depend on each other in their interface sections.

If you have a function in the current unit that has the same name as a function in another unit your wish to call, you can preface the function name with the unit name to fully qualify it.

The following example demonstrates calling functions and procedures and accessing variables in another unit:

Example:

```
{Code for unit2}
```

```
unit Unit2;
```

```
interface
```

```
procedure SomeProcedure(i : integer);
```

```
function SomeFunction(i : integer) : integer;
```

```
var
  SomeInteger : integer;

implementation

procedure SomeProcedure(i : integer);
begin
  SomeInteger := i;
end;

function SomeFunction(i : integer) : integer;
begin
  Result := SomeInteger;
  SomeInteger := i;
end;

end.
```

```
{Code for unit1's implementation section:}
```

```
implementation
```

```
uses
```

```
  Unit2;
```

```
{$R *.DFM}
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
  SomeProcedure(10);
```

```
  ShowMessage(IntToStr(SomeFunction(12)));
```

```
  ShowMessage(IntToStr(SomeInteger));
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
```

```
{Lets do the same thing as Button1Click but fully  
qualify the names}
```

```
  Unit2.SomeProcedure(10);
```

```
  ShowMessage(IntToStr(Unit2.SomeFunction(12)));
```

```
  ShowMessage(IntToStr(Unit2.SomeInteger));
```

```
end;
```

```
7/16/98 4:31:28 PM
```

Last Modified: 01-SEP-99

## Article #16301: Programmatically closing another application

Question and Answer Database

FAQ1301D.txt Programmatically closing another application

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I programmatically close another application?

Answer:

You can send the application a WM\_QUIT message;

Example:

```
PostMessage(FindWindow(nil, 'window caption'), WM_QUIT, 0, 0);
```

Where "window caption" is the caption of the window that you are sending the message to.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16307: Formatting a drive under Win32

Question and Answer Database

FAQ1307D.txt     Formatting a drive under Win32

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I format a drive under Win32?

Answer:

You can use the ShellAPI funciton ShFormatDrive().

Example:

```
const SHFMT_DRV_A = 0;
const SHFMT_DRV_B = 1;

const SHFMT_ID_DEFAULT = $FFFF;

const SHFMT_OPT_QUICKFORMAT = 0;
const SHFMT_OPT_FULLLFORMAT = 1;
const SHFMT_OPT_SYSONLY = 2;

const SHFMT_ERROR = -1;
const SHFMT_CANCEL = -2;
const SHFMT_NOFORMAT = -3;

function SHFormatDrive(hWnd : HWND;
                      Drive : Word;
                      fmtID : Word;
                      Options : Word) : Longint
  stdcall; external 'Shell32.dll' name 'SHFormatDrive';

procedure TForm1.Button1Click(Sender: TObject);
var
  FmtRes : longint;
begin
  try
    FmtRes:= ShFormatDrive(Handle,
```

```
SHFMT_DRV_A,  
SHFMT_ID_DEFAULT,  
SHFMT_OPT_QUICKFORMAT);
```

```
case FmtRes of  
  SHFMT_ERROR : ShowMessage('Error formatting the drive');  
  SHFMT_CANCEL :  
    ShowMessage('User canceled formatting the drive');  
  SHFMT_NOFORMAT : ShowMessage('No Format')  
else  
  ShowMessage('Disk has been formatted');  
end;  
except  
end;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16313: Disabling the Win95 start button

Question and Answer Database

FAQ1313D.txt Disabling the Win95 start button

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I hide and disable the Win95 start button?

Answer:

The following example both hides and unhides the Windows start button, and also enables and disables the start button.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Rgn : hRgn;
begin
  {Hide the start button}
  Rgn := CreateRectRgn(0, 0, 0, 0);
  SetWindowRgn(FindWindowEx(FindWindow('Shell_TrayWnd', nil),
                               0,
                               'Button',
                               nil),
               Rgn,
               true);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  {Turn the start button back on}
  SetWindowRgn(FindWindowEx(FindWindow('Shell_TrayWnd', nil),
                               0,
                               'Button',
                               nil),
               0,
```



```

end;
true);

procedure TForm1.Button3Click(Sender: TObject);
begin
  {Disable the start button}
  EnableWindow(FindWindowEx(FindWindow('Shell_TrayWnd', nil),
                                0,
                                'Button',
                                nil),
                false);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  {Enable the start button}
  EnableWindow(FindWindowEx(FindWindow('Shell_TrayWnd', nil),
                                0,
                                'Button',
                                nil),
                true);
end

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16319: Determining minor version number

Question and Answer Database

FAQ1319D.txt Determining minor version number

Category :Windows API

Platform :All

Product :All 32 bit

### Question:

The Delphi/Borland C++ Builder cd's are not marked with the minor version. How can I tell the version number of the CD?

### Answer:

You can tell the version number from the timestamp of the Bcb.exe, Delphi.exe or Delphi32.exe files found from the cd in the bin subdirectory of RunImage, or in the installed directory.

The time listed on the file will be the version number.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16325: Inserting a file at the cursor

Question and Answer Database

FAQ1325D.txt     Inserting a file at the cursor

Category        :IDE

Platform        :All

Product         :All 32 bit

Question:

Is there any way to insert the contents of a file at the cursor?

Answer:

Yes! Depending on your editor setting in the IDE environment options, the following keystrokes will insert the contents of a file at the cursor:

Default and Classic: Ctrl+K+R

Brief: Alt+R

Epsilon: None

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16331: Indenting a block of code

Question and Answer Database

FAQ1331D.txt Indenting a block of code

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I quickly indent a block of code in the IDE?

Answer:

With the default key mapping, as set in the editor preferences, the keys shift+ctrl+I will indent a block, and shift+ctrl+U will un-indent a block.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16337: Controlling shortcut keystrokes in a string grid

Question and Answer Database

FAQ1337D.txt     Controlling shortcut keystrokes in a string grid  
Category        :Windows API  
Platform        :All  
Product         :All 32 bit

### Question:

I have a string grid that is read only. I am trapping any keystrokes sent to the string grid. I have a button that has a caption of E&xit. When ever the user presses the x key (even if the alt key is not pressed) in the string grid, the button click is executed. How can I prevent this?

### Answer:

The following example demonstrates trapping the CM\_DIALOGCHAR message at the form level. This will give you an opportunity to respond to a dialog character only if the alt key is held down, preventing the default handling of the character shortcuts.

### Example:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    StringGrid1: TStringGrid;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure StringGrid1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
  private
    { Private declarations }
    procedure CMDialogChar(var Message: TCMDialogChar);
      message CM_DIALOGCHAR;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  Button1.Caption := 'E&xit';  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  Application.Terminate;  
end;
```

```
procedure TForm1.StringGrid1KeyDown(Sender: TObject;  
var Key: Word; Shift: TShiftState);  
begin  
  ShowMessage('Grid keypress = ' + Char(Key));  
  Key := 0;  
end;
```

```
procedure TForm1.CMDDialogChar(var Message: TCMDDialogChar);  
begin  
  if ssAlt in KeyDataToShiftState(Message.KeyData) then  
    inherited;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16343: Disabling ALT-F4 to prevent form closing

Question and Answer Database

FAQ1343D.txt Disabling ALT-F4 to prevent form closing

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I disable the Alt+F4 key combination to keep my form from closing?

Answer:

If the intent is to make sure the user doesn't exit prematurely before performing some required task, use the OnCloseQuery event of the form and set the CanClose property appropriately.

If you need to override the default behavior for normal form handling and prevent the Alt+F4 keystrokes from being responded to, then set the forms KeyPreview property to true and trap the key during the forms KeyDown event.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  KeyPreview := true;
end;
```

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if ((ssAlt in Shift) and (Key = VK_F4)) then
    Key := 0;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16349: Prevent window update

Question and Answer Database

FAQ1349D.txt Prevent window update

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I temporarily prevent any given window from updating?

Answer:

Make a call to the Windows API function LockWindowUpdate with the handle of the Window you wish to lock as a parameter. Send a zero as a parameter to enable updating and to restore the drawing.

```
LockWindowUpdate(Mem01.Handle);
```

```
.
```

```
.
```

```
LockWindowUpdate(0);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16355: Preventing multiple application instances

Question and Answer Database

FAQ1355D.txt Preventing multiple application instances

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I guarantee that only one instance of my program executes? Is it possible to restore the previous instance and shut down if I start more than one instance?

Answer:

Under Win16, you can check the value of `hPrevInst` at startup. If it is non-zero, then another instance of your application is running, and you can take steps to abort the current instance. Under Win32, `hPrevInst` is always zero and cannot be relied upon. You should attempt to create a uniquely named mutex. If the creation fails, then there is another instance running.

The following example attempts to create a named mutex. If the mutex fails, a unique system wide custom windows message is registered with the system, and is then broadcast to all top level windows. The second instance then shuts down. The previous instance has also registered this unique custom system wide message, and has trapped the window procedure of `Form1` to respond to this message. If the message is received, then `Form1` is restored (if minimized) and then made to be the top level window. Care is taken in the example to minimize the effect of Delphi/Borland C++ Builder's hidden application window, and its effect on the task bar icon for `Form1`.

Example:

```
{The code for OneInstance.dpr}
```

```
program OneInstance;
```

```
uses
```

```
    Windows,
```

```

Forms,
Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  {Attempt to create a named mutex}
  CreateMutex(nil, false, 'MyApp');
  {if it failed then there is another instance}
  if GetLastError = ERROR_ALREADY_EXISTS then begin
    {Send all windows our custom message - only our other}
    {instance will recognise it, and restore itself}
    SendMessage(HWND_BROADCAST,
                RegisterWindowMessage('MyApp'),
                0,
                0);
    {Lets quit}
    Halt(0);
  end;
  Application.Initialize;
  {Tell Delphi to un-hide it's hidden application window}
  {This allows our instance to have a icon on the task bar}
  Application.ShowMainForm := true;
  ShowWindow(Application.Handle, SW_RESTORE);
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

{The code for unit1.pas}

unit Unit1;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }

```



```
end;  
  
procedure TForm1.FormDestroy(Sender: TObject);  
begin  
  {Set form1's window proc back to it's original procedure}  
  SetWindowLong(Form1.Handle,  
                GWL_WNDPROC,  
                LongInt(OldWindowProc));  
end;  
  
begin  
  {Tell Delphi to hide it's hidden application window for now to avoid}  
  {a "flash" on the taskbar if we halt due to another instance}  
  ShowWindow(Application.Handle, SW_HIDE);  
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16361: Programatically installing printer driver

Question and Answer Database

FAQ1361D.txt     Programatically installing printer driver

Category     :Windows API

Platform     :All

Product      :All 32 bit

Question:

My application uses a custom print dirver. Is it possible to silently install a print driver in Windows 3.1 or Windows 95 without user intervention?

Answer:

The following example silently installs a print driver. You will need to copy the print driver files to the Windows\System directory and add the following entries to the Win.Ini file.

Notes:

In the follwong example:

```
DriverName = Name of the driver;  
DRVFILE - Filename of the driver minus the extension  
         (unless it is not ".drv").
```

Example:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
  s : array[0..64] of char;  
begin  
  WriteProfileString('PrinterPorts',  
                    'DriverName',  
                    'DRVFILE,FILE:,15,45');  
  WriteProfileString('Devices',  
                    'DriverName',  
                    'DRVFILE,FILE:');  
  StrCopy(S, 'PrinterPorts');  
  SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(@S));  
  StrCopy(S, 'Devices');  
  SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(@S));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16367: Simultaneous printing to multiple printers

Question and Answer Database

FAQ1367D.txt - Simultaneous printing to multiple printers

Category :Printing  
Platform :All Windows  
Product :

Question:

Is it possible to print to more than one printer at a given time?

Answer:

Yes! The following example demonstrates how to print to more than one printer at a given time. The exmple does not use TPrinter, since TPrinter does not support printing to multiple printers at a given time.

Example:

```
unit multiprn;

interface

{$IFDEF WIN32}
    uses
        Windows, Messages, SysUtils, Classes, Graphics, Controls,
        Forms, Dialogs, StdCtrls;
{$ELSE}
    uses
        SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
        Controls, Forms, Dialogs, StdCtrls;
{$ENDIF}

type
    TForm1 = class(TForm)
        Button1: TButton;
        ListBox1: TListBox;
        ListBox2: TListBox;
        ListBox3: TListBox;
```

```

procedure FormCreate(Sender: TObject);
procedure ListBox1Click(Sender: TObject);
procedure ListBox2Click(Sender: TObject);
procedure ListBox3Click(Sender: TObject);
procedure Button1Click(Sender: TObject);
private
  { Private declarations }
  procedure GetTheListOfPrinters;
public
  { Public declarations }
end;

```

```

var
  Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

type
  PdcArray = ^TdcArray;
  TdcArray = array[0..0] of hdc;

```

```

procedure TForm1.GetTheListOfPrinters;

```

```

var
  p : pChar;
  p2 : pChar;
  i : integer;
  sDriver : string;
  sPort : string;

```

```

begin
  GetMem(p, 32767);
  p2 := p;
  {Get a list of the printer names from the win.ini file.}
  {The list will be a buffer filled with strings, each seperated}
  {by a null character, with the final string terminated with}
  {a two null characters}
  if GetProfileString('devices', nil, '', p, 32767) <> 0 then begin
    {loop though the null terminated strings. We know we}
    {have reached the end when p2 equals a null character}
    while p2^ <> #0 do begin
      ListBox1.Items.Add(StrPas(p2));
      {increment the pointer past the null to get the next string}
      p2 := @p2[lStrLen(p2) + 1];
    end;
  end;

```



```

end;
GetMem(p2, 32767);
{Get the driver and port names for each printer found}
for i := 0 to (ListBox1.Items.Count - 1) do begin
  StrPCopy(p2, ListBox1.Items[i]);
  if GetProfileString('devices', p2, '', p, 32767) <> 0 then begin
    sDriver := StrPas(p);
    sPort := sDriver;
    {The driver is the portion of the string before the comma}
    Delete(sDriver, Pos(',', sDriver), Length(sDriver));
    {The port is the portion of the string after the comma}
    Delete(sPort, 1, Pos(',', sPort));
    ListBox2.Items.Add(sDriver);
    ListBox3.Items.Add(sPort);
  end;
end;
FreeMem(p2, 32767);
FreeMem(p, 32767);
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  GetTheListOfPrinters;
  {Allow the user to select multiple printers}
  ListBox1.MultiSelect := true;
  ListBox2.MultiSelect := true;
  ListBox3.MultiSelect := true;
end;

```

```

procedure TForm1.ListBox1Click(Sender: TObject);
var
  i : integer;
begin
  {Select same indexes in ListBoxes 2 and 3 as are selected in Listbox1}
  for i := 0 to ListBox1.Items.Count - 1 do begin
    ListBox2.Selected[i] := ListBox1.Selected[i];
    ListBox3.Selected[i] := ListBox1.Selected[i];
  end;
end;

```

```

procedure TForm1.ListBox2Click(Sender: TObject);
var
  i : integer;
begin
  {Select same indexes in ListBoxes 1 and 3 as are selected in Listbox2}

```

```

for i := 0 to ListBox2.Items.Count - 1 do begin
    ListBox1.Selected[i] := ListBox2.Selected[i];
    ListBox3.Selected[i] := ListBox2.Selected[i]
end;
end;

procedure TForm1.ListBox3Click(Sender: TObject);
var
    i : integer;
begin
    {Select same indexes in ListBoxes 1 and 2 as are selected in Listbox3}
    for i := 0 to ListBox3.Items.Count - 1 do begin
        ListBox1.Selected[i] := ListBox3.Selected[i];
        ListBox2.Selected[i] := ListBox3.Selected[i]
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    dcs : PdcArray; {a pointer to an array of hdc's}
    dcsCount : integer; {count of valid hdc's}
    dc : hdc; {test hdc}
    i : integer; {counting variable}
    pPrinter : pChar; {pointer to a printer name}
    pDriver : pChar; {pointer to a printer driver filename}
    pPort : pChar; {pointer to a port}
    DocInfo: TDocInfo; {document information for the spooler}
    {$IFDEF WIN32}
        osv : TOSVERSIONINFO; {to test for Windows NT}
    {$ENDIF}
begin
    {If there are no printers selected then exit}
    if ListBox1.SelCount = 0 then exit;
    {If Range Checking is on then remember the setting and turn it off}
    {This will allow use to access the dcs[i] array past element zero}
    {without a compiler or runtime error}
    {$IFOPT R+}
        {$DEFINE CKRANGE}
        {$R-}
    {$ENDIF}
    {Allocate the array of dcs}
    GetMem(dcs, sizeof(hdc) * ListBox1.SelCount);
    dcsCount := 0;
    {Loop through the printers that are selected and create dc's for each}
    for i := 0 to (ListBox1.Items.Count - 1) do begin

```

```

{Loop through the list box to find the selected printers}
if ListBox1.Selected[i] then begin
  {Allocate and get the Printer, Driver and port from the listboxes}
  {in the form of a null terminated string}
  GetMem(pPrinter, Length(ListBox1.Items[i]) + 1);
  GetMem(pDriver, Length(ListBox2.Items[i]) + 1);
  GetMem(pPort, Length(ListBox3.Items[i]) + 1);
  StrPCopy(pPrinter, ListBox1.Items[i]);
  StrPCopy(pDriver, ListBox2.Items[i]);
  StrPCopy(pPort, ListBox3.Items[i]);
  {Attempt to create a dc - notes 1) Not all printers can}
  {support multiple dc's at one time. 2) CreateDc requires}
  {different parameters if Win32 andor Win32 under NT.}
  {Sixteen bit win apps get standard handling}
  {$IFDEF WIN32}
    GetVersionEx(osv);
    if osv.dwPlatformId = VER_PLATFORM_WIN32_NT then
      dc := CreateDc(pDriver, pPrinter, nil, nil) else
      dc := CreateDc(nil, pPrinter, nil, nil);
  {$ELSE}
    dc := CreateDc(pDriver, pPrinter, pPort, nil);
  {$ENDIF}
  FreeMem(pPrinter, Length(ListBox1.Items[i]));
  FreeMem(pDriver, Length(ListBox2.Items[i]));
  FreeMem(pPort, Length(ListBox3.Items[i]));
  {If the dc is valid, then lets save it to our array}
  {and increment our count}
  if dc <> 0 then begin
    dcs^[dcsCount] := dc;
    inc(dcsCount);
  end;
end;
end;
{If we ended up with a valid array of dc's then let's print}
if dcsCount > 0 then begin

  {Fill Out the DocInfo structure for the spooler}
  {and start a document for each printer}
  GetMem(DocInfo.lpszDocName, 32);
  for i := 0 to (dcsCount - 1) do begin
    FillChar(DocInfo, sizeof(DocInfo), #0);
    DocInfo.cbSize := SizeOf(DocInfo);
    StrPCopy(DocInfo.lpszDocName, 'Test Doc' + IntToStr(i));
    StartDoc(dcs^[i], DocInfo);
  end;
end;

```

```
FreeMem(DocInfo.lpszDocName, 32);
```

```
{Start a page for each printer}  
for i := 0 to (dcsCount - 1) do  
    StartPage(dcs^[i]);  
{Print something}  
for i := 0 to (dcsCount - 1) do  
    TextOut(dcs^[i], 200, 200, 'Test', 4);
```

```
{End a page for each printer}  
for i := 0 to (dcsCount - 1) do  
    EndPage(dcs^[i]);
```

```
{End the document for each printer}  
for i := 0 to (dcsCount - 1) do  
    EndDoc(dcs^[i]);
```

```
{Delete the dc for each printer}  
for i := 0 to (dcsCount - 1) do  
    DeleteDc(dcs^[i]);
```

```
end; {dcsCount}
```

```
{Free our array of printers}  
FreeMem(dcs, sizeof(hdc) * ListBox1.SelCount);
```

```
{If range checking was on when we started then turn it back on}  
{$IFDEF CKRANGE}  
    {$UNDEF CKRANGE}  
    {$R+}  
{$ENDIF}  
end;
```

```
end.
```

4/2/99 11:23:18 AM

Last Modified: 01-SEP-99

## Article #16373: Stack Overflow, Runtime error 202;

Question and Answer Database

FAQ1373D.txt Stack Overflow, Runtime error 202;

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I avoid getting a Stack overflow runtime error 202?

Answer:

Simply put, stack overflows are caused by putting too much on the stack. Usually, they are caused by recursive procedures that never end. A good example would be creating an event handler for the TMemo's onChange event, and making a change to the Memo during the processing of the event. Every time the OnChange event gets fired, another change is made, so the OnChange event gets fired again in an almost endless loop. The loop finally ends when the stack overflows, and the application crashes.

Here is an example of a recursive procedure:

```
procedure RecursiveBlowTheStack;  
begin  
  RecursiveBlowTheStack;  
end;
```

Sometimes, a stack overflow is caused by too many large procedures. Each procedure calls another procedure, until the stack simply overflows. This can be remedied by breaking up large procedures into smaller ones. A good rule of thumb in regard to a procedures size is if the procedure's source code takes up more than a screen, its time to break it down into smaller procedures.

Finally, stack overflows can be caused by creating very large local variables inside a procedure, or passing a large variable by value to another procedure. Consider the passing of string variables. If the string is 255 characters (plus the length byte), if passed by value, you are actually taking up 256 bytes off the stack. If the procedure you are calling passes the string by value to yet another procedure,

the string now takes 512 bytes of stack space. Passing the string (or other variable) as a var or const parameter takes only 4 bytes, since var and const parameters are simply pointers to the actual data. You can also create large variables on the heap by dynamic allocation of the memory;

The following code demonstrates two procedures BlowTheStack(), and NoBlowTheStack(). The BlowTheStack procedure attempts to allocate a large local variable designed to be large enough to crash the application. The NoBlowTheStack() procedure allocates the same large variable but allocates it on the heap so the function will succeed.

```
type
  PBigArray = ^TBigArray;
{$IFDEF WIN32}
  TBigArray = array[0..10000000] of byte;
{$ELSE}
  TBigArray = array[0..64000] of byte;
{$ENDIF}
```

```
procedure BlowTheStack;
```

```
var
```

```
  BigArray : TBigArray;
```

```
begin
```

```
  BigArray[0] := 10;
```

```
end;
```

```
procedure NoBlowTheStack;
```

```
var
```

```
  BigArray : PBigArray;
```

```
begin
```

```
  GetMem(BigArray, sizeof(BigArray^));
```

```
  BigArray^[0] := 10;
```

```
  FreeMem(BigArray, sizeof(BigArray^));
```

```
end;
```

Finally, the following code demonstrates creating procedures that accept large variables as parameters. The PassByValueAnCrash() procedure is designed to crash since the value parameter is too large for the stack. The PassByVar(), PassByPointer(), and PassByConst will succeed, since these procedures only use 4 bytes of stack space. Note that you cannot modify a parameter passed as const, as a const parameter is assumed to be read only.

Example:

```
procedure PassByValueAnCrash(BigArray : TBigArray);  
begin  
    BigArray[0] := 10;  
end;
```

```
procedure PassByVar(var BigArray : TBigArray);  
begin  
    BigArray[0] := 10;  
end;
```

```
procedure PassByPointer(BigArray : PBigArray);  
begin  
    PBigArray^[0] := 10;  
end;
```

```
procedure PassByConst(const BigArray : TBigArray);  
begin  
    ShowMessage(IntToStr(BigArray[0]));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16379: Closing popup menu in system tray application

Question and Answer Database

FAQ1379D.txt Closing popup menu in system tray application

Category :Windows API

Platform :All

Product :All 32 bit

Question:

Sometimes, my popup menu in my tray application does not close when it loses focus. How can I get it to close?

Answer:

When processing the message to pop up the menu, you need to set the foreground window, then send a WM\_NULL message after the menu popup.

```
procedure TForm1.WndProc(var Msg : TMessage);
```

```
var
  p : TPoint;
begin
  case Msg.Msg of
    WM_USER + 1:
      case Msg.lParam of
        WM_RBUTTONDOWN: begin
          SetForegroundWindow(Handle);
          GetCursorPos(p);
          PopupMenu1.Popup(p.x, p.y);
          PostMessage(Handle, WM_NULL, 0, 0);
        end;
      end;
    end;
  inherited;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16385: Keeping a glyph's transparent area transparent.

Question and Answer Database

FAQ1385D.txt Keeping a glyph's transparent area transparent.

Category :VCL

Platform :All

Product :All 32 bit

Question:

Under some video modes, the transparent area of the standard TBitBtn glyphs turns gold or maroon. How can I keep the standard TBitBtn's glyph transparent?

Answer:

The following example demonstrates a technique where the transparent portion of the glyph is filled with the background color of the button, making the glyph appear transparent.

Example:

```
function InitStdBitBtn(BitBtn : TBitBtn;
                      kind : TBitBtnKind) : bool;
var
  Bm1 : TBitmap;
  Bm2 : TBitmap;
begin
  Result := false;
  if Kind = bkCustom then exit;
  Bm1 := TBitmap.Create;
  case Kind of
    bkOK : Bm1.Handle := LoadBitmap(hInstance, 'BBOK');
    bkCancel : Bm1.Handle := LoadBitmap(hInstance, 'BBCANCEL');
    bkHelp : Bm1.Handle := LoadBitmap(hInstance, 'BBHELP');
    bkYes : Bm1.Handle := LoadBitmap(hInstance, 'BBYES');
    bkNo : Bm1.Handle := LoadBitmap(hInstance, 'BBNO');
    bkClose : Bm1.Handle := LoadBitmap(hInstance, 'BBCLOSE');
    bkAbort : Bm1.Handle := LoadBitmap(hInstance, 'BBABORT');
    bkRetry : Bm1.Handle := LoadBitmap(hInstance, 'BBRETRY');
    bkIgnore : Bm1.Handle := LoadBitmap(hInstance, 'BBIGNORE');
    bkAll : Bm1.Handle := LoadBitmap(hInstance, 'BBALL');
  end;
end;
```

```
Bm2 := TBitmap.Create;
Bm2.Width := Bm1.Width;
Bm2.Height := Bm1.Height;
bm2.Canvas.Brush.Color := ClBtnFace;
bm2.Canvas.BrushCopy(Rect(0, 0, bm2.Width, bm2.Height),
                    Bm1,
                    Rect(0, 0, Bm1.Width, Bm1.Height),
                    bm1.Canvas.Pixels[0,0]);

bml.Free;
LockWindowUpdate(BitBtn.Parent.Handle);
BitBtn.Kind := kind;
BitBtn.Glyph.Assign(bm2);
LockWindowUpdate(0);
bm2.Free;
Result := true;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    InitStdBitBtn(BitBtn1, bkOk);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16390: Transparent property to work for a CoolBar background bitmap

Question and Answer Database

FAQ1390D.txt      Transparent property to work for a CoolBar background  
bitmap

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I get the transparent property to work for a background  
bitmap assigned to a CoolBar component?

Answer:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    Bm1 : TBitmap;
```

```
    Bm2 : TBitmap;
```

```
begin
```

```
    Bm1 := TBitmap.Create;
```

```
    Bm2 := TBitmap.Create;
```

```
    Bm1.LoadFromFile('c:\download\test.bmp');
```

```
    Bm2.Width := Bm1.Width;
```

```
    Bm2.Height := Bm1.Height;
```

```
    bm2.Canvas.Brush.Color := CoolBar1.Color;
```

```
    bm2.Canvas.BrushCopy(Rect(0, 0, bm2.Width, bm2.Height),
```

```
                          Bm1,
```

```
                          Rect(0, 0, Bm1.width, Bm1.Height),
```

```
                          ClWhite);
```

```
    bm1.Free;
```

```
    CoolBar1.Bitmap.Assign(bm2);
```

```
    bm2.Free;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16395: How can I determining if a given machine has a math coprocessor.

Question and Answer Database

FAQ1395D.txt    How can I determining if a given machine has a math coprocessor.

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I determine in code if a given machine either has a math coprocessor or if Windows is supplying emulation?

Answer:

Contrary to popular belief, not all 486/586/686/ and Pentium clones have floating point units. Under Windows 3.1, applications could take advantage of this knowledge and choose to do fixed point math or use other means to improve performance. The following example shows how to check for the existance of a floating point unit under both Win16 and Win32.

Example:

```
{ $IFDEF WIN32 }
  uses Registry;
{ $ENDIF }
```

```
function HasCoProcessor : bool;
```

```
{ $IFDEF WIN32 }
```

```
var
```

```
  TheKey : hKey;
```

```
{ $ENDIF }
```

```
begin
```

```
  Result := true;
```

```
{ $IFNDEF WIN32 }
```

```
  if GetWinFlags and Wf_80x87 = 0 then
```

```
    Result := false;
```

```
{ $ELSE }
```

```
  if RegOpenKeyEx(HKEY_LOCAL_MACHINE,
```

```
    'HARDWARE\DESCRIPTION\System\FloatingPointProcessor',
```

```
    0,
```

```
KEY_EXECUTE,  
TheKey) <> ERROR_SUCCESS then result := false;  
RegCloseKey(TheKey);  
{ $ENDIF }  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  if HasCoProcessor then  
    ShowMessage('Has CoProcessor') else  
    ShowMessage('No CoProcessor - Windows Emulation Mode');  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16400: How can I dial out through the modem under Win32?

Question and Answer Database

FAQ1400D.txt    How can I dial out through the modem under Win32?

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I dial out through the modem under Win32?

Answer:

You can use the Windows API function CreateFile() to get a handle to the com port, and use standard file I/O to communicate with the given port.

Example:

```
var
  hCommFile : THandle;

procedure TForm1.Button1Click(Sender: TObject);
var
  PhoneNumber : string;
  CommPort    : string;
  NumberWritten : LongInt;
begin
  PhoneNumber := 'ATDT 1-555-555-1212' + #13 + #10;
  CommPort    := 'COM2';
  {Open the comm port}
  hCommFile := CreateFile(PChar(CommPort),
                          GENERIC_WRITE,
                          0,
                          nil,
                          OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL,
                          0);

  if hCommFile=INVALID_HANDLE_VALUE then
  begin
    ShowMessage('Unable to open '+ CommPort);
    exit;
```

```
end;
```

```
{Dial the phone}
```

```
NumberWritten:=0;
```

```
if WriteFile(hCommFile,
```

```
    PChar(PhoneNumber)^,
```

```
    Length(PhoneNumber),
```

```
    NumberWritten,
```

```
    nil) = false then begin
```

```
    ShowMessage('Unable to write to ' + CommPort);
```

```
end;
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
```

```
    {Close the port}
```

```
    CloseHandle(hCommFile);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16405: Retrieving BDE system information

Question and Answer Database

FAQ1405D.txt Retrieving BDE system information

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How can I get BDE system status such as size of buffer and heap used, or number of currently loaded drivers, active clients, sessions, databases or cursors?

Answer:

Use the BDE api call DbiGetSysInfo. The following example outputs the results to a text log file.

implementation

```
uses dbierrs, dbiprocs, dbitypes;
```

```
{ $R *.DFM }
```

```
procedure LogDbiSysInfo(const LogFile: string);
```

```
var
```

```
    Info: SYSInfo;
```

```
begin
```

```
    DbiGetSysInfo(Info);
```

```
    with TStringList.Create do
```

```
    try
```

```
        Clear;
```

```
        Add(Format('BUFFER SPACE: %d', [Info.iBufferSize]));
```

```
        Add(Format('HEAP SPACE: %d', [Info.iHeapSpace]));
```

```
        Add(Format('DRIVERS: %d', [Info.iDrivers]));
```

```
        Add(Format('CLIENTS: %d', [Info.iClients]));
```

```
        Add(Format('SESSIONS: %d', [Info.iSessions]));
```

```
        Add(Format('DATABASES: %d', [Info.iDatabases]));
```

```
        Add(Format('CURSORS: %d', [Info.iCursors]));
```

```
        SaveToFile('c:\logcurs.txt');
```

```
    finally
```

```
        Free;
```



```
end;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    table1.Open;  
    table2.open;  
    table3.Open;  
    LogDbiSysInfo('c:\logcurs.txt');  
end;  
  
end.
```

Output for logcurs.txt:

-----

```
BUFFER SPACE: 128  
HEAP SPACE:   160  
DRIVERS:      2  
CLIENTS:      2  
SESSIONS:     2  
DATABASES:    4  
CURSORS:      3
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16409: What does the error 'Token not found'

Question and Answer Database

FAQ1409C.txt    What does the error 'Token not found'

Category        :Database Issues

Platform        :All

Product         :C++Builder 1.x

Question:

I have a TQuery and TDataSource. In the TStrings property of the TQuery I have 'SELECT \* FROM dbo.AnyTable' where dbo is the database on my SQL server. When I set active to TRUE I get the error: 'Token not found. Token :dbo. line number:1'.

What's wrong?

Answer:

If the RequestLive property is set to true, then enclose the owner and tables names in quotes:

Ex:

```
SELECT * FROM "dbo.table"
```

If request live is false, don't use quotes.

Ex:

```
SELECT * FROM dbo.table
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16410: How can I use a true DWORD value passed to my Delphi application?

Question and Answer Database

FAQ1410D.txt     How can I use a true DWORD value passed to my Delphi application?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I use a true DWORD value passed to my Delphi application?

Answer:

One solution is to convert the DWORD to a comp.

Example:

```
{ $IFOPT Q+ }
  { $DEFINE OVERFLOWSON }
  { $Q- }
{ $ENDIF }

function DWordToComp(dw : longint) : comp;
var
  c : comp;
begin
  if u >= 0 then
    c := dw else begin
    c := $7FFFFFFF;
    if dw = -1 then
      c := c + c + 1
    else
      c := c + abs($7FFFFFFF - dw);
    end;
  result := c;
end;

{ $IFDEF OVERFLOWSON }
  { $UNDEF OVERFLOWSON }
  { $Q+ }
{ $ENDIF }
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16415: Going to the next compiler error

Question and Answer Database

FAQ1415D.txt    Going to the next compiler error

Category       :IDE

Platform       :All

Product        :All 32 bit

Question:

Is there a faster way to go to the next compiler error than clicking the error with the mouse?

Answer:

Yes. The Alt-F8 will take you to the next compiler error, and Alt-F7 will take you to the previous error.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16420: Changing the input caret

Question and Answer Database

FAQ1420D.txt Changing the input caret

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to change the caret in an edit or other windows control capable of accepting keyboard input in response to a keyboard action?

Answer:

Yes! The following example demonstrates how to create two color bitmaps: a "smiley" and a "frowny" face bitmap, and attach them to the caret of an edit control. The edit control's window procedure must be trapped to do this to create a superclass condition when responding to windows focus messages. To do this we will replace the address of the Edit control's main window procedure with our own, and make calls to the Edit control's old window procedure as necessary. The example shows the "smiley" face caret when normally typing, and displays a "frowning" face caret when you backspace.

Example:

```
unit caret1;
```

```
interface
```

```
{ $IFDEF WIN32 }
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls;
```

```
{ $ELSE }
```

```
uses
```

```
  WinTypes, WinProcs, Messages, SysUtils, Classes, Graphics,  
  Controls, Forms, Dialogs, StdCtrls;
```

```
{ $ENDIF }
```

```

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    CaretBm : TBitmap;
    CaretBmBk : TBitmap;
    OldEditsWindowProc : Pointer;
  end;

```

```

var
  Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

type
{$IFDEF WIN32}
  WParameter = LongInt;
{$ELSE}
  WParameter = Word;
{$ENDIF}
  LParameter = LongInt;

```

```

{New windows procedure for the edit control}

```

```

function NewWindowProc(WindowHandle : hWnd;
                        TheMessage   : WParameter;
                        ParamW       : WParameter;
                        ParamL       : LParameter) : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}

```

```

begin
  {Call the old edit controls windows procedure}
  NewWindowProc := CallWindowProc(Form1.OldEditsWindowProc,
                                   WindowHandle,
                                   TheMessage,
                                   ParamW,
                                   ParamL);

  if TheMessage = WM_SETFOCUS then begin
    CreateCaret(WindowHandle, Form1.CaretBm.Handle, 0, 0);

```





```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  {Unhook the edit controls window procedure and clean up}
  SetWindowLong(Edit1.Handle,
                GWL_WNDPROC,
                LongInt(OldEditsWindowProc));
  CaretBm.Free;
  CaretBmBk.Free;
end;

end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16425: Is there any way to record a keyboard macro in the IDE?

Question and Answer Database

FAQ1425D.txt Is there any way to record a keyboard macro in the IDE?

Category :IDE

Platform :All

Product :All 32 bit

Question:

Is there any way to record a keyboard macro in the IDE?

Answer:

Yes! Depending on your editor setting in the IDE environment options, the following keystrokes will record and play back macros:

Default

Ctrl+Shift+P Plays back a key macro

Ctrl+Shift+R Starts and stops the recording a key macro

Classic

Ctrl+Shift+P Plays back a keyboard macro

Ctrl+Shift+R Starts and stops the recording a key macro

Brief

F7 Starts and Stops the recording a key macro

F8 Plays back a keyboard macro

Epsilon

Ctrl+X+( Records a keyboard macro

Ctrl+X+) Ends a keyboard macro recording

Ctrl+X+e Plays back the the last keyboard macro recorded

Ctrl+X+E Plays back the the last keyboard macro recorded

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16430: How can I tell in code if my application is running on Windows NT

Question and Answer Database

FAQ1430D.txt    How can I tell in code if my application is running on Windows NT

Category        :VCL  
Platform        :All  
Product         :All 32 bit

Question:

How can I tell in code if my application is running on Windows NT?

Answer:

The following example demonstrates how to detect if your application is running under Windows NT. Note that the code will compile under both 16 and 32 bit platforms.

Example:

```
{ $IFDEF WIN32 }
  const WF_WINNT = $4000;
{ $ENDIF }

function IsNT : bool;
{ $IFDEF WIN32 }
  var
    osv : TOSVERSIONINFO;
{ $ENDIF }
begin
  result := true;
{ $IFDEF WIN32 }
  GetVersionEx(osv);
  if osv.dwPlatformId = VER_PLATFORM_WIN32_NT then exit;
{ $ELSE }
  if ((GetWinFlags and WF_WINNT) = WF_WINNT ) then exit;
{ $ENDIF }
  result := false;
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  if IsNt then
    ShowMessage('Running on NT')
  else
    ShowMessage('Not Running on NT');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16435: How can I detect if a given key was pressed during application during startup?

Question and Answer Database

FAQ1435D.txt How can I detect if a given key was pressed during application during startup?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I detect if a given key was pressed during application startup?

Answer:

Use the Windows API function `GetKeyState()` to determine if a key is pressed in the main project source. To get to the main project's source file, select from the IDE's main menu, select "View">>"ProjectSource".

Example:

```
program Project1;

uses
  Windows,
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  if GetKeyState(vk_F8) < 1 then
    MessageBox(0, 'F8 was pressed during startup', 'MyApp', mb_ok);
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

7/16/98 4:31:28 PM



## Article #16440: Duplicate resource errors

Question and Answer Database

FAQ1440D.txt Duplicate resource errors

Category :Miscellaneous

Platform :All

Product :All 32 bit

### Question:

How can I avoid getting duplicate resource errors when linking a language dependent resource file under Delphi 3.0 or Borland C++ Builder?

### Answer:

If the resource identifier has an even number of characters, you may get duplicate resource errors. Rename the resource to a name that has an odd number of characters.

Example resource scripts:

```
{This compiles fine}
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US
Hello BITMAP "hienglsh.bmp"
```

```
LANGUAGE LANG_SPANISH, SUBLANG_SPANISH_MEXICAN
Hello BITMAP "hispansh.bmp"
```

```
{This does not compile}
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US
Helo BITMAP "hienglsh.bmp"
```

```
LANGUAGE LANG_SPANISH, SUBLANG_SPANISH_MEXICAN
Helo BITMAP "hispansh.bmp"
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16445: How can I detect if a non-menu key has been pressed?

Question and Answer Database

FAQ1445D.txt How can I detect if a non-menu key has been pressed?

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I detect if a non-menu key has been pressed while a menu is displayed?

Answer:

Create a message handler for the WM\_MENUCHAR character message.

Example:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, Menus;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    MainMenu: TMainMenu;
```

```
    One1: TMenuItem;
```

```
    Two1: TMenuItem;
```

```
    Three1: TMenuItem;
```

```
  private
```

```
    { Private declarations }
```

```
    procedure WmMenuChar(var m : TMessage); message WM_MENUCHAR;
```

```
  public
```

```
    { Public declarations }
```

```
end;
```

```
var
```

```
  Form1: TForm1;
```



implementation

```
{ $R *.DFM }
```

```
procedure TForm1.WmMenuChar(var m : TMessage);  
begin  
  Form1.Caption := 'Non standard menu key pressed';  
  m.Result := 1;  
end;  
  
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16450: Changing the default project directory

Question and Answer Database

FAQ1450D.txt Changing the default project directory

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I change the default directory Delphi uses to save projects in?

Answer:

Right click on the Windows shortcut icon used to start the program. Select properties from the menu, and change the default working directory the program starts in.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16455: Getting bin names from your printer

Question and Answer Database

FAQ1455D.txt - Getting bin names from your printer

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How do I retrieve the bin names of all bins for a particular printer using DeviceCapabilities?

Answer:

Here's a very simple app to do this. Just start a brand new project in Delphi and drop down a TButton and a TMemo. Then double click the button and paste this code snippet over the corresponding code in the implementation of Unit1.

```
uses
  WinSpool;

const
  PName = 'DELPHI III';
  PPort = '\\CONAN\DELPHI_III';

procedure TForm1.Button1Click(Sender: TObject);
var
  i : Integer;
  p : PChar;
begin
  GetMem(p,24*DeviceCapabilities(PName,PPort,DC_BINNAMES,nil,nil));
  with Mem01, Lines do begin
    Clear;
    for i:=1 to DeviceCapabilities(PName,PPort,DC_BINNAMES,p,nil) do
      Add(p+24*(i-1));
    end;
  FreeMem(p);
end;
```

4/2/99 11:23:40 AM

Last Modified: 01-SEP-99

## Article #16460: Changing the port of a printer

Question and Answer Database

FAQ1460D.txt - Changing the port of a printer

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How can I change the port a given printer will use in code?

Answer:

You can use the SetPrinter method of TPrinter to change the port. The following example demonstrates changing the port to print to a file.

Example:

```
uses Printers;

{$IFDEF WIN32}
  const MAX_PATH = 144;
{$ENDIF}

procedure TForm1.Button1Click(Sender: TObject);
var
  pDevice : pChar;
  pDriver : pChar;
  pPort   : pChar;
  hDMode  : THandle;
  PDMode  : PDEVMODE;
begin
  if PrintDialog1.Execute then begin
    GetMem(pDevice, cchDeviceName);
    GetMem(pDriver, MAX_PATH);
    GetMem(pPort, MAX_PATH);
    Printer.GetPrinter(pDevice, pDriver, pPort, hDMode);
    Printer.SetPrinter(pDevice, PDriver, 'FILE:', hDMode);
    FreeMem(pDevice, cchDeviceName);
```

```
FreeMem(pDriver, MAX_PATH);  
FreeMem(pPort, MAX_PATH);  
Printer.BeginDoc;  
Printer.Canvas.TextOut(100, 100, 'Delphi Is RAD!');  
Printer.EndDoc;  
end;  
end;
```

4/2/99 11:23:47 AM

Last Modified: 01-SEP-99

## Article #16465: Getting the actual size of a TComboBox

Question and Answer Database

FAQ1465D.txt Getting the actual size of a TComboBox

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I tell the actual size of a TComboBox in it's dropped down state before it drops down?

Answer:

During the FormShow event, send the combo box a CB\_SHOWDROPDOWN message twice, once to drop it down, and once to retract it. Then send a CB\_GETDROPPEDCONTROLRECT message, passing the address of a TRect. When the SendMessage call returns, the TRect will contain the rectangle that would enclose the ComboBox in it's dropped down state, relative to the screen. You can then call ScreenToClient to convert the TRect's coordinates to be relative to the client area of the form.

Example:

```
var
  R : TRect;

procedure TForm1.FormShow(Sender: TObject);
var
  T : TPoint;
begin
  SendMessage(ComboBox1.Handle,
              CB_SHOWDROPDOWN,
              1,
              0);
  SendMessage(ComboBox1.Handle,
              CB_SHOWDROPDOWN,
              0,
              0);
  SendMessage(ComboBox1.Handle,
              CB_GETDROPPEDCONTROLRECT,
```

```
        0,  
        LongInt(@r));  
t := ScreenToClient(Point(r.Left, r.Top));  
r.Left := t.x;  
r.Top := t.y;  
t := ScreenToClient(Point(r.Right, r.Bottom));  
r.Right := t.x;  
r.Bottom := t.y;  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form1.Canvas.Rectangle(r.Left,  
                           r.Top,  
                           r.Right,  
                           r.Bottom );  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16470: Converting a number from one base to another

Question and Answer Database

FAQ1470D.txt     Converting a number from one base to another  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

Question:

How can I convert a number from one base to another base?

Answer:

The following function will convert a number from one base to a number of another base:

```
procedure RadixStr(NumStr : pChar;  
                  Radix  : LongInt;  
                  ResultStr : pChar;  
                  NewRadix : LongInt;  
                  var ErrorCode : LongInt);
```

The RadixStr() function takes a pointer to a null terminated string containing a number of one base, and fills a buffer with a null terminated string containing the number converted to another base.

Parameters:

NumStr: A pointer to a null terminated string containing the numeric string to convert:

Radix: The base of the number contained in the NumStr parameter. The base must be in the range of 2 to 36;

ResultStr : A pointer to a null terminated string buffer to place the resulting numeric string. The buffer should be sufficiently large to hold the resulting string.

NewRadix: The base to use in the conversion. The base must be in the range of 2 to 36;

ErrorCode: Upon return, contains the return code 0 if successful, or the character number of the offending character contained in the buffer NumStr.

Examples of calling the RadixStr() function:

```
{Convert Hex to Decimal}
RadixStr('FF',
        16,
        lpBuffer,
        10,
        Code);
```

Should return the string '255' in lpbuffer^.

```
{Convert Decimal to Binary}
RadixStr('255',
        10,
        lpBuffer,
        2,
        Code);
```

Should return the string '11111111' in lpbuffer^.

```
{Convert Hex to Octal}
RadixStr('FF',
        16,
        lpBuffer,
        8,
        Code);
```

Should return the string '377' in lpbuffer^.

```
{Function code}
```

```
procedure RadixStr(NumStr : pChar;
                  Radix : LongInt;
                  ResultStr : pChar;
                  NewRadix : LongInt;
                  var ErrorCode : LongInt);
```

```
var
    RadixChar : array[0..35] of Char;
    v : LongInt;
    i : LongInt;
    p : LongInt;
```

```

    c : Integer;
begin
    if ((Abs(Radix) < 2) or
        (Abs(Radix) > 36)) then begin
        ErrorCode := p;
        Exit;
    end;
    StrLCopy(ResultStr, NumStr, StrLen(NumStr));
    for i := 0 to 35 do begin
        if i <= 9 then
            RadixChar[i] := Char(48 + (i))
        else
            RadixChar[i] := Char(64 + (i - 9))
        end;
    end;
    v := 0;
    for i := 0 to (StrLen(ResultStr) - 1) do begin
        ResultStr[i] := UpCase(ResultStr[i]);
        p := Pos(ResultStr[i], PChar(@RadixChar)) - 1;
        if ((p < 0) or
            (p >= Abs(Radix))) then begin
            ErrorCode := i;
            Exit;
        end;
        v := v * Abs(Radix) + p;
    end;
    if v = 0 then begin
        ResultStr := '0';
        ErrorCode := 0;
        exit;
    end else begin
        i:=0;
        repeat
            ResultStr[i] := RadixChar[v mod NewRadix];
            v := v div NewRadix;
            Inc(i)
        until v = 0;
        if Radix < 0 then begin
            ResultStr[i] := '-';
            ResultStr[i + 1] := #0
        end else
            ResultStr[i] := #0;
        p := StrLen(ResultStr);
        for i := 0 to ((p div 2) - 1) do begin
            ResultStr[i] := Char(Byte(ResultStr[i]) xor
                Byte(ResultStr[(p - i) - 1]));

```

```
ResultStr[(p - i) - 1] := Char(Byte(ResultStr[(p - i) - 1]) xor  
                                Byte(ResultStr[i]));
```

```
ResultStr[i] := Char(Byte(ResultStr[i]) xor  
                    Byte(ResultStr[(p - i) - 1]))
```

```
end;
```

```
ResultStr[p] := #0;
```

```
ErrorCode := 0;
```

```
end;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16475: Getting printer capabilities

Question and Answer Database

FAQ1475D.txt - Getting printer capabilities

Category :Printing  
Platform :All  
Product :

Question:

How can I detect if a printer is capable of printing graphics, large bitmaps, and DIBs?

Answer:

You can call the `GetDeviceCaps` function and test the capability against the `RASTERCAPS` capability. Note that if the Device does support graphics, but does not support another raster capability such as "StretchDib", Windows will handle the call by using lower level graphic primitives to accomplish the task.

Example:

```
if (GetDeviceCaps(Printer.Canvas.Handle, RASTERCAPS) and
    RC_NONE) = RC_NONE then
NoGraphics := True;

if (GetDeviceCaps(Printer.Canvas.Handle, RASTERCAPS) and
    RC_STRETCHDIB) = RC_STRETCHDIB then
DeviceDirectly SupportsStretchDIB := True else
WindowsWillSimulateStretchDIB := True;
```

4/2/99 11:23:56 AM

Last Modified: 01-SEP-99

## Article #16480: Changing the location of the object repository

Question and Answer Database

FAQ1480D.txt Changing the location of the object repository

Category :IDE

Platform :All

Product :All 32 bit

Question:

Is it possible to specify a different location for the object repository?

Answer:

Modify the following registry entry:

HKEY\_CURRENT\_USER\Software\Borland\Delphi\2.0\Repository\BaseDir

HKEY\_CURRENT\_USER\Software\Borland\Delphi\3.0\Repository\BaseDir

and place the full path to the new location of the delphi32.dro file.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16485: Getting property information

Question and Answer Database

FAQ1485D.txt    Getting property information

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I tell if a given object has a given property at runtime?

Answer:

The following example shows how to determine if an object has a property available, and if so, how to change the property:

```
function HasProperty(Obj : TObject;
                    Prop : string) : PPropInfo;
begin
  Result := GetPropInfo(Obj.ClassInfo, Prop);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  p : pointer;
begin
  p := HasProperty(Button1, 'Color');
  if p <> nil then
    SetOrdProp(Button1, p, clRed) else
    ShowMessage('Button has no color property');

  p := HasProperty(Label1, 'Color');
  if p <> nil then
    SetOrdProp(Label1, p, clRed) else
    ShowMessage('Label has no color property');

  p := HasProperty(Label1.Font, 'Color');
  if p <> nil then
    SetOrdProp(Label1.Font.Color, p, clBlue) else
    ShowMessage('Label.Font has no color property');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16490: Moving controls at runtime

Question and Answer Database

FAQ1490D.txt Moving controls at runtime

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I move a control at runtime using the mouse?

Answer:

You can trap the OnMouseDown event, remembering the x and y coordinates of the mouse and set the mouse capture. You can then track mouse movements with the OnMouseMove event, moving the control until the OnMouseUp event is fired. You then will want to place the control in it's final position and release capture of the mouse.

The following example shows how to move a TButton component around on the form using the mouse. The movement is initiated by the user holding down the control key when the mouse button is activated.

Example:

type

```
TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure Button1MouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
  procedure Button1MouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
private
  { Private declarations }
public
  { Public declarations }
  MouseDownSpot : TPoint;
```

```

    Capturing : bool;
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1MouseDown(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    if ssCtrl in Shift then begin
        SetCapture(Button1.Handle);
        Capturing := true;
        MouseDownSpot.X := x;
        MouseDownSpot.Y := Y;
    end;
end;

procedure TForm1.Button1MouseMove(Sender: TObject;
Shift: TShiftState; X, Y: Integer);
begin
    if Capturing then begin
        Button1.Left := Button1.Left - (MouseDownSpot.x - x);
        Button1.Top := Button1.Top - (MouseDownSpot.y - y);
    end;
end;

procedure TForm1.Button1MouseUp(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
    if Capturing then begin
        ReleaseCapture;
        Capturing := false;
        Button1.Left := Button1.Left - (MouseDownSpot.x - x);
        Button1.Top := Button1.Top - (MouseDownSpot.y - y);
    end;
end;

```



## Article #16495: re-mark a Delphi 16-Bit executable to be a Windows 4.0 application

Question and Answer Database

FAQ1495D.txt - re-mark a Delphi 16-Bit executable to be a Windows 4.0 application

Category :Delphi 1.x  
Platform :All Windows  
Product :

Question:

Is it possible to re-mark a Delphi 16-Bit executable to be a Windows 4.0 application, and why have I been instructed to do this? Is it also possible to tell what kind of executable image a given file is and it's expected windows version before I load it?

Answer:

There are several reasons why you would want to mark a 16-Bit executable to be a Windows 4.0 application.

- 1) Thunks created using the Microsoft Thunk Compiler will not work unless the 16-bit thunk DLL is marked for version 4.0.
- 2) Print drivers and other sub system driver files created for Windows 95 must be marked for version 4.0 even though they may be 16-bit executable images.
- 3) All standard control windows owned by a version 4.0 application are drawn with a chiseled 3D look.
- 4) Applications marked for Windows version 4.0 receive the full benefit of new user interface features available in Windows 95.
- 5) New window messages, such as WM\_STYLECHANGED, are sent to applications that are marked as a version 4.0 application.
- 6) Printer DCs are reset during StartPage() in version 4.0 applications.
- 7) Executables marked as a Windows version 4.0 executable will load on Windows 95, Windows NT versions 3.5, Win32s version 1.15, and all

later versions, but will not load on earlier versions of Windows such as Windows 3.1.

Note: Windows applications marked as 3.1 executables allow other applications to share GDI resources such as font or bitmap handles. Windows 95 does not clean up objects left undeleted by a 16-bit version 3.x application. Objects created by a 16-bit version 4.0-based applications are not shared, and are freed automatically when the application terminates.

The following is an example of how to read, examine, and remark the expected windows version of any Windows executable, dynamic link library, or .vxd file. Note: Any non-zero file checksums are not recalculated. You cannot mark a file that is currently in use. To build the project, you will need to create a form that contains two buttons, one label, an OpenFileDialog component, and two spin edit components.

Example:

```
unit Unit1;

interface

{$IFDEF WIN32}
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Spin;
{$ELSE}
uses
  WinTypes, WinProcs, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Spin;
{$ENDIF}

type
  TForm1 = class(TForm)
    SpinEdit1: TSpinEdit;
    SpinEdit2: TSpinEdit;
    Button1: TButton;
    Button2: TButton;
    OpenFileDialog: TOpenDialog;
    Label1: TLabel;
  procedure FormCreate(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure Button2Click(Sender: TObject);
```

```
private
  { Private declarations }
  ExeType : word;
  hdroffset : longint;
  procedure GetExeHeader(FileName : string);
  procedure RemarkExeHeader(FileName : string);
public
  { Public declarations }
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
const IMAGE_UNKNOWN_SIGNATURE = $0;
const IMAGE_DOS_SIGNATURE      = $5A4D; {MZ}
const IMAGE_OS2_SIGNATURE      = $454E; {NE}
const IMAGE_OS2_SIGNATURE_LE   = $454C; {LE}
const IMAGE_VXD_SIGNATURE      = $454C; {LE}
const IMAGE_NT_SIGNATURE       = $4550; {PE}

const NE_IMAGE_DLL              = $8000; {File is a DLL}

const IMAGE_NT_OPTIONAL_HDR_MAGIC = $10B;
const IMAGE_ROM_OPTIONAL_HDR_MAGIC = $107;

const IMAGE_FILE_RELOCS_STRIPPED      = $0001; {Relocation info stripped}
const IMAGE_FILE_EXECUTABLE_IMAGE    = $0002; {File is executable}
const IMAGE_FILE_LINE_NUMS_STRIPPED  = $0004; {Line numbers stripped}
const IMAGE_FILE_LOCAL_SYMS_STRIPPED = $0008; {Local symbols stripped}
const IMAGE_FILE_BYTES_REVERSED_LO   = $0080; {machine word bytes rev}
const IMAGE_FILE_32BIT_MACHINE       = $0100; {32 bit word machine}
const IMAGE_FILE_DEBUG_STRIPPED      = $0200; {Debug info stripped}
const IMAGE_FILE_SYSTEM              = $1000; {System File}
const IMAGE_FILE_DLL                 = $2000; {File is a DLL}
const IMAGE_FILE_BYTES_REVERSED_HI   = $8000; {machine word bytes rev}

const IMAGE_FILE_MACHINE_UNKNOWN = $0;
const IMAGE_FILE_MACHINE_I386    = $14c; {Intel 386}
const IMAGE_FILE_MACHINE_R3000B  = $160; {MIPS big-endian}
const IMAGE_FILE_MACHINE_R3000L  = $162; {MIPS little-endian}
const IMAGE_FILE_MACHINE_R4000   = $166; {MIPS little-endian}
```

```
const IMAGE_FILE_MACHINE_R10000 = $168; {MIPS little-endian}
const IMAGE_FILE_MACHINE_ALPHA = $184; {Alpha_AXP}
const IMAGE_FILE_MACHINE_POWERPC = $1F0; {IBM PowerPC Little-Endian}
```

```
const IMAGE_NUMBEROF_DIRECTORY_ENTRIES = 16;
```

```
{ $IFDEF WIN32 }
type DWORD = LongInt;
{ $ENDIF }
```

```
type
  PIMAGE_DOS_HEADER = ^TIMAGE_DOS_HEADER;
  TIMAGE_DOS_HEADER = { $IFDEF WIN32 } packed { $ENDIF } record
    e_magic : word;           { Magic number }
    e_cblp : word;            { Bytes on last page of file }
    e_cp : word;              { Pages in file }
    e_crlc : word;           { Relocations }
    e_cparhdr : word;        { Size of header in paragraphs }
    e_minalloc : word;       { Minimum extra paragraphs needed }
    e_maxalloc : word;       { Maximum extra paragraphs needed }
    e_ss : word;             { Initial (relative) SS value }
    e_sp : word;             { Initial SP value }
    e_csum : word;           { Checksum }
    e_ip : word;             { Initial IP value }
    e_cs : word;             { Initial (relative) CS value }
    e_lfarlc : word;         { File address of relocation table }
    e_ovno : word;           { Overlay number }
    e_res : array[0..3] of word; { Reserved words }
    e_oemid : word;          { OEM identifier (for e_oeminfo) }
    e_oeminfo : word;        { OEM information; e_oemid specific }
    e_res2 : array[0..9] of word; { Reserved words }
    e_lfanew : longint;      { File address of new exe header }
  end;
```

```
type
  PIMAGE_OS2_HEADER = ^TIMAGE_OS2_HEADER;
  TIMAGE_OS2_HEADER = { $IFDEF WIN32 } packed { $ENDIF } record
    ne_magic : word;           { Magic number }
    ne_ver : char;             { Version number }
    ne_rev : char;             { Revision number }
    ne_enttab : word;          { Offset of Entry Table }
    ne_cbenttab : word;        { Number of bytes in Entry Table }
    ne_crc : longint;          { Checksum of whole file }
    ne_flags : word;           { Flag word }
    ne_autodata : word;        { Automatic data segment number }
```

```

ne_heap : word;           {Initial heap allocation}
ne_stack : word;         {Initial stack allocation}
ne_csip : longint;       {Initial CS:IP setting}
ne_sssp : longint;       {Initial SS:SP setting}
ne_cseg : word;          {Count of file segments}
ne_cmod : word;          {Entries in Module Reference Table}
ne_cbnrestab : word;     {Size of non-resident name table}
ne_segtab : word;        {Offset of Segment Table}
ne_rsrctab : word;       {Offset of Resource Table}
ne_restab : word;        {Offset of resident name table}
ne_modtab : word;        {Offset of Module Reference Table}
ne_imptab : word;        {Offset of Imported Names Table}
ne_nrestab : word;       {Offset of Non-resident Names Table}
ne_cmovent : word;       {Count of movable entries}
ne_align : word;         {Segment alignment shift count}
ne_cres : word;          {Count of resource segments}
ne_exetyp : word;        {Target Operating system}
ne_flagsothers : word;   {Other .EXE flags}
ne_pretthunks : word;    {offset to return thunks}
ne_psegrefbytes : word;  {offset to segment ref. bytes}
ne_swaparea : word;      {Minimum code swap area size}
ne_expver : word;        {Expected Windows version number}
end;

```

type

```

PIMAGE_VXD_HEADER = ^TIMAGE_VXD_HEADER;
TIMAGE_VXD_HEADER = {$IFDEF WIN32} packed {$ENDIF} record
  e32_magic : word;           {Magic number}
  e32_border : byte;          {The byte ordering for the VXD}
  e32_worder : byte;          {The word ordering for the VXD}
  e32_level : DWORD;          {The EXE format level for now = 0}
  e32_cpu : word;             {The CPU type}
  e32_os : word;              {The OS type}
  e32_ver : DWORD;            {Module version}
  e32_mflags : DWORD;         {Module flags}
  e32_mpages : DWORD;         {Module # pages}
  e32_startobj : DWORD;       {Object # for instruction pointer}
  e32_eip : DWORD;            {Extended instruction pointer}
  e32_stackobj : DWORD;       {Object # for stack pointer}
  e32_esp : DWORD;            {Extended stack pointer}
  e32_pagesize : DWORD;       {VXD page size}
  e32_lastpagesize : DWORD;   {Last page size in VXD}
  e32_fixupsize : DWORD;      {Fixup section size}
  e32_fixupsum : DWORD;       {Fixup section checksum}
  e32_ldrsize : DWORD;        {Loader section size}

```



```

e32_ldrsum : DWORD;      {Loader section checksum}
e32_objtab : DWORD;      {Object table offset}
e32_objcnt : DWORD;      {Number of objects in module}
e32_objmap : DWORD;      {Object page map offset}
e32_itermap : DWORD;     {Object iterated data map offset}
e32_rsrctab : DWORD;     {Offset of Resource Table}
e32_rsrcnt : DWORD;      {Number of resource entries}
e32_restab : DWORD;      {Offset of resident name table}
e32_enttab : DWORD;      {Offset of Entry Table}
e32_dirtab : DWORD;      {Offset of Module Directive Table}
e32_dircnt : DWORD;      {Number of module directives}
e32_fpagetab : DWORD;    {Offset of Fixup Page Table}
e32_frectab : DWORD;     {Offset of Fixup Record Table}
e32_impmod : DWORD;      {Offset of Import Module Name Table}
e32_impmodcnt : DWORD;   {NumEntries in Import Module Name Table}
e32_impproc : DWORD;     {Offset of Import Procedure Name Table}
e32_pagesum : DWORD;     {Offset of Per-Page Checksum Table}
e32_datapage : DWORD;    {Offset of Enumerated Data Pages}
e32_preload : DWORD;     {Number of preload pages}
e32_nrestab : DWORD;     {Offset of Non-resident Names Table}
e32_cbnrestab : DWORD;   {Size of Non-resident Name Table}
e32_nressum : DWORD;     {Non-resident Name Table Checksum}
e32_autodata : DWORD;    {Object # for automatic data object}
e32_debuginfo : DWORD;   {Offset of the debugging information}
e32_debuglen : DWORD;    {length of the debugging info. in bytes}
e32_instpreload : DWORD; {# of instance pages in preload section}
e32_instdemand : DWORD;  {# of inst pages in demand load section}
e32_heapsize : DWORD;    {Size of heap - for 16-bit apps}
e32_res3 :
    array[0..11] of byte; {Reserved words}
e32_winresoff : DWORD;
e32_winreslen : DWORD;
e32_devid : word;        {Device ID for VxD}
e32_ddkver : word;      {DDK version for VxD}
end;

```

type

```

PIMAGE_DATA_DIRECTORY = ^TIMAGE_DATA_DIRECTORY;
TIMAGE_DATA_DIRECTORY = {$IFDEF WIN32} packed {$ENDIF} record
    VirtualAddress : DWORD;
    Size : DWORD;
end;

```

type

```

PIMAGE_FILE_HEADER = ^TIMAGE_FILE_HEADER;

```

```
TIMAGE_FILE_HEADER = {$IFDEF WIN32} packed {$ENDIF} record
    Machine : word;
    NumberOfSections : word;
    TimeDateStamp : DWORD;
    PointerToSymbolTable : DWORD;
    NumberOfSymbols : DWORD;
    SizeOfOptionalHeader : word;
    Characteristics : word;
end;
```

type

```
PIMAGE_OPTIONAL_HEADER = ^TIMAGE_OPTIONAL_HEADER;
TIMAGE_OPTIONAL_HEADER = {$IFDEF WIN32} packed {$ENDIF} record
    Magic : word;
    MajorLinkerVersion : byte;
    MinorLinkerVersion : byte;
    SizeOfCode : DWORD;
    SizeOfInitializedData : DWORD;
    SizeOfUninitializedData : DWORD;
    AddressOfEntryPoint : DWORD;
    BaseOfCode : DWORD;
    BaseOfData : DWORD;
    {NT additional fields}
    ImageBase : DWORD;
    SectionAlignment : DWORD;
    FileAlignment : DWORD;
    MajorOperatingSystemVersion : word;
    MinorOperatingSystemVersion : word;
    MajorImageVersion : word;
    MinorImageVersion : word;
    MajorSubsystemVersion : word;
    MinorSubsystemVersion : word;
    Reserved1 : DWORD;
    SizeOfImage : DWORD;
    SizeOfHeaders : DWORD;
    CheckSum : DWORD;
    Subsystem : word;
    DllCharacteristics : word;
    SizeOfStackReserve : DWORD;
    SizeOfStackCommit : DWORD;
    SizeOfHeapReserve : DWORD;
    SizeOfHeapCommit : DWORD;
    LoaderFlags : DWORD;
    NumberOfRvaAndSizes : DWORD;
    DataDirectory : array [0..IMAGE_NUMBEROF_DIRECTORY_ENTRIES - 1] of
```

```
TIMAGE_DATA_DIRECTORY;  
end;
```

```
type
```

```
PIMAGE_ROM_OPTIONAL_HEADER = ^TIMAGE_ROM_OPTIONAL_HEADER;  
TIMAGE_ROM_OPTIONAL_HEADER = {$IFDEF WIN32} packed {$ENDIF} record  
    Magic : word;  
    MajorLinkerVersion : byte;  
    MinorLinkerVersion : byte;  
    SizeOfCode : DWORD;  
    SizeOfInitializedData : DWORD;  
    SizeOfUninitializedData : DWORD;  
    AddressOfEntryPoint : DWORD;  
    BaseOfCode : DWORD;  
    BaseOfData : DWORD;  
    BaseOfBss : DWORD;  
    GprMask : DWORD;  
    CprMask : array[0..3] of DWORD;  
    GpValue : DWORD;  
end;
```

```
{$IFNDEF WIN32}
```

```
function MakeWord(A, B: Byte): Word;
```

```
begin
```

```
    Result := A or B shl 8;
```

```
end;
```

```
{$ENDIF}
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    Label1.Caption := '';
```

```
    Button1.Caption := 'GetExeHdr';
```

```
    Button2.Caption := 'MarkExe';
```

```
    Button2.Enabled := false;
```

```
    SpinEdit1.MinValue := 0;
```

```
    SpinEdit1.MaxValue := 255;
```

```
    SpinEdit1.Value := 0;
```

```
    OpenDialog1.Options := [ofNoChangeDir,  
                            ofPathMustExist,  
                            ofFileMustExist];
```

```
    OpenDialog1.Filter := 'All Files (*.*)|*.*';
```

```
    Exetype := IMAGE_UNKNOWN_SIGNATURE;
```

```
end;
```

```
procedure TForm1.GetExeHeader(FileName : string);
```

```

var
  f : file;
  image_dos_header : TIMAGE_DOS_HEADER;
  image_file_header : TIMAGE_FILE_HEADER;
  image_optional_header : TIMAGE_OPTIONAL_HEADER;
  Image_OS2_Header : TIMAGE_OS2_HEADER;
  Image_VXD_Header : TIMAGE_VXD_HEADER;
  Signature : word;
  w : word;
begin
  Exetype := IMAGE_UNKNOWN_SIGNATURE;
  Labell.Caption := '';
  AssignFile(f, FileName);
  Reset(f, 1);
  while Pos('\', FileName) > 0 do
    Delete(FileName, 1, Pos('\', FileName));
  if FileSize(f) < sizeof(image_dos_header) then begin
    Labell.Caption := FileName + ' is an unknown file type';
    CloseFile(f);
    Exit;
  end;
  BlockRead(f, Signature, sizeof(Signature));
  if Signature <> IMAGE_DOS_SIGNATURE then begin
    Labell.Caption := FileName + ' is an unknown file type';
    CloseFile(f);
    Exit;
  end;
  Labell.Caption := FileName + ' has the DOS executable header';
  Seek(f, $18);
  BlockRead(f, w, sizeof(w));
  if w <> $40 then begin
    CloseFile(f);
    Exit;
  end;
  Seek(f, 0);
  BlockRead(f, image_dos_header, sizeof(image_dos_header));
  if FileSize(f) < (image_dos_header.e_lfanew + 2) then begin
    CloseFile(f);
    Exit;
  end;
  Seek(f, image_dos_header.e_lfanew);
  BlockRead(f, Signature, sizeof(Signature));
  if Signature = IMAGE_OS2_SIGNATURE then begin
    if FileSize(f) < (image_dos_header.e_lfanew +
      sizeof(IMAGE_OS2_HEADER)) then begin

```

```

    CloseFile(f);
    Exit;
end;
Seek(f, FilePos(f) - 2);
HdrOffset := FilePos(f);
BlockRead(f, image_OS2_header, sizeof(image_OS2_header));
Exetype := IMAGE_OS2_SIGNATURE;
if image_OS2_header.ne_flags and NE_IMAGE_DLL = NE_IMAGE_DLL then
    Labell.Caption := FileName +
                    ' has the new executable DLL format'
else
    Labell.Caption := FileName +
                    ' has the new executable format';
    SpinEdit1.Value := HiByte(image_OS2_header.ne_expver);
    SpinEdit2.Value := LoByte(image_OS2_header.ne_expver);
end else
if Signature = IMAGE_NT_SIGNATURE then begin
    if FileSize(f) < (image_dos_header.e_lfanew +
                    sizeof(image_file_header) +
                    sizeof(image_optional_header) +
                    2) then begin

        CloseFile(f);
        Exit;
    end;
    Seek(f, FilePos(f) + 2);
    BlockRead(f, image_file_header, sizeof(image_file_header));
    HdrOffset := FilePos(f);
    BlockRead(f,
                image_optional_header,
                sizeof(image_optional_header));
    Exetype := IMAGE_NT_SIGNATURE;

    if image_file_header.Characteristics and IMAGE_FILE_DLL =
        IMAGE_FILE_DLL then
        Labell.Caption := FileName +
                        ' has the portable executable DLL format'
    else
        Labell.Caption := FileName +
                        ' has the portable executable format';
        SpinEdit1.Value := image_optional_header.MajorSubsystemVersion;
        SpinEdit2.Value := image_optional_header.MinorSubsystemVersion;
    end else
if Signature = IMAGE_VXD_SIGNATURE then begin
    Seek(f, FilePos(f) - 2);
    HdrOffset := FilePos(f);

```

```

    BlockRead(f, image_VXD_header, sizeof(image_VXD_header));
    Label1.Caption := FileName + ' has the VXD format';
    Exetype := IMAGE_VXD_SIGNATURE;
    SpinEdit1.Value := LoByte(image_VXD_header.e32_os);
    SpinEdit2.Value := HiByte(image_VXD_header.e32_os);
end;
CloseFile(f);
Button2.Enabled := true;
end;

procedure TForm1.RemarkExeHeader(FileName : string);
var
    f : file;
    image_optional_header : TIMAGE_OPTIONAL_HEADER;
    Image_OS2_Header : TIMAGE_OS2_HEADER;
    Image_VXD_Header : TIMAGE_VXD_HEADER;
begin
    AssignFile(f, FileName);
    Reset(f, 1);
    while Pos('\', FileName) > 0 do
        Delete(FileName, 1, Pos('\', FileName));
    Seek(f, HdrOffset);
    case Exetype of
        IMAGE_OS2_SIGNATURE : begin
            BlockRead(f, image_OS2_header, sizeof(image_OS2_header));
            image_OS2_header.ne_expver :=
                MakeWord(Byte(SpinEdit2.Value), Byte(SpinEdit1.Value));
            Seek(f, HdrOffset);
            BlockWrite(f, image_OS2_header, sizeof(image_OS2_header));
        end;
        IMAGE_NT_SIGNATURE : begin
            HdrOffset := FilePos(f);
            BlockRead(f,
                image_optional_header,
                sizeof(image_optional_header));
            image_optional_header.MajorSubsystemVersion := SpinEdit1.Value;
            image_optional_header.MinorSubsystemVersion := SpinEdit2.Value;
            Seek(f, HdrOffset);
            BlockWrite(f,
                image_optional_header,
                sizeof(image_optional_header));
        end;
        IMAGE_VXD_SIGNATURE : begin
            BlockRead(f, image_VXD_header, sizeof(image_VXD_header));

```

```
image_VXD_header.e32_os :=  
    MakeWord(Byte(SpinEdit1.Value), Byte(SpinEdit2.Value));  
Seek(f, HdrOffset);  
BlockWrite(f, image_VXD_header, sizeof(image_VXD_header));  
end;
```

```
end;
```

```
CloseFile(f);
```

```
ShowMessage(FileName + ' remarked');
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    if OpenFileDialog1.Execute then
```

```
        GetExeHeader(OpenDialog1.FileName);
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
```

```
    if Exetype <> IMAGE_UNKNOWN_SIGNATURE then
```

```
        ReMarkExeHeader(OpenDialog1.FileName);
```

```
end;
```

```
end.
```

4/2/99 11:28:24 AM

Last Modified: 01-SEP-99

## Article #16500: Does Delphi have an equivalent to the Visual Basic SendKeys function?

Question and Answer Database

FAQ1500D.txt Does Delphi have an equivalent to the Visual Basic SendKeys function?

Category :VCL

Platform :All

Product :All 32 bit

Question:

Does Delphi have an equivalent to the Visual Basic SendKeys function?

Answer:

The following example demonstrates procedures that provide the capability of sending keystrokes to any window control capable of receiving keyboard input. You may use this technique to toggle the num lock, caps lock, and scroll lock keys under Windows NT. This same technique works for toggling caps lock and scroll lock keys under Windows 95, but it will not work for num lock.

Note that there are four procedures provided: SimulateKeyDown(), SimulateKeyUp(), SimulateKeystroke(), and SendKeys(), to allow greater control in your ability to send keystrokes.

The SimulateKeyDown(), SimulateKeyUp(), and SimulateKeystroke() procedures expect a virtual key code (like VK\_F1).

The SimulateKeystroke() procedure accepts an extra parameter that is useful when simulating the PrintScreen key. When extra is set to zero, the entire screen will be captured to the windows clipboard. When extra is set to one, only the active window will be captured.

The four button click methods demonstrate the use of these functions:

ButtonClick1 - Toggles the cap lock.

ButtonClick2 - Captures the entire screen to the clipboard.

ButtonClick3 - Capture the active window to the clipboard.

ButtonClick4 - Set the focus to an edit control and sends it a string.

Example:

```
procedure SimulateKeyDown(Key : byte);
```



```

begin
    keybd_event(Key, 0, 0, 0);
end;

procedure SimulateKeyUp(Key : byte);
begin
    keybd_event(Key, 0, KEYEVENTF_KEYUP, 0);
end;

procedure SimulateKeystroke(Key : byte;
                             extra : DWORD);
begin
    keybd_event(Key,
                extra,
                0,
                0);
    keybd_event(Key,
                extra,
                KEYEVENTF_KEYUP,
                0);
end;

procedure SendKeys(s : string);
var
    i : integer;
    flag : bool;
    w : word;
begin
    {Get the state of the caps lock key}
    flag := not GetKeyState(VK_CAPITAL) and 1 = 0;
    {If the caps lock key is on then turn it off}
    if flag then
        SimulateKeystroke(VK_CAPITAL, 0);
    for i := 1 to Length(s) do begin
        w := VkKeyScan(s[i]);
        {If there is not an error in the key translation}
        if ((HiByte(w) <> $FF) and
            (LoByte(w) <> $FF)) then begin
            {If the key requires the shift key down - hold it down}
            if HiByte(w) and 1 = 1 then
                SimulateKeyDown(VK_SHIFT);
            {Send the VK_KEY}
            SimulateKeystroke(LoByte(w), 0);
            {If the key required the shift key down - release it}
            if HiByte(w) and 1 = 1 then

```

```

        SimulateKeyUp(VK_SHIFT);
    end;
end;
{if the caps lock key was on at start, turn it back on}
if flag then
    SimulateKeystroke(VK_CAPITAL, 0);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    {Toggle the cap lock}
    SimulateKeystroke(VK_CAPITAL, 0);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    {Capture the entire screen to the clipboard}
    {by simulating pressing the PrintScreen key}
    SimulateKeystroke(VK_SNAPSHOT, 0);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    {Capture the active window to the clipboard}
    {by simulating pressing the PrintScreen key}
    SimulateKeystroke(VK_SNAPSHOT, 1);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    {Set the focus to a window (edit control) and send it a string}
    Application.ProcessMessages;
    Edit1.SetFocus;
    SendKeys('Delphi Is RAD!');
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16505: Using Tapi

Question and Answer Database

FAQ1505D.txt Using Tapi

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I use TAPI to dial the telephone for a voice call?

Answer:

The following example shows how to interface with tapi to make a voice call.

Example:

```
{tapi Errors}
const TAPIERR_CONNECTED          = 0;
const TAPIERR_DROPPED           = -1;
const TAPIERR_NOREQUESTRECIPIENT = -2;
const TAPIERR_REQUESTQUEUEFULL  = -3;
const TAPIERR_INVALIDDESTADDRESS = -4;
const TAPIERR_INVALIDWINDOWHANDLE = -5;
const TAPIERR_INVALIDDEVICECLASS = -6;
const TAPIERR_INVALIDDEVICEID    = -7;
const TAPIERR_DEVICECLASSUNAVAIL = -8;
const TAPIERR_DEVICEIDUNAVAIL    = -9;
const TAPIERR_DEVICEINUSE        = -10;
const TAPIERR_DESTBUSY           = -11;
const TAPIERR_DESTNOANSWER       = -12;
const TAPIERR_DESTUNAVAIL        = -13;
const TAPIERR_UNKNOWNWINHANDLE   = -14;
const TAPIERR_UNKNOWNREQUESTID   = -15;
const TAPIERR_REQUESTFAILED      = -16;
const TAPIERR_REQUESTCANCELLED   = -17;
const TAPIERR_INVALIDPOINTER     = -18;

{tapi size constants}
const TAPIMAXDESTADDRESSSIZE     = 80;
const TAPIMAXAPPNAMESSIZE        = 40;
```

```
const TAPIMAXCALLEDPARTYSIZE      = 40;
const TAPIMAXCOMMENTSIZE          = 80;
const TAPIMAXDEVICECLASSSIZE      = 40;
const TAPIMAXDEVICEIDSIZE         = 40;
```

```
function tapiRequestMakeCallA(DestAddress : PAnsiChar;
                              AppName : PAnsiChar;
                              CalledParty : PAnsiChar;
                              Comment : PAnsiChar) : LongInt;
  stdcall; external 'TAPI32.DLL';
```

```
function tapiRequestMakeCallW(DestAddress : PWideChar;
                              AppName : PWideChar;
                              CalledParty : PWideChar;
                              Comment : PWideChar) : LongInt;
  stdcall; external 'TAPI32.DLL';
```

```
function tapiRequestMakeCall(DestAddress : PChar;
                              AppName : PChar;
                              CalledParty : PChar;
                              Comment : PChar) : LongInt;
  stdcall; external 'TAPI32.DLL';
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  DestAddress : string;
  CalledParty : string;
  Comment : string;
begin
  DestAddress := '1-555-555-1212';
  CalledParty := 'Frank Borland';
  Comment := 'Calling Frank';
  tapiRequestMakeCall(pChar(DestAddress),
                     PChar(Application.Title),
                     pChar(CalledParty),
                     PChar(Comment));
```

```
end;
```

```
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16510: Setting custom tabstops in a ListBox.

Question and Answer Database

FAQ1510D.txt     Setting custom tabstops in a ListBox.

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I set custom tab stops in a TListbox control?

Answer:

You will need to send a LB\_SETTABSTOPS message to the Listbox.

Note: The Listbox tabwidth property must be set to a number other than zero for the LB\_SETTABSTOPS message to work.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
  ListBox1.TabWidth := 1;
```

```
  ListBox1.Items.Add('one'+#9+'one');
```

```
  ListBox1.Items.Add('two'+#9+'two');
```

```
  ListBox1.Items.Add('three'+#9+'three');
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  DialogUnitsX : LongInt;
```

```
  PixelsX : LongInt;
```

```
  i : integer;
```

```
  TabArray : array[0..4] of integer;
```

```
begin
```

```
  DialogUnitsX := LoWord(GetDialogBaseUnits);
```

```
  PixelsX := SpinEdit1.Value;
```

```
  for i := 1 to 5 do begin
```

```
    TabArray[i - 1] :=
```

```
      ((PixelsX * i) * 4) div DialogUnitsX;
```

```
  end;
```

```
  if SendMessage(ListBox1.Handle,
```

```
      LB_SETTABSTOPS,
```

```
5,  
    LongInt(@TabArray)) = 0 then  
    ShowMessage('Tabs Not Set');  
    Listbox1.Refresh;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16515: Compile Error: Too many files open

Question and Answer Database

FAQ1515D.txt     Compile Error: Too many files open

Category        :IDE

Platform        :All

Product         :All 32 bit

### Question:

I have a large project consisting of many units. I added a unit and now I get an error "Too many files open". How can I work around this?

### Answer:

Move your project to a directory with a short path name, or use the dos subst command to substitute a drive letter for the directory name. If you move the project to another directory, be sure to only move the necessary files used to compile the application. Borland's Ti3214 - "Moving Projects Between Machines or Directories" is also a helpful resource. ie: .dpr, .pas, .res etc.. Note: Do not move the .dsm files, as they have path information built into them, and will be automatically regenerated.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16520: How can I create a transparent form?

Question and Answer Database

FAQ1520D.txt    How can I create a transparent form?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I create a transparent form?

Answer:

The following example demonstrates how to create a transparent form.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.Brush.Style := bsClear;
  Form1.BorderStyle := bsNone
end;
```

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Application.Terminate;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16525: Using WM\_COPYDATA.

Question and Answer Database

FAQ1525D.txt Using WM\_COPYDATA.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I use the WM\_COPYDATA message to pass information between 16-bit and 32-bit applications?

Answer:

The following example shows how to use the WM\_COPYDATA message to send a record containing data between a 16-bit and 32-bit application. This method can also be used to call 16-bit or 32-bit dll from a 16-bit or 32-bit application (known as thunking). Thunking can be accomplished by creating an invisible window to respond to the WM\_COPYDATA message, and call the dll in your behalf.

Compile the following unit source code under both 16-bit and 32-bit versions of Delphi. Use separate directories for the two projects. Each project will require a form, a button, and a memo component on it's form.

To use the WM\_COPYDATA message, you must send the address of a TCopyDataStruct to the receiving window. The dwData parameter of the TCopyDataStruc is a user defined parameter that allows you to store any 32 bit value you wish, for additional information. The cbData member of the TCopyDataStruc contains the size of the data pointed to by the lpData member of the TCopyDataStruc. The lpData parameter of the TCopyDataStruct can point to any data structure you wish.

Note that any pointers contained in the structure you send will not be valid from the receiving application since windows has no way to convert them. Note that the data that is passed to the receiving application is only valid during the call. If the receiving application needs to retain the information passed, it should make a copy of the data that is local to the receiving application. The receiving application should not modify the data that is passed during the call. If modifications are necessary, the receiving application

should make a local copy of the data, and send a WM\_COPYDATA message back to the sending application using the modified copy.

It's worth noting that the WM\_COPYDATA message will work going to and from any combination of 16/32 bit applications. Finally, be aware that the call to SendMessage will not return until the message is processed. Also be aware that you should never use PostMessage() with the WM\_COPYDATA message.

```
unit Unit1;
```

```
interface
```

```
 {$IFDEF WIN32}
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls;
```

```
 {$ELSE}
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls;
```

```
 {$ENDIF}
```

```
const WM_COPYDATA = $004A;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    Mem1: TMemo;
```

```
    Button1: TButton;
```

```
    procedure FormCreate(Sender: TObject);
```

```
    procedure Button1Click(Sender: TObject);
```

```
  private
```

```
    { Private declarations }
```

```
    procedure WMCopyData(var m : TMessage); message WM_COPYDATA;
```

```
  public
```

```
    { Public declarations }
```

```
end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
 {$R *.DFM}
```

```

type
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = record
    dwData: LongInt;
    cbData: LongInt;
    lpData: Pointer;
  end;

type
  PRecToPass = ^TRecToPass;
  TRecToPass = packed record
    s : string[255];
    i : integer;
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  {$IFDEF WIN32}
    Form1.Caption := 'My 32-Bit App'
  {$ELSE}
    Form1.Caption := 'My 16-Bit App'
  {$ENDIF}
end;

procedure TForm1.WMCopyData(var m : TMessage);
begin
  Mem1.Lines.Add('Sending Window Handle := ' +
    IntToStr(m.WParam));
  Mem1.Lines.Add('Data Size := ' +
    IntToStr(PCopyDataStruct(m.LParam)^.cbData));
  Mem1.Lines.Add('User Defined Data Param := ' +
    IntToStr(PCopyDataStruct(m.LParam)^.dwData));
  Mem1.Lines.Add('TRecToPass.s := ' +
    PRecToPass(PCopyDataStruct(m.LParam)^.lpData)^.s);
  Mem1.Lines.Add('TRecToPass.i := ' +
    IntToStr(
      PRecToPass(PCopyDataStruct(m.LParam)^.lpData)^.i));
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  h : THandle;
  cd : TCopyDataStruct;
  rec : TRecToPass;
begin

```

```
{$IFDEF WIN32}
h := FindWindow(nil, 'My 16-Bit App');
rec.s := 'Hello World From: My 32-Bit App';
rec.i := 32;
cd.dwData := 3232;
{$ELSE}
h := FindWindow(nil, 'My 32-Bit App');
rec.s := 'Hello World From: My 16-Bit App';
rec.i := 16;
cd.dwData := 1616;
{$ENDIF}
cd.cbData := sizeof(rec);
cd.lpData := @rec;
if h <> 0 then
    SendMessage(h, WM_COPYDATA, Form1.Handle, LongInt(@cd));
end;
```

end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

[bdn.borland.com](http://bdn.borland.com)

## **Article #16527: Borland Assist for Delphi/400**

Borland no longer distributes or supports Delphi/400. Delphi/400 is now distributed by [System Objects](#)

Last Modified: 30-JAN-02

## Article #16530: Selecting a cell in a DBGrid

Question and Answer Database

FAQ1530D.txt     Selecting a cell in a DBGrid

Category        :VCL

Platform        :All

Product         :All 32 bit

### Question:

How can I programmatically place a grid into edit mode and move the cursor to a predetermined position in the edit box?

### Answer:

Place the grid in editor mode, then get a handle to the edit box and send a EM\_SETSEL message, passing the initial position where you want the cursor to appear, and the position of the ending selection (if you wish to highlight text).

The following example sets the cursor to the second character in the edit control, and does not select any text.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  h : THandle;
begin
  Application.ProcessMessages;
  DbGrid1.SetFocus;
  DbGrid1.EditorMode := true;
  Application.ProcessMessages;
  h := Windows.GetFocus;
  SendMessage(h, EM_SETSEL, 2, 2);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16535: Trapping non client areas

Question and Answer Database

FAQ1535D.txt Trapping non client areas

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I trap messages in the non client areas of my form, such as the title bar?

Answer:

Create a message handler for one of the variety of WM\_NC (non client) messages available to trap. (search the Windows API help file for WM\_NC). This example shows how to track mouse movement in all of the non client areas of the form.

Example:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
  private
```

```
    procedure WMNCMOUSEMOVE(var Message: TMessage);
```

```
    message WM_NCMOUSEMOVE;
```

```
    { Private declarations }
```

```
  public
```

```
    { Public declarations }
```

```
  end;
```

```
var
```

```
  Form1: TForm1;
```

implementation

{ \$R \*.DFM }

```
procedure TForm1.WMNCMOUSEMOVE(var Message: TMessage);
var
  s : string;
begin
  case Message.wParam of
    HTERROR : s := 'HTERROR';
    HTTRANSPARENT : s := 'HTTRANSPARENT';
    HTNOWHERE : s := 'HTNOWHERE';
    HTCLIENT : s := 'HTCLIENT';
    HTCAPTION : s := 'HTCAPTION';
    HTSYSTEMMENU : s := 'HTSYSTEMMENU';
    HTSIZE : s := 'HTSIZE';
    HTMENU : s := 'HTMENU';
    HTHSCROLL : s := 'HTHSCROLL';
    HTVSCROLL : s := 'HTVSCROLL';
    HTMINBUTTON : s := 'HTMINBUTTON';
    HTMAXBUTTON : s := 'HTMAXBUTTON';
    HTLEFT : s := 'HTLEFT';
    HTRIGHT : s := 'HTRIGHT';
    HTTOP : s := 'HTTOP';
    HTTOPLEFT : s := 'HTTOPLEFT';
    HTTOPRIGHT : s := 'HTTOPRIGHT';
    HTBOTTOM : s := 'HTBOTTOM';
    HTBOTTOMLEFT : s := 'HTBOTTOMLEFT';
    HTBOTTOMRIGHT : s := 'HTBOTTOMRIGHT';
    HTBORDER : s := 'HTBORDER';
    HTOBJECT : s := 'HTOBJECT';
    HTCLOSE : s := 'HTCLOSE';
    HTHELP : s := 'HTHELP';
  else s := '';
  end;
  Form1.Caption := s;
  Message.Result := 0;
end;

end.
```

7/16/98 4:31:28 PM



Last Modified: 01-SEP-99

## Article #16545: Changing the printer setup between pages

Question and Answer Database

FAQ1545D.txt - Changing the printer setup between pages

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How can I change the printer setup between printed pages?

Answer:

This is normally accomplished by retrieving a copy of the device mode structure that is associated with the current printer, making the necessary changes to the device independent portion of the structure, and then calling the Windows API function `ResetDc()` in between pages.

Since the `ResetDc()` function is disabled by Windows during the printing of a page, and the `TPrinter` unit does not have an event that is fired between pages, a work around is required.

While it is possible to insert and modify code directly in the implementation section of `TPrinter`, it is not possible to make changes to the interface portion of `TPrinter` and still maintain stability in the VCL and other third party components. With the advent of packages, rebuilding the main VCL package is also not an ideal solution. Finally, `TPrinter` is one of the few VCL classes that does not lend itself to subclassing, making it difficult to add an event that would fire between pages.

Before the advent of packages, the work around consisted of creating an additional unit that sat between `TPrinter` and the unit(s) containing your printing code. A `uses` statement was added to the `TPrinter` unit in the implementation section that pointed to the new unit (avoiding any change to `TPrinter`'s interface section). Finally, code was added to the `TPrinter` unit that would call a function pointer in the new unit between each printed page. You could then hook into `TPrinter` by adding the new unit to the `uses` clause of the unit(s) you

are printing from, and reassigning the function pointer to point to a function in your unit. While this may sound complex, in practice, it was a clean solution to an interesting problem, that worked well with both VCL components and custom printing code produced directly in your application.

With the advent of packages, a new solution is needed, since the TPrinter unit is compiled into the main VCL system package, where rebuilding is not recommended. Here we will provide such a solution in the form of a single unit that encapsulates both the work around and the printing code. The code is designed so it will compile under all versions of Delphi and Borland's C++ Builder. Since there will be no hooks added to the TPrinter unit, this solution will only work with custom printing code provided from your application, and not work with VCL and other "non-aware" third part components that offer a print method, or code utilizing the AssignPrn() function.

The solution presented allows changes to the page setup by making direct calls to the Windows API to simulate TPrinter's NewPage method, and calling the Windows API function ResetDc() between pages. Note that not all printers support the ResetDc() escape. Under 16 bit versions of Windows, testing for escape support by calling Escape(QueryEscapeSupport...) returns the correct value for the ResetDc undocumented escape code #128, but Windows 95 instead steals the call, and returns true even if the driver does not support it. The only way to test if the ResetDc() function is supported is to make a call directly to the function itself. To avoid printing blank pages in the event of an error, the example calls the ResetDC function and passes Printer.Handle as the DC before the printing actually starts to test for support of the function. This is ok, as the call is transformed by Windows to an escape to the print driver, and the dc is not actually passed but is instead used by Windows to identify the driver. If the print driver does not support the ResetDc() escape, we will accomplish changing the printer's settings by printing each page as separate print job. Since we are bypassing TPrinter's NewPage method, and since the print job may be accomplished by printing separate pages, the PageNumber property of TPrinter will not remain accurate through out the print job and should be tracked by the application. Finally, if abort method of TPrinter must be called, it should be called in a section of code after only after a StartPage command.

Note: See TDEVMODE in the Delphi 1.02 help file or DEVMODE in the 32 bit versions of Delphi and C++ Builder for other settings you can change, such as the bin, paper sizes and so forth.

Example:

```
unit Unit1;

interface

{$IFDEF WIN32}
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Printers;
{$ELSE}
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Printers, Print;
{$ENDIF}

type
  TForm1 = class(TForm)
    PrintDialog1: TPrintDialog;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
    function PageSetup : bool;
    function NewPage : bool;
  public
    { Public declarations }
    PageNumber : integer;
    CanReset : bool;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{$IFDEF WIN32}
const MAX_PATH = 144;
{$ENDIF}

function TForm1.PageSetup : bool;
```

```

var
  pDevice : pChar;
  pDriver : pChar;
  pPort   : pChar;
  hDMode  : THandle;
  PDMode  : PDEVMODE;
  PrnHandle : THandle;
begin
  result := false;
  GetMem(pDevice, cchDeviceName);
  GetMem(pDriver, MAX_PATH);
  GetMem(pPort, MAX_PATH);
  Printer.GetPrinter(pDevice, pDriver, pPort, hDMode);
  if hDMode <> 0 then begin
    pDMode := GlobalLock(hDMode);
    if pDMode <> nil then begin

      {Change your printing settings here}
      if not odd(PageNumber) then begin
        pDMode^.dmFields := pDMode^.dmFields or dm_PaperSize;
        pDMode^.dmPaperSize := DMPAPER_LETTER;
      end else begin
        pDMode^.dmFields := pDMode^.dmFields or dm_PaperSize;
        pDMode^.dmPaperSize := DMPAPER_LEGAL;
      end;

      if Printer.Printing then
        PrnHandle := Printer.Canvas.Handle
      else
        PrnHandle := Printer.Handle;
    {$IFDEF WIN32}
      if ResetDc(PrnHandle, pDMode^) <> 0 then
    {$ELSE}
      if ResetDc(PrnHandle, pDMode) <> 0 then
    {$ENDIF}
        CanReset := true else
        CanReset := false;
      Result := true;
      GlobalUnlock(hDMode);
    end;
  end;
  FreeMem(pDevice, cchDeviceName);
  FreeMem(pDriver, MAX_PATH);
  FreeMem(pPort, MAX_PATH);
end;

```

```

function TForm1.NewPage : bool;
begin
  Result := true;
  if CanReset then
    EndPage(Printer.Canvas.Handle) else
    Printer.EndDoc;
  Inc(PageNumber);
  if PageSetUp then begin
    if CanReset then begin
      StartPage(Printer.Canvas.Handle);
      Printer.Canvas.Refresh;
    end else
      Printer.BeginDoc;
  end else begin
    if CanReset then begin
      StartPage(Printer.Canvas.Handle);
      Printer.Abort;
    end;
    Result := false;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if PrintDialog1.Execute then begin

    {Setup printing}
    PageNumber := 1;
    if not PageSetUp then begin
      ShowMessage('Unable to customize printer settings');
      exit;
    end;
    Printer.BeginDoc;

    {Print page one}
    Printer.Canvas.TextOut(100,100, 'Test Page 1');
    if not NewPage then begin
      ShowMessage('Unable to customize printer settings');
      exit;
    end;

    {Print page two}
    Printer.Canvas.TextOut(100,100, 'Test Page 2');
    if not NewPage then begin

```

```
ShowMessage('Unable to customize printer settings');  
exit;  
end;
```

```
{Print page three}
```

```
Printer.Canvas.TextOut(100,100, 'Test Page 3');
```

```
Printer.EndDoc;
```

```
end;
```

```
end;
```

```
end.
```

4/2/99 11:24:05 AM

Last Modified: 01-SEP-99

## Article #16550: Text in Windows metafiles

Question and Answer Database

FAQ1550D.txt    Text in Windows metafiles

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

### Question:

Sometimes, some text in a metafile I display from my application comes up in the wrong font. The file displays properly in xyz Graphics program. Why?

### Answer:

The VCL simply wraps standard Windows functions for displaying metafiles. As the metafile is scaled, the font width and height are scaled as well. Many times, the Windows font mapper will simply substitute a different font that the font that was originally selected in an attempt to satisfy the request. This can also be seen in many graphics programs if you arbitrarily scale the metafile in different directions. Note that some "hi end" graphic programs actually read the metafile record by record, bypassing the Windows GDI entirely to overcome the effects of the Windows font mapper.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16555: How can I get the serial number of an Audio CD?

Question and Answer Database

FAQ1555D.txt     How can I get the serial number of an Audio CD?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I get the serial number of an Audio CD?

Answer:

The cd may or may not have a serial number and/or Universal Product Code embedded in the cd. The MCI Windows extension does provide this functionality through the MCI\_INFO\_MEDIA\_IDENTITY command. This command will return a unique ID string consisting of 16 hexadecimal digits.

Example:

```
uses MMSystem, MPlayer;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  mp : TMediaPlayer;
```

```
  msp : TMCI_INFO_PARMS;
```

```
  MediaString : array[0..255] of char;
```

```
  ret : longint;
```

```
begin
```

```
  mp := TMediaPlayer.Create(nil);
```

```
  mp.Visible := false;
```

```
  mp.Parent := Application.MainForm;
```

```
  mp.Shareable := true;
```

```
  mp.DeviceType := dtCDAudio;
```

```
  mp.FileName := 'D:';
```

```
  mp.Open;
```

```
  Application.ProcessMessages;
```

```
  FillChar(MediaString, sizeof(MediaString), #0);
```

```
  FillChar(msp, sizeof(msp), #0);
```

```
  msp.lpstrReturn := @MediaString;
```

```
  msp.dwRetSize := 255;
```

```
ret := mciSendCommand(Mp.DeviceId,  
                      MCI_INFO,  
                      MCI_INFO_MEDIA_IDENTITY,  
                      longint(@msp));  
  
if Ret <> 0 then begin  
  MciGetErrorString(ret, @MediaString, sizeof(MediaString));  
  Mem1.Lines.Add(StrPas(MediaString));  
end else  
  Mem1.Lines.Add(StrPas(MediaString));  
mp.Close;  
Application.ProcessMessages;  
mp.free;  
end;  
  
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16560: Error: 'Mismatch in datapacket'

Question and Answer Database

FAQ1560D.txt     Error: 'Mismatch in datapacket'

Category     :Multitier/Remote Datasets

Platform     :All

Product      :Delphi 3.x

Question:

When I use LoadFromStream on the TClientDataSet,  
I receive an EDBClient exception "Mismatch in datapacket".

Answer:

You may not be resetting the position of the stream.  
This procedure works...

```
var
Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    CDS.SaveToStream(Stream);
    // Reset to the beginning of the data packet.
    // Forgetting this step is one way to produce the error.
    Stream.Position := 0;
    CDS.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16565: Accessing new interfaces in a RemoteDataModule

Question and Answer Database

FAQ1565D.txt    Accessing new interfaces in a RemoteDataModule  
Category        :Multitier/Remote Datasets  
Platform        :All  
Product         :Delphi 3.x

Question:

If I want to support several interfaces on the same RemoteDataModule and in the typelib editor, add new interfaces, and make them members of the original coClass. How do I access these other interfaces. If I try (AppServer as IAnother) or IAnother(AppServer), I get a compilation error.

Answer:

Try the syntax:

```
(IDispatch(RemoteServer.AppServer) as IAnother)
```

Also, this will only work if you are using DCOM as the transport.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16575: Reassigning events at runtime

Question and Answer Database

FAQ1575D.txt Reassigning events at runtime

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I reassign an event at runtime?

Answer:

The following example demonstrates reassigning an event.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Button2SavedEvent : TNotifyEvent;
  Button3SavedEvent : TNotifyEvent;
begin
  {Save the events}
  Button2SavedEvent := Button2.OnClick;
  Button3SavedEvent := Button3.OnClick;

  {Clear button 2's OnClick event}
  Button2.OnClick := nil;

  {Set button 3's OnClick to Button2's old OnClick Event}
  Button3.OnClick := Button2SavedEvent;

  { Do something here }

  {Take us back to the original events}
  Button2.OnClick := Button2SavedEvent;
  Button3.OnClick := Button3SavedEvent;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16580: Showing two forms allowing the first to retain focus

Question and Answer Database

FAQ1580D.txt Showing two forms allowing the first to retain focus

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to show a form, but allow the form showing before it to remain focused, and still positioned behind the newly shown form?

Answer:

The follow code demonstrates showing a non auto-created form without changing the focus of the original form.

```
uses Unit2;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    Form2 := TForm2.Create(Application);
```

```
    Form2.Visible := FALSE;
```

```
    ShowWindow(Form2.Handle, SW_SHOWNA);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16585: Disabling close button of a window

Question and Answer Database

FAQ1585D.txt Disabling close button of a window

Category :Windows API

Platform :All

Product :All 32 bit

Question:

Is it possible to disable the close button of any given Window?

Answer:

Yes, the following example disables the close button (and close option from the system menu) of the given Window.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  hwndHandle : THANDLE;
  hMenuHandle : HMENU;
begin
  hwndHandle := FindWindow(nil, 'Untitled - Notepad');
  if (hwndHandle <> 0) then begin
    hMenuHandle := GetSystemMenu(hwndHandle, FALSE);
    if (hMenuHandle <> 0) then
      DeleteMenu(hMenuHandle, SC_CLOSE, MF_BYCOMMAND);
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16595: Creating a Resource only DLL

Question and Answer Database

FAQ1595D.txt     Creating a Resource only DLL

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How do I create a Resource only DLL?

Answer:

Create and build an empty DLL project, that contains a resource link reference to the .res file that contains your resources.

Example:

```
library ResTest;
```

```
uses
```

```
    SysUtils;
```

```
{ $R MYRES.RES }
```

```
begin
```

```
end.
```

To use you resource only DLL, simply load the dll and the resources you wish to use:

Example:

```
{ $IFDEF WIN32 }
```

```
const BadDllLoad = 0;
```

```
{ $ELSE }
```

```
const BadDllLoad = 32;
```

```
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```



```
h : THandle;  
Icon : THandle;
```

```
begin
```

```
  h := LoadLibrary('RESTEST.DLL');
```

```
  if h <= BadDllLoad then
```

```
    ShowMessage('Bad Dll Load')
```

```
  else begin
```

```
    Icon := LoadIcon(h, 'ICON_1');
```

```
    DrawIcon(Form1.Canvas.Handle, 10, 10, Icon);
```

```
    FreeLibrary(h);
```

```
  end;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16600: Deleting an executable while it is running

Question and Answer Database

FAQ1600D.txt Deleting an executable while it is running

Category :Miscellaneous

Platform :All

Product :All 32 bit

Question:

How can I delete my executable while it is running?

Answer:

Under Win32, unless you are running from a removable drive, you cannot delete a running executable. You can have Windows delete the executable the next time Windows is ran by adding an entry to the RunOnce key in the Windows registry under:

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce

You can name the key anything you like, and specify a command line to another executable or to a dos command passed to command.com.

Example:

uses

Registry;

procedure TForm1.Button1Click(Sender: TObject);

var

reg: TRegistry;

begin

reg := TRegistry.Create;

with reg do begin

RootKey := HKEY\_LOCAL\_MACHINE;

LazyWrite := false;

OpenKey('Software\Microsoft\Windows\CurrentVersion\RunOnce',  
false);

WriteString('Delete Me!', 'command.com /c del FILENAME.EXT');

```
    CloseKey;  
    free;  
end;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

**Article #16605: Windows Dialer dials out when I am already connected to the internet**

Question and Answer Database

FAQ1605D.txt - Windows Dialer dials out when I am already connected to the internet

Category :Internet/WEB  
Platform :All-32Bit  
Product :

Question:

Whenever I try to use the Internet controls, the Windows dialer attempts to dial out, even if I am connected to the internet via the Network. What can I do?

Answer:

You can disable this feature of the operating system by going to the Windows Control Panel, select the Internet icon, and go to the Connections page. Un-Select the check box "Connect to the internet as needed".

4/2/99 11:44:35 AM

Last Modified: 01-SEP-99

## Article #16610: displaying an application associated icons

Question and Answer Database

FAQ1610D.txt displaying an application associated icons

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I display the icon that is associated with a given file type?

Answer:

Use the ShellApi function `ExtractAssociatedIcon()` to retrieve the icon that is associated with the file.

Example:

```
uses ShellApi;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    Icon : hIcon;
```

```
    IconIndex : word;
```

```
begin
```

```
    IconIndex := 1;
```

```
    Icon := ExtractAssociatedIcon(HInstance,  
                                Application.ExeName,  
                                IconIndex);
```

```
    DrawIcon(Canvas.Handle, 10, 10, Icon);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16615: getting DLL function parameter listings

Question and Answer Database

FAQ1615D.txt getting DLL function parameter listings

Category :Miscellaneous

Platform :All

Product :All 32 bit

Question:

How can I get a list of the function parameters of a given function in a Dynamic Link Library (DLL) at runtime?

Answer:

While you can get the list of exported functions in a DLL, you cannot get the parameter list. To get this information, you would need get the documentation from the distributor of the DLL.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16620: Loading a locale file into the THTML component

Question and Answer Database

FAQ1620D.txt - Loading a locale file into the THTML component

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

How do I load a local file in the HTML control?

Answer:

Use the file command in the URL:

Example:

```
HTML1.RequestDoc('file:///C:/Netscape/Navigator/bookmark.htm');
```

4/2/99 11:46:38 AM

Last Modified: 01-SEP-99

## Article #16625: Printing a web page using the HTML control

Question and Answer Database

FAQ1625D.txt - Printing a web page using the HTML control

Category :Printing  
Platform :All  
Product :

Question:

How do I print a WEB page with the HTML control?

Answer:

Use either the HTML control's AutoPrint method, or alternatively use the PrintPage method.

Example using AutoPrint:

```
uses Printers;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    OldCur: TCursor;
```

```
begin
```

```
    OldCur := Screen.Cursor;
```

```
    with Printer do begin
```

```
        BeginDoc;
```

```
        HTML1.AutoPrint(handle);
```

```
        Title := HTML1.URL;
```

```
        EndDoc;
```

```
    end;
```

```
    Screen.Cursor := OldCur;
```

```
end;
```

4/2/99 11:24:16 AM

Last Modified: 01-SEP-99



## Article #16630: Illegal Operation, Windows Lockup, IDE

Question and Answer Database

FAQ1630D.txt Illegal Operation, Windows Lockup, IDE

Category :IDE

Platform :All

Product :All 32 bit

### Question:

How can I prevent Windows 95 from locking up when my application performs an illegal operation when run from the IDE?

### Answer:

When the dialog box announces the Illegal Operation, do not immediately press the "Done" button. Instead, go back to the IDE and select "Run->Program Reset" from the IDE menu (or press Ctrl-F2 inside the IDE) to release the Debug Hook, and then go back to the dialog box and press the "Done" button.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16635: doing work while application is Idle

Question and Answer Database

FAQ1635D.txt    doing work while application is Idle

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I have my application perform work when it is idle?

Answer:

You can create an OnIdle event to be called whenever your application is waiting for messages to be processed. Simply declare the event handler procedure in the private section of the forms declaration:

```
{ Private declarations }  
procedure IdleEventHandler(Sender: TObject; var Done: Boolean);
```

In the implementation section, define the procedure:

```
procedure TForm1.IdleEventHandler(Sender: TObject;  
  var Done: Boolean);  
begin  
  {Do a small bit of work here}  
  Done := false;  
end;
```

Then assign the Application's OnIdle event to point to your new procedure(You may do this wherever you like, but a good place would be the Forms OnCreate method):

```
Application.OnIdle := IdleEventHandler;
```

OnIdle is called only once, as the application transitions into an idle state. It is not called continuously unless Done is set to False. If Done is False, WaitMessage is not called. Applications that set Done to False consume an inordinate amount of CPU time that affects overall system performance.

You can set the boolean variable "done" to true when you want to stop the event handler from executing.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16640: passing functions/procedure to other functions/procedures

Question and Answer Database

FAQ1640D.txt    passing functions/procedure to other functions/  
procedures

Category        :Object Pascal

Platform        :All

Product         :All 32 bit

Question:

Can I pass a function or procedure to another function or procedure?

Answer:

Yes. You must declare a procedure or function type to use as the parameter type, then you can pass the address of the procedure that you wish to pass. You must be careful to pass functions or procedures that are compatible with the function or procedure expected, as little error checking will be provided by the compiler.

Example:

```
type
  AFunctionType = function(IntIn : integer) : integer;

function AddProc(IntIn : integer) : integer;
begin
  Result := IntIn + 1;
end;

function SubProc(IntIn : integer) : integer;
begin
  Result := IntIn - 2;
end;

procedure PassAFunction(var IntIn : integer;
                        fn : AFunctionType);
begin
  IntIn := fn(IntIn);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i : integer;

begin
  i := 10;

  PassAFunction(i, @AddProc);
  ShowMessage(IntToStr(i));

  PassAFunction(i, @SubProc);
  ShowMessage(IntToStr(i));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16645: printing a TStringGrid or a TDBGrid

Question and Answer Database

FAQ1645D.txt - printing a TStringGrid or a TDBGrid

Category :Printing  
Platform :All  
Product :

Question:

How do I Print a TStingGrid / TDBGrid?

Answer:

You will need to create two loops, one to print each row, and one to print each column using TPrinter's TextOut method. You should determine the widest line of text in the grid you will need to accommodate and adjust your font size as necessary, or decide if you need to print some of the columns on a separate page.

4/2/99 11:24:30 AM

Last Modified: 01-SEP-99

## Article #16650: printing in color

Question and Answer Database

FAQ1650D.txt - printing in color

Category :Printing

Platform :All

Product :

Question:

How do I print in color?

Answer:

Generally, no special effort is required to print in color assuming the printer is a color printer and is set to its native color mode. Windows will automatically convert color to black and white if the printer does not support color. If you need to change the color mode in code, you can access the DevMode structure of the printer driver.

Example:

```
uses Printers;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    Device : array[0..255] of char;
```

```
    Driver : array[0..255] of char;
```

```
    Port   : array[0..255] of char;
```

```
    hDMode : THandle;
```

```
    PDMode : PDEVMODE;
```

```
begin
```

```
    with Printer do begin
```

```
        PrinterIndex := PrinterIndex;
```

```
        GetPrinter(Device, Driver, Port, hDMode);
```

```
    if hDMode <> 0 then begin
```

```
        pDMode := GlobalLock(hDMode);
```

```
        if pDMode <> nil then begin
```

```
            pDMode.dmFields := pDMode.dmFields or dm_Color;
```

```
            pDMode.dmColor := DMCOLOR_COLOR;
```

```
GlobalUnlock(hDMode);
end;
end;

PrinterIndex := PrinterIndex;
BeginDoc;
Canvas.Font.Color := clRed;
Canvas.TextOut(100,100, 'Red As A Rose!');
EndDoc;
end;
end;
```

4/2/99 11:24:38 AM

Last Modified: 01-SEP-99



## Article #16655: rotating a Metafile

Question and Answer Database

FAQ1655D.txt rotating a Metafile

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I rotate a Metafile?

Answer:

You will need to enumerate through the metafile records using the Windows API functions EnumMetaFile() and EnumEnhMetaFile(). As each Metafile object is returned, you will need to rotate the coordinates of each object. Rectangles will need to be converted to polygons, and ellipses will need to be rendered to polylines under Win16 or replaced with paths under Win32. Fonts should be remapped to TrueType fonts (if necessary) since most non-TrueType fonts do not rotate.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16660: saving TFont information

Question and Answer Database

FAQ1660D.txt saving TFont information

Category :VCL

Platform :All

Product :All 32 bit

Question:

How do I save a TFont object to a File/Registry/ini/Database?

Answer:

You should instead save the font's attributes (name, size, and so forth), not the font object itself. When you read the font information back into your program, you should test to make sure the font still exists before trying to use it.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16664: Colored text on a TStatusBar

Question and Answer Database

FAQ1664D.txt Colored text on a TStatusBar

Category :VCL

Platform :All

Product :All 32 bit

Question:

Why do changes to the StatusBar's font color not have any affect?

Answer:

The status bar is a standard Windows control, and as such, displays the font in the `clBtnText` value, which is set via the Control Panel. This color is black by default, but it can vary due to the user's color scheme. Other standard Windows controls, such as buttons, exhibit this identical behavior. The StatusBar and its associated panels have an owner-draw capability that allow you to draw text in any colors you want. Be sure to change the `Style` property of the `TStatusBar.Panels` to `OwnerDraw`.

Example:

```
procedure TForm1.StatusBar1DrawPanel(StatusBar: TStatusBar;
  Panel: TStatusBarPanel; const Rect: TRect);
begin
  if Panel = StatusBar.Panels[0] then begin
    StatusBar.Canvas.Font.Color := clRed;
    StatusBar.Canvas.TextOut(Rect.Left, Rect.Top, 'Panel - 0')
  end else begin
    StatusBar.Canvas.Font.Color := clGreen;
    StatusBar.Canvas.TextOut(Rect.Left, Rect.Top, 'Panel - 1');
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16668: Typecasting

Question and Answer Database

FAQ1668D.txt    Typecasting

Category       :Object Pascal

Platform       :All

Product        :All 32 bit

Question:

Many Windows functions claim to want PChar parameters in the documentation, but they are defined as requiring LongInts. Is this a bug?

Answer:

No, this is where "typecasting" is used. Typecasting allows you to fool the compiler into thinking that one type of variable is of another type for the ultimate in flexibility. The last parameter of the Windows API function SendMessage() is a good example. It is documented as requiring a long integer, but commonly requires a PChar for certain messages (i.e. WM\_WININICHANGE). Generally, the variable you are typecasting from must be the same size as the variable type you are casting it to. In the SendMessage example, you could typecast a PChar as a longint, since both occupy 4 bytes of memory:

Example:

```
var
  s : array[0..64] of char;
begin
  StrCopy(S, 'windows');
  SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(@S));
end;
```

Pointers are the easiest to typecast, and are the most flexible since you can pass anything to the called procedure. They are also the most dangerous for the same reason. You can also use untyped variable parameters for convenience, although var parameters are really pointers behind the scenes.

Example:

```
type
  PBigArray = ^TBigArray;
  TBigArray = Array[0..65000] of char;

procedure ZeroOutPointerVariable(P : pointer; size : integer);
var
  i : integer;

begin
  for i := 0 to size - 1 do
    PBigArray(p)^[i] := #0;
  end;

procedure ZeroOutUntypedVariable(var v, size : integer);
var
  i : integer;

begin
  for i := 0 to size - 1 do
    TBigArray(v)[i] := #0;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  s : string[255];
  i : integer;
  l : longint;

begin
  s := 'Hello';
  i := 10;
  l := 2000;

  ZeroOutPointerVariable(@s, sizeof(s));
  ZeroOutPointerVariable(@i, sizeof(i));
  ZeroOutPointerVariable(@l, sizeof(l));

  s := 'Hello';
  i := 10;
  l := 2000;

  ZeroOutUntypedVariable(s, sizeof(s));
  ZeroOutUntypedVariable(i, sizeof(i));
```

```
ZeroOutUntypedVariable(1, sizeof(1));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16672: IDE highlighting the incorrect line

Question and Answer Database

FAQ1672D.txt IDE highlighting the incorrect line

Category :IDE

Platform :All

Product :All 32 bit

### Question:

On one project, the IDE insists on highlighting the incorrect line for different conditions. For example, when a syntax error is highlighted, the line above the error is highlighted or when I set breakpoints by choosing a blue dot in the gutter, it does not "line up" with the text line. How can I fix this?

### Answer:

This condition is usually caused by opening the file in a different editor than the editor provided by the IDE. If a line of code is somehow modified and then saved back to the disk using only a carriage return for a line terminating character (instead of a carriage return + line feed sequence), the IDE may get confused. To fix the problem, load the file into an editor that will save each line with a carriage return + line feed sequence.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16676: Determining the version/release of Delphi 3

Question and Answer Database

FAQ1676D.txt - Determining the version/release of Delphi 3

Category :Delphi 3.x  
Platform :All Windows  
Product :Delphi3.x,

Question:

How do I find out what exact version/release of Delphi 3 I have?

Answer:

You can determine the version/release of Delphi 3 by checking the modified time stamp of the file "dclldb30.dpl". Do step 1A if you want to check the version/release currently installed on your system, or step 1B if you want to check the version/release on your Delphi 3 CD. Then follow steps 2 and 3.

1A. Do a find files or folders search for "dclldb30.dpl" on the drive where Delphi 3 was installed. (by default in the "...\borland\delphi 3\bin" directory).

1B. With the Delphi 3 CD inserted, do a find files or folders search for "dclldb30.dpl" on the CD-ROM drive.

2. Right click on the file and go to properties, then read the modified time stamp.

3. Time stamp 3:00:00 A.M. is version 3.00; 3:01:00 A.M. is version 3.01; 3:02:00 A.M. is version 3.02; etc. Version 3.01 is release 1; version 3.02 is release 2; etc. At the time of this writing, the newest version of Delphi 3 was 3.02 release 2).

You can also get:

"/devsupport/delphi/download\_files/vcnp.zip" version checking utility

"/devsupport/delphi/download\_files/vcp.exe" version checking utility (Package EXE)

4/1/99 12:41:30 PM

Last Modified: 01-SEP-99



## Article #16680: Component Palette Corruption Problems

Question and Answer Database

FAQ1680D.txt      Component Palette Corruption Problems

Category        :Install Issues

Platform        :All

Product         :Delphi 3.x

### Question:

I installed Delphi 3 successfully; no errors occurred. Delphi 3 starts up OK, but the component palette display is corrupted: part of one icon appears on part of the next; images are "drifting" off the buttons; the wrong images on buttons, etc. Restarting the computer does not help. What's going on?

(Note: Similar cases have been reported from Delphi 2. Same workarounds/solutions may also apply to Delphi 2.)

### Answer:

USERHTML

The problem is most likely a compatibility conflict with the video driver. Borland tests its products against generic S3 drivers provided by Microsoft or S3 Inc. Any drivers compliant with this standard should not cause this problem with Delphi 3. Video drivers that do not fully comply with this standard may or may not function correctly with Delphi 3. Refer to [Part I](#) and [Part II](#) if you using any of the following high performance video card types with Delphi 3: [Matrox Millennium](#), [Diamond Viper](#), [Neo Magic](#) or a "[Non-Standard Souped-up S3 Card](#)". Otherwise, refer to [Part II](#).

### Part I.

It has been reported that certain workstation configurations that include the above high performance video cards, using their default high performance video drivers, may cause the Delphi 3 component palette to appear corrupted. Borland and cooperative video card vendors are currently investigating these reports. See [Part II](#) if the following workarounds or solutions are not successful:

## Matrox Millennium:

Borland and Matrox are investigating the issue, but neither have been able to reproduce the problem using the Matrox Millennium video card. Possible workarounds that customers have reported are:

- Turning off "Bitmap Caching" in the Matrox driver options.
- Switching to 16-bit or 24-bit Color Mode.
- (Contact Matrox for support on these workarounds.)

## Diamond Viper:

For The Diamond Viper 330, Diamond has advised using the generic S3 drivers provided by Microsoft or S3 Inc. as the solution for their video cards.

(Contact Diamond for support on this solution.)

## Neo Magic:

For The Neo Magic 128XD video card, a reported workaround is to change the video mode to 16-bit or 24-bit color.

(Contact Neo Magic for support on this workaround.)

## "Non-Standard Souped-up S3 Cards":

A "Non-Standard Souped-up S3 Card" is a video card built around the S3 chip which is running drivers to which the vendor has added performance tricks above and beyond the generic S3 drivers. S3 chips are used in several different video cards from different vendors (STB, Orchid, Number9, Diamond, etc.). Please contact your specific video card vendor about getting a compatible generic S3-compliant video driver. Also see Part II directly below.

## Part II.

Try running Delphi 3 under Safe Mode in Windows 95 or VGA Mode under Windows NT. This loads only standard-generic video drivers which are compliant with the standard S3 drivers provided by

Microsoft or S3 Inc.

If this fixes the display of the component palette, then the problem is most likely a compatibility conflict with the video driver. You should get an updated version of your video driver from your video card vendor or switch your default driver to a generic compatible driver from Microsoft or S3 Inc. (which may be available on your Windows 95 or Windows NT CD-ROM).

As of this writing, the above workarounds are the only ones known to Inprise. Since Inprise and the related video card vendors have been unable to reproduce this issue, these workarounds have been provided by Inprise users, and have not been directly tested by Inprise. If none of these workarounds solves your problem, then please post a detailed message to the [INPRISE Newsgroups](#) forum [borland.public.install.delphi](#) and/or contact your video card vendor.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16683: Runtime Error 255

Question and Answer Database

FAQ1683D.txt - Runtime Error 255

Category :Install Issues  
Platform :All-32Bit  
Product :All32Bit,

### Question:

I am getting runtime error 255 when using Delphi 3. What does it mean?

### Answer:

This error is reported if there is a hardware or other exception that isn't caught (and that is not one of the error codes listed below). As of this writing, Error 255 in Delphi 3 appears to be caused by certain video drivers, or by workstations using the AMD NexGen x586 CPU. If your PC is not using the AMD NexGen x586, then the problem is probably the video driver. Try changing driver configuration, getting a newer video driver, or using the generic S3 drivers provided by Microsoft or S3 Inc.

In the case of the AMD NexGen x586, AMD no longer officially supports it, but there is a "Windows CPU Identifier tool" reported as fixing this runtime error in a file called "winid20.exe", packaged in an Update Kit called <ftp://ftp.amd.com/pub/nexgen/updtpi31.exe>

(this "update kit" is actually a self-extracting executable file, from

which you extract, and then execute, the "winid20.exe" tool. For further

details, please refer to AMD's

<http://www.amd.com/products/cpg/nx586/support/toolbox.html>

Nx586 Processor Toolbox home page.)

Note: if your PC uses the VESA Local Bus instead of the PCI bus, you'll want to download the package <ftp://ftp.amd.com/pub/nexgen/updtvl30.exe> instead.

Listing of Delphi 3 Runtime Errors:

```
STATUS_INTEGER_DIVIDE_BY_ZERO:   ErrCode := 200;
STATUS_ARRAY_BOUNDS_EXCEEDED:   ErrCode := 201;
STATUS_FLOAT_OVERFLOW:          ErrCode := 205;
STATUS_FLOAT_INEXACT_RESULT,
STATUS_FLOAT_INVALID_OPERATION,
STATUS_FLOAT_STACK_CHECK:       ErrCode := 207;
STATUS_FLOAT_DIVIDE_BY_ZERO:    ErrCode := 200;
STATUS_INTEGER_OVERFLOW:        ErrCode := 215;
STATUS_FLOAT_UNDERFLOW,
STATUS_FLOAT_DENORMAL_OPERAND:  ErrCode := 206;
STATUS_ACCESS_VIOLATION:        ErrCode := 216;
STATUS_PRIVILEGED_INSTRUCTION:  ErrCode := 218;
STATUS_CONTROL_C_EXIT:          ErrCode := 217;
STATUS_STACK_OVERFLOW:          ErrCode := 202;
```

4/2/99 11:53:57 AM

Last Modified: 01-SEP-99

## Article #16687: Doing a complete reinstallation of Delphi 3

Question and Answer Database

FAQ1687D.txt - Doing a complete reinstallation of Delphi 3

Category :Install Issues  
Platform :All-32Bit  
Product :Delphi3.x, Delphi4.x,

Question:

I need to do a complete reinstallation of Delphi 3. How do I do this?

Answer:

Uninstall Delphi 3 using the following five steps:

Back up all project files;

Optional: back up aliases by backing up idapi32.cfg, using it to replace the newly installed idapi32.cfg after reinstallation;

It is recommended that all third party and custom components be reinstalled after the complete reinstallation of Delphi 3; Use the windows add/remove programs utility to remove Delphi 3, answering "yes" to remove all files; Manually remove the "Delphi 3" directory from your system.

Install Delphi 3 using the following five steps:

Insert the Delphi 3 CD into CD-ROM Drive;

If auto-play is enabled, the Delphi 3 Setup Launcher will automatically run. If the Delphi 3 Setup Launcher does not automatically run, run the file install.exe in the root directory of your Delphi 3 CD;

Follow the on screen instructions to install Delphi 3;

From Optional Uninstall #2 above: back up idapi32.cfg by copying over the newly installed idapi32.cfg;

Install all third party and custom components.

4/2/99 11:54:57 AM



## Article #16691: Calls to the abort method fail to compile

Question and Answer Database

FAQ1691D.txt      Calls to the abort method fail to compile

Category        :VCL

Platform        :All

Product         :All 32 bit

### Question:

When I use any of the BDE access units (DbiTypes, DbiProcs, DbiErrs), any attempt to raise a "silent exception" using calls to the abort procedure in my unit fail to compile, and I receive the error "Statement expected, but expression of type 'Integer' found". I have looked for the DbiTypes.pas, DbiProcs.pas, and DbiErrs.pas files to check for an inconsistency, but I am unable to find these files. Where are the files located, and how can I get around this error?

### Answer:

The DbiTypes, DbiProcs, DbiErrs units are aliased to the "BDE" unit, as declared under the IDE's main menu for Projects->Options->Directories/Conditionals->Unit Aliases. The source code for the DBE unit is located in Delphi's "doc" directory, and is named "BDE.INT". In this file, you will find that there is a constant named ABORT that has a declared value of -2. Since you want to use the procedure Abort() that is located in the SysUtils unit, you will need to fully qualify your call to the abort procedure by prefixing the name of the SysUtils unit to the Abort procedure.

### Example:

```
SysUtils.Abort;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16695: Changing the main form of an app. at runtime

Question and Answer Database

FAQ1695D.txt Changing the main form of an app. at runtime

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to change the main form of an application dynamically at runtime?

Answer:

It is possible to change the applications main form dynamically at runtime, but only during the applications start up. To do this, you will need to choose "View->Project Source" from the IDE's main menu, and edit the applications initialization code, changing the forms creation order in response to some condition. Note: You will need re-edit this code if you add additional forms to the project, as the IDE's code wizard will not be expecting the changes that you have made to the main project file, and you may find that the creation code for the additional forms you have added may have been inserted in an unexpected position within the code.

```
begin
  Application.Initialize;
  if SomeCondition then begin
    Application.CreateForm(TForm1, Form1);
    Application.CreateForm(TForm2, Form2);
  end else begin
    Application.CreateForm(TForm2, Form2);
    Application.CreateForm(TForm1, Form1);
  end;
  Application.Run;
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16699: Showing dbgrid is disabled

Question and Answer Database

FAQ1699D.txt Showing dbgrid is disabled

Category :VCL

Platform :All

Product :All 32 bit

Question:

Some of the database components (notably the TDBGrid component) does not turn gray or give any visual notification when the control is disabled. How can I provide this functionability?

Answer:

The following example demonstrates changing the text color of the control to clGray when the control is disabled, giving a visual indication that the control is disabled.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DbGrid1.Enabled := false;
  DbGrid1.Font.Color := clGray;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  DbGrid1.Enabled := true;
  DbGrid1.Font.Color := clBlack;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16703: Detecting the full path/filename of a DLL

Question and Answer Database

FAQ1703D.txt Detecting the full path/filename of a DLL

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I detect (from a dynamic link library) the full path and file name of where the DLL is running from?

Answer:

The following example demonstrates a dll function that will detect the full path of where the dll was loaded from.

Example:

```
uses Windows;
```

```
procedure ShowDllPath stdcall;
```

```
var  
    TheFileName : array[0..MAX_PATH] of char;
```

```
begin  
    FillChar(TheFileName, sizeof(TheFileName), #0);  
    GetModuleFileName(hInstance, TheFileName, sizeof(TheFileName));  
    MessageBox(0, TheFileName, 'The DLL file name is:', mb_ok);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16707: Programmatically dropping down a menu

Question and Answer Database

FAQ1707D.txt Programmatically dropping down a menu

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I programmatically drop a menu down in code?

Answer:

The following example demonstrates both dropping down a menu and selecting a menu item by synthesizing menu shortcut keys. If your application does not have shortcut keys assigned to the menu items, you can send combinations of the VK\_MENU, VK\_LEFT, VK\_DOWN, and VK\_RETURN keys to programmatically navigate the menu.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
//Allow button to finish painting in response to the click
  Application.ProcessMessages;
  {Alt Key Down}
  keybd_Event(VK_MENU, 0, 0, 0);
  {F Key Down - Drops the menu down}
  keybd_Event(ord('F'), 0, 0, 0);
  {F Key Up}
  keybd_Event(ord('F'), 0, KEYEVENTF_KEYUP, 0);
  {Alt Key Up}
  keybd_Event(VK_MENU, 0, KEYEVENTF_KEYUP, 0);
  {F Key Down}
  keybd_Event(ord('S'), 0, 0, 0);
  {F Key Up}
  keybd_Event(ord('S'), 0, KEYEVENTF_KEYUP, 0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16711: Executing the explorer find file dialog

Question and Answer Database

FAQ1711D.txt Executing the explorer find file dialog

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I execute the Windows Explorer Find File Dialog Box?

Answer:

The following example demonstrates using DDE to execute Explorer's find file dialog. The example opens the dialog in the Directory "C:\Download".

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with TDDEClientConv.Create(Self) do begin
    ConnectMode := ddeManual;
    ServiceApplication := 'explorer.exe';
    SetLink( 'Folders', 'AppProperties');
    OpenLink;
    ExecuteMacro('[FindFolder(, C:\DOWNLOAD)]', False);
    CloseLink;
    Free;
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16715: Radiogroups and focus

Question and Answer Database

FAQ1715D.txt Radiogroups and focus

Category :VCL

Platform :All

Product :All 32 bit

Question:

When tabbing through the controls on a form, a radio group requires two tab keystrokes if an item is selected, but only one if no item is selected. Since two tab keystrokes are required in this case one of the tab keystrokes gives no visual indication that the focused control is the RadioGroup. Is there any way to make the behavior more consistent?

Answer:

Setting the TabStop property of the RadioGroup to false should correct the problem, yet still allow the tab key to function, so long as one item is selected in the RadioGroup.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16719: Trapping the Windows system key 'PrintScreen'

Question and Answer Database

FAQ1719D.txt Trapping the Windows system key 'PrintScreen'

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I globally trap the Windows system key "PrintScreen"?

Answer:

The following example demonstrates registering a hot key with the system to globally trap the windows printscreen key.

Example:

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
    procedure WMHotKey(var Msg : TWMHotKey); message WM_HOTKEY;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

const id_SnapShot = 101;

procedure TForm1.WMHotKey (var Msg : TWMHotKey);
begin
  if Msg.HotKey = id_SnapShot then
```

```
    ShowMessage('GotIt');  
end;  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    RegisterHotKey(Form1.Handle,  
                  id_SnapShot,  
                  0,  
                  VK_SNAPSHOT);  
end;  
  
procedure TForm1.FormDestroy(Sender: TObject);  
begin  
    UnRegisterHotKey (Form1.Handle, id_SnapShot);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16723: Accessing HKEY\_LOCAL\_MACHINE under NT without Admin. rights

Question and Answer Database

FAQ1723D.txt      Accessing HKEY\_LOCAL\_MACHINE under NT without Admin.  
rights

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

When using the TRegistry component under NT, a user with less than "administrator" rights is unable to access information stored in the HKEY\_LOCAL\_MACHINE. How can I work around this?

Answer:

The problem is caused by the fact that TRegistry (and the derived TRegInifile) always opens a key with KEY\_ALL\_ACCESS, even if only KEY\_READ would be needed. You can avoid this by going back to using the API registry functions (RegOpenKey et al.), or create a new class from the TRegistry component, and change the new TRegistry class to add an access property.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16727: Detecting how many jobs are in the print spooler

Question and Answer Database

FAQ1727D.txt    Detecting how many jobs are in the print spooler  
Category       :Windows API  
Platform        :All  
Product         :All 32 bit

Question:

Is there any way to detect how many jobs are in the Windows print spooler at any given time?

Answer:

The Windows print spooler regularly broadcasts a system wide WM\_SPOOLERSTATUS message each time a job is added or deleted from the spooler que. The following example demonstrates trapping for this message.

Example:

```
type
  TForm1 = class(TForm)
    Label1: TLabel;
  private
    { Private declarations }
    procedure WM_SpoolerStatus(var Msg : TWMSPOOLERSTATUS);
      message WM_SPOOLERSTATUS;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WM_SpoolerStatus(var Msg : TWMSPOOLERSTATUS);
begin
```

```
Label1.Caption := IntToStr(msg.JobsLeft) +  
                  ' Jobs currently in spooler';  
msg.Result := 0;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16731: Bitwise AND evaluates different under Delphi 3

Question and Answer Database

FAQ1731D.txt     Bitwise AND evaluates different under Delphi 3  
Category        :Object Pascal  
Platform        :All  
Product         :Delphi 3.x

### Question:

Under Delphi 3 (all versions), on some occasions, bitwise arithmetic using "AND" against integers does not always render the same results as Delphi 2. How can I get around this?

### Answer:

This is a known problem, where in certain circumstances, the compiler produces a sign extended negative result in a bitwise "AND" operation. The solution is to compare the result against zero, instead of comparing the result as being greater than zero. The following example demonstrates both the problem and the solution.

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    IntVal : Integer;  
begin  
    IntVal := 128 + 1;  
    if (IntVal and 128) > 0 then  
        ShowMessage('This never executes!');  
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    IntVal : Integer;  
begin  
    IntVal := 128 + 1;  
    if (IntVal and 128) <> 0 then  
        ShowMessage('This is correct!');  
end;
```



## Article #16735: Dialogs displaying 'No to all' button instead of mbHelp

Question and Answer Database

FAQ1735D.txt Dialogs displaying 'No to all' button instead of mbHelp  
Category :VCL  
Platform :All  
Product :Delphi 3.x

### Question:

When I upgraded to Delphi 3, some of my dialogs now display a button with a caption title of "No to all" where they used to display the mbHelp button. The Delphi help file and documentation give no mention to this additional button. What is this new button, and how can I prevent this behavior?

### Answer:

Delphi 3 introduced two new buttons for the message dialog boxes: mbNoToAll and mbYesToAll. The new button constants were inserted where mbHelp used to reside. The most likely cause of the problem is that you are using a hard coded ordinal value or perhaps are manipulating the value used to set the button type requested in a non portable fashion, instead of relying on more portable constants such as mbHelp.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16739: '\$2109 Cannot initialize the BDE' error when deploying InstallShield application

Question and Answer Database

FAQ1739D.txt '\$2109 Cannot initialize the BDE' error when deploying InstallShield application  
Category :Install Issues  
Platform :All  
Product :Delphi 3.x

### Question:

I get error \$2109 "Cannot initialize the Borland Database Engine" when running an application deployed with InstallShield Express. What's wrong?

### Answer:

A problem with either ISExpress or your setup configuration in ISExpress has caused the following entry to be left out of the registry:

HKEY\_LOCAL\_MACHINE\SOFTWARE\BORLAND\DATABASE ENGINE | DLLPATH.  
This entry is probably empty on the deployed machine's registry. Enter the path to IDAPI32.DLL (usually \borland\common files\bde)

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16743: Raize components: 'Can't Load Package' error

Question and Answer Database

FAQ1743D.txt Raize components: 'Can't Load Package' error

Category :Install Issues

Platform :All

Product :Delphi 3.x

Question:

Raize components: "Can't Load Package" error: What does this error mean?

Answer:

When installing the RzLstpck.dpc file, Component | "Install Packages..." | "Add...", make sure to click the "Directories..." button and set the correct System entry to the correct location of your Windows\System directory.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16747: Local InterBase: 'Internal Error; Near: query shellmgr' error

Question and Answer Database

FAQ1747D.txt - Local InterBase: 'Internal Error; Near: query shellmgr' error

Category :Install Issues  
Platform :All-32Bit  
Product :

Question:

Local InterBase: "Internal Error; Near: query shellmgr" error:  
What is the cause and how do I fix this error?

Answer:

This is actually a problem with some Dell machines. The problem is associated with Dell Pentium Pro machines with NT 4.0 pre-installed. There is a setting in the registry that is modified so that Dell can load up an explorer extension.

Workaround:

Run Regedit

Modify the key: HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT  
\CurrentVersion\WinLogon\Shell

Originally this will say: Explorer.exe userhook.

Modify the key to say: Explorer.exe

Reboot the machine

This is documented on InstallShield's website and it is not a problem in the IB install.

4/2/99 11:56:25 AM

Last Modified: 01-SEP-99

## Article #16751: Cannot write EXE or DLL to disk.

Question and Answer Database

FAQ1751D.txt      Cannot write EXE or DLL to disk.

Category        :Compiler

Platform        :All

Product         :All 32 bit

### Question:

Occasionally, when building a dynamic link library or executable file, the IDE responds that it cannot write the exe or dll file. How can I get around this?

### Answer:

This will happen if the executable or dynamic link library is still in memory. If you have executed the file outside of the IDE, and then try to compile your project, the Windows system will not allow the compiler to overwrite the file. The only recourse is to make sure the application is not running, or your dll is not in memory when you attempt to compile the project.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16754: How can I have a TBitBtn component that has a word wrapped caption?

Question and Answer Database

FAQ1754D.txt    How can I have a TBitBtn component that has a word wrapped caption?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I have a TBitBtn component that has a word wrapped caption?

Answer:

The following example demonstrates drawing word wrapped text directly on a TBitBtn's glyph.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  R : TRect;
  N : Integer;
  Buff : array[0..255] of Char;
begin
  with BitBtn1 do begin
    Caption := 'A really really long caption';
    Glyph.Canvas.Font := Self.Font;
    Glyph.Width := Width - 6;
    Glyph.Height := Height - 6;
    R := Bounds(0, 0, Glyph.Width, 0);
    StrPCopy(Buff, Caption);
    Caption := '';
    DrawText(Glyph.Canvas.Handle,
              Buff,
              StrLen(Buff),
              R,
              DT_CENTER or DT_WORDBREAK or DT_CALCRECT);
    OffsetRect(R,
              (Glyph.Width - R.Right) div 2,
              (Glyph.Height - R.Bottom) div 2);
```

```
DrawText (Glyph.Canvas.Handle ,  
          Buff ,  
          StrLen (Buff) ,  
          R ,  
          DT_CENTER or DT_WORDBREAK) ;
```

```
end ;
```

```
end ;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16758: Bitmaps and captions on TBitBtn

Question and Answer Database

FAQ1758D.txt     Bitmaps and captions on TBitBtn

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

When I use BitBtn, the Caption(text) and bitmap file cannot be seen at the same time. How can I work around this?

Answer:

This can happen if the bitmap is too large. The TBitBtn class displays the bitmap first, and then displays the caption above, below, to the right, or to the left, of the bitmap (depending on the Layout property). If the bitmap is as wide as the button, then there isn't any room to display the caption.

If you are trying to use bitmaps which are the same size as your button (and you want the caption displayed on top of the bitmap), you will need to create a bitmap image with the captioned text drawn onto the bitmap's canvas.

Example:

```
var
  bm : TBitmap;
  OldBkMode : integer;
begin
  bm := TBitmap.Create;
  bm.Width := BitBtn1.Glyph.Width;
  bm.Height := BitBtn1.Glyph.Height;
  bm.Canvas.Draw(0, 0, BitBtn1.Glyph);
  OldBkMode := SetBkMode(bm.Canvas.Handle, Transparent);
  bm.Canvas.TextOut(0, 0, 'The Caption');
  SetBkMode(bm.Canvas.Handle, OldBkMode);
  BitBtn1.Glyph.Assign(bm);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16762: Detecting print errors

Question and Answer Database

FAQ1762D.txt - Detecting print errors

Category :Printing  
Platform :All  
Product :

### Question:

How can I detect printing errors (like the printer is out of paper) when printing from my application?

### Answer:

Print jobs are generally added to the Windows print spooler, where the job will be spooled to the printer at some future time (perhaps even after your application has been shut down). Even if the spooler is not used, it is the print driver that takes the responsibility of dealing with errors at the printer level. The only way to handle printer errors at application level is to print to a dedicated printer through a custom device driver that is built for that specific printer.

4/2/99 11:24:46 AM

Last Modified: 01-SEP-99

## Article #16766: Disabling system menu in code

Question and Answer Database

FAQ1766D.txt Disabling system menu in code

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I disable the forms system menu and the Minimize, Maximize, and Close Buttons in code at Runtime?

Answer:

The following demonstrates disabling the buttons and system menu on the form's caption bar.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {Disable}
  Form1.BorderIcons := Form1.BorderIcons -
                        [biSystemMenu, biMinimize, biMaximize];
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  {Enable}
  Form1.BorderIcons := Form1.BorderIcons +
                        [biSystemMenu, biMinimize, biMaximize];
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16770: Avoiding flicker on DrawCell

Question and Answer Database

FAQ1770D.txt     Avoiding flicker on DrawCell

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

When I set the font color on a DrawGrid's DrawCell event using  
Font.Color := clRed; I get an endless loop of flicker. How can I get  
around this?

Answer:

Be sure to fully qualify the canvas that you are using.

Example:

```
    If (Row = 0) then begin
        DrawGrid1.Canvas.Font.Color := clRed;
        DrawGrid1.Canvas.TextOut(Rect.Left,Rect.Top, IntToStr(Col));
    end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16774: How can I determine the names of the installed comm ports?

Question and Answer Database

FAQ1774D.txt    How can I determine the names of the installed comm ports?

Category        :Windows API  
Platform        :All  
Product         :All 32 bit

Question:

How can I determine the names of the installed comm ports?

Answer:

The following example demonstrates enumerating the communications ports that are installed and listed in the Win32 registry.

Example:

```
uses Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
  reg : TRegistry;
  ts : TStringList;
  i : integer;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.OpenKey('hardware\devicemap\serialcomm',
             false);
  ts := TStringList.Create;
  reg.GetValueNames(ts);
  for i := 0 to ts.Count -1 do begin
    Mem1.Lines.Add(reg.ReadString(ts.Strings[i]));
  end;
  ts.Free;
  reg.CloseKey;
  reg.free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16778: Extracting Icons

Question and Answer Database

FAQ1778D.txt     Extracting Icons

Category        :Windows API

Platform        :All

Product         :All 32 bit

### Question:

How can I use the undocumented ShellAPI function `ExtractIconEx()` to extract the small and large icons from a given file?

### Answer:

You can use the undocumented ShellAPI function `ExtractIconEx()` to extract both large and small icons from an executable file, dynamic-link library (DLL), or icon file.

The function is prototyped as:

```
function ExtractIconEx(lpszFile: PAnsiChar;
                       nIconIndex: Integer;
                       phiconLarge : PhIconArray;
                       phiconSmall: PhIconArray;
                       nIcons: UINT): UINT; stdcall;
```

### Parameters

#### `lpszFile`:

Pointer to a null-terminated string specifying the name of the executable, DLL, or icon file to extract the icon from.

#### `nIconIndex`:

Specifies the index of the icon to retrieve. If this value is 0, the function returns the handle of the first icon in the specified file. If the value is -1 and `phiconLarge` and `phiconSmall` are both nil, the function returns the total number of icons in the executable, DLL, or icon file to extract the icon from. If the file is an executable or DLL file, the return value is the number of `RT_GROUP_ICON` resources. If the file is an .ICO file, the return value is 1.

phiconLarge:

Pointer to an array of icon handles for the large icons returned. This parameter can be nil if only one icon is to be extracted.

phiconSmall

Pointer to an array of icon handles for the small icons returned. This parameter can be nil if only one icon is to be extracted.

nIcons

Specifies the number of icons to extract.

Return Value

The return value is the handle to an icon if only one icon is to be extracted. If there were no icons found in the file, the return value is 0.

The following example demonstrates using the ExtractIconEx() function. Note that the functions are prototyped differently than as prototyped in the ShellAPI unit, to allow the full usage of the function.

```
type ThIconArray = array[0..0] of hIcon;
type PhIconArray = ^ThIconArray;
```

```
function ExtractIconExA(lpszFile: PAnsiChar;
    nIconIndex: Integer;
    phiconLarge : PhIconArray;
    phiconSmall: PhIconArray;
    nIcons: UINT): UINT; stdcall;
external 'shell32.dll' name 'ExtractIconExA';
```

```
function ExtractIconExW(lpszFile: PWideChar;
    nIconIndex: Integer;
    phiconLarge: PhIconArray;
    phiconSmall: PhIconArray;
    nIcons: UINT): UINT; stdcall;
external 'shell32.dll' name 'ExtractIconExW';
```

```
function ExtractIconEx(lpszFile: PAnsiChar;
    nIconIndex: Integer;
    phiconLarge : PhIconArray;
    phiconSmall: PhIconArray;
    nIcons: UINT): UINT; stdcall;
```

```
external 'shell32.dll' name 'ExtractIconExA';
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    NumIcons : integer;  
    pTheLargeIcons : phIconArray;  
    pTheSmallIcons : phIconArray;  
    LargeIconWidth : integer;  
    SmallIconWidth : integer;  
    SmallIconHeight : integer;  
    i : integer;  
    TheIcon : TIcon;  
    TheBitmap : TBitmap;
```

```
begin
```

```
    NumIcons :=
```

```
    ExtractIconEx('C:\Program Files\Borland\Delphi 3\BIN\delphi32.exe',  
                -1,  
                nil,  
                nil,  
                0);
```

```
if NumIcons > 0 then begin
```

```
    LargeIconWidth := GetSystemMetrics(SM_CXICON);  
    SmallIconWidth := GetSystemMetrics(SM_CXSMICON);  
    SmallIconHeight := GetSystemMetrics(SM_CYSMICON);  
    GetMem(pTheLargeIcons, NumIcons * sizeof(hIcon));  
    GetMem(pTheSmallIcons, NumIcons * sizeof(hIcon));  
    FillChar(pTheLargeIcons^, NumIcons * sizeof(hIcon), #0);  
    FillChar(pTheSmallIcons^, NumIcons * sizeof(hIcon), #0);
```

```
    ExtractIconEx('C:\Program Files\Borland\Delphi 3\BIN\delphi32.exe',  
                0,  
                pTheLargeIcons,  
                pTheSmallIcons,  
                numIcons);
```

```
{ $IFOPT R+ }
```

```
    { $DEFINE CKRANGE }
```

```
    { $R- }
```

```
{ $ENDIF }
```

```
for i := 0 to (NumIcons - 1) do begin
```

```
    DrawIcon(Form1.Canvas.Handle,  
            i * LargeIconWidth,  
            0,  
            pTheLargeIcons^[i]);
```

```
    TheIcon := TIcon.Create;
```

```
    TheBitmap := TBitmap.Create;
```

```
TheIcon.Handle := pTheSmallIcons^[i];
TheBitmap.Width := TheIcon.Width;
TheBitmap.Height := TheIcon.Height;
TheBitmap.Canvas.Draw(0, 0, TheIcon);
TheIcon.Free;
Form1.Canvas.StretchDraw(Rect(i * SmallIconWidth,
                               100,
                               (i + 1) * SmallIconWidth,
                               100 + SmallIconHeight),
                          TheBitmap);
```

```
    TheBitmap.Free;
```

```
end;
```

```
{ $IFDEF CKRANGE }
```

```
    { $UNDEF CKRANGE }
```

```
    { $R+ }
```

```
{ $ENDIF }
```

```
FreeMem(pTheLargeIcons, NumIcons * sizeof(hIcon));
```

```
FreeMem(pTheSmallIcons, NumIcons * sizeof(hIcon));
```

```
end;
```

```
end;
```

```
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16782: Adjusting number and with of columns in a TFileListBox

Question and Answer Database

FAQ1782D.txt Adjusting number and with of columns in a TFileListBox

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to adjust the number of columns and width of columns in a TFileListBox component?

Answer:

The following example demonstrates casting a FileListBox component to a TDirectoryListBox component so additional columns can be added and sized.

Example:

```
with TDirectoryListBox(FileListBox1) do begin
  Columns := 2;
  SendMessage(Handle,
    LB_SETCOLUMNWIDTH,
    Canvas.TextWidth('WWWWWWWWW.WWW'),
    0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16786: Notifying forms of global changes

Question and Answer Database

FAQ1786D.txt     Notifying forms of global changes

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I tell all created (and even hidden), forms of an application that some global option has changed?

Answer:

One way is to create a user message, and use the preform method to send the message to all forms in the Screen.Forms array.

Example:

```
{Code for Unit1}
```

```
const
  UM_MyGlobalMessage = WM_USER + 1;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Button1: TButton;
    procedure FormShow(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    { Private declarations }
  private
    procedure UMMyGlobalMessage(var AMessage: TMessage); message
      UM_MyGlobalMessage;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

{ \$R \*.DFM }

uses Unit2;

procedure TForm1.FormShow(Sender: TObject);

begin

    Form2.Show;

end;

procedure TForm1.UMMyGlobalMessage(var AMessage: TMessage);

begin

    Label1.Left := AMessage.WParam;

    Label1.Top := AMessage.LParam;

    Form1.Caption := 'Got It!';

end;

procedure TForm1.Button1Click(Sender: TObject);

var

    f: integer;

begin

    for f := 0 to Screen.FormCount - 1 do

        Screen.Forms[f].Perform(UM\_MyGlobalMessage, 42, 42);

end;

{Code for Unit2}

const

    UM\_MyGlobalMessage = WM\_USER + 1;

type

    TForm2 = class(TForm)

        Label1: TLabel;

    private

        { Private declarations }

        procedure UMMyGlobalMessage(var AMessage: TMessage); message

            UM\_MyGlobalMessage;

    public

        { Public declarations }

end;

var

```
Form2: TForm2;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm2.UMMyGlobalMessage(var AMessage: TMessage);  
begin  
  Label1.Left := AMessage.WParam;  
  Label1.Top  := AMessage.LParam;  
  Form2.Caption := 'Got It!';  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16790: How can I activate different hints for each cell in a grid

Question and Answer Database

FAQ1790D.txt    How can I activate different hints for each cell in a grid

Category        :VCL  
Platform        :All  
Product         :All 32 bit

Question:

How can I activate different hints for each cell in a grid control?

Answer:

The following example demonstrates tracking the mouse movements in a StringGrid component. If the mouse moves to a different cell in the grid, a new hint window is displayed, showing the row and column numbers of the cell.

Example:

```
type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    procedure StringGrid1MouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    Col : integer;
    Row : integer;
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  StringGrid1.Hint := '0 0';
  StringGrid1.ShowHint := True;
end;

procedure TForm1.StringGrid1MouseMove(Sender: TObject;
Shift: TShiftState; X, Y: Integer);
var
  r : integer;
  c : integer;
begin
  StringGrid1.MouseToCell(X, Y, C, R);
  if ((Row <> r) or
    (Col <> c)) then begin
    Row := r;
    Col := c;
    Application.CancelHint;
    StringGrid1.Hint := IntToStr(r) + #32 + IntToStr(c);
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16794: Dynamically drawing a transparent image

Question and Answer Database

FAQ1794D.txt Dynamically drawing a transparent image

Category :VCL

Platform :All

Product :All 32 bit

Question:

How do I use a TImageList component to dynamically draw a transparent image?

Answer:

The following example demonstrates the dynamic creation of a TImageList component used to draw a transparent bitmap.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bm : TBitmap;
  il : TImageList;
begin
  bm := TBitmap.Create;
  bm.LoadFromFile('C:\Download\TEST.BMP');
  il := TImageList.CreateSize(bm.Width,
                             bm.Height);

  il.DrawingStyle := dsTransparent;
  il.Masked := true;
  il.AddMasked(bm, clRed);
  il.Draw(Form1.Canvas, 0, 0, 0);
  bm.Free;
  il.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16797: which rows have been selected in a TDBGrid

Question and Answer Database

FAQ1797C.txt    which rows have been selected in a TDBGrid  
Category        :Database Issues  
Platform        :All  
Product         :C++Builder 1.x

Question:

In a MultiSelect DBGrid, how can I determine which rows have been selected?

Answer:

Use the SelectedRows property of the TDBGrid component.

Example:

```
for (int i =0; i < DBGrid1->SelectedRows->Count -1; i++)  
    Table1->GotoBookmark(TBookMark(DBGrid1->SelectedRows->Items  
[i]));
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16798: Porting strings from 16 to 32-bit

Question and Answer Database

FAQ1798D.txt Porting strings from 16 to 32-bit

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

When I port my code from 16 to 32 bit, record types that contain strings do not function correctly. What has happened?

Answer:

Traditionally, strings that are defined with no length parameter would default to having a length of 255 characters. With the advent of 32 bit Delphi, strings with no defined length default to a long string. Long strings are pointers to a reference counted AnsiString. Long string types are not recommended for use in structures that are intended to be written or read from a disk file. To port your code correctly, you need to explicitly set the expected length of the string in the record, otherwise, a long string is used in place of the original string in the record, leading to unexpected results.

Example:

```
ShortStringRec : record
  s : string[255]; {this uses a short string - ok}
end;
```

```
LongStringRec : record
  s : string; {this uses a long string - not ok}
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16801: Distributing IDAPI with your EXE

Question and Answer Database

FAQ1801C.txt     Distributing IDAPI with your EXE

Category        :Database Issues

Platform        :All

Product         :C++Builder 1.x

Question:

Is IDAPI necessary for data access in BCB? Can you 'bundle' IDAPI inside of a BCB EXE so that your distributed application does not need to install IDAPI on your user's computers?

Answer:

IDAPI is necessary for data access in BCB. BCB comes with InstallShield express that allows you to create an installable image that will include the needed IDAPI redistributabl files.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16802: Overwrite in TMemo and TEdit

Question and Answer Database

FAQ1802D.txt Overwrite in TMemo and TEdit

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I get the TEdit and TMemo controls to have a overwrite capability?

Answer:

The Windows TEdit and TMemo controls have no overwrite capability. It is possible to emulate this behavior however, by setting the SelLength property of the edit or memo control to one during the processing of the KeyPress event. This causes the character at the current position of the caret to be overwritten.

The following example demonstrates emulation of an overwrite capability of a TMemo component. The state of the overwrite mode can be toggled by pressing the insert key.

Example:

```
type
  TForm1 = class(TForm)
    Mem1: TMemo;
    procedure Mem1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure Mem1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
    InsertOn : bool;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.Mem1KeyDown(Sender: TObject; var Key: Word;  
  Shift: TShiftState);
```

```
begin
```

```
  if (Key = VK_INSERT) and (Shift = []) then
```

```
    InsertOn := not InsertOn;
```

```
end;
```

```
procedure TForm1.Mem1KeyPress(Sender: TObject; var Key: Char);
```

```
begin
```

```
  if ((Mem1.SelLength = 0) and
```

```
    (not InsertOn)) then
```

```
    Mem1.SelLength := 1;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 12-NOV-99

## Article #16806: Small bitmaps in a TPopUpMenu

Question and Answer Database

FAQ1806D.txt Small bitmaps in a TPopUpMenu

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I place small bitmaps in a TPopUpMenu component?

Answer:

The following example demonstrates adding a bitmap to a PopUpMenu item using the Windows API function `SetMenuItemBitmaps()`. This function takes a handle to the popup menu, the zero based position of the menu item you wish to add the bitmaps to, and two bitmap handles, (one bitmap specifying the image to be shown in the menu unchecked state, and another specifying the image to be shown in the menu checked state).

type

```
TForm1 = class(TForm)
  PopupMenu1: TPopupMenu;
  Pop11: TMenuItem;
  Pop21: TMenuItem;
  Pop31: TMenuItem;
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
```

private

```
{ Private declarations }
  bmUnChecked : TBitmap;
  bmChecked : TBitmap;
```

public

```
{ Public declarations }
```

end;

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  bmUnchecked := TBitmap.Create;
  bmUnchecked.LoadFromFile(
    'C:\Program Files\Borland\BitBtns\ALARMRNG.BMP');
  bmChecked := TBitmap.Create;
  bmChecked.LoadFromFile(
    'C:\Program Files\Borland\BitBtns\CHECK.BMP');
  {Add the bitmaps to the item at index 1 in PopUpMenu}
  SetMenuItemBitmaps(PopUpMenu1.Handle,
    1,
    MF_BYPOSITION,
    BmUnchecked.Handle,
    BmChecked.Handle);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  bmUnchecked.Free;
  bmChecked.Free;
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  pt : TPoint;
begin
  pt := ClientToScreen(Point(x, y));
  PopUpMenu1.Popup(pt.x, pt.y);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16810: Refreshing the windows desktop

Question and Answer Database

FAQ1810D.txt Refreshing the windows desktop

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I cause the Windows Desktop icons to refresh?

Answer:

The following example refreshes the Windows Desktop icons.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    SendMessage(FindWindow('Progman', 'Program Manager'),
                WM_COMMAND,
                $A065,
                0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16814: Explanation of rounding

Question and Answer Database

FAQ1814D.txt    Explanation of rounding

Category       :Object Pascal

Platform       :All

Product        :All 32 bit

Question:

Since Delphi's Round() function uses "bankers rounding" where the value is rounded to the nearest even number, how can I round a floating point number using the more traditional means, where fractional values less than .5 round down, and fractional values of .5 and greater round up?

Answer:

The following function demonstrates rounding down numbers with fractional values of less than .5, and rounding up numbers with fractional values of .5 and greater.

Example:

```
function RoundUp(X: Extended): Extended;
begin
  Result := Trunc(X) + Trunc (Frac(X) * 2);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16818: StretchDraw on an icon

Question and Answer Database

FAQ1818D.txt     StretchDraw on an icon

Category        :VCL

Platform        :All

Product         :All 32 bit

### Question:

When I attempt to use the TCanvas.StretchDraw method to draw a zoomed view of an Icon, the size does not change. How can I get around this?

### Answer:

Icons always draw at the system default icon size. To display a zoomed view of an icon image, copy the icon image to a bitmap, then use the TCanvas.StretchDraw method.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TheBitmap : TBitmap;
begin
  TheBitmap := TBitmap.Create;
  TheBitmap.Width := Application.Icon.Width;
  TheBitmap.Height := Application.Icon.Height;
  TheBitmap.Canvas.Draw(0, 0, Application.Icon);
  Form1.Canvas.StretchDraw(Rect(0,
                                0,
                                TheBitmap.Width * 3,
                                TheBitmap.Height * 3),
    TheBitmap);

  TheBitmap.Free;
end;
```

7/16/98 4:31:28 PM



Last Modified: 01-SEP-99

## Article #16822: How can I determine the week number of a given day in the year?

Question and Answer Database

FAQ1822D.txt    How can I determine the week number of a given day in the year?

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How can I determine the week number of a given day in the year?

Answer:

This is a highly debated subject, as the first week of the year may be considered to be before or after Jan 1. The following function provides a starting point for WeekOfYear calculations.

Example:

```
function WeekOfYear(ADate : TDateTime) : word;
var
  day : word;
  month : word;
  year : word;
  FirstOfYear : TDateTime;
begin
  DecodeDate(ADate, year, month, day);
  FirstOfYear := EncodeDate(year, 1, 1);
  Result := Trunc(ADate - FirstOfYear) div 7 + 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(IntToStr(WeekOfYear(Date)));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16825: How do I change a specific IDAPI configuration setting?

Question and Answer Database

FAQ1825C.txt    How do I change a specific IDAPI configuration setting?

Category        :Database Issues

Platform        :All

Product         :C++Builder 1.x

Question:

How do I change a specific IDAPI configuration setting?

How do I know the current settings?

Answer:

The settings change depending on the database or server. To find the current settings, you can display the TStringList returned from GetAliasParams in a TMemo, then you can modify the alias by replacing specific indexed strings within the TStringList.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TStringList MyList;

    MyList = new TStringList();

    Session1->GetAliasParams('DBDEMOS', MyList);
    Session1->Memo1->Lines->Assign(MyList);
    Session1->MyList->Strings[0] = "PATH=d:\\borland\\BCB\\Demos\\
\\Data';
    Session1->ModifyAlias('DBDEMOS', MyList);

    delete MyList;
}
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16826: Sending a message to all controls on a form

Question and Answer Database

FAQ1826D.txt      Sending a message to all controls on a form

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Is there a way to send a message to all the controls on a form?

Answer:

You can use `Screen.Forms[i].Broadcast(msg);` where `[i]` is the index of the form you wish to broadcast the message to. `Broadcast` works with all `TWinControls` components, and sends messages to all children in the `Controls` array. If one of your children wants to handle the message and have `Broadcast` exit, then set `Msg.Result` to a `NonZero` value during the processing of the message.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16830: Resizing an array

Question and Answer Database

FAQ1830D.txt     Resizing an array

Category     :Compiler

Platform     :All

Product      :All 32 bit

Question:

How can I resize an array?

Answer:

You cannot resize a non-dynamic array in Pascal. You can create and resize a dynamically created array. To do this, you must create the dynamic array, turn range checking off, and access the array members via a variable only, or you will receive runtime and compile time errors. Since you will access the array through a pointer variable, you can dynamically resize the array by creating a new array in memory, copy all the valid elements of the original array to the new array, free the memory for the original array, and assign the new array's pointer back to the original array pointer.

Example:

```
type
  TSomeArrayElement = integer;
  PSomeArray = ^TSomeArray;
  TSomeArray = array[0..0] of TSomeArrayElement;

procedure CreateArray(var TheArray : PSomeArray;
                     NumElements : longint);
begin
  GetMem(TheArray, sizeof(TSomeArrayElement) * NumElements);
end;

procedure FreeArray(var TheArray : PSomeArray;
                   NumElements : longint);
begin
  FreeMem(TheArray, sizeof(TSomeArrayElement) * NumElements);
end;
```

```

procedure ReSizeArray(var TheArray : PSomeArray;
                     OldNumElements : longint;
                     NewNumElements : longint);
var
    TheNewArray : PSomeArray;
begin
    GetMem(TheNewArray, sizeof(TSomeArrayElement) * NewNumElements);
    if NewNumElements > OldNumElements then
        Move(TheArray^,
             TheNewArray^,
             OldNumElements * sizeof(TSomeArrayElement)) else
        Move(TheArray^,
             TheNewArray^,
             NewNumElements * sizeof(TSomeArrayElement));
    FreeMem(TheArray, sizeof(TSomeArrayElement) * OldNumElements);
    TheArray := TheNewArray;
end;

```

```

procedure TForm1.Button1Click(Sender: TObject);
var
    p : PSomeArray;
    i : integer;
begin
    {$IFOPT R+}
    {$DEFINE CKRANGE}
    {$R-}
    {$ENDIF}

    CreateArray(p, 200);
    for i := 0 to 199 do
        p^[i] := i;
    ResizeArray(p, 200, 400);
    for i := 0 to 399 do
        p^[i] := i;
    ResizeArray(p, 400, 50);
    for i := 0 to 49 do
        p^[i] := i;
    FreeArray(p, 50);

    {$IFDEF CKRANGE}
    {$UNDEF CKRANGE}
    {$R+}
    {$ENDIF}
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16834: How can I place a string that is padded with spaces into a

Question and Answer Database

FAQ1834D.txt    How can I place a string that is padded with spaces into a

Category        :Database/VCL

Platform        :All

Product         :All 32 bit

Question:

How can I place a string that is padded with spaces into a database?

Answer:

The BDE will trim trailing spaces from a string field. It is possible to choose another character to act in the place of trailing spaces, then replace the characters at runtime with spaces.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16838: Timed beeps from the PC speaker

Question and Answer Database

FAQ1838D.txt Timed beeps from the PC speaker

Category :Windows API

Platform :All

Product :All 32 bit

### Question:

How can I beep the PC Speaker several times in a row, and have a small delay between beeps that is reliable across different machines with different clock speeds?

### Answer:

The following demonstrates using a delay function that can be used to beep the PC speaker.

### Example:

```
procedure Delay(ms : longint);
{$IFDEF WIN32}
var
  TheTime : LongInt;
{$ENDIF}
begin
{$IFDEF WIN32}
  Sleep(ms);
{$ELSE}
  TheTime := GetTickCount + ms;
  while GetTickCount < TheTime do
    Application.ProcessMessages;
{$ENDIF}
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageBeep(word(-1));
  Delay(200);
  MessageBeep(word(-1));
  Delay(200);
end;
```

```
MessageBeep(word(-1));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16841: TQuery, error 'Creating Cursor handle'

Question and Answer Database

FAQ1841C.txt TQuery, error 'Creating Cursor handle'

Category :Database Issues

Platform :All

Product :C++Builder 1.x

### Question:

I am getting the error creating curser handle, even though the tquery has a select statement. Same code else where is fine.

### Answer:

Do not Fill a Query->SQL in the OnOpen Event.

5/13/98 10:09:30 AM

Last Modified: 01-SEP-99

## Article #16842: How can I place a bitmap in a Metafile?

Question and Answer Database

FAQ1842D.txt    How can I place a bitmap in a Metafile?

Category        :VCL

Platform        :All

Product        :All 32 bit

Question:

How can I place a bitmap in a Metafile?

Answer:

The following example demonstrates drawing a bitmap into a metafile.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  m : TmetaFile;
  mc : TmetaFileCanvas;
  b : tbitmap;
begin
  m := TMetaFile.Create;
  b := TBitmap.create;
  b.LoadFromFile('C:\SomePath\SomeBitmap.BMP');
  m.Height := b.Height;
  m.Width := b.Width;
  mc := TMetafileCanvas.Create(m, 0);
  mc.Draw(0, 0, b);
  mc.Free;
  b.Free;
  m.SaveToFile('C:\SomePath\Test.emf');
  m.Free;
  Image1.Picture.LoadFromFile('C:\SomePath\Test.emf');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16845: What are IDAPI and SQL links

Question and Answer Database

FAQ1845C.txt    What are IDAPI and SQL links

Category        :Database Issues

Platform        :All

Product         :C++Builder 1.x

Question:

What is the definition of 'IDAPI'? What is 'SQL Links'?

Answer:

IDAPI is the Integrated Database Application Program Interface. BDE is a way to access multiple data sources with a consistent API. The distinction is in the terminology: IDAPI is the API for the BDE. It includes all the functions necessary to access, manipulate, etc. the data; BCB, dBASE for Windows, Quattro Pro, and Paradox for Windows use these functions to access data: You can use them yourself in your programs. If you look at the BCB source you will see these functions used. They are prefaced with "Dbi" (e.g. DbiCreateTable).

SQL Links is a collection of native drivers that enable you to connect to remote database servers.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16846: Images conflicts between different versions of IE

Question and Answer Database

FAQ1846D.txt Images conflicts between different versions of IE  
Category :Miscellaneous  
Platform :All  
Product :All 32 bit

### Question:

If I create an application on a machine with Internet Explorer 4 and deploy on machines without Internet Explorer 4, they see no images in my Toolbar and/or Coolbar. How can I get my images to be visible?

### Answer:

The Imagelist streaming implementation and other features of the Coolbar and Toolbar have changed between Internet Explorer 3.02 and Internet Explorer 4. Applications using Win32 common controls developed under Internet Explorer 4.0 may not work correctly under Internet Explorer 3.02 (and vice-versa).

The way to work around the imagelist streaming problem is to not stream the images to the .dfm. Instead, place the bitmaps in a resource file, then load them into the imagelist at runtime using the following code:

```
procedure LoadResourcedBMP(ID: string; Images: TImageList);
var
  B: TBitmap;
begin
  B := TBitmap.Create;
  try
    B.LoadFromResourceName(hInstance, ID);
    Images.AddMasked(B, B.TransparentColor);
  finally
    B.Free;
  end;
end;
```



7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16849: Informix Support

Question and Answer Database

FAQ1849C.txt Informix Support

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

What versions of Informix (Online, I-NET) do the SQL Links support?

Answer:

SQL Links will work fine with all versions of the Informix sever, including 5.01.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16850: Debugging into the VCL source code

Question and Answer Database

FAQ1850D.txt     Debugging into the VCL source code

Category        :Database/VCL

Platform        :All

Product         :All 32 bit

Question:

Is it possible to step through the VCL code with using integrated debugging?

Answer:

Yes! If you have the Developer or Client Server versions of Delphi or C++ Builder, simply add the path to the VCL directory at the beginning of your Library search path. To get there from the IDE's MainMenu, select: Tools > Environment Options, and then select the Library page from the dialog, and look for the Library Path option.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16854: Big-endian and little-endian formatted integers

Question and Answer Database

FAQ1854D.txt Big-endian and little-endian formatted integers

Category :Compiler

Platform :All

Product :All 32 bit

Question:

How can I create big-endian and little-endian formatted integers?

Answer:

The big-endian and little-endian format simply swaps the HiByte and LoBytes and HiWords and LoWords of a given integer or word value. The following demonstrates swapping the values of a given word or longint. Calling the Endian functions a second time reverts the values back to their original format.

```
function EndianWord(w : word) : word;
begin
    result := swap(w);
end;

function EndianInt(i : integer) : integer;
begin
    result := swap(i);
end;

function EndianLong(L : longint) : longint;
begin
    result := swap(L shr 16) or
              (longint(swap(L and $ffff)) shl 16);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    w : word;
    i : integer;
    l : LongInt;
begin
    w := 10;
```

```
i := -10;
l := 1024;
w := EndianWord(w);
i := EndianInt(i);
l := EndianLong(l);
{again!}
w := EndianWord(w);
i := EndianInt(i);
l := EndianLong(l);
Mem1.Lines.Add(IntToStr(w));
Mem1.Lines.Add(IntToStr(i));
Mem1.Lines.Add(IntToStr(l));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16858: Displaying documents in the default browser

Question and Answer Database

FAQ1858D.txt - Displaying documents in the default browser

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

How can I start the default browser with an URL in a new browser window?

Answer:

The following example attempts to start the specified URL in a new browser window. Note: Not all versions of Internet Explorer will respond with a new window.

Example:

```
uses
  DdeMan,
{$IFDEF WIN32}
  Registry; {We will get it from the registry}
{$ELSE}
  IniFiles; {We will get it from the win.ini file}
{$ENDIF}

{$IFDEF WIN32}
  const MAX_PATH = 144;
{$ENDIF}

function GetProgramAssociation (Ext : string) : string;
var
{$IFDEF WIN32}
  reg: TRegistry;
  s : string;
{$ELSE}
  WinIni : TIniFile;
  WinIniFileName : array[0..MAX_PATH] of char;
```

```

s : string;
{$ENDIF}
begin
{$IFDEF WIN32}
s := '';
reg := TRegistry.Create;
reg.RootKey := HKEY_CLASSES_ROOT;
if reg.OpenKey('.') + ext + '\shell\open\command',
    false) <> false then begin
{The open command has been found}
s := reg.ReadString('');
reg.CloseKey;
end else begin
{perhaps thier is a system file pointer}
if reg.OpenKey('.') + ext,
    false) <> false then begin
s := reg.ReadString('');
reg.CloseKey;
if s <> '' then begin
{A system file pointer was found}
if reg.OpenKey(s + '\shell\open\command',
    false) <> false then
{The open command has been found}
s := reg.ReadString('');
reg.CloseKey;
end;
end;
end;
end;
{Delete any command line, quotes and spaces}
if Pos('%', s) > 0 then
Delete(s, Pos('%', s), length(s));
if ((length(s) > 0) and
(s[1] = '"')) then
Delete(s, 1, 1);
if ((length(s) > 0) and
(pos('"', s) > 0)) then
Delete(s, pos('"', s), Length(s));
while ((length(s) > 0) and
(s[length(s)] = #32)) do
Delete(s, Length(s), 1);
{$ELSE}
GetWindowsDirectory(WinIniFileName, sizeof(WinIniFileName));
StrCat(WinIniFileName, '\win.ini');
WinIni := TIniFile.Create(WinIniFileName);
s := WinIni.ReadString('Extensions',

```

```

                                ext,
                                '');
WinIni.Free;
{Delete any command line}
if Pos(' ^', s) > 0 then
    Delete(s, Pos(' ^', s), length(s));
{$ENDIF}
result := s;
end;

procedure StartNewBrowserWindow(URL : string);
var
    DDEConv : TDDEClientConv;
    URLFired : bool;
    App : string;
    UpApp : string;
    p : array[0..MAX_PATH] of char;
begin
    UrlFired := false;
    App := GetProgramAssociation('HTM');
    UpApp := Uppercase(App);
    Delete(App, Pos('.EXE', UpApp), length(App));
    if Pos('NETSCAPE.EXE',
        UpApp) > 0 then begin
        DDEConv:=TDDEClientConv.Create(nil);
        DDEConv.ServiceApplication := App;
        if DDEConv.SetLink('NETSCAPE' , 'WWW_OpenURL') then
            if DDEConv.RequestData(URL +
                ',,0x0,0x0') <> nil then
                if DDEConv.SetLink('NETSCAPE', 'WWW_Activate') then
                    URLFired := DDEConv.RequestData('0xFFFFFFFF,0x0') <> nil;
        DDEConv.Free;
    end else
    if Pos('IEXPLORE.EXE',
        UpApp) > 0 then begin
        DDEConv:=TDDEClientConv.Create(nil);
        DDEConv.ServiceApplication := App;
        if DDEConv.SetLink('iexplore', 'WWW_OpenURL') then
            if DDEConv.RequestData(URL + ',,0') <> nil then
                if DDEConv.SetLink('iexplore', 'WWW_Activate') then
                    URLFired := DDEConv.RequestData('0,0') <> nil;
        DDEConv.Free;
    end;
    if UrlFired = false then
        WinExec(StrPCopy(@p, URL), SW_SHOWNORMAL);

```



```
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    StartNewBrowserWindow('http://www.borland.com');  
    StartNewBrowserWindow('http://www.yahoo.com');  
end;
```

4/2/99 11:46:29 AM

Last Modified: 01-SEP-99

## Article #16861: Mask edit for TDBEdit controls

Question and Answer Database

FAQ1861C.txt Mask edit for TDBEdit controls

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

How to I create a mask for a TDBEdit control?

Answer:

Edit masks are applied to the fields in the table (TField components) and not in the data controls themselves.

Double-click on the TTable icon and add all the fields you want from your table. When a field is highlighted, its properties show up in the object inspector, including an edit mask. Linking the TDBEdit and any of the other data controls to this dataset will follow the edit mask rules for the fields set this way.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16862: Disabling Full Window Drag

Question and Answer Database

FAQ1862D.txt    Disabling Full Window Drag

Category       :Windows API

Platform        :All

Product         :All 32 bit

### Question:

The redraw time for my canvas takes too long if the user drags the form outside the desktop window. How can I determine if the user has the "Full Window Drag" feature enabled in Windows, so I can take steps to temporarily disable my redraws?

### Answer:

The Following example demonstrates checking if full windows drag is enabled.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  b : bool;
begin
  SystemParametersInfo(SPI_GETDRAGFULLWINDOWS, 0, @b, 0);
  if not b then
    ShowMessage('Full Window Drag is not enabled') else
    ShowMessage('Full Window Drag is enabled');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16865: Packing dBase tables

Question and Answer Database

FAQ1865C.txt     Packing dBase tables

Category     :Database Issues

Platform     :All

Product      :C++Builder 1.x

Question:

Is there a BDE API or a DLL available to pack a dBASE table?

Answer:

The BDE includes a function to pack dBASE tables, called `DbiPackTable()`.

Include the `BDE.HPP` header file. Then call the BDE function as follows:

```
DBIPackTable(Table1->DbHandle, Table1->Handle,  
             "TABLENAME.DBF", szDBASE, TRUE);
```

Note: The table must be opened in exclusive mode.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16866: MaxLength property of a DBEdit is not automatically set

Question and Answer Database

FAQ1866D.txt    MaxLength property of a DBEdit is not automatically set  
Category        :Database/VCL  
Platform        :All  
Product         :Delphi 3.x

### Question:

In Delphi 1.0 and 2.0, the TDBEdit control would automatically set the max field length (MaxLegth) and wouldn't allow you to enter more characters than the size specified for the TField Object. Why has Delphi 3 removed this "feature"?

### Answer:

This new behavior allows you to set the value different than the field size. Delphi 1 and Delphi 2 overrided a change and put back the field size length. In Delphi 3 though, the maximum length is set, by default, to zero and will allow you to specify and enter any MaxLength for a TDBEdit, but it only writes as many characters to the database as the actual TField size property allows.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16869: How can I assign a Null value to a numeric field in a table?

Question and Answer Database

FAQ1869C.txt    How can I assign a Null value to a numeric field in a table?

Category        :Database Issues

Platform        :All

Product         :C++Builder 1.x

Question:

How can I assign a Null value to a numeric field in a table?

Answer:

Use the Clear method of any TField object.

```
Table1Amount->Clear();
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16870: How can I get the default Windows folders

Question and Answer Database

FAQ1870D.txt How can I get the default Windows folders

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I get the default Windows folders?

Answer:

The following example demonstrates retrieving the entire list of default shell folders to a TMemo component. The folders retrieved should include: Favorites, Desktop, Programs, Fonts, SendTo, Start Menu, Templates, Startup, Recent and NetHood.

Example:

```
uses Registry;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    reg : TRegistry;
```

```
    ts : TStringList;
```

```
    i : integer;
```

```
begin
```

```
    reg := TRegistry.Create;
```

```
    reg.RootKey := HKEY_CURRENT_USER;
```

```
    reg.LazyWrite := false;
```

```
    reg.OpenKey(
```

```
        'Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders',  
        false);
```

```
    ts := TStringList.Create;
```

```
    reg.GetValueNames(ts);
```

```
    for i := 0 to ts.Count - 1 do begin
```

```
        Mem1.Lines.Add(ts.Strings[i] +
```

```
            ' = ' +
```

```
            reg.ReadString(ts.Strings[i]));
```

```
    end;
```

```
    ts.Free;
```

```
    reg.CloseKey;  
    reg.free;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16873: Getting current record from a TDBGrid

Question and Answer Database

FAQ1873C.txt Getting current record from a TDBGrid

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

How do you tell which record and which field of a TDBGrid is current?

Answer:

Here is a method to keep track of the current column and row. The following code in the method `MyDBGridDrawDataCell` updates the variables `Col` and `Row` (which must not be local to the method) every time the grid is redrawn. Using this code you can assume that `Col` and `Row` point to the current column and row respectively.

```
int Col, Row;

void __fastcall TForm1::DBGrid1DrawDataCell(TObject *Sender,
      const TRect &Rect, TField *Field, TGridDrawState State)
{
    int RowHeight;
    if (State.Contains(gdFocused)
        {
            RowHeight = Rect.Bottom - Rect.Top;
            Row = (Rect.Top / RowHeight) - 1;
            Col = Field->Index;
        }
    }
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16874: Changing the font in a hint window

Question and Answer Database

FAQ1874D.txt Changing the font in a hint window

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I change the font of a hint window?

Answer:

The following example traps the `Application.OnShowHint` event and changes the font.

Example:

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    procedure MyShowHint(var HintStr: string;
                        var CanShow: Boolean;
                        var HintInfo: THintInfo);
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.MyShowHint(var HintStr: string;
                            var CanShow: Boolean;
                            var HintInfo: THintInfo);

var
  i : integer;
```

```
begin
  for i := 0 to Application.ComponentCount - 1 do
    if Application.Components[i] is THintWindow then
      with THintWindow(Application.Components[i]).Canvas do begin
        Font.Name      := 'Arial';
        Font.Size      := 18;
        Font.Style     := [fsBold];
        HintInfo.HintColor := clWhite;
      end;
    end;
  end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnShowHint := MyShowHint;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16878: Converting a TColor value to a HTML Color string

Question and Answer Database

FAQ1878D.txt - Converting a TColor value to a HTML Color string

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

How can I convert a TColor value to a HTML Color string?

Answer:

The following example demonstrates converting a TColor to a RGB color, and formatting the RGB color to be shown as hexadecimal digits.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TheRgbValue : TColorRef;
begin
  if ColorDialog1.Execute then begin
    TheRgbValue := ColorToRGB(ColorDialog1.Color);
    ShowMessage(Format( '%.2x%.2x%.2x' ,
                        [GetRValue(TheRGBValue),
                         GetGValue(TheRGBValue),
                         GetBValue(TheRGBValue)]));
  end;
end;
```

4/2/99 11:46:48 AM

Last Modified: 01-SEP-99

## Article #16881: What is the equivalent in BCB to a Pdox TCursor

Question and Answer Database

FAQ1881C.txt    What is the equivalent in BCB to a Pdox TCursor

Category        :Database Issues

Platform        :All

Product         :C++Builder 1.x

Question:

What is the equivalent in BCB of Paradox's TCursor and what is a TCursor in Paradox?

Answer:

The TTable component.

A TCursor in Paradox is a pointer to the data in a table, a pointer that enables you to manipulate data at the table level, record level, and field level without having to display the table.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16882: Using JPEGs from resource files

Question and Answer Database

FAQ1882D.txt Using JPEGs from resource files

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I load a JPEG image from a resource file that is linked with my application?

Answer:

The following demonstrates creating a resource file containing a JPEG image, and loading the JPEG file from the resource file. The resulting JPEG image is displayed in a Image component.

1) Create a text file with the extension of ".rc". The text file should be named something different than the project name or any unit name in your application to avoid any confusion for the compiler. The text file should contain the following line:

```
MYJPEG JPEG C:\DownLoad\MY.JPG
```

Where:

"MYJPEG" is the name you wish to name the resource

"JPEG" is the user defined resource type.

"C:\DownLoad\MY.JPG" is the path and filename of the JPEG file.

For our example we will name the file "foo.rc" .

Now run the BRCC32.exe (Borland Resource CommandLine Compiler) program found in the Delphi/C++ Builders bin directory giving the full path to the rc file:

```
C:\DelphiPath\BIN\BRCC32.EXE C:\ProjectPath\F00.RC
```

You should now have a compiled resource file named the same as the ".rc" file you compiled with the extension of ".res".

The following demonstrates using the embedded JPEG in your application

```
{Link the res file}  
{$R FOO.RES}
```

```
uses Jpeg;
```

```
procedure LoadJPEGFromRes(TheJPEG : string;  
                          ThePicture : TPicture);
```

```
var
```

```
  ResHandle : THandle;  
  MemHandle : THandle;  
  MemStream : TMemoryStream;  
  ResPtr     : PByte;  
  ResSize    : Longint;  
  JPEGImage : TJPEGImage;
```

```
begin  
  ResHandle := FindResource(hInstance, PChar(TheJPEG), 'JPEG');  
  MemHandle := LoadResource(hInstance, ResHandle);  
  ResPtr     := LockResource(MemHandle);  
  MemStream := TMemoryStream.Create;  
  JPEGImage := TJPEGImage.Create;  
  ResSize := SizeOfResource(hInstance, ResHandle);  
  MemStream.SetSize(ResSize);  
  MemStream.Write(ResPtr^, ResSize);  
  FreeResource(MemHandle);  
  MemStream.Seek(0, 0);  
  JPEGImage.LoadFromStream(MemStream);  
  ThePicture.Assign(JPEGImage);  
  JPEGImage.Free;  
  MemStream.Free;
```

```
end;  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  LoadJPEGFromRes('MYJPEG', Image1.Picture);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16885: error Index out of range with TTable FindNearest or FindKey

Question and Answer Database

FAQ1885C.txt error Index out of range with TTable FindNearest or FindKey

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

Why do I get 'Index out of range' when I use TTable::FindNearest and TTable::FindKey on a dBASE table with an expression index?

Answer:

FindKey and FindNearest are not meant to work with dBASE expression indexes. Instead, use TTable::GoToKey and TTable::GotoNearest, which will work fine with dBASE expression indexes.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99



## Article #16886: System Modal Windows on WinNT

Question and Answer Database

FAQ1886D.txt      System Modal Windows on WinNT

Category        :Windows API

Platform        :All

Product         :All 32 bit

### Question:

When I try to create a system modal window, it seems to work on Windows 3.1 and Windows 95, but not on Windows NT. How can I get a system modal window to Work under Window NT.

### Answer:

Windows NT does not allow system modal windows to be modal on a system wide basis.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16889: error initializing Database engine

Question and Answer Database

FAQ1889C.txt error initializing Database engine

Category :Database Issues

Platform :All

Product :C++Builder 1.x

### Question:

When I try to run a database application from BCB, I get an exception EDatabaseError with message 'An error occurred while attempting to initialize the Borland Database Engine (Error \$2C09)'.

### Answer:

Add SHARE.EXE to your AUTOEXEC.BAT file or add DEVICE=VSHARE.386 to the [386Enh] section of your SYSTEM.INI file and reboot.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16890: Playing MPEG files in Delphi apps

Question and Answer Database

FAQ1890D.txt    Playing MPEG files in Delphi apps

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I play an MPEG file in Delphi apps?

Answer:

Providing the Windows MMSYSTEM layer has a MPEG decoder installed, you can easily use the MultiMedia play a MPEG file by setting the filename to the path of the MPEG file.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MediaPlayer1.FileName := 'C:\Download\rsgrow.mpg';
  MediaPlayer1.Open;
  MediaPlayer1.Display := Panel1;
  MediaPlayer1.DisplayRect := Panel1.ClientRect;
  MediaPlayer1.Play;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16894: Running apps on Windows startup

Question and Answer Database

FAQ1894D.txt    Running apps on Windows startup

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I get my application to run every time Windows starts?

Answer:

The following demonstrates how to add your application to the Windows boot sequence. Code is demonstrated for both the Win32 and Win16 platforms.

uses

```
  Registry, {For Win32}  
  IniFiles; {For Win16}
```

```
{$IFDEF WIN32}
```

```
  const MAX_PATH = 144;
```

```
{$ENDIF}
```

```
{For Win32}
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  reg: TRegistry;
```

```
begin
```

```
  reg := TRegistry.Create;
```

```
  reg.RootKey := HKEY_LOCAL_MACHINE;
```

```
  reg.LazyWrite := false;
```

```
  reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\Run',  
              false);
```

```
  reg.WriteString('My App', Application.ExeName);
```

```
  reg.CloseKey;
```

```
  reg.free;
```

```
end;
```

```
{For Win16}
```

```
procedure TForm1.Button2Click(Sender: TObject);
var
  WinIni : TIniFile;
  WinIniFileName : array[0..MAX_PATH] of char;
  s : string;
begin
  GetWindowsDirectory(WinIniFileName, sizeof(WinIniFileName));
  StrCat(WinIniFileName, '\win.ini');
  WinIni := TIniFile.Create(WinIniFileName);
  s := WinIni.ReadString('windows',
                        'run',
                        '');
  if s = '' then
    s := Application.ExeName else
    s := s + ';' + Application.ExeName;
  WinIni.WriteString('windows',
                    'run',
                    s);
  WinIni.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16898: Alloting CPU time

Question and Answer Database

FAQ1898D.txt Alloting CPU time

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I change the amount of CPU time that is allotted to my application?

Answer:

The following example demonstrates changing the CPU priority given an application. Care should be taken when changing the applications priority, as values that are too high may cause the system to become unresponsive. For more information, please see the Win32 help file for the `SetThreadPriority()` function.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ProcessID : DWORD;
  ProcessHandle : THandle;
  ThreadHandle : THandle;
begin
  ProcessID := GetCurrentProcessID;
  ProcessHandle := OpenProcess(PROCESS_SET_INFORMATION,
                              false,
                              ProcessID);
  SetPriorityClass(ProcessHandle, REALTIME_PRIORITY_CLASS);
  ThreadHandle := GetCurrentThread;
  SetThreadPriority(ThreadHandle, THREAD_PRIORITY_TIME_CRITICAL);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16902: Yielding cycles to other applications

Question and Answer Database

FAQ1902D.txt Yielding cycles to other applications

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I yield to the system during a lengthy operation without allowing message processing to continue in my application?

Answer:

Call the Windows API Sleep() function passing zero as a parameter.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16905: creating indexes with the TTable create function

Question and Answer Database

FAQ1905C.txt    creating indexes with the TTable create function  
Category       :Database Issues  
Platform        :All  
Product         :C++Builder 1.x

Question:

Why is it that when I create a table using the TTable component's CreateTable method it creates the fields correctly but does not create the indexes even though I do a

```
NewTable->IndexDefs->Assign(Table1->IndexDefs)?
```

Answer:

This is the correct way to transfer the index definition to NewTable, however, the IndexDefs property of Table1 may not be up-to-date so you need to call the Update method of Table1's IndexDefs property prior to its assignment to NewTable as shown in this example:

```
NewTable->Active = false;  
NewTable->DatabaseName = "DBDEMOS";  
NewTable->TableName = "Temp";  
NewTable->TableType = ttParadox;  
NewTable->FieldDefs->Assign(Table1->FieldDefs);  
NewTable->Table1->IndexDefs->Update();     // Do an update first  
NewTable->IndexDefs->Assign(Table1->IndexDefs);  
NewTable->CreateTable();
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16906: Reducing the resource requirements of TNotebook and TTabbedNotebook

Question and Answer Database

FAQ1906D.txt Reducing the resource requirements of TNotebook and TTabbedNotebook  
Category :VCL  
Platform :All  
Product :All 32 bit

Question:

Is there any way to reduce the resource requirements of the TNotebook and TTabbedNotebook components?

Answer:

Yes. It is possible to reduce the requirements of the pages in the TNotebook and TTabbedNotebook components by destroying the handles of the components that are not on the current page. The following example demonstrates calling the protected DestroyHandle method of the page, by subclassing the notebook to get access to the protected method.

```
type TMyTabbedNotebook = class(TTabbedNotebook);
type TMyNotebook = class(TNotebook);

procedure TForm1.TabbedNotebook1Change(Sender: TObject;
                                       NewTab: Integer;
                                       var AllowChange: Boolean);
begin
    with TabbedNotebook1 do
        TMyTabbedNotebook(
            TWinControl(Pages.Objects[PageIndex])).DestroyHandle;
end;

procedure TForm1.TabSet1Change(Sender: TObject; NewTab: Integer;
                               var AllowChange: Boolean);
begin
    with Notebook1 do
        TMyNotebook(TWinControl(Pages.Objects[PageIndex])).DestroyHandle;
    Notebook1.PageIndex := NewTab;
end;
```

```
    AllowChange := true  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16909: Accessing MS Access data with BCB

Question and Answer Database

FAQ1909C.txt     Accessing MS Access data with BCB

Category     :Database Issues

Platform     :All

Product      :C++Builder 1.x

### Question:

I have had no success getting at Microsoft Access data using BCB other than a simple TTable view. Using TQuery I can get a read-only view to work, but I cannot a read/write view to work. After the login screen I am presented with an exception message like 'Passthrough SQL connection must be shared'.

### Answer:

Use the Database Engine Configuration to change the 'SQLPASSTHRU MODE' option in the alias associated with your Access database from its default blank value to 'SHARED AUTOCOMMIT' (without the quotes).

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16910: Changing the root key

Question and Answer Database

FAQ1910D.txt Changing the root key

Category :VCL

Platform :All

Product :All 32 bit

Question:

According to the documentation for the TRegIniFile component, I should be able to change the root key. I have tried, but the key does not change. How can I change the root key?

Answer:

The following demonstrates changing the root key for the TRegIniFile component.

Example:

```
uses Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
  WinIni : TRegIniFile;
begin
  WinIni := TRegIniFile.Create('');
  WinIni.RootKey := HKEY_LOCAL_MACHINE;
  WinIni.WriteString('Frank',
                    'Borland',
                    'Writes Fast Code!');

  WinIni.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16913: Is there a programmatic way to add an alias to the IDAPI.CFG

Question and Answer Database

FAQ1913C.txt Is there a programmatic way to add an alias to the IDAPI.CFG

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

Is there a programmatic way to add an alias to the IDAPI.CFG file?

Answer:

The BDE includes a function called `DbiAddAlias()`. See the BDE API help file for more information on this.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16914: Show a AVI clip in full screen zoom mode

Question and Answer Database

FAQ1914D.txt     Show a AVI clip in full screen zoom mode  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

Question:

How can I show a AVI clip in full screen zoom mode?

Answer:

The following example demonstrates playing a AVI file in full screen mode by displaying the AVI on a second form that has the form border style set to bsNone and has no border icons. Note that the second form has a panel with it's alignment set to alClient, and a MediaPlayer component that has a Notify event. We will show the second form maximized, and when the AVI file has played, the Notify event will close the form.

Example:

```
{Code for Form 1}
```

```
uses Unit2;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form2.Show;  
    Form2.WindowState := wsMaximized;  
    Form2.MediaPlayer1.Notify := false;  
    Form2.MediaPlayer1.Display := Form2.Pane11;  
    Form2.MediaPlayer1.FileName := 'C:\TheWall\DELCAR2.AVI';  
    Form2.MediaPlayer1.Open;  
    Form2.MediaPlayer1.DisplayRect := Form2.ClientRect;  
    Form2.MediaPlayer1.Play;  
end;
```

```
{Code for Form 2}
```

```
procedure TForm2.MediaPlayer1Notify(Sender: TObject);  
begin  
    if MediaPlayer1.NotifyValue = nvSuccessful then  
        Form2.Close;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16917: Max Fields for dBase & Paradox Tables

Question and Answer Database

FAQ1917C.txt Max Fields for dBase & Paradox Tables

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

What is the maximum number of fields dBASE and Paradox tables can have (per table)?

Answer:

dBASE tables can have a maximum of 254 fields, Paradox tables can have 255.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16918: Play an AVI file that has sound, silently

Question and Answer Database

FAQ1918D.txt    Play an AVI file that has sound, silently  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

Question:

How can I play an AVI file that has sound silently without adjusting the volume?

Answer:

The following example demonstrates playing an AVI file silently by turning off the MCI audio channels in code.

Example:

```
uses MMSystem;

procedure SetMediaAudioOff(DeviceID : word);
var
  SetParm : TMCISetParams;
begin
  SetParm.dwAudio := MCI_SET_AUDIO_ALL;
  mciSendCommand(DeviceID,
                 MCI_SET,
                 MCI_SET_AUDIO or MCI_SET_OFF,
                 Longint(@SetParm));
end;

Procedure SetMediaAudioOn(DeviceID : word);
var
  SetParm : TMCISetParams;
begin
  SetParm.dwAudio := MCI_SET_AUDIO_ALL;
  mciSendCommand(DeviceID,
                 MCI_SET,
                 MCI_SET_AUDIO or MCI_SET_ON,
                 Longint(@SetParm));
```

```
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  {Play AVI file silently}  
  MediaPlayer1.FileName := 'C:\TheWall\DELCAR2.AVI';  
  MediaPlayer1.Display := Panel1;  
  MediaPlayer1.Open;  
  MediaPlayer1.Play;  
  SetMediaAudioOff(MediaPlayer1.DeviceId);  
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
  {Play AVI with sound}  
  MediaPlayer1.FileName := 'C:\TheWall\DELCAR2.AVI';  
  MediaPlayer1.Display := Panel1;  
  MediaPlayer1.Open;  
  MediaPlayer1.Play;  
  SetMediaAudioOn(MediaPlayer1.DeviceId);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16922: Error warnings too long

Question and Answer Database

FAQ1922D.txt    Error warnings too long

Category       :IDE

Platform        :All

Product         :Delphi 3.x

Question:

Some errors are too long to view in the Delphi 3 Warning window. How can I get the full text of the warning message?

Answer:

Hold the cursor over the error, and a hint window will display the full text of the given error.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16925: TTable AddIndex() with ixUnique and Paradox tables

Question and Answer Database

FAQ1925C.txt TTable AddIndex() with ixUnique and Paradox tables

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

Why can't I use the ixUnique option when creating indexes for Paradox tables with the AddIndex method of the TTable component?

Answer:

The index options used in the AddIndex method of the TTable component are table specific. For example, the ixUnique option works with dBASE tables but not Paradox. The following table shows how these options apply to dBASE and Paradox tables.

Index Options	dBASE	Paradox
ixUnique	*	
ixDescending	*	*
ixNonMaintained	*	*
ixPrimary		*
ixCaseInsensitive		*

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16926: Flush binary file from disk cache to disk

Question and Answer Database

FAQ1926D.txt Flush binary file from disk cache to disk

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I guarantee that information I have written to a binary file is flushed from the disk cache and actually written to the disk?

Answer:

The following example demonstrates flushing the disk cache of given binary file, so that data is guaranteed to be written to the disk.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f : file;
  i : integer;
begin
  i := 10;
  AssignFile(f, 'C:\Download\Test.Bin');
  Rewrite(f, 1);
  BlockWrite(f, i, sizeof(i));
  FlushFileBuffers(TFileRec(f).Handle);
  CloseFile(f);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16929: How do I do a locate on a non-indexed field?

Question and Answer Database

FAQ1929C.txt    How do I do a locate on a non-indexed field?

Category     :Database Issues

Platform     :All

Product      :C++Builder 1.x

Question:

How do I do a locate on a non-indexed field?

Answer:

The following function can be added to your to your unit and called as follows:

```
bool __fastcall Locate(const TTable *oTable, const TField *oField,
                      const Char* SvALUE);
```

Table1 is your table component, Table1LName is TField you've added with the fields editor (double click on the table component) and 'Beman' is the name you want to find.

```
// Locate will find SValue in a non-indexed table
bool __fastcall Locate(const TTable *oTable, const TField *oField,
                      const Char* SvALUE)
{
    TBookmark bmPos;
    bool Locate, bFound;

    Locate = false;
    bFound = false;

    if (!oTable->Active)
        return;

    if (oTable->FieldDefs->IndexOf(oField->FieldName) < 0)
        return;

    bmPos = oTable->GetBookmark;

    oTable->DisableControls;
```

```
oTable->First;

while(!oTable->EOF)
{
    if(oField->AsString == sValue)
    {
        Locate = true;
        bFound = true;
        break;
    }
    else
        oTable->Next;
}

if (!bFound)
    oTable->GotoBookMark(bmPos);

oTable->FreeBookMark(bmPos);
oTable->EnableControls;
}
```

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99



## Article #16930: Determine the last access time of a given file

Question and Answer Database

FAQ: FAQ1930D - Determine the last access time of a given file

Category: Windows API

Platform: All Windows

Product:

Question:

How can I determine the last access time of a given file?

Answer:

The following example demonstrates retrieving the last date and time a given file was accessed. Note: Not all file systems support the last access time.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SearchRec : TSearchRec;
  Success : integer;
  DT : TFileTime;
  ST : TSystemTime;
begin
  Success := SysUtils.FindFirst('C:\autoexec.bat',
                                faAnyFile,
                                SearchRec);

  if (Success = 0) and
     (( SearchRec.FindData.ftLastAccessTime.dwLowDateTime <> 0)
     or ( SearchRec.FindData.ftLastAccessTime.dwHighDateTime <> 0))
  then
    begin
      FileTimeToLocalFileTime(SearchRec.FindData.ftLastAccessTime,DT);
      FileTimeToSystemTime(DT,ST);
      Mem1.Lines.Clear;
      Mem1.Lines.Add('AutoExec.Bat was last accessed at:');
```

```
Mem1.Lines.Add('Year := ' + IntToStr(st.wYear));
Mem1.Lines.Add('Month := ' + IntToStr(st.wMonth));
Mem1.Lines.Add('DayOfWeek := ' + IntToStr(st.wDayOfWeek));
Mem1.Lines.Add('Day := ' + IntToStr(st.wDay));
Mem1.Lines.Add('Hour := ' + IntToStr(st.wHour));
Mem1.Lines.Add('Minute := ' + IntToStr(st.wMinute));
Mem1.Lines.Add('Second := ' + IntToStr(st.wSecond));
Mem1.Lines.Add('Milliseconds := ' + IntToStr(st.wMilliseconds));
end;
SysUtils.FindClose(SearchRec);
end;
```

6/28/99 10:23:30 AM

Last Modified: 01-SEP-99

## Article #16933: using return instead of tab or mouse to move off a control

Question and Answer Database

FAQ1933C.txt using return instead of tab or mouse to move off a control

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

Is there a way to use the return key for data entry, instead of tab or the mouse?

Answer:

Use this code for an Edit's OnKeyPress event.

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char *Key)
{
    if (Key == "#13")
    {
        SelectNext(dynamic_cast (Sender), true, true );
        Key = "#0";
    }
}
```

This causes Enter to behave like tab. Now, select all controls on the form you'd like to exhibit this behavior (not Buttons) and go to the Object Inspector and set their OnKeyPress handler to EditKeyPress. Now, each control you selected will process Enter as Tab. If you'd like to handle this at the form (as opposed to control) level, reset all the controls OnKeyPress properties to blank, and set the \_form\_'s OnKeyPress property to EditKeyPress. Then, change Sender to ActiveControl and set the form's KeyPreview property to true:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char *Key)
{
    if (Key == "#13")
    {
        SelectNext(dynamic_cast (ActiveControl), true, true );
        Key = "#0";
    }
}
```

```
}  
}
```

This will cause each control on the form (that can) to process  
Enter as Tab.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16934: Formatting and displaying integers with commas

Question and Answer Database

FAQ1934D.txt    Formatting and displaying integers with commas  
Category        :Object Pascal  
Platform        :All  
Product         :All 32 bit

Question:

How can I easily display an integer value with commas every three digits?

Answer:

The following example demonstrates displaying an integer value with thousand separators. Note that the actual character used to separate the digits will depend on the local (country) system settings.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i : integer;
begin
  i := 12345678;
  Mem1.Lines.Add(FormatFloat('#, ', i));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16938: Create hints for Menu items

Question and Answer Database

FAQ1938D.txt Create hints for Menu items

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I create hints for Menu items?

Answer:

The following example demonstrates adding a hint handler to your application for handling menu selections. Hints will be displayed in the status panel's caption.

Example:

```
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    MainMenu: TMainMenu;
    MenuItemFile: TMenuItem;
    MenuItemOpen: TMenuItem;
    MenuItemClose: TMenuItem;
    OpenDialog1: TOpenDialog;
    procedure FormCreate(Sender: TObject);
    procedure MenuItemCloseClick(Sender: TObject);
    procedure MenuItemOpenClick(Sender: TObject);
  private
    { Private declarations }
    procedure HintHandler(Sender: TObject);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Panell.Align := alBottom;  
    MenuItemFile.Hint := 'File Menu';  
    MenuItemOpen.Hint := 'Opens A File';  
    MenuItemClose.Hint := 'Closes the Application';  
    Application.OnHint := HintHandler;  
end;
```

```
procedure TForm1.HintHandler(Sender: TObject);  
begin  
    Panell.Caption := Application.Hint;  
end;
```

```
procedure TForm1.MenuItemCloseClick(Sender: TObject);  
begin  
    Application.Terminate;  
end;
```

```
procedure TForm1.MenuItemOpenClick(Sender: TObject);  
begin  
    if OpenFileDialog1.Execute then  
        Form1.Caption := OpenFileDialog1.FileName;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16942: Native Access driver and ISAPI/NSAPI

Question and Answer Database

FAQ1942D.txt - Native Access driver and ISAPI/NSAPI

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

Why can't I use the native Access driver in an ISAPI/NSAPI DLL?

Answer:

Because neither DAO 3.0 or DAO 3.5 are thread-safe.

A web server (such as IIS) will always spawn different threads and load the ISAPI DLL in the context of the thread. The ISAPI DLL is then talking to the BDE which talks to the DAO engine (not thread-safe).

So, if you need to write an ISAPI/NSAPI DLL talking to Access 95/97 tables, you'll need to use Microsoft's ODBC driver. The ODBC driver doesn't use the DAO engine, and is thread-safe.

4/2/99 11:43:26 AM

Last Modified: 01-SEP-99



## Article #16945: Problems with MS Access

Question and Answer Database

FAQ1945C.txt Problems with MS Access

Category :Database Issues

Platform :All

Product :BC++ 5.x

### Question:

I'm having trouble with using BDE 3.5 to talk to a Microsoft Access table. Specifically, if the table has a primary key, and I create a secondary key which uses one of the fields in the primary key, my C++ program no longer recognizes the primary key. What do I need to do?

### Answer:

You need to have a secondary key which uses the `_first record_` in the primary key. It is not necessary that all secondary keys do this, just that one of them does.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16949: Clearing the window in a console application

Question and Answer Database

FAQ1949D.txt Clearing the window in a console application

Category :Miscellaneous

Platform :All

Product :All 32 bit

Question:

How can I guarantee to clear my window in a console application?

Answer:

The following example demonstrates clearing the console window by using the screensize information passed back from the Windows API function `GetConsoleScreenBufferInfo()` to write several blank lines.

Example:

```
program Project1;

{$APPTYPE CONSOLE}

uses
  Windows;

{$R *.RES}

var
  sbi : TConsoleScreenBufferInfo;
  i : integer;
begin
  Writeln('A Console Applicaiton');
  Writeln('Press Enter To Clear The Screen');
  GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE),
    sbi);
  Readln;
  GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE),
    sbi);
  for i := 0 to sbi.dwSize.y do
    writeln;
  Writeln('Press Enter To End');
```

Readln;  
end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16953: Changing the owner of a component at runtime

Question and Answer Database

FAQ1953D.txt Changing the owner of a component at runtime

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to change the owner of a component at runtime?

Answer:

You can change a component's ownership after creation via the `InsertComponent()` and `RemoveComponent()` methods available to all components. The Delphi help file contains examples for using these methods.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16957: Accessing a grid's column/row by name

Question and Answer Database

FAQ1957D.txt    Accessing a grid's column/row by name

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Is it possible to access a grid row or column by name?

Answer:

The following example presents two functions: `GetGridColumnByName()` and `GetGridRowByName()` that return the given column or row that contains the specified caption.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    StringGrid1.Rows[1].Strings[0] := 'This Row';
    StringGrid1.Cols[1].Strings[0] := 'This Column';
end;

function GetGridColumnByName(Grid : TStringGrid;
                             ColName : string): integer;

var
    i : integer;
begin
    for i := 0 to Grid.ColCount - 1 do
        if Grid.Rows[0].Strings[i] = ColName then begin
            Result := i;
            exit;
        end;
    Result := -1;
end;

function GetGridRowByName(Grid : TStringGrid;
                           RowName : string): integer;

var
    i : integer;
begin
    for i := 0 to Grid.RowCount - 1 do
```

```
    if Grid.Cols[0].Strings[i] = RowName then begin
        Result := i;
        exit;
    end;
Result := -1;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Column : integer;
    Row : integer;
begin
    Column := GetGridColumnByName(StringGrid1, 'This Column');
    if Column = -1 then
        ShowMessage('Column not found') else
        ShowMessage('Column found at ' + IntToStr(Column));
    Row := GetGridRowByName(StringGrid1, 'This Row');
    if Row = -1 then
        ShowMessage('Row not found') else
        ShowMessage('Row found at ' + IntToStr(Row));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16960: Converting ASCII files to Paradox tables

Question and Answer Database

FAQ1960C.txt     Converting ASCII files to Paradox tables

Category     :Database Issues

Platform     :All

Product      :C++Builder 1.x

Question:

How do I import my ASCII file into a Paradox table? Can the Database Desktop do it?

Answer:

No, the Database Desktop can't do it.

Drop 1 TBatchMove and 2 TTables on a form. Set BatchMove1::Mode to batCopy, and Source and Destination to Table1 and Table2, respectively. Set Table1::TableName to an empty Paradox table created with the same structure as the Ascii file. Set Table2::TableName to any non-existent table name such as FOO.TXT, and set Table2::TableType to ttASCII. Double-click on the form and type, BatchMove1->Execute() into the created OnCreate method. Press F9 to build and run. The files FOO.TXT and FOO.SCH should be created.

Now copy the ASCII text file to the directory where FOO.TXT is located. Open FOO.SCH and changed the first line to read the name of the name of the ASCII data file instead of "FOO". Add statements for delimiter and separator, and change Filetype= to "varying" if appropriate. Rename FOO.SCH to the name of the ASCII data file .SCH. Back in your program, change Table2::TableName to the name of your ASCII data file. Change BatchMove1::Source to Table2 and BatchMove1::Destination to Table1. F9 to rebuild and run.

You can now open the paradox table in Database Desktop and see your data. If the data did not splice correctly into the target fields, you will need to modify your schema file (.sch).

Also see the file ASCII DRV.TXT in Delphi3, or TI2844 "Using The

ASCII Driver With Comma-delimited Files", or Delphi3\Help\Examples  
\Batchmv.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99



## Article #16961: Reducing flicker on component redraw

Question and Answer Database

FAQ1961D.txt Reducing flicker on component redraw

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is there any way to reduce component flicker when the component is redrawn

Answer:

Adding the csOpaque flag to the ControlStyle of the component should eliminate the repainting of the component's background.

Example:

```
constructor TMyControl.Create;  
begin  
    inherited;  
    ControlStyle := ControlStyle + [csOpaque];  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16965: Assigning a comp type to an integer

Question and Answer Database

FAQ1965D.txt     Assigning a comp type to an integer

Category     : Object Pascal

Platform     : All

Product      : All 32 bit

### Question:

I am using a comp type (64 bit integer) and would like to assign the value to an integer. Using the Trunc function is too slow. Is there a better way?

### Answer:

The VCL defines a type called TLargeInteger that is a comp type that is broken down into the integer components Low and High. You could simply cast the comp as a TLargeInteger and access one of it's members, and avoid using floating point functions to get at it's underlying integer value.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  c : comp;
  i : integer;
begin
  c := $FFFFFF;
  // i := c; This won't work - incompatible types!
  i := Trunc(c); // This works but has additional overhead;
  i := TLargeInteger(c).LowPart;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16969: Testing if you are connected to the internet

Question and Answer Database

FAQ1969D.txt - Testing if you are connected to the internet

Category :Internet/WEB  
Platform :All-32Bit  
Product :

Question:

How can I tell at runtime if I am connected to the internet?

Answer:

You can use the TCP component to retrieve the Local IP address. If it is "0.0.0.0" then there is no connection.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if TCP1.LocalIp = '0.0.0.0' then  
        ShowMessage('Your not connected!');  
end;
```

4/2/99 11:47:13 AM

Last Modified: 01-SEP-99

## Article #16972: Rebuilding Paradox Index files

Question and Answer Database

FAQ1972C.txt Rebuilding Paradox Index files

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

How do I rebuild an index file for a Paradox table?

Answer:

You can do a couple things.

1. Delete the index file, restructure the table, and add the index back to the table. This will automatically rebuild the index.
2. By using the BDE API you can actually repair the table. An example of this is shown using VCL(Delphi and BCB), and OWL. This example can be found in Borlands web page in the product area under BCB->BDE.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #16973: Setting the number of fixed columns in a TDbGrid

Question and Answer Database

FAQ1973D.txt     Setting the number of fixed columns in a TDbGrid

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Is there any way to set the number of fixed columns in a TDbGrid?

Answer:

The following example demonstrates casting a TDbGrid component to a TStringGrid component to surface the FixedCols property.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TStringGrid(DbGrid1).FixedCols := 2;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16977: Changing the default form size

Question and Answer Database

FAQ1977D.txt Changing the default form size

Category :IDE

Platform :All

Product :All 32 bit

### Question:

How can I change the default form size that Delphi / C++ Builder uses?

### Answer:

While the default form size is hard coded in the IDE, you can simply create a form to your liking, and save the form template in the repository for future reuse.

To add a form to the repository, create your new form and save it to a directory (usually to the Objrepos directory located in the Delphi or CBuilder directory), then right click on the form and choose "Add to repository". After completing the information requested in the repository dialog, you will be able to add your new default form to any project by selecting "File->New" from the IDE's main menu, and choosing your new form.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16981: Getting the handle of a Window containing a DOS box

Question and Answer Database

FAQ1981D.txt    Getting the handle of a Window containing a DOS box  
Category       :Windows API  
Platform        :All  
Product         :All 32 bit

### Question:

How can I get the handle of the Window containing a DOS box or other console mode application?

### Answer:

The following example demonstrates using the Windows API function FindWindow() to retrieve the handle of a console mode application. Note that the WndClass of a console window is different under Windows 95 and Windows NT, and the title of the Window may contain a full path name under Windows NT.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  info : TOSVersionInfo;
  ClassName : string;
  Title : string;
begin
  {See if we're running on Win95 or NT.}
  info.dwOSVersionInfoSize := sizeof(info);
  GetVersionEx(info);
  if (info.dwPlatformId = VER_PLATFORM_WIN32_NT) then begin
    ClassName := 'ConsoleWindowClass';
    Title := 'Command Prompt';
  end else begin
    ClassName := 'tty';
    Title := 'MS-DOS Prompt';
  end;
  ShowMessage(IntToStr(FindWindow(PChar(ClassName), PChar(Title))));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #16985: Refreshing the TDriveComboBox drive list

Question and Answer Database

FAQ1985D.txt Refreshing the TDriveComboBox drive list

Category :VCL

Platform :All

Product :All 32 bit

### Question:

How can I get the TDriveComboBox component to refresh its' list, given that networked drives can be added, deleted, and remapped, and drives can be hot swapped by using plug and play devices?

### Answer:

The following example demonstrates surfacing the protected BuildList() method of the TDriveComboBox component to regenerate the list of current disk drives.

### Example:

```
type
  TNewDriveComboBox = class(TDriveComboBox)
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Drive : char;
begin
  Drive := DriveComboBox1.Drive;
  TNewDriveComboBox(DriveComboBox1).BuildList;
  DriveComboBox1.Drive := Drive;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16989: Removing drives listed in a TDriveComboBox

Question and Answer Database

FAQ1989D.txt Removing drives listed in a TDriveComboBox

Category :VCL

Platform :All

Product :All 32 bit

### Question:

On many laptop computers, there is a possibility that no floppy drive is installed. Is it possible to remove drives that are listed in the TDriveComboBox component to avoid listing non existent drives?

### Answer:

The following example demonstrates removing drives that are not ready for use from the TDriveComboBox component. Note that on many computers, there will be a significant delay in checking for a plug and play floppy drive.

### Example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i : integer;
  OldErrorMode : Word;
  OldDirectory : string;
begin
  OldErrorMode := SetErrorMode(SEM_NOOPENFILEERRORBOX);
  GetDir(0, OldDirectory);
  i := 0;
  while i <= DriveComboBox1.Items.Count - 1 do begin
    {$I-}
    ChDir(DriveComboBox1.Items[i][1] + ':\');
    {$I+}
    if IoResult <> 0 then
      DriveComboBox1.Items.Delete(i) else
      inc(i);
  end;
  ChDir(OldDirectory);
  SetErrorMode(OldErrorMode);
```

end;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #16993: Determining if you have Delphi v3.02 installed

Question and Answer Database

FAQ1993D.txt - Determining if you have Delphi v3.02 installed

Category :Delphi 3.x  
Platform :All-32Bit  
Product :Delphi3.x,

### Question:

I am updating many machines in our office to Delphi 3.02. How can I determine if a given machine has been upgraded?

### Answer:

Check the Bin directory for DELPHI.EXE. If it's file time is listed as 3:00, you have 3.0. If it's file time is listed 3:01, you may have either 3.01 or 3.02. In this case, check the time stamp of TLIBIMP.EXE. If it's file time is listed as 3.02, then you have 3.02.

You can also get:

```
"/devsupport/delphi/download_files/vcnp.zip">version checking utility  
"/devsupport/delphi/download_files/vcp.exe">version checking utility  
(Package EXE)
```

4/1/99 2:40:58 PM

Last Modified: 01-SEP-99

## Article #16996: Defining PRIMARYLANGID(), SUBLANGID(), and MAKELANGID() macros

Question and Answer Database

FAQ1996D.txt Defining PRIMARYLANGID(), SUBLANGID(), and MAKELANGID()  
( ) macros

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I define the PRIMARYLANGID(), SUBLANGID() and MAKELANGID() macros that are used with the Windows API functions that accept language identifiers?

Answer:

The follow example demonstrates functional equivalents for the PRIMARYLANGID(), SUBLANGID(), and MAKELANGID() macros.

```
function PRIMARYLANGID(lgid : Word) : LongInt;
begin
    result := lgid and $3FF;
end;
```

```
function SUBLANGID(lgid : Word) : LongInt;
begin
    result := lgid shr 10;
end;
```

```
function MAKELANGID(sPrimaryLanguage : Word;
                   sSubLanguage : Word) : Word;
begin
    result := (sSubLanguage shl 10) or
              sPrimaryLanguage;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17000: Forcing edit mode in a TListView via a keypress

Question and Answer Database

FAQ2000D.txt Forcing edit mode in a TListView via a keypress

Category :VCL

Platform :All

Product :All 32 bit

### Question:

When the user clicks on a caption in the listview, it goes into edit mode. How can I force the edit mode by pressing F2 or a similar key?

### Answer:

The following example demonstrates trapping the F2 key in a Listview control and forcing the caption into edit mode.

### Example:

```
procedure TForm1.ListView1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Ord(Key) = VK_F2 then
    ListView1.Selected.EditCaption;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17004: Differences between Create(Self) and Create(Application)

Question and Answer Database

FAQ2004D.txt Differences between Create(Self) and Create  
(Application)

Category :VCL

Platform :All

Product :All 32 bit

Question:

What's the difference between TForm.Create(Self) and  
TForm.Create(Application)?

Answer:

Self is an implicit variable which can only be used within class  
methods, and refers to the current class instance. So "Self" within a  
method of the TForm1 class refers to the current instance of TForm1.

When you create a form or component, you pass its owner in the  
constructor. When a form or component is freed, it automatically frees  
all the components which it owns as well.

So if you pass an Application as the owner when you create a form  
instance, the form instance will be automatically freed when the  
Application is freed.

If you pass a form instance as the owner when you create another form  
instance, the second form will be freed when the first is freed.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17008: Using the Shell API function SHBrowseForFolder()

Question and Answer Database

FAQ2008D.txt Using the Shell API function SHBrowseForFolder()  
Category :Windows API  
Platform :All  
Product :All 32 bit

Question:

How can I use the Shell API function SHBrowseForFolder() to let the user browse for a folder?

Answer:

The following example demonstrates using the SHBrowseForFolder() function to allow the user to select a given directory object.

Example:

```
uses ShellAPI, ShlObj;

procedure TForm1.Button1Click(Sender: TObject);
var
  TitleName : string;
  lpItemID : PItemIDList;
  BrowseInfo : TBrowseInfo;
  DisplayName : array[0..MAX_PATH] of char;
  TempPath : array[0..MAX_PATH] of char;
begin
  FillChar(BrowseInfo, sizeof(TBrowseInfo), #0);
  BrowseInfo.hwndOwner := Form1.Handle;
  BrowseInfo.pszDisplayName := @DisplayName;
  TitleName := 'Please specify a directory';
  BrowseInfo.lpszTitle := PChar(TitleName);
  BrowseInfo.ulFlags := BIF_RETURNONLYFSDIRS;
  lpItemID := SHBrowseForFolder(BrowseInfo);
  if lpItemID <> nil then begin
    SHGetPathFromIDList(lpItemID, TempPath);
    ShowMessage(TempPath);
    GlobalFreePtr(lpItemID);
  end;
end;
```



end;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17012: Adding accelerator characters to TTabsheets

Question and Answer Database

FAQ2012D.txt Adding accelerator characters to TTabsheets

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I make an accelerator character work for a Tabsheet?

I put accelerator characters in the captions for each Tabsheet of a PageControl, but when I try to tab the pages using accelerator key, it beeps and does nothing.

Answer:

The following example demonstrates trapping the CM\_DIALOGCHAR message to surface the accelerator messages for the tab sheets of a PageControl.

Example:

```
type
  TForm1 = class(TForm)
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    TabSheet2: TTabSheet;
    TabSheet3: TTabSheet;
  private
    { Private declarations }
    procedure CMDialogChar(var Msg:TCMDialogChar);
      message CM_DIALOGCHAR;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.CMDDialogChar(var Msg:TCMDDialogChar);
var
  i:Integer;
begin
  with PageControll do begin
    if Enabled then
      for i := 0 to PageControll.PageCount - 1 do
        if ((IsAccel(Msg.CharCode, Pages[i].Caption)) and
          (Pages[i].TabVisible)) then begin
          Msg.Result:=1;
          ActivePage := Pages[i];
          exit;
        end;
      end;
    end;
  inherited;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17016: Limiting the amount of text entered in a TEdit

Question and Answer Database

FAQ2016D.txt Limiting the amount of text entered in a TEdit

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I limit the amount of text that is entered into a TEdit control so the text entered does not exceed the width of the TEdit control?

Answer:

The following two examples demonstrate limiting the text in a edit control to the amount of text that fits in the client area of the control to avoid scrolling text. The first example sets the MaxLength property to the number of "W" characters that will fit in the control. The "W" character is chosen, since it is most likely to be the widest character in the font. This method will work well for fixed pitch fonts, but will produce less than desirable effects when using a variable width font. The second method traps the edit control's key press method, and measures the text width of the current string in the edit control and the width of the character that will be added. If the width exceeds the client area of the edit control, the key is discarded, and a MessageBeep is produced.

Examples:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  cRect : TRect;
  bm : TBitmap;
  s : string;
begin
  Windows.GetClientRect(Edit1.Handle, cRect);
  bm := TBitmap.Create;
  bm.Width := cRect.Right;
  bm.Height := cRect.Bottom;
  bm.Canvas.Font := Edit1.Font;
  s := 'W';
```

```
while bm.Canvas.TextWidth(s) < CRect.Right do
  s := s + 'W';
if length(s) > 1 then begin
  Delete(s, 1, 1);
  Edit1.MaxLength := Length(s);
end;
end;
```

{Alternatively}

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
var
  cRect : TRect;
  bm : TBitmap;
begin
  if ((Ord(Key) <> VK_TAB) and
    (Ord(Key) <> VK_RETURN) and
    (Ord(Key) <> VK_LEFT) and
    (Ord(Key) <> VK_BACK)) then begin
    Windows.GetClientRect(Edit1.Handle, cRect);
    bm := TBitmap.Create;
    bm.Width := cRect.Right;
    bm.Height := cRect.Bottom;
    bm.Canvas.Font := Edit1.Font;
    if bm.Canvas.TextWidth(Edit1.Text + Key) > CRect.Right then begin
      Key := #0;
      MessageBeep(-1);
    end;
    bm.Free;
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17020: Detecting if the system time has been changed

Question and Answer Database

FAQ2020D.txt Detecting if the system time has been changed

Category :Windows API

Platform :All

Product :All 32 bit

Question:

Is it possible to detect if the system time has been changed by another application?

Answer:

The following example demonstrates trapping the WM\_TIMECHANGE message. Note: Applications that change the system time should broadcast a WM\_TIMECHANGE message to all top level windows.

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
    procedure WMTIMECHANGE(var Message: TWMTIMECHANGE);
      message WM_TIMECHANGE;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMTIMECHANGE(var Message: TWMTIMECHANGE);
begin
  Form1.Caption := 'Time Changed';
end;
```

Last Modified: 01-SEP-99

## Article #17024: Trapping for when a user is done resizing a window

Question and Answer Database

FAQ2024D.txt Trapping for when a user is done resizing a window

Category :Windows API

Platform :All

Product :All 32 bit

Question:

I want to trap when the user is finished sizing or moving my window. I trapped the WM\_SIZE and the WM\_MOVE messages, but I receive many of these messages, and seems impossible to tell when the user has completed the operation. How can I tell when the user is finished moving and sizing the window?

Answer:

The following example demonstrates trapping the WM\_EXITSIZEMOVE message to determine when the user has exited a Window sizing or moving event. Although the message is documented as available only under Windows NT, it does work equally as well under Windows 95. Note that you can also trap the WM\_ENTERSIZEMOVE to determine when the user is initiating a window move or size operation.

Example:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    procedure WMEXITSIZEMOVE(var Message: TMessage);
      message WM_EXITSIZEMOVE;
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```



```
procedure TForm1.WMEXITSIZEMOVE(var Message: TMessage);  
begin  
    Form1.Caption := 'Finished Moving and sizing';  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17028: Using the WIN API high resolution performance counter

Question and Answer Database

FAQ2028D.txt Using the WIN API high resolution performance counter

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I make use of the high resolution performance counter available under Win32?

Answer:

The following example demonstrates using the TLargeInteger type to query the performance counter. Note that the TLargeInteger type (64 bit integer) can be converted and referenced as a whole by casting it to a Comp type.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  li : TLARGEINTEGER;
begin
  QueryPerformanceFrequency(li);
  ShowMessage(FloatToStr(Comp(li)));
  QueryPerformanceCounter(li);
  ShowMessage(FloatToStr(Comp(li)));
  QueryPerformanceCounter(li);
  ShowMessage(FloatToStr(Comp(li)));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17032: Keyboard shortcut to a component that has no caption

Question and Answer Database

FAQ2032D.txt    Keyboard shortcut to a component that has no caption  
Category        :VCL  
Platform        :All  
Product         :All 32 bit

### Question:

How can I create a keyboard shortcut to a component that has no caption?

### Answer:

One work around is to leverage the FocusControl property of TLabel. The following example demonstrates using an invisible label control to create a keyboard shortcut to a memo control. The shortcut will respond to the Alt+M key combination. To use this example, place a label, a memo, and several other controls that receive focus. Run the application, move focus to a control other than "Memo1", and key in -M. You should see focus move to Memo1.

### Example:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Label1.Visible := false;  
    Label1.Caption := '&M';  
    Label1.FocusControl := Memo1;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17036: Getting modem status under Win32

Question and Answer Database

FAQ2036D.txt Getting modem status under Win32

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I get the modem status under Win32?

Answer:

The following example demonstrates retrieving the current state of the modem control-register values under Win32.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  CommPort : string;
  hCommFile : THandle;
  ModemStat : DWord;
begin
  CommPort := 'COM2';

  {Open the comm port}
  hCommFile := CreateFile(PChar(CommPort),
                        GENERIC_READ,
                        0,
                        nil,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        0);

  if hCommFile = INVALID_HANDLE_VALUE then
  begin
    ShowMessage('Unable to open '+ CommPort);
    exit;
  end;

  {Get the Modem Status}
```

```
if GetCommModemStatus(hCommFile, ModemStat) <> false then begin
  if ModemStat and MS_CTS_ON <> 0 then
    ShowMessage('The CTS (clear-to-send) is on.');
```

if ModemStat and MS\_DSR\_ON <> 0 then  
 ShowMessage('The DSR (data-set-ready) is on.');

if ModemStat and MS\_RING\_ON <> 0 then  
 ShowMessage('The ring indicator is on.');

if ModemStat and MS\_RLSD\_ON <> 0 then  
 ShowMessage('The RLSD (receive-line-signal-detect) is  
on.');

```
end;
```

```
{Close the comm port}
  CloseHandle(hCommFile);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17039: BDE 4.51 causes Application Error

Question and Answer Database

FAQ2039C.txt BDE 4.51 causes Application Error

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

when building a program that interacts with MS Access using the BDE 4.51 and C++Builder, I find that programs I run under the debugger crash with an exception. The programs work fine when run from windows, and I can access the data fine at designtime. What is happening?

Answer:

The DAO, like the VCL, uses exceptions internally to propogate information and control structures. The C++Builder debugger is designed so as to trigger a user interrupt when a certain number of internal exceptions have been tracked. That interrupt causes the exception message to appear.

The program being debugged has not crashed, and it is perfectly reasonable to resume execution without doing anything other than clicking on the exception message box. The interruption is only intended to serve as a notification about what is happening behind the scenes.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #17040: Disabling a radio button in a TRadioGroup

Question and Answer Database

FAQ2040D.txt    Disabling a radio button in a TRadioGroup

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Is it possible to disable a given radio button in a RadioGroup?

Answer:

The following example demonstrates accessing a individual radio button in a TRadioGroup component. Note that the RadioGroup.Controls is a zero based array.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    TRadioButton(RadioGroup1.Controls[1]). Enabled := False;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17044: Click a speed button in code

Question and Answer Database

FAQ2044D.txt Click a speed button in code

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I click a SpeedButton component in code? I tried using SendMessage but it does not work since Speedbuttons do not have handles.

Answer:

The following example demonstrates using the Perform method of TControl to send the message.

Example:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  ShowMessage('clicked');
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SpeedButton1.Perform(WM_LBUTTONDOWN, 0, 0);
  SpeedButton1.Perform(WM_LBUTTONUP, 0, 0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17048: What is the tag property useful for?

Question and Answer Database

FAQ2048D.txt    What is the tag property useful for?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

What is the tag property useful for?

Answer:

The tag property is a storage area that is associated with most all components, and can be used to store any 32 bit type such as pointers to data structures and other objects. The tag property is not used by the system, and is included for the programmers convenience.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17052: Using an instance of TPrinter

Question and Answer Database

FAQ2052D.txt Using an instance of TPrinter

Category :VCL

Platform :All

Product :All 32 bit

Question:

When I create an instance of TPrinter using TPrinter.Create, I get an exception. How can I get around this?

Answer:

There is no need to create an instance of TPrinter using TPrinter.Create, as an instance of TPrinter (called Printer) is automatically created when you use the Printers unit.

Example:

```
uses Printers;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    Printer.BeginDoc;
```

```
    Printer.Canvas.TextOut(100, 100, 'Hello World!');
```

```
    Printer.EndDoc;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17056: Customizing TTrackBar

Question and Answer Database

FAQ2056D.txt Customizing TTrackBar

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I create a trackbar where the track has a thin line on which the slider moves along, rather than a rather big white space which the slider moves along?

Answer:

The following example demonstrates creating a descendant of the TTrackbar component that overrides the CreateParams method, and clears the TBS\_ENABLESELRANGE flag from the params style. This results in a Trackbar component that has a small thin track for the slider to move upon. Note that the TBS\_ENABLESELRANGE constant is declared in the CommCtrl unit.

Example:

```
uses CommCtrl, ComCtrls;
```

```
type TMyTrackBar = class(TTrackBar)
```

```
  procedure CreateParams(var Params: TCreateParams); override;  
end;
```

```
procedure TMyTrackBar.CreateParams(var Params: TCreateParams);  
begin  
  inherited;  
  Params.Style := Params.Style and not TBS_ENABLESELRANGE;  
end;
```

```
var  
  MyTrackbar : TMyTrackbar;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin
```

```
MyTrackBar := TMyTrackbar.Create(Form1);  
MyTrackbar.Parent := Form1;  
MyTrackbar.Left := 100;  
MyTrackbar.Top := 100;  
MyTrackbar.Width := 150;  
MyTrackbar.Height := 45;  
MyTrackBar.Visible := true;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17060: Detecting tab key press

Question and Answer Database

FAQ2060D.txt Detecting tab key press

Category :VCL

Platform :All

Product :All 32 bit

Question:

Since the KeyPress and the KeyDown events do not get called for the tab key, how do I trap the tab key at the form level?

Answer:

At form level, the tab key is generally handled by Windows. The following example demonstrates creating a CM\_Dialog message handler to trap for Dialog keys. The code surfaces the tab character through the KeyPress event.

Example:

```
type
  TForm1 = class(TForm)
  private
    procedure CMDialogKey( Var msg: TCMDialogKey );
    message CM_DIALOGKEY;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.CMDialogKey(var msg: TCMDialogKey);
begin
  if msg.Charcode <> VK_TAB then
    inherited;
end;
```

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = VK_TAB then
    Form1.Caption := 'Tab Key Down!';
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17064: adding system menu items to a form

Question and Answer Database

FAQ2064D.txt adding system menu items to a form

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I add a custom item to the system menu of my application?

Answer:

The following example demonstrates using the Windows API function `AppendMenu()` to append a new menu item to the system menu. We will define a new system command

constant : `SC_MyMenuItem` that will be used to identify our new item to the system. After adding the menu item, we will trap the `WM_SYSCOMMAND` message to test if our new menu item was selected.

Example:

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    procedure WMSysCommand(var Msg: TWMSysCommand);
      message WM_SYSCOMMAND;
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
const
```

```
SC_MyMenuItem = WM_USER + 1;
```

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  AppendMenu(GetSystemMenu(Handle, FALSE), MF_SEPARATOR, 0, '');  
  AppendMenu(GetSystemMenu(Handle, FALSE),  
    MF_STRING,  
    SC_MyMenuItem,  
    'My Menu Item');  
end;
```

```
procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);  
begin  
  if Msg.CmdType = SC_MyMenuItem then  
    ShowMessage('Got the message') else  
    inherited;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17068: Resizing StringGrid Columns

Question and Answer Database

FAQ2068D.txt Resizing StringGrid Columns

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to automatically size a column in a TStringGrid component to fit the text of the largest string in the column?

Answer:

The following example demonstrates auto sizing a column in a TStringGrid Component.

Example:

```
procedure AutoSizeGridColumn(Grid : TStringGrid;
                             column : integer);
var
  i : integer;
  temp : integer;
  max : integer;
begin
  max := 0;
  for i := 0 to (Grid.RowCount - 1) do begin
    temp := Grid.Canvas.TextWidth(grid.cells[column, i]);
    if temp > max then max := temp;
  end;
  Grid.ColWidths[column] := Max + Grid.GridLineWidth + 3;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  AutoSizeGridColumn(StringGrid1, 1);
end;
```



## Article #17071: Endless looping with the TMediaPlayer

Question and Answer Database

FAQ2071D.txt    Endless looping with the TMediaPlayer

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I get an AVI to loop continually with the TMediaPlayer?

Answer:

The following example demonstrates looping an AVI file by taking advantage of the MediaPlayer's Notify event to play the AVI file again... and again... and again!

Example:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  with MediaPlayer1 do
    if NotifyValue = nvSuccessful then begin
      Notify := True;
      Play;
    end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17075: Frame count of an AVI

Question and Answer Database

FAQ2075D.txt     Frame count of an AVI

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I tell how many frames are in a given AVI file and how long the AVI file will play?

Answer:

The TMediaPlayer provides this information by setting the TimeFormat property and querying the Length property. The following example demonstrates retrieving the number of frames and the play time of a given AVI file.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MediaPlayer1.TimeFormat := tfFrames;
  ShowMessage('Number of frames = ' +
    IntToStr(MediaPlayer1.Length));
  MediaPlayer1.TimeFormat := tfMilliseconds;
  ShowMessage('Number of milliseconds = ' +
    IntToStr(MediaPlayer1.Length));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17079: drawing transparent text onto a TBitmap

Question and Answer Database

FAQ2079D.txt drawing transparent text onto a TBitmap

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I draw transparent text on a TBitmap?

Answer:

The following example demonstrates drawing transparent text on the canvas of a TBitmap.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldBkMode : integer;
begin
  Image1.Picture.Bitmap.Canvas.Font.Color := clBlue;
  OldBkMode := SetBkMode(Image1.Picture.Bitmap.Canvas.Handle,
                        TRANSPARENT);
  Image1.Picture.Bitmap.Canvas.TextOut(10, 10, 'Hello');
  SetBkMode(Image1.Picture.Bitmap.Canvas.Handle,
            OldBkMode);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17083: displaying seconds, minutes, tracks of an AVI

Question and Answer Database

FAQ2083D.txt displaying seconds, minutes, tracks of an AVI

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I display the real time track selection and current play time of an Audio CD track?

Answer:

The following example demonstrates displaying the current track and play time of an AudioCD, updating the values every second by utilizing a timer event.

Example:

```
uses
    MMSystem;

procedure TForm1.Timer1Timer(Sender: TObject);
var
    Trk : Word;
    Min : Word;
    Sec : Word;
begin
    with MediaPlayer1 do begin
        Trk := MCI_TMSF_TRACK(Position);
        Min := MCI_TMSF_MINUTE(Position);
        Sec := MCI_TMSF_SECOND(Position);
        Label1.Caption := Format('%.2d', [Trk]);
        Label2.Caption := Format('%.2d:%.2d', [Min, Sec]);
    end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17087: Clearing the contents of a Canvas

Question and Answer Database

FAQ2087D.txt Clearing the contents of a Canvas

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I clear the contents of a canvas?

Answer:

Simply draw a rectangle in the desired color.

Example:

```
Canvas.Brush.Color := ClWhite;  
Canvas.FillRect(Canvas.ClipRect);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17091: Shift, Ctrl, Alt key checking

Question and Answer Database

FAQ2091D.txt    Shift, Ctrl, Alt key checking

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I determine if the Shift, Alt, or Ctrl key is depressed at any given time?

Answer:

The following example demonstrates checking if the Shift key is pressed during a menu item selection. The example presents functions for testing the status of the Alt, Ctrl, and shift keys.

Example:

```
function CtrlDown : Boolean;
var
  State : TKeyboardState;
begin
  GetKeyboardState(State);
  Result := ((State[vk_Control] And 128) <> 0);
end;
```

```
function ShiftDown : Boolean;
var
  State : TKeyboardState;
begin
  GetKeyboardState(State);
  Result := ((State[vk_Shift] and 128) <> 0);
end;
```

```
function AltDown : Boolean;
var
  State : TKeyboardState;
begin
  GetKeyboardState(State);
```

```
Result := ((State[vk_Menu] and 128) <> 0);  
end;
```

```
procedure TForm1.MenuItem12Click(Sender: TObject);  
begin  
  if ShiftDown then  
    Form1.Caption := 'Shift' else  
    Form1.Caption := '';  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17095: making a Delphi DLL with as little in the uses clause as possible

Question and Answer Database

FAQ2095D.txt making a Delphi DLL with as little in the uses clause as possible

Category :VCL  
Platform :All  
Product :All 32 bit

Question:

I want to create a small dynamic link library that contains only a few small functions. I do not need any additional units, but Delphi insists that the uses clause is incorrect if I remove the standard uses clause. The code does compile to a very small dll if I do remove the uses clause. May I disregard the error when I load the project?

Answer:

While it is ok to disregard the error, it would be better to create a unit that would contain the code for your functions. This way the main project file can have a uses statement that will satisfy the IDE.

Example:

```
(* Main project file *)

library Project1;

uses
  Unit1;

exports
  AddLong index 1 name 'ADDLONG' resident;

begin
end.

(* unit1's code *)

unit Unit1;
```

interface

```
function AddLong(a : longint;  
                b : longint) : longint  
{ $IFDEF Win32 } stdcall; { $ELSE } ; export; { $ENDIF }
```

implementation

```
function AddLong(a : longint;  
                b : longint) : longint;
```

```
begin  
  AddLong := a + b;  
end;
```

end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17099: Moving a form by clicking in its client area

Question and Answer Database

FAQ2099D.txt Moving a form by clicking in its client area

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I make a form move by clicking and dragging in the client area instead of on the caption bar?

Answer:

You could trap the mouse move messages, and move the form based on the net movement of the mouse. An easy way to achieve the same result is to make Windows believe the caption bar was clicked on by trapping and modifying the WM\_NCHITTEST message.

Example:

type

```
TForm1 = class(TForm)
private
  { Private declarations }
public
  procedure WMNCHitTest(var M: TWMNCHitTest);
    message WM_NCHITTEST;
  { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.WMNCHitTest(var M: TWMNCHitTest);
begin
```

```
inherited;  
{If the client has been clicked, make Windows believe}  
{it was the caption bar that was clicked on}  
  if M.Result = htClient then  
    M.Result := htCaption;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17103: Keeping an application minimized

Question and Answer Database

FAQ2103D.txt Keeping an application minimized

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I keep my application minimized as an icon?

Answer:

The following example demonstrates trapping the WM\_QUERYOPEN message to prevent the application from restoring from an iconized state. Note that the Window state of the form must be set to Minimized in the object inspector for this method to work.

Example:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
    procedure WMQueryOpen(var Msg : TWMQueryOpen);
      message WM_QUERYOPEN;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMQueryOpen(var Msg : TWMQueryOpen);
begin
  Msg.Result := 0;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17107: Clearing the recent Documents from the Start Menu

Question and Answer Database

FAQ2107D.txt Clearing the recent Documents from the Start Menu

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I clear the list of recent documents in the Windows Start menu?

Answer:

Call the Windows API function SHAddToRecentDocs() passing nil as the filename parameter.

Example:

uses

ShlOBJ;

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
  SHAddToRecentDocs(SHARD_PATH, nil);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17110: Changing RichEdit Font styles with HotKey combinations

Question and Answer Database

FAQ2110D.txt Changing RichEdit Font styles with HotKey combinations

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I change the RichEdit font style in response to a dedicated keystroke combination?

Answer:

The following example demonstrates changing the font style in a TRichEdit component the following key combinations are pressed:

Ctrl + B - Turns on and off Bold styling

Ctrl + I - Turns on and off Italic styling

Ctrl + S - Turns on and off Strikeout styling

Ctrl + U - Turns on and off Underline styling

Note: It is possible to set the font to include multiple styles.

Example:

```
const
  KEY_CTRL_B = 02;
  KEY_CTRL_I = 9;
  KEY_CTRL_S = 19;
  KEY_CTRL_U = 21;

procedure TForm1.RichEdit1KeyPress(Sender: TObject; var Key: Char);
begin
  case Ord(Key) of
    KEY_CTRL_B: begin
      Key := #0;
      if fsBold in (Sender as TRichEdit).SelAttributes.Style then
        (Sender as TRichEdit).SelAttributes.Style :=
          (Sender as TRichEdit).SelAttributes.Style - [fsBold] else
        (Sender as TRichEdit).SelAttributes.Style :=
```

```

        (Sender as TRichEdit).SelAttributes.Style + [fsBold];
end;
KEY_CTRL_I: begin
    Key := #0;
    if fsItalic in
        (Sender as TRichEdit).SelAttributes.Style then
            (Sender as TRichEdit).SelAttributes.Style :=
                (Sender as TRichEdit).SelAttributes.Style - [fsItalic]
        else
            (Sender as TRichEdit).SelAttributes.Style :=
                (Sender as TRichEdit).SelAttributes.Style + [fsItalic];
        end;
KEY_CTRL_S: begin
    Key := #0;
    if fsStrikeout in
        (Sender as TRichEdit).SelAttributes.Style then
            (Sender as TRichEdit).SelAttributes.Style :=
                (Sender as TRichEdit).SelAttributes.Style - [fsStrikeout]
        else
            (Sender as TRichEdit).SelAttributes.Style :=
                (Sender as TRichEdit).SelAttributes.Style + [fsStrikeout];
        end;
KEY_CTRL_U: begin
    Key := #0;
    if fsUnderline in
        (Sender as TRichEdit).SelAttributes.Style then
            (Sender as TRichEdit).SelAttributes.Style :=
                (Sender as TRichEdit).SelAttributes.Style - [fsUnderline]
        else
            (Sender as TRichEdit).SelAttributes.Style :=
                (Sender as TRichEdit).SelAttributes.Style + [fsUnderline];
        end;
    end;
end;
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17114: using the Lookup method to return values from multiple columns

Question and Answer Database

FAQ2114D.txt using the Lookup method to return values from multiple columns

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How do I use the Lookup method to return values from multiple columns?

Answer:

The result is a variant array -- one index to the array for every column specified in the KeyValues parameter:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyResults: Variant;
  MyKeyFields, MyKeyValues: String;
  MySearchForValue: Integer;
begin
  MyKeyFields := 'CustNo';
  MySearchForValue := 1351;
  MyReturnColumns := 'Company;Addr1';
  MyResults := Table1.Lookup(MyKeyFields, MySearchForValue,
                             MyKeyValues);
  if not VarIsNull(MyResults) then
    ShowMessage('Company: ' + MyResults[0] + ' Address: ' +
                MyResults[1]);
  { With DBDemos data this returns
    "Company: Sight Diver Address: 1 Neptune Lane" }
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17118: Copying files using the Standard Windows Copy file dialog box

Question and Answer Database

FAQ2118D.txt Copying files using the Standard Windows Copy file dialog box

Category :Windows API  
Platform :All  
Product :All 32 bit

Question:

Is there any way to copy a group of files and use the same file copy animation dialog box that the Windows Explorer uses?

Answer:

The following example demonstrates using the SHFileOperation function to copy a group of files and display a progress dialog. You can also use the following flags to delete, move and rename a group of files.

TO\_COPY  
FO\_DELETE  
FO\_MOVE  
FO\_RENAME

Note: The buffer that contains the file names to copy must end with a double null terminating character;

Example:

```
uses ShellAPI;

procedure TForm1.Button1Click(Sender: TObject);
var
  Fo      : TSHFileOpStruct;
  buffer  : array[0..4096] of char;
  p       : pchar;

begin
  FillChar(Buffer, sizeof(Buffer), #0);
  p := @buffer;
  p := StrECopy(p, 'C:\Download\1.ZIP') + 1;
```

```
p := StrECopy(p, 'C:\Download\2.ZIP') + 1;
p := StrECopy(p, 'C:\Download\3.ZIP') + 1;
StrECopy(p, 'C:\Download\4.ZIP');

FillChar(Fo, sizeof(Fo), #0);
Fo.Wnd      := Handle;
Fo.wFunc    := FO_COPY;
Fo.pFrom    := @Buffer;
Fo.pTo      := 'D:\';
Fo.fFlags   := 0;
if ((SHFileOperation(Fo) <> 0) or
    (Fo.fAnyOperationsAborted <> false)) then
    ShowMessage('Cancelled')
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17122: Using an Icon as the Glyph in a TSpeedButton

Question and Answer Database

FAQ2122D.txt Using an Icon as the Glyph in a TSpeedButton

Category :VCL

Platform :All

Product :All 32 bit

Question:

Is it possible to use an icon as the Glyph in a TSpeedButton component?

Answer:

The following example demonstrates painting an Icon's image onto a SpeedButton's Glyph.

Example:

```
uses ShellApi;

procedure TForm1.FormShow(Sender: TObject);
var
  Icon: TIcon;
begin
  Icon := TIcon.Create;
  Icon.Handle :=
    ExtractIcon(0,
                'C:\WINDOWS\notepad.exe',
                1);
  SpeedButton1.Glyph.Width := Icon.Width;
  SpeedButton1.Glyph.Height := Icon.Height;
  SpeedButton1.Glyph.Canvas.Draw(0, 0, Icon);
  Icon.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17126: Getting stronger type checking

Question and Answer Database

FAQ2126D.txt    Getting stronger type checking

Category        :VCL

Platform        :All

Product         :All 32 bit

### Question:

How can I get Delphi to perform stronger type checking on user defined types. Example: If I create a user defined type that descends from a double, I can pass a variable of this new type to any function that expects a double. I want Delphi to provide stronger type checking and produce a warning in this instance.

### Answer:

The following example demonstrates Delphi's new Strong type checking types, allowing you to define types that require stronger type checking at compile time.

### Example:

```
type TStrongType = type Double;
type TWeakType = Double;

procedure AddWeakType(var d : TWeakType);
begin
  d := d + 1;
end;

procedure AddStrongType(var d : TStrongType);
begin
  d := d + 1;
end;

procedure AddDoubleType(var d : Double);
begin
  d := d + 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  d : Double;
  s : TStrongType;
  w : TWeakType;
begin
  AddDoubleType(d);           { compiles fine }
  AddDoubleType(w);           { compiles fine }
  AddDoubleType(s);           { <- compile error }
  AddDoubleType(double(s));   { compiles fine }

  AddWeakType(d);             { compiles fine }
  AddWeakType(w);             { compiles fine }
  AddWeakType(s);             { <- compile error }
  AddWeakType(TWeakType(s)); { compiles fine }

  AddStrongType(d);           { <- compile error }
  AddStrongType(TStrongType(d)); { compiles fine }
  AddStrongType(w);           { <- compile error }
  AddStrongType(TStrongType(w)); { compiles fine }
  AddStrongType(s);           { compiles fine }
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17130: Getting a string representation of a Windows error code

Question and Answer Database

FAQ2130D.txt Getting a string representation of a Windows error code

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I get a string representation of the last Windows error value returned by the Windows GetLastError?

Answer:

Use the RTL function SysErrorMessage(GetLastError).

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {Cause a Windows system error message to be logged}
  ShowMessage(IntToStr(1StrLen(nil)));
  ShowMessage(SysErrorMessage(GetLastError));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17134: Getting the path to the directory where Windows was installed from

Question and Answer Database

FAQ2134D.txt Getting the path to the directory where Windows was installed from

Category :VCL  
Platform :All  
Product :All 32 bit

Question:

I need to install some files from the Windows Setup disk in code. On many computers, the Windows setup files reside in CAB files that are installed on the user's hard drive and in other cases, the system is installed from a CD. Where can I find where Windows was originally installed from?

Answer:

The following example demonstrates retrieving the source of the Windows setup files.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  reg: TRegistry;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\SETUP',
             false);
  ShowMessage(reg.ReadString('SourcePath'));
  reg.CloseKey;
  reg.free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17138: responding when the user minimizes or maximizes a Form

Question and Answer Database

FAQ2138D.txt    responding when the user minimizes or maximizes a Form

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I respond to the user minimizing or maximizing my form before the resizing actually takes place?

Answer:

The following example demonstrates trapping the Windows WM\_SYSCOMMAND message. If the command specifies to minimize or maximize the form, we will beep the PC speaker instead;

Example:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
    procedure WMSysCommand(var Msg: TWMSysCommand);
      message WM_SYSCOMMAND;
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.WMSysCommand;
begin
  if (Msg.CmdType = SC_MINIMIZE) or
    (Msg.CmdType = SC_MAXIMIZE) then
```

```
    MessageBeep(0) else
inherited;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17142: Creating a custom word break procedure

Question and Answer Database

FAQ2142D.txt Creating a custom word break procedure

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I create a custom word break procedure for the TEdit, TMemo, or the TRichEdit components?

Answer:

The following example implements a custom work break procedure and attaches it to a TMemo component. Note that the implementation of the word break proc simply returns that it is ok to break the word at the current character. Additional custom handling can be further implemented. For additional information, please see the documentation for the EM\_SETWORDBREAKPROC message in the Win32 Help File.

```
var
  OriginalWordBreakProc : pointer;
  NewWordBreakProc : pointer;

function MyWordBreakProc(LPTSTR  : pchar;
                        ichCurrent : integer;
                        cch : integer;
                        code : integer) : integer
  {$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
  result := 0;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  OriginalWordBreakProc := Pointer(
    SendMessage(Mem1.Handle,
                EM_GETWORDBREAKPROC,
                0,
                0));
```

```
{$IFDEF WIN32}
  NewWordBreakProc := @MyWordBreakProc;
{$ELSE}
  NewWordBreakProc := MakeProcInstance(@MyWordBreakProc,
                                       hInstance);
{$ENDIF}
SendMessage(Mem1.Handle,
            EM_SETWORDBREAKPROC,
            0,
            longint(NewWordBreakProc));
```

end;

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  SendMessage(Mem1.Handle,
              EM_SETWORDBREAKPROC,
              0,
              longint(@OriginalWordBreakProc));
```

```
{$IFNDEF WIN32}
  FreeProcInstance(NewWordBreakProc);
{$ENDIF}
```

end;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17146: Creating a Lookup field at runtime

Question and Answer Database

FAQ2146D.txt Creating a Lookup field at runtime

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How can I create a lookup field at runtime?

Answer:

Creating a lookup field at runtime differs in only a few ways from creating any standard TField object. Three properties of a field that must be set in order to function as a Lookup field are LookUpDataset, LookUpKeyFields and LookUpResultField. The sample code below demonstrates how to create a Lookup field at runtime using the two DBDEMOS tables 'customer.DB' and 'orders.DB'.

Example:

uses

```
Forms, Classes, Controls, StdCtrls, Db, DBTables, DBCtrls;
```

type

```
TForm1 = class(TForm)
```

```
Table1: TTable; // DBDemos customer table
```

```
Table2: TTable; // DBDemos orders table
```

```
Button1: TButton;
```

```
DBLookupComboBox1: TDBLookupComboBox;
```

```
DataSource1: TDataSource;
```

```
Table2CustNo: TFloatField; // CustNo key field object used for
```

Lookup

```
procedure Button1Click(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
public
```

```
{ Public declarations }
```

```
end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  with TStringField.Create(Table2) do begin
    FieldName := 'MyLookup';
    FieldKind:= fkLookup;
    DataSet := Table2;
    Name := DataSet.Name + FieldName;
    KeyFields:= 'CustNo';
    LookUpDataset:= Table1;
    LookUpKeyFields:= 'CustNo';
    LookUpResultField:= 'Company';
    DbLookupComboBox1.DataField:= FieldName;
    DataSource1.DataSet:= DataSet;
    Table2.FieldDefs.Add(Name, ftString, 20, false);
  end;
  DbLookupComboBox1.DataSource:= DataSource1;
  Table1.Active:= True;
  Table2.Active:= True;
end;

end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17150: 'Object not found' with STANDARD database

Question and Answer Database

FAQ2150D.txt 'Object not found' with STANDARD database

Category :Database/VCL

Platform :All

Product :Delphi 3.x

Question:

I'm getting "Object not found" error when opening a STANDARD alias with a TDatabase. What is wrong?

Answer:

Delphi 3.0 is attempting to get information on an alias parameter that does not exist on the STANDARD database alias you are trying to open. With the BDE Administrator (BDEADMIN.EXE), delete and re-create the alias.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17154: Sharing data dictionaries

Question and Answer Database

FAQ2154D.txt      Sharing data dictionaries

Category        :Database/VCL

Platform        :All

Product         :All 32 bit

Question:

How do I share data dictionaries?

Answer:

The data dictionary is merely a table created in any database. To create, right click on the DB Explorer Dictionary tab, provide a unique dictionary name, select a database from the list to create the dictionary table in, provide a unique table name (you can leave BDESDD), and description is optional. Once the dictionary is created, you can see on the dictionary table exists on the Databases tab, under the database you specified. To share, the database and table specified in dictionary creation must exist in a shareable location, such as on an SQL server. At the second location, select from the Dictionary menu, Register. Provide any name for Dictionary name (using a company standard of some sort and everyone use the same name here is recommended), select the database table that contains the previously created dictionary table, table name, and description is once again optional. In cases where database tables are qualified with an owner name, the datadictionary table will also have this same feature/limitation. The same issue will be true for each table field being described -- i.e. if the field is myownerA.field1, table myownerb.field1 will not see the attributes described.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17158: 'Your application is not enabled for use with this driver' error

Question and Answer Database

FAQ2158D.txt 'Your application is not enabled for use with this driver' error

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

What is the cause and how do I fix this error?

Answer:

This error will be raised if an application created with Delphi Standard or Delphi Professional attempts to use any SQL Link driver (InterBase, Oracle, Informix, Sybase, MS-SQL). This error can also be raised if an application created with Delphi Standard attempts to use an ODBC driver. In order to create programs that use Borland's SQL Links, you must have Delphi Client / Server and to use ODBC, you must have at least Delphi Professional.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17162: open or create a project using the TReport component

Question and Answer Database

FAQ2162D.txt open or create a project using the TReport component

Category :Database/VCL

Platform :All

Product :Delphi 3.x

Question:

How can I open or create a project using the TReport ReportSmith component. Also, can I open an existing project from Delphi 2 in Delphi 3?

Answer:

Yes. The TReport component is included in Delphi 3, but by default it is not included on the palette. If you wish to use the TReport component to create a new project you need to make it available for use on the palette. To do this choose Component | Configure Palette and move the TReport component from [Hidden] to a visible Palette (ie QReport).

NOTE: Delphi 3 does not ship with ReportSmith. You will need to install ReportSmith from your Delphi 2 CD.

NOTE: report.dcu and rsconsts.dcu are not in the search path by default. These files are located in "Delphi 3\Lib\Delphi2" directory and can either be moved to the "Delphi 3\Lib" directory or the path can be added to Delphi's search directory.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17166: Specify a DefaultExpression for a TField object

Question and Answer Database

FAQ2166D.txt Specify a DefaultExpression for a TField object

Category :Database/VCL

Platform :All

Product :Delphi 3.x

Question:

How do I specify a DefaultExpression for a TField object?

Answer:

If you set up a field attribute set and associate that set with a field in your table that will work. If you set the value in the object inspector it will let you enter in a string but it will not reflect the value at runtime. If you try to set the TField.DefaultExpression property at runtime like this:

```
MyField.DefaultExpression := 'MyValue';
```

It will compile but you will not get a default value at runtime when you create a new record in the table by, say, clicking on the + on the DBNavigator. To get the default value to take at runtime the code assignment needs to be:

```
MyField.DefaultExpression := ''MyValue'';
```

In the Object Inspector. You just need to put 'MyValue'. Use single quotes in the Object Inspector.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17170: Getting values in a TDBGrid before they are posted

Question and Answer Database

FAQ2170D.txt Getting values in a TDBGrid before they are posted

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How do I know what is being typed into the grid?

Answer:

You can see what is being typed into a TDBGrid by looking at the TInPlaceEdit control of the TDBGrid. Care should be taken to make sure that the TInPlaceEdit control is created before trying to use it. The following function will show the data in the column of the grid that is being edited:

```
procedure TForm1.DBGrid1KeyUp(Sender: TObject;
    var Key: Word; Shift: TShiftState);
var
    B: byte;
begin
    for B := 0 to DBGrid1.ControlCount - 1 do
        if DBGrid1.Controls[B] is TInPlaceEdit then
            begin
                with DBGrid1.Controls[B] as TInPlaceEdit do
                    begin
                        Labell.Caption := 'Text = ' + Text;
                    end;
            end;
    end;
```

Please reference TInPlaceEdit in the Delphi 3 help file for more information on it's functionality.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17174: Invalid class string' error during connect

Question and Answer Database

FAQ2174D.txt Invalid class string' error during connect

Category :Multitier/Remote Datasets

Platform :All

Product :Delphi 3.x

Question:

I'm specifying the ServerName for my RemoteServer, and when trying to connect I get "Invalid class string". What's wrong?

Answer:

You're not specifying the ServerGUID. You always need the ServerGUID. The ServerName, however, is optional.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17178: 'Cannot load IDAPI service library' error during connect

Question and Answer Database

FAQ2178D.txt 'Cannot load IDAPI service library' error during connect

Category :Multitier/Remote Datasets

Platform :All

Product :Delphi 3.x

Question:

I try to connect to my Remote Server and I get a "Cannot load IDAPI service library" error message on the client machine. What is wrong?

Answer:

This error is being propagated from the server where the actual problem resides. This error is a good indication that IDPROV32.DLL has not been properly installed on the server. Copy this file to the BDE directory on the server and register it through the use of a proper tool like TRegSvr.exe or RegSrv32.exe.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17182: Remote datasets for multiple clients

Question and Answer Database

FAQ2182D.txt Remote datasets for multiple clients

Category :Multitier/Remote Datasets

Platform :All

Product :Delphi 3.x

Question:

My dataset works fine when only one client connects to the dataset; What do I have to do in order to make it work for multiple clients?

Answer:

For your TSession component, set AutoSessionName to True and for your TDatabase component, set HandleShared to True.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17186: Error reading Database1.SessionName

Question and Answer Database

FAQ2186D.txt - Error reading Database1.SessionName

Category :VCL  
Platform :All-32Bit  
Product :

Question:

"Error reading Database1.SessionName. Duplicate database name 'MyDB'." error: What is wrong?

Answer:

For your TDatabase component, set HandleShared to True.

4/2/99 12:00:02 PM

Last Modified: 01-SEP-99

## Article #17190: Error reading Session1.SessionName

Question and Answer Database

FAQ2190D.txt - Error reading Session1.SessionName

Category :VCL  
Platform :All-32Bit  
Product :

Question:

"Error reading Session1.SessionName. Duplicate session name 'MySession'." error: What is wrong?

Answer:

For your TSession component, set AutoSessionName to True.

4/2/99 12:04:14 PM

Last Modified: 01-SEP-99

## Article #17194: Client application hang

Question and Answer Database

FAQ2194D.txt Client application hang

Category :Multitier/Remote Datasets

Platform :All

Product :Delphi 3.x

Question:

Client application hangs. What is wrong?

Answer:

Most likely your application server has raised an exception (on the server machine) that isn't being handled correctly. You have to add exception handling to your application server. It could also be because your server is hung for other reasons and can't service your request.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17198: 'Provider name was not recognized by the server' error

Question and Answer Database

FAQ2198D.txt 'Provider name was not recognized by the server' error

Category :Multitier/Remote Datasets

Platform :All

Product :Delphi 3.x

Question:

What is the cause and how do I fix this error?

Answer:

The problem can be caused by making changes to the server, but not rebuilding the entire project, so the EXE and TLB were out of sync. Simply rebuild the server with 'Build All'.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17202: 'Could not find interface IDataBroker in type library' error

Question and Answer Database

FAQ2202D.txt 'Could not find interface IDataBroker in type library' error

Category :Multitier/Remote Datasets

Platform :All

Product :Delphi 3.x

Question:

What is the cause and how do I fix this error?

Answer:

DataBroker is defined in the "Borland standard VCL type library", which is found in bin\stdvcl32.dll. Use REGSVR32.EXE or TREGSVR to register the type library. This may have failed in the install.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17206: Blank form at design or run time

Question and Answer Database

FAQ2206D.txt Blank form at design or run time

Category :Quick Reports

Platform :All

Product :Delphi 3.x

### Question:

Using QuickReports 2, I have a report with only a title band and some QRLabels, at design or run time I can not print or preview the form (It is blank). What is wrong?

### Answer:

Most likly, QuickReport doesn't think you any Data to print or preview. At runtime you can tell QR to print using this techinque: On the Report's OnNeedData event, add this code:

```
If Form1.tag = 0 then
begin
    MoreData := True;
    Form1.Tag := 1;
end else
    MoreData := False;
```

This tells the QR component that it got data, go ahead and print. Then the component on the next time into the event to stop - no longer needs data avoiding an infinite loop. Of course Form1's tag property is simple being used as a flag, thus can be replaced by any other flaging scheme.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17210: Saving a QuickReport custom preview to text

Question and Answer Database

FAQ2210D.txt Saving a QuickReport custom preview to text

Category :Quick Reports

Platform :All

Product :Delphi 3.x

Question:

How do I save a QuickReport custom preview to text?

Answer:

Add QREXTRA to your uses clause and the following code to your "Save button" in your custom preview:

```
procedure TfrmPreview.ToolButton3Click(Sender: TObject);
begin
    frmReport.QuickRep1.ExportToFilter
        (TQRAsciiExportFilter.Create('c:\report.txt'));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17214: TQRExpr does not recognize DisplayFormat settings

Question and Answer Database

FAQ2214D.txt TQRExpr does not recognize DisplayFormat settings

Category :Quick Reports

Platform :All

Product :Delphi 3.x

Question:

What is the easiest workaround?

Answer:

The workaround is to simply get rid of the TQRExpr component and use a TQRDBText component instead. This component honors DisplayFormat settings.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17218: Automating Word 7

Question and Answer Database

FAQ2218D.txt Automating Word 7  
Category :ActiveX/OLE/COM/ActiveForm  
Platform :All  
Product :All 32 bit

Question:  
How do I automate Word 7?

Answer:  
You can get to any of the interfaces exposed by the word automation server. These can be found by loading MSWORD8.OLB into Delphi which will display the type library information. You can also use the Application's WordBasic property to get at the VB used in Word. The following sample demonstrates both avenues:

```
implementation

uses ComObj;

{$R *.DFM}

var
  V: OleVariant;

procedure TForm1.Button1Click(Sender: TObject);
begin
  V := CreateOleObject('Word.Application');
  V.ShowMe;
  V.WordBasic.FileNew;
  V.WordBasic.Insert('test');
  V.Run('mymac');
  V.WordBasic.FileSave;
end;

end.
```

Last Modified: 01-SEP-99

## Article #17221: How to use Access97 drivers with BDE 4.51

Question and Answer Database

FAQ2221C.txt     How to use Access97 drivers with BDE 4.51

Category     :Database Issues

Platform     :All

Product      :C++Builder 3.x

Question:

How do I use the native Access drivers with the BDE 4.51 that is shipping with BCB3?

Answer:

Redeploying application with Access 97

1. In BDE Administrator open idapi.cnf from the InstallShield redist directory. Under the Configuration tab, set DLL32 to IDDA3532.DLL for the MSACCESS driver. Apply changes.

2. Copy idapi.cnf to idapi32.cnf in the InstallShield redist directory.

3. In InstallShield Groups & Files dialog, add IDDA3532.DLL to the BDE/IDAPI File Group.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #17222: New methods not showing in code insight window

Question and Answer Database

FAQ2222D.txt    New methods not showing in code insight window

Category        :ActiveX/OLE/COM/ActiveForm

Platform        :All

Product         :Delphi 3.x

### Question:

I'm writing an in-process server and I'm adding some methods to it. However, they do not show up in the code-insight window, and my code doesn't compile. What is wrong?

### Answer:

You need to either reimport the type library into the client, or just use the type library wrapper (myserver\_tlb.pas) that was created for you when you rebuilt the in-process server. If you imported the type library from the server and then modified the server, the wrapper for the server's type library will be out-dated.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17226: 'EOleSys..operation unavailable' error

Question and Answer Database

FAQ2226D.txt 'EOleSys..operation unavailable' error

Category :ActiveX/OLE/COM/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

EOleSys..operation unavailable" error using GetActiveOleObject.  
What is the cause and how do I resolve this error?

Answer:

This occurs when using a Delphi automation server, or when the automation server (such as word.basic) is not running.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: OleVariant;
begin
  V := GetActiveOleObject('Word.Basic');
  V.FileNew;
  V.Insert('test');
end;
```

GetActiveOleObject is defined in ComObj.pas. It converts the classname to a guid, and passes the guid to the Windows api call GetActiveObject.

```
function GetActiveOleObject(const ClassName: string): IDispatch;
var
  ClassID: TCLSID;
  Unknown: IUnknown;
begin
  ClassID := ProgIDToClassID(ClassName);
  OleCheck(GetActiveObject(ClassID, nil, Unknown));
  OleCheck(Unknown.QueryInterface(IDispatch, Result));
end;
```

The GetActiveOleObject uses an interface called IRunningObjectTable. We're not automatically registering to this table, so to have this functionality you must get



this interface and use it's methods to register.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17230: Access violations with VC++ created OCX controls

Question and Answer Database

FAQ2230D.txt    Access violations with VC++ created OCX controls

Category       :ActiveX/OLE/COM/ActiveForm

Platform       :All

Product        :Delphi 3.x

Question:

AV's occur when using OCX created in VC++. The same OCX works OK in Visual Basic, but not in Delphi 3. What is wrong?

Answer:

Versions of VC++ earlier than 4.2b have issues with event handling. There's a patch for VC++ on Microsoft's website.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17234: 'Error saving I(Interface)'

Question and Answer Database

FAQ2234D.txt 'Error saving I(Interface)'

Category :ActiveX/OLE/COM/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

"Error saving I(Interface): The parameter is incorrect" error:  
I'm trying to build an automation server or ActiveForm/ActiveX  
and I'm trying to save the project and getting a "Error saving  
I(Interface): The parameter is incorrect". What is wrong?

Answer:

The OLE DLLs on your machine are old. Install Internet Explorer  
off of the CD to upgrade those. Or take them from the RUNIMAGE  
directory on the Delphi 3.0 CD.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17238: 'Error 0: RLINK32 error opening file (TypeLibrary).tlb' error

Question and Answer Database

FAQ2238D.txt 'Error 0: RLINK32 error opening file (TypeLibrary).tlb' error

Category :ActiveX/OLE/COM/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

I'm trying to build an automation server or ActiveForm/ActiveX and I'm getting a "Error 0: RLINK32 error opening file my\_type\_library.tlb". What is wrong?

Answer:

The OLE DLLs on your machine are old. Install Internet Explorer off of the CD to upgrade those. Or take them from the RUNIMAGE directory on the Delphi 3.0 CD.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17242: ActiveForm not displaying in Internet Explorer

Question and Answer Database

FAQ2242D.txt    ActiveForm not displaying in Internet Explorer  
Category        :ActiveX/OLE/COM/ActiveForm  
Platform        :All  
Product         :Delphi 3.x

Question:

Why do ActiveX or ActiveForm sometimes not display in Internet Explorer? All that shows up is the .htm page with a blank square with a little red X in it.

Answer:

Its likely that you selected the licensing option when you created the ActiveForm and you are not deploying the .LIC file with your .OCX file.

Usually licensing is not used with ActiveForm/ActiveXs that are being used to enhance a web site because you are giving the control away for free.

To disable the design time licensing locate the initialization section in your ActiveForm's XXXImpl file and replace the second to last parameter of the TActiveXControlFactory.Create call to an empty string like so:

```
initialization
  TActiveXControlFactory.Create(
    ComServer,
    TAnimateX,
    TAnimate,
    Class_AnimateX,
    1,
    '',      0);
end.
```

So, when should you use Design-Time Licensing?

Your control should use design-time only when you want to sell the ActiveX or ActiveForm to other developers who will then sell

their finished applications to end-users. That is, the control works in a design-time environment (ie, Delphi, C++Builder, VB, etc.) when the LIC file is present, but it only works in a run-time environment (ie, the end-user application) when the .LIC file is not present.

If you're deploying your ActiveX over the web, then you are by definition deploying to end-users (as opposed to developers) and you should never require a design-time license.

Also, if the ActiveForm is unsigned, the Internet Explorer must have its "Active content security" level set to medium. To do this, go to the control panel and select Internet. Go to the security tab and then press the Safety Level button. Make sure the level is set to medium.

NOTE: This is only advised when designing your own controls. Potentially rogue ActiveX controls could damage data on the machine!

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17245: Keeping the Local IBServer from starting up automatically

Question and Answer Database

FAQ2245C.txt Keeping the Local IBServer from starting up automatically

Category :Database Issues

Platform :All

Product :C++Builder 1.x

Question:

How can I keep the Local InterBase Server from automatically starting when I start Windows?

Answer:

There are three methods that you can use:

- 1) Right click on the Local InterBase Server icon and select "Startup Configuration" then select "Manual Startup".
- 2) Try taking the Local InterBase Server out of you Startup Menu, or comment it from the run/load lines of your Win.ini file.
- 3) Run REGEDIT.EXE, find the key:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

and remove manually the IBSERVER entry.

5/13/98 10:10:09 AM

Last Modified: 01-SEP-99

## Article #17246: How can I get serial number of my drive

Question and Answer Database

FAQ2246D.txt    How can I get serial number of my drive

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I get volume information, such as the serial number, from my drivers?

Answer:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  VolumeName,
  FileSystemName        : array [0..MAX_PATH-1] of Char;
  VolumeSerialNo        : DWord;
  MaxComponentLength,
  FileSystemFlags       : Integer;
begin
  GetVolumeInformation('C:\',VolumeName,MAX_PATH,@VolumeSerialNo,
    MaxComponentLength,FileSystemFlags,
    FileSystemName,MAX_PATH);
  Mem1.Lines.Add('VName = '+VolumeName);
  Mem1.Lines.Add('SerialNo = $'+IntToHex(VolumeSerialNo,8));
  Mem1.Lines.Add('CompLen = '+IntToStr(MaxComponentLength));
  Mem1.Lines.Add('Flags = $'+IntToHex(FileSystemFlags,4));
  Mem1.Lines.Add('FSName = '+FileSystemName);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17250: 'TActiveFormX declaration missing or incorrect' error

Question and Answer Database

FAQ2250D.txt 'TActiveFormX declaration missing or incorrect' error

Category :ActiveX/OLE/COM/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

What does this error mean and how do I solve the problem?

Answer:

This is usually caused by changing the name of an ActiveForm in the incorrect order (see README.TXT). If the name of the CoClass is changed first and then does a refresh, an AV will occur.

Proceeding to use the Object Inspector to change the name of the form will cause "TActiveFormX declaration missing or incorrect".

To fix this he will need to open the .DFM file and change the line:

```
object ActiveFormX: TActiveFormX
```

```
to,
```

```
object MyForm: TMyForm
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17254: Missing ProviderName in TClientDataSet

Question and Answer Database

FAQ2254D.txt - Missing ProviderName in TClientDataSet

Category :DCOM/MultiTier/ActiveForm  
Platform :All-32Bit  
Product :

Question:

I don't see entries in the TClientDataSet ProviderName property.  
What's wrong?

Answer:

On your server application, in the remote data module, right click your dataset (TTable, Tquery, etc) and select "Export [MyDataset] from data module". Rebuild the server application. You should now see the provider supplied in the list of the TClientDataSet.

"Exporting from data module" adds a "Get\_MyDataset" function to the protected section of the remote datamodule's CoClass that returns an IProvider interface.

4/2/99 1:03:04 PM

Last Modified: 01-SEP-99

## Article #17258: Propagating constraints from middle-tier to client

Question and Answer Database

FAQ2258D.txt - Propagating constraints from middle-tier to client

Category :Multi-tier  
Platform :All-32Bit  
Product :

Question:

How do I propagate constraints from Middle-tier to Client?

Answer:

This example uses the demos found in  
\Delphi 3\Demos\DB\CLIENTDS\Empedit.dpr and \server.dpr.  
Empedit.dpr represents the client tier, server.dpr runs on  
the middle tier, and the database, DBDEMOS, is located also on  
the middle tier, but in a real world scenario would be an SQL  
database running on yet a third machine. This example uses the  
Employee.db table. In order to use the following, you will need  
to set up DCOM and have the two examples running before adding  
the complication of constraints to the mix.

The general steps are:

Define constraints in the data dictionary  
Associate the data dictionary attributes to a TField  
Access the TField from a ClientDataSet.

First, define an attribute set in the data dictionary that  
includes a constraint. For instance, an attribute set called  
EmpNoA could have a CustomConstraint of "EmpNo > 100" and a  
ConstraintErrorMessage of "EmpNo is less than 100". Define  
static TFields in Server.dpr. Bring up the EmpServer datamodule  
and double click EmpQuery to bring up the fields editor. In the  
fields editor, right-click the EmpNo TField, select Associate  
Attributes, and select EmpNoA from the Associate Attributes  
dialog. Now when selecting the EmpNo TField in the fields  
editor you should see CustomConstraint and  
ConstraintErrorMessage matching the data dictionary entry. Open  
EmpEdit.dpr and add a TDBEdit to EmployeeForm. Set the

DataField property to EmpNo and the DataSource property to EmpData (the TClientDataSet). Compile and run Server.dpr on the middle tier machine, then on the client machine run the EmpEdit example application. Attempt to post a record where EmpNo is greater than 100; your custom error message will appear.

4/2/99 1:29:56 PM

Last Modified: 01-SEP-99

## Article #17262: 'DAX Error: Unexpected failure.' error

Question and Answer Database

FAQ2262D.txt 'DAX Error: Unexpected failure.' error  
Category :ActiveX/OLE/COM/ActiveForm  
Platform :All  
Product :Delphi 3.x

Question:

How do I resolve this error?

Answer:

This usually happens when the application server has raised an exception during initialization. Add exception handling to your server.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17266: Determining Drive Type

Question and Answer Database

FAQ2266D.txt Determining Drive Type

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can my application tell if a given drive is a CD, Network, Ramdisk, or Removable drive?

Answer:

Use the Windows API function `GetDriveType()`.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  case GetDriveType('C:\') of
    0          : ShowMessage('The drive type cannot be
determined');
    1          : ShowMessage('The root directory does not exist');
    DRIVE_REMOVABLE: ShowMessage('The disk can be removed');
    DRIVE_FIXED   : ShowMessage('The disk cannot be removed');
    DRIVE_REMOTE  : ShowMessage('The drive is remote (network
drive');
    DRIVE_CDROM   : ShowMessage('The drive is a CD-ROM drive');
    DRIVE_RAMDISK : ShowMessage('The drive is a RAM disk');
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17270: Client launching

Question and Answer Database

FAQ2270D.txt - Client launching

Category :ActiveX/OLE/COM/ActiveForm  
Platform :All-32Bit  
Product :

### Question:

What do I need to do to make my client launch and access my server through DCOM?

### Answer:

One setup that usually work is if you install Service Pack 3 on all NT 4.0 machines. If you are going to use any Win95 clients then you'll have to install DCOM for Win95 on those machines. You'll also need Microsoft Internet Explorer 3.02 on the client machines. On the machine were the application server is going to run (an NT 4.0 machine) you'll do each of the following:

Enable the Guest account. This is disabled by default.  
Make sure the application server is on a local drive on the machine.  
Run the application server once to register it.  
In DCOM Config (dcomcnfg.exe) for the specific server object:

On the Identity tab, select "The interactive user".  
On the Security tab, select Custom Access Permissions, and make sure Everyone is added and allowed access.  
On the Security tab, select Custom Launch Permissions, and make sure Everyone is added and allowed to launch.

4/2/99 12:08:31 PM

Last Modified: 01-SEP-99

## Article #17274: 'DAX Error: Server execution failed' error

Question and Answer Database

FAQ2274D.txt 'DAX Error: Server execution failed' error

Category :ActiveX/OLE/COM/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

What am I doing wrong and how do I resolve this error?

Answer:

Make sure you followed the instructions in FAQ "Client launching" in this section. If you did and it still doesn't work, try running the server once manually.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17278: 'DAX Error: Access is denied' error

Question and Answer Database

FAQ2278D.txt 'DAX Error: Access is denied' error

Category :ActiveX/OLE/COM/ActiveForm

Platform :All

Product :Delphi 3.x

Question:

What am I doing wrong and how do I resolve this error?

Answer:

Make sure you followed the instructions in "What do I need to do to make my client launch and access my server through DCOM?" FAQ. If you did and it still doesn't work, try running the server once manually.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17282: Old ActiveForm information displayed

Question and Answer Database

FAQ2282D.txt - Old ActiveForm information displayed

Category :ActiveX/OLE/COM/ActiveForm  
Platform :All-32Bit  
Product :

### Question:

I updated my ActiveForm and I hit my web page in Internet Explorer, but I get the old ActiveForm. What am I doing wrong?

### Answer:

Even if you use version info, you'll see this happening in some cases. Usually it goes away if you unregister the OCX (in C:\WINDOWS\OCCACHE\ or C:\WINNT\OCCACHE) using REGSVR32.EXE.

The recommended way of doing this is:

Exit Internet Explorer.

Run "regsvr32.exe /u c:\windows\occache\MyActiveX.ocx"

Restart Internet Explorer and hit the web page again.

If you deploy extra files that you ActiveX is using you might be helped by knowing that all the stuff is being pulled down to your OCCACHE directory and if Internet Explorer determines that there is a conflict - it'll put some files in OCCACHE\CONFLICT.x (x is a number) directories.

4/2/99 12:08:51 PM

Last Modified: 01-SEP-99

## Article #17286: What directory is the ActiveForm in

Question and Answer Database

FAQ2286D.txt - What directory is the ActiveForm in

Category :ActiveX/OLE/COM/ActiveForm  
Platform :All-32Bit  
Product :

Question:

I need to know the exact directory that my ActiveForm is located in; How do I get that information?

Answer:

GetModuleFilename(HInstance, ModuleName, SizeOf(ModuleName)).  
ModuleName can be declared as an array [0..255] of Char for instance. For your information:

Application.ExeName gives you the path to IEXPLORE.EXE.  
GetCurrentDirectory usually gives you C:\Windows\Desktop.

4/2/99 12:09:11 PM

Last Modified: 01-SEP-99

## Article #17290: How do I launch the Control Panel (and its applets)

Question and Answer Database

FAQ2290D.txt    How do I launch the Control Panel (and its applets)  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

Question:

How do I launch the Control Panel (and its applets)?

Answer:

This will launch the control panel itself:

```
WinExec('rundll32 shell32.dll,Control_RunDLL',SW_SHOW);
```

"Display Properties" - Background:

```
WinExec('rundll32 shell32.dll,Control_RunDLL desk.cpl,,0',SW_SHOW);
```

"Display Properties" - Screen Saver:

```
WinExec('rundll32 shell32.dll,Control_RunDLL desk.cpl,,1',SW_SHOW);
```

"Display Properties" - Appearance:

```
WinExec('rundll32 shell32.dll,Control_RunDLL desk.cpl,,2',SW_SHOW);
```

"Display Properties" - Settings:

```
WinExec('rundll32 shell32.dll,Control_RunDLL desk.cpl,,3',SW_SHOW);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17294: 'Invalid filename' error

Question and Answer Database

FAQ: FAQ2294D - 'Invalid filename' error

Category: Internet/WEB

Platform: All-32Bit

Product: All32Bit,

Question:

When a client hits my ISAPI DLL it says "Invalid filename". My Paradox/dBASE tables are on a Novell server. What's wrong?

Answer:

You're missing the drive mapping for your IUSR\_XXX account. Your tables are on the Novell server, so you need a drive mapping to get there.

Background information:

Your webserver is running IIS as a service. Nobody is logged in interactively.

A client hits the web page with the ISAPI DLL. IIS is logging you in on the server as IUSR\_XXX. In order for the ISAPI DLL to see your tables on the Novell server, you will need the drive mapping for that user.

6/28/99 2:26:43 PM

Last Modified: 01-SEP-99

## Article #17298: 'Invalid configuration parameter for alias

Question and Answer Database

FAQ: FAQ2298D - 'Invalid configuration parameter for alias

Category: Internet/WEB

Platform: All-32Bit

Product: All32Bit,

Question:

"Invalid configuration parameter for alias {alias\_name}"  
error with ODBC: I get this error when trying to access a  
database thru an ODBC DSN connection on the ISAPI/NSAPI server.  
What am I doing wrong?

Answer:

In order for ODBC to be used as a service and to be accessed by  
all users (IIS guest account), you need to create a non-standard  
SYSTEM DSN, not the default USER DSN.

6/28/99 2:26:59 PM

Last Modified: 01-SEP-99

## Article #17302: Conditional defines for all Delphi versions

Question and Answer Database

FAQ2302D.txt - Conditional defines for all Delphi versions

Category :Compiler  
Platform :All-32Bit  
Product :All32Bit,

Question:

What are the conditional defines for Delphi

Answer:

Delphi defines the following standard conditional symbols.

VER80 - Delphi 1.x  
VER90 - Delphi 2.x  
VER93 - CBuilder 1.0  
VER100 - Delphi 3.x

For example the following would compile in the correct uses statement in Delphi 2 and Delphi 3 respectively.

```
{$IFDEF VER90}  
uses system, windows, oleaut;  
{$ENDIF}
```

```
{$IFDEF VER100}  
uses system, windows, comobj;  
{$ENDIF}
```

The version numbers for Delphi 1, 2 and 3 correspond to version 8.0, 9.0 and 10.0 of the Object Pascal compiler. For additional information on conditional compiles and compiler directives see appendix B in the Delphi 3 users guide.

4/2/99 12:10:06 PM

Last Modified: 01-SEP-99

## Article #17306: Launching a web browser

Question and Answer Database

FAQ2306D.txt - Launching a web browser

Category :Internet/WEB  
Platform :All-32Bit  
Product :

Question:

How can I launch the computer's current web browser with a specific web site URL?

Answer:

Call the Win32 API function called HlinkNavigateString which is declared in the UrlMon unit.

Example call:

```
HlinkNavigateString(nil, 'http://www.borland.com');
```

If you're calling this from an ActiveForm then call it like this:

```
HlinkNavigateString(ComObject, 'http://www.borland.com');
```

4/2/99 11:47:47 AM

Last Modified: 01-SEP-99



## Article #17310: TClientSocket and TServerSocket

Question and Answer Database

FAQ2310D.txt - TClientSocket and TServerSocket

Category :Internet/WEB  
Platform :All-32Bit  
Product :

Question:

Why don't the Sockets receive more than 8k worth of data?

Answer:

Because the IP layer breaks a stream of data into 8k chunks, the developer must explicitly insert a length integer at the beginning of the stream telling the receiving end how much data to expect, more importantly, how many 8k packets to receive.

Because the Socket components are just wrappers for the WinSock and not Protocols with header information (as mentioned above), the developer must concern themselves with said "padding" of data. This requires:

NOTE: The implementation can vary by the developer so this will remain generic.

padding the data at the source with a length integer,  
extracting the length integer at the destination,  
counting the number of bytes received at the destination  
and matching those with the length integer.

If TotalBytesReceived <> LengthInteger then know the next packet is a continuation, otherwise know the next packet is a different stream altogether.

NOTE: This is handled in HTTP & FTP protocols via header information (ie, 'Content\Length:').

4/2/99 11:47:55 AM

Last Modified: 01-SEP-99

## Article #17314: Installing old Delphi 2 components

Question and Answer Database

FAQ2314D.txt - Installing old Delphi 2 components

Category :Miscellaneous

Platform :All-32Bit

Product :

Question:

When installing a component into a package in Delphi 3, why is Delphi 3 complaining about not finding an associated PAS file? All I should need is the DCU file.

Answer:

That's true only if the component was compiled using Delphi 3. If it's a Delphi 2 DCU file, it will tell you "Fatal Error" and complain about not finding an associated PAS file. In order to install such a component, you will need the source code and recompile the PAS file.

There are two solutions to the problem:

Have the vendor of the component re-compile the component with Delphi 3.

If you have the source to the component, re-compile the component with Delphi 3 and use the resultant DCU.

4/2/99 12:11:03 PM

Last Modified: 01-SEP-99

## Article #17318: Rebuilding components

Question and Answer Database

FAQ2318D.txt Rebuilding components

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

There is no longer the option to do Component/ Rebuild Library. If I make changes to the source code of my component, how do get the changes to take effect?

Answer:

All components now reside in packages, to see the changes, start the package editor, and do a compile. The package editor can be started by doing a file/open, and changing the "File type" to be "Delphi Package Source"

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17322: Using Visual Basic arrays in Delphi

Question and Answer Database

FAQ2322D.txt - Using Visual Basic arrays in Delphi

Category :Miscellaneous  
Platform :All-32Bit  
Product :

Question:

How do I pass arrays from VB to Delphi?

Answer:

Arrays can be passed as variants:

VB module code:

```
Attribute VB_Name = "Module1"  
Declare Function TestMin Lib "c:\windows\system\NoelSArr"  
    (Nums As Variant) As Integer
```

VB form code:

```
Dim A As Variant  
Private Sub Command1_Click()  
    A = Array(4, 3)  
    MsgBox (TestMin(A))  
End Sub
```

Delphi DLL code:

```
library NoelSArray;  
.  
.  
function TestMin(const Nums: Variant): integer; export; stdcall;  
var  
    p1: Variant;  
begin  
    p1 := VarArrayCreate([0, 1], VT_I4);  
    p1:= Nums;  
    if (p1[0] < p1[1]) then  
        result:= p1[0]  
    else  
        Result:= p1[1];
```

end;

4/2/99 12:11:27 PM

Last Modified: 01-SEP-99

## Article #17326: Specify an overlay image for a node in a treeview

Question and Answer Database

FAQ2326D.txt Specify an overlay image for a node in a treeview

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

How do I specify an overlay image for a node in a treeview?

Answer:

You have to call TImageList.Overlay first. Example:

```
ImageList1.Overlay(Image_Index, Overlay_Index);
```

```
TreeView1.Items[Item_Index].OverlayIndex := Overlay_Index;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17330: Opening an application's help file

Question and Answer Database

FAQ2330D.txt    Opening an application's help file

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

How do I bring up the index in my application's help file?

Answer:

Application.HelpCommand(HELP\_FINDER,0). For further info see help on the Windows API call WinHelp.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17334: TToolBar and TSpeedButton misaligned

Question and Answer Database

FAQ2334D.txt - TToolBar and TSpeedButton misaligned

Category :Miscellaneous  
Platform :All-32Bit  
Product :Delphi3.x, Delphi4.x,

### Question:

When I use TSpeedButtons with the Flat property set to True on a TToolBar they look misaligned and strange. What's wrong?

### Answer:

You probably installed IE4, which has a new comctl32.dll. If you use NT3.51 don't set Flat to True, if you use Win95 or NT4.0 use TToolButtons instead of TSpeedButtons.

4/2/99 12:22:11 PM

Last Modified: 01-SEP-99



## Article #17338: Intel's Lan Desk management Suite causes lockup

Question and Answer Database

FAQ2338D.txt Intel's Lan Desk management Suite causes lockup

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

Intel's Lan Desk management Suite causes lockup with debugger:  
Why does the internal debugger cause PC to lock up when  
hitting a breakpoint?

Answer:

Intel's Lan Desk management Suite (v2.0.0), a PC Anywhere-like product used by Help Desk organizations to view help desk customer PCs, causes WUSER.EXE (v4.01) to run. This hasn't been verified, but WUSER may be using an interrupt usually reserved for debugging.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17341: Where is CERTINST.DOC ?

Question and Answer Database

FAQ2341C.txt    Where is CERTINST.DOC ?

Category       :Database Issues

Platform        :All

Product         :C++Builder 3.x

Question:

Where is CERTINST.DOC?

DEPLOY.TXT says it is on the web.

Answer:

This file is elusive. Usually you won't find it. In any case, in general, the only certified install programs are Install Shield Express and apparently, occasionally, Wyse Technology's install program.

Only these two are BDE aware.

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99

## Article #17342: Declaring a SafeArray in the type library editor

Question and Answer Database

FAQ2342D.txt Declaring a SafeArray in the type library editor

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

How do I declare a SafeArray in the type library editor?

Answer:

You must specify the component type, for example "procedure MyProc(Items: TSafeArray(Integer));" specifies a SafeArray with a component type of Integer. See page 42-10 in the Developer's Guide for more info.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17346: Setting a transparent background color for TImageList

Question and Answer Database

FAQ2346D.txt    Setting a transparent background color for TImageList  
Category        :Miscellaneous  
Platform        :All  
Product         :Delphi 3.x

Question:

How do I set the transparent color in an TImageList so the background color doesn't show?

Answer:

Use the TImageList.AddMasked(Bitmap: TBitmap; MaskColor: TColor) method. You can either pass the transparent color yourself in the second parameter or clDefault to let the imagelist take the bottom-left pixel's color. TSpeedButton and TBitBtn use this method. Check-out buttons.pas (around line 560) to see an example.

7/16/98 4:31:28 PM

Last Modified: 25-FEB-00

## Article #17350: DBGrid fixed leftmost column

Question and Answer Database

FAQ2350D.txt DBGrid fixed leftmost column

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

How do I make the leftmost column in my DBGrid fixed, so it stays when I scroll left and right in the grid?

Answer:

Declare a type THack = class(TDBGrid). Then in run-time, do THack(DBGrid1).FixedCols := 2. Of course, you'd have to set it to 1 if you exclude dgIndicator from the Options in the grid component.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17354: Component palette shown incorrectly - Matrox Millenium card

Question and Answer Database

FAQ2354D.txt - Component palette shown incorrectly - Matrox Millenium card

Category :Install Issues  
Platform :All-32Bit  
Product :

Question:

How do I solve the problem?

Answer:

This problem has occurred on a few Matrox Millennium boards and Diamond Stealth 3D 2000 boards. The workaround is to try a different version of the vendor's video driver or try to use Windows' version (even VGA/SVGA) of the video driver.

One solution for the Matrox Millennium board may be the following:

download the latest version of video-driver  
(v3.70 as of October 3rd)  
do normal installation  
deactivate the "Use device bitmap caching" in the  
"MGA PowerDesk Settings"

4/2/99 12:17:27 PM

Last Modified: 01-SEP-99

## Article #17358: Using FindFirst to search for files.

Question and Answer Database

FAQ2358D.txt Using FindFirst to search for files.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I handle the memory loss when I use the example for FindFirst to search for files?

Answer:

You need to qualify calls to FindFirst, FindNext, and FindClose to include the SysUtils unit.

Example:

```
begin
  Result := SysUtils.FindFirst(Path, Attr, SearchRec);
  while Result = 0 do
  begin
    ProcessSearchRec(SearchRec);
    Result := SysUtils.FindNext(SearchRec);
  end;
  SysUtils.FindClose(SearchRec);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17362: Getting an handle to a window in another application.

Question and Answer Database

FAQ2362D.txt Getting an handle to a window in another application.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get the handle of another application's window and make that window the active window?

Answer:

Using the Windows API function `FindWindow()` is the simplest and most straightforward method of locating a given window provided you know the exact window caption or class name. If you know only part of the window's caption (ie: 'Netscape - ' + 'Some Unknown URL', you need to use the Windows `EnumWindows()` function to enumerate all the active windows, then call the Windows API functions `GetWindowText()` and `GetClassName` to see if you find a match for the window that you are searching for.

The following example returns the window handle of the first window enumerated that contains a partial match of the window title and an exact match of the window's `ClassName` (if given) and brings that window to the foreground.

```
type
  PFindWindowStruct = ^TFindWindowStruct;
  TFindWindowStruct = record
    Caption : string;
    ClassName : string;
    WindowHandle : THandle;
  end;

function EnumWindowsProc(hWindow : hWnd;
                        lParam : LongInt) : Bool
{$IFDEF Win32} stdcall; {$ELSE} ; export; {$ENDIF}
var
  lpBuffer : PChar;
```



```

WindowCaptionFound : bool;
ClassNameFound : bool;

begin
  GetMem(lpBuffer, 255);
  Result := True;
  WindowCaptionFound := False;
  ClassNameFound := False;

  try
    if GetWindowText(hWnd, lpBuffer, 255) > 0 then
      if Pos(PFindWindowStruct(lParam).Caption, StrPas(lpBuffer)) > 0
        then WindowCaptionFound := true;

    if PFindWindowStruct(lParam).ClassName = '' then
      ClassNameFound := True else
      if GetClassName(hWnd, lpBuffer, 255) > 0 then
        if Pos(PFindWindowStruct(lParam).ClassName, StrPas
(lpBuffer))
          > 0 then ClassNameFound := True;

    if (WindowCaptionFound and ClassNameFound) then begin
      PFindWindowStruct(lParam).WindowHandle := hWnd;
      Result := False;
    end;

  finally
    FreeMem(lpBuffer, sizeof(lpBuffer^));
  end;
end;

function FindAWindow(Caption : string;
                    ClassName : string) : THandle;
var
  WindowInfo : TFindWindowStruct;

begin
  with WindowInfo do begin
    Caption := Caption;
    ClassName := ClassName;
    WindowHandle := 0;
    EnumWindows(@EnumWindowsProc, LongInt(@WindowInfo));
    FindAWindow := WindowHandle;
  end;
end;

```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TheWindowHandle : THandle;
begin
  TheWindowHandle := FindAWindow('Netscape - ', '');
  if TheWindowHandle = 0 then
    ShowMessage('Window Not Found!') else
    BringWindowToTop(TheWindowHandle);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17366: Using an AVI file with TAnimate

Question and Answer Database

FAQ2366D.txt - Using an AVI file with TAnimate

Category :Miscellaneous  
Platform :All-32Bit  
Product :

Question:

How can I resource an AVI so the TAnimate control doesn't have to rely upon an external file?

Answer:

There is an AVI Type, which is not used often, called 'AVI' and if you create a .RES file, Resource file, with this type it will allow you to use the resource as an AVI.

The steps are:

Create a new RES file. (Resource Workshop is good but it must be a 32-bit resource file)  
Add the AVI to the RES Project as a New Type specified as 'AVI'. (ie File | "Add to project..." in Resource Workshop and followed by clicking on the "New Type" button)

NOTE:Make sure to edit the Name of the Resource (from 'AVI\_1') to a more meaningful name before saving the project. (maybe 'About') - case sensitive.

Save the RES Project to a name (ie RORY.RES)  
Back in Delphi 3, add the RES file in with the \$R Compiler directive (ie {\$R 'C:\AVI\RORY.RES' } )  
Programmatically, set the TAnimate's RESName property to the Resource name. (Not the Type; AVI)

Example:

```
Animat1.RESName := 'About';  
Animat1.Active := True;
```

4/2/99 12:14:28 PM

Last Modified: 01-SEP-99

## Article #17370: Migrating a project from Delphi 1 or 2 to Delphi 3

Question and Answer Database

FAQ2370D.txt Migrating a project from Delphi 1 or 2 to Delphi 3

Category :Miscellaneous

Platform :All

Product :Delphi 3.x

Question:

What files should I move when migrating a project from Delphi 1 or 2 to Delphi 3?

Answer:

The files required to move a project from Delphi 1 or Delphi 2 to Delphi 3 are just the DFM, PAS, and DPR. It is also important to move any resource files (.res), that have custom resources like cursors, icons, strings, etc. All the other files are going to be recreated by the compiler when a build all is done inside of Delphi 3. It is especially important not to bring over the DCU files as the format changes from version to version. You may get unknown identifier errors or other strange behaviors if you try to bring over DCU files.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17378: Checking drive ready status.

Question and Answer Database

FAQ2378D.txt    Checking drive ready status.

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I check to see if a drive is ready without triggering a Windows error dialog box?

Answer:

You can use the Windows API function SetErrorMode() to suppress the Window's critical Error dialog.

Example:

```
function IsDriveReady(DriveLetter : char) : bool;
var
  OldErrorMode : Word;
  OldDirectory : string;
begin
  OldErrorMode := SetErrorMode(SEM_NOOPENFILEERRORBOX);
  GetDir(0, OldDirectory);
  {$I-}
  ChDir(DriveLetter + ':\');
  {$I+}
  if IoResult <> 0 then
    Result := False
  else
    Result := True;

  ChDir(OldDirectory);
  SetErrorMode(OldErrorMode);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if not IsDriveReady('A') then
```

```
ShowMessage('Drive Not Ready') else  
  ShowMessage('Drive is Ready');  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17382: External function failure when passing boolean parms.

Question and Answer Database

FAQ2382D.txt External function failure when passing boolean parms.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

Why do some external functions fail when passed a boolean parameter of True?

Answer:

In Delphi 3, the value "True" for data types ByteBool, WordBool, and LongBool is now represented as -1 for compatibility with Microsoft Visual Basic. Note that many compilers represent "True" as either non-zero or a positive 1. When passing values to non Visual Basic applications, you should consider using the following technique to avoid incompatibilities:

```
LongBool(Abs(True));
```

When accepting boolean values from any external source, you should always test against the value "False". The following technique should always work, since the value for false is always zero:

```
if BoolValPassed <> False then DoSomething.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17386: No default printer selected error message.

Question and Answer Database

FAQ2386D.txt - No default printer selected error message.

Category :Printing  
Platform :All-32Bit  
Product :

Question:

What is the cause of the error message: 'No default printer selected'?

Answer:

This message may be caused by running an older version of the IBM AS400 emulation software. An update from IBM generally fixes the problem.

4/2/99 11:24:54 AM

Last Modified: 01-SEP-99

## Article #17390: Creating a non-VCL console project.

Question and Answer Database

FAQ2390D.txt     Creating a non-VCL console project.

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I create a small footprint, non-VCL, non-console project?

Answer:

Create a new application, then, from the View... Menu option, start the "Project Manager", and remove all units from the project. Then from the View... menu option, select "Project Source" and edit it so that it looks like the below example. Note: You can add and use additional units, so long as you do not use any VCL components.

Example:

```
program Project1;

{$R *.RES}

uses SysUtils;

var
  f : TextFile;

begin
  AssignFile(f, 'TestFile.Txt');
  ReWrite(f);
  Writeln(f, 'Test');
  Close(f);
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17394: Printing to file.

Question and Answer Database

FAQ2394D.txt - Printing to file.

Category :Printing  
Platform :All  
Product :

Question:

How do I print to a file without changing the printer's setup, or getting a prompt for a file to print to?

Answer:

To do this you simply need to change the initialization of the DocInfo structure in TPrinter from:

```
lpszOutput := nil;
```

to:

```
lpszOutput := 'C:\SomePath\SomeFileName';
```

You can make a permanent change to TPrinter, but for the ultimate in flexibility, you can create an interface unit to make the filename easy to change from your application.

{ Note: Modifications to the VCL are made at the developers risk and

any effects caused by modifications are not the responsibility of Borland International. }

The safest way to make changes to the VCL is to create a new directory called "patched", and copy the Printers.Pas unit to the new directory, then make your changes (with comments) to the copied file. The you must add the path to your patched directory to the beginning of your library path, then restart Delphi/C++ Builder, and rebuild your project after making the following changes. Note that the library path setting can be found off the main menu of:

Delphi 1 : Options... Environment... Library

Delphi 2 : Tools... Options... Library  
Delphi 3 : Tools... Environment Options... Library  
C++ Builder : Options... Environment... Library

Instructions for the patch:

1) Create a new unit in the patched directory called "PrnPage.pas" and include the following code in the unit:

```
unit PrnPage;  
  
interface  
  
var FileNameToPrintTo : Pointer;  
  
implementation  
  
begin  
    FileNameToPrintTo := nil;  
end.
```

2) Now open the Printers.Pas file in your patched directory. Search for the line that reads "uses Consts;" (or in Delphi 16 it will read "uses WinProcs, Consts;");

and add the PrnPage unit to the uses clause:

```
uses Consts, PrnPage;
```

3) Now search for line that reads "lpszOutput := nil;"

and change it to read: lpszOutput := FileNameToPrintTo;

4) Save your changes, and restart Delphi / C++ Builder.

5) To test your changes, try the following code. Note: Be sure to use the "Build All" option when compiling your project for the first time.

```
uses Printers, PrnPage;
```

```
{$IFDEF WIN32}
```

```
    const MAX_PATH = 144;
```

```
{$ENDIF}
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  lpPrnFile : pChar;
begin
  GetMem(lpPrnFile, MAX_PATH +1);

  try
    StrPCopy(lpPrnFile, 'C:\Download\TestIt.Prn');
    FileNameToPrintTo := @lpPrnFile^;

    with Printer do begin
      BeginDoc;
      Canvas.TextOut(10, 10, 'Test');
      EndDoc;
    end;

    FileNameToPrintTo := nil;

  finally
    FreeMem(lpPrnFile, MAX_PATH +1);
  end;
end;
```

4/2/99 11:25:03 AM

Last Modified: 01-SEP-99

## Article #17398: Getting the longfilename from a file.

Question and Answer Database

FAQ2398D.txt Getting the longfilename from a file.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get the long file name of a file or directory from the short file name?

Answer:

You can use the Win32\_Find\_Data member of TSearchRec to extract this information.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SearchRec : TSearchRec;
  Success : integer;
begin
  Success := SysUtils.FindFirst('C:\Download\dial-u~1.htm',
                                faAnyFile,
                                SearchRec);

  if Success = 0 then begin
    ShowMessage(SearchRec.FindData.CFileName);
  end;
  SysUtils.FindClose(SearchRec);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17406: Getting the filename and path of a local table.

Question and Answer Database

FAQ2406D.txt Getting the filename and path of a local table.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I get the full filename (including path) of a local table?

Answer:

The following example was adapted from the BDE32.HLP file, located in the Borland/CommonFiles/BDE directory:

implementation

```
{ $R *.DFM }
```

```
uses DbiTypes, DbiProcs;
```

```
function fDbiFormFullName(Tbl: TTable): String;
```

```
var
```

```
  Props: CurProps;
```

```
  Buffer1 : array[0..DBIMAXPATHLEN] of char;
```

```
  Buffer2 : array[0..DBIMAXPATHLEN] of char;
```

```
begin
```

```
  Check(DbiGetCursorProps(Tbl.Handle, Props));
```

```
  StrPCopy(Buffer1, Tbl.TableName);
```

```
  Check(DbiFormFullName(Tbl.DBHandle,
```

```
    @Buffer1,
```

```
    Props.szTableType,
```

```
    @Buffer2));
```

```
  Result := StrPas(Buffer2);
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
  Mem1.Lines.Add(fDbiFormFullName(Table1));
```

```
end;
```

Notes:

Table\_You\_Are\_Using.Active Must be True.  
Works on Local Tables.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17410: Temporarily turning off range checking.

Question and Answer Database

FAQ2410D.txt Temporarily turning off range checking.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I temporarily turn off range checking for a block of code, then turn it back on if it was originally enabled ?

Answer:

You can do this in code by using "IFOPT" and "DEFINE".

type

```
  PSomeArray = ^TSomeArray;
```

```
  TSomeArray = array[0..0] of integer;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  p : PSomeArray;
```

```
  i : integer;
```

```
begin
```

```
{ $IFOPT R+ }
```

```
  { $DEFINE CKRANGE }
```

```
  { $R- }
```

```
{ $ENDIF }
```

```
  GetMem(p, sizeof(integer) * 200);
```

```
  try
```

```
    for i := 1 to 200 do
```

```
      p[i] := i;
```

```
  finally
```

```
    FreeMem(p, sizeof(integer) * 200);
```

```
  end;
```

```
{ $IFDEF CKRANGE }
```

```
  { $UNDEF CKRANGE }
```

```
{ $R+ }  
{ $ENDIF }  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17414: Getting the taskbar handle.

Question and Answer Database

FAQ2414D.txt    Getting the taskbar handle.

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I obtain Windows Taskbar handle?

Answer:

```
hTaskbar := FindWindow('Shell_TrayWnd', Nil );
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17418: Why am I having trouble displaying and/or printing a very large bitmap?

Question and Answer Database

FAQ2418D.txt Handling bitmap display.

Category :Windows API

Platform :All

Product :All 32 bit

### Question:

I created a very large bitmap image that refuses to display or print on some systems. What can I do?

### Answer:

There are a number of factors that can affect the size of the largest displayable bitmap. Some versions of Windows can only work with a Bitmap that is no larger than 1.75 megabytes in size. The display driver can also play an important role, even during printing, as the printer driver may rely on the display driver to do some of the work. Some display drivers can only work with a bitmap that is no larger than the screen resolution, and may have trouble with bitmaps that use a different color resolution. The safest method to use when working with large bitmaps, is to create smaller bitmaps from the larger one (in code), and display the larger bitmap in sections called "bands".

7/16/98 4:31:28 PM

Last Modified: 21-MAR-00

## Article #17422: Copying a information from one memo field to another using TBlobStream

Question and Answer Database

FAQ2422D.txt Copying a information from one memo field to another using TBlobStream

Category :Database/VCL  
Platform :All  
Product :All 32 bit

Question:

How do I copy information from one memo field in one table to another memo field in another table?

Answer:

Here's an example using a TBlobStream:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BS1, BS2: TBlobStream;
begin
  BS1 := TBlobStream.Create(Table1Notes,bmRead);
  try
    Table2.Edit;
    BS2 := TBlobStream.Create(Table2MyBlob,bmReadWrite);
    try
      BS2.CopyFrom(BS1,BS1.Size);
    finally
      BS2.Free;
    end;
  finally
    BS1.Free;
  end;
  Table2.Post;
end;
```

7/16/98 4:31:28 PM



## Article #17426: Broadcasting a message.

Question and Answer Database

FAQ2426D.txt Broadcasting a message.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I broadcast a message (such as WM\_WININICHANGE) to all top level Windows?

Answer:

The Windows API function `SendMessage()` takes a 4 parameters. The first is the window handle. To broadcast to all top level Windows, use `HWND_BROADCAST`. The second parameter is the message itself, and then the `WParam` and the `LParam` parameters for the given message. For the `WM_WININICHANGE` message, `wParam` should be zero, and `LParam` should be either the address of a null terminated section name that changed, or `nil` if you want each window to examine all sections for the change. Passing a null value is not recommended unless you have modified several sections due to performance considerations.

Example:

```
var
  s : array[0..64] of char;
begin
  StrCopy(S, 'windows');
  SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(@S));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17430: Registering a file extension and context menu operations

Question and Answer Database

FAQ2430D.txt     Registering a file extension and context menu operations

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I register a file extension (file type) and how do I register context menu operations?

Answer:

Here's a sample that registers a sample extension (.myext). A sample application (MyApp.Exe) is going to handle this extension. It also registers one default action as well as two other context menu operations. You may have to logout and login in after running this code to get the icon to show up correctly.

uses

  Registry;

procedure TForm1.Button1Click(Sender: TObject);

var

  R : TRegIniFile;

begin

  R := TRegIniFile.Create('');

  with R do begin

    RootKey := HKEY\_CLASSES\_ROOT;

    WriteString('.myext', '', 'MyExt');

    WriteString('MyExt', '', 'Some description of MyExt files');

    WriteString('MyExt\DefaultIcon', '', 'C:\MyApp.Exe,0');

    WriteString('MyExt\Shell', '', 'This\_Is\_Our\_Default\_Action');

    WriteString('MyExt\Shell\First\_Action', '', 'This is our first action');

    WriteString('MyExt\Shell\First\_Action\command', '',

      'C:\MyApp.Exe /LotsOfParamaters %1');

    WriteString('MyExt\Shell\This\_Is\_Our\_Default\_Action', '',



```
    'This is our default action');
WriteString('MyExt\Shell\This_Is_Our_Default_Action\command',
    '', 'C:\MyApp.Exe %1');
WriteString('MyExt\Shell\Second_Action', '', 'This is our second
action');
WriteString('MyExt\Shell\Second_Action\command',
    '', 'C:\MyApp.Exe /TonsOfParameters %1');
Free;
end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17434: Custom InPlace Editor in StringGrid

Question and Answer Database

FAQ2434D.txt Custom InPlace Editor in StringGrid

Category :VCL

Platform :All

Product :Delphi 3.x

Question:

How can I popup my own inplace editor (such as a combobox) in a string grid cell?

Answer:

The following example demonstrates popping up a Combobox as an inplace editor to a Stringgrid component.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  {The combobox height is not settable, so we will}
  {instead size the grid to fit the combobox!}
  StringGrid1.DefaultRowHeight := ComboBox1.Height;
  {Hide the combobox}
  ComboBox1.Visible := False;
end;
```

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  {Get the ComboBox selection and place in the grid}
  StringGrid1.Cells[StringGrid1.Col,
                    StringGrid1.Row] :=
    ComboBox1.Items[ComboBox1.ItemIndex];
  ComboBox1.Visible := False;
  StringGrid1.SetFocus;
end;
```

```
procedure TForm1.ComboBox1Exit(Sender: TObject);
begin
  {Get the ComboBox selection and place in the grid}
```

```

StringGrid1.Cells[StringGrid1.Col,
                    StringGrid1.Row] :=
    ComboBox1.Items[ComboBox1.ItemIndex];
ComboBox1.Visible := False;
StringGrid1.SetFocus;
end;

procedure TForm1.StringGrid1SelectCell(Sender: TObject; Col,
    Row: Integer; var CanSelect: Boolean);
var
    R: TRect;
begin
    if ((Col = 3) AND
        (Row <> 0)) then begin
        {Size and position the combo box to fit the cell}
        R := StringGrid1.CellRect(Col, Row);
        R.Left := R.Left + StringGrid1.Left;
        R.Right := R.Right + StringGrid1.Left;
        R.Top := R.Top + StringGrid1.Top;
        R.Bottom := R.Bottom + StringGrid1.Top;
        ComboBox1.Left := R.Left + 1;
        ComboBox1.Top := R.Top + 1;
        ComboBox1.Width := (R.Right + 1) - R.Left;
        ComboBox1.Height := (R.Bottom + 1) - R.Top;
        {Show the combobox}
        ComboBox1.Visible := True;
        ComboBox1.SetFocus;
    end;
    CanSelect := True;
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17438: Delphi startup failure.

Question and Answer Database

FAQ2438D.txt Delphi startup failure.

Category :IDE

Platform :All

Product :All 32 bit

### Question:

I was working on a project, and shut down Delphi/C++ Builder after saving the project. Now Delphi/C++ Builder will not start up. What should I do?

### Answer:

Check to make sure the path to the project (including file names) did not contain any unusual characters (such as a comma). If you find that the path did contain an unusual character, use `regedit.exe` to edit the keys:

HKEY\_CURRENT\_USER/Software/Borland/C++Builder/1.0/ClosedProjects

HKEY\_CURRENT\_USER/Software/Borland/C++Builder/1.0/Closed Files

or if you were running Delphi go to

HKEY\_CURRENT\_USER/Software/Borland/Delphi/2.0/ClosedProjects

HKEY\_CURRENT\_USER/Software/Borland/Delphi/2.0/Closed Files

or

HKEY\_CURRENT\_USER/Software/Borland/Delphi/2.0/ClosedProjects

HKEY\_CURRENT\_USER/Software/Borland/Delphi/2.0/Closed Files

and modify the paths containing the unusual characters. You can replace the offending characters with any usual character.

Shut down RegEdit, and Delphi/C++ Builder should now start.



## Article #17442: Creating Non-Visual Components Without Icons

Question and Answer Database

FAQ2442D.txt      Creating Non-Visual Components Without Icons

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How I do to create a nonvisual component that does not show up on the component palette at design time (such as a TField)?

Answer:

Nonvisual components without icons are useful for components that are always subcomponents associated with other components.      Use RegisterNoIcon instead of RegisterComponent, and then create the component as you would normally.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17446: Compiler options.

Question and Answer Database

FAQ2446D.txt    Compiler options.

Category       :IDE

Platform       :All

Product        :All 32 bit

Question:

How can I quickly embed the current compiler options into the current unit?

Answer:

Press Ctrl-o, then press the o key;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17450: Turning on Screen saver with code.

Question and Answer Database

FAQ2450D.txt Turning on Screen saver with code.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I turn the Windows Screen Saver on in code?

Answer:

The following function tests to see if the Windows Screen Saver is available, and if so, turns it on:

```
function TurnScreenSaverOn : bool;
var
  b : bool;
begin
  result := false;
  if SystemParametersInfo(SPI_GETSCREENSAVEACTIVE,
                          0,
                          @b,
                          0) <> true then exit;
  if not b then exit;
  PostMessage(GetDesktopWindow, WM_SYSCOMMAND, SC_SCREENSAVE, 0);
  result := true;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17454: Returning application's exit code.

Question and Answer Database

FAQ2454D.txt    Returning application's exit code.

Category     : Object Pascal

Platform     : All

Product      : All 32 bit

Question:

How do I return an exit code to the calling application or batch file from a Windows Console mode application written in Delphi?

Answer:

Call the Halt procedure, passing the exit code you wish to return.

Example:

```
begin
  Halt(2)
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17458: Extracting an icon from a .exe or .dll.

Question and Answer Database

FAQ2458D.txt     Extracting an icon from a .exe or .dll.

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I Extract an icon from another executable or dll in code?

Answer:

Use the Windows API function `ExtractIcon()`, passing it the instance handle of your application, the path name of the application you wish to extract the icon from, and the number of the icon you wish to extract.

Example:

```
var
  TheIcon: TIcon;
begin
  TheIcon := TIcon.Create;
  TheIcon.Handle := ExtractIcon(hInstance,
                                'C:\SOMEPATH\SOMEPROG.EXE',
                                0);
  {Do something with the icon}
  TheIcon.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17462: Determining if Delphi Running.

Question and Answer Database

FAQ2462D.txt Determining if Delphi Running.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I tell if Delphi / C++ Builder is running?

Answer:

Use the example program in the faq for finding a Window.

Note that the main Delphi / C++ Builder class name is  
TAppBuilder;

```
if FindWindow('TAppBuilder', Nil) <> 0 Then  
    ShowMessage('Delphi and or C++ Builder is running');
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17466: Checking if TrueType fonts are available.

Question and Answer Database

FAQ2466D.txt      Checking if TrueType fonts are available.

Category       :Windows API

Platform       :All

Product        :All 32 bit

Question:

How can I tell if TrueType fonts are available on a given system?

Answer:

The following example should report if the TrueType engine is available, enabled, and at least one TrueType font is installed:

```
function IsTrueTypeAvailable : bool;
var
  {$IFDEF WIN32}
    rs : TRasterizerStatus;
  {$ELSE}
    rs : TRasterizer_Status;
  {$ENDIF}
begin
  result := false;
  if not GetRasterizerCaps(rs, sizeof(rs)) then exit;
  if rs.WFlags and TT_AVAILABLE <> TT_AVAILABLE then exit;
  if rs.WFlags and TT_ENABLED <> TT_ENABLED then exit;
  result := true;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17470: Why Delphi drawing functions draw one pixel short.

Question and Answer Database

FAQ2470D.txt Why Delphi drawing functions draw one pixel short.

Category :VCL

Platform :All

Product :All 32 bit

Question:

Why do line drawing methods like MoveTo and LineTo always draw a pixel short?

Answer:

This is the way most graphics systems work, including Windows. The VCL simply maps calls to the GDI functions. If you need a given line to go all the way to the last coordinate, then simply add one pixel.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17474: Recompiling complib.

Question and Answer Database

FAQ2474D.txt    Recompiling complib.

Category       :IDE

Platform       :All

Product        :Delphi 2.x

Question:

Can I recompile complib with the command line compiler?

Answer:

There is currently no way recompile the component library using the command line compiler.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

**Article #17478: KeyStrokes: Opening files in Delphi editor.**

Question and Answer Database

FAQ2478D.txt KeyStrokes: Opening files in Delphi editor.

Category :IDE

Platform :All

Product :All 32 bit

Question:

Is there a quick way to open up a file in the Delphi/C++ Builder IDE?

Answer:

Position the cursor over the file name and press Ctrl-Enter. This will work for any file in any directory including .pas and .h files listed in both "uses" and "include" statements.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17482: Getting the Windows version in code?

Question and Answer Database

FAQ2482D.txt Getting the Windows version in code?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I retrieve the Windows version in code?

Answer:

Here is an example that should work under all versions of Windows.

```
{ $IFDEF WIN32 }
function GetVersionEx(lpOs : pointer) : BOOL; stdcall;
  external 'kernel32' name 'GetVersionExA';
{ $ENDIF }

procedure GetWindowsVersion(var Major : integer;
                             var Minor : integer);

var
  { $IFDEF WIN32 }
  lpOS, lpOS2 : P OSVersionInfo;
  { $ELSE }
  l : longint;
  { $ENDIF }
begin
  { $IFDEF WIN32 }
  GetMem(lpOS, SizeOf(T OSVersionInfo));
  lpOS^.dwOSVersionInfoSize := SizeOf(T OSVersionInfo);
  while getVersionEx(lpOS) = false do begin
    GetMem(lpOS2, lpOS^.dwOSVersionInfoSize + 1);
    lpOS2^.dwOSVersionInfoSize := lpOS^.dwOSVersionInfoSize + 1;
    FreeMem(lpOS, lpOS^.dwOSVersionInfoSize);
    lpOS := lpOS2;
  end;
  Major := lpOS^.dwMajorVersion;
  Minor := lpOS^.dwMinorVersion;
  FreeMem(lpOS, lpOS^.dwOSVersionInfoSize);
```



```
{$ELSE}  
  l := GetVersion;  
  Major := LoByte(LoWord(l));  
  Minor := HiByte(LoWord(l));  
{$ENDIF}  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
var  
  Major : integer;  
  Minor : integer;  
begin  
  GetWindowsVersion(Major, Minor);  
  Mem1.Lines.Add(IntToStr(Major));  
  Mem1.Lines.Add(IntToStr(Minor));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17486: How do I create a power function?

Question and Answer Database

FAQ2486D.txt    How do I create a power function?

Category        :Object Pascal

Platform        :All

Product         :All 32 bit

Question:

How do I create a power function?

Answer:

The following example shows how to use the example power function to raise three to the power of 2:

```
function Power(x, y : extended) : extended;
begin
  result := exp(y*ln(x));
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(FloatToStr(Power(3, 2)));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17490: Local Interbase Startup Configuration.

Question and Answer Database

FAQ2490D.txt - Local Interbase Startup Configuration.

Category :Install Issues  
Platform :All-32Bit  
Product :

Question:

How can I keep the Local InterBase Server from automatically starting when I start Windows?

Answer:

There are three methods that you can use:

Right click on the Local InterBase Server icon and select "Startup Configuration" then select "Manual Startup".

Try taking the Local InterBase Server out of you Startup Menu, or comment it from the run/load lines of your Win.ini file.

Run REGEDIT.EXE and activate the key:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run and remove manually the IBSERVER entry.

4/2/99 11:56:34 AM

Last Modified: 01-SEP-99

## Article #17494: Preventing Components from Resizing

Question and Answer Database

FAQ2494D.txt Preventing Components from Resizing

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I prevent resizing of my component at design-time?

Answer:

Have your constructor set the default size of the object. Override the SetBounds method, and then check the component state. If it is in design mode (csDesigning in ComponentState) simply pass the default values for width and heights to the inherited SetBounds, otherwise you pass the parameters unchanged.

Example:

```
procedure TVu.SetBounds(ALeft : integer;
                       ATop : integer;
                       AWidth : integer;
                       AHeight : integer);
begin
  if csdesigning in componentstate then begin
    AWidth := 50;
    AHeight := 50;
    inherited;
  end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17498: Embedding a quote in a string.

Question and Answer Database

FAQ2498D.txt Embedding a quote in a string.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I embed a quote in a string since it is a reserved delimiter?

Answer:

You can either use two single quote characters where a single quote is required, or define a constant whose value is character number 39. If you have many quotes to embed (this can happen in a complex SQL statement), the second method is preferred.

Example:

```
const Quote = #39;
```

```
{This line will not compile}
```

```
ShowMessage('The smith's yard is greener than ours');
```

```
{This line will compile}
```

```
ShowMessage('The smith's yard is greener than ours');
```

```
{This line will compile}
```

```
ShowMessage('The smith' + Quote + 's yard is greener than ours');
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17502: How do I restart Windows in code.

Question and Answer Database

FAQ2502D.txt    How do I restart Windows in code.

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I restart Windows in code?

Answer:

The Windows API function `ExitWindows()` is incorrectly documented. There are three constants available for the first parameter, and the second parameter is used when exiting Windows to start a DOS program:

`EW_RESTARTWINDOWS`

`EW_REBOOTSYSYSTEM`

`EW_EXITANDEXECAPP`

Example:

```
ExitWindows(EW_RESTARTWINDOWS, 0 );
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17506: Implementing the VB TwipsPerPixel in Delphi

Question and Answer Database

FAQ: FAQ2506D - Implementing the VB TwipsPerPixel in Delphi

Category: VCL

Platform: All Windows

Product:

Question:

I need the equivalent of the VB TwipsPerPixel functions.  
How can I implement the same functionality in Delphi?

Answer:

The following example demonstrates two functions  
(TwipsPerPixelX, and TwipsPerPixelY) that implement  
the same functionality in Delphi.

Example:

```
function TwipsPerPixelX(Canvas : TCanvas) : Extended;
begin
    result := 1440 /
        GetDeviceCaps(Canvas.Handle,
            LOGPIXELSX);
end;

function TwipsPerPixelY(Canvas : TCanvas) : Extended;
begin
    result := 1440 /
        GetDeviceCaps(Canvas.Handle,
            LOGPIXELSY);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage(FloatToStr(TwipsPerPixelX(Form1.Canvas)));
    ShowMessage(FloatToStr(TwipsPerPixelY(Form1.Canvas)));
end;
```

end;

8/3/99 1:47:39 PM

Last Modified: 01-SEP-99



## Article #17510: Converting 16bit resource to 32bit.

Question and Answer Database

FAQ2510D.txt     Converting 16bit resource to 32bit.

Category     :Object Pascal

Platform     :All

Product     :All 32 bit

Question:

How can I convert a 16-bit resource file to a 32 bit resource file?

Answer:

If you have the original source file (.rc) then you can simply recompile the .rc file to a .res file using the Borland Resource Command Line Compiler (brcc32.exe) located in the Delphi/C++ Builders bin directory. If you only have a .res file to work with, you will need to use a quality resource compiler/decompiler such as Borland's Resource Workshop. Versions of the Borland's Resource Workshop later than 4.5 can extract, compile, and decompile both 16 and 32 bit resource files from a variety of sources including .res, .exe, .dll, drv, and .cpl files. The Borland Resource Workshop version 4.5 ship with the Borland RAD Pack product line.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17514: Creating resource files.

Question and Answer Database

FAQ2514D.txt Creating resource files.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

What are my options for creating resource files?

Answer:

You can use the Borland Command Line Resource Compiler (BRCC.EXE and BRCC32.EXE) that ships with Delphi and C++ Builder, or a WYSIWYG resource editor such as Borland's Resource Workshop 5.0 that ships with the Borland RAD Pack. The Resource Workshop 5.0 can compile and decompile both 16 and 32 bit resources from .rc, .res, .exe, .dll, .drv, .vbx, .cpl, .ico, .bmp, .rle, .dlg, fnt, and cur files.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17518: Using the WinApi GetObject to get a bitmap.

Question and Answer Database

FAQ2518D.txt Using the WinApi GetObject to get a bitmap.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How to I gain access to the TBitmap types used by the Windows API function GetObject()?

Answer:

The WinTypes/Windows, and Graphics units have two different definitions of TBitmap. You can qualify the one you want by prefacing the unit name like this:

var

```
AWindowsTBitmap : Windows.TBitmap;
```

```
AVclTBitmap : Graphics.TBitmap;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17522: calling windows Beep() function

Question and Answer Database

FAQ2522D.txt calling windows Beep() function

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

I want to call the Windows Beep() function, but I get the error "Too many parameters" when compiling. How can I get around this?

Answer:

The Beep() procedure is declared both in the SysUtils unit and the Windows unit. The following example demonstrates calling both procedures.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    SysUtils.Beep;
    Windows.Beep(100, 1000);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17526: Performing undo on an edit control.

Question and Answer Database

FAQ2526D.txt Performing undo on an edit control.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I programmatically undo changes in an edit type control?

Answer:

You can use:

```
Mem1.Perform(EM_UNDO, 0, 0);
```

If you want to check whether undo is available, so you can enable or disable a menu item choice, you can check the "Undo status" like this:

```
If Mem1.Perform(EM_CANUNDO, 0, 0) <> 0 then begin  
  {Undo is possible}  
end;
```

To perform a "Redo" simply "Undo" a second time.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17533: Embedding a copyright symbol in a string?

Question and Answer Database

FAQ2533D.txt Embedding a copyright symbol in a string?

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I embed a copyright symbol in a caption?

Answer:

For most fonts, typing ALT+0169 in the (object inspector) will produce a copyright symbol. In code, use `chr(169)` to produce a copyright symbol. If you are using the symbol font, the char code will be 211.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17537: Adding TrueType fonts to windows in code.

Question and Answer Database

FAQ2537D.txt Adding TrueType fonts to windows in code.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I add a TrueType font to Windows 95 in code;

Answer:

You should copy the file to the Windows\Fonts directory, then add an entry to the registry key:

```
"Software\Microsoft\Windows\CurrentVersion\Fonts".
```

The entry will be the font name and the path (if different than the Windows\Fonts directory). Once the registry entry is written, you should make a call to the Windows API function `AddFontResource()`, then broadcast a system-wide `WM_FONTCHANGE` message. Finally, you should make a call to the Windows API function `RemoveFontResource()`, to remove the resource lock on the font file, and broadcast a second system-wide `WM_FONTCHANGE` message.

Here is an example:

```
uses Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
  reg: TRegistry;
  b : bool;
begin
  CopyFile('C:\DOWNLOAD\FP000100.TTF',
          'C:\WINDOWS\FONTS\FP000100.TTF', b);
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.LazyWrite := false;
```

```
reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\Fonts',
            false);
reg.WriteString('TESTMICR (TrueType)', 'FP000100.TTF');
reg.CloseKey;
reg.free;
{Add the font resource}
AddFontResource('c:\windows\fonts\FP000100.TTF');
SendMessage(HWND_BROADCAST, WM_FONTCHANGE, 0, 0);
{Remove the resource lock}
RemoveFontResource('c:\windows\fonts\FP000100.TTF');
SendMessage(HWND_BROADCAST, WM_FONTCHANGE, 0, 0);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17544: Converting resources from 16bit to 32bit.

Question and Answer Database

FAQ2544D.txt     Converting resources from 16bit to 32bit.

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I convert a 16 bit res file to a 32 bit res file?

Answer:

Use Borland's WYSIWYG Resource Workshop 5.0 resource editor that ships with the Borland RAD Pack to decompile the 16 bit .res file to a .rc file. Once you have a .rc file, you can close the project, set your preferences in Resource Workshop to "Win32", reload the .rc file, then save as a .res file. Note: The Resource Workshop 5.0 can compile and decompile both 16 and 32 bit resources from .rc, .res, .exe, .dll, .drv, .vbx, .cpl, .ico, .bmp, .rle, .dlg, fnt, and cur files.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17547: TDateTime does not set the field out of 1900

Question and Answer Database

FAQ2547C.txt     TDateTime does not set the field out of 1900  
Category        :Database Issues  
Platform        :All  
Product         :C++Builder 3.x

### Question:

I have a Table, assume paradox but the issue is the same with all tables, and one of the fields is a DATE field. I get to the date field by using `TDataSet.FieldName ("MyDateField").AsString` I read the string into a `AnsiString` and then create the `TDateTime` with the `AnsiString` (`TDateTime dt (AnsiString)`). then I decode the `TDateTime` to change the year to something in the 2000 and then set it back to the dataset using `TDataSet.FieldName ("MyDateField").AsString=dt.DateString();` The field is then not set to the 2000 year but back to the 1900 year. What the heck is the problem. Is `TDateTime` not year 2000 compliant?

### Answer:

No, `TDateTime` is absolutely year 2000 compliant. The problem is that you are setting the field as a string and something is being lost in the process. What you need to do is read in the `TDateTime` as a `TDateTime` by using a `TDateField` or a `TDateTimeField`. Instead of this:

```
AnsiString as = DataSet->FieldName ("FieldName")->AsString;
TDateTime dt (as, TDateTime::Date);
short int y, m, d;
dt.DecodeDate (&y, &d, &y);
y = 2001;
TDateTime dt2 (y, m, d);
DataSet->Edit ();
DataSet->FieldName ("FieldName")->AsString = dt2.DateString();
```

Do this:

```
TDateField *df = (TDateField*)DataSet->FieldName ("FieldName");
```

```
TDateTime dt = df->AsDateTime;  
short int y, m, d;  
dt.DecodeDate (&y, &d, &y);  
y = 2001;  
TDateTime dt2 (y, m, d);  
DataSet->Edit ();  
df->AsDateTime = dt2;
```

It will now work fine.

Y.T.

5/13/98 10:40:33 AM

Last Modified: 01-SEP-99

## Article #17548: Copying contents of a control to the clipboard.

Question and Answer Database

FAQ2548D.txt Copying contents of a control to the clipboard.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get a control in another window to copy it's contents to the Windows clipboard?

Answer:

Send a WM\_COPY message to the active window to tell it to copy its contents to the clipboard. If the control does not support the message, it is simply ignored, and no exception is raised.

Example:

```
SendMessage(Edit1.handle, WM_COPY, 0, 0);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17552: Checking available disk space on large drives.

Question and Answer Database

FAQ2552D.txt      Checking available disk space on large drives.

Category       :Windows API

Platform       :All

Product        :All 32 bit

Question:

How do I check for available disk space on a drive larger than 2 gigabytes?

Answer:

You will need to call the Windows API function `GetDiskFreeSpaceEx()` and convert the returned integers to doubles, since integers greater than 2 gigabytes are not supported in Delphi.

Example:

```
function GetDiskFreeSpaceEx(lpDirectoryName: PAnsiChar;
  var lpFreeBytesAvailableToCaller : Integer;
  var lpTotalNumberOfBytes: Integer;
  var lpTotalNumberOfFreeBytes: Integer) : bool;
  stdcall;
  external kernel32
  name 'GetDiskFreeSpaceExA';
```

```
procedure GetDiskSizeAvail(TheDrive : PChar;
  var TotalBytes : double;
  var TotalFree : double);
var
  AvailToCall : integer;
  TheSize : integer;
  FreeAvail : integer;
begin
  GetDiskFreeSpaceEx(TheDrive,
    AvailToCall,
    TheSize,
    FreeAvail);
```

```

{$IFOPT Q+}
{$DEFINE TURNOVERFLOWON}
{$Q-}
{$ENDIF}
if TheSize >= 0 then
    TotalBytes := TheSize else
if TheSize = -1 then begin
    TotalBytes := $7FFFFFFF;
    TotalBytes := TotalBytes * 2;
    TotalBytes := TotalBytes + 1;
end else
begin
    TotalBytes := $7FFFFFFF;
    TotalBytes := TotalBytes + abs($7FFFFFFF - TheSize);
end;

if AvailToCall >= 0 then
    TotalFree := AvailToCall else
if AvailToCall = -1 then begin
    TotalFree := $7FFFFFFF;
    TotalFree := TotalFree * 2;
    TotalFree := TotalFree + 1;
end else
begin
    TotalFree := $7FFFFFFF;
    TotalFree := TotalFree + abs($7FFFFFFF - AvailToCall);
end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    TotalBytes : double;
    TotalFree : double;
begin
    GetDiskSizeAvail('C:\',
                    TotalBytes,
                    TotalFree);
    ShowMessage(FloatToStr(TotalBytes));
    ShowMessage(FloatToStr(TotalFree));
end;

```



## Article #17555: Dropping a combobox's list in code.

Question and Answer Database

FAQ2555D.txt Dropping a combobox's list in code.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get a ComboBox to drop down its list box in code?

Answer:

Send the ComboBox a CB\_SHOWDROPDOWN message, passing a 1 (true) in the WParam parameter to drop the list down, or a 0 (false) to hide it.

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    SendMessage(ComboBox1.Handle,  
                CB_SHOWDROPDOWN,  
                1,  
                0);
```

```
    ComboBox1.SetFocus;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17559: Setting a form's maxwidth in code.

Question and Answer Database

FAQ2559D.txt    Setting a form's maxwidth in code.

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I get a Form's width to be less than 120 (approx) pixels?

Answer:

This is controlled and overridden by Windows. Under some versions of Windows, you may be successful by creating a message handler to respond to the WM\_GETMINMAXINFO message:

Example:

```
{Place this code in the private section of the Form declaration}
procedure WMGetMinMaxInfo(var Message: TWMGetMinMaxInfo);
  message WM_GETMINMAXINFO;
```

```
{Place this code in the Form implementation section}
procedure TForm1.WMGetMinMaxInfo(var Message: TWMGetMinMaxInfo);
begin
  with message do begin
    MinMaxInfo^.ptMaxSize.x := 20;
    MinMaxInfo^.ptMaxSize.y := 20;
    MinMaxInfo^.ptMinTrackSize.x := 20;
    MinMaxInfo^.ptMinTrackSize.y := 20;
  end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17563: Getting the windows directory.

Question and Answer Database

FAQ2563D.txt Getting the windows directory.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I determine where the Windows directory is?

Answer:

You can make a call to the Windows API function

GetWindowsDirectory().

If you want the system directory, simply call the Windows API function

GetSystemDirectory().

Example:

```
{ $IFDEF WIN32 }
```

```
const MAX_PATH = 144;
```

```
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  a : Array[0..MAX_PATH] of char;
```

```
begin
```

```
  GetWindowsDirectory(a, sizeof(a));
```

```
  ShowMessage(StrPas(a));
```

```
  GetSystemDirectory(a, sizeof(a));
```

```
  ShowMessage(StrPas(a));
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17567: Getting the size of the desktop.

Question and Answer Database

FAQ2567D.txt Getting the size of the desktop.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I find the size of the Desktop minus the taskbar?

Answer:

Call the Windows API function `SystemParametersInfo()` passing the `SPI_GETWORKAREA` parameter along with the address of the rectangle structure to receive the coordinates.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r : TRect;
begin
  SystemParametersInfo(SPI_GETWORKAREA,
                      0,
                      @r,
                      0);
  Mem1.Lines.Add(IntToStr(r.Top));
  Mem1.Lines.Add(IntToStr(r.Left));
  Mem1.Lines.Add(IntToStr(r.Bottom));
  Mem1.Lines.Add(IntToStr(r.Right));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17571: Hiding the windows taskbar.

Question and Answer Database

FAQ2571D.txt Hiding the windows taskbar.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I hide the Windows Task Bar?

Answer:

First call the Windows API function `FindWindow()` to retrieve the handle to the TaskBar Window, then call the Windows API function `ShowWindow()` passing the predefined constant `SW_HIDE`.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  hTaskBar : THandle;
begin
  hTaskbar := FindWindow('Shell_TrayWnd', Nil);
  ShowWindow(hTaskBar, SW_HIDE);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
var
  hTaskBar : THandle;
begin
  hTaskbar := FindWindow('Shell_TrayWnd', Nil);
  ShowWindow(hTaskBar, SW_SHOWNORMAL);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17575: Keeping an application in a minimized state?

Question and Answer Database

FAQ2575D.txt Keeping an application in a minimized state?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I keep my application in a iconized (minimized) state?

Answer:

First, you must set the WindowState property of the form to Minimized. Then you must create a message handler to respond to the Windows WM\_QueryOpen message, to let Windows know that you do not want the form to resize.

Example:

{Place this code in the private section of the Form declaration}

```
procedure WMQueryOpen(VAR Msg : TWMQueryOpen); message WM_QUERYOPEN;
```

{Place this code in the Form implementation section}

```
procedure TForm1.WMQueryOpen(VAR Msg : TWMQueryOpen);
```

```
begin
```

```
    Msg.Result := 0;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17582: Getting the current line number of a TMemo?

Question and Answer Database

FAQ2582D.txt Getting the current line number of a TMemo?

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I determine the current line number of a TMemo?

Answer:

You can use the Windows EM\_LINEFROMCHAR message to get the current (zero based) line number (position of the caret) of any edit control.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  LineNumber : integer;
begin
  LineNumber := SendMessage(Memo1.Handle,
                           EM_LINEFROMCHAR,
                           word(-1),
                           0);
  ShowMessage(IntToStr(LineNumber));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17591: Calling DeleteFile function

Question and Answer Database

FAQ2591D.txt     Calling DeleteFile function

Category     :Object Pascal

Platform     :All

Product      :All 32 bit

Question:

I want to call the Windows DeleteFile() function, but I get the error "Incompatible types: 'string' and 'Pointer'" when compiling. How can I get around this?

Answer:

The DeleteFile() procedure is declared both in the SysUtils unit and the Windows unit. The following example demonstrates calling both procedures. The version declared in the SysUtils unit accepts strings, and the Windows version accepts PChars.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s : string;
  a : array[0..MAX_PATH - 1] of char;
begin
  s := 'C:\SomeFile';
  SysUtils.DeleteFile(s);
  a := 'C:\SomeFile';
  Windows.DeleteFile(@a);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17595: Using far pointers.

Question and Answer Database

FAQ2595D.txt Using far pointers.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I instruct the compiler that I require a far pointer instead of a near pointer?

Answer:

In 16 bit versions of Delphi, all pointers are far pointers. When running in a 32 bit environment, all pointers are considered to be huge near pointers as there is no concept of far pointer that cross process address space without thunking.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17599: String Trim functions.

Question and Answer Database

FAQ2599D.txt String Trim functions.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

Where are the equivalent functions for the Visual Basic Trim functions?

Answer:

There are many ways to implement these and similar functions. The following example is a great start for a string handling unit add to your programming tool box.

```
unit StrFunc;

interface

procedure LTrim(var s : string);

procedure RTrim(var s : string);

procedure Trim(var s : string);

implementation

{$R *.DFM}

procedure LTrim(var s : string);
begin
  while ((length(s) > 0) and
        (s[1] = #32)) do
    Delete(s, 1, 1);
end;

procedure RTrim(var s : string);
begin
```

```
while ((length(s) > 0) and
      (s[length(s)] = #32)) do
  Delete(s, Length(s), 1);
end;
```

```
procedure Trim(var s : string);
begin
  LTrim(s);
  RTrim(s);
end;
```

end.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17603: Virtual key constants.

Question and Answer Database

FAQ2603D.txt Virtual key constants.

Category :VCL

Platform :All

Product :All 32 bit

Question:

Where are the VK\_Key constants A-Z and 0-9 declared?

Answer:

They are not declared in Delphi, since they are simply the ordinal value for those keys.

VK\_0 thru VK\_9 are the same as ASCII '0' thru '9' (\$30 - \$39)

VK\_A thru VK\_Z are the same as ASCII 'A' thru 'Z' (\$41 - \$5A).

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17607: Changing the Windows wallpaper in code?

Question and Answer Database

FAQ2607D.txt - Changing the Windows wallpaper in code?

Category :Windows API  
Platform :All-32Bit  
Product :

Question:

How do I change the Windows wallpaper in code?

Answer:

Call the Windows API function `SystemParametersInfo()` passing the `SPI_SETDESKWALLPAPER` parameter along with the filename of the new bitmap to use.

Example:

```
SystemParametersInfo(SPI_SETDESKWALLPAPER,  
                    0,  
                    PChar('C:\SOME_PATH\SOME.BMP'),  
                    SPIF_SENDWININICHANGE);
```

4/2/99 12:26:15 PM

Last Modified: 01-SEP-99

## Article #17610: Passing TPrinter handle to GetDeviceCaps()

Question and Answer Database

FAQ2610D.txt    Passing TPrinter handle to GetDeviceCaps()

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

Is there any way to call the GetDeviceCaps() function before I start my print job?

Answer:

Yes, You can pass the Printer.Handle property to the GetDeviceCaps() function instead of the Printer.Canvas.Handle property.

Example:

```
{Before BeginDoc;}
HRes := GetDeviceCaps(Printer.Handle, HORZRES);
VRes := GetDeviceCaps(Printer.Handle, VERTRES);

Printer.BeginDoc;

{After BeginDoc;}
InchX := GetDeviceCaps(Printer.Canvas..Handle, LOGPIXELSX);
InchY := GetDeviceCaps(Printer.Canvas.Handle, LOGPIXELSY);

Printer.Canvas.TextOut(InchX, InchY, 'Delphi Is RAD!');

Printer.EndDoc;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17614: Memo Input and line breaks.

Question and Answer Database

FAQ2614D.txt Memo Input and line breaks.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I prevent Delphi from storing the contents of a Memo input at design time as separate lines in the project/executable?

Answer:

Set the TMemo's WordWrap property to false in design mode. If you desire word wrap at runtime, set the TMemo's WordWrap property to true during the form's create event.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17618: Changing windows default printer in code.

Question and Answer Database

FAQ2618D.txt - Changing windows default printer in code.

Category :Printing  
Platform :All-32Bit  
Product :

Question:

How do I change the default Windows printer?

Answer:

You need to change the printer, driver, and port as listed in the device key of the Windows section in the Windows.ini file, then broadcast a system wide WM\_WININICHANGE message.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  WinIni : TIniFile;
  WinIniFileName : array[0..MAX_PATH] of char;
  s : array[0..64] of char;
begin
  GetWindowsDirectory(WinIniFileName, sizeof(WinIniFileName));
  StrCat(WinIniFileName, '\win.ini');
  WinIni := TIniFile.Create(WinIniFileName);
  try
    WinIni.WriteString('windows',
                      'device',
                      'HP LaserJet Series II,HPPCL,LPT1:');
  finally
    WinIni.Free;
  end;

  StrCopy(s, 'windows');
  SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(@s));
end;
```

4/2/99 11:25:11 AM

Last Modified: 01-SEP-99



## Article #17622: Raster operations.

Question and Answer Database

FAQ2622D.txt Raster operations.

Category :VCL

Platform :All

Product :All 32 bit

### Question:

When using a canvas, the return value from the pixels property returns an invalid color of -1 or hex \$FFFFFFFF. How can I get around this?

### Answer:

This can happen if the requested pixel is not in the current clipping region. This can also happen on devices that do not support the Windows API functions GetPixel() and SetPixel(). You should check the return value from the GetDeviceCaps() to check to see if a device supports raster operations before using these functions. A printer and/or metafile canvas is a likley candidate for a device that the pixels property may fail on.

### Example:

```
if (GetDeviceCaps(Printer.Canvas.Handle, RASTERCAPS)
    and RC_NONE) = RC_NONE then
NoRasterOperations := True;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17626: How do I create a Resource only DLL?

Question and Answer Database

FAQ2626D.txt    How do I create a Resource only DLL?

Category        :Object Pascal

Platform        :All

Product         :All 32 bit

Question:

How do I create a Resource only DLL?

Answer:

Create and build an empty DLL project, that contains a resource link reference to the .res file that contains your resources.

Example:

```
library ResTest;
```

```
uses
```

```
    SysUtils;
```

```
{ $R MYRES.RES }
```

```
begin
```

```
end.
```

To use you resource only DLL, simply load the dll and the resources you wish to use:

Example:

```
{ $IFDEF WIN32 }
```

```
const BadDllLoad = 0;
```

```
{ $ELSE }
```

```
const BadDllLoad = 32;
```

```
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
h : THandle;  
Icon : THandle;
```

```
begin
```

```
  h := LoadLibrary('RESTEST.DLL');
```

```
  if h <= BadDllLoad then
```

```
    ShowMessage('Bad Dll Load')
```

```
  else begin
```

```
    Icon := LoadIcon(h, 'ICON_1');
```

```
    DrawIcon(Form1.Canvas.Handle, 10, 10, Icon);
```

```
    FreeLibrary(h);
```

```
  end;
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17630: Using GetTimeZoneInformation() function?

Question and Answer Database

FAQ2630D.txt Using GetTimeZoneInformation() function?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

What are the constants for the return values for the Windows API function GetTimeZoneInformation()?

Answer:

The following defines the return values for the GetTimeZoneInformation() function.

```
const TIME_ZONE_ID_UNKNOWN = 0;  
const TIME_ZONE_ID_STANDARD = 1;  
const TIME_ZONE_ID_DAYLIGHT = 2;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17634: Trying to delete a running .exe?

Question and Answer Database

FAQ2634D.txt Trying to delete a running .exe?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I delete my executable while it is running?

Answer:

Under Win32, unless you are running from a removable drive, you cannot delete a running executable. You can have Windows delete the executable the next time Windows is ran by adding an entry to the RunOnce key in the Windows registry under:

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce

You can name the key anything you like, and specify a command line to another executable or to a dos command passed to command.com.

Example:

uses

Registry;

procedure TForm1.Button1Click(Sender: TObject);

var

reg: TRegistry;

begin

reg := TRegistry.Create;

with reg do begin

RootKey := HKEY\_LOCAL\_MACHINE;

LazyWrite := false;

OpenKey('Software\Microsoft\Windows\CurrentVersion\RunOnce',  
false);

WriteString('Delete Me!', 'command.com /c del FILENAME.EXT');

```
    CloseKey;  
    free;  
end;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17638: Determining is a canvas is a Metafile Canvas?

Question and Answer Database

FAQ2638D.txt     Determining is a canvas is a Metafile Canvas?

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I determine in code if a canvas is a Metafile Canvas?

Answer:

You can check the return value of the Windows API function `GetDeviceCaps()` to check if a given dc is used by a Metafile.

Example:

```
if (GetDeviceCaps(Canvas.Handle, TECHNOLOGY) and  
    DT_METAFILE) = DT_METAFILE then  
IsMetafileDc := True;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17641: Access97 ODBC with BDE 4.51

Question and Answer Database

FAQ2641C.txt      Access97 ODBC with BDE 4.51

Category        :Database Issues

Platform        :All

Product         :C++Builder 3.x

### Question:

I have BCB Pro 1.0, Upgraded BDE to 4.5, and have Office 97 installed on my machine.

When i attempt to connect to the access97 database, i get :

Cannot load an IDAPI service library.

File : C:\Program Files\Common Files\Microsoft Shared\DAO\dao3032.dll

Alias : AliasName

### Answer:

Office 97 comes with DAO3.5, but the BDE4.51 defaults to the DAO3.0 driver. Open the BDE Admin and goto the native access driver and change the dll32 to IDDA3532.dll. The Registry change is only necessary if you wish to use DAO3.0

7/2/98 10:32:32 AM

Last Modified: 01-SEP-99



## Article #17642: Disabling autodial when using internet controls.

Question and Answer Database

FAQ2642D.txt - Disabling autodial when using internet controls.

Category :Internet/WEB  
Platform :All-32Bit  
Product :

### Question:

Whenever I try to use the Internet controls, the Windows dialer attempts to dial out, even if I am connected to the internet via the Network. What can I do?

### Answer:

You can disable this feature of the operating system by going to the Windows Control Panel, select the Internet icon, and go to the Connections page. Un-Select the check box "Connect to the internet as needed".

4/2/99 11:48:08 AM

Last Modified: 01-SEP-99

## Article #17646: Displaying a file's associated icon.

Question and Answer Database

FAQ2646D.txt    Displaying a file's associated icon.

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I display the icon that is associated with a given file type?

Answer:

Use the ShellApi function `ExtractAssociatedIcon()` to retrieve the icon that is associated with the file.

Example:

```
uses ShellApi;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    Icon : hIcon;
```

```
    IconIndex : word;
```

```
begin
```

```
    Icon := ExtractAssociatedIcon(HInstance,  
                                  'C:\SomePath\SomeFile.ext',  
                                  IconIndex);
```

```
    DrawIcon(Form1.Canvas.Handle, 10, 10, Icon);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

**Article #17649: Can I get function params from a dll.**

Question and Answer Database

FAQ2649D.txt Can I get function params from a dll.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I get a list of the function parameters of a given function in a Dynamic Link Library (DLL) at runtime?

Answer:

While you can get the list of exported functions in a DLL, you cannot get the parameter list. To get this information, you would need get the documentation from the distributor of the DLL.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## **Article #17653: Loading a local html file in the HTML control**

Question and Answer Database

FAQ2653D.txt - Loading a local html file in the HTML control

Category :Internet/WEB

Platform :All-32Bit

Product :

Question:

How do I load a local file in the HTML control?

Answer:

Use the file command in the URL:

Example:

```
HTML1.RequestDoc('file:///C:/Netscape/Navigator/bookmark.htm');
```

4/2/99 11:48:20 AM

Last Modified: 01-SEP-99

## Article #17657: Printing a WEB page with the HTML control?

Question and Answer Database

FAQ2657D.txt - Printing a WEB page with the HTML control?

Category :Internet/WEB  
Platform :All-32Bit  
Product :

Question:

How do I print a WEB page with the HTML control?

Answer:

Use either the HTML control's AutoPrint method, or alternatively use the PrintPage method.

Example using AutoPrint:

```
uses Printers;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
var
```

```
    OldCur: TCursor;
```

```
begin
```

```
    OldCur := Screen.Cursor;
```

```
    with Printer do begin
```

```
        BeginDoc;
```

```
        HTML1.AutoPrint(handle);
```

```
        Title := HTML1.URL;
```

```
        EndDoc;
```

```
    end;
```

```
    Screen.Cursor := OldCur;
```

```
end;
```

4/2/99 11:43:08 AM

Last Modified: 01-SEP-99

## Article #17661: Preventing system lockup in Win95.

Question and Answer Database

FAQ2661D.txt Preventing system lockup in Win95.

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I prevent Windows 95 from locking up when my application performs an illegal operation when run from the IDE?

Answer:

When the dialog box announces the Illegal Operation, do not immediately press the "Done" button. Instead, go back to the IDE and select "Run|Program Reset" from the IDE menu (or press Ctrl-F2 inside the IDE) to release the Debug Hook, and then go back to the dialog box and press the "Done" button.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17665: Using the Application's OnIdle Event.

Question and Answer Database

FAQ2665D.txt Using the Application's OnIdle Event.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How do I have my application perform work when the computer is idle?

Answer:

You can create an OnIdle event to be called whenever your application is waiting for messages to be processed. Simply declare the event handler procedure in the private section of the forms declaration:

```
{ Private declarations }  
procedure IdleEventHandler(Sender: TObject; var Done: Boolean);
```

In then implementation section, define the procedure:

```
procedure TForm1.IdleEventHandler(Sender: TObject;  
    var Done: Boolean);  
begin  
    {Do a small bit of work here}  
    Done := false;  
end;
```

Then assign the Application's OnIdle event to point to your new procedure(You may do this wherever you like, but a good place would be the Forms OnCreate method):

```
Application.OnIdle := IdleEventHandler;
```

OnIdle is called only once, as the application transitions into an idle state. It is not called continuously unless Done is set to False If Done is False, WaitMessage is not called. Applications that set Done to False consume an



inordinate amount of CPU time that affects overall system performance.

You can set the boolean variable "done" to true when you want to stop the event handler from executing.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17669: EHRTRW.DLL not found

Question and Answer Database

FAQ2669D.txt - EHRTRW.DLL not found

Category :AS400  
Platform :All-32Bit  
Product :

Question:

How come I'm getting the error "EHRTRW.DLL not found"?

Answer:

This error occurs with 16 bit Delphi/400 when using the Rumba router.  
You must add the following to the WIN.INI:

```
[CO400]
RTRW = False?
```

4/1/99 2:44:24 PM

Last Modified: 01-SEP-99

## Article #17673: 'Index List' error

Question and Answer Database

FAQ2673D.txt - 'Index List' error

Category :AS400  
Platform :All-32Bit  
Product :

Question:

How come I'm getting a "Index List" error?

Answer:

This errors occurs after updating your ClientObjects/400 version. To solve the problem delete all the files in the \DDS directory. The \DDS directory contains file and field descriptions. Select Build All from your Delphi project menu.

4/1/99 2:44:32 PM

Last Modified: 01-SEP-99

## Article #17677: Passing a function to a procedure or vice versa.

Question and Answer Database

FAQ2677D.txt Passing a function to a procedure or vice versa.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

Can I pass a function or procedure to another function or procedure?

Answer:

Yes. You must declare a procedure or function type to use as the parameter type, then you can pass the address of the procedure that you wish to pass. You must be careful to pass functions or procedures that are compatible with the function or procedure expected, as little error checking will be provided by the compiler.

Example:

```
type
  AFunctionType = function(IntIn : integer) : integer;

function AddProc(IntIn : integer) : integer;
begin
  Result := IntIn + 1;
end;

function SubProc(IntIn : integer) : integer;
begin
  Result := IntIn - 2;
end;

procedure PassAFunction(var IntIn : integer;
                        fn : AFunctionType);
begin
  IntIn := fn(IntIn);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i : integer;
begin
  i := 10;

  PassAFunction(i, @AddProc);
  ShowMessage(IntToStr(i));

  PassAFunction(i, @SubProc);
  ShowMessage(IntToStr(i));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17681: How do I Print a TStringGrid / TDBGrid?

Question and Answer Database

FAQ2681D.txt    How do I Print a TStringGrid / TDBGrid?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I Print a TStringGrid / TDBGrid?

Answer:

You will need to create two loops, one to print each row, and one to print each column using TPrinter's TextOut method. You should determine the widest line of text in the grid you will need to accommodate, and adjust your font size as necessary, or decide if you need to print some of the columns on a separate page.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17685: How do I print in color?

Question and Answer Database

FAQ2685D.txt    How do I print in color?

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I print in color?

Answer:

Generally, no special effort is required to print in color assuming the printer is a color printer and is set to its native color mode. Windows will automatically convert color to black and white if the printer does not support color. If you need to change the color mode in code, you can access the DevMode structure of the printer driver.

Example:

```
uses Printers;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  Device : array[0..255] of char;
```

```
  Driver : array[0..255] of char;
```

```
  Port   : array[0..255] of char;
```

```
  hDMode : THandle;
```

```
  PDMode : PDEVMODE;
```

```
begin
```

```
  with Printer do begin
```

```
    PrinterIndex := PrinterIndex;
```

```
    GetPrinter(Device, Driver, Port, hDMode);
```

```
  if hDMode <> 0 then begin
```

```
    pDMode := GlobalLock(hDMode);
```

```
    if pDMode <> nil then begin
```

```
      pDMode.dmFields := pDMode.dmFields or dm_Color;
```

```
      pDMode.dmColor := DMCOLOR_COLOR;
```

```
      GlobalUnlock(hDMode);
```

```
    end;  
end;  
  
PrinterIndex := PrinterIndex;  
BeginDoc;  
Canvas.Font.Color := clRed;  
Canvas.TextOut(100,100, 'Red As A Rose!');  
EndDoc;  
end;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17692: Saving a TFont Object.

Question and Answer Database

FAQ2692D.txt Saving a TFont Object.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How do I save a TFont object to a File/Registry/ini/Database?

Answer:

You should instead save the font's attributes (name, size, and so forth), not the font object itself. When you read the font information back into your program, you should test to make sure the font still exists before trying to use it.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17696: Listbox selected property

Question and Answer Database

FAQ2696D.txt Listbox selected property

Category :VCL

Platform :All

Product :All 32 bit

Question:

When I try to set the selected property of a ListBox component, I receive an exception that the Index is out of bounds. How do I get around this?

Answer:

The selected property should only be used with ListBox components that have the MultiSelect property set to true. If you are working with a ListBox that has the MultiSelect property set to false, use the ItemIndex property to select the item.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('1');
  ListBox1.Items.Add('2');
  {This will fail on a single selection ListBox}
  // ListBox1.Selected[1] := true;
  ListBox1.ItemIndex := 1; {This is ok}
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17700: Getting a list of valid time zones from windows.

Question and Answer Database

FAQ2700D.txt Getting a list of valid time zones from windows.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I retrieve a list of valid time zones from the system?

Answer:

The following example lists the time zones known to Windows.

Example:

```
uses Registry;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    reg : TRegistry;
```

```
    ts : TStringList;
```

```
    i : integer;
```

```
begin
```

```
    reg := TRegistry.Create;
```

```
    reg.RootKey := HKEY_LOCAL_MACHINE;
```

```
    reg.OpenKey(
```

```
'SOFTWARE\Microsoft\Windows\CurrentVersion\Time Zones',  
        false);
```

```
    if reg.HasSubKeys then begin
```

```
        ts := TStringList.Create;
```

```
        reg.GetKeyNames(ts);
```

```
        reg.CloseKey;
```

```
        for i := 0 to ts.Count - 1 do begin
```

```
            reg.OpenKey(
```

```
'SOFTWARE\Microsoft\Windows\CurrentVersion\Time Zones\' +  
                ts.Strings[i],
```

```
                false);
```

```
            Mem1.Lines.Add(ts.Strings[i]);
```

```
            Mem1.Lines.Add(reg.ReadString('Display'));
```

```
    Mem1.Lines.Add(reg.ReadString('Std'));
    Mem1.Lines.Add(reg.ReadString('Dlt'));
    Mem1.Lines.Add('-----');
    reg.CloseKey;
end;
ts.Free;
end else
reg.CloseKey;
reg.free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17704: What types of Routers are able to be used?

Question and Answer Database

FAQ2704D.txt - What types of Routers are able to be used?

Category :AS400  
Platform :All-32Bit  
Product :

Question:

What types of Routers are able to be used with the Delphi/400 product?

Answer:

In general, the routers accepted are:  
PCS, CA, NS Router, SNA Server, SNA Client, and Rumba.

4/1/99 2:44:42 PM

Last Modified: 01-SEP-99

## Article #17707: Deploying the BDE: InstallShield can't find BDE drivers

Question and Answer Database

FAQ2707C.txt    Deploying the BDE:    InstallShield can't find BDE drivers

Category        :Database Issues

Platform        :Windows 95

Product         :    C++Builder3.x

Question:

I'm trying to deploy a full version of the BDE with my application. When I use InstallShield Express, though, when I create a new BDE alias there is a page entitled: Step 3 of 4. On this page, there is a combo box entitled type. This combo box is supposed to contain driver types, but it's empty. When I look at the help, it says that InstallShield is looking for driver information from a key in the registry called HKEY\_LOCAL\_MACHINE\SOFTWARE\Borland\Database Engine\Settings\DRIVERS. This key isn't there!

Answer:

The reason for this problem is that the BDE configuration settings are set up for Windows 95/NT AND Windows 3.1. This means that all the driver information is stored in the idapi.cfg file, and none of it is put into the registry.

To change this:

1. Go into the BDE Administration
2. Go to Object\Options and change the setting from Win3.1 and Win95/NT to Win95/NT only.
3. Go to Object\Save As Configuration and save over the old Idapi.cfg file.
4. Check your registry. The DRIVERS key should have been added.
5. Uninstall InstallShield
6. Reinstall InstallShield

Now you should see the drivers listed in that combo box.

6/8/98 4:58:30 PM

Last Modified: 01-SEP-99

## Article #17708: BCD Number Truncate to only 4 Decimal Places

Question and Answer Database

FAQ2708D.txt - BCD Number Truncate to only 4 Decimal Places

Category :Database/VCL

Platform :All-32Bit

Product :

Question:

When I use the BDE setting ENABLE BCD, all my data is truncated to 4 decimal places. Is this as designed?

Answer:

Delphi uses the Delphi Currency type for holding the BCD values internally. This is actually a signed 64 bit integer (comp) with the last 4 digits used for the decimal places.

If a BCD value from BDE that contains more than 4 decimals, an exception is raised.

The only way to work with more decimal places than 4 is to call GetData on the field and work directly with the FMTBCD structure.

4/2/99 12:12:52 PM

Last Modified: 01-SEP-99



## Article #17712: Requirements for running CO400INS

Question and Answer Database

FAQ2712D.txt - Requirements for running CO400INS

Category :AS400  
Platform :All-32Bit  
Product :

Question:

I've added my key via CO400INS, yet when I run CO400/ABOUT my key is not entered.

Answer:

Make sure the CO400 subsystem is not running  
When running CO400INS make sure you are logged in as QSECOFR.

4/1/99 2:44:52 PM

Last Modified: 01-SEP-99

## Article #17716: Changing font color on standard win controls.

Question and Answer Database

FAQ2716D.txt Changing font color on standard win controls.

Category :VCL

Platform :All

Product :All 32 bit

Question:

Why do changes to the StatusBar's font color not have any affect?

Answer:

The status bar is a standard Windows control, and as such, displays the font in the `clBtnText` value, which is set via the Control Panel. This color is black by default, but it can vary due to the user's color scheme. Other standard Windows controls, such as buttons, exhibit this identical behavior. The StatusBar and its associated panels have an owner-draw capability that allow you to draw text in colors you want.

Example:

```
procedure TForm1.StatusBar1DrawPanel(StatusBar: TStatusBar;
  Panel: TStatusPanel; const Rect: TRect);
begin
  if Panel = StatusBar.Panels[0] then begin
    StatusBar.Canvas.Font.Color := clRed;
    StatusBar.Canvas.TextOut(Rect.Left, Rect.Top, 'Panel - 0')
  end else begin
    StatusBar.Canvas.Font.Color := clGreen;
    StatusBar.Canvas.TextOut(Rect.Left, Rect.Top, 'Panel - 1');
  end;
end;
```

7/16/98 4:31:28 PM



## Article #17720: Typecasting a PChar to a longint.

Question and Answer Database

FAQ2720D.txt    Typecasting a PChar to a longint.

Category       :Windows API

Platform       :All

Product        :All 32 bit

Question:

Many Windows functions claim to want PChar parameters in the documentation, but they are defined as requiring LongInts. Is this a bug?

Answer:

No, this is where "typecasting" is used. Typecasting allows you to fool the compiler into thinking that one type of variable is of another type for the ultimate in flexibility. The last parameter of the Windows API function SendMessage() is a good example. It is documented as requiring a long integer, but commonly requires a PChar for some messages (WM\_WININICHANGE). Generally, the variable you are typecasting from must be the same size as the variable type you are casting it to. In the SendMessage example, you could typecast a PChar as a longint, since both occupy 4 bytes of memory:

Example:

```
var
  s : array[0..64] of char;
begin
  StrCopy(S, 'windows');
  SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(@S));
end;
```

Pointers are the easiest to typecast, and are the most flexible since you can pass anything to the called procedure, and the most dangerous for the same reason. You can also use untyped variable parameters for convenience, although var parameters are really pointers behind the scenes.

Example:

```

type
  PBigArray = ^TBigArray;
  TBigArray = Array[0..65000] of char;

procedure ZeroOutPointerVariable(P : pointer; size : integer);
var
  i : integer;

begin
  for i := 0 to size - 1 do
    PBigArray(p)^[i] := #0;
  end;

procedure ZeroOutUntypedVariable(var v, size : integer);
var
  i : integer;

begin
  for i := 0 to size - 1 do
    TBigArray(v)[i] := #0;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  s : string[255];
  i : integer;
  l : longint;

begin
  s := 'Hello';
  i := 10;
  l := 2000;

  ZeroOutPointerVariable(@s, sizeof(s));
  ZeroOutPointerVariable(@i, sizeof(i));
  ZeroOutPointerVariable(@l, sizeof(l));

  s := 'Hello';
  i := 10;
  l := 2000;

  ZeroOutUntypedVariable(s, sizeof(s));
  ZeroOutUntypedVariable(i, sizeof(i));
  ZeroOutUntypedVariable(l, sizeof(l));

```

end;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17724: How do I use Paths and Geometric Pens?

Question and Answer Database

FAQ2724D.txt How do I use Paths and Geometric Pens?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I use Paths and Geometric Pens?

Answer:

You will need to call the Windows API function `ExtCreatePen()` to create a geometric pen style, and use the `Canvas.Handle` property to use the Windows Path functions.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  lb : TLogBrush;
begin
  lb.lbStyle := BS_SOLID;
  lb.lbColor := RGB(255, 0, 0);
  lb.lbHatch := 0;
  Form1.Canvas.Pen.Handle :=
    ExtCreatePen(PS_GEOMETRIC or
                PS_INSIDEFRAME or
                PS_ENDCAP_SQUARE or
                PS_JOIN_BEVEL,
                20,
                lb,
                0,
                nil);
  BeginPath(Form1.Canvas.Handle);
  Form1.Canvas.MoveTo(100, 30);
  Form1.Canvas.LineTo(200, 100);
  Form1.Canvas.LineTo(300, 30);
  EndPath(Form1.Canvas.Handle);
  StrokePath(Form1.Canvas.Handle);
```

```
Form1.Canvas.Pen.Width := 1;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17728: How do I convert a color to a shade of gray?

Question and Answer Database

FAQ2728D.txt How do I convert a color to a shade of gray?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I convert a color to a shade of gray?

Answer:

The following example shows how to convert an RGB color to the equivalent color of gray using the same method that a black and white television would render a color broadcast:

```
function RgbToGray(RGBColor : TColor) : TColor;
var
  Gray : byte;
begin
  Gray := Round((0.30 * GetRValue(RGBColor)) +
                (0.59 * GetGValue(RGBColor)) +
                (0.11 * GetBValue(RGBColor)));
  Result := RGB(Gray, Gray, Gray);
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Shape1.Brush.Color := RGB(255, 64, 64);
  Shape2.Brush.Color := RgbToGray(Shape1.Brush.Color);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17732: Overcomming Stream read error.

Question and Answer Database

FAQ2732D.txt Overcomming Stream read error.

Category :VCL

Platform :All

Product :All 32 bit

Question:

When I start 16 Bit Delphi, I get the error "Stream Read Error" or "Invalid Stream Format". How can I get Delphi to start properly?

Answer:

This is usually caused by a corrupt "Delphi.dmt" file located in Delphi's \Bin\ directory. To replace the file, simply copy the file:

D:\DELPHI16\RUNIMAGE\Delphi\BIN\Delphi.DMT

from the CD to Delphi's \Bin\ directory, then clear the read only attribute from the file.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17736: Drawing in scale without using Windows Mapping Modes?

Question and Answer Database

FAQ2736D.txt Drawing in scale without using Windows Mapping Modes?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I draw in scale without using Windows Mapping Modes?

Answer:

You will need to use the Windows API function `GetDeviceCaps()` to retrieve the "logical" number of pixels per inch for the display in both the X and y directions, and multiply your coordinates by the returned values. Screen measurements are based on a concept of "Logical Pixels" and may not always reflect a true measurement. Printers should give satisfactory results when measurement are compared against a ruler. When printing, delete the lines for `GetDc()` and `ReleaseDc()` and simply pass `Printer.Canvas.Handle` to the `GetDeviceCaps()` function.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dc : hdc;
  InchX : integer;
  InchY : integer;
begin
  dc := GetDc(0);
  InchX := GetDeviceCaps(dc, LOGPIXELSX);
  InchY := GetDeviceCaps(dc, LOGPIXELSY);
  ReleaseDC(0, dc);
  Form1.Canvas.Rectangle(Round(0 * InchX),
                        Round(0 * InchY),
                        Round(1.5 * InchX),
                        Round(1.5 * InchY));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17740: How can I eject a CD-ROM in code?

Question and Answer Database

FAQ2740D.txt    How can I eject a CD-ROM in code?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How can I eject a CD-ROM in code?

Answer:

You can use the Windows API function `GetDriveType()` to test if the drive is a CD-ROM drive then use the `TMediaPlayer` (even if the CD is not an Audio CD) to eject the CD.

Example:

```
function IsDriveCD(Drive : char) : longbool;
var
  DrivePath : string;
begin
  DrivePath := Drive + ':\';
  result := LongBool(GetDriveType(PChar(DrivePath)) and DRIVE_CDROM);
end;

function EjectCD(Drive : char) : bool;
var
  mp : TMediaPlayer;
begin
  result := false;
  Application.ProcessMessages;
  if not IsDriveCD(Drive) then exit;
  mp := TMediaPlayer.Create(nil);
  mp.Visible := false;
  mp.Parent := Application.MainForm;
  mp.Shareable := true;
  mp.DeviceType := dtCDAudio;
  mp.FileName := Drive + ':\';
  mp.Open;
```

```
Application.ProcessMessages;  
mp.Eject;  
Application.ProcessMessages;  
mp.Close;  
Application.ProcessMessages;  
mp.free;  
result := true;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if not EjectCD('D') then  
        ShowMessage('Not A CD Drive');  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17744: Extracting RGB components of a color.

Question and Answer Database

FAQ2744D.txt     Extracting RGB components of a color.

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I extract the Red, Green, and Blue components of a given color?

Answer:

Use the Window API functions `GetRValue()`, `GetGValue()`, and `GetBValue()`.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Canvas.Pen.Color := clRed;
  Mem1.Lines.Add('Red := ' +
                 IntToStr(GetRValue(Form1.Canvas.Pen.Color)));
  Mem1.Lines.Add('Red := ' +
                 IntToStr(GetGValue(Form1.Canvas.Pen.Color)));
  Mem1.Lines.Add('Blue := ' +
                 IntToStr(GetBValue(Form1.Canvas.Pen.Color)));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17748: How do I create an icon from a bitmap?

Question and Answer Database

FAQ2748D.txt     How do I create an icon from a bitmap?

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I create an icon from an bitmap?

Answer:

You must create two bitmaps, a mask bitmap (called the "AND" bitmap) and a image bitmap (called the XOR bitmap). You can pass the handles to the "AND" and "XOR" bitmaps to the Windows API function `CreateIconIndirect()` and use the returned icon handle in your application.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  IconSizeX : integer;
  IconSizeY : integer;
  AndMask : TBitmap;
  XOrMask : TBitmap;
  IconInfo : TIconInfo;
  Icon : TIcon;
begin
  {Get the icon size}
  IconSizeX := GetSystemMetrics(SM_CXICON);
  IconSizeY := GetSystemMetrics(SM_CYICON);

  {Create the "And" mask}
  AndMask := TBitmap.Create;
  AndMask.Monochrome := true;
  AndMask.Width := IconSizeX;
  AndMask.Height := IconSizeY;

  {Draw on the "And" mask}
```



```

AndMask.Canvas.Brush.Color := clWhite;
AndMask.Canvas.FillRect(Rect(0, 0, IconSizeX, IconSizeY));
AndMask.Canvas.Brush.Color := clBlack;
AndMask.Canvas.Ellipse(4, 4, IconSizeX - 4, IconSizeY - 4);

{Draw as a test}
Form1.Canvas.Draw(IconSizeX * 2, IconSizeY, AndMask);

{Create the "XOr" mask}
XOrMask := TBitmap.Create;
XOrMask.Width := IconSizeX;
XOrMask.Height := IconSizeY;

{Draw on the "XOr" mask}
XOrMask.Canvas.Brush.Color := ClBlack;
XOrMask.Canvas.FillRect(Rect(0, 0, IconSizeX, IconSizeY));
XOrMask.Canvas.Pen.Color := clRed;
XOrMask.Canvas.Brush.Color := clRed;
XOrMask.Canvas.Ellipse(4, 4, IconSizeX - 4, IconSizeY - 4);

{Draw as a test}
Form1.Canvas.Draw(IconSizeX * 4, IconSizeY, XOrMask);

{Create a icon}
Icon := TIcon.Create;
IconInfo.fIcon := true;
IconInfo.xHotspot := 0;
IconInfo.yHotspot := 0;
IconInfo.hbmMask := AndMask.Handle;
IconInfo.hbmColor := XOrMask.Handle;
Icon.Handle := CreateIconIndirect(IconInfo);

{Destroy the temporary bitmaps}
AndMask.Free;
XOrMask.Free;

{Draw as a test}
Form1.Canvas.Draw(IconSizeX * 6, IconSizeY, Icon);

{Assign the application icon}
Application.Icon := Icon;

{Force a repaint}
InvalidateRect(Application.Handle, nil, true);

```

```
{Free the icon}  
  Icon.Free;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 21-MAR-00

## Article #17752: How do I create a bitmap from an icon?

Question and Answer Database

FAQ2752D.txt     How do I create a bitmap from an icon?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I create a bitmap from an icon?

Answer:

Use `Bitmap.Canvas.Draw` to draw the icon on the `Bitmap`.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TheIcon : TIcon;
  TheBitmap : TBitmap;
begin
  TheIcon := TIcon.Create;
  TheIcon.LoadFromFile('C:\Program Files\Borland\IcoCur32\EARTH.ICO');
  TheBitmap := TBitmap.Create;
  TheBitmap.Height := TheIcon.Height;
  TheBitmap.Width := TheIcon.Width;
  TheBitmap.Canvas.Draw(0, 0, TheIcon);
  Form1.Canvas.Draw(10, 10, TheBitmap);
  TheBitmap.Free;
  TheIcon.Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17756: Does my CD-ROM Drive contain an Audio CD?

Question and Answer Database

FAQ2756D.txt Does my CD-ROM Drive contain an Audio CD?  
Category :VCL  
Platform :All  
Product :All 32 bit

Question:

How can I tell if a given CD-ROM Drive contains an Audio CD?

Answer:

You can use the Windows API function `GetDriveType()` to test if the drive is a CD-ROM drive then use the Windows API function `GetVolumeInformation()` to test if the VolumeName is 'Audio CD'.

Example:

```
function IsAudioCD(Drive : char) : bool;
var
  DrivePath : string;
  MaximumComponentLength : DWORD;
  FileSystemFlags : DWORD;
  VolumeName : string;
begin
  Result := false;
  DrivePath := Drive + ':\';
  if GetDriveType(PChar(DrivePath)) <> DRIVE_CDROM then exit;
  SetLength(VolumeName, 64);
  GetVolumeInformation(PChar(DrivePath),
                      PChar(VolumeName),
                      Length(VolumeName),
                      nil,
                      MaximumComponentLength,
                      FileSystemFlags,
                      nil,
                      0);
  if lStrCmp(PChar(VolumeName), 'Audio CD') = 0 then result := true;
```

```
end;

function PlayAudioCD(Drive : char) : bool;
var
  mp : TMediaPlayer;
begin
  result := false;
  Application.ProcessMessages;
  if not IsAudioCD(Drive) then exit;
  mp := TMediaPlayer.Create(nil);
  mp.Visible := false;
  mp.Parent := Application.MainForm;
  mp.Shareable := true;
  mp.DeviceType := dtCDAudio;
  mp.FileName := Drive + ':';
  mp.Shareable := true;
  mp.Open;
  Application.ProcessMessages;
  mp.Play;
  Application.ProcessMessages;
  mp.Close;
  Application.ProcessMessages;
  mp.free;
  result := true;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if not PlayAudioCD('D') then
    ShowMessage('Not an Audio CD');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17759: How can I tell if the mouse is over my form?

Question and Answer Database

FAQ2759D.txt How can I tell if the mouse is over my form?

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I tell if the mouse is over my form?

Answer:

You can use the `GetCapture()` windows API function to capture the mouse. See the Windows documentation for additional information and limitations on mouse capturing.

Example:

```
procedure TForm1.FormDeactivate(Sender: TObject);
begin
    ReleaseCapture;
end;
```

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
begin
    If GetCapture = 0 then
        SetCapture(Form1.Handle);
    if PtInRect(Rect(Form1.Left,
                    Form1.Top,
                    Form1.Left + Form1.Width,
                    Form1.Top + Form1.Height),
                ClientToScreen(Point(x, y))) then
        Form1.Caption := 'Mouse is over form' else
        Form1.Caption := 'Mouse is outside of form';
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17763: Is my machine connected to the network.

Question and Answer Database

FAQ2763D.txt Is my machine connected to the network.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can my application tell if the machine is connected to a network under Windows 95?

Answer:

You can call the Windows API function `GetSystemMetrics()` and pass the `SM_NETWORK` flag.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if GetSystemMetrics(SM_NETWORK) AND $01 = $01 then
    ShowMessage('Machine is attached to network') else
    ShowMessage('Machine is not attached to network');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17767: Storing icons in your application for use at runtime?

Question and Answer Database

FAQ2767D.txt Storing icons in your application for use at runtime?

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I store icons in my application and use them at runtime?

Answer:

Create a resource file containing the icons you wish to use, then use the Windows API function LoadIcon(),

Example:

```
{ $R MYRES.RES }
```

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    h : hIcon;  
begin  
    h := LoadIcon(hInstance, 'ICON_1');  
    Application.Icon.Handle := h;  
    InvalidateRect(Application.Handle, nil, true);  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17771: How do I draw using LoEnglish Mapping Modes?

Question and Answer Database

FAQ2771D.txt      How do I draw using LoEnglish Mapping Modes?

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I draw using LoEnglish Mapping Modes?

Answer:

You will need to call the Windows API functions `SetMapMode()` and `SetViewportOrgEx()` to change the mapping mode and the origin, then draw using coordinates that are oriented from the bottom of the canvas.

Notes:

- 1) Change the drawing mode back as soon as you are finished drawing, or you may get unexpected results.
- 2) Screen measurements are based on a concept of "Logical Pixels" and may not always reflect a true measurement. Printers should give satisfactory results when measurement are compared against a ruler.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldMapMode : integer;
  OldOrigin : TSize;
  Inch : integer;
begin
  Inch := 100;
  with Form1.Canvas do begin
    SetViewportOrgEx(Handle,
                     0,
                     ClientHeight,
                     @OldOrigin);
```

```
OldMapMode := SetMapMode(Handle,  
                           MM_LOENGLISH);  
Rectangle(1 * inch, 1 * inch,  
          0 * inch, 0 * inch);  
SetMapMode(Handle,  
            OldMapMode);  
SetViewportExtEx(Handle,  
                  OldOrigin.cx,  
                  OldOrigin.cy,  
                  nil);
```

end;

end;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17774: Non VCL Bitmap buffering

Question and Answer Database

FAQ2774D.txt Non VCL Bitmap buffering

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I create a non VCL memory display context and bitmap for additional drawing performance?

Answer:

You will need to use low level Windows API functions.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dc : hdc;
  MemDc : hdc;
  MemBitmap : hBitmap;
  OldMemBitmap : hBitmap;
begin
  {Get the handle to the screen's dc}
  dc := GetDc(0);

  {Create and retrieve a handle to a memory dc based on the screen}
  MemDc := CreateCompatibleDc(dc);

  {Create a bitmap that is compatible with the display.}
  {Note: if you pass "MemDc" to CreateCompatibleBitmap()}
  {instead of "dc", you will get a monochrome bitmap!}
  MemBitmap := CreateCompatibleBitmap(dc, 100, 100);

  {Release the screen dc}
  ReleaseDc(0, dc);

  {Select the bitmap surface into the MemDc}
```

```
{remembering the default bitmap}
  OldMemBitmap := SelectObject(MemDc, MemBitmap);

{Draw on the MemoryDc}
  PatBlt(MemDc, 0, 0, 100, 100, WHITENESS);
  Ellipse(MemDc, 0, 0, 100, 100);

{Copy the MemDc to the Form Canvas}
  BitBlt(Form1.Canvas.Handle,
        100, 100,
        100, 100,
        MemDc,
        0, 0,
        SRCCOPY);

{Select the default bitmap back into the memory dc}
  SelectObject(MemDc, OldMemBitmap);

{Note: You can now use the memory bitmap handle with}
{functions such as GetDiBits()}

{Delete the Memory Bitmap}
  DeleteObject(MemBitmap);

{Delete the MemoryDc}
  DeleteDc(MemDc);

end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17777: Positioning the caret in an edit control.

Question and Answer Database

FAQ2777D.txt    Positioning the caret in an edit control.

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I position the cursor (caret) in an edit or similar type control?

Answer:

You can use the Delphi SelStart() and SelectLength() methods.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.SetFocus;
    {Move the caret to the second character}
    Edit1.SelStart := 2;
    {Do not select any text}
    Edit1.SelLength := 0;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17780: Color problems in 256 color mode.

Question and Answer Database

FAQ2780D.txt    Color problems in 256 color mode.

Category        :VCL

Platform        :All

Product         :All 32 bit

### Question:

When I am drawing graphics with my application in 256 color mode (or less), some pen and text colors do not match my brush colors.

### Answer:

Unless you are using a palette, Windows will attempt to convert pen and text colors to the nearest pure color available. You can determine the nearest pure color, a given color will map to, by calling the Windows API function `GetNearestColor()`. If you wish to use dithered colors for a pen, you can also set the pen style to the `psInsideFrame` style. When using this style, the pen draws lines within the frame of closed shapes that specify a bounding rectangle instead of centering the line over the shape. For the `psInsideFrame` style to have an effect, the pen width must be larger than 1 pixel.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var NearestColor : TColorRef;
begin
  Form1.Canvas.Brush.Color := RGB(143, 53, 22);
  Form1.Canvas.Pen.Color := RGB(143, 53, 22);
  Form1.Canvas.Pen.Width := 10;
  Form1.Canvas.Ellipse(10, 10, 100, 100);
  Form1.Canvas.Pen.Style := psInsideFrame;
  Form1.Canvas.Ellipse(10, 110, 100, 200);
  Mem1.Lines.Add(
    'The nearest displayable color to RGB(143, 53, 22) is:');
  NearestColor :=
```

```
    GetNearestColor(Form1.Canvas.Handle,  
                    RGB(143, 53, 22));  
Mem1.Lines.Add('Red := ' +  
               IntToStr(GetRValue(NearestColor)));  
Mem1.Lines.Add('Red := ' +  
               IntToStr(GetGValue(NearestColor)));  
Mem1.Lines.Add('Blue := ' +  
               IntToStr(GetBValue(NearestColor)));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17784: How do I recurse sub directories?

Question and Answer Database

FAQ2784D.txt    How do I recurse sub directories?

Category       :Object Pascal

Platform        :All

Product         :All 32 bit

Question:

How do I recurse sub directories?

Answer:

You will need to use Delphi's FindFirst() and FindNext() procedures to create a list of all the sub directories in a directory. For each sub directory, you will need to use Delphi's FindFirst() and FindNext() procedures, again to add any sub directories of the found directories to the list. Continue working down the list until all the directories have been processed.

Example:

```
procedure GetDirectories(const DirStr : string;
                        ListBox : TListBox);

var
  DirInfo: TSearchRec;
  r : Integer;
begin
  r := FindFirst(DirStr + '\*.*', FaDirectory, DirInfo);
  while r = 0 do begin
    Application.ProcessMessages;
    if ((DirInfo.Attr and FaDirectory = FaDirectory) and
        (DirInfo.Name <> '.') and
        (DirInfo.Name <> '..')) then
      ListBox.Items.Add(DirStr + '\' + DirInfo.Name);
    r := FindNext(DirInfo);
  end;
  SysUtils.FindClose(DirInfo);
end;

procedure GetFiles(const DirStr : string;
```

```
        ListBox : TListBox);
```

```
var
  DirInfo: TSearchRec;
  r : Integer;
begin
  r := FindFirst(DirStr + '\*.*', FaAnyfile, DirInfo);
  while r = 0 do begin
    Application.ProcessMessages;
    if ((DirInfo.Attr and FaDirectory <> FaDirectory) and
        (DirInfo.Attr and FaVolumeId <> FaVolumeID)) then
      ListBox.Items.Add(DirStr + '\' + DirInfo.Name);
    r := FindNext(DirInfo);
  end;
  SysUtils.FindClose(DirInfo);
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
var
  i : integer;
begin
  ListBox1.Items.Clear;
  ListBox2.Items.Clear;
  ListBox1.Items.Add('C:\Delphi');
  GetDirectories('C:\Delphi', ListBox1);
  i := 1;
  while i < ListBox1.Items.Count do begin
    GetDirectories(ListBox1.Items[i], ListBox1);
    Inc(i);
  end;
end;
```

```
procedure TForm1.ListBox1Click(Sender: TObject);
```

```
begin
  ListBox2.Clear;
  GetFiles(ListBox1.Items[ListBox1.ItemIndex],
           ListBox2);
end;
```

Note: Recursing directories can take up a lot of memory. It is suggested that you consider creating a linked list or a temporary file to store your list entries instead of using a component (such as a Memo or StringList), as there are limits to the number of entries these components can have.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17787: Sending a file to the Windows Recycle Bin?

Question and Answer Database

FAQ2787D.txt Sending a file to the Windows Recycle Bin?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I send a file to the Windows Recycle Bin?

Answer:

Use the ShellAPI function SHFileOperation().

```
uses ShellAPI;
```

```
procedure SendToRecycleBin(FileName: string);
```

```
var
```

```
    SHF: TSHFileOpStruct;
```

```
begin
```

```
    with SHF do begin
```

```
        Wnd := Application.Handle;
```

```
        wFunc := FO_DELETE;
```

```
        pFrom := PChar(FileName);
```

```
        fFlags := FOF_SILENT or FOF_ALLOWUNDO;
```

```
    end;
```

```
    SHFileOperation(SHF);
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    SendToRecycleBin('c:\Download\Test.gif');
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17791: Justifying edit text.

Question and Answer Database

FAQ2791D.txt Justifying edit text.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I create a single line edit control that is right or centered justified?

Answer:

You can use a TMemo component, since the TEdit component does not directly support center and right justification. In addition, you will need to prevent the user from pressing the Enter, Ctrl-Enter, and a variety of arrow key combinations to prevent more than one line to be added to the memo. This can be accomplished by scanning the TMemo's text string for carriage return and line feed characters and deleting them during the TMemo's Change and KeyPress events. Optionally, you could replace any carriage returns with spaces to accommodate a multi-line paste operation.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Memo1.Alignment := taRightJustify;
    Memo1.MaxLength := 24;
    Memo1.WantReturns := false;
    Memo1.WordWrap := false;
end;
```

```
procedure MultiLineMemoToSingleLine(Memo : TMemo);
var
    t : string;
begin
    t := Memo.Text;
    if Pos(#13, t) > 0 then begin
```

```
while Pos(#13, t) > 0 do
  delete(t, Pos(#13, t), 1);
while Pos(#10, t) > 0 do
  delete(t, Pos(#10, t), 1);
Memo.Text := t;
end;
end;

procedure TForm1.Memo1Change(Sender: TObject);
begin
  MultiLineMemoToSingleLine(Memo1);
end;

procedure TForm1.Memo1KeyPress(Sender: TObject; var Key: Char);
begin
  MultiLineMemoToSingleLine(Memo1);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17798: Read and BlockRead file function failure in 32bit.

Question and Answer Database

FAQ2798D.txt Read and BlockRead file function failure in 32bit.

Category :Object Pascal

Platform :All

Product :All 32 bit

### Question:

When I port my code from 16 bit Delphi to 32 bit Delphi, the read and BlockRead function no longer seem to work correctly.

Is this a bug?

### Answer:

No. In 32 bit environments, records are automatically aligned on 32 bit boundaries when the \$A+ compiler directive is in effect. This is done by padding the record so each record will be located on a 32 bit boundary. If you are reading in data from a file containing non-aligned records, the data may not be read in correctly if you are using automatic record alignment. To turn off automatic record alignment for a given record type, use the "packed" record attribute. Globally turning off record alignment with the \$A- is not recommended, as much of the system requires record alignment to be on. The following example demonstrates the size difference of a packed and unpacked record.

### Example:

```
type TSomeRec = record
    w1 : word;
    b1 : byte;
    w2 : word;
end;
```

```
type TSomePackedRec = packed record
    w1 : word;
    b1 : byte;
    w2 : word;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r : TSomeRec;
  s : TSomePackedRec;
begin
  ShowMessage(IntToStr(SizeOf(r))); {displays 6}
  ShowMessage(IntToStr(SizeOf(s))); {displays 5};
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17800: Record alignment in 32bit pascal.

Question and Answer Database

FAQ2800D.txt Record alignment in 32bit pascal.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

When I try to call a function in a DLL and pass a record, the field members are not aligned in the same way the dll expects. The DLL was constructed with a C compiler using record member alignment. How can I achieve a like record alignment with Delphi?

Answer:

The following example demonstrates padding a record to 32 bit boundaries. Adjustments will need to be made to interface with the compiler directives that were in place when the dll was built, as there are differing rules that can be in force during compile time in regards to record alignment. Some compilers may have options to (for instance) guarantee that every field is not only 32 bit aligned, but each field will also be located on an even 32 bit boundary. It may even be possible that the given compiler guarantees that the entire record is aligned to an even number of 32 bit "chunks".

```
type SomeAlignedRecord = {$IFDEF WIN32} packed {$ENDIF} record
    AByteField : byte;
    {add three bytes to align field to 32 bits}
    UnusedPaddingA : array[1..3] of byte;
    AWordField : word;
    {add two bytes to align field to 32 bits}
    UnusedPaddingB : array[1..2] of byte;
    ACharField : char;
    {add three bytes to align field to 32 bits}
    UnusedPaddingC : array[1..3] of byte;
    {align record to an even number of 32 bit chunks}
    FinalUnusedPadding : array[1..4] of byte;
end;
```

Last Modified: 01-SEP-99

## Article #17804: Calling windows RegisterClass function.

Question and Answer Database

FAQ2804D.txt Calling windows RegisterClass function.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

I want to call the Windows RegisterClass() function, but I get the error "Incompatible types: 'TPersistentClass' and 'TWndClassA'" when compiling. How can I get around this?

Answer:

The RegisterClass() function is declared both in the Classes unit and the Windows unit. To call the Windows version, simply prefix the call with the unit name.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
  wc : TWndClass;
begin
  Windows.RegisterClass(wc)
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17807: Sending keystroke #255 to a windows control.

Question and Answer Database

FAQ2807D.txt      Sending keystroke #255 to a windows control.

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Since the `keybd_event()` function only accepts key values up to 254, how can I send the keystroke #255 to a Windows control?

Answer:

This can be very useful for entering a value in an edit control for a foreign, symbol, or wingding character. The following example shows how to stuff the keyboard with the #255 character and direct it to any Window control. Note: This method should not be used to enter characters that could normally be processed by the `keybd_event()` function.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  KeyData : packed record
    RepeatCount : word;
    ScanCode : byte;
    Bits : byte;
  end;
begin
  {Let the button repaint}
  Application.ProcessMessages;
  {Set the focus to the window}
  Edit1.SetFocus;
  {Send a right so the char is added to the end of the line}
  SimulateKeyStroke(VK_RIGHT, 0);
  {Let the app get the message}
  Application.ProcessMessages;
  FillChar(KeyData, sizeof(KeyData), #0);
  KeyData.ScanCode := 255;
  KeyData.RepeatCount := 1;
```

```
SendMessage(Edit1.Handle, WM_KEYDOWN, 255,
    LongInt(KeyData));
KeyData.Bits := KeyData.Bits or (1 shl 30);
KeyData.Bits := KeyData.Bits or (1 shl 31);
SendMessage(Edit1.Handle, WM_KEYUP, 255,
    LongInt(KeyData));
KeyData.Bits := KeyData.Bits and not (1 shl 30);
KeyData.Bits := KeyData.Bits and not (1 shl 31);
SendMessage(Edit1.Handle, WM_CHAR, 255,
    LongInt(KeyData));
Application.ProcessMessages;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17810: TQuery RequestLive results in error "Table/View does not exist."

Question and Answer Database

FAQ2810C.txt TQuery RequestLive results in error "Table/View does not exist."

Category :Database Issues

Platform :Win95/NT

Product : C++Builder3.x

Question:

With Oracle the TQuery-RequestLive results in error "Table/View doesn't exist",

with Interbase it works.

How to use TQuery for update?

Answer:

All your tables and columns need to be uppercase when you change to RequestLive. This is the most common culprit when you see this message.

8/4/98 9:18:05 AM

Last Modified: 01-SEP-99

## Article #17811: Freeing an object stored in a list.

Question and Answer Database

FAQ2811D.txt Freeing an object stored in a list.

Category :VCL

Platform :All

Product :All 32 bit

Question:

When I add an object to a TStrings component, how should I free the object?

Answer:

Simply call the objects free method. The following example attaches a icon object to the strings property of a listbox during the forms creation, and frees the icon when the form is destroyed.

Example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Icon: TIcon;
begin
  Icon := TIcon.Create;
  Icon.LoadFromFile('C:\Program Files\BorlandImages\CONSTRUC.ICO');
  ListBox1.Items.AddObject('Item 0', Icon);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ListBox1.Items.Objects[0].Free;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17814: StrPCopy and long strings

Question and Answer Database

FAQ2814D.txt StrPCopy and long strings

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

StrPCopy does not seem to work with long strings. How can I get around this limitation?

Answer:

Use the StrPLCopy function instead.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17818: How can I draw directly on the Windows Desktop?

Question and Answer Database

FAQ2818D.txt     How can I draw directly on the Windows Desktop?

Category     :Windows API

Platform     :All

Product      :All 32 bit

Question:

How can I draw directly on the Windows Desktop?

Answer:

The following example uses the dc returned from `GetDc(0)` with the Windows API drawing functions to draw on the Windows Desktop.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dc : hdc;
begin
  dc := GetDc(0);
  MoveToEx(Dc, 0, 0, nil);
  LineTo(Dc, 300, 300);
  ReleaseDc(0, Dc);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17822: Copying a 400 table to a different type of table.

Question and Answer Database

FAQ2822D.txt - Copying a 400 table to a different type of table.

Category :AS400  
Platform :All-32Bit  
Product :

Question:

Is there a procedure that will allow me to copy  
a 400 Table to a different type of table?

Answer:

Yes. It is the 'CopyTable' Procedure(found within the  
TTable400).

4/1/99 2:45:17 PM

Last Modified: 01-SEP-99

## Article #17825: CO400 and AnyNet

Question and Answer Database

FAQ2825D.txt - CO400 and AnyNet

Category :AS400  
Platform :All-32Bit  
Product :

Question:

Can the ClientObjects/400 TCP/IP Version be run over Anynet?

Answer:

The Delphi/400 TCP/IP version connects to the AS/400 through the Windows WINSOC.DLL. Anynet is APPC over TCP/IP, so this approach requires using the SNA Protocol.

Note: Performance using this configuration is slow.

4/1/99 2:45:29 PM

Last Modified: 01-SEP-99

## Article #17828: Making ini file changes permanent.

Question and Answer Database

FAQ2828D.txt Making ini file changes permanent.

Category :Windows API

Platform :All

Product :All 32 bit

### Question:

After writing to a ini file, how can the ini file cache be flushed so the changes made are permanent?

### Answer:

Call the Windows API function `WriteProfileString()` or `WritePrivateProfileString()` and pass nil parameters for the section, key, and string values.

### Example:

```
WriteProfileString(nil, nil, nil);
```

```
WritePrivateProfileString(nil, nil, nil, FileName);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17831: Nested fields in DBGrid

Question and Answer Database

FAQ2831D.txt :Nested fields in DBGrid

Category :Database/VCL

Platform :Win95/NT

Product :Delphi4.x,

Question:

How do I display nested fields from an Oracle 8 table like SQL Explorer does?

Answer:

Add the TFields to the DataSource and nested fields will appear.

12/01/1998 1:45:05 PM

Last Modified: 01-SEP-99

## Article #17834: Covert string of (zeros and ones) to binary.

Question and Answer Database

FAQ2834D.txt Covert string of (zeros and ones) to binary.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I convert a string of zeros and ones to the integer it represents?

Answer:

The following example shows how to convert a "BinaryString" to a long integer by using the shl (shift bits left) macro.

Example:

```
function BinStringToLongInt(BinString : string) : longint;
var
  i : integer;
  Num : longint;
begin
  Num := 0;
  for i := 1 to length(BinString) do
    if BinString[i] = '1' then
      Num := (Num shl 1) + 1 else
      Num := (Num shl 1);
  Result := Num;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Mem1.Lines.Add(IntToStr(BinStringToLongInt('11111111')));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17836: Bad PC First port on line1

Question and Answer Database

FAQ2836D.txt - Bad PC First port on line1

Category :AS400  
Platform :All-32Bit  
Product :

Question:

What does the error "Bad PC First port on line1" installing with TCP/IP proptocol to Delphi/400?

Answer:

You must enter a port number in the the edit box labled "1st port pc". In the TCP/IP configuration for the AS/400, default values for the 1st pc port can read 5103, NB PC Port can read 5203.

4/1/99 2:45:42 PM

Last Modified: 01-SEP-99

## Article #17838: Finding the version info for OS/400

Question and Answer Database

FAQ2838D.txt - Finding the version info for OS/400

Category :AS400  
Platform :All-32Bit  
Product :

Question:

How can I find out which release of OS/400 I am using?

Answer:

On the 400 Command Prompt type:

'GO LICPGM' No quotes...

Choose option #10 (Display installed License programs)

Press F11 and look for Operating system/400 Entry.

4/1/99 2:45:51 PM

Last Modified: 01-SEP-99



## Article #17841: Linking an animated cursor to a resource file.

Question and Answer Database

FAQ2841D.txt Linking an animated cursor to a resource file.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How do I link an animated cursor to a resource file, then load it from my application?

Answer:

You will need to save the ani file into a res file as user data, (detailed in Borland's Technical information article Ti-3209 - "How to use a user defined resource"). Once you have successfully linked the resource to your application, you will need to save the data to a temporary file, then use the LoadImage command to get a handle to the animated cursor.

7/16/98 4:31:28 PM

Last Modified: 21-MAR-00

## Article #17845: DataSetFields don't display in Nested Table

Question and Answer Database

FAQ2845D.txt :DataSetFields don't display in Nested Table  
Category :Database/VCL  
Platform :Win95/NT  
Product :

Question:

How come fields from my dataset are not appearing in the DataSetFields property of my Nested Table component.

Answer:

Add the appropriate TFields to the DataSet and they will appear.

12/01/1998 1:45:42 PM

Last Modified: 01-SEP-99

## Article #17848: External Exception EEFACE when connecting

Question and Answer Database

FAQ2848C.txt External Exception EEFACE when connecting

Category :Database Issues

Platform :Win95/NT

Product : C++Builder1.0 C++Builder3.x

Question:

I have BDE 4.0 or BDE 4.51 and I am getting External Exception EEFACE when opening an MSACCESS Native Driver Alias in the BDE Administrator. How can I avoid this exception?

Answer:

Install BDE 5.0, which is available for download at:

<http://www.inprise.com/devsupport/bde/bdeupdate.html>

8/18/98 8:50:15 AM

Last Modified: 01-SEP-99

## Article #17849: Adding install to Win95 Add/Remove programs.

Question and Answer Database

FAQ2849D.txt Adding install to Win95 Add/Remove programs.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I get my program listed in the control panels  
"Add/Remove programs" applet?

Answer:

You can use Install Shield Express that ships with Delphi and C++ Builder, or you can add this in code using the TRegistry object to add a new key to the registry under the key:

```
HKEY_LOCAL_MACHINE\  
SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\  

```

The key that you add should be unique, and should reflect the name of your program: "My Program Name".

Under your new key, you will need to add two items: The name to display for your program in the control panel applet list, and the command line to your uninstall program:

Example:

```
uses Registry;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    reg: TRegistry;
```

```
begin
```

```
    reg := TRegistry.Create;
```

```
    reg.RootKey := HKEY_LOCAL_MACHINE;
```

```
    reg.LazyWrite := false;
```

```
    reg.OpenKey(  
        'SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\MyProgam',
```

```
    true);  
reg.WriteString('DisplayName', 'My Program');  
reg.WriteString('UninstallString',  
  'C:\Program Files\Borland\Delphi 3\MyUninstall.exe UninstalMe');  
reg.CloseKey;  
reg.free;  
end;
```

In addition to uninstalling you application, your uninstall program should delete the registry keys you added in the previous example.

Example:

```
uses Registry;  
  
procedure TForm1.Button1Click(Sender: TObject);  
var  
  reg: TRegistry;  
begin  
  reg := TRegistry.Create;  
  reg.RootKey := HKEY_LOCAL_MACHINE;  
  reg.LazyWrite := false;  
  reg.DeleteKey(  
'SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\MyProgam');  
  reg.free;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17852: Query with a string does not work properly

Question and Answer Database

FAQ2852D.txt - Query with a string does not work properly

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I am doing a Query against the 400 where I specify a string,  
example: `SELECT * FROM CUST WHERE CUSTNAME = "Sparky"`  
But it is not working properly. What am I doing wrong?

Answer:

The 400 likes Single Quotes and not double quotes.  
You should be doing:

```
SELECT * FROM CUST
WHERE CUSTNAME = 'Sparky'
```

4/1/99 2:46:02 PM

Last Modified: 01-SEP-99

## Article #17855: CO400INS library created, not CO400

Question and Answer Database

FAQ2855D.txt - CO400INS library created, not CO400

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

When I'm don installing Delphi/400 the CO400INS library created, instead of the CO400 library.

### Answer:

CO400INS is a temporary library for the installation of Delphi/400. At the conclusion of a normal install, the CO400 library has been installed and CO400INS has been removed. Early versions of ClientAccess data transfer and remote command functions may not work correctly and this can prevent the installation program from creating the CO400 library and transferring data. Check with IBM for the latest PTF's relating to data transfer and remote command functions.

4/1/99 2:46:14 PM

Last Modified: 01-SEP-99

## Article #17857: PtInRegion pixel function

Question and Answer Database

FAQ2857D.txt PtInRegion pixel function

Category :Windows API

Platform :All

Product :All 32 bit

### Question:

I am having trouble getting the Windows API function PtInRegion() to work correctly in Windows 3.1 and Windows 95 without generating an application error or GPF. What can I do to determine if a point is in a region reliably?

### Answer:

The Windows 3.1 and Windows 95 GDI heap is limited in regards to scan converting large and complex regions such as Ellipses, Polygons, and PolyPolygons. One "Brute Force" technique is to create an "in memory" monochrome bitmap and draw the required shape in black. You can then test to see if a given point in the bitmap is black or white. If the test pixel is black, then the given point is "in" the given region.

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bm : TBitmap;
begin
  bm := TBitmap.Create;
  bm.Monochrome := true;
  bm.Height := 100;
  bm.Width := 100;
  bm.Canvas.Brush.Color := clWhite;
  bm.Canvas.FillRect(Rect(0, 0, 100, 100));
  bm.Canvas.Brush.Color := clBlack;
  bm.Canvas.Pen.Color := clBlack;
  bm.Canvas.Ellipse(10, 10, 90, 90);
  if bm.Canvas.Pixels[20, 20] = clBlack then
    ShowMessage('Point : (20, 20) is in region') else
```



```
ShowMessage('Point : (20, 20) is not in region');  
if bm.Canvas.Pixels[50, 50] = clBlack then  
    ShowMessage('Point : (50, 50) is in region') else  
    ShowMessage('Point : (50, 50) is not in region');  
bm.Free;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17859: F004 Error

Question and Answer Database

FAQ2859D.txt - F004 Error

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I'm getting "F004" Error, when installing Delphi/400?

Answer:

This error occurs only when using the SNA Protocol.  
It has two meanings:

The SNA server is not started.

The license limit is reached (number of clients you declare when installing SNA on  
NT server.

4/1/99 2:46:29 PM

Last Modified: 01-SEP-99

## Article #17861: How can I close the CD Tray in code?

Question and Answer Database

FAQ2861D.txt    How can I close the CD Tray in code?

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How can I close the CD Tray in code?

Answer:

Make a call to the MMSystem function `mciSendCommand` sending the `MCI_SET_DOOR_CLOSED` command.

Example:

```
uses MMSystem;

procedure CloseCD(Drive : char);
var
  mp : TMediaPlayer;
begin
  result := false;
  Application.ProcessMessages;
  mp := TMediaPlayer.Create(nil);
  mp.Visible := false;
  mp.Parent := Application.MainForm;
  mp.Shareable := true;
  mp.DeviceType := dtCDAudio;
  mp.FileName := Drive + ':';
  mp.Open;
  Application.ProcessMessages;
  mciSendCommand(mp.DeviceID,
  MCI_SET, MCI_SET_DOOR_CLOSED, 0);
  Application.ProcessMessages;
  mp.Close;
  Application.ProcessMessages;
  mp.free;
  result := true;
end;
```

end;

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    CloseCD('D');  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17863: Setting break on exception in Delphi 4

Question and Answer Database

FAQ2863D.txt - Setting break on exception in Delphi 4

Category :Delphi 4.x  
Platform :All Windows  
Product :

Question:

Where is the "Break on Exception" check box in Delphi 4

Answer:

Follow menu: tools | Debugger Options | Language Exceptions | Stop on  
Delphi Exceptions

4/1/99 2:41:17 PM

Last Modified: 01-SEP-99

## Article #17865: Where is "View | Project Source"

Question and Answer Database

FAQ2865D.txt - Where is "View | Project Source"

Category :Delphi 4.x  
Platform :All-32Bit  
Product :Delphi4.x,

Question:

What happened to the "View | Project Source" menu item from Delphi 3?

Answer:

In order to view the project source code under Delphi 4 navigate to the "Project | View Source" menu item.

4/1/99 2:41:32 PM

Last Modified: 01-SEP-99

## Article #17867: Creating a file association for my application?

Question and Answer Database

FAQ2867D.txt Creating a file association for my application?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I create a file association for my application?

Answer:

In Win32, create a new registry entry under the HKEY\_CLASSES\_ROOT root key that points to the file extension, the command line to invoke, and the icon to display. Under Win16, simply file extension and the command line to invoke in the [Extensions] sections of Win.ini.

Example:

uses

Registry, {For Win32}

IniFiles; {For Win16}

{For Win32}

procedure TForm1.Button1Click(Sender: TObject);

var

reg: TRegistry;

begin

reg := TRegistry.Create;

reg.RootKey := HKEY\_CLASSES\_ROOT;

reg.LazyWrite := false;

{Add Program Support}

reg.OpenKey('.bor\shell\open\command',  
true);

{Invoke the program passing the file name as the first parameter}

reg.WriteString('',

'C:\Program Files\Borland\Delphi 3\Project1.exe %1');

{Add Icon Display}

reg.CloseKey;

```
reg.OpenKey('.bor\DefaultIcon',
            true);
{Use the first icon in the executable to display}
reg.WriteString('',
                'C:\Program Files\Borland\Delphi 3\Project1.exe,0');
reg.CloseKey;
reg.free;
end;
```

```
{For Win16}
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
var
    WinIni : TIniFile;
    WinIniFileName : array[0..MAX_PATH] of char;
    s : array[0..64] of char;
```

```
begin
    GetWindowsDirectory(WinIniFileName, sizeof(WinIniFileName));
    StrCat(WinIniFileName, '\win.ini');
    WinIni := TIniFile.Create(WinIniFileName);
    WinIni.WriteString('Extensions',
                      'bor',
                      'C:\PROGRA~1\BORLAND\DELPHI~1\PROJECT1.EXE ^.bor');
    WinIni.Free;
    StrCopy(S, 'Extensions');
    SendMessage(HWND_BROADCAST, WM_WININICHANGE,
                0, LongInt(@S));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17869: How to move Client Objects to a different AS/400

Question and Answer Database

FAQ2869D.txt - How to move Client Objects to a different AS/400

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How do I move Client Object to a different AS/400?

Answer:

Install ClientObjects/400 on the second machine.

On the old AS/400 run CO400UNS. CO400UNS will clear the validation key and provide a "Reference" to allow the ClientObjects/400 license to be transferred.

Get a new validation key from Inprise or the salesperson that provided the first validation key. To get this new key you will need to supply the "Reference" (provided by CO400UNS), the serial number and the model of the new AS/400.

Run CO400INS on the new AS/400 to enter the new key.

4/1/99 2:46:40 PM

Last Modified: 01-SEP-99

## Article #17871: Simulating a mouse move in code.

Question and Answer Database

FAQ2871D.txt Simulating a mouse move in code.

Category :VCL

Platform :All

Product :All 32 bit

### Question:

Some components do not respond to a cursor change until the user moves the mouse. Is there a way to simulate a mouse move?

### Answer:

The following example shows how to "nudge the mouse" without user intervention.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  pt : TPoint;
begin
  Application.ProcessMessages;
  Screen.Cursor := CrHourglass;
  GetCursorPos(pt);
  SetCursorPos(pt.x + 1, pt.y + 1);
  Application.ProcessMessages;
  SetCursorPos(pt.x - 1, pt.y - 1);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17873: Viewing currently loaded DLLs/Modules from within Delphi 4

Question and Answer Database

FAQ2873D.txt - Viewing currently loaded DLLs/Modules from within Delphi 4

Category :Delphi 4.x  
Platform :All-32Bit  
Product :Delphi4.x,

### Question:

Is there a way to view all the currently loaded DLLs on my system via Delphi 4?

### Answer:

Yes, this can be done through the IDE by navigating to the "View | Debug Windows | Modules" menu item. This will pop up a window of currently loaded modules (DLL's, packages, etc.)

4/1/99 2:41:44 PM

Last Modified: 01-SEP-99

## Article #17875: Delphi .dcu format

Question and Answer Database

FAQ2875D.txt Delphi .dcu format

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I decompile a DCU file, so I may modify it to work with later versions of Delphi / C++ Builder?

Answer:

The DCU file format is undocumented and may change from version to version. You will need the source code (.pas file) to modify/update the DCU.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17877: 'Setup cannot find any system', using SNA Server

Question and Answer Database

FAQ2877D.txt - 'Setup cannot find any system', using SNA Server

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I use SNA Server. When installing the AS/400 part of Delphi/400, the setup program displays error "Setup cannot find any system".

### Answer:

Check the configuration of SNA server on NT server :

For SNA server 2.11, verify the "APPC LU PROPERTIES" and select "Member of Default Outgoing local APPC lu pool".  
For SNA server 3.0, verify the "LOCAL APPC LUs" members, one of these must have the option "Member of Default Outgoing local APPC lu pool" selected.

4/1/99 2:43:02 PM

Last Modified: 01-SEP-99

## Article #17879: Detecting display changes.

Question and Answer Database

FAQ2879D.txt Detecting display changes.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How do I detect, and how should I handle dynamic changes in the display?

Answer:

To detect changes in the display, create a message handler to trap the WM\_DISPLAYCHANGE message. If you have a simple application, you may need to do little in response to this message. If your application makes use of graphics, you may want to restart the application, as there may be many changes to the system that may be difficult to overcome as the screen resolution, color depth and default fonts may change.

Here is an example of of a message handler:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
    procedure WMDisplayChange(var Message: TMessage);
      message WM_DISPLAYCHANGE;
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.WMDisplayChange(var Message: TMessage);  
begin  
  {Do Something here}  
  inherited;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17881: Executing and application by extention.

Question and Answer Database

FAQ2881D.txt Executing and application by extention.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I execute the application that is associated with a file extension?

Answer:

Use the ShellAPI function ShellExecute().

Example:

```
uses ShellAPI;

procedure TForm1.Button1Click(Sender: TObject);
var
  DocFileName : string;
  DocFileDir  : string;
begin
  DocFileName := 'C:\Download\somefile.htm';
  DocFileDir  := ExtractFileDir(DocFileName);
  ShellExecute(Form1.Handle,
               nil,
               pChar(DocFileName),
               nil,
               pChar(DocFileDir),
               SW_SHOWNORMAL);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17883: Handling command line params with spaces

Question and Answer Database

FAQ2883D.txt Handling command line params with spaces

Category :Object Pascal

Platform :All

Product :All 32 bit

### Question:

When I pass a file name containing spaces to my application, ParamStr(1) only shows the part of the filename before the first space. How can I retrieve file names containing spaces?

### Answer:

Since the space character is a delimiter for ParamStr, you will need to add together all the parameters that are passed to your application (using the ParamCount variable), or access the entire paramstring via the CmdLine variable, and parse out any additional commands. The following example shows how to get the entire command line including any embedded spaces:

### Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s : string;
begin
  s := StrPas(CmdLine);
  {$IFDEF WIN32}
  Delete(s, 1, Pos(' ' , s) + 1);
  {$ENDIF}
  ShowMessage(s);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17885: Transfer initialization (1) =error 55

Question and Answer Database

FAQ2885D.txt - Transfer initialization (1) =error 55

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

What causes the error Transfer initialization (1) =error 55

Answer:

This error occurs when are not logged in as QSECOFR and attempt to perform an operation with CO400 that requires QSECOFR rights.

4/1/99 2:47:26 PM

Last Modified: 01-SEP-99

## Article #17887: Creating new constructors

Question and Answer Database

FAQ2887D.txt    Creating new constructors

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

Is it possible to create a form that accepts additional parameters for the create method?

Answer:

Simply replace the Create constructor to your form class to accept additional parameters.

Example:

```
unit Unit2;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs;
```

```
type
```

```
  TForm2 = class(TForm)
```

```
  private
```

```
    { Private declarations }
```

```
  public
```

```
    constructor CreateWithCaption(aOwner: TComponent;  
                                  aCaption: string);
```

```
    { Public declarations }
```

```
  end;
```

```
var
```

```
  Form2: TForm2;
```

```
implementation
```

```
{ $R *.DFM }
```

```
constructor TForm2.CreateWithCaption(aOwner: TComponent;  
                                     aCaption: string);
```

```
begin  
    Create(aOwner);  
    Caption := aCaption;  
end;
```

```
uses Unit2;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin  
    Unit2.Form2 :=  
        Unit2.TForm2.CreateWithCaption(Application, 'My Caption');  
    Unit2.Form2.Show;  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17889: Reading dos environment variables

Question and Answer Database

FAQ2889D.txt    Reading dos environment variables

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I retrieve a DOS environment variable such as the current path?

Answer:

You can make a call to the Windows API function `GetDOSEnvironment()` under Win16 or the `GetEnvironmentStrings()` function under Win32.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  p : pChar;
begin
  Mem1.Lines.Clear;
  Mem1.WordWrap := false;
  {$IFDEF WIN32}
  p := GetEnvironmentStrings;
  {$ELSE}
  p := GetDOSEnvironment;
  {$ENDIF}
  while p^ <> #0 do begin
    Mem1.Lines.Add(StrPas(p));
    inc(p, lStrLen(p) + 1);
  end;
  {$IFDEF WIN32}
  FreeEnvironmentStrings(p);
  {$ENDIF}
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17891: Not authorized to use RSTLIB command

Question and Answer Database

FAQ2891D.txt - Not authorized to use RSTLIB command

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I'm getting the error  
"Not authorized to use RSTLIB command"?

Answer:

This error may occur when attempting to install ClientObjects/400 to the AS/400 when not connected using the QSECOFR profile. Verify that your logon and your initial connection (as seen in Client Access | AS/400 Connections, for instance) are QSECOFR.

4/1/99 2:47:48 PM

Last Modified: 01-SEP-99

## Article #17893: Protection Error

Question and Answer Database

FAQ2893D.txt - Protection Error

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I just upgraded my AS/400 OS from V3R6 to V3R7 and now I am getting a "Protection Error" whenever I try to access the AS/400 using CO400 from Delphi.

### Answer:

The error is raised when the Delphi/400 is installed to the AS/400 more than once. Re-run the CO400INS program and re-enter your license key.

4/1/99 3:02:19 PM

Last Modified: 01-SEP-99



## Article #17895: Getting the Lowest common denominator.

Question and Answer Database

FAQ2895D.txt Getting the Lowest common denominator.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How do I convert two numbers representing a ratio to a string that represents a fraction of the lowest common denominator?

Answer:

The following example demonstrates a technique based on equations originally solved by Euclid in approx. 300 B.C. :

Example:

```
function GetFracStr(Num1 : integer;
                   Num2 : integer) : string;
var
  N1 : integer;
  N2 : integer;
  Tmp : integer;
begin
  if Num1 < Num2 then begin
    N1 := Num1;
    N2 := Num2;
  end else begin
    N1 := Num2;
    N2 := Num1;
  end;
  Tmp := N2 mod N1;
  while Tmp <> 0 do begin
    N2 := N1;
    N1 := Tmp;
    Tmp := N2 mod N1;
  end;
  result := IntToStr(Num1 div N1) + '/' + IntToStr(Num2 div N1);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Memo1.Lines.Add(GetFracStr(2, 64));  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17897: Delphi/400 requires QSECOFR login

Question and Answer Database

FAQ2897D.txt - Delphi/400 requires QSECOFR login

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Why does the Client Objects AS/400 piece need to be installed by QSECOFR?

Answer:

Since QSECOFR uses the Security Officer profile, this user has access to all parts of the machine as well as all other user profiles. Delphi/400 needs access to specific parts of the AS/400 for the installation of their product. These areas on the 400 may be restricted to specific users due to security reasons, so it is required that you sign on as QSECOFR

4/1/99 3:02:31 PM

Last Modified: 01-SEP-99

## Article #17899: Placing multiple copies of a component on a form.

Question and Answer Database

FAQ2899D.txt Placing multiple copies of a component on a form.

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I paste multiple copies of a component onto a form from the component palette?

Answer:

Hold the Shift key down when you select the component from the palette. A blue outline will appear around the border of the component. Now click several times on the form in different locations. With each click, you will get a new copy of the component. To stop this action, single click on the original component on the component palette without holding the Shift key.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17901: How do I launch a control panel applet?

Question and Answer Database

FAQ2901D.txt    How do I launch a control panel applet?

Category       :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I launch a control panel applet?

Answer:

You can launch a control panel applet by calling WinExec to execute control.exe, passing the name of the control panel applet as a command line parameter. Applets files (.cpl) can be found in the Windows system directory. Note: Some of the applet files will not be found in the Windows system directory. Those applets can be executed simply by name.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    WinExec('C:\WINDOWS\CONTROL.EXE TIMEDATE.CPL',
           sw_ShowNormal);
    WinExec('C:\WINDOWS\CONTROL.EXE MOUSE',
           sw_ShowNormal);
    WinExec('C:\WINDOWS\CONTROL.EXE PRINTERS',
           sw_ShowNormal);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17903: APPC Allocate Error No retry

Question and Answer Database

FAQ2903D.txt - APPC Allocate Error No retry

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I get "APPC Allocate Error No retry"  
when setting TAS400.Active to True?

Answer:

The APPC router is not configured properly.  
You must have E32APPC.DLL in \windows\system.

4/1/99 3:02:47 PM

Last Modified: 01-SEP-99

## Article #17905: Integer to 'Binary String'

Question and Answer Database

FAQ2905D.txt Integer to 'Binary String'

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I convert an integer to a string of zeros and ones representing the binary values of the bits?

Answer:

The following example shows how to convert an integer to a "BinaryString" of zeros and ones by using the "shl" (shift bits left) macro to isolate the bits, and the "and" operator to test the bits.

Example:

```
function LongIntToBinString(BinValue : longint) : string;
```

```
var
  i : integer;
  s : string;
begin
  s := '';
  for i := 31 downto 0 do
    if (BinValue and (1 shl i)) <> 0 then
      s := s + '1' else
      s := s + '0';
  Result := s;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.Add(LongIntToBinString($FF));
end;
```





## Article #17907: Detecting PlugNPlay hardware changes?

Question and Answer Database

FAQ2907D.txt Detecting PlugNPlay hardware changes?

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I correctly detect PlugNPlay hardware changes?

Answer:

You should create a WM\_DEVICECHANGE message handler to detect changes in PlugNPlay hardware.

Example:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
    procedure WMDeviceChange(var Message: TMessage);
      message WM_DEVICECHANGE;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

const DBT_DEVICEARRIVAL = $8000;
const DBT_DEVICEQUERYREMOVE = $8001;
const DBT_DEVICEQUERYREMOVEFAILED = $8002;
const DBT_DEVICEREMOVEPENDING = $8003;
const DBT_DEVICEREMOVECOMPLETE = $8004;
```

```

const DBT_DEVICETYPESPECIFIC = $8005;
const DBT_CONFIGCHANGED = $0018;

procedure TForm1.WMDeviceChange(var Message: TMessage);
var
  s : string;
begin
{Do Something here}
  case Message.wParam of
    DBT_DEVICEARRIVAL :
      s := 'A device has been inserted and is now available';
    DBT_DEVICEQUERYREMOVE: begin
      s := 'Permission to remove a device is requested';
      ShowMessage(s);
      {True grants premission}
      Message.Result := integer(true);
      exit;
    end;
    DBT_DEVICEQUERYREMOVEFAILED :
      s := 'Request to remove a device has been canceled';
    DBT_DEVICEREMOVEPENDING :
      s := 'Device is about to be removed';
    DBT_DEVICEREMOVECOMPLETE :
      s := 'Device has been removed';
    DBT_DEVICETYPESPECIFIC :
      s := 'Device-specific event';
    DBT_CONFIGCHANGED :
      s:= 'Current configuration has changed'
  else s := 'Unknown Device Message';
end;
ShowMessage(s);
inherited;
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17909: Launching your default browser on a URL.

Question and Answer Database

FAQ2909D.txt Launching your default browser on a URL.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How do I start up and pass an URL to the default browser?

Answer:

Use the ShellAPI function ShellExecute.

Example:

```
uses ShellAPI;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    ShellExecute(Form1.Handle,  
                nil,  
                'http://www.borland.com',  
                nil,  
                nil,  
                SW_SHOWNORMAL);
```

```
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17911: Creating a Win95 Shell icon application.

Question and Answer Database

FAQ2911D.txt     Creating a Win95 Shell icon application.

Category        :Windows API

Platform        :All

Product         :All 32 bit

Question:

How do I create a Delphi/C++ Builder application that runs in the system tray?

Answer:

This is done via the ShellAPI function `Shell_NotifyIcon()`. The following example shows how to create an application that uses this function. The example application shows a different icon in the system tray every second and shows the time in the title. The application responds to a right click of the mouse by showing a menu allowing the user to display a form or quit the application. Attempts to close the form result in the form hiding, but the tray application continues to run. Note that no task bar icon is shown for the form. To compile the application, you will need to create a form with a `TPopupMenu` and a `TTimer` component. You will also need to make modifications to the main project source file (`TrayIt.dpr`).

Example:

```
{TrayIt.dpr}
```

```
program TrayIt;
```

```
uses
```

```
  Windows,
```

```
  Forms,
```

```
  TrayIt1 in 'TrayIt1.pas' {Form1};
```

```
{$R *.RES}
```

```
begin
```

```
  Application.Initialize;
```

```
Application.ShowMainForm := False;
Application.CreateForm(TForm1, Form1);
ShowWindow(Application.Handle, SW_HIDE);
Application.Run;
end.
```

```
{TrayIt1.pas}
```

```
unit TrayIt1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, Menus, ShellAPI, ExtCtrls;
```

```
type
```

```
TForm1 = class(TForm)
  PopupMenu1: TPopupMenu;
  Open1: TMenuItem;
  Exit1: TMenuItem;
  Timer1: TTimer;
  procedure FormCreate(Sender: TObject);
  procedure Open1Click(Sender: TObject);
  procedure Exit1Click(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure Timer1Timer(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
procedure WndProc(var Msg : TMessage); override;
```

```
public
```

```
{ Public declarations }
```

```
IconData : TNotifyIconData;
```

```
IconCount : integer;
```

```
end;
```

```
var
```

```
Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.WndProc(var Msg : TMessage);
```

```
var
```

```

    p : TPoint;
begin
    case Msg.Msg of
        WM_USER + 1:
            case Msg.lParam of
                WM_RBUTTONDOWN: begin
                    GetCursorPos(p);
                    PopupMenu1.Popup(p.x, p.y);
                end
            end;
        end;
    inherited;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    BorderIcons := [biSystemMenu];
    IconCount := 0;
    IconData.cbSize := sizeof(IconData);
    IconData.Wnd := Handle;
    IconData.uID := 100;
    IconData.uFlags := NIF_MESSAGE + NIF_ICON + NIF_TIP;
    IconData.uCallbackMessage := WM_USER + 1;
    IconData.hIcon := Application.Icon.Handle;
    StrPCopy(IconData.szTip, Application.Title);
    Shell_NotifyIcon(NIM_ADD, @IconData);
    Timer1.Interval := 1000;
    Timer1.Enabled := true;
end;

procedure TForm1.Open1Click(Sender: TObject);
begin
    Form1.Show;
    ShowWindow(Application.Handle, SW_HIDE);
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
    Shell_NotifyIcon(NIM_DELETE, @IconData);
    Application.ProcessMessages;
    Application.Terminate;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin

```

```
Action := caNone;
Form1.Hide;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  case(IconCount) of
    0 : IconData.hIcon := LoadIcon(0, IDI_APPLICATION);
    1 : IconData.hIcon := LoadIcon(0, IDI_ASTERISK);
    2 : IconData.hIcon := LoadIcon(0, IDI_EXCLAMATION);
    3 : IconData.hIcon := LoadIcon(0, IDI_HAND);
    4 : IconData.hIcon := LoadIcon(0, IDI_QUESTION);
    5 : IconData.hIcon := Application.Icon.Handle;
  end;
  inc(IconCount);
  if IconCount > 5 then
    IconCount := 0;
  Application.Title := TimeToStr(Now);
  StrPCopy(IconData.szTip, Application.Title);
  Shell_NotifyIcon(NIM_MODIFY, @IconData);
end;

begin
  ShowWindow(Application.Handle, SW_HIDE);
end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17913: Making sure changes survive a power failure

Question and Answer Database

FAQ2913D.txt Making sure changes survive a power failure

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How do I make sure that my data is saved to disk and has the greatest chance of surviving something unexpected, like a power failure?

Answer:

?

?

- Call DbtSaveChanges

?

- Call TTable.FlushBuffers

?

- Set LOCAL SHARE to TRUE in the BDE Configuration Tool

?

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17915: Calculating a point using angles and distance

Question and Answer Database

FAQ2915D.txt     Calculating a point using angles and distance

Category     : Object Pascal

Platform     : All

Product      : All 32 bit

Question:

How do I calculate the x and y coordinates of a point that is at some distance and angle away?

Answer:

The following example shows how to convert a polar coordinate to a rectangular coordinate:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Angle : Double;
  x : Double;
  y : Double;
  Distance : Double;
  Radians : Double;
begin
  Distance := 100;
  Angle := 270;
  Radians := Angle * DegToRad;
  x := Round(Distance * Cos(Radians));
  y := Round(Distance * Sin(Radians));
  ShowMessage(FloatToStr(x) + ' ' + FloatToStr(y));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17917: Calculating an angle from two points

Question and Answer Database

FAQ2917D.txt     Calculating an angle from two points

Category     : Object Pascal

Platform     : All

Product      : All 32 bit

Question:

How can I tell the given angle expressed by two points?

Answer:

The following example shows how to tell the angle of a given point from the coordinate 0, 0. In this example, the caption of the form shows the current point and angle from the center of the form. The y coordinates are inverted to show a more natural north orientation where north is directly up.

Example:

```
const RADTODEG = 180 / Pi;

function AngleOfPt(pt : TPoint) : double;
begin
  Result := 0;
  if ((Pt.x = 0) and (Pt.y < 0)) then
    Result := 270 else
  if ((Pt.x = 0) and (Pt.y > 0)) then
    Result := 90 else
  if ((Pt.x > 0) and (Pt.y >= 0)) then
    Result := ArcTan(Pt.y / Pt.x) * RADTODEG else
  if ((Pt.x < 0) And (Pt.y > 0)) then
    Result := 180 - (ArcTan(Pt.y / Abs(Pt.x))* RADTODEG) else
  if ((Pt.x < 0) And (Pt.y <= 0)) then
    Result := 180 + (ArcTan(Pt.y / Pt.x) * RADTODEG) else
  if ((Pt.x > 0) and (Pt.y < 0)) then
    Result := 360 - (ArcTan(Abs(Pt.y) / Pt.x) * RADTODEG) else
  Result:=0;
end;
```

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift:
TShiftState; X, Y: Integer);
var
  XYMove : TPoint;
  RotateAngle : Double;
begin
  XYMove := (Point(x - (Form1.Width div 2),
                  (Form1.Height div 2) - y));
  RotateAngle :=
    AngleOfPt(XYMove);
  Form1.Caption := IntToStr(XYMove.x) + #32 +
                  IntToStr(XYMove.y) + ' = ' +
                  FloatToStr(RotateAngle);
end;
```

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Form1.Canvas.MoveTo(Form1.Width div 2,
                      0);
  Form1.Canvas.LineTo(Form1.Width div 2,
                      Form1.Height);
  Form1.Canvas.MoveTo(0,
                      Form1.Height div 2);
  Form1.Canvas.LineTo(Form1.Width,
                      Form1.Height div 2);
end;
```

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Invalidate;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17919: Changing delphi default directory

Question and Answer Database

FAQ2919D.txt Changing delphi default directory

Category :IDE

Platform :All

Product :All 32 bit

Question:

How can I change the default directory Delphi / Borland C++ Builder uses to save projects in?

Answer:

Right click on the Windows shortcut icon used to start the program. Select properties from the menu, and change the default working directory the program starts in.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17921: Changing the printer port in code.

Question and Answer Database

FAQ2921D.txt Changing the printer port in code.

Category :Windows API

Platform :All

Product :All 32 bit

Question:

How can I change the port a given printer will use in code?

Answer:

You can use the SetPrinter method of TPrinter to change the port. The following example demonstrates changing the port to print to a file.

Example:

```
uses Printers;
```

```
{ $IFDEF WIN32 }  
const MAX_PATH = 144;  
{ $ENDIF }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    pDevice : pChar;
```

```
    pDriver : pChar;
```

```
    pPort   : pChar;
```

```
    hDMode  : THandle;
```

```
    PMode   : PDEVMODE;
```

```
begin
```

```
    if PrintDialog1.Execute then begin
```

```
        GetMem(pDevice, cchDeviceName);
```

```
        GetMem(pDriver, MAX_PATH);
```

```
        GetMem(pPort, MAX_PATH);
```

```
        Printer.GetPrinter(pDevice, pDriver, pPort, hDMode);
```

```
        Printer.SetPrinter(pDevice, PDriver, 'FILE:', hDMode);
```

```
        FreeMem(pDevice, cchDeviceName);
```

```
        FreeMem(pDriver, MAX_PATH);
```

```
FreeMem(pPort, MAX_PATH);
Printer.BeginDoc;
Printer.Canvas.TextOut(100, 100, 'Delphi Is RAD!');
Printer.EndDoc;
end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17923: TCombobox drop down size.

Question and Answer Database

FAQ2923D.txt TCombobox drop down size.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I tell the actual size of a TComboBox in it's dropped down state before it drops down?

Answer:

During the FormShow event, send the combo box a CB\_SHOWDROPDOWN message twice, once to drop it down, and once to retract it. Then send a CB\_GETDROPPEDCONTROLRECT message, passing the address of a TRect. When the SendMessage call returns, the TRect will contain the rectangle that would enclose the ComboBox in it's dropped down state, relative to the screen. You can then call ScreenToClient to convert the TRect's coordinates to be relative to the client area of the form.

Example:

```
var
  R : TRect;

procedure TForm1.FormShow(Sender: TObject);
var
  T : TPoint;
begin
  SendMessage(ComboBox1.Handle,
              CB_SHOWDROPDOWN,
              1,
              0);
  SendMessage(ComboBox1.Handle,
              CB_SHOWDROPDOWN,
              0,
              0);
  SendMessage(ComboBox1.Handle,
              CB_GETDROPPEDCONTROLRECT,
```

```
        0,  
        LongInt(@r));  
t := ScreenToClient(Point(r.Left, r.Top));  
r.Left := t.x;  
r.Top := t.y;  
t := ScreenToClient(Point(r.Right, r.Bottom));  
r.Right := t.x;  
r.Bottom := t.y;  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form1.Canvas.Rectangle(r.Left,  
                           r.Top,  
                           r.Right,  
                           r.Bottom );  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17928: Finding distance between two points.

Question and Answer Database

FAQ2928D.txt Finding distance between two points.

Category :Object Pascal

Platform :All

Product :All 32 bit

Question:

How can I find the distance between two points?

Answer:

The following example shows how to tell the distance between two points. In this example, the caption of the form shows the current point and distance from the center of the form. The y coordinates are inverted to show a more natural north orientation where north is directly up.

Example:

```
function Distance(Pt1 : TPoint;
                  Pt2 : TPoint) : Double;
var
  dx : LongInt;
  dy : LongInt;
begin
  dx:= pt1.x - pt2.x;
  dy:= pt1.y - pt2.y;
  Result := Sqrt((Dx * Dx) + (Dy * Dy));
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift:
TShiftState; X, Y: Integer);
var
  Pt1 : TPoint;
  Pt2 : TPoint;
  Dist : Double;
begin
  Pt1.x := Form1.Width div 2;
  Pt1.y := Form1.Height div 2;
```

```
Pt2.x := x;
Pt2.y := y;
Dist := Distance(Pt1, Pt2);
Form1.Caption := IntToStr(Pt2.x - Pt1.x) + #32 +
                 IntToStr(Pt1.y - Pt2.y) + ' = ' +
                 FloatToStr(Dist);
end;
```

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Form1.Canvas.MoveTo(Form1.Width div 2,
                      0);
  Form1.Canvas.LineTo(Form1.Width div 2,
                      Form1.Height);
  Form1.Canvas.MoveTo(0,
                      Form1.Height div 2);
  Form1.Canvas.LineTo(Form1.Width,
                      Form1.Height div 2);
end;
```

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Invalidate;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17930: Determining if an object supports a property.

Question and Answer Database

FAQ2930D.txt Determining if an object supports a property.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I tell if a given object has a given property at runtime?

Answer:

The following example shows how to determine if an object has a property available, and if so, how to change the property:

```
function HasProperty(Obj : TObject;
                    Prop : string) : PPropInfo;
begin
    Result := GetPropInfo(Obj.ClassInfo, Prop);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    p : pointer;
begin
    p := HasProperty(Button1, 'Color');
    if p <> nil then
        SetOrdProp(Button1, p, clRed) else
        ShowMessage('Button has no color property');

    p := HasProperty(Label1, 'Color');
    if p <> nil then
        SetOrdProp(Label1, p, clRed) else
        ShowMessage('Label has no color property');

    p := HasProperty(Label1.Font, 'Color');
    if p <> nil then
        SetOrdProp(Label1.Font.Color, p, clBlue) else
        ShowMessage('Label.Font has no color property');
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17932: Determining if a combobox is in dropped state.

Question and Answer Database

FAQ2932D.txt Determining if a combobox is in dropped state.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I determine if a ComboBox is in a dropped down state?

Answer:

Send a CB\_GETDROPPEDSTATE message to the ComboBox.

Example:

```
if SendMessage(ComboBox1.Handle,  
               CB_GETDROPPEDSTATE,  
               0,  
               0) = 1 then begin  
{do something}  
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17934: How do I plot a line?

Question and Answer Database

FAQ2934D.txt    How do I plot a line?

Category        :VCL

Platform        :All

Product         :All 32 bit

Question:

How do I plot a line?

Answer:

The easiest way to step along each point in a line segment is to call the Windows API function `LineDDA()`. `LineDDA()` takes the x and y start and end points of the line you wish to plot, and the address of a function in your program, and calls this function for each pixel that needs to be drawn. `LineDDA()` accepts an additional parameter "LParam" that gets passed back to your function each time it is called. The value of LParam can be anything you wish, and is useful for passing handles and pointers. Some typecasting of this parameter may be necessary to achieve the functionality you desire.

The following example shows how to plot a line from 10,10 to 100,100, and draw a circle along the line if the x coordinate of the pixel position is an even multiple of 10.

Example:

```
procedure DDAProc(x : integer;
                 y : integer;
                 LParam : LongInt)
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
  if x mod 10 = 0 then
    TCanvas(LParam).Ellipse(x - 10,
                           y - 10,
                           x + 10,
                           y + 10);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  LineDDA(10, 10,
    100, 100,
    @DDAProc,
    {$IFDEF WIN32}
    LongInt(Form1.Canvas)
    {$ELSE}
    Form1.Canvas
    {$ENDIF}
  );
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17936: Moving a control at runtime with the mouse.

Question and Answer Database

FAQ2936D.txt Moving a control at runtime with the mouse.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I move a control at runtime using the mouse?

Answer:

You can trap the OnMouseDown event, remembering the x and y coordinates of the mouse and set the mouse capture. You can then track mouse movements with the OnMouseMove event, moving the control until the OnMouseUp event is fired. You then will want to place the control in it's final position and release capture of the mouse.

The following example shows how to move a TButton component around on the form using the mouse. The movement is initiated by the user holding down the control key when the mouse button is activated.

Example:

type

```
TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure Button1MouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
  procedure Button1MouseUp(Sender: TObject; Button:
    TMouseButton; Shift: TShiftState; X, Y: Integer);
private
  { Private declarations }
public
  { Public declarations }
  MouseDownSpot : TPoint;
```



```

    Capturing : bool;
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1MouseDown(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    if ssCtrl in Shift then begin
        SetCapture(Button1.Handle);
        Capturing := true;
        MouseDownSpot.X := x;
        MouseDownSpot.Y := Y;
    end;
end;

procedure TForm1.Button1MouseMove(Sender: TObject;
Shift: TShiftState; X, Y: Integer);
begin
    if Capturing then begin
        Button1.Left := Button1.Left - (MouseDownSpot.x - x);
        Button1.Top := Button1.Top - (MouseDownSpot.y - y);
    end;
end;

procedure TForm1.Button1MouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    if Capturing then begin
        ReleaseCapture;
        Capturing := false;
        Button1.Left := Button1.Left - (MouseDownSpot.x - x);
        Button1.Top := Button1.Top - (MouseDownSpot.y - y);
    end;
end;

```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17938: Debugging CORBA clients and servers

Question and Answer Database

FAQ2938D.txt :Debugging CORBA clients and servers

Category :CORBA

Platform :Win95/NT

Product :

Question:

How do I debug a CORBA client and server at the same time?

Answer:

You will need two instances of Delphi 4. One will be running and debugging the server, the other will be running and debugging the client. Always make sure that you start the server before you start the client, and always make sure you shut all clients down before you shut the server down, otherwise you may get unpredictable results.

12/01/1998 1:40:48 PM

Last Modified: 01-SEP-99

## Article #17940: Placing a grid in edit mode programatically.

Question and Answer Database

FAQ2940D.txt Placing a grid in edit mode programatically.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I programmatically place a grid into edit mode and move the cursor to a predetermined position in the edit box?

Answer:

Place the grid in editor mode, then get a handle to the edit box and send a EM\_SETSEL message, passing the initial position where you want the cursor to appear, and the position of the ending selection (if you wish to highlight text).

The following example sets the cursor to the second character in the edit control, and does not select any text.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  h : THandle;
begin
  Application.ProcessMessages;
  DbGrid1.SetFocus;
  DbGrid1.EditorMode := true;
  Application.ProcessMessages;
  h := Windows.GetFocus;
  SendMessage(h, EM_SETSEL, 2, 2);
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17942: APPC LU NAME ARE INVALID OR NOT FOUND

Question and Answer Database

FAQ2942D.txt - APPC LU NAME ARE INVALID OR NOT FOUND

Category :AS400  
Platform :All-32Bit  
Product :

Question:

How come I am getting "APPC LU NAME ARE INVALID OR NOT FOUND" when running Delphi/400.

Answer:

Reasons for this error can be:

The router was not signed on with QSECOFR authority during the install.

ClientObjects/400 is not installed to the AS/400

You are using a TCP/IP transport to use the SNA version of ClientObjects/400 or you're using an SNA router in combination with the TCP/IP version of ClientObjects/400.

If you're using SNA Server: the server is not started.

If you're using SNA Server: In the SNA Server Admin application, LU Properties, "Member Default Outgoing LU Pool" check box must be selected.

If you're using TCP/IP: TCP/IP is not started on the AS/400 (STRTCP command on the AS/400).

If you're using TCP/IP: The subsystem is not started on the AS/400 (STRSBS CO400/CO400TCP).

4/1/99 3:29:12 PM

Last Modified: 01-SEP-99

## Article #17944: Trapping messages in non client area of a form.

Question and Answer Database

FAQ2944D.txt Trapping messages in non client area of a form.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How can I trap messages in the non client areas of my form, such as the title bar?

Answer:

Create a message handler for one of the variety of WM\_NC (non client) messages available to trap. (search the Windows API help file for WM\_NC). This example shows how to track mouse movement in all of the non client areas of the form.

Example:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
  private
```

```
    procedure WMNCMOUSEMOVE(var Message: TMessage);
```

```
    message WM_NCMOUSEMOVE;
```

```
    { Private declarations }
```

```
  public
```

```
    { Public declarations }
```

```
  end;
```

```
var
```

```
  Form1: TForm1;
```

implementation

{ \$R \*.DFM }

```
procedure TForm1.WMNCMOUSEMOVE(var Message: TMessage);
var
  s : string;
begin
  case Message.wParam of
    HTERROR : s := 'HTERROR';
    HTTRANSPARENT : s := 'HTTRANSPARENT';
    HTNOWHERE : s := 'HTNOWHERE';
    HTCLIENT : s := 'HTCLIENT';
    HTCAPTION : s := 'HTCAPTION';
    HTSYSTEMMENU : s := 'HTSYSTEMMENU';
    HTSIZE : s := 'HTSIZE';
    HTMENU : s := 'HTMENU';
    HTHSCROLL : s := 'HTHSCROLL';
    HTVSCROLL : s := 'HTVSCROLL';
    HTMINBUTTON : s := 'HTMINBUTTON';
    HTMAXBUTTON : s := 'HTMAXBUTTON';
    HTLEFT : s := 'HTLEFT';
    HTRIGHT : s := 'HTRIGHT';
    HTTOP : s := 'HTTOP';
    HTTOPLEFT : s := 'HTTOPLEFT';
    HTTOPRIGHT : s := 'HTTOPRIGHT';
    HTBOTTOM : s := 'HTBOTTOM';
    HTBOTTOMLEFT : s := 'HTBOTTOMLEFT';
    HTBOTTOMRIGHT : s := 'HTBOTTOMRIGHT';
    HTBORDER : s := 'HTBORDER';
    HTOBJECT : s := 'HTOBJECT';
    HTCLOSE : s := 'HTCLOSE';
    HTHELP : s := 'HTHELP';
    else s := '';
  end;
  Form1.Caption := s;
  Message.Result := 0;
end;

end.
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17946: Creating a PolyPolygon using a point array.

Question and Answer Database

FAQ2946D.txt Creating a PolyPolygon using a point array.

Category :VCL

Platform :All

Product :All 32 bit

Question:

How do I create a PolyPolygon using a point array as shown in the example of Polygon?

Answer:

Polygon is a method of TCanvas, and has been adapted to use a dynamic point array as a parameter. The PolyPolygon() function is a Windows GDI function and requires a pointer to a point array.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  PtArray : array[0..9] of TPOINT;
  PtCounts : array[0..1] of integer;
begin
  PtArray[0] := Point(0, 0);
  PtArray[1] := Point(0, 100);
  PtArray[2] := Point(100, 100);
  PtArray[3] := Point(100, 0);
  PtArray[4] := Point(0, 0);
  PtCounts[0] := 5;
  PtArray[5] := Point(25, 25);
  PtArray[6] := Point(25, 75);
  PtArray[7] := Point(75, 75);
  PtArray[8] := Point(75, 25);
  PtArray[9] := Point(25, 25);
  PtCounts[1] := 5;
  PolyPolygon(Form1.Canvas.Handle,
              PtArray,
              PtCounts,
```



2) ;

end ;

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17948: Converting a RGB color to a CMYK color

Question and Answer Database

FAQ2948D.txt - Converting a RGB color to a CMYK color

Category :Printing  
Platform :All  
Product :

Question:

How do I convert an RGB color to a CMYK color?

Answer:

The following functions RGBTOCMYK() and CMYKTORGB() demonstrate how to convert between RGB and CMYK color spaces. Note: There is a direct relationship between RGB colors and CMY colors. In a CMY color, black tones are achieved by printing equal amounts of Cyan, Magenta, and Yellow ink. The black component in a CMY color is achieved by reducing the CMY components by the minimum of (C, M, and Y) and substituting pure black in its place producing a sharper print and using less ink. Since it is possible for a user to boost the C,M and Y components where boosting the black component would have been preferable, a ColorCorrectCMYK() function is provided to achieve the same color by reducing the C, M and Y components, and boosting the K component.

Example:

```
procedure RGBTOCMYK(R : byte;  
                   G : byte;  
                   B : byte;  
                   var C : byte;  
                   var M : byte;  
                   var Y : byte;  
                   var K : byte);  
  
begin  
  C := 255 - R;  
  M := 255 - G;  
  Y := 255 - B;
```

```

if C < M then
  K := C else
  K := M;
if Y < K then
  K := Y;
if k > 0 then begin
  c := c - k;
  m := m - k;
  y := y - k;
end;
end;

```

```

procedure CMYKTORGB(C : byte;
                   M: byte;
                   Y : byte;
                   K : byte;
                   var R : byte;
                   var G : byte;
                   var B : byte);

```

```

begin
  if (Integer(C) + Integer(K)) < 255 then
    R := 255 - (C + K) else
    R := 0;
  if (Integer(M) + Integer(K)) < 255 then
    G := 255 - (M + K) else
    G := 0;
  if (Integer(Y) + Integer(K)) < 255 then
    B := 255 - (Y + K) else
    B := 0;
end;

```

```

procedure ColorCorrectCMYK(var C : byte;
                           var M : byte;
                           var Y : byte;
                           var K : byte);

```

```

var
  MinColor : byte;
begin
  if C < M then
    MinColor := C else
    MinColor := M;
  if Y < MinColor then
    MinColor := Y;
  if MinColor + K > 255 then
    MinColor := 255 - K;

```

```

    C := C - MinColor;
    M := M - MinColor;
    Y := Y - MinColor;
    K := K + MinColor;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    R : byte;
    G : byte;
    B : byte;
    C : byte;
    M : byte;
    Y : byte;
    K : byte;
begin
    R := 151;
    G := 81;
    B := 55;
    Mem1.Lines.Add('R = ' + IntToStr(R));
    Mem1.Lines.Add('G = ' + IntToStr(G));
    Mem1.Lines.Add('B = ' + IntToStr(B));
    Mem1.Lines.Add('-----');
    RGBTOCMYK(R, G, B, C, M, Y, K);
    Mem1.Lines.Add('C = ' + IntToStr(C));
    Mem1.Lines.Add('M = ' + IntToStr(M));
    Mem1.Lines.Add('Y = ' + IntToStr(Y));
    Mem1.Lines.Add('K = ' + IntToStr(K));
    Mem1.Lines.Add('-----');
    CMYKTORGB(C, M, Y, K, R, G, B);
    Mem1.Lines.Add('R = ' + IntToStr(R));
    Mem1.Lines.Add('G = ' + IntToStr(G));
    Mem1.Lines.Add('B = ' + IntToStr(B));
    Mem1.Lines.Add('-----');
    RGBTOCMYK(R, G, B, C, M, Y, K);
    c := c + 2;
    m := m + 2;
    y := y + 2;
    ColorCorrectCMYK(C, M, Y, K);
    Mem1.Lines.Add('C = ' + IntToStr(C));
    Mem1.Lines.Add('M = ' + IntToStr(M));
    Mem1.Lines.Add('Y = ' + IntToStr(Y));
    Mem1.Lines.Add('K = ' + IntToStr(K));
end;

```

4/2/99 11:26:15 AM

Last Modified: 01-SEP-99

## Article #17952: Metafile fonts and the VCL

Question and Answer Database

FAQ2952D.txt Metafile fonts and the VCL

Category :VCL

Platform :All

Product :All 32 bit

### Question:

Sometimes, some text in a metafile I display from my application comes up in the wrong font. The file displays properly in xyz Graphics program. Is this a bug in the VCL?

### Answer:

No. The VCL simply wraps standard Windows functions for displaying metafiles. As the metafile is scaled, the font width and height are scaled as well. Many times, the Windows font mapper will simply substitute a different font that the font that was originally selected in an attempt to satisfy the request. This can also be seen in many graphics programs if you arbitrarily scale the metafile in different directions. Note that some "hi end" graphic programs actually read the metafile record by record, bypassing the Windows GDI entirely to overcome the effects of the Windows font mapper.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17954: "APPC MEMORY ALLOCATION ERROR" with Delphi 400

Question and Answer Database

FAQ2954D.txt - "APPC MEMORY ALLOCATION ERROR" with Delphi 400

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

What causes the error "APPC MEMORY ALLOCATION ERROR"?

Answer:

This can occur when using the PCS router and when there is insufficient low memory. Try adding "SetDOS16M 1M" to the Autoexec.bat to allow allocation of extended memory.

4/1/99 3:29:57 PM

Last Modified: 01-SEP-99

## Article #17956: Connecting to a remote CORBA server

Question and Answer Database

FAQ2956D.txt :Connecting to a remote CORBA server

Category :CORBA

Platform :Win95/NT

Product :Delphi4.x,

Question:

My CORBA client and server work great together on the same machine, but when I try running the client from a remote machine it doesn't work. What's wrong?

Answer:

In addition to the RepositoryId, you need to specify both the HostName and the ObjectName for your TCorbaConnection component.

12/01/1998 1:41:17 PM

Last Modified: 01-SEP-99



## Article #17958: Error 'There are no fonts installed'.

Question and Answer Database

FAQ2958D.txt     Error 'There are no fonts installed'.

Category        :VCL

Platform        :All

Product         :All 32 bit

### Question:

When I execute the FontDialog with the Device property set to fdBoth or fdPrinter, I get an error "There are no fonts installed".

### Answer:

This setting should show fonts that are compatible with either the Printer or both the Printer and the Screen. The following is an example of calling the Windows ChooseFont dialog directly to provide a list of fonts that are compatible with both the screen and the printer:

### Example:

```
uses
  Printers,
  CommDlg;

procedure TForm1.Button1Click(Sender: TObject);
var
  cf : TChooseFont;
  lf : TLogFont;
  tf : TFont;
begin
  if PrintDialog1.Execute then begin
    GetObject(Form1.Canvas.Font.Handle,
              sizeof(lf),
              @lf);
    FillChar(cf, sizeof(cf), #0);
    cf.lStructSize := sizeof(cf);
    cf.hWndOwner := Form1.Handle;
    cf.hdc := Printer.Handle;
    cf.lpLogFont := @lf;
```

```
cf.iPointSize := Form1.Canvas.Font.Size * 10;
cf.Flags := CF_BOTH or
           CF_INITTOLOGFONTSTRUCT or
           CF_EFFECTS or
           CF_SCALABLEONLY or
           CF_WYSIWYG;
cf.rgbColors := Form1.Canvas.Font.Color;
if ChooseFont(cf) <> false then begin
  tf := TFont.Create;
  tf.Handle := CreateFontIndirect(lf);
  tf.Color := cf.RgbColors;
  Form1.Canvas.Font.Assign(tf);
  tf.Free;
  Form1.Canvas.TextOut(10, 10, 'Test');
end;
end;
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17960: Can't create local description file

Question and Answer Database

FAQ2960D.txt - Can't create local description file

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

What causes the error "Can't create local description file"

Answer:

This occurs in a deployed application when the application can not find the description of the AS400 files (Data Description Specifications or DDS).

Verify that the following entries are in your WIN.INI file:  
[CO400]

PATH\_DDS = The path to the DDS(C:\CO400\DDS for example).

4/1/99 3:30:21 PM

Last Modified: 01-SEP-99

## Article #17962: Making the drop-down list appear faster

Question and Answer Database

FAQ2962D.txt Making the drop-down list appear faster

Category :Database/VCL

Platform :All

Product :All 32 bit

Question:

How do I get the drop-down list (pick list) to appear faster?  
I don't want to have to click 4 times every single time.

Answer:

In your OnCellClick event handler for your TDBGrid, do this:

```
keybd_event(VK_F2,0,0,0);  
keybd_event(VK_F2,0,KEYEVENTF_KEYUP,0);  
keybd_event(VK_MENU,0,0,0);  
keybd_event(VK_DOWN,0,0,0);  
keybd_event(VK_DOWN,0,KEYEVENTF_KEYUP,0);  
keybd_event(VK_MENU,0,KEYEVENTF_KEYUP,0);
```

This is the same as pressing F2 followed by Alt+DownArrow.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17964: Why isn't a network install of Delphi 3 supported?

Question and Answer Database

FAQ2964D.txt Why isn't a network install of Delphi 3 supported?

Category :Install Issues

Platform :All

Product :Delphi 3.x

Question:

Why isn't a network install of Delphi 3 supported?

Answer:

Delphi 3 was not designed to be installed on a network. Borland makes no claims that a network install of Delphi 3 will work.

It may work for some users if all work machines install Delphi 3 to the same location on the network. This requires many rights, including:

- \* Writing shared DLLs and OCXs to NT4's System32 directory (usually C:\WinNT\System32)
- \* Writing registry entries to HKEY\_LOCAL\_MACHINE
- \* Writing registry entries to HKEY\_CURRENT\_USER
- \* Writing registry entries to HKEY\_CLASSES\_ROOT (part of registering OCXs)

Delphi 3 expects all shared DLLs, packages and OCXs to be locally installed.

Delphi 3 expects all registry entries to be local.

Many problems may arise when trying to install Delphi 3 on a network. Possibly the most important one is the network software and hardware itself. Unforeseen problems may occur with any network configuration.

For more information, please see the TI entitled "Installing and

using Delphi 3 under NT4".

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17966: Getting the summary function to work

Question and Answer Database

FAQ2966D.txt Getting the summary function to work

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

I cannot get the summary function to work? What's wrong?

Answer:

Most likely you have forgot to set the Master property. Master should be pointed to the QuickRep or QRSubDetail component you want to base the aggregation upon.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17968: Adding sums of fields

Question and Answer Database

FAQ2968D.txt Adding sums of fields

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

I want to sum several fields and then sum those sums -  
how do I do this?

Answer:

Just create a entire new sum expression, adding all  
the fields together:

```
QRExpr1.Expression := Sum(Field1);  
QRExpr2.Expression := Sum(Field2);  
QRExpr3.Expression := Sum(Field1+Field2);
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## **Article #17970: Is there a TQRDBMemo?**

Question and Answer Database

FAQ2970D.txt Is there a TQRDBMemo?

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

Why is there no TQRDBMemo component?

Answer:

Use TQRDBText. It does the job a TQRDBMemo would do.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17972: Switching between different data sources

Question and Answer Database

FAQ2972D.txt     Switching between different data sources

Category     :Quick Reports

Platform     :All

Product      :All 32 bit

Question:

In QuickReport 1 all components was connected to a DataSource, now they are connected to a DataSet. The other solution made it much easier to switch between different tables. Why did this change? How do I use it?

Answer:

This change was done to improve speed, code size and to make QuickReport more stable. To switch between different tables and queries it's easiest to use the TQRExpr component to display field data. TQRExpr will pick up the field from any dataset connected to the report, regardless of its name. Then simply change the QuickRep.DataSet property.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17974: Improving performance

Question and Answer Database

FAQ2974D.txt     Improving performance

Category     :Quick Reports

Platform     :All

Product      :All 32 bit

Question:

Is there any special trick to improve performance?

Answer:

Most importantly - add persistent field objects for your tables. Create expressions with as few functions as possible. Instead of doing `Sum(Field1)+Sum(Field2)` you should do `Sum(Field1+Field2)`. The last one is a lot faster.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17976: Printing Delphi controls using QuickReports

Question and Answer Database

FAQ2976D.txt - Printing Delphi controls using QuickReports

Category :Quick Reports

Platform :All-32Bit

Product :

Question:

Can I print any standard Delphi control using QR?

Answer:

No, a control has to be descending from TQRPrintable to be printed on a QuickReport.

However, if you want, you can drop a TQRImage on a QuickReport and then do something like this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DC : HDC;
begin
  DC := GetDC(Button1.Handle);
  BitBlt(QuickReport2.QRImage1.Canvas.Handle,0,0,
        Button1.Width,Button1.Height,DC,0,0,SrcCopy);
  ReleaseDC(Button1.Handle,DC);
  QuickReport2.Preview;
end;
```

4/2/99 12:18:28 PM

Last Modified: 01-SEP-99

## Article #17978: Using additional data sets in expressions

Question and Answer Database

FAQ2978D.txt Using additional data sets in expressions

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

The expressions only pick up field values from datasets referenced in my reports. How can I use additional datasets in expressions?

Answer:

In the QuickRep.BeforePrint event you can add additional datasets to the QuickRep.AllDataSets list property:

```
MyReport.AllDataSets.Add(SomeOtherTable).
```

You can now use fields from this (these) tables in your expressions.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17980: Stretching memos

Question and Answer Database

FAQ2980D.txt     Stretching memos

Category        :Quick Reports

Platform        :All

Product         :All 32 bit

Question:

I set AlignToBottom to True and use a stretching memo field.  
The result is not what I expected. Why not?

Answer:

Stretching memos are not supported in page footers. A band with AlignToBottom set to True acts like a page footer and therefore does not allow a stretching (memo)component.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17982: Setting the caption of the preview window

Question and Answer Database

FAQ2982D.txt    Setting the caption of the preview window

Category        :Quick Reports

Platform        :All

Product         :All 32 bit

Question:

How do I set the caption of the preview window?

Answer:

The preview window will get the report title as caption.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17984: Printing to another printer than the default one

Question and Answer Database

FAQ2984D.txt    Printing to another printer than the default one  
Category        :Quick Reports  
Platform        :All  
Product         :All 32 bit

Question:

How do I print to a printer different than the Windows default printer?

Answer:

Set `QuickRep.PrinterSettings.PrinterIndex` to a value corresponding to the printer you want to print to. This value is the same as for the Delphi `TPrinter.PrinterIndex`. Set to `-1` to print to the default printer again.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #17986: Printing calculated text

Question and Answer Database

FAQ2986D.txt    Printing calculated text

Category       :Quick Reports

Platform       :All

Product        :All 32 bit

Question:

I have a calculated table field and it has line breaks in the text and those line breaks don't print.

Answer:

Delphi defines calculated table text fields as TStringfield and QuickReport handles that type of field differently than it handles actual memo fields. To output a calculated text field, use a QRMemo component and set QRMemo.Lines.Text to the value of the calculated field at runtime.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17988: Cannot find A2E table

Question and Answer Database

FAQ2988D.txt - Cannot find A2E table

Category :AS400  
Platform :All Windows  
Product :DelphiAS400,

Question:

How come I'm getting the error "Cannot find A2E table" when making TAS400.Active set to TRUE?

Answer:

This error message refers to the ASCII to EBCDIC tables contained in files in a path defined by WIN.INI in the [CO400] section. Verify the entire path exists and that the \*.EBC and \*.ASC files exist along that path. Example of WIN.INI fragment:

```
[CO400]
.
.
c:\co400\convert\us1.ebc
c:\co400\convert\us1.asc.
```

4/1/99 3:30:57 PM

Last Modified: 01-SEP-99

## Article #17990: A sum(if(.....,0)) always returns 0

Question and Answer Database

FAQ2990D.txt A sum(if(.....,0)) always returns 0

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

The following expression always returns 0 when using the QRExpr component in Delphi 3:

```
Sum(if(Query1.CartonID = 'SI', Query1.Caliper, 0))
```

Why?

Answer:

Change the 0 to 0.0 and the Sum function will work as expected.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17992: Font problems on NT 4.0

Question and Answer Database

FAQ2992D.txt    Font problems on NT 4.0

Category       :Quick Reports

Platform        :All

Product         :All 32 bit

### Question:

When reports are previewed on a NT computer fonts sometimes changes randomly. How can I fix this?

### Answer:

There is a known printing problem with Windows NT 4.0 when you have multiple font changes on the printer canvas. Installing Service Pack 3 for NT 4.0 solves this problem.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17994: Printing a footer on only the last page

Question and Answer Database

FAQ2994D.txt    Printing a footer on only the last page

Category       :Quick Reports

Platform       :All

Product        :All 32 bit

Question:

Is there a way to print a band as footer only on the last page of a report.

Answer:

Instead of using a footer band, use a summary band and set it's AlignToBottom property to true.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17996: Getting AddPrintable to work

Question and Answer Database

FAQ2996D.txt Getting AddPrintable to work

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

I can't get AddPrintable to work.

Answer:

Do something like this:

```
with TQRDBText(DetailBand1.AddPrintable(TQRDBText)) do
begin
  DataField := 'Test';
  // Do other stuff
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #17998: No default printer selected when using Quickreports

Question and Answer Database

FAQ2998D.txt - No default printer selected when using Quickreports

Category :Quick Reports  
Platform :All-32Bit  
Product :

Question:

I get a exception error when running our application on a machine without a default printer.

Answer:

QuickReport requires that a default printer be installed. The report is prepared using information from the printer driver. The following code is one way to check and see if a default printer has been defined.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Device, Driver, Port    : array [0..255] of Char;
  Mode                   : Integer;
begin
  Printer.GetPrinter(Device,Driver,Port,Mode);
  if Device <> '' then
    ShowMessage(Device)
  else
    ShowMessage('You do not have a default printer defined');
end;
```

4/2/99 12:18:51 PM

Last Modified: 01-SEP-99

## Article #18000: The Count function

Question and Answer Database

FAQ3000D.txt    The Count function

Category        :Quick Reports

Platform        :All

Product         :All 32 bit

Question:

Why doesn't the expression Count(Table1.Field1) return the right value?

Answer:

The Count function in the expression builder returns a number that increments by 1 with every call to Count. It does not behave like the SQL Count function.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99



## Article #18002: Using a TQRExpr component at run-time

Question and Answer Database

FAQ3002D.txt Using a TQRExpr component at run-time

Category :Quick Reports

Platform :All

Product :All 32 bit

Question:

How do I check the value of QRExpr control at runtime?

Answer:

The Value property of a TQRExpr component is of type TQREvResult. You can check Value.Kind to see what the data type is. The following table shows how to retrieve the results of the expression based on the value of Value.Kind:

Value.Kind	The result is in
resInt	Value.IntResult
resDouble	Value.DblResult
resString	Value.StrResult
resBool	Value.BooResult
resError	there was an error

The constants are defined in the QRPrntr unit.

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #18004: Detecting if there's a disk in a drive

Question and Answer Database

FAQ3004D.txt    Detecting if there's a disk in a drive  
Category        :Miscellaneous  
Platform        :All  
Product         :All 32 bit

Question:

How do I detect if there's a disk in a drive? How do I avoid the critical error box that Windows displays?

Answer:

Here's a short example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldErrorMode : Integer;
  fp           : TextFile;
begin
  try
    OldErrorMode := SetErrorMode(SEM_FAILCRITICALERRORS);
    try
      AssignFile(fp, 'A:\foo.bar');
      Reset(fp);
      CloseFile(fp);
    finally
      SetErrorMode(OldErrorMode);
    end;
  except
    on E:EInOutError do
      if E.ErrorCode = 21 then
        ShowMessage('Drive A: is not ready...');
    end;
  end;
end;
```

Last Modified: 01-SEP-99

## Article #18006: Converting strings to dates

Question and Answer Database

FAQ3006D.txt     Converting strings to dates

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

I want to convert strings to dates, but StrToDate doesn't do what I want. I want to convert "December 6, 1969" to a TDateTime. What should I do?

Answer:

StrToDate only converts numbers, so if you have month names in your string, you'll have to use VarToDateTime instead. Here are some examples:

```
var
  D1, D2, D3 : TDateTime;
begin
  D1 := VarToDateTime('December 6, 1969');
  D2 := VarToDateTime('6-Apr-1998');
  D3 := VarToDateTime('1998-Apr-6');
  ShowMessage(DateToStr(D1)+' '+DateToStr(D2)+' '+
              DateToStr(D3));
end;
```

7/16/98 4:31:28 PM

Last Modified: 01-SEP-99

## Article #18008: Where is the 40COMUPD file

Question and Answer Database

FAQ3008D.txt :Where is the 40COMUPD file

Category :Install Issues

Platform :All Windows

Product :Delphi4.x,

Question:

The Delphi 4 README file states that I should run a file called 40COMPUPD to update my COMCTL32.DLL if an older one gets installed by another product. I cannot find that file.

Answer:

The Delphi 4 README file incorrectly identified the file as 40COMPUPD. The correct file name is 40COMUPD and it is located in the \INFO\UPDATES directory on your Delphi 4 CD.

11/30/1998 12:25:07 PM

Last Modified: 01-SEP-99

## Article #18010: Cannot find 'CO40032.DLL'

Question and Answer Database

FAQ3010D.txt - Cannot find 'CO40032.DLL'

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I'm getting Cannot find 'CO40032.DLL' when I deploy my application?

Answer:

CO40032.DLL must be deployed to the target system in the \windows\system (16 bit) or \windows\system32 (32 bit) directory. It's recommended that you use the install provided with the deployment version, which will properly deploy the DLL.

4/1/99 3:31:12 PM

Last Modified: 01-SEP-99

## Article #18012: Cannot create local description file

Question and Answer Database

FAQ3012D.txt - Cannot create local description file

Category :AS400  
Platform :Win95/NT  
Product :DelphiAS400,

Question:

Why am I getting a "Can't create local description file" when I deploy my Delphi/400 program?

Answer:

This error occurs with Delphi 1 deployed applications. WIN.INI must be updated so that the [CO400] section contains an entry that provides the directory containing local description files. For example:

```
[CO400]
.
.
PATH_DDS=C:\CO400\DDS
```

4/1/99 3:32:22 PM

Last Modified: 01-SEP-99

## Article #18014: Object Activation

Question and Answer Database

FAQ3014D.txt :Object Activation

Category :CORBA

Platform :Win95/NT

Product :Delphi4.x,

Question:

I'm trying to use the Object Activation Daemon to activate my server dynamically. Why do I get a TRANSIENT exception after the server launches?

Answer:

You need to make sure that the connection to your database is made active in run-time, and not in design-time. Also make sure that you TClientDataSets are made active at run-time, not at design-time.

12/01/1998 1:41:55 PM

Last Modified: 01-SEP-99



## Article #18016: Protection Error, Runtime version

Question and Answer Database

FAQ3016D.txt - Protection Error, Runtime version

Category :AS400  
Platform :All-32Bit  
Product :

Question:

How come I am getting the error "Protection Error, Runtime version" with Delphi/400?

Answer:

Possible Causes:

You are trying to use the run-time Components during Design Time.

You are exceeding the limit of Developers that are licensed to use the Delphi/400 components.

Your PC may have locked up and been rebooted without killing the job on the AS/400 thus making the 400 think that there are two developers with the same key running the same job. To check for currently running jobs on the AS/400:

Close Delphi.

Execute "WRKACTJOB" on the AS/400 to see the active job. Look for the PC controller. If the controller is in "CPCW" it means that the job is running. To kill the job, choose option "4" and finally "ENTER" to confirm.

The 60 day evaluation period has expired.

Developer must have installed product using the QSECOFR sign-on. You have installed Delphi/400 twice. By default you have a temp key for two months but just for the first install.

4/1/99 3:27:27 PM

Last Modified: 01-SEP-99

## Article #18018: 'Record not found' or 'End Of File'

Question and Answer Database

FAQ3018D.txt - 'Record not found' or 'End Of File'

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I'm getting "Record not found" or "End Of File" when Opening AS/400 file in IDE.

Answer:

These errors occur when running an application in the IDE even if trapped within a try..except block.

Uncheck 'Break On Exception' in TOOLS | OPTIONS | PREFERENCES.

4/1/99 3:32:12 PM

Last Modified: 01-SEP-99

## Article #18020: Signal IO error

Question and Answer Database

FAQ3020D.txt - Signal IO error

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I'm getting the error "Signal IO error"  
in QPJOBLOG for Delphi/400 jobs.

Answer:

This is normal and not an error but a function of Delphi/400.  
This function is called to get and send the AS/400 system  
message to the PC.

4/1/99 3:28:21 PM

Last Modified: 01-SEP-99

## Article #18022: SQL open error

Question and Answer Database

FAQ3022D.txt - SQL open error

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I'm getting the error "SQL open error" when using a TQuery400 or TSQL400 containing a Date Field?

Answer:

Check the date formatting. For example,  
'WHERE DTINVOICE='96-02-15', the format should be  
'WHERE DTInvoice='1996-02-25'.

4/2/99 12:19:22 PM

Last Modified: 01-SEP-99

## Article #18024: Unknown Field Name

Question and Answer Database

FAQ3024D.txt - Unknown Field Name

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

How come I'm getting the error "Unknown Field Name" with Delphi/400?

### Answer:

This error may occur when using field names that contain "#" as is common for the AS/400. This may be illegal on the PC depending on the translation table declared in the configuration of the router.

Note: SNA Server version 3 should be 3.01.1113. Early versions of 3.0 do a poor job of translating EBCDIC-ASCII for special characters.

Also note: the translation table must also accommodate the language features of the OS/400, i.e. French special characters, etc.

4/1/99 3:33:06 PM

Last Modified: 01-SEP-99

## Article #18026: Finding the RepositoryId of a CORBA Server

Question and Answer Database

FAQ3026D.txt :Finding the RepositoryId of a CORBA Server

Category :CORBA

Platform :Win95/NT

Product :Delphi4.x,

### Question:

Where can I find the RepositoryId for a CORBA server? I need it in order to put it in the client's TCorbaConnection.

### Answer:

Open up the \_TLB.PAS file that got generated for your server's type library and go to the end of it. You'll see the RepositoryId as the first parameter to the call to CorbaFactoryCreateStub in the CreateInstance class function for your CORBA factory.

12/01/1998 1:42:14 PM

Last Modified: 01-SEP-99

## Article #18028: IDAPI Service Library not Found

Question and Answer Database

FAQ3028D.txt - IDAPI Service Library not Found

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I'm using Delphi/400. How come I'm getting an "IDAPI Service Library Not Found" error?

Answer:

This error usually occurs in deployments.  
There are two conditions where this error can occur.

IDC0400.DLL is not in the BDE directory.  
Some of client DLLs (CO40032.DLL, TCRTR32, E32TCPIP.DLL, etc)  
are in the application directory and some are in Windows\System32.

If you use the install from the Deployment CD both of these problems will not occur.

4/1/99 3:33:50 PM

Last Modified: 01-SEP-99



## Article #18030: Delphi/400 Crash in Fields Editor

Question and Answer Database

FAQ3030D.txt - Delphi/400 Crash in Fields Editor

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Adding many fields to the fields edit causes a crash.

Answer:

Obtain the latest Delphi/400 patch from:

"<http://libra.netmastersllc.com/borland400/patches.html>"  
to solve this problem.

4/1/99 3:35:28 PM

Last Modified: 01-SEP-99

## Article #18032: Preventing changing pages in a page control

Question and Answer Database

FAQ3032D.txt : Preventing changing pages in a page control  
Category : VCL  
Platform : Win95/NT/98  
Product : All32Bit,

Question:

How do I prevent the change of a page in a TPageControl?

Answer:

```
uses
  CommCtrl;

procedure TForm1.PageControl1Changing(Sender: TObject;
  var AllowChange: Boolean);
var
  Info : TTCHitTestInfo;
  P    : TPoint;
  idx  : Integer;
begin
  GetCursorPos(P);
  Info.pt := (Sender as TPageControl).ScreenToClient(P);
  Info.flags := TCHT_ONITEM;
  idx := SendMessage((Sender as TPageControl).Handle, TCM_HITTEST, 0,
  Integer(@Info));
  Caption := 'idx = '+IntToStr(idx);
end;

// Of course, you'd set AllowChange to False given certain
// values of idx, instead of setting the Caption...
```

11/24/1998 3:14:45 PM

Last Modified: 01-SEP-99

## Article #18034: Crash in lookup field or dblookuplist

Question and Answer Database

FAQ3034D.txt - Crash in lookup field or dblookuplist

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I'm using Delphi/400. How come I crash in a lookup field or dblookuplist?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 3:36:59 PM

Last Modified: 01-SEP-99

## Article #18036: Delphi/400 : Queries against \*LIBL

Question and Answer Database

FAQ3036D.txt - Delphi/400 : Queries against \*LIBL

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Trying to execute a sql query on an alias with property  
Libraryname set to "\*LIBL" doesn't work.

Example :

MyTable is in MyLib Library and MyLib is included in \*LIBL

Select \* from MyTable. returns "invalid expresion" exception.

Answer:

Set explicitly the libraryname in your statement :

```
select * from MyLib/MyTable.
```

4/1/99 3:37:33 PM

Last Modified: 01-SEP-99

## Article #18038: Composite key with blank field inside

Question and Answer Database

FAQ3038D.txt - Composite key with blank field inside

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I have Delphi 3/400. Accessing a record by Findkey or FindNearest with a composite key including a blank field doesn't work.

### Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 3:38:22 PM

Last Modified: 01-SEP-99

## Article #18040: Opening a file with many indexes is slow

Question and Answer Database

FAQ3040D.txt - Opening a file with many indexes is slow

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I have Delphi 3/400. How how when I open a table with many indexes it is slow?

### Answer:

To solve this problem first obtain the latest Delphi/400 patch from:

"<http://libra.netmastersllc.com/borland400/patches.html>"

If you Add or Delete logical files on your AS/400 set the LOCAL INDEXES property of the IDCO400 driver to FALSE to regenerate local indexes.

4/1/99 3:40:13 PM

Last Modified: 01-SEP-99

**Article #18042: Only one connection is available via Sna/Server router**

Question and Answer Database

FAQ3042D.txt - Only one connection is available via Sna/Server router

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come when I launch 2 Delphi3/400 applications, I get an  
"INVALID CONVID" error on the second.

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 3:40:42 PM

Last Modified: 01-SEP-99

## Article #18044: Delphi 3/400: Record blocking doesn't work

Question and Answer Database

FAQ3044D.txt - Delphi 3/400: Record blocking doesn't work

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Using `dbisetprop(hDBIObj(table1.handle),$4050009E,NUMRECORDS);`  
to read NUMRECORDS in one pass, has no effect.

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

As of patch 2 the Property Value is now \$4050009

4/1/99 3:41:31 PM

Last Modified: 01-SEP-99



## Article #18046: If the last value of key is blank, findKey doesn't find any record

Question and Answer Database

FAQ3046D.txt - If the last value of key is blank, findKey doesn't find any record

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Delphi 3/400. FindKey['Value1',''] always return False.

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 3:42:02 PM

Last Modified: 01-SEP-99

## Article #18048: Client DataSet doesn't work with Delphi3/400

Question and Answer Database

FAQ3048D.txt - Client DataSet doesn't work with Delphi3/400

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Using a Tquery in a Multi-Tiered application result of blanks records or hangs client application.

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 3:42:31 PM

Last Modified: 01-SEP-99

## Article #18050: Update, AppendUpdate Modes for BatchMove don't work

Question and Answer Database

FAQ3050D.txt - Update, AppendUpdate Modes for BatchMove don't work

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

Delphi 3/400. How come batch move does not work correctly if the mode is BatAppend and BatUpdate?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 3:43:13 PM

Last Modified: 01-SEP-99

## Article #18052: Low Speed AS/400 with TCP/IP desynchronises

Question and Answer Database

FAQ3052D.txt - Low Speed AS/400 with TCP/IP desynchronises

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I'm using Delphi 3/400 to connect via TCP/IP to a slow AS/400. The conversation is getting desynconised. How do I solve this problem?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:03:23 PM

Last Modified: 01-SEP-99

## Article #18054: Delphi 3/400: 2 TQueries on the Same Form

Question and Answer Database

FAQ3054D.txt - Delphi 3/400: 2 TQueries on the Same Form

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come I can't use 2 TQueries on the same form?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:04:45 PM

Last Modified: 01-SEP-99

## Article #18056: TQuery with NULL values causes hang

Question and Answer Database

FAQ3056D.txt - TQuery with NULL values causes hang

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come Delphi hangs when I run a SQL query based on an AS/400 file with NULL values.

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:05:11 PM

Last Modified: 01-SEP-99

## Article #18058: Delphi 3/400: OnFilterRecord causes hang

Question and Answer Database

FAQ3058D.txt - Delphi 3/400: OnFilterRecord causes hang

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come when I use OnFilterRecord event to filter records on a table, Delphi hangs?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:05:47 PM

Last Modified: 01-SEP-99

## **Article #18060: Cancel does not unlock locked record**

Question and Answer Database

FAQ3060D.txt - Cancel does not unlock locked record

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I have Delphi 3/400. How come when a record is locked,  
Cancel doesn't unlock it?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/2/99 12:19:48 PM

Last Modified: 01-SEP-99



**Article #18062: Cannot write more than 223 characters in a dataarea**

Question and Answer Database

FAQ3062D.txt - Cannot write more than 223 characters in a dataarea

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How can I write more than 233 characters to a dataarea?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:06:46 PM

Last Modified: 01-SEP-99

## Article #18064: Not all fields allowed with Decision Cube

Question and Answer Database

FAQ3064D.txt - Not all fields allowed with Decision Cube

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I have Delphi 3/400. How can I make all legal fields to used in the Decsion cube?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:07:24 PM

Last Modified: 01-SEP-99

## Article #18066: Delphi 3/400: Invalid Parameter with BDE Admin

Question and Answer Database

FAQ3066D.txt - Delphi 3/400: Invalid Parameter with BDE Admin

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

After installing D3/400 with Patch 7, BDE invalid parameter occurs in BDE Administrator.

### Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:07:54 PM

Last Modified: 01-SEP-99

## Article #18068: Delphi 3/400: Delphi pads out strings with nulls

Question and Answer Database

FAQ3068D.txt - Delphi 3/400: Delphi pads out strings with nulls

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

Storing data on the AS/400 in text fields with Delphi (and other products that null out strings) ends up filling the text fields with trailing nulls, just as stored in Delphi.

### Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:08:43 PM

Last Modified: 01-SEP-99

## Article #18070: Delphi 3/400:4 Digit integers are not SmallInt

Question and Answer Database

FAQ3070D.txt - Delphi 3/400:4 Digit integers are not SmallInt

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come Integers with less than 5 digits are not smallints?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:09:18 PM

Last Modified: 01-SEP-99

## Article #18072: Delphi 3/400: Modifying Memo fields

Question and Answer Database

FAQ3072D.txt - Delphi 3/400: Modifying Memo fields

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

How come Modifying Memo fields doesn't work?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:09:48 PM

Last Modified: 01-SEP-99

## Article #18074: Delph 3/400: Float fields are rounded

Question and Answer Database

FAQ3074D.txt - Delph 3/400: Float fields are rounded

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I have Delphi 3/400. How come my Float Fields are being rounded?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:10:17 PM

Last Modified: 01-SEP-99

## Article #18076: Delphi 3/300:Closing TQuery results in a GPF

Question and Answer Database

FAQ3076D.txt - Delphi 3/300:Closing TQuery results in a GPF

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I have Delphi 3/400. How I can make closing my TQuery not make me GPF?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

4/1/99 4:10:55 PM

Last Modified: 01-SEP-99



## Article #18078: Delphi 3/400:Table list displays only 499 tables

Question and Answer Database

FAQ3078D.txt - Delphi 3/400:Table list displays only 499 tables

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

Question:

I have Delphi 3/400. How come I only see 499 tables in a table list?

Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

There is no longer a limit.

4/1/99 4:11:23 PM

Last Modified: 01-SEP-99

## Article #18080: Delphi 3/400: ApplyUpdates doesn't work

Question and Answer Database

FAQ3080D.txt - Delphi 3/400: ApplyUpdates doesn't work

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I have Delphi 3/400. How come when using a MIDAS connection, ApplyUpdates doesn't work.

### Answer:

To solve this problem obtain the latest Delphi/400 patch from:  
"<http://libra.netmastersllc.com/borland400/patches.html>"

There is no way to know if a field is NULL or contains NOTHING. so, the Resolver will replace each field with no data by a NULL. In most case, this will generate a error on the AS/400. (if your fields don't allow NULL). That's why you should had something like this on the server side :

```
procedure TDemoServer.MyProviderGetData(DataSet: TClientDataSet);
var
  i:integer;
begin
  with DataSet do
    begin
      while not EOF do
        begin
          Edit;
          for i:=0 to fieldcount-1 do
            begin
              if fields[i].IsNull then
                fields[i].asString:='';
            end;
          Post;
          Next;
        end;
    end;
end;
```

end;  
end;

4/1/99 4:12:08 PM

Last Modified: 01-SEP-99

## Article #18082: Getting a cool menu in your app

Question and Answer Database

FAQ3082D.txt :Getting a cool menu in your app

Category :VCL

Platform :Win95/NT

Product :Delphi4.x,

Question:

I want to have a menu that looks like Delphi 4's menu in my app.  
How can I do that?

Answer:

1. Drop down a TControlBar. Align it to Client.
2. Drop down a TToolBar inside of the TControlBar.
3. Set Flat and ShowCaptions of the TToolBar to True.
4. Create as many TToolButtons on the TToolBar as you'd like.
5. Set Grouped to True for all TToolButtons. This will enable your menu to pop up menu after menu as you sweep across the menu (if a menu is already pulled down).
6. Drop down a TMainMenu, and make sure it's *\*not\** attached to the form it's on.
7. Create the entire menu, with all its entries.
8. For each TToolButton, set its MenuItem to the corresponding menu item in the TMainMenu.

12/01/1998 1:39:32 PM

Last Modified: 01-SEP-99

## Article #18084: Running 16-bit Delphi on Windows NT4

Question and Answer Database

FAQ3084D.txt      Running 16-bit Delphi on Windows NT4

Category        :Miscellaneous

Platform        :All

Product         :All 32 bit

Question:

When running 16-bit Delphi on Windows NT 4.0, running or debugging in the IDE locks up the system. What causes this and how can it be fixed?

Answer:

USERHTML

The problem is caused by running 16-bit Delphi on a 32-bit operating system it was never intended nor certified to run on. A known conflict between 16-bit Delphi and NT 4.0 is enabling Delphi's environment options "Minimize on run" and "Integrated Debugging" at the same time. Disabling the "Minimize on run" usually solves this problem, and is recommended as a workaround.

5/26/98 3:12:16 PM

Last Modified: 01-SEP-99

## Article #18086: Installing Delphi from a hard drive.

### Question:

How do I install Delphi from a local hard drive?

### Answer:

You will need approximately 460 MB of hard drive space to do the hard drive install (for Delphi 5 Enterprise version). 230 MB for the complete installation and free space to hold the files temporarily while doing the installation.

1. Put the Delphi CD in the drive and choose "Browse the CD, or view the CD in Windows Explorer."
2. Copy the "Install" folder to the root directory of the local drive you will be installing from (i.e. not inside any subfolders).
3. Copy the "Runimage" folder to the root directory of the same drive. This may take a very long time. Verify that the entire folder was copied correctly. Right-click on the folder and choose properties. This folder should contain the same number of files that the one on the CD-ROM contained.
4. Copy all of the remaining files (autorun.inf, delphi.ico, deploy.txt, etc.) to the root directory of the same drive.
5. Reboot into Safe Mode or VGA mode (optional) and double click on the "Install.exe" that you now have on your hard drive and install Delphi as you normally would.
6. Restart your computer and verify that Delphi runs correctly. Once you have done this you can delete all the files that you copied over from the CD-ROM.

Last Modified: 10-OCT-00

## Article #18088: "PPC PARTNER LU NAME NAME NOT FOUND OR INVALID" error

Question and Answer Database

FAQ3088D.txt - "PPC PARTNER LU NAME NAME NOT FOUND OR INVALID" error

Category :AS400  
Platform :Win95/NT  
Product :DelphiAS400,

Question:

Why am I getting an error of: PPC PARTNER LU NAME NAME NOT FOUND OR INVALID when trying to connect to my AS/400 using Delphi/4000?

Answer:

This error can occur when the port to the server is configured incorrectly. For Delphi 3, the correct port is 5003. For Delphi 4 the correct port is 4003.

4/1/99 4:12:32 PM

Last Modified: 01-SEP-99

## Article #18090: Creating Delphi 4 Style Toolbars/Menus

Question and Answer Database

FAQ3090D.txt :Creating Delphi 4 Style Toolbars/Menus  
Category :Miscellaneous  
Platform :Win95/NT  
Product :Delphi4.x,

Question:

How do I create Delphi 4 Style toolbars (like MS Office 97)?

Answer:

Do the following:

- 1) Drop a ControlBar component onto a form.
- 2) Set the Align property of the ControlBar to alTop.
- 3) Drop a ToolBar component onto the ControlBar.
- 4) Set the Align property of the ToolBar to alNone.
- 5) Set the Flat property of the ToolBar to TRUE.
- 6) Set the ShowCaptions property of the ToolBar to TRUE.
- 7) Right Click on the ToolBar and choose New Button;  
repeat a few times.
- 8) Drop a PopUpMenu for each ToolButton you added  
to the ToolBar onto the form.
- 9) Right click on each PopUpMenu component and choose  
Menu Designer- create Menu Items for each.
- 10) Set the DropDownMenu property of each ToolButton on  
the ToolBar to its respective PopUpMenu.
- 11) Set the AutoSize property of the ToolButtons to TRUE.

12/01/1998 3:05:01 PM

Last Modified: 01-SEP-99



## Article #18727: Merging the configuration file has failed

Question and Answer Database

FAQ: FAQ3727B - Merging the configuration file has failed

Category: Install Issues

Platform: Win95/NT

Product: BC++5.x, C++Builder1.0, C++Builder3.x, Delphi2.x, Delphi3.x,

Question:

I'm deploying my BDE application and the install generates the error "Merging the configuration file has failed". What is causing this?

Answer:

The most likely cause of this error is using a BDE Deployment tool with BDE 5 or later, such as InstallShield, that doesn't know about BDE 5 files.

Specifically what causes the error is that BDE 5.0 uses a new mechanism to for language drivers. The error occurs because the language driver being merged can't be found.

To solve the problem:

1. When you update your BDE, install to the same directory the BDE is already installed to.
2. Add BANTAM.DLL, \*.BTL and CHARSET.CVB from your BDE directory to the BDE/IDAPI Files group.

Another possible causes of this problem are:

1. Running an application that uses the BDE while deploying your application.
2. The target machine's preexisting Idapi32.cfg file is set to read-only or is in any other way inaccessible.

3. The IDAPI configuration file on the target machine is corrupt.

For additional information see document Q100670 on  
<http://support.installshield.com>

7/13/99 11:04:11 AM

Last Modified: 01-SEP-99

## Article #18737: Oracle 8 and returning the REF CURSOR from a stored procedure

Question and Answer Database

FAQ3737B.txt Oracle 8 and returning the REF CURSOR from a stored procedure

Category :Database(Oracle8)

Platform :Win95/NT

Product : BDE 5.0 (fixed in BDE 5.01)

Question:

Using Delphi 4 (w/BDE 5) with the Oracle 8.0.4 client and the Oracle 8 SQL Link driver, I am trying to execute a stored procedure that returns a REF CURSOR type. No matter what I seem to try I get the error "General SQL error ORA-01036: illegal variable name/number" when I try to execute the procedure. What am I doing wrong?

Answer:

When using BDE 5.0 there is a known problem with returning a REF CURSOR type from a stored procedure when using the Oracle 8 SQL Link Driver (in BDE 5.0). BDE 5.01 or later resolves the problem.

Last Modified: 16-FEB-00

## Article #18749: Unknown Internal Operating System Error

Question and Answer Database

FAQ3749B.txt      Unknown Internal Operating System Error  
Category        :BDE  
Platform        :Win95/NT  
Product        :    BC++5.x      C++Builder1.0      C++Builder3.x      Delphi2.x  
Delphi3.x      Delphi4.x

Question:

Why am I getting an Unknown Internal Operating System Error when opening a Paradox or dBASE Table?

Answer:

This error can occur when attempting to open a Paradox or dBASE table whose language driver is not installed on your computer. In most circumstances this means a bad BDE install and BDE should be reinstalled.

One way to determine a likely bad install is by running BDE administator and attempting to modify the langauge driver for the Paradox driver. If only one language driver is available, it indicates a bad install.

This problem is most likely to occur with BDE 5.0 when deploying with a deployment tool that does not know about BDE 5.0. The problem occurs because BDE 5.0 includes a new mechanism for language drivers. To deploy BDE 5.0 language driver features with an older development tool be sure include BANTAM.DLL and \*.BTL from the BDE directory on your deployment machine.

3/26/99

Last Modified: 01-SEP-99

## Article #18752: Creating NonMaintained or .NDX indexes on dBASE Tables

Question and Answer Database

FAQ3752B.txt      Creating NonMaintained or .NDX indexes on dBASE Tables  
Category        :BDE  
Platform        :All Windows  
Product        :    C++Builder1.0    C++Builder3.x    Delphi2.x    Delphi3.  
x    Delphi4.x

Question:

TTable.AddIndex does not contain functionality to create NonMaintained or .NDX style indexes.

How can I create these types of indexes?

Answer:

TTable.AddIndex does not contain functionality to create this style index. You can use DbAddIndex instead.

Please refer to the BDE API Help file for examples on how to use DbAddIndex.

3/26/99

Last Modified: 01-SEP-99

## Article #18758: Registering service applications under NT

Question and Answer Database

FAQ3758D.txt - Registering service applications under NT

Category :Delphi 4.x  
Platform :Windows NT  
Product :Delphi4.x,

Question:

How do I register service applications I have created in Delphi 4 with the operating system?

Answer:

Run the service with the command line parameter "-install".  
(ie., "MyService -install" will do the trick.)

4/1/99 2:41:55 PM

Last Modified: 01-SEP-99

## Article #18761: Automating Word 8 (word97)

Question and Answer Database

FAQ3761D.txt :Automating Word 8 (word97)  
Category :ActiveX/OLE/COM/ActiveForm  
Platform :All Windows  
Product :All32Bit,

Question:

How do I automate word 8 (aka word97)?

Answer:

You can get to any of the interfaces exposed by the word automation server. These can be found by loading the file "MSWORD8.OLB" into Delphi which will display the type library information. The following code sample demonstrates creating a new word document and inserting text into the document via automation.

uses

ComObj;

procedure TForm1.Button1Click(Sender: TObject);

var

Word97: Variant;

begin

Word97 := CreateOLEObject('Word.Application');

Word97.Documents.Add;

Word97.Selection.TypeText('Wow BOB woW');

Word97.Visible := True;

Word97:= UnAssigned;

end;

3/10/99 10:15:08 AM

Last Modified: 01-SEP-99

## Article #18763: Error: Too many fields in table create

### Question:

I'm getting the error "Too many fields in table create" when either using DbiCreateTable (BDE API) or TTable.Create (Delphi) or TTable->Create (C++ Builder). What is the limit to the number of fields I can create in a table using these functions?

### Answer:

The number of fields varies by database. See <http://community.borland.com/article/0,1410,15159,00.html> for a complete listing. If your database supports a larger number of fields you may create a table with a larger number of fields using a TQuery instead.

Last Modified: 08-SEP-00



## Article #18765: Detecting if a mouse has a wheel

Question and Answer Database

FAQ3765D.txt - Detecting if a mouse has a wheel

Category :VCL  
Platform :Win95/NT  
Product :Delphi4.x,

Question:

How can I detect if the currently installed mouse has a wheel?

Answer:

By taking advantage of the new global mouse object which contains information relating to the mouse. The mouse object has a boolean property called "WheelPresent" which is true if a mouse wheel is present or false if none is detected.

4/2/99 12:20:23 PM

Last Modified: 01-SEP-99

## Article #18767: Oracle's Oracle 7.x ODBC drivers, VARCHAR2 >255

Question and Answer Database

QNA :Oracle's Oracle 7.x ODBC drivers, VARCHAR2 >255  
Category :Database (ODBC)  
Platform :Win95/NT  
Product : All32Bit

### Question:

I am unable to see or edit data from my VARCHAR2 fields when they are defined as >255 in size.

### Answer:

This is a problem that has been seen with only the Oracle ODBC drivers.

"Live" TQueries can be used in this situation.

The Oracle SQL Links drivers and Intersolv Oracle ODBC drivers do not exhibit the same behavior.

As a general rule ODBC drivers should be ODBC level 2 compliant when used with the Borland Database Engine.

3/26/99

Last Modified: 01-SEP-99

## Article #18770: IBM CAE 5 PTF 3 and 4 for DB2 Universal server and client

Question and Answer Database

FAQ3770B.txt IBM CAE 5 PTF 3 and 4 for DB2 Universal server and client

Category :Database (DB2)

Platform :Win95/NT

Product : All32Bit

Question:

When working with IBM's CAE v5 and IBM Universal server with the Borland DB2 SQL Link driver for Win95 and WinNT I sometimes receive unusual errors.

Answer:

While the versions of the Borland DB2 SQL Links driver <=5.0 have not been tested with IBM's CAE 5 and Universal server you should apply the IBM PTF versions 3 and/or 4 to both the CAE client and Universal server.

3/26/99

Last Modified: 01-SEP-99

## Article #18772: Informix SE, errors while working with Borland Informix SQL Links

Question and Answer Database

FAQ3772B.txt Informix SE, errors while working with Borland Informix SQL Links

Category :Database (Informix)

Platform :Win95/NT

Product : All32Bit

Question:

I sometimes receive strange errors while working with an Informix SE server and the Informix SQL Links driver.

Answer:

Only versions of Informix ONLINE are certified for use with the Borland SQL Links.

Informix SE is not a tested or supported server when using the Borland

Informix SQL Links driver.

All versions ( $\geq$  V3.0) of the 32bit Informix SQL Links driver are certified against Informix 7 server. SQL Links v5 includes an Informix 9

driver that has been certified with the Informix 9 server.

While previous versions of Informix ONLINE have not been tested with the

32bit Informix SQL Links driver most functionality should be available

as implied by Informix's ESQL/C (versions 7 and 9).

3/26/99

Last Modified: 01-SEP-99

## Article #18774: Microsoft SQL Server "Unexpected EOF from SQL Server"

Question and Answer Database

FAQ3774B.txt Microsoft SQL Server "Unexpected EOF from SQL Server"  
Category :Database (MSSQL)  
Platform :All Windows  
Product : All32Bit

### Question:

I receive the error "Unexpected EOF from SQL Server" 109 10008 "Bad token from SQL Server" when working with my MS SQL Server. Why am I receiving this error?

### Answer:

Banyan Vines Network, MS SQL Server 6.x, A select with an "order by" will cause a "connection is already in use..." error MS's ISQL will bring up the error "Unexpected EOF from SQL Server" Error: "109 The pipe has been ended." [OS/2 and NT] "10008 Bad token rom SQL Server: datastream processing out of sync." Often caused by an abruptly terminated network connection, which causes a damaged Tabular Data Stream token to be read by the client. Please see Microsoft's knowledge Base article Q73215 for more details on this error.

3/26/99

Last Modified: 01-SEP-99

## Article #18776: Microsoft SQL Server error 17824 Unable to write to ListenOn connection

Question and Answer Database

QNA :Microsoft SQL Server error 17824 Unable to write to ListenOn connection  
Category :Database (MSSQL)  
Platform :Win95/NT  
Product : All32Bit

### Question:

I am receiving the error "17824 Unable to write to ListenOn connection 'xxx', oginname 'xxx', hostname 'xxx'." when working with my MS SQL server.

What is causing this error?

### Answer:

Please see the Microsoft SQL server documentation for further information on this topic.

After establishing a number of concurrent application connections to the MS SQL Server the following error is encountered:

MS SQL Server error "17824 Unable to write to ListenOn connection 'xxx', loginname 'xxx', hostname 'xxx'."

### POSSIBLE CAUSE:

It is possible that the PROCEDURE CACHE percentage is set too low.

Please see your MS SQL Server documentation and NT Event Viewer for more information on debugging network errors and tuning your MS SQL Server's PROCEDURE CACHE.

Relevant SQL Server options: MAX PROCEDURE CACHE ... MAX WORKER THREADS

### PROCEDURE CACHE

The procedure cache holds entries for each procedure and other maintenance overhead.

Settings for the PROCEDURE CACHE include:

MAX PROCEDURE CACHE ACTIVE %  
MAX PROCEDURE CACHE USED%  
PROCEDURE CACHE USED%

MAX WORKER THREADS

The max worker threads option allows control over the number of threads allocated to the user pool.

3/26/99

Last Modified: 01-SEP-99

## Article #18778: Sybase DBLIB, Access Violation in KERNEL32.DLL

Question and Answer Database

FAQ3778B.txt     Sybase DBLIB, Access Violation in KERNEL32.DLL  
Category        :Database (Sybase)  
Platform        :Win95/NT  
Product         :     All32Bit

### Question:

My application is crashing in KERNEL32.DLL when a connection is attempted using the SQL Links DBLIB driver (SQLSYB32.DLL) and the Sybase 11.1 Win95/NT client.

Why is this happening?

### Answer:

This problem has been addressed in BDE/SQL Links v4.01 and above.

(Please check Delphi, C Builder, and Visual dBASE Client Server product release info for more information on new versions of the BDE and SQL Links)

DBLIB and CTLIB connection types to Sybase servers through Borland SQL Links are determined by the Sybase driver DLL32 param (either SQLSYB32.DLL or SQLSSC32.DLL)

SQLSYB32.DLL     (DBLIB) - loads Sybase LIBSYBDB.DLL  
SQLSSC32.DLL     (CTLIB) - loads Sybase LIBCT.DLL

### Tested using:

Sybase 11.1 LIBSYBDB.DLL dated 9/4/96

If the 11.1 LIBSYBDB.DLL is replaced with a Sybase 10 LIBSYBDB.DLL (dated 8/2/95) then the connection is OK.





## Article #18780: Sybase CTLIB, multiple stored procedure result sets

Question and Answer Database

FAQ3780B.txt     Sybase CTLIB, multiple stored procedure result sets  
Category     : Database (Sybase)  
Platform     : Win95/NT  
Product      :     All32Bit

Question:

I wish to return result set from multiple stored procedures or open tables/tqueries after returning results from a stored procedure but receive this error instead:

```
ct_send(): user api layer: external error: This routine cannot be called because another command structure has results pending.
```

What can I do to avoid this error?

Answer:

Sybase (CTLIB SYBSSC32.DLL) SQL Links Driver, BDE 4.x

Solution:

Use a tquery to "open" the stored procedure.

(For additional info please see the topic on Sybase in the BDE32.HLP file by using the help file Index tab to search for SYBASE)

Delphi Example:

Add "BDE" to your unit's uses clause

Button Click procedure to open results from a Tquery ("exec storedproc"):

```
procedure TForm1.Button2Click(Sender: TObject);  
begin
```

```
  query1.prepare;  
  dbisetprop(hdbiobj(query1.StmtHandle), stmtEXECASCURSOR, 1);
```

```
query1.open;
```

In this example:  
the StoredProc name is LINEITEM\_RSLT:

```
CREATE PROCEDURE LINEITEM_RSLT AS  
(  
SELECT * FROM lineitem  
)
```

Tquery1 sql strings for this example would be:

```
exec LINEITEM_RSLT
```

3/26/99

Last Modified: 01-SEP-99

## Article #18782: DB2 Win3.1 and Win95/WinNT combined installation.

Question and Answer Database

FAQ3782B.txt DB2 Win3.1 and Win95/WinNT combined installation.

Category :Database (DB2)

Platform :Win95/NT

Product : All32Bit

Question:

What can I do to allow connections from both the Win 3.x and Win 95/NT IBM CAE on the same PC?

Answer:

Both the 16 and 32 bit IBM CAE use a file called DB2ODBC.DLL. Each install may overwrite the copy of the previous install's version (16 vs. 32bit) of the file. If the \BIN dir of both installs is in the environment path and the corresponding copy of DB2ODBC.DLL is copied to its respective directory after each install then the \WINDOWS\SYSTEM copy of DB2ODBC.DLL can be deleted or renamed.

3/26/99

Last Modified: 01-SEP-99

## Article #18784: IBM DB2 CLI driver errors SQL0551N and SQL0805N

Question and Answer Database

FAQ3784B.txt IBM DB2 CLI driver errors SQL0551N and SQL0805N

Category :Database (DB2)

Platform :Win95/NT

Product : All32Bit

Question:

```
[IBM][CLI Driver][DB2/NT] SQL0551N "MYUSERNAME" does not have the
privilege to perform "GRANT" on object "NULLID.SQLL16W4".
SQLSTATE=42501
```

What can I do to help address the cause of this error?

Answer:

The following error may appear either when trying to list tables via a DB2 alias or when a table is chosen to be viewed on an OS/2 or NT DB2 server:

```
[IBM][CLI Driver][DB2/NT] SQL0551N "MYUSERNAME" does not have the
privilege to perform "GRANT" on object "NULLID.SQLL16W4".
SQLSTATE=42501
```

```
[IBM][CLI Driver][DB2/NT] SQL0805N Package "NULLID.SQLL66W4" was not
found. SQLSTATE=51002 "GRANT" on object.
```

It may be necessary to issue the following after logging in as the DBA:

```
grant execute on package NULLID.SQLL16W4 TO PUBLIC
```

3/26/99

Last Modified: 01-SEP-99

## Article #18786: BDE >=4.0, Intersolve ODBC 3.0, TQuery CachedUpdates, TUpdateSql, ApplyUpdates, null/blank string

Question and Answer Database

FAQ3786B.txt    BDE >=4.0, Intersolve ODBC 3.0, TQuery CachedUpdates, TUpdateSql, ApplyUpdates, null/blank string  
Category     : Database (ODBC)  
Platform     : Win95/NT  
Product      : All32Bit

Question:

When working with BDE 4.0, Intersolve ODBC 3.0, Delphi TQuery CachedUpdates, TUpdateSql I receive the error "General SQL error. [INTERSOLV][ODBC Oracle driver]Character, decimal, and binary parameters cannot have a precision of zero. Error in parameter n."  
What might be causing this error?

Answer:

The problem does not occur when using CachedUpdates w/TTables

"General SQL error. [INTERSOLV][ODBC Oracle driver]Character, decimal, and binary parameters cannot have a precision of zero. Error in parameter n."

Add a new string value called "Workarounds" to:

\HKEY\_CURRENT\_USER\Software\ODBC\ODBC.INI\MYODBCDSN and set its value to 1048576

Currently under investigation.

From /IVODBC/READ.ME (Intersolv 3.0 ODBC readme):

"WorkArounds=1048576. Some applications incorrectly specify a precision of 0 for character types when the value will be SQL\_NULL\_DATA. This option overrides the specified precision and sets the precision to 256."

3/26/99

Last Modified: 01-SEP-99

## Article #18788: Oracle Integer to BDE or Delphi float, ENABLE INTEGER

Question and Answer Database

FAQ3788B.txt Oracle Integer to BDE or Delphi float, ENABLE INTEGER  
Category :Database (Oracle)  
Platform :Win95/NT  
Product : All32Bit

### Question:

I've created an Oracle table using the INTEGER type but the BDE shows the INTEGER field as a FLOAT even though I have ENABLE INTEGERS set to True.

### Answer:

The Oracle server will set the fields precision to '\*' and the scale to '0' (MYFIELD(\*,0) for example) when a field is created as INTEGER. This translates to an Oracle precision of 38. The maximum precision for Oracle NUMBERS INTEGERS and SMALLINTS is 38. The maximum precision for Delphi/BDE INTEGERS is 10 (-2,147,483,648 to 2,147,483,647 or 9,999,999,999).

3/26/99

Last Modified: 01-SEP-99



## Article #18789: SQL LINKS >=4.0 BLOB SIZE parameter

Question and Answer Database

FAQ3789B.txt SQL LINKS >=4.0 BLOB SIZE parameter

Category :BDE Features

Platform :Win95/NT

Product : All32Bit

Question:

What is the BDE BLOB SIZE parameter used for?

Answer:

[\*Please note that the behavior described below applies only to BDE/  
SQL

Links >=4.0. The Version 4.01 BLOB SIZE param applies to viewing  
"dead"

results as well as live results containing text/blob data if BLOB  
SIZE

is set higher than 32k]

Increasing the BLOB SIZE from the default of 32 will currently  
prevent

opening "dead" (unindexed or Request Live = false) results from  
returning

text data when opening. In other words if BLOB SIZE is increased from  
32

then only TQueries that are opened as live results will return data.

From SQL Links 4.0 Help:

BLOB SIZE

Determines the fetch buffer size for dead BLOBs. Applications that  
deal with dead BLOBs using dead table opens or queries or batchmoves  
can set a maximum limit on the size of BLOBs to fetch. Setting this  
parameter to 64 means your application can fetch BLOBs of up to 64K.  
This parameter does not apply to live table opens.

Default Value: 32

Range : >32 and <1000

\*

Interbase BLOBs are always "live" and are therefore not affected by the BLOB SIZE param

\*

DB/2

Binding a large buffer for fetching dead blobs by setting the BLOB SIZE

setting to max (65535) DB2 returns SQL error "SQL0973N Not enough storage is available in the QUERY\_HEAP to process the statement. SQLSTATE=57011"

Note:

This is a vendor restriction. Increasing the QUERY\_HEAP on the server should solve the problem.

3/26/99

Last Modified: 01-SEP-99

## Article #18798: SQL Explorer reports Oracle 8 database as Oracle 7 database

Question and Answer Database

FAQ3798B.txt SQL Explorer reports Oracle 8 database as Oracle 7 database

Category :Database(Oracle8)

Platform :Win95/NT

Product : BC++5.x C++Builder1.0 C++Builder3.x Delphi2.x

Delphi3.x Delphi4.x

Question:

Why does SQL Explorer report the database type as Oracle 7.3.2 when I connect to an Oracle 8 database?

Answer:

SQL Explorer queries the SM\$VERSION table using the sql statement

```
"select 'Oracle ' || VERSION_TEXT from SM$VERSION"
```

to determine the version of the database and it returns 7.3.2.

To verify that you are connecting to an Oracle 8 database query GV\$VERSION.

example: select BANNER from GV\$VERSION

3/26/99

Last Modified: 01-SEP-99

## Article #18804: Access Violation when opening BDE Object in DataModule of DLL

Question and Answer Database

FAQ3804B.txt    Access Violation when opening BDE Object in DataModule of DLL

Category       :Database/VCL

Platform        :All Windows

Product        :   C++Builder1.0   C++Builder3.x   Delphi2.x   Delphi3.x  
x   Delphi4.x

Question:

Why do I receive an Access Violation when I open any BDE object (Session, Database, Stored Procedure, Table) contained in a Data Module in my DLL?

Answer:

Forms and Data Modules in DataModules in DLLs are not autocreated. You instantiate them yourself. The most likely place to do this is in DLL entry point.

Where DMod is the name of your Data Module, in C++ Builder the code would look as follows:

```
if reason = DLL_PROCESS_ATTACH
    DMod = new TMod(NULL);
```

in Delphi it looks like:

```
if reason = DLL_PROCESS_ATTACH
    DMod := TMod.Create(nil);
```

3/26/99

Last Modified: 01-SEP-99

## Article #18806: Sybase/"record has been modified by another user" error with datetime fields

Question and Answer Database

FAQ3806B.txt Sybase/"record has been modified by another user" error with datetime fields

Category :Database (Sybase)

Platform :Win95/NT

Product :BDE versions prior to 5.10

### Question:

I have a table in a Sybase database (version 10.x or 11.x) that has a datetime field. I was able to use/edit data in this table until I applied a patch to my Sybase client (versions affected below). Now after applying the patch, whenever I attempt to update records that were written to the table with the unpatched client I get the error "record has been modified by another user". What is wrong?

### Answer:

As of Open Client Version 11.1.1, CT-Lib now uses a small datetime as its default. (A small datetime keeps year, month, day, hour & minute). This means that the 11.1.1 client will not even attempt to send second and millisecond data to the server when you post a record. If the data was written with second and millisecond precision, there is no way of reselecting it using Open Client 11.1.1. (for example 11:30:23.111 doesn't equal 11:30:00.000) In versions earlier than this (for example 11.1.0) CT-Lib used long datetime as its default. (A long datetime keeps everything a small datetime does plus seconds and milliseconds). As a result, when you upgrade your Sybase client you will see this problem.

One possible workaround is to change your Update Mode to be WhereKeyOnly or WhereChanged, but if the datetime field is part of the key or you change the value of the datetime field you will still see this problem. You could also use small datetime fields when you are designing your tables and avoid this problem entirely. If you have a database which requires second and/or millisecond precision for the datetime fields, you have a few options:

1. Use Open Client 11.1.0 or 10.0.4 with CT-Lib
2. Use Open Client 11.1.1 with DB-Lib
3. Write your own implementation of second and millisecond times in another field and use it in conjunction with a short date.

If these alternatives are not acceptable to you, we encourage you to provide this feedback to Sybase Customer Service or your Sybase sales representative.

### Versions Affected:

Open Client 11.5, 11.1.1, 11.1 with ebf7398 applied, 10 with ebf7438.



## Article #18808: Delphi4, Oracle 7.x stored procedures with multiple parameters with one being a REF Cursor

Question and Answer Database

FAQ3808B.txt Delphi4, Oracle 7.x stored procedures with multiple parameters with one being a REF Cursor

Category :Database (Oracle)

Platform :Win95/NT

Product : Delphi4.x

Question:

I have a stored procedure on my Oracle 7.x server that has multiple parameters with one of them being a REF Cursor type. When I try to execute this procedure I get "invalid cursor" errors.

I used to be able to execute this procedure in Delphi 3, why doesn't it work in Delphi 4?

Answer:

Parameter binding behavior was changed in Delphi 4.

In Delphi 4 if you don't give a parameter a value it assumes that the value you wanted was NULL. This causes a problem for the REF Cursor type in Oracle because the server doesn't accept NULL as an input value

for a REF Cursor type. The workaround for this issue is to set the value

of the REF Cursor parameter to 0 (zero) at either design time or at runtime before the procedure is executed. This change in behavior will

be accounted for in a later version of the BDE (> 5.0).

3/26/99

Last Modified: 01-SEP-99

## Article #18823: Sybase 11.5 server causes problems for SQL Explorer

Question and Answer Database

FAQ3823B.txt - Sybase 11.5 server causes problems for SQL Explorer

Category :Database (Sybase)

Platform :Win95/NT

Product :C++Builder1.0, C++Builder3.x, Delphi2.x, Delphi3.x, Delphi4.x

### Question:

When connecting to a Sybase 11.5 server the following problems occur in SQL Explorer:

1. Type is shown as SYBASE: Unknown Version: Mapping error: Bad template: SQL Server/([0-9]+).
2. Only Columns, Indices and Triggers and shown below tables in the tree.
3. Selecting the text tab of table which has a numeric field results in the error dialog "Property Precision does not exist".

### Answer:

You can solve this problem by editing the DBX.DBI file in your bin directory. Find the line the reads:

```
SYBASE:IdentityRE=SQL Server/([0-9]+).
```

change it to:

```
SYBASE:IdentityRE=[SQL Server][Adaptive Server Enterprise]/([0-9]+).
```

This will resolve the problem.

Note that this problem does not occur in Delphi 5 or later, or C++ Builder 5 or later because the DBX.DBI file has been updated to include this fix.

Last Modified: 01-MAR-00



## Article #18825: Preventing changing to a page in a page control

Question and Answer Database

FAQ3825D.txt :Preventing changing to a page in a page control  
Category :VCL  
Platform :Win95/NT  
Product :Delphi3.x, Delphi4.x,

Question:

How do I prevent a user from tabbing to a specific page in a page control?

Answer:

```
uses
  CommCtrl;

procedure TForm1.PageControl1Changing(Sender: TObject;
  var AllowChange: Boolean);
var
  Info : TTCHitTestInfo;
  Index : Integer;
begin
  with Info do begin
    GetCursorPos(pt);
    pt := PageControl1.ScreenToClient(pt);
    flags := TCHT_ONITEM;
  end;
  Index := TabCtrl_HitTest(PageControl1.Handle,@Info);
  AllowChange := Index <> 2; // Or another test that makes sense
end;
```

12/01/1998 1:40:07 PM

Last Modified: 01-SEP-99

## Article #18827: Installing Delphi 4 From a Network Location

Question and Answer Database

FAQ3827D.txt :Installing Delphi 4 From a Network Location  
Category :Install Issues  
Platform :Win95/NT  
Product :Delphi4.x,

Question:

How do I install Delphi 4 on my machine from some place on the network?

(This does not include installing Delphi ON a network, only from a network. Delphi is intended to be installed on the machine that will be running it, not on a network server.)

Answer:

1a) Copy the needed files or the entire Delphi 4 CD to a location your machine can access on the network. The files needed for Delphi 4 installation are all of the files in the root directory of the CD (license.txt, inprise.gif, install.exe, etc.) as well as the entire Install and Runimage directories.

1b) Alternately, you could simply share the CD Drive.

2) On the machine where you will be installing Delphi, map a drive directly to that location so that the license.txt, install.exe, and other files are in the root directory of the drive. (You will get Error 113 if the files are not in the correct location).

3) Run install.exe from the drive you just mapped.

NOTE: Make sure that the filenames of the original files on the CD are not truncated when you copy them to your network, and make sure that you can read long filenames over your network.



## Article #18829: Vendor Init when using Oracle 7.2 with Delphi4/BDE5

Question and Answer Database

FAQ3829B.txt - Vendor Init when using Oracle 7.2 with Delphi4/BDE5

Category :Database (Oracle)  
Platform :Win95/NT  
Product :Delphi4.x,

### Question:

I have Delphi4/BDE5 and I am trying to connect to an Oracle server using the Oracle 7.2 client. Every time I try to connect I get a "Vendor Init error". I have read the Developer support Web pages and read the available documentation and everything is configured correctly. Why can't I connect?

### Answer:

The lowest version of the Oracle 7 client supported by BDE 5 is 7.3.x

3/26/99 10:41:04 AM

Last Modified: 01-SEP-99

## Article #19235: Oracle 8 and writing Data to a LOB (blob/clob) field.

Question and Answer Database

FAQ4235B.txt - Oracle 8 and writing Data to a LOB (blob/clob) field.

Category :Database(Oracle8)  
Platform :Win95/NT  
Product :Delphi4.x,

### Question:

I am using Oracle 8.0.4 as my database backend and when I attempt to write data to a Blob/Clob field I keep getting the error "ORA-22990: LOB locators cannot span

transactions". The code I am using used to work against Oracle 7 servers but against Oracle 8 it keeps raising this error. What is wrong?

### Answer:

As of Oracle 8, Oracle mandates that in order to write data to a LOB datatype ( Blobs and Clobs are subtypes of the LOB type) that you must be in a transaction. this is due to the

changes in the way oracle implemented the LOB data type. To get around this issue start a

transaction before you attempt to write data to a LOB data type.

for example if your code was (Delphi example)

```
Table1.Insert;  
Table1KEYFLD.AsInteger := new_val;  
Table1TESTFLD.LoadFromFile('D:\data\test.bmp');  
Table1.Post;
```

change it to

```
Databasel.StartTransaction;  
Table1.Insert;
```

```
Table1KEYFLD.AsInteger := new_val;  
Table1TESTFLD.LoadFromFile('D:\data\test.bmp');  
Table1.Post;  
Database1.Commit;
```

and this will get you around the issue.  
For more information on LOB data types see the Oracle 8 documentation.

3/26/99 10:51:08 AM

Last Modified: 01-SEP-99

## Article #19238: Date Field Precision For Borland Database Formats

Question and Answer Database

FAQ4238C.txt :Date Field Precision For Borland Database Formats  
Category :Database Issues  
Platform :All Windows  
Product :C++Builder1.0, C++Builder3.x,

Question:

What is the Date Field precision for Borland database formats?

Answer:

Date Field Precision for Paradox, Visual dBASE and InterBase are as follows:

Paradox

Date Field Range is from January 1, 9999 BC to December 31, 9999 AD. Paradox correctly handles leap years and leap centuries and checks all dates for validity. Paradox treats all BC years as leap years.

dBase

In dBASE III+ to Visual dBASE for Windows 5.5  
Date Field Range is from January 1, 0000 to December 31, 9999. For all versions of dBASE from version III+ to the current Visual dBASE for Windows 5.5, date fields are stored as a string in the format YYYYMMDD. In dBASE III+ a new SET CENTURY command was added for the approaching 21st century. SET CENTURY ON allows display and editing of the 4 digit year. SET CENTURY OFF limits display and entry of the year to 2 digits. All calculations and storage involving date fields handle the year correctly regardless of the state of SET CENTURY. With SET CENTURY OFF, the command, STORE {01/01/2000} TO MYDATE, will display the year as 00 even though it is stored correctly on disk and in memory as the string 20000101. Note that the YEAR function always returns a 4 digit year.

InterBase

Date Field Range is from January 1, 100 to December 11, 5941. InterBase supports a DATE data type that stores dates as two 32-bit longwords. An InterBase DATE data type includes information about year, month, day of the month, and time.

12/16/1998 12:14:28 PM

Last Modified: 01-SEP-99



## Article #19240: SQL against Access tables with special characters in field names, such as #

Question and Answer Database

FAQ4240B.txt - SQL against Access tables with special characters in field names, such as #

Category :Database (MSACCESS)

Platform :All Windows

Product :C++Builder1.0, C++Builder3.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How do I execute SQL against an Access table that has a special character in a field name, such as a #? Whenever I try my statement I get a "General SQL Error".

Answer:

Place [] around the field name. Example:

```
SELECT * FROM Table WHERE Table.[Phone #] = '(831)431-1000'
```

3/26/99 10:24:18 AM

Last Modified: 01-SEP-99

## Article #19242: Vendor INIT error when connecting to DB2 Version 5

Question and Answer Database

FAQ4242B.txt - Vendor INIT error when connecting to DB2 Version 5

Category :Database (DB2)

Platform :Win95/NT

Product :BC++5.x, C++Builder1.0, C++Builder3.x, Delphi2.x,  
Delphi3.x, Delphi4.x, VdBase7.x,

Question:

I have DLL32 for my DB2 driver in BDE Admin set to SQLDB2V5.DLL. How come whenever I try to connect I get a Vendor Init error?

Answer:

The most likely cause of this error is that a minimum of DB2 Version 5, FixPack 4 is not installed. Another cause is that the DB2/BIN directory is not in your path.

3/26/99 10:28:46 AM

Last Modified: 01-SEP-99

## Article #19244: SQL Error returns error when trying to get TableList for DB2

Question and Answer Database

FAQ4244B.txt - SQL Error returns error when trying to get TableList for DB2

Category :Database (DB2)  
Platform :Win95/NT  
Product :C++Builder1.0, C++Builder3.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How come when I try to get a list of tables for my DB2 Database I get an error?

Answer:

This error occurs when connecting to the DB2 Version 5 Client and a minimum of FixPack 4 is not installed.

3/26/99 10:29:07 AM

Last Modified: 01-SEP-99

## Article #19254: Interbase, cached updates and Blob not found error

Question and Answer Database

FAQ4254B.txt - Interbase, cached updates and Blob not found error

Category :Database (InterBase)  
Platform :Win95/NT  
Product :All32Bit,

Question:

My application is querying an Interbase table that contains blob fields and I am using cached updates. I can do some editing of the blob fields but after editing a few records I begin to get the error 'BLOB NOT FOUND'. Why am I getting this error and what can I do to avoid it?

Answer:

This issues arises because of the way that Interbase accesses blob fields. Interbase Blob handles are not guaranteed to stay consistent across transactions and therefore the record buffer that is cached may contain a Blob handle that is no longer valid (BLOB NOT FOUND). There are a few workarounds you can try:

1. Use a live query instead of cached updates
2. Use a TTable instead of cached updates
3. Use MIDAS
4. Refresh the query after each post (not recommended--too expensive)
5. Force an update of all blob fields when posting:

Example of #5:

```
TForm1.Query1BeforePost(DataSet: TDataSet);  
var  
    i: Integer;  
    temp: String;  
begin  
    for i := 0 to Query1.FieldCount-1 do
```

```
if (Query1.Fields[i].IsBlob) then
begin
    temp := Query1.Fields[i].AsString;
    Query1.Fields[i].AsString := temp;
end;
end;
```

This will force an update of all blob fields and refresh their handles.

3/26/99 11:00:00 AM

Last Modified: 01-SEP-99

## Article #19255: MS Access and tryingto detect Null blobs using TBlobField.IsNull method

Question and Answer Database

FAQ4255B.txt - MS Access and tryingto detect Null blobs using TBlobField.IsNull method

Category :Database (MSACCESS)  
Platform :Win95/NT  
Product :C++Builder3.x, Delphi3.x, Delphi4.x,

Question:

I have an MS Access table with blob fields in it. When I use the TBlobField.IsNull method it always returns True whether the blob field has data or is Null. This same test works fine with Paradox and if do a query against the MS access table to look for null blobs (for example select \* from blobtest where blob1 IS NULL) it detects and shows the records with empty blobs correctly. What is wrong and how can I work around it?

Answer:

The DAO engine does not know without opening the Blob if it is null or not. The BDE would have to open the all the Blob fields at every getrecord call for the TBlobField.IsNull

function to work correctly. This would be a huge performance hit even though IsNull may

never be used. A workaround is to use .AsString and check the length:

before:

```
(assume that Table1blob1 is a TBlobField)
if Table1blob1.IsNull then
```

after:

```
if Length(Table1blob1.AsString) = 0) then
```

3/26/99 11:00:59 AM

Last Modified: 01-SEP-99

## **Article #19257: Installation error -113**

Question and Answer Database

FAQ4257D.txt :Installation error -113

Category :Install Issues

Platform :All Windows

Product :Delphi

Question:

What does installation "Error ?113" mean?

Answer:

The machine that you are trying to install Delphi on has a 16-bit CD-ROM driver installed on it. The driver does not recognize long file names and therefore cannot find some of the files necessary for installation. Contact the vendor of the CD-ROM drive for an updated 32 bit driver or the computer system manufacturer (such as Dell, Micron or Gateway).

11/30/1998 12:28:29 PM

Last Modified: 19-JUN-00



## Article #19258: Installation error -117

Question and Answer Database

FAQ4258D.txt :Installation error -117  
Category :Install Issues  
Platform :All Windows  
Product :Delphi3.x, Delphi4.x, DelphiAS400,

Question:

What is installation "Error ?117"?

Answer:

A certain batch of CDs produced by Inprise contained a strange glue-like substance on them that causes this error to popup because a track on the CD cannot be read.

The solution is to clean the CD using Windex or some equivalent cleaner to remove the offending substance. The other possible cause of this error is a damaged or scratched CD, if cleaning the CD does not make the error cease then the customer should call the order desk (800-331-0877) to order a replacement CD.

11/30/1998 12:29:49 PM

Last Modified: 01-SEP-99

## Article #19259: Missing DATAPUMP.EXE error when deploying BDE with ISExpress

Question and Answer Database

FAQ: FAQ4259D - Missing DATAPUMP.EXE error when deploying BDE with ISExpress

Category: Miscellaneous

Platform: Win95/NT

Product: Delphi4.x,

Question:

What is this "Missing DATAPUMP.EXE" error when trying to deploy BDE with Installshield Express?

Answer:

Download and replace the files included in the installshield express update available from the Borland website at <http://www.borland.com/devsupport/delphi/downloads/> The cause of this error is that ISExpress is trying to deploy the DATAPUMP when it should not.

7/29/99 2:33:10 PM

Last Modified: 01-SEP-99

## Article #19260: Missing component palette under NT

Question and Answer Database

FAQ4260D.txt :Missing component palette under NT  
Category :Install Issues  
Platform :Windows NT  
Product :

### Question:

No errors during installation, but when I run Delphi there are no components on the component palette! Why?

### Answer:

Most likely you are using NT and need to use the "Registry Keys Only" installation choice as the currently logged in user to get all the necessary DLL's and registry key entries to run Delphi 4 under this user profile. Most likely Delphi 4 was installed by an administrator or a user with administrator rights and you are logged in as another user and trying to use Delphi 4.

Delphi fails to load properly because it cannot locate registry keys essential for running.

11/30/1998 1:19:07 PM

Last Modified: 01-SEP-99

## Article #19261: Random access violations in the IDE

Question and Answer Database

FAQ4261D.txt :Random access violations in the IDE  
Category :Install Issues  
Platform :Win95/NT  
Product :Delphi4.x,

Question:

Delphi used to work fine but now I get random access violations in the IDE! Why?

Answer:

You most likely installed a bad component or package that was not written/compiled for Delphi 4. Try uninstalling custom components one by one (or package by package) until the problem goes away, then get in touch with the component vendor regarding this issue.

12/01/1998 2:59:06 PM

Last Modified: 01-SEP-99

## Article #19262: Installation error "\_ins5176" at end of installation

Question and Answer Database

FAQ4262D.txt :Installation error "\_ins5176" at end of installation  
Category :Install Issues  
Platform :Windows 95  
Product :Delphi3.x, Delphi4.x,

### Question:

I get installation "Error \_ins5176" at the end of my installation, what does this mean?

Delphi seems to run fine after the install and reboot.

### Answer:

If the install continues and everything seems to work fine, then it can be ignored (the error happens during installation cleanup).

Otherwise, it's much more serious and there's no easy solution. Try uninstalling the product via the control panel's "Add/Remove applications" applet. This usually happens on a machine that already has another version of Delphi on it and has to do with the failure to register an OCX that is already registered.

3/10/99 10:36:43 AM

Last Modified: 01-SEP-99

## Article #19263: Multiple splash screens keep popping up

Question and Answer Database

FAQ4263D.txt :Multiple splash screens keep popping up  
Category :Install Issues  
Platform :Win95/NT  
Product :Delphi3.x, Delphi4.x,

Question:

Why do multiple splash screens keep popping up?

Answer:

Either you have a bad CD-ROM driver, or a slightly damaged CD that one computer may read fine but this particular computer is not as forgiving. Try a hard drive installation or update the CD-ROM driver. You will need to contact customer service in the case of a damaged CD.

12/01/1998 3:01:32 PM

Last Modified: 01-SEP-99

## Article #19264: "ICON IMAGE IS NOT VALID" error when installing

Question and Answer Database

FAQ4264D.txt : "ICON IMAGE IS NOT VALID" error when installing  
Category : Install Issues  
Platform : Win95/NT  
Product : Delphi3.x, Delphi4.x,

Question:

What does the "ICON IMAGE IS NOT VALID" error mean?

Answer:

Chances are good that you are using a Micron(or other factory assembled system) computer which has a proprietary version of the OS (95, NT, 98). You may be able to successfully install by uninstalling and re-installing, or in the worse case scenario you may need to install a non-Micron-proprietary OS (official OEM version of 95/98/NT)

12/01/1998 3:02:13 PM

Last Modified: 01-SEP-99

## Article #19271: Blob have limit of 1000000 bytes when connecting via ODBC

Question and Answer Database

FAQ4271B.txt - Blob have limit of 1000000 bytes when connecting via ODBC

Category :Database (ODBC)  
Platform :All Windows  
Product :All32Bit,

Question:

How come I'm getting the error "Invalid Blob Length" when trying to post a blob to an ODBC datasource when the blob is greater than 1000000 bytes?

Answer:

This is a current limitation of the BDE ODBC driver. Native drivers do not have this limitation.

3/26/99 1:42:05 PM

Last Modified: 01-SEP-99



## Article #19272: IDCO400.DLL and IDCO400.CFG not marked as shared DLLs.

Question and Answer Database

FAQ4272D.txt - IDCO400.DLL and IDCO400.CFG not marked as shared DLLs.

Category :AS400  
Platform :Win95/NT  
Product :DelphiAS400,

### Question:

I have Delphi/400 Development and Deployment installed. How come when I uninstall either one of them IDCO400.DLL and IDCO400.CFG are deleted?

### Answer:

These files are not marked as shared DLLs in

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\CurrentVersion\SharedDlls

so they are deleted during uninstall. You can either add the keys in the registry to make them be not be deleted, or make make backups before unstaill.

Note: This problem is only a problem if both the deployment and the developer installs are done on the same computer.

4/1/99 11:27:40 AM

Last Modified: 01-SEP-99

## Article #19274: Non-Blob column in table required to perform operation with MSSQL

Question and Answer Database

FAQ4274B.txt - Non-Blob column in table required to perform operation

Category :Database (MSSQL)

Platform :32 Bit

Product :BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, Delphi 5.x

Question:

How come I'm getting the error

"Non Blob column in table required to perform operation" when posting to my MSSQL database?

Answer:

There are two conditions where this error occurs:

1. You have an identity field and a field that is NOT NULL (required) and you post a record.

With BDE versions prior 5.10, all you can do is ignore the error and continue. The problem is resolved simply by getting BDE 5.10.

2. You have an identity field which is the primary key and a key field, you have a text field, and you post a record.

With BDE versions prior 5.10, all you can do is ignore the error and continue. If you have BDE 5.10 and Delphi, you can turn on the AutoRefresh property of your DataSet to resolve the problem.

If you have BDE 5.10 and development system other than Delphi 5, you can call DbisetProp and set curAUTOREFETCH (which is equal to 0x00050017) to true.

- o C++ Builder Example:

```
#include "bde.hpp"
#define curAUTOREFETCH 0x00050017;
Table1->Open();
DbisetProp(Table1->Handle, curAUTOREFETCH, TRUE);
```

- o Delphi version other than 5 example:

```
uses bde;  
const  
  curAUTOREFETCH = $00050017;  
...  
Table1.Open;  
DbiSetProp(hDbiObj(Table1.Handle), curAUTOREFETCH, Longint(TRUE))
```

11/16/99

Last Modified: 16-NOV-99

## Article #19279: Cannot Load IDAPI Service library when connecting to Access

Question and Answer Database

FAQ4279B.txt - Cannot Load IDAPI Service library when connecting to Access

Category :Database (MSACCESS)  
Platform :Win95/NT  
Product :BC++5.x, C++Builder1.0, C++Builder3.x, Delphi2.x,  
Delphi3.x, Delphi4.x, VdBase7.x,

Question:

I have DAO and BDE installed correctly, yet I'm getting the error "Cannot load IDAPI Service library". How come?

Answer:

The DAO requires ODBC to be installed. If ODBC is not installed this error will occur.

3/26/99 10:34:25 AM

Last Modified: 01-SEP-99

## Article #19280: Vendor INIT failed error when connecting via Delphi/400

Question and Answer Database

FAQ4280D.txt - Vendor INIT failed error when connecting via Delphi/400

Category :AS400  
Platform :All Windows  
Product :DelphiAS400,

### Question:

Connecting to my AS/400 via Delphi/400 used to work fine. Now whenever I try to connect I get the error "Vendor INIT failed". How come?

### Answer:

Your license key on the AS/400 has expired. To confirm this, run CO400/ABOUT on the 400. If it shows your license has expired, run CO400/CO400INS and enter your license key.

4/1/99 11:28:01 AM

Last Modified: 01-SEP-99

## Article #19285: How to prevent closing of an undocked ToolBar

Question and Answer Database

FAQ4285D.txt :How to prevent closing of an undocked ToolBar

Category :Miscellaneous

Platform :Win95/NT

Product :Delphi4.x,

Question:

How do I prevent a user from closing (hiding) an undocked ToolBar?

Answer:

Here's one possible solution. You have to create your own TToolDockForm descendant, and in it you want to make sure that the border icon is gone as well as preventing Alt+F4.

type

```
TMyToolDockForm = class(TToolDockForm)
    constructor Create(AOwner: TComponent); override;
    procedure DoClose(var Action: TCloseAction); override;
end;
```

```
constructor TMyToolDockForm.Create(AOwner: TComponent);
```

```
begin
```

```
    inherited Create(AOwner);
```

```
    BorderIcons := [];
```

```
end;
```

```
procedure TMyToolDockForm.DoClose(var Action: TCloseAction);
```

```
begin
```

```
    Action := caNone;
```

```
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
var
```

```
    i : Integer;
```

```
begin
```

```
    for i:=0 to ComponentCount-1 do
```

```
        if Components[i] is TToolBar then
```

```
            TToolBar(Components[i]).FloatingDockSiteClass :=
```

```
TMyToolDockForm;  
end;
```

3/10/99 10:38:05 AM

Last Modified: 01-SEP-99

## Article #19288: Error: "BORDBK40.DLL is missing or not registered"

Question and Answer Database

FAQ4288D.txt :Error: "BORDBK40.DLL is missing or not registered"  
Category :Install Issues  
Platform :All  
Product :Delphi4.x,

### Question:

What does "BORDBK40.DLL is missing or not registered" error mean during installation or startup of Delphi 4?

### Answer:

If the installation does not complete the cause of the problem is likely due to a software conflict. Some anti-virus software and certain NT Internet Services have been known to prevent BORDBK40.DLL and/or HTML.OCX from being registered with the operating system. In this case the best solution is to try eliminating the software in question from the picture and then try reinstalling Delphi 4. If that doesn't pose a viable solution you can also try step 1 and 2 outlines below to manually register the BORDBK40.DLL or HTML.OCX.

If the install does complete, and you get this message when running Delphi 4:

- 1) First, try running TREGSVR manually to register the dll. TREGSVR can be found in the "Delphi 4/bin" directory.
- 2) To register the DLL you would type "tregsvr bordbk40.dll" in the directory where the DLL resides. (This works providing you have the "Delphi 4/bin" directory on your path, otherwise you would include the path to bordbk40.dll)
- 3) If TREGSVR says the dll "has been successfully registered", but Delphi 4 doesn't work, then you have probably uninstalled Delphi 4 AFTER applying Update 2, and then re-installed without applying Update 2. Hence, the DLL is installed correctly but Delphi 4.00 does not know how to use the Delphi 4.01 version of the DLL. To correct this problem install update pack 2 and then your version of Delphi 4 will correspond to the version of the DLL on your system.



3/10/99 12:33:43 PM

Last Modified: 01-SEP-99

## Article #19291: Paradox errors when using SQL servers and cached updates

Question and Answer Database

FAQ4291D.txt :Paradox errors when using SQL servers and cached updates

Category :Database/VCL

Platform :Win32

Product :All32Bit,

Question:

Why am I getting a paradox lock or table busy error when I'm accessing data on a SQL server in my application (not paradox)?

Answer:

You are probably using Cached Updates in you project. When the BDE caches your data it is stored inside a paradox table. This happens when an application is being run off a server and there is more then one user executing the application. It is easily fixed by setting the paradox NetFile Directory and Private Directory to a temp directory such as " C:\ " (see the documentation for TSession for PrivateDir and NetFileDir properties)

3/10/99 10:22:25 AM

Last Modified: 01-SEP-99

**Article #19294: '#\$(\*(#' is not a valid identifier.**

Question and Answer Database

FAQ4294D.txt : '#\$(\*(#' is not a valid identifier.

Category :Install Issues

Platform :All Windows

Product :Delphi4.x,

Question:

Whenever I and save, even a blank project, I get "' is not a valid identifier" and if I try and save again I get "' is not a valid identifier."

Answer:

Uninstall Delphi, make sure no other software is running, especially antivirus software, and reinstall Delphi.

Alternately, a customer reported that ImageFox from ACDSystems was the source of his problems, but this has not been confirmed or disproved.

3/10/99 10:37:20 AM

Last Modified: 01-SEP-99

## Article #19296: In Memory Tables use System Language Driver

Question and Answer Database

FAQ4296B.txt - In Memory Tables use System Language Driver

Category :BDE Features  
Platform :All Windows  
Product :BC++4.5x, BC++5.x, C++Builder1.0, C++Builder3.x,  
Delphi2.x, Delphi3.x, Delphi4.x,

Question:

If I use DbiCreateinMemTable what language driver does it use?

Answer:

The language driver used is the System Language driver as specified in BDE Administrator by going to the Configuration tab, selecting System | INIT, LANGDRIVER.

3/26/99 10:31:50 AM

Last Modified: 01-SEP-99

## Article #19297: Deleting all records from dBASE table may work differently between 16 bit and 32 bit.

Question and Answer Database

FAQ4297B.txt - Deleting all records from dBASE table may work differently between 16 bit and 32 bit.

Category :BDE Features  
Platform :All Windows  
Product :BC++4.5x, BC++5.x, C++Builder1.0, C++Builder3.x,  
Delphi2.x, Delphi3.x, Delphi4.x, VdBase5.5x, VdBase5.6,  
VdBase7.x,

### Question:

In 16 bit Visual dBASE, DELETE without a WHERE clause causes records to be physically deleted. In 32 bit, it just marks them for deletion.

### Example

```
USE Foo
DELETE FROM Foo
```

### Answer:

This is design change. Other 32 bit clients may be effected as well. To insure that records are physically deleted from a dBASE table always issue a pack.

3/26/99 10:32:22 AM

Last Modified: 01-SEP-99

**Article #19312: AS/400 DataBaseName setting in BDE alias.**

Question and Answer Database

FAQ4312D.txt - AS/400 DataBaseName setting in BDE alias.

Category :AS400  
Platform :WIN32  
Product :Delphi3.x, Delphi4.x, DelphiAS400,

Question:

What is the DataBaseName setting for an AS/400 BDE alias used for?

Answer:

In the BDE Administrator you can set the DataBaseName to any of the AS/400 names that you have already setup in the Delphi AS/400 utility. After setting the DataBaseName you must apply the changes (control-a) and then you can test your connection.

4/1/99 4:13:03 PM

Last Modified: 01-SEP-99

## Article #19379: The beta version of the NetManage InterNet Controls has expired

Question and Answer Database

FAQ4379D.txt - The beta version of the NetManage InterNet Controls has expired

Category :Delphi 2.x  
Platform :All-32Bit  
Product :Delphi2.x,

### Question:

How come I get the error

"The beta version of the NetManage InterNet Controls has expired"  
whenever I try to  
use the NetManage controls from Delphi 2.

### Answer:

The controls inadvertantly contained a timebomb. Download  
[www.borland.com/devsupport/delphi/download\\_files/nmpatch.zip](http://www.borland.com/devsupport/delphi/download_files/nmpatch.zip)  
to resolve the problem.

4/1/99 11:09:26 AM

Last Modified: 01-SEP-99

## Article #19382: Oracle fields of type MLSLABEL are not supported

Question and Answer Database

FAQ4382B.txt - Oracle fields of type MLSLABEL are not supported

Category :Database (Oracle)  
Platform :All Windows  
Product :All, All, All,

Question:

How come I can't see data in fields of type MLSLABEL on my Oracle database?

Answer:

Fields of type MLSLABEL are unsupported.

3/26/99 10:41:18 AM

Last Modified: 01-SEP-99



## Article #19384: BatchMove of Time field from Paradox to Oracle results in timestamp in the year 3000.

Question and Answer Database

FAQ4384B.txt - BatchMove of Time field from Paradox to Oracle results in timestamp in the year 3000.

Category :Database (Oracle)

Platform :All Windows

Product :All-CBuilder, BC++4.5x, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How come when I batch move my Paradox table to my Oracle table I have timestamp data in the year 3000.

Answer:

This can occur when you batchmove time fields.

Oracle does not have an exact match, so a timestamp field is created. Oracle has no Time field, so an arbitrary date is put into the DATE field with the original time.

3/26/99 10:30:37 AM

Last Modified: 01-SEP-99

## Article #19387: How can I generate an error for SELECT statements that divide by zero?

Question and Answer Database

FAQ4387B.txt - How can I generate an error for SELECT statements that divide by zero?

Category :BDE Features  
Platform :All Windows  
Product :All, All, All,

Question:

How can I generate an error for SELECT statements that divide by zero?

Answer:

How and whether you can do this depends on the database. SQL Links will raise errors an error if the client raises an error that is above informational. Informational errors (such as seen with Sybase) cannot be raised.

3/26/99 10:32:43 AM

Last Modified: 01-SEP-99

## Article #19389: What is a good BDE Alternative?

Question and Answer Database

FAQ4389C.txt :What is a good BDE Alternative?

Category :Database Issues

Platform :All-32Bit

Product :All-CBuilder,

Question:

I am trying to find out what the advantages and disadvantages of using the BDE are? Are there BDE alternatives and what advantages and disadvantages do they have?

Answer:

Check out Kyle Cordes's Software Site. He discusses in depth BDE and MIDAS alternatives:

<http://www.kylecordes.com/>

2/22/1999 1:47:00 PM

Last Modified: 01-SEP-99

## Article #19395: Delphi is not brought to the front when breakpoints are hit in Windows 98

Question and Answer Database

FAQ4395D.txt :Delphi is not brought to the front when breakpoints are hit in Windows 98

Category :IDE

Platform :Windows 98

Product :All32Bit,

Question:

In Windows 98, why is it that when I hit a breakpoint or an Exception occurs,

Delphi isn't brought to the front of all other windows?

All it does is blink the Delphi icon in the Windows taskbar.

Answer:

This is a Windows 98 feature. You can turn it off by editing the WIN.INI file

under the section header entitled "[Compatibility95]"

```
[Compatibility95]
```

```
(Add the line below)
```

```
DELPHI32=0X00000002
```

3/10/99 10:30:11 AM

Last Modified: 01-SEP-99

## Article #19396: GPF in module MACXW4.DRV

Question and Answer Database

FAQ4396D.txt :GPF in module MACXW4.DRV  
Category :Install Issues  
Platform :Windows 95/98  
Product :All32Bit,

### Question:

I just installed Delphi and when I access the Environment Options or open up one of my projects, I get an error message that 'Delphi32 caused a GPF in module MACXW4.DRV at?' and then Delphi shuts down. How do I fix this?

### Answer:

This is more than likely a video driver related problem. Downloading the latest video drivers for your video card should solve the problem. You can also try turning hardware acceleration off. To turn off hardware acceleration: Right click on the desktop, select properties from the popup menu. Click on the "Settings" tab and then click on the "Advanced" button and then select the "Performance" tab. The final step is to turn the accelerator slider down to none and reboot for the changes to take effect.

3/10/99 10:36:13 AM

Last Modified: 01-SEP-99

## Article #19414: "Socket subsystem error" during install of InterBase

Question and Answer Database

FAQ4414C.txt : "Socket subsystem error" during install of InterBase  
Category : Database Issues  
Platform : All-32Bit  
Product : C++Builder3.x, C++Builder4.x,

### Question:

During installation of InterBase on Windows 95 or Windows NT, the configuration of the network service fails and reports the error message "Socket subsystem error."

### Answer:

This error occurs on systems that use the Novell Winsock TCP/IP driver.

InterBase 5.5 supports only the Microsoft Winsock TCP/IP driver. Some versions of the Novell Winsock TCP/IP driver are not compatible with the Microsoft driver. Installing or running InterBase on a platform with a Novell TCP/IP driver is not supported.

Remove the Novell TCP/IP protocol driver using Control Panel | Network | Protocols. Install the Microsoft TCP/IP protocol driver from the Windows NT and 95 installation media.

3/1/1999 8:10:03 AM

Last Modified: 01-SEP-99

## Article #19417: Updating your AS/400 OS release does not effect your Delphi/400 install

Question and Answer Database

FAQ4417D.txt - Updating your AS/400 OS release does not effect your Delphi/400 install

Category :AS400  
Platform :All Windows  
Product :DelphiAS400,

Question:

I'm updating my OS release on my AS/400. Will this effect my Delphi4/00 install?

Answer:

No. The key on your AS/400 is based on the model number and serial number of your AS/400. These do not change when install a new AS/400 OS release.

4/1/99 11:28:27 AM

Last Modified: 01-SEP-99

## Article #19428: Error: APPC Allocate Error Security Not Valid

Question and Answer Database

FAQ4428D.txt - Error: APPC Allocate Error Security Not Valid

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

How come I'm getting the error "APPC Allocate Error Security Not Valid" when I try to connect to my AS/400 using Delphi/400.

### Answer:

This error occurs when the port number to your AS/400 is incorrect. The correct port number for Delphi 3 is 5003. The correct port number for Delphi 4 is 4003.

Delphi 2/400 and Delphi 3/400 use the CO400 library on the AS/400. Delphi 4/400 uses the CO404 library. The way Delphi 4/400 decides to use the CO404 library is by reading Version=4 from the CO400 section of WINI.INI. If Version=4 is missing it connects to the CO400 library instead.

The Delphi 4/400 Deployment install does not update WIN.INI, so by default it attempts to connect to the CO400 library. So, for Delphi 4/400 Deployments if you enter a port number of 4003 and do not update WIN.INI, you will receive the error APPC Allocate Error Security Not Valid unless you either update WIN.INI, or have the CO400 library installed.

4/1/99 11:28:43 AM

Last Modified: 01-SEP-99



## Article #19430: Error: Nested Transactions Not supported

Question and Answer Database

FAQ4430B.txt - Error: Nested Transactions Not supported

Category :Database (InterBase)  
Platform :All-32Bit  
Product :All, All, All,

Question:

Q. How do I update a live view inside a transaction against Interbase? I always get the error "Nested Transactions not supported".

Answer:

To update an updateable view you must use an UPDATE SQL command. Otherwise, you cannot update an updateable view inside a transaction.

3/26/99 11:00:13 AM

Last Modified: 01-SEP-99

## Article #19432: DbiOpenTableList does not list Informix views

Question and Answer Database

FAQ4432B.txt - DbiOpenTableList does not list Informix views

Category :Database (Informix)  
Platform :All-32Bit  
Product :All, All, All,

Question:

How come Synonyms are not listed for the tablename property of a TTable, yet they are list under Synonmys in SQL Explorer?

Answer:

A TTable uses DbiOpenTableList to get its list of tables. For Informix, Synonyms are not implemented in this function. SQL Explorer uses SQL to gets Synonym list.

3/26/99 10:33:59 AM

Last Modified: 01-SEP-99

## Article #19436: Access CrossTab queries queries

Question and Answer Database

FAQ4436B.txt - Access CrossTab queries queries

Category :Database (MSACCESS)  
Platform :All-32Bit  
Product :All-CBuilder, BC++5.x, Delphi2.x, Delphi3.x,  
Delphi4.x, VdBase7.x,

Question:

Can the BDE MSACCESS driver execute crosstab queries?

Answer:

Yes. Here is an example created by using the Crosstab expert from Access against the Northwind sample database.

```
TRANSFORM Count([OrderID]) AS [The Value]
SELECT [ShipVia], Count([OrderID]) AS [Total Of OrderID]
FROM Orders
GROUP BY [ShipVia]
PIVOT [ShipPostalCode];
```

3/26/99 10:33:03 AM

Last Modified: 01-SEP-99

## Article #19438: Incorrect Application Icon

Question and Answer Database

FAQ4438D.txt - Incorrect Application Icon

Category :Miscellaneous  
Platform :All Windows  
Product :All32Bit,

### Question:

Windows shows the wrong icon for my application in the Windows Explorer and on Short Cuts. When my application is run however, the correct icon appears on the Task Bar and in the Alt-Tab list.

### Answer:

When Windows shows the default icon for an application in a Short Cut or in the Window's Explorer it uses the first icon resource that appears in the executable file. Icon resources that are linked into an application are ordered alphabetically. The resource name that Delphi uses for an application's icon is "MAINICON". If you link additional icon resources into your application you will need to make sure to name them so that they are ordered alphabetically after the resource name "MAINICON". The correct icon appears when the application is launched because in the window creation and registration process the icon resource name for the application is specified explicitly.

4/1/99 10:55:19 AM

Last Modified: 01-SEP-99

## Article #19439: Creating a custom TInplaceEdit for TDBGrid

Question and Answer Database

FAQ4439D.txt - Creating a custom TInplaceEdit for TDBGrid

Category :Database/VCL  
Platform :All Windows  
Product :All32Bit,

Question:

How can I customize the cell editor for a TDBGrid?

Answer:

You should create a custom TInPlaceEdit for use within Grids.

Following is an example component:

```
unit CustomEditorGrid;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  Grids, DBGrids;

type
  TMyInplaceEdit = class(TInplaceEdit)
  public
    procedure DblClick; override;
  end;

  TCustomEditorGrid = class(TDBGrid)
  private
    { Private declarations }
  protected
    { Protected declarations }
    function CreateEditor: TInplaceEdit; override;
  public
    { Public declarations }
  published
```

```
    { Published declarations }  
end;
```

```
procedure Register;
```

```
implementation
```

```
const
```

```
    SomeDefaultValue = 'Default Value';
```

```
procedure Register;
```

```
begin
```

```
    RegisterComponents('Custom', [TCustomEditorGrid]);
```

```
end;
```

```
{ TCustomEditorGrid }
```

```
function TCustomEditorGrid.CreateEditor: TInplaceEdit;
```

```
begin
```

```
    Result := TMyInplaceEdit.Create(Self);
```

```
end;
```

```
{ TMyInplaceEdit }
```

```
procedure TMyInplaceEdit.DblClick;
```

```
begin
```

```
    inherited DblClick;
```

```
    if (Text = '') then
```

```
        Text := SomeDefaultValue;
```

```
end;
```

```
end.
```

4/1/99 11:32:33 AM

Last Modified: 01-SEP-99

## Article #19440: Delphi 3/400 and Delphi 4/400 installed on the same machine

Question and Answer Database

FAQ4440D.txt - Delphi 3/400 and Delphi 4/400 installed on the same machine

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I have Delphi 3/400 and Delphi 4/400 installed on the same machine. I see that the last one installed controls what library I connect to the AS/400 regardless of whether I'm using Delphi 3 or Delphi 4. For example, if I install Delphi 3/400, then Delphi 4/400, then connect using Delphi 3, I connect to the Delphi 3 library (CO400). How do I make each connect to the correct library?

### Answer:

The version= line in the CO400 section of WIN.INI controls what library is connected to for the Delphi 4/400 version. If version=4, the CO404 library is used. If the line is not found, the CO400 library is used.

So, to resolve the problem, make sure Delphi 4/400 is installed last and control which library is connected to by editing WIN.INI.

4/1/99 4:13:28 PM

Last Modified: 01-SEP-99

## Article #19443: Generating auxiliary tables for Local SQL

Question and Answer Database

FAQ4443B.txt - Generating auxiliary tables for Local SQL

Category :BDE Features  
Platform :All Windows  
Product :All32Bit,

Question:

When I insert or delete records with Database Desktop using Local SQL a INSERTED or DELETED table is generated. How do I do the same with any BDE tool?

Answer:

Here's a Delpi example. It assumes Query2 has a SQL statement that deletes or inserts records from a Paradox or dBASE table.

```
Query2.Prepare;  
Check(DbiSetProp(hDbiObj(Query2.stmtHandle), stmtAUXTBLS, LongInt  
(True)));  
Query2.ExecSQL;
```

3/26/99 2:25:24 PM

Last Modified: 01-SEP-99



## Article #19444: Live Queries do not support TO\_-Date function

Question and Answer Database

FAQ4444D.txt - Live Queries do not support TO\_-Date function

Category :Database/VCL  
Platform :All Windows  
Product :All32Bit,

Question:

I have a query using Local SQL that includes the TO\_DATE function. Can I make this a live result set?

Answer:

No. Live Result sets do not support the TO\_DATE function,

4/1/99 11:32:53 AM

Last Modified: 01-SEP-99

## **Article #19447: curNATIVEHNDL, dbNATIVEHNDL, stmtNATIVEHNDL nor supported for Access**

Question and Answer Database

FAQ4447B.txt - curNATIVEHNDL, dbNATIVEHNDL, stmtNATIVEHNDL nor supported for Access

Category :Database (MSACCESS)  
Platform :All-32Bit  
Product :All32Bit,

Question:

How come I get an error of capability not supported when using dbiGetProp with curNATIVEHNDL, dbNATIVEHNDL, stmtNATIVEHNDL against Access?

Answer:

These properties are not supported in the access driver.

3/26/99 1:17:27 PM

Last Modified: 01-SEP-99

## Article #19452: 0 Dates and Informix

Question and Answer Database

FAQ4452B.txt - 0 Dates and Informix

Category :Database (Informix)  
Platform :All Windows  
Product :All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

### Question:

Informix documentation says that if you assign 0 to a date the result should be 12/31/1899, yet if I assign a 0 to an Informix date in Delphi or C++ Builder, the result is 12/30/1899. How come?

### Answer:

Delphi and C++ Builder assign a 0 based date to 12/30/1899. When they send the date to the server they do not send a zero but send a 12/30/1899 instead.

3/26/99 10:33:46 AM

Last Modified: 01-SEP-99

## Article #19455: Live query against ADT's are not supported

Question and Answer Database

FAQ4455B.txt - Live query against ADT's are not supported

Category :Database (Oracle)

Platform :All-32Bit

Product :All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How come when I attempt to perform a live query against an Oracle 8 table involving an ADT I get an error of Invalid field name or Invalid Column Name.

Answer:

Live queries against ADT's are not supported.

3/26/99 10:41:36 AM

Last Modified: 01-SEP-99

## **Article #19457: Filters are not supported against ADTs.**

Question and Answer Database

FAQ4457B.txt - Filters are supported against ADTs

Category :Database (Oracle)

Platform :All-32Bit

Product : BDE 5.0 and later

Question:

Why do I get an error of "Capability not supported" when I try to perform a filter using an ADT?

Answer:

ADT's are not supported in Filters.

Last Modified: 23-FEB-00

## Article #19459: BLOBs and CLOBs not supported in REF child table

Question and Answer Database

FAQ4459B.txt - BLOBs and CLOBs not supported in REF child table

Category :Database (Oracle)  
Platform :All-32Bit  
Product :C++Builder4.x, Delphi4.x,

Question:

How come I'm getting an error of Capability not supported when I I attempt to access a BLOB or CLOB in an Oracle 8 REF child table?

Answer:

This capability is not supported.

3/26/99 10:41:59 AM

Last Modified: 01-SEP-99

## Article #19461: IN Not supported for Live Queries against SQL Databases

Question and Answer Database

FAQ4461B.txt - IN Not supported for Live Queries against SQL Databases

Category :Database (Miscellaneous)

Platform :All-32Bit

Product :All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How come when I do a query such as:

```
Select * from kanton where kanton in (1,5,6)
```

I get an error of "Capability not supported"? This works in Delphi 1 and against Local Databases.

Answer:

This functionality is not implemented in the 32 bit BDE.

3/26/99 10:36:18 AM

Last Modified: 01-SEP-99

## Article #19462: Creating Floats in Access Table results in integers being created

Question and Answer Database

FAQ4462B.txt - Creating Floats in Access Table results in integers being created

Category :Database (ODBC)  
Platform :All-32Bit  
Product :All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How come when I'm using ODBC to connect to ODBC Access and I create a Table with TTable.CreateTable, fields which are type float result in being type integer?

Answer:

This is a problem specific to Microsoft's implementation of the Access ODBC driver. The problem does not occur with other ODBC datasources. The solution is to either use SQL to create the table or use the Native driver instead.

3/26/99 10:37:14 AM

Last Modified: 01-SEP-99



## Article #19464: Fieldtypes that get translated to BCD

Question and Answer Database

FAQ4464B.txt - Fieldtypes that get translated to BCD

Category :Database (Miscellaneous)  
Platform :All Windows  
Product :All-CBuilder, BC++4.5x, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, JBuilderC/S1.0,

Question:

What fieldtypes get translated to BCD when Enable BCD is true?

Answer:

When BCD is enabled, the following data types are translated to BDE logical data type fldBCD.

Interbase:

DECIMAL, NUMERIC

Oracle

NUMBER, FLOAT

Mssql

MONEY, SMALLMONEY, DECIMAL, NUMERIC

Sybase

MONEY, SMALLMONEY, DECIMAL, NUMERIC

Dbase

NUMBER, FLOAT

Paradox

BCD

3/26/99 10:36:30 AM

Last Modified: 01-SEP-99

## Article #19465: Default values not imported to Data Dictionary for Access

Question and Answer Database

FAQ4465B.txt - Default values not imported to Data Dictionary for Access

Category :Database (MSACCESS)  
Platform :All-32Bit  
Product :All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

I'm creating a data dictionary to my Access database. How come default values are not imported.

Answer:

This feature is not supported at this time.

3/26/99 11:03:22 AM

Last Modified: 01-SEP-99

## Article #19467: Can't Create Data Dictionaries for ODBC connections to Access and Oracle

Question and Answer Database

FAQ4467B.txt - Can't Create Data Dictionaries for ODBC connections to Access and Oracle

Category :Database (Miscellaneous)  
Platform :All-32Bit  
Product :C++Builder1.0, C++Builder3.x, C++Builder4.x,  
Delphi2.x, Delphi3.x, Delphi4.x,

### Question:

How come when I try to create a data dictionary for an ODBC connection to Oracle I get an error of Cannot Create Data Dictionary and doing the same I get an error of Invalid use of Keyword.

### Answer:

Creating Data Dictionaries on these data sources is not supported at this time. For Access, you can use the Native driver and for Oracle you can use the SQL Links driver instead.

3/26/99 10:36:41 AM

Last Modified: 01-SEP-99

## Article #19469: Oracle 8 RAW fields are limited to 255 bytes

Question and Answer Database

FAQ4469B.txt - Oracle 8 RAW fields are limited to 255 bytes

Category :Database (Oracle)  
Platform :All-32Bit  
Product :C++Builder4.x, Delphi4.x,

Question:

How come I can only enter 255 bytes in an Oracle 8 RAW field?

Answer:

This is a current limitation of the Oracle 8 driver.

3/26/99 10:42:10 AM

Last Modified: 01-SEP-99

## Article #19470: Delphi/400 deployment fails if BDE partial install is deployed

Question and Answer Database

FAQ4470D.txt - Delphi/400 deployment fails if BDE partial install is deployed

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Question:

I'm trying to install the Delphi/400 Deployment and I'm getting an error indication it cannot find the BDE. What is wrong?

### Answer:

The Deployment looks in the registry at the registry entry DLLPATH in HKEY\_LOCAL\_MACHINE\SOFTWARE\Borland\Database Engine to find the BDE. If this entry is missing or pointing to an invalid directory this problem will occur.

There are four possible conditions to cause this error.

1. The BDE has not been installed.
2. Only a partial install of the BDE has performed. Partial installs do not update DLLPath.
3. The registry has been edited to point to an incorrect location.
4. The files in directory where DLLPATH points have been moved or deleted.

4/1/99 11:30:14 AM

Last Modified: 01-SEP-99

## Article #19490: ODBC and DB2 don't do Prepare's

Question and Answer Database

FAQ4490B.txt - ODBC and DB2 don't do Prepare's

Category :Database (DB2)

Platform :32 bit Windows

Product : ALL 32 Bit BDE

### Question:

How come I see no performance improvement when preparing a query against an ODBC datasource or a DB2 datasource?

### Answer:

IN BDE versions prior to 5.10, The BDE connection to DB2 and ODBC do not support prepare. Performing a prepare causes a no-op to occur.

BDE 5.10 and later supports Prepare in a DB2 connection. However, executing two prepared statements in a row will generate a General SQL Error. To avoid getting the General SQL Error Unprepare after each SQL execution and then Prepare again before executing the statement again.

ODBC still does not support Prepare.

Last Modified: 01-MAR-00

## Article #19495: Row Level locking not supported in drivers other than Paradox, dBASE and FoxPro

Question and Answer Database

FAQ4495B.txt - Row Level locking not supported in drivers other than Paradox, dBASE and FoxPro

Category :Database (Miscellaneous)  
Platform :All Windows  
Product :All, All, All,

Question:

How can I get row level locking with a driver other than Paradox, dBASE or Fox (Access, MSSQL, Sybase, Interbase, Informix, DB2, ODBC, Oracle)?

Answer:

Row level locking is not supported for any driver other than Paradox, dBASE or FoxPro. Using Transactions may give some of the desired support.

3/26/99 10:34:51 AM

Last Modified: 01-SEP-99

## Article #19497: DbGetSysVersion does not report minor releases

Question and Answer Database

FAQ4497B.txt - DbGetSysVersion does not report minor releases

Category :Database (Miscellaneous)  
Platform :All-32Bit  
Product :All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

I have BDE 5.01. When I use DbGetSysVersion it reports I have version 500. How come it doesn't report 5.01? How do I get the current version of the BDE I'm using programatically?

Answer:

DbGetSysVersion is generally only updated for major releases. Therefore, version 5.01 didn't cause it to be modified. You find out the exact version by looking at the version information of IDAPI32.DLL.

3/26/99 10:36:04 AM

Last Modified: 01-SEP-99



## Article #19504: Special SQL syntax required to INSERT into BLOBs/CLOBs when parameters are >= 4K.

Question:

Why am I having trouble inserting/updating Oracle 8 BLOB and/or CLOB datatypes where the data >=4k?

Answer:

Oracle SQL requires a different SQL syntax when parameters to BLOBs and/or CLOBs have more than 4K worth of data. This means the following SQL statement:

```
INSERT INTO TESTBLOB (ICOMPONENTID, GIMAGE) VALUES (:CID, :GIMAGE)
```

will result in the error:

EDBEngineError with message 'General SQL error.  
ORA-22275: invalid LOB locator specified'

To resolve the problem for a blob do the following:

1. Use the following SQL statement:

```
insert into TESTBLOB (ICOMPONENTID, GIMAGE) values(:CID, empty_blob() ) returning  
GIMAGE into :GIMAGE
```

2. Make the parameter type for GIMAGE be ftOraBlob

To resolve the problem for a clob do the following:

1. Use the following SQL statement:

```
insert into TESTBLOB (ICOMPONENTID, GIMAGE) values(:CID, empty_clob() ) returning  
GIMAGE into :GIMAGE
```

2. Make the parameter type for GIMAGE be ftOraClob

## Article #19505: Sybase / MSSQL ARITHABORT ON and "Divide by zero occurred." error.

Question and Answer Database

FAQ: FAQ4505B - Sybase / MSSQL ARITHABORT ON and "Divide by zero occurred." error.

Category: Database (Sybase)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Why doesn't the Sybase SQL Link driver raise an error when a "Divide by zero" error occurs on the server?

Answer:

The MSSQL and Sybase SQL Links drivers do not raise server errors with severity levels 10 and below for the reasons explained below.

Setting ARITHABORT ON should change the severity level of arithmetic errors (see below).

With MSSQL Server "ARITHABORT ON" appears to change the severity level of the "Divide by zero" error from 10 to 16 whereas Sybase does not change its level from 10 (as expected the underlying Sybase server behavior does change though).

Both the MSSQL and Sybase documentation describe severity levels the same way (see below).

MSSQL descriptions for levels 0 through 16:

Severity Level 0 and 10: Status Information (Reported as Level 0)

These messages are not errors; they provide additional information after certain statements have executed.

[errors with levels 0 or 10 are not raised by the BDE]

Severity Levels 11 through 16

These messages indicate errors that can be corrected by the user.

Sybase:

-----  
set arithabort on  
select 1/0

Server Message: Number 3607, Severity 10  
Divide by zero occurred.  
(0 rows affected)

-----  
set arithabort off  
select 1/0

Server Message: Number 3621, Severity 10  
Command has been aborted.  
(0 rows affected)

MSSQL:

-----  
set arithabort off  
select 1/0

(null)

(1 row(s) affected)

Divide by zero occurred.

This error is documented as being  
Error: 3607  
Severity Level: 10  
Description: Divide by zero occurred.

-----  
set arithabort on  
select 1/0

Msg 8134, Level 16, State 1  
Divide by zero error encountered

7/13/99 10:54:39 AM

Last Modified: 01-SEP-99

## Article #19506: Informix Online and ROW LEVEL LOCKING (LOCK MODE ROW)

Question and Answer Database

FAQ: FAQ4506B - Informix Online and ROW LEVEL LOCKING (LOCK MODE ROW)

Category: Database (Informix)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Is it possible to do row level locking when working with an Informix server?

Answer:

All settings and options for row level locking are done at the table and database level.

The client application does not require any modifications or specific Informix server support to benefit from Informix Online row level locking.

Please see your Informix documentation for specific server and table settings.

Example Informix table create (please see your Informix documentation for specific information):

```
CREATE TABLE ROWLOCKTEST
(
FLD1 CHAR(5),
FLD2 FLOAT
)
LOCK MODE ROW
```

LOCK MODE PAGE is the default if not specified.

ALTER TABLE can be used to change the LOCK MODE for a table.

Example:

- 1) Start a transaction from the Informix client (TDatabase.starttransaction)
- 2) Modify a row and post (do not commit yet (TDatabase.commit))
- 3) From another client or connection attempt a change to the updated, uncommitted row. The second client should time out.
- 4) All rows not currently involved in a transaction can be modified.

If the Informix table in the above example had been created or altered with

LOCK MODE PAGE then rows adjacent (within the same locked page) to the row or rows currently part of a pending transaction will also be locked.

7/13/99 10:20:05 AM

Last Modified: 01-SEP-99

## Article #19507: Oracle database link, cursor, ORA-03106: fatal two-task communication protocol error

Question and Answer Database

FAQ: FAQ4507B - Oracle database link, cursor, ORA-03106: fatal two-task communication protocol error

Category: Database (Oracle)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Why am I receiving the Oracle error ORA-03106: "fatal two-task communication protocol error" when using an Oracle database link and Oracle cursors?

Answer:

```
SQL> print rcl
ERROR:
ORA-03106: fatal two-task communication protocol error
ORA-02063: preceding line from MARSLNK
```

As indicated in the SQL Plus output above this problem can be reproduced without using the BDE and/or BDE clients (Delphi, BCB, etc.).

The example SQL Plus output shows examples show the creation of an Oracle database link, synonym, stored procedure, package, package body, along with the execution and "printing" of the Oracle cursor data.

=====

```
create public database link MARSLNK using 'srvr_tcp.borland.world'
```

=====

EXECUTE PROCEDURE AND FETCH DATA FROM CURSORS VIA SQL\*PLUS:

```
variable rctl refcursor;
execute cursortest_4(:rctl);
```

```
print rctl
```

```
variable rcl refcursor;
execute cursortest_2(:rcl);
```

```
print rcl
```

=====

```
create or replace procedure cursortest_4 (rcl in out DMOD.rctl) AS
```

```
begin
```

```
open rcl for
```

```
SELECT CUSTOMER_N, NAME
```

```
FROM IDXCUSTM
```

```
ORDER BY UPPER(NAME);
```

```
end;
```

=====

```
create or replace procedure cursortest_2
```

```
(rcl in out cursorpack.rctl) AS
```

```
begin
```

```
open rcl for
```

```
SELECT *
```

```
FROM IDXCUSTM;
```



```
end;

=====

create or replace package cursorpack IS

type rctl is ref cursor return IDXCUSTOM%rowtype;

end;
```

```
=====

create or replace PACKAGE DMOD

IS

type rctl is ref cursor;

end;
```

```
=====
```

Log in to the Oracle server as the administrator to create the public database link:

SQL\*Plus: Release 3.3.4.0.0 - Production on Thu Oct 22 16:36:15 1998

Copyright (c) Oracle Corporation 1979, 1996. All rights reserved.

Connected to:  
Oracle7 Server Release 7.3.4.0.0 - Production  
With the distributed, replication and parallel query options  
PL/SQL Release 2.3.4.0.0 - Production

```
SQL> CREATE PUBLIC DATABASE LINK MARSLNK USING 'srvr_tcp.borland.world';
```

Database link created.

```
=====
```

Log into the Oracle server as user to create the synonym, packages

and procedures followed by execution of the procedures and printing/fetching the cursor rows:

SQL\*Plus: Release 3.3.4.0.0 - Production on Thu Oct 22 16:40:35 1998

Copyright (c) Oracle Corporation 1979, 1996. All rights reserved.

Connected to:

Oracle7 Server Release 7.3.4.0.0 - Production

With the distributed, replication and parallel query options

PL/SQL Release 2.3.4.0.0 - Production

```
SQL> CREATE SYNONYM IDXCUSTM FOR IDXCUST@MARSLNK;
```

Synonym created.

```
SQL> create or replace PACKAGE DMOD
```

```
2
```

```
2 IS
```

```
3
```

```
3 type rctl is ref cursor;
```

```
4
```

```
4 end;
```

```
5 /
```

Package created.

```
SQL> create or replace package cursorpack IS
```

```
2
```

```
2 type rctl is ref cursor return IDXCUSTM%rowtype;
```

```
3
```

```
3 end;
```

```
4 /
```

Package created.

```
SQL> create or replace procedure cursortest_2
```

```
2
```

```
2 (rcl in out cursorpack.rctl) AS
```

```
3
```

```
3 begin
```

```
4
```

```
4 open rcl for
```

```
5
5  SELECT *
6
6  FROM IDXCUSTM;
7
7  end;
8  /
```

Procedure created.

```
SQL> create or replace procedure cursortest_4 (rc1 in out DMOD.rctl)
AS
2
2  begin
3
3  open rc1 for
4
4  SELECT CUSTOMER_N, NAME
5
5  FROM IDXCUSTM
6
6  ORDER BY UPPER(NAME);
7
7  end;
8  /
```

Procedure created.

```
SQL> variable rctl refcursor;
SQL> execute cursortest_4(:rctl);
```

PL/SQL procedure successfully completed.

```
SQL> print rctl
ERROR:
ORA-03106: fatal two-task communication protocol error
ORA-02063: preceding line from MARSLNK
```

no rows selected

```
SQL> variable rcl refcursor;
SQL> execute cursortest_2(:rcl);
```

PL/SQL procedure successfully completed.

SQL> print rc1

ERROR:

ORA-03106: fatal two-task communication protocol error

ORA-02063: preceding line from MARSLNK

no rows selected

SQL>

7/13/99 10:49:35 AM

Last Modified: 01-SEP-99

## Article #19508: Raising MSSQL and Sybase errors with a severity <=10 do not get raised by SQL Links.

Question and Answer Database

FAQ: FAQ4508B - Raising MSSQL and Sybase errors with a severity <=10 do not get raised by SQL Links.

Category: Database (MSSQL)

Platform: All-32Bit

Product: All-32Bit,

Question:

Why don't server errors with a severity <=10 get raised by SQL Links?

Answer:

According to MS documentation severity Level 0 and 10 are for Status Information only (Reported as Level 0) These messages are not errors; they provide additional information after certain statements have executed.

Errors w/severity <=10 are also not returned by MS ODBC.

ADDITIONAL INFORMATION AND EXAMPLES

```
create procedure myerrorproc as
begin
RAISERROR ('my raiserror',11,127)
end
```

```
Using WISQL32:
exec myerrorproc
```

Returns the following:

```
Msg 50000, Level 11, State 127
my raiserror
```

```
create procedure myerrorproc2 as
begin
RAISERROR ('my raiserror',10,127)
end
```

```
Using WISQL32:
exec myerrorproc2
```

Returns the following:  
my raiserror

Info from MS SQL Server knowledgebase:

RAISERROR Statement (version 6.5):  
"Messages with severity levels 10 and under are not errors, but they do provide additional information."

Severity Levels 0 through 18

Severity Level 0 and 10: Status Information (Reported as Level 0) These messages are not errors; they provide additional information after certain statements have executed.

Severity Levels 11 through 16

These messages indicate errors that can be corrected by the user.

Severity Level 17: Insufficient Resources

These messages indicate that the statement has caused SQL Server to run out of resources (such as locks or disk space for the database) or to exceed some limit set by the SA.

These system limits include the number of databases that can be open at the same time and the number of connections allowed to SQL Server. Limits are stored in the sysconfigures table in the master database and can be changed with the sp\_configure system stored procedure. (For details about using sp\_configure, see the Microsoft SQL Server Transact-SQL Reference)

Level 17 messages that indicate you have run out of space can usually be corrected by the database owner. Other level 17 messages are best addressed by the SA.

Severity Level 18: Nonfatal Internal Error Detected  
These messages indicate that there is some type of internal software problem, but the statement finishes, and the connection to SQL Server is maintained. For example, a level 18 message occurs when SQL Server detects that a decision about the access path for a particular query has been made without a valid reason. The SA should be informed every time a level 18 message occurs.

7/26/99 10:10:23 AM

Last Modified: 01-SEP-99

## Article #19557: New Interbase SQL Links support in BDE 5.01 (SQLINT32.DLL v5.01.24)

Question and Answer Database

FAQ: FAQ4557B - New Interbase SQL Links support in BDE 5.01 (SQLINT32.DLL v5.01.24)

Category: Database (InterBase)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Does the Interbase SQL Links driver support Roles and additional transaction behavior?

Answer:

Please see the BDEREADME.TXT for additional information regarding BDE and SQL Links v5.01.

Please note that the new BDE ALIAS configuration parameters (COMMIT RETAIN, ROLE NAME, WAIT ON LOCKS) will show up only in newly created ALIASES.

InterBase Changes

-----  
Transaction Isolation Levels & InterBase  
-----

By adding 4096 to the setting of DRIVER FLAGS in the Borland Database Engine (BDE) configuration, you can specify that the InterBase SQL Links driver should use soft commits. Soft commits are a feature of InterBase that let the driver retain the cursor when committing changes. Soft commits improve performance on updates to large sets of data. When not used, the BDE must re-fetch all the records, even for a single record change. With soft commit the cursor is retained, and a re-fetch is not needed. Soft commits are never used in explicit transactions started by



BDE client applications. The soft commit property corresponds to the COMMIT RETAINING transaction option in the InterBase documentation.

DRIVER FLAGS	Isolation level	Commit type
-----	-----	-----
0	Read committed	hard commit
512	Repeatable read	hard commit
4096	Read committed	soft commit
4608	Repeatable read	soft commit

The 4096 driver flag bit only affects implicit transaction behavior. Use the COMMIT RETAINING property to control the default commit behavior of explicit transactions.

## Roles

-----  
InterBase allows users to log in by specifying a role. ROLE NAME is a new BDE configuration parameter that allows BDE clients to pass a role name. ROLE NAME can also be passed as an optional parameter when using a TDatabase component (Delphi and C++Builder).

This information supersedes that in the InterBase 5.5 Operations Guide, which indicates that the BDE has no property for specifying a role name.

## Transaction WAIT

-----  
Currently, the InterBase SQL Links driver uses NOWAIT when resolving lock conflicts and returns an error immediately if there is a resource conflict. Setting the WAIT ON LOCK configuration parameter allows SQL Links to use WAIT on InterBase transactions. This can also be done by setting the r/w database property dbWAITONLOCK to TRUE.

## Explicit transactions

-----  
The new BDE configuration parameter COMMIT RETAIN allows explicit transactions to use soft commit. The new database property dbCOMMITRETAIN also does this. Only explicit transactions are affected. The existing driver flags are still valid for explicit transactions.

COMMIT RETAINING = TRUE      calls isc\_commit\_retaining()

COMMIT RETAINING = FALSE    calls isc\_commit\_transaction()

7/13/99 10:21:26 AM

Last Modified: 01-SEP-99

## Article #19558: MSSQL SQL Links driver and the LANGDRIVER setting

Question and Answer Database

FAQ: FAQ4558B - MSSQL SQL Links driver and the LANGDRIVER setting

Category: Database (MSSQL)

Platform: All-32Bit

Product: All-32Bit,

Question:

Why might I be seeing duplicated rows being displayed in my application when using the MSSQL SQL Links driver w/MSSQL server version 6.5 or later?

Answer:

Please see the Delphi 4 README.TXT, BDEREADME.TXT, BDE32.HLP and SSQLLNK32.HLP for additional information regarding BDE language drivers and Borland SQL Links.

BDE/Microsoft SQL Server/Default Language Driver

-----  
If the LANGDRIVER parameter of a BDE alias for the Microsoft SQL Server driver is left blank, the Paradox ASCII driver will be used by default. Beginning with version 6.5, MS-SQL Server changed the default sort order from BINARY to NOCASE (52). As a result, an index created on CHAR or VARCHAR fields using the default language driver (Paradox ASCII) produces duplicate rows for an MS-SQL table if mixed case data is found in that field. Workaround: select an appropriate language driver in the LANGDRIVER parameter when configuring a BDE alias for MS-SQL (for example, US users would select the driver Borland ENG Latin-1). Note: this only applies to SQL Server version 6.5 and above, with default sort order.

7/26/99 10:11:02 AM



## Article #19559: BDE 12550 Invalid configuration parameter error

Question and Answer Database

FAQ: FAQ4559B - BDE 12550 Invalid configuration parameter error

Category: BDE

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

What might be causing the error BDE 12550 Invalid configuration parameter?

Answer:

12550 Invalid configuration parameter.

One possible cause for the error:

ODBC ALIASES.

When creating and/or using a BDE alias that is using an ODBC DSN make sure that the ALIAS parameter "ODBC DSN" is set to the DSN that you wish to use and that has been created using ODBC Manager. Available ODBC DSN's can be chosen from the parameter's drop-down edit using the BDE Administrator.

Another possible cause:

Using ISAPI app and ODBC against NT web server. ODBC DSN on NT machine should be set to "system" instead of "user".

7/13/99 10:10:56 AM

Last Modified: 01-SEP-99

## Article #19560: "lock timeout" error, TIMEOUT and MAX QUERY TIME parameters.

Question and Answer Database

FAQ: FAQ4560B - "lock timeout" error, TIMEOUT and MAX QUERY TIME parameters.

Category: Database (MSSQL)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

What does the lock timeout error mean and what can I do to change the timeout?

Answer:

"lock timeout" error, TIMEOUT and MAX QUERY TIME parameters.

By default, SQL statements sent to MS SQL and Sybase servers using DB-Lib are sent using the synchronous query submission API. Earlier versions of SQL Links used the asynchronous query submission API. To use the asynchronous query submission API, add 2048 to the current value of DRIVER FLAGS or set it to 2048 if it's blank.

To control the time-out for synchronous query submission use the MSSQL and Sybase driver TIMEOUT param.

To control the time-out for Asynchronous query submission use the MSSQL and Sybase driver and ALIAS MAX QUERY TIME param.

7/13/99 11:01:08 AM

Last Modified: 01-SEP-99

## Article #19561: Oracle user functions, ORA-06553 expression of of wrong type

Question and Answer Database

FAQ: FAQ4561B - Oracle user functions, ORA-06553 expression of of wrong type

Category: Database (Oracle)

Platform: All-32Bit

Product: All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

When using my user functions I receive the error:  
ORA-06553 expression of of wrong type.  
What might be causing this error?

Answer:

Oracle SQL Plus example:

```
SQL> SELECT MYFUNCTION('ASTRING') FROM MYTABLE
      2 /
SELECT MYFUNCTION('ASTRING') FROM MYTABLE
      *
```

ERROR at line 1:

ORA-06552: PL/SQL: Statement ignored

ORA-06553: PLS-382: expression is of wrong type

In this example the function has been defined with a return type of boolean which is not a valid field type.

In other words the return type cannot be returned as part of a sql result (see below).

USER FUNCTIONS FROM THE ORACLE 8 SQL REFERENCE:

You can write your own user functions in PL/SQL to provide functionality that is not available in SQL or SQL functions. User functions are used in a SQL statement anywhere SQL functions can be used; that is, wherever

expression can occur.

For example, user functions can be used in the following:

- the select list of a SELECT command
- the condition of a WHERE clause
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- the VALUES clause of an INSERT command
- the SET clause of an UPDATE command

For a complete description on the creation and use of user functions, see Oracle8 Application Developer's Guide.

Oracle user functions, ORA-06553 expression of of wrong type

7/13/99 10:50:43 AM

Last Modified: 01-SEP-99



## Article #19565: MSSQL 7 SQL Links (v5.01.23) feature support, MS DBLIB (NTWDBLIB.DLL)

Question and Answer Database

FAQ: FAQ4565B - MSSQL 7 SQL Links (v5.01.23) feature support, MS DBLIB (NTWDBLIB.DLL)

Category: Database (MSSQL)

Platform: All-32Bit

Product: All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Why aren't all of the new features and types supported in the new version of the MSSQL SQL Links driver?

Answer:

Please see the BDEREADME.TXT for additional information regarding BDE and SQL Links v5.01.

New MS-SQL Server 7 Driver

-----  
The SQL Links (not available with all editions of Inprise programming tools) driver for Microsoft SQL Server now supports version 7, in addition to support in the same driver for version 6.5.

SQL Links uses DBLIB (native API for MS-SQL Server). DBLIB only implements the MS-SQL Server 6.5 feature set. This means that new types in MS-SQL Server 7 like GUID, Unicode, and CHAR columns longer than 255 characters are not supported. Microsoft will not be adding support for these new types to DBLIB, so the SQL Links driver does not support these new MS-SQL Server 7 types.

Both Microsoft and Inprise strongly recommend that you use the client software appropriate for the version of MS-SQL Server used. That is, use version 6.x client software only with 6.x server software and version 7 client software only

with version 7 server software.

7/13/99 10:31:37 AM

Last Modified: 01-SEP-99

## Article #19566: SQL Explorer left pane cannot be resized.

Question and Answer Database

FAQ: FAQ4566B - SQL Explorer left pane cannot be resized.

Category: Delphi 4

Platform: All-32Bit

Product: All-32Bit,

Question:

When I launch DB/SQL Explorer the left pane does not display anything and I cannot resize it anymore. What can I do to fix this?

Answer:

Search your registry for all references to DbX.MainSplit and change the value to something greater than 0 (zero).

This is the registry entry for DB Explorer when it is launched from the Delphi 4 IDE:

```
[HKEY_CURRENT_USER\Software\Borland\Delphi\4.0\DbXForm]
"DbX.MainSplit"="300"
```

7/13/99 11:03:08 AM

Last Modified: 01-SEP-99

## Article #19602: Computer freezes at splash screen when Delphi is started.

Question and Answer Database

FAQ4602D.txt - Computer freezes at splash screen when Delphi is started.

Category :Install Issues  
Platform :Windows 95 & 98  
Product :Delphi4.x, Delphi5.x,

### Question:

When I start Delphi I see the splash screen, then my system freezes. How can I fix this?

### Answer:

This is most likely a compatibility conflict with the video driver. Borland tests its products against generic S3 drivers provided by Microsoft or S3 Inc. Any drivers compliant with this standard should be okay. Video drivers that do not fully comply with this standard may or may not function correctly with Delphi.

To verify that you have a video driver issue try changing the hardware acceleration in the display properties to 'None'. To do this go to Control Panel\DisplaySettings\Advanced Properties\Performance (a setting in the middle may work also). Restart the computer and try opening Delphi. If the problem is gone then you know it is a video driver issue.

### If the system still freezes:

In Windows 95 try starting in safe mode.

In Windows 98, in the SYSTEM.INI file under the [display] section, add the line 'BusThrottle = 1' (no quotes). Save the file, reboot and try running Delphi again.

Here are some more permanent solutions:

1. Install the Delphi 4 service pack 3. This offers a work-around by changing the way the component palette icon image list is loaded at startup. This fix is included in Delphi 5.

2. Start Delphi with the "-ns" switch: At the Run command line type "delphi32.exe" -ns. Make sure the -ns is outside the quotes (It may be necessary to type the entire path to the delphi32.exe file inside the quotes). This will bypass the splash screen.
3. Try going into the menu Component|Install Packages and remove some packages for components you don't need.
4. Install a new video driver.

4/1/99 3:23:58 PM

Last Modified: 10-OCT-00

## Article #19603: Computer freezes when Delphi is started under Windows 98

Question and Answer Database

FAQ4603D.txt - Computer freezes when Delphi is started under Windows 98

Category :Install Issues  
Platform :Windows 98  
Product :Delphi4.x,

### Question:

My system freezes when I start Delphi ? I don't even see the splash screen.

How can I fix this?

### Answer:

Delphi 4 will not run on Windows 98 if the display is set to 16 colors or less.

This may cause the system to freeze during installation also.

This can be fixed by changing the display color palette.

Go to Control Panel/Display/Settings/, change the color palette setting to

something greater than 16 color, such as 256 color, High Color (16-bit) or

True Color (32-bit). If you do not have these options you may need to

update your video driver.

4/1/99 3:23:40 PM

Last Modified: 10-OCT-00

## Article #19614: Safe Mode and VGA mode.

Question and Answer Database

FAQ: FAQ4614D - Safe Mode and VGA mode.

Category: Install Issues

Platform: Windows 95, Windows 98, Windows NT, Windows ME, Windows 2000, Windows XP

Product: All32Bit,

Question:

What is Safe Mode or VGA mode?

Answer:

Safe mode is the method for booting into a simple system configuration under Windows 95/98/ME/2000/XP.

VGA mode is the diagnostic mode under Windows NT.

When Windows boots into Safe mode or VGA mode, it uses basic default settings that will run the operating system with minimal functionality, meaning it loads a bare bones operating system. Hardware, like your CD-ROM drive, may not function in Safe mode or VGA mode because the drivers are not loaded. Because of this it may be necessary to copy the Delphi/C++Builder installation files to the hard drive before restarting in safe mode (See FAQ titled "Installing Delphi 4 from a Hard Drive").

10/24/02 2:00:21 PM

Last Modified: 24-OCT-02

## Article #19615: Starting in Safe Mode or VGA mode.

Question and Answer Database

FAQ: FAQ4615D - Starting in Safe Mode or VGA mode.

Category: Install Issues

Platform: All Windows

Product: All32Bit,

Question:

How do I start my computer in Safe Mode or VGA mode?

Answer:

SAFE MODE (Windows 95):

There are two ways:

1. Hold down the SHIFT key as you restart until the Windows screen comes up.
2. When you restart, watch the screen for the text "Starting Windows 95".

When you see this hit F8 and choose Safe Mode from the option menu which comes up.

VGA MODE (Windows NT):

Windows NT should offer a menu at start up where you can select VGA mode.

If you do not see this option it may not be installed. Look at Control Panel->

System-> StartupShut Down->System Startup for a list of the operating systems installed on you computer.

6/21/99 2:00:46 PM

Last Modified: 11-OCT-00



## Article #19616: Delphi 4 running slowly.

Question and Answer Database

FAQ: FAQ4616D - Delphi 4 running slowly.

Category: IDE

Platform: All Windows

Product: Delphi4.x,

Question:

Why is Delphi 4 running slower than Delphi 3?

Answer:

Make sure the Delphi 4 Update Pack 2 has been installed. The debugger hooks were slow starting and finishing which caused slowness. Update Pack 3 is also available but should not significantly alter the execution speed.

6/21/99 2:04:54 PM

Last Modified: 11-OCT-00

## Article #19618: Error "Can't find PVCS" when installing Delphi Professional.

Question and Answer Database

FAQ: FAQ4618D - Error "Can't find PVCS" when installing Delphi Professional.

Category: Install Issues

Platform: All Windows

Product: Delphi4.x,

Question:

What does the error "Can't find PVCS" when installing Delphi Professional version mean?

Answer:

This is an error in the install program and can be ignored. The version control software (PVCS) comes with Delphi Client/Server but not Delphi Professional. However PVCS can be installed into Delphi professional if you own the standalone version of PVCS (available from intersolve).

6/21/99 2:03:09 PM

Last Modified: 11-OCT-00

## Article #19652: Creating a lookup field in a TDBGrid for Builder

Question and Answer Database

FAQ: FAQ4652C - Creating a lookup field in a TDBGrid for Builder

Category: Database Issues

Platform: All Windows

Product: All-CBuilder,

Question:

How do I create a lookup field in a TDBGrid in C++ Builder

Answer:

1. Drop 2-TTable's, 1-TDataSource and 1-TDBGrid on a form.
2. Connect Table1 to DataSource1 to DBGrid1: In the object inspector, set these attributes.
  - a. DataSource1.DataSet = Table1
  - b. DBGrid1.DataSource = DataSource1
3. Setup Table1: In the object inspector for Table1, set these attributes.
  - a. DatabaseName = BCDEMOS
  - b. TableName = Customer
  - c. Active = True
4. Setup Table2: In the object inspector for Table2, set these attributes.
  - a. DatabaseName = BCDEMOS
  - b. TableName = Orders
  - c. Active = True
5. Add all of the fields for Table1 by bringing up the Fields Editor:
  - a. Double click on Table1
  - b. Right click on Fields Editor
  - c. Add Fields. Add all of them
6. Add a new field for Table1.
  - a. Right click on Fields Editor, and select New Field.

7. Specify the following parameters for the newly added field.
  - a. Name: CustOrder
  - b. Type: String
  - c. Size: 30
  - d. Select Lookup
  - e. Key Fields: CustNo - Field in Table1 to store value
  - f. DataSet: Table2 - Table lookup is being done on
  - g. LookUpKeys: CustNo - This Key gets copied to KeyField
  - h. Result Field: OrderNo - Value to display to the user in the drop down box

8. Run the application

8/4/99 11:54:12 AM

Last Modified: 01-SEP-99

## Article #19655: SET NOCOUNT ON causes "Couldn't perform the edit because another user changed the record" error

Question and Answer Database

FAQ: FAQ4655B - SET NOCOUNT ON causes "Couldn't perform the edit because another user changed the record" error

Category: Database (MSSQL)

Platform: All Windows

Product: BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

How come after I execute SET NOCOUNT ON on MSSQL server all updates result in the error

"Couldn't perform the edit because another user changed the record"?  
BDE error 10259.

Answer:

The BDE uses rows effected to determine if an UPDATE is successful. SET NOCOUNT ON tells MSSQL to return 0 for rows effected for all UPDATE statements and thereby causing all UPDATES to fail.

7/28/99 9:07:21 AM

Last Modified: 01-SEP-99

## Article #19657: FoxPro IDX files

Question and Answer Database

FAQ: FAQ4657B - FoxPro IDX files

Category: Database (FoxPro)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Does the BDE support FoxPro IDX files?

Answer:

The BDE supports FoxPro tables via its native driver up to version 2.6. IDX files were added in a FoxPro version after 2.6. Because of this the native driver does not support IDX files. If you have ODBC driver which can be configured to open IDX files whenever a table is opened then this would be a viable alternative.

7/13/99 10:18:30 AM

Last Modified: 01-SEP-99

## Article #19658: How do I get the version information for a file?

Question and Answer Database

FAQ: FAQ4658D - How do I get the version information for a file?

Category: Windows API

Platform: All-32Bit

Product: All32Bit,

Question:

How do I get the version information for a file?

Answer:

In order to get the file version information for a file you have to make several API calls. Here is an example that checks your version of shell32.dll and returns True if it is greater than or equal to 4.71.

```
function TForm1.CheckShell32Version: Boolean;

    procedure GetFileVersion(FileName: string; var Major1, Major2,
        Minor1, Minor2: Integer);
    { Helper function to get the actual file version information }
    var
        Info: Pointer;
        InfoSize: DWORD;
        FileInfo: PVSFixedFileInfo;
        FileInfoSize: DWORD;
        Tmp: DWORD;
    begin
        // Get the size of the FileVersionInformation
        InfoSize := GetFileVersionInfoSize(PChar(FileName), Tmp);
        // If InfoSize = 0, then the file may not exist, or
        // it may not have file version information in it.
        if InfoSize = 0 then
            raise Exception.Create('Can't get file version information for
            '
                + FileName);
        // Allocate memory for the file version information
```

```

GetMem(Info, InfoSize);
try
    // Get the information
    GetFileVersionInfo(PChar(FileName), 0, InfoSize, Info);
    // Query the information for the version
    VerQueryValue(Info, '\', Pointer(FileInfo), FileInfoSize);
    // Now fill in the version information
    Major1 := FileInfo.dwFileVersionMS shr 16;
    Major2 := FileInfo.dwFileVersionMS and $FFFF;
    Minor1 := FileInfo.dwFileVersionLS shr 16;
    Minor2 := FileInfo.dwFileVersionLS and $FFFF;
finally
    FreeMem(Info, FileInfoSize);
end;
end;

var
    tmpBuffer: PChar;
    Shell32Path: string;
    VersionMajor: Integer;
    VersionMinor: Integer;
    Blank: Integer;
begin
    tmpBuffer := AllocMem(MAX_PATH);
    // Get the shell32.dll path
    try
        GetSystemDirectory(tmpBuffer, MAX_PATH);
        Shell32Path := tmpBuffer + '\shell32.dll';
    finally
        FreeMem(tmpBuffer);
    end;

    // Check to see if it exists
    if FileExists(Shell32Path) then
        begin
            // Get the file version
            GetFileVersion(Shell32Path, VersionMajor, VersionMinor, Blank,
Blank);
            // Do something, such as require a certain version
            // (such as greater than 4.71)
            if (VersionMajor >= 4) and (VersionMinor >= 71) then
                Result := True
            else
                Result := False;
        end
    end
end

```



```
else
  Result := False;
end;
```

6/21/99 2:08:09 PM

Last Modified: 01-SEP-99

## Article #19660: Access and QuickReports

Question and Answer Database

FAQ: FAQ4660B - Access and QuickReports

Category: Database (MSACCESS)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

How come whenever I preview a QuickReport using the Access Native driver I crash?

Answer:

QuickReports uses threads when previewing reports. The Microsoft DAO which the BDE uses for its native Access driver is not thread safe. If your application requires previewing of QuickReports or any sort of threading support and you use an Access database you must use ODBC instead.

7/13/99 10:22:44 AM

Last Modified: 01-SEP-99

## Article #19661: How do I set the Wallpaper on Windows98/2000 when ActiveDesktop is turned on?

Question and Answer Database

FAQ: FAQ4661D - How do I set the Wallpaper on Windows98/2000 when ActiveDesktop is turned on?

Category: Miscellaneous

Platform: Windows98/2000

Product: All32Bit,

Question:

How do I set the Wallpaper on Windows98/2000 when ActiveDesktop is turned on?

Answer:

You may have noticed that using SystemParametersInfo to change the wallpaper when ActiveDesktop is turned on doesn't work. The reason is because you need to use the IActiveDesktop COM interface. Using SystemParametersInfo still works, but it doesn't update the wallpaper.

Note that the IActiveDesktop interface requires a Shell32.dll version  $\geq 4.71$ . The document titled "Get a File Version" demonstrates how to check the file version and uses the shell32.dll as an example.

Here is an example of using IActiveDesktop to work with the wallpaper. It assumes that you have 3 labels on a form and two buttons with the default names.

uses

```
ComObj, // For CreateComObject and Initialization/Finalization of COM
```

```
ShlObj; // For IActiveDesktop
```

```
{ The CLASS ID for ActiveDesktop is not defined in ShlObj, while the IID is so we define it here. }
```

```
const
```

```
CLSID_ActiveDesktop: TGUID = '{75048700-EF1F-11D0-9888-006097DEACF9}';
```

```
{ Demonstrate getting the Wallpaper }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
ActiveDesktop: IActiveDesktop;
```

```
CurrentWallpaper: string;
```

```
CurrentPattern: string;
```

```
WallpaperOptions: TWallpaperOpt;
```

```
tmpBuffer: PWideChar;
```

```
begin
```

```
    // Create the ActiveDesktop COM Object
```

```
    ActiveDesktop := CreateComObject(CLSID_ActiveDesktop) as  
IActiveDesktop;
```

```
    // We now need to allocate some memory to get the current Wallpaper.
```

```
    // However, tmpBuffer is a PWideChar which means 2 bytes make
```

```
    // up one Char. In order to compensate for the WideChar, we
```

```
    // allocate enough memory for MAX_PATH*2
```

```
tmpBuffer := AllocMem(MAX_PATH*2);
```

```
try
```

```
    ActiveDesktop.GetWallpaper(tmpBuffer, MAX_PATH*2, 0);
```

```
    CurrentWallpaper := tmpBuffer;
```

```
finally
```

```
    FreeMem(tmpBuffer);
```

```
end;
```

```
if CurrentWallpaper <> '' then
```

```
    Label1.Caption := 'Current Wallpaper: ' + CurrentWallpaper
```

```
else
```

```
    Label1.Caption := 'No Wallpaper set';
```

```
    // Now get the current Wallpaper options.
```

```
    // The second parameter is reserved and must be 0.
```

```
WallpaperOptions.dwSize := SizeOf(WallpaperOptions);
```

```
ActiveDesktop.GetWallpaperOptions(WallpaperOptions, 0);
```

```
case WallpaperOptions.dwStyle of
```

```
    WPSTYLE_CENTER: Label2.Caption := 'Centered';
```

```
    WPSTYLE_TILE: Label2.Caption := 'Tiled';
```

```
    WPSTYLE_STRETCH: Label2.Caption := 'Stretched';
```

```
    WPSTYLE_MAX: Label2.Caption := 'Maxed';
```

```
end;
```

```
    // Now get the desktop pattern.
```

```
    // The pattern is a string of decimals whose bit pattern
```

```
// represents a picture. Each decimal represents the on/off state
// of the 8 pixels in that row.
tmpBuffer := AllocMem(256);
try
  ActiveDesktop.GetPattern(tmpBuffer, 256, 0);
  CurrentPattern := tmpBuffer;
finally
  FreeMem(tmpBuffer);
end;
if CurrentPattern <> '' then
  Label3.Caption := CurrentPattern
else
  Label3.Caption := 'No Pattern set';
end;
```

```
{ Demonstrate setting the wallpaper }
```

```
procedure TForm1.Button2Click(Sender: TObject);
var
  ActiveDesktop: IActiveDesktop;
begin
  ActiveDesktop := CreateComObject(CLSID_ActiveDesktop)
    as IActiveDesktop;
  ActiveDesktop.SetWallpaper('c:\downloads\images\test.bmp', 0);
  ActiveDesktop.ApplyChanges(AD_APPLY_ALL or AD_APPLY_FORCE);
end;
```

6/21/99 2:05:55 PM

Last Modified: 01-SEP-99

## Article #19662: "Invalid Row ID" when migrating from Oracle 7 to Oracle 8

Question and Answer Database

FAQ: FAQ4662D - "Invalid Row ID" when migrating from Oracle 7 to Oracle 8

Category: Database(Oracle8)

Platform: All-32Bit

Product: All32Bit,

Question:

I just upgraded from Oracle 7 to Oracle 8 and now I occasionally get an "Invalid Row Id" error. I am using the new Oracle 8 Row Id's. How can I fix this error?

Answer:

This error may occur when posting to tables that contain LONG/Blob fields. More than likely, you are using the old SQL Link driver. Insure that you have the BDE 5.01 installed. In the BDE Administrator go to the Configuration tab and select Drivers->Native->Oracle. Make sure that the DLL32 is set to SQLORA8.DLL and the VENDOR INIT is set to OCI.DLL. Another possible solution may be to set the TTable or TQuery's UpdateMode property to upWhereKeyOnly.

6/21/99 1:58:59 PM

Last Modified: 01-SEP-99

## Article #19663: MSACCESS Tables with System Database on a CDROM

Question and Answer Database

FAQ: FAQ4663B - MSACCESS Tables with System Database on a CDROM

Category: Database (MSACCESS)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

How do I open a MSACCESS database that includes a system database on a CDROM?

Answer:

Exclusive must be set to true for the TDatabase.

7/13/99 10:23:56 AM

Last Modified: 01-SEP-99

## Article #19664: General SQL error # 2147221164

Question and Answer Database

FAQ: FAQ4664B - General SQL error # 2147221164

Category: Database (MSACCESS)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

How come when I try to connect to my Access database I'm getting a General SQL Error with an error from the DAO of # 2147221164?

Answer:

DAO350.DLL is not registered. Reinstalling the DAO will solve the problem in most cases.

7/13/99 10:25:06 AM

Last Modified: 01-SEP-99



## Article #19666: SQL Anywhere and Blobs>32K when RequestLive=False

Question and Answer Database

FAQ: FAQ4666B - SQL Anywhere and Blobs>32K when RequestLive=False

Category: Database (ODBC)

Platform: All-32Bit

Product: BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

If I RequestLive=False with a TQuery, can I get blob data from SQL Anywhere greater than 32K?

Answer:

Not at this time. SQL Anywhere is not a certified driver and operates differently than any driver Inprise has certified.

With other drivers, if Blob Size is set large enough, Data is Fetched successfully. With SQL Anywhere, even if Blob Size is large enough, the driver returns an error and fetching the data is unsuccessfull.

TQueries with RequestLive=True appear to work because the exact size of the buffer is requested. Whereas when requestlive=false, your blobsize setting is used. If buffersize is too large, SQL Anywhere returns an error, whereas other ODBC Datasources do not.

7/13/99 10:36:38 AM

Last Modified: 01-SEP-99

## Article #19667: Invalid Field Type when creating ODBC DataSource

Question and Answer Database

FAQ: FAQ4667B - Invalid Field Type when creating ODBC DataSource

Category: Database (ODBC)

Platform: All Windows

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

I'm using BatchMove to create a table in an ODBC DataSource. How come I'm getting the error "Invalid Field Type"?

Answer:

This error occurs for ODBC drivers where SQLGetTypeInfo(SQL\_DOUBLE) returns a result set. In this case, the result is float is attempted to be created on the DataSource even though the DataSource does not support floats. A classic example of this is Microsoft or Oracle's ODBC driver for Oracle.

There two possible solutions:

1. Use the Native driver (SQL Links) driver for the Database if one exists
2. Create the table yourself and use BatchAppend instead of BatchCopy.

7/13/99 10:35:27 AM

Last Modified: 01-SEP-99

## Article #19669: Oracle 7 faster than Oracle 8

Question and Answer Database

FAQ: FAQ4669B - Oracle 7 faster than Oracle 8

Category: Database (Oracle)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

I've noticed that fetching data against Oracle 7 is faster than Oracle 8. How come?

Answer:

If you are not using Object types the Oracle 8 driver can see improved performance turning off OBJECT MODE in BDE Administrator.

The Oracle 8 driver performs at the same level as Oracle 7 driver when the Oracle 7 driver has ROWSET SIZE set to 1. However, the Oracle 7 driver can have marked performance by increasing this parameter. The Oracle 8 driver does not support this parameter at this time and so therefore cannot reach the performance of the Oracle 7 driver at this time.

7/13/99 10:52:06 AM

Last Modified: 01-SEP-99

## Article #19674: Shared Memory Conflict error

Question and Answer Database

FAQ: FAQ4674B - Shared Memory Conflict error

Category: Database (Miscellaneous)

Platform: All Windows

Product: All-CBuilder, BC++4.5x, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

When I launch my second BDE application I'm getting the error Shared Memory Conflict (\$210D, 8461). How can I solve this?

Answer:

This error occurs when the memory location the BDE is attempting load at is being used another application. In most cases, this happens under Windows

NT. When this error occurs you need to changed the Shared Memory Location in BDE Administrator. While it is theoretically possible that many different

memory adresses exist that may resolve the problem, under Windows NT the address which we have found to always work in the past is 7000.

If 7000 does

not work try 7F00. For Windows 95/98, the best address to try is 1000.

7/13/99 10:32:58 AM

Last Modified: 01-SEP-99

## Article #19676: Unable to resolve records. Table name not found

Question and Answer Database

FAQ: FAQ4676D - Unable to resolve records. Table name not found

Category: Multi-tier

Platform: All-32Bit

Product: Delphi3.x, Delphi4.x,

Question:

I'm using ClientDatasets and my query has results from 2 tables. How come when I insert, delete or update records I get the error "Unable to resolve records. Table name not found"? I've run SQL Monitor and notice that the UPDATE statement just says WHERE w.

Answer:

To update multiple tables you must generate the SQL yourself. If you have Delphi 3, you can use an UpdateSQLProvider. If you have Delphi 4 you must create 3 TQueries (One each for INSERT, DELETE and UPDATE statements). From the TProviders BeforeUpdateRecord event determine which query to use by the updatekind parameter and then execute the appropriate query (populating the parameters along the way).

6/21/99 2:06:46 PM

Last Modified: 01-SEP-99

## Article #19694: "Invalid field type" database error with MSSQL server 7

### Question:

When I try to activate my Table/Query I get an error stating "invalid field type". What does this mean?

### Answer:

Microsoft no longer supports DB-LIB with the release of MS SQL Server 7. This means DB-LIB doesn't have support for the 4 new data types that MS SQL 7 offers: NTEXT, NCHAR, NVARCHAR and UNIQUEIDENTIFIER (a GUID). In order to successfully use new types offered by SQL Server 7 and above you'll need to use CT-LIB. With Delphi 5 we recommend you use ADO.

It is possible that you are using one of these datatypes inadvertently. For example when you use the replication feature in MS SQL 7 a new field called "rowguid" of type UNIQUEIDENTIFIER is automatically added. Since this will cause the invalid field type error in the BDE, replication cannot be used with the BDE.

Last Modified: 11-OCT-00

**Article #19698: Oracle wrapped stored procedure and the Oracle SQL Links driver.**

Question and Answer Database

FAQ: FAQ4698B - Oracle wrapped stored procedure and the Oracle SQL Links driver.

Category: Database (Oracle)

Platform: All-32Bit

Product: All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Can the Oracle SQL Links driver use Oracle procedures that have been "wrapped".

Answer:

Yes.

Oracle wrapped stored procedures behave just like "unwrapped" procedures.

The client application knows nothing about the fact that the procedure has been encoded by Oracle's WRAP23.EXE (or WRAP80.EXE).

Please refer to your Oracle documentation for information on the PL/SQL Wrapper utility.

7/13/99 10:53:10 AM

Last Modified: 01-SEP-99

## Article #19699: Database Desktop and the number of displayed digits after the decimal point.

Question and Answer Database

FAQ: FAQ4699B - Database Desktop and the number of displayed digits after the decimal point.

Category: BDE

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

Why does SQL Explorer display all digits to the right of the decimal point where Database Desktop does not?

Answer:

The Database Desktop uses your computer's Regional Settings | Number setting to determine the display format for numbers.

Run Control Panel | Regional Settings and click on the Number tab. Adjust the "No. of digits after decimal" to the desired number of digits that you wish displayed.

7/13/99 10:09:53 AM

Last Modified: 01-SEP-99



## Article #19700: Obtaining a list of available BDE language drivers

Question and Answer Database

FAQ: FAQ4700B - Obtaining a list of available BDE language drivers

Category: BDE

Platform: All-32Bit

Product: Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How can I retrieve a list of available BDE language drivers?

Answer:

The following Delphi procedure returns a formatted list of available BDE language drivers.

The procedure can be called as shown in the buttonclick method below.

Add BDE and DBTables to your unit's uses clause.

```
procedure GetLdList(Lines: TStrings);
var
  hCur: hDBICur;
  LD: LDDesc;
  cnt: integer;
begin
  // get a cursor to the in-mem table containing language
  // driver information...
  cnt:=0;
  check(dbiinit(nil));
  Check(DbiOpenLdList(hCur));
  try
    while (DbiGetNextRecord(hCur, dbiNOLOCK, @LD, nil) =
      DBIERR_NONE) do begin
      cnt:=cnt+1;
      Lines.Add(format('%4d %-6s%-10s %-10s%5s %-10s %-10s',
        [cnt, 'Name:', LD.szName, 'Code Page:', IntToStr(LD.iCodePage),
        'Description:', LD.szDesc]));
    end;
  end;
end;
```

```
    end;
finally
    Check(DbiCloseCursor(hCur));
    check(dbiexit);
end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
getldlist(memo1.lines);
end;

end.
```

7/13/99 10:12:07 AM

Last Modified: 01-SEP-99

## Article #19701: Unrecognized Database Format error when opening Access table

Question and Answer Database

FAQ: FAQ4701B - Unrecognized Database Format error when opening Access table

Category: Database (MSACCESS)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, Delphi2.x, Delphi3.x, VdBase7.x,

Question:

How come I'm getting a recognized Database Format error when opening an Access table with BDE?

Answer:

The table is probably an Access 2000 table. Another possibility is corrupted table.

The BDE uses DAO which has not been updated by MicroSoft since 1997. Access 2000 does not use DAO but uses ADO instead. Access 97 and BDE use the same DAO engine. Opening the table in Access 97 should result in the same error.

7/13/99 10:26:02 AM

Last Modified: 01-SEP-99

## Article #19727: \$2B05 "Permission denied." MS Terminal Server, application using the Borland Database Engine

Question and Answer Database

FAQ: FAQ4727B - \$2B05 "Permission denied." MS Terminal Server, application using the Borland Database Engine

Category: BDE

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

After logging into the Terminal Server the application is loaded which is followed by the BDE error:  
11013 \$2B05 "Permission denied."

Answer:

Terminal Server appears to create a temp folder for each user including appropriate rights. If the server is not brought down properly then the temp dir is not deleted and a subsequent user that logs in may be assigned an existing temp dir with someone else's rights.

A possible solution would be to delete all files and temp directories in the temp directory when the NT server boots up each time.

7/13/99 10:08:36 AM

Last Modified: 01-SEP-99

## Article #19729: Maximum number of DBPROCESSES already allocated

Question and Answer Database

FAQ: FAQ4729B - Maximum number of DBPROCESSES already allocated

Category: BDE Features

Platform: All-32Bit

Product: All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

What can I do to avoid the Maximum number of DBPROCESSES... error?

Answer:

Use the BDE MSSQL and Sybase driver MAX DBPROCESSES parameter to adjust the maximum number of DBPROCESSES allowed per client.

7/13/99 10:16:17 AM

Last Modified: 01-SEP-99

## Article #19731: NT Authentication/Integrated security

Question and Answer Database

FAQ: FAQ4731B - NT Authentication/Integrated security

Category: Database (MSSQL)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

If Integrated Security is enabled on my MSSQL sql server how do I make the BDE MSSQL SQL Links driver use Integrated Security?

Answer:

Leave user name and password blank in your application's TDatabase and make sure that the USER NAME in the BDE ALIAS and MSSQL driver settings is blank (used as a default when an ALIAS parameter is blank).

7/13/99 10:13:32 AM

Last Modified: 01-SEP-99

## Article #19741: No such method error on BOA\_init()

Question and Answer Database

FAQ: FAQ4741J - No such method error on BOA\_init()

Category: CORBA

Platform: All Windows

Product:

Question:

I can compile my distributed application without error, but at runtime I get a "no such method error" on org.omg.CORBA.orb.BOA\_Init()

Answer:

this error can be remedied by using an explicit cast on the orb, for example:

```
org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init  
( );
```

8/4/99 2:08:39 PM

Last Modified: 01-SEP-99

## Article #19742: ODBC Connections via Microsoft ODBC Text driver

Question and Answer Database

FAQ: FAQ4742B - ODBC Connections via Microsoft ODBC Text driver

Category: Database (ODBC)

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

Whenever I open a table via Microsoft's ODBC Text driver I get the error "Invalid table name". How come?

Answer:

The BDE calls SQLTables to determine whether a table name exists. The BDE does not expect table names returned by SQLTables to have an extension, but

Microsoft's Text driver does include extensions (Because unlike other drivers, it allows for tables with different extensions within the same database.)

and therefore the tablename cannot be verified.

There are two possible workarounds to the problem:

1. Use text files without extensions. To configure Microsoft's ODBC Text driver to do this run ODBC Administrator, configure your DSN, click Options, and \*. as an extension.
2. Use the BDE Native Text driver instead.

7/13/99 10:45:44 AM

Last Modified: 01-SEP-99



## Article #19744: Unable to use IDE tools to access and/or update tables in Informix database

Question and Answer Database

FAQ: FAQ4744J - Unable to use IDE tools to access and/or update tables in Informix database

Category: Database (Informix)

Platform: All Windows

Product:

Question:

I am unable to access my Informix database via most of the IDE database tools (e.g. JDBC Explorer, SQL Builder, and Browse Table functionality is lost)

Answer:

It appears that with older Informix JDBC drivers (problem reported against 1.22.JC1 specifically), there is a communication breakdown that causes an invalid schema name, "informix", to be returned on the table. As most of the IDE database tools rely on the schema name when internally querying the table, this results in poor behavior:

1) one will experience errors of the type: "the specified table (informix.  
) is not in the database" when attempting to open the table via JDBC Explorer's "Data" page 2) the table will appear to have no columns defined when inspecting via the "browse tables" dialog or the database tree in the left-hand pane of the JDBC Explorer Note that queries can still be performed successfully either via code or via the "Enter SQL" page of JDBC Explorer. The workaround for this problem is to update your JDBC driver to the latest available from the vendor. Visit the download area of Informix's website (<http://www.informix.com>), or go directly to <http://www.intraware.com:81/shop/product.html?PLNE=000148> for more information on the latest JDBC

drivers for Informix. 8/4/99 2:09:45 PM

Last Modified: 01-SEP-99

## Article #19773: External Exception EEFACE under Winnt 4.0

Question and Answer Database

FAQ: FAQ4773C - External Exception EEFACE under Winnt 4.0

Category: Database Issues

Platform: Windows NT

Product: All-CBuilder,

Question:

When I try to connect a MS Access table to my TTable component I get the External Exception "EEFACE". Is there any way to work around this?

Answer:

A solution is to modify your registry.

- 1) Run regedit
- 2) Open HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT  
\Currentversion\Image file execution options
- 3) Create a new key for your application. For example "Project1.exe"
- 4) Create a new value:  
Value Name: DisableHeapLookAside  
Data Type: REG\_SZ

You should now be able to connect to the table.

8/4/99 11:54:28 AM

Last Modified: 01-SEP-99

## Article #19778: Editing record generates BDE Error 10259 (\$2813)

Question and Answer Database

FAQ: FAQ4778B - Editing record generates BDE Error 10259 (\$2813)

Category: BDE Features

Platform: All Windows

Product: Delphi4.x,

Question:

How come when I edit a record with more than 15 digits after the decimal place I get BDE error 10259 (\$2813) "Couldn't perform the edit because another user changed the record."?

Answer:

The BDE has a limit of 15 decimal digits after the decimal place. What is happening to cause this error is that the BDE refetches the row the before you edit it to make sure it still exists. To do so, it uses the data it originally fetched. It only fetches up to 15 digits, so the data the BDE has and data on the server don't match. The server returns that the fetch returned no records which causes the BDE to generate its error.

7/28/99 10:30:24 AM

Last Modified: 01-SEP-99

## Article #19779: Character fields from Access ODBC connection are reported as being double their true size

Question and Answer Database

FAQ: FAQ4779B - Character fields from Access ODBC connection are reported as being double their true size

Category: Database (ODBC)

Platform: All-32Bit

Product: All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

How come when I use Microsoft's latest ODBC driver for Access the BDE returns the field size for character fields as being double their actual size?

Answer:

This problem occurs because the BDE calls SQLCOLUMNS and then looks at BUFFER\_LENGTH to determine the size of a column. With this driver BUFFER\_LENGTH is returned as double what the COLUMN\_SIZE is.

Here is what Microsoft used to document what BUFFER\_LENGTH should return:

The length in bytes of data transferred on an SQLGetData, SQLFetch, or SQLFetchScroll operation if SQL\_C\_DEFAULT is specified. For numeric data, this size may be different than the size of the data stored on the data source.

This value is the same as the COLUMN\_SIZE column for character or binary data.

Here is what Microsoft now documents what BUFFER\_LENGTH should return:

The length in bytes of data transferred on an SQLGetData, SQLFetch,

or  
SQLFetchScroll operation if SQL\_C\_DEFAULT is specified. For numeric data, this size may be different than the size of the data stored on the data source.  
This value might be different than COLUMN\_SIZE column for character data.

Inprise has verified this problem to exist in versions 4.00.3711.08 and 4.00.4202.00 of the Access ODBC driver. Inprise has contacted Microsoft and they have confirmed this new change in the ODBC specification. Inprise is currently evaluating how to resolve this issue in a future release that does not effect other ODBC drivers adversely.

Errors that may be caused by problem are "Record/Key Deleted" and "Size Mismatch".

The workarounds to this problem are:

1. Use an older version of the ODBC driver.
2. Use the BDE Access Native driver.
3. Use ADO with Delphi 5.

7/28/99 10:46:47 AM

Last Modified: 01-OCT-99

## Article #19781: BatchMove from TEXT doesn't create Time or TimeStamp fields.

Question and Answer Database

FAQ: FAQ4781B - BatchMove from TEXT doesn't create Time or TimeStamp fields.

Category: BDE Features

Platform: All Windows

Product: All-CBuilder, BC++4.5x, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, dBaseDOS, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

How when I use MatchMode BatCopy of BatchMove and the source is a text file with time or timestamp data I always get Alpha data in the destination table?

Answer:

The text driver doesn't support time or timestamp data types. You must alter the table once batchmove is complete and change the field type yourself.

7/28/99 11:20:50 AM

Last Modified: 01-SEP-99

## Article #19784: Oracle MSLABEL fields

Question and Answer Database

FAQ: FAQ4784B - Oracle MSLABEL fields

Category: Database (Oracle)

Platform: All-32Bit

Product: All-CBuilder, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x,

Question:

Does the BDE support Oracle MSLABEL fields?

Answer:

In most cases it is not supported. The BDE returns it as string field with size of zero. The VCL ignores string fields that have a size of zero.

7/28/99 11:23:16 AM

Last Modified: 01-SEP-99



## Article #19785: Dates with single digit years operate differently with a leading zero

Question and Answer Database

FAQ: FAQ4785B - Dates with single digit years operate differently with a leading zero

Category: BDE Features

Platform: All-32Bit

Product: All-CBuilder, BC++5.x, C++Builder1.0, C++Builder3.x, C++Builder4.x, Delphi2.x, Delphi3.x, Delphi4.x, VdBase7.x,

Question:

If I have a Paradox alpha field in which I restructure it and change the field to a date or datetime field, how come 1/1/1 results in 1/1/0001 whereas the date 1/1/01 results in the date 1/1/2001?

Answer:

The BDE treats 2 digit years specially. 2 digits years of 50-99 are in 20th Century (1900's). 2 digits years of 0-49 are in the 21st Century (2000's).

7/28/99 11:19:29 AM

Last Modified: 01-SEP-99

## Article #20073: How to avoid "Random" Exceptions being raised when using TJPEGImage or the JPEG unit.

### How to avoid "Random" Exceptions being raised when using TJPEGImage or the JPEG unit.

#### Question:

Sometimes when I try to load a JPEG into a TImage, I get an exception message even though I wrap the LoadFromFile call in a try...except. How can I avoid such exceptions, and why do they happen?

#### Answer:

When loading a JPEG from a file, sometimes it will load fine, but contain a corrupt JPEG image. When an attempt is made to draw the image, an exception will be raised. Quite often, the programmer may try to suppress errors with a procedure such as:

```
procedure TForm1.btnOpenDefaultClick(Sender: TObject);
begin
    if OpenPictureDialog1.Execute then
        try
            Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
        except end;
    end;
```

But, unexpectedly, JPEG's sometimes display an exception message and it appears that the creator of the JPEG unit hard-coded a ShowMessage to display the exception. The problem is that the LoadFromFile procedure executes fine, and the actual drawing of the JPEG to the TImage is what is causing the problem. The drawing of the image is implicitly called after the image has loaded from the file.

The way around this problem is to explicitly control when any possible errors happen. You can capture any errors that happen in the loading of the image, and/or in the drawing of the image.

Below is an example of how to do this. The project consists of a unit (called MainFrm.pas). It has a TImage and an OpenPictureDialog placed on the form. Two buttons are also placed on it, one named btnOpenCatch and another called btnOpenDefault. The btnOpenCatchClick procedure demonstrates how each exception can be caught and displayed the the user. Alternatively, the exception could be "eaten" with an empty except section. The btnOpenDefaultClick displays the default way of opening an image and trying to eat an exception, which may still display a JPEG error. Also notice the addition of the JPEG unit to the uses clause.

To create a corrupt JPEG that displays this error, open a JPEG image in a hex editor and change the first few bytes or the last few bytes.

```
unit MainFrm;
```

```
interface
```

## uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, ExtDlgs, ExtCtrls, StdCtrls, JPEG;

## type

```
TForm1 = class(TForm)
  btnOpenCatch: TButton;
  Image1: TImage;
  OpenPictureDialog1: TOpenPictureDialog;
  btnOpenDefault: TButton;
  procedure btnOpenCatchClick(Sender: TObject);
  procedure btnOpenDefaultClick(Sender: TObject);
```

## private

```
  { Private declarations }
```

## public

```
  { Public declarations }
```

```
end;
```

## var

```
Form1: TForm1;
```

## implementation

```
{$R *.DFM}
```

```
procedure TForm1.btnOpenCatchClick(Sender: TObject);
```

## var

```
  Picture: TPicture;
  Bitmap: TBitmap;
```

## begin

```
  if OpenPictureDialog1.Execute then
```

## begin

```
  Picture := TPicture.Create;
```

## try

```
  // Try to load the picture from a file.
  // I'm wrapping it in a try..except to capture
  // any exceptions that happen in loading the image from
  // a stream.
```

## try

```
  Picture.LoadFromFile(OpenPictureDialog1.FileName);
```

## except

```
  on E: EInvalidGraphic do
```

## begin

```
  ShowMessage('Capturing Load Error: ' + E.Message);
```

```
  Exit; // Exit the procedure
```

```
  end;
```

```
end;
```

```

Bitmap := TBitmap.Create;
try
  try
    // Assign the Graphic to the Bitmap; any drawing errors/exceptions
    // will happen here, and will be captured and shown to the user
    Bitmap.Assign(Picture.Graphic);
  except
    // Eat invalid graphic messages (which may be raised
    // when a jpeg is being drawn)
    on E: EInvalidGraphic do
      begin
        ShowMessage('Capturing Assign (Drawing) Error: ' + E.Message);
        Exit; // Exit the procedure
      end;
    end;
  // Now draw the picture on the screen's TImage
  Image1.Picture.Bitmap.Assign(Bitmap);
finally
  Bitmap.Free;
end;
finally
  Picture.Free;
end;
end; // OpenPictureDialog1.Execute
end;

procedure TForm1.btnOpenDefaultClick(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    try
      Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
    except end;
  end;
end.

```

Another thing to notice is that the OpenPictureDialog does not contain this fix. It will display some JPEG error messages when you click on a corrupt JPEG and it attempts to display a preview image.

Last Modified: 03-JAN-00

## Article #20090: System Resource Capacities in Windows 95

Running out of system resources in Windows 95 can give the Delphi exception "EWin32Error" with further details such as "in VCL40.DPL". EWin32Error is raised when Windows returns an error code. Following are some Windows 95 limits. The logical pens and brushes limit is probably the most significant for Delphi users since it will limit the number of visual components an application can create simultaneously. These were found on the Microsoft web page at: <http://msdn.microsoft.com/library/winresource/dnwin95/s70b0.htm>

```
*****
Windows 95 System Limits
*****
Windows Menu handles      32K
Timers                    Unlimited
COM and LPT ports        Unlimited
Items per listbox        32K
Data per listbox         Unlimited
Data per edit control    Unlimited
Regions                  Unlimited
Physical pens and brushes Unlimited
Logical pens and brushes All in 64K segment
Logical fonts            750 - 800
Installed fonts          1000
Device contexts          16K
*****
```

Last Modified: 16-DEC-00

## Article #20100: "Expected type..." error using ADO to access numbers in Oracle Table.

### Question:

Why am I getting "Expected type was..." error using ADO to access numbers in an Oracle table?

### Answer:

The problem lies in the fact that Integer fields in Oracle are being represented as TFloatfield in Delphi. The root of this issue is the translation done by the OLE DB PROVIDER. NUMBER and INTEGER fields in Oracle are being translated as DBTYPE\_NUMERIC by the OLE DB PROVIDER. This can be verified by looking at the MDAC SDK documentation, which is available off the Microsoft web page. If the provider never returns an Integer OLE DB Data Type, you'll never get a TField that is a TIntegerField type.

Last Modified: 11-OCT-00

## **Article #20158: Hidden or missing IDE**

Occasionally in Delphi 5 the IDE will 'disappear'. This is to say you will launch Delphi, and most or all of your IDE will not be visible. Delphi is clearly running and working, as you can still use your hot keys and run and compile a program, you just can't see such useful items such as the object inspector or your component tool bar.

This is caused by Delphi saving bogus information in its desktop preference files, or '.dsk' files. Delphi saves information about your environment options and settings in these files. In order to see your IDE again, all you need to do is delete these offending files. This will in no way harm Delphi or your project, as Delphi will dynamically generate these files as it needs them. You will however lose any environment settings that you had set previously.

Last Modified: 30-NOV-99

## Article #20179: How do I execute a stored procedure with TIBSQL?

### QUESTION:

How do I execute a stored procedure with TIBSQL?

### ANSWER:

To take advantage of the light weight TIBSQL component for executing stored procedures, execute the procedure with the SQL statement 'execute procedure procedure\_name'. The results are passed back through a TIBXSQLEDA object. A TIBXSQLEDA is an object which encapsulates an InterBase API data type and contains all of the return parameters for the stored procedure. TIBSQL.Current returns the current TIBXSQLEDA object associated with the TIBSQL.

The following code example executes a stored procedure, new\_key, that does not take any parameters but does return a value. The 'new\_key' procedure is defined as follows:

```
create procedure new_key returns (newkey integer)
as
begin
    newkey = gen_id(vote_pui_gen, 1);
end
```

For more information on creating stored procedures refer to the InterBase "Data Definition Guide", and "Programmer's Guide".

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    IBSQL1.SQL.Clear;
    IBSQL1.SQL.Add('execute procedure new_key');
    IBSQL1.ExecQuery;
    Form1.Caption := IBSQL1.Current.Vars[0].AsString;
end;
```

If the procedure returns multiple parameters, you can iterate through the Vars array to extract all of the parameters.

Last Modified: 18-JAN-00



## **Article #20180: Garbage added to end of Memo field with ODBC connection**

When the BDE inserts blob data into a table via an ODBC connection it indicates the length of the blob via a size parameter. Some ODBC drivers ignore this parameter and attempt to determine the size by looking for a NULL character instead. ODBC drivers which are reported to operate this way are Advantage and PostgreSQL. This may result in garbage appearing at the end of the blob. To resolve the problem obtain an updated ODBC driver from the ODBC vendor which determines the size via the parameter, not by looking for a NULL.

Last Modified: 01-DEC-99

## Article #20195: "Cannot load an IDAPI service library" error with a message of "SQLORA8.DLL"

### Question:

I recently moved from Oracle 7 to 8, and when I try to open my alias up in the BDE administrator, I get a "Cannot load an IDAPI service library" error with a message of "SQLORA8.DLL". How do I fix this?

### Answer:

There are several things that can cause this problem. The first thing to make sure of is that you have the Oracle 8 client installed. If you can use the Oracle 8 client tools (such as SQL Plus) to successfully connect, then you may have a path issue. The easiest way to test this is to copy the SQL Plus EXE from the Oracle bin directory to your desktop and see if you can run it. The SQL Plus EXE is typically in your c:\ORANT\BIN or c:\ORA95\BIN directory (depending on the OS, and assuming it is installed on the C drive). If you try to run it from the desktop and you get an error about it not finding a DLL, then you have a path issue.

The way to fix a path issue is to right click on the file c:\autoexec.bat and select Edit. Look for the line that starts with something like:

```
set PATH=C:\....
```

Add REM to the start of that line to comment it out, and add a new line that looks like this:

For Windows NT:

```
set PATH=C:\ORANT\BIN;
```

For Windows 95/98:

```
set PATH=C:\ORA95\BIN;
```

After that, reboot your machine and it should work fine.

Last Modified: 01-DEC-99

**Article #20234: TeamSource Error(4): Invalid extension, in file '...\Archives\root.tsdata'**

**QUESTION:**

Why am I getting Error(4): Invalid extension, in file 'C:\Teamsource\Archives\root.tsdata' when I try to open the Remote screen in TeamSource?

**ANSWER:**

This is caused by a bug in TeamSource. The documentation for Productions states that a Production can be Defined with Source = \*.RC and Target = \*.RES, but after these values are entered TeamSource adds a period in front of them so the result is '.\*.RC' and '.\*.RES'. These values are accepted but unfortunately cause the error when the project is reopened in the future.

**WORKAROUND:**

Do not use the asterisk notation, just enter the extension. If you've already entered this notation you can recover by deleting the 'root.tsdata.z' file, but you will then need to re-do check ins.

Last Modified: 11-OCT-00

## Article #20243: Conflict between IntelliMouse driver v2.2d and TDBLookupComboBox

### QUESTION:

Why is my application freezing when I pass the IntelliMouse pointer over a TDBLookupComboBox?

### ANSWER:

There is a conflict between the IntelliMouse driver v2.2d and TDBLookupComboBox. The error is reproduced by running an application which has a TDBLookupComboBox on a machine with IntelliMouse v2.2d. Click on the arrow in the box to show the dropdown list. Now move the mouse wheel, and the application freezes. The application must be killed to recover.

This issue does not occur with a TComboBox because TDBLookupComboBox is not a TComboBox descendent.

### WORKAROUND

This issue was fixed in Delphi 5. Here is a workaround for Delphi 4 which requires adding two lines to DBCTRLS.PAS. The .DCU file will need to be rebuilt after adding these lines:

```
// This line takes care of the AV. ListLink.DataSet could be  
// nil after the WM_MOUSEWHEEL wasn't handled and the Intellimouse  
// driver decided to send a WM_VSCROLL
```

```
procedure TDBLookupListBox.WMVScroll(var Message: TWMVScroll);  
begin
```

```
    SearchText := '';  
(* ** * ADD THIS LINE * ** *)  
    if ListLink.DataSet = nil then Exit;
```

```
    with Message, ListLink.DataSet do  
end;
```

```
// This takes care of the "hang". The control is getting a  
// WM_KILLFOCUS immediately after the *second* WM_MOUSEWHEEL.  
// Focus is retained by the drop-down list, making it appear  
// that the application had stopped responding }
```

```
procedure TDBLookupComboBox.CloseUp(Accept: Boolean);
```

```
var  
    Msg: TMsg;  
    ListValue: Variant;
```

```
begin
  if FListVisible then
    begin
      if GetCapture <> 0 then SendMessage(GetCapture, WM_CANCELMODE, 0, 0);
      (**** ADD THIS LINE ****)
      SetFocus;

      ListValue := FDataList.KeyValue;
    ...
  end;
```

Last Modified: 28-DEC-00

**Article #20330: BDE Administrator not responding.**

**QUESTION:**

Why after loading BDE 5.0 on Windows '98 machine with SR2, does the BDE Administrator not respond? There is no error message, it just freezes. Delphi is running fine.

**ANSWER:**

This is due to a conflict with Microsoft's generic SVGA video driver. If you switch to the card manufacture's updated driver, make sure you have 256 colors. This should solve the problem. You may need to experiment with different drivers.

Last Modified: 17-DEC-99

**Article #20362: "Could not perform corba dispatch, no interface repository found" error.**

**QUESTION:**

Why am I getting the error "Could not perform corba dispatch, no interface repository found"?

**ANSWER:**

There was an issue between Visibroker 3.4 which comes with JBuilder 3, and Delphi 5 which resulted in the loss of some CORBA functionality in Delphi when connecting to a JBuilder 3 server.

A new version of Visibroker 3.3 for Delphi 5 Enterprise is now available which resolves this. The update is for Delphi 5 Enterprise only and is available for free download from our web site at:

<http://www.borland.com/visibroker/delphi/>

This release includes an IDL2PAS compiler. IDL2PAS is a tool which implements isiBroker's IDL to Pascal compiler. IDL2PAS takes one or more IDL files as input and generates the corresponding Pascal classes for the client side, including the client stubs. Visit the URL above for more details.

Last Modified: 11-OCT-00

## Article #20386: The integrated debugger and NT

### QUESTION:

My simple database application works on Windows 95 and 98, but gives me access violations on NT. I can run the files I create from the desktop in NT, but not from the IDE, why?

### ANSWER:

In order to solve this issue, you must go to Delphi's 'Tools | Debugging Options' on the main menu and locate the 'General' tab. On this page you will find a checkbox that reads 'Integrated Debugger'. You will want to make sure that this is turned off. After this Delphi should run normally.

Last Modified: 04-JAN-00



## **Article #20393: Parameters to Access Long Text fields limited to 255 characters**

A TQuery that updates an Access LongText field via a Parameter or a TUpdateSQL Component that does the same will fail if number of characters in the parameter is more than 255. If the connection is done via ODBC it will result in a BDE error of "General SQL Error" and then an error from the ODBC of "String data, right truncated (null)". If the connection is done via the BDE Native driver it will result in an error of "External Exception EEFACE".

The reason the ODBC driver error occurs is because the Access ODBC driver has a limit of 255 characters for parameters.

The reason the error occurs with the Native driver is the DAO limits parameters to 255 characters and the BDE uses the Microsoft DAO client to use Access tables.

The best workaround to this problem is to use ADO with Delphi 5.

Another partial solution is to use a TQuery with requestlive=true, or a TTable.

Last Modified: 04-JAN-00

## **Article #20394: Versions of Oracle Supported**

Here is a list Oracle clients that BDE 5.10, supports:

- Oracle 7.33
- Oracle 7.34
- Oracle 8.04
- Oracle 8.15

BDE 5.2.0.2 supports Oracle 8.16 as well.

Last Modified: 10-JUL-01

## **Article #20395: Versions of DB2 supported by BDE 5.10**

Below are the versions of DB2 client supported by BDE  
5.10

DB/2 version 2.x

DB/2 V5 (UDB)

Any later DB2 version is not supported at this time.

Last Modified: 21-DEC-99

## Article #20420: Parameter "ParamName" Not Found error when using an ADOQuery and Parameters

### Question:

When using an ADOQuery with Parameters, and trying to access the Parameters with: `ADOQuery.Parameters.ParamByName('ParamName').Value` I get a 'Parameter "ParamName" Not Found' error. How can I fix this?

### Answer:

You can easily fix this by creating the parameters before calling `ADOQuery.Parameters.ParamByName('ParamName').Value` by calling: `ADOQuery.Parameters.ParseSQL(ADOQuery.SQL.Text, True)`.

Last Modified: 03-JAN-00

## **Article #20421: Why are my Data Access components not showing?**

### **Why are my Data Access components not showing?**

**In the "Quick Start" booklet, the tutorial uses components on the data access tab; Why don't I have that tab?**

The "Standard" version of Delphi 5, does not come with the "BDE" or "Code explorer". Therefore, there will not be a Data Access tab available when following the tutorial in the "Quick Start" booklet. That tutorial was developed for the "Enterprise" and "Professional" versions only. However there is an appropriate tutorial available for all versions of Delphi here: <http://www.borland.com/techpubs/delphi/> also, the address is listed at the bottom of the README.txt file. The tutorial is towards the bottom, under the heading Tutorials, downloads.

Last Modified: 19-JAN-00

## Article #20433: Interpreting Cryptic Jpeg Error Number's

### Interpreting Cryptic Jpeg Error Number's

By Corbin Dunn

If you have used the TJpegImage class you may be familiar with the "JPEG error #xx" error that comes up on occasion. You may want to know exactly what went wrong, and how to get more information based on that number.

Luckily, the TJpegImage pascal source is included on the Delphi CD. Looking at the source, you will notice that it is a wrapper around some C code. There is one particular file called "jerror.h" which contains an enumeration of error strings. Based on the offset in this enum, you can find out what a given error number means.

For ease of use, I converted the "jerror.h" file to pascal in the unit JpegErrors.pas. To use this unit, simply reference the string array cJpegErrors, passing a given error number to find the error string, such as:

```
Label1.Caption := cJpegErrors[StrToInt(Edit1.Text)]
```

Note that the JpegErrors.pas unit was based on the Jpeg source that comes with Delphi 5. If the order of the error messages in "jerror.h" ever changes, the JpegErrors.pas will not be in sync with it and has to be updated.

[Download a sample program and the JpegErrors.pas unit](#)

Last Modified: 30-DEC-99

## **Article #20468: How to enable the cut and copy operations for the TWebBrowser component in Delphi 5.**

### **Why don't the cut and copy operations work for the TWebBrowser component in Delphi 5?**

If you are going to use the TWebBrowser and want to enable cut and copy, you will need to add these lines of code at the bottom of the your unit:

```
initialization  
    OleInitialize(nil);
```

```
finalization  
    OleUninitialize;
```

You will also need to add ActiveX to your uses clause.

Last Modified: 19-JAN-00

## **Article #20553: How do I Debug an In-Proc COM Server?**

How to do I Debug an In-Proc COM Server?

The steps to debug an in-proc COM server are on page 45-6 in the Delphi 5 Developer's Guide (or DG. PDF on Delphi 5 CD) titled "Testing a COM Object". However, there is one step missing in order to get this to work correctly. You also need to enable "Include Remote Debugger Symbols" in Project->Options->Linker (and be sure to do a Build All).

Last Modified: 21-JAN-00



## **Article #20559: General SQL Error when accessing a memo field in MS SQL 6.5.**

### **Question:**

When connecting to MS SQL Server 6.5, and trying to access long text data (a memo field) with something such as "FieldByName('FieldName')", I get a "General SQL Error."

### **Answer:**

This could be caused by not having the proper service packs installed on MS SQL Server. Installing Service Pack 4 has been known to fix this issue.

A "General SQL Error" is returned when the database server returns some message that the BDE does not know how to handle.

Last Modified: 23-FEB-00

## Article #20566: Master/Detail relationships with data-aware controls.

### Question:

Why are my Master/Detail relationships not showing the correct detail in the data-aware controls when I scroll through the master?

### Answer:

This is a known bug in Delphi 5 and will be fixed. It seems to only happen with queries. Using persistent fields will make it worse. Here is a good workaround which was posted on the borland newsgroup. It requires editing and recompiling the file 'dbctrls.pas' which comes with Delphi.

First find the file 'dbctrls.dcu' on your machine. If you did a default install it should be at 'C:\Program Files\Borland\Delphi5\Source\Vcl'. Rename this to something like 'dbctrls\_ORIGINAL.dcu' so you can get it back later if you want. Next start a new, blank application and open the file 'C:\Program Files\Borland\Delphi5\Source\Vcl\dbctrls.pas'. Add Dbctrls to the uses clause of the blank project. Find the definition for procedure TFieldDataLink.UpdateField; and change the code to the following. You may want to just comment out the code that is there in order to save it.

```
procedure TFieldDataLink.UpdateField;
var
  TempField : TField;
begin
  if Active and (FFieldName <> '') then
  begin
    if Assigned(FControl) then
      TempField := GetFieldProperty(DataSource.DataSet, FControl, FFieldName)
    else
      TempField := DataSource.DataSet.FieldByName(FFieldName);
    if TempField <> FField then
      SetField(TempField)
    else
      begin
        FField := NIL;
        SetField(TempField);
      end;
  end
  else
    SetField(nil);
end;
```

Finally, save and Build All Projects (Projects menu). If you get an error that 'dbctrls.dcu' cannot be found then you need to go into Tools|Environment Options|Library and add the path C:\Program Files\Borland\Delphi5\Source\Vcl (or whatever your actual path to the Vcl directory is) to the Library Path.

Now you should be able to compile and create a new 'dbctrls.dcu' file, and your data-aware controls should be

updating.

Last Modified: 11-OCT-00

## Article #20580: How SQL Links determines if an MSSQL table can refreshed.

Q. How does SQL Links determine if an MSSQL Table can be refreshed or not?

A. The following query is run when the table is opened:

```
select x.name, x.indid, x.status, INDEX_COL('owner.tablename', x.indid, c.colid) from sysindexes x,
syscolumns c where x.id = object_id ('owner.tablename') and indid between 1 and 254 and x.id = c.id
and c.colid <= x.keycnt order by x.indid, c.colid ASC
```

Note that owner and tablename are case sensitive.

Another query that is useful in working on this problem is listing all index for a table. The following query will do so:

```
SELECT b.* from dbo.sysobjects a, sysindexes b, sysusers c where a.name='tablename' and a.id=b.id
and c.uid=a.uid and c.name='ownername'
```

Note that tablename and username are case sensitive.

Last Modified: 11-FEB-00

## Article #20590: Stored Procedures not working in MS-SQL 7

### Question:

Why can't I use my MS-SQL 6.5 stored procedures in MS-SQL 7.0?

### Answer:

Although the BDE can be used to connect to MS-SQL 7 tables, it only supports MS-SQL 6.5 features. Here's why.

In this case the new feature in MS-SQL 7 that is not supported is the ability to have two stored procedures with the same name and different parameters. You will notice Stored Procedures in MS-SQL 7 have a ;1 or ;2 after the name. If you have a procedure with a ;1 after it you should be able to remove the ;1 and use it without a problem. If it has a ;2 or higher after it you should rename the procedures which have the same name, and remove the ;# and then it should work okay.

Last Modified: 11-OCT-00

## **Article #20632: What version of PVCS does TeamSource™ require?**

### **What version of PVCS do I need, to use the auto-merge feature of TeamSource?**

You only need the basic PVCS engine. Make sure when you install TeamSource that the PVCS directory is in the path. Then check the PVCS option and TeamSource will be able to detect if you have PVCS. Once detected, Team Source will generate the IPVCS.tsx file and place this in the Delphi5\Bin directory.

Last Modified: 04-JUN-01

## Article #20641: How do I change an ISAPI DLL project to a CGI project, or vice-versa?

### Building Web Applications

By Corbin Dunn

**Question:** How do I change an ISAPI dll project to a CGI project, or vice-versa?

**Answer:** The easiest way to do this is with conditional build and defining the target in the source. By simply commenting the {\$define dll} line, you can build an EXE instead of a DLL.

Create a new Web Server Application by choosing "File->New, Web Server Application", and selecting a "ISAPI/NSAPI Dynamic Link Library." After the project has been created, save it, and click "Project->View Source." Copy and paste the sample code from below into the project source.

#### Download a sample project

#### Sample Code

```
// Comment or uncomment the line below to change from an ISAPI dll
// to a CGI app or vice-versa
//{$define dll}

{$ifndef dll}
program Project1;
{$APPTYPE CONSOLE}
{$else}
library Project1;
{$endif}

uses
  WebBroker,
  {$ifdef dll}
  ISAPIApp,
  {$else}
  CGIApp,
  {$endif}
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

{$R *.RES}

{$ifdef dll}
```

**exports**

```
    GetExtensionVersion,  
    HttpExtensionProc,  
    TerminateExtension;  
{$endif}
```

**begin**

```
    Application.Initialize;  
    Application.CreateForm(TWebModule1, WebModule1);  
    Application.Run;  
end.
```

Last Modified: 23-FEB-00



**Article #20644: Loading an ISAPI DLL with Informix results in Error.**

Q. Why, when I load an ISAPI DLL that opens a connection to an Informix database, do I get the error "The dynamic link library ISQLT09A.dll could not be found in the specified path", or "The dynamic link library ISQLT07C.dll could not be found in the specified path", yet when I connect in any other BDE application I can connect fine?

A. IDAPI DLLs use the IUSR\_computername user account and therefore don't have access to the user path. The error indicates that the Informix client can't be found in the path for the IUSR\_computername account. To solve the problem, add the Informix and Informix/bin directories to your System path using the System control panel and reboot.

Last Modified: 01-MAR-00

## **Article #20646: ISAPI DLL when connecting to Informix gives sql.iem error**

Q. Why, when my ISAPI DLL connects an Informix database, do I get the error "Cannot open file 'sql. iem'", yet in any other BDE application I cannot connect fine?

A. ISAPI DLLs use the IUSR\_computername account. The problem here is that the IUSR\_computername account can't see the appropriate registry entries. To resolve the problem run regedit and copy HKEY\_CURRENT\_USERSoftwareInformix to HKEY\_USERS. DEFAULTSoftwareInformix.

Last Modified: 25-FEB-00

**Article #20659: Delphi 5 trial edition premature expiration**

**QUESTION:**

Why has my trial version of Delphi 5 expired before the 60 days are up?

**ANSWER:**

There is a bug in the trial edition timer which can cause it to expire early on some machines. We have a fix available for this. In order to receive a copy of the fix please go [here](#) and submit an install web case. We will then email you the fix.

Last Modified: 26-JUN-01

## **Article #20661: "Operation not applicable" when opening ClientDataSet.**

### **QUESTION:**

Why am I getting the error "Operation not applicable" when I open a TClientDataset with many master-detail datasets?

### **ANSWER:**

There is a limit of 15 master-detail relationships when using nested datasets.

There are two ways you can set up master/detail relationships in client datasets using nested tables.

One is to define them explicitly using the Fields Editor. At design time, right click the client dataset and choose Fields Editor. Add a new persistent field to your client dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of your detail table.

A second way that you may not be aware of is to obtain records that contain nested details from a provider component. When a provider component represents the master table of a master/detail relationship, it automatically creates a nested dataset field to represent the details for each record.

Last Modified: 11-OCT-00

## Article #20689: How do I set the desktop icon color in code, or make the text behind them transparent?

**Question:** How do I set the desktop icon color in code, or make the text behind them transparent?

**Answer:** To do this, you will have find the "SysListView32" window (which is the desktop icon list view window). First, you will have to find the main parent window called "Progman". It has a child window called "SHELLDLL\_DefView" that has the "SysListView32". You can use the API call `FindWindow` to find the "Progman" window, and then `FindWindowEx` to find the child window. When you finally have the right handle to the "SysListView32" window, you can use the `ListView` macro's `ListView_SetTextBkColor` and `ListView_SetTextColor` to set the color's to whatever you want.

Below is a sample unit that does this for you. Simply use the procedure `SetDesktopIconColor` to set the desktop icon color. If the parameter `Trans` is true, it makes the background transparent, otherwise it used the given `Background` color.

### Download the DeskIcons.pas file

#### DeskIcons.pas Unit

```
unit DeskIcons;  
  
interface  
uses Graphics; // Definition of TColor  
  
procedure SetDesktopIconColor(Foreground, Background: TColor; Trans: Boolean);  
procedure SetDefaultIconColors;  
  
implementation  
uses Windows, CommCtrl; // Definition of HWND and ListView_XXXXX  
  
procedure SetDesktopIconColor(Foreground, Background: TColor; Trans: Boolean);  
{  
  This procedure set's the desktop icon text color  
  to a given color with the option to add a transparent background.  
}  
var  
  Window: HWND;  
begin  
  // Find the right window with 3 calls  
  Window := FindWindow('Progman', 'Program Manager');  
  // FindWindowEx is used to find a child window  
  Window := FindWindowEx(Window, HWND(nil), 'SHELLDLL_DefView', '');  
  // SysListView32 is the desktop icon list view  
  Window := FindWindowEx(Window, HWND(nil), 'SysListView32', '');  
  // Use the macro to set the background color to clear  
  if Trans then
```

```

    ListView_SetTextBkColor(Window, $ffffffff) // back color
else
    ListView_SetTextBkColor(Window, Background); // back color
    ListView_SetTextColor(Window, Foreground); // foreground color
    // now send a redraw to the icons to redraw the new color
    ListView_RedrawItems(Window, 0, ListView_GetItemCount(Window) - 1);
    UpdateWindow(Window); // force the redraw to take effect immediatly
end;

procedure SetDefaultIconColors;
{ This set's the colors to be whatever is currently
  stored by windows }
var
    Kind: Integer;
    Color: TColor;
begin
    Kind := COLOR_DESKTOP;
    Color := GetSysColor(COLOR_DESKTOP);
    SetSysColors(1, Kind, Color);
end;

end.

```

Last Modified: 23-FEB-00

## Article #20716: Customizing the time/date display of the TDateTimePicker

### Question:

The TDateTimePicker's time display defaults to showing hours, minutes, and seconds. I would prefer not to show seconds, but there doesn't seem to be a way of changing the format. Is there a way?

### Answer:

The underlying API control can be customized using the DTM\_SETFORMAT window message. Use the Perform message to send this message and its parameters to the control.

### Example in C++:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    DateTimePicker1->Perform(DTM_SETFORMAT, NULL, "hh:mm tt");
    // will yeild a display without seconds, like this: "8:30 PM"
}
```

### Example in Object Pascal:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    s: String;
begin
    s := 'hh:mm tt';
    DateTimePicker1.Perform(DTM_SETFORMAT, DWord(nil) ,DWord(s));
end;
```

Consult the Windows API documentation [here](#) for further information on formatting strings.

Note: This technique bypasses the VCL wrapper around the date time picker by sending a message directly to the Windows control. This causes one problem. If you send a message that sets the formatting to show date elements while the Kind property is set to dtkTime (or vice versa), the date elements will display correctly, but the user may not be able to change those elements (or elements will change incorrectly). Therefore, it is suggested that you be sure the elements you display correspond to the control's Kind.





**Article #20755: Does the BDE use Client side or server side cursors for MSSQL?**

Q. Does the BDE SQL Links driver to MSSQL use client side or server side cursors?

A. Server side cursors are used. The driver does not use the client side cursor API at all.

Last Modified: 29-FEB-00

## **Article #20782: How to obtain IDAPI32.LIB and IDAPI.H**

Q. I need to get the files IDAPI32.LIB and IDAPI.H. Can I download them? How do I get them?

A. The BDE is not free. These files are included with Borland C++ 5.x. To obtain Borland C++ 5.02 you must purchase C++ Builder Profession or Enterprise edition and Borland Borland C++ 5.02 is included on the Companions Tools CD.

Last Modified: 01-MAR-00

## **Article #20791: Versions of the BDE that support MSSQL 7**

Q. What versions of the BDE support MSSQL 7?

A. SQLMSS32.DLL is the BDE SQL Links driver used to connect to MSSQL 7. Any version of this DLL later than 5.01(23) supports MSSQL 7.

This applies to all 32 bit versions of Delphi and C++ Builder.

Last Modified: 06-APR-00

## **Article #20802: Access Violation when opening any Access Table**

Q. If I attempt to get a list of tables in SQL Explorer against an Access database, or just execute a SQL statement I get an Access violation, even though the Microsoft Access product works just fine. What can cause this?

A. In general terms your DAO install is bad in some regard. Obtaining a newer version of DAO350.DLL or OLEAUT32.DLL generally solves the problem. See <http://support.microsoft.com/support/kb/articles/Q231/8/41.ASP?LNG=ENG&SA=ALLKB> for additional information.

Last Modified: 29-FEB-00

**Article #20903: Missing icons for the office components on the Server tab.**

**QUESTION:**

Why am I missing icons for the office components on the Server tab?

**ANSWER:**

This was caused by an import error, and has been fixed in Delphi 5 update pack 1 which is currently available for download from our website at

<http://www.borland.com/devsupport/delphi/downloads>.

Last Modified: 11-OCT-00

## **Article #21480: Why can't I connect to an Access database using the BDE and native MSACCESS driver?**

### **Question:**

Why can't I connect to an Access database using the BDE and native MSACCESS driver?

### **Answer:**

#### ***Access 2000:***

You cannot connect to an Access 2000 database using the BDE's native MSACCESS driver. It is recommended that you upgrade to Delphi 5 and use ADO. You could also connect using the BDE with ODBC.

#### ***Access 95 or 97:***

You may get the message "BDE error 13059" and "General SQL error". If you are trying to connect to Access 95 or 97 databases using the BDE's native MSACCESS driver, then you need to have DAO 3.0 installed for Access 95 or DAO 3.5 for Access 97 (DAO = Data Access Objects for Visual Basic). You can install the DAO by doing a custom installation of MS Office and selecting only to install Data Access Objects for Visual Basic. You must also have the correct DLL32 setting in the BDE Administrator for the DAO you have installed: IDDA3532.DLL for Access 97, IDDAO32.DLL for Access 95.

Last Modified: 03-AUG-01

## **Article #21484: How do I prevent my application's main window from showing when my application loads?**

### **Question:**

How do I prevent my application's main window from showing when my application loads?

### **Answer:**

There are some cases in which you may not want the application's main window to show unless a certain criteria is met (i.e. successful user/password authentication, successful connection to database, etc.) or you may not want the window to show at all. To do this, set the Application's ShowMainForm property to false (`Application.ShowMainForm := false`) in the main project file before `Application.Run`, the `OnCreate` of one of your auto-created forms, or in the initialization section of one of your units.

Last Modified: 29-MAR-00

## Article #21516: How to capture the URL and title of the current webpage being viewed.

### Question:

## How can I see if Netscape is running and capture the current URL?

### Answer:

This code sample will allow you to see if Netscape is running. Also, it will put the URL and page title into one string. This code will also work for IExplorer. Just replace the Netscape with IExplorer. Place a TEdit component on the form, to view the current URL and page title.

Uses

```
DdeMan;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  DDE: TDdeClientConv;
```

```
begin
```

```
  DDE := TDdeClientConv.Create(self);
```

```
  if DDE.SetLink( 'Netscape', 'WWW_GetWindowInfo' ) then
```

```
    Edit1.Text := DDE.RequestData( '0xFFFFFFFF', sURL, sTitle' )
```

```
  else
```

```
    ShowMessage( 'Netscape is not running' );
```

```
  DDE.Free;
```

```
end;
```

Last Modified: 05-APR-00



## Article #21519: Items disappearing from the Uses tab in the Type Library editor.

### Question:

Why are Type Libraries disappearing from the Uses tab in the Type Library editor?

### Answer:

The Delphi type library editor is designed to remove type libraries listed on the Uses tab which are not actually used by the type library currently open. Unfortunately there is a bug with this process which does not account for indirect dependencies. For example, library A uses an interface from library B and library B uses an interface from library C. Library A also has a function with a return type which is defined in C. Because Library A does not explicitly use any interfaces from C, C is removed from the Uses tab, but then subsequently refreshing and compiling a unit which uses library A's pas file, A\_TLB.pas, will result in a compiler error along the lines of "Undeclared identifier".

Fortunately there is a simple work around for this. Any type libraries which need to remain on the Uses tab, but are being automatically removed, can be forced to remain by creating a dummy interface which uses an interface from that type library. To do this open your type library and go to the Uses tab. Add the libraries you need. Now create a new interface and call it something like Dummy1. Select a parent interface from the type library you wish to retain. This will force it to remain on the Uses tab. On the Flags tab check 'hidden'. Now refresh and you should be all set.

Last Modified: 11-OCT-00

## Article #21543: How do I get a list of CDROM drives on a computer?

### Question:

How do I get a list of CDROM drives on a computer?

### Answer:

```
procedure GetCDROMDrives(list: TStrings);  
var  
    DriveBits: set of 0..25;  
    Drives,DriveNum: integer;  
    DriveLetter: string;  
begin  
    list.Clear;  
    Drives := GetLogicalDrives;  
    if Drives <> 0 then  
    begin  
        integer(DriveBits) := Drives;  
        for DriveNum := 0 to 25 do  
        begin  
            if (DriveNum in DriveBits) then  
            begin  
                DriveLetter := char(DriveNum+Ord('A'))+':';  
                if GetDriveType(PChar(DriveLetter)) = DRIVE_CDROM then  
                    list.Add(DriveLetter);  
            end;  
        end;  
    end;  
end;
```

Last Modified: 05-APR-00

## **Article #21548: New Interbase driver configuration parameter COMMIT RETAIN.**

### **Question:**

What is the COMMIT RETAIN property and why can't I see it in the BDE Administrator?

### **Answer:**

COMMIT RETAIN is a new Interbase driver configuration parameter. It was introduced for explicit transactions to retain cursors on commit. It will not appear in existing aliases but should appear in new Interbase aliases, along with two other new parameters - WAIT ON LOCKS and ROLE NAME.

If these new properties do not appear when creating a new Interbase alias try this fix: After installing the latest version of the BDE, run BDE Administrator and select Object|Merge Configuration. In the Open dialog, enter SQL\_INT.CNF as the file name. This file should be in the main BDE directory. See the BDE help for more information.

Last Modified: 11-OCT-00

**Article #21549: Code sample to detect if there is a sound device present on a computer.**

**Question:**

**How can I detect if a sound device is present on the computer were my application is running?**

**Answer:**

Use this code sample to detect if there is a sound device on the machine your application is running on. Make sure you add the unit MMSystem to your uses clause.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if waveOutGetNumDevs > 0 then
        ShowMessage( 'Wave-Device present' )
    else
        ShowMessage( 'No Wave-Device present' );
end;
```

Last Modified: 05-FEB-02

## Article #21550: Converting an integer into a binary string

### Question:

How do I convert an integer into a binary string?

### Answer:

```
function IntToBinStr(num: integer): string;  
var  
    i: integer;  
begin  
    for i := 0 to 31 do  
        Result := IntToStr((num shr i) and 1)+Result;  
end;
```

Last Modified: 05-APR-00

## Article #21552: RichEdit in a COM server is not being correctly created

### Question:

I have a RichEdit control in my COM Server application that uses the tmFree threading model for COM objects, and the RichEdit control is not being created. How can I get around this?

### Answer:

This is related to a bug in the VCL for Delphi 5 and you can [click here to read more about it](#). (It is also mentioned in the readme file for Delphi 5).

One possible workaround given was to override CreateParams for a TRichEdit and add the ES\_EX\_NOCALLOLEINIT style to it. However, this may not always work. Another possible solution is to capture the WM\_NCCREATE message and set the proper params at that point. Below is a component that will do that for you.

### [Download FixedRichEdit.pas](#)

#### Sample Code

```
unit FixedRichEdit;  
  
interface  
  
uses  
    Windows, Messages, Classes, Controls, RichEdit, ComCtrls;  
  
type  
    TFixedRichEdit = class(TRichEdit)  
    protected  
        procedure WMNCCreate(var message: TWMNCCreate); message WM_NCCREATE;  
    end;  
  
procedure Register;  
  
implementation  
  
procedure TFixedRichEdit.WMNCCreate(var message: TWMNCCreate);  
var  
    gwLEx : Longint;  
begin  
    gwLEx := GetWindowLong(WindowHandle, GWL_EXSTYLE);  
    gwLEx := gwLEx or ES_EX_NOCALLOLEINIT;  
    message.CreateStruct^.dwExStyle := gwLEx;  
end;
```

```
    inherited;  
end;  
  
procedure Register;  
begin  
    RegisterComponents('Samples', [TFixedRichEdit]);  
end;  
  
end.
```

Last Modified: 05-APR-00

## Article #21553: Getting "unmapped SQL error code 10038" when connecting to MS SQL.

### Question:

Why am I getting the error "Unmapped SQL error code 10038 - Attempt to initiate new SQL server operation with results pending"?

### Answer:

One reason we have found for this error is an incomplete connection to the server via a VPN and/or proxy server. We have Debugged the BDE code up to the point of a dialup connection to a server failing with the 10038 error. It failed during the single MS DBLIB function call to connect to the server. There aren't any changes that can be made or options that can be set that will "fix" this. It seems to do with how a MS DBLIB connection behaves across a firewall and/or VPN. Each subsequent action against the server (executing an sql statement of any kind) resulted in the "...results pending error".

You might find that other tools don't do this. MS's Windows ISQL is a good example. The main difference there is that it will try to reconnect when the initial connection appears to been dropped or failed to succeed. The BDE does not do this because many of its operations rely on a persistent connection to carry out supported features."

There is now a way to work around this. There is a new TDatabase param called "CANCEL QUERY". It can be set to either "dbcancel" or "dbcquery". In addition to params like:

```
USER NAME=varick  
PASSWORD=mypass
```

a TDatabase's params list can include

```
CANCEL QUERY=dbcquery
```

"dbcancel" is the default. "dbcquery" was added to take care of those situations where the MSSQL server is unable to process a "dbcancel" of pending results which may be the case when getting this error.

But before this property can be used in a TDatabase the parameter must be added for the MSSQL driver. This can be done by adding the following registry entry to the DB OPEN section of the MSSQL driver (using REGEDIT.EXE):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Borland\Database Engine\Settings\DRIVERS\MSSQL\DB  
OPEN
```

```
Name: CANCEL QUERY Data: ""
```



Without the registry change the CANCEL QUERY param in the TDatabase has no affect (always dbcancel). Once the registry entry had been made to the MSSQL driver the CANCEL QUERY param could be used to control the cancel mode (dbcancel or dbcanquery).

Another possible workaround might be to use ODBC instead as it does not use DBLIB.

Last Modified: 11-OCT-00

## Article #21564: Initializing an array or record to null.

### Question:

How Do I initialize an array or record to null?

### Answer:

By using the FillChar procedure. Here is an example:

```
procedure TForm1.Button1Click(Sender: TObject);  
type  
  TIntRec = record  
    i1: integer;  
    i2: integer;  
    i3: integer;  
    i4: integer;  
    i5: integer;  
  end;  
var  
  IntArray: array[0..4] of integer;  
  i: integer;  
  s: string;  
  IntRec: TIntRec;  
begin  
  // the array:  
  FillChar(IntArray,SizeOf(IntArray),0); // if you comment out this  
line, you will have random values in the array  
  for i := low(IntArray) to high(IntArray) do  
    s := s+IntToStr(IntArray[i]);  
  form1.Caption := 'IntArray: '+s;  
  // the record:  
  FillChar(IntRec,SizeOf(IntRec),0); // if you comment out this line,  
you will have random values in the record  
  form1.Caption := form1.Caption+' '+IntToStr(IntRec.i1)  
+IntToStr(IntRec.i2)+IntToStr(IntRec.i3)+IntToStr(IntRec.i4)+IntToStr  
(IntRec.i5);  
end;
```

## **Article #21573: Why is my system freezing when I start Delphi, just after closing the Just-in-time debugging dialog box?**

### **Question:**

Why does my system freeze after I see the message "The Just-in-time debugger is set to ". In order for it to work correctly, it needs to be changed to "C:\Program Files\Borland\Delphi5\Bin" -aargs %ld %ld"?

### **Answer:**

This issue probably has nothing to do with Just-in-time debugging. It is most likely a video driver issue that happens to be showing up right after the just-in-time dialog box closes. To verify this try turning off the hardware acceleration on your machine under display properties (Win 95 and Win 98 only), restart your machine and restart Delphi. If this fixes the issue you know it was caused by the video driver. See the following for further explanation:

<http://community.borland.com/article/0,1410,19602,00.html>

Last Modified: 11-OCT-00

## Article #21592: A final prompt, asking user if they really want to exit.

### Question:

How do I prompt a user, after they select a close method, for a response that they really want to close down the application?

### Answer:

Put this sample code into the OnCloseQuery event of your form. After a close method has been invoked on the form, a message will pop up asking the user if they really want to exit. If the user selects yes, the application will close normally, if they say no, the application will stay open.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
  if (mrNo = MessageDlg('You really want to exit?', mtInformation,
[mbYes, mbNo], 0)) then
    begin
      CanClose := False;
    end;
end;
```

Last Modified: 05-APR-00

## Article #21701: Clearing a console window screen

### Question:

How do I clear a console screen?

### Answer:

```
function ClearConsoleScreen: boolean;
const
    BUFSIZE = 80*25;
var
    Han,Dummy: LongWord;
    buf: string;
    coord: TCoord;
begin
    Result := false;
    Han := GetStdHandle(STD_OUTPUT_HANDLE);
    if Han <> INVALID_HANDLE_VALUE then
        begin
            if SetConsoleTextAttribute(han, FOREGROUND_RED or
FOREGROUND_GREEN or FOREGROUND_BLUE) then
                begin
                    SetLength(buf, BUFSIZE);
                    FillChar(buf[1], Length(buf), ' ');
                    if WriteConsole(han, PChar(buf), BUFSIZE, Dummy, nil) then
                        begin
                            coord.X := 0;
                            coord.Y := 0;
                            if SetConsoleCursorPosition(han, coord) then
                                Result := true;
                        end;
                    end;
                end;
            end;
        end;
end;
```

## **Article #21703: Oracle 8 problems introduced in BDE 5.1.1.**

Previous to BDE 5.1.1, setting ROWSET SIZE for an alias connected to Oracle 8 did nothing. BDE 5.1.1 added this as a new feature as a performance enhancement to improve the speed when many records are fetched.

However, this new feature introduces 3 new problems:

- When the number of rows returned from a TTable or TQuery matches a multiple of the setting of ROWSET SIZE data will be displayed twice.
- If the last row of a TTable or TQuery has fields which are blank (NULL), the corresponding field in the first record will be blank as well. This problem does not occur when using the Oracle 8.1.5 client (8i). When using the Oracle 8.0.4 client R&D has determined this problem to be caused by a bug in the Oracle 8.0.4 client. Therefore, ROWSET SIZE must be set to 0 when using the 8.0.4 client.
- Getting a list of tables in SQL Explorer may result in the error "Translate Error. Value out of Bounds."

The workaround for all of these problems at this time to set ROWSET SIZE to 0.

## Article #21721: Loading Params for some Stored Procs in Oracle Packages causes AV

As of BDE 5.1.0, Developer Support has seen a number of conditions where a Stored Proc is created correctly inside of a package, yet it will not operate correctly. This problem was introduced in version 5.1.0 as a result of fixing a bug where you crashed loading params for a stored proc with greater than 10 parameters. The conditions where this problem occur are:

- In SQL Explorer the valid property for the package and the package body returns Yes.
- Either the Oracle 8.0.4 or 8.1.5 client is used against a database of the same version.
- In the BDE Administrator settings for the Oracle native driver, DLL32 is set to SQLORA8.DLL and Vendor Init is set to OCI.DLL.
- Either Prepare is called for the Stored Proc, or the Params editor is accessed in Delphi or C++ Builder.

Under these conditions you either get an AV, or are exited from the IDE of Delphi or C++ Builder.

As of this time we don't have an exact description for what Stored Procedures work and which one do not. However; our observation is that most work. For those that have this problem, the workaround is to move the Stored Procedure out of a package and into a standalone Stored Procedure.

As of this time we believe this problem to be in the Oracle client. It is possible a future version of the Oracle client may resolve the problem.

Last Modified: 24-OCT-00

## **Article #21725: Closing and reopening a prepared query with Oracle 8 causes memory leak.**

### **Question:**

When I close and reopen a query with Oracle 8 I notice a memory leak. What causes this?

### **Answer:**

Every open of a Prepared query with Oracle 8 causes the HeapSize as reported from DbGetSysInfo to grow. When HeapSize equals MemSize an Out of Memory exception occurs. The only workaround at this time is UnPrepare the query after every execution of the Query.

This error is know to occur with BDE 5.0, 5.0.1, 5.1.0 and 5.1.1. with Oracle clients 8.0.4 and 8.1.5.

At this time, we believe this problem to be caused by a memory leak in the Oracle client. A future version of the Oracle client may resolve this error.

Last Modified: 12-APR-00



## Article #21726: Logging in with an incorrect login results in different errors with Oracle 7 and 8

### Question:

Why do I get different errors, when I login with an incorrect user name or password, using Oracle 7 vs. Oracle 8.

### Answer:

When connecting to Oracle 7, DLL32 in BDE Administrator is normally set to SQLORA32.DLL. When connecting to Oracle using SQLORA32.DLL the connection is made via calling the Oracle OCI function ologon.

When connecting to Oracle 8, DLL32 in BDE Administrator is normally set to SQLORA8.DLL. When connecting to Oracle using SQLORA8.DLL the connection is made via calling the Oracle OCI function OciBegin Session.

The reason SQLORA8.DLL and SQLORA32.DLL raise different errors is because ologon and OciBeginSession generate different errors when an incorrect login is attempted.

If you connecting to an Oracle 8 database, but like the errors that SQLORA32.DLL generates, here is a way to get the most out of SQLORA8.DLL and SQLORA32.DLL.

```
//Place this code it the location where connecting to the database
occurs
var
  DriverParams : TStringList;
begin
  DriverParams := TStringList.Create;
  try
    DriverParams.Add('DLL32=SQLORA32.DLL');
    Session.ModifyDriver('Oracle', DriverParams);
    Databasel.Open;
    //if no exception is raised, then close database and reopen with
SQLORA8.DLL
    Databasel.Close;
    DriverParams.Clear;
    DriverParams.Add('DLL32=SQLORA8.DLL');
    Session.ModifyDriver('Oracle', DriverParams);
    Databasel.Open;
  finally
    DriverParams.Free;
  end;
end;
```

Last Modified: 13-APR-00

## Article #21727: General SQL Error when connecting to MSSQL

### Question:

I can connect to MSSQL using ODBC, ADO and Enterprise Manager. Yet when I try to connect with BDE I get a General SQL Error. What is wrong?

### Answer:

It appears the Database Name property of your alias is invalid. To simply test whether you can connect, make the database name property blank. This will cause you to connect to the default MSSQL database on the server.

Last Modified: 13-APR-00

## Article #21728: Application Taskbar Button

### Question:

Why do operations I attempt to perform on the taskbar button for my main VCL form have no effect? (e.g. FlashWindow)

### Answer:

When calling Win32 functions or handling window messages associated with an application's taskbar button, you must use the global TApplication's window handle (TApplication::Handle).

This window handle is the parent of all the application's forms' handles. Many Win32 functions require window handles that are not that of child windows. FlashWindow is an example of such a function. It can flash the title bar and/or the task bar button of a window. If the window is a child window, then it can only flash the title bar of the window. Supplying Application->Handle for the HWND parameter of FlashWindow will allow flashing of the taskbar.

Last Modified: 02-MAY-00

## Article #21742: TTable.Edit always executes SELECT statement

### Question:

Whenever I start editing a record, even with Cached Updates ON, why is a SELECT statement executed?

### Answer:

The BDE needs to know what the current values are in the record so that when it goes to do an UPDATE it knows what record to change.

Last Modified: 25-APR-00

**Article #21773: Delphi 3 Midas apps not usable with BDE 5.1.1.**

The IDAPI32.DLL file included with BDE 5.1.1 does not include two exports, DsProviderGetDataPacket and dsResolver. This will cause Midas applications written in Delphi 3 to fail when used with it.

The workaround at this time is to use the IDAPI32.DLL file that comes with BDE 5.1.0.

Last Modified: 27-APR-00

**Article #21774: Read only access to FoxPro is recommended.**

Based on extended testing of the Native BDE FoxPro driver, Inprise/Borland advises against any use of this driver other than for read only access. Usage other than read only may result in crashes, Access Violations or corruption of indexes.

Last Modified: 02-AUG-00

## Article #21821: BDE uses small packet sizes

### Question:

I've noticed that the BDE uses small packet sizes. Is there are way to adjust this?

### Answer:

While the BDE does not explicitly request or know about packet sizes, the end result is that they are always small. In a LAN environment this works out fine. However, in a WAN environment this can result in a serious degradation of performance. Using BDE to ODBC or ADO may cause less degradation to occur.

Last Modified: 18-MAY-00



## Article #22086: How to tell if a form has been created (or any object)?

### Question:

How to determine if a form has been created (or any object)?

### Answer:

If you creating and freeing an object several times, and you need to be able to tell if the object is currently created or freed, set the object's reference to nil everytime you free it. For example:

```
MyObject.Free;  
MyObject := nil;
```

Or in Delphi 5 or higher:

```
FreeAndNil(MyObject);
```

Then you can simply use the Assigned function to test if the object is nil or not.

Last Modified: 20-JUN-00

## **Article #22090: Internal Error 2140 with NT Service Application and ADO**

### **Question:**

Why am I getting internal error 2140 from my service application that uses ADO?

### **Answer:**

You need to call CoInitialize (and CoUnInitialize when you're done). You can use the OnStart or OnStop events.

Last Modified: 31-MAY-00

## **Article #22100: What version of Delphi 5 does the Code Explorer come with?**

### **Question:**

Where is the Code Explorer, why can't I see it in my installation of Delphi 5?

### **Answer:**

The Standard version of Delphi 5 does not come with Code Explorer. The Professional and Enterprise versions both come with Code Explorer. For a description on the different products that come with the different versions of Delphi 5, please see our feature matrix: <http://www.borland.com/delphi/productinfo/featurelist/> .

Last Modified: 31-MAY-00

## Article #22101: How to embed Adobe Acrobat into your application.

### Question:

How can I embed Adobe Acrobat into my application so users can read PDF files?

### Answer:

This document describes the process on how to use Adobe Acrobat to display files inside of your application.

1. First, you will have to ensure that you have the latest version of Acrobat on your machine. You can [download it from Adobe](#).
2. Next you will need to import the type library for Acrobat, and be sure to click the Install button to install it into the IDE:

Import Picture

3. You can now drop a TPDF component onto your form and use it however you like.

### -- Tips --

---

To set the file that you want open, use the src property, such as:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    // This example assumes that you have a TOpenDialog  
    // and TPDF dropped onto your form  
    OpenFileDialog.Filter := 'PDF Files (*.pdf)|*.pdf';  
    if OpenFileDialog.Execute then  
        Pdf1.src := OpenFileDialog.FileName;  
end;
```

---

If you don't want the Acrobat toolbar to be shown, you can call **Pdf1.setShowToolbar(false);**. You can then have custom buttons that allow the user to navigate through the PDF document by calling **gotoFirstPage**, **gotoLastPage**, **setZoom(percent: Single)**, etc. To find out a list of all the functions you can call, open the file that you imported (it is probably called PdfLib\_TLB.pas), and look at all the functions in the class **TPdf = class(TOLEControl)**. To easily browse to this file, you can place the cursor over the PdfLib\_TLB name in the uses clause and press Ctrl-Enter.

Last Modified: 31-MAY-00

## Article #22113: Indices Vs. Keys

**Question:** When dealing with databases, what is the difference between keys and indices?

**Answer:** An index is a marker in a table which other tables can operate upon. A key is a value that a table uses internally to order its data.

Imagine you had a table of customers, working with a table of orders. Your orders would be keyed by their order numbers, and your customers by their customer numbers.

This is fine until we want to view our orders in terms of which customers made them. We would then use an index into the orders table. Using this index, we could find all of the orders which were made by a certain customer, even though the table itself isn't ordered in this fashion.

Last Modified: 27-OCT-00

## Article #22114: How to use sets in code, such as adding to TDBGrid.Options, at run time

**Question:** I know how to use the object inspector to change the value of a set, such as TDBGrid.Options, but how do I manipulate sets in code?

**Answer:** To use sets, you will have to use square brackets to refer to set elements. Sets can have at most 256 elements in them. Here is an example of using and declaring sets:

### Sample Code

```
type
// Here are some examples of declaring set types.
// A variable of TMyMonth can take one and only one of
// one of its members.
TMyMonth = (Jan, Feb, Mar, Apr, May);
// TMyMonthSet can be an empty set (no months in it), or
// contain any combination of the above months.
TMyMonthSet = set of TMyMonth;
// TMySubMonthSet is a subset of the above set, only able
// to take on the first 3 months
TMySubMonthSet = set of Jan..Mar;

// TMyIntSet can be an empty set, or any combination of integers
// in the range of 1 to 15.
TMyIntSet = set of 1..15;
// TMySubIntSet can be an empty set, or any combination of integers
// in the range of 1 to 5.
TMySubIntSet = set of 1..5;

// Here are some examples of declaring set variables
var
  MyMonth: TMyMonth; // can only be one month (NOT a set)
  MyMonthSet: TMyMonthSet; // can be more than one month
  MyIntSet: TMyIntSet; // can be a combination of ints from 1 to 15 only
begin
  // The TMyMonth type can only take on one value at a time
  MyMonth := May;

  // Sets can take on more than one value, and notice the use of square
  // brackets
  MyMonthSet := [Jan, Mar, May];
  // MySubMonths can only take the values Jan..Mar. If you try to set it
  // to May, you won't get a compile time error, but the set will be empty.

  MyIntSet := [20]; // 20 is past the range for the set..so this should do nothing!
  if MyIntSet = [] then
    ShowMessage('MyIntSet is empty! (as expected...)');

  // Here are some common operations you can do with sets:
  // + Union operator
  MyMonthSet := MyMonthSet + [Feb, Apr]; // Add 2 months to the set
```

```
// - Difference operator
MyMonthSet := MyMonthSet - [MyMonth]; // Remove May from the set
MyMonthSet := MyMonthSet - [Jan..Apr]; // Remove Jan through Apr
// The set should now be empty (use the equality operator = to test sets)
if MyMonthSet = [] then
    ShowMessage('MyMonthSet is empty! (as expected...)');

// * Intersection operator
MyMonthSet := [Jan, Feb] * [Jan, Apr];
if MyMonthSet = [Jan] then
    ShowMessage('MyMonthSet is Jan! (as expected...)');

// <= Subset operator
if [Jan, Apr] <= [Jan..Apr] then
    ShowMessage('[Jan, Apr] <= [Jan..Apr], as expected...');
// <= Subset operator is the opposite of the above

// To test if an ordinal is in the set, us the "in" operator
if Jan in [Jan..Feb] then
    ShowMessage('Jan in set [Jan..Feb]');
end.
```

Last Modified: 31-MAY-00



## Article #22115: Setting default value for decimal places in the BDE.

### Question:

How can I control the number of decimal places in a floating point number when I do a BatchMove to a text file?

### Answer:

Field formats in a text file are stored in a schema file which has the same name as the text file with a .sch extension. When doing a batch move to a text file (batCopy) a .sch file is automatically created. All floating point fields are provided by default with two decimal places of accuracy. After the schema file is created you can edit it and change the number of decimal places to another value. This new value will be used in future batch moves.

The default number of decimal places is controlled by the BDE parameter DECIMALDIGITS which can be found in the BDE administrator under: Configuration | System | Formats | Number. Since this is a BDE parameter it can also be changed at runtime by changing the value in the registry at:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Borland\Database Engine\Settings  
\SYSTEMFORMATSNUMBER

Last Modified: 27-OCT-00

## **Article #22213: Non-TThread Threading in Delphi, Access Violation, Invalid Pointer Operation.**

### **Question:**

Why do I get "Access violation" or "invalid pointer operation" when doing multi-threading in Delphi when not using the TThread class or the BeginThread function (i.e. using the CreateThread API function, or writing an Entera server)?

### **Answer:**

If you use non-Delphi threads (not using TThread or BeginThread) in a Delphi application, then you must manually set the IsMultiThread variable from the System unit to true. See "IsMultiThread variable" in the online help files for more information.

Last Modified: 19-JUN-00

## Article #22269: Delphi 5 on Windows 2000 and Windows ME

### Question:

Can I use Delphi 5 with Windows 2000?

### Answer:

Delphi 5 is not fully certified to be used on Windows 2000 because it came out before Windows 2000. This means that officially we cannot support it, however there are many people using it without a problem. We have heard of only a couple isolated cases of people having issues that did not occur on Windows NT. The same is true for Windows ME although we've not seen as many cases where people are using it.

Be sure to install [Delphi 5 update pack 1](#) as there were a couple Win2K modifications made.

Last Modified: 14-DEC-00

## Article #22390: Auto-Increment fields in InterBase

### Question:

I have an InterBase table, but I can't figure out how to make an Auto-Increment field for it. How do I do this?

### Answer:

InterBase doesn't have a convenient AutoInc datatype like Paradox, so we have to do a little bit of work to get this functionality. Start out by creating an Integer field that is appropriate for the size of the values you want in your AutoInc field. From there, we need a trigger to calculate the value for the field:

```
CREATE TRIGGER autoinc FOR tablename
BEFORE INSERT AS
BEGIN
    NEW.autoinc_field_name = GEN_ID(generator_name, 1);
END
```

We aren't done yet. We need a persistent location to store the current AutoInc value. InterBase's way to deal with this is called a generator:

```
CREATE GENERATOR generator_name;
SET GENERATOR generator_name TO x;
```

GEN\_ID() is a built-in InterBase function specially designed to increment the generator specified by the first parameter by the value in the second parameter. The SET GENERATOR statement initializes the generator to the value provided by *x*. This is useful if your database is based on something like a customer number, and you want this customer number to start at a specific value rather than zero. And that is all there is to it. From here on, you can insert away and this trigger will stick the correct value into autoinc\_field\_name. For further information on how to work with Auto-Increment fields in your application, read the TI "Working With Auto-increment Field Types" at <http://community.borland.com/article/0,1410,16123,00.html>.

Last Modified: 18-JUL-00

## Article #22391: Row-Level Locking in MSSQL 7

### Question:

I'm having a hard time trying to make my application acquire one of those neat row-level locks that were introduced in MSSQL 7. How do I do this?

### Answer:

Row-level locking is definitely a powerful feature that was recently added to MSSQL server. Unfortunately, the current version of the BDE (5.11) does not support this functionality. The only way to get this out of your SQL Server is to bypass the BDE entirely with, for example, an ADO connection. In that situation, the lock is acquired by this simple SQL statement:

```
SELECT * FROM tablename WITH (UPDLOCK ROWLOCK) WHERE condition
```

*tablename* is the name of your table and *condition* is a conditional statement that should force only one record to be returned. The WITH clause will cause all rows returned to be locked with a row level lock, but if you need multiple rows locked you might want to look at another type of lock. This statement must be run in the context of a transaction, and this lock will persist until that transaction is committed or rolled back.

If you then run the MSSQL stored procedure `sp_lock()` during the timeframe of the transaction, you will see that your table has a lock of type RID, which denotes a row level lock. If you were to fire this exact SQL statement in a BDE application, `sp_lock()` would show a lock of either type TAB (table level) or PAG (page level).

Last Modified: 18-JUL-00

**Article #22506: Delphi 1 no longer available.**

**Question:**

Why is Delphi 1 no longer shipped with the latest version of Delphi?

**Answer:**

Delphi 1 was shipped with Delphi 2,3 and 4 because it could still be used to create 16 bit applications for Windows 3.1, but it was not shipped with Delphi 5. This is because Delphi 1 has not been Y2K tested and is not considered Y2K compliant.

Last Modified: 11-OCT-00

**Article #22584: With an inproc server that uses Sharemem, the caller AV's on exit.**

**QUESTION:**

Why is the caller getting an Access Violation on exit, when I use Sharemem?

**ANSWER:**

The problem is a call made to borlndmm.dll. The unload sequence on Win95 machines is at fault. The code runs fine on Win98, WinNT4, and Win2K. Linking the following unit will resolve the issue and the DLL does get unloaded.

```
unit Win95ShareMemFix;

interface

implementation

uses
  Windows, ActiveX, Sysutils;

initialization

  if (Win32Platform = VER_PLATFORM_WIN32_WINDOWS)
    and (Win32MajorVersion = 4) and (Win32MinorVersion <= 0) then
      Win32Check(CoLoadLibrary('borlndmm.dll', true) <> 0);

end.
```

Last Modified: 31-JUL-00

**Article #22586: Activating the clientdataset closes the query.**

**QUESTION:**

Why does my query close when I activate my clientdataset component?

**ANSWER:**

This is normal behavior in Delphi 5. When the clientdataset is activated and it is reading it's data from a query, the client dataset will close the query after it is done reading in the data. This is different from Delphi 4 because Delphi 4 uses Midas 2 which is not stateless, and Delphi 5 uses Midas 3 which is stateless. So Midas applications written in Delphi 4 may need to be redesigned to adhere to the stateless aspect of Midas 3.

Last Modified: 31-JUL-00



**Article #22597: When Quick Report printer settings are changed during a preview at run time, they will have no affect.**

**QUESTION:**

Why don't my printer setting take affect at run time?

**ANSWER:**

The Quick Report set up dialog of the report preview only works on the current print job. The setting are loaded and cannot be changed until the printing is done. You will need to change the settings again for the second print out. The settings are not saved and will need to be changed for each consecutive print.

Last Modified: 02-AUG-00

## Article #22609: Changing Quick Report BandType at runtime.

### QUESTION:

How do I change the BandType property of QRBand at run time?

### ANSWER:

Once you have assigned a BandType to a QRBand, you need to deactivate the band before you can change it's band type:

```
theReport.Bands.HasPageFooter := FALSE;  
theReport.Bands.PageFooter := nil;  
QRBand1.BandType := rbPageFooter;  
theReport.Bands.HasPageFooter := TRUE;  
theReport.Bands.HasPageFooter := QRBand1;
```

Last Modified: 31-JUL-00

**Article #22610: How to assign the Delphi global printer object, to the Quick Reports global printer object.**

**QUESTION:**

How do I assign the Delphi global printer object, to the Quick Report global printer object?

**ANSWER:**

If you are using Quick Reports and want to use Delphi's print dialogs, then you need to assign Delphi's global printer object to Quick Report's global printer object. Here is a code snippet that shows how to do this.

```
uses Printers;  
  
ReportForn.QuickRepl.PrinterSettings.PrinterIndex := Printer.PrinterIndex;
```

Last Modified: 14-AUG-00

## Article #22696: Error: "Type mismatch"

### QUESTION:

Why am I getting type mismatch error when I try to query my oracle table?

### ANSWER:

The numeric type that oracle uses allows a length field and a precision field. So declaring a field to be Numeric (10,4) specifies a number with 10 digits, 4 of which to be used for precision.

```
Declaration    | Valid input  
-----
```

```
Numeric (5,2)  | 345.23
```

```
Numeric(3)    | 345      *
```

\*Note: This Declaration appears to be an integer but is actually a floating-point number.

In both cases these numbers are stored as floating point Numbers. If your tables have type TIntegerField where a numeric field was declared, then you will get an error on activation. Technically the BDE handles Numeric fields as floats, even if they look like integers. If you are sure that the values in table are integers then you can use the Trunc function, which returns an Int64.

Last Modified: 14-AUG-00

## Article #22866: How to send a response early in a web application action.

### Question:

How to send a response early in a web application action.

### Answer:

Typically, with web applications, you set `Response.Content` to something in an Action and let the content be sent after the action is done. But what if you want to constantly stream out data to the client web browser? To do this, you will have to use `TWebRequest.WriteString` in combination with `TWebResponse.SendResponse`.

The trick is to not set `Response.Content` before calling `Response.SendResponse`. Setting `Response.Content` to a string and then calling `SendResponse` causes the Response to send a "Content-Length:" to the web browser client. The web browser will then expect nothing longer than the `Response.Content` and will close the connection early.

To keep the connection open, the first thing you should do is send `Response.SendResponse`. This properly fills up the Response header, but doesn't set the "Content-Length:". You can then use `Request.WriteString` to write individual strings directly to the web browser at any time.

Here is an example action that should give you an idea of how to do this:

```
// This example doesn't add the proper HTML starting  
// tags for simplicity  
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;  
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
var  
    I: Integer;  
    S: string;  
begin  
    // Important that this is first!  
    Response.SendResponse;  
    S := 'About to send time stamps<br>';  
    // Send the content early by accessing the Request  
    // directly and writing to it.  
    Request.WriteString(S);  
    // Send some Responses and pause to simulate lengthy operations  
    for I := 1 to 3 do  
    begin  
        // Pause for about 3 secs  
        Sleep(3000);
```

```
// Set another response  
S := TimeToStr(Now) + '<br>';  
Request.WriteString(S);  
end;  
end;
```

Last Modified: 24-AUG-00

## Article #22867: How to turn off ISAPI DLL caching on Windows 2000 and IIS5

### Question:

How do I turn off ISAPI DLL caching on Windows 2000 and Internet Information Services 5 (IIS5)?

### Answer:

You may want to turn off DLL caching to allow you to better debug ISAPI DLL's. Note that if you do turn it off, it is best to turn it back on when you are ready to use your DLL as it greatly improves performance.

Click on Start->Settings->Control Panel->Administrative Tools->Internet Services Manager. Right click on your website and select Properties:



IIS Manager Screen Shot

Select the "Home Directory" tab, and click on Configuration...:

Configuration

Uncheck "Cache ISAPI applications":



uncheck cache extensions

Last Modified: 24-AUG-00

## Article #22940: Can you drag and drop existing SQL structures directly onto a form?

**Question:** What is the easiest way to implement SQL Tables, Sequences, Procedures, etc. that have already been created, into the Data Access components?

**Answer:** By dragging and dropping pre-created SQL structures onto Builder/Delphi forms.

Once the structures have been created, they may be accessible from within the components themselves. To do this, right-click on your Data Access Component and choose "Explore." This will open the SQL Explorer.

Next, navigate and open the database alias you are working with. You should be presented with a list of whatever structure groups are currently available. Within these are the structures themselves. Find the specific structure you would like to implement and drag it over to your Builder/Delphi form and drop it. A component will be created and its properties filled in with the applicable data. This is much easier than typing in the properties by hand, and is an efficient way to eliminate implementation errors that could occur.

Last Modified: 24-AUG-00

## Article #22985: Error "Unable to write to registry" during Interbase 5.5 install.

### Question:

Why did my Interbase 5.5 installation stop around 10% with the error 'unable to write to registry'?

### Answer:

Another possible error message is "Failed to open registry key". Here are 3 possible solutions:

1) InterBase 5.5 expecting the following registry key to exist and it has not yet been created:

HKEY\_LOCAL\_MACHINE-->Software-->Microsoft-->Windows-->CurrentVersion-->SharedDLLs

2) Other software has added enough values to this key to hit some OS limit to the number of values that can be registered for this key. Check to see if there are extraneous or duplicate values under this key and delete them.

3) A path length over 128 characters can cause problems when installing Interbase but it usually gives a different error message (see the file Install.txt which comes with Delphi for more information).

Last Modified: 11-OCT-00

## Article #23006: Passing nil to a variable parameter.

### QUESTION:

How do I pass nil to a variable parameter of a method?

### ANSWER:

Create the pointer type, assign it nil and pass it in by dereferencing it.

```
var
  value: PInt;
begin
  value := nil;
  NextVal(value^);
end;
```

Now to use the passed in parameter you will have to check for nil first by getting the address of the Integer.

```
function NextVal(var value: Integer): HRESULT; stdcall;
begin
  if @value <> nil then
    ShowMessage('value ' + IntToStr(value))
  else
    ShowMessage('passed nil');

  Result := 0;
end;
```

Last Modified: 15-SEP-00

**Article #23007: How to access more than one table with the QuickRep component.**

**QUESTION:**

I have more than table that I need my QuickRep component to access, what other Dataset can I use?

**ANSWER:**

The QuickRep component can only be connected to one dataset at a time. If you have data from more than one table that needs to be accessible to the QuickRep component, use a TQuery with a join statement.

Last Modified: 15-SEP-00

## Article #23008: Variable alignment problems in record type.

### Question:

Why am I getting garbage returned from my record variables?

### Answer:

This is most likely a problem with field alignment. If you are using an application in Delphi 5 that was migrated from an earlier version you should know that alignment changed in Delphi 5. See Delphi 5 help under *Contents / Upgrading to Delphi 5 / Compatibility issues / Default alignment*.

If this is not your issue there is also a bug in Delphi 5 you should be aware of. If no compiler directive is specified the default is `{ $ALIGN ON }` or simply `{ $A+ }`. Aligned means the record is unpacked and each field takes a fixed amount of space in memory based on the datatype. Packing is done to save memory.

The bug in Delphi can cause packing or inconsistent alignment to occur even when alignment is on, `{ $A+ }`. It occurs when multiple record fields are declared in the same line. Moreover, this alignment is not consistent, varying between 2, 4 and 6.

The solution is easy; declare each field in a new line. Here's an example that demonstrates the issue with Extended types. Since they are all Extended types they should all use the same amount of space in memory. Therefore the difference between each of the memory addresses should all be the same, but notice they are not. We've only observed this behavior with Extended and Real48 types. We've heard reports of it happening with other data types but have not been able to reproduce it.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  test1 : record
    field1, field2 : Extended;    // Packed
    field3 : Extended;          // Not packed
    field4 : Extended;          // Not packed
  end;
begin
  showmessage('Address of field1: ' + IntToStr(longint(@test1.field1))
+ #10#13
+ 'Address of field2: ' + IntToStr(longint(@test1.field2))
+ #10#13
+ 'Address of field3: ' + IntToStr(longint(@test1.field3))
+ #10#13
+ 'Address of field4: ' + IntToStr(longint(@test1.field4)));
end;
```

Last Modified: 11-OCT-00

## Article #23009: Auto Increment Fields in a ClientDataSet

### QUESTION:

How can I auto increment my field in a dataset using TClientDataSet?

### ANSWER:

If you are using a TClientDataSet to create local data set, which will be saved locally to disk, then there are a couple of ways to automatically increment a field. The first technique requires the use of an aggregate.

Examine the code below:

```
.
.
var
    NewID: Integer;
begin
    //Assuming that your table has been created
    with ClientDataSet do
    begin
        AggregatesActive := False;
        Aggregates.Clear;
        with Aggregates.Add do
        begin
            Expression := 'Max(AField)';
            AggregateName := 'Runtime';
            Active := True;
        end;

        AggregatesActive := True;

    try
        NewID := Aggregates[0].Value;
        // ...or alternatively
        // NewID := Aggregates.Find('Runtime').Value;
        Inc(NewID);
    except
        on E: Exception do SomeHandle; //Probably shouldn't throw exception
    end;

    // Next set field value to NewID
    Insert;
    FieldByName('AField').AsInteger := NewID;
end;
```



In this example we first deactivate any aggregates, as well as clearing them. Clearing them is optional, if you choose not to clear then be sure to use the proper index value when indexing into `Aggregates[]`. When we add the new aggregate we set the `Expression` to `Max('AField')`, give it a name, and activate it. This aggregate will keep track of the maximum value in your auto incremented field. So when you insert a new record you can set the value of the auto incremented field to the current value of the aggregate, add 1.

The second technique of auto incrementing requires changing the current index of the `TClientDataSet`. The first thing to do is to create an `Index` with the value of its `Fields` property set to the field name for which you want to auto increment. Once the index is created you need to set it to the `ClientDataSet.IndexName` property. This will cause the resulting data set to order itself in descending order based on the values in the column. You can then call `Last()` to move the cursor to the end of this data set. Next, get the value out of the field and add 1. This will be the value of your next of the next record's auto increment field.

Last Modified: 08-SEP-00

## Article #23011: Using the requery method in place of the refresh method.

### QUESTION:

How can I speed up ADO's slow refresh for my table?

### ANSWER:

Use the requery method, this will update your tables faster. Call requery to refresh the recordset by re-executing the original command or SQL statement that produced the record set. Here is an example on how to implement requery:

```
procedure TFrmListOpen.TableRefresh;
var
  SavePlace : TBookmark;
begin
  {Using a standard refresh takes too long.  This code will perform a }
  {refresh much quicker.}
  SavePlace := DM.TableOpenEvents.GetBookmark;
  DM.TableOpenEvents.Requery;
  DM.TableOpenEvents.GotoBookMark(SavePlace);
  DM.TableOpenEvents.FreeBookMark(SavePlace);
end;
```

Last Modified: 08-SEP-00

## Article #23012: Checking the BDE version programmatically.

### QUESTION:

How can I check for the BDE version programmatically?

### ANSWER:

Your application might need a certain version of the BDE to run properly. So you will want to make sure that version is installed. Here is a code sample that shows you how to check which version of the BDE is installed.

```
uses
  Forms,
  DB,
  BDE,
  Dialogs,
  SysUtils;

{$R *.RES}

var
  ThisVersion : SYSVersion;
begin
  DbtGetSysVersion(ThisVersion);
  ShowMessage('BORLAND DATABASE ENGINE VERSION = ' + IntToStr(ThisVersion.iVersion));
end.
```

Last Modified: 15-SEP-00

## Article #23175: Finding all files (or files of a certain type) in a given directory

### Question:

Given a directory name, how do I find all files in that directory (or files of a specific type, such as \*.jpg)? How to I optionally search through all subdirectories?

### Answer:

You can use FindFirst and FindNext functions in SysUtils. Here is an example:

```
{ procedure GetFiles -
  Parameters:
    ADirectory: string; // A directory to search for files
    Files: TStringList; // The list of files to put the result in
    SubFolders: Boolean; // If True, it searches subfolders too }

procedure GetFiles(const ADirectory: string; Files: TStringList;
SubFolders: Boolean);
    // Helper function to remove any slashes or add them if needed
function SlashSep(const Path, S: string): string;
begin
    if AnsiLastChar(Path) ^ <> '\' then
        Result := Path + '\' + S
    else
        Result := Path + S;
    end;
var
    SearchRec: TSearchRec;
    nStatus: Integer;
begin
    // First find all the files in the current directory
    // You could put something else instead of *.* , such as *.jpeg or *.
gif
    // to find only files of those types.
    nStatus := FindFirst(PChar(SlashSep(ADirectory, '*.*')), 0,
SearchRec);
    while nStatus = 0 do
        begin
            Files.Add(SlashSep(ADirectory, SearchRec.Name));
            nStatus := FindNext(SearchRec);
        end;
    FindClose(SearchRec);
    // Next look for subfolders and search them if required to do so
```

```

if SubFolders then
begin
    nStatus := FindFirst(PChar(SlashSep(ADirectory, '*.*')),
faDirectory,
    SearchRec);
    while nStatus = 0 do
    begin
        // If it is a directory, then use recursion
        if ((SearchRec.Attr and faDirectory) <> 0) then
        begin
            if ( (SearchRec.Name <> '.') and (SearchRec.Name <> '..') )
then
                GetFiles(SlashSep(ADirectory, SearchRec.Name), Files,
SubFolders);
            end;
            nStatus := FindNext(SearchRec)
        end;
        FindClose(SearchRec);
    end;
end;

```

[Click here to download the source and sample program from Code Central.](#)

**NOTE:** Delphi 4's FindClose function has a bug in it and can make this code run in an infinite loop. You will need to change the FindClose function in SysUtils.pas to look like this:

```

procedure FindClose(var F: TSearchRec);
begin
    if F.FindHandle <> INVALID_HANDLE_VALUE then
    begin
        Windows.FindClose(F.FindHandle);
        F.FindHandle := INVALID_HANDLE_VALUE;
    end;
end;

```

Last Modified: 07-NOV-00

## Article #25128: Creating a BDE alias programmatically from Delphi

Problem:

Need to create a BDE alias programmatically

Solution:

When you create a delphi project and include the DB unit it creates an instance of a TSession component called Session. This component has methods to allow the creation of a BDE alias.

If your application includes any of the database components the DB unit is automatically included in the project. If you do NOT use any of the database components you will have to manually add the DB unit to have access to the Session variable. Alternatively you could add your own TSession variable to your project. (TSession is a component in the Data Access tab of the component palette).

The following is a code fragment from an program that creates a new BDE alias:

----- Begin Code Fragment -----

```
{ procedure to create new BDE alias }
{ it creates an InterBase alias with the specified }
{ database and username }
{ All BDE parameters not specified will use the defaults }

procedure TForm1.Button1Click(Sender: TObject);
var
    AParams: TStringList;
begin
    AParams := TStringList.Create;

    AParams.add('Server Name=' + 'lurch:d:c:ibtmprestored.gdb');
    AParams.add('User Name=' + 'SYSDBA');
    Session.AddAlias('NewAlias', 'INTRBASE', AParams);
```

```
Session.SaveConfigFile;  
AParams.Free;
```

```
end;
```

Last Modified: 27-SEP-00

## Article #25183: Handling the "Capability not supported" error

### Problem:

An application written using the BDE, Delphi, or C++ Builder may return an error "Capability not supported."

### Solution:

This error is returned by the BDE when the BDE parses an SQL string to be sent to a server and the syntax of the string is not supported by the BDE.

See below for more information on how the BDE sends pass through SQL strings to DBMS servers and ODBC data sources.

The BDE, and Delphi and C++ Builder applications that use data-aware components send SQL strings to DBMS servers and ODBC data sources depending on the setting of the SQLQRYMODE parameters.

The settings of the SQLQRYMODE parameter are listed in the BDE Administrator Help (formerly known as the BDE Configuration Help) file, and are paraphrased below:

- The SQL string is sent to the server or ODBC data source first. If an error is returned the string is then parsed and ran by the BDE.

SERVER - The SQL string is sent to the server or ODBC data source, if an error is returned then the error is placed on the BDE error stack and the appropriate BDE error is returned.

LOCAL - The SQL string is parsed and ran by the BDE.

When the BDE parses and runs the SQL string due to the SQLQRYMODE being set to blank or LOCAL, if the SQL string to be parsed is not supported by the BDE the error "Capability not supported" is returned.

The SQLQRYMODE parameter is found in both BDE alias definitions and in BDE driver definitions. The BDE will look for this parameter first in the alias definition, if it is not found or the entry not a recognized value or is corrupted then the BDE will



look at this parameter in the driver definition. Alias definitions can be created by using TDatabase component, by using the DbOpenDatabase() and DbAddAlias() BDE API calls, and by using the BDE Administrator (formerly the BDE Configuration) utility.

Last Modified: 29-SEP-00

## Article #25187: An EDatabase Exception is generated by a Delphi or C++ Builder application

### Problem:

A Delphi generated application is generating an EDatabase exception.

### Solution:

An exception was generated in the Delphi application code, or in the Delphi VCL Source code.

To locate the cause of the exception do the following steps from within the Delphi development environment:

Set the "Break on exception" checkbox and the "Integrated debugging" checkbox in the Tools | Options | Preferences menu to checked. In Delphi 1.0 the menu choices are: Option | Environment | Preferences. Run the "Build All" in the Project menu to generate debug information. Run the application and repeat the steps that produced the EDatabase exception.

When the error occurs and the cause is not obvious from within the application source code it is possible to go inside the VCL source code and have the debugger stop the application on the line of the VCL source that generates the EDatabase exception. To do this requires having the Enterprise or Client/Server edition of Delphi or C++ Builder.

If these versions of are not owned, obtaining the source code to the VCL components requires upgrading to the versions of Delphi mentioned above, or purchasing the RTL source code. To do either of the two choices consult the Borland Website at [www.borland.com](http://www.borland.com)

for the Borland Customer Service phone number.

To use the VCL source code to further track down the source of the EDatabase exception do the following:

Look up the help topic for the component that is generating the error. On this page in the upper left is a line that says "Unit". On the line underneath is the source file that contains the VCL source. Copy the source file noted on the help page from the Program Files\Delphi\Source\Vcl directory to the application directory. Copy the files db.pas and dbtables.pas as well, since these files are used extensively by the TDatabase, TTable and TQuery components. Repeat the steps above for reproducing the exception.

Last Modified: 29-SEP-00

## Article #25188: Handling BDE Engine Errors

### Problem:

A Delphi or C++ Builder generated application produces a BDE Engine exception.

### Solution:

An exception was produced in the Delphi generated application due to an error from the RDBMS server (i.e InterBase), the BDE or an ODBC driver.

Using the Borland BDE Configuration Utility (BDECFG.EXE, or BDEADMIN.EXE)

to examine the SQLQRYMODE property of the InterBase or Visigneic ODBC alias being used in the application. If the setting is or set to LOCAL change the setting to SERVER. Save the changes and exit the utility.

If the application is using a TDatabase component, bring the application up in Delphi and left double-click the mouse on the TDatabase component to bring up the Database Login dialog. Change the SQLQRYMODE property in the parameters list box to SERVER, if the property is present. Save and do a Build All.

At this point exiting the Delphi IDE and starting it again may be required so that the new changes in the BDE aliases will be re-read and accounted for.

### SQLQRYMODE

#### Settings

=====

: The BDE sends the SQL query string to the RDBMS server or the ODBC data

source and if an error is returned the BDE will then parse the string itself and

will send the parsed string (or strings) to the server.

LOCAL: The BDE will parse the strings and send the parsed string (or strings) to the

server.

SERVER: The BDE will send the SQL query string to the RDBMS SERVER or the ODBC datasource. If an error occurs then this will be returned through the BDE.

By changing the setting to SERVER the error returned will be in direct relation to the SQL query that was requested by the Delphi application. Next, run the application and try to reproduce the exception.

When the exception is reproduced the BDE will take the resulting error from the server and will map it through one of its existing error codes and messages. If the error is "General SQL Error: " this indicates that an error occurred on the server and the BDE did not have a comparable or corresponding error message to map it to so it returned this error. The

portion will be the error string that is returned from the server (or ODBC datasource). This is the easy case, match the error string with the errors listed in the InterBase Language Reference, or with a listing of ODBC errors.

If another error is produced then it is uncertain if the error is from the server, the BDE, or an ODBC driver, this is the not-so-easy case. Delphi provides a way to look at where the error is coming from, follow the instructions in the following paragraphs.

To further track down the source of the exception start by placing a memo on the main form. This memo will be used to display further information that is available in the BDE Engine Exception object. Next implement the following code snippet into the main form of the application:

```
unit dbughelp;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, DB;
```

```
type
```

```
  TForm1 = class(TForm)
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure MyHandler(Sender: TObject; E:Exception);
  end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := MyHandler;
end;
```

```
procedure TForm1.MyHandler(Sender: TObject; E: Exception);
```

```
var
```

```
  i: Word;
  F: EBDEEngineError;
```

```
begin
```

```
  Memo1.Lines.Add('== Begin Entries for Exceptions ==');
  Memo1.Lines.Add(E.ClassName);
  if (E is EBDEEngineError) then begin
    F := EBDEEngineError(E);
    Memo1.Lines.Add('ErrorCount = ' + IntToStr(F.ErrorCount));
    for i := 0 to F.ErrorCount - 1 do begin
      Memo1.Lines.Add(' ');
      Memo1.Lines.Add('-----');
      Memo1.Lines.Add(' ');
      Memo1.Lines.Add('NativeError = ' + IntToStr(F.Errors[i].
NativeError));
      Memo1.Lines.Add('Message = ' + F.Errors[i].Message);
      Memo1.Lines.Add('ErrorCode = ' + IntToStr(F.Errors[i].
```

```
ErrorCode));  
    end;  
end;  
Memo1.Lines.Add('== End Entries for Exceptions ==');  
end;
```

end.

In the FormCreate procedure (i.e OnCreate event) the application global exception handler is re-routed to a custom exception handler, the Myhandler procedure. Within the MyHandler procedure the error information in the BDE Engine Exception object is placed in the memo component. After the BDE Engine Exception information is placed in the memo look for the NativeError codes. The errors will follow a specific pattern for InterBase , the BDE and for ODBC. The patterns of the error messages are:

InterBase: The errors will follow the pattern of -xxx, like a -902 error.

Use the InterBase Language Reference manual to look up the error and the error message.

BDE: Download the DBI Error checking utility from the BDE Technical Support pages of the [www.borland.com](http://www.borland.com) website. Run the utility and type in the error number, the corresponding error message will be returned.

ODBC: Try consulting an ODBC reference for the error code.

Last Modified: 29-SEP-00

## Article #25207: Automatically Generating Data to Create Large Tables Using Delphi

### Problem:

In testing applications, sometimes a large table is desired for a variety of reasons.

Some of the reasons might be to test performance, backing up and restoring sets of data, and reporting statistical data. Creating a large table by hand, or by several persons can be a time-consuming task that can be automated and performed quickly by using the approach given below.

### Solution:

The approach given below uses Delphi to connect to the InterBase server, however it could easily be modified for any database.

Before starting to construct the program that will generate records, if they are to be viewed in the Delphi generated application, run the BDE Configuration Utility. On the InterBase alias that will be used change the Enable BCD property to TRUE. This step not required but is recommended for tables with NUMERIC and DECIMAL columns. BDE will treat NUMERIC and DECIMAL columns differently based upon the setting of this property, the chart below will illustrate how BDE treats these columns:

	ENABE BCD is FALSE	ENABLE BCD is TRUE
NUMERIC(4)	TSmallIntField	TSmallIntField
NUMERIC(9)	TIntegerField	TBCDField
NUMERIC(10)	TFloatField	TBCDField
DECIMAL(4)	TIntegerField	TBCDField
DECIMAL(9)	TIntegerField	TBCDField
DECIMAL(10)	TFloatField	TBCDField

The ENABLE BCD setting affects how the BDE retrieves data from



InterBase by use of a SELECT statement. When the columns are translated into the TSmallIntField and TIntegerField types the decimal data is truncated, otherwise if the columns are translated to TFloatField or TBCDField types then the whole and decimal values are retained. The ENABLE BCD setting has no bearing on the INSERT statement that will be used later on to insert data into the table.

In the application the Interface section of the .pas will look similar to:

```
unit gendatal;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  StdCtrls, DB, DBTables;

type
  TForm1 = class(TForm)
    Query1: TQuery;
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    procedure gen_alpha      (var in_string:   String; in_length: Word);

    procedure gen_smallint  (var in_smallint: Smallint);
    procedure gen_integer   (var in_integer:  Integer);
    procedure gen_float     (var in_float:    Single);
    procedure gen_double    (var in_double:   Double);
    procedure gen_decimal   (var in_decimal:  Double; precision,
      scale: Word);
    procedure gen_integer_range (var in_integer: Integer; in_low,
      in_high: Integer);
    procedure gen_date_1900   (var in_date:   TDateTime);

    function  gen_rand: Extended;
  end;
end;
```

```
public
end;
```

```
var
  Form1: TForm1;
```

The application is divided up into several functions that perform the generation of fictitious data.

The functions are as follows:

```
// Generate CHAR and VARCHAR data for strings that are from 1 to
// 255 characters in width
procedure gen_alpha      (var in_string:   String; in_length: Word);

// Generate a SMALLINT value
procedure gen_smallint  (var in_smallint: Smallint);

// Generate an INTEGER value
procedure gen_integer   (var in_integer:  Integer);

// Generate a FLOAT value
procedure gen_float     (var in_float:    Single);

// Generate a DOUBLE PRECISION value
procedure gen_double    (var in_double:   Double);

// Generate a date in the 1900's range (i.e 20th Century)
procedure gen_date_1900(var in_date:     TDateTime);

// Generate a DECIMAL or a NUMERIC value
procedure gen_decimal   (var in_decimal:  Double; precision,
  scale: Word);

// Generate a value used by the procedures gen_integer, gen_float,
// gen_double, and gen_decimal
function  gen_rand: Extended;

// Generate a value used by gen_decimal, and gen_1900_date
procedure gen_integer_range (var in_integer: Integer; in_low,
  in_high: Integer);
```

The functions used in the interface of this application and the TQuery component that is used to

insert the data into the table will be covered later. The code for each procedure is shown below:

In this procedure an array of possible characters to include in the character string is constructed private to this procedure. In this array the blank space character is available twice, at the beginning and the end, and the comma character is available. What is not included here is the period character and other punctuation character, include these punctuation characters as desired. Please note that in Delphi 1.0 (16-bit) strings are limited to 255 characters. In Delphi v3.0 and higher, (32-bit) strings can be up to 4Gig in size, though InterBase supports CHAR and VARCHAR types only up to 32,767 (32k) bytes in length.

```
procedure TForm1.gen_alpha(var in_string: String; in_length: Word);
const
  char_array_size: Word = 55;
  char_array: Array[0..54] of Char =
    (' ', ' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
     'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
     'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
     'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
     'y', 'z', ' ');
var
  i, j: Word;
begin
  // CHAR and VARCHAR types cannot exceed 32767, these lines of
  // code ensure this limit is not exceeded.
  if (in_length > 32767) then
    in_length := 32767;

  j := Random(in_length - 1) + 1;
  SetLength(in_string, j);
  for i := 1 to j do
    in_string[i] := char_array[Random(char_array_size)];
end;

procedure TForm1.gen_smallint(var in_smallint: Smallint);
begin
  // This calculation provides a range of
  // -32,768 to 32,767, the range of a smallint
```

```
in_smallint := 32767 - Trunc(65536 * gen_rand);  
end;
```

In generating a random integer value and extended variable is used internally in the procedure.

An extended variable is used because the internal algorithm uses a value that ranges from 0 to 4,294,967,295 and integer values above 2147483647 are not supported in Delphi (i.e. Delphi does not support unsigned long values).

```
procedure TForm1.gen_integer(var in_integer: Integer);  
var  
    temp_extended: Extended;  
begin  
    // This calculation provides a range of  
    // -2,147,483,648 to 2,147,483,647, the range of a integer  
  
    temp_extended := 4294967295.0 * gen_rand;  
    if temp_extended <= 2147483647 then  
        in_integer := Integer(2147483647 - Trunc(temp_extended))  
    else  
        in_integer := -1 * Integer(Trunc(temp_extended - 2147483647));  
end;
```

In generating a random float value the format of the float value as defined in the Object Pascal Language Guide is used to construct the float value. Specifically, the s, e, and f components are separately obtained and then placed in an integer variable (which is of the same size as a float), and then once this is done the value is copied into the float. Please note in constructing a float value it is possible to create a NAN (not a number), infinity, and minus infinity values. The procedure below does not create these values, if it did this would raise an exception in Delphi.

```
procedure TForm1.gen_float(var in_float: Single);  
var  
    s, e:      Byte;  
    f:        Longint;  
    temp_integer: Integer;
```

```

begin
  temp_integer := 0;

  // Obtain values for the component elements of in_float
  s := Trunc(2* gen_rand);
  e := Trunc(255 * gen_rand);

  f := Trunc(8388608 * gen_rand);

  // Place the components in the proper location in in_float
  temp_integer := temp_integer or s;
  temp_integer := temp_integer shl 8;
  temp_integer := temp_integer or e;
  temp_integer := temp_integer shl 23;
  temp_integer := temp_integer or f;

  // Move the data into the float variable
  Move(temp_integer, in_float, sizeof(temp_integer));
end;

```

In generating random double precision values the format double precision type that is defined in the Object Pascal Language Guide is used. Specifically the s, e, and f components are separately generated. Since a double precision value is 64-bits wide two long integers, which are 32-bits in length, are used to hold the values during computation of the double precision value. To this end the s, e, and 20 bits of the f components are held in one integer, and the remaining 32-bits of f are held in the other integer. In the final stages the contents of the two integers are copied to the correct locations inside the double precision number to produce a random value. Please note in constructing a double precision value it is possible to create a NAN (not a number), infinity, and minus infinity values. The procedure below does not create these values, if it did this would raise an exception in Delphi.

```

procedure TForm1.gen_double(var in_double: Double);
var
  e, s: Word;

```

```

f_low, low_integer, hi_integer: Integer;
temp_char_array: array[0..7] of char;
begin

    low_integer := 0;
    hi_integer := 0;

    // Get s component of double
    s := Trunc(2 * gen_rand);
    low_integer := low_integer or s;

    // Get e component of double
    low_integer := low_integer shl 11;
    e := Trunc(2046 * gen_rand);
    low_integer := low_integer or e;

    // Low 20 bits of f component of double
    low_integer := low_integer shl 20;
    f_low := Trunc(2097152 * gen_rand);
    low_integer := low_integer or f_low;

    // High 32 bits of f component of double
    gen_integer(hi_integer);

    Move(hi_integer, temp_char_array[0], 4);
    Move(low_integer, temp_char_array[4], 4);
    Move(temp_char_array, in_double, 8);
end;

```

```

procedure TForm1.gen_date_1900      (var in_date: TDateTime);
var
    days, millisecs: Integer;
begin
    gen_integer_range(days, Integer(0), Integer(36524));
    gen_integer_range(millisecs, 1, 99999);

    in_date := StrToFloat(FormatFloat('00000', days) + '.' +
        FormatFloat('00000', millisecs));
end;

```

In this procedure the range array is used to signify the range of numbers to generate using the `gen_integer_range` procedure when generating integers within a certain

range. The double\_format array is used to indicate a specific string formatting used in the algorithm used to generate the DECIMAL or NUMERIC value.

```
procedure TForm1.gen_decimal (var in_decimal: Double; precision,
scale: Word);
const
  range: array[1..9] of Integer = (9, 99, 999, 9999, 99999, 999999,
9999999,
                                99999999, 999999999);
  double_format: array[1..16] of String =
    ('0', '00', '000', '0000', '00000', '000000', '0000000',
'000000000',
    '0000000000', '00000000000', '000000000000', '0000000000000',
'000000000000000',
    '0000000000000000', '00000000000000000', '000000000000000000');
var
  whole, decimal:      Word;
  whole_int, decimal_int: Integer;
  whole_double, decimal_double: Double;
begin
  if ((precision > 15) or (precision < 1) or (scale > 15) or (scale <
1)) then begin
    in_decimal := 0.0;
    exit;
  end;

  if (scale > precision) then begin
    in_decimal := 0.0;
    exit;
  end;

  whole := precision - scale;
  decimal := scale;

  // generate whole number portion
  if (whole = 0) then
    whole_int := 0
  else begin
    if (whole <= 9) then begin
      gen_integer_range(whole_int, -1 * range[whole], range[whole]);
      whole_double := whole_int;
    end
    else begin
```

```

    gen_integer_range(whole_int, -1 * range[9], range[9]);
    whole_double := Int(whole_int * exp((whole - 9)*ln(10)));
    gen_integer_range(whole_int, 0, (whole - 9));
    whole_double := whole_double + whole_int;
end;
end;

// generate decimal number portion
if (decimal = 0) then
    decimal_int := 0
else begin
    if (decimal <= 9) then begin
        gen_integer_range(decimal_int, 0, range[decimal]);
        decimal_double := decimal_int;
    end
    else begin
        gen_integer_range(decimal_int, 0, range[9]);
        decimal_double := Int(decimal_int * exp((decimal - 9)*ln(10)));
        gen_integer_range(decimal_int, 0, (decimal - 9));
        decimal_double := decimal_double + decimal_int;
    end;
end;
end;

in_decimal := StrToFloat(FormatFloat(double_format[whole+1],
whole_double) +
    '.' + FormatFloat(double_format[decimal], decimal_double));
end;

```

This procedure is used internally by other procedures that require generating a random integer value within a certain range.

```

procedure TForm1.gen_integer_range (var in_integer: Integer; in_low,
in_high: Integer);
var
    low, hi, delta: Integer;
begin
    if (in_low <= in_high) then begin
        low := in_low;
        hi := in_high;
    end
    else begin
        low := in_high;
        hi := in_low;
    end;
end;

```



```

end;

if ((hi - low) > 2147483647) then begin
    in_integer := 0;
    Exit;
end;

delta := hi - low + 1;
in_integer := Trunc(delta * gen_rand) + low;
end;

```

Place each procedure and function shown above in the Implementation section of the .pas file. Next, place a TQuery, two TEdit, and a TButton Component on the form. Give the TQuery component focus on the form and in the Object Inspector make the entry in the SQL property similar to the following entry:

```

INSERT INTO InputTble
(Col1, Col2, Col3)
VALUES
(:parm1, :parm2, :parm3)

```

The names of the columns and their data type in the table depend on the structure of the table. Next, set data type of the parm variables in the Params property. The InterBase data types will match up with the following BDE data types:

InterBase Type:	BDE Type:
=====	=====
CHAR	String
VARCHAR           String	
SMALLINT	Smallint
INTEGER	Integer
FLOAT	Float
DOUBLE PRECISION	Float
NUMERIC           Float	
DECIMAL	Float
DATE	Date

One data type is missing here, the BLOB data type. BDE v3.5 and previous does not accept blob parameters to pass to InterBase BLOB columns. This is also why there

is no procedure the  
generate random BLOB data.

For the TButton component set up an OnClick event and have the OnClick event look like:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:          Longint;
  in_string:  String;
  in_integer: Integer;
  in_decimal86: Double;
begin
  in_integer := 0;
  in_decimal := 0;
  Databasel.StartTransaction;
  try
    for i := 0 to StrToInt(Edit1.Text) - 1 do begin
      gen_alpha(in_string, 128);
      gen_integer(in_integer);
      gen_decimal(in_decimal86, 8, 6);
      Query1.ParamByName('parm1').AsString := in_string;
      Query1.ParamByName('parm2').AsInteger := in_integer;
      Query1.ParamByName('parm3').AsFloat := in_decimal86;
      Query1.ExecSQL;
      Edit2.Text := IntToStr(i);
      Application.ProcessMessages;
    end;
  except
    else Databasel.Rollback;
  end;
  if Databasel.InTransaction then
    Databasel.Commit;
end;
```

Using the Object Inspector, set up an OnCreate and an OnClose event for the form and have them look similar to:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Start up the random number generator in the runtime library
  Randomize;
end;
```

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    // Free the form upon closing
    Action := caFree;
end;
```

When running the application type the number of records that are desired to be generated in the Edit1 control and click on the Button12 control to start the process of adding records.

The Edit2 control will display the number of records added.

Last Modified: 02-OCT-00

## Article #25213: Handling "General SQL error: violation of FOREIGN KEY constraint . . ." error

### Problem:

When using a master-detail relationship the error "General SQL error: violation of FOREIGN KEY constraint . . ." error messages appear when I try to simultaneously post records in the master and detail table.

### Solution:

The VCL code posts the records to the detail table first, then posts the records to the master table.

This is done for local tables (dBASE, Paradox, and ASCII), ODBC data sources, InterBase, Oracle, and other supported RDBMS servers. In posting the details records first, since the master record is not yet in the table, an error is generated on the server and passed to the BDE. In case of local tables the error is generated in the BDE.

One easy way of overcoming the exception is to do the following:

- In the BeforeEdit event of the detail TTable or TQuery object place the following lines of code:

```
// If using TQuery object replace "TTable" with "TQuery"

if ((DataSet as TTable).MasterSource.State = dsInsert) then
    (TTable(DataSet).MasterSource as TTable).Post;
```

What this will do is that when record data is typed in on the detail table, the detail table will switch its state into dsInsert mode. The BeforeInsert event will fire off the above code that will place the new master record in the master table. If the new record in the detail table is aborted and the new record remains in the master table. What will happen in the application is that the new master is still added first (no change here), the only difference is that the new master and

detail records are not added at the click of a "Post" button or other such triggering UI component.

Two other alternatives that are more difficult to implement are suggested below:

- Remove the foreign key constraint and require that the client applications check for the key in the master table.
- Do not use the MasterSource and MasterFields properties in the TTable and TQuery objects and manage the master-detail relationship in the application code.

Last Modified: 02-OCT-00

## Article #25223: Converting Data in Paradox Tables to InterBase Using Delphi

### Problem:

I would like to transfer data from Paradox tables to InterBase. I do not want to use SQL to perform their conversion from Paradox to InterBase. What are my options?

### Solution:

One option is to use database desktop's Tools| Utilities | Copy menu choices to convert the Paradox table to InterBase.

Another option is to use the Data Pump.

A 3rd option is to export the table into ASCII and use SQL for the conversion manually. Note that this option will not bring in blob fields.

Note: While these steps have been written with Paradox and InterBase tables in mind, they will work for any database.

Last Modified: 02-OCT-00

## Article #25238: How to retrieve an InterBase Blob in Delphi using SaveToFile?

### Problem:

This is an example of how to retrieve an InterBase Blob in Delphi.

### Note:

While this example was designed specifically for Interbase, it can easily be made to apply to any database.

### Solution:

IB V4.x

Delphi 3.0

There are several ways of retrieving InterBase Blob data within Delphi.

This example uses the SaveToFile method.

- \* Put TDatabase, TQuery, TDataSource and TDBGrid on the form.
- \* set the TDatabase's fields as follow:
  - > Alias name to the BDE Alias.
  - > DatabaseName to something you wish.
  - > Connected to true
- \* Set the TQuery fields as follow:
  - > SQL to the sql statement to be executed, ie select \* from table1
  - > DatabaseName to the same name you've named in the previous step.
- \* Set the TDataSource field DataSet to the name of the Tquery component. Default is query1.
- \* Set the TDBGrid field DataSource to the name of the TDataSource component. Default is Datasource1.
- \* Go back to TQuery component and double click.
- \* Right click on the blank space on the windows said "form1.query". And choose add fields.
- \* Select all the fields in the list and added.
- \* Drop a TButton on the form.
- \* Double click on the Tbutton.
- \* Add the following lines between begin and end;

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
querylimage.savetofile('c:testimage.jpg');
end;
```

Note: image is the field name defined as blob.  
And c:testimage.jpg is where the location of the  
image file to be created.

- \* Save the project/form/unit. and Run.
- \* Click on the Button will retrieve a blob entry and create file  
image.jpg.

Last Modified: 02-OCT-00



## Article #25249: Calling an IB stored procedure using TStoredProc

### Problem:

Looking for an example of how to call an InterBase stored procedure using Delphi's TStoredProc component.

### Solution:

The information in this article applies to:

- \* InterBase v4.x
- \* InterBase v5.x
- \* Delphi 1.x, 2.x, 3.x, 4.x

Here is an example of how to call an InterBase stored procedure using Delphi's TStoredProc component.

The stored procedure retrieves and returns the next value from a generator. You must create this stored procedure and generator in the database.

### Creating the generator:

```
=====
create generator cust_no_gen;
```

### Here is the InterBase Stored procedure:

```
=====
create procedure procl returns(e_no integer)
as begin
    e_no=gen_id(cust_no_gen,1);
end
```

-----  
-----  

### Here are the steps to accomplish the task in Delphi:

- \* Put TDatabase, TStoredProc, TButton, TLabel components on the form.
- \* Set the following properties for TDatabase component:
  - > AliasName to the BDE alias pointing to the database created
  - > Fill in DatabaseName to be used by other database components

> Connected to True

\* Set the following properties for TStoredProc component:

- > Databasename to the name you set for your TDatabase component
- > StoredProcName to the name of the stored procedure in the database (You can select from drop down list)

\* Double click on the TButton component and add the following code in between begin and end.

```
storedproc1.execproc;  
labell1.caption:=storedproc1.parambyname('e_no').asstring;
```

The result would look like this:

=====

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
storedproc1.execproc;  
labell1.caption:=storedproc1.parambyname('e_no').asstring;  
end;  
end.
```

- \* Save the project/form/unit and Run.
- \* Click on the Button and the next generated number will be displayed.

Some benefits of getting the gen id first before inserting it:

=====

- \* The next generated number will be visible, so it is known exactly what's inserted.
- \* Using trigger to call the generator in Delphi often results in error "Record Key Deleted". Calling the generator using TstoredProc is a workaround for this error.

Last Modified: 02-OCT-00

## Article #25276: Vendor Initialization Failed error when trying to start InterBase application

### Problem:

When an application tries to start it gets the error:

```
Vendor Initialization Failed
Idapi Service Library
```

And complains that it cannot find gds32.dll

### Solution:

Things to check are:

1) check Registry for InterBase entries:

```
HKEY_LOCAL_MACHINESOFTWAREBorlandInterBaseCurrentVersion
```

2) Make sure gds32.dll is in the propert location (It should be in the windowssystem

```
directory for win95 and the winntsystem32 directory for NT)
```

3) Make sure that the gds32.dll is the correct version for the installed kit.

The registry entries for InterBase will tell you the installed version. Make

sure that the gds32.dll has the same version. In win95 and NT4.0 you can

right mouse click on the gds32.dll file and select the properties menu option.

This will bring up a dialog which will tell you the version of the dll.

4) Make sure that the file msvcrt40.dll exists in the system directory. This file

is required for InterBase to operate. This file is installed by the operating

system on NT4.0, but not with NT 3.51 or win95. The InterBase installation

will install this file on these operating systems.

BDE specific suggestions:

-----

- 1) connect with wisql to see if InterBase is setup properly. If the wisql connection is successful that would suggest that the BDE is improperly configured.
  
- 2) Make sure that the database and InterBase files reside on local drives. InterBase will not operate when either the InterBase files or the database file reside on a mapped drive.

Last Modified: 26-OCT-00

## Article #25287: SQL Monitor

Problem:

What is SQL Monitor?

Solution:

SQL monitor is one of the tools shipped the Enterprise versions of C++ Builder and Delphi.

SQL monitor is used to display and debug the information sent through the

BDE. It allows you see what is actually happening behind the scenes and make sure that your BDE applications are being efficient.

If installed in the default location, SQL Monitor will be located in the (bin) directory of the Borland tool.

Last Modified: 11-OCT-00

## Article #25294: Insufficient memory to complete operation

### Problem:

Why am I getting the message insufficient memory to complete operation?

### Solution:

You may have hit one of the BDE fixed, but configurable, memory limitations.

Use the BDE Administrator (BDEADMIN.EXE) to increase MEMSIZE and SHAREDMEMSIZE settings.

You may also wish to re-evaluate the application hitting this limitation because it is likely that the application is a major resource hit on the machine.

Not using auto-create forms and opening tables and queries as needed can free up some of these resources.

Last Modified: 12-OCT-00

## Article #25337: When installing BDE based products they overwrite IB v5 gds32.dll with v4.2.1 dl

### Problem:

If you have installed and are running InterBase version 5.0 on Windows NT/95 and then install a BDE based product will over write the IB v5 gds32.dll with the IB v4.2.1 gds32.dll.

### Solution:

Until the BDE changes their installation programs to locate our software in its new registry location this will continue to happen.

### Default Registry Entries:

#### v4.21:

```
HKEY_LOCAL_MACHINE
  SOFTWARE
    Borland
      InterBase
```

#### v5.00:

```
HKEY_LOCAL_MACHINE
  SOFTWARE
    InterBase Corp
      InterBase
```

To avoid this problem you must:

- a) Safely shut down the InterBase server.
- b) Rename the gds32.dll to gds32.500 (or something you like)
- c) Install the BDE based product
- d) Rename the gds32.dll to gds32.421 (or something you like)
- e) Rename gds32.500 to gds32.dll
- f) Restart the InterBase server.

Last Modified: 17-OCT-00

## Article #25360: Setting the BDE DRIVER FLAGS parameter for individual aliases

### Problem:

How do I set the BDE DRIVER FLAGS parameter to different values for different aliases?

I have several BDE aliases defined and I want the DRIVER FLAGS parameter to be set differently for each of the BDE aliases.

### Solution:

There is NO way to set the DRIVER FLAGS parameter for individual BDE aliases.

The only way to set it is for the InterBase SQL Links driver. This change affects ALL BDE aliases created using this driver.

Last Modified: 18-OCT-00



## Article #25364: How to insert an InterBase BLOB in Delphi using LoadFromFile?

### Problem:

There is no example of how to insert an InterBase Blob in Delphi.

### Solution:

The information in this article applies to:

- \* InterBase v4.x
- \* InterBase v5.x
- \* Delphi all releases thru 4.x

This is one way of inserting BLOB in Delphi. There are several ways to accomplish this task. This example is just one way of doing it. This example is using the LoadFromFile method.

Create a table in InterBase with a field that can store non-text Blobs:

```
create table table1 (images blob sub_type 0);
```

- 
- \* Put TQuery and a Button on the form.
  - \* Set the TQuery's properties:
    - > Alias name to the BDE Alias pointing to the database.
    - > SQL to the sql statement to be executed,  
i.e. select images from table1
    - > RequestLive to True.
    - > Active to true.
  - \* Go back to TQuery component and double click which will bring up a window titled "form1.query".
  - \* Right click on the blank space of form1.query and choose add fields.
  - \* Select images in the list and add.
  - \* Double click on the TButton.
  - \* Add the following lines

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  query1.append;  
  query1images.loadfromfile('c:testimage.bmp');  
  query1.post;  
end;
```

Note: images is the field name defined as blob.

And c:testimage.bmp is where the location of the image file to be inserted.

- \* Save the project/form/unit and Run.
- \* Click on the Button will insert a blob entry.

To verify the Blob is inserted:

- \* Drop a Tdatasource and TDBImage component on the form.
- \* Set the properties of Tdatasource:
  - > Dataset to "query1"
- \* Set the properties of TDBImage:
  - >Datasource to "datasource1"
  - >Datafield to images
- \* Save the project and run. You should see the images displayed.

Note: By default, DBImage will only display bitmaps.

Last Modified: 18-OCT-00

## **Article #25382: Error installing from InterBase: String variable is not large enough for strin**

### Problem:

Trying to install InterBase from various products may result in the error " String variable is not large enough for string. Check the string declarations. Error: 401"

In most cases, this is a benign error. You can just click the OK box and continue. This error usually occurs when you have a very long path statement. In most cases, the only way to make it not occur is to shorten your path statement.

In some other conditions, checking the PATH value for the registry key: HK\_Current\_userenvironment and if it's a hex value delete it or convert it to a character string may resolve the problem as well.

Last Modified: 01-NOV-00

## Article #25392: Getting 'Error creating cursor handle' when using TStoredProc

### Problem:

When using TStoredProc and try to execute an InterBase stored procedure the error 'Error creating cursor handle' is generated.

### Solution:

This error is the message that accompanies an ENoResultSet exception thrown by the BDE. This error occurs, because the application tried to execute an SQL statement and retrieve a result set that did not exist.

When using TStoredProcs with InterBase execute procedures the application must use the TStoredProc.ExecProc method instead of the TStoredProc.Open method (or setting active property to true).

(See Data Definition Guide for more information on execute stored procedures)

Last Modified: 27-OCT-00

## Article #25433: Using BLOBs in Stored Procedures through the BDE

Problem:

Example using Blobs in Stored Procedures, then accessing them via BDE and Delphi

Solution:

Test case done with

IB V5.0  
Delphi 3.0  
BDE 4.51

```
----- IB CODE -----  
  
/* start procedure */  
set term ^^;  
  
create procedure t2_get_call_details_proc(call_no integer)  
returns (customer_no integer,  
        phone_no varchar(20),  
        ib_version char(20),  
        os_version varchar(20),  
        problem blob sub_type text segment size 80,  
        comment blob sub_type text segment size 100)  
as  
begin  
  
begin  
  for select customer_no,  
            phone_no,  
            gds_version,  
            operating_sys_version,  
            problem_description,  
            comment  
from call_logs  
where call_log_no = :call_no  
into :customer_no,  
      :phone_no,  
      :ib_version,  
      :os_version,  
      :problem,
```

```
      :comment
```

```
do  
suspend;  
end
```

```
end^^  
commit^^
```

```
set term ;^^  
/* end procedure */
```

```
commit;
```

```
----- DELPHI CODE -----
```

```
procedure TfrmMain.cmdGetDetailsClick(Sender: TObject);  
begin  
  // Close details  
  begin  
    qyGetDataCallLog_2.Close;  
    qyGetDataCallLog_2.UnPrepare;  
  end;  
  // Get Call Details  
  begin  
    db1.StartTransaction;  
    qyGetDataCallLog_2.SQL.Clear;  
    qyGetDataCallLog_2.SQL.Add  
      ('select * from t2_get_call_details_proc(:call_no);');  
    qyGetDataCallLog_2.Prepare;  
    qyGetDataCallLog_2.Params[0].AsInteger := StrToInt(edtCall_No.  
Text);  
    qyGetDataCallLog_2.Open;  
    db1.Commit;  
  end;  
end;
```

Last Modified: 18-OCT-00

## **Article #25435: Can a Delphi Application Connect to an InterBase Database on Any Platform?**

### Problem:

I want to develop a Delphi application using InterBase as a backend. I need it to be able to connect to an InterBase database on whatever operating system it happens to reside.

### Solution:

The information in this article applies to:

- \* All InterBase Versions

Yes, this is possible with a Delphi application using either the BDE/SQL Links or ODBC. Depending on how transportable you want the application to be, you may need to programmatically make it possible for the user to set up their own datasources.

The InterBase client library will connect to an InterBase database running on any supported platform. Because the BDE/SQLLinks driver uses the InterBase client, Delphi applications are able to connect to InterBase databases on any of the supported platforms.

Last Modified: 26-OCT-00

## Article #25493: How to connect to an InterBase database using the BDE API

### Problem:

Sometimes it is necessary to remove the VCL layer of code in a Delphi or C++ Builder application in order to track down a connection problem. To do this requires writing to the BDE API.

### Solution:

The following example below are some excerpts from a Delphi application that uses the BDE API to directly connect to an InterBase database. In order to run this example add the following units to the uses clause:

```
DB, DBTables, BDE
```

In the example the OnCreate event of the form is used to initialize the BDE engine, and to create a BDE alias in memory. The OnClose event shuts down the BDE engine. The OnClick event of a pushbutton is used to perform the connection to the database. If an error is returned by the BDE API function then the Check() function will generate an EDBEngineError exception and the appropriate error dialog will be shown.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DbiInit(nil);
  Check(DbiAddAlias(nil,
                   'TempIB',           // This is the name of the
in-memory alias
                   'INTRBASE',
                   'SERVER NAME: myserver:/usr/interbase/examples.v4/
employee.gdb; USER NAME: SYSDBA; PASSWORD: masterkey',
                   True));
end;
```

// NOTE: The formatting of the SERVER NAME indicates what protocol to use:

```
//
```



```
// Local connection:
// TCP/IP connection:      :
// NetBEUI connection:    ///  
// SPX connection:        @
```

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Check(DbiExit);  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Check(DbiOpenDatabase('TempIB',  
                           'INTRBASE',  
                           dbiREADWRITE,  
                           dbiOPENSHARED,  
                           'masterkey',  
                           0,  
                           nil,  
                           nil,  
                           hDb));  
end;
```

```
// The memory variable hDb is defined as follows:      hDb: hDBIDb;
```

```
end.
```

Last Modified: 26-OCT-00

## Article #25594: Is it possible to connect to InterBase from a Delphi app without using the BDE?

### Problem:

For performance reasons, some customers would like to avoid using the BDE

or ODBC to connect to InterBase from thier Delphi or C++Builder apps. Is this possible?

### Solution:

Yes, it is possible to program directly to the InterBase API using C or C++

(API = Application Program Interface) Please refer to the Programmer's Guide in the InterBase documentation.

There are also two third-party libraries available for download from our website, that were designed for this purpose.

Greg Deatz' FreeIBComponents ( <http://www.interbase.com/download/components.html> )

is a Delphi add-on which directly accesses the InterBase API.

With Jason Wharton's IB Objects ( <http://www.ibobjects.com/> )

"you can eliminate both the BDE and ODBC from your applications. Access will be direct to the InterBase API."

As of Delphi 5 and C++ Builder 5 you can also you InterBase Express (IBX).

## Article #25617: Performance techniques when using BDE and Delphi TTables with InterBase

### Problem:

With the default BDE configuration of the native InterBase driver, when the Delphi component TTable is used in order to fetch a set of rows, an InterBase cursor is setup in order to track the specific row that is being referenced in the dataset. When any modification to the rows is made (insert/update/delete), the Interbase API function `isc_commit_transaction()` is called resulting in a commit of the data and a loss of cursor.

The closing of the InterBase cursor triggers a Delphi "fetchall" which reloads all of the data accross the connection.

Other actions against the selected data, such as `locatelist`, `recordcount`, and using filters also result in a "fetchall" command in the default environment (driver flags set to NULL or 0).

With large data sets, this can be a very slow and network intensive operation. Multiple users doing this against the same datasets simultaneously can easily slow down the server significantly.

### Solution:

There are several possibilities for reducing the impact of using TTable components with large data sets.

Some of those suggested are :

1. In the BDE Configuration utility, you can set the `MaxRows` parameter to a restrictive value so that only the specified number of rows will be fetched.

This is, in general a good idea when it is possible for the user, in advance, to narrow down the search so that it is very likely that the specified number of rows will contain the desired information.

To make this user friendly, it may be necessary to code the search such that the next set of rows after the current set

can be easily obtained (such as a "Show Next" button).

2. It is possible to modify the InterBase driver flags on a global level to change the driver options for record locking and cursor retention. Since these flags are global to an application, care is required in order to ensure correct behavior.

If the BDE -> Configuration -> Drivers -> Native -> Intrbase ->drivers

flag is set to 4096 or 4608, the InterBase API call upon commit will

typically be changed from `isc_commit_transaction` to `isc_commit_retaining`. This will not close the InterBase cursor, hence avoiding the resultant "fetchall".

Note: the `TDatabase.commit` method ignores this flag and issues an `isc_commit_transaction` which will close the cursor and trigger the "fetchall".

Sideeffects:

=====

Setting the driver flag to 4096

-----

This may lead to potential deadlocks of multiple users attempt to modify the same row at the same time.

If it is likely that multiple users will attempt to modify the same rows frequently, this solution is not workable.

Also, when the user is done with the information, a hard commit should be done to free up the locks established on the table.

Setting the driver flag to 4608

-----

Any changes to data made by other users during the time this transaction is open will not be available since the transaction is scoped as read-committed at the time of the original request to fetch this information.

If it is important for one user to see changes made by another user in a timely manner, this solution is not suitable.

When a user is done with modifications, a hard commit will allow other changes to the database to be viewed.

See article in InterCom Vol 1 Number 2 (about handling deadlocks) for more information on possible driver flags.

Last Modified: 26-OCT-00

## **Article #25619: Do I need Local InterBase on the client machines along with my BDE application?**

### Problem:

Do I need to install Local Interbase on the client machines, or does InstallShield install the necessary parts on the client machine to allow them to use the local database?

### Solution:

Depending which version of Delphi you are using (the Professional or the Enterprise version) here is what the InstallShield version of both is able to do in terms of distributing and installing InterBase along with applications:

#### Professional:

InstallShield Express distributes applications, the BDE and a local version of the InterBase SQL Link driver (remote connections are not permitted). It does not distribute any of the other SQL Link drivers (DLL's). As such the BDE engine can only access ASCII, Paradox, dBASE and local InterBase files directly. Contact Inprise Corp. for information on obtaining the remote InterBase SQL Links driver. The BDE can also go through ODBC drivers, but InstallShield Express does not automatically distribute ODBC drivers.

#### Client/Server:

In the Enterprise version of Delphi and C++ Builder the InstallShield Express has all of the capabilities that are present in the Professional version plus some added capabilities. The SQL Link drivers can optionally be bundled in with the BDE, this includes the remote SQL Link driver for InterBase. When the InterBase SQL Link driver is included in the distributable install, the InterBase Client software is automatically bundled in as part of the SQL Link.

The InterBase Client software is freely distributable with no

royalties required and no copy limitations.

To distribute the Local InterBase Server requires obtaining the InterBase Deployment Kit which will license the purchaser to be able to distribute the Local InterBase Server to up to 20 seats or 100 seats depending on the Deployment kit that is purchased.

Last Modified: 25-OCT-00

## Article #25620: An Introduction to the TBatchMove Component in Delphi and C++ Builder

### Problem:

What is the TBatchMove component and how do I use it with InterBase (or any other database)?

### Solution:

An Introduction to the TBatchMove component

=====

#### 1. What is it?

TBatchMove is a VCL component that lets you perform operations on groups of records in a table.

The operations are named with intuitive names that basically tell what they do.

Listed below are the operations or "modes" that the TBatchMove component supports.

- batAppend => appends all records from the source table to the end of the destination table.

- batAppendUpdate => Appends all records from the source table to the end of the destination

table and updates

those records with the same primary index in the

destination table with

the same records from the source table.

- batCopy => Copies a table and its records to a new table

- batDelete => Delete all records in the destination table that also appear in the source table.

- batUpdate => Update existing records in the destination table with their counterparts


in the source table.

#### 2. How do I use it?

Listed below are steps to setup and run a TBatchMove component to



upsized Paradox data to InterBase.

- a. Put two TTable objects on the form, one for source and one for destination.
  1. Click on "Data Access" tab.
  2. Click on table icon.
  3. Click on form.
  4. Press  to go to the Object Inspector.
  5. Define the Database property for table1 to be the BDE alias that points to the directory where Paradox tables are located. In this case, we are using the DBDEMOS alias that is installed by default when you do a full install of Delphi 3.
  6. Define the TableName property in object inspector for table1 to be the Paradox table name, In this example, the table name will be the "orders" sample table that comes with Delphi. Since we have already specified the location of the tables, you can easily fill in the TableName property setting by selecting the table from the picklist of the TableName property.
  7. Repeat steps 2.a.1 - 2.a.4 for table2
  8. Define the database on table2 to be an alias that points to the InterBase database, in this case employee.gdb, by selecting the BDE alias that points to the employee database.
  9. Define TableName property for table2 to be the table name you want created in InterBase. For simplicity, call it "orders" as well.
  10. Configure the TBatchMove component
    - a. Put the TBatchMove component on the form.
      1. Click on the Mode property's edit box.
      2. Click on the BatchMove icon.

3. Click on the form.
- b. Define the source and destination for TBatchMove
  1. Hit `F4` to go to the object inspector.
  2. Set the Source property of TBatchMove to be table1.
  3. Set the Destination property of TBatchMove to be table2.
- d. Specify the mode for TBatchMove
  1. Click on the Mode property's edit box.
  2. Click on the down arrow that appears to get a picklist of modes to choose from.
  3. Click on "batCopy" to select it from the list.
- d. Activate TBatchMove
  1. Right click on TBatchMove component
  2. Click on "Execute" on the pop-up menu that appears.
- e. What happens when TBatchMove fails?

You can tell TBatchMove to document what happened when you run a batch move operation by setting the following properties. They tell Delphi to create a local Paradox table containing records that were not successfully part of the batchmove operation.

ChangedTablename => Records that were modified by the batch move operation

KeyvioltableName => Records that could not be added because they violated a uniqueness constraint.

ProblemTableName => Records that could not be added for one reason or another.

## Article #25621: How to create an About Box in Delphi or C++ Builder

Problem:

What are the steps to create an about box?

Solution:

Go to the Main Menu

Click on File | New Form

Go to the Object Inspector

Click on the Name property & Type: AboutBox

Click on the Caption property & Type: About This Program

Click on the BorderStyle property and select bsDialog

ADD 3 TEXT LABELS

Click on the Standard Tab

\*PROGRAM NAME

Click on the Label (giant A in your component palette) and place it on your form.

Go to the Object Inspector

Click on the Caption property & Type: The name of your program

\*VERSION

Click on the Label (giant A in your component palette) and place it on your form.

Go to the Object Inspector

Click on the Caption property & Type: The Version of your program

\*COPYRIGHT STATEMENT

Click on the Label (giant A in your component palette) and place it on your form.

Go to the Object Inspector

Click on the Caption property & Type: Your copyright statement

(To type the ? hold down the ALT KEY and type: 0169 and release the ALT KEY)

CREATE ICON

Click on the Additional Tab in the Component Palette

Click on the Image Button (Light Blue Horizon picture) and place it on the left side of your form

(You should see an empty square on your

form at this point)

Go to Object Inspector

Click on AutoSize property and change it to True

Click on: Picture property

Double click: the Value column.- The Picture Editor dialog box is displayed

Click: Load button

Navigate to: Common Files | Borland Shared | Images | Icons & select an icon

Click: OK

#### \*OK BUTTON

Click on the Additional Tab in the Component Palette

Click on the OK Button & place near the bottom of your form

Go to Object Inspector

Click on Kind - Change to bkok

#### \*BEVEL FRAME

Click on the Additional Tab in the Component Palette

Click on the Bevel Button and carefully move to the form and drag it around the 3 text lines & release

Go to Object Inspector

Click on the Shape property and change to bsFrame. You now have a 3D frame.

Last Modified: 25-OCT-00

## Article #25625: How to Set Up a C++ Builder or Delphi Form to View InterBase Data

### Problem:

This document gives step by step instructions on how to view data in an InterBase table from a C++ Builder form.

### Solution:

1. Make sure SQL Link for InterBase is installed on your computer.
2. Set up a BDE alias to point to InterBase database.
  - a. Bring up BDE Administrator.
  - b. Select Object | New from the menu.
  - c. Select "INTRBASE" from the "DataBase Driver name" list.
  - d. Enter a name for the BDE alias.
  - e. Enter the path to the database in space next to "SERVER NAME"
  - f. Enter your InterBase user name in the "User Name" space.
  - g. Select Object | Apply to save the alias.
  - h. Exit BDE administrator.
2. Start C++ Builder or Delphi
3. Create a new form by selecting File | New | Form from the menu.
4. Place a Ttable object on the form.
  - a. Click on the "Data Access" tab on the component palette.
  - b. Click on the Ttable object.
  - c. Click on the form to place the Ttable object on the form.
5. Define the Ttable object to be a table in the InterBase database.
  - a. Hit `F12` to bring up the Object Inspector.
  - b. Click on the "DatabaseName" property.
  - c. Click on down arrow that appears.
  - d. Select the database alias you created in step 1.
  - e. Click on TableName property.
  - f. Select the table name from your database's list of tables.

6. Put a DataSource on the form.
  - a. Click on Data Access Tab.
  - b. Click on DataSource button.
  - c. Click on the form to place the DataSource on the form.
  - d. Hit `Enter` to get to the Object Inspector.
  - e. Click on the space next to the "DataSet" property.
  - f. Select the Ttable object created earlier from the drop down list.
  
7. Put a DBGrid on the form.
  - a. Click on "Data Controls" tab on component palette.
  - b. Click on DBGrid object.
  - c. Click on form to place the dbGrid on the form.
  - d. Hit `Enter` to get to the Object Inspector.
  - e. Click on DataSource property.
  - f. Select DataSource created earlier.
  
8. Activate the Ttable object.
  - a. Click on Ttable object.
  - b. Hit `Enter` to get to object Inspector.
  - c. Click next to "Active" property.
  - d. Select "True" from drop down list.
  
9. Press `F5` to compile and run the form.

Last Modified: 24-OCT-00

## Article #25636: Using Form Wizard to Quickly Create a Delphi Form

### Problem:

This document will show how to quickly and easily make a Delphi form that is bound to a table.

### Solution:

The steps below demonstrate how to use the Form Wizard to quickly create a form that is bound to a table in a Delphi application. It is assumed that the user has done a full install of Delphi.

1. Start Delphi.
2. Select File | New Application from the menu.
3. Select Database | Form Wizard from the menu.
4. You will now see a form with Radio buttons for selecting Form Options and DataSet options.
5. For this example, we will use the defaults of :
  - Form Options set to "Create a simple form.
  - Data Options set to "Create a form using Ttable objects"
6. Hit the button.
7. Click on the white area under "Drive or Alias Name" heading. This will bring up a list of drives and aliases to choose from.
8. Select "DBDemos" as the alias you want to work with.
9. Select the "customer" table by double clicking on it.
10. Click on the ">>" button to include all fields from the customer table in the form.
11. Click on button.
12. Click on radio button next to "In a Grid"

13. Click on
14. Click on "Form and Data Module"
15. Click on "Finish" button.
16. See the form come up.
17. Hit to see data in form

Last Modified: 24-OCT-00



## Article #25639: Connecting via ODBC using Delphi or C++ Builder

Problem:

This document will cover the followings:

- \* Setting up an ODBC data source
- \* Connecting from Delphi or C++ Builder using the ODBC data source
- \* Displaying SQL Monitor log for the ODBC connection

Solution:

Setting up an ODBC data source

In order to make a connection from 3rd party products using ODBC, an ODBC data source is needed. A data source stores connection information

such as user name, password, location of database&etc. This is how to set up a data source:

? Start up ODBC Administrator. ODBC Administrator can be started by:

- 1) Going to Control Panel and double click on 32bit ODBC. OR
- 2) Starting the utility called "32bit ODBC Administrator" if you have the ODBC SDK installed.

? Stay at the User Data Source tab and click on Add. This will bring up another window titled "Create New Data Source".

? Pick the ODBC driver that to be used. Pick the InterBase driver which is called "InterBase 5.x driver by Visigenic (\*.gdb)" and then click on Finish.

This will bring up a new window with the title "InterBase ODBC Configuration".

? Fill in the blank fields in this window:

- 1) Data Source Name: Make up a name for your data source.
- 2) Description: This is the description of the data source. It's not required.
- 3) Network Protocol: Choose the protocol from the drop down list.
- 4) Database: Fill in the physical full path to the database including the database name to server.
- 5) Server: Fill in the server name. If you choose the protocol

"local", this will default to the local server.

6) Username: Fill in the database user name.

7) Password: Fill in the password corresponding to the above user name.

Optional for the new InterSolv ODBC driver for InterBase:

8) Go to the Advanced tab and fill in the CharacterSet and Roles.

? Clicking the OK button will bring back to the main form. You should see

the newly added user data source there.

Note: A user data source is a data source visible to the user whereas a system data source is visible to the system.

### Connecting from Delphi using the ODBC data source

ODBC connection from Delphi is very similar to connecting using BDE from Delphi. Here is an example of connecting using the Tquery component.

This example will also display the results of a sql statement.

? Drop a Tquery, a Tdatasource, and a Tdbgrid component on a Delphi form.

? Set the following properties for the Tquery component:

1) DatabaseName: Pick from the list the data source name you just created in

ODBC Administrator.

2) SQL: Input the sql statement to be executed. For example: "select \* from table1".

3) Active: Set to True to connect. And supply user name and password on connection.

? Set the following properties for the Tdatasource component:

1) Data Set: Set to the name of the Tquery component, or "query1" in this case.

? Set the following properties for the TDBGrid component:

1) Data Source: Set to the name of the Tdatasource component, or "data source1"

in this case.

? Now you can see the returned results from select statement in the dbgrid area.

## Displaying SQL Monitor Log for the ODBC connection

This section will display the SQL Monitor Log for ODBC and BDE. It will show the difference of the two logs.

? To turn on the SQL Monitor log: Go to the Database and choose SQL Monitor.

? Make the Tquery's connection to active from above. And you should see a log generated in the SQL Monitor window.

Here is a sample of what it looks like:

```
1      13:57:29  Log started for: Delphi 4
2      13:57:34  SQL Prepare: InterBase InterSolv Driver (*.g -
select * from test_table
3      13:57:34  SQL Execute: InterBase InterSolv Driver (*.g -
select * from test_table
4      13:57:34  SQL Vendor: ODBC - SQLAllocStmt
5      13:57:34  SQL Vendor: ODBC - SQLExecDirect
6      13:57:34  SQL Vendor: ODBC - SQLNumResultCols
7      13:57:34  SQL Vendor: ODBC - SQLDescribeCol
8      13:57:34  SQL Vendor: ODBC - SQLDescribeCol
9      13:57:34  SQL Misc: InterBase InterSolv Driver (*.g - Set
rowset size
10     13:57:34  SQL Vendor: ODBC - SQLBindCol
11     13:57:34  SQL Vendor: ODBC - SQLBindCol
12     13:57:34  SQL Stmt: InterBase InterSolv Driver (*.g - Fetch
13     13:57:34  SQL Vendor: ODBC - SQLSetStmtOption
14     13:57:34  SQL Vendor: ODBC - SQLExtendedFetch
15     13:57:34  SQL Data Out: InterBase InterSolv Driver (*.g -
Column = 1, Name = TESTING, Type = fldTIMESTAMP, Precision = 19,
Scale = 0, Data = 3/31/1998 10:42:52:0
16     13:57:34  SQL Data Out: InterBase InterSolv Driver (*.g -
Column = 2, Name = TESTDATE, Type = fldTIMESTAMP, Precision = 19,
Scale = 0, Data = 3/30/1998 14:4:39:0
17     13:57:34  SQL Stmt: InterBase InterSolv Driver (*.g - Fetch
18     13:57:34  SQL Vendor: ODBC - SQLSetStmtOption
19     13:57:34  SQL Vendor: ODBC - SQLExtendedFetch
```

```
20      13:57:34  SQL Stmt: InterBase InterSolv Driver (*.g - EOF
21      13:57:34  SQL Stmt: InterBase InterSolv Driver (*.g - Reset
22      13:57:34  SQL Vendor: ODBC - SQLFreeStmt
```

Here is a sample SQL monitor log using the BDE for the same operation:

```
1      14:06:45  SQL Prepare: INTRBASE - select * from test_table
2      14:06:45  SQL Vendor: INTRBASE - isc_dsqli_allocate_statement
3      14:06:45  SQL Vendor: INTRBASE - isc_start_transaction
4      14:06:45  SQL Vendor: INTRBASE - isc_dsqli_prepare
5      14:06:45  SQL Vendor: INTRBASE - isc_dsqli_sql_info
6      14:06:45  SQL Vendor: INTRBASE - isc_vax_integer
7      14:06:45  SQL Transact: INTRBASE - XACT (UNKNOWN)
8      14:06:45  SQL Vendor: INTRBASE - isc_commit_retaining
9      14:06:45  SQL Execute: INTRBASE - select * from test_table
10     14:06:45  SQL Vendor: INTRBASE - isc_dsqli_execute
11     14:06:45  SQL Stmt: INTRBASE - Fetch
12     14:06:45  SQL Vendor: INTRBASE - isc_dsqli_fetch
13     14:06:45  SQL Data Out: INTRBASE - Column = 1, Name =
TESTING, Type = fldTIMESTAMP, Precision = 1, Scale = 0, Data =
3/31/1998 10:42:52:0
14     14:06:45  SQL Data Out: INTRBASE - Column = 2, Name =
TESTDATE, Type = fldTIMESTAMP, Precision = 1, Scale = 0, Data =
3/30/1998 14:4:39:0
15     14:06:45  SQL Stmt: INTRBASE - Fetch
16     14:06:45  SQL Vendor: INTRBASE - isc_dsqli_fetch
17     14:06:45  SQL Stmt: INTRBASE - EOF
18     14:06:45  SQL Stmt: INTRBASE - Reset
19     14:06:45  SQL Vendor: INTRBASE - isc_dsqli_free_statement
20     14:06:45  SQL Transact: INTRBASE - XACT Commit
21     14:06:45  SQL Vendor: INTRBASE - isc_commit_transaction
```

Note: What's after SQL Vendor is the layer beyond Delphi. For ODBC, it's ODBC and for BDE is the "INTRBASE". Also pay special attention to the function follows the name. ODBC uses ODBC function calls and where as BDE uses InterBase api function calls.

Last Modified: 27-OCT-00

## Article #25644: What are Delphi and C++ Builder Action Lists and how do I use them?

Problem:

What are Delphi and C++ Builder Action Lists and how do I use them?

Solution:

What are Action Lists?

Action lists are a way of linking controls that perform similar tasks.

Why use Action Lists?

This is a time saver. Once controls are linked, changes made to an action list item will propagate to all components tied to that action item.

Action List Demo Objective:

To make the File | New command use an action list item

Start	New application
Add	MainMenu component to the form
Double click	MainMenu component - To import File Menu template
Right click	Form1.MainMenu
Select	Insert from Template
Select	File Menu
Click	OK
Add	ActionList component to the form
Double click	ActionList component This will display the Action
List editor	
Click	New Action - This action will be responsible for
the New File	
Click	Object Inspector
Click Name	Change to: actNewFile
Click Caption	Change to: &New
Click Shortcut	Change to: CTRL+N
Double click	MainMenu component
Click	Object Inspector
Click Action	Change to: actNewFile You can also use the down
arrow to select this	
Go to	Action List Editor

Double click      actNewFile  
Click             Object Inspector  
Click             Events tab    This will display 3 properties:  
OnExecute, OnHint, & OnUpdate  
Double click      OnExecute  
Enter             ShowMessage( Create New File );  
Save All

If we ran the application now, selecting File | New would trigger the message Create New File

The benefit of saving time occurs when you have multiple components.

Add             Button component to the form  
Click             Properties tab of the Object Inspector  
Click Action     Change to: actNewFile - You can also use the down  
arrow to select this  
Add             CheckBox component to the form  
Click             Object Inspector  
Click Action     Change to: actNewFile

Doing the above will now tie the Button and CheckBox component properties to that action

Run the application

Change one line of code, save all, and then run the application again.

That change will propagate to all components in the action list we created.

Last Modified: 24-OCT-00

## Article #25645: A possible explanation for 'Name not Unique in this context' errors

### Problem:

My app was working fine, when I went to add some fields into one of my tables. When I try to add the fields using the field editor, I get an error message stating 'Name is not unique in this context'.

What happened?

### Solution:

The information in this document is relevant to all versions of C++Builder and Delphi.

The error, 'Name is not unique in this context' means there's already a component (field or otherwise) on the form with the same name.

One instance where this error can occur is if you have a form that is descended from another form.

If you create the field on the parent form, then try to explicitly add the field to the child form as well, rather than inheriting it, this error can occur.

source: list serve

Last Modified: 24-OCT-00

## Article #25647: Using Variables & For Loops in Delphi

### Problem:

Using Variables & For Loops in Delphi to cause a piece of code to get executed a certain number of consecutive iterations based on certain criteria.

### Solution:

The information in this article applies to:

\* All versions of Delphi

1. From the Standard tab of the Component Palette drop a Button & a Memo Field on the Form.
2. Enlarge the Memo Field vertically by dragging the top & bottom edges.
3. Double click the Button to open the Event Handler.
4. Enter the code as shown in the procedure body:

```
procedure TForm1.Button1Click(Sender: TObject); // this line is pre-
coded in the event handler
```

```
var {this declares a section of code where you can define variables}
I : Integer; {define 'I' as an Integer variable}
```

```
Begin // This line is pre-coded in the event handler
```

```
Mem1.Lines.Clear; {clears characters previously displayed in memo
field}
```

```
For I := 0 to 5 do {incremental 'For Loop'}
  Mem1.Lines.Add('This is iteration ' + IntToStr(I)); {display
string}
{'IntToStr' converts the increment variable from an integer to a
string so that it can be displayed}
```

```
Mem1.Lines.Add(' '); {display blank line}
```

```
For I := 5 downto 0 do {decremental 'For Loop'}
  Mem1.Lines.Add('This is iteration ' + IntToStr(I));
```



end;

end.

5. Click the 'Run' button on the Toolbar (or press F9).

Last Modified: 24-OCT-00

## Article #25648: Peep Show Picture Viewer

### Problem:

Creating a graphics viewing application for viewing bitmap, icon, and metafiles (\*.bmp, \*.ico, \*.mtf).

### Solution:

The information in this article applies to:

#### Creating the Main Form

Step 1: Select File|New Application from the File Menu.

Step 2: For the main form, change the Name property to 'MainForm'.

Step 3: Change the Caption property to 'Peep Show'.

Step 4: Change the Height to 450 and the Width to 575 (or other suitable values for you display resolution).

Step 5: Change the FormStyle to fsMDIForm.

#### Adding a Menu

Step 1: Click the Standard tab of the Component palette and click the MainMenu button.

Step 2: Click on the form to place a MainMenu component on the form.  
(position unimportant doesn't  
show at runtime)

Step 3: Change the Name property to MainMenu.

Step 4: Double-click the MainMenu component to display the Menu Designer.

Step 5: Place your cursor over the Menu Designer and click you right mouse button. Choose Insert  
from Template from the context menu. The Insert  
Template dialog box appears.

Step 6: Chosse MDI Frame Menu and click OK. The menu is displayed in

the Menu Designer.

Step 7: Click the system close box on the Menu Designer to close it.

### Configuring the File|Open and File|Save Dialog Boxes

Step 1: Click the Dialogs tab on the Component palette. Choose an `OpenPictureDialog` component and place it on the form. The `OpenPictureDialog` component's icon can be placed anywhere on the form.

Step 2: Change the Name property of the Open dialog box to `OpenPictureDialog`.

Step 3: Change the title property to `Open a Picture for Viewing`.

Step 4: Add a `SavePictureDialog` component.

Step 5: Change the Name property of the component to `SavePictureDialog` and the Title property to `Save a Picture`.

### Coding the File|Open and File|Save As Menu Items

Step 1: On the main form, choose `File|Open` from the menu. An event handler is created for that menu item and the Code Editor is displayed.

Step 2: Type code so that the event handler looks like this:

```
procedure TmainForm.Open1Click(Sender: TObject);
var
    Child : TChild;
begin
    If OpenPictureDialog.Execute then begin
Child := Tchild.Create(Self);
with Child.Image.Picture do begin
    LoadFromFile(OpenPicturedialog.FileName);
    Child.Clientwidth := Width;
        Child.ClientHeight := Height;
end;
Child.Caption := ExtractFileName(OpenPictureDialog.
FileName);
Child.Show;
```

```
        end;  
    end;
```

Step 3: Press F12 to switch back to the form. Now choose File|Save As from the menu. The File|Save As event handler is displayed.

Step 4: Type code so that the File|Save As event handler looks like this:

```
    procedure TmainForm>SaveAs1Click(Sender: TObject);  
    begin  
        if SavePictureDialog>Execute then  
with ActiveMDIChild as Tchild do  
        Image.Picture.SaveToFile(SavePictureDialog.FileName);  
        end;
```

(Choose File|Save All to save project)

## Coding the Window Menu

Step 1: Switch back to the form by pressing F12. Choose Window|Tile from the form's menu.

Step 2: You need to enter only a single line of code for the event handler: Tile;

Step 3: Switch back to the form and repeat the process for Window|Cascade: Cascade;

Step 4: Repeat the steps for the Window|Arrange All: ArrangeIcons;

## Creating the MDI Child Form

Step 1: Create a new form using the toolbar button or the File menu.

Step 2: Change the Name property to Child. The Caption property can be ignored because you will be setting the dialog box's caption at runtime.

Step 3: Change the FormStyle property to fsMDIChild. This is necessary for the form to be treated as an MDI child window.

Step 4: Click the Additional tab on the Component palette. Click the

Image button and place an  
Image component anywhere on the form.

Step 5: Change the Name property to Image.

Step 6: Change the Stretch property to True.

Step 7: Change the Align property to alClient. The Image component expands to fill the client area of the form.

Step 8: Choose File|Save and save the form's unit as MDIChild.

Step 9: Switch to the Code Editor. Click the PctViewU tab. Now choose File|Use Unit from the main menu, select the MDIChild unit, and click OK.

## Finishing Touches

To prevent the child window from being created at start up do the following:

Step 1: Choose Project|Options from the main menu.

Step 2: If necessary, click the Forms tab. The list of forms to auto-create is displayed.

Step 3: Click the child from and then click the > button. This removes the child from from the auto-create list and puts it in the Available froms list.

To make the close button actually close the window instead of minimizing it (the MS default) do the following:

Step 1: Bring up the child window from int the Form Designer. Be sure the form itself is selected and not the Image component on the form (choose the Child from the Component Selector at the top of the object Inspector, if necessary).

Step 2: Double-click the Value column next to the OnClose event in the Object Inspector. Add a line of code to the event havndler so that it looks

like this:

```
procedure Tchild.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := caFree;  
end;
```

The End!

Last Modified: 24-OCT-00

## Article #25665: Handling exceptions in Delphi

This article provides some information on setting up error handling with Delphi.

One of the big advantages of using exceptions is that you do not have to check the result of every function call that you make. Typical C programming involves making a function call (such as `allocmem`) and checking the result to see if the function succeeded or not. More than likely, the function will always work. Having an exception being raised when an error does happen, makes less work for the programmer.

The best way to catch exceptions is by defensive programming. Any code which could possibly raise an exception can be wrapped in one of two blocks: a **try ... except** block or a **try ... finally** block.

Here is an example of using **try ... except**:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    try  
        // Some code which may raise an exception  
    except  
        on E: EConvertError do  
            begin  
                // Handle only EConvertError, or any derived classes  
            end;  
        on E: Exception do  
            begin  
                // Handle only Exception, or any derived classes  
            end;  
        end;  
end;
```

Once an exception is raised, the exception handler jumps immediately to the nearest **except** (or **finally**) block. Notice how you can pick exactly what exceptions you want to handle. Also, notice that `EConvertError` derives from `Exception`. You must list more specific (derived) exceptions before generic exceptions. Also, if you do not catch the exception (such as, if we had not put **E: Exception** in the handler), then the exception would continue to propagate outside of the **try ... except** block, and be caught else where.

The big difference between a **try ... except** block and a **try ... finally** block is that the **try ... except** block is only executed if an exception was raised, while the **try ... finally** is **always** executed. Because **finally** blocks are always executed, it doesn't make sense to catch specific exceptions with the **on** keyword, so this is not possible in them. **Try ... finally**'s are typically used 3 to 1 times over **try ...**

**except**'s, as handling for an exceptional case it is much rarer than simply doing resource management. Normal resource management with a **try ... finally** block typically looks like this:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    Form: TForm1;  
begin  
    Form := TForm1.Create(Self); // Allocate some memory  
    try  
        // Some code which may raise an exception  
        Form.DoSomethingWhichMayRaiseAnException;  
    finally  
        Form.Free; // Always free the memory.  
    end;  
end;
```

The Form will always be freed. One may ask why the line **Form := TForm1.Create(Self);** is not inside the **try ... finally** block. The reason is simple: what if Create raised an exception? Then you would be freeing the Form variable when it was never allocated! Now, the compiler is smart enough to realize this, and would give you a warning if you put the Form inside the block:

```
[Warning] Unit1.pas(34): Variable 'Form' might not have been  
initialized
```

The next question may be what to do with the exception once you catch it. Sometimes, you may want to ignore certain exceptions (such as EConvertError), or execute certain code when an exception has been raised. If you don't handle the exception (or catch with a try...except block), then the default exception handler will show a dialog with the exception's message. You can customize this for your entire application by placing a **TApplicationEvents** component on your form and writing code for the **OnException** event. A reason for doing this might be to log the exception do a file. Or, you may want to display a custom dialog. I have an application which automatically closes the dialog after 15 seconds with no mouse clicks. Very handy!

Now, raising an exception is simple:

```
raise Exception.Create('some exception!');
```

Notice that it is possible to raise an exception inside of a **try...except/finally** block. If you want to re-raise the current exception, you can simply use **raise**:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    Form: TForm1;  
begin
```



```
Form := TForm1.Create(Self); // Allocate some memory
try
  // Some code which may raise an exception
  Form.DoSomethingWhichMayRaiseAnException;
except
  on E: Exception do
    raise; // re-raise the currently caught exception
end;
end;
```

Note that it is wrong to try and re-raise it with:

```
raise E; // DON't DO THIS!
```

After you catch an exception, it is implicitly freed by the compiler (since, it has to be freed at some time!). Re-raising it with just **raise** doesn't free the current exception object. Calling **raise E**, where E is the caught exception, means that the compiler will implicitly free E twice!

This should give you a pretty good overview as to how to handle exceptions with Delphi.

Last Modified: 01-MAY-01

## Article #25669: Can I create stored procedures via Delphi or C++ Builder?

### Problem:

Can I create InterBase stored procedures in my InterBase database using Delphi or C++ Builder?

### Solution:

\*\*\*\*\*


This document refers to:

- \* InterBase versions 4.X and above.

\*\*\*\*\*

Yes, you can create stored procedures via Delphi. You can use the TQuery component to pass the SQL statements needed to create the stored procedure on the InterBase server. One caveat you will want to be aware of is that you need to have two colons in front of memory variables instead of one when you call the stored procedure.

Listed below are steps to create an example of defining stored Procedures via Delphi or C++ Builder:

1. Select File | New Application from the menu.  
--This will bring up a new application with a blank form.
2. Click on the button icon on the standard tab of the component palette.
3. Click on the "Data Access" tab.
4. Click on the Query icon.
5. Hit  to bring up the Object Inspector.
6. Set the DatabaseName property to your InterBase alias.
7. Click on the three dots in the space next to the "SQL" property.
8. You can now enter the text of your stored procedure.
9. Type in the text of the stored procedure definition code:

```
create procedure myproc returns (a integer)
  as
  begin
    a=5;
  end
```

10. After you have entered the code for your stored procedure, click on the button.
11. Next, you will need to tell Delphi to run your stored procedure code. To that, do the steps listed below:
  - a. Click on the button.
  - b. Hit to bring up the Object Inspector.
  - c. Click on the Events tab.
  - d. DoubleClick on the space next to "OnClick"
  - e. Enter in the following code after the line with the word 'begin' on it:  
query1.execSQL;
12. Push to run the program.

Last Modified: 23-OCT-00

## Article #25670: How to Use Delphi or C++ Builder to Run a Query in a Text file

### Problem:

I would like to run a query saved in a text file using Delphi or C++ Builder. How do I do that?

### Solution:

### Step by Step Instructions on How to Run a Query from a File Using Delphi

=====

1. Create an alias in the BDE to point to the database.
2. Create a file called 'myquery.sql' in your c: root directory that contains the following line:  

```
select * from employee
```
3. Start Delphi.
4. File | New Application
5. Place a button on the form. You can do this by clicking on the button icon on the Standard page of the component palette and then clicking on the form.
6. Click on the Data Access tab of the component palette.
7. Click on TQuery icon and then on the form to place a TQuery on the form.
8. Hit `Alt+O` to bring up the Object Inspector.
9. Define the TQuery's DatabaseName property to be the alias that points to the employee database.
10. Click on DataSource icon and then on the form to place a DataSource on the form.
11. Hit `Alt+O` to bring up the Object Inspector.

12. Define the DataSet property of the DataSource object to be Query1.
13. Click on the Data Controls tab on the component palette.
14. Click on DBGrid icon and then on the form to place a DBGrid on the form.
15. Hit `Alt+V` to go back to the Object Inspector.
16. Define the DBGrid's DataSource property to be DataSource1.
17. Click on the button again.
18. Hit `Alt+V` to go to the Object Inspector.
19. Click on the 'Events' tab.
20. Double Click in the white space next to the OnClick event.
21. Place the following code in the button's OnClick event code:

```
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.LoadFromFile('C:MYQUERY.SQL');
Query1.Open;
```
22. Run the application by hitting `F5`.
23. Press the button to see the query run.

Note: You can also read a query into the TQuery object with SQL Builder.

Use the File | Import from file menu in SQL Builder choices to do this.

For more information about accessing SQL Builder, please refer

to document number 25679.

## **Article #25671: Where are the Delphi 4 InterBase Certificate Keys and IDs located?**

### Problem:

Can't find the Certificate Key and ID for installing InterBase from the Delphi 4 CD.

### Solution:

The information in this article applies to:

- \* Delphi 4
- \* C++Builder 3

As with the InterBase installation from the C++Builder CD, the license keys are located in the IB5 directory on the Delphi 4 CD in a file named `ibkey.txt`.

Last Modified: 23-OCT-00

## Article #25679: How to create SQL queries with SQL Builder

### Problem:

Here is a quick walk-through of creating a query with SQL Builder. SQL Builder is an interactive tool that is included with Delphi and C++ Builder. It allows a user to create an SQL query using a graphical interface. Complete instructions on displaying results using a dbGrid component is also covered.

### Solution:

\*\*\*\*\*

Note: The information in this document was tested with C++ Builder 3.0.

\*\*\*\*\*

#### A. Setup Display Components

=====

1. File | New Application
2. Click on the Data Access tab.
3. Click on the Query icon and then on form to place Tquery component on the form.
4. Hit `F4` to go to the Object Inspector.
5. Click on space next to DatabaseName.
6. Select the alias you will be using from the pull down menu,  
For this example, we will use the "DBDEMOS" alias that ships with the BDE.
7. Click on DataSource icon and then on form to place DataSource on the form.
8. Hit `F4` to go to the object Inspector.
9. Click on the space next to the "DataSet" property.
10. Enter "Query1" (without the quotation marks) as the DataSet.
11. Click on Data Controls tab.
12. Click on DBGrid icon and then on the form.
13. Hit `F4` to go to the Object Inspector.

14. Click on space next to the "DataSource" property.

15. Click on downarrow and select "DataSource1".

## B. Set Up Query

=====

1. Right click on query component.

2. Select SQL Builder from menu.

3. Click on down arrow next to "Table" drop down list.

4. Select "customer" from the list.

5. See the list of fields in customer table appear in a box.

6. Click on checkbox next to the table name.

7. Notice that this checks all the fields.

8. If you hit the SQL button at this time, you can see the SQL statement that was generated for you.

9. Click on File | Exit to get out of the SQL Builder.

10. When prompted to save changes to the query, click on "Yes"

11. Your SQL query is now built and stored in the tquery component. You can see that the query got built by looking at the SQL property of the Tquery component.

12. To see the results of the query, you will need to set the active property on the

Tquery to true. Here are the steps to do that:

- a. Right click on the tquery component
- b. Hit to get to the Object Inspector.
- c. Click on space next to Active property.
- d. Select True from the list.

11. Hit to run the form. You will see the results of your query in a grid.

Last Modified: 23-OCT-00



## Article #25682: Delphi Training: Accepting User Input Using An Edit Box

### Problem:

An example of using an Edit Box to accept user input with Delphi.

### Solution:

EDIT BOX (Accepting User Input)

Wayne Shaddock

October 2, 1998

### This demo uses:

- A TForm
- A TButton
- A TEdit
- 2 TLabel

It demonstrates the use of an edit box to take input from the console and display to the user some information about the input data.

1. Open a new project.
2. From the Standard tab on the Component Palette place a button, 2 labels, and an edit box on the form.
3. Arrange the components as desired. (The first label will display some instructions when the button is clicked so it should be placed above the edit box. The second label will give information about the user's input, so it should be placed below or to the right of the edit box)
4. Double click the button to create an event handler and type in the following code in the body of the Button1.Click procedure:

```
Label1.Caption := 'Enter an integer';
```

5. Double-click the edit box and create a variable declaration section right before the body of the procedure. Declare the following variable:

```
X : Integer;
```

6. Type the following code in the body of the Edit1.Change procedure:

```
X := StrToInt(Edit1.Text);  
if X > 10 then  
    Label2.Caption := 'You entered a number greater than 10'  
else if  
X <= 10 then  
    Label2.Caption := 'You entered a number less than or equal to 10';
```

7. Click the 'Run' arrow on the toolbar to compile and run the program.

Last Modified: 23-OCT-00

## Article #25683: Using Control Characters With Delphi

### Problem:

An example of using control characters with Delphi to format text displayed in a label component.

### Solution:

Using Control Characters With Delphi

Wayne Shaddock

?1998

#### 1) Open a new project:

From the 'File' menu select New Application

#### 2) Place two buttons on the form:

From the Standard tab of the Component Pallet double-click the button icon twice

#### 3) Place two labels on the form:

From the Standard tab of the Component Pallet double-click the label icon twice

#### 4) Change the captions of the two buttons:

Click Button1 to select it and change the 'Caption' property to 'No Control Character' (without the single quotes)

Click Button2 to select it and change the 'Caption' property to 'Control Character' (without the single quotes)

#### 5) Change the captions of the two labels:

Click Label1 to select it and change the 'Caption' property to 'Without Carraige Return Control Character' (without the single quotes)

Click Label1 to select it and change the 'Caption' property to 'With Carraige Return Control Character'

(without the single quotes)

6) Code the Event Handler for Button1:

Double-click Button1 to invoke the Event Handler. Declare a variable declaration section by typing the keyword 'var' between the keyword 'procedure' and 'begin'.

Declare a string variable by typing the following below the keyword 'var':

```
S1 : string;
```

Type the following code in the body of the procedure:

```
S1 := 'This is a string with no carriage return control character.';
Label1.Caption := S1;
```

7) Code the Event Handler for Button2:

Following the example for the first button enter the code for the event handler as follows:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S1, S2 : string;
begin
  S1 := 'This is a string with';
  S2 := 'a carriage return control character.';
  Label2.Caption := S1 + #10 + S2;
end;
```

9) Compile and run the project: press the F9 key.

Pressing the button labeled 'No Control Character' results in the first label displaying a text string as a single line. Pressing the other button results in a text string being displayed in the second label on two lines because of the inclusion of the control character '#10' in the code for Button2's event handler.

Last Modified: 23-OCT-00

## Article #25704: InterBase array datatype and the BDE

### Problem:

I have just created a table that contains a field with an array of integers defined as one of the fields. How can I access it from my Delphi or C++ Builder application using the BDE?

### Solution:

\*\*\*\*\*

The information in this article applies to:

\*InterBase versions 3.3, 4.X, and 5.X

\*Delphi versions 1.X, 2.X, 3.X, and 4.0

\*\*\*\*\*

Delphi and C++ Builder's data aware components do not support arrays. One has to use

InterBase API calls to get data into and out of the arrays. Thus you must either write your own array handling code, or use a component set which already does this.

Last Modified: 23-OCT-00

## Article #25707: Creating a Worker Thread for Background Processing

Problem:

Creating a Worker Thread for Background Processing

Solution:

This information in this document applies to:

- \* InterBase 5.x
- \* Delphi 32-bit

\*\*\*\*\*

Steps to Create a Background Thread

\*\*\*\*\*

1. Create new Application

2. Create a descendant of TThread (we'll call it TWorkerThread). You can do this by

selecting the File | New menu option and choosing the "Thread Object" item.

3. Save your unit (we'll call it WorkerThreadUnit)

4. Add WorkerThreadUnit to interface uses section of TMainForm. This is added to the

interface section, opposed to the implementation section, because we are going to

create a variable of this class and need this when declaring the main form's class.

```
{ begin code snippet }  
interface
```

```
uses
```

```
    WorkerThreadUnit;
```

```
{ end code snippet }
```

5. Create a variable of TWorkerThread. Here is a code snippet showing a variable of

type TWorkerThread being declared.

```

{ begin code snippet }
type
  TMainForm = class(TForm)
  private
  public
    { Public declarations }
    workerThread: TWorkerThread;
  end;
{ end code snippet }

```

6. Add TMainForm's unit (MainUnit) to the implementation section of WorkerThreadUnit. We add it to the implementation section, because we will use it in our classes declaration, not its definition.

```

{ begin code snippet }
implementation

  uses SysUtils, MainUnit;
{ end code snippet }

```

7. Fill in the Execute method of TWorkerThread to do what you want when thread is created. This method will be called when you create an instance of your thread.

```

{ begin code snippet }
procedure TWorkerThread.Execute;
begin
  { Place thread code here }
  .
  .
  .
end;
{ end code snippet }

```

8. Create a method to be used for all your user interface code. This needs to be in a separate method so that we can invoke it with the Synchronize method of the TThread class (we'll call it UpdateUI)

```

{ begin code snippet }
procedure TWorkerThread.UpdateUI;
begin

```

```

    { do user interface stuff here }
    .
    .
    .
end;
{ end code snippet }

```

9. In TWorkerThread's Execute method call the Synchronize method and pass it the UpdateUI method as a parameter.

```

{ begin code snippet }
procedure TWorkerThread.Execute;
begin
    { Place thread code here }
    FreeOnTerminate := true;
    Synchronize(UpdateUI);
end;
{ end code snippet }

```

10. Now, in your main form create an instance of your TWorkerThread class where ever you need a new thread. Here is an example showing an instance of our

TWorkerThread class being created when Button1 is clicked.

```

{ begin code snippet }
procedure TMainForm.Button1Click(Sender: TObject);
begin
    workerThread := TWorkerThread.Create(false);
end;
{ end code snippet }

```

\*\*\*\*\*

Notes

\*\*\*\*\*

\* Delphi requires (there are a few exceptions) that all user interface code be executed in the context of the application's primary thread.

\* TThread provides a method called Synchronize to allow some of its methods to be executed in the context of the application's primary thread.



\* TThread is an object pascal object that encapsulates a Win32 API thread object.

The parameter to the TThread.Create method controls whether the thread is created

in a suspended state. If true is passed in then the thread will be created, but its execute

will be suspended. If false is passed in the thread will be created and will execute

immediately. In the above example I passed in false so that the worker thread would

execute as soon as it was created.

\* The thread associated with a TThread object will be terminated as soon as the

TThread.Execute method is done executing.

\* However, the TThread object will not automatically be destroyed when the

underlying thread terminates. To destroy the TThread object you can:

- \* Explicitly call the TThread.Destroy method

- \* Set the TThread's property FreeOnTerminate to true in the TThread.Execute

method. If FreeOnTerminate is true then the TThread object will be

automatically destroyed when the underlying thread is terminated.

Last Modified: 23-OCT-00

## Article #25745: Why can't I display data in descending order in Delphi?

### Problem:

Why can't I display data in descending order in Delphi or C++ Builder?

### Solution:

If you are using a TTable, descending indexes can be a problem. A way around this

is to use TQuery and specify the SQL order by clause. The downside to this method

is refresh is not automated.

Last Modified: 23-OCT-00

## Article #25774: I am getting permission denied errors when I run a Delphi or C++ Builder data-aware application.

Problem:

The errors are:

Cannot find file e:tempdboxusers.lck

Permission Denied

Solution:

In some sequences of events BDE will store data in temporary Paradox tables. Some of the scenarios where BDE creates temporary Paradox tables

are with heterogeneous joins, and with CachedUpdates. BDE will look to where the Paradox PrivateDir is set to be. This will be based on the drive and directory that the application is launched from. If the application is launched from a networked or mapped drive then the temporary Paradox tables will be created on that drive and network rights will be an issue. In most cases it will be sufficient to route the Paradox PrivateDir setting to the local drive. The code snippets below show two possible approaches to doing this in Delphi. These approaches require the use of the Session object, specifically the Session.PrivateDir member of the object. This also requires having DBTables in the Uses clause. One approach is to reassign the Session.PrivateDir to the Windows temporary directory, by calling the GetTempPath() Windows API function call. Add Windows to the Uses clause

so that GetTempPath() can be called. The other approach is to assign a hard coded path to Session.PrivateDir.

Uses Windows, DBTables

```
const
    temp_path_len = 512;
var
    temp_path: array[1 .. 512] of char;

//
// Place this code snippet at the
// appropriate place in the code
//
GetTempPath(temp_path_len, @temp_path);
Session.PrivateDir := StrPas(@temp_path);
```

```
//  
// or  
//  
// Session.PrivateDir := 'C:Temp';
```

Last Modified: 26-SEP-00

## Article #25807: Purpose of the interbase Client32 files

### Problem:

What is the purpose of the interbase Client32 files in the Pro versions of C++Builder and Delphi?

### Solution:

This information applies to InterBase 5.x for Win95 and NT

To allow you to have a pre-made install of the Client-only software with a predefined license key. You can install this on as many workstations as you want.

Source: Bill Karwin

Last Modified: 12-OCT-00

## Article #25831: How do I use the InterBase API security functions to add, modify and delete users

### Problem:

Since there are no VCL controls to add, modify, or delete users for an InterBase server I am not sure how to do this task. The BDE also does not allow me to add, modify or delete users. I need to be able to do this task within a program? How do I do it?

### Solution:

InterBase has surfaced the following functions to add, modify and delete users:

```
ISC_STATUS isc_add_user(ISC_STATUS *status, USER_SEC_DATA *
user_sec_data)
```

```
ISC_STATUS isc_modify_user( ISC_STATUS *status, USER_SEC_DATA
*user_sec_data)
```

```
ISC_STATUS isc_delete_user( ISC_STATUS *status, USER_SEC_DATA
*user_sec_data)
```

The following program below shows how to define these functions in a Delphi program and how to call them in a Delphi application:

```
unit add_user1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
StdCtrls, ExtCtrls;
```

```
const
```

```
    sec_uid_spec = $01;
    sec_gid_spec = $02;
    sec_server_spec = $04;
    sec_password_spec = $08;
    sec_group_name_spec = $10;
    sec_first_name_spec = $20;
    sec_middle_name_spec = $40;
```

```
    sec_last_name_spec = $80;  
sec_dba_user_name_spec = $100;  
    sec_dba_password_spec = $200;
```

```
    sec_protocol_tcpip = 1;  
sec_protocol_netbeui = 2;  
    sec_protocol_spx = 3;  
    sec_protocol_local = 4;
```

```
type
```

```
    ISC_STATUS = Integer;  
P_ISC_STATUS = ^ISC_STATUS;
```

```
    USER_SEC_DATA = record  
        sec_flags: Smallint;  
        uid: Integer;  
        gid: Integer;  
        protocol: Integer;  
        server: PChar;  
        user_name: PChar;  
        password: PChar;  
        group_name: PChar;  
        first_name: PChar;  
        middle_name: PChar;  
        last_name: PChar;  
        dba_user_name: PChar;  
        dba_password: PChar;
```

```
    end;
```

```
P_USER_SEC_DATA = ^USER_SEC_DATA;
```

```
TForm1 = class(TForm)  
    add_user_name: TEdit;  
    add_password: TEdit;  
    Button1: TButton;  
    Label1: TLabel;  
    Label2: TLabel;  
    Bevel1: TBevel;  
    Bevel2: TBevel;  
    Bevel3: TBevel;  
    mod_user_name: TEdit;  
    mod_password: TEdit;  
    Button2: TButton;  
    Label3: TLabel;  
    Label4: TLabel;  
    Label5: TLabel;
```

```

Label6: TLabel1;
del_user_name: TEdit;
del_password: TEdit;
Button3: TButton;
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
private
    function add_user_by_name_password(user_name: PChar; password:
PChar): ISC_STATUS;
    function mod_user_by_name_password(user_name: PChar; password:
PChar): ISC_STATUS;
    function del_user_by_name_password(user_name: PChar; password:
PChar): ISC_STATUS;
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

function isc_add_user(status: P_ISC_STATUS; user_sec_data:
P_USER_SEC_DATA):
    ISC_STATUS; stdcall; external 'gds32.dll';

function isc_modify_user(status: P_ISC_STATUS; user_sec_data:
P_USER_SEC_DATA):
    ISC_STATUS; stdcall; external 'gds32.dll';

function isc_delete_user(status: P_ISC_STATUS; user_sec_data:
P_USER_SEC_DATA):
    ISC_STATUS; stdcall; external 'gds32.dll';

function TForm1.add_user_by_name_password(user_name, password:
PChar): ISC_STATUS;
var
    p_status: P_ISC_STATUS;
    status: array[0..19] of ISC_STATUS;
    user_sec_data1: USER_SEC_DATA;
begin
    // have p_status point at the ISC_STATUS array

```



```

p_status := @status[0];

// set up the USER_SEC_DATA elements
user_sec_data1.protocol      := sec_protocol_tcpip;
user_sec_data1.server        := 'aquaman';
user_sec_data1.user_name     := user_name;
user_sec_data1.password      := password;
user_sec_data1.dba_user_name := 'sysdba';
user_sec_data1.dba_password  := 'masterkey';

user_sec_data1.sec_flags := sec_password_spec or sec_server_spec or
                             sec_dba_user_name_spec or
sec_dba_password_spec;

result := isc_add_user(p_status, @user_sec_data1);
end;

function TForm1.mod_user_by_name_password(user_name: PChar; password:
PChar): ISC_STATUS;
var
    p_status: P_ISC_STATUS;
    status: array[0..19] of ISC_STATUS;
    user_sec_data1: USER_SEC_DATA;
begin
    // have p_status point at the ISC_STATUS array
    p_status := @status[0];

    // set up the USER_SEC_DATA elements
    user_sec_data1.protocol      := sec_protocol_tcpip;
    user_sec_data1.server        := 'aquaman';
    user_sec_data1.user_name     := user_name;
    user_sec_data1.password      := password;
    user_sec_data1.dba_user_name := 'sysdba';
    user_sec_data1.dba_password  := 'masterkey';

    user_sec_data1.sec_flags := sec_password_spec or sec_server_spec or
                             sec_dba_user_name_spec or
sec_dba_password_spec;

    result := isc_modify_user(p_status, @user_sec_data1);
end;

function TForm1.del_user_by_name_password(user_name: PChar; password:
PChar): ISC_STATUS;
var

```

```

p_status: P_ISC_STATUS;
status: array[0..19] of ISC_STATUS;
user_sec_data1: USER_SEC_DATA;
begin
  // have p_status point at the ISC_STATUS array
  p_status := @status[0];

  // set up the USER_SEC_DATA elements
  user_sec_data1.protocol      := sec_protocol_tcpip;
  user_sec_data1.server       := 'aquaman';
  user_sec_data1.user_name    := user_name;
  user_sec_data1.password     := password;
  user_sec_data1.dba_user_name := 'sysdba';
  user_sec_data1.dba_password := 'masterkey';

  user_sec_data1.sec_flags := sec_password_spec or sec_server_spec or
                             sec_dba_user_name_spec or
sec_dba_password_spec;

  result := isc_delete_user(p_status, @user_sec_data1);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  add_user_by_name_password(@add_user_name.Text[1], @add_password.
Text[1]);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  mod_user_by_name_password(@mod_user_name.Text[1], @mod_password.
Text[1]);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  del_user_by_name_password(@del_user_name.Text[1], @del_password.
Text[1]);
end;

end.

```



## Article #25849: Displaying 4 digit years in Delphi applications

### Problem:

When date fields are displayed in the data aware controls the year is always displayed with 2 digits. How can I display years with 4 digits instead?

### Solution:

The information in this article applies to:

- \* BDE
- \* InterBase Express (IBX)
- \* ADO

When connecting to an InterBase database date datatypes are always returned to the client using InterBase's date format: ISC\_QUAD. All other databases have their own way of representing 4 digit years. It is the client's responsibility to display the date information in a format suitable for the particular application.

When using Delphi with either the BDE or IBX year data is displayed with 2 digits by default.

There are many ways to change this behavior and customize the displayed date format. Two such ways will be explained here.

### Changing Regional Settings for the System

-----

One way to adjust the display format of date fields is to change the system's settings for displaying dates. In the control panel applet "Regional Settings" there is a Date tab that allows the system's date format to be customized. You can change the value for the combo box "Short date style"

### Changing the Short Date Format for the Application

-----

Another way to adjust the display formats for dates is to set the variable ShortDateFormat. This variable is defined in the Sysutils unit. You can customized the date display format for the application by setting this variable. For more information search on ShortDateFormat in the Delphi help.

Here is an example that sets the variable to display 4 digit years

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ShortDateFormat := 'mm/dd/yyyy';
end;
```

## Changing the Date Display Format for a Particular Field

---

You can also customize the date display format for individual fields. This is accomplished by creating a persistent field component for the field and setting the `DisplayFormat` property for the component.

The easiest way to create a persistent field is to use the Fields editor in Delphi. After dropping your dataset component on a form, right click on the component and select "Fields Editor...". Right clicking in the Fields editor brings up a menu which allows you to "Add all fields"

Once the fields are created you can customize the properties for the date field.

Select the date field's `Field` object from the list. Then in the object inspector you can set the `DisplayFormat` property for the date field. You can also change the property in code as demonstrated in the following snippet of code.

Note: `Query1HIRE_DATE` is the field component that was created from the Fields editor.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Query1HIRE_DATE.DisplayFormat := 'mm-dd-yyyy';
end;
```

Last Modified: 27-OCT-00

## Article #25870: How do I use the InterBase Express (IBX) SQL monitor?

### Problem:

Use the TIBSQLMonitor component to monitor the dynamic SQL that passes through the InterBase server. You can write an application that can view only its own SQL statements, or you can write a generic SQL monitor application that monitors the dynamic SQL of all applications built with InterBase Express (IBX).

Use the TIBSQLMonitor component to watch dynamic SQL taking place in all InterBase data access applications both before and after they have been compiled.

### Solution:

To build a simple SQL monitoring application, follow these steps:

1. Open a new form in Delphi.
2. Add a Memo component to the form and clear the Lines property.
3. Add a TIBSQLMonitor component to the form
4. Double-click the OnSQL event and add the following line of code:  
`Mem1.Lines.Add(EventText);`
5. Compile the application.

You can now start another IBX application and monitor the code.

Last Modified: 19-OCT-00

**Article #25915: BDE Merge Module now available.**

A BDE Merge Module is now available for download from our ftp site at:

<ftp://ftp.borland.com/pub/bde/devsupport/updates/>

There are two files available here:

bdemm5ent.exe

bdemm5pro.exe

for the Enterprise and Professional versions of Delphi. French and German versions are also available:

bdemm5pf.exe

bdemm5ef.exe

bdemm5pg.exe

bdemm5eg.exe

This BDE merge module provides support for the Microsoft Windows Installer (MSI). The Merge Module contains all the BDE files, along with registry and configuration information needed to install those files.

Last Modified: 11-OCT-00

## Article #25954: Where are my components?

### Question

Where are my components?

### Answer

Windows NT and 2000:

Most likely Delphi was installed by an administrator or a user with administrator rights and the user currently logged on does not have administrator rights. The component library is failing to load because it does not have the registry keys available that indicate where the packages are located. Install first as Admin, then do a 'Registry Settings Only' install for each user to get all the necessary DLL's and registry key entries to run Delphi under a specific user profile.

To do this login as the user and run the Delphi install. Around 3 screens in you'll get to the screen where you need to choose the install directories. At the bottom of this screen is a check box to select for Registry Settings Only. Complete the install as usual. It should be much faster since no files are copied.

If it looks like the components are there but the bitmap images are missing then try uninstalling and reinstalling each package. This can be done very easily by going into the Delphi menu option Component | Install Packages. Uncheck each of the packages and then check it again. Say "Yes" to any message boxes that appear asking about the removal of related packages. I recommend you start by removing "Borland Database Components" since this will remove several others as well.

Last Modified: 13-OCT-00



## Article #26068: Resolving DFM corruption issues with CVS.

Many developers have complained of problems using source control with their Builder/Delphi projects. After checking their form files out of source control systems such as CVS and attempting to open the form in Builder or Delphi, the IDE reports an error "Invalid stream format." and refuses to display the form. After some research, it was found that CVS was adding bytes to the form's DFM file, inserting a line-feed (integer value of 10) character before each carriage return character (integer value of 13).

The cause is CVS opening and closing this file as if it were a text file. In Builder/Delphi 4, DFM files were stored as binary files, reason being that a binary DFM could be rendered to a form in the Builder/Delphi IDE faster than a text file could. Through the IDE, a developer could view and edit the DFM as a text file, but it was still stored in binary format.

Version 5 of Builder and Delphi now store DFM files as plain text by default, giving you the option of storing DFMs in binary format. Projects converted from version 4 to version 5 continue to store their DFMs in binary format until set differently.

To prevent your forms from being corrupted by CVS, you should either store your DFMs as text or let CVS know that your DFMs are binary files (consult the CVS documentation for these settings).

If your DFM files have already been corrupted, it might be possible to fix them using this tool: [FIXDFM.EXE](#). This tool is unsupported by Borland so don't send any email if you have problems with it. What it does is search your corrupted binary DFM file for the byte pattern 10 13 (linefeed, carriage return) and removes the 10. Keep in mind the possibility that the program will modify cr/lf combos that were in the DFM originally and not added by CVS.

Last Modified: 13-MAR-02

**Article #26072: Delphi 5 is unable to start due to problems with a missing export in a .BPL file.**

**Question:**

When I try to start Delphi 5, I get an error about:

"The VCLX5.BPL file is linked to missing export VCL50.BPL:@Forms@Finilization\$qqrv"  
or something very similar to this. How can I fix this problem?

**Answer:**

This problem typically occurs when you have a corrupted .bpl file on your system. The solution is to uninstall Delphi, search for any BPL's on your system (a good search query would be: \*50.bpl), and delete the files. If you don't want to delete them (because you may have some custom .bpl's on your system), copy them to a temporary folder that is on your desktop (it will not be on your system path so they won't cause any problems). Then, reinstall Delphi.

Last Modified: 18-OCT-00

## Article #26107: Where is the file dsgnintf.pas?

### Question:

I'm looking for dsgnintf.pas - where is it?

### Answer:

This file comes on the Delphi 5 Professional and Enterprise version CD. It is listed under the directory Runimage\Delphi50\Source\Toolsapi

This file does not come with the Delphi trial version and is not available for download. Please note that this is a design time package and should not be in a run time unit. The Borland/Delphi license agreement prohibits distribution of this file in run time code.

Last Modified: 28-DEC-00

## Article #26111: Manually uninstalling Delphi

### Question:

How do I manually uninstall Delphi?

### Answer:

When uninstalling a program it is best to use the utility Add/Remove Programs found in the Control Panel in Windows, but sometimes this does not work properly. Here is a guide to help you uninstall Delphi manually.

There are four major areas where you will need to delete: The Delphi directory, the Borland Shared directory, the System32 directory and the Windows Registry.

1. The Delphi directory is easy. Just go into Windows Explorer and delete the entire directory called Delphi5. This should be under the directories Program Files\Borland.
2. The Borland Shared directory has more than one possible location but the most common is Program Files\Common Files\Borland Shared. If you are removing all Borland software that uses the BDE from your machine (i.e. Delphi and C++ Builder) you may be able to just delete the entire directory. Be aware that there may be other programs that use the BDE such as Corel paradox. It may be a good idea to always leave the BDE directory unless you are certain.
3. The System32 directory is located in your operating system directory, for example WINNT \System32. It will be the trickiest because it contains many files and only a portion of them are used by Delphi. The key here is to look on your Delphi CD for the System32 directory to find which files to delete. The files contained under the Runimage directory on the CD are in more or less the same directory structure as they are on your machine. The main exception is that the Windows\System32 directory located under Runimage\Delphi50\ on the CD contains items that are actually contained in the OS directory on your machine.

One tip is that Delphi files of the same version have the same date and time, so if you go into the System32 directory and sort by 'Modified' you should be able to group all the Delphi files together so you can delete them all at once.

4. In order to make changes to the registry you must open to utility regedit.exe. Go to the Start menu and select run, then type 'regedit'. Once in the registry open the entry HKEY\_LOCAL\_MACHINE \SOFTWARE\Borland\Delphi. Right click on the version of Delphi you are deleting such as 5.0 and select delete. If desired remove other directories under here as well. Now you should simply do a search on the registry for 'delphi' and 'borland' and make your own judgment as to whether there are other entries you want to delete.

One final step you may want to take is to remove all the file extension associations with Delphi. If you

did a search on the registry in step four then you may have already done this. It can be done by opening Windows Explorer and going to the menu View | Options. The extensions associated with Delphi 5 are: dfm, dpr, dpk and pas.

Last Modified: 24-OCT-00

## Article #26151: Unable to install Delphi 5 after having the Trial edition fully uninstalled

### Question:

After uninstalling the Delphi 5 Trial Edition, I went to install a purchased copy of Delphi 5, but it keeps telling me to uninstall the Trial Edition. How can I get around this problem?

### Answer:

The registry did not properly get cleared by the Delphi 5 Trial uninstaller. Click Start->Run, and type "regedit". Expand HKEY\_CURRENT\_USER, Software, Borland, and then Delphi. Select the key (a folder in the left panel) labeled 5.0 and press delete. You will be prompted to delete the key (select Yes). Next, expand HKEY\_LOCAL\_MACHINE, Software, Borland, and then Delphi. Select the key (a folder in the left panel) labeled 5.0 and press delete. *WARNING: Be very careful when using regedit to not delete any other keys as it can make your system unstable.*

Last Modified: 07-NOV-00

## Article #26402: How to print an HTML document from inside of your application

### Developing Internet Applications

By [Corbin Dunn](#)

Delphi Developer Support

**Question:** How do I print an HTML document from inside an application? Or, how do I print from a TWebBrowser control?

**Answer:** Using Internet Explorer 5 makes this very simple to do, but involves a little work to get it done. Let's say that you want to allow your program to print HTML documents - either from the web directly, or from a file. The main problem programmers encounter is that TWebBrowser does not have a Print method. If you have ever looked inside of MSHTML.pas you will notice that IHTMLWindow3 has a print method that we can call - the only trick is in acquiring an IHTMLWindow3 interface from a TWebBrowser. This can be done by using TWebBrowser component (it doesn't have to be visible), calling WebBrowser.Navigate to go the location you want to print, and then in the NavigateComplete event doing the print:

```
procedure TForm1.WebBrowser_V1NavigateComplete(Sender: TObject;  
    const pDisp: IDispatch; var URL: OleVariant);  
var  
    HTMLDoc: IHTMLDocument2;  
    HTMLWnd: IHTMLWindow2;  
    HTMLWindow3: IHTMLWindow3;  
begin  
    HTMLDoc := (Sender as TWebBrowser).Document as IHTMLDocument2;  
    if HTMLDoc = nil then  
        raise Exception.Create('Couldn't convert the WebBrowser to an  
IHTMLDocument2');  
    HTMLWnd := HTMLDoc.parentWindow;  
    HTMLWindow3 := HTMLWnd as IHTMLWindow3;  
    // Finally, we get to the print method  
    HTMLWindow3.print;  
end;
```

[Download the sample source](#) for this project from [Borland Code Central](#)

Last Modified: 12-DEC-00

## Article #26407: Dynamically changing the MainForm of an Application when it starts.

### Developing Applications

By [Corbin Dunn](#)

Delphi Developer Support

**Question:** I want to dynamically change the MainForm for my Application based on a command line switch or registry setting. How can I do this?

**Answer:** The MainForm for your Application will be the first form created with a call to `Application.CreateForm`. Whenever this form is closed, your whole application will close. You may want to change which form this is based on a command line switch or registry setting. Here is a code sample from the Application source on how to do this:

```
program Project1;

uses
  Forms,
  Classes, // For TComponentClass
  SysUtils, // For FindCmdLineSwitch
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.RES}

var
  InstanceClass: TComponentClass;
  FormReference: TForm;

begin
  Application.Initialize;

  if FindCmdLineSwitch('second', ['-','/'], True) then
  begin
    InstanceClass := TForm2;
    FormReference := Form2;
  end
  else
  begin
    InstanceClass := TForm1;
    FormReference := Form1;
  end;
end;
```



```
Application.CreateForm( InstanceClass, FormReference );  
Application.Run;
```

**end.**

[Download the complete sample project from Borland CodeCentral](#)

Last Modified: 12-DEC-00

**Article #26408: Help pages are coming up blank in Delphi.**

**Question:**

Why are my help topics coming up blank?

**Answer:**

There is a Windows imposed limit to the number of help files that can be opened in one help screen. To fix this, in Delphi, go into Help | Customize and remove help files that you do not think you will need. You should remove them from the Link and Index screens.

Last Modified: 12-DEC-00

## Article #26416: How to move a string into a TMemoryStream or TFileStream.

**Question:** How do I move a string into a TMemoryStream (or a TFileStream)?

**Answer:** Here is a code sample of how to do this with a TMemoryStream. It could easily be extended to work with a TFileStream too:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    SourceString: string;  
    MemoryStream: TMemoryStream;  
begin  
    SourceString := 'Hello, how are you doing today?';  
    MemoryStream := TMemoryStream.Create;  
    try  
        // Write the string to the stream. We want to write from  
SourceString's  
        // data, starting at a pointer to SourceString (this returns a  
pointer to  
        // the first character), dereferenced (this gives the actual  
character).  
        MemoryStream.WriteBuffer(Pointer(SourceString)^, Length  
(SourceString));  
        // Go back to the start of the stream  
        MemoryStream.Position := 0;  
        // Read it back in to make sure it worked.  
        SourceString := 'I am doing just fine!';  
        // Set the length, so we have space to read into  
        SetLength(SourceString, MemoryStream.Size);  
        MemoryStream.ReadBuffer(Pointer(SourceString)^, MemoryStream.  
Size);  
        Caption := SourceString;  
    finally  
        MemoryStream.Free;  
    end;  
end;
```

## **Article #26523: Is Turbo Pascal still available?**

### **Question**

Is Turbo Pascal available for purchase or free download?

### **Answer**

We are no longer selling 16 bit development tools because they are not considered to be Y2K compliant. This includes Turbo Pascal 7 and Turbo Pascal for Windows 1.5 (the latest version of each). Copies of these products may still be available in some stores.

However, Turbo Pascal 5.5 is available for free download from our community site [here](#). Please note this product is no longer supported.

Delphi 5 is our current Pascal windows development tool. It is a descendent of Turbo Pascal and is capable of creating console applications as well as Windows apps. The Standard version of Delphi 5 costs about the same as Turbo Pascal did and is available from our web site at [shop.borland.com](http://shop.borland.com).

Last Modified: 04-JAN-01

## Article #26563: How do I split the editor?

### Question:

How do I split the editor? I want to be able to compare two files side by side.

### Answer:

The editor in C++Builder 5 and Delphi 5 do not directly support this functionality, but by using multiple edit windows and the Save Desktop feature, you can accomplish the same feat.

### Try this:

On Builder/Delphi's main menu, click "View" | "New Edit Window". A new edit window should appear. Close any docked windows you don't need, such as the Class Explorer, and arrange your two edit windows such that they are side by side.

Naturally, you don't want to take the time to rearrange the windows every time you need a split view, so why not save this desktop configuration?

Click "View" | "Desktops" | "Save Desktop...", enter a name such as "Split View", and click OK.

Now, by using either "View" | "Desktops" or the Desktop pull-down menu (it's one of the menu bands on the Builder/Delphi main menu), you can switch between standard single editor view, and split view with a couple of mouse clicks.

Last Modified: 14-MAR-02

## Article #26609: BORDBK50.DLL is missing or not registered.

### Question:

What do I do about the error "BORDBK50.DLL is missing or not registered"?

### Answer:

If the installation does not complete then the cause of the problem is likely due to a software conflict. Some anti-virus software and certain NT Internet Services have been known to prevent bordbk50.dll from being registered with the operating system. In this case the best solution is to try eliminating the software in question and then try reinstalling Delphi 5.

If that is not a viable solution, try manually registering the bordbk50.dll (or borbdk51.dll if you have installed the update pack). Search for this file on your machine to verify that you have it. It is most likely in the folder C:\Program Files\Common Files\Borland Shared\Debugger. Run TREGSVR manually to register the dll. TREGSVR can be found in the Delphi5\bin directory. To register the DLL open a DOS window and change to the directory where bordbk50.dll is located, then type:

```
tregsvr bordbk50.dll
```

You may also need to include the full path to the location of tregsvr.

Last Modified: 11-JAN-01

## Article #26634: What is "Error 113" during installation?

### Question:

What is installation error 113?

### Answer:

This error occurs when there are problems reading files from the CD. Delphi requires a 32-bit CD-ROM driver in order to read long file names. If you have a 16-bit CD-ROM driver you will need to update to 32-bit. The reason this happens with Delphi (and usually not with other programs) is because Delphi installs by copying the Runimage directory, which contains long file names, from the CD to the hard drive. Typically, other installation programs unpack a compressed file. Contact the manufacturer's web site for an update.

To verify that you have a 16-bit driver:

- Check device manager in Control Panel - if no CD-ROM then it's 16-bit.
- Go to Explorer and view the Runimage directory on the CD. If there is a tilde (~) in any of the file names then the name is long and is being truncated.

Possible fix:

- Autoexec.bat and Config.sys - remove any reference to the 16-bit CD-ROM and restart. Windows might load new drivers. If not, manually add the new drivers.

Other Issues:

- This error can happen when someone is installing off of a network and has not actually mapped a drive to the correct location (they simply share the folder). Setup needs to see the Install and Runimage folders in the ROOT directory of a drive. Right click My Computer->Map Network Drive...
- This can also happen if you are trying to do a hard drive install, but have not put the files in the ROOT of the drive.

Last Modified: 15-JAN-01

**Article #26667: Saving your project's breakpoint settings.**

**Question:**

Whenever I close and re-open my project, all of my breakpoints are lost. Is there any way to save my breakpoints?

**Answer:**

Yes. Under **Tools/Environment Options/AutoSave options** check the box labeled **Project Desktop**. The IDE online help says that this option saves what editor files you had open, but does not tell you it also saves your breakpoint options.

Last Modified: 14-MAR-02



## Article #26668: How to set the default form font in Delphi.

### Question:

How can I set the default form font in Delphi?

### Answer:

The value for the default font is stored in the registry. Open the registry editor by entering **regedit** at the RUN prompt. Go to the key

HKEY\_CURRENT\_USER\Software\Borland\Delphi\5.0

Here you will add a new key called FormDesign (with no space). To do right click on the 5.0 and chose New | Key. Name the key **FormDesign**. Now right click on this new key and choose New | String Value. Name this string value **DefaultFont** (no space). Now right click on this new string value entry in the right window and choose Modify. Enter the name of the default font you would like, followed by a comma and the default size. For example:

Arial,10

Now the next time you open Delphi this should be the default font for your forms. You can even go a little further with this if you wish by adding other font properties such as

Arial,10, bold, italic

Last Modified: 19-JAN-01

## Article #26731: How to display the current user name.

### QUESTION:

How can I get the name of the user currently logged onto the system.

### ANSWER:

The code sample below displays the current user's name.

```
function CurrentUserName:String;  
var  
    u: array[0..127] of Char;  
    sz:DWord;  
begin  
    sz:=SizeOf(u);  
    GetUserName(u,sz);  
    Result:=u;  
end;  
  
procedure TForm1.Button1Click(Sender:  
TObject);  
begin  
    ShowMessage(CurrentUsername);  
end;
```

Last Modified: 02-FEB-01

**Article #26735: How to use the Find Error entry under the search heading of the menu bar.**

**QUESTION:**

How do I use the Find Error tool?

**ANSWER:**

If you are getting an AV, and are not sure what line this is happening on. You can use this feature to help scope down where this error might be occurring. This is especially helpful for tracking down, those irregular AV's.

The first thing to do when the AV occurs, is to write down the address. This Hex address we are interested in is the number that occurs in this sentence: "Access violation at address 0043F156 in module..." However, to use this properly, the application must be in Debug mode.

You can put the application into debug mode; by setting a break point some place in the application. Next, go to Search -> Find Error, after selecting this, a dialog window will show up. Put the Hex address number in the box "0043F156".

You actually don't need to put in the 004; this number specifies the preferred load address of the compiled image. You can check this number by going to Project->Options, then viewing the linker tab. It will appear in the Memory sizes portion off of the Linker tab.

Now hit the OK button, it should stop on the line that caused the error.

Last Modified: 05-FEB-02

## Article #26752: How to detect if the current user is logged in administrator.

### QUESTION:

How can I check if the current user is logged in as administrator?

### ANSWER:

Below is a code sample to check if the current user is logged in as administrator.

```
function IsAdmin: Boolean;
var
  hAccessToken: THandle;
  ptgGroups: PTokenGroups;
  dwInfoBufferSize: DWORD;
  psidAdministrators: PSID;
  x: Integer;
  bSuccess: BOOL;
begin
  Result := False;
  bSuccess := OpenThreadToken(GetCurrentThread, TOKEN_QUERY, True,
    hAccessToken);
  if not bSuccess then
  begin
    if GetLastError = ERROR_NO_TOKEN then
      bSuccess := OpenProcessToken(GetCurrentProcess, TOKEN_QUERY,
        hAccessToken);
  end;
  if bSuccess then
  begin
    GetMem(ptgGroups, 1024);
    bSuccess := GetTokenInformation(hAccessToken, TokenGroups,
      ptgGroups, 1024, dwInfoBufferSize);
    CloseHandle(hAccessToken);
    if bSuccess then
    begin
      AllocateAndInitializeSid(SEcurity_NT_AUTHORITY, 2,
        SECURITY_BUILTIN_DOMAIN_RID, DOMAIN_ALIAS_RID_ADMINS,
        0, 0, 0, 0, 0, 0, psidAdministrators);
      {$R-}
      for x := 0 to ptgGroups.GroupCount - 1 do
```

```
    if EqualSid(psidAdministrators, ptgGroups.Groups[x].Sid) then
    begin
        Result := True;
        Break;
    end;
    {$R+}
    FreeSid(psidAdministrators);
end;
FreeMem(ptgGroups);
end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if isAdmin then
        begin
            ShowMessage('Logged in as Administrator');
        end;
end;
```

Last Modified: 02-FEB-01

## Article #26754: How to detect the computer name.

### QUESTION:

How can I get the name of the computer my application is running on?

### ANSWER:

On a Form, add a TButton and a TEdit box. Declare your function and add the GetCompName code, shown below, to the implementation section of the Unit. Next, in the TButton's OnClick event add the code shown. Now run the project and click the button.

```
function GetCompName: string;

var
    Form1: TForm1;

implementation

{$R *.dfm}

function GetCompName: string;
var
    dwLength: dword;
begin
    dwLength := 253;
    SetLength(result, dwLength+1);
    if not Windows.GetComputerName(pchar(result), dwLength) then
        raise exception.create('Computer name not detected');
    result := pchar(result);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.Text := GetCompName;
end;
```

## Article #26889: Why does my Application run slowly under Linux?

### Question

Why does my Linux application run so much slower than my windows application?

### Answer

Linux performs fixups of external functions at runtime instead of at load time. This creates noticeable performance loss. To work around this, use static linking of your exe's.

You might also notice a significant performance improvement if you run your application without debugging turned on, or outside the IDE.

Last Modified: 13-NOV-01

## Article #26890: Error -10 while installing

### Question

I got an Error -10 when I tried to install Kylix, why?

### Answer

The Error -10 is a generic install error. The usual cause of this is due to a previous version of Kylix that wasn't uninstalled completely. Check the RPM database to see if there are still components from Kylix installed, use the command `rpm -qa | grep kylix` to check for installed packages. If there are any packages still on the system, use `rpm -e <PACKAGE>` to remove them.

This can also be caused by a bug in the RPM manager that causes it to incorrectly check package dependencies. If this is the cause, then you can use the `-m` switch on the installer.

Last Modified: 29-SEP-01



## Article #26891: What are the Differences in the install paths for Kylix?

### Question

What is the difference between the Install Path and the Sym Link Path?

### Answer

The Install path is where all the kylix files will be installed, whereas the symlink path is where symbolic links to those files will be installed. The links should be installed to a location that all the users of kylix will have access. Note, if you are installing as the root, the the installer will want to install Kylix to the /root/kylix directory. This is generally not the best option. Kylix should be installed to a common directory to all users that are going to be using it.

Last Modified: 25-MAY-01

## Article #26937: Why am I getting the error "Operation must use an updateable query"?

### Question:

Why am I getting the error "Operation must use an updateable query" when editing a Paradox table using the Microsoft ODBC driver on my deployed application?

### Answer:

The Microsoft ODBC driver uses the Borland Database Engine (BDE) for updateable queries. It does not use it for read-only queries. One solution is to deploy the BDE with your application. A better solution is to use the Paradox driver that comes with the BDE.

This ODBC driver is used by Microsoft to import Paradox tables into Access. An import is read-only so it doesn't need the BDE. If you attempt to edit a Paradox table in MS Access without the BDE on your machine you should receive an error as well. We have been asked how to acquire the BDE in order to use Paradox tables in Access. Unfortunately the BDE is not sold as a stand-alone product. It is only available with Delphi and C++Builder.

If you are using ADO and TClientDataset you may receive the error: "Object or provider is not capable of performing the requested operation". This error can occur even if the BDE is on the machine. We have not verified the exact cause of this error but the solution is the same: don't use the Microsoft Paradox ODBC driver for updates - use the native Paradox driver instead.

Last Modified: 21-FEB-01

## Article #27022: How do I get a TRichEdit to scroll in code?

### Question:

How do I get a TRichEdit to automatically scroll down as I add new lines?

### Answer:

This functionality is included in a TMemo but not a TRichEdit. You just need to add one line of code to get it to scroll down:

```
begin
  RichEdit1.Lines.Add('New text');
  RichEdit1.Perform(EM_SCROLL, SB_LINEDOWN, 0);
end;
```

Last Modified: 06-MAR-01

## **Article #27031: Error "Application is not licensed to use this feature" when running a Delphi application.**

### **Question:**

What is the cause of the error "Application is not licensed to use this feature"?

### **Answer:**

This error occurs when you try to use components from one version of Delphi, such as Enterprise, with a different version that does not include those components, such as Standard. For example if you try to install the database components from the Enterprise version of Delphi into a Standard installation of Delphi.

Another possible cause is if you have the Standard version of Delphi installed on your machine and you install the Professional or Enterprise version without first uninstalling the Standard version. This error may also occur if you move a project from one version to another and keep the same compiled units (.dcu). It is best to delete the .dcu files and let them be recreated automatically.

*Note: It is fine to have two different release versions of Delphi on your machine, such as Delphi 4 and Delphi 5, but not two versions of the same release such as Delphi 5 Standard and Delphi 5 Enterprise.*

Last Modified: 07-MAR-01

## Article #27039: Setting a Kylix TListView component's subitem images to nothing

**Question:** In Kylix, I have a TListView with a ViewStyle of vsReport that has the Images property set to a valid TImageList. Whenever I add subitems to an Item in the TListView, they have images (icons) set to them. How can I prevent this from happening?

**Answer:** This is due to a bug in the CLX VCL. There are two workarounds that you can do. First, you can modify the VCL source. To do this, open up QComCtrls.pas and save it in your projects current directory. Then, look for the following procedure and add the suggested lines of code:

```
procedure TCustomViewItem.ImageIndexChange(ColIndex, NewIndex:
Integer);
var
    Pixmap: QPixmapH;
begin
    if ViewControlValid and HandleAllocated and Assigned(ViewControl.
Images) then
        begin
            Pixmap := ViewControl.Images.GetPixmap(NewIndex);
            if Assigned(Pixmap) then
                QListViewItem_setPixmap(Handle, ColIndex, Pixmap)
                // Remove the ; from the last line and add the following fix:
            else
                begin
                    Pixmap := QPixmap_create;
                    QListViewItem_setPixmap(Handle, ColIndex, Pixmap);
                    QPixmap_destroy(Pixmap);
                end;
                // End of the fix
            end;
        end;
end;
```

The above solution will not work if you build with runtime packages. The fix for this must be done each time you add an item to the TListView. First, add Qt to your uses list, as we need to make some calls directly to the Qt library. Then, add the following work around whenever you add an Item to the TListView:

```
var
    Pixmap: QPixmapH;
begin
    with ListView.Items.Add do
        begin
```

```
Caption := 'Some Caption for the Item';
SubItems.Add('Some sub item');
// Work around for a bug in setting the image to nothing
Pixmap := QPixmap_create;
QListViewItem_setPixmap(Handle, 1, QPixmap);
QPixmap_destroy(Pixmap);
// End work around for the bug
end;
end;
```

The 1 passed to `QListViewItem_setPixmap` is the index of the `SubItem` that you want fixed, with the first one being 1, the second 2, etc (so, it is NOT a 0-based index, but rather a 1-based index).

Last Modified: 09-NOV-01

## Article #27041: TIBLicensingService component not found on Component Pallet.

TIBLicensingService is not installed on the IBX component pallet. If you need to use this component in your application it must be created dynamically.

Delphi example:

```
uses IBServices;  
  
var  
    IBLicensingService1 : TIBLicensingService;  
  
...  
  
IBLicensingService1 := TIBLicensingService.Create(nil);
```

Last Modified: 19-MAR-01

## Article #27043: Trying to revoke a drop target that has not been registered with TWebBrowser.

TWebBrowser.ExecWB can generate the error "Trying to revoke a drop target that has not been registered" under 2 different conditions:

1. TWebBrowser is not yet ready to accept the command. For following may generate this error:

```
Navigate('e:\pictures\picture1.jpg');  
ExecWB(OLECMDID_PRINT, OLECMDEXECOPT_DONTPROMPTUSER);
```

To resolve the problem use QueryStatusWB and wait for it to return OLECMDF\_SUPPORTED + OLECMDF\_ENABLED. Below is an example that fixes the problem:

```
Navigate('e:\pictures\picture1.jpg');  
while QueryStatusWB( OLECMDID_PRINT) <> OLECMDF_SUPPORTED +  
OLECMDF_ENABLED do  
Forms.Application.ProcessMessages;  
ExecWB(OLECMDID_PRINT, OLECMDEXECOPT_DONTPROMPTUSER);
```

2. Even though you use QueryStatusWB, Copy and Paste may still fail. Add the following to the end of your unit to resolve the problem:

```
initialization  
OleInitialize(nil);
```

```
finalization  
OleUninitialize;
```

Last Modified: 19-MAR-01



## Article #27067: Trouble Shooting Errors Received After Adding a Unit to the Uses Section of Your Unit

### Question:

Why am I getting the error "file not found: 'xxx.dcu' " when I add a unit to the uses section, if I know it should be there?

### Answer:

Here are some possibilities. First go to the menu option Tools | Environment Options | Library and make sure the Library Path includes \$(DELPHI)\lib.

A second possibility is that the unit name may have changed between Delphi and Kylix. For example the unit Printers is now QPrinters.

Finally, remember that Linux is case sensitive, so qprinters.dcu and Qprinters.dcu would not be found in the example above - only QPrinters.dcu.

Last Modified: 27-APR-01

## Article #27128: How to custom draw menu items

### Custom drawing of menu items

By [Corbin Dunn](#)

Delphi Developer Support

First, take the opportunity to [download the complete source](#) for this project from CodeCentral.

While working an application I wanted to have a custom drawn popup menu that separated the menu into a series of groups. This would allow the user to easily differentiate each group. With Delphi, this is very easy to do. I put a **TPopupMenu** on my form and set the **OwnerDraw** property to True. I then set the **OnDrawItem** event for all of my "header" menu items. Then, I added the following code:



```
procedure TForm1.CustomMenuDrawItem(Sender: TObject; ACanvas:
TCanvas;
  ARect: TRect; Selected: Boolean);
var
  LeftPos: Integer;
  TopPos: Integer;
  TextLength: Integer;
  Text: string;
begin
  Text := (Sender as TMenuItem).Caption;
  ACanvas.Brush.Color := clGray;
  ACanvas.FillRect(ARect);
  ACanvas.Font.Color := clWhite;
  ACanvas.Font.Style := [fsBold];
  // Draw right in the middle of the menu
  TopPos := ARect.Top +
    (ARect.Bottom - ARect.Top - ACanvas.TextHeight('W')) div 2;
```

```
TextLength := Length(Text);  
if TextLength > (ARect.Right - ARect.Left) then  
    LeftPos := ARect.Left + 3  
else  
    LeftPos := ARect.Left + (ARect.Right - ARect.Left -  
        ACanvas.TextWidth(Text)) div 2;  
ACanvas.TextOut(LeftPos, TopPos, Text);  
end;
```

Pretty simple!

Last Modified: 26-MAR-01

## Article #27129: Selecting an item in a TListView when the checkbox is checked

### Selecting an item in a TListView when the checkbox is checked

By [Corbin Dunn](#)

Delphi Developer Support

You may have noticed some peculiar behavior with a TListView that has check boxes. Checking a check box does not select the underlying item. There is any easy way to solve this problem. In the **OnMouseDown** event handler of the TListView add the following code:

```
procedure TForm1.ListView1MouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Item: TListItem;
begin
  // Did we hit a checkbox? If so, make sure the item it
  // represents is selected.
  Item := ListView1.GetItemAt(X, Y);
  if Item <> nil then
    if ListView1.Selected <> Item then
      ListView1.Selected := Item;
end;
```

This should solve that problem!

If you want some example source code with this solution in it, [download the project](#) from CodeCentral.

Last Modified: 26-MAR-01

## Article #27138: How to create Data Link File with Windows 2000

ADO comes installed on Windows 2000. You have to download it and install it for earlier Windows versions. When you install it, Data Link File shows up in the New Menu under Windows Explorer, making it quite easy to create a Data Link File. Below are two ways to easily create a Data Link file with Windows 2000 and Windows Explorer.

1. Select New | Text Document. Rename the resulting file to have a UDL extension. Double click on the file.
2. Open Notepad and place the following in it:

```
REGEDIT4
[HKEY_CLASSES_ROOT\.UDL\ShellNew]
@="MSDASC"
"Data"=hex:ff,fe,5b,00,6f,00,6c,00,65,00,64,00,62,00,5d,00,0d,00,0a,00,3b,00,
20,00,45,00,76,00,65,00,72,00,79,00,74,00,68,00,69,00,6e,00,67,00,20,00,61,
00,66,00,74,00,65,00,72,00,20,00,74,00,68,00,69,00,73,00,20,00,6c,00,69,00,
6e,00,65,00,20,00,69,00,73,00,20,00,61,00,6e,00,20,00,4f,00,4c,00,45,00,20,
00,44,00,42,00,20,00,69,00,6e,00,69,00,74,00,73,00,74,00,72,00,69,00,6e,00,
67,00,0d,00,0a,00,50,00,52,00,4f,00,56,00,49,00,44,00,45,00,52,00,3d,00,6d,
00,73,00,64,00,61,00,73,00,71,00,6c,00,3b,00,0d,00,0a,00
```

Save the file with a .REG extension. Double click on the file. This will cause this to be automatically be entered in the appropriate place in your registry. After restarting Windows Explorer, Microsoft Data Link will be in the New menu.

## Article #27166: Assigning an Oracle BLOB greater than 32K must be done using AsMemo

If you create a table as such:

```
create table foo  
(f1 int not null,  
f2 long)
```

then insert data such as:

```
insert into foo values (:1, :2)
```

when assigning values to the second parameter you must the AsMemo property. Failing to do this may result in an Access Violation in SQLORA32.DLL at 4D629F73.

Example:

```
var  
s : string;  
counter : integer;  
begin  
s := "";  
for counter := 1 to 32768 do  
s := s + 'A';  
with Query1 do  
begin  
Params[0].Value := 1;  
//Params[1].Value := s; //This will cause an AV when ExecSQL is executed.  
//Params[1].AsString := s; //This will cause an AV when ExecSQL is executed.  
//Params[1].AsBlob := s; //Generates Oracle Error  
Params[1].AsMemo := s;  
ExecSQL;  
end;
```

Last Modified: 10-APR-01

## Article #27227: WHERE clause with greater than 256 ANDs or ORs generates size of optimizer block exceeded error

Q. What can cause the error Size of optimizer block exceeded?

A. This error occurs when the WHERE clause of an UPDATE or SELECT has greater than 256 ANDs or ORs.

Example:

```
create table foo( f1 int,f2 int,f3 int,f4 int,f5 int,f6 int,f7 int,f8 int,f9 int,f10 int,f11 int,f12 int,f13 int,f14
int,f15 int,f16 int,f17 int,f18 int,f19 int,f20 int,f21 int,f22 int,f23 int,f24 int,f25 int,f26 int,f27 int,f28 int,
f29 int,f30 int,f31 int,f32 int,f33 int,f34 int,f35 int,f36 int,f37 int,f38 int,f39 int,f40 int,f41 int,f42 int,f43
int,f44 int,f45 int,f46 int,f47 int,f48 int,f49 int,f50 int,f51 int,f52 int,f53 int,f54 int,f55 int,f56 int,f57 int,
f58 int,f59 int,f60 int,f61 int,f62 int,f63 int,f64 int,f65 int,f66 int,f67 int,f68 int,f69 int,f70 int,f71 int,f72
int,f73 int,f74 int,f75 int,f76 int,f77 int,f78 int,f79 int,f80 int,f81 int,f82 int,f83 int,f84 int,f85 int,f86 int,
f87 int,f88 int,f89 int,f90 int,f91 int,f92 int,f93 int,f94 int,f95 int,f96 int,f97 int,f98 int,f99 int,f100 int,
f101 int,f102 int,f103 int,f104 int,f105 int,f106 int,f107 int,f108 int,f109 int,f110 int,f111 int,f112 int,
f113 int,f114 int,f115 int,f116 int,f117 int,f118 int,f119 int,f120 int,f121 int,f122 int,f123 int,f124 int,
f125 int,f126 int,f127 int,f128 int,f129 int,f130 int,f131 int,f132 int,f133 int,f134 int,f135 int,f136 int,
f137 int,f138 int,f139 int,f140 int,f141 int,f142 int,f143 int,f144 int,f145 int,f146 int,f147 int,f148 int,
f149 int,f150 int,f151 int,f152 int,f153 int,f154 int,f155 int,f156 int,f157 int,f158 int,f159 int,f160 int,
f161 int,f162 int,f163 int,f164 int,f165 int,f166 int,f167 int,f168 int,f169 int,f170 int,f171 int,f172 int,
f173 int,f174 int,f175 int,f176 int,f177 int,f178 int,f179 int,f180 int,f181 int,f182 int,f183 int,f184 int,
f185 int,f186 int,f187 int,f188 int,f189 int,f190 int,f191 int,f192 int,f193 int,f194 int,f195 int,f196 int,
f197 int,f198 int,f199 int,f200 int,f201 int,f202 int,f203 int,f204 int,f205 int,f206 int,f207 int,f208 int,
f209 int,f210 int,f211 int,f212 int,f213 int,f214 int,f215 int,f216 int,f217 int,f218 int,f219 int,f220 int,
f221 int,f222 int,f223 int,f224 int,f225 int,f226 int,f227 int,f228 int,f229 int,f230 int,f231 int,f232 int,
f233 int,f234 int,f235 int,f236 int,f237 int,f238 int,f239 int,f240 int,f241 int,f242 int,f243 int,f244 int,
f245 int,f246 int,f247 int,f248 int,f249 int,f250 int,f251 int,f252 int,f253 int,f254 int,f255 int,f256 int,
f257 int);
```

```
update foo set f1=1 where f1=1 and f2=2 and f3=3 and f4=4 and f5=5 and f6=6 and f7=7 and f8=8 and
f9=9 and f10=10 and f11=11 and f12=12 and f13=13 and f14=14 and f15=15 and f16=16 and f17=17
and f18=18 and f19=19 and f20=20 and f21=21 and f22=22 and f23=23 and f24=24 and f25=25 and
f26=26 and f27=27 and f28=28 and f29=29 and f30=30 and f31=31 and f32=32 and f33=33 and f34=34
and f35=35 and f36=36 and f37=37 and f38=38 and f39=39 and f40=40 and f41=41 and f42=42 and
f43=43 and f44=44 and f45=45 and f46=46 and f47=47 and f48=48 and f49=49 and f50=50 and f51=51
and f52=52 and f53=53 and f54=54 and f55=55 and f56=56 and f57=57 and f58=58 and f59=59 and
f60=60 and f61=61 and f62=62 and f63=63 and f64=64 and f65=65 and f66=66 and f67=67 and f68=68
and f69=69 and f70=70 and f71=71 and f72=72 and f73=73 and f74=74 and f75=75 and f76=76 and
f77=77 and f78=78 and f79=79 and f80=80 and f81=81 and f82=82 and f83=83 and f84=84 and f85=85
```

and f86=86 and f87=87 and f88=88 and f89=89 and f90=90 and f91=91 and f92=92 and f93=93 and f94=94 and f95=95 and f96=96 and f97=97 and f98=98 and f99=99 and f100=100 and f101=101 and f102=102 and f103=103 and f104=104 and f105=105 and f106=106 and f107=107 and f108=108 and f109=109 and f110=110 and f111=111 and f112=112 and f113=113 and f114=114 and f115=115 and f116=116 and f117=117 and f118=118 and f119=119 and f120=120 and f121=121 and f122=122 and f123=123 and f124=124 and f125=125 and f126=126 and f127=127 and f128=128 and f129=129 and f130=130 and f131=131 and f132=132 and f133=133 and f134=134 and f135=135 and f136=136 and f137=137 and f138=138 and f139=139 and f140=140 and f141=141 and f142=142 and f143=143 and f144=144 and f145=145 and f146=146 and f147=147 and f148=148 and f149=149 and f150=150 and f151=151 and f152=152 and f153=153 and f154=154 and f155=155 and f156=156 and f157=157 and f158=158 and f159=159 and f160=160 and f161=161 and f162=162 and f163=163 and f164=164 and f165=165 and f166=166 and f167=167 and f168=168 and f169=169 and f170=170 and f171=171 and f172=172 and f173=173 and f174=174 and f175=175 and f176=176 and f177=177 and f178=178 and f179=179 and f180=180 and f181=181 and f182=182 and f183=183 and f184=184 and f185=185 and f186=186 and f187=187 and f188=188 and f189=189 and f190=190 and f191=191 and f192=192 and f193=193 and f194=194 and f195=195 and f196=196 and f197=197 and f198=198 and f199=199 and f200=200 and f201=201 and f202=202 and f203=203 and f204=204 and f205=205 and f206=206 and f207=207 and f208=208 and f209=209 and f210=210 and f211=211 and f212=212 and f213=213 and f214=214 and f215=215 and f216=216 and f217=217 and f218=218 and f219=219 and f220=220 and f221=221 and f222=222 and f223=223 and f224=224 and f225=225 and f226=226 and f227=227 and f228=228 and f229=229 and f230=230 and f231=231 and f232=232 and f233=233 and f234=234 and f235=235 and f236=236 and f237=237 and f238=238 and f239=239 and f240=240 and f241=241 and f242=242 and f243=243 and f244=244 and f245=245 and f246=246 and f247=247 and f248=248 and f249=249 and f250=250 and f251=251 and f252=252 and f253=253 and f254=254 and f255=255 and f256=256 and f257=257;

Generates the error:

Statement failed, SQLCODE = -904

size of optimizer block exceeded

This error has been reported to occur in conjunction with the error unknown username or password. This error is not related to the problem.

A common cause for having an UPDATE statement that includes every field in a table occurs when using a TTable or TQuery with Delphi or C++ Builder and the UpdateMode property is set to UpWhereAll. Changing the UpdateMode to upWhereChanged or upWhereKeyOnly solves the problem for most applications. If you need further customizations, a TUpdateSQL component can be used in conjunction with your TQuery or TTable to further control which fields are used in the WHERE clause of your UPDATE statement.

Last Modified: 17-APR-01



## Article #27255: Finding the column and row of the cursor in a TRichEdit

### Finding the column and row of the cursor in a TRichEdit

By [Corbin Dunn](#)

Delphi Developer Support

You may have come across the problem of trying to find out the current column and row position of the cursor in a rich edit. This is pretty easy to do, and the best time to do it is in the **Application.OnIdle** event:

```
procedure TForm1.ApplicationEvents1Idle(Sender: TObject;  
    var Done: Boolean);  
var  
    Row, Col: Integer;  
begin  
    // Find out what line we are on, based on the current selection's  
    // end (start + length).  
    Row := SendMessage(Richedit1.Handle, EM_EXLINEFROMCHAR, 0,  
        RichEdit1.SelStart + RichEdit1.SelLength);  
    // Find the column, from the selection end, minus the char we are  
on  
    Col := RichEdit1.SelStart + RichEdit1.SelLength -  
        SendMessage(Richedit1.Handle, EM_LINEINDEX, -1, 0);  
    // Display it in a status bar  
    StatusBar1.SimpleText := IntToStr(Row+1) + ' ' + IntToStr(Col+1)  
        + ' ' + DateTimeToStr(Now);  
end;
```

[Download the complete source from Borland CodeCentral](#)

Last Modified: 27-APR-01

**Article #27265: Corrupted help files.**

**Question:**

My Delphi 5 help files seem to be corrupted. When I use the index, and select a topic, often an entirely different topic comes up. Is there some way to repair the help system?

**Answer:**

Usually, this is the result of a bad .gid file. WinHelp creates the .gid file the first time you open the help, and stores information on how to locate topics quickly. It's a hidden file by default so you may need to go into Folder Options to select the option to see hidden files. Try deleting all .gid files in your help directory and see if the problem goes away.

Last Modified: 27-APR-01

**Article #27266: BDE on multiprocessor machines.**

**Question:**

Are there any know issues with the BDE on multiprocessor machines?

**Answer:**

Yes. We have had reports that in very high traffic situations the BDE can cause an application to freeze or possibly give BDE Error \$2A04. Your best solution if you are faced with this situation is probably to switch to a single processor machine or set the affinity mask of your process to run on just one processor. You may also want to consider a different form of connectivity such as ADO.

Last Modified: 27-APR-01

## Article #27267: Setting process affinity for your application.

### Question:

Is there a way to make my application run on one processor of a multiprocessor machine?

### Answer:

Yes. Here are a couple ways you can set process affinity. The first is done while your application is running. Go into the Task Manager and click the Processes tab. Right click on your process and choose Set Affinity. Clear either CPU0 or CPU1.

The second is done in code. Here is an example of setting and getting process affinity:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    vHandle : Cardinal;  
    vMaskProcess, vMaskSystem : cardinal;  
begin  
    vHandle := Form1.Handle;  
    SetProcessAffinityMask(vHandle, 0); // or 1  
    GetProcessAffinityMask(vHandle, vMaskProcess, vMaskSystem);  
    showmessage(inttostr(vMaskProcess));  
    showmessage(inttostr(vMaskSystem));  
end;
```

Last Modified: 17-MAY-01

## Article #27298: Finding product information from Borland's website for previous versions of Delphi

### QUESTION:

**Where can I find product information for previous versions of Delphi?**

### ANSWER:

You can find information on previous versions of Delphi by following these steps:

1. Start out at Borland's Home Page: [www.borland.com](http://www.borland.com)

2. Near the top, you should see the following:

Corporate Info | Products | Downloads | Services | Support | Community

Click on [Support](#)

3. You are now at the Developer Support Main Page. To the left, you will see a listing of various products underneath Updates/Patches.

Click on [Delphi](#)

4. On the top left, underneath Corporate Info, you will see Delphi. Underneath Delphi, you will see Datasheet(PDF), What's New and various other links. Click on [Previous Versions](#)

5. You can now select the previous version of Delphi you are wanting information about and navigate accordingly.

Last Modified: 10-MAY-01

**Article #27308: Does Delphi 5 include Resource Workshop and InstallShield Express?**

**QUESTION:**

**Does Delphi 5 include Resource Workshop and InstallShield Express?**

**ANSWER:**

InstallShield and Resource Workshop are provided with Delphi 5 Professional and Enterprise but are not included in the Standard Version.

Last Modified: 10-MAY-01

## **Article #27312: What databases can be used with dbExpress?**

### **Question**

Will dbExpress allow access to file based databases such as DBase, Paradox and FoxPro?

### **Answer**

No. It currently works with DB2, Interbase, MySQL and Oracle.

Last Modified: 10-MAY-01

## Article #27321: How to get UpdateMode to work for TQuery on a SQL Server 7.0 table.

### QUESTION:

Why are all the fields on my SQL Server table used for identifying a record even though I have UpdateMode set to upWhereKeyOnly?

### ANSWER:

The problem is that the BDE is not identifying a primary key for your table. There may be two things required to solve this problem. The first is to verify that the table has been created with a primary key declared. You can verify that a primary key exists using SQL Explorer and viewing the text of the table, or by looking at the constraints placed on that table.

If you have verified that a primary key exists and you are still getting the error then you may need to change the SQL statement that is sent to the server. Try specifying the owner of the table before the table name (separated with a . ). For example if the owner is SYSADMIN and the table is Foo, then the SQL statement might look something like:

```
select * from SYSADMIN.Foo;
```

Last Modified: 15-MAY-01



## Article #27329: How to retrieve the position of the cursor.

### QUESTION:

How can I get the current coordinates of the mouse?

### ANSWER:

The windows unit provides a wrapper around the Windows API function, `GetCursorPos`. Passing a `TPoint` into the function will retrieve the coordinates of the cursor. The following code illustrates how to get the values of the coordinates of the cursor from a button click.

```
procedure Form1.Button1Click(Sender: TObject);
var
  foo: TPoint;
begin
  GetCursorPos(foo)
  ShowMessage( '(' + IntToStr(foo.X) + ' , ' + IntToStr( foo.Y ) + ')' );
end;
```

Last Modified: 17-MAY-01

**Article #27349: I installed my application with InstallShield and there are files missing.**

**Question:**

When I deploy my application with Installshield and try to run it, I get an error "Can't find VCL50.bpl. What's going on?"

**Answer:**

You have built your application using runtime packages. If these packages are not installed on the target machine, it will return an error like the above. You need to either install these packages or change your Project|Options|Packages settings to build your executable without these packages.

Last Modified: 05-FEB-02

## Article #27369: Why won't my server socket reconnect?

### Question

Why won't my server socket reconnect ?

### Answer

The Linux TCP stack has a default wait of 2 minutes before it will let you bind another server socket to that port. This can be circumvented by adding a `SetSocketOpt()` call in the `OnCreateHandle` event. The actual is as follows.

```
Var  
    I : integer;  
begin  
    SetSocketOpt(sock, SOL_SOCKET, SO_REUSEADDR, @I, sizeof(I));  
end;
```

Last Modified: 04-JUN-01

## Article #27370: Why won't the virtual key codes work in my program?

### Question

How can I determine what virtual key was pressed in Kylix?

### Answer

First you have to add QT to the uses clause of your program, then you can start using the virtual key codes in Kylix as described in the help files, i.e. `Key_Return`.

Last Modified: 04-JUN-01

**Article #27373: VisiBroker applications created with Delphi 5 Enterprise on Windows 2000**

**QUESTION:**

**I installed VisiBroker that comes with Delphi 5 Enterprise and I can't get my VisiBroker applications to work on Windows 2000. What is wrong?**

**ANSWER:**

The version of VisiBroker that ships with Delphi 5 Enterprise is VisiBroker 3.3 for C++. As you can see from the following supported platform page <http://www.borland.com/techpubs/visibroker/platforms/previous-platforms.html#vbcpp33> Windows 2000 is not a supported operating system. However, Delphi 6 Enterprise comes with VisiBroker 4.1 which does work on Windows 2000.

Last Modified: 05-FEB-02

**Article #27375: Can you use the Naming Service with the version of VisiBroker that comes with Delphi 5 Enterprise?**

**QUESTION:**

**I am interested in using the Naming Service with Visibroker and was wondering if this feature is supported with VisiBroker for Delphi 5 Enterprise?**

**ANSWER:**

The version of VisiBroker that ships with Delphi 5 Enterprise is VisiBroker 3.3 for C++. This means you have all the capabilities that are offered with VisiBroker 3.3, one of them being the Naming Service. For a complete listing of all the features and instructions on how to use them, please consult the VisiBroker for C++ 3.3 documentation found at: <http://www.borland.com/techpubs/books/vbcpp/vbcpp33/index.html>

Last Modified: 01-JUN-01

## Article #27379: Install option is inactive for the Trial Version of Delphi 5

### QUESTION:

I received the install password for the Delphi 5 trial, but the Install option is inactive. Why is that?

### ANSWER:

A lot of customers have had issues with installing the trial version due to a misunderstanding of Step 2: Register Now ! People click on the Register Now option and they are routed to the Borland Community site where they create a member account. While creating a member account you choose your username and a password, however this is not the password nor the registration that is required when installing the Delphi 5 Trial. After becoming a member of the Borland Community Site, you actually have to go to the community site and log in. Only then are you presented with the next step - completing a brief survey. After completing the survey, you will be at the next step, which is where you actually obtain your Trial Edition Password. In this step you must enter the System ID that you obtained at the very beginning of your installation (Step 1: Get your System ID to unlock the Trial.) Once you enter the System ID and enter your e-mail address (be sure and enter the proper e-mail address), you will be e-mailed your trial edition password and then Step 3 of the install should be active.

Last Modified: 01-JUN-01

## Article #27394: Borland Kylix Install Frequently Asked Questions

### Error -10 when trying to install Kylix

The Error -10 is a generic install error. The usual cause of this is due to a previous version of Kylix that wasn't uninstalled completely. You will need to check the RPM database to see if there are still components from Kylix installed. This can be done with command **rpm -qa | grep kylix**. For each of the resulting packages (if there are none, then Kylix is completely uninstalled), use **rpm -e *PackageName*** to remove them. Then, try reinstalling Kylix.

If you have Kylix completely uninstalled (the **rpm -qa | grep kylix** command returns nothing), and after choosing "Begin Install" from the installation, the installer aborts with an error code -10, then the problem may be do with the your locale settings. Typing the command "locale" may show **LC\_NUMERIC="de\_DE"** (the German locale, in this case). Some versions of RPM (mostly newer ones) respect this setting and will use a comma as the decimal separator. This causes a problem for the Kylix installer. The installer needs a period as the decimal separator. The workaround is to reset the **LC\_NUMERIC** environment like this:

```
export LC_NUMERIC="en_US"
```

Once installation is finished, you can reset this value by setting it to the original locale. This problem can occur for any locale which does not use a period as the decimal separator.

### Should I install Kylix as root or a regular user?

Installing Kylix as root has the advantage of allowing all users to access the Kylix installation, provided you install it into a "shared" location. I reccomend installing Kylix as root into the directory /usr/local/kylix. However, if you are going to be the only user using Kylix, you may want to install it as that user into your home directory.

To install as root, type **su** from the command prompt and enter the super user password. Then, run the Kylix installer.



## Article #27405: Borland Delphi Presales Frequently Asked Questions

### Can Delphi output Java byte code?

No. Use [JBuilder](#) for Java development.

### Are the ADO components in Delphi compatible with Microsoft's Jet engine 4?

Yes.

### Can I use components written in an earlier version of Delphi than the one I have?

Yes, though Delphi components from previous versions must be recompiled from source code in Delphi 5. This is true for all 32bit versions of Delphi.

### What is TeamSource?

TeamSource is a workflow management tool for helping application development teams manage their daily tasks in a shared development environment. TeamSource uses a version control system, such as PVCS, for storing and retrieving shared files, but goes beyond simple version control to manage and coordinate the process of using a parallel model of source control.

### Does PVCS work with Delphi?

Yes, but PVCS does not ship with Delphi.

### What is the Integrated Translation Environment?

The Integrated Translation Environment (ITE) is a suite of tools that facilitates software localization and simultaneous development of applications for different locales. Because of its integration with the Delphi IDE, the ITE lets you manage multiple localized versions of an application as part of a single Delphi project group. The ITE ships with the Enterprise and higher editions of Delphi.

### Can I add the Integrated Translation Environment to my Professional or Standard version?

No. You will need to upgrade to the Enterprise or Architect version to get the ITE.

### What are some of the differences between Delphi Standard/Professional/Enterprise/Architect?

See <http://www.borland.com/delphi/index.html>

### Can I still take advantage of all the new language and compiler features if I only have the standard version?

Yes, the core compiler is the same throughout the different versions of Delphi being offered (e.g. Standard, Professional, Enterprise, etc.)

### Is there a Translation Suite for Delphi?

Yes, Delphi Enterprise now ships with Borland Translation Suite.

### **Can I develop Windows CE applications with Delphi?**

No.

### **Can I take advantage of the new data types MS SQL Server 7 offers?**

No, not with the SQL Links driver. Microsoft no longer supports DB-LIB with the release of MS SQL Server 7 which means that DB-LIB doesn't have support for the 4 new data types that SQL 7 offers - namely NTEXT, NCHAR, NVARCHAR and GUID. In order to successfully use new types offered by SQL Server 7 and above you'll need to use CT-LIB, or use dbGo.

### **What vendors/versions of ODBC drivers have Delphi and the BDE been tested with?**

Please see the list in the BDE readme.txt or online at <http://www.borland.com/devsupport/bde/odbccertinfo.html>

Last Modified: 16-APR-03

## Article #27406: What do I do if I am getting missing or corrupted file messages?

### Question:

What do I do if I am getting missing or corrupted file messages?

### Answer:

You will often find "default" versions of these files on the CD for your Borland product.

First, search your system for the files listed in the error message by going to Start | Find | Files or Folders...

Type the name of the file, Look in: your local hard drive and click Find now.

If the file(s) are not on your system then copy them from the Borland CD to your hard drive. If the file(s) are on your hard drive then try checking the path to make sure the files are in the correct directories. If you still get the error then try copying the "default" file(s) from the Borland CD over the file(s) on your system, replacing them. Be sure to remove the Read-Only attributes.

Last Modified: 13-JUN-01

## Article #27407: How do I determine if I have a bad CD?

### Question:

How do I determine if I have a bad CD?

### Answer:

It is extremely unusual to have received a CD that is corrupt. In many cases, the CD-ROM itself is not reading the CD properly. One of the following actions may correct the problem, or allow the installation to proceed:

- Try a hard drive installation.
- If the CD-ROM is external, try turning the CD-ROM drive on it's side. As strange as this may sound, in many cases it will change the behavior.
- If you are using Windows drivers, switch to the manufacturer's CD-ROM drivers (try the reverse if you are using the manufacturer's drivers). Also, check with your manufacturer to obtain a more recent copy of their driver.
- Compare the file size of any files that you suspect are corrupt to the file size of those on your CD. If you can copy the installation directory from the CD to the hard drive, and the file sizes match, then the CD is most likely fine -- the problem is elsewhere.
- If the CD-ROM drive simply will not read from the CD at all (and works on other CDs), then replacing the CD with an identical CD will probably not work either. This is usually due to the fact that there are different CD-ROM formats and not all drives can read all formats. Be sure to check for firmware updates for your CD drive. Alternately, you could try using another CD-ROM drive (or copy the files over the network from another machine).

## Article #27411: Borland Delphi Install Frequently Asked Questions

### My system freezes when I start Delphi - I don't even see the splash screen.

Check display colors - Delphi cannot run in Windows 98 with 16 colors. If that does not fix the issue then add the following line to the system.ini file under the [display] section: BusThrottle=1 (reboot afterwards). This tells the Video card that it cannot dominate the Bus and must give some time to the OS. It looks as if the problem is the video driver is not releasing the Bus back to the OS after it switches to main memory.

### When I start Delphi I see the splash screen, then my system freezes.

The problem is most likely a compatibility conflict with the video driver. Borland tests its products against generic S3 drivers provided by Microsoft or S3 Inc. Any drivers compliant with this standard should not cause this problem with Delphi. Video drivers that do not fully comply with this standard may or may not function correctly with Delphi.

Possible fixes:

- Turn Hardware acceleration off (Win 95) or down to none (Win 98).
- NT - turn off bitmap caching.
- Delphi 4 - Download Update Patch 3 (contains a work-around).
- Try starting in Safe Mode (Win 95) or VGA mode (NT).
- Contact your specific video card vendor about getting a compatible generic S3-compliant video driver.
- Try variations of color depth and resolution.

### There were no errors during installation, but when I run Delphi there are no components on the component palette.

#### Windows 2000:

If you are running Delphi 6 this most commonly occurs if you install 3rd party components before running Delphi at least once after installation. Registry settings are made for Delphi the first time it is run (and subsequently the first time any additional user runs it). If components are installed before those settings are in place things can get messed up [technical term]. There is a utility that comes with Delphi 6 to fix this. Located in the bin directory or on the CD, it is called **D6RegClean.exe**. Running this (ignore warning messages) and then running Delphi will restore your component palette. You will then need to re-register and reinstall any additional components.

*Note: This utility is designed specifically for issues with Windows 2000. If you are using a different OS please see other suggestions below.*

#### Windows NT:

Delphi 5 or earlier: Most likely Delphi was installed by an administrator or a user with administrator rights and the user currently logged on does not have administrator rights. The component library is failing to load because we don't have the registry keys that tell us where the packages are located.

Install first as Admin, then do a 'Registry Keys Only' install for each user to get all the necessary DLL's and registry key entries to run Delphi under a specific user profile.

Delphi 6: This shouldn't happen :) Try uninstalling and reinstalling.

### **Windows 95/98:**

It is possible for an older version of this file to be installed (overwriting the current version) by another product. If this happens then run the comctl32.dll update on the Delphi CD. It is in the Info/updates directory and is called 40comupd.exe for Delphi 4. This file is also updated with Delphi 4 Update Pack 3. For Delphi 5, the file is in the install directory on the CD. For Delphi 6 the file is called:50comupd.exe, and is located in InfoExtrasComctl on the CD.

**Delphi starts fine, but the component palette display is corrupted. Part of one icon appears on part of the next, images are drifting off the buttons, the wrong images on buttons, etc. Restarting the computer does not help.**

The problem is most likely a compatibility conflict with the video driver. Borland tests its products against generic S3 drivers provided by Microsoft or S3 Inc. Any drivers compliant with this standard should not cause this problem with Delphi 4 or 5. Video drivers that do not fully comply with this standard may or may not function correctly with Delphi.

Possible fixes:

- Try running with video hardware acceleration turned down to none. This will identify it as a video driver conflict if it eliminates the problem.
- Try running Delphi with Computer in safe mode (Win95) or VGA mode (NT).
- Try running the update 40comupd.exe found in the infoupdates folder of the Delphi 4 CD. For Delphi 5 run the Dcom95.exe in the MDAC directory or the 40comupd.exe from the install directory on the CD. This updates the file comctl32.dll. (Note: installing Internet Explorer 5.x or newer will also update this file).
- Contact your specific video card vendor about getting a compatible generic S3-compliant video driver.
- Try variations of color depth and resolution.

**I installed Update Pack 3 for Delphi 4, and now my toolbar icons are corrupted.**

This is a video driver problem. This update changes the way the Image List is handled in order to correct the problem with the component palette (see Component Palette FAQ). This is a rare problem and may require reinstalling Delphi without the service pack if the video driver issues cannot be handled another way.

**What is installation error 113?**

This error occurs when there are problems reading files from the CD. Delphi requires a 32-bit CD-ROM driver in order to read long file names. If you have a 16-bit CD-ROM driver you will need to update to 32-bit. The reason this happens with Delphi (and usually not with other programs) is because Delphi installs by copying the Runimage directory, which contains long file names, from the CD to the hard drive. Typically, other installation programs unpack a compressed file. Contact the manufacturer's web site for an update.

To verify that you have a 16-bit driver:

- Check device manager in Control Panel - if no CD-ROM then it's 16-bit.
- Go to Explorer and view the Runimage directory on the CD. If there is a tilde (~) in any of the file names then the name is long and is being truncated.

Possible fix:

- Autoexec.bat and Config.sys - remove any reference to the 16-bit CD-ROM and restart. Windows might load new drivers. If not, manually add the new drivers.

Other Issues:

- This error can happen when someone is installing off of a network and has not actually mapped a drive to the correct location (they simply share the folder). Setup needs to see the Install and Runimage folders in the ROOT directory of a drive. Right click My Computer->Map Network Drive...
- This can also happen if you are trying to do a hard drive install, but have not put the files in the ROOT of the drive.

### **What is installation error 117?**

A certain batch of CD's produced by Borland contained a glue-like substance on them that causes this error because a track on the CD cannot be read. The solution is to clean the CD using a glass cleaner like Windex. The other possible cause of this error is a damaged or scratched CD. If cleaning the CD does not make the error go away then customer should call the order desk (1-800-932-9994) to order a replacement CD.

### **I get installation Error \_ins5176 at the end of my installation.**

This is an Install Shield error. If the install continues and everything seems to work fine, then it can be ignored (the error happens during installation cleanup). Otherwise, it's much more serious and there's no easy solution. Try uninstalling the product, including registry entries, and doing a hard-drive installation in SAFE mode. This usually happens on a machine that already has another version of Delphi on it and has to do with the failure to register an OCX that is already registered.

### **What does the error Can't find PVCS when installing Delphi 4 Pro mean?**

This is an error in the install program and can be ignored. PVCS comes with Delphi 4 C/S but not Delphi 4 Pro, but can be installed with Delphi 4 Pro if customer already has PVCS.

### **What does BORDBK60.DLL or (BORDBK50.DLL for Delphi 5),(BORDBK40.DLL for Delphi 4) is missing or not registered error mean during installation or startup of Delphi?**

If the installation does not complete the cause of the problem is likely due to a software conflict. Some anti-virus software and certain NT Internet Services have been known to prevent bordbk60.dll and/or html.ocx from being registered with the operating system. In this case the best solution is to try eliminating the software in question and then try reinstalling Delphi 6. If that is not a viable solution, try step 1 below to manually register the bordbk60.dll (BORDBK50.DLL for Delphi 5),or (borbdbk40.dll for Delphi 4) or html.ocx. If the install does complete:

1. Run TREGSVR manually to register the dll. TREGSVR can be found in the Delphi 6, 5 and 4 / bin directory. To register the DLL you would type:

```
tregsvr bordbk60.dll
```

in the directory where the DLL resides (assuming you have the Delphi 6, 5 or 4 /bin directory on your path, otherwise you would include the path to bordbk60.dll).

Delphi 4: If TREGSVR says the dll has been successfully registered, but Delphi 4 doesn't work, then you have probably uninstalled Delphi 4 AFTER applying Update 2, and then reinstalled without applying Update 2. Hence, the DLL is installed correctly but Delphi 4.00 does not know how to use the Delphi 4.01 version of the DLL. To correct this problem install update pack 2 and then your version of Delphi 4 will correspond to the version of the DLL on your system.

### **How do I know which version of Delphi I have?**

#### **Delphi 6, 5, 4:**

The menu option Help/About tells you what build version you have and what service packs are installed.

#### **Delphi 3:**

Delphi 3 did not update the about box. Look at the time stamp for Delphi 32.exe if it is 3:00 am it's unpatched, 3:01 you have 3.01 and 3:02 you have 3.02. There is also a Delphi/BDE version utility available on the Borland web page under DOWNLOADS.

### **How do I completely uninstall and reinstall Delphi?**



1. Back up all project files which are located under the Delphi directory.
2. Optional: back up aliases by backing up idapi32.cfg, using it to replace the newly installed idapi32.cfg after reinstallation; It is recommended that all third party and custom components be reinstalled after the complete reinstallation of Delphi.
3. Use the Windows add/remove programs utility to remove Delphi, answering yes to remove all files
4. Manually remove the Delphi directory from your system.
5. If you are using Delphi 6 on Windows 2000, run the utility D6RegClean.exe off of the CD.
6. Install Delphi using the following steps: Insert the Delphi CD into CD-ROM Drive. If auto-play is enabled, the Delphi Setup Launcher will automatically run. If the Delphi Setup Launcher does not run automatically, run install.exe in the root directory of your Delphi CD. Follow the on-screen instructions.
7. If you backed up your aliases, copy the backed up version of idapi32.cfg over the newly installed idapi32.cfg
8. Install all third party and custom components.

### **How do I install Delphi from a local hard drive?**

You will need approximately 184 MB of hard drive space to temporarily store the installation files, 160 MB for the complete installation and free space to hold temporary files, on the drive that has windows while the installation program is running.

1. Put the Delphi CD in the drive and choose Browse the CD.
2. Copy the Install folder to the root directory (i.e. not inside any subfolders) of the local drive you will be installing from.
3. Copy the Runimage folder to the root directory of the same drive. This may take a very long time. Verify that the entire folder was copied correctly. Right-click on the folder and choose properties. This folder should contain the same number of files that the one on the CD-ROM contained.
4. Copy all of the remaining files (autorun.inf, delphi.ico, deploy.txt, etc.) to the root directory of the same drive.
5. Reboot into Safe Mode or VGA mode and double click on the install.exe that you now have on your hard drive and install Delphi as you normally would.
6. Reboot in normal mode and verify that Delphi runs correctly. Once you have done this you can delete all the files that you copied over from the CD-ROM.

### **What is Safe Mode or VGA Mode?**

Safe mode is the method for booting into a simple system configuration under Windows 95\*. VGA mode is the diagnostic mode under Windows NT. When Windows boots into Safe mode or VGA mode, it uses basic default settings that will run the operating system with minimal functionality, meaning it loads a bare bones operating system. Hardware, like your CD-ROM drive, may not function in Safe mode or VGA mode because the drivers are not loaded. Because of this it may be necessary to copy the Delphi installation files to the hard drive before restarting in safe mode (See Hard Drive Install FAQ).

\*Windows 98 has a safe mode but it has a different functionality and may not be helpful in resolving install issues. Specifically, it loads the current video driver instead of the default VGA driver provided by Microsoft, so starting in safe mode in Windows 98 will not resolve issues related to the video driver. You will need to manually change the driver.

## **How do I start my computer in Safe mode (Windows 95) or VGA mode (Windows NT)?**

Safe Mode: There are two ways:

1. Hold down the SHIFT key as you reboot until the Windows screen comes up.
2. When you restart, watch the screen for the text Starting Windows 95. When you see this hit F8 and choose Safe Mode from the option menu which comes up.

VGA Mode: Windows NT should offer a menu at start up where you can select VGA mode.

## **Why does the splash screen keep popping up?**

Either you have a bad CD-ROM driver, or a slightly damaged CD that one computer may read fine but this particular computer is not as forgiving. Try a hard drive installation or update the CD-ROM driver. You will need to contact customer service in the case of a damaged CD.

## **Why is Delphi 4 running slower than Delphi 3?**

Make sure Service Pack 2 and 3 have been installed. The debugger hooks were slow starting and finishing which caused slowness.

## **What does the ICON IMAGE IS NOT VALID error mean?**

Chances are good that you are using a Micron (or other factory assembled system) computer which has a proprietary version of the OS (95, NT, 98). You may be able to successfully install Delphi by uninstalling and re-installing the OS, or in the worst case scenario you might need to install a non-Micron-proprietary OS. Also, check to see if there is more than one copy of bordbk60.dll for Delphi 6, bordbk50.dll for Delphi 5 or bordbk40.dll for Delphi 4 on the machine.

## **Delphi used to work fine but now I get random access violations in the IDE.**

You most likely installed a bad component or package that was not written or compiled for Delphi 6, 5 or 4. Try uninstalling custom components one by one (or package by package) until the problem goes away, then get in touch with the component vendor regarding this issue.

**Whenever I save, even a blank project, I get '<garbage>' is not a valid identifier. If I try and save again I get '<garbage>' is not a valid identifier.**

Uninstall Delphi. Make sure no other software is running, especially antivirus software, and reinstall Delphi. Alternately, a customer reported that ImageFox from ACDSYSTEMS was the source of his problems, but this has not been confirmed or disproved.

**I just installed Delphi and when I access the Environment Options or open up one of my projects, I get an error message: 'Delphi32 caused a GPF in module MACXW4.DRV at...' and then Delphi shuts down.**

More than likely this is a video driver related problem. Downloading the latest video drivers for your video card should solve the problem. You can also try turning hardware acceleration off.

**What does BDE Error 8712 - Object not found mean?**

Be sure there is only one version of BDE installed on machine. To verify this, search for idapi\*.dll. Also, try renaming idapi.cfg or turning off virtual alias to see if its a ODBC problem.

**Why am I having preview and paper size problems in QuickReports? (Delphi 4 only)**

The latest version of QuickReports, version 3.0.5, has several bug fixes including some related to custom paper size and print preview. This version comes with Delphi 5. An upgrade is available for Delphi 4 users from the QuSoft web page at:

<http://www.qusoft.com/scripts/download.dll>

The file is called QR305SD4.EXE

**I am getting runtime error 255 when using Delphi 3.**

This error is reported if there is a hardware or other exception that isn't caught (this error is caught in Delphi 4). Error 255 in Delphi 3 appears to be caused by certain video drivers, or by workstations using the AMD NexGen x586 CPU. If your PC is not using the AMD NexGen x586, then the problem is probably the video driver. Try changing driver configuration, getting a newer video driver, or using the generic S3 drivers provided by Microsoft or S3 Inc.

Last Modified: 19-APR-02

**Article #27428: Error "database unavailable" when connecting to Interbase via IIS.**

**Question:**

Why am I getting the error "database unavailable" when connecting to Interbase via IIS?

**Answer:**

Go into the services window on the machine your apps are running on and make sure that all services such as Interbase server, IIS Admin and web Publishing and Interbase Server have the "Allow service to interact with desktop" option checked. This is accessed by right clicking the service and going to properties | Log On.

Note: This may not be the same as "Unavailable Database" which is generally caused by not having your Interbase server running.

Last Modified: 15-JUN-01

**Article #27442: SIGSEGV error when running MySQL on Kylix.**

**Question:**

Why am I getting the error SIGSEGV when attempting to run a dbExpress demo in Kylix?

**Answer:**

DBExpress may not currently work with MySQL v3.32. It should work fine with v3.22.32.

Last Modified: 09-NOV-01

## Article #27459: What is the message "Generating font matrix" when starting Kylix?

### Question:

Why am I getting a message that says "Generating font matrix. Please wait" when I start Kylix?

### Answer:

This should only happen the first time you start Kylix using the startkyix script. The font matrix is used by the Wine implementation of the Win32 API and you should wait until it finishes. There have been reports of this appearing to take forever in some Linux versions. If it seems to take too long you can try canceling it and Kylix should still run okay.

Last Modified: 09-NOV-01

## Article #27463: My console application generates a Runtime Error when it starts for no apparent reason.

### QUESTION:

My Console Application compiles fine, but when I run it I get a Runtime Error 6. What is this error and how do I fix it?

### ANSWER:

Runtime error 6 is defined in the online help as:

```
106      Invalid numeric format  Reported by Read or Readln if a numeric value
read from a text file does not conform to the proper numeric format.
```

The most probable cause of this error is that the compiler directive for APPTYPE has not been set to console. Your console application you should set the APPTYPE to CONSOLE at the beginning of you application. If you are using dcc32 to compile you program you could also pass the /CC command line option. Your console application should look something like this:

```
program OneCoolDelphiProgram1;

{$APPTYPE CONSOLE} //This tells the compiler that the program is a console type

uses
  SysUtils;

begin
  //Insert super cool Delphi Code here
end.
```

Last Modified: 03-JUL-01

**Article #27478: What version of TeeChart is supplied with Delphi 6?**

**QUESTION:**

**What version of TeeChart is supplied with Delphi 6?**

**ANSWER:**

The version supplied with Delphi 6 is TeeChart Standard 4.04.

Last Modified: 09-JUL-01



## Article #27486: Where can I obtain the help file for the VtChart component?

### QUESTION:

I can't bring up help for the VtChart component found under the ActiveX tab. It says contact your application vendor for the updated help file. Where can I obtain the help file for the VtChart component?

### ANSWER:

The Windows Help engine on Windows 95, 98, and Millennium uses a 16-bit list box in the Help Topics dialog's Index tab, and can display no more than 32,767 keywords. If this limitation is exceeded, the Index is blank. Rather than adding this help file to the main Delphi help system, simply open up the help file when you need to use it. The help file can be found in the following directory (for default installs):

C:\Program Files\Borland\Delphi5\OcxVciImpressVcfi.hlp

Simply navigate out to the directory where this file is located and double click on it (using Windows Explorer) and it should open right up.

Last Modified: 10-JUL-01

**Article #27506: How to change file attributes programmatically.**

**ANSWER:**

It might be convenient to change file attributes during the run time of your application. For example you might want to create a temporary file to take advantage of Windows' ability to manage the file in memory for quicker access. To change a file's attributes at runtime you can use the function **SetFileAttributes()**. To see exactly what attributes you can set see the help for the Windows SDK ( From the Start menu go to the Delphi menu and choose Help | Windows SDK ).

Last Modified: 10-JUL-01

**Article #27511: Insufficient disk space on C: drive while installing Delphi to D: drive.**

**Question:**

Why am I getting an error that there is not enough space on C: when I'm installing to D:?

**Answer:**

Delphi needs to copy many files to the Windows System32 directory and requires around 28 MB of space on the drive where this directory is located - even if it is not the directory where you are installing Delphi.

Last Modified: 13-JUL-01

## Article #27522: Why is there no "Save" button in the save dialog box?

### Question:

Why is there no 'Save' button in the save dialog box?

### Answer:

In the Save dialog box if you click on one of the directories listed then the Save button caption will change to 'Open'. If you then click the open button it will open the directory you have selected. If you have selected the '.' (Linux only), clicking on open will bring you up to the parent directory of the directory you are currently in. In order to get the button to say 'Save' again just click somewhere else, such as in the File name edit box.

Last Modified: 17-JUL-01

## **Article #27541: Problem generating font matrix in Kylix.**

### **Question:**

My install of Kylix was successful but after startup with "startkylix" I see the message "generating font matrix" forever. What should I do about this?

### **Answer:**

If you installed as root the font matrix will be created for each user the first time they start Kylix. How long it takes depends on the number of fonts that you have installed and the speed of your machine. If it seems to freeze or last a long time you should still be able to run Kylix even if you terminate the font matrix generation.

### **Possible workarounds**

This problem could be caused by a bad truetype font. If you have any Windows tt-fonts installed try removing them. Another thing to try is to change the order of the font search path in XF86Config or in your tt-fontserver. Try to put the tt-fonts at the end of the list.

Here is another suggestion from our newsgroups:

Rename the file "transdlg" located in your Kylix bin directory to "transdlg.renamed" and then run Kylix.

This is an issue we are currently looking into. This document will be updated when more information is available.

Last Modified: 01-NOV-01

## Article #27543: How do I set a custom CVS PServer port?

### Question:

How do I configure the CVS PServer to use a different port value than the default?

### Answer:

You may do this from the commandline in the following way:

Windows:

```
set CVS_PSERVER_PORT=<custom port value>
```

Linux:

```
export CVS_PSERVER_PORT=<custom port value>
```

Last Modified: 30-JUL-01

**Article #27544: Some links and/or executables are missing after running Kylix install.**

**Question:**

I installed Kylix but now I can't find certain files or links on my machine. Why were they not installed?

**Answer:**

When you install Kylix you are prompted for two destination directories - one for files and one for links. If you installed as a user but specified a directory for your links that the user does not have write access to, such as `/usr/local/bin`, then the file will not be written because only root can write to this directory. The solution is to install everything in directories that the installing user has full access to such as `/home/username/kylix` and `/home/username/bin`.

Last Modified: 01-NOV-01

## **Article #27545: Error "permission denied" when trying to run Kylix**

### **Question:**

Why am I getting the error "permission denied" when I try to run Kylix?

### **Answer:**

If you installed as root and accepted the defaults then Kylix was installed under the /root directory. By default the permissions for this directory are 700 so only the root user can access this directory tree for read, write or execute. If users besides root are to use Kylix it must be installed in a directory all will have access to. See the file INSTALL that comes with Kylix for more information.

Last Modified: 30-JUL-01



**Article #27547: The Buttons are missing or grayed out in Database Desktop.**

**PROBLEM:**

**Database desktop is not loading correctly, all the buttons are missing.**

**SOLUTION:**

This problem is due to a compatibility issue with Windows 2000 and Paradox.

In Order to resolve this issue you must use the "Application Compatibility Tool". It is located in the resource kit on the Windows 2000 CD. You can also download the Application Compatibility Tool directly from Microsoft's website:

<http://msdn.microsoft.com>

Once the resource kit is installed then select

Start

Programs

Windows 2000 Support Tools

Tools

Application Compatibility Tool

Where it says "Start the following program" browse to the location of the executable for DataBase Desktop and select it (dbd32.exe) or you can simply type in the full path and file name.

For Operating System select "Windows NT4 SP5".

Select "Make the above check box settings permanent".

Last Modified: 14-AUG-01

## Article #27549: How do I create an executable file using the command line directive in Linux with Kylix?

### QUESTION:

How do I create an executable file, i.e. foo.exe, using dcc in Kylix?

### ANSWER:

Create the project file, i.e. foo.dpr, using VI, Pico or some other text writing tool. Next, at the command line type: dcc foo. You can pass in flags like -BE: dcc -BE foo, the will build and execute the file. There are a number of different flags you can pass in, look at the help for dcc for a full listing and description.

```
program foo
```

```
uses
```

```
    SysUtils;
```

```
begin
```

```
    writeln('Hello World');
```

```
    readln;
```

```
end.
```

Last Modified: 03-AUG-01

**Article #27550: Error message "Failed to create process" when trying to use the Corba Server and Client Application wizards.**

**QUESTION:**

**I am getting an error message "Failed to create process" when trying to use the Corba Server and Client Application wizards in Delphi 6. What is causing this?**

**ANSWER:**

This problem derives from a user not having enough rights. In order to use the Corba Client and Server Wizards, you must have full administrative rights. Without administrative privileges, you will get the "Failed to create process error" or some error relating to this. Once the user is given administrative rights, the wizards will work correctly.

Last Modified: 03-AUG-01

**Article #27551: Getting Fatal Error "File not found: 'OrbPas.dcu'" when using a Corba Object or Corba Data Module in Delphi 6.**

**QUESTION:**

**Getting Fatal Error "File not found: 'OrbPas.dcu'" when using a Corba Object or Corba Data Module in Delphi 6. Why won't this work?**

**ANSWER:**

If you look at the Corba Compatibility notes in the Delphi 6 readme, you will see the following:  
The TypeLib interface and CorbaConnection component are only compatible with the VisiBroker 3.3 ORB.  
If you installed Visibroker 4.1 when you installed Delphi 6, this means you cannot use the Corba wizards found under the File | New | Other | Multitier tab. If you need these components, you will have to uninstall Visibroker 4.1 and install Visibroker 3.3.

Last Modified: 03-AUG-01

## Article #27553: What can I use in Kylix, as a possible replacement for ShellExecute in Windows?

### QUESTION:

How do I launch a Linux application from Kylix?

### ANSWER:

There is a method you can use to execute a shell command that will launch the application: `system()`. This executes the command you specify in the string parameter, you will need to include **libc** in your uses clause. There is a return code of -1 if the call was not successful. You could do something like this:

```
TForm.ButtonClick(Sender: TObject);
var
    idError: integer;

begin
    idError:= libc.system('emacs');
    if idError = -1 then
        ShowMessage('Error: Program did not start');
end;
```

Last Modified: 05-FEB-02

## **Article #27554: How to create a console application in Kylix.**

### **QUESTION:**

How do I create a console application in Kylix?

### **ANSWER:**

To create a console application in Kylix, you need to make sure you check the "Use Launcher Application" check box. The check box is located on the Run Parameters dialog page, in the Local tab. To get to that dialog go to: Run | Parameter. Now, if you run your application, you will notice that a console window appears, if you are in Xwindow.

Last Modified: 05-FEB-02

**Article #27555: How to change the shape of your forms.**

**QUESTION:**

How can I change the shape of my form?

**ANSWER:**

There are a number of Windows API calls you can make to do this. You can use these calls on any component that is descendant of TWinControl, like TForm. Here is a list of the different API calls:

```
CreateEllipticRgn  
CreateEllipticRgnIndirect  
CreatePolygonRgn  
CreatePolyPolygonRgn  
CreateRectRgn  
CreateRectRgnIndirect  
CreateRoundRectRgn
```

The example I use is CreateEllipticRgn, here is one way of using it. You could put this call with in a button click.

```
SetWindowRgn(Form1.handle,CreateEllipticRGN(0,0,Width,Height),True);
```

Last Modified: 03-AUG-01

## Article #27561: How to capture the current cursor icon and draw it.

### QUESTION:

How do I get the handle of the current cursor and draw on my form?

### ANSWER:

There is a Windows API call to do this: DrawIconEx(). This function draws an icon or cursor in the area specified by the X and Y coordinates, that are parameters to this function. You can pass it a given handle and it will draw that icon. Here is an example that sets the cursor handle and draws the icon.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    HCursor : THandle;
begin
    HCursor:= Screen.Cursors[Ord(Screen.Cursor)];//Gets the cursor handle.
    //
    //          X    Y          Size
    DrawIconEx(Form1.Canvas.Handle, 150, 150, HCursor, 32, 32,
                0, 0, DI_NORMAL);//Draws to canvas
end;
```

Last Modified: 03-AUG-01



## Article #27572: How to stop IIS when you are writing web applications as ISAPI dll's

**Question:** How to I completely restart Internet Information Services (IIS) in order to unload my ISAPI DLL so I can rebuild it?

**Answer:** On Windows 2000 with IIS5, use the command iisreset from a command prompt. On Windows NT4 with IIS4, use the commands:

```
mtxstop  
net stop iisadmin /y  
net start w3svc
```

You can place these commands in a .bat file to easily restart your server at will.

Last Modified: 04-AUG-01

## Article #27578: Oracle Stored Procedures Generate Access Violations in ORA805.dll

### QUESTION:

Oracle stored procedure worked fine with BDE 5.0.1 but later versions generate Access Violations. Is this an incompatibility with older versions of Delphi?

### ANSWER:

The problem isn't a bug in SQL links, nor is it a backwards compatibility problem with the BDE. The D4 shipping of SQLORA8.DLL didn't use the OCI 8 calls to get the stored procedure schema info, instead we were using OCI 7 calls. It is the attempt to get the stored procedure schema with OCI 8 calls that is causing the access violation.

The current work around is to use the BDE 5.0.1.

Last Modified: 04-AUG-01

**Article #27579: How to use TADOTable Seek procedure.**

**QUESTION:**

When I use TADOTable.Seek I get an access violation, "Object or provider is not capable of performing requested operation." In the help it is documented that all I need is Access 2000 and the Jet 4 Provider. How do I get Seek to work?

**ANSWER:**

The online help is slightly miss documented for this procedure. There are two other properties that need to be set in order to successfully execute the seek procedure. You can download a demo project from Code Central [here](#).

To avoid the access violation you have to set two properties for TADOTable. The first is to set CursorLocation := clUseServer. The second step is to set the TableDirect property to True. You can also specify the IndexName property to be the primary key of the table, but this seems to be optional.

Last Modified: 04-AUG-01

**Article #27588: Lost trial key.**

**Question:**

How can I get a key for the trial version sent to me if I lost or never received one?

**Answer:**

Go back to the community page where you completed the survey. Login and click the link to complete the survey and it will bring you to the screen where you can order the serial number and have it sent again. Be sure to verify that your email address is correct.

Last Modified: 06-AUG-01

**Article #27593: Error "Unsupported calling convention: REGISTER" in SOAP Application.**

**QUESTION:**

**Why am I getting the following Debugger Exception Notification: "Project Client.exe raised exception class Exception with message 'Unsupported calling convention: REGISTER'. Process stopped. Use step or run to continue." after creating a SOAP application and attempting to run the client?**

**ANSWER:**

This problem occurs because the stdcall calling convention was not used in the function declaration in the server. Include stdcall at the end of the function definition and the problem will be resolved. In the Server Interface file and in the Server Implementation file the function definition should look similar to this:

```
function Foo: String; stdcall;
```

The reason stdcall must be used is that the default calling convention is Register which passes parameters from left to right; that is, the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. In Windows, the operating system API's use stdcall which pass parameters from right to left.

Last Modified: 14-AUG-01

## Article #27594: Does Kylix support ODBC?

### Question:

Does kylix support ODBC?

### Answer:

You can connect using an ODBC driver via dbExpress in Kylix. There is an ODBC dbExpress gateway driver on the Kylix companion CD. It is also available on [Code Central](#) or the [Easy Soft](#) web site.

Last Modified: 01-NOV-01

## Article #27595: Does Kylix support Postgres?

### Question:

Can kylix connect to PostgreSQL Server?

### Answer:

The Kylix 2 dbExpress PostgreSQL/RedHat database driver is available for download from the [Kylix 2 registered users website](#).

You can also use [ODBC](#).

Last Modified: 18-DEC-01

## Article #27600: What is DBGo?

### Question:

What is DBGo?

### Answer:

DBGo is another name used for the ADO add on components for the Professional version of Delphi - also known as ADOExpress.

Last Modified: 06-AUG-01



**Article #27602: Do the ADO components come with the professional version of Delphi 6?**

**Question:**

Do the ADO components (DBGo) come with the Professional version of Delphi 6? It states in the documentation that they do not.

**Answer:**

The documentation is wrong. These components did not come with the Professional version of Delphi 5, they were an optional add-on, but they do come with the Professional version of Delphi 6.

Last Modified: 06-AUG-01

**Article #27613: I can't find ADO Express for Delphi on shop.borland.com. Where can I buy ADO Express for Delphi 5?**

**QUESTION:**

**Where can I obtain ADO Express for Delphi 5?**

**ANSWER:**

ADO Express is now included in Delphi 6 Professional and Enterprise Editions. It is no longer available for Delphi 5.

Last Modified: 16-APR-02

**Article #27647: What features come with the Delphi 6 free trial?**

**Question:**

What features and extras come with the Delphi 6 free trial?

**Answer:**

The trial version includes Delphi 6 Enterprise, Visibroker, and Interbase. InstallShield is not included.

Last Modified: 13-AUG-01

**Article #27656: Not enough disk space to install Delphi 6 on Windows 98.**

**Question:**

On Windows 98 I am told that I don't have enough space to install the product, even though I have plenty of Disk space?

**Answer:**

This is a problem with the MSI. Usually, you can proceed to exit the installer and then try install over again. The second time you try to install there shouldn't be any problems.

Last Modified: 14-AUG-01

## Article #27699: Heterogeneous queries with MSSQL 7

### Question:

Why am I getting blank data or the error "Could not find object" when trying to do a heterogeneous query in Delphi?

### Answer:

The BDE does not support the new unicode data types introduced in MSSQL7 ([more info](#)). This will prevent you from doing the heterogeneous query if the columns you are linking are one of these data types. One workaround is to do the heterogeneous join using MS SQL 7 server and then access the data from Delphi using ADO. See the following for more: <http://msdn.microsoft.com/msdnmag/issues/0800/cutting/cutting0800.asp>

Last Modified: 28-AUG-01

**Article #27702: Why reusing or inheriting from a Remote Data Module is not a good idea.**

**QUESTION:**

Why can't I reuse or Inherit from my newly created Remote Data Module(RDM), that I just placed into the Object Repository?

**ANSWER:**

The reason is, each Remote Data Module(RDM) needs to have a unique GUID and class ID associated with it. The GUIDs and class IDs are added to the registry, where the system references them to see which object they represent. You can copy your RDM from the Repository, however you will need to change the class ID and the GUID to a unique number. Though, I suggest leaving this chore to the system.

Last Modified: 28-AUG-01

**Article #27703: How to find what the user path is for their Environment Variables.**

**QUESTION:**

How can I get the path of the different Environment Variables in Windows2000?

**ANSWER:**

There is a function call that returns a string. You can use it in a button click and see that it returns the path of the variable you choose.

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
    ShowMessage(GetEnvironmentVariable('Temp'));  
end;
```

Last Modified: 31-AUG-01

## Article #27704: How to download a file from the Internet.

### QUESTION:

How do I download a file from a URL location?

### ANSWER:

There are a number of ways to do this, below is a simple code snippet that will give you an idea of how to do this. Make sure to add ShellApi, and UrlMon to your uses clause.

```
function DownloadInternetFile(Source, Dest : String): Boolean;
begin
  try
    Result := URLDownloadToFile(nil, PChar(Source), PChar(Dest), 0, nil) = 0
  except
    Result := False;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  SourceString, DestinationString: string;
begin
  //File location on the Internet.
  SourceString := 'http://www.borland.com/images/homepage/del6homemural.gif';
  //File destination.
  DestinationString := 'C:Tempdel6homemural.gif';

  if DownloadInternetFile(SourceString, DestinationString) then
  //This will display the file from your browser.
    ShellExecute(Application.Handle, PChar('Open'), PChar(DestinationString), PChar
(''), nil, SW_NORMAL)
  else
    ShowMessage('Error during Download ' + SourceString);
end;
```

Last Modified: 31-AUG-01



## Article #27706: How to invoke the Windows ShortCut Dialog wizard.

### QUESTION:

How do I invoke the Create Shortcut dialog wizard for my application?

### ANSWER:

You will need to add ShellApi and Registry to your uses clause. Next, pass in the directory were you want your shortcut, you may want a variable here to get input from the user. For this example I use Temp:InvokeShortcutDialog('c:Temp');. However, to save it to the DeskTop you would instead do this: LaunchShortcutDialog('C:Documents and Settings"User Name"Desktop'); or use a string variable: PathShortcut : string, so the function call looks like this: LaunchShortcutDialog(PathShortcut);.

```
function InvokeShortcutDialog(Directory : string) : boolean;
var
  Reg : TRegistry;
  CmdLine : string;
begin
  Result := false;
  Reg := TRegistry.Create;
  try
    Reg.Rootkey := HKEY_CLASSES_ROOT;
    if Reg.OpenKeyReadOnly('.LNKShellNew') then
      begin
        CmdLine := Reg.ReadString('Command');
        CmdLine := StringReplace(CmdLine, '%1', Directory, []);
        Result := True;
        WinExec(PChar(CmdLine), SW_SHOWNORMAL );
      end
    finally
      Reg.free;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  InvokeShortcutDialog('c:Temp');
end;
```

Last Modified: 31-AUG-01

## Article #27707: How to close your application using the Escape key.

### QUESTION:

How can I make my application close when the "Escape" key is pressed?

### ANSWER:

Make sure that you set the KeyView property of the Form to True. This will make keyboard events occur on the form, before the active control. You should set this in the create event of the form. Below, you will find a code sample that demonstrates how to do this.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.KeyPreview := True;
end;
```

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  //The Escape key is #27.
  if key = #27 then
    Close;
end;
```

Last Modified: 16-SEP-01

## Article #27708: Error 10 while installing Kylix

### Question:

What is causing Error -10 while installing Kylix?

### Answer:

It's possible that a previous version of Kylix was not uninstalled completely or that there is a problem with the RPM. [[more info](#)]. There are a couple ways to work around this. The first is to install as a user other than root. If you would rather not do this you can use the "-m" option on the setup script. So you would enter: `./setup.sh -m` from a vterm. This tells the script to not use the RPM.

Last Modified: 01-NOV-01

## Article #27718: How can I make the background color different for alternating lines of text in a TListBox in Delphi?

### QUESTION:

How can I make the background color different for alternating lines of text in a TListBox in Delphi?

### ANSWER:

Here is one way of accomplishing this. After dropping a TListBox on your form, you must change the Style property of the TListBox to lbOwnerDrawFixed. If you fail to change the Style property, the OnDrawItem event will never be called. Put the following code in the OnDrawItem event of your TListBox:

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index:
Integer;
  Rect: TRect; State: TOwnerDrawState);
var
  myColor: TColor;
  myBrush: TBrush;
begin
  myBrush := TBrush.Create;
  with (Control as TListBox).Canvas do // draw on control canvas, not
on the form
  begin
    if not Odd(Index) then
      myColor := clSilver
    else
      myColor := clYellow;

    myBrush.Style := bsSolid;
    myBrush.Color := myColor;

    Windows.FillRect(handle, Rect, myBrush.Handle);

    Brush.Style := bsClear;
    // display the text
    TextOut(Rect.Left, Rect.Top, (Control as TListBox).Items
```

```
[ Index ] ) ;  
    MyBrush.Free ;  
end ;  
end ;
```

Last Modified: 19-SEP-01

## **Article #27719: Action Manager Component referenced in Text Editor Tutorial is Missing**

### **Problem:**

**When attempting to do the Text Editor Tutorial for Delphi 6 Personal, I can't find the Action Manager Component.**

### **Solution:**

The Text Editor Tutorial found in the Quick Start manual incorrectly states that it is for all editions of Delphi. The Action Manager Component is not included in Delphi 6 personal. This component is only included in the Professional and Enterprise version. This is a bug in our documentation and has already been addressed in the latest Help File updates. To obtain the latest help files, go to the following site <http://www.borland.com/techpubs/delphi/v6/updates/per.html>

After installing the new help files, in the IDE select Help | Delphi Help and under the contents tab you will see Tutorials. The updated Tutorial can be found there.

Last Modified: 16-JAN-03

Article #27726: WebBrowser1.GoBack and WebBrowser1.GoForward creating 'Unspecified error'.

## Problem:

Using TWebBrowser GoBack method at the beginning of the history list or using the GoForward method at the end of the history list causes 'Unspecified Error'. According to the Delphi Help *"GoBack has no effect unless the history list contains additional documents."*

## Solution:

This is a documentation error in the Delphi Help. If you look in Microsoft's Platform SDK documentation you will find the following: *"if the beginning of the history list has been reached the IWebBrowser2::GoBack method should not be used."*

If you want to have a button or similar component that will allow you to GoBack and another button that will let you GoForward using the TWebBrowser component, here is an example of the best way to proceed that will avoid the unspecified error.

In your application drop a TWebBrowser component on the form and two button components. Name the first Button btnBack and the next button btnForward. For both of these buttons, **change the enabled property to False**. In the OnClick event of btnBack the code should look like this:

```
procedure TForm1.btnBackClick(Sender: TObject);
begin
    WebBrowser1.GoBack;
end;
```

In the onClick event of btnForward the code will look like this:

```
procedure TForm1.btnForwardClick(Sender: TObject);
begin
    WebBrowser1.GoForward;
end;
```

These buttons are currently disabled and we only want them enabled when appropriate. To do this we will use the TWebBrowser's OnCommandStateChange event. The code should look like this:



```
procedure TForm1.WebBrowser1CommandStateChange(Sender: TObject;  
  Command: Integer; Enable: WordBool);  
begin  
  case Command of  
    CSC_NAVIGATEBACK: btnBack.Enabled := Enable;  
    CSC_NAVIGATEFORWARD: btnForward.Enabled := Enable;  
  end;  
end;
```

This will avoid throwing the "unspecified error".

Last Modified: 16-SEP-01

## Article #27753: How to make sure the function DLLPROC is called when your DLL is unloaded.

### QUESTION:

Why isn't DLLPROC being called when my DLL is unloaded?

### ANSWER:

Due to RTL changes, the DLLProc was not being called appropriately. This change will fix the change that you may have noticed. Once you have completed the change, you will need to recompile the RTL and copy the new System.dcu into your project directory. Your project will use the newly created System.dcu with the fix included. In System.pas, navigate down to this method @@skipTLSproc.

```
@@skipTLSproc:
```

```
    { Call any DllProc }

    PUSH     ECX
    MOV      ECX,[ESP+8]//Changed from 4 to 8.
    TEST     ECX,ECX
    JE       @@noDllProc
    MOV      EAX,[EBP+12]
    MOV      EDX,[EBP+16]
    CALL     ECX
```

Last Modified: 16-SEP-01

## Article #27757: How to create Interbase stored procedures that return parameters with TIBQuery.

### QUESTION:

When I try to create my stored procedure using TIBQuery I get either EIBInterbaseError 'SQL error code = -104 Token unknown ...' or EIBClientError 'XSQLDA index out of range'. How do I create a stored procedure in Delphi using IBX?

### ANSWER:

Most likely your stored procedure is returning parameters. If this is the case then you will need to do two things to your IBQuery. The first is to set ParamCheck to false. The second is to clear out the Params property. Assuming that your stored procedure is legal the code to create the stored procedure might look like this in Delphi:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    IBQuery.Params.Clear;  
    IBQuery.ParamCheck := False;  
    IBQuery.ExecSQL;  
end;
```

Last Modified: 16-SEP-01

## Article #27779: Where is proxies.dcu?

### Question:

Why am I getting the error "File not found 'Proxies.dcu'"

### Answer:

This error can occur when DesignEditors is added to the Uses clause. Do not statically link in the code in the ToolsAPI folder, it is there for reference purposes only. If this is a run time package then you should separate out the design time items and put them into a design time package. You should then add DesignIde.dcp to the requires section of your design time package.

Please see the following article for more information: <http://community.borland.com/article/0,1410,27717,00.html>

Last Modified: 05-FEB-02

## Article #27780: Missing components in Delphi 6.

### Question:

Why am I missing components after installing Delphi 6?

### Answer:

This most commonly occurs if you had a previous edition of Delphi 6 that you uninstalled. It is possible that there are registry entries left over from that installation because of the new way Delphi is creating the startup registry configuration. It can also occur if you run the stand alone register program or install third party components, which will create the D6 registry key, before actually running Delphi 6 for the first time.

The solution for any of these is to run the utility **d6RegClean.exe** that comes with Delphi 6, and then restart Delphi before installing anything else. d6RegClean.exe should be located in the bin directory or on the CD.

Also be sure you are not installing two different editions of Delphi 6 on the same machine (such as Personal and Enterprise).

Last Modified: 06-JUN-02

## Article #27784: How can I create an identical binary executable in Delphi?

### QUESTION:

Does Delphi create binary identical executable files?

### ANSWER:

No, Delphi adds Timestamps in the executable. At this point there are not any flags you can set to get an identical binary executable. The Timestamps coincide with the file datestamp, so you could use this aspect to see if anyone has changed the application in anyway. Timestamps can also be used as a simple version control system, by comparing the executables. The compiler uses the system clock for the stamp.

Last Modified: 25-SEP-01

## Article #27799: How to convert numbers into roman numerals.

### QUESTION:

How can I convert decimals into Roman numerals?

### ANSWER:

Drop a button, two edit boxes and two labels on the form. Add the code below. Enter numbers into the edit boxes, then press the button, to see the conversion in the labels' captions.

```
function DecToRom(Dec: LongInt): String;
const
  Nums : Array[1..13] of Integer =
    (1, 4, 5, 9, 10, 40, 50, 90, 100,
     400, 500, 900, 1000);
  RomanNums: Array[1..13] of string =
    ('I', 'IV', 'V', 'IX', 'X', 'XL',
     'L', 'XC', 'C', 'CD', 'D', 'CM', 'M');
var
  i: Integer;
begin
  Result := '';
  for i := 13 downto 1 do
    while (Dec >= Nums[i]) do
      begin
        Dec := Dec - Nums[i];
        Result := Result + RomanNums[i];
      end;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := DecToRom(StrToInt(Edit1.Text));
  Label2.Caption := DecToRom(StrToInt(Edit2.Text));
end;
```

## Article #27805: How to get the date of the last day in the current month.

### QUESTION:

How do can I find out the date, of the last day in the current month?

### ANSWER:

Drop a button on a form and add the code snippit below. The show message dialog will reveal the date.

```
function LastDayCurrMon: TDate;
var
  y, m, d: word;
begin
  decodedate(now, y, m, d);
  m := m + 1;
  if m > 12 then
  begin
    y := y + 1;
    m := 1;
  end;
  result := encodedate(y, m, 1) - 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(DateToStr(LastDayCurrMon));
end;
```

Last Modified: 09-OCT-01



**Article #27827: Making changes to my dBase tables causes McAfee to scan tables.**

**Question:**

How can I prevent McAfee from scanning my tables when I insert/update my table?

**Answer:**

There are no programmatic work arounds, nor are there any settings that can be changed in the BDE to prevent McAfee from scanning your dbf files. Any files that are modified by the BDE can cause McAfee to scan these files. This is due to the configuration of McAfee, which can be changed if so desired. Check your McAfee user manuals for further instructions on how to do this.

Last Modified: 09-OCT-01

## Article #27845: How to debug an Apache Shared Module

### Question:

I am running Apache on Windows and want to know how to debug Apache Shared Modules?

### Answer:

It is a straight forward task to debug Shared Modules in Delphi. The only thing that needs to be done is to set the Host Application and Parameters for the Shared Module's Project. From the Delphi menu bar go to Run | Parameters. Set the Host Application to point to Apache.exe, and specify the following parameters: -X -w -f "c:\path to\httpd.conf".



When you run the project be sure that IIS is not running. If you need IIS to run while Apache is running then change the Port value stored in httpd.conf.

Last Modified: 09-MAY-02

## Article #27846: Problems installing Delphi 6 Professional or Enterprise editions after uninstalling the free personal edition.

### Question:

Why am I getting the error "Can't load package c:borlanddelphi6Binproide60.bpl" when I run Delphi 6?

### Answer:

If you had the free personal edition installed previously on the same machine, and you registered the Professional or Enterprise edition with the same ID numbers, it is possible that the registry settings from the Personal edition are not being overwritten. This may be a problem with the registration system. The easiest way to resolve it is to register with a different name/email address than the one you used for the downloaded personal edition. This may require uninstalling everything and starting over.

When removing Delphi from your machine be sure to use Add/Remove programs from the Control Panel.

If you are using Windows 2000 you can try running the utility D6RegClean.exe located in the Delphi bin directory (or on the CD in the InfoExtras folder) after you unistall the Personal edition and before installing the Professional edition.

Note: You can not have two different *editions*, such as Delphi 6 Personal and Delphi 6 Professional, installed simultaneously. You can have different *versions* though, such as Delphi 5 and Delphi 6.

Last Modified: 08-MAR-02

## Article #27890: Working around time zone issues with Web Broker

### Question:

Why are my short term cookies not working in WebBroker?

### Answer:

When using Cookies in webbroker, it assumes all times are in GMT for cookies so setting a cookie to expire in 2 minutes will not work unless if your timezone is GMT.

Here is a workaround.

```
uses
  {$IFDEF MSWINDOWS}Windows, {$ENDIF}
  {$IFDEF LINUX}Libc, {$ENDIF}
  DateUtils;

resourcestring
  RSFailedTimeZoneInfo = 'Failed attempting to retrieve time zone
information.';

function OffsetFromUTC: TDateTime;
{$IFDEF LINUX}
var
  T: TTime_T;
  TV: TTimeVal;
  UT: TUnixTime;
begin
  gettimeofday(TV, nil);
  T := TV.tv_sec;
  localtime_r(@T, UT);
  // __tm_gmtoff is the bias in seconds from the UTC to the current
time.
  // so I multiply by -1 to compensate for this.
  Result := -1*(UT.__tm_gmtoff / 60 / 60 / 24);
end;
{$ENDIF}
{$IFDEF MSWINDOWS}
var
  iBias: Integer;
  tmez: TTimeZoneInformation;
begin
```

```

// Copied from IdGlobal.pas
case GetTimeZoneInformation(tmez) of
  TIME_ZONE_ID_INVALID:
    raise EFailedToRetrieveTimeZoneInfo.Create
(RSFailedTimeZoneInfo);
  TIME_ZONE_ID_UNKNOWN   :
    iBias := tmez.Bias;
  TIME_ZONE_ID_DAYLIGHT :
    iBias := tmez.Bias + tmez.DaylightBias;
  TIME_ZONE_ID_STANDARD :
    iBias := tmez.Bias + tmez.StandardBias;
else
  raise EFailedToRetrieveTimeZoneInfo.Create
(RSFailedTimeZoneInfo);
end;
{We use ABS because EncodeTime will only accept positive values}
Result := EncodeTime(Abs(iBias) div 60, Abs(iBias) mod 60, 0, 0);
end;
{$ENDIF}

```

Use it as: Expires := Now + (however long in TDateTimeFormat)+OffsetFromUTC;

such as: Expires := IncMinute(Now, DefaultTimeout) + OffsetFromUTC;

Last Modified: 16-OCT-01

## Article #27908: Where is TReport in Delphi 6?

### Question:

Where is the ReportSmith component TReport in Delphi 6?

### Answer:

ReportSmith was removed from Delphi after Delphi 2. It was replaced by QuickReports. Since that time the TReport component still shipped with Delphi in order for developers to maintain backward compatibility, but the component was hidden. In Delphi 6 the component has been completely removed from Delphi. However it is still possible to compile a project developed with TReport because the file Report.dcu is still included on the Delphi 6 CD, in the directory

D:\Install\program files\Borland\Delphi6\Lib\Delphi2

In order to compile the project that contains TReport copy the two files Report.dcu and RSCONSTS.dcu to your Delphi6\Lib directory.

Last Modified: 27-FEB-03

## Article #27914: Borland license information is either invalid or missing

### Problem:

Able to successfully register Delphi with Borland. A dialog box pops up stating the registration was successful. The only button on the dialog box is Exit, and once you click exit an error message pops up: Borland license information is either invalid or missing.

### Solution:

Navigate to the bin directory of Delphi 6. If you installed to the default directory it is located at:

C:\Program Files\Borland\Delphi6\bin

Run the file: "**D6RegClean.exe**". Now, insert the Delphi 6 CD and run the installer. Select the option to repair Delphi 6. Once the repair has completed, start Delphi and complete the registration process. This time it will complete successfully.

Last Modified: 26-DEC-02

## **Article #27924: Internal Error 2381, "Directory Does not Exist"**

### **QUESTION:**

When I am installing Delphi 6 I get an Internal Error that the directory doesn't exist and then the installation fails. How can I install Delphi 6?

### **ANSWER:**

This error is thrown by MSI when it can't find a directory. Typically this is due to permission problem that the user doesn't have. To ensure that you have the necessary permissions on your drive(s), please try the following:

- 1) In Windows Explorer, select the folder.
- 2) Right-click the folder and click Properties on the shortcut menu.
- 3) On the Security tab, click Permissions.
- 4) Select the Everyone group and click Full Control in the Type of Access list.
- 5) Check the Replace Permissions on Subdirectories check box.
- 6) Click OK.
- 7) Click Yes to the prompt confirming whether you want to replace permissions on all subdirectories.
- 8) Run Delphi Installer.

Last Modified: 26-OCT-01



## Article #27949: Unofficial WebSnap updates and bug fixes

### Unofficial WebSnap Bug Fixes

By Corbin Dunn [cdunn@nospam.borland.com](mailto:cdunn@nospam.borland.com)

Remove the nospam. from my email address when mailing me

#### Unofficial fixes - page 2 (these are post service pack 2 issues)

In general, applying these fixes require you to rebuild some of the VCL. The easiest way to do this, is to go to **Tools | Environment Options** and from the **Library** tab add **\$(DELPHI)\Source\Internet** to your Library Path.

These fixes are unofficial, are not supported by Borland, and are to be used at your own risk.

Quick Jumps:

[Crashes or hangs in your Web Application when it shuts down \(ie: ISAPI dll is unloaded, CGI exits, or Apache is stopped\).](#)

[Cannot perform this operation on a closed dataset](#)

[Transparent tags in any TPageProducer will have spaces "eaten"...](#)

[WebAppDebugger: Exception EOleException in module ModuleName.exe. Catastrophic failure](#)

[The OnPrepareStrings event of TCustomStringsValuesList never gets called.](#)

["Runtime Error 217" when unloading an ISAPI dll/CGI \(or crash of webserver on unloading of the app\)](#)

[WebSnap apps leak memory if any of the events in the SessionsService are used](#)

[When assigning things in script, you get an access violation at address xxx in module 'oleaut32.dll'.](#)

[Read of address 00000000](#)

[The application called an interface that was marshalled for a different thread](#)

[If you use the AdapterPageProducer to generate script, the SelOptions JScript will not show a value of 0...](#)

[An error occurred while attempting to initialize the Borland Database Engine \(error \\$2B05\)](#)

[If you are using the TDatasetAdapter, and the keyfield is a string which contains a single or double quote in it, it will not be properly encoded in actions.](#)

[Calculated fields in a dataset cause the TDatasetAdapter to always raise the exception "Field %s changed by another user".](#)

[Where are some good articles and updated documentation on WebSnap?](#)

[More additional information](#)

[On XP: Cannot create script engine: "SignedJavaScript". Error: 80040154](#)

[Memory leak fixes in WebSnap](#)

AV's when using the TDataSetAdapter and an exception is raised...

AV's when using MIDAS and TSocketConnection with WebSnap

AV's when using MIDAS and TSocketConnection with WebSnap (upon shutdown of the application).

This is fixed in Delphi 6 update pack 2.

When using the TDataSetAdapter, AV's will happen if an exception is raised in the opening/closing/next/prior of a dataset.

This is fixed in Delphi 6 update pack 2.

Memory leak fixes. TAutoObjectFactory objects that are created in WebAuto. TAbstractScriptComServerObject.RegisterScriptClass are never freed. Another 84 bytes are lost on each request with another bug.

This is fixed in Delphi 6 update pack 2.

The error: "Cannot perform this operation on a closed dataset" will happen if you click the Prior button in a WebSnap application, and you don't have any keyfields set.

This is fixed in Delphi 6 update pack 2.

Transparent tags in any TPageProducer will have spaces "eaten" in the tags if the ScriptEngine property is anything but blank. For instance, <#TagName ParamName="Value"> will become <#TagNameParamName="Value">.

This is fixed in Delphi 6 update pack 2.

After applying the Delphi 6 update pack 1 attempting to recompile and use the Web App Debugger always gives: "Exception EOleException in module ModuleName.exe. Catastrophic failure".

This is fixed in Delphi 6 update pack 2.

The OnPrepareStrings event of TCustomStringsValuesList never gets called.

This is fixed in Delphi 6 update pack 2.

"Runtime Error 217" when unloading an ISAPI dll or the termination of a CGI application happens. Or, your web server simply crashes when trying to unload a WebSnap ISAPI dll.

This is fixed in Delphi 6 update pack 2.

WebSnap apps leak memory if any of the events in the SessionsService are used (OnEndSession or OnStartSession).

This is fixed in Delphi 6 update pack 2.

If you try to assign something to an Adapter field's value in script, you get an:  
[Error] Unit2.htm(1): Access violation at address 779DB637 in module 'oleaut32.dll'. Read of address 00000000

Or, you try to assign something to the adapter itself, such as: <% Modules.  
PageProducerPage2.Adapter1 = 'alksjfkljsadf' %>

This is fixed in Delphi 6 update pack 2.

When running your WebSnap app you sometimes get: "The application called an interface that was marshalled for a different thread"

Or:

The error: -2147417842 (0x8001010e) is returned from any WebSnap application compiled as an ISAPI dll and run under IIS5 (Windows 2000) with the Application Protection on that directory is set to "Medium (Pooled)".

Note: This is fixed in Delphi 6 update pack 2, by having the Wizard generate this code for you. If you have existing apps, you will still have to apply this fix

The reason this happens is because the CoInitFlags are not being set to COINIT\_MULTITHREADED for an ISAPI dll (but, it is required!).

Add the following to the uses list of the main .dpr file:  
ActiveX, ComObj

and have the following in the begin..end:

begin

```
CoInitFlags := COINIT_MULTITHREADED;  
Application.Initialize;  
...
```

CoInitFlags MUST be set before Application.Initialize.

If you use the AdapterPageProducer to generate script, the SelOptions JScript will not show a value of 0 for a given name/value pair with a TDatasetValuesAdapter. For example, if you have a Value of 0 in the database, you will see:

```
<select size="0" name="A">  
<option>Bug</option>  
//          ^^^^^^ MUST BE <option value= "0">Bug</option>      !!!!!  
<option value="1">A</option>  
<option selected value="2">B</option>  
</select>
```

This is fixed in Delphi 6 update pack 2.

When using the BDE, I get the following error when starting my web application: "An error occurred while attempting to initialize the Borland Database Engine (error \$2B05)"

**Note:** This is a setup issue, not a Delphi programming problem. **Cause:** This error happens because the anonymous user that IIS starts your process as does not have permission to write to some temp directories (Typically, for security reasons, it can't write to the directory where inetinfo.exe is located in your System32\inetsrv directory.)

**Solution:** The Paradox tables were not designed for multiple users hitting the database at the same time. The best solution is to use a SQL based database, such as Interbase, Oracle, or MS SQL Server.

If you must use Paradox and the BDE, then be sure to have a TSession on your WebDataModule. Set AutoSessionName to true. The key thing is to set the NetFileDir and PrivateDir to something that the anonymous IIS account can access. A good solution would be to set the NetFileDir to: C:\Temp\Net and the PrivateDir to: C:\Temp. The two cannot point to the same directory. In addition, it is best to uniquely set the PrivateDir for each webmodule, in the OnCreate event (typically, you will create a directory based on the current thread id, and set the PrivateDir to this). Otherwise, you may see each other's data from different threads.

However, the NetDir MUST be shared across all processes and threads.

Next, you must give the IUSR\_MACHINE\_NAME and IWAM\_MACHINE\_NAME accounts full read and write permissions to the following directories:

1. The BDE directory
2. The base directory for your NetDir and PrivateDir
3. Your actual temp directory, probably C:\WINNT\Temp
4. Your directory that contains the database files
5. If all else fails, give read permissions to C:\WINNT, and then write permissions to C:\WINNT.

If you are using the TDataSetAdapter, and the keyfield is a string which contains a single or double quote in it, it will not be properly encoded in actions. This causes javascript errors on the client side browser, and the inability to do a locate on given records with quotes in the keyfield name.

This is fixed in Delphi 6 update pack 2.

Calculated fields in a dataset cause the TDataSetAdapter to always raise the exception "Field %s changed by another user". It is not possible to update a webpage using the TDataSetAdapter if the dataset contains calculated fields.

This is fixed in Delphi 6 update pack 2.

On XP, WebSnap apps give: Cannot create script engine: "SignedJavaScript". Error: 80040154

This is fixed in Delphi 6 update pack 2.

In any WebBroker application compiled with Delphi 6 update pack 1, the following symptoms may happen when unloading an ISAPI dll, exiting a CGI, closing down a WebAppDebugger project, or stopping Apache that contains a DSO:  
access violations, runtime errors, strange exceptions being raised.

This is fixed in Delphi 6 update pack 2.

Where are some good articles and updated documentation on WebSnap?

[WebSnap Server-side Scripting](#)

[New help files](#)

[UsingWebSnap.pdf](#)

[A database-enabled Web user list](#)

[Getting started with WebSnap in Delphi 6 Enterprise](#)

[Loggin' in ain't hard to do](#)

[Paging Dr. WebSnap!](#)

[Persistent user sessions in WebSnap](#)

[Access rights in WebSnap](#)

[WebSnap is now in session](#)

[WebSnap: using Sessions and TStringValuesList and server side includes \(source code from corbin\)](#)

[WebSnap: Guestbook source code \(Kylux 2, but should work under Delphi 6\)](#)

## More additional information

**The WebSnap Pack** by Jimmy Tharpe provides the following free (postcard-ware) components to enhance WebSnap (see his website for more information):

- TDBWebUserList - A replacement for TWebUserList that validates users through a database.
- TudEndUserSessionAdapter - A replacement for TEndUserSessionAdapter that adds EndUser.DisplayName.
- TudLoginFormAdapter - A replacement for TLoginFormAdapter.
- TudStringsValuesList - A replacement for TStringValuesList.

## Article #27958: Uninstalling Delphi 6 prompts for the CD even after saving the installation information to the hard drive

### Problem

During the installation of Delphi 6 the option to save the installation information to the hard drive was selected. When uninstalling Delphi 6, the following message is displayed: **"The feature you are trying to use is on a CD-ROM or other removable disk that is not available. Insert the 'Borland Delphi 6' disk and click OK."**

### Solution

The windows installer doesn't always seem to know where to look for the "Borland Delphi 6.msi" file that you saved to your hard drive when you selected the option to store the installation information. Instead of inserting your Delphi 6 CD when presented with the message described above, select Browse and navigate to the directory where you installed Delphi 6 and select the "Borland Delphi 6.msi" file. The uninstall will now proceed normally.

Last Modified: 01-NOV-01

## Article #27966: Steps to uninstall and reinstall Delphi 6

### Problem

Delphi 6 isn't working correctly and has been messed up from the initial installation. Running the Delphi 6 Repair from the installation CD didn't fix the problems so now a full reinstall seems to be necessary.

### Solution

1. Go into the control panel and select **Add/Remove Programs**.
2. Select Remove Delphi 6. Proceed with the Uninstall.
3. Open up Windows Explorer. Navigate to the directory where Delphi 6 was installed. If you have projects that are saved in the **Delphi6\Projects** directory, move them to a different location outside of the Delphi6 folder. Delete the Delphi6 folder and all its sub-directories.
4. In Windows, click on Start and select Run. Type in:  
**regedit**
5. Navigate to **HKEY\_LOCAL\_MACHINE\Software\Borland\Delphi**  
If you see a **6.0** folder, select it and hit Delete.
6. Navigate to **HKEY\_CURRENT\_USER\Software\Borland\Delphi**  
If you see a **6.0** folder, select it and hit Delete.
7. Close regedit and reboot the machine. Ensure the user account that Delphi 6 will be installed on has administrative rights, as this is necessary to install Delphi.
8. Stop all background process that are running on the machine. This also means **stop/disable any anti virus program that is running**.
9. Place the Delphi 6 CD in the CD-ROM drive and install Delphi 6. Before installing the Delphi 6 Update or any third party components, start Delphi 6 and ensure it is working correctly.



## Article #27980: Problems registering Kylix

### Question:

Why are my codes not being accepted? Why am I being asked to register again after I've already installed and registered Kylix?

### Answer:

This most often occurs when Kylix (any edition) has previously been installed on the same machine. Here are some suggestions:

- Don't paste codes - type them in.
- If you installed Kylix as root, and are running as root or a user:
  - Rename your `$HOME/.borland` directory to `$HOME/.borland.bak` (or just delete it).
  - Start Kylix.
  - Register the Software again (you should still have your serial number, authorization key and registration code).
- If you installed Kylix as a user:
  - Remove 'registry.bin' file from `$HOME/.borland` directory.
  - Start Kylix.
  - Register the Software again.

If the previous suggestions fail then run the 'locale' command on your console window. If you have the default locale setting:

```
LANG=POSIX
LC_XXX=POSIX
LC_ALL=POSIX, C or blank
```

then, could you try to set 'en\_US' to your locale settings (or whatever you locale may be). To do this enter the following in a console window:

```
export LANG=en_US
export LC_ALL=en_US
```

then, use 'locale' command again to verify that the settings are correct.

### Other notes:

Different installation locations won't help since the registration information are always stored in the same place `/home//.borland` or `/root/.borland`.

You will have to register the Software for every machine you install Kylix on, and for every user who runs Kylix on a machine.

Last Modified: 28-OCT-02

## Article #27987: Error "InstallShield for Windows is unable to start..." when installing Delphi.

### Question:

What can I do about the message "InstallShield for Windows is unable to start. Ensure that the Windows Installer engine has been installed properly" while attempting to install Delphi?

### Answer

There is a utility called **instmsiw.exe** that comes on the Delphi CD. Run this (by double clicking it) and this will reinstall the Microsoft Installer on your machine.

Last Modified: 06-NOV-01

## Article #28032: Missing components in Kylix.

### Question:

Why am I missing components in Kylix after I just installed it?

### Answer:

**When you launch Kylix, did you get any errors indicating that a package could not be loaded? If so there may be other issues and you may need to reinstall. If you downloaded the open edition please be advised that it does not include all of the components so you will only see 4 tabs of components. If you are seeing one or no tabs then try the following:**

**If you install as root and launch kylix as regular user, then as the regular user execute the following commands:**

```
cd $HOME/.borland
rm delphi60*
cd
startkylix
```

**Last Modified: 16-NOV-01**

## Article #28034: Manually unistalling Kylix.

### Question

I am trying to reinstall Kylix after deleting a previous installation and I'm getting an error stating that an installation already exists. How can I remove it?

### Answer:

Kylix should be removed by running the uninstall. If this was not done or did not remove everything properly you can manually remove Kylix.

- \* Delete all files in your .borland and Kylix directories:
- \* Check your installed packages with the rpm command:

```
rpm -qa | grep kylix
```

Uninstall these packages with rpm command (or other utility to add/remove/modify rpm packages). You can use '-e' option for the rpm command to uninstall rpm packages. After you uninstall all of the kylix packages from your system you should be able to reinstall kylix.

Last Modified: 16-NOV-01

## Article #28057: How to run executables created in Kylix

### Question:

How do I run an executable created in Kylix?

### Answer:

Kylix produces ELF binaries so in order to run them outside the IDE you need to first run 'source kylixpath' in the kylix2/bin directory. This sets up the necessary environment variables. Then, in a command window, go the directory where your compiled binary is and enter **./Project1** if your executable is called Project1.

Last Modified: 29-NOV-01

## Article #28061: Getting error -145 when trying to apply the Delphi 6 Update

### Problem

Getting error -145 when trying to apply the Delphi 6 Update

### Solution

This problem usually occurs because the data in the RootDir key in the Windows Registry is blank. Use regedit, navigate to **HKEY\_LOCAL\_MACHINE\Software\Borland\Delphi\ 6.0** Look at the Data in the RootDir key. It should point to the location where you installed Delphi 6. In most cases where I have seen error -145 this value was blank. If yours is blank, modify the RootDir key and set the Value Data to the path where your Delphi 6 is installed. If you installed to the default location it would be:

**c:\Program Files\Borland\Delphi6**

Once you do this, retry the Delphi 6 Update.

Last Modified: 24-SEP-02

## Article #28063: Is Visibroker for C++ 4.5 supported with Delphi 6?

### Question

Is Visibroker for C++ 4.5 supported with Delphi 6 Enterprise?

### Answer

No, when you install Delphi 6 Enterprise and choose to install VisiBroker, the two versions of VisiBroker that you can choose from are VisiBroker 3.3 or VisiBroker 4.1 for C++. The .dll's that ship with Delphi that support VisiBroker were built and tested against the VisiBroker for C++ 4.1 ORB, so using VisiBroker for C++ 4.5 is currently not supported with Delphi 6 Enterprise Version 6.190.

Last Modified: 13-DEC-01



## Article #28070: How can I avoid expanding a tree node when double-clicking on it?

### Question

How can I avoid expanding a tree node when double-clicking on it?

### Answer

Handle the OnExpanding event. Use the Node parameter to determine if it's a leaf or not (i.e. Node.Level or Node.HasChildren). Also look at GetHitTestInfoAt to determine where the double click occurred because you will want to allow expanding when the user clicks the +.

Last Modified: 13-DEC-01

**Article #28072: How can I tell if I have downloaded the correct Delphi 6 Update?**

## **Question**

How can I tell if I have downloaded the correct Delphi 6 Update?

## **Answer**

As long as you downloaded the file after October 3rd 2001 you will have the correct Update for Delphi 6. The original update with the variants problem was only up on the Borland site for a couple of days before being withdrawn.

Last Modified: 03-JUN-03

## **Article #28084: Installing the EasySoft ODBC driver from InterBase 6 from Delphi 6**

Q. How do I install the EasySoft ODBC driver for InterBase 6 from Delphi 6.

A. Run SETUP.EXE from the ib6\ODBC directory.

Note that this install only installs a driver with a 7 day license. To get an unlimited driver you must either purchase the [media kit](#) or contact [EasySoft](#).

Last Modified: 06-DEC-01

## Article #28118: How to change Kylix directory permissions for all users when installed as root.

### Question:

How do I change the directory permissions for Kylix so all users can run it?

### Answer:

Even if you installed into the /usr directory you may need to change the permissions for that directory so other users can run Kylix. To do this open a terminal window. If you are not currently logged in as root you can temporarily log in as root within the terminal window by entering **su** and then entering your password. Then, as root, enter the following:

```
cd /usr  
chmod -R 755 kylix/
```

As an alternative to using `chmod` you may be able to change directory permissions from within a graphical directory browser on your machine, by right clicking on the **kylix** directory.

Last Modified: 13-DEC-01

## Article #28176: Cannot access deployed EJB in BES 5.0 with a Delphi Client

### Problem

I installed Borland Enterprise Server 5.0 and deployed the EuroConverter EJB found in the EJB Demo that ships with Delphi 6. I cannot access the deployed EJB with the Delphi Client. When I try running the Delphi Client I keep getting Object Not Exist error.

### Solution

First off, ensure that the osagent port inside Borland Enterprise Server is set to the same port number as the osagent that is running on the client machine.

In order for a Delphi Client to talk to an EJB deployed on BES 5.0, you must configure the Borland Enterprise Server so that SIDL is enabled. With Borland Application Server, this was always enabled as a default, however with BES 5.0 the default setting has SIDL disabled. The easiest way to enable SIDL is using the BES console. Do the following after starting up the BES server:

- Start the Console
- Expand your server
- Expand the Partition where you deployed the Enterprise Java Bean.
- Under the Partition, you will see Deployed Modules and Partition Services. Expand Partition Services.
- Select the EJB Container, right click on it and select Configure.
- Click the check box next to Enable SIDL. Click Okay.
- You will be presented with the message: "The partition needs to be restarted in order for this change to take effect. Do you want to restart the partition now?". Click Yes.

Now when you start your Delphi client it will be able to access the deployed EJB.

Last Modified: 05-FEB-02

## Article #28178: Possible solutions for IDE Internal Error in Kylix 2.

### Question:

Why am I getting the an IDE Internal Error when loading Kylix?

### Answer: Here are some suggestions

If it is a conflict with the QT library paths you can try entering the following in a terminal window (command prompt):

```
$export LD_LIBRARY_PATH=  
$export LTDL_LIBRARY_PATH=  
$startkylix
```

If it is not a QT conflict, then another possibility for Redhat 7.1 is to try running XConfigurator as root from a term window. For other distributions this may or may not work. It is possible that this is a problem with XFree86 V4.0.3 so you can try upgrading to XFree86 V4.1 or downgrading to V3.3.6, or just comment the pex5 line in your /etc/X11/XF86Config (or XF86Config-4) and restart X:

```
Section "Module"  
    Load      "xtt"  
    Load      "ddc"  
    Load      "GLcore"  
    Load      "dbe"  
    Load      "dri"  
    Load      "extmod"  
    Load      "glx"  
#    Load     "pex5"  
    Load      "record"  
    Load      "xie"
```

Last Modified: 31-DEC-01

## Article #28182: Timeout problems with SOAP services in Delphi 6.

### Question:

I'm writing SOAP servers in Delphi 6 with service pack 1 and IndyApp. How can I set the timeout of SOAP requests, to adapt it to the type of the requests?

### Answer:

IF the client is Delphi 6, SP1 then the problem is that Borland uses WinInet which has problems with TimeOuts. There's the InternetSetOption API to configure timeouts but it has been broken for a very long time now:

<http://support.microsoft.com/default.aspx?scid=kb;EN-US;q176420>

It used to be that only the SEND\_TIMEOUT and CONNECT\_TIMEOUT were failing but it seems that as of IE5, RECEIVE\_TIMEOUT also timed out.

There's a workaround involving another thread that Borland did not implement. It is documented at:

<http://support.microsoft.com/default.aspx?scid=kb;en-us;Q224318>

Last Modified: 31-DEC-01

## Article #28185: How to keep seconds from being included in the display when using TDateTimePicker

### QUESTION:

How can you keep seconds from being included in the display when using TDateTimePicker?

### ANSWER:

- Note: This only applies when you have the TDateTimePicker Kind property set to dtkTime.

Drop a TDateTimePicker component on your Form. Now **add CommCtrl to the uses** clause of your unit. Place the following code in the FormCreate event:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    DateTime_SetFormat(DateTimePicker1.Handle,  
        'hh:mm tt');  
end;
```

For more formatting options, look up DateTime\_SetFormat from the Microsoft Platform SDK.

Last Modified: 02-JAN-02



## Article #28186: How can one prevent a TForm from being moved or resized?

### QUESTION:

How can one disable a TForm from being moved or resized?

### ANSWER:

Place the following code in the FormCreate event:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    hMenuHandle : HMENU;  
begin  
    hMenuHandle := GetSystemMenu(Form1.Handle, FALSE); //Get the handle  
of the Form  
    if (hMenuHandle <> 0) then  
        DeleteMenu(hMenuHandle, SC_MOVE, MF_BYCOMMAND); //disable moving  
        DeleteMenu(hMenuHandle, SC_SIZE, MF_BYCOMMAND); //disable resizing  
end;
```

If you also want to prevent the user from minimizing and maximizing the form, in the Object Inspector, select the form you are concerned with and under the BorderIcons, set the biMinimize and biMaximize properties to False.

Last Modified: 02-JAN-02

## Article #28187: Delphi 6 on Windows XP

### Question:

Can I use Delphi 6 with Windows XP?

### Answer:

Delphi 6 was not fully certified to be used on Windows XP because it came out before Windows XP. However the second update pack is certified for XP so be sure to install this.

Last Modified: 23-APR-02

## **Article #28192: Klyix 2 Not Registered**

You may have successfully register Kylix 2, but when you try to start Kylix you get an error that it isn't registered.

In the ./borland directory is a file "registry.bin". Remove registry.bin and try registering the product again. This should resolve the problem.

Last Modified: 30-OCT-02

## Article #28214: How to make the TDateTimePicker display blank.

### Question:

How do I make the TDateTimePicker display blank?

### Answer:

A little trick is to use the letters **gg** in the format property. This format pattern displays the period or era. This pattern is ignored if the date to be formatted does not have an associated period or era string - which TDateTimePicker does not - thus the display is blank. When a date is subsequently selected you can then set the format to "" in the OnChange or OnClick event handler in order to get it to display again.

The format property was added in Delphi 6. In earlier versions add CommCtrl to the uses clause of your unit and use the following code:

```
DateTime_SetFormat(DateTimePicker1.Handle, 'gg');
```

Last Modified: 09-JAN-02

## Article #28236: Component properties do not appear in the Translation Manager

### Problem:

Certain component properties, such as DBGrid column titles, do not appear in the Translation Manager, causing them to be stuck in the native language the application was developed in.

### Solution:

Try changing the default value of the property. For example, change the `DBGrid.Columns.Column1.Title.Caption` in the IDE to something other than the default. Then click Project, Languages, Update Resource DLLs. The properties should then appear in the Translation Manager.

### Explanation:

The different languages in your project are governed by resource DLLs. When you generate a resource DLL, the wizard examines the project forms' DFM files, looking for strings and values you may want to change between languages. If you examine an example form's DFM file, you'll find that the DFM contains no data for the column title. This is the reason the TM does not contain these properties.

But then, how can the column titles appear at design time and run-time? The titles are defaulting to the corresponding field name on the table it's representing. The IDE saves space in the DFM by omitting properties that can fall back on their default values, but that doesn't help the Translation Manager do it's job.

When we explicitly assign a value to these properties, we force the IDE to write these property values to the DFM which, in turn, makes the Translation Manager aware of them.

Last Modified: 13-MAR-02

## Article #28270: ISAPI DLL used to work with Delphi 5 but doesn't work in Delphi 6

My Delphi5 Isapi Dlls run fine from Netscape, but when I create an Isapi DLL in Delphi6 it fails to work in a Netscape FastTrack WebServer.

Delphi6 adds a Unit to the project that didn't exist in Delphi5. This new unit will keep the ISAPI DLL from loading into Netscape Server. To correct this problem comment out the "ISAPIThreadPool" unit from your Delphi6 source and recompile.

Your new DPR for an ISAPI dll might look something like this:

```
library ISAPI_Prog;  
  
uses  
    WebBroker,  
    // ISAPIThreadPool, comment this out for Netscape  
    ISAPIApp,  
    Unit1 in 'Unit1.pas' {WebModule1: TWebModule};  
  
{ $R *.RES }  
  
exports  
    GetExtensionVersion,  
    HttpExtensionProc,  
    TerminateExtension;  
  
begin  
    Application.Initialize;  
    Application.CreateForm(TWebModule1, WebModule1);  
    Application.Run;  
end.
```

**Article #28276: Access violation in dcc60.dll and Internal Error LA33 in Delphi 6.**

**Question:**

What is causing an access violation in dcc60.dll and then internal error LA33 in Delphi 6?

**Answer:**

If you are using the Delphi 6 trial edition or you are running a project in Delphi 6 that was created with the Delphi 6 trial edition then the problem is with the .dcu files. The DCU format for the trial version is not compatible with that for the regular version. This is intentional.

In order to resolve this you will need to rebuild everything in the edition you are currently using. If rebuilding doesn't work for some reason, you can delete the .dcu files, which will force them to be recreated.

Last Modified: 24-JAN-02

## Article #28277: ItemIndex property of TStaticActionList and TVirtualActionList does not represent which item is selected.

### Question:

Why is the ItemIndex property of TStaticActionList and TVirtualActionList not indicating which item is selected?

### Answer:

This is the designed behavior. In the help for TCustomListAction.OnItemSelected it states "Before this event is called, the list action first updates the ItemIndex property of any client controls that descend from TCustomListControl."

And in the help for TCustomListAction.ItemIndex "Note: List actions do not supply ItemIndex to all clients. The ItemIndex property controls only the ItemIndex property of clients that descend from TCustomListControl."

TStaticListAction does not descend from TCustomListControl.

If you want to get the selected text for these components you can do it like this:

```
ShowMessage( (Control as TCustomActionCombo).SelText );
```

Last Modified: 24-JAN-02



## Article #28298: Loading a TListView component that contains SubItems.

### QUESTION:

How do I load the SubItems of a TListView component?

### ANSWER:

Add a TListView to a form. Next, right click the ListView components to invoke the sub-menu, and select Columns Editor. Add two columns and set the Width property to 300. In the TListView properties set the ViewStyle to vsReport. Drop a TButton on the form and add the code sample below to the button's OnClick event.

```
procedure TfrmMain.Button1Click(Sender: TObject);
var
    NewItem: TListItem;
begin
    NewItem := ListView1.Items.Add;
    NewItem.Caption := 'FirstItem'; //some string.
    NewItem.SubItems.Add('SecondItem'); // next string.
end;
```

Last Modified: 05-FEB-02

## Article #28301: How to make your main form fill up the Desk Top space.

### QUESTION:

How do I make sure my main form occupies all of the desktop space when my application is run?

### ANSWER:

Below is a code sample that demonstrates the process.

```
procedure TForm1.FormCreate(Sender: TObject);
var
    DeskTopForm: TRect;
begin

(* You'll find more options in the Win32.hlp file under SystemParametersInfo *)
    SystemParametersInfo(SPI_GETWORKAREA, 0, Pointer(@DeskTopForm), 0);

    Top := DeskTopForm.Top;
    Left := DeskTopForm.Left;
    Height := DeskTopForm.Bottom - DeskTopForm.Top + 1;
    Width := DeskTopForm.Right - DeskTopForm.Left + 1;
end;
```

Last Modified: 05-FEB-02

## Article #28321: How to create an example of your problem for Borland Developer Support

### Question:

The engineer working on my case wants a small example that demonstrates the issue. What should I provide?

### Answer:

Many developer support cases deal with problems that simply do not occur in code generated by the IDE. In these situations, it is very helpful to the engineer working on your case to have a small example that demonstrates the problem so they can do testing on their end. Having the engineer work out their own example of the problem takes more time which means a delayed solution. Also, not knowing exactly what to send to your engineer can lead to a lot of back-and-forth exchanges which can further delay the resolution. What follows is a list of suggestions to help you provide a good example of the issue to your engineer.

1. **Make sure your example is complete in itself:** An incomplete example will only make the case take longer. For instance, an example of your run-time database connectivity failure which doesn't compile due to missing auxilliary files. Be sure to either include the necessary files to get the example to run or remove the references to unnecessary files before sending it to the engineer. Test it before you send it to make sure it functions in illustrating the errant behavior.
2. **Make the example atomic:** That is to say, make it so that it cannot be reduced to something simpler. The example should only include source code and libraries related to the issue. This may require you to rework your code before sending it but your case will go faster this way.
3. **Try to make it generic:** Examples that are dependant on specific third-party software are much harder to deal with. Borland Developer Support does not have every permutation of database, application server, utility component set and operating system available to them for testing. Everywhere you can, remove third-party components and use the database or app server that came bundled with the IDE. It is understandable that sometimes this is not possible, but doing this when you can will speed up the resolution of your issue considerably.
4. **Make the example a complete project:** An example that can be opened as a normal IDE project is always preferable to an unmanaged heap of source code. Most of our products use a "project" abstraction as a way of managing not only source code, but also auxilliary libraries and build options. Sending a complete project is like sending a road map of your application.

5. **Send it in the structure it appears in on your drive:** Meaning once you have created an example that demonstrates the problem, the whole problem and nothing but the problem, zip it up right there and send the whole thing, folders and all. This will help eliminate time spent recreating your machine's directory structure and time spent hunting down missing files.

Following these guidelines will give your engineer the tools needed to solve the case quickly.

Last Modified: 30-AUG-02

## Article #28322: How to generate steps to reproduce your issue for Borland Developer Support

### Question:

My engineer asked me for a sequence of steps that causes my issue to occur. What should I provide?

### Answer:

A sequence of steps to reproduce an issue is the best thing you can give to the engineer working on your problem. Reproducible problems are easier to solve which means quicker resolution for you.

The more detailed the steps are, the better. Making one step per action (like filling in a field or selecting a radio button) is great but can be a little lengthy. In general, one step per dialog or equivalent action is acceptable. Below are two examples of steps given to create an EJB example, one lacking and one satisfactory:

1. Create an EJB.
2. Put it in your module.
3. Build the project.

This lacks many important details that the engineer would need to track down the problem. Don't provide steps like this. Instead, use the following as a model:

1. Create a new Project: File -> New Project...
2. Accept the defaults and hit Finish
3. Create a new EJB Module: File -> New... -> Enterprise -> EJB Module
4. Accept the defaults and hit OK.
5. Add a bean: File -> New... -> Enterprise -> Enterprise JavaBean 1.x
6. Select the EJB Module you just created. Click Next.
7. Accept defaults. Make sure Stateless Session Bean is selected. Click Next.
8. Accept defaults and click Finish.
9. Build the project: Project -> Rebuild Project...

You can see in this example that steps, values, and selections are distinct. There are many ways to do the same thing in JBuilder so the more detail you provide, the quicker the case will go.

## Article #28323: What to have ready before you call Borland Developer Support

### Question:

What resources should I have in front of me when I call Developer Support?

### Answer:

You will definitely want all of the following information before calling Borland Developer Support:

1. Your product name, version number, and SKU (Professional, Enterprise, etc.)
2. Your operating system type and version number.

Generally, this information is also helpful:

1. Sample code or project that illustrates the issue ready to email.
2. Steps to reproduce the problem.

You should be at the computer exhibiting the issue. Not having the machine at hand makes it nearly impossible to solve the case. If you are not the developer/user who encountered the issue, please try to have them on hand to answer questions. Providing all of these things will let the engineer working on your problem operate at maximum efficiency.

## Article #28327: How to detect the CPU type of the computer the application is running on.

### QUESTION:

How can I find out what type of CPU the computer has that my application is running on?

### ANSWER:

Below is a code sample that demonstrates the process. Make sure you add Registry to your uses clause.

```
function CPUType: string;
var
  Reg: TRegistry;
begin
  CPUType := '';
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    if Reg.OpenKey('\Hardware\Description\System\CentralProcessor\0', False) then
      CPUType := Reg.ReadString('Identifier');
  finally
    Reg.Free;
  end;
end;
```

Last Modified: 05-FEB-02

## Article #28328: Getting the date of a file using Delphi.

### QUESTION:

How can one get the date of a file programmatically?

### ANSWER:

Place a TOpenDialog component on a form along with a TButton and a TLabel. Place the following code in Button1.OnClick event:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    OpenDialog1.Execute; //Use the TOpenDialog component to select the  
file we are interested in  
    Label1.Caption := DateToStr(FileDateToDateTime(FileAge(OpenDialog1.  
FileName)));  
end;
```

Last Modified: 05-FEB-02



**Article #28329: How does one change the default directory that Delphi uses to save projects?**

**QUESTION:**

**How does one change the default directory that Delphi uses to save projects?**

**ANSWER:**

Select the Shortcut Icon that you use to start up Delphi. Right click on the icon and select **Properties**. Where it says **Start in:** replace the string in that field with the directory you would like Delphi to default to when you save projects. For example, the string I entered is:  
**C:\MyDocs\Delphi\Projects\**

Last Modified: 05-FEB-02

**Article #28330: How to clear the document listing found in the Windows Start Menu.**

**QUESTION:**

**How can one clear the document listing found in the Windows Start Menu?**

**ANSWER:**

Add ShlObj to the uses of your application. Place a TButton on a form. In the Button1.OnClick event place the following code:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    SHAddToRecentDocs(SHARD_PATH, nil);  
end;
```

Last Modified: 05-FEB-02

## Article #28366: Unofficial WebSnap Fixes Page 2

### Unofficial WebSnap Bug Fixes Page 2

By Corbin Dunn [cdunn@nospam.borland.com](mailto:cdunn@nospam.borland.com)

Remove the nospam. from my email address when mailing me

In general, applying these fixes require you to rebuild some of the VCL. The easiest way to do this, is to go to **Tools | Environment Options** and from the **Library** tab add **\$(DELPHI)\Source\Internet** to your Library Path. Or, you can copy the patches files to your current project directory and do a build all. These will only fix applications that are built without runtime packages.

These fixes are unofficial, are not supported by Borland, and are to be used at your own risk.

#### Quick Jumps:

The OnValidate events of an Adapter and AdapterField don't get called on a regular adapter when an Action for that Adapter is executed.

Required property of AdapterField not working when the user inputs nothing into the field

Using the TDataSetAdapterImageField with Bitmaps in a database BLOB field will always give an "Invalid bitmap" error when trying to view the image.

#### Required property of AdapterField not working when the user inputs nothing into the field.

The fix for this problem is to modify the following function in WebAdapt.pas as follows:

```
function TAdapterUpdateField.ImplCheckValueChange(
  AActionRequest: IActionRequest; AFieldIndex: Integer): Boolean;
var
  FieldValue: IActionFieldValue;
  I: Integer;
  Value: Variant;
begin
  // Test single value
  with GetActionFieldValues(AActionRequest) do
    FieldValue := Values[AFieldIndex];
  if GetValueCount = ConvertValueCount(FieldValue) then
    begin
      Result := False;
      if IsMultiValueField then
        for I := 0 to FieldValue.ValueCount do
          begin
            try
              if ConvertValues(FieldValue, I) <> GetValues(I) then
                Result := True;
            except
              Result := True;
            end;
            if Result then break;
          end
        else
          // Corbin - Add the Required check as a fix.
```

```

Value := GetValue;
if VarIsEmpty(Value) then
begin // I added the "or Required" on the line below
  if (ConvertValue(FieldValue) <> '') or Required then
    Result := True
end
else
  try
    if ConvertValue(FieldValue) <> Value then
      Result := True;
    except
      Result := True;
    end;
end
else
  Result := False;
end;

```

**The OnValidate events of an Adapter and AdapterField don't get called on a regular adapter when an Action for that Adapter is executed.**

The fix for this problem is to modify the following function in WebAdapt.pas as follows:

```

procedure TCustomAdapterAction.ImplExecuteActionRequest(AActionRequest:
IActionRequest;
  AActionResponse: IActionResponse);
var
  S: TStrings;
begin
  // corbin - make sure the adapter updates its records before calling the OnExecute
event
  Adapter.UpdateRecords; // Add this line before the rest
  // ....

```

Or, you can call (YourAdapterName).UpdateRecords as the first thing in your OnExecute event of the AdapterAction.

**Using the TDataSetAdapterImageField with Bitmaps in a database BLOB field will always give an "Invalid bitmap" error when trying to view the image.**

The fix for this problem is to modify the following function in DBAdaptImg.pas as follows. Goto line 238 (in RenderAdapterImage):

```

    else if (Bytes[0] = Ord('B')) and { Do not localize}
      (Bytes[1] = Ord('M')) then // Windows bitmap { Do not localize}
    {$IFDEF MSWINDOWS}
    begin // added begin
      ContentStream.Seek(0, soFromBeginning); // Missing in update pack 2 -
corbin - go back over the start of the bytes before converting
      ConvertBitmapToJpeg
    end // added end
    {$ENDIF MSWINDOWS}

```

I missed adding the seek back over BM before converting the bitmap to a jpeg.

Last Modified: 25-FEB-02

## **Article #28400: Problems using ADO in a WebSnap application.**

### **Question:**

Why am I getting an internal server error (IIS) or an error that CoInitialize has not been called (Apache) in my WebSnap application?

### **Answer:**

In any multi-threaded application you must manually call CoInitialize at the correct times if you use COM. Using ADO is using COM. In ISAPI, CGI or DSO applications each request is in a different thread, so you must call CoInitialize for each thread. Try calling it in the OnActivate, and call CoUninitialize in the OnDeactivate (requires you to use the ActiveX unit). Make sure your tables and connections are not open at design time.

It works in the web app debugger because CoInitialize has been called in the client web app (since, it is a com server).

Last Modified: 13-FEB-02

## Article #28411: Why can't I run my Kylix application outside of the IDE?

### Question:

What do I need to do to run a Kylix application outside of the IDE?

### Answer:

You need to have your `LD_LIBRARY_PATH` variable set to your bin directory. Your bin directory is located under your Kylix directory. You can set this by sourcing the `kylxpath` script. From an xterm window go to the Kylix bin directory and enter

```
source kylxpath
```

See the README file for more details.

Last Modified: 19-FEB-02

## Article #28413: Where is the browser pane in the web page editor in Kylix?

### Question:

Where is the browser tab in the web page editor?

### Answer:

Kylix 2 does not have a browser pane in the web page editor. This feature exists in Delphi 6 but was not implemented in Kylix 2. It is erroneously mentioned in the on-line help.

The web page editor is invoked when you double click on an AdapterPageProducer component, or by going to the WebPageItems property in the Object Inspector.

*Note: The preview tab in code explorer should still be present. [\[If not, click here\]](#)*

Last Modified: 19-FEB-02



## Article #28414: Where is the web page preview pane in the code editor?

### Question:

Where is the HTML preview pane in the code editor?

### Answer:

The tab for the preview pane may simply be out of sight if your window is too narrow. Try making the code explorer window wider. The Preview tab is always the farthest to the right.

If you are using Kylix and you still can't see the preview tab you should see the *Browser Preview feature issues* section of the README file, in your Kylix directory regarding the Mozilla Preview Widget. This is an installation option for Kylix that you may have left out.

Last Modified: 19-FEB-02

## Article #28417: Why can't I get breakpoints to work when debugging a COM+ or MTS app in Delphi?

### Question:

Why can't I get breakpoints to work when debugging a COM+ or MTS app in Delphi?

### Answer:

Here are some possibilities:

You may need to set the *Include remote debug symbols* option under Project | Options | Linker.

Make sure to shutdown the COM+ application that contains the DLL first. This can be done from the Component Services Admin utility.

Check that the Shutdown Timeout value is not set to some small amount (see Delphi help for more).

Last Modified: 19-FEB-02

## Article #28419: Web App Debugger incorrectly formats dates based on the current regional settings

The Web App Debugger in Delphi 6 (or 6.01 or 6.02) has a bug in it where it incorrectly formats dates based on the current regional settings. This can hide other bugs which may only appear when you deploy your application to a real web server such as IIS or Apache.

Luckily, the source for the Web App Debugger is included with Delphi 6. Open up the project WebAppDbg.dpr in \$(DELPHI)\source\internet. Then, open the file SvrHTTP.pas. On line 678, you will see the start of the procedure TWebServerRequest.SetDateVariable. Modify it as follows:

```
procedure TWebServerRequest.SetDateVariable(Index: Integer;
  Value: TDateTime);
begin
  // This previously encoded the date based on the current locale,
  // which would hide other bugs if you didn't use the web app
  debugger.
  StringVariable[Index] :=
    Format(FormatDateTime('%s', dd '%s' yyyy hh:mm:ss 'GMT',
Value), { do not localize }
  [DayOfWeekStr(Value), MonthStr(Value)]);
  // StringVariable[Index] := DateTimeToStr(Value) // Old encoding
end;
```

Compile the application and copy the executable to your \$(DELPHI)\bin directory. That should be it! You will now have a patched version of the Web App Debugger.

Last Modified: 20-FEB-02

## Article #28431: Problems with IBExpress when installing the Delphi 6 Update Pack 2

### PROBLEM:

After installing the Delphi 6 Update Pack 2, when starting up Delphi the following error is displayed: *The procedure entry point @Ibxmlheader@initialization\$qqrv could not be located in the dynamic link library ibxpress60.bpl*

### RESOLUTION:

IBX update 6.03 for Delphi 6 must be installed after installing Update Pack 2 for Delphi 6. This can be obtained at:

<http://codecentral.borland.com>

Once at the codecentral website, click on Find and do a search by ID number. The ID number to enter is 17555

The following description explains how to resolve the problem after downloading the updated IBX:

If you apply update pack 2 to D6, the ibx 6.03 update should be run afterwards. If you have ibx 6.02 installed and install the update pack 2 for D6, you will get a function not found when loading D6 the next time. 6.03 will solve this. Alternatively you can delete the ibxpress60.bpl from the directory and reinstall 6.02. The update pack 2 does not have the 6.5 XMLHeader code in its bpl, but does not update dclib60.bpl which as of IBX 6.02 will look for these functions.

IBX 6.02's version is less than D6's update pack 2 so the installer will not just overwrite which is why you need to remove the update pack 2 version. IBX 6.03 does have a higher version number so will overwrite just fine.

Last Modified: 28-FEB-02

## Article #28445: Endless Registration Loop when Trying to Register Delphi 6

Delphi reports that I have registered the product successfully, however when Delphi launches it reports a message that Delphi 6 isn't registered.

### **Answer:**

The problem seems to happen mainly on Windows 2000, XP and there are three easy steps to get Delphi back on the "up and up." The process will require that you have you CD handy, so make sure it is available before you begin.

1. Locate the utility D6RegClean in the Delphi6\Bin directory. Run this utility. \*Note\* That any third party components will need to be reinstalled once this process has been completed.
2. Next put your CD into the CDROM and run the repair install.
3. The final thing to do is to register again. You can either run D6Reg, which is located in the Delphi6\Bin directory, or you can start Delphi.

Last Modified: 28-FEB-02

**Article #28447: How to disable mouse clicking in a TWebBrowser**

**QUESTION:**

**How does one disable the mouse from being used inside a TWebBrowser?**

**ANSWER:**

Place a TPanel on your form and drop the TWebBrowser on top of the TPanel. Then, change the TPanel's Enabled property to false. This will disable the use of the mouse inside the WebBrowser.

Last Modified: 28-FEB-02

**Article #28451: How can I use a dialog window to have a user select a folder without a file name?**

**QUESTION:**

**How can I use a dialog window to have a user select a folder without a file name?**

**ANSWER:**

You can accomplish this without using a TOpenDialog component. First, add FileCtrl to the Uses. Now the SelectDirectory function can be used. Drop a TButton component on your Form. Place the following code in the button's onClick event to get a demonstration of how this function works:

```
procedure TForm1.Button1Click(Sender: TObject);  
const  
    SELDIRHELP = 1000;  
var  
    dir: String;  
begin  
    dir := 'C:';  
    if SelectDirectory(dir, [sdAllowCreate, sdPerformCreate, sdPrompt],  
SELDIRHELP) then  
        Button1.Caption := dir;  
end;
```

For an explanation of the parameters that SelectDirectory takes, look up the function inside Delphi Help.

Last Modified: 28-FEB-02

## Article #28457: How to disable a Menu item with out greying it out.

### QUESTION:

How can I disable a Menu item, with out greying it out?

### ANSWER:

There are times when you find that you want to disable an option while not greying it out completely. Drop a TMainMenu component on a Form and add atleast two menu items. Next drop a TButton on the Form and in the OnClick event, add the code show below. The code sample below demonstrates how to disable a Menu item without greying it out.

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Menu: TMenu;  
begin  
    Menu := Application.MainForm.Menu;  
    // The first Menu item will be disabled, but not greyed out.  
    EnableMenuItem(Menu.Handle, Menu.Items[0].Command, MF_DISABLED);  
end;
```

Last Modified: 05-MAR-02



**Article #28467: How do you display the current line number of a selected record in a DBGrid?**

**QUESTION:**

**How do you display the current line number of a selected record in a DBGrid?**

**ANSWER:**

Substitute the Table1 component for whatever dataset you are using to provide data to your DBGrid.

**ShowMessage(IntToStr(Table1.RecNo));**

Last Modified: 28-FEB-02

## Article #28473: How can I log off the current user from within my application?

### QUESTION:

How can I log off the current user from within my application?

### ANSWER:

You can accomplish this using the WindowsAPI call `ExitWindows`. To see how this works, drop a `TButton` component on a form and type in the following code:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ExitWindows(0, 0); //both parameters for this call are reserved and  
    you must pass in zero's  
end;
```

Last Modified: 28-FEB-02

**Article #28475: How to ensure that a path name ends with a trailing path delimiter ('\').**

**QUESTION:**

How can I make sure that a backslash is added to the end of a path

**ANSWER:**

Delphi has a function called: `IncludeTrailingPathDelimiter`. This will return a string with a backslash '/' added to the end, if it does not already have one. Below is one example of how this function works. There is another function called: `IncludeTrailingBackslash`, this is only included for backwards compatibility (this function should not be used).

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    DirPath : string;  
begin  
    DirPath := Edit1.Text;  
    ShowMessage(IncludeTrailingPathDelimiter(DirPath));  
end;
```

Last Modified: 28-FEB-02

## Article #28478: How to remove the backslash from the end of a path.

### QUESTION:

How can I make sure there is no backslash at the end of a path?

### ANSWER:

Delphi has a function called: `ExcludeTrailingPathDelimiter`. This will return a string with one backslash '/' removed from the end if it has one (or more). Below is one example of how this function works. There is another function called: `ExcludeTrailingBackslash`, this is only included for backwards compatibility (this function should not be used).

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    DirPath : string;  
begin  
    DirPath := Edit1.Text;  
    ShowMessage(ExcludeTrailingPathDelimiter(DirPath));  
end;
```

Last Modified: 28-FEB-02

## Article #28494: *Operation Not Supported* error when calling TSQLQuery.RecordCount

### Question:

Why am I getting the error 'Operation Not Supported' when calling TSQLQuery.RecordCount?  
This error may also be preceded by the exception 'ORA-00942: table or view does not exist' in the IDE.

### Answer:

You should note from the Delphi help that reading RecordCount will generate an exception if the dataset can not determine the number of records. Do not read RecordCount if:

- The dataset represents stored procedure.
- The dataset represents a query that contains parameters.
- The dataset represents a multi-table join.

If you are not doing any of these and you are still getting the error you should note that the SQL in TSQLQuery can be case sensitive. For example, if your table name shows up in SQL Explorer or in the TableName property of TSQLTable as **CUST2** and your query is:

```
Select * from cust2
```

then you may get the error, but if your query is:

```
Select * from CUST2
```

then it should work.

Last Modified: 05-MAR-02

## Article #28514: Unofficial SOAP Bug Fixes

### Unofficial SOAP Bug Fixes

By Bruneau Babet [bbabet@nospam.borland.com](mailto:bbabet@nospam.borland.com)

Remove the nospam. from email address when mailing me

In general, applying these fixes require you to rebuild some of the VCL. The easiest way to do this, is to go to **Tools | Environment Options** and from the **Library** tab add **\$(DELPHI)\Source\Internet;\$(DELPHI)\Source\SOAP** to your Library Path.

These fixes are unofficial, are not supported by Borland, and are to be used at your own risk.

#### Quick Jumps:

- [The case of serialized Boolean members of a complex type is incorrect \(i.e. 'True'/'False' instead of 'true'/'false'\)](#)
- [Memory leak in Servers that expose WideString parameters](#)
- [Error publishing Webservice's WSDL when MSXML4 is installed](#)
- [HTTPRIO component reloads a Service's WSDL for each Webservice call](#)
- [\(Ole\)Variant array of one element are incorrectly deserialized](#)
- [Bad XML Namespaces for 'dateTime' and other XSBuiltIn types when members of Complex Type](#)

#### The case of serialized Boolean members of a complex type is incorrect (i.e. 'True'/'False' instead of 'true'/'false')

##### **Description:**

When serializing Boolean members of a TRemotable-descendant class, the value of the latter are sent as 'True' or 'False'. Some SOAP implementations, including Delphi's, will accept these values. However, many will complain, and for good reason, since the legal literal representations of a boolean are '1', '0', 'true' or 'false'.

##### **Fix:**

The fix for this problem is to modify the following function in OPToSOAPDomConv.pas as follows:

```
function TSOAPDomConv.GetObjectPropAsText(Instance: TObject;
  PropInfo: PPropInfo): WideString;
var
  I: LongInt;
  E: Extended;
  I64: Int64;
begin
  case (PropInfo.PropType)^.Kind of
    tkInteger:
      begin
        I := GetOrdProp(Instance, PropInfo);
        Result := IntToStr(I);
      end;
    tkFloat:
      begin
        E := GetFloatProp(Instance, PropInfo);
        Result := FloatToStrEx(E);
      end;
    tkWString:
```

```

    Result := GetWideStrProp(Instance, PropInfo);
tkString,
tkLString:
    Result := GetStrProp(Instance, PropInfo);
tkInt64:
    begin
        I64 := GetInt64Prop(Instance, PropInfo);
        Result := IntToStr(I64);
    end;
tkEnumeration:
    Result := GetEnumProp(Instance, PropInfo);
    begin
        Result := GetEnumProp(Instance, PropInfo);
        if PropInfo.PropType^ = TypeInfo(System.Boolean) then
            Result := Lowercase(Result);
        end;
tkChar:
    begin
        I := GetOrdProp(Instance, PropInfo);
        Result := InvString(Char(I));
    end;
tkWChar:
    begin
        I := GetOrdProp(Instance, PropInfo);
        Result := InvString(WideChar(I));
    end;
tkClass:
    ;
tkSet,
tkMethod,

tkArray,
tkRecord,
tkInterface,

tkDynArray,
tkVariant:
    raise ESOAPDomConvertError.CreateFmt(SUnexpectedDataType, [KindNameArray
[(PropInfo.PropType)^.Kind]]);

end;

end;

```

## Memory leak in Servers that expose WideString parameters

### **Description:**

Delphi SOAP fails to delete WideStrings allocated by the framework on behalf of Servers that expose WideString parameters.

### **Fix:**

The fix for this problem is to modify InvokeRegistry.pas as follows:

## type

```
{...}
```

```
TDataContext = class
```

```
protected
```

```
  FObjsToDestroy: array of TObject;
```

```
  DataOffset: Integer;
```

```
  Data: array of Byte;
```

```
  DataP: array of Pointer;
```

```
  VarToClear: array of Pointer;
```

```
  DynArrayToClear: array of TDynToClear;
```

```
  StrToClear: array of Pointer;
```

```
  WStrToClear: array of Pointer;
```

```
public
```

```
  constructor Create;
```

```
  destructor Destroy; override;
```

```
  function AllocData(Size: Integer): Pointer;
```

```
  procedure SetDataPointer(Index: Integer; P: Pointer);
```

```
  function GetDataPointer(Index: Integer): Pointer;
```

```
  procedure AddObjectToDestroy(Obj: TObject);
```

```
  procedure RemoveObjectToDestroy(Obj: TObject);
```

```
  procedure AddDynArrayToClear(P: Pointer; Info: PTypeInfo);
```

```
  procedure AddVariantToClear(P: PVarData);
```

```
  procedure AddStrToClear(P: Pointer);
```

```
  procedure AddWStrToClear(P: Pointer);
```

```
end;
```

## implementation

```
{...}
```

```
procedure TDataContext.AddStrToClear(P: Pointer);
```

```
var
```

```
  I: Integer;
```

```
begin
```

```
  { If this string is in the list already, we're set }
```

```
  for I := 0 to Length(StrToClear) - 1 do
```

```
    if StrToClear[I] = P then
```

```
      Exit;
```

```
  I := Length(StrToClear);
```

```
  SetLength(StrToClear, I + 1);
```

```
  StrToClear[I] := P;
```

```
end;
```

```
procedure TDataContext.AddWStrToClear(P: Pointer);
```

```
var
```

```
  I: Integer;
```

```
begin
```

```
  { If this WideString is in the list already, we're set }
```

```
  for I := 0 to Length(WStrToClear) - 1 do
```

```
    if WStrToClear[I] = P then
```

```
      Exit;
```



```
I := Length(WStrToClear);
SetLength(WStrToClear, I + 1);
WStrToClear[I] := P;
end;
```

```
constructor TDataContext.Create;
begin
  inherited;
end;
```

```
destructor TDataContext.Destroy;
var
```

```
  I: Integer;
  P: Pointer;
begin
  { Clean up objects we've allocated }
  for I := 0 to Length(FObjsToDestroy) - 1 do
  begin
    if (FObjsToDestroy[I] <> nil) and (FObjsToDestroy[I].InheritsFrom(TRemotable))
then
      begin
        TRemotable(FObjsToDestroy[I]).Free;
      end;
    end;
  end;
  SetLength(FObjsToDestroy, 0);
```

```
{ Clean Variants we allocated }
for I := 0 to Length(VarToClear) - 1 do
begin
  if Assigned(VarToClear[I]) then
    Variant( PVarData(VarToClear[I])^ ) := NULL;
  end;
end;
SetLength(VarToClear, 0);
```

```
{ Clean up dynamic arrays we allocated }
for I := 0 to Length(DynArrayToClear) - 1 do
begin
  if Assigned(DynArrayToClear[I].P) then
  begin
    P := Pointer( PInteger(DynArrayToClear[I].P)^ );
    DynArrayClear(P, DynArrayToClear[I].Info)
  end;
end;
SetLength(DynArrayToClear, 0);
```

```
{ Clean up strings we allocated }
for I := 0 to Length(StrToClear) - 1 do
begin
  if Assigned(StrToClear[I]) then
    PString(StrToClear[I])^ := '';
  end;
end;
SetLength(StrToClear, 0);
```

```
{ Clean up WideStrings we allocated }
```

```

for I := 0 to Length(WStrToClear) - 1 do
begin
  if Assigned(WStrToClear[I]) then
    PWideString(WStrToClear[I])^ := '';
  end;
SetLength(WStrToClear, 0);

  inherited;
end;

{...}

procedure TInvContext.AllocServerData(const MD: TIntfMethEntry);
var
  I: Integer;
  Info: PTypeInfo;
  P: Pointer;
begin
  for I := 0 to MD.ParamCount - 1 do
  begin
    P := AllocData(GetTypeSize(MD.Params[I].Info));
    SetParamPointer(I, P);
    if MD.Params[I].Info.Kind = tkVariant then
      begin
        Variant(PVarData(P)^) := NULL;
        AddVariantToClear(PVarData(P));
      end
    else if MD.Params[I].Info.Kind = tkDynArray then
      begin
        AddDynArrayToClear(P, MD.Params[I].Info);
      end
    else if MD.Params[I].Info.Kind = tkLString then
      begin
        PString(P)^ := '';
        AddStrToClear(P);
      end
    else if MD.Params[I].Info.kind = tkWString then
      begin
        PWideString(P)^ := '';
        AddWStrToClear(P);
      end;
    end;
  end;
  if MD.ResultInfo <> nil then
  begin
    Info := MD.ResultInfo;
    case Info^.Kind of
      tkLString:
        begin
          P := AllocData(sizeof(PString));
          PString(P)^ := '';
          AddStrToClear(P);
        end;
      tkWString:
        begin
          P := AllocData(sizeof(PWideString));
          PWideString(P)^ := '';
          AddWStrToClear(P);
        end;
    end;
  end;
end;

```

```

tkInt64:
  P := AllocData(sizeof(Int64));
tkVariant:
  begin
    P := AllocData(sizeof(TVarData));
    Variant( PVarData(P)^ ) := NULL;
    AddVariantToClear(PVarData(P));
  end;
tkDynArray:
  begin
    P := AllocData(GetTypeSize(Info));
    AddDynArrayToClear(P, MD.ResultInfo);
  end;
else
  P := AllocData(GetTypeSize(Info));
end;
SetResultPointer(P);
end;
end;

```

## Error publishing WebService's WSDL when MSXML4 is installed

### **Description:**

As of SP#2, Delphi's msxmldom unit will attempt to use MSXMLDOM v4.0 if the latter is present. However, this may cause the creation and Publishing of a WSDL document by a Delphi WebService to fail. The typical symptom of this failure is that the client requesting the WSDL gets back an HTML document instead; and the document contains the following error message:

*Error: This name may not contain the ':' character*

### **Fix:**

The fix to this problem is to modify XMLDoc.pas as follows: (Note that this file is in the Source/Internet directory)

```

function TXMLNode.FindNamespaceDecl(const NamespaceURI: DOMString): IXMLNode;
var
  I: Integer;
  Attr: IXMLNode;
begin
  Result := nil;
  for I := 0 to AttributeNodes.Count - 1 do
    if SameNamespace(VarToStr(AttributeNodes[I].NodeValue), NamespaceURI) and
    (AttributeNodes[I].Prefix = SXMLNS) then
    begin
      Attr := AttributeNodes[I];
      if SameNamespace(VarToStr(Attr.NodeValue), NamespaceURI) and
        ((Attr.Prefix = SXMLNS) or (Attr.NodeName = SXMLNS)) then
        begin
          Result := AttributeNodes[I];
          Break;
        end;
    end;
  end;
end;

```

```

    if (Result = nil) and Assigned(FParentNode) then
        Result := FParentNode.FindNamespaceDecl(NamespaceURI);
    end;

procedure TXMLNode.DeclareNamespace(const Prefix, URI: DOMString);
begin
    if Prefix <> '' then
        SetAttributeNS(SXMLNS+NSDelim+Prefix, SXMLNamespaceURI, URI) else
        SetAttributeNS(SXMLNS, SXMLNamespaceURI, URI);
    end;

function TXMLNode.GetPrefixedName(const Name, NamespaceURI: DOMString): DOMString;
var
    NSDecl: IXMLNode;
begin
    { The method adds a prefix to a localname based on the specified URI.
      If there is no corresponding namespace already declared or if
      the name is already prefixed, then nothing is done. }
    if (doAutoPrefix in OwnerDocument.Options) and not IsPrefixed(Name) then
        begin
            NSDecl := FindNamespaceDecl(NamespaceURI);
            if Assigned(NSDecl) and (NSDecl.LocalName <> '') then
                if Assigned(NSDecl) and (NSDecl.NodeName <> SXMLNS) then
                    Result := MakeNodeName(NSDecl.LocalName, Name)
                else
                    Result := Name;
            end else
                Result := Name;
        end;
    end;
end;

```

## HTTPRIO component reloads a Service's WSDL for each WebService call

### **Description:**

When using the WSDLLocation property of a THTTPRIO to invoke a Web Service, each invocation results in a 'GET' of the WSDL document. NOTE: This bug was introduced when proxy support was added for the retrieval of the WSDL.

### **Fix:**

The fix to this problem is to modify WSDLNode.pas as follows:

```

{ ActivateWSDL }
function ActivateWSDL(WSDL: TWSDLItems; const Name: string; const Password: string;
const Proxy: string): Boolean;
begin
    Result := True;
    try
        { if not WSDL.Active then begin }
        if not WSDL.Active then
            begin
                WSDL.StreamLoader.UserName := Name;
                WSDL.StreamLoader.Password := Password;
                WSDL.StreamLoader.Proxy := Proxy;
            end;
    end;
end;

```

```

WSDL.Load(WSDL.FileName);
{ end }
end;

except
  on E: EDOMParseError
  begin
    Result := False;
    raise EWSDLLoadException.CreateFmt(SWSDLLError, [WSDL.FileName, E.Message]);
  end;
  on Ex: Exception do
  begin
    Result := False;
    raise Ex;
  end;
end;
end;

```

### (Ole)Variant array of one element are incorrectly deserialized

#### **Description:**

When a WebService Server or Client receives data that's deserialized into an OleVariant or Variant type, if the XML data sent is an array of a single element, the deserialization logic fails to see the data as an array; This causes a problem if the consumer of the (Ole)Variant expects an array. This problem may cause a MIDAS FetchParams of one parameter to fail, for example (while the call with two or more parameters succeed).

#### **Fix:**

The fix to this problem is to modify OPToSOAPDomConv.pas as follows:

```

procedure TSOAPDomConv.WriteVarArray(RootNode, Node: IXMLNode; Name: InvString; V:
Variant);
var
  I, DimCount: Integer;
  LoDim, HiDim, Indices: array of integer;
  V1: Variant;
  ElemNode: IXMLNode;
  VAPropSet: Boolean;
begin
  if not VarIsArray(V) then
  begin
    WriteVariant(RootNode, Node, Name, V);
  end
  else
  begin
    ElemNode := Node.AddChild(Name);
    DimCount := VarArrayDimCount(V);
    SetLength(LoDim, DimCount);
    SetLength(HiDim, DimCount);
    for I := 1 to DimCount do
    begin
      LoDim[I - 1] := VarArrayLowBound(V, I);

```

```

    HiDim[I - 1] := VarArrayHighBound(V, I);
end;
SetLength(Indices, DimCount);
for I := 0 to DimCount - 1 do
    Indices[I] := LoDim[I];
VAPropSet := False;
while True do
begin
    V1 := VarArrayGet(V, Indices);
    if VarIsArray(V1) and not VarIsType(V1, varArray or varByte) then
        WriteVarArray(RootNode, ElemNode, SDefVariantElemName, V1);
    else
begin
        WriteVarArray(RootNode, ElemNode, SDefVariantElemName, V1);
        ElemNode.SetAttributeNS(SVarArrayType, SBorlandTypeNamespace, VarType(V));
    end else
begin
        WriteVariant(RootNode, ElemNode, SDefVariantElemName, V1);
        if not VAPropSet then
            begin
                SetAttributeNS(RootNode, ElemNode, SVarArrayType, SBorlandTypeNamespace,
IntToStr(VarType(V)));
                ElemNode.SetAttributeNS(SVarArrayType, SBorlandTypeNamespace, VarType(V));
                VAPropSet := True;
            end;
        end;
    end;
    Inc(Indices[DimCount - 1]);
    if Indices[DimCount - 1] > HiDim[DimCount - 1] then
        for i := DimCount - 1 downto 0 do
            if Indices[i] > HiDim[i] then
                begin
                    if i = 0 then Exit;
                    Inc(Indices[i - 1]);
                    Indices[i] := LoDim[i];
                end;
            end;
        end;
    end;
end;
end;
end;

```

## Bad XML Namespaces for 'dateTime' and other XSBuiltIn types when members of Complex Type

### **Description:**

dateTime, decimal and other types implemented as a TXSxxxx class (in XSBuiltIn.pas) have incorrect namespaces when they are members of a complex (i.e. TRemotable-descendant) type.

### **Fix:**

The fix to this problem is to modify OPToSOAPDomConv.pas as follows:

```

function TSOAPDomConv.CreateObjectNode(Instance: TObject; RootNode, Node: IXMLNode;
                                       Name, URI: InvString; UsePrefix: Boolean):
InvString;

var

    {...}

    end
    else
    begin
        ClsType := GetTypeData((PropList[I].PropType)^).ClassType;
        RemClassRegistry.ClassToURI(ClsType, ElemURI, TypeName, IsScalar);
        MultiRef := MultiRefObject(ClsType);

        if IsScalar then
        begin
            ElemNode := InstNode.AddChild(ExtPropName);
            if not RemTypeRegistry.TypeInfoToXSD((PropList[I].PropType)^, ElemURI,
TypeName) then
                raise ESOAPDomConvertError.CreateFmt(SRemTypeNotRegistered,
[GetTypeData((PropList[I].PropType)^).ClassType.ClassName]);
                if not GetTypeData((PropList[I].PropType)^).ClassType.InheritsFrom
(TRemotable) then
                    raise ESOAPDomConvertError.CreateFmt(SScalarFromTRemotables,
[GetTypeData((PropList[I].PropType)^).ClassType.ClassName]);
                    {$IFDEF OPENDOM}
                        ElemNode.SetAttributeNS(SSoapType, XMLSchemaInstNameSpace, TypeName);
                    {$ELSE}
                        AttrNodePre := FindPrefixForURI(RootNode, Node,
XMLSchemaInstNameSpace);
                        ElemNode.Attributes[MakeNodeName(AttrNodePre, SSoapType)] := TypeName;
                    {$ENDIF}

                        ElemNode.Text := TRemotableXS(Obj).NativeToXS;
                        { Create node }
                        ElemNode := CreateScalarNodeXS(RootNode, InstNode, ExtPropName,
                                                        ElemURI, TypeName,
                                                        TRemotableXS(Obj).NativeToXS, True);
            end
        else
        begin
            if not MultiRef then
            begin
                if IsObjectWriting(Obj) then
                    raise ESOAPDomConvertError.Create(SNoSerializeGraphs);

                {...}

            finally
                FreeMem(PropList, Count * SizeOf(Pointer));
            end;
        end;
    end;
end;

```

## Links:

- [Top](#)
- [The case of serialized Boolean members of a complex type is incorrect \(i.e. 'True'/'False' instead of 'true'/'false'\)](#)
- [Memory leak in Servers that expose WideString parameters](#)
- [Error publishing WebService's WSDL when MSXML4 is installed](#)
- [HTTPRIO component reloads a Service's WSDL for each WebService call](#)
- [\(Ole\)Variant array of one element are incorrectly deserialized](#)
- [Bad XML Namespaces for 'dateTime' and other XSBuiltIn types when members of Complex Type](#)

---

Last Modified: 17-MAY-02



## Article #28520: How do I search backwards in the C++Builder/Delphi editor?

### Question:

The standard editor key mapping sets F3 as the "Find Next", but there doesn't appear to be a hot-key for "Find Previous". How can I search backward?

### Answer:

There is no hot key for "Find Previous". Most of the time, people only need to "Find Previous" when they've gone too far in their "Find Next". In this situation, you can use Ctrl-Z. Technically, this will "Select previously selected text", but will effectively "Find Previous" on items already "Found".

Last Modified: 17-JUL-02

## Article #28603: Unable to locate PDOXUSRS.NET when using Novell client 4.83

### Question:

I am receiving an error message from the BDE reporting it cannot connect to PDOXUSRS.NET located on my network server.

### Answer:

There is a file locking problem with Novell Client version 4.83 and this BDE error message is a direct result. The problem has been addressed with a patch released by Novell, located here: <http://support.novell.com/servlet/tidfinder/2961866>

Last Modified: 08-APR-02

## Article #28605: Project options window and other IDE windows appear too small.

### Question:

IDE windows such as the Project Options window appear too small to work with. How can I fix this problem?

### Answer:

If you are using a Matrox Dual-Head video card, the problem may be caused by their driver software. Try right clicking on the dual-head setup icon in the Taskbar (beside the time) or enter your display options, and disable the "Save/Restore Window positions" option.

Last Modified: 17-JUL-02

## Article #28615: What causes internal error T2261?

### Question:

What causes the internal compiler error t2261?

### Answer:

This is usually caused by using a **break** or **continue** statement outside of a loop construct. Breaks are designed exclusively for use inside of a loop.

Last Modified: 12-APR-02

## Article #28616: Getting a result set from an Oracle stored procedure in Delphi or C++Builder.

### Question:

How do I get a result set from an Oracle Stored Procedure?

### Answer:

A REF CURSOR is required to return a result set from a stored procedure. You will need to put the procedure in a package. To create a package use the following SQL command:

```
CREATE PACKAGE MYPKG
IS
    TYPE CursorType IS REF CURSOR;
END MyPkg;
```

Then when you create the procedure:

```
CREATE PROCEDURE RETURN_RESULT_SET
(oCursor IN OUT MyPkg.CursorType) AS
BEGIN
    open oCursor for select * from SomeTable;
END;
```

In Delphi or C++Builder, set the TStoredProc parameter DataType as type ftCursor. At this point you should treat the TStoredProc component like a TQuery and use the Open method instead of ExecProc.

You should also be sure you have the [latest BDE](#) installed.

Last Modified: 12-APR-02

## Article #28620: Problem with IBClientDataSet component in Delphi 6.

### Question:

Why is Delphi freezing when I close a form or application containing IBClientDataSet

### Answer:

There was an issue with IBClientDataset which sometimes caused a problem with the Delphi IDE. This was resolved in an updated version of the IBX components which are available for download from Code Central.

You may want to install Delphi 6 update 2 first.

Last Modified: 16-APR-02

## Article #28662: How to get WebSnap applications to display new lines in a MEMO.

You may have noticed your WebSnap applications display MEMO's without new lines even though there are new lines in the memo.

### Answer:

When WebSnap retrieves the memo field the script variable is replaced with the text from the memo. Since the end result is HTML the browser automatically ignores new lines in the document.

One simple way you can preserve new lines is to wrap the script variable in `<PRE>` `</PRE>` tags.

You can replace the server script with something like this:

```
<PRE><% WriteColText(vDataSetAdapter1_MEMO.DisplayText) %></PRE>
```

Last Modified: 10-MAY-02

## Article #28666: Where is 2AllBooks.pdf?

### Question:

Where is the file 2allbooks.pdf?

### Answer:

This file is simply an index to the Interbase document set. This file and the corresponding documentation are not on the Delphi 6 CD, but they are available for download from our [web site](#). The file 2allbooks.pdf is contained in the [InterBase 6.0 Beta Documentation Set](#).

Last Modified: 10-MAY-02



## Article #28675: Sending an early response in a WebSnap application loses session values.

Sometimes you will want to send out information as it is being processed in your WebSnap Application. One way to do this is to call Response's `SendResponse` method. If you are using session components then you will also need to make a call to the Session's `UpdateResponse` method to preserve your session values.

An example of how to do this is listed below:

```
Session.Values['a'] := 'could be lost... unless';  
Response.Content := 'Response.SendResponse Sent';  
Session.UpdateResponse(Response); // add this line  
Response.SendResponse;           // now send the response
```

Last Modified: 10-MAY-02

## Article #28692: Can't load package dclnet60.bpl

### Question:

Why am I getting the error "Can't load package dclnet60.bpl when running Delphi?"

### Answer:

There is a conflict between Delphi 6.0 or 6.1 and C++ Builder 6, with some files they share. The solution is to install the [Delphi 6 update pack 2](#) which fixes this.

Last Modified: 17-MAY-02

## **Article #28694: Missing database login prompt.**

### **Question**

Why am I not getting a login prompt when I open a TDatabase connection?

### **Answer**

A change between Delphi 5 and Delphi 6 requires that you now add the unit DBLogDlg to the uses clause if you want the default login dialog to appear. This dialog appears when TDatabase.LoginPrompt is set to true.

Last Modified: 17-MAY-02

**Article #28699: Oracle 9i BDE certification.**

**Question**

Does the BDE 5.2 support Oracle 9i?

**Answer**

No. BDE version 5.2 is currently certified for Oracle 8.0.4, 8.1.6, and 8.1.7 but not 9i.

If you need 9i support you can use dbExpress.

For more up to date information check the readme which ships with your version.

Last Modified: 26-NOV-02

## **Article #28701: Problems installing / registering free (trial) edition of Kylix on CD.**

Some free CDs of Kylix, which were included in a trial package of Borland Linux products, are having registration problems when trying to use the option to register by phone. Specifically this happend with CDs made outside the U.S. which do not have the official Kylix label (similar to the box cover as seen on [shop.borland.com](http://shop.borland.com)).

If you are unable to get this copy installed and running the best alternative is to [download the Kylix trial edition from our web page](#).

Last Modified: 28-MAY-02

**Article #28718: Keep getting error: "Maximum number of DBPROCESSES already allocated".**

**PROBLEM:**

**Getting Error "Maximum number of DBPROCESSES already allocated" using Midas/BDE.**

**SOLUTION:**

Start the BDE Administrator and in the configuration go to the driver you are using and increase the MAX DBPROCESSES number.

This property is not available for all native drivers.

Last Modified: 06-JUN-02

## **Article #28734: Setting the Just-in time debugger default in Delphi 6.**

### **Question:**

Why can't I set the Just-in-time debugger default when starting Delphi 6?

### **Answer:**

If Delphi was installed as Administrator and you are running it as a user who does not have administrator rights then you cannot set this option. However, you should not need to set it since it was set by the Administrator who installed Delphi. Simply choose "no" and check the box (so it will not ask again).

Last Modified: 06-JUN-02

**Article #28735: Missing package libpng.so.2 when installing Kylix 2 on Mandrake Linux 8.2.**

**Question:**

Why am I getting an error stating that the package libpng.so.2 is missing when I try to install Kylix 2?

**Answer:**

There is a conflict with libpng version 2 and version 3 so Mandrake linux 8.2 does not ship with version 2, as outlined in [this article](#).

Please note that Kylix 2 is not supported on this version of Mandrake Linux - it is supported on Mandrake 8.0.

See the [installation requirements](#).

Last Modified: 10-JUN-02



**Article #28740: Trial and open editions of Kylix and Delphi do not include all files.**

**Question:**

Why can't I find the file actnres.pas in the trial edition?

**Answer:**

The source code is not included in the trial editions of Kylix and Delphi. In addition the Images directory is not included in the trial and open editions of Delphi. One of the tutorials calls for the use of these files so unfortunately you will not be able to complete the

Note: Interbase is not included with the Kylix trial but it is available in a separate download from our website.

Last Modified: 10-JUN-02

**Article #28746: Internal error 2349 while installing Delphi.**

**Question:**

What does internal error 2349 mean while installing Delphi6?

**Answer:**

This is a Windows Installer (MSI) error.

It means "Copy Resumed With Different Info".

It most commonly occurs because the CD is dirty or scratched.

If cleaning the CD does not help you may need to copy the Delphi install onto your hard drive. You can also try turning off virus protection software while running the install

If you decide to copy the install to the hard drive you only need to copy the Install directory and all the files in the root of the CD including setup.exe and install.txt.

Last Modified: 12-JUN-02

## Article #28773: Error "No invocable class registered..." in Midas/Soap app after installing Delphi 6 update pack 2.

### Question:

Why am I getting the error **No invocable class registered that implements interface SOAPMidas.IAppSOAP** of (soap action/path) "**http://www.borland.com/namespaces/Types-IAppServerSOAP**" after installing Delphi 6 update pack 2?

### Answer:

There is a new property added to TSoapConnection called UseSOAPAdapter. This property should be True if the backend is a C++ service and false if it is a Delphi service. The default in Delphi is true. If you are getting this error you should change this property to false.

### Explanation from Borland R&D:

*Delphi SOAP remotes IAppServer 'as-is'. This interface uses the safecall calling convention that is only available in Delphi. More specifically, safecall is not available in C++. In order to have C++ and Delphi interop we needed to expose SoapDM via a different interface. That new interface is IAppServerSOAP. The latter is essentially IAppServer except that the methods utilize the stdcall instead of safecall convention. This interface is used by C++Builder SOAP DataModules.*

*We struggled with whether Delphi SOAP Modules should be moved to IAppServerSOAP too. There were several reasons for and against. Eventually, we settled on leaving Delphi SOAP DM with IAppServer (for now) - mainly for compatibility reasons.*

*This means that TSoapConnection can talk to either interface. The useSOAPAdapter should be true if the backend is a C++ service and false if it is a Delphi one.*

Last Modified: 17-JUN-02

## Article #28798: How to catch a 'Tab Character' key stroke inside your TEdit control.

**Question:** How can I catch the tab character inside of my Edit controls?

**Answer:** This can easily be done by overriding the forms CMDialogKey procedure. To see how this works drop a Edit on the form and enter in the following code:

```
procedure TMyForm.CMDialogKey(Var Msg: TWMKEY);  
begin  
    if (ActiveControl = Edit1) and (Msg.Charcode = VK_TAB) then  
        begin  
            ShowMessage('Somebody Tabbed?');  
        end;  
    inherited;  
end;
```

Also, make sure to add the following line to your forms definition:

```
procedure CMDialogKey(Var Msg: TWMKey); message CM_DIALOGKEY;
```

Last Modified: 27-JUN-02

**Article #28812: Kylix freezes after clicking the Register Now button.**

**Question:**

Why is Kylix freezing after I click the Register Now button?

**Answer:**

It is possible that Kylix only appears to freeze when in actuality the registration dialog has opened but cannot be seen, and is waiting for input. Check all of the icons in the system tray or try moving the registration window by hitting Alt-Tab to see if you see another window with the registration dialog.

Last Modified: 27-JUN-02

## Article #28841: The {Program Name} is linked to the missing export OLEAUT32.DLL: VarNot

### Problem

After installing Update Pack 2 for Delphi 6, deploying even the most basic application on a machine running Windows 95 will give the following error when trying to run the app:

The {Program Name} is linked to the missing export OLEAUT32.DLL:VarNot This problem will also occur on a machine running Windows 98 without any DCOM updates applied. Windows 98SE doesn't exhibit this behavior.

### Solution

In order to resolve this issue, install the Win95 DCOM 1.2 Service Pack and on a Windows 98 Machine apply the DCOM update. This update can be obtained from <http://msdn.microsoft.com>

Last Modified: 01-JUL-02

## Article #28849: SQL Explorer Form isn't Displaying Controls Properly

### Symptoms:

Occasionally the splitter bar saves a position value when SQL Explorer exits that can't be overwritten in later runs of SQL Explorer. An example of this is sometimes experienced with the Enter SQL Tab. Once the results of a SQL statement are displayed a user could choose to make the result grid larger and totally hide the "Enter SQL" part. Every now and then the splitter won't move and appears to be stuck.

### Solution:

The settings for SQL Explorer are stored in the registry and will be updated when SQL Explorer is closed, so the first step is to close all instances of SQL Explorer. If Delphi is running be sure to close it as well. Next you should start Regedit and browse to the following location in the registry:

```
HKEY_CURRENT_USER\Software\Borland\Database Explorer\ 2.0\DbXForm
```

Here are all of the settings for SQL Explorer. For the above scenario you will see a value for Dbx.QuerySplit of 0. Changing this to any positive value should resolve the issue. If you start SQL Explorer through Delphi then the value may be stored somewhere else in the registry. You will need to change this value as well if you are still experiencing the issue.

Last Modified: 08-JUL-02

## Article #28863: Using TTable generates errors when using Filters

**Synopsis:** A "Capability not supported" error is generated when I activate a table that is using filters.

**Solution:** You will want to double check the case of the fields that you are referencing in you filter string. The field names are case sensitive, so capitalizing one letter that wasn't capitalized during the building of the table is the same as misspelling the field name.

Last Modified: 30-JUL-02



## **Article #28878: Access Violation at address ### in module vcl60.bpl or coreide60.bpl when running a project in Delphi**

### **Problem:**

Getting the error "Access violation at address ### in module vcl60.bpl" when running a project in Delphi, often followed by other AVs.

### **Solution:**

The exact cause for this is not currently known. It seems to happen sometimes with a new installation of Delphi on Windows XP.

Possible solutions:

Reset the user registry settings by running the utility D6RegClean.exe which is located in the Delphi bin directory.

Reinstall Delphi, running the utility above after uninstalling (it is also located on the CD).

It is possible that the problem is linked to setting options in Delphi, such as the desktop preferences, before first running a project for the first time.

It is important to note that the registry settings for each user are set the first time Delphi is run, so be sure to run Delphi once after installing, before installing any 3rd party components.

Last Modified: 16-JUL-02

## Article #28882: IDE Internal Error Solutions

### Question

I receive an "IDE Internal Error" message when I attempt to do practically anything in the Kylix 2 IDE. How can I resolve this problem?

### Answer

Change the driver used by XFree86 to 'fbdev' in the X configuration tool provided by your Linux distribution.

This particular solution resolved the problem on a laptop system using Trident video chipset on Mandrake 8.0 and SUSE 8.0 distributions, but may work for other distributions and chipsets. 'fbdev' is a generic, unaccelerated video driver for framebuffer devices. Use of this driver may work-around problems with the stock video driver.

A more complete solution in this situation might be to update the XF86 drivers or possibly downgrade drivers to a more stable version.

Last Modified: 17-JUL-02

**Article #28888: Network initialization failed. Lock violation with file pdoxusrs.net.**

**Question:**

What is causing the error "Network initialization failure" and "lock violation"?

**Answer:**

There is a known file sharing issue with Novell client version 4.83 for Windows NT/2000. The solution is to install the [fix](#) from the [Novell support web site](#).

Last Modified: 22-JUL-02

**Article #28889: Applying Delphi updates to the evaluation version**

**QUESTION:**

**How do I apply the Delphi Updates to my trial (evaluation) copy of Delphi Enterprise?**

**ANSWER:**

The trial version of Delphi is simply for evaluation, not for serious development. It is not the same build as the full Delphi Enterprise version that you can purchase, which means the updates for Delphi 6 will not work on the trial version.

Last Modified: 25-JUL-02

**Article #28890: Removed FastNet while installing the Indy components.**

**QUESTION:**

**I installed the WinShoes Indy components in Delphi 5 and it removed NetMasters Fastnet components. How can I get the Fastnet components back?**

**ANSWER:**

Start Delphi 5 and select Component | Install Packages. Click on the Add Button and navigate to the Delphi 5 Bin directory. Select dclnmf50.bpl and click Open. Now select OK. You can now select Component | Configure Palette. Select [All] under the Pages ListBox. Click the Default Pages button and you will now have the FastNet tab back in your component palette.

Last Modified: 25-JUL-02

## Article #28912: MySQL queries yeild only 10 columns.

### Question:

My DBExpress connection to MySQL yeilds query results that return only 10 columns. What is the problem?

### Answer:

This is usually caused by a mismatch in driver versions. Check your 'dbxdrivers.ini' file, typically found in the "/Program Files/Common Files/Borland Shared/DBExpress" folder, and make sure the driver for MySQL is "dbexpmysql.dll" and not "dbexpmys.dll". Also make sure the proper MySQL client driver 'libmysql.dll' is accessible in the system path.

Last Modified: 25-JUL-02

## Article #28920: Delphi Generates "Can't Create Delphi32. \$\$\$" Errors when Launching.

### Synopsis:

When Delphi starts it reports errors about creating a `..\Delphi6\bin\Delphi32. $$$` file. Also, once Delphi is finished loading, it appears to be missing various items from the IDE.

### Solution:

This may be a problem with the current user not having appropriate security rights. Even if the user is part of the *Administrators* group it is still possible that the security attributes set on `\Delphi6\bin` directory don't allow your user to modify the directory.

To Resolve this issue you can change the security settings for Delphi's Bin directory. To do this:

- Open Explorer and Browse to `..\Delphi6\Bin` directory
- Right Click on the Bin directory | **Properties**
- Click on **Security Tab**
- Select your user and Allow Full Control to Permissions

If this resolves the issue, then you should examine all the directories that were created during Delphi's installation to verify these directories have Full Control set.

Last Modified: 30-JUL-02

## Article #28927: How to Get the Filename of the Process that is Currently Executing.

**Question:** I'd like to get the filename of the process that is currently executing?

**Solution:** A call to `GetModuleName` will return the name of the file that is currently executing your code. You will need to pass it the correct handle to the module in order for it to correctly return the filename that it is associated with, specifically `hInstance`. Below is some sample code to return the filename:

```
function GetModName: String;  
var  
    fName: String;  
    nsize: cardinal;  
begin  
    nsize := 128;  
    SetLength(fName, nsize);  
    SetLength(fName, GetModuleFileName(hinstance, pchar(fName), nsize));  
    Result := fName;  
end;
```

Last Modified: 30-JUL-02



**Article #28944: Runtime error 230 when running Kylix application outside of the IDE.**

**Question:**

Why am I getting runtime error 230 when running my application outside of the Kylix IDE?

**Answer:**

This error generally means your application cannot find a needed library.

Tunning the command **source \kylix\bin\kylixpath** in Linux should fix this. See the file `deploy.txt` that comes with Kylix.

If this does not resolve your error you may want to chech the Kylix deploy [newsgroup](#) for more information.

Last Modified: 31-JUL-02

## Article #28963: How to Convert a String of Integers into an Array of Byte.

**Question:** How can I create an array of byte values that correspond to an integer string of arbitrary length?

**Suggestion:** The best solution is to dynamically create an array of byte that has length equal to that of the string. Once you have your array you can fill the array with the values from the string, however there is some offset since the ascii representation of the character '1' isn't equivalent to 1. Below is a sample of how one might return an array of byte:

```
interface
uses
    ...
type
    //dynamic array type for Array of Byte
    TByteArr = array of byte;

    ...
implementation

function ArrOfByte(AStr: String): TByteArr;
var
    i: integer;
begin
    SetLength( Result, Length(AStr));
    for i := 0 to Length(AStr) - 1 do
        Result[i] := ord(AStr[i + 1]) - 48;
end;
```

## Article #28964: Converting Double Types from Little Endian to Big Endian Using Variant Records.

**Question:** How can I convert my doubles from Little Endian to Big Endian?

**Solution:** The conversion is very simple. It requires that you swap the 8 underlying bytes of the double such that the first byte goes to the last, the second byte goes to the second of last, and so on. The following Record definition and procedure should convert you doubles from Little Endian to Big Endian and back again.

```
type
  //enumeration used in variant record
  BytePos = (EndVal, ByteVal);

  PEndianCnvRec = ^EndianCnvRec;

  EndianCnvRec = packed record
    case pos: BytePos of
      //The value we are trying to convert
      EndVal: (EndianVal: double);
      //Overlapping bytes of the double
      ByteVal: (Bytes: array[0..SizeOf(Double)-1] of byte);
    end;

  //A gets B's values swapped
procedure SwapBytes( A, B: PEndianCnvRec );
var
  i: integer;
begin
  for i := high(A.Bytes) downto low(A.Bytes) do
    A.Bytes[i] := B.Bytes[High(A.Bytes) - i];
end;
```

Below is an example of how one might use the above procedure. Start a New application and drop a button on the form. Then use the following code for the button click:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a,b,c: EndianCnvRec;
begin
  a.EndianVal := 23344.55555;
```

```
SwapBytes(@b,@a);  
ShowMessage(FloatToStr(b.EndianVal));  
SwapBytes(@c,@b);  
ShowMessage(FloatToStr(c.EndianVal));  
end;
```

Last Modified: 15-AUG-02

## **Article #28965: ISAPI and CGI Applications get Errors Initializing the BDE.**

**Scenario:** When my ISAPI or CGI Applications try to access the BDE they get an error when the BDE tries to initialize. The BDE appears to be installed correctly since regular desktop applications operate without any problems.

**Suggestion:** The user that runs under IIS may not have proper permissions set on the system Temp directory. One suggestions is to change the security settings to Full Control for the USERS user. Full Control must be set on Temp in addition to the directory having read, write permissions.

Last Modified: 15-AUG-02

## Article #28968: My WebSnap Application Executes the Wrong Action, or Skips the Action All Together.

**Synopsis:** Occasionally a WebSnap Application doesn't seem to execute the action associated with an AdapterPageProducer. Usually this occurs when the FormMethod is set to fmPost.

**Suggestion:** If you have a Page Module with an AdapterPageProducer on it, then you may need to alter the HTML that is generated. Specifically you will need to add an attribute to the <Form> tag. The HTML below illustrates the changes that need to be made:

```
<!-- Add the action attribute to the form tag -->
```

```
<form name="AdapterForm1" action=<%=Request.ScriptName%> method="post">
```

Last Modified: 15-AUG-02

## Article #28999: Error "Can't load package dclbdbdesign.so.6.9" when starting Kylix 3 for C++.

### Question:

What is causing the error "can't load package ...dclbdbdesign.so.6.9" "Access violation at address..." when starting Kylix 3 for C++?

### Answer:

There is a problem which seems to exist only with the Professional edition of Kylix 3 when running the C++ version, and only when running on certain versions of Linux, including RedHat 7.3 and SuSE 8.0. It may also occur on RH 7.2 if you have upgraded your kernel.

Please note that this error *does not* occur on any of the Linux versions and distributions Kylix 3 has been certified for (listed on [the Borland web site](#) or on [shop.borland.com](http://shop.borland.com)).

**Possible workarounds:** Some people have reported success by removing the SOAP components, and then restarting Kylix 3. After restarting you may even be able to reinstall the SOAP components.

Last Modified: 02-OCT-02

## **Article #29013: My ActiveX Controls are not Responding to 'Tab' and 'Cursor' Keys.**

**Synopsis:** My Active Forms and other ActiveX Controls respond to Alpha Numeric Keys, however, they are not responding to 'Tab' and 'Cursor' keys.

**Solution:** Your controls will need to handle the CM\_WANTSPECIALKEY message, which can be done by implementing the **CMWantSpecialKey** procedure.

Last Modified: 31-AUG-02



## Article #29027: How to Get the Page Name of a WebModule in a WebSnap Application.

**Question:** I'd like to create a link in one Web Module to reference another Web Module, such that when the user clicks on the link it takes them to that page.

**Solution:** The easiest way to do this is to use *Pages* script variable. If the page has been registered with the application then it should be accessible from the *Pages* variable. The example below illustrates how one would generate a HTML link to a page called YourPageName:

```
<a href="<%Response.Write(Pages.YourPageName.HREF)%>"> Click Me </a>
```

Last Modified: 12-SEP-02

## Article #29033: Kylix 3 freezes when opening or saving a file.

### Question:

Why is Kylix 3 freezing when I try to open or save a file?

### Answer:

The exact cause of this problem has not yet been determined. You should be able to get out of it by using kill -2 to shut down Kylix.

There is a problem which seems to exist only with the Professional edition of Kylix 3 on Redhat 7.3. It may also occur on RH 7.2 if you have upgraded your kernel.

Please note that this error *does not* occur on any of the Linux versions and distributions Kylix 3 has been certified for (listed on [the Borland web site](#) or on [shop.borland.com](http://shop.borland.com)). If you are experiencing this issue the solution is to move to a supported Linux distribution and version - i.e. Redhat 7.2.

**Possible workarounds:** One workaround that is reported to work instead is to download the latest kernel source (2.4.19) from <http://www.kernel.org/pub/linux/kernel/v2.4/> and build it.

Last Modified: 09-SEP-02

**Article #29034: Kylix 3 trial - compiler errors state "message catalog not found".**

**Question:**

Why are compilation errors showing up as "message catalog not found" instead of the text of a real message?

**Answer:**

This is a known issue with the Kylix 3 trial edition.

There is a fix available for this problem. Please see the Kylix 3 downloads page here: [http://www.borland.com/products/downloads/download\\_kylix.html](http://www.borland.com/products/downloads/download_kylix.html). Click the "Updates" link under the "Registered Users" section.

Last Modified: 14-APR-03

## Article #29035: Delphi 7 and Visibroker 3.3 support

### Question:

I have an application that uses Visibroker 3.3 that I wrote with Delphi 5. Can I work with this using Delphi 7?

### Answer:

Visibroker 3.3 support was removed from Delphi 7. Delphi 7 ships with Visibroker 4.5 for C++. In order to use an application that was written with Visibroker 3.3, you will have to migrate the app to Visibroker 4.5.

Last Modified: 31-AUG-02

## Article #29037: Changing the Desktop wallpaper using Delphi

### Question

How do you change the Desktop wallpaper for Windows using Delphi?

### Solution

You will use the SystemParametersInfo function found in the Windows API to set the desktop wallpaper. Drop a button on your form and in the OnClick event for the button place the following code:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    PicPath: String;  
begin  
    PicPath := 'C:\WINNT\Prairie Wind.bmp';  
    SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, pChar(PicPath),  
SPIF_SENDCHANGE)  
end;
```

Last Modified: 12-SEP-02

## Article #29045: Evaluating Corba using the Delphi 7 Trial

### Question

Installing Delphi 7 Trial didn't allow the option of installing Visibroker. Can the Delphi 7 Corba features be evaluated with the Delphi 7 Trial?

### Answer

The Corba demo's do get installed with the Delphi 7 Trial. In order to get them to work, you must install an evaluation version of Visibroker 4.5 for C++. Once you do this, you will be able to try the Corba demo's. The Corba Wizards in the Object Repository will not be installed with the Delphi 7 Trial.

Last Modified: 31-AUG-02

## Article #29048: Corrupted .pdf files in the Corba EJB demo folder

### Question

Attempting to open the .pdf files in the Delphi7\Demos\Corba\Idl2Pas\EJB directory gives the error: "There was an error opening this document. The file is damaged and could not be repaired." How can I obtain the non-corrupted pdf files? The two files that are corrupted are: Getting Started EJB.pdf  
Simplified IDL demo.pdf

### Answer

You need to obtain the working .pdf files from Borland. Create an free Install Support case from Borland's site and request the working .pdf files. The URL to create a free support case is:

[http://support.borland.com/devsupport/create\\_install\\_case.html](http://support.borland.com/devsupport/create_install_case.html)

Last Modified: 31-AUG-02

## Article #29049: Help files for Idl2Pas in Delphi 7 Enterprise

### Question

The help files for Idl2Pas in Delphi 7 Enterprise didn't get installed on the hard drive. With Delphi 6, the help files were installed in the Delphi 6\Doc\Idl2Pas directory. Where is the Help for Idl2Pas in Delphi 7?

### Answer

Due to space constraints on the Delphi 7 installation CD, the help files for Idl2Pas were left out. They can be found on the Companion Tools CD (disk 1) in the borland directory. Open up the borland directory on that CD and you will see the file:

Online+PDF Docs.zip

Copy this file to your hard drive and extract the contents to a location of your choice.

Last Modified: 31-AUG-02



**Article #29050: Mystery errors compiling Delphi project.**

**Question:**

Why am I getting errors on lines where there are no errors or on lines that do not exist?

**Answer:**

This can occur if there are hidden characters in the .pas file.

The simplest way to fix it is to open the file in a text editor such as Wordpad, add a space anywhere to be sure it resaves the file and then close the file. Then try reopening and compiling in Delphi.

Last Modified: 30-AUG-02

## Article #29054: Kylix 3 C++ compiler issue with locales where decimal separator is not a period.

The Decimal separator in C++ source code is locale sensitive. This results in incorrectly built binaries on any locales where the decimal separator is not a period ".".

You should be able to avoid this problem with following setup for bc++:

```
% unsetenv LC_ALL
% setenv LANG de_DE
% setenv LC_NUMERIC C
---
# unset LC_ALL
# export LANG=de_DE
# export LC_NUMERIC=C
```

Last Modified: 31-AUG-02

## Article #29059: Error uninstalling Delphi 6: Error 1926. Could not set file security for file Filename.msi

### Problem

When attempting to uninstall Delphi 6, the following error is raised: Error 1926. Could not set file security for file \somepath\Filename.msi and the uninstall fails. (somepath represents the path of the file listed in the error message and Filename.msi represents the file name also listed in the error message.

### Resolution

This problem is due to a lack of privileges on the specified file. Make sure you are logged in as administrator and have sufficient privileges to modify the security permissions on the file in question. Once you do this, the problem will be resolved.

Last Modified: 04-SEP-02

## Article #29064: A Better Way to End the Endless Registration Loop when Trying to Register Delphi 6

**Synopsis:** Delphi is prompting to register the product. After registering the product the an error message is displayed to register the product again. Further attempts to register the product are unsuccessful.

**Solution:** There is an alternative method to resolving the registration loop, which doesn't require D6RegClean and a repair of Delphi. Instead you can preserve your current installation by removing three registry keys and registering once more.

The registry keys can be removed using the RegEdit tool, Start | Run | RegEdit. The Keys to remove are listed below:

```
HKEY_CURRENT_USER\Software\Borland\Delphi\ 6.0\LM  
HKEY_CURRENT_USER\Software\Borland\Delphi\ 6.0\LMLIC  
HKEY_CURRENT_USER\Software\Borland\Delphi\ 6.0\LMKEY
```

Last Modified: 21-APR-03

## Article #29078: Adding QuickReports to the Delphi 7 Component Palette

### Question

How do I install the QuickReports components on the Delphi Component Palette?

### Answer

You need to install the QuickReports package (dclqrt70.bpl).

In Delphi, select **Component | Install Packages...**

Click **Add** and select the file *dclqrt70.bpl* (found in the Delphi7\Bin directory). Click **Open** and then click **OK**.

Last Modified: 12-SEP-02

## Article #29082: Where is the Rave PDF File Documentation for Delphi 7?

### Question

Where is the Rave PDF File Documentation for Delphi 7?

### Answer

These files are located on the Companion Tools CD Disk 1 and are bundled in a zip file.

**\borland\Online+PDF Docs.zip**

Unzip that on your hard drive and you will see a Rave directory with the .pdf files.

Last Modified: 20-SEP-02

## Article #29083: Where is TCorbaConnection in Delphi 7?

### Question

Where is TCorbaConnection in Delphi 7?

### Answer

TCorbaConnection was removed from Delphi 7 since it only works with VisiBroker 3.3. Support for Visibroker 3.3 has been removed from Delphi 7. Delphi 7 ships with and supports VisiBroker 4.5. Take a look at the Midas XML demo that ships with Delphi. This demo shows you how to use Corba ANY, eliminating the need to use TCorbaConnection.

Last Modified: 12-SEP-02

## Article #29091: Problem installing Delphi: Internal Error 2735.

### Problem

When installing Delphi, Internal Error 2735 is raised before the installation completes.

### Solution

Copy the Install directory and all the files in the root directory on the CD to

C:\

It is important to make sure you copy the files to the root drive. Reboot in Safe Mode and install. Reboot again to finish.

If that fails to resolve the problem, check the file attributes for the **autoexec.bat** file found in the root of your C:\ Drive. If it is set to Read Only, this can be causing the problem so simply set the file attributes to "**Archive**" and you should be able to work around this issue.

Last Modified: 18-SEP-02



**Article #29105: Delphi 7 registration wizard is missing text.**

Some people are experiencing problems with the Delphi 7 registration wizard. All of the text is missing from the dialogs and controls so it is not possible to tell which options you are choosing.

A fix for this is now available on the Delphi Developer Support page:

<http://info.borland.com/devsupport/delphi>

Last Modified: 28-OCT-02

## Article #29142: E2257 in QRvDefine.hpp when compiling TRvProject

### Question:

I am using the C++ IDE of Kylix 3 and the bundled Rave Reports 4.5 Community Edition. When I try to compile a TRvProject component, I get the following error:

```
-----  
[C++ Error] QRvDefine.hpp(147): E2257 , expected  
-----
```

The line 147 referenced in QRvDefine.hpp reads:

```
-----  
typedef TRaveUnits TRaveUnits;  
-----
```

How can this be resolved?

### Answer:

Line 147 in QRvDefine.hpp is incorrect. This issue can be resolved by correcting it to:

```
-----  
typedef double TRaveUnits;  
-----
```

Be sure to save the changes to QRvDefine.hpp, and then try compiling again.

## Article #29143: Cannot register Kylix 3 due to endless registration loop

### Question:

I have installed Kylix 3 and am having problems registering it. The registration wizard appears repeatedly and will not accept my registration information as valid, even though it is correct.

How can I register Kylix 3?

### Answer:

This could be due to a corrupt registration file. The solution is to delete the existing registration files for Kylix 3, and then re-register.

First, navigate to your `.borland` directory, which will be found in the home directory of the user under which Kylix 3 was installed. If you installed as root, this will be `/root/.borland`. If you installed as an user, this will be `/home/username/.borland`.

Inside the `.borland` directory, delete these two files: `registry.dat`, `registry.slm`. Doing so should remove the existing registration information.

Then, register Kylix 3 again, and it should work properly.

Last Modified: 30-SEP-02

**Article #29156: Kylix 3 freezes on start up.**

**Question:**

Why is Kylix freezing during startup?

**Answer:**

This is most likely a video problem. The settings on your machine may need to be adjusted. Try changing the screen size and/or color resolution. 800 X 600 and 24 bit seem to work most often. If they do not work for you then continue to experiment. If this does not work you may need to update your video driver.

If the freeze is occurring after clicking the Register Now button then you should see [this document](#) as well.

Last Modified: 30-SEP-02

## **Article #29178: Kylix 3 does not install properly on Redhat 8.0**

Kylix 3 is not installing properly on Redhat 8.0. It seems some of the files are copied to the root directory instead of the kylix3 directory, and some are not copied at all. Currently we do not workaroud for this. When we do it will be posted here.

Please note that Kylix 3 is only certified for use on RedHat 7.2.

Last Modified: 10-OCT-02

**Article #29179: Error: "d7reg.exe has encountered a problem and needs to close."**

**Question:**

Why am I getting the error "d7reg.exe has encountered a problem and needs to close. We are sorry for the inconvenience."?

**Answer:**

You may also get the error

```
"exception EReadError in module D7Reg.exe.  
Error reading image1.picture.data: Access violation at  
address 005715D6 in module qtintf70.dll."
```

This is a video driver problem. It is more likely to occur on laptops. The solution is to upgrade your video driver. You could also try different settings for your screen resolution and/or density.

Last Modified: 11-OCT-02

## Article #29200: During installation, Serial Number and Authorization are invalid

### Question

*When installing, I reach the point where serial number and authorization key are requested. I enter them, but the installer says they are invalid. What's wrong?*

### Answer

First of all, make sure you are entering the correct serial number. Since most Borland products packages actually contain multiple software installations, it is easy to accidentally use the serial number for one product on another. Before you start typing a serial number, check the bottom right corner of the printed serial number tag, and find the product description the serial number is made for. Make sure this matches the software you are running or installing.

Second, make sure you are typing the digits and numbers in correctly. Use the correct capitalization and punctuation when entering the serials. Most Borland software provides you with multiple edit boxes for each character group of the serial number, so you may omit the hyphen (-). Other products, such as Kylix, do require you to enter the hyphens between character groups.

If you are running Windows 98, you may need an update to your system's common windows controls (comctrl32.dll) which is most easily acquired by installing Internet Explorer 5 or later. On these systems, the edit boxes themselves have a problem taking input from the keyboard and despite having entered the proper information, will still not recognize the serial correctly.

Last Modified: 22-APR-03

**Article #29212: When attempting to compile, the following error comes up: 'System.pas' not found**

## **Problem**

**After installing Delphi 7, I select File | New | Application and try to compile. It fails with the error: 'System.pas' not found**

## **Resolution**

This can happen if your environment path settings inside the IDE isn't setup properly. Inside the Delphi IDE, select Tools | Environment Options and select the Library tab. In the **Library Path**, make sure you have the following: *(keep in mind that your path may vary depending on where you installed Delphi)*

c:\Program Files\Borland\Delphi7\**Lib**  
c:\Program Files\Borland\Delphi7\**Bin**  
c:\Program Files\Borland\Delphi7\**Imports**  
c:\Program Files\Borland\Delphi7\**Projects\Bpl**  
c:\Program Files\Borland\Delphi7\**Rave5\Lib**

**BPL output** directory should be:

c:\Program Files\Borland\Delphi7\Projects\**Bpl**

**DCP output** directory should be:

c:\Program Files\Borland\Delphi7\Projects\**Bpl**

The **Browsing Path** should be: *(this will vary depending on whether you have Delphi 7 Personal, Professional, or Enterprise)*

c:\Program Files\Borland\Delphi7\source\vc1;  
c:\Program Files\Borland\Delphi7\Source\Rtl;  
c:\Program Files\Borland\Delphi7\source\rtl\Corba45;  
c:\Program Files\Borland\Delphi7\source\rtl\Sys;  
c:\Program Files\Borland\Delphi7\source\rtl\Win;  
c:\Program Files\Borland\Delphi7\source\rtl\common;  
c:\Program Files\Borland\Delphi7\source\Internet;  
c:\Program Files\Borland\Delphi7\source\clx;  
c:\Program Files\Borland\Delphi7\source\websnap;  
c:\Program Files\Borland\Delphi7\Source\WebMidas;  
c:\Program Files\Borland\Delphi7\Source\Indy;



c:\Program Files\Borland\Delphi7\Source\IntraWeb;  
c:\Program Files\Borland\Delphi7\Source\Samples;  
c:\Program Files\Borland\Delphi7\Source\Soap;  
c:\Program Files\Borland\Delphi7\Source\ToolsAPI;  
c:\Program Files\Borland\Delphi7\Source\Xml

Last Modified: 30-OCT-02

**Article #29226: Cannot create new connection because in manual or distributed transaction mode**

**Problem**

**When using the MSSQL dbExpress driver with a ClientDataSet and a Provider, keep getting the following error when calling ApplyUpdate:  
*Cannot create new connection because in manual or distributed transaction mode***

**Suggested Work Around:**

Set "poFetchDetailsOnDemand" (for the Provider) to False. This is a known issue when it is set to True.

Last Modified: 30-OCT-02

**Article #29229: How can I generate a log file of the Delphi installation process?**

**Question:**

**How can I generate a log file of the Delphi installation process?**

**Answer:**

Open up a command window and type the following:

*(keep in mind that the path settings may be different depending on your machine's configuration)*

```
c:\winnt\system32\msiexec /L*V c:\install.log /i "e:\install\Borland Delphi 6.msi"
```

**Last Modified: 30-OCT-02**

## **Article #29230: How to Cut and Paste WebSnap Script from the Web Page Designer into the Code Editor.**

**Synopsis:** Using the Ctrl + C shortcut to copy WebSnap script from the web page designer doesn't work. Instead of copying the generated script, the shortcut appears to copy contents from the DFM.

**Solution:** In order to copy the script from the web page designer use the "Ctrl + Insert" keystroke. You should still be able to use the "Ctrl + V" keystroke to paste the contents back into the code editor.

Last Modified: 30-OCT-02

## Article #29231: Converting a Delphi VisiBroker 3.3 app to Delphi 7 VisiBroker 4.5

### Question:

**How can I migrate my Delphi VisiBroker 3.3 code to use Delphi 7 and VisiBroker 4.5?**

### Answer:

In Delphi 7 you can use the command line tool: `tlibimp` to convert your VisiBroker 3.3 tlb files to Corba IDL. Type the following at the command line:

*(Keep in mind that the path depends on where your tlb file is located and substitute `MyTLBFile.tlb` with the name of the type library file you wish to convert)*

```
C:\Program Files\Borland\Delphi7\Projects\Corba\>tlibimp -Ic+ MyTLBFile.tlb
```

The files that are generated are: `MyTLBFile_TLB.dcr`, `MyTLBFile_TLB.pas` and `MyTLBFile_TLB.idl`. You can now use the Client/Server wizards and pass in the newly created IDL file to generate the stubs and skeletons. Now all that must be done is to transfer the code from the VisiBroker 3.3. application to the new VisiBroker 4.5 application in Delphi 7.

Last Modified: 30-OCT-02

## Article #29232: My Kylix application won't run outside the IDE.

### Question:

Why is it when I try to run my Kylix App outside the IDE I get this message:  
"error loading shared libraries: libqtintf.so: cannot open shared file: No such file or directory"?

### Answer:

This message and similar ones occur when ../kylix/bin is not included in your path when trying to use CLX components. Running /usr/kylix/bin/kylixpath is a short fix, but you can also add the line to your .bashrc file to set the paths whenever you start a shell. Be sure to change the appropriate .bashrc (ie. for user jbrown /home/jbrown/.bashrc).

Example .bashrc:

```
-----  
#.bashrc  
  
..  
source /usr/kylix/bin/kylixpath  
..  
  
-----
```

Last Modified: 30-OCT-02

**Article #29239: How to enable .NET Compiler Warnings when using dcc32 compiler**

**Question:**

**How can I enable .NET Compiler Warnings when using the dcc32 compiler?**

**Answer:**

The Delphi 7 dcc32 compiler now supports three additional compiler warnings: Unsafe\_Type, Unsafe\_Code, and Unsafe\_Cast. These warnings are disabled by default, but can be enabled with source code compiler directives {\$WARN UNSAFE\_CODE ON}, compiler command line switches (dcc32 -W +UNSAFE\_CODE), and in Project|Options (Select Compiler Messages and select Unsafe Type, Unsafe Code and Unsafe TypeCast).

***NOTE: There is no space before or after the "+" in the command line switch***

Last Modified: 05-DEC-02

## Article #29240: "CoInitialize has not been called" when using TXMLDocument

### Question:

I have an application that uses the XML components. This application worked just fine until I decided to make it multi-threaded. Now I am getting a "CoInitialize has not been called" error when I try and use an XML component.

### Answer:

Microsoft's XML parser, which the XML components use by default, is a COM-based parser, which means that COM needs to be initialized before you can use them. Delphi calls the CoInitialize function for you when an application starts up, but that function only initializes COM on the thread it was called from. You should put a call to CoInitialize at the beginning of your TThread descendant's Execute method and a matching CoUnInitialize at the end.

### Question:

I have a TXMLDocument component on my form, and I have no problems using it. However, I tried creating a TXMLDocument at runtime, and with the same exact code that worked earlier, I am getting access violations! What is causing this?

Last Modified: 31-OCT-02



## Article #29241: Strange access violations when using TXMLDocument at runtime

### Question:

I have a TXMLDocument component on my form, and I have no problems using it. However, I tried creating a TXMLDocument at runtime, and with the same exact code that worked earlier, I am getting access violations! What is causing this?

### Answer:

Most likely, you are creating TXMLDocument like this:

```
var
  Doc: TXMLDocument;
begin
  Doc := TXMLDocument.Create(nil);
end;
```

The important thing to notice is that you are passing in **nil** as the owner. What happens then is that TXMLDocument takes on a different behavior; it acts as a reference counted object that automatically frees itself when those references reach zero. This means that the next time you do something to Doc that causes a decrease in the reference count, you run the risk of destroying the object and invalidating your reference. The problem is that so much happens to TXMLDocument behind the scenes, inside of the VCL, that it is hard to say when exactly the reference count is getting modified. The easiest way to get around this is to use an interface pointer instead of a TXMLDocument reference, like so:

```
var
  Doc: IXMLDocument;
begin
  Doc := TXMLDocument.Create(nil);
end;
```

This lets you have more complete control over the reference, allowing you to specify when exactly it gets decremented by setting the reference to **nil**.

Last Modified: 31-OCT-02

## Article #29304: Limit reached for installs using Kylix serial number

### Question:

I am attempting to register Kylix, but cannot and get the following error:

-----  
Error registering software. The limit has been reached for installs using this serial number, please contact customer service to increase the allowed install count.  
-----

How do I get around this and register Kylix?

### Answer:

Borland has a registration server that keeps count of how many times a product has been registered. The default count is five registrations. This error indicates that your serial number has already been registered five times, and the registration currently being attempted exceeds the count.

You can get this count increased by calling Borland's registration hotline at (888) 588-2230. Explain to the representative who answers the phone what product you have, as well as the full error message, and he or she (after asking for your relevant information) will be able to increase the count to allow for additional registrations.

Last Modified: 05-DEC-02

## Article #29306: Kylix 3 encounters declaration syntax errors in TIME.H

### Question:

I am using Kylix 3, and get declaration syntax errors in TIME.H when attempting to compile any project. How can I solve this problem?

### Answer:

The TIME.H declaration syntax errors can be resolved by going into the Project Options and moving the reference to /usr/include up in the Include path. Preferably, /usr/include should be moved to the very first position in the ordered Include path.

The exact cause of this problem is not yet known, but it could be related to Kylix 3 finding a different version of TIME.H elsewhere on the system. For reference, Kylix 3 contains four instances of TIME.H, in the following locations:

- /usr/include/linux
- /usr/include/bits
- /usr/include/sys
- /usr/include

Last Modified: 05-DEC-02

**Article #29313: ModelMaker demo has expired. Please check the ModelMakers tools website for more info.**

**Question:**

**I keep getting the following error message:**

***ModelMaker demo has expired. Please check the ModelMakers tools website for more info.***

**even though I installed from the Delphi 7 CD. How can I install it so it isn't a Demo version?**

**Answer:**

This occurs when you install and try to use Modelmaker without installing and registering Delphi 7. You should not need to uninstall Modelmaker, just install and register Delphi 7 and the problem should go away.

Last Modified: 05-DEC-02

## Article #29314: Upgrading from Delphi 7 Enterprise to Delphi 7 Architect

### Question:

**Is it necessary to uninstall Delphi 7 Enterprise in order to upgrade to the Architect version?**

### Answer:

In order to upgrade from Delphi 7 Enterprise to Delphi 7 Architect version, simply get the BoldSoft CD from the box and install that to your Delphi 7 Enterprise. Doing it this way, when you go Help | About, it will still show Delphi 7 Enterprise, but you will have all the BoldSoft components installed. If you want it to read Delphi 7 Architect simply uninstall the Enterprise version and install Architect.

Last Modified: 05-DEC-02

## Article #29385: Kylix 3 C++ IDE hangs at the splash screen under Mandrake 9.0

### Question:

I have installed Kylix 3 under Mandrake 9.0. The Delphi IDE works fine, but the C++ IDE does not load past the splash screen. How can I get around this problem?

### Answer:

Kylix 3 is not officially supported under Mandrake 9.0, and at this point, it is unclear what the precise cause of this problem is. However, it does seem to be specific to Mandrake 9.0, and appears related to the SOAP and dbExpress packages. One workaround is to remove the references to those packages, as follows.

-----  
#1.

Navigate to the `.borland` directory, which can be found in the home directory of the user under which Kylix 3 was installed. If you installed as root, this will be `/root/.borland`. If you installed as an user, this will be `/home/username/.borland`.

#2.

Inside the `.borland` directory, open up the `bc69rc` file in a text editor.

#3.

Scroll to the [Known Packages] section and look for references to the SOAP and dbExpress packages. These should be called `dclsoap.so.6.9` and `dclbdbx.so.6.9`, respectively.

#4.

Remove the two aforementioned package references and save the changes to `bc69rc`.  
-----

After having done the above, the C++ IDE should work. Of course, the SOAP and dbExpress packages will be unavailable. If you require them, then they can be manually added by selecting Component | Install Packages... However, be sure to uninstall them again before exiting the C++ IDE, otherwise, the problem will reoccur the next time you attempt to start it.

## Article #29390: Kylix 3 registration fails and indicates the Serial Number is for an upgrade version only

### Question:

I have installed Kylix 3 and am attempting to register it via the "Register by Phone or Web Page" option. I have submitted my Serial Number and Registration Key, and received a reg###.txt file in return. However, after placing the reg###.txt file in my user's home directory as instructed, I get the following error in the registration wizard:

-----  
The serial number is an upgrade version only of the product, and requires a prior version of the product to be registered. That registration entry was not found, registration failed.  
-----

How can I register Kylix 3?

### Answer:

This error message is a relatively general one, and does not necessarily actually indicate a problem with the Serial Number or product version. It primarily indicates that there is a problem with the reg###.txt (## is a number that depends on what edition of Kylix you have) file.

There is an outside chance that the problem is due to a corrupt registration file. I strongly recommend first removing the existing registration information according to the steps outlined in [this](#) document. If the steps in that document do not help, then you can also delete (or rename if you want to play it safe) the entire .borland directory, thereby causing Kylix 3 to regenerate all the information within it.

If deleting registry.dat, registry.slm, and .borland do not make any difference, then chances are the problem is not due to corrupt registration files, but rather a bad reg###.txt file.

A common problem is unwanted text in the reg###.txt file. Since reg###.txt is sent to you as an e-mail attachment, this can happen if you save it via a HTML-based e-mail client. Such clients sometimes insert bits of HTML into reg###.txt upon saving, thereby rendering it invalid. Open up reg###.txt in a text editor, and see if it contains any extra text - you only want the exact same text contained in the "Borland Product Registration" e-mail. If there is any text other than that, then it will be a problem. To avoid this problem, it is recommended that you save reg###.txt via a different e-mail client (perhaps a non HTML-based one that will not insert HTML), or manually copy and paste the contents of the "Borland Product Registration" e-mail into a text file named reg###.txt (rather than directly save it through the e-mail client). This will ensure that it contains no extra text and is intact.

You should be able to successfully register Kylix 3 once your registration information and reg##.txt file are correct.

Last Modified: 26-MAR-03



**Article #29395: Whom should we contact to renew our support or maintenance contract with Borland?**

**Question:**

Our support contract and/or maintenance agreement with Borland is about to expire, but we would like to renew it.

What is the best way to accomplish this?

**Answer:**

The best way to renew your contract is to call our Sales department at 800-632-2864 and ask for "Recurring Revenue."

Last Modified: 26-DEC-02

**Article #29396: How to obtain a Serial Number and Authorization key for Delphi 7 Personal Promotion**

**Problem:**

**The activation file for my Delphi 7 Personal doesn't work. What can I do?**

**Answer:**

Navigate to [http://www.borland.com/products/downloads/download\\_delphi.html](http://www.borland.com/products/downloads/download_delphi.html). In the Keys Only section click the Personal link for Version 7, fill out the survey, follow the on screen instructions, and a Serial Number and Authorization Key will be e-mailed to you.

Last Modified: 26-MAR-03

**Article #29535: The procedure entry point RtlUnhandledExceptionFilter could not be located in the dynamic link library ntdll.dll**

**Problem:**

**When starting Delphi you receive this error: The procedure entry point RtlUnhandledExceptionFilter could not be located in the dynamic link library ntdll.dll**

**Answer:**

Search for UXTHEME.dll and rename it to UXTHEME.old.

Last Modified: 26-MAR-03

## **Article #29540: Adding InterBase Server Manager to the Tools Menu of Delphi or C++ Builder**

To run InterBase Server Manager from Tools menu in Delphi or C++ Builder, Select Tools | Configure Tools, Click Add and enter the following:

Title: InterBase Server Manager

Program: rundll32

Working dir: c:\program files\borland\interbase\bin (this directory may vary depending on where you installed InterBase)

Parameters: shell32.dll,Control\_RunDLL ibmgr.cpl

Last Modified: 26-MAR-03

## Article #29541: Converting a Delphi VisiBroker 3.3 app to Delphi 7 VisiBroker 4.5

### Question:

**How can I migrate my Delphi VisiBroker 3.3 code to use Delphi 7 and VisiBroker 4.5?**

### Answer:

In Delphi 7 you can use the command line tool: `tlibimp` to convert your VisiBroker 3.3 tlb files to Corba IDL. Type the following at the command line:

*(Keep in mind that the path depends on where your tlb file is located and substitute MyTLBFile.tlb with the name of the type library file you wish to convert)*

**C:\Program Files\Borland\Delphi7\Projects\Corba\>tlibimp -Ic+ MyTLBFile.tlb**

The files that are generated are: `MyTLBFile_TLB.dcr`, `MyTLBFile_TLB.pas` and `MyTLBFile_TLB.idl`. You can now use the Client/Server wizards and pass in the newly created IDL file to generate the stubs and skeletons. Now all that must be done is to transfer the code from the VisiBroker 3.3. application to the new VisiBroker 4.5 application in Delphi 7.

Last Modified: 26-MAR-03

## Article #29558: Accessing InterBase via dbExpress with Kylix produces error: Unable to load libgds.so

If you install the dbExpress InterBase client driver, you will need to have libcrypt.so installed. Some Linux distributions omit this library or do not include it in a base install. If your distribution does not include libcrypt.so, contact the package maintainer, or search online Linux resources, such as <http://rpmfind.net/>.

Some Linux distributions provide all the libraries required to run Kylix, but do not use the naming conventions that Kylix expects. The most common problem is library names with embedded version information, while Kylix expects version-independent names. If Kylix software fails to run because of missing shared libraries, check for similarly named libraries in /lib and /usr/lib. You can then create a symbolic link to help Kylix find the library. For example, if you are missing libcrypt.so but find /lib/libcrypt.so.1, you would enter the following shell commands as root:

```
cd /lib  
ln -s libcrypt.so.1 libcrypt.so
```

Last Modified: 17-JAN-03

**Article #29562: Can't find ibx.hlp error**

**Problem:**

**After loading the help for an IBX component, click Properties and this error occurs: Can't find ibx.hlp**

**Answer:**

Rename d7ibx.hlp to ibx.hlp

Last Modified: 26-MAR-03

## Article #29573: Unable to register Delphi 7 Trial

### Problem:

**The Delphi 7 Architect Trial is installed but it won't register properly or it won't import the activation file.**

### Answer:

Here some things to check:

- 1) That the attachment is saved as a .txt file rather than some other format
- 2) That the file is saved in the Home Directory specified in the e-mail that was sent
- 3) That the file is not modified in any way. (i.e. don't delete the text of the e-mail contained in the text file)
- 4) That you don't have other reg\*.txt files present in the directory as these might interfere
- 5) That you don't have any TSR's running in the background
- 6) Additionally, you should not have a full version of Delphi 7 and the Delphi 7 Architect Trial installed at the same time
- 7) If the other 6 don't solve the problem, check if there is a .borland folder in your Home Directory. If there is, delete it and try to run the Trial again. Note: Deleting this directory will also delete registration information for JBuilder 6 and up.



**Article #29575: Checking to see if the latest RTL Update has been applied to Delphi 6**

**Question:**

**How can I verify that I have applied the latest RTL update for Delphi 6?**

**Answer:**

Open up an Explorer Window and navigate to the Windows System32 directory. Look for the file titled **rtl60.bpl**. The Date/Time stamp should read 12/5/2002 6:03 AM

Last Modified: 26-MAR-03

## Article #29586: Getting error 1629 when attempting to install InstallShield Express

### Problem:

Attempting to install InstallShield Express that comes with Delphi 7 Enterprise, an error message comes up:  
**"1629: Invalid Command Line"** and the installation aborts.

### Solution:

From the Operating System, select My Computer | Properties | Advanced, and select Environment Variables. Under User Variables, check to see what the variables 'Temp' and 'Tmp' are set to. Sometimes the installer has problems if there is a space in the path that Temp and/or Tmp point to. Change it to something without a space in it, like C:\Temp, and it should work.

Last Modified: 26-MAR-03

**Article #29587: Using a DSO on Apache 2.0.43, created with Kylix 3**

**Problem:**

**After compiling and installing Apache 2.0.39 with DSO support, deploying an .so file built with Kylix 3 doesn't work.**

**Solution:**

You need to change `MODULE_MAGIC_NUMBER_MAJOR` in `HTTPD.pas` file to the following:  
**`MODULE_MAGIC_NUMBER_MAJOR = 20020903;`**

**Last Modified: 27-MAR-03**

**Article #29588: There is not enough space on c:\ to extract this package**

**Problem:**

**When trying to install Delphi to a drive other than C: you get the error message: "There is not enough space on c: to extract this package".**

**Answer:**

Go into your System Control Panel and change the TMP system variable to point to a directory on a different drive which has enough space to install Delphi.

Last Modified: 27-MAR-03

## Article #29651: Following the directions in the websnap readme for compiling Apache for DSO support doesn't work

### Question:

Following the directions in the websnap readme for compiling Apache for DSO support didn't work. What went wrong?

### Answer:

The readme.txt file, found in the *kylix3\examples\delphi\websnap* directory has the following section:  
----- Apache DSO's ----- Creating an Apache DSO requires that the source from Apache be recompiled with a shared core. Get the source from [www.apache.org](http://www.apache.org) and configure it with:

```
./configure \  
"--with-layout=Apache" \  
"--enable-rule=SHARED_CORE" \  
"--enable-module=most" \  
"--enable-shared=max" \  
"--prefix=/usr/local/apache"
```

In order to get Apache compiled with DSO support, change the word **most** in the line:

```
"--enable-module=most" \  
to so. The entire section should look like:
```

```
./configure \  
"--with-layout=Apache" \  
"--enable-rule=SHARED_CORE" \  
"--enable-module=so" \  
"--enable-shared=max" \  
"--prefix=/usr/local/apache"
```

## Article #29707: Exception encountered when installing the Rave BE 5.08 update

### Problem:

### When installing the Rave BE 5.08 update you get this error message:

The exception unknown software exception (0x0eedfade) occured in the application at location 0x77eab2f0.

### Answer:

This usually happens when you attempt to install the update without starting Delphi for the first time. In general, it is a good idea to start Delphi after install one time before applying any updates or installing any third party components.

Last Modified: 27-MAR-03

**Article #29754: Procedure entry point cannot be found in Rave50VCLBE70.bpl**

**Problem:**

**When starting Delphi 7, this error occurs: "The procedure entry point @Rpcanvas@initialization\$qqrv could not be located in the dynamic link library Rave50VCLBE70.bpl"**

**Answer:**

This error occurs if you install the Rave BE 5.08 update, subsequently uninstall Delphi 7, reinstall Delphi 7, and then start Delphi without applying the update. You can fix this by applying the update. If you do not want to install the Rave BE update, you can solve this by uninstalling Delphi, searching for and deleting Rave50VCLBE70.bpl and Rave50CLXBE70.bpl, and then reinstalling Delphi.

Last Modified: 27-MAR-03

**Article #29755: Why can't I use SaveChartToFile and LoadChartFromFile with my TChart component?**

**Problem:**

**How can I use SaveChartToFile and LoadChartFromFile with my TChart component?**

**Answer:**

SaveChartToFile and LoadChartFromFile require teechart.dcu which does not ship with Delphi 7. To get this you will need to purchase TeeChart Professional from <http://www.steema.com>

Last Modified: 27-MAR-03



**Article #29766: Delphi 7 and FastNet Components**

**QUESTION:**

**Where are the FastNet Components in Delphi 7?**

**ANSWER:**

The Fastnet components are no longer bundled with Delphi. It appears Netmasters does have a version compatible with Delphi 7 which can be purchased at: <http://www.netmastersllc.com/>

Last Modified: 27-MAR-03

## Article #29768: TServerSocket and TClientSocket are Missing from Delphi 7.

**Question:** The TClientSocket and TServerSocket components seem to be missing from my installation of Delphi 7?

**Solution:** You will need to add the dclsockets package to the IDE. To do this go to Component | Install Packages | Add (/bin/dclsockets70.bpl).

Last Modified: 11-FEB-03

**Article #29789: Problems running WebSnapSurvey IntraWeb demo with Delphi 7.**

**Problem:**

**When trying to run WebSnapSurvey INTRAWEB demo, the following error is raised:**

***Web page not found iw/js/IWcommon.js\_5.0.43***

**Solution:**

This was a problem introduced late in the cycle with the new Web Application Development Environment (WAD). This is addressed with the latest version of IntraWeb. You can download the free update at <http://www.atozedsoftware.com>

Last Modified: 27-MAR-03

## **Article #29790: Migrating from Delphi 4: What happened to the routines FMTBCDToCurr and CurrToFMTBCD?**

### **Problem:**

**I have a project created in Delphi 4 that uses FMTBCDToCurr and CurrToFMTBCD routines. These routines don't seem to be in the newer versions of Delphi**

### **Explanation:**

The routines FMTBCDToCurr and CurrToFMTBCD have been replaced by the new BCDToCurr and CurrToBCD routines.

Last Modified: 27-MAR-03

**Article #29802: Having trouble changing the path that Delphi 6 is installed to**

**Problem:**

**I installed Delphi 6 then decided to uninstall it. Now when I install it again, I am unable to choose a different path to install Delphi to. How do I change this?**

**Answer:**

There is a setting in HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\installer\UserData\[user specific number]\Components\[some GUID] that this path is pulled from. Delete or change this setting to the directory that you want to install Delphi to.

Last Modified: 27-MAR-03

## Article #29804: Installing Kylix 3 Open Edition on Mandrake 9

### Question:

**How can I get Kylix 3 Open Edition installed and running on Mandrake 9?**

### Answer:

Open a Super User Terminal, select "Sessions/New Root Midnight Commander" and in "/", create a directory "temp"

Download or copy the file kylix3\_open.tar.gz into the "temp" directory you just created.

Check that all necessary programs are installed on your system:

Open kmenu/configuration/packaging/Remove Software or kpackage

Check for installation of the following:

kernel => 2.2 (mdk9.0 uses 2.4)

libgtk => 1.2 (mdk 9.0 uses 1.2-1.2.10-29)

libjpeg => 6.2 (mdk 9.0 uses 6.-66-25)

XIIR6 (XFree86) (mdk uses 4.2.1-3)

XFree86-dev (mdk9.0 uses 4.2.1-3)

glibc-dev (mdk9.0 uses 2.2.5-16)

Once you have verified that you have all of the necessary programs do the following:

In a Super User Terminal do:

```
cd /temp
```

```
tar xzf kylix3_open.tar.gz
```

In user terminal (NOT as root):

```
cd /temp/kylix3_open
```

```
./setup.sh
```

In Gui select "I Agree"

Now select "install"

Logout and log back in as same user to get the Borland Kylix entry in the KDE menu

In Borland Kylix 3 menu, select "register now"

In the Gui select "next"

In the Gui select "Finish"

The Gui disappears and the registration process has completed.

The Kylix Delphi and Kylix C++ IDE's can now be run.

Last Modified: 27-MAR-03

**Article #29805: Class not registered error in Delphi 7 when dropping any component installed with dclisp70.bpl onto a form**

**Problem:**

**After installing the dclisp70.bpl package into the Delphi 7 IDE, I try to drop any component added by that package onto a form and I get a class not registered error message. Why does this occur?**

**Answer:**

The Internet Solutions Pack contains NetManage components and is included for backward compatibility with Delphi 2. The ocxs' that are required to use these components haven't shipped with Delphi since Delphi 3. We recommend that you use Indy instead.

Last Modified: 12-MAY-03



## Article #29806: Class TReport not found

### Problem:

**I am migrating an application that was developed with Delphi 2. This application uses ReportSmith and used the TReport component. I keep getting the error Class TReport not found even though I copied the Report.dcu and RSCONSTS.dcu files into my Delphi Lib directory.**

### Explanation:

The files Report.dcu and RSCONSTS.dcu are shipped with the latest version of Delphi strictly for backward compatibility. What these files allow you to do is use the command line compiler to compile your existing project that contains the TReport component. You will not be able to open up this project inside the Delphi IDE and compile because the TReport component doesn't exist so your form will be unable to render this component. I would highly recommend rewriting any code that is dependent on TReport and use a reporting tool that is current and supported with the latest versions of Delphi. Delphi 7 ships with Rave so I would suggest looking into this.

Last Modified: 27-MAR-03

## Article #29821: Installation incomplete: Unable to locate ModelMaker

### Problem:

**When starting ModelMaker within the Delphi IDE you receive this error: "Installation incomplete: Unable to locate ModelMaker." It works fine when logged in as a user with Administrative rights but not with other users. Why does this happen?**

### Answer:

The IDE integration expert uses HKey\_Local\_Machine\Software\ModelMaker\ModelMaker\6.0 to detect the install dir. If you do not have Administrative rights during install some OS'es do not allow writing to HKLM and as a result the IDE experts cannot locate ModelMaker. You can workaroud this by starting ModelMaker from your Start menu or creating a shortcut on the desktop and running it from there.

Last Modified: 27-MAR-03

**Article #29826: Where are the Office '97 components in Delphi 7?**

**Problem:**

**What package contains the Office '97 components in Delphi 7**

**Answer:**

The Office '97 components can be found in Delphi7\bin\dclaxserver70.bpl

Last Modified: 27-MAR-03

## Article #29833: Intraweb Context Sensitive Help not working

### Problem:

**Pressing F1 when an Intraweb component is selected fails to bring up the help information**

### Resolution:

Start up Delphi. Select **Help | Customize**. This will start Borland OpenHelp and it should open up *D7.ohp*. Select the **Contents** tab. Select **Add Files** by clicking the add files button or right clicking in the ListView box. You must add the *Intraweb.cnt* file. In order to see this file you must change the Files of Type from \*.toc to \*.cnt. If you installed Delphi to the default location, this file can be found at:

*C:\Program Files\Borland\Delphi7\Help\IntraWeb*

You will get a message asking:

*Do you want to change the system registry path of Intraweb.cnt from C:\Program Files\Borland\Delphi7\Help\IntraWeb\Intraweb.cnt to C:\Program Files\Borland\Delphi7\Help\IntraWeb\Intraweb.cnt?*

Select **yes**.

Click on the **Index** tab. Select **Add Files**. You must add the *Intraweb.hlp* file. If you installed Delphi to the default location, this file can be found at:

*C:\Program Files\Borland\Delphi7\Help\IntraWeb*

You will get a message asking:

*Do you want to change the system registry path of INTRAWEB.HLP from C:\Program Files\Borland\Delphi7\Help\IntraWeb\INTRAWEB.HLP to C:\Program Files\Borland\Delphi7\Help\IntraWeb\INTRAWEB.HLP?*

Select **yes**.

Click on the **Link** tab. Select **Add Files**. You must add the *Intraweb.hlp* file. If you installed Delphi to the default location, this file can be found at:

*C:\Program Files\Borland\Delphi7\Help\IntraWeb*

**Save** the project inside of OpenHelp. **Close** OpenHelp. You must now go to the Delphi 7 help directory (using a utility such as Windows Explorer). If you installed to the default location go to:

***C:\Program Files\Borland\Delphi7\Help***

Erase all **.gid** files. These are hidden files so make sure you are able to view hidden files. Context Sensitive help should now work with the **Intraweb** components.

Last Modified: 27-MAR-03

**Article #29839: Where can I download the updated Registration Wizard for Delphi 7?**

**Question:**

**I am experiencing the blank registration wizard problem when I try and register Delphi 7. I heard there is an updated wizard that fixes this problem. Where can I download the updated Wizard?**

**Answer:**

Open up a browser and go to the following site:

<http://info.borland.com/devsupport/delphi/>

Click on the link that says Delphi 7 Registration Wizard.

Last Modified: 27-MAR-03

## Article #29840: CLX components not showing up under the System tab on the component palette

### Problem:

The following components aren't showing up under the System tab on the component palette:

- DirectoryTreeView
- FileListView
- FileIconView
- FileEdit
- FileHistoryComboBox
- FilderComboBox

### Resolution:

These components will only show up if you are developing a clx application. If you do **File | New | Clx Application** you will see these components under the **System** tab on the component palette.

Last Modified: 27-MAR-03

**Article #29853: Icon Image not valid error**

**Problem:**

**When starting Delphi, it says "Icon Image not Valid". How do I resolve this?**

**Answer:**

First, go into your display settings and disable Hardware Acceleration. Then, run Delphi. If that does not help, navigate to the directory which contains your delphi32.dro file and delete it. This file may be corrupt. The next time you run Delphi, this file will be regenerated, and the problem should be solved.

Last Modified: 27-MAR-03



**Article #29862: Error: "Projects macro in project group file is missing or incorrect"**

**Problem:**

**When starting Delphi or trying to create a new application you get the error "Projects macro in project group file is missing or incorrect"**

**Answer:**

To resolve this, go into Tools|Editor Options|Display and set the Right Margin property to a higher value. This sometimes occurs when too small a value is specified.

Last Modified: 27-MAR-03

## Article #29865: problems using valuetype in idl file: idl2pas: fatal error

### Problem:

When running idl2pas on a idl file that has a valuetype, the following error is raised:  
*idl2pas: fatal error: internal problem; please contact Inprise Customer Support*

### Explanation:

This isn't a bug. valuetype isn't currently supported with idl2pas. Review the documentation on idl2pas for a complete reference of the idl to Delphi mapping.

Last Modified: 27-MAR-03

**Article #29866: problems with dbExpress connecting to DB2. Get error: SQL0204N "tablename" is an undefined name. SQLSTATE-42704**

**Problem:**

When using dbExpress and accessing a table on a DB2 database, if I have a select statement in the CommandText property of a TSQLDataSet in the form:

```
select * from employee
```

the following error is raised when attempting to apply updates:

```
Project TestDBX.exe raised exception class EDatabaseError with message '[IBM][CLI Driver] [DB2/NT] SQL0204N "employee" is an undefined name. SQLSTATE-42704
```

This error is raised even after verifying that employee is a valid table in the database.

**Solution:**

The way dbExpress handles the SQL statement requires the Table Name and the Schema Name to be in Upper Case letters. In the CommandText property, change the sql statement to:

```
select * from EMPLOYEE
```

and the error listed above should go away.

Last Modified: 27-MAR-03

## **Article #29872: Cannot install Delphi because a previous version is already installed**

### **Problem:**

**You cannot install Delphi because it indicates that a previous version is already installed.**

### **Answer:**

This can occur on install if a different SKU of Delphi of the same version of Delphi was not completely uninstalled. The MSI still thinks that Delphi is installed. To resolve this, run regedit and navigate to HKEY\_CURRENT\_USER\Software\Microsoft\Installer\Products and do a search for Borland. One of the subfolders will contain a reference to the version of Delphi that you are trying to install. Export this subfolder as a safety measure and then delete it from the registry. Then, try to install Delphi and you should no longer get this error message.

Last Modified: 28-MAR-03

## Article #29900: Installing Delphi in Silent Mode

### Question:

How can I do a silent mode installation of Delphi?

### Answer:

Since Delphi has a per user license agreement, silent mode installations of the product are not supported.

Quoting directly from the license.txt file:

"You may install the Product on a reasonable number of machines provided that the Named User is the only individual permitted to use the Product"

This being stated, as long as you are adhering to the license agreement, there shouldn't be a need to automate the installation process since the product shouldn't be installed on more than a handful of machines.

Last Modified: 28-MAY-03

**Article #29994: 'Drive C: does not exist. Please verify the correct drive was given' when trying to import slip activation file**

**Problem:**

**When trying to register Delphi on a Windows 98 or Windows ME machine you receive the 'Drive C: does not exist. Please verify the correct drive was given' error after browsing to and opening your activation file.**

**Answer:**

There is a small bug in the CLX TOpenDialog which is used in the registration wizard. This error only surfaces on Windows 98 or Windows ME machines. To get around this just type the path to the activation file in the edit box rather than clicking the ellipsis to open the TOpenDialog box.

Last Modified: 28-MAY-03

## Article #29995: Rave 4.5 CE version for Linux doesn't print

### Problem:

After installing Rave 4.5 CE version on Linux (for Kylix) everything seems to work properly except I can't get the Rave Project to print.

### Solution:

This is a known issue with the Rave 4.5 CE version on Linux. This has been fixed in the Rave 5.0 BEX release. To purchase the 5.0 BEX version, go to Navrona's web site at:

<http://www.nevrona.com>

Last Modified: 28-MAY-03

## Article #29996: Enabling automatic scrolling in a TTreeView

### Problem:

**How do I emulate the scrolling behavior of Windows Explorer when dragging a folder to the top or bottom of my TTreeView?**

### Answer:

Set your TTreeView's DragMode property to dmAutomatic and then place this code in your Treeview's OnDragOver event:

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject; X, Y: Integer; State: TDragState; var Accept: Boolean);  
var  
    HitTests : THitTests;  
begin  
    HitTests := TreeView1.GetHitTestInfoAt(X,Y);  
    if (htAbove in HitTests) then  
        TreeView1.TopItem := TreeView1.TopItem.GetPrevVisible  
    else if (htBelow in HitTests) then  
        TreeView1.TopItem := TreeView1.TopItem.GetNextVisible;  
end;
```

Last Modified: 28-MAY-03



## Article #29997: Numerous users on one Windows 2000 / NT / XP machine

### Question:

Different developers share the same machine. Each developer has their own username on the same machine. Is there a way to register Delphi once on the machine and push that registration info to each additional user on that machine?

### Answer:

No, this isn't allowed as Delphi has a per user license agreement. This means each user that will be using Delphi will need to register.

Last Modified: 02-JUN-03

## Article #30002: Failed to create process error when trying to use the CORBA Server or Client Wizards

### Problem:

Running the Corba Wizards from the Object Repository (File | New | Other | Select the CORBA tab and choose the CORBA Client or CORBA Server wizard) fails after selecting the IDL file and clicking the Generate button. It fails with the message:

*Failed to create process. The system cannot find the file specified.*

### Solution:

This problem is caused by having the wrong JRE installed on the machine. Download and install the 1.3 (make sure it is 1.3.0 as later versions will not work) JRE or JDK from Sun's website at: <http://java.sun.com>. This should resolve this issue.

Last Modified: 05-JUN-03

**Article #30003: When launching Delphi 7 on a Windows 95 machine you get this error: 'The COREIDE70.bpl is linked to missing export USER32.DLL: TrackMouseEvent'**

**Problem:**

**When starting Delphi 7 on Windows 95, I get 'The COREIDE70.bpl is linked to missing export USER32.DLL: TrackMouseEvent'**

**Answer:**

Delphi 7 is not certified on Windows 95 and ultimately we do not recommend that you use Delphi on this operating system. What is happening is that in Windows 98 and later editions of Windows, the TrackMouseEvent function is contained within USER32.dll. However, in Windows 95 it relies on Comctl32.dll to provide that functionality. A potential solution is to update your Comctl32.dll which can be done by installing Internet Explorer 5.5 or later.

Last Modified: 02-JUN-03

## Article #30006: Create Interface for SOAP server

### Problem:

I created a new Soap Server Application. The dialog box came up asking: Create Interface for SOAP module? I accidently clicked no, how can I recover and create the interface?

### Solution:

With your Soap Server Application open, Select File| New| Other and choose the WebServices tab and select **Soap Server Interface**

Last Modified: 02-JUN-03

**Article #30007: Using the Rave Reporting tool, how do I make a Header repeat on each page?**

**Problem:**

**How can I make a Header repeat on each page?**

**Answer:**

One way of doing this is to create a Global Page in the Rave Visual Designer, drop a Section component on the page and then put whatever you would like in your header into the Section component. Then, on each of your pages, drop a Section component and set it's Mirror property to the Global Page's Section component.

Last Modified: 02-JUN-03

## **Article #30008: How do I insert today's date into my Rave Report?**

### **Problem:**

**I want to insert today's date into my Rave Report but can't figure out how to do it. How can I do this?**

### **Answer:**

In the Rave Visual Designer, drop a DataText component onto your page and then click the ellipsis next to your DataText component's DataField property. This will bring up the Data Text Editor. The Report variable combobox contains a long list of items such as the current date that you can use in your report by selecting the item and clicking the Insert Report Var button.

Last Modified: 02-JUN-03

## Article #30023: Unable to resolve error, 'Table is read only.'

**Question:** When using an UpdateSQL component I get the following error when trying to apply updates: 'Table is read only.'

**Solution:** There are a couple of possible reasons for getting this error. First you should check that the query is indeed not set to read only. Next if you are using the **OnUpdateRecord** event then you will need to specify the result for the **UpdateAction** parameter. If the update is successful then **UpdateAction** should be set to *uaApplied*.

Last Modified: 09-MAY-03

**Article #30058: On Windows XP you get "Error 1606. Could not access network location \." when installing Delphi**

**Problem:**

**When installing Delphi on Windows XP, an error occurs: "Error 1606. Could not access network location ."**

**Answer:**

Click [Here](#) to read the Microsoft Knowledge Base article that describes what this error means and how to get around it

Last Modified: 02-JUN-03



**bdn.borland.com**  
**Delphi Technical Information Docs**

- 10531:** BDE and the Year 2000 27-OCT-99
- 15005:** Understanding the PARADOX.NET File 11-FEB-00
- 15007:** Skip "Include error messages?" prompt 30-MAR-99
- 15008:** What to check when you cannot get an all users int 15-JUL-98
- 15017:** Interrupt 3 patch 30-MAR-99
- 15019:** Conflicts between Delphi and Netcruiser 16-JUL-98
- 15026:** MS-DOS interrupt 24 trap routine 30-MAR-99
- 15035:** Kaypro - Ever-spinning disk drive 30-MAR-99
- 15037:** Delphi End User CD-ROM can not be mounted by Novel 16-JUL-98
- 15044:** Direct Video Accessing 30-MAR-99
- 15053:** Modifying the aspect ratio in the graph unit 30-MAR-99
- 15055:** 'Not initialized for accessing network files' 16-JUL-98
- 15062:** Database Toolbox suggestions/corrections 30-MAR-99
- 15064:** INTERFACING MODULES WRITTEN IN MICROSOFT C 16-JUL-98
- 15071:** Printing graphics to a HP Laserjet 31-MAR-99
- 15073:** LINKING .OBJ OBJECT FILES FROM OTHER ASSEMBLERS 16-JUL-98
- 15080:** Printing graphics to an Epson compatible printer 30-MAR-99
- 15082:** OBJECT FILE CREATION 16-JUL-98
- 15089:** Microstar critical error handler corrections 30-MAR-99
- 15091:** LINKING TURBO C OR ASSEMBLER .OBJ OBJECT FILES 16-JUL-98
- 15098:** Handling Special Keys and Direct Memory Access 31-MAR-99
- 15100:** DATA SEGMENT USAGE OF SYSTEM UNITS 16-JUL-98
- 15107:** Changing Color of Buttons with Bitmaps 16-JUL-98
- 15109:** VARIABLES IN UNITS CHANGES NOT REFLECTED 16-JUL-98
- 15116:** DrawAxis option: no tick marks and number labels 30-MAR-99
- 15118:** UNITS LINKED WITH DIFFERENT COMPILER DIRECTIVES 16-JUL-98
- 15125:** Patch for include file not found by Turbo Debugger 30-MAR-99
- 15127:** Units compiled more than once 16-JUL-98
- 15134:** Linking error message and font files. 16-JUL-98
- 15136:** TYPE MISMATCH ERROR COMPILING UNITS 16-JUL-98
- 15143:** Turbo Vision, 132 Column Mode 16-JUL-98
- 15145:** CREATING .TPL FILES 16-JUL-98
- 15150:** Function mapping from the Paradox Engine to BDE 15-JUL-98
- 15152:** Toggling the cursor ON/OFF 30-MAR-99
- 15154:** LINKING UNITS IN THE TURBO.TPL FILE 16-JUL-98
- 15159:** Some current internal limits of BDE 15-JUL-98

**15161:** Extend File Handles allowing more than 15. 16-JUL-98  
**15163:** TINST - NEW SETTINGS AREN'T USED 16-JUL-98  
**15168:** Local SQL Reserved Words. 15-JUL-98  
**15170:** Network File Access 16-JUL-98  
**15172:** EMS SUPPORT 16-JUL-98  
**15177:** Introduction to BDE Programming 15-JUL-98  
**15179:** editors.pas 3,4,5,6 entry patch for TP6. 30-MAR-99  
**15181:** OUT OF MEMORY ERROR 16-JUL-98  
**15186:** Comparison filters. 15-JUL-98  
**15188:** How to Print in Windows 29-JUL-99  
**15190:** SYSTEM LOCK RUNNING WITHIN IDE VS. DISK RUN 16-JUL-98  
**15194:** Using a continue node in a filter. 15-JUL-98  
**15196:** Dual Input Lines in Turbo Vision,GetData SetData 16-JUL-98  
**15198:** PICK FILES - TURBO READS FROM ANOTHER DRIVE 16-JUL-98  
**15203:** BDE Frequently Asked Questions. 15-JUL-98  
**15205:** Change Menubar at Runtime 16-JUL-98  
**15207:** CURSOR SIZE CHANGE 16-JUL-98  
**15209:** 'BLOB has been modified.', 'Index is out of date.' errors 21-MAR-00  
**15212:** Understanding the PARADOX.NET file with the BDE 15-JUL-98  
**15214:** Messaging Between Two Dialogs/Update Background 16-JUL-98  
**15216:** IDE - AVAILABLE MEMORY 16-JUL-98  
**15221:** Setting File Handles For A Windows BDE Application 15-JUL-98  
**15223:** Associating Help Features with Context Hints 16-JUL-98  
**15225:** Real number errors 16-JUL-98  
**15230:** Changing the NET DIR Programmatically 15-JUL-98  
**15232:** How to Create a Listbox and Get a Double Click 16-JUL-98  
**15234:** The location of GRAPH.TPU 16-JUL-98  
**15239:** Error solutions and limitations of Paradox Engine 15-JUL-98  
**15241:** Runtime Menubar and Statusline Change 16-JUL-98  
**15243:** Overwriting memory accidentally 16-JUL-98  
**15247:** BDE setup for Peer-To-Peer(Non-Dedicated) Networks 15-JUL-98  
**15249:** Change Foreground and Background Color 16-JUL-98  
**15251:** Out of Memory during compilation 16-JUL-98  
**15256:** Removing "Lock file has grown too large" Error 15-JUL-98  
**15258:** Simple Pick List Example 16-JUL-98  
**15260:** Exec procedure not executing subprograms 16-JUL-98  
**15265:** Sharing Violation Error with Paradox Tables. 15-JUL-98  
**15267:** Desktop Save State Using Streams 16-JUL-98  
**15269:** Communicating with the serial port 16-JUL-98  
**15274:** Steps for FAT32 Support with the BDE 3.5 15-JUL-98

**15276:** Loop Processing in Windows 16-JUL-98  
**15278:** Dumping Graphics to the printer 16-JUL-98  
**15283:** BDE Error listing 04-OCT-99  
**15285:** Creating and Using Fonts in the Resource Workshop 16-JUL-98  
**15287:** The Source of "Unit file format error" 16-JUL-98  
**15293:** Changing Fonts in Edits Controls 16-JUL-98  
**15295:** Redirecting printer output 16-JUL-98  
**15301:** Use Collections and ListBoxes. 16-JUL-98  
**15303:** Reading and Writing Files Examples 16-JUL-98  
**15309:** Different Colors in Turbo Vision. 16-JUL-98  
**15311:** DEBUGGING EXTERNAL ASSEMBLY ROUTINES 16-JUL-98  
**15317:** Turbo Pascal vs ANSI Pascal 30-MAR-99  
**15319:** INCLUDE FILE NOT FOUND BY TURBO DEBUGGER 16-JUL-98  
**15325:** Overview of Borland and Turbo Pascal 7.0 30-MAR-99  
**15327:** NESTED FOR LOOPS- How many? 16-JUL-98  
**15333:** Questions and Answers of Borland and Turbo Pascal 30-MAR-99  
**15335:** RESERVED WORD 'PACKED' 16-JUL-98  
**15341:** Creating Protected Mode Apps with Borland Pascal 30-MAR-99  
**15343:** CASE - IDENTIFIERS OTHER THAN SCALAR 16-JUL-98  
**15348:** Installing Turbo Pascal 7.0 on a Network 30-MAR-99  
**15350:** READLN - READING A STRING 16-JUL-98  
**15355:** Using the VESA16.BGI that Comes with BP 7.0 30-MAR-99  
**15357:** TASM 1.0 1.01 - ORDER OF ARG DIRECTIVE OPERANDS - 16-JUL-98  
**15363:** Using Validators in OWL Applications 16-JUL-98  
**15365:** TASM 1.0 - RETURN FAILED FROM ASSEMBLY ROUTINE - T 16-JUL-98  
**15369:** Printing the Run of a Program 22-APR-99  
**15371:** OVERLAY UNITS LOADED INTO BUFFER 16-JUL-98  
**15376:** Keyboard Problems Using Borland Pascal and TPX 16-JUL-98  
**15378:** TP OVERLAYS: RECLAIMING BUFFER MEMORY 16-JUL-98  
**15383:** Suggestions for Installing from Backups 16-JUL-98  
**15385:** OVERLAYS \*.OVR FILES 16-JUL-98  
**15390:** Creating Groups for Borland Pascal 22-APR-99  
**15396:** Patch for the STDDLG Unit 16-JUL-98  
**15398:** OVERLAY SUPPORT 16-JUL-98  
**15403:** Mouse Driver Compatibility 31-MAR-99  
**15405:** EXTERNAL METHODS 16-JUL-98  
**15410:** Recommended PIF Settings for Windows 31-MAR-99  
**15412:** RECORDS VS. OBJECTS 16-JUL-98  
**15416:** Booting clean -- tips for installing Turbo Pascal 30-MAR-99  
**15418:** PROTECTED AND PRIVATE FIELDS 16-JUL-98

**15422:** Does a Turbo Vision control have the input focus? 16-JUL-98

**15424:** CALLING ANCESTORS CONSTRUCTOR 16-JUL-98

**15428:** Declaring an array on the heap 22-APR-99

**15430:** CONSTRUCTOR CALLING ANCESTOR'S CONSTRUCTOR 16-JUL-98

**15434:** Making a help group for Borland Pascal 31-MAR-99

**15436:** CONSTRUCTOR CALL WITHIN METHOD 16-JUL-98

**15440:** Turbo Vision local menu using right mouse button 16-JUL-98

**15442:** OVERRIDE VIRTUAL METHOD CALLING ANCESTOR 16-JUL-98

**15447:** Using procedural variables in Pascal 22-APR-99

**15449:** CONSTRUCTOR CALL TO ANCESTOR WITH VIRTUALS 16-JUL-98

**15454:** Using editor toolbox with newer versions of Pascal. 30-MAR-99

**15456:** LINKER STRIPS UNUSED STATIC METHODS 16-JUL-98

**15461:** Having two dialogs on your screen at once. 16-JUL-98

**15463:** SIZEOF OBJECTS CONSTRUCTOR 16-JUL-98

**15468:** Printing in the TP/BP IDE 31-MAR-99

**15470:** FILES OF OBJECT TYPE 16-JUL-98

**15475:** Getting and setting a disks serial number 16-JUL-98

**15477:** SAVING OBJECTS TO DISK 16-JUL-98

**15481:** Creating a temporary stack. 22-APR-99

**15483:** DESTRUCTOR HEAP CORRUPTION 16-JUL-98

**15488:** How to read an enhanced keyboard 16-JUL-98

**15490:** CONSTRUCTOR USE 16-JUL-98

**15494:** Copying a desktop window into your program's window 16-JUL-98

**15496:** OBJECT EXE FILE SIZE OVERHEAD 16-JUL-98

**15501:** Configuring the Borland Pascal icon in OS/2 31-MAR-99

**15506:** Unknown runtime errors. What's error 163? 16-JUL-98

**15511:** Creating an OWL app without a border and caption 16-JUL-98

**15516:** Uninstalling Delphi 30-MAR-99

**15521:** Getting / Setting diskette serial # under Windows. 16-JUL-98

**15526:** Installing Delphi on a Network 30-MAR-99

**15531:** Enabling a horizontal scroll bar for a list box. 30-MAR-99

**15536:** Making a form selectable without the main form. 16-JUL-98

**15541:** How to get a file's date and time stamp. 16-JUL-98

**15546:** Lists the virtual key values 16-JUL-98

**15551:** How to paint the form with a bitmap. 16-JUL-98

**15556:** How to associate a string with a component. 16-JUL-98

**15561:** How close a file opened from a Delphi DLL in VB. 16-JUL-98

**15566:** Changing the border color in DOS text mode. 16-JUL-98

**15571:** How to readln longer than 255 chars. 16-JUL-98

**15576:** A VERY short primer on dynamic memory allocation. 16-JUL-98

**15578:** HEAP SPACE STILL CREATED WITH HEAP MAX 0 16-JUL-98

**15582:** How to use array of const. 16-JUL-98

**15587:** Making your own hotkeys. 16-JUL-98

**15589:** MEMORY CONTROL BLOCKS 16-JUL-98

**15593:** How to scroll your form with pgUP and pgDn. 16-JUL-98

**15595:** ALLOCATING AND RECLAIMING MEMORY USING DOS 16-JUL-98

**15599:** Trapping 8087 floating point exceptions in Pascal. 16-JUL-98

**15601:** EXEC - SETTING HEAP MEMORY 16-JUL-98

**15605:** Drag a form without clicking the caption bar 16-JUL-98

**15607:** TAKING ADVANTAGE OF IEEE FLOATING-POINT TYPES 16-JUL-98

**15613:** TRUNC - CONVERTING REAL NUMBERS INTO INTEGERS 16-JUL-98

**15617:** Accessing more than 15 files from your DOS app. 29-MAR-00

**15619:** RANDOM RANDOMIZE - RANDOM NUMBER GENERATOR FORMULA 16-JUL-98

**15623:** How to Circumvent the "index not found" Exception. 22-APR-99

**15625:** ROUNDING WITH IEEE FLOATING-POINT 16-JUL-98

**15629:** Getting version information from a Windows app. 16-JUL-98

**15630:** Compiling RWPDemo.pas in Turbo Pascal for Windows. 02-JUL-98

**15631:** INTERMEDIATE RESULTS OF REAL NUMBER EXPRESSIONS 16-JUL-98

**15635:** Using your EXE as a resource file 16-JUL-98

**15637:** SIGNIFICANT DIGITS OF 8087 FLOATING-POINT TYPES 16-JUL-98

**15641:** College Student Guide to Reading and Writing Files 16-JUL-98

**15643:** COMP TYPE USES 16-JUL-98

**15647:** Using FindFirst and FindNext to find directories. 16-JUL-98

**15649:** 80x87 NUMERIC COPROCESSOR 16-JUL-98

**15653:** Delphi Fact Sheet 30-MAR-99

**15655:** FLOATING-POINT PARAMETERS 16-JUL-98

**15659:** Delphi Quick Info Sheet 30-MAR-99

**15661:** LARGE INTEGER TYPE SUPPORT 16-JUL-98

**15665:** List of Delphi Books From Third-Party Publishers 30-MAR-99

**15667:** LINKER ELIMINATES UNUSED DATA 16-JUL-98

**15671:** Instructions for Running Delphi from CD-ROM 01-APR-99

**15673:** STRING MANIPULATION IN AN INTERRUPT ROUTINE 16-JUL-98

**15677:** Making your Delphi apps show minimized. 31-MAR-99

**15679:** NON-REENTRANT CODE 16-JUL-98

**15683:** Delphi Client/Server and Power Builder Compared 16-JUL-98

**15685:** REENTRANT CODE 16-JUL-98

**15689:** Delphi vs Visual Basic 16-JUL-98

**15691:** INTERRUPT HANDLER RULES 16-JUL-98

**15695:** Configuration of an ODBC Driver with BDE v3.5 and less 22-APR-99

**15697:** FILE HANDLES USED BY TURBO PASCAL 16-JUL-98

**15701:** Creating Database Aliases in Code 16-JUL-98  
**15703:** FINDFIRST - THE EXISTENCE OF FILES 16-JUL-98  
**15707:** Creating & Deleting TFields at run-time 16-JUL-98  
**15709:** SEEK - RANDOM-ACCESS FILES 16-JUL-98  
**15713:** How to iterate through the fields of a table 16-JUL-98  
**15715:** READ - SKIPPING OVER A READLN OR READ 16-JUL-98  
**15719:** How to match file date/time stamps. 16-JUL-98  
**15721:** RESET - READING AND WRITING TYPED UNTYPED FILES 16-JUL-98  
**15725:** Adding Graphics in Your Listboxes and Comboboxes 16-JUL-98  
**15727:** READING CHARACTERS DIRECTLY FROM SCREEN 16-JUL-98  
**15731:** Using OnHint Events Among Multiple Forms 16-JUL-98  
**15733:** WRITING TO FILE 'CON' 16-JUL-98  
**15737:** Making the Enter key work like a Tab in a TDBGrid 16-JUL-98  
**15739:** I/O BUFFER FOR TEXT FILES 16-JUL-98  
**15743:** Moving to a tab by name on a TabSet 16-JUL-98  
**15745:** FILE AND RECORD LOCKING SUPPORT 16-JUL-98  
**15749:** Calling a Delphi DLL from C 16-JUL-98  
**15751:** FILE MODE VARIABLES 16-JUL-98  
**15755:** Getting a query's Memo field as a string 16-JUL-98  
**15757:** CLEARING THE KEYBOARD BUFFER 16-JUL-98  
**15761:** How to encrypt a String 30-MAR-99  
**15763:** I/O REDIRECTION \$P128 \$G128 16-JUL-98  
**15767:** Undo in a Memo field 16-JUL-98  
**15769:** OPENING MORE THAN 15 FILES 16-JUL-98  
**15773:** Getting the Line number in a memo Field 16-JUL-98  
**15775:** CTRL-ALT-DEL TRAP KEYBOARD ISR 16-JUL-98  
**15779:** Loading Bitmaps Into dBASE And Paradox BLOB Fields 16-JUL-98  
**15781:** GRAPH DETECT VIDEO CARD AUTODETECT HARDWARE 16-JUL-98  
**15785:** Printing the contents of a TMemo or TListbox 16-JUL-98  
**15787:** REVERSE VIDEO WITH GRAPH UNIT 16-JUL-98  
**15791:** Extracting A Bitmap From A BLOB Field 16-JUL-98  
**15793:** INVERSE VIDEO TEXT 16-JUL-98  
**15797:** Bitmaps And InterBase BLOB Fields 16-JUL-98  
**15799:** HIDING THE CURSOR DOS INTERRUPTS 16-JUL-98  
**15803:** Control Font Styles 16-JUL-98  
**15805:** UNIT FILE FORMAT ERROR 16-JUL-98  
**15809:** Handling EDBEngineError Exceptions 16-JUL-98  
**15814:** How to handle text drawing in a TDBGrid 16-JUL-98  
**15816:** "INTERNAL ERROR REPORT TO BORLAND" 16-JUL-98  
**15820:** Trouble running Delphi programs from Delphi 01-APR-99

**15822:** MOVING FILES ACROSS DIRECTORIES 16-JUL-98  
**15826:** Searching your application's help file 16-JUL-98  
**15828:** DOS - PARAMSTR AND PARAMCOUNT 16-JUL-98  
**15832:** Searching Through Query Result Sets 16-JUL-98  
**15834:** RETURNING ERROR LEVELS TO BATCH FILES 16-JUL-98  
**15838:** dBASE .DBF File Structure 16-JUL-98  
**15840:** FILE HANDLES EXTENDING 16-JUL-98  
**15844:** Passing a Variable to a ReportSmith Report 31-MAR-99  
**15846:** DOS - PROCEDURE FLUSH 16-JUL-98  
**15850:** Extracting Index Data From A Table 16-JUL-98  
**15852:** DOS - ILLEGAL FILENAME; COMMAND.COM OVERWRITE 16-JUL-98  
**15856:** Redistributing the 16 bit Borland Database Engine 22-APR-99  
**15858:** DOS FILES - PROCEDURE FINDFIRST 16-JUL-98  
**15862:** Cascading Deletes With Pdox Referential Integrity 16-JUL-98  
**15864:** DOS - CHDIR PROCEDURE 16-JUL-98  
**15868:** dBASE Expression Indexes: A Primer 16-JUL-98  
**15870:** GETINFO - FREE MEMORY IN DOS 16-JUL-98  
**15874:** Removing the vertical scrollbar from a TDBGrid 16-JUL-98  
**15876:** OOP - DATA AND CODE SEGMENT SIZES 16-JUL-98  
**15882:** 4.0 TOOLBOX COMPATIBILITY 16-JUL-98  
**15886:** InterBase BLOB Fields: A Primer 16-JUL-98  
**15888:** UPGRADING 3.0 PROGRAMS WITH CHAIN FILES 16-JUL-98  
**15892:** Using The ASCII Driver With Comma-delimited Files 16-JUL-98  
**15894:** READ & READLN - DIFFERENCES BETWEEN 3.0 AND 4.0+ 16-JUL-98  
**15898:** Determining Record Number In A dBASE Table 16-JUL-98  
**15900:** RUN-TIME ERROR CODE CHANGES FROM VERSION 3.0 16-JUL-98  
**15904:** Managing disk volume labels in Delphi 16-JUL-98  
**15906:** UPGRADING 3.0 TO 4.0+ PROBLEMS 16-JUL-98  
**15910:** How to copy files in Delphi. 16-JUL-98  
**15912:** COMPILING VERSION 2.0 CODE WITH THE VERSION 4.0+ C 16-JUL-98  
**15916:** How to terminate all running applications 16-JUL-98  
**15921:** How to check to see if a drive is ready. 16-JUL-98  
**15926:** How to do pointer arithmetic in Delphi. 16-JUL-98  
**15928:** DEBUGGER - DISPLAYING FUNCTIONS IN THE WATCH WINDO 16-JUL-98  
**15932:** How to do bit-wise manipulation. 16-JUL-98  
**15934:** DEBUGGER - POINTERS CORRUPTING MEMORY? 16-JUL-98  
**15938:** Basic Delphi DLL template 16-JUL-98  
**15940:** INVALID LINKER VERSION ERROR 16-JUL-98  
**15944:** Form display with different screen resolutions. 16-JUL-98  
**15946:** DEBUGGER - STEPPING INTO A PROCEDURE 16-JUL-98

**15950:** How to keep the app iconized. 16-JUL-98  
**15952:** IS THERE A DEBUGGER AVAILABLE FOR TP 5.0+? 16-JUL-98  
**15956:** How to use a custom cursor. 16-JUL-98  
**15958:** ACCESSING MORE THAN 64K OF DATA 16-JUL-98  
**15962:** How to use a popup menu with a VBX. 31-MAR-99  
**15964:** NORMALIZING POINTERS 16-JUL-98  
**15968:** How to set a max and min form size. 16-JUL-98  
**15970:** DATA SEGMENT TOO LARGE ERROR 16-JUL-98  
**15974:** How to get the windows and DOS versions. 16-JUL-98  
**15976:** FILE MODE LOCKING/UNLOCKING ON A NETWORK 16-JUL-98  
**15980:** How to tell what kind of drive is used. 16-JUL-98  
**15982:** VARIANT RECORD LONGINT TO BYTE AND WORD 16-JUL-98  
**15986:** How to determine the current record number. 16-JUL-98  
**15988:** POST-DECLARATION TYPECASTING 16-JUL-98  
**15992:** How to automate logon for Paradox tables 16-JUL-98  
**15994:** STATIC VARIABLES LIKE C 16-JUL-98  
**15998:** packing a dBASE table 16-JUL-98  
**16000:** LARGEST GLOBAL DATA STRUCTURE 16-JUL-98  
**16004:** string manipulation routines 16-JUL-98  
**16006:** STRING SIZE LIMIT 255 CHARS? 16-JUL-98  
**16010:** WinExecAndWait 16-JUL-98  
**16015:** How to check for app already running. 16-JUL-98  
**16017:** Compatibility with MS-DOS GENERIC MACHINES 16-JUL-98  
**16021:** How to use a form several times 16-JUL-98  
**16023:** SERIAL COMMUNICATION AUXINOUT UNIT 16-JUL-98  
**16027:** Manually Installing Delphi 30-MAR-99  
**16029:** ASYNCR LOST CHARACTERS BROKEN SERIAL 16-JUL-98  
**16033:** Printing in the DOS IDE under Windows 95 22-APR-99  
**16039:** Handling " system error cannot read cdrom.bmp" 16-JUL-98  
**16045:** Different colored characters in a string grid 16-JUL-98  
**16051:** Returns the amount required to repay a debt. 16-JUL-98  
**16057:** How to click and move components at runtime. 16-JUL-98  
**16063:** Delphi 1.02 Maintenance Release Information 30-MAR-99  
**16069:** Creating Dynamic Components at Runtime 16-JUL-98  
**16075:** Loading a Custom Cursor from a RES File 16-JUL-98  
**16081:** Loading Bitmaps and Cursors from RES Files 16-JUL-98  
**16087:** SQL: Embedded Spaces in Field/Column Names 16-JUL-98  
**16093:** Dynamically Allocating Arrays 02-APR-99  
**16099:** Resource Expert: What It Is and How to Install It 16-JUL-98  
**16105:** Delphi Configuration Files 30-MAR-99



**16111:** BDE: Writing Buffer to Disk 16-JUL-98  
**16117:** Creating and Using Parameterized Queries 16-JUL-98  
**16123:** Working With Auto-increment Field Types 16-JUL-98  
**16129:** How to Populate a TDBComboBox Or TDBListBox 16-JUL-98  
**16135:** New Language Features in Delphi 2.0 30-MAR-99  
**16141:** Preventing a Form from Resizing 16-JUL-98  
**16147:** SQL: Sorting on a Calculated Column 16-JUL-98  
**16153:** SQL: Using the SUBSTRING Function 16-JUL-98  
**16159:** SQL: Summarizing a Calculated Column 16-JUL-98  
**16165:** Managing Data Segment Size 22-APR-99  
**16171:** Validating input in TEdit components 16-JUL-98  
**16177:** DDE: A simple example 01-APR-99  
**16183:** TDBGrid and Multi-Selecting Records 30-MAR-99  
**16189:** Listing the field structures of a table. 31-MAR-99  
**16195:** Showing deleted records in a dBASE table. 16-JUL-98  
**16201:** Determining a memo's number of lines showing. 16-JUL-98  
**16203:** Desqview Control Break Conflict 16-JUL-98  
**16207:** Delphi 2.0 Install Issues 30-MAR-99  
**16209:** Using the 80X87 emulator in TPW. 16-JUL-98  
**16213:** How to Validate ISBNs 22-APR-99  
**16215:** The \$G+ directive, 286 code generation, and Real M 16-JUL-98  
**16219:** Delphi 2.0 for Windows 95 & Windows NT Factsheet 30-MAR-99  
**16221:** TPW 1.0 Hot Sheet. 16-JUL-98  
**16225:** Delphi Client/Server Suite 2.0 for Windows 95 & NT 30-MAR-99  
**16227:** Status line in a MDI Application. 16-JUL-98  
**16231:** Performing database queries in a background thread 30-MAR-99  
**16233:** Changing Button Text 16-JUL-98  
**16237:** How to check a ComboBox without OnClick occurring. 16-JUL-98  
**16239:** TDW popping in dual mode in TPW. 16-JUL-98  
**16243:** TForm.MDICHildren[] Array and Form Creation 16-JUL-98  
**16245:** PChars in TPW. 16-JUL-98  
**16251:** WinCrt and Windows 3.0 16-JUL-98  
**16257:** Turbo Pascal for Windows New and GetMem procedures 16-JUL-98  
**16263:** Exec from within a Turbo Pascal for Windows progra 16-JUL-98  
**16269:** Local Heap and Global Heap 16-JUL-98  
**16275:** Turbo Pascal for Windows Strings literals in the D 16-JUL-98  
**16279:** Redistributing Applications using the ISP 30-MAR-99  
**16281:** Turbo Pascal for Windows "Can't find Debugger" err 16-JUL-98  
**16285:** How to Create a TDBGrid Lookup Field in 32-bit Delphi 30-MAR-99  
**16287:** Timer Program Example 16-JUL-98

**16297:** [Navigating a Multiselect Listbox](#) 16-JUL-98  
**16299:** [TP6 IDE Memory Documentation](#) 16-JUL-98  
**16303:** [Making your 32 bit Delphi applications "sing"](#) 26-MAR-99  
**16305:** [TP 6.0 Carton Text](#) 16-JUL-98  
**16309:** [Obtaining the Physical Path of a Table](#) 30-MAR-99  
**16311:** [TP6 Benchmarks](#) 16-JUL-98  
**16315:** [Making accelerators work with a TPageControl](#) 26-MAR-99  
**16317:** [TP6 Marketing QNAs](#) 16-JUL-98  
**16321:** [How to dynamically create a TPageControl](#) 26-MAR-99  
**16323:** [TP6.0 Hot Sheet](#) 16-JUL-98  
**16327:** [BDE Callbacks to Provide Status on Operations](#) 30-MAR-99  
**16329:** [TP6 Fact Sheet \(Draft 1.0\)](#) 16-JUL-98  
**16333:** [Accessing Paradox Tables on CD or Read-Only Drive](#) 16-JUL-98  
**16339:** [Synchronize a DLL to an Open Dataset](#) 16-JUL-98  
**16341:** [Fix for BP7 GRAFFITI example.](#) 16-JUL-98  
**16345:** [Clean-Boot 32 bit Delphi Installation](#) 01-APR-99  
**16351:** [Creating a Wallpaper Using Delphi](#) 16-JUL-98  
**16357:** [Detecting Windows Shutdown](#) 01-APR-99  
**16363:** [Returning Default Cursor after Running Queries](#) 16-JUL-98  
**16369:** [Dynamic creation and circularly linking forms](#) 16-JUL-98  
**16375:** [Avoid using Resource Heap with Tabbed Notebooks](#) 16-JUL-98  
**16377:** [Installing Acrobat Reader for Windows](#) 16-JUL-98  
**16381:** [The DocInput Object: Properties and Methods](#) 30-MAR-99  
**16392:** [Creating Class Properties](#) 01-APR-99  
**16397:** [Optimizing Oracle Connections with Windows 95](#) 22-APR-99  
**16402:** [Connecting to a 32-bit Sybase server](#) 22-APR-99  
**16407:** [Hints on Overcoming Installation Problems](#) 22-APR-99  
**16412:** [A Better Way To Print a Form](#) 22-APR-99  
**16417:** [Creating a DataAware Control for Browsing Data](#) 16-JUL-98  
**16422:** [Using InputBox, InputQuery, and ShowMessage](#) 16-JUL-98  
**16427:** [Create a new file with the .wav extension.](#) 16-JUL-98  
**16432:** [Using MS Internet Explorer 3.0 OCX in Delphi 2](#) 30-MAR-99  
**16437:** [BDE and Database Desktop Locking Protocol](#) 31-MAR-99  
**16442:** [Getting a record member char array into a memo.](#) 30-MAR-99  
**16447:** [How to Get the Most Out of DBDEMOS](#) 30-MAR-99  
**16452:** [Exposing the OnClick of the DBGrid control](#) 30-MAR-99  
**16457:** [Getting runtime properties at runtime](#) 16-JUL-98  
**16462:** [Search and replace in strings: a task made easy](#) 16-JUL-98  
**16467:** [Dynamically creating a TTable & fields at runtime](#) 31-MAR-99  
**16472:** [Activation and Use of the CPUWindow in the IDE](#) 30-MAR-99

**16477:** Passing Multidimensional Arrays as Parameters 01-APR-99  
**16482:** Implementing Drag and Drop Functionality 16-JUL-98  
**16487:** Trapping Windows Messages in Delphi 16-JUL-98  
**16497:** Direct Commands to Printer - Passthrough/Escape 22-APR-99  
**16502:** Creating a form based on a string 16-JUL-98  
**16507:** Finding the color depth of a canvas 16-JUL-98  
**16512:** Using FindFirst and the WIN\_32\_FIND\_DATA structure 16-JUL-98  
**16517:** Setting the pixels per inch property of TPrinter 01-APR-99  
**16522:** How to use a string table resource 16-JUL-98  
**16532:** TRichEdit Printing in Delphi 2 & Windows NT 4.0 30-MAR-99  
**16537:** How to use a user defined resource. 16-JUL-98  
**16542:** A better way to do pointer arithmetic 16-JUL-98  
**16547:** Assuring Proper Font Scaling When Printing 16-JUL-98  
**16552:** Delphi 3 file types with descriptions 22-APR-99  
**16557:** Moving Projects Between Machines or Directories 16-JUL-98  
**16562:** An example of drag and drop between DBGrids 30-MAR-99  
**16567:** Looping Through the Controls and Components Arrays 16-JUL-98  
**16572:** Adding ODBC Drivers in Delphi with the BDE Administrator 22-APR-99  
**16577:** Delphi/400: Activating your License Key 22-APR-99  
**16582:** Displaying System Resources in Win 9x and NT 4 31-MAR-99  
**16587:** Edit Controls that Align Under NT 4 01-APR-99  
**16592:** How do I map a network drive in Windows NT or '95? 05-OCT-99  
**16597:** Adding shortcuts to Win95/WinNT4 Desktop/StartMenu 01-APR-99  
**16602:** Minimizing Application When a Form Minimizes 16-JUL-98  
**16607:** Graying Out Enabled/Disabled Data Aware Controls 31-MAR-99  
**16612:** Exposing a multi string object in COM 31-MAR-99  
**16617:** Getting Version Information From Your Program 16-JUL-98  
**16627:** Adding an IPersistPropertyBag to Active Controls 05-AUG-99  
**16632:** Exposing a Font Property in an Automation Server 31-MAR-99  
**16637:** Resolving Focus to a DBGrid After Focus Transfers 01-APR-99  
**16642:** Importing or "Wrapping" DLL Function Calls 16-JUL-98  
**16647:** Implementing TCollection 02-APR-99  
**16652:** Delphi Titles from Leading Book Publishers 30-MAR-99  
**16657:** Pascal Titles from Leading Publishers 16-JUL-98  
**17596:** Calling a Delphi DLL from your BCB Code. 02-JUL-98  
**17624:** Assembly linked with Pascal - can't trace through 02-JUL-98  
**17795:** Removing the Vertical Scroll Bar from a TDBGrid 01-SEP-99  
**17801:** Writing a Buffer Disk 01-SEP-99  
**17805:** TI2805C: Showing Deleted Records in a dBase Table 01-SEP-99  
**18045:** Searching on a secondary index in Engine 2.0 02-JUL-98

**18107:** FAQ document 5 02-JUL-98

**18289:** Comparing connections to an AS/400 between Delphi/400 and ODBC 01-APR-99

**19020:** TURBO PROFILER SOURCE CODE VIEWING 02-JUL-98

**19021:** TP 5.0/TASM - ARG DIRECTIVE SYMBOLIC REFERENCES 02-JUL-98

**19024:** TASM - UNDEFINED SYMBOL WITH TURBO C 02-JUL-98

**19025:** TASM - LOAD AND EXECUTE OVERLAY 02-JUL-98

**19026:** TD - AT&T 6300 clock speeds up after TD is loaded. 02-JUL-98

**19027:** TD - 8087 MATH COPROCESSOR NOT RECOGNIZED 02-JUL-98

**19028:** TD - EMS VERSIONS TD 02-JUL-98

**19029:** TA - FAR JUMP OR FAR CALL TO AN ADDRESS 02-JUL-98

**19030:** TASM - EQU in masm mode vs. in ideal mode 02-JUL-98

**19031:** TLINK or TC 02-JUL-98

**19032:** TA - PROGRAM WILL ASSEMBLE UNDER MASM & NOT TASM 02-JUL-98

**19033:** TP 5.0/TD - ORDER OF ARG DIRECTIVE OPPPERANDS 02-JUL-98

**19034:** TD METAWARE HIGH C TDCONVRT 02-JUL-98

**19035:** TP 5.0/TASM - RETURN FAILED CALLING ASSEMBLY ROUT 02-JUL-98

**19036:** TA - SIZE OF STRUC IN IDEAL MODE 02-JUL-98

**19037:** TA 1.0 - TED.ASM; FORWARD SEGMENT REFERENCE PROBL 02-JUL-98

**19038:** TA - HOW TO WRITE A TSR 02-JUL-98

**19039:** TA - TYPE in IDEAL MODE 02-JUL-98

**19040:** TA - CREATING A .COM FILE TLINK /V 02-JUL-98

**19041:** TA - DEBUG .COM FILE 02-JUL-98

**19042:** TA - NONFUNCTIONAL CONDITIONAL BREAKPOINT 02-JUL-98

**19043:** TD - LOCATION POINTER SOURCE CODE 02-JUL-98

**19044:** TD - NOT ENOUGH ENVIRONMENT SPACE 02-JUL-98

**19045:** TD - HANGING ON NMI 02-JUL-98

**19046:** TD - HANGING ON IBM PC 02-JUL-98

**19047:** TD - 43/50 LINE MODE VIDEO 7 VEGA DELUX 02-JUL-98

**19048:** TASM , TLINK & use with EPROMS & intel hex files 02-JUL-98

**19049:** TD - ROMABLE CODE 02-JUL-98

**19050:** TD - TDREMOTE TIMEOUT 02-JUL-98

**19051:** TD - EXIT QUIT NULL POINTER ASSIGNMENT 02-JUL-98

**19052:** TD - CPU VIEWER 02-JUL-98

**19053:** TD - HARDWARE DEBUGGERS 02-JUL-98

**19054:** TD - TD386 CANT LOAD VIRTUAL MODE 02-JUL-98

**19055:** TD - TD and HP Vectra, unexpected interrupt 06 02-JUL-98

**19056:** TD - TD386 unexpected interrupt 06 02-JUL-98

**19057:** TD - UNEXPECTED INTERRUPT 0D 02-JUL-98

**19058:** TD - TD386 UNEXPECTED INTERRUPT 0A 02-JUL-98

**19059:** TD - ILLEGAL EXPRESSION HEX NUMBERS 02-JUL-98

**19060:** [TLINK - TLINK /V](#) 02-JUL-98

**19061:** [TD - NULL POINTER ASSIGNMENT.](#) 02-JUL-98

**19062:** [TASM 1.0 - ARG DIRECTIVE SYMBOLIC REFERENCES - THO](#) 02-JUL-98

**19063:** [TD386 and DOS 5.0 Compatibility](#) 02-JUL-98

**19064:** [IMPORT.PAS name fields empty](#) 02-JUL-98

**19301:** [How to do a "sounds like" query in Interbase](#) 23-FEB-99

**19335:** [How do I create a simple MIDAS Client Server application?](#) 16-MAR-99

**19523:** [Working with cookies](#) 01-APR-99

**19524:** [Handling Winsock errors](#) 22-APR-99

**19528:** [Converting a BMP to a JPEG and vice-versa under Delphi 3](#) 31-MAR-99

**19531:** [Installing JPEG Components that Ship with Delphi 3](#) 01-APR-99

**19532:** [Extracting a JPEG Resource from an EXE](#) 31-MAR-99

**19534:** [Creating a 32-Bit Screen Saver in 32-bit Delphi](#) 22-APR-99

**19538:** [Surfacing an Event on an ActiveForm to a Container](#) 02-APR-99

**19540:** [Drag and Drop Selected Text in between Memo Components](#) 01-APR-99

**19546:** [Description of Multi-Byte Character Support in Delphi, BDE](#) 22-APR-99

**19548:** [Creating Lookup Fields at Runtime](#) 22-APR-99

**19549:** [ShortCut Keys For Delphi](#) 22-APR-99

**19550:** [Installing and Using Delphi 3 Under NT4](#) 01-APR-99

**19555:** [Why a Network Install of Delphi 3 Isn't Supported](#) 01-APR-99

**19611:** [How to debug ISAPI dll's on Windows 98 with Personal Web Server](#) 21-JUN-99

**19646:** [How to register my program's file type with the Window's registry.](#) 21-JUN-99

**20193:** [Changing BDE Driver Settings Programmatically](#) 01-DEC-99

**20245:** [MS SQL Server 7 Stored Procedure PRINT Statements with ADO](#) 06-DEC-99

**20377:** [Maintaining State in Web Applications](#) 17-DEC-99

**20378:** [Streaming images from Web Applications](#) 17-DEC-99

**20385:** [How can I install Delphi on a Micron computer?](#) 20-DEC-99

**20450:** [Adding TextCompletion to a TComboBox](#) 30-DEC-99

**20550:** [Redirecting HTTP requests to other locations and custom response codes](#) 17-JAN-00

**20901:** [Debugging an ISAPI DLL with Delphi under NT 4.0 for IIS version 4](#) 29-FEB-00

**21772:** [Deploying a Delphi/VisiBroker Application](#) 19-APR-00

**21967:** [Oracle 7 and Muliple Threads.](#) 12-MAY-00

**22507:** [Problems comparing floating point numbers.](#) 21-JUL-00

**23024:** [How to debug ISAPI DLL's on Windows 2000 with IIS5](#) 01-SEP-00

**25920:** [Writing Clean and Safe UDFs in Delphi](#) 26-SEP-00

**26048:** [Problems using dcomcnfg to configure ActiveX library objects \(DLLs\).](#) 13-OCT-00

**26277:** [Creating transparent windows](#) 14-NOV-00

**26418:** [Creating a Web Search Wizard using the OpenTools API.](#) 12-DEC-00

**26531:** [Automating Internet Explorer to POST Form Data](#) 03-JAN-01

**26574:** [Automating Internet Explorer to Find All Links on a Web Page](#) 08-JAN-01

**26652:** Random Images Screen Saver - a complete screen saver example 17-JAN-01  
**26693:** Using blocking sockets inside of a thread, and simulating HTTP requests 24-JAN-01  
**26736:** I want to be able to use multiple aliases with a single TDatabase component. 30-JAN-01  
**26876:** Use the Windows API to generate a list of available Network Resources. 12-FEB-01  
**26977:** Finding the Windows task bar using the SHAppBarMessage API call 28-FEB-01  
**27126:** Writing COM Automation Events 26-MAR-01  
**27132:** Creating a file browser for Linux using Kylix 26-MAR-01  
**27182:** CLX styles demo for Kylix 03-APR-01  
**27363:** Using queued components in Delphi 30-MAY-01  
**27366:** Calculating Business Days on Oracle 30-MAY-01  
**27462:** Streaming Bitmaps and other Binary Data to BLOB Fields 28-JUN-01  
**27508:** How can I change a folders Icon in Delphi? 10-JUL-01  
**27879:** Using a TreeView in conjunction with a Master Detail relationship. 15-OCT-01  
**27953:** Retrieving a list of installed Applications on Windows 30-OCT-01  
**28258:** Working with TMultiReadExclusiveWriteSynchronizer 17-JAN-02  
**28325:** Encrypting and Decrypting strings in Delphi. 31-JAN-02  
**28416:** How to debug a COM+ component in Delphi. 19-FEB-02  
**28452:** How to control user input into a TEdit component. 27-FEB-02  
**28502:** How to use a parameterized query with a Remote Data Service. 08-MAR-02  
**28614:** Configuring NSAPI servers to handle ISAPI DLL requests 12-APR-02  
**28663:** How to Surface Errors in WebSnap and Still Retain Field Values. 09-MAY-02  
**28936:** How to change the default printer from Delphi. 31-JUL-02  
**28956:** How to programmatically open and close the CD-Rom drive. 31-JUL-02  
**29028:** Revamp Your Main Menu with Delphi 7 28-AUG-02

## Article #10531: BDE and the Year 2000

- Product: *BDE*
  - Version: *All*
  - Platform: *Windows 95, Windows NT, Windows 3.1*
- 

This document describes how the Borland Database Engine (BDE) operates in conjunction with the year 2000. It discusses all native drivers, SQL Links and ODBC.

What is Year 2000 compliance? Simply put, it is the ability to store and retrieve dates in any 4 digit century (the year 10000 will be a real problem!) and handle leap year days correctly. Given this minimum specification, the BDE clearly meets it. At this time, no further specification has been agreed upon by any standards committee. (As you read this document, you'll see that various vendors handle 2 digit years differently.) However, the March 1998 issue of Computer Magazine has proposed the following standard:

*Technology, including but not limited to, information technology, embedded systems, or any other electro-mechanical or processor based system, when used in accordance with its associated documentation, is capable of accurately processing, providing, providing, and/or receiving date data from, into, and between the twentieth and twenty-first centuries, and the years 1999 and 2000, including leap year calculations; provided that all other technology properly exchanges date data with it.*

All drivers handle leap year days correctly. Note that the year 2000 has a leap year day. This document does not discuss leap year days further because all drivers handle them correctly.

The one way you can insure that your programs that use the BDE are Year 2000 compliant is to always use 4 digit years. Given this one proviso, you can ignore the rest of this document. The rest of this document considers what happens when you enter 2 digit years for each driver.

With the exception of Paradox, dBASE, and FoxPro tables and when using QBE, the BDE passes dates as entered to the Client Software. No century is assumed. It is the responsibility of the connecting client to interpret the century. Below is how each driver type is known to respond to 2 digit years. Most information has been pulled from the associated vendors web site.

### **QBE**

If YEARBIASED in BDE Administrator is TRUE, all 2-digit date queries against all databases have 1900 added to them. If YEARBIASED in BDE Administrator is FALSE, all 2-digit date queries are assumed to be in the first century (Year 0001 to 0099). Because of this, all dates not in the 20th or 1st century must be expressed as 4-digit years.

### **Paradox, dBASE, FoxPro, Local SQL**

Dates entered into tables assume the current century of the local CPU.

For BDE versions less than 5.0, Local SQL against Paradox, dBASE and FoxPro tables assume the current century to be 1900. For Local SQL to use a century other than 1900 using BDE prior to 5.0 a 4-digit year must be used.

For BDE version 5.0 and later, local SQL against Paradox, dBASE and FoxPro tables assumes dates with 2 digit years to be in 1900 for years  $\geq 50$  and years to be in 2000 for years  $< 50$ . For Local SQL to use a century other than these defaults with BDE 5.0, a 4 digit year must be used.

## **Access 95**

Microsoft

Parsing on date entry. OLEAUT32. 2.20.4049 or greater is required in order for Access 95 to correctly recognize 2-digit date shortcuts as falling within a window between 1930 and 2029. Without that file update, Access assumes that all shortcut dates fall in the current century. See Microsoft for additional information.

## **Access 97**

2-digit shortcut handling:

Microsoft Access 97 interprets dates entered with a 2-digit shortcut to mean the 21st century in the following way.

1/1/00 through 12/31/29 are interpreted as 1/1/2000 through 12/31/2029

1/1/30 through 12/31/99 are interpreted as 1/1/1930 through 12/31/1999 See also [Microsoft](#)

## **Interbase**

From [InterBase](#)

InterBase provides a reasonable default interpretation of two-digit years. The InterBase server interprets a two-digit year as the nearest year within a 50-year range. For example, 2/28/48 is interpreted as 1948, whereas 2/28/47 is 2047.

## **Oracle**

From [Oracle](#)

If the current year is in the second half of the century (50 - 99)

- and a two-digit year between '00' and '49' is entered: This will be stored as a next century year. E.g. '02' entered in 1996 will be stored as '2002'.
- and a two-digit year between '50' and '99' is entered: This will be stored as a current century year. E.g.



'97' entered in 1996 will be stored as '1997'.

If the current year is in the first half of the century (00 - 49) and a two-digit year between '00' and '49' is entered:

- and a two-digit year between '00' and '49' is entered: This will be stored as a current century year. E.g. '02' entered in 2001 will be stored as '2002'.
- and a two-digit year between '50' and '99' is entered: This will be stored as a previous century year. E.g. '97' entered in 2001 will be stored as '1997'.

## **Sybase**

From Sybase

Implied century rule: <50 = 2000, >49 = 1900

Compliance issues description or comment relating to current product version:

## **Informix**

From Informix

Customers should be aware that Informix products often depend on data from other parts of the system, and Informix year 2000 compliance is not effective unless all of the customer's hardware, operating system, other software, and firmware correctly interpret and/or translate date data into a four digit year date and, when used in combination with the Informix products, properly exchange date data with it. It is the customer's responsibility to understand the manner in which applications are written to accept a date value and the habits of users entering dates into these applications. Generally, year 2000 issues associated with these problems must be addressed at the application level.

As described below, Informix has added a "DBCENTURY environment variable" feature to the Informix products listed in the table below which may help users provide year 2000 compliance in their existing applications without re-coding. Customers are encouraged to test their applications for the compliance described above. Customer applications, built using Informix products, which use only two-digit date storage may benefit from the DBCENTURY feature described below.

## **DBCENTURY Feature to Aid in the Year 2000 Solution**

The DBCENTURY environment variable was implemented to provide customers with an API-level year 2000 solution eliminating the requirement of the customer to modify application source code. The DBCENTURY environment variable is an indicator which controls how Informix products set the default century value for Date and Datetime datatypes when the user has not entered the full four digits for a year through a custom developed application. No changes are required to the application code whatsoever to take advantage of this feature. The only action to be taken by the application user is that DBCENTURY must be added to the login profile of each user on the client system, and is thereafter automatically applied to all Informix applications associated with that

login id.

For those versions of Informix products for which the DBCENTURY environment variable is supported (see table 2 below), if the DBCENTURY variable is not set in the user's profile, the Informix product will default the century value to the present century, which it obtains from the computer's system clock. If the DBCENTURY variable is set, it can be set to select a default date that is in the past, in the future, or closest to the computer's system clock. Informix anticipates that the latest setting will be the most generally applicable for the next several decades. For example, if Current date is "06/21/1998" and user enters two dates, "10/25/95" and "08/01/03", dates stored is shown below:

**Table 1: Example of DBCENTURY settings**

<b>DBCENTURY setting</b>	<b>Dates Stored</b>
P	10/25/1995, 08/01/1903
R	10/25/1995, 08/01/1903
C	10/25/1995, 08/01/2003
F	10/25/2095, 08/01/2003

If the DBCENTURY variable is not set in the user's login profile, the dates would default to "10/25/1995" and "08/01/1903", the present century. If the user enters the full four digits for the years into a four-digit year field in the application, the year will be accepted as entered, regardless of the DBCENTURY variable setting.

If the user enters into an application only two digits for the year, and the default century set through DBCENTURY is making an incorrect assumption on the range of dates to be entered, incorrect data will be stored and returned. For example, if a user enters '03 as shorthand for 1903, and the DBCENTURY variable is set to default to the closest date, the Informix product would store 2003. If the application is expected to manage dates which are more than 50 years in the past or in the future, in order to use DBCENTURY to best suit the needs of the application, please refer to the Informix documentation regarding the use of the various settings of DBCENTURY.

## **MSSQL**

From Microsoft

SQL Server allows users to specify only the last 2 digits of the year. However the year is still stored the same as if 4 digits were specified. The "two-digit year cutoff" configuration option defines how SQL Server 7.0 expands a 2-digit year to four digits. The option is specified as a 4-digit year, the default is 2049. If only two digits are specified for a year, and they are less than or equal to the last two digits of the 2-digit year cutoff option, the date is stored as being in the same century as the option. If the specified two digits are greater than the last two digits of the 2-digit year cutoff option, the date is stored as being in the century before the one specified in the option. For example, if the 2-digit year cutoff option is left at its default of 2049, the value 49 is stored as 2049 and 50 is

stored as 1950. If the 2-digit year cutoff option is set to 2029, the value 29 is stored as 2029 and 30 is stored as 1930.

## **DB2**

IBM offers a large amount of information on the year 2000. [http://www.s390.ibm.com/vse/vsehtmls/vn13sql.htm#Header\\_3](http://www.s390.ibm.com/vse/vsehtmls/vn13sql.htm#Header_3) indicates that two-digit dates that have a year less than 43 are assumed to be in the 21st century (20xx). Otherwise, they are assumed to be in the 20th century (19xx).

Please see the IBMs year 2000 web site at [IBM](#) for additional information.

## **ODBC**

Dates entered when connected via an ODBC driver are sent to the ODBC driver exactly as entered. If you enter a 4-digit year, a 4-digit year is sent. If you enter a 2-digit year, a 2-digit year is sent. The Server that the ODBC driver connects does interpretation of the date. If your ODBC driver connects to one the SQL Links drivers listed above, dates should be interpreted the same as the SQL Link driver. Otherwise, contact the manufacturer of the database that you connect to determine how 2-digits years are interpreted.

[Documentation from Microsoft on ODBC and year 2000](#)

Last Modified: 18-NOV-99

## Article #15005: Understanding the PARADOX.NET File

Technical Information Database

TI5B.txt Understanding the PARADOX.NET File

Product :BDE

### Description:

The net file regulates access to tables. Table access is enforced through the use of lock files, which are written out to the directories containing Paradox tables. A lock file points to a particular net file, which has exclusive control over the table. This means that any user wanting access to the table must use the net file that controls the table. In practice, this means that when your BDE program shares tables with Paradox users, you must initialize BDE with a netpath set to the Paradox net path. In the case of two Engine programs sharing tables, the net paths must be identical.

The message "Multiple Paradox net file found" indicates that these rules have not been followed. If you are certain that all current users have the same net path, this message usually indicates that an old lock file exists that points to a different net file. Old lock files can be deleted if care is taken to ensure that no one is currently using them. Deleting active lock files can produce unpredictable results and could cause loss of data.

**DISCLAIMER:** You have the right to use this technical information subject to the terms of the No-Nonsense License Statement that you received with the Borland product to which this information pertains.

Last Modified: 11-FEB-00

## Article #15007: Skip "Include error messages?" prompt

Technical Information Database

TI7D.txt - Skip "Include error messages?" prompt

Category :Turbo Pascal  
Platform :DOS  
Product :

### Description:

This patch automatically responds to the "Include Error Messages?" prompt and proceeds to either load or exclude the error messages (whatever is specified by the patch you apply). The message still appears, but does not wait for an answer.

ERROR MESSAGES NOTE: The new version of TINST lets you designate the path to the error messages in the event they are stored in a directory other than the one active at the time Turbo Pascal is loaded. Patches for three different implementations of

Use the DOS utility, DEBUG, to apply the following patch to a copy of your original .COM file. Copy DEBUG.COM into the directory (or onto the diskette) where your working copy of information exactly as it appears. (Conclude each line by pressing , to quit. Check your version numbers and, if correct, try again.

PC-DOS, TURBO.COM - "YES" - MESSAGES INCLUDED

YOU TYPE		THIS APPEARS
debug TURBO.COM	-	
e29CE		xxxx:29CE E8.
E9 1C 00	-	
e29ED		xxxx:29ED E8.
E9 05 00	-	
w		writing xxxx bytes
q		DOS prompt

PC-DOS, TURBO.COM - "NO - MESSAGES NOT INCLUDED

YOU TYPE

THIS APPEARS

debug TURBO.COM

-

e29CE

xxxx:29CE E8.

E9 1C 00

-

e29ED

xxxx:29ED E8.

E9 08 00

-

w

writing xxxx bytes

q

DOS prompt

Reference:

3/30/99 3:49:50 PM

Last Modified: 01-SEP-99

## Article #15008: What to check when you cannot get an all users int

Technical Notes Database

TN8B.txt      What to check when you cannot get an all users int  
Category      :InterBase  
Platform      :All  
Product       :Borland SQL Lin 4.5

Description:

Keywords: IB, interface, connect

Question: I have three users trying to access data on my InterBase server via Paradox. I can get two users in just fine, but the system refuses to let me enter a third one.

Answer: There are a number of things that can cause this. First, can you get all three users into Paradox? If you are not using runtime, you need a valid license of Paradox for each user. If you cannot get all users into Paradox, first check to make sure that you have adequate licenses of Paradox for Windows. If you have adequate licenses of Paradox, make sure that all users are pointed to the same pdoxusrs.net file. If you suspect the pdoxusrs.net file may be the culprit, delete any pdoxusrs.net and the paradox.net files that you may find, when everyone is out of Paradox. (This is just in case you may have had a damaged pdoxusrs.net.) If that does not solve the problem, we will look to possible causes on the server side.

On the server side, make sure that you have valid UNIX logins for

each user who will be trying to access the data on the InterBase server. You can use the UNIX command "uname" to do this. eg. uname -x

Next, run gsec with the -display option to find out if all three users have access to the remote database. If you find that one of your users does not have access to the remote database, you will need to do an add to put that person into the remote servers list of users.

The final thing to check is the amount of RAM on the UNIX machine.

SCO UNIX requires a minimum of 8 meg of RAM. It also requires an additional .5 Meg of RAM for each user.

GRUBTI01 - 03/10/94

Reference:

7/15/98 3:24:41 PM

Last Modified: 01-SEP-99



## Article #15017: Interrupt 3 patch

Technical Information Database

TI17D.txt - Interrupt 3 patch

Category :Turbo Pascal  
Platform :DOS  
Product :

### Description:

Turbo Pascal redefines the interrupt 3 vector to its own interrupt service routine used for handling Ctrl-C. However, the vector is permanently initialized to that interrupt routine even though the "U-" directive is specified.

This patch will permanently keep the Ctrl-C vector from being initialized, but will allow the use of DEBUG break points.

Use the DOS utility, DEBUG, to apply the following patch to a copy of your .COM file. Copy DEBUG.COM into the directory (or onto the diskette) where your working copy of .COM is located. At the DOS prompt, type the following information exactly as it

- Note:
1. DEBUG is not sensitive to upper and lower case. All addresses are listed in upper case for ease of readability.
  2. While in DEBUG, the prompt appears as a dash (-).
  3. If you do not receive the appropriate response, press "q", then , to quit. Check your version numbers and, if correct, try again.

For IBM versions of Turbo Pascal:

YOU TYPE	THIS APPEARS
DEBUG TURBO.COM	-
e0BE2	XXXX:0EB2 E8.
90	DC.

```
90          FD.
90          -
w           writing xxxx bytes
q           DOS prompt
```

For Generic versions of Turbo Pascal:

YOU TYPE	THIS APPEARS
DEBUG TURBO.COM	-
e0922	xxxx:0922 E8.
90          D1.	
90          FD.	
90          -	
w           writing xxxx bytes	
q           DOS prompt	

Reference:

3/30/99 3:49:21 PM

Last Modified: 01-SEP-99

## Article #15019: Conflicts between Delphi and Netcruiser

Technical Notes Database

TN19D.txt      Conflicts between Delphi and Netcruiser  
Category      :General Programming  
Platform       :All  
Product        :Delphi    All

### Description:

Users of Netcom's Netcruiser software have complained that, after they install Delphi, Netcruiser software exits with a GPF at startup.

It seems that Delphi installs a version of gauge.vbx which is incompatible with that used by Netcruiser. Delphi itself does not need to use gauge.vbx (it is only used by the BIGauge component), so the best solution is to remove the BIGauge component from the VBX component field, delete the offending copy of gauge.vbx, and then reinstall Netcruiser.

### Reference:

7/16/98 4:35:28 PM

Last Modified: 01-SEP-99

## Article #15026: MS-DOS interrupt 24 trap routine

Technical Information Database

TI26D.txt - MS-DOS interrupt 24 trap routine

Category :Turbo Pascal  
Platform :DOS  
Product :

### Description:

The following example routines are public domain programs that have been uploaded to our Forum on CompuServe. As a courtesy to our users that do not have immediate access to CompuServe, Technical Support distributes these routines free of charge.

However, because these routines are public domain programs, not developed by Borland International, we are unable to provide any technical support or assistance using these routines. If you need assistance using these routines, or are experiencing difficu

Thanks to Marshall Brain for the original code for this routine.

These routines provide a method for Turbo Pascal programs to trap MS-DOS interrupt 24. INT 24 is called by DOS when a "critical error" occurs, and it normally prints the familiar "Abort, Retry, Ignore?" message.

With the INT 24 handler installed, errors of this type will be passed on to Turbo Pascal as an error. If I/O checking is on, this will cause a program crash. If I/O checking is off, IOResult will return an error code. The global variable INT24Err will b

In most cases, INT24Result should be used, because INT24Err must be set back to false, and DOS sometimes restores its normal INT 24 handler after an error.

-----  
\*\*Note: Turbo's normal IOResult codes (and Turbo Access error codes)

for MS-DOS DO NOT correspond to I/O error numbers given in Appendix I of the Turbo Pascal manual, or error codes given in the I/O error nn, PC=aaaa/Program aborted message. Here is

IOResult	Turbo error	
00 (0)	00 (0)	none
01 (1)	90 (144)	record length mismatch
02 (2)	01 (1)	file does not exist
03 (3)	F1 (241)	directory is full
04 (4)	FF (255)	file disappeared
05 (5)	02 (2)	file not open for input
06 (6)	03 (3)	file not open for output
07 (7)	99 (153)	unexpected end of file
08 (8)	F0 (240)	disk write error
09 (9)	10 (16)	error in numeric format
0A (10)	99 (153)	unexpected end of file
0B (11)	F2 (242)	file size overflow
0C (12)	99 (153)	unexpected end of file
0D (13)	F0 (240)	disk write error
0E (14)	91 (145)	seek beyond end of file
0F (15)	04 (4)	file not open
10 (16)	20 (32)	operation not allowed on a logical device
11 (17)	21 (33)	not allowed in direct mode
12 (18)	22 (34)	assign to standard files is not allowed
90 (144)	90 (144)	record length mismatch

```
program CriticalError;
```

```
Const INT24Err      : Boolean=False;
      INT24ErrCode  : Byte=0;
      OldINT24      : Array [1..2] Of Integer=(0,0);
```

```
Var RegisterSet    : Record Case Integer Of
      1: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer);
      2: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
End;
```

```
Procedure INT24;
```

```
Const FCBFfuncs: Array [1..6] Of Byte=(14,15,21,22,27,28);
Begin
```

```
{ To understand this routine, you will need to read the
description of
```

```
Interrupt 24h in the DOS manual. It also helps to examine and
```



routine

like

handles

.2  
saved by

POP BX

POP CX

POP DX

POP SI

POP DI

MOV BP,SP

OR Byte Ptr [BP+0Ah],1

Flags

POP BP

POP DS

POP ES

IRET

regs.

set. }

End;

that called DOS, making it look

a simple DOS error. Turbo

the I/O error.

Pop the rest of the registers

the initial INT 21h.

Set the carry flag in the saved

reg.

Return to next instruction: all

restored, AL=error code, carry

Procedure INT24On; {Enable INT 24 trapping}

Begin

INT24Err:=False;

With RegisterSet Do

Begin

AX:=\$3524;

MsDos(RegisterSet);

If (OldINT24[1] Or OldINT24[2]) = 0 Then

Begin

OldINT24[1]:=ES;

OldINT24[2]:=BX;

End;

DS:=CSeg;

DX:=Ofs(INT24);

AX:=\$2524;

MsDos(RegisterSet);

End;

```

End;

Procedure INT24Off; {Disable INT 24 trapping. Should be done at the
                    end of the program, if you plan to be running
                    the program from within the Turbo compiler.}

Begin
  INT24Err:=False;
  If OldINT24[1]<>0 Then
    With RegisterSet Do
      Begin
        DS:=OldINT24[1];
        DX:=OldINT24[2];
        AX:=$2524;
        MsDos(RegisterSet);
      End;
    OldINT24[1]:=0;
    OldINT24[2]:=0;
  End;

Function INT24Result: Integer;
Var I : Integer;

Begin
  I:=IOResult;
  If INT24Err Then

  Begin
    I:=I+256*(INT24ErrCode+1);
    INT24On;
  End;

  INT24Result:=I;
End;

{ INT24Result returns all the regular Turbo IOResult codes if no
critical error has occurred. If a critical error, then the
following values are added to the error code from Turbo (each
is 256 times the INT24ErrorCode value returned by DOS):

256: Attempt to write on write protected disk
512: Unknown unit [internal dos error]
768: Drive not ready [drive door open or bad drive]
1024: Unknown command [internal dos error]

```



1280:	Data error (CRC)	[bad sector or drive]
1536:	Bad request structure length	[internal dos error]
1792:	Seek error	[bad disk or drive]
2048:	Unknown media type	[bad disk or drive]
2304:	Sector not found	[bad disk or drive]
2560:	Printer out of paper	[anything that the printer might signal]
2816:	Write fault	[character device not ready]
3072:	Read fault	[character device not ready]
3328:	General failure	[several meanings]

If you need the IOResult part, use

```
I:=INT24Result and 255; [masks out the INT 24 code]
```

For the INT 24 code, use

```
I:=INT24Result Shr8; [same as Div 256, except faster]
```

INT24Result clears both error codes, so you must assign it to a variable if you want to extract both codes:

```
J:=INT24Result;
```

```
Writeln('Turbo IOResult = ',J And 255);
```

```
Writeln('DOS INT 24 code = ',J Shr 8); }
```

```
{ Main program }
```

```
{ Run this with printer off (or no printer), and nothing in drive A }
```

```
Var F : File;
    I : Integer;
```

```
Procedure PrinterTest;
```

```
Begin
```

```
  WriteLn(LST,'test');
```

```
  I:=INT24Result;
```

```
  If I<>0 Then
```

```
    WriteLn('Printer error: ',I)
```

```
  Else
```

```
    WriteLn('Printer OK');
```

```
End;
```

```
Procedure FileTest;
```

```
Begin
```

```
  Assign(F,'A:FILE');
```

```
  {$I-}
```

```
  Reset(F);
```

```
  {$I+}
```

```
  I:=INT24Result;
```

```
If I<>0 Then
  WriteLn('Open failure on A:FILE : INT24Result=',I)
Else
begin
  WriteLn('A:FILE exists');
  Close(F);
end;
End;
```

```
Begin
  INT24On;
  PrinterTest;
  FileTest;
  PrinterTest;
  INT24Off;
  FileTest;
  PrinterTest;
End.
```

Reference:

3/30/99 3:11:26 PM

Last Modified: 01-SEP-99

## Article #15035: Kaypro - Ever-spinning disk drive

Technical Information Database

TI35D.txt - Kaypro - Ever-spinning disk drive

Category :Turbo Pascal  
Platform :DOS  
Product :

### Description:

Place this procedure into your program and call it when you would like your disk drives to stop spinning.

```
{ This works for Kaypro 2, 4, 1984 or later }  
procedure MotorOff;  
const  
    system = $14;                                {Kaypro System port}  
  
begin  
    port[system] :=port[system] and $EF;        {Turn off motor}  
end;
```

```
{ This works for the original Kaypro 2 and 4 }  
  
procedure MotorOff;  
const  
    system = $1C;  
begin  
    port[system] := port[system] or $140;  
end;
```

Reference:

3/30/99 3:14:33 PM

Last Modified: 01-SEP-99

## Article #15037: Delphi End User CD-ROM can not be mounted by Novel

Technical Notes Database

TN37D.txt      Delphi End User CD-ROM can not be mounted by Novel  
Category      :General Programming  
Platform       :All  
Product       :Delphi    All

### Description:

Delphi End User can not be mounted on a Novell 3.11 server running the cdnet NLM. Novell states in their documentation that "not all CD's will be mountable." Currently there seems to be no work around for this problem.

### SOLUTION:

The only solution is to use a local CD-ROM drive and copy the files to the network hard-disk.

### POSSIBLE FIXES:

If the customer is using NOVELL 3.12 or 4.01 server, then there is a patch available from Novell for CDNET.nlm, it is not stated anywhere that this will fix the problem, but it might work. Novell is currently working with Borland on this problem.

### Reference:

7/16/98 4:35:28 PM

Last Modified: 01-SEP-99

## Article #15044: Direct Video Accessing

Technical Information Database

TI44D.txt - Direct Video Accessing

Category :Turbo Pascal  
Platform :DOS  
Product :

Description:

Using Mem and MemW you may address any location on the screen and change its value. A possible use might be creating a memory mapped text editor. The following is an example of using Mem to address the text screen one byte at a time. (Exchange \$B800 for \$B000 if you have a monochrome screen.)

```
program RAM2;
```

```
Var
```

```
  j, i : integer;  
  ch : char;
```

```
begin
```

```
  j := 0;  
  for i := 1 to 2048 do  
  begin  
    Mem[$B800:j] := $80;           {Accessing video RAM  
    Mem[$B800:j + 1] := $99       a byte at a time}  
    j := j + 2  
  end;
```

```
end.
```

When accessing RAM addressing a word at a time using MemW, you must be aware of the following:

1. In text mode, when you enter a word value, it is broken down into two bytes which are placed on the screen. Consequently, word values must equal the sum of two bytes that you want sent consecutively to the screen.
2. You must first assign the low byte, then the high byte of

the word.

3. You must increase the offset by two to access the next word in memory.

```
MemArray
program RAM1;
```

```
Var
  j, i : integer;
begin
  j := 0;
  for i := 1 to 2048 do
  begin
    MemW[$B800:j] := $9908;           { Accessing video RAM a word
                                       at a time }
    j := j + 2;
  end;
end.
```

The following program demonstrates how to write directly to screen memory and save and restore text screens. These routines may cause snow on some monitors. There are routines in the Editor Toolbox to read and write to the screen without snow, but they are much harder to use and less intuitive than these ones.

```
program ScreenTest;
type
  { Row, Column }
  ScreenType = array [1..25, 1..80] of record
    Character : Char;
    Attribute : Byte;
  end;

  AnyString = string [80];    { Generic string type }
```

{ The attribute byte of this data structure is formatted as follows:

Bits 0-3 hold the foreground color, bits 4-6 hold the background color, and bit 7 indicates whether blinking is on or off.

Bit	0	1	2	3	4	5	6	7
Color Part	FG	FG	FG	FG	BG	BG	BG	BL
Color Part Value	1	2	4	8	1	2	4	1
Bit Value (Decimal)	1	2	4	8	16	32	64	128
Bit Value (Hex)	\$01	\$02	\$04	\$08	\$10	\$20	\$40	\$80 }

```

var
  ColorScreen : ScreenType absolute $B800:0;
                { Use for color screen }
  MonoScreen : ScreenType absolute $B000:0;
                { Use for monochrome screen }
  SavedScreen : ScreenType;
                { Buffer to save the screen }

function MakeAttribute(Foreground, Background : Byte;
                      Blink : Boolean) : Byte;

{ Creates an attribute byte from a combination of foreground,
background, and blinking attributes }

begin
  MakeAttribute := Foreground + (Background shl 4) +
                    (Ord(Blink) shl 7);
end; { MakeAttribute }

procedure WriteString(S : AnyString; var Screen : ScreenType;
                    Col, Row, Attrib : Byte);

{ Writes a string into screen memory starting at a particular row and
column, using Attrib as the color attribute. }

var
  Counter : Byte;

begin
  for Counter := 1 to Length(S) do
  begin
    Screen[Row, Pred(Col + Counter)].Character := S[Counter];
    Screen[Row, Pred(Col + Counter)].Attribute := Attrib;
  end;
end; { WriteString }

begin
  SavedScreen := ColorScreen;      { Save the current screen }
  Window(25, 5, 55, 15);          { Create the new window }
  GotoXY(1, 1);                   { Move inside the new window }
  TextColor(Yellow);              { Change the current colors }
  TextBackground(Red);
  ClrScr;

```

```
        { Clear the window and fill it with color }
WriteString('  Press Return to continue...', ColorScreen, 25, 5,
MakeAttribute(Yellow, Red, False));
    { Write out prompt - note that WriteString uses the
      absolute row and column, not the row and column
      of the current window }
Read;                                { Wait for return to be pressed }
ColorScreen := SavedScreen;         { Restore old screen }
end.
```

Reference:

3/30/99 3:53:27 PM

Last Modified: 01-SEP-99



## Article #15053: Modifying the aspect ratio in the graph unit

Technical Information Database

TI53D.txt - Modifying the aspect ratio in the graph unit

Category :Turbo Pascal  
Platform :All  
Product :

Description:

```
{*****  
The .BGI files supplied with the Graph unit declare an Aspect  
Ratio constant which is used to draw circles and arc segments.  
For some applications it is necessary to alter the aspect ratio.  
One such application results when trying to perform a screen dump  
to the printer. In quad-density graphics printing, it is useful  
to redefine the aspect ratio so the displayed results will be  
circular instead of ellipsoid.
```

The following program demonstrates how a programmer can alter the predefined Xasp element of the aspect ratio and to obtain the current value. Note that the Graph unit already has the procedure for GetAspectRatio.

```
***** }
```

```
program ChangeAspectRatio;
```

```
uses  
  Crt, Graph;
```

```
function GetAspectX : word;  
{ Return the current aspect ratio element Xasp }  
begin  
  GetAspectX := word(Ptr(Seg(GraphFreeMemPtr),  
                      Ofs(GraphFreeMemPtr)+277)^);  
end;
```

```
procedure SetAspectRatio(NewAspect : word);
```

```

{ Set the current aspect ratio element Xasp }
begin
    word(Ptr(Seg(GraphFreeMemPtr),
            Ofs(GraphFreeMemPtr)+277)^) := NewAspect;
end;

var
    GraphDriver, GraphMode : integer;
    Ch : char;
    S : String;

begin
    DirectVideo := false;
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, '');
    Str(GetAspectX, S);
    S := 'The initial aspect ratio was ' + S;
    OutTextXY(1,1, S);
    Circle((GetMaxX Div 2), (GetMaxY Div 4), (GetMaxY div 4));
    SetAspectRatio(5000);
    Circle((GetMaxX Div 2), 3*(GetMaxY Div 4), (GetMaxY div 4));
    Str(GetAspectX, S);
    S := 'Now the aspect ratio is ' + S;
    OutTextXY(1,(GetMaxY - 10), S);
    Ch := ReadKey;
    CloseGraph;
end.

```

Reference:

3/30/99 3:08:48 PM

Last Modified: 01-SEP-99

## Article #15055: 'Not initialized for accessing network files'

Technical Notes Database

TN55D.txt 'Not initialized for accessing network files'  
Category :General Programming  
Platform :All  
Product :Delphi All

### Description:

Installed Delphi, then cannot access Paradox network files from neither Paradox 5.0 for Windows nor dBASE for Windows. Gets 'Not initialized for accessing network files'. The solution is to copy the IDAPI.CFG file from the CDROM and rename the previous one. The problem is that Delphi uses a newer IDAPI that puts the NetDir in a new place. The install updates both places correctly, but when modifying the NetDir, it does not get saved to both places, only the new location.

### Reference:

7/16/98 4:35:28 PM

Last Modified: 01-SEP-99

## Article #15062: Database Toolbox suggestions/corrections

Technical Information Database

TI62D.txt - Database Toolbox suggestions/corrections

Category :Turbo Pascal  
Platform :All  
Product :

### Description:

This handout addresses some problems which have been reported by users of the Turbo Pascal DataBase ToolBox version 4.0 for use with Turbo Pascal 4.0.

### Correction

When accessing an Index File that allows duplicate keys and has greater than 32K records, the Index Key routines, FindKey and SearchKey will be unsuccessful in finding keys for records past 32K. By making the following changes to TACCESS.PAS, this proble

1. In the TaCompKeys procedure change the old code to look as follows:

```
begin
  if TaKeyStr(K1) = TaKeyStr(K2) then
    begin
      if not Dup or (DR1 = DR2) then
        TaCompKeys := 0
      else
        if DR1 > DR2 then
          TaCompKeys := 1
        else
          TaCompKeys := -1;
    end
  else
    if TaKeyStr(K1) > TaKeyStr(K2) then
      TaCompKeys := 1
    else
      TaCompKeys := -1;
end;
```

2. Save the changes to TACCESS.PAS. Call TaBuild for each .TYP file to create a new TACCESS.TPU file. Rebuild all Index Files.

### Recommended Suggestion

Users have reported unusual errors occurring in their DataBase ToolBox programs. Many of these errors can be solved by running TaBuild with the /W+ option to display the worksheet. The worksheet will show that 1,000 records will be the expected size of

For existing Data Files, where greater than 1,000 records are expected, it is recommended that you build a new TACCESS unit using the worksheet to set the maximum number of records. This requires that all Index Files be rebuilt.

Reference:

3/30/99 3:15:25 PM

Last Modified: 01-SEP-99

## Article #15064: INTERFACING MODULES WRITTEN IN MICROSOFT C

Technical Notes Database

TN64D.txt      INTERFACING MODULES WRITTEN IN MICROSOFT C  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. Can I link modules, written in Microsoft C, with Turbo Pascal programs?

A. Yes. M/S C .OBJ modules can be linked provided they do not use the C runtime library. The same limitations apply to M/S C modules as to Turbo C modules.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15071: Printing graphics to a HP Laserjet

Technical Information Database

TI71D.txt - Printing graphics to a HP Laserjet

Category :Printing  
Platform :All  
Product :

### Description:

```
{ The following example routines are public domain programs }  
{ that have been uploaded to our Forum on CompuServe. As a }  
{ courtesy to our users that do not have immediate access }  
{ to CompuServe, Technical Support distributes these }  
{ routines free of charge. }  
{ }  
{ However, because these routines are public domain programs, }  
{ not developed by Borland International, we are unable to }  
{ provide any Technical Support or assistance using these }  
{ routines. If you need assistance using these routines, }  
{ or are experiencing difficulties, we recommend that }  
{ you log onto CompuServe and request assistance from the }  
{ Forum members that developed the routines. }
```

### Unit HpCopy;

```
{ This unit is designed to dump graphics images produced }  
{ by Turbo Pascal 4.0's Graph Unit to a Hewlett-Packard }  
{ LaserJet printer. }  
{ }  
{ You MUST set the Aspect Ratio to 4950 before drawing a }  
{ circular object on the screen. The procedure to accomplish }  
{ this is also contained in this handout. }  
{ }  
{ If the Aspect Ratio is NOT set, the image produced by this }  
{ routine will appear ellipsoid. }
```

### Interface

Uses Crt, Dos, Graph;

Var

```
LST : Text;      { MUST Redefine because Turbo's Printer }
                  { Unit does not open LST with the File }
                  { Mode as BINARY. }
}
```

```
Procedure HPHardCopy;
{ Procedure to be called when the desired image is on the }
{ screen. }
}
```

```
Procedure SetAspectRatio( NewAspect : Word );
{ Procedure to be called to set the aspect ratio such that }
{ circular objects will appear correctly on the printout }
{ generated by HpHardCopy. NOTE that the image on the }
{ screen WILL appear ellipsoid. This is NORMAL! }
}
```

```
Function GetAspectX : Word;
{ This Function will return the currently set aspect ratio }
{ to allow the user to save the default ratio, set it to }
{ the ratio required by HpHardCopy (4950) and then restore }
{ it to the default value. }
}
```

## Implementation

```
Var
  Width, Height : Word; { Variables used to store settings }
  Vport : ViewPortType; { Used in the call GetViewSettings }
```

```
{ $F+ }
Function LSTNoFunction ( Var F : TextRec ) : Integer;
{ This function performs a NUL operation for a Reset or }
{ Rewrite on LST. }
}
```

```
Begin
  LSTNoFunction := 0;
End;
```

```
Function LSTOutPutToPrinter( Var F : TextRec ) : Integer;
{ LSTOutPutToPrinter sends the output to the Printer port }
{ number stored in the first byte of the UserData area of }
{ the Text Record. }
}
```

```
Var
  Regs : Registers;
  P : Word;
```

```
Begin
```



```

With F Do
Begin
  P := 0;
  Regs.AH := 16;
  While( P < BufPos ) and ( ( Regs.AH And 16 ) = 16 ) Do
  Begin
    Regs.AL := Ord( BufPtr^[P] );
    Regs.AH := 0;
    Regs.DX := UserData[1];
    Intr( $17, Regs );
    Inc( P );
  End;
  BufPos := 0;
End;
If( ( Regs.AH And 16 ) = 16 ) Then
  LstOutputToPrinter := 0          { No Error          }
Else
  If( ( Regs.AH And 32 ) = 32 ) Then
    LSTOutputToPrinter := 159      { Out of Paper    }
  Else
    LSTOutputToPrinter := 160;     { Device Write Fault }
End;
{$F-}

```

```

Procedure AssignLST( Port : Byte );
{ AssignLST both sets up the LST text file record as would }
{ ASSIGN, and initializes it as would a RESET.              }
{                                                            }
{ The parameter passed to this procedure corresponds to     }
{ DOS's LPT number. It is set to 1 by default, but can     }
{ easily be changed to any LPT number by changing the     }
{ parameter passed to this procedure in this unit's       }
{ initialization code.                                       }

```

```

Begin
  With TextRec( Lst ) Do
  Begin
    Handle := $FFF0;
    Mode := fmOutput;
    BufSize := SizeOf( Buffer );
    BufPtr :=
Buffer;
    BufPos := 0;
    OpenFunc :=
LSTNoFunction;

```

```

    InOutFunc :=
LSTOutPutToPrinter;
    FlushFunc :=
LSTOutPutToPrinter;
    CloseFunc :=
LSTOutPutToPrinter;
    UserData[1] := Port - 1;
End;
End;

Function GetAspectX : Word;

Begin
    GetAspectX := Word( Ptr( Seg( GraphFreeMemPtr ),
                            Ofs( GraphFreeMemPtr ) + 277 ) ^ );
End;

Procedure SetAspectRatio{ NewAspect : Word };

Begin
    Word( Ptr( Seg( GraphFreeMemPtr ),
                Ofs( GraphFreeMemPtr ) + 277 ) ^ ) := NewAspect;
End;

Procedure HPHardCopy;
{ Produces hard copy of a graph on Hewlett-Packard Laserjet }
{ printer By Joseph J. Hansen 9-15-87 }
{ Modified Extensively for compatibility with Version 4.0's }
{ Graph Unit By Gary Stoker }
{ }
{ Unlike Graphix Toolbox procedure HardCopy, this procedure }
{ has no parameters, though it could easily be rewritten to }
{ include resolution in dots per inch, starting column, }
{ inverse image, etc. }
{ }

Const DotsPerInch = '100';
    { 100 dots per inch gives full-screen }
    { width of 7.2 inches for Hercules card }
    { graphs, 6.4 inches for IBM color card }
    { and 6.4 inches for EGA card. Other }
    { allowable values are 75, 150, and 300. }
    { 75 dots per inch will produce a }
    { larger full-screen graph which may be }
    { too large to fit on an 8 1/2 inch }

```

```
    { page; 150 and 300 dots per inch will }
    { produce smaller graphs                }
```

```
CursorPosition = '5';
    { Column position of left side of graph }
Esc             = #27;
    { Escape character                      }
```

```
Var LineHeader  : String[6];
    { Line Header used for each line sent }
    { to the LaserJet printer.           }
LineLength     : String[2];
    { Length in bytes of the line to be }
    { sent to the LaserJet.             }
Y              : Integer;
    { Temporary loop Variable.          }
```

```
Procedure DrawLine ( Y : Integer );
{ Draws a single line of dots. No of Bytes sent to printer }
{ is Width + 1. Argument of the procedure is the row no, Y }
```

```
Var GraphStr    : String[255]; { String used for OutPut }
Base            : Word;        { Starting position of }
                                { output byte.           }
BitNo,          : Word;        { Bit Number worked on }
ByteNo,         : Word;        { Byte number worked on }
DataByte       : Byte;        { Data Byte being built }
```

```
Begin
FillChar( GraphStr, SizeOf( GraphStr ), #0 );
GraphStr := LineHeader;
For ByteNo := 0 to Width Do
Begin
DataByte := 0;
Base := 8 * ByteNo;
For BitNo := 0 to 7 Do
Begin
If GetPixel( BitNo+Base, Y ) > 0
Then
Begin
DataByte := DataByte + 128 Shr BitNo;
End;
End;
GraphStr := GraphStr + Chr (DataByte)
End;
```

```

Write (Lst, GraphStr)

End; {Of Drawline}

Begin {Main procedure HPCopy}
  FillChar( LineLength, SizeOf( LineLength ), #0 );
  FillChar( LineHeader, SizeOf( LineHeader ), #0 );

  GetViewSettings( Vport );
  Width := ( Vport.X2 + 1 ) - Vport.X1;
  Width := ( ( Width - 7 ) Div 8 );
  Height := Vport.Y2 - Vport.Y1;

  Write (Lst, Esc + 'E');           { Reset Printer   }
  Write (Lst, Esc+'*t'+DotsPerInch+'R'); { Set density in }
                                          { dots per inch  }
  Write (Lst, Esc+'&a'+CursorPosition+'C'); { Move cursor to }
                                          { starting col   }
  Write (Lst, Esc + '*r1A');         { Begin raster graphics }

  Str (Width + 1, LineLength);
  LineHeader := Esc + '*b' + LineLength + 'W';

  For Y := 0 To Height + 1 Do
  Begin
    DrawLine ( Y );
    DrawLine ( Y );
  End;

  Write (Lst, Esc + '*rB');           { End Raster graphics }
  Write (Lst, Esc + 'E');           { Reset printer and   }
                                          { eject page          }

End;

Begin
  AssignLST( 1 );                    { This is the parameter to change }
                                          { if you want the output to be }
                                          { directed to a different LST }
                                          { device.                  }
End.

```

Reference:

3/31/99 10:18:34 AM

Last Modified: 01-SEP-99

## Article #15073: LINKING .OBJ OBJECT FILES FROM OTHER ASSEMBLERS

Technical Notes Database

TN73D.txt      LINKING .OBJ OBJECT FILES FROM OTHER ASSEMBLERS  
Category      :Pascal  
Platform      :All  
Product       :Turbo Pascal 4.0+

### Description:

- Q. Will the \$L compiler directive work for compiler object files other than assembler?
- A. That depends on the language. TURBO requires all the code in the .OBJ to be in \*one\* CODE segment, and all the data to be in \*one\* DATA segment. With assembly language that's easy, but it may not work with some high-level language compilers. You can use Turbo C to generate .OBJ files for use by Turbo Pascal programs. An example, CPASDEMO.PAS is included on the distribution disks.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15080: Printing graphics to an Epson compatible printer

TI80D.txt     PRINTING GRAPHICS TO AN EPSON COMPATIBLE PRINTER  
Category     :BGI  
Platform     :All  
Product      :Pascal   All

```
{ The following example routines are public domain programs }  
{ that have been uploaded to our Forum on CompuServe. As a }  
{ courtesy to our users that do not have immediate access }  
{ to CompuServe, Technical Support distributes these }  
{ routines free of charge. }  
{ }  
{ However, because these routines are public domain programs, }  
{ not developed by Borland International, we are unable to }  
{ provide any Technical Support or assistance using these }  
{ routines. If you need assistance using these routines, }  
{ or are experiencing difficulties, we recommend that }  
{ you log onto CompuServe and request assistance from the }  
{ Forum members that developed these routines. }
```

Unit GraphPRN;

```
{ This unit is designed to send graphics images to Epson }  
{ Compatible and late model IBM ProPrinter Dot Matrix }  
{ printers. It takes the image from the currently active }  
{ Viewport, determined by a call to GetViewSettings, and }  
{ transfers that image to the printer. }
```

Interface

```
Uses Dos, Graph;     { Used to get the Image from the Screen }
```

Var

```
    LST : Text;
```

Procedure HardCopy (Gmode: Integer);

```
{ Procedure HardCopy prints the current ViewPort }  
{ To an IBM or Epson compatible graphics printer. }  
{ }  
{ Valid Gmode numbers are : }
```

```

{      -4 to -1 for Epson and IBM Graphic Printers      }
{      0 to 7  for Epson Printers                       }

```

## Implementation

```

Procedure HardCopy {Gmode: Integer};

```

### Const

```

  Bits : Array [0..7] of Byte = (128,64,32,16,8,4,2,1);

```

### Var

```

  X,Y,YOfs      : Integer;    { Screen location variables }
  BitData,MaxBits : Byte;      { Number of Bits to transfer }
  Vport          : ViewPortType; {Used to get view settings }
  Height, Width  : Word;       { Size of image to transfer }
  HiBit, LoBit   : Char;       { Char size of image }
  LineSpacing,   :              { Additional Info for dump }
  GraphixPrefix  : String[10]; { " " " " " " }

```

### Begin

```

  LineSpacing := #27+'3'+#24; { 24/216 inch line spacing }
  Case Gmode Of
    -1: GraphixPrefix := #27+'K'; { Std. Density }
    -2: GraphixPrefix := #27+'L'; { Double Density }
    -3: GraphixPrefix := #27+'Y'; { Dbl. Density Dbl. Speed }
    -4: GraphixPrefix := #27+'Z'; { Quad. Density }
    0..7: GraphixPrefix := #27+'*'+Chr(Gmode); { 8-Pin Bit Img }
  Else
    Exit; { Invalid Mode Selection }
  End;
  GetViewSettings( Vport ); { Get size of image to }
  Height := Vport.Y2 - Vport.Y1; { be printed }
  Width := ( Vport.X2 + 1 ) - Vport.X1;
  HiBit := Chr(Hi(Width)); {Translate sizes to char }
  LoBit := Chr(Lo(Width)); { for output to printer }
  Write( LST, LineSpacing );
  Y := 0;
  While Y < Height Do
  Begin
    Write( LST,GraphixPrefix,LoBit,HiBit );
    For X := 0 to Width-1 Do
    Begin
      BitData := 0;
      If y + 7 <= Height

```



```

    Then MaxBits := 7
Else
    MaxBits := Height - Y;
For YOfs := 0 to MaxBits do
Begin
    If GetPixel( X, YOfs+Y ) > 0
        Then BitData := BitData or Bits[YOfs];
    End;
    Write( LST, Chr(BitData) );
End;
WriteLn ( LST );
Inc(Y,8);
End;
End;

{$F+}

{
    LSTNoFunction performs a NUL operation for a Reset or
{ Rewrite on LST (Just in case)
}

Function LSTNoFunction( Var F: TextRec ): integer;
Begin
    LSTNoFunction := 0;
    { No error
}
end;

{
    LSTOutputToPrinter sends the output to the Printer
{ port number stored in the first byte of the UserData area
{ of the Text Record.
}

Function LSTOutputToPrinter( Var F: TextRec ): integer;
var
    Regs: Registers;
    P : word;
begin
    With F do
    Begin
        P := 0;
        Regs.AH := 16;
        While (P < BufPos) and ((regs.ah and 16) = 16) do
        Begin
            Regs.AL := Ord(BufPtr^[P]);
            Regs.AH := 0;
            Regs.DX := UserData[1];
            Intr($17,Regs);
            Inc(P);

```

```

    end;
    BufPos := 0;
End;
if (Regs.AH and 16) = 16 then
    LSTOutputToPrinter := 0           { No error           }
else
    if (Regs.AH and 32 ) = 32 then
        LSTOutputToPrinter := 159    { Out of Paper      }
    else
        LSTOutputToPrinter := 160;   { Device write Fault }
End;

```

```

{$F-}

{
    AssignLST both sets up the LST text file record as
    would ASSIGN, and initializes it as would a RESET. It also
    stores the Port number in the first Byte of the UserData
    area.
}

```

```

Procedure AssignLST( Port:Byte );
Begin
    With TextRec(LST) do
        begin
            Handle      := $FFF0;
            Mode        := fmOutput;
            BufSize     := SizeOf(Buffer);
            BufPtr      :=
Buffer;
            BufPos      := 0;
            OpenFunc    :=
LSTNoFunction;
            InOutFunc   :=
LSTOutputToPrinter;
            FlushFunc   :=
LSTOutputToPrinter;
            CloseFunc   :=
LSTOutputToPrinter;
            UserData[1] := Port - 1; { We subtract one because }
        end;           { Dos Counts from zero. }
    end;
end;

```

```

Begin
    AssignLST( 1 );           { Sets output printer to LPT1 by }

```

End.

```
{ default. Change this value to }  
{ a 2 to select LPT2.           }  
{ Note: BIOS only handles LPT1  }  
                                {      and      LPT2. }
```

}

7/16/98 4:33:37 PM

Last Modified: 01-SEP-99

## Article #15082: OBJECT FILE CREATION

Technical Notes Database

TN82D.txt      OBJECT FILE CREATION  
Category      :Pascal  
Platform       :All  
Product       :Turbo Pascal 4.0+

### Description:

Q. Does Turbo Pascal create object files that can be linked into other languages?  
A. Turbo Pascal 4.0+ generates .TPU (Turbo Pascal Unit) files, not .OBJ files. We've made that decision for many reasons:

1. TP 4.0+'s .TPU files are smaller than .OBJ's, and they contain symbolic information important to the support of Pascal's strict type conventions (types, constants, etc.).
2. .TPU files allow "smart linking" - elimination of unused code and data on a procedure-by-procedure basis.
3. .TPU's allow built-in project management through version 4.0+'s Make and Build commands.
4. .TPU's allow faster compilation speeds (34,000 lines per minute on a PS/2 Model 60).

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15089: Microstar critical error handler corrections

Technical Information Database

TI89D.txt - Microstar critical error handler corrections

Category :Turbo Pascal  
Platform :All  
Product :

### Description:

This handout addresses a problem reported by users of the Turbo Pascal Editor Toolbox version 4.0 for use with Turbo Pascal 4.0.

Users have found that MicroStar cannot recover from a DOS Critical Error. DOS Critical Errors occur when attempting to write or read a disk file with the disk door open, the drive empty, or the printer out of paper, etc. MicroStar reports the Critical E

The problem lies with the Interrupt 24 Handler (the DOS Critical Error Handler) implemented within MicroStar; its source may be found in the file INT24.ASM of the MicroStar source code. The INT24.ASM correctly identifies Critical Errors but does not rese

1. Load INT24.ASM into an ASCII editor (Turbo Pascal's editor will do).

2. Find Line 68 which should read as follows:

```
MOV CS:Int24ErrCode,0 ;Reset Int24ErrCode to 0
```

3. Replace this text with the following line.

```
MOV CS:Int24Err,0 ;Reset Int24Err to 0
```

4. Use the Microsoft or a compatible assembler to assemble the code to an object format (.OBJ). This must replace the INT24.OBJ supplied with the distribution disks.
5. Set the compiler destination to disk (Compile/Destination option). Load MS.PAS into the editor and perform a build

(Compile/Build option) on the entire application. Make sure the entire MicroStar source is available for recompilation.

For those without an assembler, the following Turbo Pascal program corrects the unmodified MicroStar EXE file. From the directory containing MicroStar, enter the following lines of code into the Turbo Pascal editor and run from either memory or disk.

```
program ChMS;

var
  f : file of byte;
  b : byte;

begin

Assign(f, 'MS.EXE');
reset(f);

seek( f, 129626);   b := 30;   write( f, b);
seek( f, 154257);   b := 101;  write( f, b);
seek( f, 154305);   b := 4;    write( f, b);
seek( f, 154401);   b := 103;  write( f, b);
seek( f, 154431);   b := 115;  write( f, b);
seek( f, 154701);   b := 108;  write( f, b);
seek( f, 154827);   b := 108;  write( f, b);

close(f);

end.
```

Reference:

3/30/99 3:55:36 PM

Last Modified: 01-SEP-99

## Article #15091: LINKING TURBO C OR ASSEMBLER .OBJ OBJECT FILES

Technical Notes Database

TN91D.txt      LINKING TURBO C OR ASSEMBLER .OBJ OBJECT FILES  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. Are the .OBJ files generated by Turbo C and Turbo Assembler compatible with 4.0+?

A. Yes. You can write Turbo C or Turbo Assembler routines and link the .OBJ files into your Turbo Pascal programs by using the {\$L} compiler directive. See the CTOPAS example on the distribution diskette.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15098: Handling Special Keys and Direct Memory Access

Technical Information Database

TI98D.txt - Handling Special Keys and Direct Memory Access

Category : Turbo Pascal  
Platform : All  
Product :

Description:

{  
Toggle Controls: -- allow you to check to see if a certain key was pressed or to turn off or on a certain key, such as activating the Num-Lock key.

MemW[0000:\$0417]

number	bit	
1	0	- Right Shift
2	1	- Left Shift
4	2	- Ctrl
8	3	- Alt
16	4	- Scroll Lock
32	5	- Num Lock
64	6	- Caps Lock
128	7	- Insert
256	8	-
512	9	-
1024	10	- Sys Req
2048	11	-
4096	12	- Scroll Lock Pressed
8192	13	- Num Lock Pressed
16384	14	- Caps Lock Pressed
32768	15	- Insert Pressed

Other memory locations that can be accessed to get/put information.

Clock ticks: MemW[\$0040:\$006C] updates every 58ms.

Clear Key Buffer: MemW[0000:\$041A] := MemW[0000:\$041C].



Color Address: \$B800:0000; Mono Address: \$B000:0000.

Print Screen: inline (\$CD/\$05).

}

{ example }

program TrapAlt;

Uses

Dos, Crt;

Var

i:char;

Function alt:boolean;

Begin

if MemW[0000:\$0417] and 8<>0 then

begin

alt:=true;

repeat

if keypressed then

begin

alt:=false;

exit;

end;

until MemW[0000:\$0417] and 8=0;

end

else

alt:=false;

End;

Begin

clrscr;

repeat

if keypressed then

begin

writeln('non alt');

i:=readkey;

end;

if alt then writeln('Alt key pressed');

until (i=#13);

End.

Reference:

3/31/99 10:18:47 AM

Last Modified: 01-SEP-99

## Article #15100: DATA SEGMENT USAGE OF SYSTEM UNITS

Technical Notes Database

TN100D.txt DATA SEGMENT USAGE OF SYSTEM UNITS

Category :Pascal

Platform :All

Product :Turbo Pascal 5.0 5.5

Description:

Q. How much of the 64K in the data segment is actually available to my program?

A. The amount of data segment used by the run-time library depends on which standard units you use in your program. Here is the data segment usage (in bytes) for each unit:

UNIT	Data Size
----	-----
System	664
Overlay	10
Crt	20
Dos	6
Printer	256
Graph	1070
Turbo3	256
Graph3	0
	=====
	2282

The total size of the data segment is 65,520 bytes. If you used only the System unit, the amount of data segment space left over would be

$$65520 - 664 = 64856 \text{ bytes}$$

Reference:

7/16/98 4:35:29 PM



## Article #15107: Changing Color of Buttons with Bitmaps

Technical Information Database

TI107D.txt Changing Color of Buttons with Bitmaps

Category :OWL

Platform :All

Product :Pascal All

Description:

```
{ $R bnn }
```

```
{ The above resource assumes that you have a dialog name 'dialog_1' that contains three buttons of id 101, 102, 103.
```

```
There should also be six bitmaps in the resource of value 1101, 3101, 5101 which hold the bitmaps for the red button and 1102, 3102, 5102 which hold the bitmaps for the blue button. Clicking the blue button should change the third button blue and clicking the red button should change it red.
```

```
}
```

```
uses WinTypes, WinProcs, WObjects, bwcc;
```

```
const
```

```
    id_Red      = 101;
```

```
    id_Blue    = 102;
```

```
    id_Change  = 103;
```

```
    IsRed: Boolean = False;
```

```
    IsBlue: Boolean = False;
```

```
type
```

```
    TApp = object(TApplication)
```

```
        procedure InitMainWindow; virtual;
```

```
    end;
```

```
    PMyDialog = ^TMyDialog;
```

```
    TMyDialog = object(TDialog)
```

```
        Red, Blue: array[0..2] of hBitmap;
```

```
        constructor Init(AParent :PWindowsObject; AName: PChar);
```

```
        destructor Done; virtual;
```

```
        procedure ChangeToRed(var Message: TMessage);
```

```
            virtual id_First + id_Red;
```

```
        procedure ChangeToBlue(var Message: TMessage);
```

```
virtual id_First + id_Blue;
```

```
end;
```

```
{ TApp }
```

```
procedure TApp.InitMainWindow;
```

```
begin
```

```
  MainWindow := New(PMyDialog, Init(Nil, 'dialog_1'));
```

```
end;
```

```
{ TMyDialog }
```

```
constructor TMyDialog.Init(AParent :PWindowsObject; AName:  
PChar);
```

```
begin
```

```
  TDialog.Init(AParent, AName);
```

```
  Red[0] := LoadBitMap(hInstance, PChar(1101));
```

```
  Red[1] := LoadBitMap(hInstance, PChar(3101));
```

```
  Red[2] := LoadBitMap(hInstance, PChar(5101));
```

```
  Blue[0] := LoadBitMap(hInstance, PChar(1102));
```

```
  Blue[1] := LoadBitMap(hInstance, PChar(3102));
```

```
  Blue[2] := LoadBitMap(hInstance, PChar(5102));
```

```
end;
```

```
destructor TMyDialog.Done;
```

```
var
```

```
  I: Integer;
```

```
begin
```

```
  TDialog.Done;
```

```
  if Not IsRed then
```

```
    for I := 0 to 2 do
```

```
      DeleteObject(Red[I]);
```

```
  if Not IsBlue then
```

```
    for I := 0 to 2 do
```

```
      DeleteObject(Blue[I]);
```

```
end;
```

```
procedure TMyDialog.ChangeToRed(var Message: TMessage);
```

```
begin
```

```
  SendDlgItemMessage(HWindow, id_Change, bbm_SetBits, 0,
```

```
    LongInt(
```

```
Red));
```

```
  InvalidateRect(GetDlgItem(HWindow, id_Change), Nil, True);
```

```
  IsRed := True;
```

```
  IsBlue := False;
```

```
end;
```

```
procedure TMyDialog.ChangeToBlue(var Message: TMessage);
begin
    SendDlgItemMessage(HWindow, id_Change, bbm_SetBits, 0,
        LongInt(
    Blue));
    InvalidateRect(GetDlgItem(HWindow, id_Change), Nil, True);
    IsRed := False;
    IsBlue := True;
end;
```

```
var
    App: TApp;
begin
    App.Init('Test');
    App.Run;
    App.Done;
end.
```

Reference:

7/16/98 4:33:37 PM

Last Modified: 01-SEP-99

## Article #15109: VARIABLES IN UNITS CHANGES NOT REFLECTED

Technical Notes Database

TN109D.txt      VARIABLES IN UNITS CHANGES NOT REFLECTED  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 4.0+

### Description:

- Q. I moved some variables into a unit. When I change a variable in the program, the change is not reflected in the unit used by the program. Why?
- A. Remove the duplicated variable definitions from the main program. Variables to be used in both a unit and a main program must be declared only in the interface portion of the unit.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99



## Article #15116: DrawAxis option: no tick marks and number labels

Technical Information Database

TI116D.txt - DrawAxis option: no tick marks and number labels

Category :Turbo Pascal  
Platform :All  
Product :

### Description:

The following modifications may be made to the "GSHELL.PAS" file of the Turbo Pascal Graphix Toolbox to provide the DrawAxis routine with an option to turn off the display of tick marks and number labels, as in version 1.0x.

The following routine is found in the file called "GSHELL.PAS".

Change the following lines in the DrawAxis procedure:

Line 213:

```
FROM: if YDens >= 0 then  
TO:   if YDens > 0 then
```

Line 230:

```
FROM: if (YDens >= 0) and (Ys > Y1RefGlb + 13) then  
TO:   if (YDens > 0) and (Ys > Y1RefGlb + 13) then
```

Line 240:

```
FROM: if XDens >= 0 then  
TO:   if XDens > 0 then
```

Line 260:

```
FROM: if (XDens >= 0) and (Xs > X2RefGlb shl 3 + 7 - 24) then  
TO:   if (XDens > 0) and (Xs > X2RefGlb shl 3 + 7 - 24) then
```

Reference:

Last Modified: 01-SEP-99

## Article #15118: UNITS LINKED WITH DIFFERENT COMPILER DIRECTIVES

Technical Notes Database

TN118D.txt      UNITS LINKED WITH DIFFERENT COMPILER DIRECTIVES  
Category      :Pascal  
Platform      :All  
Product      :Turbo Pascal 4.0+

### Description:

- Q. What happens if you attempt to link another unit in which the compiler directives are set differently?
- A. Compiler directives are local to the unit they are declared in. Thus, the compiler directives in one unit, or in the main program, have no effect on the directives set in another unit.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15125: Patch for include file not found by Turbo Debugger

Technical Information Database

TI125D.txt - Patch for include file not found by Turbo Debugger

Category :Turbo Pascal  
Platform :All  
Product :

### Description:

This handout describes the patches needed in order for Turbo Pascal version 5.0 to generate the correct debug information so that a \$Include file in a user defined unit can be found by Turbo Debugger. Two patches are supplied, one for TURBO.EXE and one for TPC.EXE. Both patches should be made using the DOS utility, DEBUG on copies of the original .EXE files.

### Notes:

1. DEBUG is not case sensitive to upper and lower case. All addresses are listed in upper case for ease of readability. All addresses are in hexadecimal.
2. While in DEBUG, the prompt will appear as a dash (-).
3. If you do not receive the appropriate response, press "q" followed by to quit.

### TURBO.EXE Patch

1. Rename TURBO.EXE to TURBO.BIN (DEBUG will not allow modifications to be made to a .EXE file.)

```
Prompt>ren turbo.exe turbo.bin
```

2. Load TURBO.BIN into DEBUG with the command:

```
Prompt>debug turbo.bin
```

3. Display the registers by entering "r" at DEBUG's hyphen

prompt.

-r

The system will respond with a display of the CPU register values. The following is an example:

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  ...  
DS=3C23  ES=3C23  SS=3C23  CS=3C23  IP=0100  NV UP    ...
```

Note the value of the CS register. Add 1000h to this value mentally; e.g., in the above example CS=3C23, then CS + 1000h is 4C23. This value will be referenced by ????.

4. At DEBUG's hyphen prompt, type:

```
-a ????:A41E
```

5. Enter the following Assembler code when the address is displayed:

```
????:A41E cmp si,dx
```

6. Press a second time.

7. Write the modified code to disk by entering the following at the prompt:

```
-w
```

8. Quit DEBUG by entering the following:

```
-q
```

9. At the DOS prompt, rename the .BIN file back to the .EXE file with the following command:

```
Prompt>ren turbo.bin turbo.exe
```

TPC.EXE Patch

1. Rename TPC.EXE to TPC.BIN (DEBUG will not allow modifications to be made to an .EXE file.)

```
Prompt>ren tpc.exe tpc.bin
```

2. Load TPC.BIN into DEBUG with the command:

```
Prompt>debug tpc.bin
```

3. At DEBUG's hyphen prompt, type:

```
-a a5d3
```

4. Enter the following Assembler code when the address is displayed:

```
XXXX:A5D3 cmp si,dx
```

5. Press a second time.

6. Write the modified code to disk by entering the following at the prompt:

```
-w
```

7. Quit DEBUG by entering the following:

```
-q
```

8. At the DOS prompt, rename the .BIN file back to the .EXE file with the following command:

```
Prompt>ren tpc.bin tpc.exe
```

Reference:

3/30/99 4:01:16 PM

Last Modified: 01-SEP-99

## Article #15127: Units compiled more than once

Technical Notes Database

TN127D.txt      Units compiled more than once  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 4.0+

### Description:

- Q. If two units use a third unit, does the third unit get included twice in my program?
- A. No. All your units are "linked" together when you compile your program. Only one copy of each procedure and function used is generated. There is NO duplication of run-time code. In fact, Turbo Pascal 4.0+ has "smart linking" which eliminates any unused code. Turbo Pascal 5.0+ will also "smart link" unused data from the final .EXE file.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15134: Linking error message and font files.

Technical Information Database

TI134D.txt      Linking error message and font files.

Category      :General Programming

Platform      :All

Product      :Pascal    All

### Description:

This technical information handout describes how to link in the ERROR.MSG, 4x6.FON, 8x8.FON and 14x9.FON files into your \*.EXE file so that your application no longer needs these files to be present on disk. Please make these changes to copies of your

GRAF3270.DVR	GRAFEGA.DVR	ERROR.MSG	14X9.FON
GRAFATT.DVR	GRAFHGC.DVR	4X6.FON	GDRIVER.PAS
GRAFCGA.DVR	GKERNEL.PAS	8X8.FON	

First, you must convert the ERROR.MSG, 4x6.FON, 8x8.FON, and 14x9.FON files to the OBJ format.

Use the BINOBJ.EXE utility to convert these files to the OBJ format.

For example,

```
BINOBJ ERROR.MSG ERROR.OBJ FakeErr
BINOBJ 4x6.FON 4x6.OBJ Font4x6
BINOBJ 8x8.FON 8x8.OBJ FakeFont
BINOBJ 14x9.FON 14x9.OBJ FakeFont
```

Next, delete the first 13 lines of code from the procedure EnterGraphic in each of the \*.DVR and GDRIVER.PAS files with the exception of GRAFATT.DVR file. In the GRAFATT.DVR file, delete the first 15 lines of code instead. Also, in the GRAFCGA.DVR file

s



```

{$L 14x9}           { ADD }
procedure FakeFont; external;   { ADD }

procedure EnterGraphic;
{ Enter graphics mode }
...
...
begin
  if not FontLoaded then      { ADD }
    begin                    { ADD }
      Move(Mem[Seg(FakeFont):Ofs(FakeFont)], Font, SizeOf(Font));
                              {ADD}
      FontLoaded := true;    { ADD }
    end;                      { ADD }
...
.....
.....

```

The rest of the EnterGraphic procedure remains the same.

The following changes should be made in the GKERNEL.PAS unit:

```

{$L 4x6 }           {NEW}
procedure Font4x6; external;   {NEW}

{$L Error}         {NEW}
procedure FakeErr; external;   {NEW}

procedure InitGraphic;
var
  Fil : file of CharArray;
  Tfile : text;
  Test : ^integer;
  Temp : WrkString;
  I : word;
  GTBCount : integer;           {NEW}
  GTBch : char;                 {NEW}

  Function ErrorStr : WrkString; {NEW}
begin                             {NEW}
  temp := '';                      {NEW}
  repeat                            {NEW}
    GTBch := chr(mem[Seg(FakeErr):Ofs(FakeErr) + GTBcount]);
    if not ((GTBch = #10) or (GTBch = #13)) then {NEW}
      temp := temp + GTBch;         {NEW}
    inc(GTBcount);                 {NEW}

```

```
    until(GTBch = #13);           {NEW}
    ErrorStr := temp;           {NEW}
end; { Function ErrorStr }      {NEW}
```

```
begin
  MessageGlb := true;
  BrkGlb := false;
  GrafModeGlb := false;
  GotoXY(1, 1);
  if not HardwarePresent then
  begin
    ClrScr;
    GotoXY(1, 2);
    WriteLn('Fatal Error: graphics hardware not found or not
properly activated');
    Halt;
  end;
  GetMem(ErrorProc[0], 16);
  GetMem(ErrorCode[0], 24);
  ErrorProc[0]^ := 'InitGraphic';
  ErrorCode[0]^ := 'Error.MSG missing';
  GTBcount := 0;           {NEW}
  for I := 0 to MaxProcsGlb do           {MODIFIED}
  begin
    Temp := ErrorStr;           {NEW}
    if I <> 0 then
      GetMem(ErrorProc[I], Length(Temp) + 1);
      ErrorProc[I]^ := Temp;
    end;
  for I := 0 to MaxErrsGlb do           {MODIFIED}
  begin
    Temp := ErrorStr;           {NEW}
    if I <> 0 then
      GetMem(ErrorCode[I], Length(Temp) + 1);
      ErrorCode[I]^ := Temp;
    end;
  PcGlb := ErrorStr;           {MODIFIED}
  for I := 1 to MaxWorldsGlb do
    DefineWorld(I, 0, 0, XScreenMaxGlb, YMaxGlb);
  MaxWorldGlb := 1;
  for I := 1 to MaxWindowsGlb do
  begin
    DefineWindow(I, 0, 0, XMaxGlb, YMaxGlb);
    with Stack[I] do
    begin
```

```
W.Size := 0;
Contents := nil;
end;
end;
MaxWindowGlb := 1;
Move(mem[seg(Font4x6):ofs(Font4x6)], CharSet, Sizeof(CharSet));
{NEW}
```

Delete the next 12 lines of code.

The rest of the code remains the same in the GKERNEL.PAS unit.

```
BrkGlb := True;
If RamScreenGlb Then
...
...
```

\*\*\*\*\*

Reference:

7/16/98 4:33:38 PM

Last Modified: 01-SEP-99

## Article #15136: TYPE MISMATCH ERROR COMPILING UNITS

Technical Notes Database

TN136D.txt      TYPE MISMATCH ERROR COMPILING UNITS  
Category      :Pascal  
Platform      :All  
Product       :Turbo Pascal 4.0+

### Description:

- Q. Why is the compiler reporting "Type Mismatch Error" when I have declared every variable?
- A. The most common error of this type is caused by duplicated type definitions. First some background on how Turbo Pascal 4.0+ works. All identifiers (procedures, functions, types, variables, constants) are handled by appending the module name to the front of them (ie., ProgramName.identifier or UnitName.Identifier). Now, if you have a type, MYTYPE, in Unit MyUnit and also in MyProgram you have the following TWO DIFFERENT Types :

MyUnit.MYTYPE and MyProgram.MyType.

Please notice that by no means does "MyUnit.MYTPYE" equal "MyProgram.MYTYPE". The actual contents of MyType does not even enter into this discussion and Turbo Pascal only looks at the Type name.

The Solution? Declare types only in the interface section of the original unit AND NOWHERE ELSE!. Alternatively, declare variables with the file type name (ie., if you want to pass a variable of MYTYPE to the unit MyUnit, declare the variable:  
var MyVar : MyUnit.MYTYPE;).

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15143: Turbo Vision, 132 Column Mode

Technical Information Database

TI143D.txt Turbo Vision, 132 Column Mode  
Category :TVISION  
Platform :All  
Product :Pascal All

### Description:

program TV132;

```
{ this program shows how to put a Turbo Vision application in  
132 column mode. it is written specifically for an ATI VGA  
card but can be modified for other cards. also your mouse  
drive must be able to support 132 columns. }
```

```
{ $X+ }
```

```
uses  
    objects, drivers, views, app, dialogs , menus, dos;  
const  
    cm_DoIt = 100;  
  
type  
    PMyApp = ^TMyApp;  
    TMyApp = object(TApplication)  
        procedure InitMenuBar; virtual;  
        procedure InitScreen; virtual;  
        procedure HandleEvent(var Event: TEvent); virtual;  
        procedure CMDoIt;  
    end;  
  
    PMyDialog = ^TMyDialog;  
    TMyDialog = object(TDialog)  
        procedure InitFrame; virtual;  
        procedure HandleEvent(var Event: TEvent); virtual;  
    end;  
  
    PMyFrame = ^TMyFrame;  
    TMyFrame = object(TFrame)  
        procedure Draw; virtual;  
    end;
```

```
procedure TMyFrame.Draw;
begin
end;
```

```
procedure TMyDialog.InitFrame;
var
  R: TRect;
begin
  TDialog.InitFrame;
  (*
  R.Assign(0, 0, Size.X, Size.Y);
  Frame := new(PMyFrame, init(R));
  *)
end;
```

```
procedure TMyApp.CMDoIt;
var
  D: PMyDialog;
  ILine : PInputLine;
  R: TRect;
begin
  R.Assign(5, 5, 25, 12);
  new(D, init(R, 'Do It'));
  D^.SetState(sfShadow, false);
  R.Assign(1, 1, 19, 2);
  new(ILine, init(R, 20));
  R.Assign(1, 3, 19, 4);
  D^.Insert(ILine);
  D^.Insert(new(PInputline, init(R, 20)));
  R.Assign(1, 5, 11, 7);
  D^.Insert(new(PButton, Init(R, '~O~k', cmOk, bfDefault)));
  Iline^.Select;
  Desktop^.ExecView(D);
end;
```

```
procedure TMyDialog.HandleEvent(var Event: TEvent);
begin
  if Event.What = evKeyDown then
    if Event.KeyCode = kbEnter then
      Event.KeyCode := kbTab;
  TDialog.HandleEvent(Event);
end;
```

```
procedure TMyApp.InitMenuBar;
```

```

var R:TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewItem('~D~o it', '', kbNoKey, cm_DoIt, hcNoContext,
    nil)
  )));
end;

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    if Event.Command = cm_DoIt then
      begin
        CMDoIt;
      end;
end;

procedure TMyApp.InitScreen;
{ this procedure is specific to an ATI VGA Card, you may need to
  modify if you have a different video card. }
begin
  TApplication.InitScreen;
  asm
    mov ah, 0
    mov al, 23H
    int 10H
  end;
  ScreenMode := $23;
  ScreenHeight := 25;
  ScreenWidth := 132;
end;

var
  MyApp: TMyApp;

begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.

```

Reference :

7/16/98 4:33:38 PM

Last Modified: 01-SEP-99



## Article #15145: CREATING .TPL FILES

Technical Notes Database

TN145D.txt      CREATING .TPL FILES

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. Can I create my own .TPL file?

A. Yes. However, Turbo Pascal will only use the TURBO.TPL library file. If you want to add your own units to the TURBO.TPL file, you can use the unit mover program (TPUMOVER.EXE). For example, you might want a customized version of TURBO.TPL for each of the programs you're developing. A corresponding configuration file for Turbo Pascal would specify a different Turbo directory and thus fetch the appropriate .TPL file for each of your projects.

Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15150: Function mapping from the Paradox Engine to BDE

Technical Information Database

TI150B.txt      Function mapping from the Paradox Engine to BDE

Category      :General Programming

Platform      :All

Product       :BDE 2.x

Description:

### Types

Paradox Engine	BDE	Variable name in this doc
PXCODE	DBIResult	rslt
TABLEHANDLE	hDBICur	tabH
RECORDHANDLE	pBYTE	rech
FIELDNUMBER	UINT16	iFld
LOCKHANDLE	UINT32	
RECORDNUMBER	UINT32	recNum
BLOBHANDLE	INT16	

### Functions

Paradox Engine	BDE
BLANK	<code>DbiPutField(tabH, iFld, rech, NULL);</code> (Cannot compare Field Data with BLANK in IDAPI. Use the value returned from <code>DbiGetField</code> in <code>bIsBlank</code> to determine if a field is blank.)
ISBANK()	Use the blank parameter in <code>DbiGetField(tabH, iFld, rech, pDest, &amp;bIsBlank);</code> or <code>DbiVerifyField(tabH, iFld, pSrc, &amp;bIsBlank);</code>
PXBlobClone()	Not Supported. (Private BLOBs are not supported in IDAPI)

PXBlobClose()	DbiFreeBlob(tabH, rech, iFld);
accept == FALSE	Not required in IDAPI. Note that
accept == TRUE	DbiFreeBlob(tabH, rech, iFld) needs to be called <u>after</u> adding a record containing a BLOB to a table.
PXBlobDrop()	DbiGetRecord(tabH, dbiNOLOCK, rech, NULL); DbiTruncateBlob(tabH, rech, iFld, 0); DbiModifyRecord(tabH, rech, FALSE);
PXBlobGet()	DbiGetBlob(tabH, rech, iFld, iOff, iLen, pDest, &iRead);
PXBlobGetSize()	DbiGetBlobSize(tabH, rech, iFld, piSize);
PXBlobOpenRead()	DbiOpenBlob(tabH, rech, iField, dbiREADONLY);
PXBlobOpenWrite()	DbiOpenBlob(tabH, rech, iField, dbiREADWRITE);
PXBlobPut()	DbiPutBlob(tabH, rech, iFld, iOff, iLen, pSrc);
PXBlobQuickGet()	DbiGetBlobHeading(tabH, iFld, rech, &iSize); (Paradox Tables only.)
PXDateDecode()	DbiDateDecode(dated, piMon, piDay, piYear);
PXDateEncode()	DbiDateEncode(iMon, iDay, iYear, pdated);
PXErrMsg()	DbiGetErrorString(iErrorCode, szError); or DbiGetErrorInfo(&ErrInfo); (Must be called immediately after the offending function - provides more information than DbiGetErrorString.)
PXExit()	DbiCloseDatabase(&hDb); DbiExit();
PXFldBlank()	DbiGetField(tabH, iFld, rech, NULL, &bIsBlank); or

```

DbiVerifyField(tabH, iFld, pSrc, &bIsBlank);
bIsBlank = TRUE is field is blank.

PXFldHandle()      DbiGetFieldDescs(tabH, &fldDesc);
                   fldDesc.iFldNum;

PXFldName()        DbiGetFieldDescs(tabH, &fldDesc);
                   fldDesc.szName;

PXFldType()        DbiGetFieldDescs(tabH, &fldDesc);
                   fldDesc.iFldType;
                   fldDesc.iSubType;

PXGetAlpha()       DbiGetField(tabH, iFld, rech,
                               (pBYTE)szString, &bIsBlank);

PXGetDate()        DbiGetField(tabH, iFld, rech,
                               (pBYTE)&Date, &bIsBlank);

PXGetDefaults()   DbiOpenCfgInfoList(hCfg, eOpenMode,
                                       eConfigMode, pszCfgPath,
                                       &tabH);

PXGetDoub()        DbiGetField(tabH, iFld, rech,
                               (pBYTE)&Double, &bIsBlank);

PXGetLong()        DbiGetField(tabH, iFld, rech,
                               (pBYTE)&Long, &bIsBlank);

PXGetShort()       DbiGetField(tabH, iFld, rech,
                               (pBYTE)&Short, &bIsBlank);

PXInit()           Not Supported. See PXWinInit.

PXKeyAdd()         DbiAddIndex(hDb, tabH, szTblName, szTblType,
                               &IdxDesc);

PXKeyDrop()        DbiDeleteIndex(hDb, tabH, szTblName,
                               szTblType, NULL, NULL,
                               iIndexId);

PXKeyMap()         Not Supported.
                   (Key Mapping is no longer needed - all
                    pertinent information is put in the index
                    descriptor - IdxDesc)

```

PXKeyNFlds()	<pre> DbiGetIndexSeqNo(tabH, NULL, NULL, 0,                   &amp;seqNo); DbiGetIndexDesc(tabH, seqNo, &amp;idxDesc); idxDesc.iFldsInKey; (Note that Paradox is the only database which has the concept of a primary index) </pre>
PXKeyQuery()	<pre> Not Supported. (The BDE does not have any function like this. You need to call DbiGetIndexDescs() and then look for the index you want. Once found all the information about that index is available to you in the structure.) </pre>
PXNetErrUser()	<pre> DbiErrGetErrorContext(ecUSERNAME,                       szUserName); </pre>
PXNetFileLock()	<pre> DbiAcqPersistTableLock(hDb, (pCHAR)fileName,                        Driver); </pre>
PXNetFileUnlock()	<pre> DbiRelPersistTableLock(hDb, (pCHAR)fileName,                        Driver); </pre>
PXNetInit()	<pre> Not Support. See PXWinInit. </pre>
PXNetRecGotoLock()	<pre> Not Supported. (Can be simulated by setting a bookmark on the record which is locked and then switching to that bookmark: DbiSetToBookMark(tabH, pBookMark);) </pre>
PXNetRecLock()	<pre> DbiGetRecord(tabH, dbiWRITELOCK, NULL,              NULL); </pre>
PXNetRecLocked()	<pre> DbiIsRecordLocked(tabH, edbiLock, piLocks); </pre>
PXNetRecUnlock()	<pre> DbiRelRecordLock(tabH, FALSE); or durring the update of the record - DbiModifyRecord(tabH, rech, TRUE); </pre>
PXNetTblChanged()	<pre> Need to register a cbTABLECHANGED callback. (Paradox only.) </pre>
PXNetTblLock()	<pre> DbiAcqTableLock(tabH, eLockType); </pre>

(Note that only Read and Write locks are supported by this function. For a FL on a table, open the table with the dbiOPENEXCL parameter or use the DbAcqPersistTableLock function.)

PXNetTblRefresh()	DbiForceReread(tabH);
PXNetTableUnlock()	DbiRelTableLock(tabH, eLockType);
PXNetUserName()	DbiGetNetUserName(pzName);
PXPswAdd()	DbiAddPassword(szPassword); (Paradox tables only.)
PXPswDel()	DbiDropPassword(szPassword); (Paradox tables only.)
PXPutAlpha()	DbiPutField(tabH, iFld, rech, (pBYTE)szString);
PXPutBlank()	DbiPutField(tabH, iFld, rech, NULL);
PXPutDate()	DbiPutField(tabH, iFld, rech, (pBYTE)&Date);
PXPutDoub()	DbiPutField(tabH, iFld, rech, (pBYTE)&Double);
PXPutLong()	DbiPutField(tabH, iFld, rech, (BYTE)&Long);
PXPutShort()	DbiPutField(tabH, iFld, rech, (pBYTE)&Short);
PXRawGet()	DbiSetProp((hDBIObj)tabH, curXLTMODE, xltNONE) DbiGetRecord(tabH, NULL, rech, NULL);
PXRawPut()	DbiSetProp((hDBIObj)tabH, curXLTMODE, xltNONE) DbiInsertRecord(tabH, dbiNOLOCK, rech, pDest);
PXRecAppend()	DbiAppendRecord(tabH, rech);

PXRecBufClose()           The application needs to release the memory associated with the record buffer. In 'C', call: free(rech).  
(Record buffers in IDAPI are owned by the application.)

PXRecBufCopy()           The application needs to copy the memory which is used for the record buffer. In 'C', call:  
memcpy(rechDest, rechSource, size).  
(Record buffers in IDAPI are owned by the application.)

PXRecBufEmpty()          DbiInitRecord(tabH, rech);

PXRecBufOpen()           The application needs to allocate memory for record buffer. In 'C', call:  
DbiGetCursorProps(tabH, &CurProps);  
rech = (pBYTE)malloc(CurProps.iRecBufSize \* sizeof(BYTE));  
(Record buffers in IDAPI are owned by the application.)

PXRecDelete()            DbiDeleteRecord(tabH, NULL);

PXRecFirst()             DbiSetToBegin(tabH);  
DbiGetNextRecord(tabH, dbiNOLOCK, NULL, NULL);  
(IDAPI has the concept of Beginning of file, which is before the first record).

PXRecGet()                DbiGetRecord(tabH, dbiNOLOCK, rech, NULL);

PXRecGoto()              Not supported by the SQL Drivers.  
Example for use with 'C':  
DBIResult gotoRec(hDBICur tabH,  
                  UINT32 recNum)  
{  
    DBIResult     lastError;  
    CURProps     curProps;  
  
    lastError = DbiGetCursorProps(tabH,  
                                  &curProps);  
    if (lastError != DBIERR\_NONE)

```

    {
        return lastError;
    }

    if (!strcmp(curProps.szTableType,
                szPARADOX))
    {
        lastError = DbiSetToSeqNo(tabH,
                                   recNum);
    }
    else if (!strcmp(curProps.szTableType,
                     szDBASE))
    {
        lastError = DbiSetToRecordNo(tabH,
                                       recNum);
    }
    else
    {
        lastError = DBIERR_NOTSUPPORTED;
    }

    return lastError;
}

```

```

PXRecInsert()      DbiInsertRecord(tabH, dbiNOLOCK, rech);

PXRecLast()        DbiSetToEnd(tabH);
                   DbiGetPriorRecord(tabH, dbiNOLOCK, NULL,
                                       NULL);
                   (IDAPI has the concept of End of File, which
                    is after the last record).

PXRecNext()        DbiGetNextRecord(tabH, dbiNOLOCK, NULL,
                                       NULL);

PXRecNFlds()       DbiGetCursorProps(tabH, &CurProps);
                   lFlds = CurProps.iFields;

PXRecNum()         Not supported for SQL Tables.
                   Example for use with 'C':
                   DBIResult getCurRecNum(hDBICur tabH,
                                             UINT32 *recNum)
                   {
                       DBIResult    lastError;
                       CURProps     curProps;

```



```

RECProps    recProps;

lastError = DbiGetCursorProps(tabH,
                               &curProps);
if (lastError != DBIERR_NONE)
{
    return lastError;
}

if (!strcmp(curProps.szTableType,
            szPARADOX))
{
    lastError = DbiGetSeqNo(tabH,
                            (pUINT32)recNum);
}
else if (!strcmp(curProps.szTableType,
                 szDBASE))
{
    lastError = DbiGetRecord(tabH,
                              dbiNOLOCK,
                              NULL,
                              &recProps);
    *recNum = recProps.iPhyRecNum;
}
else
{
    lastError = DBIERR_NOTSUPPORTED;
}

return lastError;
}

```

```

PXRecPrev()    DbiGetPriorRecord(tabH, dbiNOLOCK, NULL,
                          NULL);

```

```

PXRecUpdate() DbiModifyRecord(tabH, rech, TRUE);

```

```

PXSave()      DbiForceReread(tabH);

```

```

PXSetDefaults() DbiOpenCfgInfoList();

```

```

PXSetHWHandler() Not Supported

```

```

PXSrchFld()    DbiSetToKey(tabH, keySEARCHEQ, FALSE, iFlds
                          iLen, pIdxBuf);

```

```

PXSrchKey( )           DbiSwitchToIndex(&tabH, NULL, NULL, 0,
                       TRUE);
DbiSetToKey(tabH, keySEARCHEQ, FALSE, iFlds
             iLen, pIdxBuf);
(can only search on the currently active
 index)

PXTblAdd( )           DbiBatchMove(NULL, hSrcCur, NULL, hDestCur,
                       batAPPEND, 0, NULL, NULL, NULL,
                       NULL, NULL, NULL, NULL, NULL,
                       NULL, NULL, NULL, TRUE, FALSE,
                       NULL, TRUE);

PXTblClose( )        DbiCloseCursor(&tabH);

PXTblCopy( )         DbiCopyTable(hDb, TRUE, szSrcName,
                                   szPARADOX, szDestName);

PXTblCreate( )       DbiCreateTable(hDb, TRUE, &TblDesc);

PXTblCreateMode( )   DbiCreateTable(hDb, TRUE, &TblDesc);
(Use the optional parameters to change the
 level of the table.)

PXTblDecrypt( )      Only Supported for Paradox Tables
Example for use with 'C':
DBIResult decryptTable(hDBIDb hDb,
                       const char *tableName)
{
    CRTblDesc    crTblDesc;
    DBIResult    lastError;

    // Clear the buffer
    memset(&crTblDesc, 0,
           sizeof(CRTblDesc));

    // name of the table
    strcpy(crTblDesc.szTblName, tableName);

    lastError = DbiDoRestructure(hDb, 1,
                                  &crTblDesc,
                                  NULL, NULL,
                                  NULL,
                                  FALSE);
}

```



```

{
    BOOL          exists = FALSE;
    BOOL          isLocal = TRUE;
    hDBICur      tabH    = 0;
    char         *tblName;
    TBLBaseDesc  tblDesc;
    CHAR         remoteName[DBIMAXNAMELEN+1];
    DBIResult    lastError;

    if ((!strcmp(tableType, szPARADOX)) ||
        (!strcmp(tableType, szDBASE)) ||
        (!strcmp(tableType, szASCII)))
    {
        tblName = _fstrtok((pCHAR)tableName,
                          ".");
        isLocal = TRUE;
    }
    else if ((!strcmp(tableType, "ORACLE"))
             ||
             (!strcmp(tableType, "SYBASE")))
    {
        strcpy(remoteName, userName);
        strcat(remoteName, ".");
        strcat(remoteName, tableName);
        tblName = (pCHAR)tableName;
        isLocal = FALSE;
    }
    else
    {
        strcpy(remoteName, tableName);
        tblName = (pCHAR)tableName;
        isLocal = FALSE;
    }

    if (tblName == NULL)
    {
        lastError = DBIERR_INVALIDTABLENAME;
        return FALSE;
    }

    lastError = DbiOpenTableList(hDb, FALSE,
                                  TRUE, "*.*",
                                  &tabH);

    if (lastError != DBIERR_NONE)
    {

```

```

        return FALSE;
    }

    lastError = DbiSetToBegin(tabH);
    if (lastError != DBIERR_NONE)
    {
        return FALSE;
    }

    while ((DbiGetNextRecord(tabH,dbiNOLOCK,
                            (pBYTE)&tblDesc,
                            NULL))
           == DBIERR_NONE)
    {
        if (!stricmp(tblDesc.szName,
                    tblName))
        {
            // Check if the types match
            if (strcmp(tblDesc.szType,
                    tableName) &&
                (isLocal))
            {
                // keep searching it the
                // table is of the wrong
                // type
                continue;
            }
            lastError = DBIERR_NONE;
            DbiCloseCursor(&tabH);
            return TRUE;
        }
    }

    if (tabH)
    {
        DbiCloseCursor(&tabH);
    }

    lastError = DBIERR_NOSUCHTABLE;

    return exists;
}

```

PXTblMaxSize()

```

DbiCreateTable(hDb, TRUE, &TblDesc);
(This is done using the optional parameters

```



```

                                ".*.*", &tabH);
if (lastError != DBIERR_NONE)
{
    return FALSE;
}

lastError = DbiSetToBegin(tabH);
if (lastError != DBIERR_NONE)
{
    return FALSE;
}

while ((DbiGetNextRecord(tabH,
                        dbiNOLOCK,
                        (pBYTE)&tblDesc,
                        NULL))
        == DBIERR_NONE)
{
    if (!stricmp(
        tblDesc.tblBase.szName,
        tblName))
    {
        if (stricmp(
            tblDesc.tblBase.szType,
            tableType))
        {
            // Keep searching if the
            // type doesn't match
            continue;
        }
        if (
            tblDesc.tblExt.bProtected
            == TRUE)
        {
            // Table is protected
            lastError = DBIERR_NONE;
            DbiCloseCursor(&tabH);
            return TRUE;
        }
        else
        {
            lastError = DBIERR_NONE;
            DbiCloseCursor(&tabH);
            return FALSE;
        }
    }
}

```

```

        }
    }

    if (tabH)
    {
        DbiCloseCursor(&tabH);
    }

    lastError = DBIERR_NOSUCHTABLE;
}
else
{
    lastError = DbiOpenTable(hDb,
                            (pCHAR)tableName,
                            (pCHAR)tableType,
                            NULL, NULL, NULL,
                            dbiREADONLY,
                            dbiOPENSHARED,
                            xltNONE, FALSE,
                            NULL, &tabH);

    if (lastError == DBIERR_NONE)
    {
        DbiCloseCursor(&tabH);
        protect = FALSE;
        return protect;
    }
    else if (lastError ==
            DBIERR_INVALIDPASSWORD)
    {
        lastError = DBIERR_NONE;
        protect = TRUE;
        return protect;
    }
}

return protect;
}

```

PXTblRename()            DbiRenameTable(hDb, szOldName, szTableType, szNewName);

PXTblUpgrade()        DbiDoRestructure(hDb, 1, pTblDesc, pSaveAs, NULL, NULL, FALSE);  
 (Need to use the Optional Parameters to set



the level of the table. Local tables only.)

PXWinInit()

```
DbiInit(NULL);  
DbiOpenDatabase(NULL, NULL, dbiREADWRITE,  
                dbiOPENSHARED, NULL, 0,  
                NULL, NULL, &hDb);
```

Reference:

7/15/98 3:24:20 PM

Last Modified: 15-NOV-99

## Article #15152: Toggling the cursor ON/OFF

Technical Information Database

TI152D.txt - Toggling the cursor ON/OFF

Category :Turbo Pascal  
Platform :DOS  
Product :

### Description:

The following example routines are public domain programs that have been uploaded to our Forum on CompuServe. As a courtesy to our users that do not have immediate access to CompuServe, Technical Support distributes these routines free of charge.

However, because these routines are public domain programs, not developed by Borland International, we are unable to provide any technical support or assistance using these routines. If you need assistance using these routines, or are experiencing difficulties, we recommend that you log onto CompuServe and request assistance from the Forum members that developed these routines.

To turn the cursor on and off you must use BIOS interrupt \$10. For more information on this interrupt, please refer to a DOS programmer's guide. Of course, manipulating the cursor can only be done in text mode and not graphics mode.

The following procedure turns the cursor on and off in Turbo Pascal:

```
procedure SetCursor(On: Boolean);
var
  reg : Registers;
begin
  if On then { Turn cursor on }
  if Mem[$0040:$0049] = 7 then
    reg.cx := $B0C { If monochrome monitor }
  else
    reg.cx := $607 { If color monitor }
```

```
else                                { Turn cursor off }
  reg.cx := $2020;
  reg.bx := 0;
  reg.ax := $0100;    { Set the interrupt function }
  Intr($10,reg);     { Call the interrupt }
end; { procedure SetCursor }
```

This procedure was revised from the question and answer section in the Turbo Pascal Tutor and Turbo Pascal version 3.0 reference guide.

In the procedure you do not need to declare the type Registers because it is predefined for you in the DOS unit. Therefore, to use this procedure, your program must also use the DOS unit.

Reference:

3/30/99 3:59:41 PM

Last Modified: 01-SEP-99

## Article #15154: LINKING UNITS IN THE TURBO.TPL FILE

Technical Notes Database

TN154D.txt LINKING UNITS IN THE TURBO.TPL FILE

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0 5.0

### Description:

Q. The computer runs out of memory while linking a unit that's in TURBO.TPL to my program. What do I do?

A. The entire TURBO.TPL library file is loaded into memory, not just the unit, and therefore, no memory is left. The solution is to take the unit out of the library file.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15159: Some current internal limits of BDE

Some current internal limits of BDE

Description:

Here are the maximum limits for some common BDE objects. If you find you cannot reach these limits or are getting an out of memory error, increasing your SHAREDMEMSIZE in BDE Config to 4096 or more should allow you to reach these limits.

### GENERAL LIMITS

48	Clients in system
32	Sessions per client (3.5 and earlier 16 Bit, 32 Bit)
256	Session per client (4.0 32 Bit)
32	Open databases per session (3.5 and earlier 16 Bit, 32 Bit)
2048	Open databases per session (4.0 32 Bit)
32	Loaded drivers
64	Sessions in system (3.5 and earlier 16 Bit, 32 Bit)
12288	Sessions in system (4.0 32 Bit)
4000	Cursors per session
16	Entries in error stack
8	Table types per driver
16	Field types per driver
8	Index types per driver
48K	Size of configuration (IDAPI.CFG) file
64K	Size of SQL statement (RequestLive=False)
4K	Size of SQL statement (RequestLive=True)
6K	Size of SQL statement (RequestLive=True) (4.01, 32 Bit)
16K	Record buffer size (SQL or ODBC)
31	Table and field name size in characters
64	Stored procedure name size in characters

- 16 Fields in a key
- 3 File extension size in characters
- 260 Table name length in characters
- 260 Path and file name length in characters
- 15 Significant Digits

## PARADOX LIMITS

- 127 Tables open per system (4.0 and earlier 16 Bit32 Bit)
- 254 Tables open per system (4.01 32 Bit)
- 64 Record locks on one table (16Bit) per session
- 255 Record locks on one table (32Bit) per session
- 255 Records in transactions on a table (32 Bit)
- 512 Open physical files (4.0 and earlier 16 Bit32 Bit)  
(DB, PX,MB, X??, Y??, VAL,TV)
- 1024 Open physical files (4.01 32 Bit)  
(DB, PX,MB, X??, Y??, VAL,TV)
- 300 Users in one PDOXUSRS.NET file
- 255 Number of fields per table
- 255 Size of character fields
- 2 Billion records in a table
- 2 Billion bytes in .DB (Table) file
- 10800 Bytes per record for indexed tables
- 32750 Bytes per record for non-indexed tables
- 127 Number of secondary indexes per table
- 16 Number of fields in an index
- 255 Concurrent users per table
- 256 Megabytes of data per BLOB field
- 100 Passwords per session

- 15 Password length
- 63 Passwords per table
- 159 Fields with validity checks (32 Bit)
- 63 Fields with validity checks (16 Bit)
- 53 Fields with Validity checks added at onetime
- 63 Number of Sessions with Tables open on1 System

## DBASE LIMITS

- 256 Open dBASE tables per system (16 Bit)
- 350 Open dBASE tables per system (BDE 3.0 - 4.0,32 Bit)
- 512 Open dBASE tables per system (BDE 4.01, 32 Bit)
- 100 Record locks on one dBASE table (16 and 32 Bit)
- 100 Records in transactions on a dBASE table (32 Bit)
- 1 Billion records in a table
- 2 Billion bytes in .DBF (Table) file
- 4000 Size in bytes per record (dBASE 4)
- 32767 Size in bytes per record (dBASE for Windows)
- 255 Number of fields per table (dBASE 4)
- 1024 Number of fields per table (dBASE for Windows)
- 47 Number of index tags per .MDX file
- 254 Size of character fields
- 10 Open master indexes (.MDX) per table
- 220 Key expression length in characters

Last Modified: 26-FEB-01

## Article #15161: Extend File Handles allowing more than 15.

Technical Information Database

TI161D.txt      Extend File Handles allowing more than 15.

Category      :General Programming

Platform      :All

Product       :Pascal    All

### Description:

The following is public domain information that has been uploaded to our Forum on CompuServe. As a courtesy to our users that do not have immediate access to CompuServe, Technical Support distributes these routines free of charge.

However, because these routines are public domain programs, not developed by Borland International, we are unable to provide any technical support or assistance using these routines. If you need assistance using these routines, or are experiencing difficulties, we recommend that you log onto CompuServe and request assistance from the Forum members that developed these routines.

The following unit extends the maximum number of files that can be open simultaneously from 20 to 255. Files in DOS 2.0 or later are controlled by file handles. The number of file handles available to application programs is controlled by the FILES environment variable stored in a CONFIG.SYS file. If no FILES variable is established in a CONFIG.SYS file, then only 8 file handles are available. However, DOS requires 5 file handles for its own use (controlling devices such as CON, AUX, PRN, etc). This le

aves only 3 handles for use by application programs.

By specifying a value for the FILES environment variable, you can increase the number of possible file handles from 8 up to 20. Since DOS still requires 5, 15 are left for application programs. But you cannot normally increase the number of handles beyond 20.



With DOS version 3.0, a new DOS function was added to increase the number of file handles available. However, the function must be called from application programs that have previously reserved space for the new file handles.

The unit EXTEND, described below, resizes the amount of allocated memory for a Turbo Pascal program to allow space for new file handles. In doing so, it also resizes the heap by adjusting the value of FreePtr, the pointer used in FreeList management. Since the FreeList is being manipulated, the heap must be empty when the EXTEND unit is initialized. This can be guaranteed by including EXTEND as one of the first units in your program's USES statement. If any heap has been allocated when EXTEND initialize s, the program will halt with an error message.

Notice that the interface section of the EXTEND unit is empty. The initialization section of the unit takes care of the unit's entire function. Other than including EXTEND in a program's USES statement, no other operation need be preformed.

Before using EXTEND, you must specify a FILES variable in the CONFIG.SYS file such as the following:

```
FILES = 255
```

Refer to your DOS User's Guide for more information regarding the CONFIG.SYS file and the FILES variable.

The unit EXTEND is set up for 255 file handles. You can reduce this number by changing the HANDLES constant in the unit's source code and reducing the number specified for the FILES environment variable.

```
unit Extend;
{This extends the number of file handles from 20 to 255}
{DOS requires 5 for itself. Applications can use up to 250}

interface

implementation
uses dos;
const
    Handles =255;
```

```
{You can reduce the value passed to Handles if fewer files}
{are required.}
```

```
var
    reg:registers;
begin
    {Check the Dos Version}
    {This technique only works for DOS 3.0 or later}
    reg.ah:=$30;
    MsDos(reg);
    if reg.al<3 then
        begin
            writeln('Extend Unit Require DOS 3.0 or greater');
            halt(1);
        end;

        {Reset the FreePtr}
        {This reduces the heap space used by Turbo
Pascal}
    if ofs(FreePtr^)<>0 then
        {Checks to see if the Heap is empty}
        begin
            write('Heap must be empty before Extend unit
initializes');
            writeln;
            halt(1);
        end;
    FreePtr:=ptr(seg(FreePtr^)-(Handles div 8 +1), ofs(FreePtr^));

        {Determine how much memory is allocated to
program}
    {Reg.Bx will return how many paragraphs used by program}
    reg.ah:=$4A;
    reg.es:=PrefixSeg;
    reg.bx:=$FFFF;
    msdos(reg);

        {Set the program size to the allow for new
handles}
    reg.ah:=$4A;
    reg.es:=PrefixSeg;
    reg.bx:=reg.bx-(Handles div 8 +1);
    msdos(reg);

        {Error when a Block Size is not appropriate}
```

```

if (reg.flags and 1)=1 then
    begin
        Writeln('Runtime Error ',reg.ax);
        Writeln('In the Extend Unit');
        halt(1);
    end;

        {Allocate Space for Additional Handles}

reg.ah:=$67;
reg.bx:=Handles;
MsDos(reg);
end.

```

Write the following program to a separate file. This program tests the extend unit. This test should be done on systems equipped with a hard disk.

```

program TestEx;
uses extend;
type
    filearray=array[1..250] of text;
var
    f:^filearray;
    i:integer;
    s:string;

begin
        {Allocate Space for File Variable Table}
new(f);

        {Open 250 files simultaneously}
for i:=1 to 250 do
    begin
        str(i,s);
        Assign(f^[i],'Dum'+s+'.txt');
        rewrite(f^[i]);
        writeln('Open #',s);
    end;

        {Write some text to the files}
for i:=1 to 250 do
    write(f^[i],'This is a test file');

        {Close the files}
for i:=1 to 250 do
    begin

```

```
close(f^[i]);  
writeln('Closing #',i);  
end;
```

```
                                {Erase the files}  
for i:=1 to 250 do  
  begin  
    erase(f^[i]);  
    writeln('Erasing #',i);  
  end;
```

end.

Reference:

7/16/98 4:33:38 PM

Last Modified: 01-SEP-99

## Article #15163: TINST - NEW SETTINGS AREN'T USED

Technical Notes Database

TN163D.txt      TINST - NEW SETTINGS AREN'T USED  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 5.0 5.5

### Description:

Q. Why aren't the new settings used after I install TURBO.EXE using the TINST.EXE program?

A. You probably have a .TP file in the current or Turbo directory being loaded and the settings in the .TP file override the settings installed by TINST. Delete the .TP file.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15168: Local SQL Reserved Words.

Technical Information Database

TI168B.txt Local SQL Reserved Words.

Category :General Programming

Platform :All

Product :BDE All

### Description:

This file contains an alphabetical list of words reserved by Local SQL in the Borland Database Engine. Note that this file is provided as-is.

ACTIVE, ADD, ALL, AFTER, ALTER, AND, ANY, AS, ASC, ASCENDING, AT, AUTO, AUTOINC, AVG

BASE\_NAME, BEFORE, BEGIN, BETWEEN, BLOB, BOOLEAN, BOTH, BY, BYTES

CACHE, CAST, CHAR, CHARACTER, CHECK, CHECK\_POINT\_LENGTH, COLLATE, COLUMN, COMMIT, COMMITTED, COMPUTED, CONDITIONAL, CONSTRAINT, CONTAINING, COUNT, CREATE, CSTRING, CURRENT, CURSOR

DATABASE, DATE, DAY, DEBUG, DEC, DECIMAL, DECLARE, DEFAULT, DELETE, DESC, DESCENDING, DISTINCT, DO, DOMAIN, DOUBLE, DROP

ELSE, END, ENTRY\_POINT, ESCAPE, EXCEPTION, EXECUTE, EXISTS, EXIT, EXTERNAL, EXTRACT

FILE, FILTER, FLOAT, FOR, FOREIGN, FROM, FULL, FUNCTION

GDSCODE, GENERATOR, GEN\_ID, GRANT, GROUP, GROUP\_COMMIT\_WAIT\_TIME

HAVING, HOUR

IF, IN, INT, INACTIVE, INDEX, INNER, INPUT\_TYPE, INSERT, INTEGER, INTO, IS, ISOLATION

JOIN

KEY

LONG, LENGTH, LOGFILE, LOWER, LEADING, LEFT, LEVEL, LIKE,  
LOG\_BUFFER\_SIZE

MANUAL, MAX, MAXIMUM\_SEGMENT, MERGE, MESSAGE, MIN, MINUTE,  
MODULE\_NAME, MONEY, MONTH

NAMES, NATIONAL, NATURAL, NCHAR, NO, NOT, NULL,  
NUM\_LOG\_BUFFERS, NUMERIC

OF, ON, ONLY, OPTION, OR, ORDER, OUTER, OUTPUT\_TYPE, OVERFLOW

PAGE\_SIZE, PAGE, PAGES, PARAMETER, PASSWORD, PLAN, POSITION,  
POST\_EVENT, PRECISION, PROCEDURE, PROTECTED, PRIMARY,  
PRIVILEGES

RAW\_PARTITIONS, RDB\$DB\_KEY, READ, REAL, RECORD\_VERSION,  
REFERENCES, RESERV, RESERVING, RETAIN, RETURNING\_VALUES,  
RETURNS, REVOKE, RIGHT, ROLLBACK

SECOND, SEGMENT, SELECT, SET, SHARED, SHADOW, SCHEMA,  
SINGULAR, SIZE, SMALLINT, SNAPSHOT, SOME, SORT, SQLCODE,  
STABILITY, STARTING, STARTS, STATISTICS, SUB\_TYPE, SUBSTRING,  
SUM, SUSPEND

TABLE, THEN, TIME, TIMESTAMP, TIMEZONE\_HOUR, TIMEZONE\_MINUTE,  
TO, TRAILING, TRANSACTION, TRIGGER, TRIM

UNCOMMITTED, UNION, UNIQUE, UPDATE, UPPER, USER

VALUE, VALUES, VARCHAR, VARIABLE, VARYING, VIEW

WAIT, WHEN, WHERE, WHILE, WITH, WORK, WRITE

YEAR

OPERATORS:

||, -, \*, /, <>, <, >, ,(comma), =, <=, >=, ~=, !=, ^=, (, )

Reference:

7/15/98 3:24:21 PM

Last Modified: 01-SEP-99



## Article #15170: Network File Access

Technical Information Database

TI170D.txt    Network File Access  
Category    :General Programming  
Platform    :All  
Product     :Pascal    All

Description:

### FILE AND RECORD LOCKING IN TURBO PASCAL

Turbo Pascal does not have built-in support for file and record locking. It does, however, have the ability to open files for shared access. This is accomplished by setting the value of the system variable FileMode. By default, FileMode has a value of 2 which causes files opened to have single-user, read/write access. A shared, read/write mode can be had by setting FileMode to a value of 66. Whereas, a shared, read only mode is represented by the value 64.

Record and file locking is a bit more difficult. DOS version 3.0 and later provide record locking through the use of Interrupt 21H, Service 5CH. This service may be accessed through Turbo Pascal's procedure MsDos defined in the Dos unit.

Here's a routine designed to lock and unlock portions of any untyped or type file. Note, these routines may not be used with text files.

```
procedure LockBlock(var F; Offset: Longint; Length: Longint;
Lock: Boolean);
type
  LoHiRec = record
    LoWord, HiWord: Word;
  end;
var
  RecordSize: Word;
  Handle: Integer;
  File: FileRec absolute F;
  R: Registers;
```

```
begin
  Handle := File.Handle;
  RecordSize := File.RecSize;
  Offset := Offset * RecordSize;
  R.CX := LoHiRec(Offset).LoWord;
  R.DX := LoHiRec(Offset).HiWord;
  R.SI := LoHiRec(Length).LoWord;
  R.DI := LoHiRec(Length).HiWord;
  R.AH := $5C;
  if Lock then R.AL := 0 else R.AL := 1;
  MsDos(R);
  if (R.Flags and FCarry) <> 0 then
  begin
    RunError(R.AX);
  end;
end;
```

The parameter 'offset' indicates what record is to mark the beginning of the file lock. 'Length' determines how many records will be locked from this point. The 'lock' parameter determines whether the portion of the file is to be locked or unlocked.

To lock an entire file, pass 0 for offset and the results of the FileSize function for the number of records to be locked.

This routine will function as expected when using SHARE and/or MS-DOS compatible network software.

Reference:

7/16/98 4:33:38 PM

Last Modified: 01-SEP-99

## Article #15172: EMS SUPPORT

Technical Notes Database

TN172D.txt      EMS SUPPORT  
Category      :Pascal  
Platform       :All  
Product       :Turbo Pascal 5.0 5.5

### Description:

Q. Does Turbo Pascal 5.0+ support EMS?

A. Yes. Turbo Pascal 5.0+ will use up to 64K (if selected in TINST) of EMS for storing the edit buffer. In addition, you can instruct the Overlay unit to place your overlaid units on EMS. Finally, EMS.PAS on the distribution disk shows you how to access EMS memory.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15177: Introduction to BDE Programming

Technical Information Database

TI177B.txt Introduction to BDE Programming

Category :General Programming

Platform :All

Product :BDE All

### Description:

Introduction to BDE Programming

BDE v2.0

#### I) Project Setup

This section covers the basics of what needs to be done in order to set up a Borland Database Engine project or makefile.

#### II) Design Overview

A high level overview of the steps that are required to create a simple application which retrieves fields from a table

#### III) Design Specifics

Detailed description, with code examples, of the steps described in part II.

#### IV) Error Handler

This section contains the error handler used by this example.

#### I) Project Setup

=====

This document provides a quick introduction to programming with the Borland Database Engine. By the end of this section, you should have a simple EasyWin, BDE example which gets a record from a table and displays the first two fields.

The following steps must be followed when starting to write a

BDE application.

First, set up the project or makefile

MAIN.CPP	- File to contain your code
DBERR.CPP	- Error handling routine
IDAPI.LIB	- BDE Import Library
MAIN.DEF	- Module Definition file

Select the large memory model.

For this simple application, set the target to be an EasyWin application. This way we don't have to deal with any Windows UI issues.

It is also suggested to install the IDAPI.TOK file so as to have syntax highlighting for BDE functions and types (directions on this are included in the \BDE\DOC\IDAPITOK.TXT file).

The on-line help file, IDAPI.HLP, can also be incorporated into the BC 4.5 OpenHelp architecture, allowing context sensitive help on IDAPI types and functions.

Other required setup:

-----

The Borland Database Engine is fairly stack intensive, especially when doing batch operations and queries, so it is generally recommended that the stack be set to a minimum of 25k (for Windows C/C++ applications, the stack size is set in the module definition file)

Increase the number of file handles available to the application. By default, the BDE assumes that it has access to 48 physical file handles (set by the BDE Configuration utility, System page | MAXFILEHANDLES option, 5-255). Due to this default setting, it is usually best to set the number of file handles available to an application to a minimum of 68 using the Windows API function SetHandleCount. Seemingly random errors can occur when the BDE does not have access to enough file handles.

Use the Debug Layer when developing applications. The debug layer performs much stricter error checking than the regular DLL, resulting in fewer GP faults and less re-booting of your machine. It will also produce a trace output of which IDAPI functions were called by an application. Note that use of the debug layer

requires the use of both the Debug DLL (Set using the DLLSwap utility), as well as a call to the DbgDebugLayerOptions function.

Make certain to compile with 'Allocate enums as ints' selected (in the BC 4.x IDE, Options | Project | Compiler | Code Generation). A number of structures, such as CURProps, make use of enumerations. Within the DLL, these are allocated as two byte values. Turning off this option will result in your code passing only one byte. This error generally manifests itself with stack corruption problems, such as GP faults when calling or returning from a function.

Within a module to contain BDE code, include the following header files:

WINDOWS.H  
IDAPI.H

Note that order is important. The WINDOWS.H file must be included before IDAPI.H.

## II) Design Overview

=====

Before we start writing code, a brief overview of what needs to be done to get a record from a tables:

- Increase the number of file handles available to the application
- Initialize the Borland Database Engine
- Enable the debug layer
- Open a database object
- Set the database object to point to the directory containing the table
- Set the directory for temporary objects
- Open a table, creating a cursor object
- Get the properties of the table
- Using these properties, allocate memory for a record buffer
- Position the cursor on the desired record
- Get the desired record from the cursor (table)
- Get the desired field(s) from the record
- Free all resources

Note that throughout the following short example unfamiliar variable types may be used. These are IDAPI variable types that are defined in the IDAPI.H header file. For example, BYTE,

BOOL, CHAR, and FLOAT.

### III) Design Specifics

=====

Increase the number of file handles available to the application

-----

A call to the Windows API function SetHandleCount will increase the number of file handles available to an application in the Windows environment:

```
int    HandlesAvail;
int    HandlesWanted;

HandlesAvail = 0;
HandlesWanted = 68;

HandlesAvail = SetHandleCount(HandlesWanted);
if (HandlesAvail < HandlesWanted)
{
    // Display message re: not enough available
    // file handles
    return;
}
```

Note that in non-trivial cases it is recommended to determine the number of file handles that are specified in the BDE configuration file. This would be done using the DbOpenCfgInfoList function, which will be described in a later section on configuration management.

Initialize the Borland Database Engine

-----

The Borland Database Engine is initialized using the DbInit function:

```
CHKERR(DbInit(NULL));
```

Note that the CHKERR macro is defined in the DBERR.H file, which is a part of the Error Handling module. The call to DbInit allocates system resources for the client.

Enable the Debug layer

-----  
The following code is used to enable the debug layer,  
outputting trace information to a text file on disk.

```
DbiDebugLayerOptions(DEBUGON | OUTPUTTOFILE, "MYTRACE.INF");
```

Note that in certain situations, you may also want to use the  
FLUSHEVERYOP flag, which will force output to the trace  
file after every operation. While this is slower, it is useful  
when a GP fault occurs....

Open a database object  
-----

The next step that is needed is to open a database object. All  
table access must be performed within the context of a  
database. Local databases generally use what is referred to as  
the "STANDARD" database, which is what we will be using in this  
example. The preferred method would be to create an alias to a  
local directory and using that as the database. This allows for  
easy modification in the future if one day it is decided to move  
the application from using dBASE tables to using InterBase  
tables. The DbiOpenDatabase function is used to open a database:

```
hDBIDb   hDb;       // Handle to the Database
```

```
hDb = 0; // Initialize to zero for cleanup  
        // purposes
```

```
CHKERR_CLEANUP(  
    DbiOpenDatabase(NULL, // Database name  
                    // - NULL for standard Database  
    NULL, // Database type -  
          // NULL for standard Database  
    dbiREADWRITE, // Open mode - Read/Write or  
                  // Read only  
    dbiOPENSARED, // Share mode - Shared or  
                  // Exclusive  
    NULL, // Password - not needed for the  
          // STANDARD database  
    NULL, // Number of Optional Parameters  
    NULL, // Field Desc for Optional  
          // Parameters  
    NULL, // Values for the
```



```
        // Optional Parameters
        &hDb) // Handle to the database
    );
```

```
// At the end of the function
```

```
CleanUp:
```

```
    // Close only if open
    if (hDb != 0)
    {
        DbtCloseDatabase(&hDb);
    }
```

```
Set the database object to point to the directory containing
```

```
-----
the table
-----
```

The working directory next needs to be set to the directory containing the table. While it is possible to open a table in other directories, specifying the absolute path, it is generally recommended to open tables in the working directory, as a number of operations, such as getting a list of available tables, work off the current directory. The `DbtSetDirectory` function is used to set the working directory (using the default location of the BDE example tables):

```
CHKERR_CLEANUP(
    DbtSetDirectory(hDb, // Handle to the
                    // database which is being modified
                    "C:\\BDE\\EXAMPLES\\TABLES")
                    // The new working directory
    );
```

Note that the full, absolute path needs to be used. Relative paths are not supported.

```
Set the directory for temporary objects
-----
```

While not all BDE applications create temporary objects, but larger applications will at one time or another. For example, the result set from a query or the records which cause a key violation in a restructure will be placed in a temporary table. By default, this temporary, or private directory, as it is called, is the startup directory. This will cause a problem if

the application is run off a network or a CD-ROM., as the directory cannot be shared, and it must be writable. The DbiSetPrivateDir function is used to set the private directory for a client:

```
CHKERR_CLEANUP(  
    DbiSetPrivateDir("c:\\")  
        // Select a directory on a local drive that is not  
        // used by other applications.  
        );
```

Note that the full, absolute path needs to be used. Relative paths are not supported.

Open a table, creating a cursor object

-----

Next, open the table. Upon opening a table, a cursor object is created and returned to the calling application. A cursor object is basically an abstraction which allows queries and tables to be accessed in the same method:

```
hDBICur hCur;    // Handle to the cursor (table)  
CHAR     szTblName[DBIMAXNAMELEN]; // Table name - DBIMAXNAMELEN  
        // is defined in IDAPI.H  
CHAR     szTblType[DBIMAXNAMELEN]; // Table Type  
  
hCur = 0; // Initialize to zero for cleanup  
        // purposes  
  
strcpy(szTblName, "customer"); // Name of the table  
strcpy(szTblType, szPARADOX); // Type of the tables  
        // - szPARADOX is defined in IDAPI.H
```

```
CHKERR_CLEANUP(  
    DbiOpenTable(hDb, // Handle to the  
        // database to which this cursor will  
        // be related  
        szTblName, // Name of the table  
        szTblType, // Type of the table - only  
        // used for local tables  
        NULL, // Index Name - Optional  
        NULL, // IndexTagName - Optional.  
        // Only used by dBASE  
        0, // IndexId - 0 = Primary. Optional.
```

```

        // Paradox and SQL only
dbiREADWRITE, // Open Mode -
        // Read/Write or Read Only
dbiOPENSHARED, // Shared mode -
        // SHARED or EXCL
xltFIELD, // Translate mode
        // FIELD or NONE
        // FIELD: Convert data from table
        // format to C format
        // NONE: Leave data in it's native
        // state
FALSE, // Unidirectional -
        // False means can navigate forward
        // and back.
NULL, // Optional Parameters.
&hCur) // Handle to the cursor
);

```

CleanUp:

```

// Close only if open
if (hCur != 0)
{
    DbicloseCursor(&hCur); // Note the use of DbicloseCursor
                          // - there is no DbicloseTable.
}

```

Get the properties of the table

-----

We next need to determine the size of the record buffer for the table. This information is gotten from the cursor via the DbigetCursorProps function. The Cursor properties include information on the table name, size, type, number of fields, and record buffer size. More information can be found on cursor properties in the on-line help under CURProps.

```

CURProps    curProps; // Properties of the cursor

```

```

CHKERR_CLEANUP(
    DbigetCursorProps(hCur, // Handle to the cursor
                      &curProps) // Properties of the cursor
                      // (table)
);

```

curProps.iRecBufSize contains the size of the record buffer.

Using these properties, allocate memory for a record buffer

```
pBYTE      pRecBuf; // Pointer to the record buffer

pRecBuf = NULL; // For cleanup purposes

// Allocate memory for the record buffer
pRecBuf = (pBYTE)malloc(curProps.iRecBufSize * sizeof(BYTE));
if (pRecBuf == NULL)
{
    // Display some error message
    goto CleanUp;
}

CleanUp:
    if (pRecBuf)
    {
        free(pRecBuf);
    }
```

Position the cursor on the desired record

-----

The `DbiSetToBegin()` function is used to position the cursor to the "crack" before the first record in the table. Crack semantics basically allow for the current location of the cursor to be before the first record, between records, or after the last record. One of the reasons for having crack semantics is to allow the use of one function to access all records in a table. For example, in place of having to use `DbiGetRecord` the first time, and `DbiGetNextRecord` each subsequent time, it is now possible to use `DbiGetNextRecord` to get all records in a table.

```
CHKERR_CLEANUP(
    DbiSetToBegin(hCur) // Position the specified cursor
                        // to the crack before the first record
    );
```

Get the desired record from the cursor (table)

-----

The `DbiGetNextRecord` function is normally used to get a record from a table. Note that the current record of the cursor is set to the record returned by this function (the next record

in the table):

```
CHKERR_CLEANUP(  
    DbiGetNextRecord(hCur,          // Cursor from which to get the  
                                // record.  
        dbiNOLOCK, // Lock Type  
        pRecBuf,   // Buffer to store the record  
        NULL)     // Record properties -  
                // don't need in this case  
    );
```

Get the desired field(s) from the record

-----

The last step is to get the field values out of the record buffer and into some local variable. Note that in this case we are making assumptions about which field is at which ordinal position within the table, as well as the size of the field. In general, it is recommended to use `DbiGetFieldDescs` to get information about a field before retrieving it. Also note that a single function, `DbiGetField`, is used to get all fields, other than BLOBs, from a table.

```
FLOAT      custNum;  
BOOL       isBlank;
```

```
CHKERR_CLEANUP(  
    DbiGetField(hCur,          // Cursor which contains the record  
                1,             // Field Number of the "Customer" field.  
                pRecBuf, // Buffer containing the record  
                (pBYTE)&custNum, // Variable for the Customer  
                // Number  
                isBlank) // Is the field blank?  
    );
```

Free all resources

-----

After all desired operations have been performed, the resources allocated on behalf of the application need to be cleaned up. In addition to any memory explicitly allocated using `malloc` or `new`, all engine objects must also be cleaned up, including the cursor, database, and engine:

CleanUp:

```

if (pRecBuf)
{
    // Free the record buffer
    free(pRecBuf);
}
if (hCur !=0)
{
    // Close the cursor
    DbiCloseCursor(&hCur);
}
// Close only if open
if (hDb != 0)
{
    // Close the database
    DbiCloseDatabase(&hDb);
}

// Close the BDE.
DbiExit();

```

#### IV) Error handler

```

// DBERR.H

// This file contains macros to use around BDE functions
// which are not expected to fail, or which would
// result in a fatal error. More explicit error handling needs
// to be done in the application when a non-fatal
// error can occur, for example, getting an End of file error,
// DBIERR_EOF, from a DbiGetNextRecord
// loop.

#ifndef __DBERR_H
#define __DBERR_H

#include

// Macro to use when no cleanup is required

#define CHKERR(parm) DBIError(__FILE__, __LINE__, \
                             #parm, parm) ; \
                             if (GlobalDBIErr) { \
                                 return GlobalDBIErr ;}

// Macro to use when cleanup must be done within a function

```

```
// for example, to close open tables or to free allocated memory.
// Note: must have a CleanUp label defined in each function
// where this macro is used. The CleanUp
// label should start the section of a function which frees the
// resources allocated within a given function
```

```
#define CHKERR_CLEANUP(parm) DBIError(__FILE__, __LINE__, \
                                     #parm, parm) ; \
                                     if (GlobalDBIErr) { \
                                         goto CleanUp ;}
```

```
// Global variable to contain the result code
extern DBIResult GlobalDBIErr;
```

```
// Prototype for the error handling function
DBIResult DBIError(pCHAR, UINT16, pCHAR, DBIResult);
```

```
#endif
```

```
// DBERR.C
```

```
#include
#include "DBERR.h"
```

```
// The way to use this macro is to include the DBERR.H.
// Then pass an IDAPI function as a parameter to the
// macro:
```

```
//
// #define CHKERR(parm) DBIError(__FILE__, __LINE__, \
//                               #parm, parm) ; \
//                               if ( GlobalDBIErr ) { \
//                                   return GlobalDBIErr ;}
```

```
// You would then use it as such:
//     CHKERR(DbiCreateTable(hDb, bOverWrite,
//                          &crTblDsc)) ;
```

```
// Global variable to hold return value from BDE functions
DBIResult GlobalDBIErr;
```

```
// Global variables to contain the error messages. Defined
// globally to ensure that an error message can be displayed
// even if the system is out of memory (and to keep it off
// the stack)
```

```

// Contains just the BDE error message
static char szDBIStatus[(DBIMAXMSGLEN * 7)+1];

// Contains the entire message, including file name and line
// number of the offending function.
static char szMessage[(DBIMAXMSGLEN * 7)+1+110];

//=====
// Function:
//         DBIError();
//
// Input:  module name (pCHAR), line number (UINT16),
//         Engine function name (pCHAR),
//         Result (DBIResult)
//
// Return: A DBIResult value.
//
// Description:
//         This is a function which takes in the information of
//         where the error occurred and displays error
//         information in a message box.
//=====
DBIResult
DBIError (pCHAR module, UINT16 line, pCHAR function,
         DBIResult retVal)
{
    DBIErrInfo  ErrInfo; // Structure to contain the error
                    // information

    if (retVal == DBIERR_NONE)
    {
        GlobalDBIErr = DBIERR_NONE;
        return retVal;
    }
    // Don't want to call functions if the DLL is not there....
    if (retVal != DBIERR_CANTFINDODAPI)
    {
        // Get as much error information as possible
        DbiGetErrorInfo(TRUE, &ErrInfo);

        // Make certain information is returned on the correct
        // error (that this function was called
        // immediately after the function that caused the error.
        if (ErrInfo.iError == retVal)

```



```

{
    strcpy(szDBIStatus, ErrInfo.szErrCode);

    if (strcmp(ErrInfo.szContext1, ""))
    {
        strcat(szDBIStatus, ErrInfo.szContext1);
    }
    if (strcmp(ErrInfo.szContext2, ""))
    {
        strcat(szDBIStatus, ErrInfo.szContext2);
    }
    if (strcmp(ErrInfo.szContext3, ""))
    {
        strcat(szDBIStatus, ErrInfo.szContext3);
    }
    if (strcmp(ErrInfo.szContext4, ""))
    {
        strcat(szDBIStatus, ErrInfo.szContext4);
    }
}
else {
    DbiGetErrorString(retVal, szDBIStatus);
}

sprintf(szMessage,
        "Module:\t\t%s\nFunction:\t\t%s\nLine:\t\t\t%d\n"
        "Category:\t\t%d\nCode:\t\t\t%d\nError:\r\n\r\n%s\n",
        module, function, line, ErrCat(retVal),
        ErrCode(retVal), szDBIStatus);

    MessageBox(NULL, szMessage, "BDE Error",
                MB_ICONEXCLAMATION);
}
else
{
    // Display an error message if the BDE DLL's cannot be
    // found.
    MessageBox(NULL, "Cannot find Borland Database"
                " Engine files: Check the [IDAPI] section"
                " in WIN.INI.", "BDE Initialization Error",
                MB_ICONHAND | MB_OK);
}

GlobalDBIErr = retVal;
return retVal;

```

}

Reference :

7/15/98 3:24:21 PM

Last Modified: 01-SEP-99

**Article #15179: editors.pas 3,4,5,6 entry patch for TP6.**

Technical Information Database

TI179D.txt - editors.pas 3,4,5,6 entry patch for TP6.

Category :Turbo Pascal  
Platform :All  
Product :

Description:

The following patch fixes the problem of not being able to enter numbers 3 through 6 into the editor of the EDITORS.PAS example shipped with Turbo Pascal 6.0.

Open EDITORS.PAS in the TURBO editor and search for the EXISTING code. Once that is found change the EXISTING code to the CHANGE TO code. Save your changes and recompile EDITORS.PAS.

EXISTING:

```
LODSW
OR      BL,BL
JE
```

2

```
CMP     BL,DL
```

CHANGE TO:

```
LODSW
CMP     BL,DL
```

EXISTING:

```
var
  Key: Word;
```

CHANGE TO:

```
var
  ShiftState: Byte absolute
    $40:$17;
  Key: Word;
```

-----  
EXISTING:

```
JE
```

-----  
EXISTING:

```
4          begin

2:      CMP      BH,DH          Key := Event.KeyCode;
        JE

4

CHANGE TO:          CHANGE TO:

        begin

        JE

4          if (ShiftState and $03 <> 0)
        CMP      BH,DH          and (Event.ScanCode >= $47)
        JE

4          and (Event.ScanCode <= $51)
        then
            Event.CharCode := #0;
            Key := Event.KeyCode;
```

Reference:

3/30/99 4:02:24 PM

Last Modified: 01-SEP-99

## Article #15181: OUT OF MEMORY ERROR

Technical Notes Database

TN181D.txt      OUT OF MEMORY ERROR  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 4.0+

### Description:

Q. How do I avoid an Out of Memory error?

A. This error occurs when the compiler has run out of memory.

There are a number of solutions to this problem:

1. If Compiler/Destination is set to Memory, set it to Disk in the integrated environment.
2. If Options/Compile/Link buffer in the integrated environment is set to Memory, set it to Disk. Alternatively, if you're using 4.0, place a {\$L-} directive at the beginning of your program. Use the /L option to link to disk in the command-line compiler.
3. If you are using any memory-resident utilities, such as Sidekick and Superkey, remove them from memory.
4. If you are using TURBO.EXE, try using TPC.EXE instead - it takes up less memory.
5. Turn off any compiler directives which are not necessarily needed. By simply turning off range checking {\$R-} and software emulation {\$E-}, your code size will be reduced dramatically.
6. Move all units, except PRINTER.TPU, out of the TURBO.TPL file and into the installed units directory.

If none of these suggestions help, your program or unit may simply be too large to compile in the amount of memory available, and you may have to break in into two or more smaller units. Alternatively, if you're using 5.0+, you should consider using overlays.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15186: Comparison filters.

Technical Information Database

TI186B.txt      Comparison filters.  
Category        :General Programming  
Platform        :All  
Product         :BDE All

Description:

/\*

This example shows how to set a filter on the first N characters of a character field. To use this example, backup the FILTER.C file in the BDE\EXAMPLES\SNIPIT directory. Copy this file to FILTER.C in that directory. Then compile and run SNIPIT, running the 'Cursor: Setting and Using filters' example.

\*/

// BDE - (C) Copyright 1994 by Borland International

// Filter.c

#include "snipit.h"

static const char szTblName[] = "customer";

static const char szTblType[] = szPARADOX;

//=====

// Function:

//            Filter();

//

// Description:

//            This example shows how to use filters to limit the

//            result set of a table. Filters perform a function

//            similar to that of ranges, but more powerful

//            operations are supported.

//=====

void

Filter (void)

{

```

hDBIDb          hDb;          // Handle to the database.
hDBICur         hCur;        // Handle to the table.
pBYTE          pcanExpr;      // Structure containing
                                     // filter info.
hDBIFilter      hFilter;      // Filter handle.
UINT16         uSizeNodes;    // Size of the nodes in the
                                     // tree.
UINT16         uSizeCanExpr;  // Size of the header
                                     // information.
UINT16         uSizeLiterals; // Size of the literals.
UINT16         uTotalSize;    // Total size of the filter
                                     // expression.
UINT32         uNumRecs = 10; // Number of records to
                                     // display.
CANExpr        canExp;       // Contains the header
                                     // information.
UINT16         Nodes[] =     // Nodes of the filter
                                     // tree.
{
    // Offset 0
    nodeCOMPARE, // canCompare.nodeClass
    canEQ,       // canCompare.canOp
    1,           // canCompare.bCaseInsensitive
    5,           // canCompare.iParialLen (0 if
                // full length)
    12,         // canBinary.iOperand1
    20,         // canBinary.iOperand2
                // Offsets in the Nodes array

    // Offset 12
    nodeFIELD,  // canField.nodeClass
    canFIELD,   // canField.canOp
    2,          // canField.iFieldNum
    0,          // canField.iNameOffset: szField
                // is the literal at offset 0

    // Offset 20
    nodeCONST,  // canConst.nodeClass
    canCONST,   // canConst.canOp
    fldZSTRING, // canConst.iType
    6,          // canConst.iSize
    5           // canConst.iOffset: szConst is
                // the literal at offset
                // strlen(szField) + 1
};

```



```

// Name of the field for the third node of the tree.
CHAR          szField[] = "NAME";
// Value of the constant for the second node of the tree.
CHAR          szConst[] = "OCEAN";
// Return value from IDAPI functions.
DBIResult     rslt;

Screen("*** Filter Example ***\r\n");

BREAK_IN_DEBUGGER();

Screen("    Initializing IDAPI...");
if (InitAndConnect(&hDb) != DBIERR_NONE)
{
    Screen("\r\n*** End of Example ***");
    return;
}

Screen("    Setting the database directory...");
rslt = DbiSetDirectory(hDb, (pCHAR) szTblDirectory);
ChkRslt(rslt, "SetDirectory");

Screen("    Open the %s table...", szTblName);
rslt = DbiOpenTable(hDb, (pCHAR) szTblName,
                    (pCHAR) szTblType, NULL, NULL, 0,
                    dbiREADWRITE, dbiOPENSHARED, xltFIELD,
                    FALSE, NULL, &hCur);
if (ChkRslt(rslt, "OpenTable") != DBIERR_NONE)
{
    CloseDbAndExit(&hDb);
    Screen("\r\n*** End of Example ***");
    return;
}

// Go to the beginning of the table
rslt = DbiSetToBegin(hCur);
ChkRslt(rslt, "SetToBegin");

Screen("\r\n    Display the %s table...", szTblName);
DisplayTable(hCur, uNumRecs);

uSizeNodes     = sizeof(Nodes);    // Size of the nodes.
// Size of the literals.
uSizeLiterals  = strlen(szField) + 1 + strlen(szConst) + 1;

```

```

// Size of the header information.
uSizeCanExpr    = sizeof(CANExpr);
// Total size of the filter.
uTotalSize      = uSizeCanExpr + uSizeNodes + uSizeLiterals;

// Initialize the header information
canExp.iVer = 1;           // Version.
canExp.iTotalSize = uTotalSize; // Total size of the
                               // filter.
canExp.iNodes = 3;        // Number of nodes.
canExp.iNodeStart = uSizeCanExpr; // The offset in the
                               // buffer where the
                               // expression nodes
                               // start.

// The offset in the buffer where the literals start.
canExp.iLiteralStart = uSizeCanExpr + uSizeNodes;
// Allocate space for the filter expression.
pcanExpr = (pBYTE)malloc(uTotalSize * sizeof(BYTE));
if (pcanExpr == NULL)
{
    Screen("    Could not allocate memory...");
    DbiCloseCursor(&hCur);
    CloseDbAndExit(&hDb);
    Screen("\r\n*** End of Example ***");
    return;
}

// Initialize the filter expression.
memmove(pcanExpr, &canExp, uSizeCanExpr);
memmove(&pcanExpr[uSizeCanExpr], Nodes, uSizeNodes);

memmove(&pcanExpr[uSizeCanExpr + uSizeNodes],
        szField, strlen(szField) + 1); // First litteral
memmove(&pcanExpr[uSizeCanExpr + uSizeNodes +
                 strlen(szField) + 1],
        szConst, strlen(szConst) + 1); // Second litteral

rslt = DbisetToBegin(hCur);
ChkRslt(rslt, "SetToBegin");

Screen("\r\n    Add a filter to the %s table which will"
        " limit the records\r\n          which are displayed"
        " to those whose %s field starts with '%s'...",
        szTblName, szField, szConst);

```

```

rslt = DbiAddFilter(hCur, 0L, 1, FALSE, (pCANExpr)pcanExpr,
                  NULL, &hFilter);
if (ChkRslt(rslt, "AddFilter") != DBIERR_NONE)
{
    rslt = DbiCloseCursor(&hCur);
    ChkRslt(rslt, "CloseCursor");
    free(pcanExpr);
    CloseDbAndExit(&hDb);
    Screen("\r\n*** End of Example ***");
    return;
}

// Activate the filter.
Screen("    Activate the filter on the %s table...",
      szTblName);
rslt = DbiActivateFilter(hCur, hFilter);
ChkRslt(rslt, "ActivateFilter");

rslt = DbiSetToBegin(hCur);
ChkRslt(rslt, "SetToBegin");

Screen("\r\n    Display the %s table with the filter"
      " set...", szTblName);
DisplayTable(hCur, uNumRecs);

Screen("\r\n    Deactivate the filter...");
rslt = DbiDeactivateFilter(hCur, hFilter);
ChkRslt(rslt, "DeactivateFilter");

Screen("\r\n    Drop the filter...");
rslt = DbiDropFilter(hCur, hFilter);
ChkRslt(rslt, "DropFilter");

rslt = DbiCloseCursor(&hCur);
ChkRslt(rslt, "CloseCursor");

free(pcanExpr);

Screen("    Close the database and exit IDAPI...");
CloseDbAndExit(&hDb);

Screen("\r\n*** End of Example ***");

```

```

}

```

Reference :

7/15/98 3:24:21 PM

Last Modified: 01-SEP-99

## Article #15188: How to Print in Windows

Technical Information Database

FAQ: TI188D - How to Print in Windows

Category: Printing

Platform: All-32Bit

Product: Delphi4.x,

Description:

Printing in Windows is achieved with a device control associated with the printing device described in the WIN.INI. Before you start printing in your program, you will need to create a device control associated with your printer.

To do this, you will call the function GetProfileString as follows.

```
GetMem(Info, 80);
GetProfileString('windows','device',',,,', Info, 80);
```

Where Info is defined to be of type PChar. The resulting ASCIIIZ string will be placed in Info. Unfortunately, CreateDC requires three parameters, one for the driver, the printer type, and the port. If GetProfileString is successful, Info contains all the information required by CreateDC but contained in one string separated by semi-colons. These items must be parsed out of the string Info before CreateDC can be called. The following function is helpful in this task.

```
function GetItem(var S:PChar): PChar;
var
  P: PChar;
  I: Integer;
begin
  i:=0;
  while (S[I]<>',' ) and (S[I]<>#0) do
    inc(I);
  S[I]:=#0;
  GetItem:=S;
```

```
    if S[0]<>#0 then S:=    ;  
end;
```

The function `GetItem` does not copy the strings but rather turns the larger string into a structure containing smaller component strings. Now separating the driver, printer type, and port information is fairly straight forward. Assuming that the `PChar` variables `Driver`, `PrinterType`, `Port` and `TInfo` have been predeclared, you would use the following assignments.

```
TInfo:=Info;  
PrinterType:=GetItem(Info);  
Driver:=GetItem(Info);  
Port:=GetItem(Info);
```

The variable `TInfo` is necessary for a subsequent `FreeMem` call to free up the memory originally allocated for `Info`.

At last, you may now call `CreateDC` with the appropriate parameters (assuming `DC` was previously declared to be of type `HDC`).

```
DC:=CreateDC(Driver, PrinterType, Port, Nil);
```

If for some reason `CreateDC` is not successful, it will return a value of zero.

Now you are ready to start using the `Escape` function. `Escape` is a very special function as it allows access to device capabilities that are not supported by the Windows GDI. The second parameter to `Escape` represents the particular escape function to be used. Two fundamental escape functions always used when printing in Windows are `STARTDOC` and `ENDDOC`. These functions do any setup and shutdown required by the printer. Another important escape function is `NEWFRAME`. `NEWFRAME` forces a buffer flush and causes either a form feed operation or a page eject operation depending upon the type of printer used.

To print some text, the following code illustrates the process involved.

```
procedure Print;  
var  
    TInfo, Info: PChar;  
    Driver, PrinterType, Port: PChar;
```

DC: HDC;

begin

GetMem(Info, 80);

GetProfileString('windows', 'device', ',,', Info, 80);

TInfo:=Info;

Driver:=GetItem(Info);

PrinterType:=GetItem(Info);

Port:=GetItem(Info);

DC:=CreateDC(PrinterType, Driver, Port, Nil);

FreeMem(TInfo, 80);

if Escape(DC, \_STARTDOC, 8,  
StrNew('Doc Name'), nil)<0then

begin

MessageBox(Window, 'Printing can not be started', nil,  
mb\_Ok or mb\_IconStop);

exit;

end;

if Escape(DC, NEWFRAME, 0, nil, nil)<0 then

begin

MessageBox(Window, 'Paper can not be advanced', nil,  
mb\_Ok or mb\_IconStop);

exit;

end;

TextOut(DC, 10, 10, 'This text is being printed', 26);

if Escape(DC, NEWFRAME, 0, nil, nil)<0 then

begin

MessageBox(Window, 'Paper can not be advanced', nil,  
mb\_Ok or mb\_IconStop);

exit;

end;

if Escape(DC, ENDDOC, 0, nil, nil)<0 then

begin

MessageBox(Window, 'Error Printing', nil,  
mb\_Ok or mb\_IconStop);

exit;

end;

end;

Notice that if `Escape` fails that it returns a value less than zero. The actual errors are governed by the spooler constants `sp_Error`, `sp_OutOfDisk`, `sp_OutOfMemory`, and `sp_UserAbort`. It is recommended that a `NEWFRAME` call precede and follow the intended output to ensure that the printer is working with a fresh piece of paper and flushes its buffer when printing ends.

The coordinates used in `TextOut` are arbitrary. Exactly lines can be calculated from the values returned by `GetTextMetrics` as you would for normal output to the screen. See Chapter 4 of the Windows Reference Guide of TPW for a complete description of the `TTextMetric` type. The `tmHeight`, `tmAveCharWidth`, and `tmMaxCharWidth` fields will prove to be the most useful when determining line lengths and the number of characters per page. Use `GetDeviceCaps` to determine the maximum horizontal and vertical resolutions of your printer when calculating these values.

Since device controls are used, printing graphics is a simple extension of what is done to display graphics on screen. Here's a procedure that copies a bitmap to the printer using the techniques that have been discussed so far.

```
procedure HardCopy(HWindow: HWnd; BitMap: HBitmap);
var
  DC, ScreenDC, MemDC: HDC;
  BM: TBitmap;
  OldBitmap: HBitmap;
  Info, TInfo: PChar;
  Driver, PrinterType, Port: PChar;
begin
  GetMem(Info, 80);
  GetProfileString('windows', 'device', ',,', Info, 80);
  TInfo:=Info;
  Driver:=GetInfo(Info);
  PrinterType:=GetItem(Info);
  Port:=GetItem(Info);

  DC:=CreateDC(PrinterType, Driver, Port, Nil);
  Escape(DC, STARTDOC, 8, StrNew('HardCopy'), nil);
  Escape(DC, NEWFRAME, 0, nil, nil);

  ScreenDC:=GetDC(HWindow);
  MemDC:=CreateCompatibleDC(DC);
  ReleaseDC(HWindow, ScreenDC);
```



```
OldBitMap:=SelectObject(MemDC, BitMap);
GetObject(BitMap, sizeof(BM), @ BM);
BitBlt(DC, 0,0, BM.bmWidth, BM.bmHeight,
      MemDC, 0, 0, SRCCOPY);
```

```
Escape(DC, NEWFRAME, 0, nil, nil);
Escape(DC, ENDDOC, 0, nil, nil);
```

```
SelectObject(MemDC, OldBitMap);
DeleteDC(MemDC);
DeleteDC(DC);
```

```
end;
```

For clarity, this routine does not perform error checking in the manner of the previous example so use it with appropriate caution.

As with BitBlt, you may use any of the other GDI functions with a DC associated with the printer. The only difficulty that you may run into in this regard is the capabilities of the printer itself. To determine if a printer can handle bitmaps, use the GetDeviceCaps function as follows.

```
Value:=GetDeviceCaps(DC, RASTERCAPS);
Value:=Value and rc_BitBlt;
```

Assuming, DC to be associate with a printer device and Value to be declared as an integer, if the resulting contents of Value is not zero then the device has the ability to display bitmaps. See Device Capabilities under Chapter 1 of TPW's Windows Reference Guide for more on what GetDeviceCaps can do.

Frequently, programs give user's the ability to change the setting of the printer. However, printers often have greatly different capabilities and it is impossible to determine what printers will be capable of in the future. Therefore, no one dialog is appropriate to change printer setting. Because of these facts, Windows requires the manufacturers of printers to supply drivers and incorporate into these drivers a dialog used to alter printer settings.

There are two functions supported by the Windows API to call up this dialog DeviceMode and ExtDeviceMode. These functions do not exist in the API but rather are loaded from the driver directly.

Before you can call on these functions you must obtain a driver handle. Here's how this would be done knowing what we know so far.

```
GetMem(FullDriver,13);
GetProfileString('windows', 'device', ',,', Info, 80);
TInfo:=Info;
PrinterType:=GetItem(Info);
Driver:=GetItem(Info);
Port:=GetItem(Info);
StrLCat(StrCopy(FullDriver,Driver),'.DRV',12);
DriverHandle:=LoadLibrary(FullDriver);
```

PrinterType, Driver, Port, Info, and TInfo you have seen above. FullDriver is declared to be PChar and DriverHandle is a THandle. FullDriver is necessary to provide the file extension to Driver as the function LoadLibrary requires a complete file name. All printer drivers have the file extension .DRV.

ExtDeviceMode is a more advanced form of DeviceMode. It not only provides the dialog to allow user's change settings but also gives the means for the program to monitor and change the device settings. DeviceMode only provides the user dialog capability. However, ExtDeviceMode was introduced in version 3.0 of Windows and not all printer drivers provide a ExtDeviceMode function. But, all printer driver provide a DeviceMode function. So if ExtDeviceMode is not present, DeviceMode will be. The following code is appropriate for displaying the printer's dialog.

```
P:=GetProcAddress(DriverHandle, 'ExtDeviceMode');
if P<>nil then
begin
  ExtDeviceMode:=TExtDeviceMode(P);
  Size:=ExtDeviceMode(Window, DriverHandle,
    OutModeRec^, PrinterType, Port,
    InModeRec^, nil, 0);
  GetMem(OutModeRec, Size);
  GetMem(InModeRec, Size);
  ExtDeviceMode(Window, DriverHandle,
    OutModeRec^, PrinterType, Port,
    InModeRec^, nil, dm_Prompt);
  FreeMem(OutModeRec, Size);
  FreeMem(InModeRec, Size);
end
else begin
```

```
P:=GetProcAddress(DriverHandle, 'DeviceMode');
DeviceMode:=TDeviceMode(P);
DeviceMode(Window, DriverHandle, PrinterType, Port);
end;
FreeLibrary(DriverHandle);
```

P is declared to be of type TFarProc; ExtDeviceMode is of type TExtDeviceMode which is supplied by the WinTypes unit (the same applies for DeviceMode and TDeviceMode). Both OutModeRec and InModeRec are declared to be of type PDevMode. These two pointers control the input and output to ExtDeviceMode where required.

Notice that if the results of GetProcAddress(DriverHandle, 'ExtDeviceHandle'); are nil then ExtDeviceHandle cannot be provided by the printer. In this case, the code proceeds to the simple DeviceMode call.

Before ExtDeviceMode can be called to prompt the user, it is a good idea to call ExtDeviceMode with its mode parameter (the 8th parameter) set to zero. The result will be the amount of space required for its input and output, TDevMode parameters. This is important as these records may not be the same size as the TDevMode type. After memory is allocated for these records, the data fields of the input record may then be initialized (see Chapter 4 of TPW's Windows Reference guide for more on TDevMode).

Now ExtDeviceMode is called with the mode parameter set to dm\_Prompt which displays and activates the printer's dialog. This constant can be combined with the other constants dm\_Copy, dm\_Modify, and dm\_Update with Turbo's bitwise OR operator. (See Chapter 1 of TPW's Windows Reference Guide for information on these dm\_ Device mode selections).

Another important feature for printing is the Abort Dialog. An Abort Dialog is a dialog displayed when printing is taking place and gives the user of the program an opportunity to halt printing before it is finished.

The first thing to do, is create a dialog suitable for the task. Something like this .DLG file would serve (in a true DLG file all data for statement would appear on a single line). Though you may prefer to use a resource toolkit of some sort to build your dialog into a resource used by your program.

ABORTBOX DIALOG DISCARDABLE LOADONCALL PURE MOVEABLE 10, 37, 187,

67

```
STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION | 0x80L
CAPTION "Printing in Progress"
BEGIN
    CONTROL "Escape" 100, "BUTTON", WS_CHILD | WS_VISIBLE |
        WS_TABSTOP, 73, 37, 41, 16
    CONTROL "Press Escape to Stop Printing" 102, "STATIC", WS_CHILD
        | WS_VISIBLE, 44, 12, 99, 11
END
```

Next, using the Object Windows Library, you would define an object type to govern your Abort Dialog. Its actual definition needs to be no more than something like the following.

```
var
    Abort: Boolean;
    AbortWindow: HWindow;
type
    PAbortDialog = ^TAbortDialog;
    TAbortDialog = object(TDlgWindow)
        procedure SetUpWindow; virtual;
        procedure WMCommand(var Msg: TMessage);
            virtual wm_First + wm_Command;
    end;

procedure TAbortDialog.SetUpWindow;
begin
    Abort:=false;
    SetFocus(HWindow);
    AbortWindow:=HWindow;
end;

procedure TAbortDialog.WMCommand(var Msg: TMessage);
{Will be entered whenever a button, return key or escape key is
pressed.}
begin
    Abort:=true;
end;
```

As any button press, return key press, or escape key press will abort printing only a WMCommand method is needed. InitDialog is necessary to set the focus to this dialog and to set the field Abort.

Abort is used by the dialog to determine if printing has been

aborted. To prevent interrupting printing while the dialog is displayed, a Call Back function is required to be installed. Most such functions call PeekMessage which checks the application queue for a message. If there are no messages, it returns and gives control to Windows. Here's a simple example of such a function.

```
function AbortCallback(DC: HDC; Code: Integer): Bool; export;
var
  Msg: TMsg;
begin
  While (not Abort) and PeekMessage(Msg, 0, 0, 0, pm_Remove) do
    if not IsDialogMessage(AbortWindow, Msg) then
      begin
        TranslateMessage(Msg);
        DispatchMessage(Msg);
      end;
    if Abort then AbortCallback:=false else AbortCallback:=true;
  end;
```

PeekMessage yields control to Windows when a message is not available. If a message is received and it doesn't belong to the dialog, it is sent along using the customary TranslateMessage and DispatchMessage functions. After it checks to see if the user has aborted and if so, AbortCallback returns false. Otherwise, it simply returns true which indicates printing may continue.

Now you have all that you need to install you Abort Dialog. Before, your call to Escape for the STARTDOC, the following code should be executed (assuming that you have previously declared a variable AbortDialog of type PAbortDialog and a variable called AbortCallbackProc of type TFarProc to hold the address of the callback function).

```
AbortDialog:=New(PAbortDialog,
  Init(Application^.MainWindow, 'ABORTBOX'));
AbortDialog:=PAbortDialog(Application^.
  MakeWindow(AbortDialog));
AbortCallbackProc:=MakeProcInstance(@ AbortCallback,
  HInstance);
Escape(DC, SETABORTPROC, 0, AbortCallbackProc, nil);
```

Now you would call Escape with the appropriate STARTDOC, NEWFRAME, and ENDDOC parameters as illustrated before. However, it is vital to check for errors after a NEWFRAME escape call.

For printing actually occurs during the NEWFRAME. If an error or user abort occurs Escape will return a value less than zero. If this happens do not allow the ENDDOC escape call take place; for the GDI automatically terminates the operation before returning the error value. Also, if an error other than a user abort occurs, use AbortDialog^.CloseWindow to remove the dialog from view. If the user cancels the print operation, Windows removes the dialog automatically.

Reference:

7/29/99 2:39:59 PM

Last Modified: 01-SEP-99

## Article #15190: SYSTEM LOCK RUNNING WITHIN IDE VS. DISK RUN

Technical Notes Database

TN190D.txt      SYSTEM LOCK RUNNING WITHIN IDE VS. DISK RUN  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. Why does my program run fine at DOS but locks the machine when running from IDE?

A. In most cases, this system crash is caused by a memory overwrite. The most common causes for memory overwrites are:

1. Indexes out of range      (Turn range checking on {\$R+})
2. Uninitialized variables   (Write an initialization proc)
3. Pointers out of bounds    (Verify that pointers are not pointing outside of the heap space)
4. Improper use of FillChar (Be sure to use the SizeOf or Move function)
5. Illogical operations on strings

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15194: Using a continue node in a filter.

Technical Information Database

TI194B.txt Using a continue node in a filter.

Category :General Programming

Platform :All

Product :BDE All

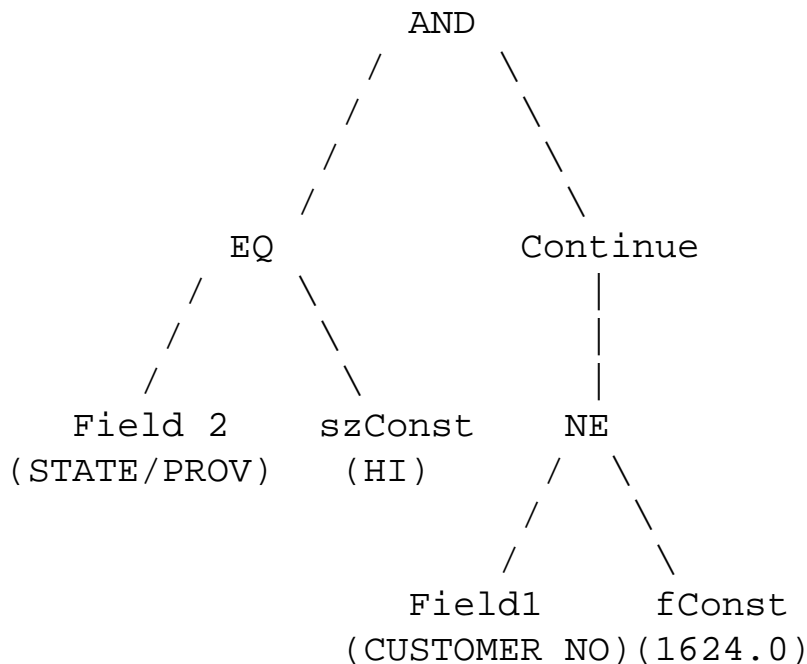
Description:

/\*

This example shows the use of a canContinue node. A Continue node is used to stop evaluating when a certain condition is false for the first time.

This filter will limit the result set to those customers living in Hawaii, you are listed in the table before the customer with ID 1624.

Picture of the filter expression:



\*/



```
// BDE - (C) Copyright 1994 by Borland International
```

```
// Filter.c
```

```
#include "snipit.h"
```

```
static const char szTblName[] = "customer";
```

```
static const char szTblType[] = szPARADOX;
```

```
//=====
```

```
// Function:
```

```
//     Filter();
```

```
//
```

```
// Description:
```

```
//     This example shows how to use filters to limit the
```

```
//     result set of a table. Filters perform a function
```

```
//     similar to that of ranges, but more powerful
```

```
//     operations are supported.
```

```
//=====
```

```
void
```

```
Filter (void)
```

```
{
```

```
    hDBIDb          hDb;          // Handle to the database.
```

```
    hDBICur         hCur;        // Handle to the table.
```

```
    pBYTE          pcanExpr;      // Structure containing  
                                // filter info.
```

```
    hDBIFilter     hFilter;       // Filter handle.
```

```
    UINT16         uSizeNodes;    // Size of the nodes in the  
                                // tree.
```

```
    UINT16         uSizeCanExpr;  // Size of the header  
                                // information.
```

```
    UINT16         uSizeLiterals; // Size of the literals.
```

```
    UINT16         uTotalSize;    // Total size of the filter  
                                // expression.
```

```
    UINT32         uNumRecs = 10; // Number of records to  
                                // display.
```

```
    CANExpr        canExp;        // Contains the header  
                                // information.
```

```
    UINT16         Nodes[] =     // Nodes of the filter tree.
```

```
{
```

```
    // Offset 0. Node 1.
```

```
    nodeBINARY,    // canBinary.nodeClass
```

```
    canAND,        // canBinary.canOp
```

```
    8,             // canBinary.iOperand1 - node 2
```

```
    34,           // canBinary.iOperand2 - node 5
```

```

// Offsets in the Nodes array

// Offset 8. Node 2.
nodeBINARY, // canBinary.nodeClass
canEQ , // canBinary.canOp
16, // canBinary.iOperand1 - node 3
24, // canBinary.iOperand2 - node 4
// Offsets in the Nodes array

// Offset 16. Node 3.
nodeFIELD, // canField.nodeClass
canFIELD, // canField.canOp
5, // canField.iFieldNum
11, // canField.iNameOffset: szField2
// is the literal at offset
// strlen(szField1) + 1

// Offset 24. Node 4.
nodeCONST, // canConst.nodeClass
canCONST, // canConst.canOp
fldZSTRING, // canConst.iType
3, // canConst.iSize
31, // canConst.iOffset: fconst is
// the literal at offset
// strlen(szField1) + 1 +
// sizeof(fConst) +
// strlen(szField2) + 1

// Offset 34. Node 5.
nodeUNARY, // canBinary.nodeClass
canCONTINUE, // canBinary.canOp
40, // canBinary.iOperand1 - node 6
// Offsets in the Nodes array

// Offset 40. Node 6.
nodeBINARY, // canBinary.nodeClass
canNE , // canBinary.canOp
48, // canBinary.iOperand1 - node 7
56, // canBinary.iOperand2 - node 8
// Offsets in the Nodes array

// Offset 48. Node 7.
nodeFIELD, // canField.nodeClass
canFIELD, // canField.canOp
1, // canField.iFieldNum

```

```

        0,          // canField.iNameOffset:
                    //   szField1 is the literal at
                    //   offset 0.

        // Offset 56. Node 1.
nodeCONST,        // canConst.nodeClass
canCONST,         // canConst.canOp
fldFLOAT,        // canConst.iType
8,               // canConst.iSize
23,              // canConst.iOffset: fconst is
                    //   the literal at offset
                    //   strlen(szField1) + 1 +
                    //   strlen(szField2) + 1
};

// Name of the field for the third node of the tree.
CHAR          szField1[] = "CUSTOMER NO";
// Name of the field for the third node of the tree.
CHAR          szField2[] = "STATE/PROV";
// Value of the constant for the second node of the tree.
FLOAT        fConst      = 1624.0;
// Value of the constant for the second node of the tree.
// Field #7
CHAR          szConst[]  = "HI";

DBIResult     rslt; // Return value from IDAPI functions.

Screen("*** Filter Example ***\r\n");

BREAK_IN_DEBUGGER();

Screen("    Initializing IDAPI...");
if (InitAndConnect(&hDb) != DBIERR_NONE)
{
    Screen("\r\n*** End of Example ***");
    return;
}

Screen("    Setting the database directory...");
rslt = DbiSetDirectory(hDb, (pCHAR) szTblDirectory);
ChkRslt(rslt, "SetDirectory");

Screen("    Open the %s table...", szTblName);
rslt = DbiOpenTable(hDb, (pCHAR) szTblName,
                    (pCHAR) szTblType,

```

```

        NULL, NULL, 0, dbiREADWRITE,
        dbiOPENSHARED, xltFIELD, FALSE, NULL,
        &hCur);
if (ChkRslt(rslt, "OpenTable") != DBIERR_NONE)
{
    CloseDbAndExit(&hDb);
    Screen("\r\n*** End of Example ***");
    return;
}

// Go to the beginning of the table
rslt = DbiSetToBegin(hCur);
ChkRslt(rslt, "SetToBegin");

Screen("\r\n    Display the %s table...", szTblName);
DisplayTable(hCur, uNumRecs);

// Size of the nodes.
uSizeNodes      = sizeof(Nodes);
// Size of the literals.
uSizeLiterals   = strlen(szField1) + 1 + sizeof(fConst) +
                  strlen(szField2) + 1 + strlen(szConst)
                  + 1;
// Size of the header information.
uSizeCanExpr    = sizeof(CANExpr);
// Total size of the filter.
uTotalSize      = uSizeCanExpr + uSizeNodes + uSizeLiterals;
// Initialize the header information
canExp.iVer     = 1; // Version.
canExp.iTotalSize = uTotalSize; // Total size of the filter.
canExp.iNodes   = 8; // Number of nodes.
canExp.iNodeStart = uSizeCanExpr; // The offset in the
                                   // buffer where the
                                   // expression nodes
                                   // start.
// The offset in the buffer where the literals start.
canExp.iLiteralStart = uSizeCanExpr + uSizeNodes;

// Allocate space for the filter expression.
pcanExpr = (pBYTE)malloc(uTotalSize * sizeof(BYTE));
if (pcanExpr == NULL)
{
    Screen("    Could not allocate memory...");
    DbiCloseCursor(&hCur);
    CloseDbAndExit(&hDb);
}

```

```

    Screen("\r\n*** End of Example ***");
    return;
}

// Initialize the filter expression.
memmove(pcanExpr, &canExp, uSizeCanExpr);
memmove(&pcanExpr[uSizeCanExpr], Nodes, uSizeNodes);

memmove(&pcanExpr[uSizeCanExpr + uSizeNodes],
        szField1, strlen(szField1) + 1); // First litteral

memmove(&pcanExpr[(uSizeCanExpr + uSizeNodes +
                  strlen(szField1) + 1)],
        szField2, strlen(szField2) + 1); // First litteral

memmove(&pcanExpr[(uSizeCanExpr + uSizeNodes +
                  strlen(szField1) + 1 +
                  strlen(szField2) + 1)],
        &fConst, sizeof(fConst)); // Second litteral

memmove(&pcanExpr[(uSizeCanExpr + uSizeNodes +
                  strlen(szField1) + 1 +
                  strlen(szField2) + 1 + sizeof(fConst))],
        szConst, strlen(szConst) + 1); // First litteral

rslt = DbisetToBegin(hCur);
ChkRslt(rslt, "SetToBegin");

Screen("\r\n    Add a bug filter to the %s table which will"
       " limit the records\r\n           which are displayed"
       " to those whose %s field is greater than %.1lf...",
       szTblName, szField1, fConst);
rslt = DbiaAddFilter(hCur, 0L, 1, FALSE, (pCANExpr)pcanExpr,
                   NULL, &hFilter);
if (ChkRslt(rslt, "AddFilter") != DBIERR_NONE)
{
    rslt = DbicloseCursor(&hCur);
    ChkRslt(rslt, "CloseCursor");
    free(pcanExpr);
    CloseDbAndExit(&hDb);
    Screen("\r\n*** End of Example ***");
    return;
}

```

```

// Activate the filter.
Screen("    Activate the filter on the %s table...",
      szTblName);
rslt = DbiActivateFilter(hCur, hFilter);
ChkRslt(rslt, "ActivateFilter");

rslt = DbiSetToBegin(hCur);
ChkRslt(rslt, "SetToBegin");

Screen("\r\n    Display the %s table with the filter"
      " set...", szTblName);
DisplayTable(hCur, uNumRecs);

Screen("\r\n    Deactivate the filter...");
rslt = DbiDeactivateFilter(hCur, hFilter);
ChkRslt(rslt, "DeactivateFilter");

Screen("\r\n    Drop the filter...");
rslt = DbiDropFilter(hCur, hFilter);
ChkRslt(rslt, "DropFilter");

rslt = DbiCloseCursor(&hCur);
ChkRslt(rslt, "CloseCursor");

free(pcanExpr);

Screen("    Close the database and exit IDAPI...");
CloseDbAndExit(&hDb);

Screen("\r\n*** End of Example ***");
}

```

Reference:

7/15/98 3:24:22 PM

Last Modified: 01-SEP-99

## Article #15196: Dual Input Lines in Turbo Vision,GetData SetData

Technical Information Database

TI196D.txt Dual Input Lines in Turbo Vision,GetData SetData  
Category :TVISION  
Platform :All  
Product :Pascal All

Description:

{

This program will create a record structure of two strings and initialize them with data then read them into a TInputLine. The program will use the SetData and GetData methods to load and store data from the TInputLine Object.

}

{ \$X+ }

program DualInputExample;

uses Objects, Drivers, Views, Menus, Dialogs, App;

const

    cmNewDialog      = 100;  
    hcMyDialog       = 300;

type

    MyData = record  
        MyStr1:String[10];                  { Create a Record Structure }  
        MyStr2:String[10];                  { for Input Lines in Dialog }  
    end;

var

    RMyData: MyData;                       { Declare the Structure }

type

    TMyApp = object(TApplication)  
        constructor Init;  
        procedure HandleEvent(var Event: TEvent); virtual;  
        procedure InitMenuBar; virtual;

```

    procedure NewDialog;
end;

PDemoDialog = ^TDemoDialog;
TDemoDialog = object(TDialog)
    Procedure HandleEvent(var Event:TEvent);virtual;
end;

constructor TMyApp.Init;
var
    R : TRect;
begin
    TApplication.Init;           { Initialize data }
    RMydata.MYstr1:='What';
    RMydata.MYstr2:='Cheese';
    GetExtent(R);
    Dec(R.B.X);
    R.A.X := R.B.X - 9; R.A.Y := R.B.Y - 1;
end;
{ TMyApp }

Procedure TDemoDialog.HandleEvent(var Event:TEvent);
begin
    TDialog.HandleEvent(Event);
    if Event.What = EvCommand then
        begin
            if event.what = EvCommand then
                case Event.Command of
                    cmOK:begin
                        GetData(RMyData); { This will retrieve the }
                                           { information corresponding to }
                                           { a previous SetData and load }
                                           { the changed values in the }
                                           { record structure }

                        TDialog.Done;
                    end;
                    cmCancel:TDialog.Done;
                end
            else
                Exit;
            end;
        clearEvent(Event);
end;

procedure TMyApp.HandleEvent(var Event: TEvent);

```



```

begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewDialog: NewDialog;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
end;

procedure TMyApp.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
        nil)),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('~D~ialog', 'F2', kbF2, cmNewDialog, hcmyDialog,
        nil)),
    nil))
  )));
end;

procedure TMyApp.NewDialog;
var
  Borland: PView;
  Dialog: PDemoDialog;
  R: TRect;
  C: Word;
begin
  R.Assign(20, 6, 60, 19);
  Dialog := New(PDemoDialog, Init(R, 'Demo Dialog'));
  with Dialog^ do
  begin
    R.Assign(3, 3, 18, 4);
    Borland := New(PInputLine, Init(R, 10));
    Insert(Borland);    { The order of insertion is directly }
                       { related to the order of data items in }
  end;
end;

```

```

                                { the data structure }
R.Assign(3, 4, 18, 5);
Borland := New(PInputLine, Init(R,10));
Insert(Borland);
R.Assign(2, 2, 10, 3);
Insert(New(PLabel, Init(R, 'Cheeses', Borland)));
R.Assign(22, 3, 34, 5);
Borland := New(PRadioButtons, Init(R,
    NewSItem('~R~unny',
    NewSItem('~M~eltd',
    nil)))
);
Insert(Borland);
R.Assign(21, 2, 33, 3);
Insert(New(PLabel, Init(R, 'Consistency', Borland)));
R.Assign(15, 8, 25, 10);
Insert(New(PButton, Init(R, '~O~k', cmOk, bfDefault)));
R.Assign(28, 8, 38, 10);
Insert(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));
end;
Dialog^.SetData(RMyData);  { Associate data to dialog }
                           { input lines }

DeskTop^.Insert(Dialog);
end;
var
    MyApp: TMyApp;

begin

    MyApp.Init;
    MyApp.Run;
    MyApp.Done;
end.

```

Reference:

7/16/98 4:33:39 PM

Last Modified: 01-SEP-99

## Article #15198: PICK FILES - TURBO READS FROM ANOTHER DRIVE

Technical Notes Database

TN198D.txt      PICK FILES - TURBO READS FROM ANOTHER DRIVE  
Category      :Pascal  
Platform        :All  
Product        :Turbo Pascal 4.0+

### Description:

- Q. Why does Turbo Pascal sometimes try to read from another drive when I run it?
- A. When you leave Turbo Pascal, it saves the name and path of the file you were last editing in a pick list. The next time you load Turbo, it checks this pick list and tries to load the file you were last editing. If the file you were last editing was in another drive, Turbo will try to read from that drive. This also occurs if you have installed another drive as your Turbo Directory.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15203: BDE Frequently Asked Questions.

Technical Information Database

TI203B.txt BDE Frequently Asked Questions.

Category :General Programming

Platform :All

Product :BDE All

### Description:

This document contains information most often provided to users of the BDE.

### FREQUENTLY ASKED QUESTIONS AND ANSWERS

-----

- 1) General
- 2) Paradox Table Specifics
- 3) dBASE Table Specifics

### GENERAL

-----

For additional information other than what is provided in this document, see:

<http://www.borland.com/devsupport/bde/faq/>

Q: Is there any "Getting started" information on the BDE?

A: Yes - get the TI177B for information on getting started with the BDE.

Q: What basic steps can I follow to make my BDE application run smoothly?

- A:
- 1) Increase stack size to 20K. (16 bit only)
  - 2) Increase the number of file handles available to your application using the Windows API function SetHandleCount. (16 bit only)
  - 3) Check the return values of each and every BDE function

call and provide some means for handling the result of any return value other than DBIERR\_NONE. (See next question also).

- 4) Take a look at TI177B, which describes the basic steps that are required in setting up a BDE application.

Q: I'm having trouble with my program, and the debugger has traced the problem into a BDE DLL. Does this mean that there is a bug in the BDE?

A: Not necessarily. Frequently if a prior call to the BDE has failed due to being passed an invalid handle or for some other reason, this will leave the BDE in an unpredictable state which will later cause a GP fault. The solution is to check the return values of each and every function call you make. The BDE sample applications contain error handling routines.

Q: Why is my application having problems when share is loaded?

A: One possibility is that share could be running out of locks or file handles. The command line 'share /L:200 /F:4096' will increase the number of files that can be locked to 200 and the memory available for files to 4096 (the default is 20 and 2048). See your DOS or WINDOWS manual for more information on share.

This information only applies to Windows 3.1. Windows for Workgroups, Windows 95 and Windows NT load share automatically. Windows 3.1 users can get this same functionality and not have to worry about running out of locks or file handles by obtaining the Windows 3.1 version of VSHARE.DLL from Microsoft.

Q: Why am I getting the error "Cannot find NetWare.DLL"?

A: An outdated netware.driv is found in the System.ini file. A newer version of this file should be gotten from your network administrator, and/or Novell.

Q: How do I use the BDE in a DLL that is called from Paradox

for Windows or dBASE for Windows?

A: Make certain to use dynamic linking when using the BDE in a DLL called from another application. This will be done automatically when using the IDAPI.LIB that ships with the BDE, but will not be done if an import library was created using IMPLIB (the shipping IDAPI.LIB is more than a standard import library).

Q: Why am I having problems getting information from the DbiGetFieldDescs function?

A: Make certain to have 'Allocate Enums as Ints' set (Options | Compiler | Code Generation in BC 4.x, Options | Project | Compiler | Code Generation in BC 5.). The FLDDesc structure, which is used by DbiGetFieldDesc, contains an enum, which within the DLL is set to the same size as an int (two bytes).

Q: Why am I having stack corruption problems? or Why is my application crashing when calling or returning from a function?

A: First, make certain that you have 25k of stack allocated for your application. Then, make certain that you have 'Allocate enums as ints' selected. A number of structures, including CURProps, make use of enumerations. As these enumerations are allocated two bytes within the DLL, we need to make certain that the application is passing two bytes as well. This is done with the 'Allocate enums as ints' option.

Q: How can I optimize BDE performance on table operations?

A: Although there are a number of ways to improve BDE performance, some general things to try are as follows:

- 1) Keep the number of maintained secondary indexes to a minimum; sometimes it is better to delete the index and recreate it than to perform a number of table operations with the indexes in place.
- 2) If possible, increase the size of the swap buffer and the number of file handles that the BDE has available to it. This will decrease the Engine's need to swap resources. Note: make certain to increase the file handles available to your

application using SetHandleCount, as well as increasing the number of file handles available to the BDE in IDAPI.CFG.

- 3) Open the table exclusively.
- 4) Batch as many operations as possible - do not read/write records one at a time. Use DbBatchMove, DbCopyTable, DbReadBlock, and/or DbWriteBlock.
- 5) When using DbWriteBlock, try to work in multiples of the physical block size, usually 2k or 4k.
- 6) If you are opening and closing one or more tables repeatedly, consider calling DbAcqPersistTableLock on a non-existent file after you initialize the BDE. This will create the .LCK file so that it will not have to be created each time a table is opened, created, etc. (Note: you'll also want to call DbRelPersistTableLock before calling DbExit).
- 7) Used Cached Updates  
Paradox Tables Only.
- 8) Work with in-memory tables when possible (Note that in-memory tables cannot be the source table to DbBatchMove).

Q: What are the current versions of the BDE?

A: <http://www.borland.com/devsupport/bde/bdever.html>

Q: Why am I getting the error 'Invalid BLOB Handle in record Buffer' when I attempt to modify a record containing a BLOB?

A: This error can occur for two different reasons.

#### Case 1 - BDE API Application

This error is caused by not setting up the record correctly. Make certain to call DbInitRecord on the record buffer before reading the record from the table.

#### CASE 2 - Any BDE Application

This error occurs when scrolling through more than 64 blobs in the results of a dead query. The solution is to either make the results of the query a live result set, or somehow limit scrolling blobs. (Such as having your SELECT statement select fewer records.)

NOTE:A future release of the BDE is planned to be able to be configured to get past the present hard coded limit.

Q: Why does my application crash when using filters?

A: Beside the general suggestion at the top of this file, as well as errors in setting up offsets of the filter, this error can be caused by setting the iPriority parameter to DbtAddFilter to 0. Make certain to set this value to 1. (There is a misprint in some copies of the BDE Users Guide to set this to 0).

Q: What's new in the 32 bit version?

A: 1) Long filename support.  
2) UNC filename support.  
3) Transactions on Paradox and dBASE tables.  
4) Cached Updates.  
5) The method for preparing queries has changed, requiring the use of a new BDEfunction, DbtQAlloc, to obtain a statement handle.  
6) BDE Configuration Information can now optionally be stored in the registry.  
7) New SQL Engine.

Q: Why am I getting the error "Too many open files" in the 16 bit version?

A: You need to call SetHandleCount and make sure the value is larger than the MAXFILEHANDLES statement in BDECFG.

Q. Can the BDE use descending indexes?

A. Only the Paradox, dBASE and FoxPro drivers support descending indexes. Selecting a descending index for any other table type will make the data appear in ascending order.

Q. With 16 bit BDE, how come I get a GPF in DBMSSOC3.DLL when connecting or disconnecting from my MSSQL database?

A. This problem can occur when using TCP/IP as the transport. The BDE calls W3DBLIB.DLL, which in turn calls DBMSSOC3.DLL. The problem occurs most commonly when using Windows NT but can occur under Windows 3.1. It has not been reported against Windows 95.



There are a number of potential solutions to this problems.

1. If the problem occurs under Windows NT, use 32 bit BDE instead.
2. Try switching providers of the TCP/IP stack. The testing for BDE 16 bit was done against Windows 3.1 using the Microsoft TCP/IP stack.
3. Use a different protocol.
4. Users have reported that replacing DBMSSOC.DLL with WDBWSKTC.DLL from the Sybase client resolves this problem.
5. For Windows 3.1 users, it has been reported you can connect using MicroSoft's 32 bit ODBC drivers, which would then use the 32 bit TCP/IP stack and solve the problem.

#### PARADOX TABLE SPECIFICS

-----  
Q: How do I access tables in a read-only directory (such as a table on a CD-ROM)?

A: A directory lock needs to be placed within the directory containing the table to prevent the BDE from attempting to create a lock file (.LCK) in that directory. An example of this is included in the SNIPIT example application, in the file RDOLOCK.C.

Q: I got "Table locked/busy" and it's neither?

A: This error happens when the application tries to lock a table when the corresponding prevent lock already exists, or when the application tries to set a prevent lock and a conflicting lock already exists. Check for old, unused lock files. Delete any .lck files that may exist after all BDE applications have terminated. Also run the TUtility program to test for table validity.

Q: Why do I get the error "Multiple Paradox Net Files Found?"

A: In addition to the case of two application referencing different physical PARADOX.NET files, this error message can be caused by old .LCK files or when two applications using the same NET file are mapped differently. Also make certain that the BDE DLL's are not left in memory. This error can also occur if differnt mapping are used to

reference the same file. For example, if h:onetwo on one system points to the same physical location as h:three on another system, the BDE will not recognize that it is only a single .NET file. Make certain that the path matches.

Q: I am working with a Peer-to-Peer network, like Windows for Workgroups, Lantastic, or Personal Netware, and I am getting "Multiple Paradox Net Files Found", what is wrong?

A: The BDE requires that everyone use the exact same directory and drive letter. Therefore, if the tables are on c:tables on the shared drive and user maps the shared drive as f: then f:tables points to the same place as the shared drive's c:tables. However, the drive letter must also match. The way around this is to subst drive c: as drive f: This way both c: and f: point to the same drive. Now f:tables on the shared drive's computer points to the c:tables and everyone is using f:tables. For more information on the subst command look in your DOS manual. However, remember that if LASTDRIVE is set to f: in your CONFIG.SYS then the last drive you can subst is f:

Q. I am getting the error "Lock file has grown too large". What is the solution?

A. Download TI256B for a detailed answer.

Q: Why am I getting the error: "Could not initialize IDAPI - not initialized for accessing network drives" when attempting to open a table on a network?

A: 1) Make certain the network control directory (NET DIR) is set to a directory on the network (not local). This is set in the BDE Configuration utility.  
2) Make certain that you have read/write/create rights to the network control directory, as well as the directory containing the table.  
3) Make certain that the private directory is set to a local directory. For example, in the Database Desktop (DBD), the private directory is set in the WIN.INI file, in the [DBD] section, PrivDir. If it is set to a directory on the network, this error can occur.  
4) Old lock files exist in the private directory. Either the application previously crashed, resulting in

improper cleanup, or another application is using the same private directory.

- 5) Under Windows for Workgroups (WfW), search your hard drive for the NWCALLS.DLL. Rename/backup the older versions of this file and run again.

If this fails to resolve the problem, you will need to make certain to not use the NetBEUI protocol as the default network protocol in Windows Setup. Select NetWare (or some other protocol) as the networking protocol, and everything should work successfully.

## dBASE TABLE SPECIFICS

-----

Q: How do I insert a Date into a dBASE table using Local SQL?

A: 'INSERT INTO myTable (myField) value ("01/01/94")'

Q: How can I create a case-insensitive index?

A: Case insensitive indexes are not directly supported, but can be simulated using an expression index with the upper() function.

Q: Why can I insert a duplicate record into a dBASE table when I have a unique index defined?

A: dBASE does not enforce the uniqueness within the table - only within the index. When the table is opened on that index, only one record with a given key value will be present in the index, but both values exist in the table.

Reference:

7/15/98 3:24:22 PM

Last Modified: 11-NOV-99

## Article #15205: Change Menubar at Runtime

Technical Information Database

TI205D.txt Change Menubar at Runtime

Category :TVISION

Platform :All

Product :Pascal All

Description:

{

This program changes the menubar at runtime as well as demonstrates how to create a menuitem that does not have a submenu.

}

{ \$X+ }

program ExampleProgram;

uses

Drivers, Objects, Views, App, Menus, Puzzle, Calendar;

const

PuzzleCmd = 100;

CalendarCmd = 101;

SwitchCmd = 102;

type

TTestMain = object(TApplication)

OtherMenu: PMenuBar;

constructor Init;

procedure Calendar;

procedure HandleEvent(var Event: TEvent); virtual;

procedure InitMenuBar; virtual;

procedure Puzzle;

procedure SwitchMenu;

end;

{ TTestMain }

constructor TTestMain.Init;

begin

```

    TApplication.Init;
end;

procedure TTestMain.Calendar;
var
    CalendarWindow: PCalendarWindow;
begin
    CalendarWindow := new(PCalendarWindow, Init);
    DeskTop^.Insert(CalendarWindow);
end;

procedure TTestMain.HandleEvent(var Event: TEvent);
begin
    TApplication.HandleEvent(Event);
    if Event.What = evCommand Then
    begin
        case Event.Command of
            PuzzleCmd    : Puzzle;
            CalendarCmd  : Calendar;
            SwitchCmd    : SwitchMenu;
        else
            Exit;
        end;
        ClearEvent(Event);
    end;
end;

Procedure TTestMain.InitMenuBar;
var
    R: TRect;
begin
    GetExtent(R);
    R.B.Y := R.A.Y+1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
        NewItem('~S~witch Menus',' ', 0, SwitchCmd, hcNoContext,
        NewSubMenu('~E~xample',hcNoContext,NewMenu(
        NewItem('~P~uzzle',' ', 0, PuzzleCmd, hcNoContext,
        NewItem('~C~alendar',' ', 0, CalendarCmd, hcNoContext,
        NewLine(
        NewItem('~Q~uit', ' ',0, cmQuit,
hcNoContext,Nil))))),Nil)))));
    OtherMenu := New(PMenuBar, Init(R, NewMenu(
        NewItem('~S~witch Menus',' ', 0, SwitchCmd, hcNoContext,
        NewSubMenu('~O~ther Menu',hcNoContext,NewMenu(
        NewItem('~B~y', ' ', 0, cmQuit, hcNoContext,

```

```

        NewLine(
           NewItem('~Q~uit', '',0, cmQuit,
hcNoContext,Nil))))),Nil)))));
end;

procedure TTestMain.Puzzle;
var
    PuzzleWindow: PPuzzleWindow;
begin
    PuzzleWindow := new(PPuzzleWindow, Init);
    DeskTop^.Insert(PuzzleWindow);
end;

procedure TTestMain.SwitchMenu;
var
    Temp: PMenuBar;
begin
    Delete(MenuBar);
    Temp := PMenuBar(MenuBar);
    MenuBar := OtherMenu;
    OtherMenu := Temp;
    Insert(MenuBar);
    MenuBar^.DrawView;
end;

var
    TestMain: TTestMain;

begin
    TestMain.Init;
    TestMain.Run;
    TestMain.Done;
end.

```

Reference:

7/16/98 4:33:39 PM

Last Modified: 01-SEP-99

## Article #15207: CURSOR SIZE CHANGE

Technical Notes Database

TN207D.txt      CURSOR SIZE CHANGE  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. How can I change the cursor size in the user interface (UI)?  
A. Send the appropriate TI to modify the cursor in Turbo.EXE.

### Reference:

7/16/98 4:35:29 PM

Last Modified: 01-SEP-99

## Article #15209: 'BLOB has been modified.', 'Index is out of date.' errors

### Technical Information Document (TI3342)

*"Index out of Date" (\$2F02), "Blob has been modified" (\$3302) errors with Paradox Tables*

- Product: *BDE*
  - Version: *All*
  - Platform: *Windows 95, Windows NT, Windows 3.1*
- 

Index out Date (\$2F02) is an error that occurs while using Paradox tables when the data in a table and a corresponding index is not consistent. In most cases (see below for the one exception), short of malicious behavior such as renaming an index, adding some data to the table, then renaming the index back, there is no programmatic way to cause this error to occur. There is no way to determine which index is out of date. All indexes must be recreated.

Blob has been modified (\$3302) is an error that occurs when the Blob portion of the record contained in the .DB file has become inconsistent with the Blob portion in the .MB file. This could occur when the write to the .DB file was successful but the .MB file did not get updated, or visa-versa.

There are a few mechanisms to fix a table where these errors have occurred.

1. First try re-starting the application. It is possible that the BDE has become unstable and is reporting incorrect errors. Also try opening the table with a different application.
2. Use Paradox 7 or 8 to run the Table Repair utility. Please see original documentation for more information.
3. Run TUtility and rebuild the table. TUtility is an unsupported utility available for download from the Borland web site in the [Utilities, programs and updates section](#).
4. Delete all indexes and recreate them (*Index out Date (\$2F02) error only*). To do this you'll need to know the structure of all your indexes (including primary) before recreating them, which means you need to know the structure of all indexes before the error occurs.

### There are 8 known possible causes for this error.

1. Incorrectly setting the LOCAL SHARE property.

Most commonly this occurs via Peer-to-Peer networking. In this case the two different database engines are on two CPUs, even though they may be the same version. See [BDE setup for Peer-To-Peer\(Non-Dedicated\) Networks](#) for additional information on Peer-to-Peer setup.

Another condition is when two different database engines execute on the same CPU concurrently and access data locally. This would be true when any combination of following are used concurrently: The Paradox Engine, BDE 16 bit, BDE 32 bit, Paradox for DOS. In this case each engine must set LOCAL SHARE to TRUE. Note that if use two applications which both use the same database engine (for example: Delphi 3 and C++ Builder 3) concurrently are run LOCAL SHARE does not need to be set to



TRUE. In this case, all locking and cached data is in a central memory pool which all BDE applications have access to. Also, if two different database engines use data remotely, LOCAL SHARE must be set to TRUE.

Code should be used at startup to check the setting of local share. Look at the [DbiOpenCfgInfoList](#) BDE API function call for more information.

2. Error transmitting data from the workstation to the server.

Most commonly, this occurs with bad network hardware (cable, card, hub, etc.). This has been determined to be a problem even though there were no other errors are detected in data transmission. To determine if this is the cause for your error, try eliminating one CPU at a time from using the data and see if the problem continues.

3. Bad VREDIR.VXD on any client accessing tables *Windows 95 ONLY*:

Several versions (notably 4.00.1113 and 4.00.1114) of the file VREDIR.VXD may need to be updated.

Reports have shown that using the original release of VREDIR.VXD (4.00.950) and a new release (4.00.1116) do not result in the errors "Blob has been modified" and/or "Index is out of date." If any one of the clients has a "bad" version of this device driver, the error can occur on any machine, not just the machine with the bad driver.

This error most likely occurs in 16Bit versions of the Borland Database Engine, although it still can occur in 32Bit versions.

For further information on the update of VREDIR.VXD, Please check the following Microsoft Articles: [Q174371](#) and [Q165403](#)

4. For Windows 95 clients only, when using data on Windows NT: Add the following key in your registry: HKEY\_LOCAL\_MACHINESystemCurrentControlSetServicesVxDVredir

Then create the string or Binary Value (either one works) with a name of DiscardCacheonOpen and make it equal to 1.

Note that this an undocumented registry entry obtained from Microsoft. Questions on its functionality should be directed to Microsoft.

5. Problem with opportunistic locking Windows NT ONLY: Try turning off opportunistic locking in the Windows NT registry: See Microsoft Article [Q129202](#)

Note: Borland internal testing has not indicated this setting to be significant. However, some Borland customers have indicated this to solve the problem.

6. Improperly closing files such as due to loss of power or restarting a workstation or the server without

closing files first may cause this problem. Paradox tables are not designed to withstand such behavior. If this is a possibility in your environment, we recommend you use a Client Server database that can recover from such conditions.

7. Extremely large numbers of indexes, especially involving Referential Integrity can cause this problem and especially when using Windows NT as the server. Borland recommends using a Client Server database under this condition. However, if you are using Windows NT as your server, switching to Novell Netware or Windows 95 as the server may resolve the problem as well.
8. The one programmatic way you can make this error occur is if you attempt to post a duplicate value to a unique, non-primary index at the same time you attempt to open the same table. This problem only occurs if local share is set to False and only occurs on local drives.

## Unverified solutions

1. Windows 95 Only: Bring up the network properties screen on all Workstations and enter the netBEUI properties screen. On the advanced tab, make sure that "Set this protocol to be the default protocol" is checked.
2. Windows 95 Only: If the previous suggestion did not work, try removing the following protocols in order. Remove one at a time and then re-test your problem:
  1. NETBIOS support for IPX/SPX-compatible Protocol
  2. TCP/IP
  3. IPX/SPX-compatible Protocol

If the problem disappears, attempt to add back in all protocols **except** for the last one that was taken out. Again, make sure netBEUI's default protocol check box is checked.

3. Windows NT Only used as a Workstation: On the Network Bindings page of the Network Properties, set the NetBEUI Protocol to be at the top of all services. The TCP/IP stack is known for having a lot of overhead that might cause timing problems. Since NT will send requests back in the same protocol as it is sent, changing the bindings on a NT machine used as a server will have no effect.

## Other resources

1. [The Delphi Magazine](http://www.itecuk.com/Delmag/Paradox.htm) has a number of interesting articles on this subject as well. See [www.itecuk.com/Delmag/Paradox.htm](http://www.itecuk.com/Delmag/Paradox.htm) for details.

## Article #15212: Understanding the PARADOX.NET file with the BDE

Technical Information Database

TI212B.txt      Understanding the PARADOX.NET file with the BDE  
Category      :General Programming  
Platform      :All  
Product       :BDE All

### Description:

When using Paradox type table with the Borland Database Engine (BDE), a Paradox net file is used in the same way that Paradox does--to keep track of the number of users. Each Paradox and BDE user is given one count. So, for example, if three Paradox and three BDE users use the same Paradox net file, the user count would be six (one for each Paradox user, and one for each BDE user).

The net file also regulates access to tables. Table access is enforced through the use of lock files, which are written out to the directories containing Paradox tables. A lock file points to a particular net file, which has exclusive control over the table. This means that any user wanting access to the table must use the net file that controls the table.

If you are using Paradox for Windows 5.0 or greater, IDAPI is used for both Paradox and BDE. The "NET DIR" path is set in the IDAPICFG.EXE or BDECFG.EXE: BDE/IDAPI Configuration Utility. Once the configuration tool is running, hi-light PARADOX, which is in the "Driver Name" List Box on the "Drivers" page. Set the "NET DIR" path to a network drive where all users that are accessing the tables have write access to.

If you are using an older version of Paradox for Windows, Paradox for DOS, or the Paradox Engine, use the appropriate method for each of these products to set the netfile path to the identical path that is set within the IDAPI/BDE Configuration Utility. In the case of two BDE programs sharing tables, the net paths must be identical.

The message "Multiple Paradox net file found" (Error 0x2C06 or 11270: DBIERR\_NETMULTIPLE) indicates that these rules have not been followed. If you are certain that all current users have the same net path, this message usually indicates that an old lock file exists that points to a different net file. Old lock files can be deleted if care is taken to ensure that no one is currently using them. Deleting active lock files can produce unpredictable results and could cause loss of data.

Reference:

7/15/98 3:24:22 PM

Last Modified: 01-SEP-99

## Article #15214: Messaging Between Two Dialogs/Update Background

Technical Information Database

TI214D.txt      Messaging Between Two Dialogs/Update Background

Category        :TVISION

Platform        :All

Product         :Pascal    All

Description:

```
{
  This program will create two modeless dialog boxes and will
  allow only one dialog to be created at any time.

  Additionally, it demonstrates how to broadcast messages to
  background windows and update information without changing
  the selected dialog.
}
```

Program DialogCommunication;

uses Objects, Drivers, Views, Menus, Dialogs, App, Crt;

const

    cmDialog1 = 100;

    cmDialog2 = 101;

    cmDialog1Button = 200;

    cmDialog2Button = 201;

type

PHelloApp = ^THelloApp;

THelloApp = object(TApplication)

    procedure MakeDialog;

    procedure HandleEvent(var Event: TEvent); virtual;

    procedure InitMenuBar; virtual;

    procedure InitStatusLine; virtual;

end;

PMyDialog1 = ^TMyDialog1;

```
TMyDialog1 = object(TDialog)
  procedure MakeDialog2;
  procedure HandleEvent(var Event:TEvent); virtual;
end;
```

```
PMyDialog2 = ^TMyDialog2;
TMyDialog2 = object(TMyDialog1)
  procedure HandleEvent(var Event:TEvent); virtual;
end;
```

```
procedure TMyDialog1.HandleEvent(var Event: TEvent);
var
  AreYouThere: PView;
begin
  TDialog.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmDialog1Button:
        begin
          AreYouThere:= Message(DeskTop, evBroadcast,
cmDialog2, nil);
          if AreYouThere = nil then
            MakeDialog2
          else
            ClearEvent(Event);
        end
      else
        Exit;
    end;
    ClearEvent(Event);
  end;
end;
```

```
procedure TMyDialog1.MakeDialog2;
var
  Dialog2: PMyDialog2;
  R: TRect;
  Button: PButton;
begin
  R.Assign(1,1,40,20);
  Dialog2:= New(PMyDialog2, init(R,'Dialog2'));
  R.Assign(10,10,20,12);
  Button:= New(PButton,Init(R,'Beep', cmDialog2Button,
bfdefault));
```

```

Dialog2^.Insert(Button);
DeskTop^.Insert(Dialog2);
end;

procedure TMyDialog2.HandleEvent(var Event: TEvent);
begin
  case Event.Command of
    cmDialog2: begin
      sound(2000); delay(10); nosound;
      Title:=newstr('Hello world');
      ReDraw;
      ClearEvent(Event);
    end;
  end;
end;
TDialog.HandleEvent(Event);
if Event.What = evCommand then
  begin
    case Event.Command of
      cmDialog2Button: begin
        Sound(1000); delay(100); NoSound;
      end;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
end;

{ THelloApp }

procedure THelloApp.MakeDialog;
var
  R:TRect;
  Button1: PButton;
  Dialog1: PMyDialog1;
begin
  R.Assign(25, 5, 65, 16);
  Dialog1:= New(PMyDialog1, init(R,'Dialog1'));
  R.Assign(16, 8, 38, 10);
  Button1:= New(PButton, Init(R,'Call Dialog2', cmDialog1Button,
bfDefault));
  Dialog1^.Insert(Button1);
  DeskTop^.Insert(Dialog1);
end;

```

```

procedure THelloApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    begin
      case Event.Command of
        cmDialog1:begin
          MakeDialog;
        end;
      else
        Exit;
      end;
      ClearEvent(Event);
    end;
end;

```

```

procedure THelloApp.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~O~pen Dialogs', hcNoContext, NewMenu(
      NewItem('~D~ialog1', '', 0, cmDialog1, hcNoContext,
        NewLine(
          NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit,
            hcNoContext, nil))))), nil)))));
end;

```

```

procedure THelloApp.InitStatusLine;
var
  R: TRect;
begin
  GetExtent(R);
  R.A.Y := R.B.Y-1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('', kbF10, cmMenu,
        NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit, nil)), nil)));
end;

```

```

var
  HelloWorld: THelloApp;

```



```
begin
  HelloWorld.Init;
  HelloWorld.Run;
  HelloWorld.Done;
end.
```

Reference:

7/16/98 4:33:39 PM

Last Modified: 01-SEP-99

## Article #15216: IDE - AVAILABLE MEMORY

Technical Notes Database

TN216D.txt     IDE - AVAILABLE MEMORY  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal 4.0+

### Description:

- Q. When I compile a program and the compile window is displayed, what does the Available Memory mean?
- A. This is the amount of available memory during compilation. It has no significance after compilation.

### Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15221: Setting File Handles For A Windows BDE Application

Technical Information Database

TI221B.txt     Setting File Handles For A Windows BDE Application  
Category     :General Programming  
Platform     :All  
Product      :BDE 16 bit

### Description:

"Setting the number of file handles for Borland Database Engine Applications"

Solving the Borland Database Engine (BDE) error, "Not Enough file handles" (Error 0x2502 or 9474: DBIERR\_NOFILEHANDLES), is not a difficult task, but it requires an understanding of how file handles are used by DOS, Windows, and your application.

At the lowest level, the number of file handles available to any application running on your computer is specified in the "FILES=nn" statement of CONFIG.SYS. For Windows running with Windows and DOS applications, a bare minimum value is 30-40. A value of 80-120 is a more realistic starting value, which you should increase according to your application's needs.

When your application starts in Windows, the number of files available to it is based on the following two values: (1) The number you specify in a call to the Windows API call 'SetHandleCount' (default = 20), and (2) the number of files specified in the \_NFILE\_ constant in Borland's RTL source (default = 20). These two values should be the same; if they're not, the lower one will apply.

When estimating the number you'll need, note that in addition to whatever files you need for the BDE and for files you want to open, a C/C++ application will open five file handles for stdin, stdout, etc., on startup.

The number of file handles the BDE itself can use is specified on the System Page in the MaxFileHandles parameter of the BDE Configuration utility. It is important to realize that the number specified by MaxFileHandles is the number of file

handles that the BDE will expect to receive for its own exclusive use, not for use by your application as a whole. The BDE doesn't allocate file handles on its own, it gets these from your application. Thus, if you make a call to SetHandleCount with a value of 20 and set the MaxFileHandles parameter to 20, all twenty handles may go to the BDE, leaving none for your application's other needs.

To sum up, the number of file handles your BDE application will need will be the sum of 1) five standard file handles, 2) the number needed by the BDE (MaxFileHandles), and 3) the number you need for other file manipulation routines. This sum should be the number you specify in SetHandleCount, and if this total is greater than 20, you'll need to change the RTL source as well.

Reference:

7/15/98 3:24:23 PM

Last Modified: 11-NOV-99

## Article #15223: Associating Help Features with Context Hints

Technical Information Database

TI223D.txt      Associating Help Features with Context Hints

Category        :TVISION

Platform        :All

Product         :Pascal    All

Description:

{

This program has context help associated with the WINDOW selection in the main menu. The context help for the radio buttons in the dialog is supported as well.

}  
{ \$X+ }

program Hints;

uses Objects, Drivers, Views, Menus, Dialogs, App, Gadgets;

```
const
  cmNewDialog      = 100;
  hcMyDialog       = 300;
  hcCheck1         = 100;           {Declare help context handles}
  hcCheck2         = hcCheck1 + 1;
  hcRunny          = 200;
  hcMelted         = hcRunny + 1;
```

```
type
  PMyStatusLine = ^TMyStatusLine;
  TMyStatusLine = object(TStatusLine)
    function Hint(AHelpCtx : Word) : String; virtual; {Help     }
                                                {context}
    procedure HandleEvent(var Event : TEvent); virtual;
  end;
```

```
TMyApp = object(TApplication)
```

```

    constructor Init;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure InitMenuBar; virtual;
    procedure InitStatusLine; virtual;
    procedure NewDialog;
end;

PDemoDialog = ^TDemoDialog;
TDemoDialog = object(TDialog)
    MyStatusLine : PMyStatusLine;
end;

function TMyStatusLine.Hint(AHelpCtx : Word):String;
begin
    { Load Hint with corresponding menu string}
    { in insertion order }
    case AHelpCtx of
        hcCheck1    : Hint := 'Tilset Cheese Selection';
        hcCheck2    : Hint := 'Jarlsberg Cheese Selection';
        hcmyDialog  : Hint := 'DIALOG selection now pressed';
        hcMelted    : Hint := 'Melted Radio Button';
        hcRunny     : Hint := 'Runny Radio pressed';
    else
        Hint := '';
    end;
end;

procedure TMyStatusLine.HandleEvent(var Event : TEvent);
begin
    TStatusLine.HandleEvent(Event);
    Update;
end;

constructor TMyApp.Init;
var
    R : TRect;
begin
    TApplication.Init;
    GetExtent(R);
    Dec(R.B.X);
    R.A.X := R.B.X - 9; R.A.Y := R.B.Y - 1;
end;
{ TMyApp }

procedure TMyApp.HandleEvent(var Event: TEvent);
begin

```

```

TApplication.HandleEvent(Event);
if Event.What = evCommand then
begin
  case Event.Command of
    cmNewDialog: NewDialog;
  else
    Exit;
  end;
  ClearEvent(Event);
end;
end;

procedure TMyApp.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
        nil)),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('~D~ialog', 'F2', kbF2, cmNewDialog, hcmyDialog,
        nil)),
    nil))
  ))) ;
end;

procedure TMyApp.InitStatusLine;
var
  R: TRect;
begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PMyStatusLine, Init(R,
    NewStatusDef(0,0,
      NewStatusKey('', kbF10, cmMenu,
        NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
          NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
            nil))),
    NewStatusDef(10,10,
      nil,nil)))));
end;

```

```

procedure TMyApp.NewDialog;
var
  Borland: PView;
  Dialog: PDemoDialog;
  R: TRect;
  C: Word;
begin
  R.Assign(20, 6, 60, 19);
  Dialog := New(PDemoDialog, Init(R, 'Demo Dialog'));
  with Dialog^ do
  begin
    R.Assign(3, 3, 18, 5);
    Borland := New(PCheckBoxes, Init(R,
      NewSItem('~T~ilset',
        NewSItem('~J~arlsberg',
          nil)))
    );
    Borland^.HelpCtx := hcCheck1;
    Insert(Borland);
    R.Assign(2, 2, 10, 3);
    Insert(New(PLabel, Init(R, 'Cheeses', Borland)));
    R.Assign(22, 3, 34, 5);
    Borland := New(PRadioButtons, Init(R,
      NewSItem('~R~unny',
        NewSItem('~M~eltd',
          nil)))
    );
    Borland^.HelpCtx := hcRunny;
    Insert(Borland);
    R.Assign(21, 2, 33, 3);
    Insert(New(PLabel, Init(R, 'Consistency', Borland)));
    R.Assign(15, 8, 25, 10);
    Insert(New(PButton, Init(R, '~O~k', cmOK, bfDefault)));
    R.Assign(28, 8, 38, 10);
    Insert(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));
  end;
  DeskTop^.ExecView(Dialog);
  Dispose(Dialog, Done);
end;
var
  MyApp: TMyApp;

begin
  MyApp.Init;
  MyApp.Run;

```



```
MyApp.Done ;  
end.
```

Reference :

7/16/98 4:33:39 PM

Last Modified: 01-SEP-99

## Article #15225: Real number errors

Technical Notes Database

TN225D.txt      Real number errors

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 5.5

### Description:

Q. Why don't I get the expected results when I compare & print real numbers?

A. The problem with real numbers in Turbo Pascal is a problem with how a real number is stored in binary form. A binary number has no decimal point and thus a real number cannot directly translate into a binary number easily. Calculations must be performed to break a real number down into it's binary representation. As with any calculation that involves division or multiplication, small rounding errors will occur. The problem you are experiencing is a rounding error that occurs during translation from a real number into it's binary representation and back. I suggest that you round the results of your calculation to the number of decimal points that you require to alleviate the problem.

### Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15230: Changing the NET DIR Programmatically

Technical Information Database

TI230B.txt      Changing the NET DIR Programmatically  
Category      :General Programming  
Platform      :All  
Product      :BDE All

### Description:

#### General:

The "NET DIR" parameter for the Paradox driver is usually indicated in the BDE Configuration Utility on the drivers tab. The NET DIR indicates where the PDOXUSRS.NET file is located. If the NET DIR is pointing to a location not containing a PDOXUSRS.NET file, one will be created there.

#### Changing the NET DIR programmatically:

It is possible to change where the NET DIR is pointing programmatically by using the DbiSetProp() function, but first, a handle to the BDE session must be acquired. The session handle can be acquired through DbiGetCurrSession(). Here is how one could change the NET DIR:

```
void main(void)
{
    hDBISes hSes;

    DbiInit(NULL);
    DbiGetCurrSession(&hSes);
    DbiSetProp(hSes, sesNETFILE, (UINT32)"c:\\temp");
    Dbi...
    ...
}
```

#### Notes:

The sesNETFILE constant is an Object Property and a clearer definition can be found by searching for "Object Property" in the BDE On-Line help. Another way to change the default NET DIR is to pass DbiInit() a DBIEnv structure that specifies a different .cfg (configuration) file other than the default IDAPI.CFG (calling DbiInit(NULL) will use IDAPI.CFG). This

method is not recommended because only one configuration file can be used for all BDE applications.

Reference:

7/15/98 3:24:23 PM

Last Modified: 01-SEP-99

## Article #15232: How to Create a Listbox and Get a Double Click

Technical Information Database

TI232D.txt     How to Create a Listbox and Get a Double Click  
Category     :TVISION  
Platform     :All  
Product     :Pascal   All

Description:

{

The following example demonstrates how to create a listbox of file items and enable the user to double click on any item. This produces a Message Box containing the listbox selected item.

}

{ \$X+ }

Program Simple\_ListBox;

Uses

  Objects, Drivers, Views, Menus, Dialogs, App, Crt, Dos, MsgBox;

Const

  cmNewCollect = 102;

Type

TMyApp = object(TApplication)

  Constructor Init;

  Procedure Initstatusline; Virtual;

  Procedure Initmenubar; Virtual;

  Procedure NewCollect; Virtual;

  Procedure HandleEvent(var Event: TEvent); Virtual;

End;

PListBox = ^RListBox;

RListBox = object(TListBox)

  Constructor Init(var Bounds:TRect; ANumCols:Word; AScrollBar:  
                                  PScrollBar);

  Procedure Process;

```
Destructor Done; Virtual;  
End;
```

```
PMyDialog = ^MyDialog;  
MyDialog = object(TDialog)  
    Constructor Init(var Bounds: TRect; MyTitle:TTitleStr);  
    Destructor Done; Virtual;  
    Procedure HandleEvent(var Event: TEvent); Virtual;  
End;
```

```
Var  
    NameList: PStringCollection;  
    PList: ^Rlistbox;  
    MyApp: TMyApp;
```

```
Constructor Mydialog.Init(var Bounds:TRect; MyTitle: TTitleStr);  
Begin  
    TDialog.Init(Bounds, MyTitle);  
End;
```

```
Destructor Mydialog.Done;  
Begin  
    TDialog.Done;  
    Dispose(NameList, Done);  
End;
```

```
Procedure MyDialog.HandleEvent(var Event:TEvent);  
var  
    Index: integer;  
Begin  
    TDialog.HandleEvent(Event);  
    if (Event.Double = true) and (Event.What = evNothing) then  
        begin  
            sound(100);  
            delay(100);  
            nosound;  
            Index:=(PList^.Focused);  
            MessageBox(PList^.GetText(Index,20) ,nil, mfOkButton);  
        end;  
    ClearEvent(Event);  
End;
```

```
Constructor TMyApp.Init;  
Begin  
    TApplication.Init;
```

```

End;

Constructor RListBox.Init(var Bounds:TRect; ANumCols:Word;
AScrollBar:
                                PScrollBar);
Begin
  TListBox.Init(Bounds, ANumCols, AScrollBar);
End;

Destructor RListBox.Done;
Begin
  TListBox.Done;
End;

Procedure TMyApp.InitStatusLine;
Var
  R:TRect;
Begin
  GetExtent(R);
  R.A.Y := R.B.Y -1;
  Statusline :=new(PStatusLine, init(R,
NewStatusDef(0,$FFFF,
NewStatusKey('~Alt-X~ Exit',kbAltX, cmQuit,
NewStatusKey('~F10~ Menu',kbF10,cmMenu,
Nil)),Nil)
  ));
End;

Procedure RListBox.Process;
var
  DirInfo: SearchRec;
Begin
  NameList:=New(PStringCollection, Init(50,10));
  With NameList^ do
  Begin
    FindFirst('*.*', Archive, DirInfo);
    while DosError = 0 do
      Begin
        Insert(Newstr(Dirinfo.Name));
        FindNext(DirInfo);
      End;
    End;
  End;
End;

Procedure TMyApp.NewCollect;

```

```

Var
  MyBox: PMyDialog;
  R: TRect;
  PBorland: ^TScrollBar;
  Col: word;
Begin
  Plist^.Process;
  R.Assign(10,10,60,20);
  MyBox := New(PMyDialog, Init(R, 'Scroll Collection'));
  Col:=1;
  R.Assign(40, 1, 41, 9);
  PBorland:= New(PScrollBar, Init(R));
  R.Assign(9, 1, 40, 9);
  Plist:=New(PListbox, Init(R,Col,PBorland));
  Plist^.Newlist(Namelist);
  MyBox^.Insert(PBorland);
  MyBox^.Insert(PList);
  Desktop^.Insert(MyBox);
End;

```

```

Procedure TMyApp.HandleEvent(var Event: TEvent);
Begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
    Begin
      case Event.Command of
        cmNewCollect: Newcollect;
      else
        ClearEvent(Event);
        Exit;
      End;
      ClearEvent(Event);
    End;
End;

```

```

Procedure TMyApp.InitMenubar;
var
  R: TRect;
Begin
  GetExtent(R);
  R.B.Y := R.A.Y +1;
  MenuBar :=new(PMenubar, Init(R, Newmenu(
  NewSubMenu('~L~istbox Menu', hcNoContext, Newmenu(
  NewItem('~B~ox', '', kbF9, cmnewcollect, hcnocontext,
  NewLine(

```



```
NewItem('~E~xit','', kbf10, cmquit, hcnoccontext,  
Nil))))), Nil)))));  
End;
```

```
PRODUCT   : Turbo Pascal                      NUMBER   : 672  
VERSION   : 6.0  
OS        : MS/PC DOS  
DATE      : June 23, 1994                     PAGE     : 2/5  
  
TITLE     : How to Create a Listbox and Get a Double Click
```

```
Begin  
  MyApp.Init;  
  MyApp.Run;  
  MyApp.Done;  
End.
```

Reference:

7/16/98 4:33:39 PM

Last Modified: 01-SEP-99

## Article #15234: The location of GRAPH.TPU

Technical Notes Database

TN234D.txt     The location of GRAPH.TPU  
Category     :Pascal  
Platform     :All  
Product     :Turbo Pascal 5.5

Description:

Q. Where is the GRAPH.TPU file?

A. The GRAPH.TPU is archived in the BGI.ARC file. Use the UNPACK.COM program to dearchive GRAPH.TPU from the BGI.ARC file. For example:

```
UNPACK BGI
```

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15239: Error solutions and limitations of Paradox Engine

Technical Information Database

TI239B.txt Error solutions and limitations of Paradox Engine

Category :General Programming

Platform :All

Product :Paradox Engine All

Description:

Limitations of the Paradox Engine

=====

The Paradox Engine version 3.01 was intended for use on DOS 3.1 through DOS 5.x, and/or Windows 3.x environments. It can also be used in conjunction with the following networks:

- 1) IBM Token Ring version 1.1x
- 2) Novell NetWare version 2.0 through 3.12
- 3) 3Com 3+ Share network version 1.5.x
- 4) DEC Pathworks version 1.x
- 5) Microsoft LAN Manager version 2.x
- 6) Banyan Vines networks version 2.1x
- 7) AT&T StarGROUP for DOS version 3.1 through 5.x

The Paradox Engine has NOT been certified on the following OS or environments:

- 1) Windows NT 3.x
- 2) Windows 95
- 3) Windows for Workgroups 3.x
- 4) MS / IBM DOS 6.x

The Paradox Engine has NOT been certified on the following networks:

- 1) Novell NetWare version 4.x
- 2) Windows for Workgroups 3.x peer-to-peer
- 3) Windows NT 3.x peer-to-peer
- 4) Lantastic

The Paradox Engine has NOT been certified on the following chipsets:

- 1) Intel Pentium (any speed)

- 2) Cyrex
- 3) AMD

The Paradox engine relies on timing for some operations. The speed of some CPUs can cause file lock contentions in multi-user environments.

#### Common Problems that occur using the Paradox Engine

=====

Internal errors: Numbers > 200 -

- 1) Memory corruption and table corruption are the main causes of these internal errors.
- 2) These errors can be caused by using the Paradox Engine on tables that are also used by any IDAPI / BDE application; This includes products such as: Visual dBASE, Paradox, Quattro Pro. To temporarily fix the error, use the TUTILITY.EXE that ships with Paradox 4.0 for DOS and repair, not verify, the table. TUTILITY is available in the PDOXDOS forum on CompuServe. There is no permanent solution to this problem.

Random GPFs when converting from BC3.x to BC4.x -

A file named SETJMP.OBJ in the BCx\LIB\COMPAT directory needs to be linked into the EXE. RTTI necessitates new jump offsets for the Paradox Engine: this file contains the new offsets.

If an application crashes with the Paradox Engine still initialized, you MUST remove the .NET file and .LCK files associated with the program.

#### Specific Error Numbers with causes and solutions

=====

"Error# 4: File is locked" -

One possible way around this error is to create a retry loop which performs the desired operation x amount of times. The engine will do a validity check of the lock after approximately four iterations, which may clear up the contention. If CPU speed is too great i.e. Pentium 60Mhz, a greater number might be needed. See TI1002.

"Error# 8: Primary index is out of date" -

This error messages can occur if the sort order of the Engine app is different from the sort order of the table. A common cause of this is using the International sort order with Paradox while using the Engine with its default ASCII sort order. The sort order can be changed by using the PXSetDefaults function.

"Error# 15: Multiple Paradox net files found"

One paradox.net ( or pdoxusrs.net ) file controls access to any given table. This error occurs when two application using different net files try to access the same table. A common cause of this error is using different drive mappings on different machines. The Paradox Engine uses the directory path ( minus the drive letter ) to determine the location of the PDOXUSERS.NET file. Configure all workstations to reference the network control directory using the same path.

This error is also caused by using the same directory for controlling version 3.5 and version 4.0 locking mechanism. This would result in both a paradox.net and pdoxusrs.net file existing within the same directory. Delete the .net files and change the net directory of one of the application.

This error will also occur if there are old lock files ( .lck ) from a previous application lying about ( through a crash, for example ) which reference a different Paradox net file. Delete the .lck files.

#### WINDOWS:

The location of the net file is set in win.ini. There is a Windows utility provided, pxengcfg.exe, which will modify the setting in the win.ini file. Use this to utility to change the net directory.

Also make certain that all workstations are using the same DLL. Having the Engine 2.0 DLL on the path before the Engine 3.0 DLL will link in the wrong version of the Engine. This will cause network concurrency issues among other things.

"Error# 21: Insufficient password rights" -

Trying to access a 4.0 table from Paradox 3.5 or PXEng 2.0.

Can also happen when trying to run Paradox 4.0/PXEng 3.0 on the network from a directory controlled by PX 3.5 ( contains a paradox.net file ). Trying to access Paradox 5 or 7 tables with PXEng 3.x. The Paradox Engine only works with v4.x or earlier tables.

#### WINDOWS:

This error can be caused by using the wrong DLL. Check the path for old versions of the Paradox Engine DLL ( from the root directory, 'dir px\*.dll /s' ).

"Error 33: Invalid Parameter" -

From PXTblEmpty():

The PXTblEmpty function from v3.0 of the engine can incorrectly return this message. Workaround: create a new table with the same structure as the old, delete the original table ( PXTblDelete() ), and copy the new table back to the old.

From PXSrchFld():

This error can occur when the table has been opened on a secondary Index and an attempt is made to search on a field which is not part of the index.

"Error# 40: Not enough memory to complete operation" -

Use PXSetDefaults() for a DOS application, or pxengcfg.exe for a windows application, to increase the amount of swap buffer space available to the engine application. The documentation for PXSetDefaults() gives more information about the swap buffer. If in the IDE, make certain that Options | Debugger | Program Heap Size is set to 640K.

From PXTblOpen - Besides the above, this message can also mean that the table being used has become corrupted. Try running TUtility.exe on the table to test for corruption.

"Error# 50: Another user deleted record" -

This message can occur when a table has been opened more than once ( from one application or multiple applications) and a record deleted in one opening of the table is the current record for the other opening of the table. This is standard engine behavior, so some method will need to be

used to reposition the record cursor ( pointer that the engine uses for the current record ). Please see ti-1003 for more information regarding this error.

"Error# 53: Invalid offset into BLOb" -

This error can occur when the offset into a BLOb is larger than the size of the BLOb. A common situation which causes this error is calling the PxBlobPut() function when the offset plus the size of the buffer are larger than the size of the BLOb specified in the PxBlobOpenWrite() statement.

"Error# 89: Record not found" -

This message indicated that an exact match was not found. This is expected behavior when doing a CLOSESTRECORD search when an exact match is not found. There is a known problem with PXSrchFld() returning this message when it actually found the record. There is a maintenance release to the Paradox Engine 3.0 available on our local BBS at ( 408 ) 431-5096 in the 'patch' section, and on CompuServe in the BDEVTOOLS forum, lib 11. The file pxall.zip contains the libraries for the maintenance release. Disks are also available from our Customer Service department.

"Error# 96: Secondary index is out of date" -

Besides having an out of date index, this message can occur if the sort order of the Engine app is different from the sort order of the table. A common cause of this is using the Intl sort order with Paradox and using the Engine with its default ASCII sort order. The sort order can be changed by using the PXSetDefaults function.

"Error# 98: Could not login on network" -

This error comes up when the application does not have read, write, and create access to the network directory specified in either PXNetInit for DOS applications, or in win.ini for windows applications ). Can also be caused by using the LOCALSHARE parameter to PXNetInit() when share.exe is not loaded.

DOS:

make certain that the path uses double backslashes as opposed to single backslashes. Strings in C and C++

treat a single backslash as the escape character. For example, "\n" is a carriage return, line feed. Use "f:\\pdx\\netdir\\" to specify f:\pdx\netdir\.

#### WINDOWS:

Run the pxengcfg.exe utility to specify the net path.

"Error# 101: End of table" -

This error can be generated when the table is opened on the primary index, but a PXSrchFld() is being done on a field which is part of a single field secondary index. The table will be sorted according to the secondary index, but the end of table will be determined by the primary index. Open the table on the secondary index and do the search.

"Error# 107: Engine not initialized with PXNetInit" -

The Paradox Engine has not been initialized in a shareable mode. This can occur when the PXNetInit function with the NOTONNET parameter was used to initialize the engine. The NOTONNET parameter, as well as the NOSHARE parameter, do not allow the creation of any locks on your system.

"Error# 118: Table is busy" -

This error happens when the application tries to lock a table when the corresponding prevent lock already exists, or when the application tries to set a prevent lock and a conflicting lock already exists. Check for old, unused lock files. Delete any .lck files that may exist after all engine apps have terminated. Also run the TUtility program to test for table validity. See pg. 31 of the Paradox Engine users guide for more information on locks.

"Error# 120: Table is not found" -

Besides not being able to find the table, this error can also occur when the sort order that the engine uses is different from the sort order of the table. A common cause of this is using the Intl sort order with Paradox and using the Engine with its default ASCII sort order. The sort order can be changed by using the PXSetDefaults function.

"Error# 126: General Hardware Error" -



This error can occur from any network error which the Paradox Engine does not specifically handle. This error can also be generated when accessing a corrupted table. Run tutility.exe on the table to test for table corruption.

"Error# 134: Can't share Paradox net file -- is SHARE.EXE loaded?" -

Besides not having share loaded, this message can come up if share is loaded but the application does not have access to the directory where paradox.net is located. This error also occurs if the net file has gotten corrupted. Delete the net file to resolve this problem.

"Error# nnn: Internal Error #nnn " -

The most likely cause of this message is some sort of memory corruption,. Make certain that the code is compiled with all warnings turned on and test stack overflow turned on. Then try doubling the size of the stack because test stack overflow does not work in Windows or with any of our functions ). Memory corruption also occurs when a previous engine function failed but was not detected. Make certain that the return code for all engine functions is being checked. Get TI-738 on memory corruption and TI-1001 on Internal Errors. Also run TUtility on the table to test for table corruption. There were also certain cases where internal errors would be erroneously generated by v3.0 of the engine which were resolved in the maintenance release. The libraries, and the libraries only, to the Paradox Engine 3.01 maintenance release are available on our local BBS at ( 408 ) 439-9096 in the "patch" section. Disks are also available from our Customer Service department.

"Error# 400: Invalid Engine Type ( Database Framework )" -

The database library for the Paradox Engine 3.0 is built with "treat enums as ints". This means that the BEngine constructor is expecting a full two byte value coming in on the stack. If the project is built without this switch on, the compiler will assume the size of the BEngineType is a byte, and it will pass an argument which the libraries constructor will not read properly. Effectively, the compiler will do this:

```
mov al, 1 ; if we used pxLocal
push ax
```

which leaves garbage in the high order byte. This would be OK if the library was expecting a byte on the stack, but because it uses a whole word, it will read the garbage from the high order byte. Re-compile your application with the "Treat enums as ints" switch.

"Error 404: Table not open" -

This error can come from the testrest.cpp example which ships with the Paradox Engine and Database Framework if share is not loaded.

"Floating point error: Domain" -

Most likely caused by compiling the Paradox Engine in the small memory model. The Paradox Engine must be compiled in the LARGE memory model. Stack overflow can also cause this problem.

DISCLAIMER: You have the right to use this technical information subject to the terms of the No-Nonsense License Statement that you received with the Borland product to which this information pertains

Reference:

7/15/98 3:24:23 PM

Last Modified: 01-SEP-99

## Article #15241: Runtime Menubar and Statusline Change

Technical Information Database

TI241D.txt Runtime Menubar and Statusline Change  
Category :TVISION  
Platform :All  
Product :Pascal All

Description:

{

This program changes menus at runtime as well as changing the statusline. This program demonstrates how to change the statusline and the active commands linked to the statusline options.

}

{ \$X+ }

```
program ExampleProgram;
```

```
uses
```

```
  Drivers, Objects, Views, App, Menus, Puzzle, Calendar;
```

```
const
```

```
  PuzzleCmd    = 100;
```

```
  CalendarCmd  = 101;
```

```
  SwitchCmd    = 102;
```

```
var
```

```
  StatusLine1: PStatusLine;
```

```
type
```

```
  TTestMain = object(TApplication)
```

```
    OtherMenu: PMenuBar;
```

```
    constructor Init;
```

```
    procedure Calendar;
```

```
    procedure HandleEvent(var Event: TEvent); virtual;
```

```
    procedure InitMenuBar; virtual;
```

```
    procedure InitStatusLine; virtual;
```

```
    procedure Puzzle;
```

```

    procedure SwitchMenu;
end;

{ TTestMain }
constructor TTestMain.Init;
begin
    TApplication.Init;
end;

procedure TTestMain.Calendar;
var
    CalendarWindow: PCalendarWindow;
begin
    CalendarWindow := new(PCalendarWindow, Init);
    DeskTop^.Insert(CalendarWindow);
end;

procedure TTestMain.HandleEvent(var Event: TEvent);
begin
    TApplication.HandleEvent(Event);
    if Event.What = evCommand Then
    begin
        case Event.Command of
            PuzzleCmd    : Puzzle;
            CalendarCmd  : Calendar;
            SwitchCmd    : SwitchMenu;
        else
            Exit;
        end;
        ClearEvent(Event);
    end;
end;

Procedure TTestMain.InitMenuBar;
var
    R: TRect;
begin
    GetExtent(R);
    R.B.Y := R.A.Y+1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
       NewItem('~S~witch Menus', '', 0, SwitchCmd, hcNoContext,
        NewSubMenu('~E~xample', hcNoContext, NewMenu(
        NewItem('~P~uzzle', '', 0, PuzzleCmd, hcNoContext,
        NewItem('~C~alendar', '', 0, CalendarCmd, hcNoContext,
        NewLine(

```

```

       NewItem('~Q~uit', '', 0, cmQuit,
hcNoContext, Nil))))), Nil)))));
    OtherMenu := New(PMenuBar, Init(R, NewMenu(
        NewItem('~S~witch Menus', '', 0, SwitchCmd, hcNoContext,
        NewSubMenu('~O~ther Menu', hcNoContext, NewMenu(
            NewItem('~B~y', '', 0, cmQuit, hcNoContext,
            NewLine(
                NewItem('~Q~uit', '', 0, cmQuit,
hcNoContext, Nil))))), Nil)))));
end;

```

```

procedure TTestmain.InitStatusLine;
var R: TRect;
begin
    GetExtent(R);
    R.A.Y := R.B.Y - 1;
    StatusLine := New(PStatusLine, Init(R,
        NewStatusDef(0, $FFFF,
            NewStatusKey('', kbF10, cmMenu,
            NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
            nil)),
        nil)
    ));
    StatusLine1 := New(PStatusLine, Init(R,
        NewStatusDef(0, $FFFF,
            NewStatusKey('', kbF10, cmMenu,
            NewStatusKey('~P~uzzle', kbAltP, PuzzleCmd,
            nil)),
        nil)
    ));
end;

```

```

procedure TTestMain.Puzzle;
var
    PuzzleWindow: PPuzzleWindow;
begin
    PuzzleWindow := new(PPuzzleWindow, Init);
    DeskTop^.Insert(PuzzleWindow);
end;

```

```

procedure TTestMain.SwitchMenu;
var
    Temp: PMenuBar;

```

```
    Temp1: PStatusLine;
begin
    Delete(MenuBar);
    Temp := PMenuBar(MenuBar);
    MenuBar := OtherMenu;
    OtherMenu := Temp;
    Insert(MenuBar);
    Delete(StatusLine);
    Temp1 := PStatusLine(StatusLine);
    StatusLine := StatusLine1;
    StatusLine1 := Temp1;
    Insert(StatusLine);
    MenuBar^.DrawView;
    StatusLine^.DrawView;
end;
```

```
var
    Main: TTestMain;
```

```
begin
    Main.Init;
    Main.Run;
    Main.Done;
end.
```

Reference:

7/16/98 4:33:39 PM

Last Modified: 01-SEP-99

## Article #15243: Overwriting memory accidentally

Technical Notes Database

TN243D.txt      Overwriting memory accidentally

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 5.5

Description:

Q. How can my program be over-writing memory?

A. The most common causes for memory overwrites are:

1. Indexes out of range            (Turn range checking on {\$R+})
2. Uninitialized variables        (Write an initialization proc)
3. Pointers out of bounds        (Verify that pointers are not pointing outside of the heap space)
4. Improper use of FillChar or Move (Be sure to use the SizeOf function)
5. Illogical operations on strings

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15247: BDE setup for Peer-To-Peer(Non-Dedicated) Networks

Technical Information Database

TI247B.txt      BDE setup for Peer-To-Peer(Non-Dedicated) Networks

Category      :General Programming

Platform      :All

Product      :BDE 3.0

### Description:

Using a BDE32 Application on a Peer-To-Peer Network

-----

A Peer-To-Peer network (a network where each machine acts as a client and a server) can be one of the following, including other network platforms that are compatible with these:

- 1) Windows 95
- 2) Windows NT
- 3) Lantastic
- 4) Netware Lite

The BDE automatically detects when tables reside on a network drive, but it cannot detect whether the tables are on a dedicated server or a server/client. Dedicated servers notify client applications that a file has been modified or locked. This functionality is not present in Peer-To-Peer (non-dedicated) networks. To achieve this functionality with Peer-To-Peer networks set "LOCAL SHARE" to TRUE in the BDE Configuration Utility on the System page. This must be done on all BDE clients that access the tables on networks listed above. This is not necessary for Novell File Server type networks.

If the tables that are being used are Paradox, there must also be a directory used for network control. This directory must also reside on the network for all client applications to use. It is good practice to have a separate directory for the application, network, and tables. The following is an example:



(Shared Directory)

```
|  
|--- (Tables Directory)  
|--- (EXE Directory)  
|--- (Network Directory)
```

There are two different BDE environments that must also be considered:

- 1) Using only BDE 32Bit applications.
- 2) Using BDE 32Bit applications along with BDE 16Bit applications.

### Setup for 32Bit Only Applications

-----

The 32Bit BDE fully supports the UNC naming convention along with long file names. It is recommended that the UNC convention is used for all BDE network connections. UNC removes the need for mapped drives. This will allow access to the tables and network directory without the user being mapped to the drive. UNC has the following syntax:

```
\\(server name)\(share name)\(path)+(file name)
```

Here is a simple example of a standard BDE alias using UNC:

```
Alias: MyUNCAlias  
Type: STANDARD  
Path: \\FooServer\FooShare\Sharedir\Tables  
Default Driver: Paradox
```

The network directory can be setup in the same fashion:

```
Drivers: Paradox  
Net Dir: \\FooServer\FooShare\Sharedir\NetDir
```

The network directory can be set at runtime using `session.netfiledir` (Delphi) or `DbiSetProp` (C++ / Delphi)

If for some reason UNC cannot be used with the 32Bit application, follow directions for using BDE 32Bit and 16Bit applications.

## Setup for 16Bit and 32Bit BDE Applications

---

Since the 16Bit Windows API does not support UNC, neither does the 16Bit BDE. To allow applications to share the tables, all clients must be mapped to the same directory on the server. If the server is also used as a client, all other clients must be mapped to the root of the drive. Drive letters from client to client do not have to be identical. Here are some examples of what will and will not work:

Client1:

Path: X:\ShareDir\Tables

Client2:

Path: X:\ShareDir\Tables

This is OK

Client1: (Also the machine with the tables):

Path: C:\ShareDir\Tables

Client2:

Path: X:\ShareDir\Tables

This is OK

Client1: (Also the machine with the tables):

Path: C:\ShareDir\Tables

Client2:

Path: X:\ShareDir\Tables

Client3:

Path: R:\ShareDir\Tables

This is OK

Client1:

Path: X:\ShareDir\Tables

Client2:

Path: X:\Tables (Where X:\Tables is actually

X:\ShareDir\Tables, but shared on the ShareDir directory)

This will not work. The BDE must be able to make the same entry into the Network Control file.

In Summary (setup for Peer-To-Peer networks):

---

16 and / or 32Bit Applications:

1) Turn "LOCAL SHARE" to TRUE in the BDE Configuration

Utility.

- 2) Do not use the UNC naming convention.
- 3) Do not use tables with long file names.
- 4) Make sure that all clients are mapped to the same directory on the server.

32Bit Only Applications:

- 1) Turn "LOCAL SHARE" to TRUE in the BDE Configuration Utility
- 2) Use the UNC naming convention to achieve a path to the network directory and table directory.

If the above steps are not followed, users could be locked out of the tables getting error:

"Directory is controlled by other .NET file."

"File: (Path1) PDOXUSRS.LCK"

"Directory: (Path2)"

OR

"Multiple .NET files in use."

"File: (Path) PDOXUSRS.LCK"

Reference:

7/15/98 3:24:23 PM

Last Modified: 01-SEP-99

## Article #15249: Change Foreground and Background Color

Technical Information Database

TI249D.txt Change Foreground and Background Color

Category :TVISION

Platform :All

Product :Pascal All

Description:

{

This program will place a new foreground and background color and pattern on the desktop. It will also set the character that is displayed for the pattern.

This example: the \$05 defines the 0 for black foreground and the 5 for purple background.

}

```
program ColorManipulation;
```

```
uses
```

```
    Dos, Objects, Drivers, Memory, Views,  
    Menus, Dialogs, App;
```

```
type
```

```
    PMyBack = ^TMyBack;  
    TMyBack = object(TBackground)  
        constructor Init(var Bounds: TRect);  
    end;
```

```
    PMyApp = ^TMyApp;  
    TMyApp = object(TApplication)  
        MyBack: PMyBack;  
        constructor Init;  
        function GetPalette:PPalette; virtual;  
    end;
```

```
function TMyApp.GetPalette: PPalette;
```

```
const  
    MyBackColor : TPalette = CColor; { sets palette to CColor }
```

```
{ items }
```

```
begin
  MyBackColor[1]:=#$05;    { TBackGround Color Constant's first }
                          { number is background and second is }
                          { foreground }
  GetPalette :=
MyBackColor;
end;
```

```
constructor TMyBack.Init(var Bounds: TRect);
begin
  TBackground.Init(Bounds, '?');{ places ASCII 178 char as }
                                { pattern for text on desktop }
end;
```

```
constructor TMyApp.Init;
var
  R:TRect;
begin
  TApplication.Init;
  GetExtent(R);
  MyBack:= New(PMyBack, init(R));
  Desktop^.Background:= MyBack;
  Desktop^.Insert(Desktop^.Background);
end;
```

```
var
  TheApp: TMyApp;
begin
  TheApp.Init;
  TheApp.Run;
  TheApp.Done;
end.
```

Reference:

7/16/98 4:33:40 PM

Last Modified: 01-SEP-99

## Article #15251: Out of Memory during compilation

Technical Notes Database

TN251D.txt Out of Memory during compilation

Category :Pascal

Platform :All

Product :Turbo Pascal 5.5

Description:

Q. What do I do about running out of memory during compilation?

A. There are a number of solutions to this problem:

1. If Compiler/Destination is set to Memory, set it to Disk in the integrated environment.
2. If Options/Compile/Link buffer in the integrated environment is set to Memory, set it to Disk. Alternatively, if you're using 4.0, place a {\$L-} directive at the beginning of your program. Use the /L option to link to disk in the command-line compiler.
3. If you are using any memory-resident utilities, such as Sidekick and Superkey, remove them from memory.
4. If you are using TURBO.EXE, try using TPC.EXE instead - it takes up less memory.
5. Turn off any compiler directives which are not necessarily needed. By simply turning off range checking {\$R-} and software emulation {\$E-}, your code size will be reduced dramatically.
6. Move all units, except PRINTER.TPU, out of the TURBO.TPL file and into the installed units directory.

If none of these suggestions help, your program or unit may simply be too large to compile in the amount of memory available, and you may have to break in into two or more smaller units. Alternatively, if you're using 5.0+, you should consider using overlays.

Reference:

Last Modified: 01-SEP-99

## Article #15256: Removing "Lock file has grown too large" Error

Technical Information Database

TI256B.txt     Removing "Lock file has grown too large" Error  
Category     :General Programming  
Platform     :All  
Product      :BDE All

### Description:

Lock file has grown too large, Decimal:9495 Hex:2517  
-----

This problem is specific to Paradox tables and can be caused in any BDE (16 or 32 Bit) application that meet some or all of the following criteria:

- 1) The Executable is in the same directory as the table.
- 2) The Private Directory is not correctly set or not set at all.
- 3a) Delphi: Having a TTable open on a paradox table and then performing multiple TQuery operations.
- 3b) C / C++: Having a table open with DbOpenTable and then performing multiple queries with DbQueryExec and/or DbQueryExecDirect.
- 4) LOCAL SHARE set to true in the BDE Configuration Utility on the System page.

To solve the problem, make sure that your application has done ALL of the following:

- 1) Under the directory where the executable is, create three new directories: TABLES, PRIV, and NET. Place all the tables for the application into the TABLES directory.
- 2) Set the session's private directory to the PRIV directory. Take the following steps according to the software you are using.

DELPHI:

```
Session.PrivateDir := ExtractFilePath(ParamStr(0)) + 'PRIV';
```



```
C / C++:  
DbiSetPrivateDir(szPath);  
// szPath is the fully qualified path (not relative)  
// to the PRIV directory.
```

- 3) Set the session's network directory to the NET directory. Take the following steps according to the software you are using.

```
DELPHI:  
Session.NetFileDir := ExtractFilePath(ParamStr(0)) + 'NET';
```

```
C / C++:  
DbiSetProp(hSes, sesNETFILE, (UINT32)szPath);  
// szPath is the fully qualified path (not relative)  
// to the NET directory.
```

```
// hSes is the current session handle. This can be  
// retrieved using the DBiGetCurrSession function.
```

- 4) If LOCAL SHARE is set to true and you are not sharing tables between different applications at the same time, change LOCAL SHARE to false.

The above steps will correct the Lock File Too Large error.

Reference:

7/15/98 3:24:23 PM

Last Modified: 01-SEP-99

## Article #15258: Simple Pick List Example

Technical Information Database

TI258D.txt      Simple Pick List Example  
Category        :TVISION  
Platform        :All  
Product         :Pascal    All

Description:

{

The following example is to demonstrate the use of a Pick List. This Pick List will read a line of data and validate the field as well as keep a history of valid commands.

}

{ \$X+ }

```
Program PickListExample;
```

```
uses Objects, App, Menus, Drivers, Views, Dialogs, MsgBox;
```

```
type
```

```
  YourKeySet = set of char;
```

```
const
```

```
  cmMakeDialog = 101;
```

```
  ValidSet    : YourKeySet = [#0..#31, '0'..'9'];
```

```
type
```

```
  PFInputLine = ^TFInputLine;
```

```
  TFInputLine = object(TInputLine)
```

```
    ValidKeys : YourKeySet;
```

```
    constructor Init(var Bounds: TRect; AMaxLen: integer;  
                      ChrSet: YourKeySet);
```

```
    procedure HandleEvent(var Event: TEvent); virtual;
```

```
    procedure GetData(var Rec); virtual;
```

```
    procedure SetData(var Rec); virtual;
```

```
    function DataSize: word; virtual;
```

```
  end;
```

```

PMyApp = ^TMyApp;
TMyApp = object(TApplication)
  procedure InitMenuBar;virtual;
  procedure InitStatusLine;virtual;
  procedure MakeDialog;
  procedure HandleEvent(var Event: TEvent);virtual;
end;

constructor TFInputLine.Init(var Bounds: TRect; AMaxLen: integer;
                             ChrSet: YourKeySet);

begin
  TInputLine.Init(Bounds,AMaxLen);
  ValidKeys:= ChrSet;
end;

procedure TFInputLine.HandleEvent(var Event: TEvent);
{
  Data validation handle event:
  Please note that validation happens BEFORE the parent
  is called. This ensures that the key is not trapped by
  the parent method.
}

var
  Number : longint;
  Code : integer;
begin
  case Event.What of
    evKeyDown :
      begin
        if not(Event.CharCode in ValidKeys) then
          ClearEvent(Event)
        else
          if Data^ <> '' then
            begin
              val(Data^, Number, Code);
              if (Code <> 0) or (Number < 0) or (Number > 65535)
then
                begin
                  Data^ := '';
                  MessageBox('Valid number is 0 to
65536.',nil,mfOkButton);
                  ClearEvent(Event);
                end;
            end;
          end;
      end;
  end;
end;

```

```

        end;
    end;
end;
TInputLine.HandleEvent(Event);
end;

procedure TFIInputLine.GetData(var Rec);
var
    Code : integer;
begin
    Val(Data^,word(Rec), Code);
end;

procedure TFIInputLine.SetData(var Rec);
begin
    Str(word(Rec),Data^);
end;

function TFIInputLine.DataSize: word;
begin
    DataSize := SizeOf(word);
end;

procedure TMyApp.InitMenuBar;
var R: TRect;
begin
    GetExtent(R);
    R.B.Y := R.A.Y + 1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
        NewSubMenu('~T~est', hcNoContext, NewMenu(
           NewItem('~D~ialog', 'F2', kbF2, cmMakeDialog, hcNoContext,
            NewLine(
               NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
                nil))))),nil)))));
end;

procedure TMyApp.MakeDialog;
var
    R : TRect;
    Dialog : PDialog;
    InputLine : PInputLine;
    Lable,Hist : PView;
begin
    GetExtent(R);
    R.Assign(10,5,60,15);

```

```

Dialog := New(PDialog, Init(R, 'Demo Dialog'));
With Dialog^ do
begin
    R.Assign(5, 4, 20, 5);
    InputLine := New(PFInputLine, Init(R, 15, ValidSet));
    Insert(InputLine);
    R.Assign(5, 3, 20, 4);
    Lable := New(PLabel, Init(R, 'InputLine', Inputline));
    Insert(Lable);
    R.Assign(21, 4, 23, 5);
    Hist := New(PHistory, Init(R, InputLine, 0));
    Insert(Hist);
end;
Desktop^.ExecView(Dialog);
Dispose(Dialog, Done);
end;

```

```

procedure TMyApp.InitStatusLine;
var
    R : TRect;
begin
    GetExtent(R);
    R.A.Y := R.B.Y - 1;
    StatusLine := New(PStatusLine, Init(R,
        NewStatusDef(0, $FFFF,
            NewStatusKey('~A~lt-X', kbAltX, cmQuit,
                nil), nil)));
end;

```

```

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
    TApplication.HandleEvent(Event);
    if Event.What = evCommand then
begin
    case Event.Command of
        cmMakeDialog: MakeDialog;
    else
        Exit;
    end;
    ClearEvent(Event);
end;
end;
end;

```

```

var
    MyApp: TMyApp;

```

```
begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.
```

Reference:

7/16/98 4:33:40 PM

Last Modified: 01-SEP-99

## Article #15260: Exec procedure not executing subprograms

Technical Notes Database

TN260D.txt      Exec procedure not executing subprograms

Category      :Pascal

Platform       :All

Product        :Turbo Pascal 5.5

### Description:

Q. Why doesn't the Exec procedure execute my subprograms?

A. Make certain that you use the {\$M} directive to set the maximum heap size to the lowest possible value. If this is done, check the value of the variable DosError to diagnose other problems. What is DosError Returning after the call:

8) Not enough Memory: User needs to lower MAX Heap  
                          {\$M Stack, Min, Max}

2) File not found: User needs to specify the full Path and extension of the command. If you're trying to execute a DOS internal command, you need to use COMMAND.COM (see DIR example in manual).

### Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15265: Sharing Violation Error with Paradox Tables.

Technical Information Database

TI265B.txt      Sharing Violation Error with Paradox Tables  
Category      :General Programming  
Platform      :All  
Product        :BDE All

### Description:

This error is most commonly caused by a "Lock File Contention". A lock file contention occurs in this situation:

One user, User "A", is accessing one or more Paradox tables in a directory. "A" is closing the last table in the directory. The BDE detects that "A" is the last program accessing the tables in the directory. Since "A" is the last user on the tables, the .LCK files are going to be deleted. At this time, User "B" attempts to open a table in the same directory as "A" (who is about to delete the .LCK files). "B" opens the .LCK files to write an entry to. This is when "A" tries to delete the .LCK files that "B" has now opened. This will cause a sharing violation on the .LCK files. The share violation usually occurs on the last client to close the table.

### Solutions:

One solution is to override Windows error routines and ignore the error. In all test cases, then a sharing violation occurs, the table is still opened or closed.

Another solution is to keep the .LCK files in the table's directories. If the BDE detects that more than one Application (or Session) is using the tables, it will not attempt to delete the .LCK files therefore solving the problem.

If the BDE and your Paradox table all reside on the



same CPU another solution is turn off Local share in BDE Administrator. If Local Share is off .LCK files will not be created in the first place, so a sharing violation should not occur. Note that if tables are on a different CPU than the BDE setting Local Share has no significance because .LCK files are always created on remote drives.

1) C / C++ / Delphi / Paradox:

Use the Windows API SetErrorMode passing the SEM\_FAILCRITICALERRORS constant. You would only need to do this before opening and closing a table and then check to see if the table has actually been open. Please see Routine 1a and 1b below for a basic Delphi (any version) routine or Routine 2a and 2b for C / C++ routines. Paradox users will need to call into the appropriate DLL where SetErrorMode resides.

2) C / C++ / Delphi / Paradox:

Create a "Dummy" Paradox table in each table directory that the application uses. Example: executable is in directory C:MyProg. This executable opens tables in two different directories - C:MyTables and D:TempTbl. Create a "Dummy" table with the Database Desktop, or any other Paradox table creation utility, in both the C:MyTables and D:TempTbl directories. Once the tables are created, create an application that opens each of the "Dummy" programs at startup. Leave this program running at all times. If the executable can be placed on a server machine, like Windows NT, that is optimal. In this scenario, whenever applications are accessing the data, the server also has the "Dummy" tables open.

3) C / C++ / Delphi / Paradox:

Leave at least one table during the entire application's run. Open one table during startup and close the table at exit. You could even open a "Dummy" table described above.

4) C / C++ / Delphi:

Use DbtAcqPersistTableLock BDE function to create a lock on a table that does not exist (Delphi or C++

only).

Syntax:

```
(Delphi) Check(DbAcqPersistTableLock(Database1.Handle,  
'NoTable1.DB', szPARADOX));
```

```
(C++) rslt = DbAcqPersistTableLock(hDb, 'NoTable2.DB',  
szPARADOX);
```

NOTE: Each instance of the application must have a unique, non-existent table name or an attempt to place a lock in the directory will fail. If this method is used, an algorithm must be used to guarantee a unique table name. If a user can only run one instance of the program at a time, the network user name can be used as a table name.

5) Paradox ObjectPAL:

If your application is apt to cause .LCK files to be created and deleted often, the easiest (and least resource-intensive) way to prevent this is to place a read lock on a non-existent table within the directory when your application starts up. (Paradox allows us to place semaphore locks on tables that don't exist.) Since read locks don't conflict with one another, all users can do this, and the net result will be that the .LCK file will not be deleted until the last user exits the system.

Syntax:

At program startup:

```
Table.attach("DummyTbl.db")
```

```
Table.lock(Read)
```

At program shut-down:

```
Table.unlock(Read)
```

```
Table.unAttach
```

NOTES - These are observations on the problem:

1) The problem does not seem to occur on NT workstation machines. The application can be either

16 or 32 bit. SetErrorMode seems to work with both cases.

2) The problem is more likely to occur if the network protocol is netBEUI. If possible use netBIOS or IPX/SPX.

3) Constant closing and opening of tables will cause this error more often.

4) The error most commonly occurs on the close of the table, not the open. In this situation the closing application tries to delete files that another application now has open.

Example Routines:

Routine 1a) (Delphi Table Open Routine)

```
-----  
procedure TForm1.OpenTable(MyTable: TTable);  
begin  
  { Set the error mode to ignore critical errors }  
  SetErrorMode(SEM_FAILCRITICALERRORS);  
  { Open the table and check if it was opened }  
  MyTable.Open;  
  if MyTable.Active = False then  
  begin  
    { Retry opening the table }  
    MyTable.Open;  
    { If an error happens again, raise an exception }  
    if MyTable.Active = False then  
      raise  
        EDatabaseError.Create('Error Opening table');  
  end;  
  { Set the error mode back to the default }  
  SetErrorMode(0);  
end;
```

Routine 1b) (Delphi Table Close Routine)

-----

```

procedure TForm1.CloseTable(MyTable: TTable);
begin
  { Set the error mode to ignore critical errors }
  SetErrorMode(SEM_FAILCRITICALERRORS);
  { Close the table and check if it was closed }
  MyTable.Close;
  if MyTable.Active = True then
  begin
    { Retry closing the table }
    MyTable.Close;
    { If an error happens again, raise an exception }
    if MyTable.Active = True then
      raise
        EDatabaseError.Create('Error Closing table');
  end;
  { Set the error mode back to the default }
  SetErrorMode(0);
end;

```

Routine 2a) (C / C++ Table Open Routine)

```

-----
DbiResult OpenTable(hDBIDb hTmpDb, pCHAR szTblName,
                   phDBICur phTmpCur)
{
  DBIResult  rslt;

  // Set the error mode to ignore critical errors
  SetErrorMode(SEM_FAILCRITICALERRORS);

  // Open the table and check if it was opened
  rslt = DbiOpenTable(hTmpDb, szTblName, NULL,
                     NULL, NULL, NULL, dbiREADWRITE, dbiOPENSHARED,
                     xltFIELD, FALSE, NULL, phTmpCur);
  if (rslt != DBIERR_NONE)

    // Retry opening the table
    rslt = DbiOpenTable(hTmpDb, szTblName, NULL,
                       NULL, NULL, NULL, dbiREADWRITE, dbiOPENSHARED,
                       xltFIELD, FALSE, NULL, phTmpCur);

  // Set the error mode back to the default
  SetErrorMode(0);
  return rslt;
}

```

Routine 2b) (C / C++ Table Open Routine)

```
-----  
DBIResult CloseTable(phDBICur phTmpCur)  
{  
    DBIResult    rslt;  
  
    // Set the error mode to ignore critical errors  
    SetErrorMode(SEM_FAILCRITICALERRORS);  
  
    // Close the table and check if it was closed  
    rslt = DbiCloseCursor(phTmpCur);  
    if (rslt != DBIERR_NONE)  
  
        // Retry closing the table  
        rslt = DbiCloseCursor(phTmpCur);  
  
    // Set the error mode back to the default  
    SetErrorMode(0);  
    return rslt;  
}
```

Reference:

7/15/98 3:24:24 PM

Last Modified: 31-MAY-00

## Article #15267: Desktop Save State Using Streams

Technical Information Database

TI267D.txt Desktop Save State Using Streams

Category :TVISION

Platform :All

Product :Pascal All

Description:

{

This program example demonstrates how to save your desktop using a stream. It will give you the ability to insert windows on the desktop and save the current state in a .DSK file and load them from the file to restore its state. Streams REQUIRE registration of descendant object types. Turbo Pascal will check that all objects in the stream have been registered and produce a compiler error otherwise.

}

Program Simple\_Stream;

Uses Objects, Drivers, Views, Menus, App;

Type

PM MyApp = ^MyApp;

MyApp = Object(TApplication)

Constructor Init;

Procedure InitStatusLine; Virtual;

Procedure InitMenuBar; Virtual;

Procedure HandleEvent(var Event: TEvent); Virtual;

Procedure NewWindow;

Destructor Done; Virtual;

End;

PM Window = ^TMyWindow;

TMyWindow = Object(TWindow)

Constructor Init(Bounds: TRect; WinTitle: String; WindowNo:

Word);

```

End;

PMyDeskTop = ^TMyDeskTop;
TMyDeskTop = object(TDeskTop)
  Windw: PMyWindow;
  Constructor Load(var S: TBufStream);
  Procedure Store(var S: TBufStream); Virtual;
End;

```

```

Const
  cmNewWin    = 101;
  cmSaveDesk  = 102;
  cmLoadDesk  = 103;

RMyDeskTop: TStreamRec = (
  ObjType : 1001;
  VmtLink : ofs(KindOf(TMyDeskTop)^);
  Load    :
TMyDeskTop.Load;
  Store   :
TMyDeskTop.Store
);

```

```

RMyWindow: TStreamRec = (
  ObjType : 1002;
  VmtLink : ofs(KindOf(TMyWindow)^);
  Load    :
TMyWindow.Load;
  Store   :
TMyWindow.Store
);

```

```

Procedure SaveDeskStream;
Var
  SaveFile:TBufStream;
Begin
  SaveFile.Init('Rdesk.dsk',stCreate,1024);
  SaveFile.Put(DeskTop);
  If Savefile.Status <>0 then
    write('Bad Save', Savefile.Status);
  SaveFile.Done;
End;

```

```

Procedure LoadDeskStream;
Var

```

```

    SaveFile:TBufStream;
Begin
    SaveFile.Init('Rdesk.dsk',stOpen,1024);
    DeskTop^.Insert(PMyDeskTop(SaveFile.Get));
    If Savefile.Status <>0 then
        write('Bad Load',Savefile.Status)
    else
        DeskTop^.ReDraw;
    SaveFile.Done;
End;

{ ** MyApp ** }
Constructor MyApp.Init;
Begin
    TApplication.Init;
    RegisterType(RMyDesktop);
    RegisterType(RDesktop);
    RegisterType(Rwindow);
    RegisterType(RMywindow);
    RegisterType(RFrame);
    RegisterType(RMenuBar);
    RegisterType(RStatusLine);
    RegisterType(RBackground);
End;

Destructor MyApp.Done;
Begin
    TApplication.Done;
End;

Procedure MyApp.InitMenuBar;
Var
    R: TRect;
Begin
    GetExtent(R);
    R.B.Y := R.A.Y + 1;
    MenuBar := New(PMenuBar, Init(R, NewMenu(
        NewSubMenu('~F~ile', hcNoContext, NewMenu(
            NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
            NewLine(
                NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
                Nil))))) ,
        NewSubMenu('~D~eskTop', hcNoContext, NewMenu(
            NewItem('~S~ave Desktop', '', kbF2, cmSaveDesk, hcNoContext,
            NewItem('~L~oad Desktop', '', kbF3, cmLoadDesk, hcNoContext,

```



```
    nil))))),nil)))));  
End;
```

```
Procedure MyApp.InitStatusLine;
```

```
Var  
  R: TRect;  
Begin  
  GetExtent(R);  
  R.A.Y := R.B.Y - 1;  
  StatusLine := New(PStatusLine, Init(R,  
    NewStatusDef(0, $FFFF,  
      NewStatusKey('', kbF10, cmMenu,  
        NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,  
          NewStatusKey('~F4~ New', kbF4, cmNewWin,  
            NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,  
              nil))))),  
    nil)))));  
End;
```

```
Procedure MyApp.HandleEvent(var Event: TEvent);
```

```
Begin  
  TApplication.HandleEvent(Event);  
  if Event.What = evCommand then  
    Begin  
      Case Event.Command of  
        cmNewWin: NewWindow;  
        cmSaveDesk: SaveDeskStream;  
        cmLoadDesk: LoadDeskStream;  
      else  
        Exit;  
      End;  
      ClearEvent(Event);  
    End;  
End;
```

```
Procedure MyApp.NewWindow;
```

```
Var  
  Window: PMyWindow;  
  R: TRect;  
Begin  
  R.Assign(0, 0, 24, 7);  
  R.Move(Random(55), Random(16));  
  Window := New(PMyWindow, Init(R, 'Demo Window', 1));  
  DeskTop^.Insert(Window);  
End;
```

```

{ ** MyDeskTop **}
Constructor TMyDeskTop.Load(Var S: TBufStream);
Begin
  TDeskTop.Load(S);
  GetSubViewPtr(S,Window);
End;

Procedure TMyDeskTop.Store(Var S: TBufStream);
Begin
  TDeskTop.Store(S);
  PutSubViewPtr(S,Window);
End;

{ ** MyWindow **}
Constructor TMyWindow.Init(Bounds: TRect; WinTitle: String;
WindowNo: Word);
Begin
  TWindow.init(Bounds,WinTitle,WindowNo);
End;

{ Main Program }
Var
  MyTApp:MyApp;

Begin
  MyTApp.Init;
  MyTApp.Run;
  MyTApp.Done;
End.

```

Reference:

7/16/98 4:33:40 PM

Last Modified: 01-SEP-99

## Article #15269: Communicating with the serial port

Technical Notes Database

TN269D.txt      Communicating with the serial port

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 5.5

Description:

Q. How do you communicate with the serial port?

A. Modify the AuxInOut unit described in the Reference

Guide as follows: To read incoming data, use the Reset procedure before reading. Also, you should read from a buffer rather than directly from the serial port.

Furthermore, the AuxInOut is not an ideal example to use as input buffering is not supported; you should use the enclosed Technical Information Handout #407 routines for this purpose.

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15274: Steps for FAT32 Support with the BDE 3.5

Technical Information Database

TI274B.txt Steps for FAT32 Support with the BDE

Category : General Programming

Platform : All

Product : BDE 3.5

### Description:

Windows 95 version 950B and Windows 98 have a file system called FAT32. BDE Versions prior to version 3.5 will not work with this file system. BDE Versions after version 3.5 have no problem with FAT32. This document discusses how to make BDE version 3.5 work with FAT32.

The FAT file system had problems with large cluster sizes with large hard disks making storage inefficient. FAT32 keeps the cluster size at 4k enabling efficient data storage.

### COMMON BDE ERRORS EXPERIENCED WITH FAT32 SYSTEMS

-----

"Invalid File Name"

"Invalid Table Name"

"File Not Found"

"Table Not Found"

NOTE: All Delphi and CBuilder exceptions are of type EDBEngine error. The errors above are displayed within this exception type.

-----

To determine if you have a FAT32 system:

1) Right click the mouse on the My Computer icon on the desktop. Select the Properties menu item. On the general tab, under System, it will say 4.00.950 B. NOTE: 4.00.950A or 4.00.950 does not support FAT32. At the time this was written 4.00.950B is the latest. Microsoft may decide to create newer builds with FAT32 support.

2) Double click on the My Computer icon on the desktop. Right click on each local drive and select the Properties menu item. If any of the drive Types is Local Disk (FAT32), then you have a FAT32 partition on your system. Note: Local Disk (FAT) is not FAT32.

If in the above steps you have determined that you have a FAT32 file system, you can obtain a version of BDE 3.5 which does not have this problem from <http://www.borland.com/devsupport/bde/oldver.html>

## OTHER OPTIONS/WORKAROUNDS IF UPGRADE IS NOT AN OPTION

-----

If for any reason, the new DLL or set of DLLs cannot be placed on the system, you must follow the proceeding criteria:

- 1) All sub-directories must be 8 characters or less.
- 2) If tables must be in a 9 character or greater sub-directory, you can still open the table by
  - a) (Delphi, CBuilder) specifying the full path and file name in TTable.Table name, leaving DatabaseName blank.
  - b) (BC++) Passing NULL to the pszAliasName parameter and using the full path and file name in DbOpenTable.  
NOTE: DbSetDirectory will not work.
- 3) Table names can be greater than 8 characters with spaces.

Reference:

7/15/98 3:24:24 PM

Last Modified: 09-NOV-99

## Article #15276: Loop Processing in Windows

Technical Information Database

TI276D.txt    Loop Processing in Windows  
Category    :General Programming  
Platform    :All  
Product     :Pascal    All

### Description:

There are two basic ways of handling loop processing in Windows that would otherwise take away control (tie up processing of your code and not allow other processes to run) of other running Window's applications. The first and probably the most flexible approach is to write a processing loop and 'YIELD' to other Windows application's request. The second approach would be to override TApplication's MessageLoop method and within the MessageLoop call your process.

### 'YIELD'

If your procedure that takes a while to execute contains some sort of outer loop, make a call to a procedure that allows other apps to run. Your time consuming code might look something like:

```
...  
While MoreToDo do  
begin  
    YieldToOthers;  
    DoSomeProcessing;  
end;
```

A few notes concerning this procedure:

- \* The call to HALT may need to be modified to do any cleanup required (i.e.: close open files, etc.).
- \* You'll need to be sensitive to re-entrant issues.

The procedure "YieldToOthers" would look like:

```

Procedure YieldToOthers;
var
  Msg : TMsg;
begin
  While PeekMessage(Msg,0,0,0,PM_REMOVE) do begin
    if Msg.Message = WM_QUIT then begin
      Application^.Done;
      halt; {!!}
    end;
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
end;

```

'MessageLoop'

This is a replacement for the TApplication MessageLoop method. It provides for an 'idle loop' where your program can continuously process its own work, and yet still yield control to windows when windows needs to do something.

```

TMyApp = object(TApplication)
  procedure MessageLoop; virtual; { add this method to your }
end;      { * descendant of TApplication * }

```

```

{ * just paste this in to your program * }
procedure TMyApp.MessageLoop;
var
  Message: TMsg;
begin
  while true do begin
    if PeekMessage(Message, 0, 0, 0, pm_Remove) then begin
      if Message.Message = wm_Quit then Exit;
      TranslateMessage(Message);
      DispatchMessage(Message);
    end
    else begin

      { ** do your stuff here ** }

    end;
  end;
end;

Status := Message.WParam;

```

end;

Reference:

7/16/98 4:33:40 PM

Last Modified: 01-SEP-99



## Article #15278: Dumping Graphics to the printer

Technical Notes Database

TN278D.txt      Dumping Graphics to the printer

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.5

Description:

Q. How do you dump graphics to a printer?

A. Please see the enclosed Technical Information

Handouts, #433 and #432, for examples of printing graphics  
on Epson compatible and HP Laser Jet printers.

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15283: BDE Error listing

Technical Information Database

TI283B.txt BDE Error listing  
Category :General Programming  
Platform :WINDOWS  
Product :BDE

### Description:

This document lists all the errors that can be returned by the BDE. This information is derived from IDAPI.H (C++) or BDE.INT (C++ Builder and Delphi). You also wish to look at the following document(s) for addition information on specific errors:

15809 - Handling EDBEngineError Exceptions  
16437 - BDE and Database Desktop Locking Protocol

Note: You can use DbtGetErrorString() to retrieve the error string for any error.

### System Related (Fatal Error)

8449 : \$2101 : Cannot open a system file.  
8450 : \$2102 : I/O error on a system file.  
8451 : \$2103 : Data structure corruption.  
8452 : \$2104 : Cannot find Engine configuration file.  
8453 : \$2105 : Cannot write to Engine configuration file.  
8454 : \$2106 : Cannot initialize with different  
configuration file.  
8455 : \$2107 : System has been illegally re-entered.  
8456 : \$2108 : Cannot locate IDAPI32 .DLL.  
8457 : \$2109 : Cannot load IDAPI32 .DLL.  
8458 : \$210A : Cannot load an IDAPI service library.  
8459 : \$210B : Cannot create or open temporary file.  
8460 : \$210C : Trying to load multiple IDAPIxx.DLL  
8461 : \$210D : Shared Memory Conflict

### Object of Interest not Found

8705 : \$2201 : At beginning of table.

8706 : \$2202 : At end of table.  
8707 : \$2203 : Record moved because key value changed.  
8708 : \$2204 : Record/Key deleted.  
8709 : \$2205 : No current record.  
8710 : \$2206 : Could not find record.  
8711 : \$2207 : End of BLOB.  
8712 : \$2208 : Could not find object.  
8713 : \$2209 : Could not find family member.  
8714 : \$220A : BLOB file is missing.  
8715 : \$220B : Could not find language driver.

#### Physical Data Corruption

8961 : \$2301 : Corrupt table/index header.  
8962 : \$2302 : Corrupt file - other than header.  
8963 : \$2303 : Corrupt Memo/BLOB file.  
8965 : \$2305 : Corrupt index.  
8966 : \$2306 : Corrupt lock file.  
8967 : \$2307 : Corrupt family file.  
8968 : \$2308 : Corrupt or missing .VAL file.  
8969 : \$2309 : Foreign index file format.

#### I/O related error

9217 : \$2401 : Read failure.  
9218 : \$2402 : Write failure.  
9219 : \$2403 : Cannot access directory.  
9220 : \$2404 : File Delete operation failed.  
9221 : \$2405 : Cannot access file.  
9222 : \$2406 : Access to table disabled because of  
previous error.

#### Resource or Limit error

9473 : \$2501 : Insufficient memory for this operation.  
9474 : \$2502 : Not enough file handles.  
9475 : \$2503 : Insufficient disk space.  
9476 : \$2504 : Temporary table resource limit.  
9477 : \$2505 : Record size is too big for table.  
9478 : \$2506 : Too many open cursors.  
9479 : \$2507 : Table is full.  
9480 : \$2508 : Too many sessions from this workstation.  
9481 : \$2509 : Serial number limit (Paradox).  
9482 : \$250A : Some internal limit (see context).  
9483 : \$250B : Too many open tables.

9484 : \$250C : Too many cursors per table.  
9485 : \$250D : Too many record locks on table.  
9486 : \$250E : Too many clients.  
9487 : \$250F : Too many indexes on table.  
9488 : \$2510 : Too many sessions.  
9489 : \$2511 : Too many open databases.  
9490 : \$2512 : Too many passwords.  
9491 : \$2513 : Too many active drivers.  
9492 : \$2514 : Too many fields in Table Create.  
9493 : \$2515 : Too many table locks.  
9494 : \$2516 : Too many open BLOBs.  
9495 : \$2517 : Lock file has grown too large.  
9496 : \$2518 : Too many open queries.  
9498 : \$251A : Too many BLOBs.  
9499 : \$251B : File name is too long for a Paradox version  
5.0 table.  
9500 : \$251C : Row fetch limit exceeded.  
9501 : \$251D : Long name not allowed for this tablelevel.  
9502 : \$251E : Insufficient shared memory available.

#### Integrity Violation

9729 : \$2601 : Key violation.  
9730 : \$2602 : Minimum validity check failed.  
9731 : \$2603 : Maximum validity check failed.  
9732 : \$2604 : Field value required.  
9733 : \$2605 : Master record missing.  
9734 : \$2606 : Master has detail records. Cannot delete or  
modify.  
9735 : \$2607 : Master table level is incorrect.  
9736 : \$2608 : Field value out of lookup table range.  
9737 : \$2609 : Lookup Table Open operation failed.  
9738 : \$260A : Detail Table Open operation failed.  
9739 : \$260B : Master Table Open operation failed.  
9740 : \$260C : Field is blank.  
9741 : \$260D : Link to master table already defined.  
9742 : \$260E : Master table is open.  
9743 : \$260F : Detail table(s) exist.  
9744 : \$2610 : Master has detail records. Cannot empty it.  
9745 : \$2611 : Self referencing referential integrity must  
be entered one at a time with no other  
changes to the table  
9746 : \$2612 : Detail table is open.  
9747 : \$2613 : Cannot make this master a detail of another  
table if its details are not empty.

9748 : \$2614 : Referential integrity fields must be indexed.  
9749 : \$2615 : A table linked by referential integrity requires password to open.  
9750 : \$2616 : Field(s) linked to more than one master.  
9751 : \$2617 : Expression validity check failed.

#### Invalid Request

9985 : \$2701 : Number is out of range.  
9986 : \$2702 : Invalid parameter.  
9987 : \$2703 : Invalid file name.  
9988 : \$2704 : File does not exist.  
9989 : \$2705 : Invalid option.  
9990 : \$2706 : Invalid handle to the function.  
9991 : \$2707 : Unknown table type.  
9992 : \$2708 : Cannot open file.  
9993 : \$2709 : Cannot redefine primary key.  
9994 : \$270A : Cannot change this RINTDesc.  
9995 : \$270B : Foreign and primary key do not match.  
9996 : \$270C : Invalid modify request.  
9997 : \$270D : Index does not exist.  
9998 : \$270E : Invalid offset into the BLOB.  
9999 : \$270F : Invalid descriptor number.  
10000 : \$2710 : Invalid field type.  
10001 : \$2711 : Invalid field descriptor.  
10002 : \$2712 : Invalid field transformation.  
10003 : \$2713 : Invalid record structure.  
10004 : \$2714 : Invalid descriptor.  
10005 : \$2715 : Invalid array of index descriptors.  
10006 : \$2716 : Invalid array of validity check descriptors.  
10007 : \$2717 : Invalid array of referential integrity descriptors.  
10008 : \$2718 : Invalid ordering of tables during restructure.  
10009 : \$2719 : Name not unique in this context.  
10010 : \$271A : Index name required.  
10011 : \$271B : Invalid session handle.  
10012 : \$271C : invalid restructure operation.  
10013 : \$271D : Driver not known to system.  
10014 : \$271E : Unknown database.  
10015 : \$271F : Invalid password given.  
10016 : \$2720 : No callback function.  
10017 : \$2721 : Invalid callback buffer length.  
10018 : \$2722 : Invalid directory.  
10019 : \$2723 : Translate Error. Value out of bounds.

10020 : \$2724 : Cannot set cursor of one table to another.  
10021 : \$2725 : Bookmarks do not match table.  
10022 : \$2726 : Invalid index/tag name.  
10023 : \$2727 : Invalid index descriptor.  
10024 : \$2728 : Table does not exist.  
10025 : \$2729 : Table has too many users.  
10026 : \$272A : Cannot evaluate Key or Key does not pass  
filter condition.  
10027 : \$272B : Index already exists.  
10028 : \$272C : Index is open.  
10029 : \$272D : Invalid BLOB length.  
10030 : \$272E : Invalid BLOB handle in record buffer.  
10031 : \$272F : Table is open.  
10032 : \$2730 : Need to do (hard) restructure.  
10033 : \$2731 : Invalid mode.  
10034 : \$2732 : Cannot close index.  
10035 : \$2733 : Index is being used to order table.  
10036 : \$2734 : Unknown user name or password.  
10037 : \$2735 : Multi-level cascade is not supported.  
10038 : \$2736 : Invalid field name.  
10039 : \$2737 : Invalid table name.  
10040 : \$2738 : Invalid linked cursor expression.  
10041 : \$2739 : Name is reserved.  
10042 : \$273A : Invalid file extension.  
10043 : \$273B : Invalid language Driver.  
10044 : \$273C : Alias is not currently opened.  
10045 : \$273D : Incompatible record structures.  
10046 : \$273E : Name is reserved by DOS.  
10047 : \$273F : Destination must be indexed.  
10048 : \$2740 : Invalid index type  
10049 : \$2741 : Language Drivers of Table and Index do not  
match  
10050 : \$2742 : Filter handle is invalid  
10051 : \$2743 : Invalid Filter  
10052 : \$2744 : Invalid table create request  
10053 : \$2745 : Invalid table delete request  
10054 : \$2746 : Invalid index create request  
10055 : \$2747 : Invalid index delete request  
10056 : \$2748 : Invalid table specified  
10058 : \$274A : Invalid Time.  
10059 : \$274B : Invalid Date.  
10060 : \$274C : Invalid Datetime  
10061 : \$274D : Tables in different directories  
10062 : \$274E : Mismatch in the number of arguments  
10063 : \$274F : Function not found in service library.

10064 : \$2750 : Must use baseorder for this operation.  
10065 : \$2751 : Invalid procedure name  
10066 : \$2752 : The field map is invalid.

#### Locking/Contention related

10241 : \$2801 : Record locked by another user.  
10242 : \$2802 : Unlock failed.  
10243 : \$2803 : Table is busy.  
10244 : \$2804 : Directory is busy.  
10245 : \$2805 : File is locked.  
10246 : \$2806 : Directory is locked.  
10247 : \$2807 : Record already locked by this session.  
10248 : \$2808 : Object not locked.  
10249 : \$2809 : Lock time out.  
10250 : \$280A : Key group is locked.  
10251 : \$280B : Table lock was lost.  
10252 : \$280C : Exclusive access was lost.  
10253 : \$280D : Table cannot be opened for exclusive use.  
10254 : \$280E : Conflicting record lock in this session.  
10255 : \$280F : A deadlock was detected.  
10256 : \$2810 : A user transaction is already in progress.  
10257 : \$2811 : No user transaction is currently in progress.  
10258 : \$2812 : Record lock failed.  
10259 : \$2813 : Couldn't perform the edit because another  
user changed the record.  
10260 : \$2814 : Couldn't perform the edit because another  
user deleted or moved the record.

#### Access Violation - Security related

10497 : \$2901 : Insufficient field rights for operation.  
10498 : \$2902 : Insufficient table rights for operation.  
Password required.  
10499 : \$2903 : Insufficient family rights for operation.  
10500 : \$2904 : This directory is read only.  
10501 : \$2905 : Database is read only.  
10502 : \$2906 : Trying to modify read-only field.  
10503 : \$2907 : Encrypted dBASE tables not supported.  
10504 : \$2908 : Insufficient SQL rights for operation.

#### Invalid context

10753 : \$2A01 : Field is not a BLOB.  
10754 : \$2A02 : BLOB already opened.

10755 : \$2A03 : BLOB not opened.  
10756 : \$2A04 : Operation not applicable.  
10757 : \$2A05 : Table is not indexed.  
10758 : \$2A06 : Engine not initialized.  
10759 : \$2A07 : Attempt to re-initialize Engine.  
10760 : \$2A08 : Attempt to mix objects from different sessions.  
10761 : \$2A09 : Paradox driver not active.  
10762 : \$2A0A : Driver not loaded.  
10763 : \$2A0B : Table is read only.  
10764 : \$2A0C : No associated index.  
10765 : \$2A0D : Table(s) open. Cannot perform this operation.  
10766 : \$2A0E : Table does not support this operation.  
10767 : \$2A0F : Index is read only.  
10768 : \$2A10 : Table does not support this operation because it is not uniquely indexed.  
10769 : \$2A11 : Operation must be performed on the current session.  
10770 : \$2A12 : Invalid use of keyword.  
10771 : \$2A13 : Connection is in use by another statement.  
10772 : \$2A14 : Passthrough SQL connection must be shared

Os Error not handled by Idapi

11009 : \$2B01 : Invalid function number.  
11010 : \$2B02 : File or directory does not exist.  
11011 : \$2B03 : Path not found.  
11012 : \$2B04 : Too many open files. You may need to increase MAXFILEHANDLE limit in IDAPI configuration.  
11013 : \$2B05 : Permission denied.  
11014 : \$2B06 : Bad file number.  
11015 : \$2B07 : Memory blocks destroyed.  
11016 : \$2B08 : Not enough memory.  
11017 : \$2B09 : Invalid memory block address.  
11018 : \$2B0A : Invalid environment.  
11019 : \$2B0B : Invalid format.  
11020 : \$2B0C : Invalid access code.  
11021 : \$2B0D : Invalid data.  
11023 : \$2B0F : Device does not exist.  
11024 : \$2B10 : Attempt to remove current directory.  
11025 : \$2B11 : Not same device.  
11026 : \$2B12 : No more files.  
11027 : \$2B13 : Invalid argument.  
11028 : \$2B14 : Argument list is too long.  
11029 : \$2B15 : Execution format error.



11030 : \$2B16 : Cross-device link.  
11041 : \$2B21 : Math argument.  
11042 : \$2B22 : Result is too large.  
11043 : \$2B23 : File already exists.  
11047 : \$2B27 : Unknown internal operating system error.  
11058 : \$2B32 : Share violation.  
11059 : \$2B33 : Lock violation.  
11060 : \$2B34 : Critical DOS Error.  
11061 : \$2B35 : Drive not ready.  
11108 : \$2B64 : Not exact read/write.  
11109 : \$2B65 : Operating system network error.  
11110 : \$2B66 : Error from NOVELL file server.  
11111 : \$2B67 : NOVELL server out of memory.  
11112 : \$2B68 : Record already locked by this workstation.  
11113 : \$2B69 : Record not locked.

#### Network related

11265 : \$2C01 : Network initialization failed.  
11266 : \$2C02 : Network user limit exceeded.  
11267 : \$2C03 : Wrong .NET file version.  
11268 : \$2C04 : Cannot lock network file.  
11269 : \$2C05 : Directory is not private.  
11270 : \$2C06 : Directory is controlled by other .NET file.  
11271 : \$2C07 : Unknown network error.  
11272 : \$2C08 : Not initialized for accessing network files.  
11273 : \$2C09 : SHARE not loaded. It is required to share  
local files.  
11274 : \$2C0A : Not on a network. Not logged in or wrong  
network driver.  
11275 : \$2C0B : Lost communication with SQL server.  
11277 : \$2C0D : Cannot locate or connect to SQL server.  
11278 : \$2C0E : Cannot locate or connect to network server.

#### Optional parameter related

11521 : \$2D01 : Optional parameter is required.  
11522 : \$2D02 : Invalid optional parameter.

#### Query related

11777 : \$2E01 : obsolete  
11778 : \$2E02 : obsolete  
11779 : \$2E03 : Ambiguous use of ! (inclusion operator).  
11780 : \$2E04 : obsolete

11781 : \$2E05 : obsolete  
11782 : \$2E06 : A SET operation cannot be included in its own grouping.  
11783 : \$2E07 : Only numeric and date/time fields can be averaged.  
11784 : \$2E08 : Invalid expression.  
11785 : \$2E09 : Invalid OR expression.  
11786 : \$2E0A : obsolete  
11787 : \$2E0B : bitmap  
11788 : \$2E0C : CALC expression cannot be used in INSERT, DELETE, CHANGETO and SET rows.  
11789 : \$2E0D : Type error in CALC expression.  
11790 : \$2E0E : CHANGETO can be used in only one query form at a time.  
11791 : \$2E0F : Cannot modify CHANGED table.  
11792 : \$2E10 : A field can contain only one CHANGETO expression.  
11793 : \$2E11 : A field cannot contain more than one expression to be inserted.  
11794 : \$2E12 : obsolete  
11795 : \$2E13 : CHANGETO must be followed by the new value for the field.  
11796 : \$2E14 : Checkmark or CALC expressions cannot be used in FIND queries.  
11797 : \$2E15 : Cannot perform operation on CHANGED table together with a CHANGETO query.  
11798 : \$2E16 : chunk  
11799 : \$2E17 : More than 255 fields in ANSWER table.  
11800 : \$2E18 : AS must be followed by the name for the field in the ANSWER table.  
11801 : \$2E19 : DELETE can be used in only one query form at a time.  
11802 : \$2E1A : Cannot perform operation on DELETED table together with a DELETE query.  
11803 : \$2E1B : Cannot delete from the DELETED table.  
11804 : \$2E1C : Example element is used in two fields with incompatible types or with a BLOB.  
11805 : \$2E1D : Cannot use example elements in an OR expression.  
11806 : \$2E1E : Expression in this field has the wrong type.  
11807 : \$2E1F : Extra comma found.  
11808 : \$2E20 : Extra OR found.  
11809 : \$2E21 : One or more query rows do not contribute to the ANSWER.  
11810 : \$2E22 : FIND can be used in only one query form at a

time.

- 11811 : \$2E23 : FIND cannot be used with the ANSWER table.
- 11812 : \$2E24 : A row with GROUPLY must contain SET operations.
- 11813 : \$2E25 : GROUPLY can be used only in SET rows.
- 11814 : \$2E26 : Use only INSERT, DELETE, SET or FIND in leftmost column.
- 11815 : \$2E27 : Use only one INSERT, DELETE, SET or FIND per line.
- 11816 : \$2E28 : Syntax error in expression.
- 11817 : \$2E29 : INSERT can be used in only one query form at a time.
- 11818 : \$2E2A : Cannot perform operation on INSERTED table together with an INSERT query.
- 11819 : \$2E2B : INSERT, DELETE, CHANGETO and SET rows may not be checked.
- 11820 : \$2E2C : Field must contain an expression to insert (or be blank).
- 11821 : \$2E2D : Cannot insert into the INSERTED table.
- 11822 : \$2E2E : Variable is an array and cannot be accessed.
- 11823 : \$2E2F : Label
- 11824 : \$2E30 : Rows of example elements in CALC expression must be linked.
- 11825 : \$2E31 : Variable name is too long.
- 11826 : \$2E32 : Query may take a long time to process.
- 11827 : \$2E33 : Reserved word or one that can't be used as a variable name.
- 11828 : \$2E34 : Missing comma.
- 11829 : \$2E35 : Missing ).
- 11830 : \$2E36 : Missing right quote.
- 11831 : \$2E37 : Cannot specify duplicate column names.
- 11832 : \$2E38 : Query has no checked fields.
- 11833 : \$2E39 : Example element has no defining occurrence.
- 11834 : \$2E3A : No grouping is defined for SET operation.
- 11835 : \$2E3B : Query makes no sense.
- 11836 : \$2E3C : Cannot use patterns in this context.
- 11837 : \$2E3D : Date does not exist.
- 11838 : \$2E3E : Variable has not been assigned a value.
- 11839 : \$2E3F : Invalid use of example element in summary expression.
- 11840 : \$2E40 : Incomplete query statement. Query only contains a SET definition.
- 11841 : \$2E41 : Example element with ! makes no sense in expression.
- 11842 : \$2E42 : Example element cannot be used more than twice

with a ! query.

- 11843 : \$2E43 : Row cannot contain expression.
- 11844 : \$2E44 : obsolete
- 11845 : \$2E45 : obsolete
- 11846 : \$2E46 : No permission to insert or delete records.
- 11847 : \$2E47 : No permission to modify field.
- 11848 : \$2E48 : Field not found in table.
- 11849 : \$2E49 : Expecting a column separator in table header.
- 11850 : \$2E4A : Expecting a column separator in table.
- 11851 : \$2E4B : Expecting column name in table.
- 11852 : \$2E4C : Expecting table name.
- 11853 : \$2E4D : Expecting consistent number of columns in all rows of table.
- 11854 : \$2E4E : Cannot open table.
- 11855 : \$2E4F : Field appears more than once in table.
- 11856 : \$2E50 : This DELETE, CHANGE or INSERT query has no ANSWER.
- 11857 : \$2E51 : Query is not prepared. Properties unknown.
- 11858 : \$2E52 : DELETE rows cannot contain quantifier expression.
- 11859 : \$2E53 : Invalid expression in INSERT row.
- 11860 : \$2E54 : Invalid expression in INSERT row.
- 11861 : \$2E55 : Invalid expression in SET definition.
- 11862 : \$2E56 : row use
- 11863 : \$2E57 : SET keyword expected.
- 11864 : \$2E58 : Ambiguous use of example element.
- 11865 : \$2E59 : obsolete
- 11866 : \$2E5A : obsolete
- 11867 : \$2E5B : Only numeric fields can be summed.
- 11868 : \$2E5C : Table is write protected.
- 11869 : \$2E5D : Token not found.
- 11870 : \$2E5E : Cannot use example element with ! more than once in a single row.
- 11871 : \$2E5F : Type mismatch in expression.
- 11872 : \$2E60 : Query appears to ask two unrelated questions.
- 11873 : \$2E61 : Unused SET row.
- 11874 : \$2E62 : INSERT, DELETE, FIND, and SET can be used only in the leftmost column.
- 11875 : \$2E63 : CHANGETO cannot be used with INSERT, DELETE, SET or FIND.
- 11876 : \$2E64 : Expression must be followed by an example element defined in a SET.
- 11877 : \$2E65 : Lock failure.
- 11878 : \$2E66 : Expression is too long.
- 11879 : \$2E67 : Refresh exception during query.

11880 : \$2E68 : Query canceled.  
11881 : \$2E69 : Unexpected Database Engine error.  
11882 : \$2E6A : Not enough memory to finish operation.  
11883 : \$2E6B : Unexpected exception.  
11884 : \$2E6C : Feature not implemented yet in query.  
11885 : \$2E6D : Query format is not supported.  
11886 : \$2E6E : Query string is empty.  
11887 : \$2E6F : Attempted to prepare an empty query.  
11888 : \$2E70 : Buffer too small to contain query string.  
11889 : \$2E71 : Query was not previously parsed or prepared.  
11890 : \$2E72 : Function called with bad query handle.  
11891 : \$2E73 : QBE syntax error.  
11892 : \$2E74 : Query extended syntax field count error.  
11893 : \$2E75 : Field name in sort or field clause not found.  
11894 : \$2E76 : Table name in sort or field clause not found.  
11895 : \$2E77 : Operation is not supported on BLOB fields.  
11896 : \$2E78 : General BLOB error.  
11897 : \$2E79 : Query must be restarted.  
11898 : \$2E7A : Unknown answer table type.  
11926 : \$2E96 : Blob cannot be used as grouping field.  
11927 : \$2E97 : Query properties have not been fetched.  
11928 : \$2E98 : Answer table is of unsuitable type.  
11929 : \$2E99 : Answer table is not yet supported under server  
alias.  
11930 : \$2E9A : Non-null blob field required. Can't insert  
records  
11931 : \$2E9B : Unique index required to perform changeto  
11932 : \$2E9C : Unique index required to delete records  
11933 : \$2E9D : Update of table on the server failed.  
11934 : \$2E9E : Can't process this query remotely.  
11935 : \$2E9F : Unexpected end of command.  
11936 : \$2EA0 : Parameter not set in query string.  
11937 : \$2EA1 : Query string is too long.  
11946 : \$2EAA : No such table or correlation name.  
11947 : \$2EAB : Expression has ambiguous data type.  
11948 : \$2EAC : Field in order by must be in result set.  
11949 : \$2EAD : General parsing error.  
11950 : \$2EAE : Record or field constraint failed.  
11951 : \$2EAF : When GROUP BY exists, every simple field in  
projectors must be in GROUP BY.  
11952 : \$2EB0 : User defined function is not defined.  
11953 : \$2EB1 : Unknown error from User defined function.  
11954 : \$2EB2 : Single row subquery produced more than one row.  
11955 : \$2EB3 : Expressions in group by are not supported.  
11956 : \$2EB4 : Queries on text or ascii tables is not supported.

11957 : \$2EB5 : ANSI join keywords USING and NATURAL are not supported in this release.  
11958 : \$2EB6 : SELECT DISTINCT may not be used with UNION unless UNION ALL is used.  
11959 : \$2EB7 : GROUP BY is required when both aggregate and non-aggregate fields are used in result set.  
11960 : \$2EB8 : INSERT and UPDATE operations are not supported on autoincrement field type.  
11961 : \$2EB9 : UPDATE on Primary Key of a Master Table may modify more than one record.  
11962 : \$2EBA : Queries on MS ACCESS tables are not supported by local query engines.  
11963 : \$2EBB : Preparation of field-level constraint failed.  
11964 : \$2EBC : Preparation of field default failed.  
11965 : \$2EBD : Preparation of record-level constraint failed.  
11972 : \$2EC4 : Constraint Failed. Expression:

#### Version Mismatch Category

12033 : \$2F01 : Interface mismatch. Engine version different.  
12034 : \$2F02 : Index is out of date.  
12035 : \$2F03 : Older version (see context).  
12036 : \$2F04 : .VAL file is out of date.  
12037 : \$2F05 : BLOB file version is too old.  
12038 : \$2F06 : Query and Engine DLLs are mismatched.  
12039 : \$2F07 : Server is incompatible version.  
12040 : \$2F08 : Higher table level required

#### Capability not supported

12289 : \$3001 : Capability not supported.  
12290 : \$3002 : Not implemented yet.  
12291 : \$3003 : SQL replicas not supported.  
12292 : \$3004 : Non-blob column in table required to perform operation.  
12293 : \$3005 : Multiple connections not supported.  
12294 : \$3006 : Full dBASE expressions not supported.  
12295 : \$3007 : Nested transactions not supported.

#### System configuration error

12545 : \$3101 : Invalid database alias specification.  
12546 : \$3102 : Unknown database type.  
12547 : \$3103 : Corrupt system configuration file.  
12548 : \$3104 : Network type unknown.

12549 : \$3105 : Not on the network.  
12550 : \$3106 : Invalid configuration parameter.

#### Warnings

12801 : \$3201 : Object implicitly dropped.  
12802 : \$3202 : Object may be truncated.  
12803 : \$3203 : Object implicitly modified.  
12804 : \$3204 : Should field constraints be checked?  
12805 : \$3205 : Validity check field modified.  
12806 : \$3206 : Table level changed.  
12807 : \$3207 : Copy linked tables?  
12809 : \$3209 : Object implicitly truncated.  
12810 : \$320A : Validity check will not be enforced.  
12811 : \$320B : Multiple records found, but only one was expected.  
12812 : \$320C : Field will be trimmed, cannot put master records into PROBLEM table.

#### Miscellaneous

13057 : \$3301 : File already exists.  
13058 : \$3302 : BLOB has been modified.  
13059 : \$3303 : General SQL error.  
13060 : \$3304 : Table already exists.  
13061 : \$3305 : Paradox 1.0 tables are not supported.  
13062 : \$3306 : Update aborted.

#### Compatibility related

13313 : \$3401 : Different sort order.  
13314 : \$3402 : Directory in use by earlier version of Paradox.  
13315 : \$3403 : Needs Paradox 3.5-compatible language driver.

#### Data Repository related

13569 : \$3501 : Data Dictionary is corrupt  
13570 : \$3502 : Data Dictionary Info Blob corrupted  
13571 : \$3503 : Data Dictionary Schema is corrupt  
13572 : \$3504 : Attribute Type exists  
13573 : \$3505 : Invalid Object Type  
13574 : \$3506 : Invalid Relation Type  
13575 : \$3507 : View already exists  
13576 : \$3508 : No such View exists

13577 : \$3509 : Invalid Record Constraint  
13578 : \$350A : Object is in a Logical DB  
13579 : \$350B : Dictionary already exists  
13580 : \$350C : Dictionary does not exist  
13581 : \$350D : Dictionary database does not exist  
13582 : \$350E : Dictionary info is out of date - needs Refresh  
13584 : \$3510 : Invalid Dictionary Name  
13585 : \$3511 : Dependent Objects exist  
13586 : \$3512 : Too many Relationships for this Object Type  
13587 : \$3513 : Relationships to the Object exist  
13588 : \$3514 : Dictionary Exchange File is corrupt  
13589 : \$3515 : Dictionary Exchange File Version mismatch  
13590 : \$3516 : Dictionary Object Type Mismatch  
13591 : \$3517 : Object exists in Target Dictionary  
13592 : \$3518 : Cannot access Data Dictionary  
13593 : \$3519 : Cannot create Data Dictionary  
13594 : \$351A : Cannot open Database

#### Driver related

15873 : \$3E01 : Wrong driver name.  
15874 : \$3E02 : Wrong system version.  
15875 : \$3E03 : Wrong driver version.  
15876 : \$3E04 : Wrong driver type.  
15877 : \$3E05 : Cannot load driver.  
15878 : \$3E06 : Cannot load language driver.  
15879 : \$3E07 : Vendor initialization failed.  
15880 : \$3E08 : Your application is not enabled for use with  
this driver.

#### Reference:

7/15/98 3:24:24 PM

Last Modified: 19-OCT-99



## Article #15285: Creating and Using Fonts in the Resource Workshop

Technical Information Database

TI285D.txt      Creating and Using Fonts in the Resource Workshop  
Category      :General Programming  
Platform      :All  
Product       :Pascal    All

### Description:

Detailed below is a step by step description of how a Windows font file (.FON) is created using Borland's Resource Workshop and Turbo Pascal for Windows.

1. With the Resource Workshop, create a Resource file and save it to a file in compiled .RES format.. This is best done by creating a resource script (a file with the extension .RC) and selecting the .RES option for MultiSave found in the File/Preferences menu.
2. In Turbo Pascal for Windows, create a library similar to the one described below.

```
library NewFont;  
{ $D FONTRES 100,96,96 : NewFont 12 (VGA res) }  
{ $M 1024, 0 }  
{ $R NewFont }  
begin  
end.
```

The font NewFont is used in this example. You may give your fonts any name desired. Of the compiler directives, the Description directive gives the resolution of the font and provides a description of the font. In the example, the values 100, 96, 96 represent the Aspect, LogPixelsX, LogPixelsY for the device normally used with the font (in this case VGA). Aspect is calculated from the expression  $(100 * \text{AspectY}) \div (\text{AspectX})$ . The values AspectX, AspectY, LogPixelsX, and LogPixelsY are accessible through the API function GetDeviceCaps.

If you like, you may have more than one set of Aspect, LogPixelX, and LogPixelY values. For NewFont, this could take the form of

the following. Each set of values is separated by a semi-colon and space.

```
{ $D FONTRES 100,96,96; 200,96,48; 133,96,72 :  
      NewFont 12 (VGA res) }
```

For font files, the description directive must start with the term FONTRES in order for the font to be recognized as a valid Windows font file. It is also very important not to separate values within a set of Aspect, LogPixelX, LogPixelY values with spaces. Windows may not recognize a file with values so delimited.

3. Compile the library just created to a .DLL file.

4. Rename the .DLL file to one with the extension .FON. In the case of NewFont, NEWFONT.DLL would be renamed to NEWFONT.FON.

5. Often, .FON are copied into the Windows directory \WINDOWS\SYSTEM and then added using the Control Panel program.

Note, Microsoft's Control Panel program may not function correctly with fonts that do encompass the entire alphabet.

Reference:

7/16/98 4:33:41 PM

Last Modified: 01-SEP-99

## Article #15287: The Source of "Unit file format error"

Technical Notes Database

TN287D.txt     The Source of "Unit file format error"  
Category     :Pascal  
Platform     :All  
Product     :Turbo Pascal 5.5

### Description:

Q. Why am I getting a "Unit file format error" when I compile my program with the new Turbo Pascal compiler?

A. You are using a unit that has been compiled with a different Turbo Pascal version. From the main program, use the BUILD option to recompile all dependent units.

### Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15293: Changing Fonts in Edits Controls

Technical Information Database

TI293D.txt Changing Fonts in Edits Controls

Category :OWL

Platform :All

Product :Pascal All

Description:

```
{ $X+ }
program EditFonts;
{
  Changing an edit controls font settings.
}

uses WObjects, WinProcs, WinTypes, Strings;

const
  id_Font = 201;

Type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure Font(var Msg: TMessage);
      virtual id_First + id_Font;
    procedure SetupWindow; virtual;
  end;

var
  MyEdit: PEdit;
  Button: PButton;

{ TMyApplication }
procedure TMyApplication.InitMainWindow;
begin
```

```

MainWindow:= new(PMyWindow,init(nil,
                    'Changing TEdit Control''s Font'));
end;

Constructor TMyWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
    TWindow.Init(AParent, ATitle);
    MyEdit := new(PEdit, Init(
Self, 200, 'Information',
                1,1,100,50,0, False));
    Button := new(PButton, init(
Self, 201, '&Font',
                100,100,100,50,True));
end;

procedure TMyWindow.SetupWindow;
begin
    TWindow.SetupWindow;
    SetFocus(Button^.hWindow);
end;

procedure TMyWindow.Font(var Msg: TMessage);
var
    LogFont: TLogFont;
    DC: HDC;
begin
    { set up the LogFont structure }
    With LogFont do
        begin
            lfHeight := 15;
            lfWidth:=0;
            lfEscapement:=0;
            lfOrientation:=0;
            lfWeight:= FW_Normal or FW_Regular;
            lfItalic:= 0;
            lfUnderLine:= 0;
            lfStrikeOut:= 0;
            lfCharSet:= ANSI_CharSet;
            lfOutPrecision:= Out_Default_Precis;
            lfClipPrecision:= Clip_Default_Precis;
            lfQuality:=Default_Quality;
            lfPitchandFamily:= Default_Pitch and ff_DontCare;
            StrCopy(lfFaceName, 'Courier');
        end;
end;

```

```
DC := GetDC(HWindow);

{ send message to change the font }
SendMessage(MyEdit^.hWindow, WM_SetFont,
            CreateFontIndirect(LogFont) , 1);
DeleteObject(CreateFontIndirect(LogFont));

{ change the button's text to 'Done' }
SetWindowText(Button^.hWindow, 'Done');
{ must send message to button's paint }
{ to update the button's text          }
SendMessage(Button^.HWindow, wm_Paint, 0, 1);
ReleaseDC(HWindow, DC);
end;

var
  MyApp: TMyApplication;
begin
  MyApp.Init('MyProgram');
  MyApp.Run;
  MyApp.Done;
end.
```

Reference:

7/16/98 4:33:41 PM

Last Modified: 01-SEP-99

## Article #15295: Redirecting printer output

Technical Notes Database

TN295D.txt     Redirecting printer output

Category     :Pascal

Platform     :All

Product      :Turbo Pascal 5.5

Description:

Q. How do you send a program's output to the printer?

A. Use the LST file variable declared in the PRINTER unit. For example:

```
Program SendToPrinter;  
Uses Printer;  
Begin  
  Writeln(LST,'This will go to the printer.');
```

End.

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15301: Use Collections and ListBoxes.

Technical Information Database

TI301D.txt Use Collections and ListBoxes.

Category :OWL

Platform :All

Product :Pascal All

Description:

{ \$X+ }

( \*

The Resource file will only have one item in it. A dialog with a listbox and an OK button. The listbox ID should be 101 and the OK Button ID should be 1. The name of the dialog is NewDialog.

\* )

```
program Initialization_ListBox_with_Collection;
```

```
{ $R CollDlg.res }
```

```
uses WinTypes, winprocs, wobjectb, strings, BWCC;
```

```
var
```

```
    st1,st2,st3: PChar;
```

```
const
```

```
    buttonid = 100;
```

```
    listid = 101;
```

```
type
```

```
    Plbtrans = ^Tlbtrans;
```

```
    Tlbtrans = record
```

```
        Strings: PStrCollection;
```

```
        Selection: integer;
```

```
    end;
```

```
type
```

```
    typePMyApp = ^TMyApp;
```



```

TMyApp = object(TApplication)
  procedure InitMainWindow; virtual;
end;

PMyMainWindow = ^TMyMainWindow;
TMyMainWindow = object(TWindow)
  But: PButton;
  Constructor Init(AName: PChar);
  Destructor Done; virtual;
  procedure SetUpWindow; virtual;
  procedure TestButton(var Msg:TMessage);
    virtual id_First + ButtonId;
end;

```

```

var
  Buf: TlbTrans;

```

```

procedure TMyApp.InitMainWindow;
begin
  MainWindow:= New(PMyMainWindow, init('Main Window'));
end;

```

```

constructor TMyMainWindow.Init(AName: PChar);
begin
  TWindow.Init(nil, AName);
  but:= New(PButton,init(
self,ButtonId,'&Test',10,10,
    50,50,false));
end;

```

```

Destructor TMyMainWindow.Done;
{ dispose of collection }
begin
  TWindow.Done;
  Dispose(But);
  FreeMem(St2,8);
  FreeMem(St3,8);
end;

```

```

procedure TMyMainWindow.SetUpWindow;
begin
  TWindow.SetUpWindow;
end;

```

```

procedure TMyMainWindow.TestButton(var Msg: TMessage);

```

```

{ Setup DialogBox with a ListBox }
var
  TheDialog: PDialog;
  LB: PListBox;
  S: PChar;
begin
  TheDialog:= new(PDialog,init(
Self,'NewDialog'));
  New(LB,InitResource(TheDialog,101));
  { create collection }
  Buf.Strings:=New(PStrCollection, init(10,10));
  GetMem(St1,8);
  StrCopy(St1,'Hello');
  Buf.Strings^.Insert(St1);
  GetMem(st2,8);
  StrCopy(St2,'Goodbye');
  Buf.Strings^.Insert(St2);
  GetMem(st3,8);
  StrCopy(St3,'Hey');
  Buf.Strings^.Insert(St3);
  Buf.Selection:=1;
  TheDialog^.TransferBuffer:=
Buf;
  Application^.ExecDialog(TheDialog);
  GetMem(S, 8);
  S:= buf.Strings^.at(Buf.Selection);
  MessageBox(hWindow,S,'Data from ListBox',mb_Ok);
  FreeMem(S, 8);
end;

var
  MyApp: TMyApp;
begin
  MyApp.Init('MyApp');
  MyApp.Run;
  MyApp.Done;
end.

```

Reference:

7/16/98 4:33:41 PM

Last Modified: 01-SEP-99

## Article #15303: Reading and Writing Files Examples

Technical Notes Database

TN303D.txt      Reading and Writing Files Examples

Category        :Pascal

Platform        :All

Product         :Turbo Pascal    5.5

Description:

Q. How do you read and write a file inside a Turbo Pascal program?

A. The following example demonstrates how to create, write and read from a text file.

```
Program FileDemo;
Var FileVar : Text;
    InString,OutString : String;
Begin
    OutString := 'Write this to a text file';
    Assign(FileVar, 'TEST.TXT');
    Rewrite(FileVar);           {Creates a file for writing}
    Writeln(FileVar,OutString);
    Close(FileVar);
    Assign(FileVar, 'TEST.TXT');
    Reset(FileVar);           {Opens an existing file for reading}
    ReadLn(FileVar,InString);
    Close(FileVar);
End.
```

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15309: Different Colors in Turbo Vision.

Technical Information Database

TI309D.txt Different Colors in Turbo Vision.

Category :TVISION

Platform :All

Product :Pascal All

Description:

Program ColorTV;

uses Objects, Drivers, Views, Menus, Dialogs, App, StdInput;

const

  cmButton = 101;

  cmNewWin = 102;

  cmNewDlg = 103;

  WinCount : integer = 0; { number of current windows }

type

PApp = ^TApp;

TApp = object(TApplication)

  procedure InitMenuBar; virtual;

  procedure InitStatusLine; virtual;

  procedure HandleEvent(var Event : TEvent); virtual;

  procedure NewWindow;

  procedure NewDialog;

  function GetPalette : PPalette; virtual;

end;

PWin = ^TWin;

TWin = object(TWindow)

  constructor Init(Bounds: TRect; WinTitle: String;  
                  WindowNo: Word);

  function GetPalette : PPalette; virtual;

end;

PInLine = ^TInLine;

```

TInLine = object(TInputReal)
  function GetPalette : PPalette; virtual;
end;

PDlg = ^TDlg;
TDlg = object(TDialog)
  function GetPalette : PPalette; virtual;
end;

{ TWin }
constructor TWin.Init(Bounds: TRect; WinTitle: String;
                     WindowNo: Word);

var
  S : string[3];
  R : TRect;
  MyInputLine : PInLine;
begin
  Str(WindowNo, S);
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  GetClipRect(Bounds);
  Bounds.Grow(-1,-1);

  { insert input string }
  R.Assign(8, 2, 20, 3);
  Insert(New(PInLine, Init(R, 12)));

  { insert button }
  R.Assign(8, 5, 20, 7);
  Insert(New(PButton, Init(R, '~O~k', cmButton, bfDefault)));

  { move to the top }
  SelectNext(false);

end;

function TWin.GetPalette : PPalette;
{ add color indexes to TWin to allow button to be displayed in }
{ the window. notice that TButton's palette starts mapping }
{ from 10 - 15, and TWindow's palette runs from 1 - 8, there- }
{ fore add the indexes for the button on the end of TWin }
{ palettes. }
const
  CWin = CBlueWindow + #0#41#42#43#44#45#64;
  PWin : string[Length(CWin)] = CWin;

```

```

begin
  GetPalette :=
  PWin;
end;

{ TApp }
procedure TApp.InitMenuBar;
var
  Menu: TRect;
begin
  GetExtent(Menu);
  Menu.B.Y := Menu.A.Y + 1;
  MenuBar := New(PMenuBar, Init(Menu, NewMenu(
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
     NewItem('~W~indow', '', kbF4, cmNewWin, hcNoContext,
      NewItem('~D~ialog', '', kbF5, cmNewDlg, hcNoContext,
      NewLine(
        NewItem('~C~lose', 'Alt-F3', kbAltF3, cmClose, hcNoContext,
        NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
        nil)))))),
    nil))));
end;

procedure TApp.InitStatusLine;
var
  Line: TRect;
begin
  GetExtent(Line);
  Line.A.Y := Line.B.Y-1;
  StatusLine := New(PStatusLine, Init(Line,
    NewStatusDef(0, $FFFF,
      NewStatusKey('', kbF10, cmMenu,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
      NewStatusKey('~F4~ Window', kbF4, cmNewWin,
      NewStatusKey('~F5~ Dialog', kbF5, cmNewDlg,
      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
      nil))))) ,
    nil));
end;

procedure TApp.NewWindow;
var
  Window : PWin;
  Box      : TRect;
begin

```

```
Inc(WinCount);
Box.Assign(0, 0, 30, 8);
Box.Move(Random(34), Random(11));
Window := New(PWin, Init(Box, 'Test Window', WinCount));
DeskTop^.Insert(Window);
end;
```

```
procedure TApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewWin : NewWindow;
      cmNewDlg : NewDialog;
      cmButton : ;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
end;
```

```
procedure TApp.NewDialog;
var
  Dialog      : PDlg;
  R           : TRect;
  C           : Word;
begin

  R.Assign(25, 5, 55, 13);
  Dialog := New(PDlg, Init(R, 'Test Dialog'));

  with Dialog^ do
  begin

    { message }
    R.Assign(9, 2, 27, 3);
    Insert(New(PStaticText, Init(R, 'Error Message')));

    { insert button }
    R.Assign(10, 5, 20, 7);
    Insert(New(PButton, Init(R, '~O~k', cmOK, bfDefault)));

  end;
```



```

C := DeskTop^.ExecView(Dialog);

Dispose(Dialog, Done);
end;

function TApp.GetPalette : PPalette;
{ this is the palette that all other views are mapped to. }
{ the first 64 (0..63) are defined, any other colors are }
{ added from 64 on. the #510 is for TWin for the back- }
{ ground color for the button. from 547 on maps the new }
{ color for TDlg and its views. }
const
  CApp = CColor + #510#547#54E#540#54F#530#53E;
  PApp : string[Length(CApp)] = CApp;
begin
  GetPalette :=
    PApp;
end;

{ TInLine }
function TInLine.GetPalette : PPalette;
{ color indexes into TWin color palette }
const
  CInLine = #4#4#3#3;
  PInLine : string[Length(CInLine)] = CInLine;
begin
  GetPalette :=
    PInLine;
end;

{ TDlg }
function TDlg.GetPalette : PPalette;
{ when replacing colors for a view must replace the entire }
{ string. notice that any color > 63 maps to a user defined }
{ color from TApp's palette (65 = 547, 66 = 54E, etc.). }
{ i.e. }
{ index 2, the active frame maps to 65 which is color 547 }
{ (or grey on red). }
{ index 3, the frame icon maps to 66->54E (yellow on red). }
{ index 46, the button shadow maps to 67->540 (black on rec). }
{ index 37, the statictext maps to 68->547 (white on red). }
{ (no reason 46 is before 38). }
{ index 41-43, button stuff 69->530 (cyan on black). }
{ index 45, button shortcut 70->53E (cyan on yellow). }

```

```
const
  CDlg = #32#65#66#35#36#68#38#39#40#69#69#69#44#70#67#47#48#49 +
         #50#51#52#53#54#55#56#57#58#59#60#61#62#63;
  PDlg : string[Length(CDlg)] = CDlg;
begin
  GetPalette :=
  PDlg;
end;

var
  MyApp : TApp;

{ main }
begin

  MyApp.Init;
  MyApp.Run;
  MyApp.Done;

end.
```

Reference:

7/16/98 4:33:41 PM

Last Modified: 01-SEP-99

## Article #15311: DEBUGGING EXTERNAL ASSEMBLY ROUTINES

Technical Notes Database

TN311D.txt      DEBUGGING EXTERNAL ASSEMBLY ROUTINES

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 5.0

### Description:

Q. Why can't I trace through my external assembly language routines which have been linked with my TP program?

A. The Pascal linker does not pull through the symbolic debug information which is necessary to do source level debugging. Therefore, you will not be able to see your TASM source code. However, if you pull up the CPU window, you can trace through the disassembled ASM code. You will not have the benefit of variable names and labels, but you will be able to see the flow of execution.

### Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15317: Turbo Pascal vs ANSI Pascal

Technical Information Database

TI317D.txt - Turbo Pascal vs ANSI Pascal

Category : Turbo Pascal  
Platform : All  
Product :

Description:

=====  
Comparing Turbo Pascal 6.0 with ANSI Pascal  
=====

-----  
Table of Contents  
-----

1. Exceptions to ANSI Pascal requirements
  2. Extensions to ANSI Pascal
  3. Implementation-dependent features
  4. Treatment of errors
- 

This appendix compares Turbo Pascal to ANSI Pascal as defined by ANSI/IEEE770X3.97-1983 in the book American National Standard Pascal Computer Programming Language (ISBN 0-471-88944-X, published by The Institute of Electrical and Electronics Engineers in New York).

- =====  
1. Exceptions to ANSI Pascal requirements  
=====

Turbo Pascal is unable to detect whether a program violates any of the exceptions listed here.

Turbo Pascal complies with the requirements of

ANSI/IEEE770X3.97-1983 with the following exceptions:

- o In ANSI Pascal, an identifier can be of any length and all characters are significant. In Turbo Pascal, an identifier can be of any length, but only the first 63 characters are significant.
  
- o In ANSI Pascal, the `^` symbol is an alternative for the `^` symbol. In Turbo Pascal, the `^` symbol is an operator, which is never treated identically with the `^` symbol.
  
- o In ANSI Pascal, a comment can begin with `{` and end with `*`), or begin with `(*` and end with `}`. In Turbo Pascal, comments must begin and end with the same set of symbols.
  
- o In ANSI Pascal, each possible value of the tag type in a variant part must appear once. In Turbo Pascal, this requirement is not enforced.
  
- o In ANSI Pascal, the component type of a file type cannot be a structured type having a component of a file type. In Turbo Pascal, this requirement is not enforced.
  
- o In ANSI Pascal, a file variable has an associated buffer variable, which is referenced by writing the `^` symbol after the file variable. In Turbo Pascal, a file variable does not have an associated buffer variable, and writing the `^` symbol after a file variable is an error.
  
- o In ANSI Pascal, the statement part of a function must contain at least one assignment to the function identifier. In Turbo Pascal, this requirement is not enforced.
  
- o In ANSI Pascal, a field that is the selector of a variant part cannot be an actual variable parameter. In Turbo Pascal, this requirement is not enforced.
  
- o In ANSI Pascal, a component of a variable of a packed type cannot be an actual variable parameter. In Turbo Pascal, this requirement is not enforced.
  
- o In ANSI Pascal, a procedural or functional parameter is declared by writing a procedure or function heading in the

formal parameter list. In Turbo Pascal, the declaration of a procedural or functional parameter is achieved through a procedural or functional type, and uses the same syntax as the declaration of other types of parameters.

- o In ANSI Pascal, the standard procedures Reset and Rewrite do not require pre-initialization of file variables. In Turbo Pascal, file variables must be assigned the name of an external file using the Assign procedure before they are passed to Reset or Rewrite.

- o ANSI Pascal defines the standard procedures Get and Put, which are used to read from and write to files. These procedures are not defined in Turbo Pascal.

- o In ANSI Pascal, the syntax `New(p,c1,...,cn)` creates a dynamic variable with a specific active variant. In Turbo Pascal, this syntax is not allowed.

- o In ANSI Pascal, the syntax `Dispose(q,k1,...,km)` removes a dynamic variable with a specific active variant. In Turbo Pascal, this syntax is not allowed.

- o ANSI Pascal defines the standard procedures Pack and Unpack, which are used to "pack" and "unpack" packed variables. These procedures are not defined in Turbo Pascal.

- o In ANSI Pascal, the term  $i \bmod j$  always computes a positive value, and it is an error if  $j$  is zero or negative. In Turbo Pascal,  $i \bmod j$  is computed as  $i - (i \operatorname{div} j) * j$ , and it is not an error if  $j$  is negative.

- o In ANSI Pascal, a goto statement within a block can refer to a label in an enclosing block. In Turbo Pascal, this is an error.

- o In ANSI Pascal, it is an error if the value of the selector in a case statement is not equal to any of the case constants. In Turbo Pascal, this is not an error; instead, the case statement is ignored unless it contains an else clause.

- o In ANSI Pascal, statements that threaten the control variable of a for statement are not allowed. In Turbo Pascal, this requirement is not enforced.

- o In ANSI Pascal, a Read from a text file with a Char-type variable assigns a blank to the variable if Eoln was True before the Read. In Turbo Pascal, a carriage return character (ASCII 13) is assigned to the variable in this situation.
- o In ANSI Pascal, a Read from a text file with an integer-type or real-type variable ceases as soon as the next character in the file is not part of a signed integer or a signed number. In Turbo Pascal, reading ceases when the next character in the file is a blank or a control character (including the end-of-line marker).
- o In ANSI Pascal, a Write to a text file with a packed string-type value causes the string to be truncated if the specified field width is less than the length of the string. In Turbo Pascal, the string is always written in full, even if it is longer than the specified field width.
- o ANSI Pascal defines the standard procedure Page, which causes all subsequent output to a specific text file to be written on a new page. This procedure is not defined in Turbo Pascal. However, the typical equivalent of Page(F) is Write(F,Chr(12)).

```
=====
2. Extensions to ANSI Pascal
=====
```

Turbo Pascal is unable to detect whether a program uses any of the extensions listed here.

The following Turbo Pascal features are extensions to Pascal as specified by ANSI/IEEE770X3.97-1983.

- o The following are reserved words in Turbo Pascal:

absolute	inline	shr
assembler	interface	string
constructor	interrupt	unit
destructor	object	virtual
external	private	uses
far	near	xor
implementation	shl	

- o An identifier can contain underscore characters (\_).
- o Integer constants can be written in hexadecimal notation; such constants are prefixed by a \$.
- o Identifiers can serve as labels.
- o String constants are compatible with the Turbo Pascal string types, and can contain control characters and other nonprintable characters.
- o Label, constant, type, variable, procedure, and function declarations can occur any number of times in any order in a block.
- o Wherever the syntax of ANSI Pascal requires a simple constant, Turbo Pascal allows the use of a constant expression (also known as a computed constant).
- o Turbo Pascal implements the additional integer types Shortint, Longint, Byte, and Word, and the additional real types Single, Double, Extended, and Comp.
- o Turbo Pascal implements string types, which differ from the packed string types defined by ANSI Pascal in that they include a dynamic-length attribute that can vary during execution.
- o Turbo Pascal implements procedural and functional types. In addition to procedural and functional parameters, these types make possible the declaration and use of procedural and functional variables.
- o The type compatibility rules are extended to make Char types and packed string types compatible with string types.
- o Variables can be declared at absolute memory addresses using an absolute clause.
- o A variable reference can contain a call to a pointer-type function, the result of which is then dereferenced to denote a dynamic variable.
- o String-type variables can be indexed as arrays to access



individual characters in a string.

- o The type of a variable reference can be changed to another type through a variable typecast.

- o Turbo Pascal implements typed constants, which can be used to declare initialized variables of all types except file types.

- o Turbo Pascal implements three new logical operators: xor, shl, and shr.

- o The not, and, or, and xor operators can be used with integer-type operands to perform bitwise logical operations.

- o The + operator can be used to concatenate strings.

- o The relational operators can be used to compare strings.

- o Turbo Pascal implements the `@` operator, which is used to obtain the address of a variable or a procedure or function.

- o The type of an expression can be changed to another type through a value typecast.

- o The case statement allows constant ranges in case label lists, and provides an optional else part.

- o Procedures and functions can be declared with external, inline, and interrupt directives to support assembly language subroutines, inline machine code, and interrupt procedures.

- o A variable parameter can be untyped (typeless), in which case any variable reference can serve as the actual parameter.

- o Turbo Pascal implements units to facilitate modular programming and separate compilation.

- o Turbo Pascal implements the following file-handling procedures and functions, which are not available in ANSI Pascal:

Append	FilePos	Rename
BlockRead	FileSize	Rmdir
BlockWrite	Flush	Seek
ChDir	GetDir	SeekEof
Close	Mkdir	SeekEoln
Erase		

o String-type values can be input and output with the Read, Readln, Write, and Writeln standard procedures.

o Turbo Pascal implements the following standard procedures and functions, which are not found in ANSI Pascal:

Addr	Inc	Ptr
Concat	Insert	Random
Copy	Int	Randomize
CSeg	Length	Release
DSeg	Lo	RunError
Dec	Mark	SPtr
Delete	MaxAvail	Seg
Exit	MemAvail	SizeOf
FillChar	Move	SSeg
Frac	Ofs	Str
FreeMem	ParamCount	Swap
GetMem	ParamStr	TypeOf
Halt	Pi	UpCase
Hi	Pos	Val

o Turbo Pascal implements further standard constants, types, variables, procedures, and functions through standard units.

=====  
3. Implementation-dependent features  
=====

The effect of using an implementation-dependent feature of Pascal, as defined by ANSI/IEEE770X3.97-1983, is unspecified. Programs should not depend on any specific path being taken in cases where an implementation-dependent feature is being used. Implementation-dependent features include:

o the order of evaluation of index expressions in a variable reference

- o the order of evaluation of expressions in a set constructor
- o the order of evaluation of operands of a binary operator
- o the order of evaluation of actual parameters in a function call
- o the order of evaluation of the left and right sides of an assignment
- o the order of evaluation of actual parameters in a procedure statement
- o the effect of reading a text file to which the procedure Page was applied during its creation
- o the binding of variables denoted by the program parameters to entities external to the program

=====  
 4. Treatment of errors  
 =====

This section lists those errors from Appendix D of the ANSI Pascal Standard that are not automatically detected by Turbo Pascal. The numbers referred to here are the numbers used in the ANSI Pascal Standard. Errors 6, 19-22, and 25-31 are not detected because they are not applicable to Turbo Pascal.

2. If t is a tag field in a variant part and f is a field within the active variant of that variant part, it is an error to alter the value of t while a reference to f exists. This error is not detected.
3. If p is a pointer variable, it is an error to reference p<sup>^</sup> if p is nil. This error is not detected.
4. If p is a pointer variable, it is an error to reference p<sup>^</sup> if p is undefined. This error is not detected.
5. If p is a pointer variable, it is an error to alter the value of p while a reference to p<sup>^</sup> exists. This error is not

detected.

42. The function call `Eoln(f)` is an error if `Eof(f)` is `True`. In Turbo Pascal this is not an error, and `Eoln(f)` is `True` when `Eof(f)` is `True`.

43. It is an error to reference a variable in an expression if the value of that variable is undefined. This error is not detected.

46. A term of the form `i mod j` is an error if `j` is zero or negative. In Turbo Pascal, it is not an error if `j` is negative.

48. It is an error if a function does not assign a result value to the function identifier. This error is not detected.

51. It is an error if the value of the selector in a case statement is not equal to any of the case constants. In Turbo Pascal, this is not an error; instead, the case statement is ignored unless it contains an else clause.

Reference:

3/30/99 1:04:56 PM

Last Modified: 01-SEP-99

## Article #15319: INCLUDE FILE NOT FOUND BY TURBO DEBUGGER

Technical Notes Database

TN319D.txt INCLUDE FILE NOT FOUND BY TURBO DEBUGGER

Category :Pascal

Platform :All

Product :Turbo Pascal 5.0

Description:

Q. TP 5.0 does not generate the correct debug information so that a \$INCLUDE file in a user defined unit can be found by Turbo Debugger.

A. Send Turbo Pascal Tech Info 456.

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15325: Overview of Borland and Turbo Pascal 7.0

Technical Information Database

TI325D.txt - Overview of Borland and Turbo Pascal 7.0

Category :Turbo Pascal

Platform :All

Product :

Description:

Borland Pascal with Objects 7.0  
Turbo Pascal 7.0  
Features and Benefits

Overview

Here are the primary features and benefits of Borland Pascal with Objects 7.0 and Turbo Pascal with Objects 7.0. These are taken from the Reviewer's guide which includes more detailed product descriptions and technical information.

Borland Pascal with Objects 7.0

Borland Pascal with Objects 7.0 is the most productive programming system available for DOS and Windows. It includes the following new features:

Program for DOS, DOS Protected Mode (DPMI) and Windows  
Professional tools for all three platforms with nothing else to buy

Free DOS Extender! Create DOS Protected Mode (DPMI) applications. Break the 640K barrier and use up to 16 MB of memory, without royalties

Create DOS protected mode Dynamic Link Libraries (DLLs)  
Reuse DLLs in DOS and Windows; and link C or C++ code easily

High capacity, multi-target IDEs

Choose the DOS or Windows IDE and target any platform

DOS and Windows ObjectBrowsers

Navigate source code quickly with full cross-reference information

Improved IDEs with color syntax highlighting, unlimited undo/redo. Makes it easier to read code and experiment with changes

Enhanced application frameworks

Add data validation to DOS and Windows applications easily

New language features and optimizations

Write more flexible code with better performance

Winsight and Winspector included

For faster debugging of Windows applications

Free Runtime library source code included (\$195 value included free!). Learn the secrets of object-oriented programming from the experts

Turbo Pascal 7.0

Turbo Pascal 7.0 is the fastest way to learn object-oriented programming. It includes the following new features:

High capacity IDE

Compile large real-mode applications within the IDE

ObjectBrowser

Navigate source code quickly with full cross-reference

Improved IDE with color syntax highlighting, unlimited undo/redo Makes it easier to read code and experiment with changes

Enhanced application frameworks

Add data validation to DOS and Windows applications easily

New tutorials on Turbo Vision and pointers

Makes it easy to learn object-oriented programming

Reference:

3/30/99 12:57:52 PM

Last Modified: 01-SEP-99



## Article #15327: NESTED FOR LOOPS- How many?

Technical Notes Database

TN327D.txt      NESTED FOR LOOPS- How many?

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 4.0+

Description:

Q. How many nested for loops are allowed?

A. The amount of code that's generated can't exceed 20K bytes.

Therefore, if you have just for loops, the depth can't exceed roughly 75 lines.

Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15333: Questions and Answers of Borland and Turbo Pascal

Technical Information Database

TI333D.txt - Questions and Answers of Borland and Turbo Pascal

Category : Turbo Pascal

Platform : All

Product :

Description:

### Borland Pascal with Objects 7.0 Questions & Answers

#### I. General Questions

1. What are the new features of Borland Pascal with Objects 7.0?

Borland Pascal with Objects 7.0 includes all of the features necessary for high productivity application development for DOS and Windows.

The major new features are:

- o Support for DOS, DOS Protected Mode (DPMI) and Windows programming
- o Ability to create DOS Dynamic Link Libraries (DLLs) that are binary compatible with Windows
- o High capacity DOS and Windows IDEs
- o Improved IDEs with color syntax highlighting and undo/redo
- o DOS and Windows ObjectBrowsers for navigating source code
- o New compiler optimizations and language features
- o Extended DOS and Windows application frameworks
- o Over 3,800 pages of documentation with many new tutorials
- o Over 4 megabytes of on-line help including the Windows API

## 2. What are the new features of Turbo Pascal 7.0?

Turbo Pascal 7.0 includes features that make it the best way to learn object-oriented programming. The major new features of Turbo Pascal 7.0 are:

- o High capacity DOS Integrated Development Environment (IDE)
- o Improved IDE with color syntax highlighting and undo/redo
- o ObjectBrowser for navigating source code
- o Additional Turbo Vision objects
- o 1,500 pages of documentation with several new tutorials
- o 1 megabyte of on-line help with copy-and-paste examples

## 3. What's the difference between Borland Pascal with Objects 7.0 and Turbo Pascal 7.0?

Borland Pascal with Objects 7.0 includes features aimed at professional DOS and Windows programmers including the ability to create DOS Protected Mode (DPMI) applications, DOS Dynamic Link Libraries (DLLs), as well as a more complete set of professional tools including Turbo Debugger, Turbo Profiler, Turbo Assembler.

Turbo Pascal 7.0 is a subset of Borland Pascal with Objects and includes the ability to create DOS real mode programs. It includes the high capacity IDE, ObjectBrowser and extended Turbo Vision application framework, but does not support DOS Protected Mode (DPMI) or Windows development.

## 4. Who uses Pascal?

Borland has shipped over 2,000,000 units of Pascal. The largest segment of Pascal customers is professional programmers, who make up 40% of the market. Pascal is widely used for internal corporate application development, by consultants and VARs in vertical markets such as health-care and construction and by commercial "shrink wrap" software developers. Borland Pascal with Objects 7.0 is targeted directly to professional programmers who want

high productivity development for DOS and Windows.

Pascal is also the most popular language for learning structured and object-oriented programming. Turbo Pascal 7.0 is targeted at new programmers who want the fastest way to learn object-oriented programming.

5. Are there commercial applications written in Pascal?

There are many commercial "shrink wrap" software products written in Pascal. These products include leading CAD packages such as Autodesk's Generic Cadd, Fast CAD, Turbo CAD, Claris' Filemaker Pro for Windows, leading tax preparation software such as Tax Cut, Turbo Tax, Parson's Personal Tax Edge, financial packages such as Money Counts, communications packages such as QModem, Tapcis, Ozcis and many others.

6. Does Borland use Pascal to develop any of its products?

Borland has a major commercial application under development using the new Borland Pascal with Objects 7.0. Borland has not yet announced the name of the application being developed in Pascal.

In addition, the DOS IDE for Borland Pascal with Objects 7.0 is written entirely in Pascal using the Turbo Vision application framework.

7. Why has the product name changed?

We've called the new professional Pascal programming system "Borland Pascal with Objects" to reflect the professional positioning of the product and provide greater consistency with the C++ product line. Borland continues to offer Turbo Pascal 7.0 and Turbo Pascal for Windows 1.5 for entry level programmers who do not need all of the advanced capabilities that professionals require.

8. What are the benefits of creating DPMI applications?

By using Borland Pascal with Objects 7.0 you can create DOS Protected Mode (DPMI) applications that break through the 640K barrier and access up to 16 megabytes of memory for code and data. This gives you higher capacity for dealing with larger amounts of data for more efficient processing.

9. How do you create DPMI applications?

You can simply select the Compile Target menu choice to DOS protected mode and recompile existing programs. The compiler will create an EXE with a real-mode stub that automatically loads the DPMI server and runtime manager and then switches to protected mode and runs your program.

10. What DPMI technology do you use? Is there a royalty charge?

The DOS extender is fully compatible with the DPMI specification and is based entirely on Borland technology and was used in the latest version of Borland's Paradox 4.0 database. No royalty payments or runtime charges are required to distribute protected-mode applications written with Borland Pascal with Objects 7.0.

11. What is the benefit of DOS DLLs? How do you use them?

By creating programs as Dynamic Link Libraries (DLLs) you are able to share the same source code and binary file between both DOS and Windows programs. For example, a typical business application might be constructed with a DOS Turbo Vision user interface and a Dynamic Link Library that does "back end" computation and data processing. The back end DLL can be shared with a Windows version of the program that uses an ObjectWindows user interface. The source code to the DLL does not require any changes to work in Windows and, in fact, does not even need to be recompiled to work in Windows.

In the same way, you can compile C or C++ code as a Dynamic Link Library using Borland C++ or any other compiler and link that code into Windows or DOS applications written with Borland Pascal with Objects. No other language offers

the ability to create and link to DOS and Windows DLLs.

12. What are the new optimizations in Borland Pascal with Objects 7.0?

Several new compiler optimizations have been added so that existing programs can be recompiled for faster performance. These include 32 bit LongInt math operations, redundant pointer load suppression, optimized sets, constant parameters and faster string and file I/O routines.

13. Does Borland Pascal with Objects 7.0 include visual resource editing?

Yes, Borland Pascal with Objects 7.0 includes Borland's Resource Workshop, a fast, powerful way to visually create user interface elements including menus, dialogs, bitmaps, cursors, icons and so on. There are also third party visual resource editing tools available for use with Turbo Vision, including Blaise's Turbo Vision Development Toolkit.

14. Do developers need to buy the Microsoft Windows Software Development Kit (SDK) to use Borland Pascal with Objects 7.0?

No, Borland Pascal with Objects includes everything you need to develop DOS and Windows applications. It is fully compatible with Windows 3.0 and 3.1. It includes:

- o Windows Integrated Development Environment
- o Borland Pascal with Objects command line compiler
- o Windows 3.1 interface files
- o ObjectWindows application framework
- o Resource Workshop
- o WinSight
- o WinSpector
- o Resource Compiler
- o Help Compiler
- o Turbo Help hypertext system
- o Sample programs
- o Complete Windows tutorials

15. Does Borland Pascal with Objects 7.0 include help on new Windows 3.1 API functions?

The on-line Turbo Help system has been completely expanded to provide more detailed descriptions of Windows API functions as well as new entries for all new Windows 3.1 features. There are over 4 megabytes of on-line help. In addition, we are making available a complete printed reference 3 volume set of all Windows API calls for \$39.95. This three volume set includes Pascal definitions of all API calls and data structures.

16. Is the Runtime Library Source Code included?

As a special bonus to purchasers of Borland Pascal with Objects 7.0, we will include in all upgrades a retail purchase the Application Frameworks and Runtime Library Source Code. This includes the complete source code to the Turbo Vision and ObjectWindows application frameworks and is a \$195 value.

17. Is there a CASE tool for Pascal? What is Protogen?

Since there are many professional programmers who are new to Windows development, Borland will be making available a version of the Protogen CASE tool that generates Pascal source code. Protogen allows developers to draw their user interface in Windows and then automatically generate much of the standard Windows code that is required. This makes it much easier to get started in Windows and object-oriented programming. Protogen will be made available for the special price of \$49.95 to purchasers of Borland Pascal with Objects 7.0.

18. Does Borland Pascal with Objects work with the new Paradox Engine?

Yes Borland Pascal with Objects works with the latest version of the Paradox Engine 3.0. Paradox Engine 3.0 includes support for real mode, DOS protected mode (DPMI) and Windows application development in Pascal. It also

includes a complete object-oriented database framework for Pascal programmers as well as faster and more memory efficient performance. Paradox Engine 3.0 is available separately.

19. What is the minimum hardware configurations for Borland Pascal with Objects 7.0 and Turbo Pascal 7.0?

Borland Pascal with Objects 7.0 requires an 80286 or higher processor, 2 megabytes of memory, a hard disk. For Windows development you must also have Windows 3.0 or later, graphics (EGA, Hercules, VGA or higher resolution) and a mouse.

Turbo Pascal 7.0 requires 720K of disk space and 512K of memory. A hard disk and 80286 or higher processor is required to use the high-capacity IDE.

20. How do the new products affect the existing product line?

Borland Pascal with Objects 7.0 and Turbo Pascal 7.0 replace the existing Turbo Pascal Professional 6.0 and Turbo Pascal 6.0 products. Turbo Pascal for Windows 1.5 remains on the market as an entry level object-oriented programming system for Windows.

21. What is the pricing and availability?

Borland Pascal with Objects 7.0 has a suggested retail price of \$495.00. Upgrades from any previous version of Turbo Pascal for DOS or Windows to Borland Pascal with Objects 7.0 are \$149.95. As a special bonus, Borland Pascal with Objects 7.0 includes the Application Frameworks and Runtime Library source code, a \$195 value.

Turbo Pascal 7.0 has a suggested retail price of \$149.95. Upgrades from any previous version of Turbo Pascal for DOS or Windows to Turbo Pascal 7.0 is \$89.95.

Both products will ship November 2, 1992.



Reference :

3/30/99 12:58:32 PM

Last Modified: 01-SEP-99

## Article #15335: RESERVED WORD 'PACKED'

Technical Notes Database

TN335D.txt      RESERVED WORD 'PACKED'  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 4.0+

### Description:

Q. Does the reserved word 'packed' have any effect on an array in Turbo Pascal?

A. No. There is no packing done because on an IBM PC, and compatibles, the data in the array is stored on a byte level.

### Reference:

7/16/98 4:35:30 PM

Last Modified: 01-SEP-99

## Article #15341: Creating Protected Mode Apps with Borland Pascal

Technical Information Database

TI341D.txt - Creating Protected Mode Apps with Borland Pascal

Category :Turbo Pascal

Platform :DOS

Product :

Description:

### o Benefits of Protected Mode Programming

Borland Pascal with Objects allows you to create DOS protected mode applications.

This means that your programs no longer have to be constricted by the long-standing 640 k memory limitation. This barrier has now disappeared entirely with our protected mode target that allows you to create programs that use up to 16meg of code and data.

You only need to throw a single switch inside the IDE (integrated development environment) in order to produce a protected mode application. In most circumstances there is no need to change your code at all.

Consider the following program, which reports on the available memory:

```
program MemSize;  
  
begin  
  WriteLn(MemAvail);  
end.
```

If compiled in real mode, this program will probably report that you have between 300,000 and 600,000 bytes of memory available. If you open the Compile menu, select Target, and switch from Real Mode to Protected Mode code generation, then this program will report a figure near the total amount of RAM available on your current system.

If you use the command line compiler, called BPC.EXE, then it is only necessary to add "/cp" to the command line in order to

compile a protected mode app. For instance, to compile the program listed above in real mode, one would type:

```
bpc MemSize
```

To recompile this same application in protected mode, one would type:

```
bpc/cp MemSize
```

As an unofficial test case, these two programs were run on a system with 12 megs of memory available. In the first case MemSize reported 574,144 bytes of memory available, but when compiled in protected mode, it reported 9,118,992 bytes of memory available. This is a difference of over 8 million bytes.

#### o Differences between Real and Protected Mode programming

Although most Pascal programs written under TP60 will recompile unchanged as protected mode apps, developers should be aware of the way memory is addressed in protected mode. Real mode DOS addresses typically include a segment and offset which reference an actual location in memory. But under Intel's protected mode memory scheme, the segment portion of an address is replaced with something called a selector.

Selectors are not real memory addresses but rather indexes into a descriptor table. This table is maintained by the operating system and cannot normally be accessed directly by a programmer. This means that programs which directly reference real mode memory addresses will have to be adjusted before they will run correctly in protected mode.

Fortunately, Intel has provided programmers with a DOS Protected Mode Interface (DPMI) specification, which forms a gateway between protected mode and real mode. Borland's new protected mode DOS extender is DPMI compliant, so this interface is available to experienced programmers who want to address real mode memory directly. As a result, Borland Pascal protected mode apps have almost unlimited access to real mode.

Your protected mode applications will also run inside a Windows DOS box. This is possible because Windows itself runs in protected mode and is somewhat DPMI compliant.

There are no royalties involved in using this new DOS extender technology, but BP.EXE, the new Pascal IDE, requires at least a 286 machine with two megs of memory to run. If you can

meet these requirements then the world of protected mode programming is now available to you at the flip of a switch.

For more information as to the internals of the DOS extender technology, Borland has made available the Open Architecture Handbook which goes in to some detail about this new and exciting area of programming.

Reference:

3/30/99 3:10:14 PM

Last Modified: 01-SEP-99

## Article #15343: CASE - IDENTIFIERS OTHER THAN SCALAR

Technical Notes Database

TN343D.txt      CASE - IDENTIFIERS OTHER THAN SCALAR  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 3.0+

### Description:

- Q. Can you use identifiers other than scalar in the case statement?
- A. Case statements allow the following ordinal types: Char, Boolean, Integer, and user-defined enumeration.

### Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15348: Installing Turbo Pascal 7.0 on a Network

Technical Information Database

TI348D.txt - Installing Turbo Pascal 7.0 on a Network

Category :Turbo Pascal  
Platform :All  
Product :

Description:

### Configuring Turbo Pascal on Network Systems

First, it must be clear to users that network requirements are not considered in the design or development of Turbo Pascal. The intended target system is an IBM PC, AT, PS/2 or 100% compatible with at least 448K of available DOS RAM, 5 megabytes hard disk space, and at least one floppy drive.

Through years of use, it has been found that Turbo Pascal happens to work satisfactorily on a number of network systems. The following guidelines may be used if installation on a network system is required.

- o Install Turbo Pascal from the original diskettes from a node with both Read and Write capability in the target directory of the server.
- o Ensure that all nodes requiring access to Turbo Pascal have at least Read rights in the directory where Turbo Pascal has been installed.
- o Have each node run Turbo Pascal at a directory where both Read and Write actions are allowed. Preferably, the current directory would reside on a local drive or at least a directory designated for use by only a single node. It is to this private disk space that Turbo Pascal's swap files and configuration files are to be saved.
- o Allow Turbo Pascal to create source and executable code

either on a local drive or at a directory designated for use by only a single node. If programs are to be created on a network, provide both Read and Write access to the appropriate areas of the network drive.

Instructions for granting access rights for a particular network system are found in documentation provided by its manufacturer. Check with the System Administrator or the manufacturer for specific information.

Before installing any software on a network, be certain that you are licensed for each node able run the software simultaneously with other nodes. If you are uncertain about any aspect of your licensing, contact your local Borland office immediately.

Reference:

3/30/99 12:59:09 PM

Last Modified: 01-SEP-99



## Article #15350: READLN - READING A STRING

Technical Notes Database

TN350D.txt      READLN - READING A STRING

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

- Q. If I read in a string, an integer, and a character, with one Readln, how come there's a carriage return in the character variable?
- A. When a string is read in there is no way to determine if the other values are strings or non-strings. This means that regardless of a string's length, the other values are placed into the string variable, although they may not be visible. The solution is to read the string separately with a Readln.

Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15355: Using the VESA16.BGI that Comes with BP 7.0

Technical Information Database

TI355D.txt - Using the VESA16.BGI that Comes with BP 7.0

Category :Turbo Pascal  
Platform :All  
Product :

Description:

VESA DRIVERS

With the release of Borland Pascal for Objects 7.0, programmers can now use BGI graphics to obtain 16 color resolutions of 800 X 600, 1024 X 768, and 1280 X 1024.

HOW TO USE THE VESA16.BGI DRIVER

The VESA unit included with this Tech Info Sheet shows you a simple way to use the new VESA driver in your own program. To get started, you need only include the word VESA in your uses clause and then call the included Initialize procedure. So the simplest possible program using the VESA driver might look like this:

```
program Simple;  
uses  
    Graph,  
    Vesa;  
begin  
    Initialize('c:\bp\bgi');  
    OutText('Hello');  
    ReadLn;  
    CloseGraph;  
end.
```

This program assumes, of course, that you have the VESA unit available on disk and that your BGI drivers are kept in the C:\BP\BGI sub directory. If the system this program is run on

does not support the VESA BIOS then the code in the VESA unit will automatically switch your computer into the highest available mode.

The next few paragraphs describe a somewhat more complex program, also included in this TI, which shows you how to use the new high resolution modes. This program is called VesaSamp.Pas. It uses the VESA unit, also listed below, to load the BGI VESA graphics driver into memory. The program then shows you the highest available resolution on your system.

In order to simplify the process of entering graphics mode, the VESA UNIT provides you with an easy to use Initialize procedure which takes the location of the BGI drivers as its sole parameter. This procedure will automatically switch you into graphics mode, and also initialize several useful variables.

As presented below, program VesaSamp.Pas hardcodes the location of the BGI drivers as being in the C:\BP\BGI sub directory. If this is not the case on your system, then you need to change this code before running the program. For instance, if you store the BGI drivers in the D:\PASCAL\DRIVERS sub directory, then you should change the call that reads:

```
Initialize('c:\bp\bgi');
```

so that it reads:

```
Initialize('d:\pascal\drivers');
```

If you have the BGI drivers available in the current sub directory, then you would initialize the graphics system with the following code:

```
Initialize('');
```

After correctly setting up the Initialize procedure, the VesaSamp program will pop into graphics mode and show you the highest available resolution.

## WHAT IS THE VESA BIOS?

The VESA BIOS represents a standard for accessing SUPER VGA resolutions. The problem it addresses is that there are many different Super VGA boards on the market, but no common API for addressing them.

To attempt to bring some sort of sanity to this madness, the Video Electronics Standards Association (VESA) defined a set of

BIOS extensions that allow programmers to ask the video adapter about its abilities. These BIOS extensions are sometimes implemented directly in ROM and sometimes loaded into memory by a special TSR. Many of these TSRs are available on the Comuserve Information Service (1-800-848-8990).

Since it is always possible that the video card on any particular system might not support the VESA extensions, you should be aware that your program might not actually run in a high resolution mode. But the Borland Graphics Interface will find the highest available mode and then switch you into it.

The rest of this TI contains the sample code which shows you how to use the VESA BGI driver. You should be aware of the fact that it is necessary for the VESA .PAS unit to automatically detect whether or not you have compiled your program as a protected mode application. In other words, if you want to copy code from the VESA unit directly into your program, make sure you are getting the DOS code for DOS apps and the DPMI code for DPMI apps.

```
program VesaSamp;
{
  A test program for unit Vesa
  which provides Super VGA Resolutions
}
uses
  Graph,
  Vesa;

{ Converts an integer to a string }
function Int2Str(L : LongInt) : string;
var
  S : string;
begin
  Str(L, S);
  Int2Str := S;
end; { Int2Str }

var
  S: String;
begin
  Initialize('c:\bp\bgi');
  Rectangle(0,0,MaxX,MaxY);
  SetColor(LightGreen);
  SetBkColor(Blue);
  SetTextStyle(TriplexFont, HorizDir, 8);
```

```

S := 'MaxX = ' + Int2Str(MaxX);
OutTextXY((MaxX Div 2) - (TextWidth(S) div 2),
          (MaxY div 2) - 220, S);
S := 'MaxY = ' + Int2Str(MaxY);
OutTextXY((MaxX Div 2) - (TextWidth(S) div 2), (MaxY div 2) -
80, S);
S := 'MemAvail = ' + Int2Str(MemAvail);
OutTextXY((MaxX Div 2) - (TextWidth(S) div 2), (MaxY div 2) +
60, S);
ReadLn;
CloseGraph;
end.

```

```

{*****}
{***** VESA UNIT *****}
{*****}

```

```

unit Vesa;
{

```

Add this unit to programs that want Vesa support for high resolutions such as 800 X 600, 1024 X 768 and 1280 X 1024. If Vesa is not supported on a system the code automatically defaults to an available driver such as EGAVGA.

To Initialize the graphics system, just call Initialize with the path to VESA.BGI as the sole parameter. Pass an empty string if the driver is in the current subdirectory.

The code will automatically detect if you are running in protected mode or real mode.

Examples:

```

Initialize('c:\bp\bin');
Initialize('');

```

```

}
Interface

```

```

uses
  Graph {$IfDef DPMS}, WinApi {$EndIf};

```

```

const
  VESA16Modes: array[0..2] of Word =
    ($0102, $0104, $0106);

```

```

type

```

```
VgaInfoBlock = record
  VESASignature: array[0..3] of Byte;
  VESAVersion: Word;
  OEMStringPtr: Pointer;
  Capabilities: array[0..3] of Byte;
  VideoModePtr: Pointer;
end;
```

```
var
  MaxColor,
  MaxX, MaxY: Integer;
```

```
procedure Initialize(PathToDriver: String);
implementation
```

```
var
  VESA16      : Integer; { Driver number of 16 color driver }
```

```
function GetHighestCap(Table: Pointer; Modes: Word; Size:
Integer): Integer;
  near; assembler;
```

```
asm
```

```
    XOR     AX,AX
    LES     DI, Table
```

```
1:
```

```
    MOV     SI, Modes
    ADD     SI, Size
    ADD     SI, Size
    SUB     SI, 2
    MOV     BX, ES:[DI]
    CMP     BX, 0FFFFH
    JE
```

```
4
```

```
    INC     DI
    INC     DI
    MOV     CX,Size
```

```
2:
```

```
    CMP     BX,[SI]
    JZ
```

```
3
```

```
    DEC     SI
    DEC     SI
    LOOP
```

```

2
3:
    CMP     AX,CX
    JA
1
    MOV     AX,CX
    JMP
1
4:
    DEC     AX
end;

```

```

{$IFDEF DPMI}
type
  TRealRegs = record
    RealeDI: Longint;
    RealeSI: Longint;
    RealeBP: Longint;
    Reserved: Longint;
    RealeBX: Longint;
    RealeDX: Longint;
    RealeCX: Longint;
    RealeAX: Longint;
    RealFlags: Word;
    RealeS: Word;
    RealDS: Word;
    RealFS: Word;
    RealGS: Word;
    RealIP: Word;
    RealCS: Word;
    RealSP: Word;
    RealSS: Word;
  end;

```

```

function DetectVesa16: Integer; far; assembler;
var
  Segment, Selector, VesaCap: Word;
asm
{$IFOPT G+}
    PUSH     0000H
    PUSH     0100H
{$ELSE}
    XOR     AX,AX

```

```
PUSH    AX
INC     AH
PUSH    AX
```

```
{ $ENDIF }
```

```
CALL    GlobalDosAlloc
MOV     Segment,DX
MOV     Selector,AX
MOV     DI,OFFSET RealModeRegs
MOV     WORD PTR [DI].TRealRegs.RealSP, 0
MOV     WORD PTR [DI].TRealRegs.RealSS, 0
MOV     WORD PTR [DI].TRealRegs.RealeAX, 4F00H
MOV     WORD PTR [DI].TRealRegs.RealeES, DX
MOV     WORD PTR [DI].TRealRegs.RealeEDI, 0
MOV     AX,DS
MOV     ES,AX
MOV     AX,0300H
MOV     BX,0010H
XOR     CX,CX
INT     31H
MOV     DI,OFFSET RealModeRegs
MOV     AX,grError
PUSH    AX
CMP     WORD PTR [DI].TRealRegs.RealeAX,004FH
JNZ
```

```
Exit
```

```
POP     AX
MOV     ES,Selector
XOR     DI,DI
CMP     ES:[DI].VgaInfoBlock.VESASignature.Word[0], 'EV'
JNZ
```

```
Exit
```

```
CMP     ES:[DI].VgaInfoBlock.VESASignature.Word[2], 'AS'
JNZ
```

```
Exit
```

```
MOV     AX,0000
MOV     CX,1
INT     31H
MOV     VesaCap,AX
MOV     DX,ES:[DI].VgaInfoBlock.VideoModePtr.Word[2]
MOV     CX,4
XOR     AX,AX
```

```
Convert:
```

```
SHL     DX,1
RCL     AX,1
```



LOOP

Convert

```
ADD    DX,ES:[DI].VgaInfoBlock.VideoModePtr.Word[0]
ADC    AX,0
MOV    CX,AX
MOV    BX,VesaCap
MOV    AX,0007H
INT    31H
INC    AX
XOR    CX,CX
MOV    DX,0FFFFH
INT    31H
MOV    ES,BX
PUSH   ES
PUSH   DI
```

{ \$IFOPT G+ }

```
PUSH   OFFSET Vesa16Modes
PUSH   0003H
```

{ \$ELSE }

```
MOV    SI, OFFSET Vesa16Modes
PUSH   SI
MOV    AX, 5
PUSH   AX
```

{ \$ENDIF }

```
CALL   GetHighestCap
PUSH   AX
MOV    BX,VesaCap
MOV    AX,0001H
INT    31H
```

Exit:

```
PUSH   Selector
CALL   GlobalDosFree
POP    AX
```

end;

{ \$ELSE }

function DetectVesa16: Integer; far; assembler;

var

VesaInfo: array[0..255] of Byte;

asm

```
MOV    AX,SS
MOV    ES,AX
LEA    DI,VesaInfo
MOV    AX,4F00H
INT    10H
```

```
CMP    AX,004FH
MOV    AX,grError
JNZ
```

Exit

```
CMP    ES:[DI].VgaInfoBlock.VESASignature.Word[0], 'EV'
JNZ
```

Exit

```
CMP    ES:[DI].VgaInfoBlock.VESASignature.Word[2], 'AS'
JNZ
```

Exit

```
LES    DI,ES:[DI].VgaInfoBlock.VideoModePtr
PUSH   ES
PUSH   DI
MOV    AX, OFFSET Vesal6Modes
PUSH   AX
MOV    AX,3
PUSH   AX
CALL   GetHighestCap
```

Exit:

end;

{ \$ENDIF }

```
procedure Initialize(PathToDriver: String);
```

```
var
```

```
    MaxColor,
    ErrorCode,
    GraphMode,
    GraphDriver: Integer;
```

```
begin
```

```
    VESA16 := InstallUserDriver('VESA16',
DetectVESA16);
```

```
    GraphDriver := Detect;
```

```
    InitGraph(GraphDriver, GraphMode, PathToDriver);
```

```
    ErrorCode := GraphResult;
```

```
    if ErrorCode <> grOK then begin
```

```
        WriteLn('Graphics error: ', GraphErrorMsg(ErrorCode));
```

```
        ReadLn;
```

```
        Halt;
```

```
    end;
```

```
    MaxX := GetMaxX;
```

```
    MaxY := GetMaxY;
```

```
    MaxColor := GetMaxColor;
```

```
end;
```

```
end.
```

Reference:

3/30/99 2:58:27 PM

Last Modified: 01-SEP-99

## Article #15357: TASM 1.0 1.01 - ORDER OF ARG DIRECTIVE OPERANDS -

Technical Notes Database

TN357D.txt      TASM 1.0 1.01 - ORDER OF ARG DIRECTIVE OPERANDS -  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 5.0 5.5

### Description:

- Q. In what order should the operands to the Arg directive appear? Should they be the same as the order in the calling procedure or in reverse?
- A. The answer depends upon the language that you're using and whether or not you're using the .model memory model, language option.

Case 1: You are using the full segment definitions or you are using the .model directive without specifying a language.

The leftmost symbol in the Arg list is always nearest the top of the stack (TOS). Thus, if your proc definition is

```
myproc proc far
arg first:word, second:word
```

first will evaluate to [BP + 06H], the size of the four byte far return address plus the size of the two byte BP register pushed onto the stack. The symbol, second, will evaluate to [BP + 08H], two bytes further from TOS.

Again, the key point to remember is that the leftmost symbol is nearest TOS.

If you are using Pascal or Basic, arguments are pushed from left to right onto the stack. This is called Pascal calling syntax or Pascal calling conventions. In this case, the last argument in the list is nearest TOS. For example:

```
{ $F+ }
procedure myproc(one:integer; two:integer); external;
```

```
{ $F- }  
...  
procedure myproc(first:integer; second:integer);  
...
```

The variable, first, is pushed onto the stack followed by second. Thus, second is nearest TOS. In this case, you would want to set up your Arg list as follows:

```
myproc proc far  
arg second:word, first:word  
push bp  
mov bp,sp
```

The symbol, second, would evaluate to [BP + 06H], the symbol nearest TOS, and first would evaluate to [BP + 08H].

To get the correct results with Basic or Pascal while using the simplified segment directives or .model without the language option, you must reverse the order of the arguments as they appear in the high level language calling routine's parameter list.

If you are calling the Assembler routine from a C module and you are NOT using Pascal calling conventions, then the arguments are pushed onto the stack from right to left. Thus, the leftmost argument is nearest TOS and there is no need to reverse the order to the symbols in the Arg list.

If you wish to use Pascal calling conventions in Turbo C, use the -p option for TCC or the Pascal keyword. Pascal calling syntax can also be enabled from within TC by setting the /Options/Compiler/Code generation/Calling convention to Pascal.

Case 2: You are using the .model memory model, language directive.

All you need to do is put the Arg list symbols in the same order as they appear in the calling routine's parameter list. This is the easiest method, by far. Remember to include the comma after the memory model in the .model directive and to specify the language. If you are using Turbo Pascal, just use .model TPascal.

Reference :

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15363: Using Validators in OWL Applications

Technical Information Database

TI363D.txt Using Validators in OWL Applications

Category :General Programming

Platform :All

Product :Pascal All

### Description:

Starting with version 7.0, Borland Pascal introduced Validator objects for Turbo Vision and the Object Windows Library. These objects help guide the user when they are entering data in Edit fields. The purpose of this TI is to introduce the user to the idea of using Validators in OWL applications.

Validators are very flexible objects which can play several different roles in your program. A typical validator might ensure that the user can only enter numbers within a certain range into a particular field. Another common use for Validators might be to ensure that users enter stock numbers that begin with two alphanumeric characters, then include a dash followed by four numbers. An example of this type of stock number might look like this: WD-3344. These are only two examples of the many different types of validators you can create for your programs.

Validators are extremely easy to use. In most cases you only need to add one line to your code. Suppose you want to add a validator to an edit field which will ensure that the user can only enter numbers between five and eight. First you would initialize your edit field, just as you have always done:

```
AEdit := New(PEdit, InitResource(
Self, Cm_NumEdit, 50));
```

The next step is to initialize the validator. The following line of code turns the trick:

```
AEdit^.SetValidator(New(PRangeValidator, Init(5, 8)));
```

That's all you need to do. Now the only characters the user can enter into the control will have to fall within the specified range.

When compiled, the program and resource script included in this Tech Info Sheet help demonstrate many of the primary features of the validator unit. Included in the example are two range validators, four picture validators and one custom validator. The range validators ensure that input falls inside a particular numeric range. One of the picture validators helps guide the user while he enters a date, and another ensures that the user follows a valid format while entering a social security number. The other validators guide the user while he enters a stock number and a special entry that must begin with the letters 'MIKE' followed by a single number. In order to show you how to create your own Validators, we include a descendant of TPxPictureValidator called TSamValidator which gives you an idea of how to control input on a character by character basis.

Validators usually work on a letter by letter or field by field basis. That is, they will check a user's input either as he enters each character, or later on, when he exits a field. If you want to check the data only when the user closes an entire screenful of data, then you should examine the TMyEdit object, presented below. In this object, we override the TEdit methods that would normally ensure that a user's input was checked as he leaves a field. As a result, the data is checked only when the user closes the dialog in which the edit fields reside.

```
{*****}
{
{ Turbo Pascal 7.0 for Windows }
{ Demo program }
{ Copyright (c) 1992 by Borland International }
{ }
{*****}
```

```
program ValidExp;
```

```
uses WinTypes, WinProcs, OWindows, ODialogs, Strings, Validate,
BWCC;
```

```
{ $R ValidExp }
```

```
const
```



```
Cm_Dialog      = 101;
Cm_NumEdit    = 102;
Cm_NumEdit2   = 103;
Cm_DateEdit   = 104;
Cm_StockEdit  = 105;
Cm_SSNEdit    = 106;
Cm_SamEdit    = 107;
Cm_MikeEdit   = 108;
```

```
type
```

```
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;
```

```
type
```

```
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    constructor Init(AParent: PWindowsObject; AName: PChar);
    procedure cmDialog(var Msg: TMessage);
      virtual cm_First + cm_Dialog;
  end;
```

```
  PMyEdit = ^TMyEdit;
  TMyEdit = Object(TEdit)
    procedure WMKillFocus(var Msg: TMessage);
      virtual wm_First + wm_KillFocus;
    procedure WmGetDlgCode(var Msg: TMessage);
      virtual wm_First + wm_GetDlgCode;
  end;
```

```
  PMyDialog = ^TMyDialog;
  TMyDialog = Object(TDialog)
    constructor Init(AParent: PWindowsObject; AName: PChar);
  end;
```

```
  PSamValidator = ^TSamValidator;
  TSamValidator = object(TPxPictureValidator)
    function IsValid(const S: string): Boolean; virtual;
    procedure Error; virtual;
  end;
```

```
procedure TMyEdit.WMKillFocus(var Msg: TMessage);
begin
  DefWndProc(Msg)
end;
```

```
procedure TMyEdit.WMGetDlgCode(var Msg: TMessage);
begin
    DefWndProc(Msg);
end;
```

```
{-----}
{ TMyWindow's method implementations: }
{-----}
```

```
constructor TMyWindow.Init(AParent: PWindowsObject;
                           AName: PChar);
begin
    TWindow.Init(AParent, AName);
    Attr.Menu := LoadMenu(HInstance, 'Menu_1');
end;
```

```
procedure TMyWindow.cmDialog(var Msg: TMessage);
var
    D: PDialog;
begin
    D := New(PMyDialog, Init(
Self, 'Dialog_1'));
    Application^.ExecDialog(D);
end;
```

```
{-----}
{ TMyWindow's method implementations: }
{-----}
```

```
constructor TMyDialog.Init(AParent: PWindowsObject;
                           AName: PChar); var
    AEdit: PEdit;
    MyEdit: PMyEdit;
begin
    TDialog.Init(AParent, AName);
    AEdit := New(PEdit, InitResource(
Self, Cm_NumEdit, 50));
    AEdit^.SetValidator(New(PRangeValidator, Init(5, 8)));
    MyEdit := New(PMyEdit, InitResource(
Self, Cm_NumEdit2, 10));
    MyEdit^.SetValidator(New(PRangeValidator, Init(1, 10)));
    AEdit := New(PEdit, InitResource(
Self, Cm_DateEdit, 50));
```

```

AEdit^.SetValidator(New(PPXPictureValidator,
                        Init('{#[#]}/{#[#]}/{##[##]}' , True)));
AEdit := New(PEdit, InitResource(
Self, Cm_StockEdit, 50));
AEdit^.SetValidator(New(PPXPictureValidator,
                        Init('&##-####' , True)));
AEdit := New(PEdit, InitResource(
Self, Cm_SSNEdit, 50));
AEdit^.SetValidator(New(PPXPictureValidator,
                        Init('###-##-####' , True)));
AEdit := New(PEdit, InitResource(
Self, Cm_SamEdit, 50));
AEdit^.SetValidator(New(PSamValidator, Init('&&#', True)));
AEdit := New(PEdit, InitResource(
Self, Cm_MikeEdit, 50));
AEdit^.SetValidator(New(PPXPictureValidator,
                        Init('Mike#' , True)));

```

```
end;
```

```

{-----}
{ TSamValidator's method implementations:  }
{-----}

```

```
function IsNumber(Chr: Char): Boolean; near; assembler;
asm
```

```

    XOR    AL,AL
    MOV    Ch,Chr
    CMP    Ch,'0'
    JB

```

```

1
    CMP    Ch,'9'
    JA

```

```

1
    INC    AL

```

```

1:
end;
```

```

function TSamValidator.IsValid(const S: string): Boolean;
var
    S1,S2: array[0..5] of Char;
begin

```

```
StrPCopy(S2, S);
IsValid := False;
StrLCopy(S1, S2, 3);
if StrIComp(S1, 'Sam') = 0 then IsValid := True
else Exit;
if IsNumber(S2[4]) then IsValid := True;
end;
```

```
procedure TSamValidator.Error;
begin
  MessageBox(0,
    'Enter the word Sam followed by a number. Example: Sam1',
    nil, mb_IconExclamation + mb_OK);
end;
```

```
{-----}
{ TMyApplication's method implementations: }
{-----}
```

```
procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PMyWindow,
    Init(nil, 'Sample ObjectWindows Program'));
end;
```

```
{-----}
{ Main program: }
{-----}
```

```
var
  MyApp: TMyApplication;

begin
  MyApp.Init('MyProgram');
  MyApp.Run;
  MyApp.Done;
end.
```

```
MENU_1 MENU
BEGIN
  MENUITEM "E&xit", 24340
  MENUITEM "&Dialog", 101
END
```

```

DIALOG_1 DIALOG 37, 28, 196, 163
  STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
WS_SYSMENU CLASS "BorDlg"
CAPTION "Validator Dialog"
BEGIN
  CONTROL "", 102, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 124, 12, 59, 12
  CONTROL "", 103, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 123, 29, 59, 12
  CONTROL "", 104, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 123, 46, 59, 12
  CONTROL "", 105, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 123, 63, 59, 12
  CONTROL "", 106, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 123, 80, 59, 12
  CONTROL "", 107, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 123, 97, 59, 12
  CONTROL "", 108, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, 123, 114, 59, 12
  CONTROL "Button", 2, "BorBtn", 1 | WS_CHILD | WS_VISIBLE |
WS_TABSTOP, 53, 138, 32, 20
  CONTROL "Button", 1, "BorBtn", 1 | WS_CHILD | WS_VISIBLE |
WS_TABSTOP, 112, 138, 32, 20
  CONTROL "Valid entries are 1-10 (No Tab)", -1, "STATIC",
SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP, 19, 12, 101, 10
  CONTROL "Enter a date", -1, "STATIC", SS_LEFT | WS_CHILD |
WS_VISIBLE | WS_GROUP, 80, 46, 40, 10
  CONTROL "Enter Stock Number (AANN-NNNN)", -1, "STATIC",
SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP, 11, 63, 110, 10
  CONTROL "Enter Social Security Num", -1, "STATIC", SS_LEFT |
WS_CHILD | WS_VISIBLE | WS_GROUP, 34, 80, 86, 8
  CONTROL "Enter SAM followed by number", -1, "STATIC",
SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP, 20, 97, 100, 10
  CONTROL "Enter MIKE followed by number", -1, "STATIC",
SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP, 17, 114, 103, 10
  CONTROL "", 106, "BorShade", 32774 | WS_CHILD | WS_VISIBLE,
6, 7, 184, 125
  CONTROL "Valid entries are 5-8 (Tab Ok)", -1, "STATIC",
SS_LEFT | WS_CHILD | WS_VISIBLE | WS_GROUP, 19, 29, 96, 10
END

```

Reference:

7/16/98 4:33:42 PM

Last Modified: 01-SEP-99

## Article #15365: TASM 1.0 - RETURN FAILED FROM ASSEMBLY ROUTINE - T

Technical Notes Database

TN365D.txt     TASM 1.0 - RETURN FAILED FROM ASSEMBLY ROUTINE - T  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal 4.0+

### Description:

Q. When I call my Assembler routine from Turbo Pascal, everything works OK. However, it does not seem to return properly.

Why?

A. Any routine declared in the main program or in the implementation section of a unit, is by default a near call. Your Assembly language routine, assembled to an .OBJ file, is brought into the Pascal module as if it were declared and defined in that module. Thus, by default, any call to that Assembly language routine is a near call. If you have defined your Assembly language routine as a proc far, the Assembler will generate a corresponding RETF. Thus, a near call is made, 4 bytes are popped off of the stack when the RETF is executed, and control will be returned to the wrong code segment. If you want that call to be a far call in your Pascal module, be sure to bracket the function or procedure declaration with {\$F+} and {\$F-}. Always make sure that the distance declarations and definitions agree.

### Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15369: Printing the Run of a Program

Technical Information Database

TI369D.txt - Printing the Run of a Program

Category :Turbo Pascal  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Educators teaching structured programming in PASCAL often require students to submit a printout of the "Run" of assigned programs. The "Run" represents the output of the program as it normally appears on screen. In mainframe environments, printing the "Run" is nothing unusual. Time and access are often very limited in academic computer rooms and such a printout can be useful when correcting code off-line.

The Integrated Development Environment of Turbo Pascal has no provision for printing the "Run" of a program. On a personal computer, output may take a variety of forms; simple text, graphics, sound etc. Printing a "run" in the fashion of an academic mainframe is only relevant to code producing simple text as output. More substantial programs involving color and graphics must use other means to print the "run". Techniques dealing with graphics and color are not addressed in this document.

### Printing the Run Using the DOS Version of Turbo Pascal

In DOS, the operating system is capable of echoing to printer characters sent to the screen. This feature only functions from the DOS prompt and only for Turbo Pascal programs not using CRT or GRAPH units. The echoing function is toggled on and off with the key sequence .

1. Set the destination of the compiler to disk via the Compile|Destination menu selection.
2. Compile the program. Doing so will generate a .EXE file.
3. Exit Turbo Pascal.



4. Ensure your printer is on and on-line.
5. At a DOS prompt, hold down the key marked `Alt` and press the letter "P". Then let up on both keys.
6. Type the name of your program. Do not include the three letter extension. For instance, the program TEST.PAS is taken as just TEST. Screen and printer output will occur simultaneously.
7. When the program is done, hold down the key marked `Alt` and press the letter "L". Then let up on both keys. Next hold down `Alt` again and press the letter "P". Let up on both keys.

## Printing the Run Using the Windows Version of Turbo Pascal

Turbo Pascal for Windows is not ideally suited for introductory Computer Science coursework. Such coursework normally relies on some terminal-type interface being involved. Windows by its very nature suspends all terminal-like devices available when running DOS alone. It is designed to function exclusively in an event-driven, graphical environment. Display functions are provided by Windows and not by Turbo Pascal. Hence, standard PASCAL statements such as `Write` and `Writeln` do not function with the screen display as they do with the DOS version. For Computer Science coursework, it is strongly recommended that the DOS version of Turbo Pascal be used.

A unit called `WinCRT` ships with Turbo Pascal for Windows. This unit causes an output window to be displayed when the statements `Write`, `Writeln`, `Read`, or `Readln` are used. It mimics the function of a terminal device and serves as a temporary aid for those learning to use Windows API functions. This unit allows some traditional programming to be done but provides nothing to "print" a run.

If printing the output of a program in Windows is absolutely required, follow these steps.

1. Run the program.
2. Enlarge the output window to fill the entire screen.
3. Press the `Alt` key. The `Alt` key will copy the contents of the Windows Desktop to the clipboard.
4. Bring up the Windows Paintbrush program.
5. Create an image large enough to hold the most of the screen. See Microsoft Windows documentation for

details.

6. Using the Edit|Paste command, copy the contents of the clipboard into the image used by PaintBrush.
7. To print, use the PaintBrush's printing commands as detailed in the Microsoft Windows documentation.

This method will only print the visible portion of the Window. If program output has scrolled from view, it may need to be scrolled back into view and the screen again copied to the clipboard, copied from there to Paintbrush and then printed. For programs with significant output, this process may need to be repeated many times.

Reference:

4/22/99 12:40:58 PM

Last Modified: 01-SEP-99

## Article #15371: OVERLAY UNITS LOADED INTO BUFFER

Technical Notes Database

TN371D.txt      OVERLAY UNITS LOADED INTO BUFFER  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 5.0 5.5

### Description:

- Q. Is there any way to determine what overlay units are loaded into the overlay buffer?
- A. Using Turbo Pascal 5.0+, there is no defined method for determining which units are loaded into the overlay buffer area.

### Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15376: Keyboard Problems Using Borland Pascal and TPX

Technical Information Database

TI376D.txt    Keyboard Problems Using Borland Pascal and TPX  
Category    :General Programming  
Platform    :All  
Product     :Pascal    All

### Description:

At times, users of Borland Pascal or Turbo Pascal's TPX compilers find difficulty using their keyboards. Usual symptoms include the following.

1. The cursor keys persistently highlighting text.
2. The cursor keys randomly producing numeric characters instead of moving the cursor.
3. Keypresses behaving as if the shift key were stuck.
4. Keystrokes producing apparently random and unrelated characters.

The problem is usually related to timing issues. In 16-bit protected mode, the addressing mode used forces all memory accesses to be directed through a table of selectors rather than to physical memory directly. Thereby, making each memory access slightly slower than seen using real mode. On most systems, the difference in timing is insignificant and no ill effects are seen. However, some systems in certain configurations may experience the problems noted above.

To address this problem, most users are able to use the KEYB.COM program that accompanies MS-DOS. Place the following statement into your AUTOEXEC.BAT file.

```
C:\DOS\KEYB US
```

Then restart your computer system. It is important to run KEYB.COM before entering the IDE. It is not appropriate to run this program from the Dos Shell selection of the IDE's file menu.

Reference :

7/16/98 4:33:42 PM

Last Modified: 01-SEP-99

## Article #15378: TP OVERLAYS: RECLAIMING BUFFER MEMORY

Technical Notes Database

TN378D.txt TP OVERLAYS: RECLAIMING BUFFER MEMORY

Category :Pascal

Platform :All

Product :Turbo Pascal 5.5

Description:

Q. How do I reclaim the memory used by overlay buffer in TP5.5?

A. The following example demonstrates how to do so.

```
unit Marker;
```

```
interface
```

```
procedure RestoreHeap; procedure RestoreOverlay;
```

```
implementation
```

```
var OldHeapPtr,OldHeapOrg,Temp:pointer;
```

```
var OldHeapPtr,OldHeapOrg,Temp:pointer;
```

```
procedure RestoreHeap; begin
```

```
  Release(Temp); {2. Release all dynamic variables}
```

```
  OldHeapOrg:=HeapOrg; {3. Save Current Heap state }
```

```
  OldHeapPtr:=HeapPtr;
```

```
  HeapOrg:=ptr(OvrHeapOrg,$0); {4. Set Heap to Origin of Overlay  
Buffer}
```

```
  HeapPtr:=HeapOrg;
```

```
  Mark(Temp); {5. Mark the origin of this heap }
```

```
end;
```

```
procedure RestoreOverlay; begin
```

```
  Release(Temp); {6. Release all dynamic variables }
```

```
  HeapOrg:=OldHeapOrg; {7. Restore heap pointers }
```

```
  HeapPtr:=OldHeapPtr; end;
```

```
begin
```

```
  mark(temp); {1. Mark the beginning of heap after overlay buffer}
```

end.  
end.

The unit is to be used before any unit that places items into the heap during their initialization. You would call RESTOREHEAP before using the dynamic variables requiring memory used by the overlay buffer. A call to OVRCLEARBUF must come before any call to RESTOREHEAP to ensure that the overlay buffer is empty and to force a reload of overlays when you are done. Note, that all dynamic variables residing on the heap, before a RESTOREHEAP call is made, are lost. (But, you could code around this by manipulating the free list).

RESTOREOVERLAY clears the heap again and restores the heap pointers to point above the overlay buffer. You may then start using overlaid procedures and functions again.

This is a simple example and may not be suitable for all purposes.

Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15383: Suggestions for Installing from Backups

Technical Information Database

TI383D.txt      Suggestions for Installing from Backups

Category      :General Programming

Platform      :All

Product       :Pascal    All

### Description:

At times, Borland software is purchased using a disk format different from that used by your computer. For instance, your computer may use 3 1/2" diskettes but the software came with 5 1/4" diskettes or vice versa. Often folks will take the software to a friend using both diskette formats and make a set of disks in the format that they need.

However, when running the install program, you will find the program asking for more diskettes than you have or failing to do everything the manual indicates it will do. These problems result from moving the software from one format to another. The install program depends upon the original diskette format.

To resolve the problem, call Borland's Disk Replacement Line at (800) 621-3132 and order a set of diskettes in the proper format. Be sure to have your serial number handy when calling this service.

If immediate installation is required and sufficient hard disk space is available, you may copy all files from all diskettes to a single temporary directory on the hard disk. Run install from this directory on the hard disk. When install is finished, delete files from the temporary directory. Then you will be able to run the software normally.

### Reference:





## Article #15385: OVERLAYS \*.OVR FILES

Technical Notes Database

TN385D.txt      OVERLAYS \*.OVR FILES  
Category      :Pascal  
Platform       :All  
Product       :Turbo Pascal 5.0 5.5

### Description:

Q. How can I transfer a large overlay file onto a floppy disk?

A. If the file does not fit on a single density disk, then transfer it to a double density disk. In the latter case, your application will run only from the double density disk or a hard disk.

### Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15390: Creating Groups for Borland Pascal

Technical Information Database

TI390D.txt - Creating Groups for Borland Pascal

Category :Turbo Pascal  
Platform :All-32Bit  
Product :

### Description:

The install program for Borland Pascal with Objects is able to create and configure a Borland Pascal with Objects group for the Windows Program Manager. It presumes that the Windows shell being used is the Program Manager or some other utility that is DDE compatible with the Program Manager. If another shell is being used that is not DDE compatible with the Program Manager (File Manager for instance), a DDE error will occur when the Borland Pascal with Objects group is being created. This error can be ignored; the performance of Borland Pascal with Objects will not be adversely affected.

At times, the program used to create the Borland Pascal with Objects group is not automatically executed when Windows is run. In this case, it is possible to run this program from the Program Manager directly. From the Program Manager's File|Run menu enter the following command line.

```
GROUPS.EXE GROUPS.B$$
```

### Reference:

4/22/99 1:01:15 PM

Last Modified: 01-SEP-99

## Article #15396: Patch for the STDDLG Unit

Technical Information Database

TI396D.txt Patch for the STDDLG Unit  
Category :TVISION  
Platform :All  
Product :Pascal All

### Description:

When using the TFileDialog object with Borland Pascal 7.0, users may find that errors occur when navigating through subdirectories using the "..\" symbol in the files listbox.

The problem lies in the STDDLG.PAS file which normally resides in the \BP\RTL\TV subdirectory. After opening the file, search on the word RelativePath to find the following code:

```
function RelativePath(var S: PathStr): Boolean;  
begin  
  S := LTrim(RTrim(S));  
  RelativePath := not (S <> '') and ((S[1] = '\') or (S[2] =  
    ':')));  
end;
```

To resolve the problem, change the fourth line so that it looks like this:

```
RelativePath := not ((S <> '') and ((S[1] = '\') or (S[2] =  
  ':'))));
```

When you are done, you can copy your new TPU or TPP file into the \BP\UNITS subdirectory.

This problem only appears in Borland Pascal 7.0. STDDLGS shipping with Turbo Pascal 7.0 is correct.

### Reference:

7/16/98 4:33:43 PM

Last Modified: 01-SEP-99

## Article #15398: OVERLAY SUPPORT

Technical Notes Database

TN398D.txt      OVERLAY SUPPORT  
Category      :Pascal  
Platform      :All  
Product       :Turbo Pascal 5.0 5.5

### Description:

Q. Are overlays supported in 5.0+?

A. Yes! See the example program OVRDEMO.PAS and refer to the Turbo Pascal manual for information on overlays.

### Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15403: Mouse Driver Compatibility

Technical Information Database

TI403D.txt - Mouse Driver Compatibility

Category :Turbo Pascal

Platform :All

Product :

Description:

Mouse Drivers Known To Function With Borland/Turbo Pascal

From time to time, users of Borland/Turbo Pascal find that after running a program or returning from a DOS shell the mouse cursor disappears. The function of the mouse seems to be unaffected as button presses activate the menu bar and status line. However, the mouse cursor itself is gone and the user can not easily determine where the mouse is on screen.

This anomaly is frequently seen as a deficiency in Borland/Turbo Pascal, particularly as this effect is rarely if ever seen with products by other vendors.

In fact, the mouse driver in use is outdated or incomplete. As Borland/Turbo Pascal is a program able to run and debug programs, it must save the state of the mouse each time a user program is run or debugged. It must also save the state of the mouse when a DOS shell initiated. For this and all other mouse functions, the IDE relies on the mouse driver installed in either the CONFIG.SYS or AUTOEXEC.BAT file. These problems occur when a mouse driver is being used that is unable to properly save or restore its state.

Here is a list of mouse drivers known to Borland to correctly implement the Save/Restore mouse state functions.

Microsoft 6.1 or later  
Logitech 4.0 or later  
Mouse Systems 6.22 or later  
Genius 9.06 or later

Correcting such a problem with the mouse driver involves updating the mouse driver to a version compatible with those listed here. For mouse systems not described in this list, contact the manufacturer for an updated mouse driver. If an updated mouse driver is not available for a particular mouse system, it may well be worth considering replacing the mouse system with one known to function correctly.

PRODUCT : Borland/Turbo Pascal NUMBER : 1260  
VERSION : 6.0 & Up  
OS : DOS  
DATE : June 23, 1994 PAGE : 2/2  
TITLE : Mouse Driver Compatibility

Reference:

3/31/99 10:19:07 AM

Last Modified: 01-SEP-99



## Article #15405: EXTERNAL METHODS

Technical Notes Database

TN405D.txt      EXTERNAL METHODS  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 5.5

### Description:

Q. Can methods within an object be external?

A. Yes. Virtual and Static methods can be written as external code. There is no difference between an external Virtual or Static method. External Constructors and Destructors are difficult to write due to the Prolog and Epilog code within them.

### Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15410: Recommended PIF Settings for Windows

Technical Information Database

TI410D.txt - Recommended PIF Settings for Windows

Category :Turbo Pascal  
Platform :DOS  
Product :

### Description:

Every DOS program that runs under Windows will have an associated PIF file. These files designate how much memory an application uses and how it uses other aspects of your computer, such as the video memory.

If you do not associate a particular PIF file with a DOS application then it will use the `_Default.PIF` file which Microsoft places automatically in the Windows subdirectory. You should also be aware that whenever you open a DOS box via the MSDOS icon, then by default Windows uses the `DOSPrmpt.PIF` file, which Windows installs during setup. This PIF file will control the settings of an application that is started from inside a DOS box.

During the default installation Turbo Pascal 7.0 and Borland Pascal 7.0 both place correctly configured PIF files in the BIN subdirectory for the protected mode IDEs and debuggers such as BP, TPX and TDX. Borland Pascal will automatically create a Program Manager group with icons which are associated by default with these PIF files. This same chore is not performed automatically during the Turbo Pascal installation, so if you own that product you should be sure to associate TPX with the correct PIF file if you want to start it from a Windows icon. There is no need to associate real mode applications such as Turbo.EXE with a PIF file since they will normally run fine under either the `_Default.PIF` or the `DOSPrmpt.PIF`.

It is extremely easy to associate a DOS executable with a particular PIF file. To do so, follow the normal procedures for setting up a Windows Icon, except instead of associating an

executable with a particular icon, you should associate it with a particular PIF file. The PIF file itself will know where your executable is and how to start it.

If you want to know more about the set up or use of PIF files you should read the lengthy and detailed description of this subject in the Microsoft Windows Users Guide which shipped with your copy of Windows.

Here are the default PIF settings for the PIF files that shipped with BP7 and TP7:

=====

BP PIF:

Program FileName: BP.EXE

Window Title: Borland Pascal 7.0

Optional Parameters: /R+

Video Memory: Text

Memory Requirements

KB Required: 128

KB Desired: 640

EMS Memory

KB Required: 0

KB Limit: 1024

XMS Memory

KB Required: 0

KB Limit: 2048

Display Usage: Full Screen

Close Window on Exit: TRUE

Advanced Options

Background Priority: 50

Foreground Priority: 100

Detect Idle Time: TRUE

Emulate Text Mode: TRUE

Allow Fast Paste: TRUE

Application Shortcut Key: None

=====

TDX PIF:

Program FileName: TDX.EXE

Window Title: TDX

Video Memory: Text  
Memory Requirements  
  KB Required: 128  
  KB Desired: 640  
EMS Memory  
  KB Required: 0  
  KB Limit: 1024  
XMS Memory  
  KB Required: 0  
  KB Limit: 2048  
Display Usage: Full Screen  
Close Window on Exit: TRUE

Advanced Options  
Background Priority: 50  
Foreground Priority: 100  
Detect Idle Time: TRUE  
Emulate Text Mode: TRUE  
Allow Fast Paste: TRUE  
Application Shortcut Key: None

=====

TPX PIF:  
Program FileName: TPX.EXE  
Window Title: Turbo Pascal 7.0  
Optional Parameters: /R+  
Video Memory: Text  
Memory Requirements  
  KB Required: 128  
  KB Desired: 640  
EMS Memory  
  KB Required: 0  
  KB Limit: 1024  
XMS Memory  
  KB Required: 0  
  KB Limit: 2048  
Display Usage: Full Screen  
Close Window on Exit: TRUE

Advanced Options  
Background Priority: 50  
Foreground Priority: 100  
Detect Idle Time: TRUE  
Emulate Text Mode: TRUE

Allow Fast Paste: TRUE  
Application Shortcut Key: None

Reference:

3/31/99 10:10:25 AM

Last Modified: 01-SEP-99

## Article #15412: RECORDS VS. OBJECTS

Technical Notes Database

TN412D.txt RECORDS VS. OBJECTS

Category :Pascal

Platform :All

Product :Turbo Pascal 5.5

Description:

Q. What things can be done with Records that can not be done with Objects?

A. You cannot have:

1. Variant Objects.
2. Objects with absolutes.
3. Directly nested Objects.

You can have:

1. Pointers to objects.

Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15416: Booting clean -- tips for installing Turbo Pascal

Technical Information Database

TI416D.txt - Booting clean -- tips for installing Turbo Pascal

Category :Turbo Pascal

Platform :All

Product :

Description:

This TI explains how to boot clean. Booting clean involves bypassing both your CONFIG.SYS and AUTOEXEC.BAT files. This is often desirable if you have reason to believe that some of the calls in these files might be conflicting with the normal operation of Turbo Pascal. For instance, trouble installing Turbo Pascal is often the result of conflicts with other programs loaded by CONFIG.SYS or AUTOEXEC.BAT files. On other occasions, conflicts with drivers or TSRs loaded in a CONFIG.SYS or AUTOEXEC.BAT file may cause problems with protected mode technology.

If you are using DOS 6, you can boot clean simply by rebooting your system and waiting till you see the words "Starting MS-DOS". While that text is on your screen, press and release the F5 key or press and hold down the SHIFT key.

Here is a second method:

Format a bootable floppy disk that can be used in Drive A. You should refer to your DOS manuals for instructions on creating such a disk. However, on many systems, all you will need to do is place an unused disk and drive and then type:

```
FORMAT A: /S
```

Be sure that there are no important files on this disk, as it will be completely reformatted and all information on it will be lost.

Create on that disk a CONFIG.SYS file having only the statement

```
FILES=20
```

and an autoexec.bat file having only the statements

```
PROMPT $p$g  
PATH=C:\BP\BIN;C:\;C:\DOS
```

The path statement listed above assumes that you are installing Borland Pascal on to the C drive of your system and using the default subdirectories. Your path may be different than this. For instance, if you are installing Turbo Pascal, you should substitute the letters TP where BP is shown here.

Some systems may be using a hard drive that requires a special driver such as DMDRVR.SYS or Stacker. If you absolutely must, leave these entries in your configuration file. You can now 'boot clean' by inserting that disk into Drive A and rebooting your computer.

If you are having difficulties running any part of the product under Microsoft Windows, you will need a modified clean boot to run Windows. Add to your CONFIG.SYS file the statement the following line:

```
DEVICE=C:\DOS\HIMEM.SYS
```

and change the path statement in your autoexec.bat file to

```
PATH=C:\BP\BIN;C:\;C:\DOS;C:\WINDOWS
```

Also, ensure the statements in the WIN.INI and SYSTEM.INI files in your Windows directories match these:

```
WIN.INI  
    load=  
    run=  
SYSTEM.INI  
    shell=progman.exe
```

Reference:



3/30/99 12:51:28 PM

Last Modified: 01-SEP-99

## Article #15418: PROTECTED AND PRIVATE FIELDS

Technical Notes Database

TN418D.txt      PROTECTED AND PRIVATE FIELDS

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.5

Description:

Q. Does Turbo Pascal 5.5 support Protected or Private fields?

A. No, it does not.

Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15422: Does a Turbo Vision control have the input focus?

Technical Information Database

TI422D.txt Does a Turbo Vision control have the input focus?  
Category :TVISION  
Platform :All  
Product :Pascal All

### Description:

The following program demonstrates how to detect if a control in a Turbo Vision dialog has received the focus. The key is the `cmReceivedFocus` message, which gets sent to the owner of a control whenever that control is about to receive the focus. Here is sample code showing how to respond to the message:

```
procedure TTrainDialog.HandleEvent(var Event: TEvent);
begin
  if (Event.What = EvBroadCast)
  and (Event.InfoPtr = MyInputLine)
  and (Event.Command = cmReceivedFocus) then begin
    MessageBox('Got cmReceivedFocus Message, nil, mfOkCancel);
  end;
  TDialog.HandleEvent(Event);
end;
```

The above code sample assumes that an inputline has been inserted into a dialog. Whenever the user TABs to the input line, then the `cmReceivedFocus` message is sent to the control's owner. Note that its not sent to the control itself, but to the control's owner! Note also that this is a broadcast message. Of course its not enough just to pick up on broadcast `cmReceivedFocus` messages. You also have to detect which control has just received the focus. To do this, you can use the `InfoPtr` field of the incoming `TEVENT` record. This field will contain a pointer to the control that is about to receive the focus. The example code here literally compares the pointer to a second copy of a pointer to that control. If they are equal, then you know that you have the right control. Of course, the above code assumes that you have explicitly saved and declared a pointer to the inputline, as shown in the included example

program.

Now that you understand how `cmReceivedFocus` messages work, you might be interested in seeing another way of handling the same problem. This is to override the `SetState` function which actually generates `cmReceivedFocus` messages. In the program shown below, the following code fragment will enable and disable the state of a command on the statusline, depending on whether or not the control has the focus.

```
procedure TMyButton.SetState(AState: Word; Enable: Boolean);
begin
  inherited SetState(AState, Enable);
  if (AState = sfFocused) and (Enable = False) then
    DisableCommands([cmFoo]);
  if (AState = sfFocused) and (Enable = True) then
    EnableCommands([cmFoo]);
end;
```

The key to the above code is that it checks for `sfFocused` messages to find out whether the control is receiving or losing the focus.

When running the following program, watch the status line to see which control has the focus.

```
}
program DlgFocus;
uses
  App, Dialogs, Drivers, Menus, MsgBox, Validate, Views, Objects;

const
  cmDialogBox = 101;
  cmTest      = 102;
  cmButton    = 103;
  cmInput     = 104;
  InputLength = 128;
  TheMessage = 'The Input line just received focus.' + #13#10 +
               'Chose OK to set Ok in the Inputline' + #13#10 +
               'Chose Cancel to set Cancel in the InputLine';

type
  PMyButton = ^TMyButton;
  TMyButton = Object(TButton)
    procedure SetState(AState: Word; Enable: Boolean); virtual;
end;
```

```

PTrainDialog = ^TTrainDialog;
TTrainDialog = Object(TDialog)
    MyInputLine: PInputLine;
    constructor Init(Bounds: TRect; ATitle: String);
    procedure HandleEvent(var Event: TEvent); virtual;
end;

```

```

TMyApp = Object(TApplication)
    constructor Init;
    procedure InitStatusLine; virtual;
    procedure HandleEvent(var Event: TEvent); virtual;
    procedure DialogBox;
end;

```

```

procedure TMyButton.SetState(AState: Word; Enable: Boolean);
begin
    inherited SetState(AState, Enable);
    if (AState = sfFocused) and (Enable = False) then
        DisableCommands([cmButton]);
    if (AState = sfFocused) and (Enable = True) then
        EnableCommands([cmButton]);
end;

```

```

constructor TTrainDialog.Init(Bounds: TRect; ATitle: String);
var
    R: TRect;
    Control: PView;
    S: String;
begin
    TDialog.Init(Bounds, ATitle);
    R.Assign(2, 2, 37, 4);
    Insert(New(PMyButton, Init(R, 'Test', cmTest, bfNormal)));
    R.Assign(3, 5, 37, 6);
    MyInputLine := New(PInputLine, Init(R, InputLength));
    Insert(MyInputLine);
    R.Assign(2, 4, 24, 5);
    Insert(New(PLabel, Init(R, 'Delivery instructions',
        MyInputLine)));
    R.Assign(2, 7, 37, 9);
    Insert(New(PButton, Init(R, 'O~k~', cmOk, bfDefault)));

    S := '32';
    SetData(S);
end;

```

```

procedure TTrainDialog.HandleEvent(var Event: TEvent);
var
  Result: Word;
  S: String;
begin
  if (Event.What = EvBroadCast) and
    (Event.InfoPtr = MyInputLine) then
    case Event.Command of
      cmReceivedFocus: EnableCommands([cmInput]);
      cmReleasedFocus: DisableCommands([cmInput]);
    end;
  TDialog.HandleEvent(Event);
end;

```

```

constructor TMyApp.Init;
begin
  inherited Init;
  DisableCommands([cmInput, cmButton]);
end;

```

```

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = EvCommand then begin
    case Event.Command of
      cmDialogBox: DialogBox;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
end;

```

```

procedure TMyApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('', kbF10, cmMenu,
        NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
          NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialogBox,
            NewStatusKey('~Alt-F4~ Button Focused?',
              kbAltF4, cmButton,

```

```
        NewStatusKey('~Alt-F5~ InputLine Focused?',
                    kbAltF5, cmInput,
                    nil))))) ,
    nil)
));
end;

procedure TMyApp.DialogBox;
var
    R: TRect;
    D: PDialog;
begin
    R.Assign(20,5,60,15);
    D := New(PTrainDialog, Init(R, 'Hit Tab, watch statusline'));
    if ValidView(D) <> Nil then DeskTop^.ExecView(D);
    Dispose(D, Done);
end;

var
    A: TMyApp;
begin
    A.Init;
    A.Run;
    A.Done;
end.
```

Reference:

7/16/98 4:33:44 PM

Last Modified: 01-SEP-99

## Article #15424: CALLING ANCESTORS CONSTRUCTOR

Technical Notes Database

TN424D.txt      CALLING ANCESTORS CONSTRUCTOR

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 5.5

Description:

Q. How do you call a constructor from an ancestor's object?

A. You can call an ancestor's constructor directly from within the constructor of the current object. For example:

Type

```
    Type1 = Object
           Constructor Init;
```

```
End;
```

```
    Type2 = Object ( Type1 )
           Constructor Init;
```

```
End;
```

```
Constructor Type1.Init;
```

```
Begin
```

```
End;
```

```
Constructor Type2.Init;
```

```
Begin
```

```
    Type1.Init;
```

```
End;
```

```
...
```

Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99



## Article #15428: Declaring an array on the heap

Technical Information Database

TI428D.txt - Declaring an array on the heap

Category :Object Pascal  
Platform :All-32Bit  
Product :

### Description:

The following program, HEAPARY.PAS, demonstrates how to place an array on the heap. Programmers need to do this when they get errors saying that there are "too many variables" or that they are out of data space.

The solution to the above errors is simply to move an array up on the heap by creating a pointer to the array. Notice that the actual definition of the array is in the TYPE section, not the VAR section. If you write a statement like:

```
var MyArray: array[0..100] of Char
```

then you are allocating memory in the data segment, which is what you want to avoid. The solution is to define the array in the TYPE section, and then declare it in the VAR section:

```
var MyArray: PMyArray;
```

where PMyArray is a pointer to an array. To see the complete code for this type of example, refer to the HEAPARY program shown below.

The \$X is just so HEAPARY.PAS can use ReadKey without returning a result, the FlushKeyBuffer and Pause methods are thrown in case you are interested. The body of the program and the Type declaration are the parts you want to look at. Always remember to dispose memory

allocated with New!

Program HeapAry;

{ \$X+ }

Uses

Dos,  
Crt;

Type

PMMyArray = ^TMyArray;  
TMyArray = array[0..999] of LongInt;

procedure FlushKeyBuffer;

var Recpack : registers;

begin

with recpack do begin  
Ax := (\$0c shl 8) or 6;  
Dx := \$00ff;

end;

Intr(\$21, recpack);

end;

procedure Pause;

begin

FlushKeyBuffer; { Make sure key buffer is empty! }  
ReadKey; { Pause }

end;

var

MyAry: PMMyArray;  
i: Integer;

begin

ClrScr;

New(MyAry); { Allocate the memory on the heap }

for i := 0 to 999 do { Fill out array }

MyAry<sup>[i]</sup> := i;

for i := 500 to 510 do { Write the array to the screen }

WriteLn(MyAry<sup>[i]</sup>);

Pause;

Dispose(MyAry); { Dispose memory! }

end.

Reference :

4/22/99 1:01:59 PM

Last Modified: 01-SEP-99

## Article #15430: CONSTRUCTOR CALLING ANCESTOR'S CONSTRUCTOR

Technical Notes Database

TN430D.txt      CONSTRUCTOR CALLING ANCESTOR'S CONSTRUCTOR

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.5

Description:

Q. Should my current object's constructor call it's ancestor's constructors?

A. As a rule of thumb, this is the correct thing to do. It is okay to not do it, but it does allow initialization of whatever the previous constructors did.

Reference:

7/16/98 4:35:31 PM

Last Modified: 01-SEP-99

## Article #15434: Making a help group for Borland Pascal

Technical Information Database

TI434D.txt - Making a help group for Borland Pascal

Category :Turbo Pascal

Platform :All

Product :

Description:

Introduction

-----

Borland Pascal has many text files in the \BP\DOC directory that can be very helpful, but not quickly accessible from within Windows. This document will show you how you can create a program group that contains the additional help icons by using the GROUPS.EXE program provided with Borland Pascal, and creating a text file to tell it where the files are.

Instructions

-----

The steps to create the help group are the following:

- 1) You need to load up an editor program with a copy of the GROUPS.HLP text file. To do this:

Load up an editor, the DOS command EDIT for example, and either type in the example file below, or load this instruction file and strip off all the text except for the example code. DO NOT include the --TOP-OF-FILE-- and --END-OF-FILE-- in your text file.

Use the SAVE AS command to save the text file as:

C:\WINDOWS\GROUPS.HLP

This will make sure the file is saved under the Windows directory.

- 2) Change the directories in the example file to match the directories on your system.

In the included example file, the location of the Pascal root directory is C:\BP. If your Pascal files are in the directory D:\BP, use a Global Search and Replace to search for 'C:\BP' and replace it with 'D:\BP'. This will let the GROUPS program know where your Pascal help and icon files are.

If your Windows files are not located in the C:\WINDOWS directory, you will have to do another Global Search and Replace and replace 'C:\WINDOWS' with whatever the path is to your Windows files, D:\WINDOWS for example.

- 3) Under Windows, you will need to run the GROUPS.EXE file along with the text file you've just edited as its parameter. To do this, go to the FILE menu and select RUN. In the field for Command Line, type

```
GROUPS GROUPS.HLP
```

This should run the GROUPS program, and it should ask you if you'd like to create a Windows program group. Click on Yes.

If an error box pops up, you can try typing

```
C:\WINDOWS\GROUPS C:\WINDOWS\GROUPS.HLP
```

assuming that your Windows files are located in the C:\WINDOWS directory.

- 4) Look at the icons, they should all be yellow question marks with titles underneath. Also double click on some of the icons and make sure that an editor screen comes up with a copy of the help file in it.

If there are problems:

If either the icon is wrong, or the editor shows an empty screen, the directory setting for the Pascal files is

wrong. Make sure that the Pascal directory entry is of the form C:\BP, or D:\BORLANDP, or something similar that has a valid path to your Pascal directory.

If the editor doesn't come up, there is an error with the Windows directory path entry. It should be something like C:\WINDOWS.

If the icons don't appear but the Group Box does appear, there is a format error in the GROUPS.HLP file. Make sure the file follows the example shown here.

If the Group Box doesn't appear at all, GROUPS.EXE couldn't load the text file GROUPS.HLP. Make sure the path to the file is valid, and that the file exists where the path is pointing to.

#### Example File

-----

#### IMPORTANT:

In this example file, the lines after '[icons1]' have been split before the '='. The actual text file must be implemented with each help icon command on it's own line. You must combine the lines that start with an '=' with the line above it. For example, you could position the cursor on the '=' and press BACKSPACE to bring the line up and append it to the line above it.

Each line should have the form:

```
ExecuteFile=WorkingDir,Icon Caption,IconFile
```

When this example text file is finished, it should contain 28 lines only and will create 24 help icons.

#### NOTE:

Since \BP\DOC\utils.td is too large to fit into the NOTEBOOK.EXE editor, WRITE.EXE has been substituted.

FILENAME: C:\WINDOWS\GROUPS.HLP

[group1]

caption=Borland Pascal Help Files

file=BPH7.GRP

[icons1]

WINHELP.EXE C:\BP\BIN\BPW.HLP

=C:\WINDOWS,BPW Reference,C:\BP\BIN\help.ico

WINHELP.EXE C:\BP\BIN\WORKHELP.HLP

=C:\WINDOWS,Workshop Reference,C:\BP\BIN\help.ico

WINHELP.EXE C:\BP\BIN\WINSIGHT.HLP

=C:\WINDOWS,WinSight Reference,C:\BP\BIN\help.ico

WINHELP.EXE C:\BP\BIN\WINSPECTR.HLP

=C:\WINDOWS,WinSpector Reference,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC>manual.rw

=C:\WINDOWS,Workshop Manual Update,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\readme.rw

=C:\WINDOWS,Workshop RLE4 Notes,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\custcntl.rw

=C:\WINDOWS,Creating Custom Controls,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\bwccapi.rw

=C:\WINDOWS,Custom Controls Tech Notes,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\bwccstyl.rw

=C:\WINDOWS,Custom Controls Style Notes,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC>manual.prf

=C:\WINDOWS,Profiler Manual Update,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\helpme!.prf

=C:\WINDOWS,Profiler Reference,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\readme.td

=C:\WINDOWS,DB/PF/ASM Release Notes,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC>manual.td

=C:\WINDOWS,Debugger Manual Update,C:\BP\BIN\help.ico

write.exe C:\BP\DOC\utils.td

=C:\WINDOWS,Debugger Utilities Reference,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\helpme!.td

=C:\WINDOWS,Debugger Tips,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\hdwdebug.td

=C:\WINDOWS,Hardware Interface Notes,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\dbgboard.td

=C:\WINDOWS,Hardware Driver Notes,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\dpmiuser.doc

=C:\WINDOWS,DPMI Reference,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\temc.doc

=C:\WINDOWS,Turbo Macro Reference,C:\BP\BIN\help.ico

notepad.exe C:\BP\DOC\utils.doc

=C:\WINDOWS,Utilities Reference,C:\BP\BIN\help.ico



```
notepad.exe C:\BP\DOC>manual.tv
=C:\WINDOWS,Turbo Vision Reference,C:\BP\BIN\help.ico
notepad.exe C:\BP\DOC\helpme!.tsm
=C:\WINDOWS,Turbo Assembler Reference,C:\BP\BIN\help.ico
notepad.exe C:\BP\DOC\readme.tsm
=C:\WINDOWS,PROCDESC PROCTYPE Reference,C:\BP\BIN\help.ico
notepad.exe C:\BP\DOC\tcref.tsm
=C:\WINDOWS,TCREF Reference,C:\BP\BIN\help.ico
```

-----END-OF-FILE-----

## Additional Information

### How it works:

The GROUPS.EXE program takes a text file as input and creates a Windows group according to the data in the file. The text file is of the form:

```
ExecuteFile=WorkingDir,Icon Caption,IconFile
```

The help files will be displayed with NOTEPAD.EXE in the Windows directory. You can use any Windows editor to display the files by changing the first field from 'notepad.exe' to whatever editor you'd like to use.

The WinHelp help files will also be included in the help group and are preceded by 'WINHELP.EXE'. The help icon is the same as the original help files, and can be changed by naming a different .ICO file in the last field.

The GROUPS.EXE program is also the same one used by the install program to create the Pascal program group. To recreate the original program group, execute the command

```
GROUPS GROUPS.B$$
```

The GROUPS.B\$\$ is created by the installation process, and is located in the Windows directory.

### Reference:

3/31/99 10:10:53 AM

Last Modified: 01-SEP-99

## Article #15436: CONSTRUCTOR CALL WITHIN METHOD

Technical Notes Database

TN436D.txt      CONSTRUCTOR CALL WITHIN METHOD

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 5.5

Description:

Q. I am calling my constructor from within a method, why am I having problems?

A. The problem will arise when the constructor loads the new VMT pointer. It loads the pointer to the VMT for the constructor's table, not the instances. Therefore if a descendant calls an ancestor's constructor, the descendant's VMT will now point to the ancestors VMT. The problem now occurs when the descendant tries to call a method that was defined after the ancestor. The VMT entry for this method is unknown. Look at the following example:

Type

  L1 = Object

    Constructor Init;

    Procedure First; Virtual;

End;

  L2 = Object ( L1 );

    Constructor Init;

    Procedure Second; Virtual;

End;

Constructor L1.Init;

Begin

End;

Constructor L2.Init;

Begin

End;

Procedure L1.First;

Begin

  Init;

```
End;
```

```
Procedure L2.Second;
```

```
Begin
```

```
  Init;
```

```
End;
```

```
Var
```

```
  L : L2;
```

```
Begin
```

```
  L.Init;  { This calls L2.Init and loads a pointer to }  
           { the L2 VMT into L. }  
           }
```

```
  L.First; { This will call L1.First, which in turn calls }  
           { L1.Init because as far as the procedure is }  
           { concerned, the Self pointer is a pointer }  
           { to an object of type L1. }  
           }
```

```
  L.Second; { This is undefined. Since the VMT now }  
            { pointed to by L is L1's, the pointer to }  
            { method Second is undefined. Therefore, the }  
            { call to this method is undefined. }  
            }
```

```
  ...
```

\*\*\*Note: This also applies to polymorphic procedures.

Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15440: Turbo Vision local menu using right mouse button

Technical Information Database

TI440D.txt Turbo Vision local menu using right mouse button  
Category :TVISION  
Platform :All  
Product :Pascal All

### Description:

This is an example Turbo Vision program that demonstrates using local menus that are activated with the right mouse button and are positioned on screen by the mouse position.

```
program Localmenu;
uses
  Drivers, Objects, Views, App, Menus;

const
  cmNothing = 101;
type

  TMyApp = object(TApplication)
    LocalMenu: PMenu;
    LocalMenuOpen: boolean;
    constructor Init;
    procedure GetEvent(var Event: TEvent); virtual;
  end;

constructor TMyApp.Init;
begin
  inherited Init;
  LocalMenu := NewMenu(
    NewItem('Item~1~', '', 0, cmNothing, hcNOContext,
    NewItem('Item~2~', '', 0, cmNothing, hcNoContext,
    nil)));
  LocalMenuOpen := false;
  EventMask := EventMask or evBroadcast;
end;

procedure TMyApp.GetEvent(var Event: TEvent);
```

```

var
  MousePt: TPoint;
  R: TRect;
  Box: PMenuBox;
  NewEvent: TEvent;
  Code: Word;
begin
  inherited GetEvent(Event);
  if ((Event.What = evMouseDown) AND (Event.Buttons =
      mbRightButton) AND NOT LocalMenuOpen)
  then
  begin
    LocalMenuOpen := true;
    MousePt := Event.Where;
    if (MousePt.X > 70) then MousePt.X := 70;
    if (MousePt.Y > 20) then MousePt.Y := 20;
    ClearEvent(Event);
    R.Assign(MousePt.X, MousePt.Y, MousePt.X+11, MousePt.Y+2);
    Box := new(PMenuBox, Init(R, LocalMenu, nil));
    Code := ExecView(Box);
    Dispose(Box);
    LocalMenuOpen := false;
    NewEvent.What := evBroadcast;
    NewEvent.Command := code;
    PutEvent(NewEvent);
  end;
end;
var
  MyApp: TMyApp;
begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.

```

Reference:

7/16/98 4:33:44 PM

Last Modified: 01-SEP-99

## Article #15442: OVERRIDE VIRTUAL METHOD CALLING ANCESTOR

Technical Notes Database

TN442D.txt      OVERRIDE VIRTUAL METHOD CALLING ANCESTOR  
Category       :Pascal  
Platform        :All  
Product         :Turbo Pascal 5.5

### Description:

- Q. Can I override a virtual method and force a call to the ancestor objects method?
- A. No. Late binding will always call the current method and it defeats the purpose of object oriented programming to go around this feature.

### Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15447: Using procedural variables in Pascal

Technical Information Database

TI447D.txt - Using procedural variables in Pascal

Category :Object Pascal  
Platform :All-32Bit  
Product :

### Description:

The included program, FUNC1.PAS shows how to use procedural variables.

It creates both a pointer to a function and a pointer to a procedure and shows how to use them.

Func1 creates a pointer to a function called MyFunc, calls it once and then passes it to a procedure which then proceeds to call it. Therefore this examples shows how to assign and call a procedural variable, and also how to pass it to another procedure which can then call it.

The following lines demonstrate how to create types that can point at a procedure or function:

```
type
  TMyFunc = function(i: Integer): Integer;
  TMyProc = procedure;
```

Notice that the actual name of the function or procedure is left out, but that it is necessary to declare any variables passed to function or procedure, and to specify any return types from a function.

Its important to understand that TMyFunc and TMyProc are really pointers to functions, even though there is no need to call new and dispose, and no need to dereference the pointer with the caret symbol ("^").



Notice also that both MyProcedure and MyFunc are declared "Far". This is absolutely necessary when working with procedural variables, though of course functions declared in the header to a unit will automatically be far when called from another module.

```
program Funcl;  
  
type  
  TMyFunc = function(i: Integer): Integer;  
  TMyProc = procedure;  
  
procedure MyProcedure; far;  
begin  
  WriteLn('Hi');  
end;  
  
function MyFunc(I: Integer): Integer; far;  
begin  
  WriteLn(i);  
end;  
  
procedure Foo(F: TMyFunc);  
begin  
  F(5);  
end;  
  
var  
  F: TMyFunc;  
  P: TMyProc;  
begin  
  F := MyFunc;  
  F(10);  
  Foo(F);  
  P := MyProcedure;  
  P;  
end.
```

Reference:

4/22/99 1:02:30 PM

Last Modified: 01-SEP-99

## Article #15449: CONSTRUCTOR CALL TO ANCESTOR WITH VIRTUALS

Technical Notes Database

TN449D.txt      CONSTRUCTOR CALL TO ANCESTOR WITH VIRTUALS

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.5

Description:

Q. If a descendant of a virtual object defines no virtual methods of its own, does it need to call the ancestor's constructor?

A. If an object is a descendant of a virtual object, it must call that ancestor's constructor or it's own constructor. Even if the new object does not define any virtuals of its own. For example:

Type

```
A = Object
```

```
    Constructor Init;
```

```
    Procedure AA; Virtual;
```

```
End;
```

```
B = Object ( A )
```

```
End;
```

For each instance of A and of B, Init must be called or the Virtual Method Table pointer will not be loaded correctly.

Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15454: Using editor toolbox with newer versions of Pascal.

Technical Information Database

TI454D.txt - Using editor toolbox with newer versions of Pascal.

Category :Turbo Pascal  
Platform :All  
Product :

### Description:

This file is for owners of Turbo Pascal 6 or 7 who want to use the editor from the editor toolbox that shipped with Turbo Pascal Version 4.0. To begin, find the file called BINED.PAS that shipped with Toolbox. You should make the following change to BINED.PAS:

```
Procedure ResetBinaryEditor(var EDData: EdCB);  
  
var X: Integer; { <<--- ADD THIS LINE; ITS THE FIX! }  
  
begin  
    EditNew(EdData);  
end;
```

The addition of the above line sets up a stack frame for this proc.

Here's some additional text taken from the TP6 HELPM!.DOC:

With the exception of the Turbo Editor Toolbox, the 4.0 toolboxes will compile with Turbo Pascal 6.0. The Turbo Editor Toolbox needs the minor source code changes described below in order to work with the new heap manager:

MicroStar

-----

In MSVARS.PAS:

- 1) On line 219 after "var"  
    Add "FreePtr: Pointer;"
- 2) On line 295 after "begin"  
    Add "FreePtr := Ptr(Seg(HeapEnd^) - \$1000, 0);"

In INVOKE.PAS:

- 1) On line 18 after "Dos"  
Add ", MsVars"

FirstEd

-----

In EDVARS.PAS:

- 1) On line 139  
Add "FreePtr: Pointer;"  
On line 207  
Add "FreePtr := Ptr(Seg(HeapEnd^) - \$1000, 0);"

The TP6 notes then go on to talk about a new copy of BINED.OBJ. This is a fictitious copy and does not exist. Don't try to find it. Instead, add "THE FIX!" shown above.

Reference:

3/30/99 12:47:49 PM

Last Modified: 01-SEP-99

## Article #15456: LINKER STRIPS UNUSED STATIC METHODS

Technical Notes Database

TN456D.txt LINKER STRIPS UNUSED STATIC METHODS

Category :Pascal

Platform :All

Product :Turbo Pascal 5.5

### Description:

Q. It appears that my static methods are getting stripped from my program by the smart linker when they are unused. Yet, my virtual methods are not. How come?

A. Static methods can be stripped because it can be determined at link time what methods will be called. Virtual methods can not be stripped because the program will not know what methods will be used, and what will not, until run time. This is because of late binding.

### Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15461: Having two dialogs on your screen at once.

Technical Information Database

TI461D.txt Having two dialogs on your screen at once.

Category :TVISION

Platform :All

Product :Pascal All

Description:

```
{
  The following program demonstrates one technique for placing
  two dialogs on the screen at the same time in a Turbo Vision
  application. The key to this process is to make the first
  dialog a modeless dialog, and then the second can be either
  modal or modeless, depending on your needs. In this case,
  the second dialog is modal.
```

```
When a modal dialog is on the screen, the user must close it
before getting access to the other features in an application.
For instance, a user cannot use the main programs menu while
modal dialog is on the screen. This is not true of modeless
dialogs. Modal dialogs are started with ExecView or ExecDialog.
Modeless dialogs are "Inserted" into the desktop.
```

```
}

{$X+}
Program TwoDlgs;
```

Uses

```
  App,
  Dialogs,
  Drivers,
  Menus,
  MsgBox,
  Views,
  Objects;
```

const

```
  cmDialogBox = 101;
  cmPress2 = 102;
```

```

type
  Consistency = (Solid, Runny, Melted);
  Cheese = (None, Hvarsti, Tilset, Jarlsberg);

PTrainDialog = ^TTrainDialog;
TTrainDialog = Object(TDialog)
  constructor Init(Bounds: TRect; ATitle: String);
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure Press2;
end;

TMyApp = Object(TApplication)
  constructor Init;
  procedure InitMenuBar; virtual;
  procedure InitStatusLine; virtual;
  procedure HandleEvent(var Event: TEvent); virtual;
  procedure DialogBox;
end;

constructor TTrainDialog.Init(Bounds: TRect; ATitle: String);
var
  R: TRect;
  Control: PView;
begin
  TDialog.Init(Bounds, ATitle);

  R.Assign(2, (Bounds.B.Y - Bounds.A.Y) - 12,
    (Bounds.B.X - Bounds.A.X) - 2,
    (Bounds.B.Y - Bounds.A.Y) - 3);
  Insert(New(PButton, Init(R, 'Click Here',
    cmPress2, bfNormal)));
end;

procedure TTrainDialog.HandleEvent(var Event: TEvent);
begin
  TDialog.HandleEvent(Event);
  if Event.What = EvCommand then begin
    case Event.Command of
      cmPress2: Press2;
      else Exit;
    end;
  ClearEvent(Event);
end;

```



```

end;

procedure TTrainDialog.Press2;
var
  D: PDialog;
  R: TRect;
  Control: Word;
begin
  R.Assign(48,3,74,14);
  D := New(PDialog, Init(R, 'Another Dialog'));

  R.Assign(4, 2, 22, 10);
  D^.Insert(New(PButton, Init(R, 'Click To Close',
    cmOk, bfDefault)));

  Control := DeskTop^.ExecView(D);
  Dispose(D, Done);
end;

constructor TMyApp.Init;
var
  R: TRect;
begin
  TApplication.Init;
end;

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = EvCommand then begin
    case Event.Command of
      cmDialogBox: DialogBox;
    else
      Exit;
    end;
  ClearEvent(Event);
end;
end;

procedure TMyApp.InitMenuBar;
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;

```

```

MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~D~ialogs', 'Alt-D', kbAltD,
      cmDialogBox, hcNoContext,
    NewLine(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
        nil))))),
  nil))
));
end;

```

```

procedure TMyApp.InitStatusLine;

```

```

var R: TRect;

```

```

begin

```

```

  GetExtent(R);

```

```

  R.A.Y := R.B.Y - 1;

```

```

  StatusLine := New(PStatusLine, Init(R,

```

```

    NewStatusDef(0, $FFFF,

```

```

      NewStatusKey('', kbF10, cmMenu,

```

```

      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,

```

```

      NewStatusKey('~Alt-D~ Dialog', kbAltD, cmDialogBox,

```

```

      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,

```

```

      nil))))),

```

```

    nil)

```

```

  ));

```

```

end;

```

```

procedure TMyApp.DialogBox;

```

```

var

```

```

  R: TRect;

```

```

  D: PDialog;

```

```

  Control: Word;

```

```

begin

```

```

  R.Assign(20,5,60,21);

```

```

  D := New(PTrainDialog, Init(R, 'Training Dialog'));

```

```

  DeskTop^.Insert(D);

```

```

end;

```

```

var

```

```

  A: TMyApp;

```

```

begin

```

```

  A.Init;

```

```

  A.Run;

```

```

  A.Done;

```

```

end.

```

Reference:

7/16/98 4:33:45 PM

Last Modified: 01-SEP-99

## Article #15463: SIZEOF OBJECTS CONSTRUCTOR

Technical Notes Database

TN463D.txt      SIZEOF OBJECTS CONSTRUCTOR

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.5

Description:

Q. Why does `Sizeof(MyObject)` return a size 2 bytes larger than I expect when no virtual methods are used?

A. By placing a constructor in your object, you're making the object virtual.

Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15468: Printing in the TP/BP IDE

Technical Information Database

TI468D.txt - Printing in the TP/BP IDE

Category :Printing  
Platform :All  
Product :

### Description:

How to print from the Pascal Integrated Development Environment  
=====

All printing is handled from the printer filter (prnfltr.exe) executable that is located in the bin directory. The source code for the prnfltr was included in both the Turbo Pascal 7.0 and the Borland Pascal 7.0 packages, although in different places. PRNFLTR.PAS is found in either the TPBIN directory or in the BPEXAMPLESUTILS directory. (Also look at the file LISTER.PAS in the BPEXAMPLESDOS or TPEXAMPLES. It is an excellent alternative to the standard printing options available in Borland Pascal.) From this file we can see the various printing options for PRNFLTR. The options can be changed from under File | Printer Setup | Command Line. The \$NOSWAP does not change.

For example, if changing from /EPSON support to /HP support, the command line should change from this:

```
$NOSWAP /EPSON
```

To this :

```
$NOSWAP /HP
```

Once the changes have been made to the command line for the PRNFLTR, be sure to save the current configuration (this will keep you from having to change the configuration every time you wish to print). The current configuration can be saved by choosing Option | Save menu option.

## OPTIONS FOR PRNFLTR

=====

```
{*****}
{
{ Printer output filter example }
{ Copyright (c) 1992 by Borland International }
{
{*****}
```

The following printers are supported:

EPSON and compatibles

HP LaserJet II, III, IIP, IID, IIID, IIISi and compatibles  
(Italics are available on IIIx, IIP)

ADOBE(R) PostScript(R)

ASCII (simply strips the highlight codes  
before sending to Lst)

Command line options:

```
/EPSON - Output EPSON printer codes
/HP - Output HP LaserJet codes
/PS - Output PostScript
/ASCII - Strip highlight codes (Default)

/Lxx - Lines per page (Default 55)
/Txx - Tabsize (Default 8)
/O[file] - Output to file or device (Default LPT1)
```

## DEFAULT LPT1

=====

The default port for printing is lpt1. If you do not have your printer set up on lpt1 the way to change the default is by using the /O[file] option, replacing [file] with whatever port the printer is on (such as lpt2).

Example (printing to an HP from lpt2):

```
$NOSWAP /HP /Olpt2
```

PRINTING PROGRAM OUTPUT

=====

If you want to print the output of your program (i.e. what happens when the program is run) the document TI1255 will describe the process.

"CANNOT FIND PRNFLTTR"

=====

If you are getting the error prnfltr cannot be found, it can be solved by placing the BIN directory in the path statement of the AUTOEXEC.BAT.

For example, if your AUTOEXEC.BAT's path statement says this:

```
path=c;;c:dos;c:windows;
```

The path statement should be changed to reflect the full path to the bin directory.

Example:

```
path=c;;c:dos;c:windows;c:bpbin;
```

Then restart your machine and try again to print.

\*\*\*

PRINTING FROM A LASER (OR A LASERJET) PRINTER

=====

If the file that you would like to print is less than a page long, sometimes it will be unable to print. The two possible solutions to this are as follows.

- 1) Put carriage returns into the document until it is more than a page long.
- 2) Also, placing a page feed character at the end of the document can sometimes solve the problem.

To create a page feed character, press the control key

and while holding it down press the P key. Keeping the control key pressed (still), then press the L key. (Recap: control P L ). A symbol should appear on the screen where the cursor was. That is the page feed character.

## TROUBLE SHOOTING

=====

If you have selected the option for the type of printer (or compatible printer) that you have and it is still not correctly printing, there are several possible things that could solve the error.

The first thing that could solve the error is to run a "clean boot".

Try a different option for the printer. From /HP and /Epson, switch to /ASCII. Also, disable the option "Send highlighting escape codes" (which appears just above the command line under File | Printer Setup) as well as using the /ASCII option could solve the problem.

If you are still not able to print after trying several combinations of options, it is possible that your printer is not correctly set up to work from DOS. (If you are on a network it is also possible that your machine is not correctly configured to print to the network printer. Check with your network administrator to see if you are correctly connected.) To test for your printer setup, try printing straight from DOS. Printing from a word processor or from Windows is not comparable because those products provide their own drivers for printing. It is possible to print from DOS using copy or print.

```
copy test.c lpt1 {if the printer is set up for lpt1}
print test.c     {if the printer is set up for lpt1}
```

If neither of those commands works then it may be necessary to reconfigure your printer for printing from DOS before you will be able to print from the Pascal Integrated Development Environment.

Reference:



3/31/99 10:11:34 AM

Last Modified: 01-DEC-99

## Article #15470: FILES OF OBJECT TYPE

Technical Notes Database

TN470D.txt      FILES OF OBJECT TYPE  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 5.5

### Description:

Q. Why can't I make a file of ObjectType?

A. Because by the rules of polymorphism, any descendant of ObjectType would be type compatible and be able to be written to disk as well. The problem with this is that the descendants may be (and usually are) of a larger size than the ObjectType itself. Pascal's file structure require all records to be the same size, otherwise, how do you know which size object do you read in? For an example on how to do Object disk I/O, please see the STREAMS example on the distribution diskettes.

### Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15475: Getting and setting a disks serial number

Technical Information Database

TI475D.txt     Getting and setting a disks serial number  
Category     :General Programming  
Platform     :All  
Product      :Pascal    All

Description:

The following program shows how to read and write the serial number on a disk drive. It uses interrupt \$21 function \$69 and reads the information back and forth into a record with the following declaration:

```
TSerPacket = record
  Info:Word;
  SerNo: Longint;
  Vol: Array[1..11] of char;
  FileType: Array[1..8] of char;
end;
```

The first word of the above record should always contain zero, the second contains the serial number, the third the volume label and the fourth the FileType, which is usually either FAT 16 or FAT 12.

When setting a new serial number, the programs first reads the current information, and then sets the new serial number and writes it to disk.

The code shown below provides the user with three functions:

```
GetSerialInfo: Get the current serial number
SetSerialInfo: Set a new serial number
DisplaySerialInfo: Display the information in the current
                   variable of type TPacket.
```

Notice that DisplaySerialInfo does not write information to a disk drive, but only displays that information to

the user via the screen. All of these routines depend on a global variable of type TSerPacket, which is called SPacket. It makes no sense to call DisplaySerialInfo before calling GetSerialInfo.

The drive specification is passed as a number, where the default drive is zero, drive A is one, drive B is two, etc.

```
program SetDiskSer;
```

```
{ $A+, B-, D+, E+, F-, G+, I+, L+, N-, O-, P-, Q-, R-, S-, T-, V+, X+, Y+ }  
{ $M 16384, 0, 655360 }
```

```
type
```

```
  TSerPacket = record  
    Info: Word;           { Info level always zero }  
    SerNo: Longint;      { Disk }  
    Vol: Array[1..11] of char;  
    FileType: Array[1..8] of char;  
  end;
```

```
var
```

```
  SPacket: TSerPacket;
```

```
function GetSerialInfo(Drive: Byte): Byte; assembler;
```

```
asm
```

```
  mov ah, 69h  
  mov al, 0 { This is where you 1 set serial, 0 to get serial  
            number }  
  mov bl, 1 { drive number 0 = default, 1 = A, 2 = B , 3 = C }  
  mov dx, offset [SPacket]  
  int 21h
```

```
end;
```

```
function SetSerialInfo(Drive: Byte; SerNo: LongInt): Boolean;
```

```
var
```

```
  S: String;
```

```
begin
```

```
  GetSerialInfo(Drive);  
  SPacket.SerNo := SerNo;
```

```
asm
```

```
  mov ah, 69h
```

```

mov al, 1 { This is where you 1 set serial, 0 to get serial
          number }
mov bl, Drive { drive number 0 = default, 1 = A, 2 = B ,
              3 = C }
mov dx, offset [SPacket]
int 21h
end;
end;

```

```

procedure DisplaySerialInfo;

```

```

var
  S: String;
begin
  WriteLn('Info level: ', SPacket.Info);
  WriteLn('Serial Num: ', SPacket.SerNo);
  FillChar(S, SizeOf(S), #0);
  Move(SPacket.Vol[1], S[1], 11);
  S[0] := #11;
  WriteLn('Vol: ', S);
  FillChar(S, SizeOf(S), #0);
  Move(SPacket.FileType, S[1], 8);
  S[0] := #8;
  WriteLn('Type: ', S);
end;

```

```

begin
  SetSerialInfo(1, 12); { One means Drive A, 16 is the serial
                        number }
  GetSerialInfo(1);
  DisplaySerialInfo;
  ReadLn;
end.

```

Reference:

7/16/98 4:33:45 PM

Last Modified: 01-SEP-99

## Article #15477: SAVING OBJECTS TO DISK

Technical Notes Database

TN477D.txt      SAVING OBJECTS TO DISK  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 5.5

### Description:

- Q. Can I save my objects to a disk file like I can a record structure?
- A. We have provided an example program on the distribution diskette, STREAMS, which documents how to save an object to disk.

### Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15481: Creating a temporary stack.

Technical Information Database

TI481D.txt - Creating a temporary stack.

Category :Object Pascal  
Platform :All-32Bit  
Product :

Description:

{  
The following code shows how to create a temporary stack for use during ISRs and in other situations where the default stack might not be big enough.

The following program intentionally creates a very small default stack by using the \$M compiler directive. It then calls a function which pushes \$2000 bytes onto the stack, thereby setting up a situation which should cause a 202 error.

To avoid the error, the program creates a temporary stack by allocating \$4000 bytes on the heap and moving this value into SS. It then sets SP to point at the end of this new stack and proceeds to call the function, which does not fail because of the new temporary stack.

This code has been tested in both real and protected mode.  
}

```
{ $M $800, 0, $12000 }
```

```
procedure BreakDefaultStack;  
var  
  a:array[0..$2000] of char;  
begin  
  Writeln('Hello');  
end;
```

```
var
```

```
aSS, aSP: Word;  
Stck: pointer;
```

```
begin  
  getmem(Stck, $4000);  
  Word (Stck^) := 1;  
  
  asm  
    mov aSP, sp  
    mov ax, ss  
    mov aSS, ax  
    mov ax, word ptr [Stck]  
    mov bx, word ptr [stck+2]  
    add ax, $4000-2  
    cli  
    mov ss, bx  
    mov sp, ax  
    sti  
  end;
```

```
BreakDefaultStack;
```

```
asm  
  mov ax, aSS  
  mov ss, ax  
  mov sp, aSP  
end;
```

```
  freemem(Stck, $4000);  
end.
```

Reference:

4/22/99 1:03:13 PM

Last Modified: 01-SEP-99



## Article #15483: DESTRUCTOR HEAP CORRUPTION

Technical Notes Database

TN483D.txt      DESTRUCTOR HEAP CORRUPTION

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.5

Description:

Q. Why is my destructor fragging my heap?

A. Use:

```
Dispose(ptr,done);
```

instead of:

```
ptr^.done;  
Dispose(done);
```

Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15488: How to read an enhanced keyboard

Technical Information Database

TI488D.txt      How to read an enhanced keyboard

Category        :General Programming

Platform        :All

Product         :Pascal    All

### Description:

{

The code included in this example shows how to read an enhanced keyboard using Turbo Pascal. That is, it detects if the F11, F12, etc keys are pressed. It provides a substitute for the ReadKey and KeyPressed functions.

This TI includes two source files, the first a unit called ENHKEY.PAS and the second a test program called TEST.PAS. ENHKEY.PAS contains three routines:

KeyEPressed: Works just like the CRT routine called KeyPressed, except it detects keypresses on enhanced keys.

NewReadKey: Works very much like ReadKey, except it detects enhanced keys.

ReadEKey:      This is the raw readkey function. It returns a word.

The high order word contains the scan code and the low word contains the regular key. Some users might not want to call this function directly, but instead might want to access it through the NewReadKey function, which acts much more like the original ReadKey function from the CRT unit.

The code in the EnhKey unit depends on interrupt 16h, functions 10 and 11, both of which assume the presence of AT or better computer. In this day and age, that's a fairly safe bet, but you should be aware

that this code will not run on an old XT.

Notice that if NewReadKey returns zero the first time it is called, you can grab the scan code in the global variable ScanCode, or you can call NewReadKey a second time to return the ScanCode.

```
unit EnhKey;
```

```
interface
```

```
var
```

```
    ScanCode: Byte;
```

```
function KeyEPressed: Boolean;
```

```
function ReadEKey: Word;
```

```
function NewReadKey: Char;
```

```
implementation
```

```
function KeyEPressed: Boolean; assembler;
```

```
asm
```

```
    mov ah, $11
```

```
    int 16h
```

```
    mov ax, 1
```

```
    jnz
```

```
True
```

```
    xor ax, ax
```

```
True:
```

```
end;
```

```
function ReadEKey: Word; assembler;
```

```
asm
```

```
    mov ah, 10h
```

```
    int 16h
```

```
end;
```

```
function NewReadKey: Char;
```

```
var
```

```
    Ch: Word;
```

```
begin
```

```
    if ScanCode <> 0 then begin
```

```

    NewReadKey := Char(ScanCode);
    ScanCode := 0;
    exit;
end;
Ch := ReadEKey;
if Lo(Ch) = 0 then begin
    ScanCode := Hi(Ch);
    NewReadKey := #0;
    exit;
end;
NewReadKey := Char(Lo(Ch));
end;

begin
    ScanCode := 0;
end.
begin
    ScanCode := 0;
end.

{ ===== The Test Program ===== }

program Test;
uses
    EnhKey;

var
    Ch: Char;
    i: Integer;
begin
    i := 0;
    while (Ch <> #27) do begin
        if KeyEpressed then begin
            Ch := NewReadKey;
            if Ch = #0 then begin
                Ch := NewReadKey;
                WriteLn('Enhanced: ', Ch)
            end else
                WriteLn('Normal: ', Ch);
            inc(i);
        end;
    end;
end.
end.

```

Reference:

7/16/98 4:33:45 PM

Last Modified: 01-SEP-99

## Article #15490: CONSTRUCTOR USE

Technical Notes Database

TN490D.txt      CONSTRUCTOR USE  
Category      :Pascal  
Platform      :All  
Product       :Turbo Pascal 5.5

Description:

Q. What are the three purposes of a Constructor?

- A. 1. Insert the address of the VMT into the Object variable.  
    (Implicit)
2. To allocate memory for the Object variable.  
    (Implicit)
3. To initialize the Object variable.  
    (Explicit)

Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15494: Copying a desktop window into your program's window

Technical Information Database

TI494D.txt      Copying a desktop window into your program's window  
Category        :General Programming  
Platform        :All  
Product         :Pascal    All

Description:

```
{  
  The following program shows how to copy the contents of the  
  current desktop window into the main window of your program.  
}
```

```
{*****}  
{  Copy DeskTop Demo program  
{*****}  
{
```

Here's the beginning of the resource:

```
-----  
MENU_1 MENU  
BEGIN  
    MENUITEM "Blit_Upper_Left_Corner_of_Desktop", 101  
END  
-----
```

End of Resource (Don't include lines)

```
}  
program MyProgram;  
  
uses WinTypes, WinProcs, OWindows;  
{ $R BMPDESK }  
  
const  
    idBlitIt = 101;  
  
type  
    TMyApplication = object(TApplication)  
        procedure InitMainWindow; virtual;  
    end;
```

```

type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    constructor Init(AParent: PWindowsObject; Name: PChar);
    destructor Done; virtual;
    procedure BlitIt(var Msg: TMessage);
      virtual Cm_First + idBlitIt;
  end;

{-----}
{ TMyWindow's method implementations: }
{-----}

constructor TMyWindow.Init(AParent: PWindowsObject; Name: PChar);
begin
  inherited Init(AParent, Name);
  Attr.Menu := LoadMenu(HInstance, 'Menu_1');
end;

destructor TMyWindow.Done;

begin
  inherited Done;
end;

procedure TMyWindow.BlitIt(var Msg: TMessage);
var
  DeskDc: HDC;
  TempDC, PaintDC: HDC;
  MyBitMap: HBitmap;
  R: TRect;
begin
  DeskDc := GetDC(GetDeskTopWindow);
  PaintDC := GetDC(HWindow);
  GetClientRect(HWindow, R);
  BitBlt(PaintDC, 0, 0, R.right, R.bottom, DeskDC, 0, 0,
SRCCopy);
  ReleaseDC(HWindow, PaintDC);
  ReleaseDC(GetDeskTopWindow, DeskDC);
end;

{-----}
{ TMyApplication's method implementations: }
{-----}

```



```
procedure TMyApplication.InitMainWindow;
begin
    MainWindow := New(PMyWindow, Init(nil, 'Sample ObjectWindows
        Program' ));
end;
```

```
{-----}
{ Main program: }
{-----}
```

```
var
    MyApp: TMyApplication;

begin
    MyApp.Init('MyProgram');
    MyApp.Run;
    MyApp.Done;
end.
```

Reference:

7/16/98 4:33:46 PM

Last Modified: 01-SEP-99

## Article #15496: OBJECT EXE FILE SIZE OVERHEAD

Technical Notes Database

TN496D.txt      OBJECT EXE FILE SIZE OVERHEAD

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 5.5

Description:

Q. How much overhead will result in the \*.EXE file from using the object oriented style?

A. The overhead will result from the pointer from the object to its method. This is a 4 byte pointer, so there isn't that much extra code generated.

Reference:

7/16/98 4:35:32 PM

Last Modified: 01-SEP-99

## Article #15501: Configuring the Borland Pascal icon in OS/2

Technical Information Database

TI501D.txt - Configuring the Borland Pascal icon in OS/2

Category :Turbo Pascal  
Platform :All  
Product :

### Description:

The BORLAND PASCAL DOS Protected Mode Integrated Development Environment (IDE) is a DOS application that is invoked by executing BP.EXE. When OS/2 migrates BP.EXE, it migrates it as a WINDOWS application rather than as a DOS application. Therefore, when double-clicking on the BORLAND PASCAL (BP.EXE) icon, the program will fail to load properly.

This document explains how to configure the BORLAND PASCAL (BP.EXE) icon to run successfully as a DOS application.

Boot up OS/2 and follow these steps:

1. Highlight the BORLAND PASCAL (BP.EXE) icon that was created during migration.
2. With the mouse pointing at the highlighted BORLAND PASCAL icon, right mouse click and a menu will appear.
3. Left mouse click on the ARROW in the upper right hand corner of the OPEN line.
4. Left mouse click on SETTINGS.
5. If you are not at the PROGRAM tab, left mouse click on the PROGRAM tab.
6. Delete everything in the PATH & FILE NAME box. When the PATH & FILE NAME box is empty, insert an asterisk (\*) into the PATH & FILE NAME box.

7. Insert the following into the PARAMETERS box:

```
/c c:\bp\bin\bp.exe
```

Be sure to substitute the correct path to BP.EXE if you deviated from the suggested defaults.

The statement, `/c c:\bp\bin\bp.exe`, specifies that the command interpreter is to execute `c:\bp\bin\bp.exe` and then return to the command interpreter when finished.

8. Left mouse click on the SESSION tab.

9. Left mouse click on the DOS FULL-SCREEN button.

10. Left mouse click on DOS SETTINGS.

11. Scroll down to DPMI\_DOS\_API in the SETTING box, highlight DPMI\_DOS\_API.

12. In the VALUE box, choose ENABLED.

13. SAVE and EXIT.

The BORLAND PASCAL icon is now configured to run as a DOS Protected Mode application. You can now double-click on the BORLAND PASCAL icon and the IDE will load successfully.

Reference:

3/31/99 10:18:58 AM

Last Modified: 01-SEP-99

## Article #15506: Unknown runtime errors. What's error 163?

Technical Information Database

TI506D.txt Unknown runtime errors. What's error 163?

Category :General Programming

Platform :All

Product :Pascal All

### Description:

Pascal sometimes returns runtime errors that aren't listed in the manuals. If you ever get an IORESULT code in the range 150-199, just subtract 131 from it and then look in your DOS reference manual (or in the list presented below) for the relevant critical errors.

One error that users see quite often is number 163, which is a sharing violation. It occurs when two users try to access the same file at the same time.

Here's a list of critical errors. The first column has the error number in decimal format, the second has the number in hex, and the third provides an explanation of the error:

DEC	HEX	EXPLANATION
01	-- 01	Invalid function number
02	-- 02	File not found
03	-- 03	Path not found
04	-- 04	Too many open files (no handles left)
05	-- 05	Access denied
06	-- 06	Invalid handle
07	-- 07	Memory control blocks destroyed
08	-- 08	Insufficient memory
09	-- 09	Invalid memory block address
10	-- 0A	Invalid environment
11	-- 0B	Invalid format
12	-- 0C	Invalid access mode (open mode is invalid)
13	-- 0D	Invalid data
14	-- 0E	Reserved
15	-- 0F	Invalid drive specified
16	-- 10	Attempt to remove current directory
17	-- 11	Not same device

18 -- 12 No more files  
19 -- 13 Attempt to write on a write-protected diskette  
20 -- 14 Unknown unit  
21 -- 15 Drive not ready  
22 -- 16 Unknown command  
23 -- 17 CRC error  
24 -- 18 Bad request structure length  
25 -- 19 Seek error  
26 -- 1A Unknown media type  
27 -- 1B Sector not found  
28 -- 1C Printer out of paper  
29 -- 1D Write fault  
30 -- 1E Read fault  
31 -- 1F General failure  
32 -- 20 Sharing violation  
33 -- 21 Lock violation  
34 -- 22 Invalid disk change  
35 -- 23 FCB unavailable  
36 -- 24 Sharing buffer overflow  
37 -- 25 Reserved  
38 -- 26 Unable to complete file operation (DOS 4.x)  
39-49 -- 27-31 Reserved  
50 -- 32 Network request not supported  
51 -- 33 Remote computer not listening  
52 -- 34 Duplicate name on network  
53 -- 35 Network name not found  
54 -- 36 Network busy  
55 -- 37 Network device no longer exists  
56 -- 38 NetBIOS command limit exceeded  
57 -- 39 Network adapter error  
58 -- 3A Incorrect network response  
59 -- 3B Unexpected network error  
60 -- 3C Incompatible remote adapter  
61 -- 3D Print queue full  
62 -- 3E No space for print file  
63 -- 3F Print file deleted  
64 -- 40 Network name deleted  
65 -- 41 Access denied  
66 -- 42 Network device type incorrect  
67 -- 43 Network name not found  
68 -- 44 Network name limit exceeded  
69 -- 45 NetBIOS session limit exceeded  
70 -- 46 Temporarily paused  
71 -- 47 Network request not accepted  
72 -- 48 Print or disk redirection is paused

73 -- 49-4F Reserved  
80 -- 50 File already exists  
81 -- 51 Reserved  
82 -- 52 Cannot make directory entry  
83 -- 53 Fail on INT 24  
84 -- 54 Too many redirections  
85 -- 55 Duplicate redirection  
86 -- 56 Invalid password  
87 -- 57 Invalid parameter  
88 -- 58 Network device fault  
89 -- 59 Function not supported by network (DOS 4.x)  
90 -- 5A Required system component not installed (DOS 4.x)

Reference:

7/16/98 4:33:46 PM

Last Modified: 01-SEP-99

## Article #15511: Creating an OWL app withoug a border and caption

Technical Information Database

TI511D.txt      Creating an OWL app withoug a border and caption

Category      :OWL

Platform      :All

Product      :Pascal    All

Description:

Sometimes programmers of OWL applications want to open a main window that has no border, no caption, and which fills the entire screen. The following example shows how to do this.

To get started, override the Init Constructor. Then set the Windows style to WS\_VISIBLE and WS\_POPUP:

```
Attr.Style := ws_Visible or ws_PopUp;
```

After doing this, the program will appear with no border and no caption. However, the window will not be visible unless you define its size. To give the program a shape, assign it a width and height:

```
Attr.X := 100;
```

```
Attr.Y := 100;
```

```
Attr.H := 200;
```

```
Attr.W := 200;
```

Of course, if you want to show the window in a maximized state, then you can simply send it a WM\_SYSCOMMAND message from the programs SETUPWINDOW procedure:

```
procedure TStepWindow.SetupWindow;
```

```
begin
```

```
  inherited SetupWindow;
```

```
  PostMessage(HWindow, WM_SYSCOMMAND, SC_MAXIMIZE, 0);
```

```
end;
```

To sum up, the shortest way from point A to point B is to



first give the window the WS\_POPUP style, and then maximize it with the WM\_SYSCOMMAND message. If you want to create a window that is not maximized, then be sure to set the window's width and height. The example below shows a maximized window:

```
program NoCaption;

uses WinTypes, WinProcs, OWindows, Strings;

type
  PStepWindow = ^TStepWindow;
  TStepWindow = object(TWindow)
    constructor Init(P: PWindowsObject; AName: PChar);
    procedure SetUpWindow; virtual;
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
      virtual;
    procedure WMLButtonDown(var Msg: TMessage);
      virtual wm_First + wm_LButtonDown;
  end;

  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

constructor TStepWindow.Init(P: PWindowsObject; AName: PChar);
begin
  inherited Init(P, AName);
  Attr.Style := ws_Visible or WS_PopUp;
  { Attr.H := 200;
    Attr.W := 200; }
end;

procedure TStepWindow.SetUpWindow;
begin
  inherited SetUpWindow;
  PostMessage(HWindow, WM_SYSCOMMAND, SC_MAXIMIZE, 0);
end;

procedure TStepWindow.WMLButtonDown(var Msg: TMessage);
begin
  CloseWindow;
end;
```

```
procedure TStepWindow.Paint(PaintDC: HDC;
                           var PaintInfo: TPaintStruct);
var
  S: array[0..100] of Char;
begin
  StrCopy(S, 'Click on Window to close it');
  TextOut(PaintDC, 10, 10, S, StrLen(S));
end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PStepWindow, Init(nil, 'Steps'));
end;

var
  MyApp: TMyApplication;

begin
  MyApp.Init('Steps');
  MyApp.Run;
  MyApp.Done;
end.
```

Reference:

7/16/98 4:33:46 PM

Last Modified: 01-SEP-99

## Article #15516: Uninstalling Delphi

Technical Information Database

TI516D.txt - Uninstalling Delphi

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

Uninstalling Delphi (16 bit)

Contents

Introduction

- A) Deleting Directories
- B) Deleting the Program Group
- C) Removing .INI files
- D) Editing WIN.INI
- E) Editing WINHELP.INI
- F) Editing your AUTOEXEC.BAT
- G) Removing files placed in WINDOWS and WINDOWS\SYSTEM directories

Introduction

This document describes the process of "uninstalling" Delphi from your system. It is a good idea to have a back up of your system files, due to the fact that you will need to use the 'delete' command.

This document assumes the product was installed to C: drive, with default directories and full (default) installation settings. Any deviations from the default settings may require adjustments/changes to the instructions below.

A) Deleting Directories

If you have any program that uses IDAPI, please disregard any mention of deleting it or its directory below.

If you wish to delete directories using DOS commands, see section 1 below. To use Windows, see section 2.

- 1) To erase directories from your hard drive on a system with DOS 6.0 or greater, you need to be in the root directory (i.e. C:\> ) and type DELTREE

and

hit return(Example: C:\>deltree delphi). Answer Yes when it prompts to delete all subdirectories (see below for the list of directories to delete). For versions of DOS lower than 6.0, you need to go into each directory and type DEL \*.\* until all directories are empty. Then DOS will allow you to delete the directory names with the command RD

- 2) From Windows, go to File Manager. Using File Manager,

select the

folder. Then hit the Delete key.

Answer Yes to All when prompted to delete subdirectories.

The directories you need to delete are:

C:\Delphi	C:\Rptsmith
C:\Rs_run	C:\Dbd
C:\Iblocal	C:\Idapi
* C:\Informix	

\* - Client/ Server version only

## B) Deleting the Program Group

To delete the program group 'Delphi' in Windows 3.1 and Windows NT, highlight the group without opening it, and press the Delete key. Under Windows 95, the folders will have been removed by deleting the directories in the previous section.

### C) Removing .INI files

Delete the following .INI files from your \WINDOWS directory  
(CAUTION: if you have any other Borland products or any programs that use ODBC, back up the .INI files listed below before deleting them. Then test all other programs to insure they run normally!):

```
DELPHI.INI
BORHELP.INI
DBD.INI
INTERBAS.INI
* ODBC.INI
* ODBCINST.INI
* ODBCISAM.INI
RPTSMITH.INI
RS_RUN.INI
RS_SQLIF.INI
VSL.INI
MULTIHELP.INI
```

\* - files used by ODBC applications

### D) Editing WIN.INI

Make a backup of your WIN.INI file. Then, in your \WINDOWS directory, remove the following sections from your WIN.INI file (CAUTION: If any other program uses Paradox Engine, IDAPI, or Database Desktop, use discretion in deleting these lines!):

```
[DDE Servers]
DBD=C:\DBD\DBD
```

```
[DBD]
WORKDIR=C:\DBD
PRIVDIR=C:\DBD\DBDPRIV
```

```
[Interbase]
RootDirectory=C:\IBLOCAL
```

```
* [IDAPI]
DLLPATH=C:\IDAPI
CONFIGFILE01=C:\IDAPI\IDAPI.CFG
```

```
* [Borland Language Drivers]
LDPath=C:\IDAPI\LANGDRV
```

\* - DO NOT delete these if other programs  
use IDAPI

Remember to save your new WIN.INI.

#### E) Editing WINHELP.INI

Remove the following lines from WINHELP.INI located in  
your  
WINDOWS directory:

```
BDECFG.HLP=C:\IDAPI
DBD.HLP=C:\DBD
DELPHI.HLP=C:\DELPHI\BIN
WINAPI.HLP=C:\DELPHI\BIN
CWG.HLP=C:\DELPHI\BIN
CWH.HLP=C:\DELPHI\BIN
LOCALSQL.HLP=C:\DELPHI\BIN
VQB.HLP=C:\DELPHI\BIN
SQLREF.HLP=C:\IBLOCAL\BIN
WISQL.HLP=C:\IBLOCAL\BIN
RPTSMITH.HLP=C:\rptsmith
RS_DD.HLP=C:\rptsmith
SBL.HLP=C:\rptsmith
RS_RUN.HLP=C:\rptsmith
RCEXPRT.HLP=C:\DELPHI\RCEXPRT
DRVDBASE.HLP=C:\WINDOWS\SYSTEM
DRVPARDX.HLP=C:\WINDOWS\SYSTEM
ODBCINST.HLP=C:\WINDOWS\SYSTEM
```

## F) Editing your AUTOEXEC.BAT

First, make a backup of your AUTOEXEC.BAT file. Then edit your AUTOEXEC.BAT and remove C:\IBLOCAL\BIN; and C:\IDAPI; from your PATH statement if they are present.

(Example:

```
(OLD)  PATH = C:\;C:\WINDOWS;C:\DOS;C:\IDAPI;C:\IBLOCAL\BIN;
```

```
(NEW)  PATH = C:\;C:\WINDOWS;C:\DOS;)
```

Remember to save your new AUTOEXEC.BAT.

## G) Removing files from your WINDOWS and WINDOWS\SYSTEM directories

Please note: These files are listed for reference ONLY. It IS NOT recommended that you delete these files. Deletion of certain files used by other programs WILL cause your system to crash.

CAUTION!!!!CAUTION!!!!CAUTION!!!!CAUTION!!!!CAUTION!!!!  
CAUTION!!!!

Many files shown in this section are shared by other programs. Deleting them may cause other programs not to work INCLUDING WINDOWS. If you wish to remove files from these directories, please do so ONLY at your own discretion. One possible method would be to delete a file, and run ALL other programs on your system to make sure they work. Copies of all files listed below are located on the Delphi CD in the runimage directory. You may copy files from the CD if one that is required is inadvertently deleted. Please be careful.

CAUTION!!!!CAUTION!!!!CAUTION!!!!CAUTION!!!!CAUTION!!!!  
CAUTION!!!!

Files placed or updated in your \WINDOWS directory by  
Delphi:

DELPHI.CBT	M3OPEN.DLL
M3OPEN.EXE	MBW.EXE
MFTP.EXE	MHPARPA.DLL
MNETONE.EXE	MNOVLWP.DLL
MPATHWAY.DLL	MPCNFS.EXE
MPCNFS2.EXE	MPCNFS4.DLL
MSOCKLIB.DLL	MVWASYNC.EXE
MWINTCP.EXE	WINSOCK.DLL

Files placed or updated in your \WINDOWS\SYSTEM  
directory by  
Delphi:

BIBAS04.DLL	BIFLT04.DLL
BIMDS04.DLL	BIPDX04.DLL
BIPDX04.HLP	BIUTL04.DLL
BIVBX11.DLL	BLBAS04.DLL
BLINT04.DLL	BLINT04.HLP
BLMDS04.DLL	BLODBC.LIC
BLUTL04.DLL	BOLE16D.DLL
BTRV110.DLL	BWCC.DLL
CHART2FX.VBX	COMPOBJ.DLL
* COREWIN.DLL	CTL3D.DLL
CTL3DV2.DLL	* DBNMP3.DLL
DRVACCSS.HLP	DRVBTRV.HLP
DRVDBASE.HLP	DRVEXCEL.HLP
DRVFOX.HLP	DRVTEXT.HLP
GAUGE.VBX	* LDLLSQLW.DLL
MSJETDSP.DLL	MULTIHLP.DLL
NWCALLS.DLL	NWIPXSPX.DLL
* NWNETAPI.DLL	ODBC.DLL
ODBCADM.EXE	ODBCCURS.DLL
ODBCINST.DLL	ODBCINST.HLP
OLE2.DLL	OLE2.REG
OLE2CONV.DLL	OLE2DISP.DLL
OLE2NLS.DLL	OLE2PROX.DLL
* ORA6WIN.DLL	* ORA7WIN.DLL
PICT.VBX	PXENGCFG.EXE



PXENGWIN.DLL	QEBI.LIC
RED110.DLL	SIMADMIN.DLL
SIMBA.DLL	STDOLE.TLB
STORAGE.DLL	STRESS.DLL
SWITCH.VBX	TXTISAM.DLL
TYPELIB.DLL	* W3DBLIB.DLL
XBS110.DLL	XLSISAM.DLL

\* - Client/ Server version only

All .BOR files are backup files that Delphi renamed and replaced with updated or modified files, you may use these files in comparison to the files you have changed. If you have installed any software after the .BOR file was created or the .BOR file is the same as the file it replaced, the .BOR file can be deleted.

Reference:

3/30/99 3:47:03 PM

Last Modified: 01-SEP-99

## Article #15521: Getting / Setting diskette serial # under Windows.

Technical Information Database

TI521D.txt Getting / Setting diskette serial # under Windows.

Category :General Programming

Platform :All

Product :Pascal All

### Description:

(\*

This example demonstrates how to get/set the volume serial number of a diskette under Windows or DOS Protected Mode. This sample demonstrates two important interrupts: Int \$21, service \$69 (Get/Set media info) and Int \$31 (simulate real mode interrupt from DPMI).

NOTE: To use the following code under DPMI instead of Windows, replace all of the occurrences of "WinProcs" with "WinAPI" in the USES clauses.

\*)

```
unit MediaID;  
{ Use this unit in your program }
```

### Interface

### Type

```
PMediaID = ^TMediaID;  
TMediaID = Record  
  InfoLvl:   Word;  
  SerialNum: Longint;  
  VolLabel:  Array[0..10] of Char;  
  FileSys:   Array [0..7] of Char;  
end;
```

```
TRMCS = record      { Real mode call structure }  
  DI, SI, BP, Reserved, BX, DX, CX, AX : Longint;  
  Flags, ES, DS, FS, GS, IP, SP, SS: Word;  
end;
```

```
FUNCTION GetMediaInfo( Drive: WORD; MID: PMediaID ): Boolean;
```

```

FUNCTION SetMediaInfo( Drive: WORD; MID: PMediaID ): Boolean;
  { MID should be a real mode pointer, that is, segment:offset }
  { Allocate memory for MID pointer using GlobalDOSAlloc }
  { Drive: 0=default, 1=A, 2=B, etc. }

```

## Implementation

```

uses WinAPI;

```

```

var

```

```

    R: TRMCS;

```

```

function GetMediaInfo( Drive: WORD; MID: PMediaID ): Boolean;

```

```

var

```

```

    ErrorFlag : byte;

```

```

begin

```

```

    GetMediaInfo := True;

```

```

    R.BX := Drive;    { 0 = Default, 1 = A, 2 = B, etc }

```

```

    R.DS := HiWord(Longint(MID)); { Segment (not selector) of MID }

```

```

    R.DX := LoWord(Longint(MID)); { Offset of MID }

```

```

    R.AX := $6900;    { Real mode service $69, subfunc 0 }

```

```

asm

```

```

    mov bx, 0021h { set flags to $00, Real mode interrupt $21 }

```

```

    mov cx, 0     { copy 0 words from protected mode stack }

```

```

    mov ax, seg R

```

```

    mov es, ax    { selector of real mode call structure }

```

```

    mov di, offset R { offset of real mode call structure }

```

```

    mov ax, 0300h { DPMI simulate real mode interrupt }

```

```

    int 31h

```

```

    pushf        { push flags onto stack }

```

```

    pop  bx      { pop flags into bx }

```

```

    test bx, 0   { error check: is CF flag 0? }

```

```

    je

```

```

done      { if so, then there was no error }

```

```

error:

```

```

    mov  ErrorFlag, 1 { set error flag }

```

```

done:

```

```

end;

```

```

if ErrorFlag = 1 then

```

```

    GetMediaInfo := False; { return False on error }

```

```

end;

function SetMediaInfo( Drive: WORD; MID: PMediaID ): Boolean;
var
  ErrorFlag : byte;
begin
  SetMediaInfo := True;
  R.BX := Drive;    { 0 = Default, 1 = A, 2 = B, etc }
  R.DS := HiWord(Longint(MID)); { Segment of MID }
  R.DX := LoWord(Longint(MID)); { Offset of MID }
  R.AX := $6901;    { Real mode service $69, subfunc 1 }
  asm
    mov bx, 0021h { Real mode interrupt $21 }
    mov cx, 0
    mov ax, seg R
    mov es, ax    { real mode selector call structure }
    mov di, offset R { offset of real mode call structure }
    mov ax, 0300h { DPMI simulate real mode interrupt }
    int 31h
    pushf        { push flags onto stack }
    pop  bx      { pop flags into bx }
    test bx, 0   { error check: is CF flag 0? }
    je
  done          { if so, then there was no error }

  error:
    mov  ErrorFlag, 1 { set error flag }

done:
  end;
  if ErrorFlag = 1 then
    SetMediaInfo := False; { return False on error }
  end;

begin
end.  {*** End of MediaID unit ***}

```

Here is a small sample program that demonstrates how to use the MediaID unit:

```

program DiskInfo;

```

```
uses MediaID, WinProcs, WinCRT;

var
  SReal, SProt : PMediaId;
  LongHandle : Longint;
begin
  { allocate some lower memory }
  LongHandle := GlobalDOSAlloc(SizeOf(TMediaID));
  { real mode pointer }
  SReal := PMediaID(Ptr(HiWord(LongHandle), 0));
  { protected mode pointer }
  SProt := PMediaID(Ptr(LoWord(LongHandle), 0));
  GetMediaInfo(3, SReal);      { call function }
  writeln(SProt^.SerialNum);  { write serial # }
  GlobalDOSFree(LoWord(LongHandle)); { free allocated memory }
end. {*** End of DiskInfo program ***}
```

Reference:

7/16/98 4:33:46 PM

Last Modified: 01-SEP-99

## Article #15526: Installing Delphi on a Network

Technical Information Database

TI526D.txt - Installing Delphi on a Network

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

----Installing Delphi to a network drive----

Note:

Delphi was never intended to be run from a network, and Borland makes no claims in it's ability to do so. This information is provided solely as a starting point and is NOT intended to be a cookbook.

This TI is assuming a basic Windows 3.1 or 3.11 installation. if you are using another operating system, appropriate adjustments will, of course, need to be made.

\*\*\*\*\*

1. Install Delphi to the Network from any Workstation, but be sure to check the "Windows Lan" option if Windows is installed on a server and it's directories are read only.
2. Edit DELPHI.INI in the windows directory.  
-enter:  
    [Globals]  
    PrivateDir = your local directory  
-add:  
    [Library]  
    SearchPath = Netdrive:\DELPHI\LIB  
    ComponentLibrary= your local directory\COMPLIB.DCL

This will allow each user to have his own private options, private DFM file, as well as a private Component Library. If a public component library is desired, the ComponentLibray line

may be left out.

3. Copy COMPLIB.DCL and DELPHI.DMT from the DELPHI\BIN directory to the local private directories.
4. Copy DELPHI.INI to all machines to be running Delphi.

If Windows is on the network, everyone will need to set up their local directories identically, since it will only have that one INI file to work from.

5. If the WINSYS option was selected at install time, there was a DELPHI\WINSYS directory created. There is a bug in the install engine, and not all of the needed files get included. Copy the files from: CDROM:\RUNIMAGE\WINDOWS and CDROM:\RUNIMAGE\WINDOWS\SYSTEM into the WINSYS directory. Then this directory will need to be included in the path. If you wish to copy these files into the WINDOWS\SYSTEM directory, be sure to back up those directories first, in case of file collisions. Care should be taken to not overwrite newer versions of these files. You may find the DOS program REPLACE.EXE useful in accomplishing this.
6. If there was no WINSYS directory created (Step 5), Copy CD:RUNIMAGE\WINDOWS and CD:\RUNIMAGE\WINODWS\SYSTEM directories to all Workstations. Put these files in a separate directory and include it in the path.(This will make uninstalling much easier.)
7. Copy the group file created in the windows directory to all other stations to be using Delphi.
8. Use Program Manager to create the group on these machines.
9. Modify the path on all workstations that will run Delphi as follows:

```
PATH=C:\IBLOCAL\BIN;C:\DELPHI\BIN
```

All users must load the DOS command SHARE(Assuming windows 3.1) before using Delphi.

10. The following line should be added to each user's AUTOEXEC.BAT:  
  
SHARE /F:4096 /L:40

11. Modify the WIN.INI on all machines (assuming a local windows installation,) adding the following:

```
[IDAPI]
DLLPATH=X:\IDAPI;C:\IDAPI
CONFIGFILE01=X:\IDAPI\IDAPI.CFG
```

```
[Borland Language Drivers]
LDPath=X\IDAPI\LANGDRV
```

```
[BWCC]
BitmapLibrary=BWCC.DLL
```

```
[Interbase]
RootDirectory=X:\IBLOCAL
```

```
[Paradox Engine]
UserName=PxEEngine
NetNamePath=X:\
MaxTables=64
RecBufs=64
MaxLocks=64
MaxFiles=64
SwapSize=64
NetNameFDSM=
```

```
[DDE Servers]
DBD=X:\DBD\DBD
```

```
[DBD]
WORKDIR=X:\DBD
PRIVDIR=C:\DBD\DBDPRIV
```

Where X is the Network Drive and path installed to.  
Note that PRIVDIR is set locally.

Note, Delphi will install numerous INI files to the Windows directory. These files can be copied over to other workstations only if the corresponding INI files do not already reside there. (They shouldn't) If this is the case you will have to manually attach these files to the ends of the existing ones.

The INI files that Delphi installs are:



RS\_SQLIF INI  
WINHELP INI  
MULTIHELP INI  
DELPHI INI  
ODBCINST INI  
ODBC INI  
RPTSMITH INI  
RS\_RUN INI  
ODBCISAM INI

The configuration described above should allow for the installed location to be set read only after it is installed, but this was only tested with a local installation of Windows.

Reference:

3/30/99 12:59:51 PM

Last Modified: 01-SEP-99

## Article #15531: Enabling a horizontal scroll bar for a list box.

Technical Information Database

TI531D.txt - Enabling a horizontal scroll bar for a list box.

Category :General Programming  
Platform :All Windows  
Product :All32Bit,

### Description:

To enable horizontal scroll bars in list boxes you have send the listbox a LB\_SETHORIZONTALTEXT message. The sample code illustrates how to do this in the OnCreate of a form. The form simply has one listbox on it and a button. Clicking on the button removes the horizontal scroll bar.

-- Source code

```
unit Unit1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs,  
StdCtrls;
```

```
type
```

```
TForm1 = class(TForm)
```

```
ListBox1: TListBox;
```

```
Button1: TButton;
```

```
procedure FormCreate(Sender: TObject);
```

```
procedure Button1Click(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
public
```

```
{ Public declarations }
```

```
end;
```

```
var
```

```
Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
var
```

```
    LongestLength: Integer;
```

```
    LongestString: String;
```

```
    PixelLength: Integer;
```

```
    i: Integer;
```

```
begin
```

```
    Listbox1.Items.Add('Line 1');
```

```
    Listbox1.Items.Add('Line 2 A very long string that is past the size  
of the listbox');
```

```
    Listbox1.Items.Add('Line 3');
```

```
    if Listbox1.Items.Count > 1 then
```

```
    begin
```

```
        LongestLength := 0;
```

```
        LongestString := '';
```

```
        // go through all the items to find the longest one  
        for i := 0 to Listbox1.Items.Count - 1 do
```

```
        begin
```

```
            if Length(Listbox1.Items[i]) > LongestLength then
```

```
            begin
```

```
                LongestString := Listbox1.Items[i];
```

```
                LongestLength := Length(Listbox1.Items[i]);
```

```
            end;
```

```
        end;
```

```
        // Now convert the longest string length to pixel size
```

```
        PixelLength := Listbox1.Canvas.TextWidth(LongestString);
```

```
        // Add a little extra to it to make it look nice
```

```
        PixelLength := PixelLength + Listbox1.Canvas.TextWidth('W');
```

```
        // Send an LB_SETHORIZONTALTEXT to the list box to apply
```

```
        // the horizontal scrollbar
```

```
        SendMessage(ListBox1.Handle, LB_SETHORIZONTALTEXT,  
                    PixelLength, 0);
```

```
    end;
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
{ This hides the listbox scrollbar by sending a
  string length of 0 message }
begin
  // Send a lb_SetHorizontalExtent to the list box with
  // 0 as the length of the string
  SendMessage(ListBox1.Handle, LB_SETHORIZONTALTEXT,
    0, 0);
end;

end.
```

Reference:

3/30/99 12:23:29 PM

Last Modified: 01-SEP-99

## Article #15536: Making a form selectable without the main form.

Technical Information Database

TI536D.txt Making a form selectable without the main form.

Category :VCL

Platform :All

Product :Delphi All

Description:

Q: How do I make it so that only the form I select comes to the top? (i.e. without the main form)

A: Try this in any secondary window that you DON'T want dragging the program along:

```
...
private {This goes in the form's type declaration.}
  { Private declarations }
  procedure CreateParams(VAR Params: TCreateParams); override;
...

procedure TForm2.CreateParams(VAR Params: TCreateParams);
begin
  Inherited CreateParams(Params);
  Params.WndParent := GetDesktopWindow;
end;
```

By setting the form's parent window handle to the desktop, you remove the link that would normally force the whole application to come to the top when this form comes to the top.

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15541: How to get a file's date and time stamp.

Technical Information Database

TI541D.txt     How to get a file's date and time stamp.

Category     :General Programming

Platform     :All

Product      :Delphi   All

Description:

Q:   How do I get a file's date and time stamp?

A:

```
function GetFileDate(TheFileName: string): string;
var
  FHandle: integer;
begin
  FHandle := FileOpen(TheFileName, 0);
  try
    Result := DateTimeToStr(FileDateToDateTime(FileGetDate(FHandle)));
  finally
    FileClose(FHandle);
  end;
end;
```

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15546: Lists the virtual key values

Technical Information Database

TI546D.txt Lists the virtual key values

Category :Windows API

Platform :All

Product :Delphi All

Description:

Q: What are the values for the virtual keys?

A:

vk_LButton	=	\$01;	
vk_RButton	=	\$02;	
vk_Cancel	=	\$03;	
vk_MButton	=	\$04;	{ NOT contiguous with L & RBUTTON }
vk_Back	=	\$08;	
vk_Tab	=	\$09;	
vk_Clear	=	\$0C;	
vk_Return	=	\$0D;	
vk_Shift	=	\$10;	
vk_Control	=	\$11;	
vk_Menu	=	\$12;	
vk_Pause	=	\$13;	
vk_Capital	=	\$14;	
vk_Escape	=	\$1B;	
vk_Space	=	\$20;	
vk_Prior	=	\$21;	
vk_Next	=	\$22;	
vk_End	=	\$23;	
vk_Home	=	\$24;	
vk_Left	=	\$25;	
vk_Up	=	\$26;	
vk_Right	=	\$27;	
vk_Down	=	\$28;	
vk_Select	=	\$29;	
vk_Print	=	\$2A;	
vk_Execute	=	\$2B;	
vk_SnapShot	=	\$2C;	

```
{ vk_Copy      = $2C not used by keyboards }
  vk_Insert    = $2D;
  vk_Delete    = $2E;
  vk_Help      = $2F;
{ vk_A thru vk_Z are the same as their ASCII equivalents: 'A' thru
'Z' }
{ vk_0 thru vk_9 are the same as their ASCII equivalents: '0' thru
'9' }
```

```
vk_NumPad0    = $60;
vk_NumPad1    = $61;
vk_NumPad2    = $62;
vk_NumPad3    = $63;
vk_NumPad4    = $64;
vk_NumPad5    = $65;
vk_NumPad6    = $66;
vk_NumPad7    = $67;
vk_NumPad8    = $68;
vk_NumPad9    = $69;
vk_Multiply   = $6A;
vk_Add        = $6B;
vk_Separator  = $6C;
vk_Subtract   = $6D;
vk_Decimal    = $6E;
vk_Divide     = $6F;
vk_F1         = $70;
vk_F2         = $71;
vk_F3         = $72;
vk_F4         = $73;
vk_F5         = $74;
```

```
vk_F6         = $75;
vk_F7         = $76;
vk_F8         = $77;
vk_F9         = $78;
vk_F10        = $79;
vk_F11        = $7A;
vk_F12        = $7B;
vk_F13        = $7C;
vk_F14        = $7D;
vk_F15        = $7E;
vk_F16        = $7F;
vk_F17        = $80;
vk_F18        = $81;
vk_F19        = $82;
```



vk\_F20 = \$83;  
vk\_F21 = \$84;  
vk\_F22 = \$85;  
vk\_F23 = \$86;  
vk\_F24 = \$87;  
vk\_NumLock = \$90;  
vk\_Scroll = \$91;

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15551: How to paint the form with a bitmap.

Technical Information Database

TI551D.txt     How to paint the form with a bitmap.

Category     :VCL

Platform     :All

Product      :Delphi   All

Description:

Q:   How do I paint the background of my form with a bitmap as tiles?

A:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs;
```

```
type
```

```
  TForm1 = class(TForm)  
    procedure FormCreate(Sender: TObject);  
    procedure FormPaint(Sender: TObject);  
  private  
    { Private declarations }  
  public  
    { Public declarations }  
  end;
```

```
var
```

```
  Form1: TForm1;  
  Bitmap: TBitmap;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);  
begin
```

```
Bitmap := TBitmap.Create;
Bitmap.LoadFromFile('C:\WINDOWS\cars.BMP');
end;

procedure TForm1.FormPaint(Sender: TObject);
var
  X, Y, W, H: LongInt;
begin
  with Bitmap do begin
    W := Width;
    H := Height;
  end;
  Y := 0;
  while Y < Height do begin
    X := 0;
    while X < Width do begin
      Canvas.Draw(X, Y, Bitmap);
      Inc(X, W);
    end;
    Inc(Y, H);
  end;
end;

end.
```

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15556: How to associate a string with a component.

Technical Information Database

TI556D.txt     How to associate a string with a component.

Category     :VCL

Platform     :All

Product      :Delphi   All

Description:

Q: Is there a way to associate a string with each component?

A: Since the Tag property is a longint, you can type cast it as a Pointer or PChar. So, you can basically store a pointer to a record by using the Tag property.

Note: You're not going to be able to store the string, or pointer rather, at design time. This is something you'll have to do at run time. Take a look at this example:

```
var
  i: integer;
begin
  for i := 0 to ComponentCount - 1 do

    if Components[i] is TEdit then
      Components[i].Tag := LongInt(NewStr('Hello '+IntToStr(i)));
end;
```

Here, I loop through the components on the form. If the component is a TEdit, I assign a pointer to a string to its Tag property. The NewStr function returns a PString (pointer to a string). A pointer is basically the same as a longint or better, occupies the same number of bytes in memory. Therefore, you can type cast the return value of NewStr as a LongInt and store it in the Tag property of the TEdit component. Keep in mind that this could have been a pointer to an entire record. Now I'll use that value:

```
var
  i: integer;
begin
```

```

for i := 0 to ComponentCount - 1 do
  if Components[i] is TEdit then begin
    TEdit(Components[i]).Text := PString(Components[i].Tag)^;
    DisposeStr(PString(Components[i].Tag));
  end;
end;

```

Here, again I loop through the components and work on only the TEdits. This time, I extract the value of the component's Tag property by typecasting it as a PString (Pointer to a string) and assigning that value to the TEdit's Text property. Of course, I must dereference it with the caret (^) symbol. Once I do that, I dispose of the string stored in the edit component. Important note: if you store anything in the TEdit's Tag property as a pointer, you are responsible for disposing of it also.

FYI, Since Delphi objects are really pointers to class instances, you can also store objects in the Tag property. As long as you remember to Free them.

Three methods spring to mind to use Tags to access strings that persist from app to app.

1. If your strings stay the same forever, create a string resource in Resource Workshop (or equiv) and use the Tags as indexes into your string resource.
2. Use TIniFile and create a section for your strings, and give each string a name with number so that your ini file has a section like this:

```

[strings]
string1=Aristotle
string2=Plato
string3=Well this is Delphi, after all

```

Then you can fetch them back out this way:

```

var s1: string;
...
s1 := IniFile1.ReadString('strings', 'string'+IntToStr(Tag), '');

```

3. Put your strings into a file, with each followed by a carriage return. Read them into a TStringList. Then your Tags

become an index into this stringlist:

```
StringList1.LoadFromFile('slist.txt');  
...  
s1 := StringList1[Tag];
```

Given the way Delphi is set up, I think the inifile method is easiest.

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15561: How close a file opened from a Delphi DLL in VB.

Technical Information Database

TI561D.txt      How close a file opened from a Delphi DLL in VB.

Category      :Application Interop

Platform      :All

Product       :Delphi 1.0

Description:

Q: How do I close a file that was opened in a DLL (Delphi made) and called from VB?

A: This is a known problem. It comes from the fact that VB closes the 5 DOS standard handles (0..4) at startup. So the open file routine will reuse one of these handles to open the first disk file. That is not a problem in using the file, but the Pascal Close routine has a build-in safety feature: it refuses to close a file that has one of the standard handles! That is a Good Thing under DOS but screws up the works in your situation since the file opened by the DLL is never closed, not even when the DLL goes down! VC++ is obviously less restricted and will close a standard handle.

You can fix this problem yourself. Instead of using the Pascal Close/CloseFile routine to close the file in the DLL, use one of these:

```
Procedure ReallyCloseFileVar(Var F); Assembler;
```

```
{ F should be a file type }
```

```
Asm
```

```
  les  bx, F                { store F in es:bx }
  mov  bx, word ptr es:[bx] { store handle in bx }
  mov  ah, $3E              { function 3Eh = close file }
  call Dos3Call             { execute int 21h }
```

```
End;
```

```
Procedure ReallyCloseFileHandle(FileHandle: word); assembler;
```

```
{ FileHandle is the DOS file handle }
```

```
asm
```

```
  mov  bx, Handle { store handle in bx }
  mov  ah, $3E    { function 3Eh = close file }
```

```
    call DOS3Call    { execute int 21h }  
end;
```

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99



## Article #15566: Changing the border color in DOS text mode.

Technical Information Database

TI566D.txt Changing the border color in DOS text mode.

Category :General Programming

Platform :All

Product :Pascal All

### Description:

Ever wonder how to change the color of the border around your screen in DOS text mode? It's actually very simple - just call DOS interrupt \$10, function \$0B. This sample code illustrates how. You just drop the BorderColor procedure into any of your DOS text mode programs.

Program ChBorder;

```
procedure BorderColor(NewColor: byte); assembler;
{ simply call this procedure and pass it the number }
{ of the color to which you would like the text-mode }
{ border to change. }
asm
    mov ah, 0Bh
    mov bh, 00h
    mov bl, NewColor
    int 10h
end;

begin
    BorderColor(2);
end.
```

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15571: How to readln longer than 255 chars.

Technical Information Database

TI571D.txt      How to readln longer than 255 chars.

Category      :General Programming

Platform      :All

Product       :Delphi 1.0

Description:

Q: How can I readln() from a file when the lines are longer than 255 bytes?

A: ReadLn will accept an array [0..something] of Char as buffer to put the read characters in and it will make a proper zero-terminated char out of them. The only limitation is this: the compiler needs to be able to figure out the size of the buffer at compile time, which makes the use of a variable declared as PChar and allocated at run-time impossible.

Workaround:

Type

```
{use longest line you may encounter here}
```

```
TLine = Array [0..1024] of Char;
```

```
PLine = ^TLine;
```

Var

```
pBuf: PLine;
```

```
...
```

```
New( pBuf );
```

```
...
```

```
ReadLn( F, pBuf^ );
```

To pass pBuf to functions that take a parameter of type PChar, use a typecast like PChar( pBuf ).

Note: you can use a variable declared as of type TLine or an equivalent array of char directly, of course, but I tend to allocate anything larger than 4 bytes on the heap...

Reference :

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15576: A VERY short primer on dynamic memory allocation.

Technical Information Database

TI576D.txt     A VERY short primer on dynamic memory allocation.  
Category     :General Programming  
Platform     :All  
Product     :Delphi 1.0

Description:

Q: How do I reduce the amount of memory taken from the data segment? (or How do I allocate memory dynamically?)

A:

Let's say your data structure looks like this:

```
type
  TMyStructure = record
    Name: String[40];
    Data: array[0..4095] of Integer;
  end;
```

That's too large to be allocated globally, so instead of declaring a global variable,

```
var
  MyData: TMyStructure;
```

you declare a pointer type,

```
type
  PMyStructure = ^TMyStructure;
```

and a variable of that type,

```
var
  MyDataPtr: PMyStructure;
```

Such a pointer consumes only four bytes of the data segment.

Before you can use the data structure, you have to allocate it

on the heap:

```
New(MyDataPtr);
```

and now you can access it just like you would global data. The only difference is that you have to use the caret operator to dereference the pointer:

```
MyDataPtr^.Name := 'Lloyd Linklater';  
MyDataPtr^.Data[0] := 12345;
```

Finally, after you're done using the memory, you deallocate it:

```
Dispose(MyDataPtr);
```

Reference:

7/16/98 4:33:47 PM

Last Modified: 01-SEP-99

## Article #15578: HEAP SPACE STILL CREATED WITH HEAP MAX 0

Technical Notes Database

TN578D.txt      HEAP SPACE STILL CREATED WITH HEAP MAX 0  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 5.5

### Description:

- Q. Compiling to disk with my heap maximum set to 0 still creates heap space when running .EXE file. Why?
- A. This has to do with the way DOS allocates memory for .EXE files in chunks of 512 bytes, rather than in chunks of 16 bytes. As a result of DOS's allocation, you always get 0-496 bytes more than you request.

### Reference:

7/16/98 4:35:33 PM

Last Modified: 01-SEP-99

## Article #15582: How to use array of const.

Technical Information Database

TI582D.txt     How to use array of const.  
Category     :Object Pascal  
Platform     :All  
Product      :Delphi All

Description:

Q: How do I use "array of const"?

A: An array of const is in fact an open array of TVarRec (a predeclared Delphi type you can look up in the online help). So the following is Object Pascal psuedocode for the general battle plan:

```
procedure AddStuff( Const A: Array of Const );
Var i: Integer;
Begin
  For i:= Low(A) to High(A) Do
    With A[i] Do
      Case VType of
        vtExtended: Begin
          { add real number, all real formats are converted to
            extended automatically }
          End;
        vtInteger: Begin

          { add integer number, all integer formats are converted
            to LongInt automatically }
          End;
        vtObject: Begin
          If VObject Is DArray Then
            With DArray( VObject ) Do Begin
              { add array of doubles }
            End
          Else If VObject Is IArray Then
            With IArray( VObject ) Do Begin
              { add array of integers }
            End;
          End;
        End;
      End;
```

```
End; { Case }  
End; { AddStuff }
```

For further information see "open arrays" in the on-line help.

Reference:

7/16/98 4:33:48 PM

Last Modified: 01-SEP-99



## Article #15587: Making your own hotkeys.

Technical Information Database

TI587D.txt Making your own hotkeys.

Category :VCL

Platform :All

Product :Delphi All

Description:

Q: How can I trap for my own hotkeys?

A: First: set the form's KeyPreview := true;

Then, you do something like this:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (ssCtrl in Shift) and (chr(Key) in ['A', 'a']) then
    ShowMessage('Ctrl-A');
end;
```

Reference:

7/16/98 4:33:48 PM

Last Modified: 01-SEP-99

## Article #15589: MEMORY CONTROL BLOCKS

Technical Notes Database

TN589D.txt      MEMORY CONTROL BLOCKS

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. How many memory control blocks does a Turbo Pascal program use?

A. Just one.

Reference:

7/16/98 4:35:33 PM

Last Modified: 01-SEP-99

## Article #15593: How to scroll your form with pgUP and pgDn.

Technical Information Database

TI593D.txt     How to scroll your form with pgUP and pgDn.

Category     :VCL

Platform     :All

Product     :Delphi   All

### Description:

Q.     How can you do scrolling functions in a TForm component using keyboard commands? For example, scrolling up and down when a PgUp or PgDown is pressed. Is there some simple way to do this or does it have to be programmed by capturing the keystrokes and manually responding to them?

A.     Form scrolling is accomplished by modifying the VertScrollbar or HorzScrollbar Position properties of the form. The following code demonstrates how to do this:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
const
  PageDelta = 10;
begin
  With VertScrollbar do
    if Key = VK_NEXT then
      Position := Position + PageDelta
    else if Key = VK_PRIOR then
      Position := Position - PageDelta;
end;
```

### Reference:

7/16/98 4:33:48 PM

Last Modified: 01-SEP-99

## Article #15595: ALLOCATING AND RECLAIMING MEMORY USING DOS

Technical Notes Database

TN595D.txt      ALLOCATING AND RECLAIMING MEMORY USING DOS  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

- Q. What Dos functions can I use to reclaim memory to Dos?  
A. You essentially must write your own GetMem and FreeMem routines. GetMem would make a call to Dos function \$48 and FreeMem would make a call to Dos function \$49. For more details on these functions please consult an IBM PC programmer's guide.

### Reference:

7/16/98 4:35:33 PM

Last Modified: 01-SEP-99

## Article #15599: Trapping 8087 floating point exceptions in Pascal.

Technical Information Database

TI599D.txt    Trapping 8087 floating point exceptions in Pascal.  
Category    :General Programming  
Platform    :All  
Product     :Pascal    All

### Description:

```
(*  
This unit allows you to selectively handle floating point  
errors - such a operations on NANs or infinity numbers -  
generated by the 8087 numeric coprocessor or emulation.  
This unit basically implements Pascal versions of C's  
_control87() and _stat87() functions and offers startup  
code for a user-defined int75h handler. For complete  
documentation on all of the constants in this unit, you  
should reference either the C documentation on _control87(),  
_stat87(), or signal(), or reference a 8087 programmer's guide.  
)
```

```
unit Except87;
```

```
{  
  NOTES: This unit is intended only for use on machines with an  
         80286 processor or higher.  
  
         If you intend to hook interrupt 75h while using this unit,  
         you should call Int75Startup as the first line in your  
         interrupt handler or very, very bad things will happen.  
}  
  
{ $IfNDef Windows }  
  
{ $N+, E+ }  
  
{ $Else }  
  
{ $N+ }
```

```
{ $EndIf }
```

```
interface
```

```
const
```

```
 { 387 Status Word format }  
 SW_INVALID      = $0001; { Invalid operation }  
 SW_DENORMAL     = $0002; { Denormalized operand }  
 SW_ZERODIVIDE   = $0004; { Zero divide }  
 SW_OVERFLOW     = $0008; { Overflow }  
 SW_UNDERFLOW   = $0010; { Underflow }  
 SW_INEXACT      = $0020; { Precision (Inexact result) }  
 SW_STACKFAULT   = $0040; { Stack fault }
```

```
 { 387 Control Word format }  
 MCW_EM          = $003f; { interrupt Exception Masks }  
   EM_INVALID    = $0001; { invalid }  
   EM_DENORMAL   = $0002; { denormal }  
   EM_ZERODIVIDE = $0004; { zero divide }  
   EM_OVERFLOW   = $0008; { overflow }  
   EM_UNDERFLOW  = $0010; { underflow }  
   EM_INEXACT    = $0020; { inexact (precision) }
```

```
 MCW_IC          = $1000; { Infinity Control }  
   IC_AFFINE     = $1000; { affine }  
   IC_PROJECTIVE = $0000; { projective }
```

```
 MCW_RC          = $0c00; { Rounding Control }  
   RC_CHOP       = $0c00; { chop }  
   RC_UP         = $0800; { up }  
   RC_DOWN       = $0400; { down }  
   RC_NEAR       = $0000; { near }
```

```
 MCW_PC          = $0300; { Precision Control }  
   PC_24         = $0000; { 24 bits }  
   PC_53         = $0200; { 53 bits }  
   PC_64         = $0300; { 64 bits }
```

```
function Control87(New, Mask: Word): Word;
```

```
function Status87: Word;
```

```
procedure Int75Startup; far;
```

```
implementation
```

```
procedure Int75Startup; assembler;
```

```
asm { this is the entry code for your int 75 handler }
    xor al, al { Clear BUSY latch}
    out 0F0h, al
    mov al, 20H { End-of-interrupt}
    out 0A0h, al
    out 20h, al
end;
```

```
function Control87(New, Mask: Word): Word; assembler;
```

```
var
    Control: Word;
asm
    fstcw Control
    mov ax, New
    mov bx, Mask
    and ax, bx
    not bx
    fwait
    mov dx, Control
    and dx, bx
    mov Control, ax
    fldcw Control
    xchg ax, dx
end;
```

```
function Status87: Word; assembler;
```

```
var
    Status: Word;
asm
    fstsw Status
    fwait
    mov ax, Status
    and ax, SW_INVALID+SW_ZERODIVIDE+SW_OVERFLOW+SW_UNDERFLOW+SW_INEXACT
end;
```

```
end.
```

```
(***** This is a test program for the Except87 unit *****)
```

```
program TestFP;
```

```
{ $N+ }
```

```
uses Except87, DOS;
```

```
var
  f: single;
  p: pointer;
  i: word;

procedure Int75Handler; interrupt;
begin
  Int75Startup;
  i := 1;
end;

begin
  i := 0;
  Control87(0, MCW_EM);
  GetIntVec($75, p);
  SetIntVec($75,
Int75Handler);
  f := 99999999999.99;
  f := f / 0.0;
  writeln('Coprocessor status: ', status87);
  SetIntVec($75, p);
  writeln('Did int75 fire?: ', i);
end.
```

Reference:

7/16/98 4:33:48 PM

Last Modified: 01-SEP-99



## Article #15601: EXEC - SETTING HEAP MEMORY

Technical Notes Database

TN601D.txt      EXEC - SETTING HEAP MEMORY  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. How do I set the heap memory when executing a child process?

A. Probably the best way to do this is through trial and error.

Set the heap maximum to some large value, and run the program.

If it runs out of memory, then divide the maximum by 2, and so on, until the parent and child programs have enough memory.

The same can be done with the heap minimum. Also, if you have access to CompuServe, you can download a routine that swaps the parent program between memory and disk.

### Reference:

7/16/98 4:35:33 PM

Last Modified: 01-SEP-99

## Article #15605: Drag a form without clicking the caption bar

Technical Information Database

TI605D.txt Drag a form without clicking the caption bar

Category :VCL

Platform :All

Product :Delphi All

Description:

{  
Q: How can I make a form move by clicking and dragging in the client area

instead of on the caption bar?

A: The easiest way to do this is to "fool" Windows into thinking that you're actually clicking on the caption bar of a form. Do this by handling the `wm_NCHitTest` windows message as shown in the sample

unit  
below.

TIP: If you want a captionless borderless window similar to a floating toolbar, set the Form's Caption to an empty string, disable all of the `BorderIcons`, and set the `BorderStyle` to `bsNone`.  
}

```
unit Dragmain;
```

```
interface
```

```
uses
```

```
SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
Forms, Dialogs, StdCtrls;
```

```
type
```

```
TForm1 = class(TForm)
```

```
  Button1: TButton;
```

```
  procedure Button1Click(Sender: TObject);
```

```
private
```

```
  procedure WMNCHitTest(var M: TWMNCHitTest); message wm_NCHitTest;
```

```

end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMNCHitTest(var M: TWMNCHitTest);
begin
  inherited;           { call the inherited message handler }
  if M.Result = htClient then { is the click in the client area? }
    M.Result := htCaption; { if so, make Windows think it's }
                          { on the caption bar. }
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
end;

end.

```

{ The text representation of the .DFM file is below:

```

object Form1: TForm1
  Left = 203
  Top = 94
  BorderIcons = []
  BorderStyle = bsNone
  ClientHeight = 273
  ClientWidth = 427
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 16
  object Button1: TButton
    Left = 160
    Top = 104
    Width = 89
    Height = 33
    Caption = 'Close'
  end
end

```

```
    TabOrder = 0
    OnClick = Button1Click
end
end
}
```

Reference:

7/16/98 4:33:48 PM

Last Modified: 01-SEP-99

## Article #15607: TAKING ADVANTAGE OF IEEE FLOATING-POINT TYPES

Technical Notes Database

TN607D.txt      TAKING ADVANTAGE OF IEEE FLOATING-POINT TYPES  
Category      :Pascal  
Platform      :All  
Product       :Turbo Pascal    5.0 5.5

### Description:

- Q. What is the best approach to taking advantage of the new IEEE floating-point types?
- A. The new IEEE floating-point types are available when you compile your program with `{N+}` and you have a math coprocessor; they are also available if you don't have a coprocessor, but specify `{N+,E+}`. The 8087 emulator has greater precision, but is significantly slower than the fast, 6-byte, software-only reals. When developing programs that will be compiled and run on machines without the 8087 coprocessor, consider the tradeoffs of speed (built-in reals) vs. precision (8087 hardware/emulation) and make the appropriate choice.

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15613: TRUNC - CONVERTING REAL NUMBERS INTO INTEGERS

Technical Notes Database

TN613D.txt TRUNC - CONVERTING REAL NUMBERS INTO INTEGERS

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. How do you convert real numbers into integers?

A. Use the Round or Trunc function and assign the result to an integer:

```
var i : integer;
begin
  i := round(12.43); i := trunc(22.22);
end.
```

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15617: Accessing more than 15 files from your DOS app.

Technical Information Database

TI617D.txt     Accessing more than 15 files from your DOS app.

Category     :General Programming

Platform     :All

Product      :Pascal   All

### Description:

DOS imposes a limit of 20 open files for any one program. DOS, friendly

as it is, uses 5 of these file handles internally - leaving your application with only 15 available file handles. Fortunately, the 15-file barrier is easily broken. Just add the GetMoreHandles() function to

your application, and call it in the beginning of your program, passing it

the number of handles you would like your application to have available.

The number you pass must be in the range of 20 to 255, and it must not exceed the number specified in the FILES= statement of your CONFIG.SYS.

uses DOS;

```
function GetMoreHandles(NumHandles: Word): Boolean;
```

```
\{
```

```
  NumHandles is the number of file handles you want your application
  to have available.  If an error occurs, function will return False,
  and any DOS error will be reported in the DOSError global variable.
  NOTE: NumHandles must be less than or equal to the number specified
  in the FILES= statement in your CONFIG.SYS.
```

```
\}
```

```
Var
```

```
  ReturnVal: Boolean;
```

```
begin
```

```
  ReturnVal := True;  \{ assume success \}
```

```
  \{ Check to make sure DOS version is greater than 3.3 \}
```

```
  if (((DOSVersion and $00FF) = 3) and ((DOSVersion and $FF00) < 3))
```

```
or
```

```
    ((DOSVersion and $00FF) < 3) then ReturnVal := False;
\{ Can't allocate more than 255 or less than 20 handles \}
if (NumHandles > 255) or (NumHandles < 20) then ReturnVal := False;
if ReturnVal then asm
    mov ax, 6700h           \{ function 67h \}
    mov bx, NumHandles     \{ requested handles in bx \}
    int 21h                \{ DOS call \}
    jae @@NotAnError       \{ CF set on error \}
    mov ReturnVal, 00h     \{ return false on error \}
    mov DOSError, ax       \{ ax contains the error code \}
    @@NotAnError:
end;
GetMoreHandles := ReturnVal;
end;
```

Reference:

7/16/98 4:33:49 PM

Last Modified: 29-MAR-00



## Article #15619: RANDOM RANDOMIZE - RANDOM NUMBER GENERATOR FORMULA

Technical Notes Database

TN619D.txt      RANDOM RANDOMIZE - RANDOM NUMBER GENERATOR FORMULA

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. What is the formula used by the random number generator in Turbo Pascal?

A. The random number generator in Turbo Pascal uses 32-bit arithmetic. It maintains a 32-bit 'seed', which is treated as an unsigned integer. Each call to Random changes the seed by the formula  $\text{seed} := \text{multiplier} * \text{seed} + 1 \text{ mod } 4294967296$ . Turbo Pascal uses the multiplier, 134775813. The modulus, 4294967296, is 2 to the power 32. The multiplier was chosen carefully for its nice randomness properties. The seed will take on every 32-bit integer before repeating.

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15623: How to Circumvent the "index not found" Exception.

Technical Information Database

TI623D.txt - How to Circumvent the "index not found" Exception.

Category :Database Programming  
Platform :All-32Bit  
Product :

### Description:

Q: How do I open a dBASE table without the required MDX file?  
I keep getting an "Index not found..." exception.

A: When you create a dBASE table with a production index (MDX), a special byte is set in the header of the DBF file. When you subsequently attempt to re-open the table, the dBASE driver will read that special byte, and if it is set, it will also attempt to open the MDX file. When the MDX file cannot be opened, an exception is raised.

To work around this problem, you need to reset the byte (byte 28 decimal) in the DBF file that causes the MDX dependency to zero.

The following unit is a simple example of how to handle the exception on the table open, reset the byte in the DBF file, and re-open the table.

```
unit Fixit;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, DB, DBTables, Grids, DBGrids;

type
  TForm1 = class(TForm)
    Table1: TTable;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;
```

```
private
  { Private declarations }
public
  { Public declarations }
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
const
  TheTableDir = 'c:\temp\';
  TheTableName = 'animals.dbf';
```

```
procedure RemoveMDXByte(dbFile: String);
{ This procedure accepts a DBF file as a parameter. It will patch }
{ the DBF header, so that it no longer requires the MDX file }
```

```
const
  Value: Byte = 0;
```

```
var
  F: File of byte;
```

```
begin
  AssignFile(F, dbFile);
  Reset(F);
  Seek(F, 28);
  Write(F, Value);
  CloseFile(F);
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
{ This procedure is called in response to a button click. It }
{ attempts to open a table, and, if it can't find the .MDX file, }
{ it patches the DBF file and re-execute the procedure to }
{ re-open the table without the MDX }
```

```
begin
  try
    { set the directory for the table }
    Table1.DatabaseName := TheTableDir;
    { set the table name }
    Table1.TableName := TheTableName;
    { attempt to open the table }
    Table1.Open;
```

```
except
  on E:EDBEngineError do
    { The following message indicates the MDX wasn't found: }
    if Pos('Index does not exist. File', E.Message)>0 then begin
      { Tell user what's going on. }
      MessageDlg('MDX file not found. Attempting to open
                  without index.', mtWarning, [mbOk], 0);
      { remove the MDX byte from the table header }
      RemoveMDXByte(TheTableDir + TheTableName);
      { Send the button a message to make it think it was }
      { pressed again. Doing so will cause this procedure to }
      { execute again, and the table will be opened without }
      { the MDX }
      PostMessage(Button1.Handle, cn_Command, bn_Clicked, 0);
    end;
  end;
end;

end.
```

Reference:

4/22/99 1:04:07 PM

Last Modified: 01-SEP-99

## Article #15625: ROUNDING WITH IEEE FLOATING-POINT

Technical Notes Database

TN625D.txt      ROUNDING WITH IEEE FLOATING-POINT

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

### Description:

Q. How does rounding work with IEEE floating-point numbers?

A. The 8087 math coprocessor uses a different method for rounding numbers than what you may be used to. In order to achieve a more even distribution of values, the 8087 uses a method sometimes called "Banker's Rounding." This method dictates that a number will always be rounded to the nearest EVEN number. Note that this is quite different than always rounding UP. Here are a couple of examples:

Round(0.5) = 0

Round(1.5) = 2

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15629: Getting version information from a Windows app.

Technical Information Database

TI629D.txt     Getting version information from a Windows app.  
Category     :General Programming  
Platform     :All  
Product      :Pascal    All

Description:

```
{  
  This sample program demonstrates how to use the  
  GetFileVersionInfo, GetFileVersionInfoSize, and  
  VerQueryValue API functions to obtain version  
  information from a Windows EXE or DLL containing  
  a version information resource.  
}
```

```
program GetVer;
```

```
uses WinCRT, WinTypes, WinProcs, Ver;
```

```
const  
  FileName: PChar = 'c:\windows\system\mmsystem.dll';
```

```
type  
  PLongInt = ^Longint;
```

```
var  
  VSize, VHandle: Longint;  
  Buffer: PChar;  
  Length, LangID, CharSetID: Word;  
  TranslationInfo, Result: Pointer;  
  StringFileInfo: Array[0..49] of Char;  
  LangCharSetIDArray: Array[1..2] of Word;
```

```
begin  
  { Get size of version info }  
  VSize := GetFileVersionInfoSize(FileName, VHandle);  
  { Allocate version info buffer }  
  GetMem(Buffer, VSize + 1);  
  { Get version info }
```

```

if GetFileVersionInfo(FileName, VHandle, VSize, Buffer) then
  { Get translation info for Language / CharSet IDs }
  if VerQueryValue(Buffer, '\VarFileInfo\Translation',
                    TranslationInfo, Length) then
begin
  LangCharSetIDArray[1] := LoWord(PLongint(TranslationInfo)^);
  LangCharSetIDArray[2] := HiWord(PLongint(TranslationInfo)^);
  { Get comments - this field is often blank }
  wvsPrintf(StringFileInfo, '\StringFileInfo\%04x%04x\Comments',
            LangCharSetIDArray);
  VerQueryValue(Buffer, StringFileInfo, Result, Length);
  Writeln('Comments: ', PChar(Result));
  { Get company name }
  wvsPrintf(StringFileInfo,
            '\StringFileInfo\%04x%04x\CompanyName',
            LangCharSetIDArray);
  VerQueryValue(Buffer, StringFileInfo, Result, Length);
  Writeln('Company: ', PChar(Result));
  { Get file description }
  wvsPrintf(StringFileInfo,
            '\StringFileInfo\%04x%04x\FileDescription',
            LangCharSetIDArray);
  VerQueryValue(Buffer, StringFileInfo, Result, Length);
  Writeln('File description: ', PChar(Result));
  { Get file version }
  wvsPrintf(StringFileInfo,
            '\StringFileInfo\%04x%04x\FileVersion',
            LangCharSetIDArray);
  VerQueryValue(Buffer, StringFileInfo, Result, Length);
  Writeln('File version: ', PChar(Result));
  { Get internal name }
  wvsPrintf(StringFileInfo,
            '\StringFileInfo\%04x%04x\InternalName',
            LangCharSetIDArray);
  VerQueryValue(Buffer, StringFileInfo, Result, Length);
  Writeln('Internal name: ', PChar(Result));
  { Get legal copyright info }
  wvsPrintf(StringFileInfo,
            '\StringFileInfo\%04x%04x\LegalCopyright',
            LangCharSetIDArray);
  VerQueryValue(Buffer, StringFileInfo, Result, Length);
  Writeln('Copyright: ', PChar(Result));
  { Get trademarks }
  wvsPrintf(StringFileInfo,
            '\StringFileInfo\%04x%04x\LegalTrademarks',

```

```

        LangCharSetIDArray);
VerQueryValue(Buffer, StringFileInfo, Result, Length);
Writeln('Trademarks: ', PChar(Result));
{ Get original filename }
wvsPrintf(StringFileInfo,
        '\StringFileInfo\%04x%04x
\OriginalFilename',
        LangCharSetIDArray);
VerQueryValue(Buffer, StringFileInfo, Result, Length);
Writeln('Original filename: ', PChar(Result));
{ Get private build info }
wvsPrintf(StringFileInfo,
        '\StringFileInfo\%04x%04x\PrivateBuild',
        LangCharSetIDArray);
VerQueryValue(Buffer, StringFileInfo, Result, Length);
Writeln('Private build info: ', PChar(Result));
{ Get product name }
wvsPrintf(StringFileInfo,
        '\StringFileInfo\%04x%04x\ProductName',
        LangCharSetIDArray);
VerQueryValue(Buffer, StringFileInfo, Result, Length);
Writeln('Product name: ', PChar(Result));
{ Get product version }
wvsPrintf(StringFileInfo,
        '\StringFileInfo\%04x%04x\ProductVersion',
        LangCharSetIDArray);
VerQueryValue(Buffer, StringFileInfo, Result, Length);
Writeln('Product version: ', PChar(Result));
{ Get special build info }
wvsPrintf(StringFileInfo,
        '\StringFileInfo\%04x%04x\SpecialBuild',
        LangCharSetIDArray);
VerQueryValue(Buffer, StringFileInfo, Result, Length);
Writeln('Special build info: ', PChar(Result));
end;
FreeMem(Buffer, VSize + 1);
end.

```

Reference:

7/16/98 4:33:49 PM



Last Modified: 01-SEP-99

## Article #15630: Compiling RWPDemo.pas in Turbo Pascal for Windows.

Technical Notes Database

TN630C.txt      Compiling RWPDemo.pas in Turbo Pascal for Windows.

Category      :Pascal

Platform      :All

Product       :RW 1.0

Description:

QUESTION:

Why do I get a Type Mismatch error at line 191 when compiling RWPDemo.Pas?

ANSWER:

RWPDemo needs a compile directive to be turned on. The trick is to place then BWCC compiler directive in the Options Compiler dialog in Conditional Define field. Then the application must be compiled using BUILD. If BUILD is not used the some of the units that are used by the demo will not be compiled.

Reference:

7/2/98 10:42:10 AM

Last Modified: 01-SEP-99

## Article #15631: INTERMEDIATE RESULTS OF REAL NUMBER EXPRESSIONS

Technical Notes Database

TN631D.txt INTERMEDIATE RESULTS OF REAL NUMBER EXPRESSIONS

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. Are the intermediate results of real number expressions stored in the 8087 registers?

A. Yes. In all versions, all intermediate calculations of real number expressions are stored on the coprocessor stack.

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15635: Using your EXE as a resource file

Technical Information Database

TI635D.txt Using your EXE as a resource file

Category :General Programming

Platform :All

Product :Pascal All

Description:

```
{
  This program demonstrates how to use your EXE
  file as a resource. You should run this program
  twice - once to write info to the EXE and once to
  read info out.
}
```

```
program foo;
```

```
uses Objects;
```

```
type
```

```
  PMyObject = ^TMyObject;
```

```
  TMyObject = object(TObject)
```

```
    AString: String;
```

```
    constructor Init(S: String);
```

```
    constructor Load(var S: TStream);
```

```
    procedure Store(var S: TStream);
```

```
  end;
```

```
constructor TMyObject.Init(S: String);
```

```
begin
```

```
  inherited Init;
```

```
  AString := S;
```

```
end;
```

```
constructor TMyObject.Load(var S: TStream);
```

```
begin
```

```
  inherited Init;
```

```
  S.Read(AString, SizeOf(AString));
```

```
end;
```

```

procedure TMyObject.Store(var S: TStream);
begin
    S.Write(AString, SizeOf(AString));
end;

const
    RMyObject: TStreamRec = (
        ObjType: 100;
        VmtLink: ofs(KindOf(TMyObject)^);
        Load:
TMyObject.Load;
        Store:
TMyObject.Store);

var
    Rez: PResourceFile;
    TheStream: PBufStream;
    AObject, Obj: PMyObject;

begin
    { Register my object for streaming }
    RegisterType(RMyObject);
    { Create instance of my object }
    Obj := New(PMyObject, Init('Hello world'));
    { Create instance of a stream pointing to EXE file }
    TheStream := New(PBufStream, Init(ParamStr(0), stOpen, 1024));
    { was stream created okay? }
    if TheStream^.Status = stOk then begin
        { Create instance of resource file }
        Rez := New(PResourceFile, Init(TheStream));
        { try to grab object from resource stream }
        AObject := PMyObject(Rez^.Get('My Object'));
        if AObject <> nil then
            { if found, then write object's string to screen }
            writeln('The magic string is: ' + AObject^.AString)
        else
            { if not, then write object to resource }
            Rez^.Put(Obj, 'My Object');
    end;
    { clean up }
    Obj^.Free;
    Rez^.Free;
end.

```

Reference :

7/16/98 4:33:49 PM

Last Modified: 01-SEP-99

## Article #15637: SIGNIFICANT DIGITS OF 8087 FLOATING-POINT TYPES

Technical Notes Database

TN637D.txt      SIGNIFICANT DIGITS OF 8087 FLOATING-POINT TYPES

Category      :Pascal

Platform       :All

Product        :Turbo Pascal 4.0+

Description:

Q. How many significant digits do the 8087 floating-point types provide?

A.	Type	Digits of precision
	-----	-----
	single	7-8
	double	15-16
	extended	19-20
	comp	19-20

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15641: College Student Guide to Reading and Writing Files

Technical Information Database

TI641D.txt College Student Guide to Reading and Writing Files  
Category :General Programming  
Platform :All  
Product :Delphi All

### Description:

College Student Crib Notes to:

- 1) Printing out your programs' output to a file.
- 2) Printing out your programs' output to the printer.
- 3) Reading from an input file.

### Printing Out Your Programs Output to a File

----- --- ---- ----- ----- -- - ----

Often times, professors will require more than your honesty and good faith to receive full credit for your program. They will want to see both your program listing and the output generated from the program. But how do you do this in Delphi and our Pascal Products ???

Simply in Delphi ....

```
program CrtApp;
uses WinCrt;
var outfile: TextFile;
begin
  AssignFile(outfile, 'c:\outfile.txt');
  Rewrite(outfile);
  writeln(outfile, 'Hello World from Delphi');
  writeln(outfile, 'My Program works, and here is ' +
            'the output to prove it...');
  CloseFile(outfile);
end.
```

Simply in Pascal .....

```
Program HelloWorld;
```



```

var
  outfile: text;
begin
  assign(outfile, 'c:\output.txt');
  rewrite(outfile);
  writeln(outfile, 'Hello World');
  writeln(outfile, 'My Program works, and here is the
                output to prove it...');
  close(outfile);
end.

```

## Printing Out Your Programs' Output to the Printer

-----

In some cases, there may be a need to print output directly to the printer. This is the way.

In Delphi ...

```

program CrtApp;
uses WinCrt;
var outfile: TextFile;
begin
  assignfile(outfile, 'LPT1');
  rewrite(outfile);
  writeln(outfile, 'Hello World from Delphi');
  writeln(outfile, 'My Program works, and here is ' +
                'the output to prove it...');
  closefile(outfile);
end.

```

In Pascal ...

```

Program HelloWorld;
var
  outfile: text;
begin
  assign(outfile, 'LPT1');
  rewrite(outfile);
  writeln(outfile, 'Hello World');
  writeln(outfile, 'My Program works, and here is the
                output to prove it...');
  close(outfile);
end.

```

## Reading From an Input File

-----

Many times, it will be necessary to read information from an input file supplied by another person. This is an example of how to implement this.

In Delphi ...

```
program CrtApp;
uses WinCrt;
var
  infile, outfile: TextFile;
  num_lines, x: integer;
  line: string;
begin
  assignfile(infile, 'C:\INFILE.TXT');
  assignfile(outfile, 'C:\OUTFILE.TXT');
  reset(infile); {reset moves the pointer to the beginning}
                  {of the file.}
  rewrite(outfile); {clears the contents of the file}
  readln(infile, num_lines);
  for x:= 1 to num_lines do
  begin
    readln(infile, line);
    writeln(outfile, line);
  end;
  closefile(infile);
  closefile(outfile);
end.
```

In Pascal ...

```
Program ReadInput;
var
  infile, outfile: text;
  num_lines, x: integer;
  line: string;
begin
  assign(infile, 'C:\INFILE.TXT');
  assign(outfile, 'C:\OUTFILE.TXT');
  reset(infile); {reset moves the pointer to the beginning}
                  {of the file.}
  rewrite(outfile); {clears the contents of the file}
  readln(infile, num_lines);
```

```
for x:= 1 to num_lines do
begin
  readln(infile, line);
  writeln(outfile, line);
end;
close(infile);
close(outfile);
end.
```

```
{START INFILE.TXT}
```

```
2
```

```
Hello World
```

```
My Program works, and here is the output to prove it.
```

```
{END INFILE.TXT}
```

-----  
For further reference, please check your Programmer's Reference.  
Look for the topics (AssignFile, Assign, Reset, Rewrite, readln,  
writeln, Close, CloseFile)  
-----

This document was written and inspired by the author's sympathy  
for the starving college student-- only because he was recently  
in your shoes..!!

Reference:

7/16/98 4:33:49 PM

Last Modified: 01-SEP-99

## Article #15643: COMP TYPE USES

Technical Notes Database

TN643D.txt      COMP TYPE USES  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. What type is Comp? What is it useful for?

A. The Comp type is a cross between an integer and a real type and is available when 8087 code is generated {\$N+}. If no math coprocessor is available, specify {\$N+,E+} and the emulator will support the Comp type. The compiler treats it as a real type without an exponent. Thus Comp is useful when you need to store extremely large numbers but don't need a decimal point. For example, you might use variables of type Comp to store amounts in cents and divide the value of the variable by 100 to determine what the value in dollars and cents would be.

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15647: Using FindFirst and FindNext to find directories.

Technical Information Database

TI647D.txt Using FindFirst and FindNext to find directories.

Category :General Programming

Platform :All

Product :Pascal All

Description:

```
{
  Finding directories using Pascal's FindFirst() and
  FindNext() procedure can be a bit tricky. By default,
  FindFirst() and FindNext() will find ALL files. Not
  just directories. To determine if what FindFirst()
  and FindNext() finds is really a directory, you must
  check the Attr field of the SearchRec to determine
  if the attribute indicates that the element is a
  directory.
```

The following program adds only directory names to an array of strings.

```
}

program FindIt;

uses DOS;

const
  ArrayLimit = 512;

type
  DOSFileName = String[12];

var
  SR: SearchRec;
  A: array[1..ArrayLimit] of DOSFileName;
  i: integer;

begin
  i := 1;
  { find the first directory }
```

```
FindFirst('c:\*.*', Directory, SR);
{ check for an error }
If DosError = 0 then begin
  repeat
    { is it really a directory? }
    if (SR.Attr and Directory) <> 0 then begin
      A[i] := SR.Name; { add dir to array }
      i := i + 1;      { increment index }
    end;
    FindNext(SR);      { find next directory }
  { exit loop if a DOS error occurs }
  until DosError <> 0;
end
else
  { Write error if FindFirst() fails }
  Writeln('Can''t do it. Error code: ', DOSError);
end.
```

Reference:

7/16/98 4:33:49 PM

Last Modified: 01-SEP-99

## Article #15649: 80x87 NUMERIC COPROCESSOR

Technical Notes Database

TN649D.txt      80x87 NUMERIC COPROCESSOR

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. How will the compiler use my 80387 coprocessor?

A. The compiler will treat it as an 8087 coprocessor and none of its extended capabilities will be used.

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15653: Delphi Fact Sheet

Technical Information Database

TI653D.txt - Delphi Fact Sheet

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

Delphi Fact Sheet

Delphi is the only Windows development tool to combine the Rapid Application Development (RAD) benefits of visual component-based design with the power of an optimizing native code compiler and scalable database access.

Delphi represents the next generation in:

- Performance--with the world's fastest compiler
- Rapid Application Development--via visual Two-Way-Tools()
- Component Reuse--a true object-oriented environment
- Scalable Database Access--the fast track to client/server

The fastest way to the fastest applications

Delphi is the next generation Windows development tool, combining the most intuitive visual design environment with the unrivaled performance of a world-class optimizing native code compiler. It compiles Windows applications at more than 350,000 lines per minute.\* An automatic MAKE facility ensures that only code that has been changed is recompiled for maximum efficiency, and the built-in assembler allows for ultimate fine-tuned performance.

The resulting executable files (EXEs) are lean, highly efficient, and run up to 10 to 20 times faster than interpreted p-code. Delphi executables are immediately deployable, royalty-free, and require no runtime interpreter Dynamic Link Library (DLL), so maximum performance is guaranteed.



RAD, from prototype to production

The comprehensive suite of visual design and debugging tools accelerate development so you move seamlessly from prototyping to deployment. You'll create high-performance applications in record time with the gallery of reusable forms, project templates, programming experts, and context-sensitive intelligent help.

And smart computer-based tutorials guide you through your project and help explain concepts. The Object Inspector; provides complete access to properties and events of a selected component, with smart Property Editors for rapid design changes. The Project Manager provides an overview of all forms and code for a development project, while the ObjectBrowser() provides a graphical view of the overall object hierarchy.

The sophisticated GUI Debugger jumps directly to the point in the source code where problems occur, so you can find and fix problems quickly and easily. And full support for conditional breakpoints, watchpoints, call stack monitoring, single-step execution, and trace mode are included.

A suite of powerful Windows resource editors lets you create and modify menus, icons, cursors, and bitmaps for low-level debugging. The WinSight() utility allows you to monitor the Windows messaging flow, while WinSpector() instantly provides comprehensive diagnostic details.

Visual Two-Way-Tools for power programming

With Delphi, you can always get to your code, and everything you can do visually can also be done in code. Innovative Two-Way-Tools maximize your productivity by letting you move seamlessly between the visual design environment and the synchronized underlying source code.

Switching between the visual design mode and the intelligent BRIEF;-style source code editor is rapid and intuitive. It supports advanced editing features such as Color Syntax Highlighting, macro recording, column block marking, and regular expression support.

## Reuse components to maximize productivity

The comprehensive Visual Component Library (VCL), a collection of more than 75 objects such as dialogs, buttons, and list boxes, includes a host of additional reusable objects including database controls, notebook tabs, grids, multimedia controls, and much more!

The completely customizable Integrated Development Environment can be tailored to individual or corporate preferences. Custom reusable components and commercial VBXs are easily added to the fully reconfigurable Component Palette. You can create and install your own application and form templates. And the Delphi component-based architecture allows you to seamlessly integrate DLLs, VBBX controls, and OLE 2.0 servers into your applications.

## Integrated database support and reporting capabilities

Delphi incorporates the Borland Database Engine (BDE), providing direct access to data stored in dBASE, Paradox, and the Local InterBase Server, and to other data formats via ODBC. And data-aware controls allow manipulation of live data at design time, improving project turnaround time.

A comprehensive selection of Visual Data Objects helps make sophisticated database application design a breeze. And the included Database Desktop() utility lets you create or restructure tables and manage connections, insulating compiled Delphi applications from changes in database locations.

Delphi also includes the PC version of ReportSmith; Borland's award-winning database reporting and query tool that supports live data access at design time. ReportSmith supports a wide range of popular database formats, and

provides a variety of graphing, tabulating, and charting options.

## The ultimate complementary tool

If you prefer to work with a variety of programming tools, Delphi is the perfect addition to your toolset. It allows

you to create reusable DLLs that can be called from applications built with C++, Paradox, dBASE, PowerBuilder, Visual Basic, and other popular development tools. Delphi can also use standard VBX controls, OLE2.0 servers, DLLs created by other development tools, Microsoft multimedia, MAPI, and pen computing APIs. No matter what you program with today, the Delphi component-based design makes it a valuable addition that leverages your existing investment in code.

Delphi gives you the control and flexibility you demand with complete access to the Windows API, while shielding you from the complexities of Windows programming.

## Object-Oriented Programming

The underlying Object Pascal language has been enhanced to fully support exception handling. All your GPFs, disk I/O errors, and other dreaded problems are automatically trapped. And it offers all of the benefits of a structured programming language and a true object-oriented development tool, including polymorphism, inheritance, and encapsulation. So Delphi applications are incredibly robust and deliver superior solutions for all of your mission-critical projects.

Enhanced compiler features such as conditional compilation and smart linking make it even easier to rapidly design professional applications intended for a variety of target environments.

## Fast track to Windows 95

Delphi applications developed for the Windows 3.1 operating environment will also run under Windows 95 and Windows NT. Upgrading them to full 32-bit performance simply involves a single-click recompile with the forthcoming Windows 95 version of Delphi. No rewriting of code is necessary!

## Smooth scaling to client/server

Delphi provides an easy path to the fast-growing market of client/server applications development. Transparent local SQL development is possible with Delphi, using the built-in Local InterBase Server, which enables you to rapidly develop

high-performance ANSI SQL-92 compliant applications for standalone systems.

To make the transition from Delphi to professional client/server development and deployment using external database hosts, upgrade to Delphi Client/Server. It's a quick and easy process requiring no additional coding. Delphi Client/Server features high-performance native drivers for Oracle, Sybase, Informix, and InterBase, along with team development support, a Visual Query Builder tool, and much more.

Delphi, the developer's solution

- Quickly and easily customize the Component Palette.
- Delphi gives you everything you need for Rapid Application Development.
- Manage all your code and visual objects with the Object Inspector.
- Two-Way-Tools give you complete control of your code.
- The Visual Component Library lets you build complete database applications in minutes.

Quotes:

"It's going to change our lives, you know."

--J.D. Hildebrand, Editor

Windows Tech Journal

"When it comes to keeping required coding to a minimum, Delphi excels..."

--Windows Sources

December 1994

"Delphi gives developers exactly what they need to create complete applications, posthaste."

--Susan Ryan

InfoWorld

"5 Stars--Excellent"

--PC/Computing

February 1995

Delphi Specifications

## Optimizing Native Code Compiler

- Compiles at over 350,000 lines per minute\*
- Create fast standalone EXEs with no runtime interpreter  
Dynamic Link Library (DLL)
- Applications run up to 10 to 20 times faster than  
interpreted p-code
- Create DLLs that work with C++, dBASE, Paradox, Visual  
Basic, PowerBuilder, and others
- Access to all Windows API functions and messages
- Optimized case statements, sets, 32-bit math operations,  
string and file routines, and more
- Math coprocessor and emulator support
- Automatic built-in MAKE facility
- Conditional compilation
- Smart Linker removes unused objects and code
- Command-line compiler and MAKE facility
- Built-in assembler for tuning performance
- Linker optimization for smaller EXEs

## Integrated Development Environment

- Integrated visual form designer with over 75 components
- Two-Way-Tools automatically synchronize code and visual  
representations
- Object Inspector allows visual customization of  
components without writing code
- Built-in tools for alignment, scaling, sizing, and tab  
order
- ObjectBrowser displays object hierarchy, units, globals,  
and code references
- Project Manager displays all files and forms
- Customizable environment including SpeedBar, Component  
Palette, editor, and browser
- Intuitive Icon, Bitmap, and Menu editors
- Open environment for adding your own tools, experts,  
components, and Property Editors

## Visual Component Library (VCL)

- Customizable palette of over 75 reusable components
- Support for the latest Microsoft systems technologies  
including OLE 2.0, DDE, VBXs, DLLs, MAPI, and ODBC
- Standard components for menus, bitmapped buttons, masked  
edit fields, panels, graphics, notebook tabs, grids,  
outlines, list boxes, combo boxes, check boxes, labels,  
and more
- Visual Data Objects for accessing databases, tables,

queries, reports, SQL stored procedures, as well as data-aware grids, navigators, lookup lists, edit fields, list boxes, combo boxes, memos, bitmaps, and more

- "Live" design-time data access
- Extend the VCL at any time with third-party libraries or with your own objects created with Delphi
- Database Field Editor for validation rules, display format, edit mask, and field width
- Standard dialog objects for file operations, printing, font and color selection, and searching
- System objects for accessing OLE 2.0 servers, DDE, multimedia and file, and directory lists
- Support for MDI, printing, and graphics
- 2-D and 3-D charts with included ChartFX control
- Compatible with hundreds of VBX controls
- Includes sample objects with complete source code written in Delphi
- VCL source code available separately

#### Object Pascal Language

- High-performance, structured, object-oriented language
- Complete support for inheritance, polymorphism, and encapsulation
- Control over privacy with Public, Private, Protected, and Published reserved words
- Create components with properties and events
- Use inheritance to customize any object
- Automatic runtime type information and object persistence
- Automatic, extensible exception handling
- Support for open arrays, user-defined types, objects, and pointers
- Advanced language support for delegation and class references
- Separate compilation of units and DLLs
- Over 150 library routines for mathematical operations, string manipulation, text formatting, and file management
- WinCRT unit for creating standard Pascal "console" programs
- Compiles Borland Pascal 7.0 for Windows code

#### Powerful Editing

- Full-featured BRIEF-style editing
- Unlimited undo and redo
- No limit on file size

- Macro record and playback
- Column block marking
- Regular expression (GREG-style) search
- Customizable Color Syntax Highlighting
- Selectable editor keystrokes with support for BRIEF and Epsilon

#### Integrated Debugging

- Integrated GUI Debugger with single-step and trace
- Conditional breakpoints and watchpoints
- Call stack view for tracing code execution
- Evaluate and modify expressions and variables
- WinSight Windows message trace utility
- WinSpector postmortem analysis tool
- Compatible with Turbo Debugger for Windows (available separately)

#### Borland Database Engine

- High-performance engine with native drivers for dBASE, Paradox, and Local InterBase Server
- Fully scalable support for migrating applications from desktop to client/sserver
- Royalty-free deployment of database engine
- Space-efficient deployment of database applications
- Includes Database Desktop for managing database aliases as well as creating and restructuring tables
- ODBC support for Access, Btrieve, Excel, DB2, AS/400, Ingres, HP ALBASE/SQL, and gateways like IBM DDCS/2, Micro Decisionware and Sybase Net-gateway. (available separately)
- Delphi Client/Server includes SQL Link native drivers for InterBase, Oracle, Sybase, Microsoft SQL Server, and Informix. (available separately)

#### Local InterBase 4.0 Server

- High-performance ANSI SQL-92 compliant
- Ideal for "off-line" development and single-user applications
- Computed fields
- Outer joins and join expressions in the where clause
- Complex data including Binary Large Objects (BLOBs()) and multidimensional arrays
- Advanced features including stored procedures, triggers, and constraints
- Fully scalable to InterBase on NT, NetWare, and UNIX

platforms

- Local InterBase Server Deployment Kit available separately

### ReportSmith 2.5

- Award-winning database reporting and query tool
- Access ReportSmith from the Delphi Tools menu and TReport component
- Features "live" report writing
- Adaptive data access works with any size database
- Prebuilt report templates and styles
- Crosstab and mailing label reports
- Multilevel sorting
- Custom group specifications
- Onscreen Print Preview mode
- Free distribution of ReportSmith runtime module
- Fully compatible with ReportSmith SQL edition

### Documentation and Help

- Extensible gallery of application templates and experts
- Seven Interactive Tutors() with "live" interaction
- Over 5Mb of on-line help
- Five manuals with over 1,200 pages of documentation: Delphi User's Guide, Component Writer's Guide, Database Application Developer's Guide, Creating Reports, and InterBase User's Guide

### Minimum System Requirements

- Intel 386-based PC or higher
- Microsoft Windows 3.1 or later, 100%-compatible version
- 6Mb of extended memory or higher
- 30Mb hard disk space
- CD-ROM drive (3.5" disks available separately)

### Networks Supported

Any Microsoft Windows 3.1 compatible network, including OS/2 2.1, Novell NetWare, Windows for Workgroups 3.11, and Windows NT 3.11.

\*Pentium 90MHz

Copyright 1995 Borland International, Inc. All rights reserved. All Borland product names are trademarks of Borland International, Inc. Borland's DELPHI products and services are not associated with or sponsored by Delphi



Internet, an on-line service and Internet access provider.  
Corporate Headquarters: 100 Borland Way, Scotts Valley,  
California 95066-3249, 408-431-1000. Offices in: Australia  
(61-02-911-1000), Belgium (32-2-47-99-903), Brazil (55-11-  
851-5326), Canada (416-229-6000), Chile (56-2-233-7113),  
Denmark (45-48-14-00-01), France (33-1-41-23-11-00), Germany  
(49-6103-9790), Hong Kong (852-2540-4380), Italy (39-2-26-91-  
51), Japan (81-03-5350-9370), Korea (82-02-551-2795), Latin  
American Headquarters in U.S.A. (408-431-1074), Malaysia (60-  
03-230-2618), Netherlands (31-020-540-54-00), New Zealand  
(64-9-443-8890), Singapore (65-339-8122), Spain (34-1-650-72-  
50), Sweden (46-8-634-35-00), Switzerland (41-031-761-2604)  
Taiwan (886-2-718-6627), and United Kingdom (44-734-321-150)

Reference:

3/30/99 3:00:47 PM

Last Modified: 01-SEP-99

## Article #15655: FLOATING-POINT PARAMETERS

Technical Notes Database

TN655D.txt      FLOATING-POINT PARAMETERS

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

- Q. How are floating-point numbers passed to procedures in versions 4.0 and 5.0+?
- A. In version 5.0+, the numbers are passed using the CPU stack. However, in version 4.0, they are passed via the 80x87 coprocessor stack.

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15659: Delphi Quick Info Sheet

Technical Information Database

TI659D.txt - Delphi Quick Info Sheet

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

Borland Delphi for Windows Quick Info Guide

RADical performance Windows development

Delphi is the only Windows development tool to combine the Rapid Application Development (RAD) benefits of visual component-based design, with the power of an optimizing native code compiler and scalable database access.

Delphi represents the next generation in:

- Performance--with the world's fastest compiler
- Rapid Application Development--via visual Two-Way-Tools
- Component Reuse--a true object-oriented environment
- Scalable Database Access--the fast track to client/server

The fastest way to the fastest applications

Delphi is the next-generation Windows development tool, combining the visual design environment with the unrivaled performance of a world-class optimizing native code compiler. Delphi includes the world's fastest compiler, capable of compiling Windows applications at more than 350,000 lines per minute, resulting in executable files (EXEs) that are robust and lightning-fast.

Delphi executables are also immediately deployable and completely royalty-free, and no runtime interpreter Dynamic Link Library (DLL) is required. Delphi applications

typically execute up to 10 to 20 times faster than interpreted p-code.

Only Borland can offer this unique combination of integrated tools including its award-winning report writer, ReportSmith, the Borland Database Engine, and the Local InterBase Server.

RAD, from prototype to production

Delphi accelerates Windows development by providing a comprehensive suite of visual design and debugging tools that move you seamlessly from prototyping to deployment of finished applications. Delphi's Two-Way-Tools help you optimize your programming time by switching effortlessly between visual design and the underlying source code.

Reuse components to maximize productivity

Included with Delphi is the comprehensive Visual Component Library (VCL), a collection of more than 75 standard Windows objects such as dialogs, buttons, and list boxes, along with a host of additional reusable objects including database controls, notebook tabs, grids, multimedia controls, and much more! Delphi's Integrated Development Environment (IDE) is fully customizable. Custom reusable components, external DLLs, and commercial VBXs can be rapidly added to Delphi's Component Palette.

Delphi is a true object-oriented development tool, including polymorphism, inheritance, and encapsulation. Delphi's component-based architecture allows you to seamlessly integrate DLLs, VBX controls, and OLE 2.0 servers into your applications.

Smooth scaling to client/server

Delphi has been designed to provide an easy path to the fast-growing market of client/server applications development. Transparent local SQL development is possible with Delphi, using the built-in Local InterBase Server, which enables you to rapidly develop high-performance ANSI SQL-92 compatible applications for standalone systems.

"Delphi is Visual Basic done right." --PC/Computing,

February 1995

## Delphi Product Information:

### Product

- Delphi for Windows CD-ROM only. Estimated street price \$350\*
- Special Introductory Offer of \$199.95 for first 90 days (3/1/95-5/31/95).
- \$50.00 rebate coupon inside for Pascal owners. 3.5" disks available separately from Borland for \$19.95 plus \$5.00 shipping.

### Contents

Delphi for Windows  
Local InterBase Server  
ReportSmith 2.5 PC version

#### Manuals:

Delphi User's Guide  
Delphi Database Application Developer's Guide  
Delphi Component Writer's Guide  
Creating Reports  
InterBase User's Guide

### Target Audience

Programmers, VARs, Pascal developers, C++ developers, Visual Basic Pro developers, PowerBuilder developers, Paradox and dBASE developers.

### Delphi Accessories:

#### Turbo Assembler

\$99.95  
3.5" HD disks

#### Description:

Whether you're writing time-critical systems or speeding up your Delphi applications, the Turbo Assembler ensures your applications are as robust and efficient as possible.

#### Buyers:

Windows developers who want to build fast applications or

add fast code to existing applications. Programmers using Microsoft Assembler who want a way to do faster, more flexible assembly language programming.

### Visual Solutions Pack

\$99.95

3.5" HD disks

#### Description:

With the Visual Solutions Pack you can add spreadsheets, databases, word processing, graphics, and more to your Delphi applications with virtually no coding.

#### Buyers:

Windows developers using Delphi or any other development tool that supports VBX format custom controls.

### Paradox 5.0 for Windows

\$495

CD-ROM or 3.5" HD disks

Special limited-time upgrade price of \$129.95 for suite owners of competing databases. \$30 rebate for previous Paradox owners.

#### Description:

With Paradox you get relational database power made easy. DLLs and OLE 2.0 controls built with Delphi are completely compatible with your Paradox applications.

#### Buyers:

Windows database developers who want to build fast, efficient, database applications.

### Borland Delphi support services

Fast Fax for Detailed Information: 1-800-408-0001

TechFax for Technical Information: 1-800-822-4269

Connections Developer Program: 1-800-353-2211

Free Install Support: (408) 461-9195

Credit Card Advisor Line: 1-800-330-3372

#### On-line Services

- On CompuServe, type GO BORLAND
- On BIX, type JOIN BORLAND

- On GENie, type BORLAND
- Borland Download Bulletin Board Service: (408) 431-5096
- For other Assist Support Services: 1-800-523-7070

#### Attention Borland resellers

- For information on obtaining evaluation copy order forms:  
1-800-408-0001
- To request collateral: (408) 431-1950
- For more information: (408) 461-9000

#### Minimum System Requirements

- Intel 386-based PC or higher
- Microsoft Windows 3.1 or later, 100%-compatible version
- 6Mb of extended memory or higher
- CD-ROM drive (3.5" floppy disks available separately)
- 30Mb hard disk space

#### Satisfaction guaranteed!

You can buy Delphi for Windows with complete assurance. If for any reason you are not fully satisfied with your purchase, you can return it to Borland within 90 days. No questions asked!

\*Scholar price available through authorized Borland Educational resellers. Call 1-800-847-7797.

Copyright 1995 Borland International, Inc. All rights reserved. All Borland product names are trademarks of Borland International, Inc. Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Brazil, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Switzerland, Taiwan, and United Kingdom.

Reference:

3/30/99 3:01:15 PM

Last Modified: 01-SEP-99

## Article #15661: LARGE INTEGER TYPE SUPPORT

Technical Notes Database

TN661D.txt      LARGE INTEGER TYPE SUPPORT

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. Does Turbo Pascal 4.0+ support large integers?

A. Yes. TP 4.0+ has virtually every incarnation of 8-, 16-, and 32-bit integers: shortint, integer, longint, byte, and word.

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99



## Article #15665: List of Delphi Books From Third-Party Publishers

Technical Information Database

TI665D.txt - List of Delphi Books From Third-Party Publishers

Category :Delphi 1.x

Platform :All

Product :

### Description:

The Delphi Programmer Explorer

by J. Duntemann/J. Mischel/D. Taylor

Coriolis Group

ISBN: 1-883577-25-X \$39.99

A new type of tutorial: Theory and practice alternate in short chapters, with the emphasis on creating useful software starting on the very first page.

Delphi for Dummies

by Neil Rubenking

IDG Press

ISBN: 1-56884-200-7 \$19.99

Readers will learn about Borland's new language in the easy to understand style of the Dummies series.

Teach Yourself Delphi

by Devra Hall

MIS Press

ISBN: 1-55828-390-0 \$27.95

Here is a complete, self-guided tour to the new development environment from Borland, encompassing all the features of the language and all the tools, tricks, and advantages of Delphi.

Delphi Nuts and Bolts

by Gary Cornell and Troy Strain

Osborne-McGraw-Hill

ISBN: 0-07-882-136-3 \$24.95

If you are an experienced programmer and want a fast introduction to Delphi, this book is for you.

Software Engineering with Delphi

by Edward C. Webber, J. Neal Ford, and Christopher R. Webber  
Prentice Hall Professional, Trade & Reference  
A guide to developing client/server applications with an emphasis  
on Delphi's object-oriented tools.

Delphi by Example

by Blake Watson

Que

ISBN: 1-56529-757-1 \$29.99

Aimed at the beginning programmer who has no prior experience  
with other languages or development products, this book presents  
basic concepts of programming along with a clear explanation of  
the key development tools that are part of Delphi.

Using Delphi, Special Edition

by John Matcho and Eric Uber

Que

ISBN: 1-56529-823-3 \$29.99

This 3-part tutorial on the most important Delphi features covers  
how to install the product and develop applications using  
Delphi's visual tools, explores the Windows application  
development process, and deals with some advanced programming  
topics.

Developing Client/Server Applications with Delphi

by Vince Killen and Bill Todd

Sams Publishing

Walks the reader through the development process of creating  
real-world C/S applications, explaining in detail what the  
thought processes must be even before any code is written.

Delphi Developer's Guide

by Xavier Pacheco/Steve Teixeira

Sams Publishing

ISBN: 0-672-30704-9 \$45.00

Intermediate to advanced guide to developing applications using  
Delphi.

Master Delphi  
by Charlie Calvert  
Sams Publishing  
ISBN: 0-672-30499-6 \$45.00  
Comprehensive tutorial/reference for intermediate programming  
with Delphi.

Teach Yourself Delphi in 21 Days  
by Andrew Wozniewicz  
Sams Publishing  
ISBN: 0-672-30470-8 \$29.99  
Introduces Delphi to the beginning programmer and includes  
question-and-answer section at end of each less to test readers  
progress as they learn.

Mastering Delphi  
by Marco Cantu  
Sybex  
ISBN: 0-7821-1739-2 \$29.99  
Introduces programmers to Delphi's features and techniques,  
including secrets of the environment, the programming language,  
the custom components and Windows programming in general.

Delphi How-To  
by Gary Frerking  
Waite Group Press  
ISBN: 1-57169-019-0 \$34.95  
Presents large collection of programming problems and their  
solutions in standard, easy-to-use reference format, including  
unique solutions that use VBX controls and easy ways to build  
multimedia projects with Delphi.

Developing Windows Applications Using Delphi  
by Paul Penrod  
John Wiley  
ISBN: 0-471-11017-5 \$29.95  
This introduction for traditional C programmers who want to make  
the transition to rapid application development also provides  
detailed instructions for building sophisticated Windows  
applications and for creating graphical interfaces.

Instant Delphi  
by Dave Jewell  
Wrox Press  
ISBN: 1-874416-57-5 \$19.95

Instant Delphi is the fast-paced tutorial guide for the programmer who wants to get up to speed on the Delphi product as quickly as possible.

Reference:

3/30/99 3:01:40 PM

Last Modified: 01-SEP-99

## Article #15667: LINKER ELIMINATES UNUSED DATA

Technical Notes Database

TN667D.txt LINKER ELIMINATES UNUSED DATA

Category :Pascal

Platform :All

Product :Turbo Pascal 5.0 5.5

### Description:

Q. Does the built-in linker eliminate unused data?

A. Yes. Unused code AND data are stripped when you compile to disk. However, if more than one variable is defined in the same VAR block and any one is used, the others will not be stripped from the .EXE. For example:

```
var  A, B: integer;
var  C: integer;
begin
  A:= 0;
end.
```

In this example, although variable B is never used, it was defined in the same block as a variable A. Therefore, B will not be linked out. Variable C will be removed from the .EXE as it is not used and is not in the same VAR block.

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15671: Instructions for Running Delphi from CD-ROM

Technical Information Database

TI671D.txt - Instructions for Running Delphi from CD-ROM

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

RUNNING DELPHI FROM THE CD-ROM

-----

Delphi can be run from a CD-ROM directly. However, it is highly recommended that at least a minimum installation be seriously considered before opting for running Delphi from CD-ROM.

For a minimum installation, 33MB of available hard disk space is required.

Running from a CD-ROM, 9MB of available hard disk space is required.

To configure the system to run Delphi from the CD-ROM,

1. Copy all files found in the CD's \RUNIMAGE\WINDOWS directory to the Windows directory of the hard disk.
2. Copy all files found in the CD's \RUNIMAGE\WINDOWS\SYSTEM directory to the Windows SYSTEM directory of the hard disk.
3. Edit DELPHI.INI in a text editor like Windows Notepad. Look for the section titled [Globals]. Make entry for the PrivateDir setting that resides on a writeable disk, preferably a local hard disk.

For instance, if a directory on the C: drive where chosen called DELPRIV, the relevant section of DELPHI.INI would look as follows.

[Globals]

PrivateDir=C:\DELPRIV\

Now, Delphi can be run from the \RUNIMAGE\DELPHI\BIN directory of the CD-ROM.

#### MODIFYING COMPONENTS

-----

To allow modification of the component library used by Delphi while running from a CD ROM an copy of the Component Library must exist on a writeable disk. Use the following steps as a guide.

1. Copy COMPLIB.DCL from the CD's \RUNIMAGE\DELPHI\BIN directory to a desired directory on the hard disk.
2. Run Delphi.
3. Bring down the Options menu and select Open Library.
4. Specify the directory and file name for the COMPLIB.DCL copied to the hard disk.

Reference:

4/1/99 4:32:43 PM

Last Modified: 01-SEP-99

## Article #15673: STRING MANIPULATION IN AN INTERRUPT ROUTINE

Technical Notes Database

TN673D.txt      STRING MANIPULATION IN AN INTERRUPT ROUTINE  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 3.0+

### Description:

Q. Can you manipulate strings inside an interrupt routine?

A. You can manipulate strings if you treat them as arrays of characters. You must use your own manipulation routines to do so and not the one's provided. In Turbo Pascal versions 4.0+ you may process strings with routines provided.

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99



## Article #15677: Making your Delphi apps show minimized.

Technical Information Database

TI677D.txt - Making your Delphi apps show minimized.

Category :VCL  
Platform :All-32Bit  
Product :

### Description:

Q: When I select the "Run Minimized" option in Program Manager to attempt to make my Delphi application execute in a minimized state, the Delphi application seems to ignore the setting and run normally. Why is this, and how to I fix it?

A: Delphi's Application object creates a hidden "application window," and it is that window, rather than your main form, that is being sent the command to show minimized. To fix this, make your main form's OnCreate event handler look like this:

```
procedure TForm1.FormCreate(Sender: TObject);
{$IFDEF WIN32}           { Delphi 2.0 (32 bit) }
var
  MyInfo: TStartupInfo;
{$ENDIF}
begin
{$IFDEF WIN32}           { Delphi 2.0 (32 bit) }
  GetStartupInfo(MyInfo);
  ShowWindow(Handle, MyInfo.wShowWindow);
{$ENDIF}
{$IFDEF WINDOWS}       { Delphi 1.0 (16 bit) }
  ShowWindow(Handle, cmdShow);
{$ENDIF}
end;
```

In other words, for 16 bits, just pass cmdShow to ShowWindow. For 32 bits you need to obtain the start up info by calling the GetStartupInfo procedure, which fills in a TStartupInfo record, and then pass TStartupInfo.wShowWindow to ShowWindow.

Reference :

3/31/99 10:13:43 AM

Last Modified: 01-SEP-99

## Article #15679: NON-REENTRANT CODE

Technical Notes Database

TN679D.txt      NON-REENTRANT CODE  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 4.0+

### Description:

Q. Which procedures/functions generate non-reentrant code?

A. All disk I/O routines and dynamic memory allocations will be non-reentrant. By disk I/O routines, we mean any routine that calls DOS, so this includes a number of procedures and functions in the DOS unit. Some code in the Graph, Graph3, and Crt units will generate non-reentrant code as well. Floating-point routines used to be non-reentrant in TP 3.0x, but the 6-byte library is now fully reentrant. The emulator, however, is not. Nor is the support code for the 8087 chip, unless special coding is included.

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15683: Delphi Client/Server and Power Builder Compared

Technical Information Database

TI683D.txt Delphi Client/Server and Power Builder Compared

Category :Product Info

Platform :All

Product :Delphi 1.0

Description:

An Evaluation of

Client/Server Development Tools:

A Comparison Between

Delphi Client/Server and PowerBuilder Enterprise

Prepared by Michael Lant, Sphere Data Systems

Table of Contents

Introduction: An Overview of Client/Server Tools

PCs and the Evolving Client/Server Model

The Next Generation

What is Delphi?

Redefining Client/Server Development

The Delphi Component Model

Performance

Compiled vs Interpreted Code

Data Access - A New Approach

Rapid Application Development (RAD)

Minimum Coding, Maximum Flexibility

DataWindows vs Reusable Forms

Component Reuse

The Delphi Inheritance Model

A DataWindows Alternative

Exception Handling

Database Scalability

## Conclusion

### Introduction: An Overview of Client/Server Tools

Client/server development tools remain one of the fastest-growing sectors of the software industry. Forrester Research predicts that the number of client/server developers will climb from 128,400 in 1994 to 698,000 in 1996, while META Group research suggests that over 90% of all new applications will be based on the client/server model. This is an overwhelming endorsement of the popularity of client/server computing.

Client/server computing, however, is not the panacea that many have been predicting. Complex environments, with mixed data sources and immature development tools, have made client/server development a complex undertaking. Researchers such as the Gartner Group have found that the most pressing issues facing current developers of client/server applications include performance of deployed applications, rapid application development (RAD), component reuse and database scalability. Although these concerns are not unique to client/server application development, they are perhaps even more important in this sector, because of the well-documented limitations of first-generation client/server tools.

### PCs and the Evolving Client/Server Model

PCs facilitated the downsizing era, by providing inexpensive platforms for the production of reasonably capable solutions that addressed genuine business needs. These applications brought with them the promise of freedom from host-based technologies and shortened development cycles. But as these PC- and network-based applications grew, user counts expanded and more functionality was necessary. PC database applications frequently "hit the wall."

In addition, all database applications have at their very heart business rules. These rules control processes such as creating new customers and data validation. Historically, PC application development has placed the enforcement of business rules into the

user interface, so programmers would create a screen and write code to respond to user input. This code would perform a specific action based on the presence, or value of, certain information. Such actions might include allowing or preventing the posting of changes to customer data. But the user interface is certainly not the best place to store business rules.

The advent of client/server brought with it a new model that allows for business rules to be stored with the data itself. This is accomplished through the use of triggers and stored procedures created in the actual database. These procedures would enact routines to perform much the same processing that might be performed in a PC application, except that the host handles this task for all of its clients. This approach ensured that there was only one source for the business rules, since they were stored with the data. In many ways, this approach is enticing, but in reality it is plagued with problems.

When business rules are created to be enforced at the server level, they tend to be difficult to write and debug. According to David Sarna and George Febish in the September 1994 issue of *Datamation*: "These are not the tools for creating and maintaining good code. Doing it this way is SQL-abuse; SQL as a query language was never designed to do this kind of procedural computation."

With this model, as an application grows, so do the number of rules that must be created and controlled by the database. Rather than focusing on the efficient storage and retrieval of data, server resources become tied down enforcing business rules - such as ensuring that a last name is supplied for each new customer. Again, Sarna and Febish point out that "many of your business and validation rules are needed so often that they must sit on the client, or performance will go straight to hell."

## The Next Generation

Borland's Delphi is truly a next generation client/server development tool. Although it fully supports the placement of business rules in the user interface or on the server (or even a combination of both), it also presents the possibility of another model that separates the business rules from both the user interface and the server. This approach has the potential to provide greater performance, better scalability, better code reuse and an ideal platform for RAD. Delphi makes this possible

because it employs an open architecture, whereby developers can use or create a wide variety of specialized components when creating applications.

Comparative reports of software tools typically focus on feature matrices or checklists to relate the features of one product versus those of its competitors - and in this context Delphi would certainly emerge as a winner. Checklists, however, do not necessarily help developers understand how one product could be far more effective than another. Thus, the content of this white paper is focused on the needs of client/server application developers, and how Delphi addresses those needs.

What is Delphi?

Delphi represents the next generation of client /server and general Windows application development tools. It is a product of Borland's long-standing leadership in programming languages, database development tools, and object-oriented programming (OOP) technology. Delphi is unique in providing a completely visual integrated development environment (IDE) that is built around a powerful object-oriented compiled language (Object Pascal). The IDE shields programmers from the complexity of Windows development, by encapsulating the Windows application programming interface (API) inside robust, reusable components.

The implementation of a component-based architecture enhances power and flexibility. Components range from simple items (such as buttons, text edit regions or sliders) through to powerful specialized tools (such as high-performance database components used for table access, queries and batch updates). While most client/server development tools include a limited selection of these basic components, Delphi ships with more than 75 reusable components.

Developers may modify Delphi components, create their own, or purchase third-party components (which include text processors, graphics, communications and statistical utilities, among others). Once installed on Delphi's component palette, these additional components become part of the IDE, so that Delphi can be tailored to whatever the developer (or team of developers) specifically requires.

Because of the unique way in which components are implemented in

Delphi, it offers similar open architecture benefits to those of the ISA bus, for example. A component is a self-contained block of code that includes all functionality to work within a Delphi application. Just like inserting a new PC expansion card, any new component added to the library becomes part of Delphi and is indistinguishable from those that ship with the product. This open, component-based structure means that Delphi developers are not tied to any particular development methodology, and may use whatever tools or components are necessary to accomplish a task.

### Redefining Client/Server Development

First generation visual client/server development tools (such as PowerBuilder) feature environments where all components are tightly integrated and dependent upon one another. Such an approach does not encourage modification or customization of existing components, because when a component is significantly altered, it has the potential of changing the way in which it interacts with others. This not only impedes the evolution of the base product, but it also has the potential to introduce unpredictable behavior.

Delphi, on the other hand, has as its foundation a true object-oriented language, with the result that its components are essentially autonomous objects. Component changes are therefore unlikely to affect overall product stability. Also, problems with any specific component behavior can often be corrected by the developer via subclassing, and thereby overriding the errant behavior. This also holds true for components created by a developer or purchased from third-party suppliers.

### Delphi Component Types

Delphi's Visual Component Library (VCL) includes more than 75 pre-built components, which can be used in applications and deployed freely. These include the following:

Standard - Menus, Edit, Label, Buttons, Scroll Bar;

Additional - Bitmap Button, SpeedButton, Notebook, Outline, DrawGrid, Image, Multimedia; Data Access - Database, Table, Query, Batch Move, Report;

Data Controls - DataGrid, DBNavigator, DBLabel, DBEdit, DBImage, ComboBoxes;



Dialogs - Open, Save, Font, Color, Print, Print Setup, Search/Replace, Tabbed Notebook;

System - Timer, FileListBox, Drive and Directory Combo OLE, DDE;

VBX - Chart, Picture, Gauge;

Samples - Gauge, ColorGrid, SpinButton, SpinEdit, Outline, Calendar.

Delphi developers can also create their own components, by modifying existing Delphi components or building them from scratch. The source code for Delphi's VCL is available to help developers in this task. Newly-developed components (or those purchased from third-party developers) can be added to Delphi's component palette.

## Performance

In a client/server environment, overall performance depends upon a combination of factors, including efficiency of the database server, network bandwidth, "middleware" or driver performance, data access techniques, and speed of the user interface. A well-designed architecture should leverage all of these characteristics. In almost all respects, Delphi outperforms PowerBuilder by a significant margin, due to its optimizing compiler, efficient VCL components and high-performance native SQL drivers. Some sample benchmark results are listed below:

Operation (All times in seconds)	Delphi	PowerBuilder
String parse to measure non-database performance; reading file and splitting record into substrings	2.7	22.8
Use a query to load a form from a 20,000 record table. Form supports searching and filtering.	4.6	70.9
Post a record (single order)	1.4	1.3
Apply a filter on 20,000 record table	3.0	6.2

Update record	1.5	1.1
Search for a value, 20,000 records	1.1	1.5

User Interface Performance and Compiled vs Interpreted Code  
 The user interface (UI) is the most visible portion of any application. UI performance covers obvious functionality such as screen updates, along with the underlying processing of data (including numerical calculations and string manipulations). It is the portion that users interact with every day, and it is also the first place where performance differs noticeably between PowerBuilder and Delphi.

Delphi uses an optimizing native code compiler to generate standalone executable files (.EXEs) containing machine code instructions. These executable files are self-contained and, other than the database engine and the data itself, require no external files to run. Although PowerBuilder creates an executable file, it is not truly compiled. When a PowerBuilder application is run, source code instructions are converted to machine code before the instructions may be executed. This extra level of translation introduces significant run-time overhead and seriously degrades performance.

To support its runtime interpretation, PowerBuilder requires external support files in the form of runtime interpreter Dynamic Link Libraries (DLLs). In addition to the application code, these DLLs must be loaded into memory at runtime, creating a larger memory "footprint" and leaving less memory for processing and buffering data. As a result, Delphi applications execute between 10 and 20 times faster than applications created using p-code ("pseudo-code") interpreters such as PowerBuilder, even on target systems with minimal memory.

In moving from PowerBuilder 3.0a to 4.0, PowerSoft tuned its runtime interpreter to achieve slightly better performance on systems fitted with 16M of memory. This performance boost often comes at a cost, however, as final applications can increase from 20 to 70 percent in size. Despite minor enhancements, there is no way for an interpreted p-code language to even approach the execution speed of a compiled language like that of Delphi.

## Data Access - A New Approach

The two primary methods of implementing client/server solutions

with PowerBuilder involve placing business rules on either the client or the server. Unfortunately, neither one adequately addresses the issue of overall performance. If developers choose to place the burden of enforcing business rules on the server, they risk performance bottlenecks due to server and network overload. Yet shifting this task to the comparatively slow PowerBuilder UI is not an ideal solution. As it turns out, PowerBuilder's design makes it difficult not to attach at least some of the business rules directly to the UI.

Delphi, on the other hand, fully supports either of these techniques, as well as allowing for a more flexible model. Delphi developers can create a "virtual third-tier" layer containing business rules independent of the UI and the server. Instead, these rules can be encapsulated in non-visual components (NVCs) in a layer between the UI and database engine. Physically and conceptually, this separates the data and related rules from its representation to the user.

This approach brings the distributed computing vision of client/server another step closer to reality, and also provides a practical means of enhancing performance. All data moving to and from the application passes through this central conduit. The "third-tier" NVCs may be single Delphi components such as a TTable or TQuery or sophisticated combinations of components. By having a single access point to lookup tables (such as State, Province or Product Codes that may be used by several forms), the application need only fetch this data once - when it is started. This can greatly reduce the number of server requests, alleviating network traffic and server load.

More importantly, developers can place connections to the main database tables here as well. As users move between forms, cursors to their data would remain in place. Thus, if basic customer information were displayed in one form, moving to another form displaying credit history would invoke it already pointing to the appropriate customer. PowerBuilder has limited functionality in this area through the dwShareData function, which allows sharing of data between DataWindows. This process results in duplication of large, complex components, and unless sufficient RAM is available, this can further degrade client performance. In the same situation, Delphi reduces the number of components used, since data access and display are accomplished with separate components.

## Rapid Application Development (RAD)

Rapid Application Development (RAD) is an attempt to address the issues of building applications tailored to customer requirements and completing projects with minimal re-working. Historically, serious application development involved tremendous effort from a team of analysts who quizzed potential users to determine every possible requirement of the application. From this collection of information, volumes of carefully documented specifications were produced, then the users would sign off and programming commenced. Some time later, the development team would finally emerge with an application, and users were then expected to adopt the product.

This model had several shortcomings. Often, the people who built the application were not the same as those who did the analysis, resulting in reinterpretation of the specifications. Furthermore, users are often not well equipped to diagnose and describe their technology needs. During the development process, little communication existed between the developers and eventual users of the application. In days gone by, when terminals and batch operations were the only UI, this may have been an adequate model. However, with the advent of PCs, user expectations are significantly higher in regard to the UI characteristics of modern client/server applications. These applications must now provide access to data, and present it in ways that are most meaningful to users, with tools to assist in the analysis and interpretation of the data. As a result, most large organizations are confronted with a backlog of applications, which must be tackled with limited staff, budgets and time.

RAD is a development methodology based around an iterative process of specifying, creating and enhancing an application until the final product is completed. RAD is based on the notion that it is difficult to fully predefine a problem domain as complex as today's typical business applications. As the developers step through each iteration from prototype to finished product, the requirements become more clearly defined. Although not ideal for batch-style applications, RAD is very effective for the development of applications where the UI design is a key consideration.

Minimum Coding, Maximum Flexibility

RAD development tools should allow developers to quickly build prototypes with as little code as possible, and Delphi is very powerful in this respect. Building a database application with a classic Customer -> Orders -> Details form (including database connections) can be done with no lines of code in Delphi. By comparison, PowerBuilder developers must write a dozen lines of code to simply link to a database, then they must also write code to open the form, query the tables and synchronize the tables appropriately.

Delphi's comprehensive array of data-aware components minimizes the level of customization that must be done to deliver a necessary feature or function. Where appropriate functionality does not exist in a VCL component, developers have the option of modifying the component or using a third-party component (such as a VBX control or Delphi native component). Any component may be subclassed (exploiting inheritance) and reused in other areas.

#### DataWindows vs Reusable Forms

RAD methodology assumes that application design evolves through several phases, so a true RAD tool should allow developers to easily adapt to changes (which can be significant). Because PowerBuilder combines data access and display in a single component, some design changes can prove extremely difficult. When a customer requirement suddenly dictates a new approach to either data access or display, this often requires a rewrite of the DataWindow and/or the entire form. Since Delphi separates these two concepts, the adaptation process is much simpler.

As with most first-generation visual development tools, PowerBuilder embeds application code inside objects - with the exception of the DataWindow, which represents both the conduit and the display format of the data. A DataWindow is created in a separate section (or "painter"), and cannot have event-handling code written until it is pasted into a form, whereupon the relevant application code is then permanently attached to that DataWindow. Since code is hidden inside objects scattered about the form, it becomes difficult for developers to locate, and it also may be affected by modifications to objects on the form. Furthermore, by "hard-wiring" the DataWindow to the code, it becomes difficult to reuse that DataWindow in other forms. Delphi developers address this issue by writing event handlers, which are pointers to specific procedures, or references within components that indicate what procedure will be called when a

particular event occurs (mouse click, timer interrupt, etc). Form procedures are stored in a "unit" file, with all the code visible to the developer, so there is no need to search through various objects track down specific code fragments. Once procedures are created, they may be called by any event of any component. Event handlers can also call procedures written for any other event handler, or even directly access components or procedures residing in other forms. Deleting an object (such as a button) does not delete procedures created for that object, so that code changes are easier, faster and safer.

Since Delphi uses the same property/event model for all components, whether they are visual or non-visual, it is more consistent and easier to learn than products such as PowerBuilder. The end result is that Delphi clearly provides an ideal environment for RAD.

## Component Reuse

Inheritance is the ability to create new objects (descendants) from a base object (ancestor). A descendant inherits all characteristics of the ancestor, although these can be overridden. Inheritance is an object-oriented programming (OOP) fundamental that provides a means by which programmers can reuse code, and it encourages development of base objects with characteristics that are likely to be used again. Since descendants inherit any changes made to the parent object, changes that are made will propagate throughout any objects that descend from a base object. The resulting collection of objects is often called a class hierarchy or class library.

PowerBuilder allows developers to use inheritance to create a class library based on forms or groups of controls. The problem with the PowerBuilder approach is that it emphasizes inheritance of the user interface. Because DataWindows perform the task of both data access and data display, and because of the close binding between a DataWindow and various objects (and code) on a form, UI inheritance is the only type of inheritance that makes sense in PowerBuilder. However, what developers mostly need to preserve and reuse is not the visual representation of the data, but instead the complex logic that drives the business rules. In fact, the visual representation usually changes the most. Hence, reuse through inheritance in PowerBuilder often requires the

developer to spend significant time overriding the UI in order to preserve the underlying code. As a result, PowerBuilder forms often carry around much unwanted baggage.

Delphi takes an entirely different approach. In its first release, Delphi, supports inheritance of forms in code only, compared to PowerBuilder's support of UI inheritance. With the ability to separate business rules from the UI, a Delphi developer need only write these once. Different forms can access the same data components and inherit their business rules, but are not bound to them. Hence the developer is free to create any number of forms displaying data in whatever format is appropriate, knowing that the non-visual components supplying data to the user interface have fully encapsulated these business rules.

Such an approach would be difficult with a product such as PowerBuilder, because the underlying interpreter-based architecture is not truly object-oriented. According to Steve Benfield, quoted in the June/July 1994 issue of PowerBuilder Developers Journal: "PowerBuilder's DataWindows is not truly object-oriented. I could try to not use DataWindows and code PowerBuilder in a purely object-oriented way. However, the system would be so slow that it would be useless."

## The Delphi Inheritance Model

Inheritance is fundamental to Delphi, and all VCL components ultimately descend from an object known as TObject (the "T" prefix is an accepted standard to denote an object type). TObject has a great deal of capability built into it, and knows how to handle certain types of errors, along with how to display itself in the form designer. Many other components descend from TObject, each acquiring more capability and becoming more specialized along the way.

The familiar case of copying to the Windows clipboard is an example of where Delphi's inheritance can be most powerful. This turns out to be a complex procedure requiring more than 100 lines of code in PowerBuilder, whereas the following code sample shows how the same CopyToClipboard is handled in Delphi:

```
if (( ActiveControl ) is TCustomEdit ) then  
TCustomEdit(ActiveControl).CutToClipboard
```

This examples surfaces three important issues. The first is that it highlights how some very basic functionality is missing from PowerBuilder. The second is that Delphi developers deal with functions such as this on a much higher level than with traditional programming languages. Thirdly, the inheritance benefits are obvious. TCustomEdit is an ancestor of most of the editable components in Delphi (such as TEdit, TDBEdit and so on) In this example, the is operator checks to see if the active component is a descendant of TCustomEdit, and if so, the next line casts the active control as its ancestor - TCustomEdit. Because CutToClipboard is a method of TCustomEdit, all controls inherited from it will also inherit this method.

### A DataWindows Alternative

DataWindows are an important feature of PowerBuilder. By attempting to incorporate all database capability into a single structure, the result is a component that is extremely powerful, but highly complex. The PowerBuilder 3.0a reference manual allocates almost 150 pages to explaining the properties and syntax of the dwDescribe and dwModify functions, which are just two of the more than 130 functions used within DataWindows. According to Breck Carter, quoted in the June/July issue of PowerBuilder Developers Journal: "dwModify( ) and dwDescribe is a topic that I know has been a sticking point for many developers. These functions can be daunting with their screwy syntax and difficult-to-debug commands . . . If DataWindows' bazillion functions and events don't get you, you probably won't avoid being shocked and overwhelmed by dwModify."

PowerBuilder developers place all of their code and forms in a single .PBL application file, while Delphi applications reside in three main file types. The application itself comprises .DPR and .PAS files, while .DFM files contains the binary representation of forms (buttons, grids etc). Delphi's project manager provides a high-level view of all these files, and units from other directories or projects can be easily included. By storing units in separate files, team development is simplified and the allocation of tasks is made easier. Developers need not even be in the same physical location.

During compilation, Delphi keeps track of which files have changed since the last compile and only generates those changes. Delphi's compilation speed is the fastest in the industry, at



more than 350,000 lines of code per minute on a 66MHz Pentium fitted with 8M of RAM. A reasonably complex client/server application containing over a dozen forms will compile all files in less than a minute.

PowerBuilder's interpreter-based design makes direct comparisons with Delphi's compiled performance unrealistic. PowerBuilder developers cannot exit the code painter until their code is syntactically correct, so that moving between different code areas is greatly restricted. PowerBuilder then generates p-code from its scripting language, along with a runtime executable file. This p-code generation stage takes about the same amount of time as the complete syntax checking and compilation to machine code of a Delphi application. The PowerBuilder application, however, still requires translation and decoding overhead each time it is executed.

As the complexity of applications and size of development teams grows, so does the need to maintain control of the process. Keeping track of the latest versions of source files is an important part of this process, and Delphi accomplishes this by providing a direct interface to PVCS, a leading version control utility. Delphi will directly link to the PVCS DLL and integrate it into the development environment.

PowerBuilder developers frustrated with limitations of its programming language are forced to leave the environment and must turn to languages such as C++ or Pascal. Once they leave PowerBuilder, they must abandon any of their custom class libraries, which are not supported by other products. Delphi does not present such obstacles, as it is built around a comprehensive structured, object-oriented language featuring pass-through SQL commands and inline assembler. Since Delphi can encapsulate applications into DLL format, any Windows program (including PowerBuilder) can call a Delphi-generated DLL. In the same manner, Delphi applications can also access DLLs created using other development tools.

## Exception Handling

When developing serious client/server applications, it becomes essential to write code that is able to deal with the possibility of any process failing. Again, according to Breck Carter in the

1994 Special Issue of PowerBuilder Developers Journal: "The PowerScript language is less than ideal when it comes to error handling, Most functions can diagnose errors, but with very few exceptions the default action is to ignore the error and proceed. There are no debugging switches or compiler directives to change this behavior to trap all errors and halt. With the exception of the global SystemError event, there is little language support for automatic error detection or handling."

This lack of support from the PowerScript language makes it difficult, if not impossible to create robust error-handling routines without writing enormous amounts of code. In contrast, Delphi introduces a new and more powerful error-handling methodology. To understand how important this new approach is, we must first look at how it is done.

Fundamental to Delphi is the concept of objects. Every control in Delphi descends ultimately from a base object control called TObject. As shown in the diagram above, TObject and hence, all of its descendants already knows about errors. This knowledge is passed to the programmer by means of objects. When an error occurs, Delphi instantiates an Exception object of that exception type. For example, if a divide by zero error occurs, an EDivideByZero object is created. All forms have built into them, the means of dealing with most errors. Rather than having to write code to test for every possible failure, the following construct is used:

```
try
    { statements }
except
    on ESomething do { specific exception handling code } ;
    else {default exception-handling code } ;
end;
```

Delphi also allows developers to create their own exception types as follows:

```
type
EpasswordInvalid = class(Exception) ;
```

This new user type exception is used simply as follows:

```
if Password <> CorrectPassword then  
raise EpasswordInvalid.Create( 'Incorrect password entered' ) ;
```

These examples are trivial and meant only to display concepts. Delphi exception handling actually has capabilities far beyond this. For instance, Windows General Protection Failure (GPF) errors that are impossible to deal with in PowerBuilder applications may be handled via Delphi's EGPFault exception type. This simple construct provides enormous power and flexibility, and minimizes the amount of code that needs to be written, further shrinking the development cycle.

## Scalability

As more attention is focused on client/server development, the scope of such applications is expanding. Where once it was sufficient to provide basic data entry and inquiry capabilities, users now demand that their applications have functionality far beyond what can be provided from mainframe solutions. Delphi is the ideal tool for this type of transition, being built around Object Pascal, an enhancement of a language that is already known by more than two million developers.

Delphi does not hide code within UI objects, and is thus more familiar to mainframe developers and easier to work with than PowerBuilder. It maximizes code reuse, minimizes development turnaround time and has the performance and stability necessary for large applications. It is also infinitely extensible, due to its OOP architecture, comprehensive array of components and full-featured programming language.

Many applications are also being scaled up from PC databases written in dBASE, Clipper, FoxPro and Paradox. As these applications make the transition from PC environments into the full client/server model, they must be able to address both PC and server data. PowerBuilder has extremely poor performance when accessing PC databases, and does not provide support for Paradox tables beyond the Paradox 3.5 format, nor does it fully support the Paradox file locking mechanism.

Some organizations decide that a portion of their data should reside on a server platform such as Oracle, while other portions should remain in a PC format such as dBASE. Because of Delphi's broad database support and extremely good performance against

both PC and server data, it is the ideal tool for when mixed data access is required. Developers need only one development tool regardless of whether the data source is PC-based, server-based or both. Delphi is the only development tool that can adequately address all of these models. On the other hand, PowerBuilder has opted to largely ignore the PC database side.

## Borland Database Engine

Although not component-based itself, the component-like architecture of the Borland Database Engine (BDE) fits neatly into the architecture of Delphi. As the core technology for most of Borland's database tools including dBASE and Paradox, and used by millions of database users, the BDE has evolved into an extremely stable, high-performance technology that can deal with almost any database format. It may be easily configured to access popular server formats (including Oracle, Sybase, Informix and InterBase) using high-performance SQL Links. PC database formats such as dBASE and Paradox are accessed directly via drivers built into BDE, which also includes support for the wealth of ODBC drivers.

The BDE is also sold as a separate product, as all of its functionality is accessible to developers via an API. However, Delphi developers are most likely to use the native data-aware components to interact with the BDE, although it is possible to write directly to the API or even bypass it altogether if required. This open architecture provides a level of power and flexibility not possible within PowerBuilder.

## Local InterBase Server

The Local InterBase Server is a locally-installed Windows version of Borland's powerful InterBase workgroup server. Unlike the Watcom SQL engine that ships with PowerBuilder, this product is a full-featured SQL database, including:

- Stored Procedures and triggers
- Automatic two-phase commit
- Explicit transaction management
- Declarative referential integrity
- Event alerters
- Simultaneous access to multiple databases and transactions
- Multi-generational architecture
- Updatable views

## Outer joins

## User defined functions (UDFs)

Because the Local InterBase Server supports most of the functionality of the high-end InterBase server, developers can create SQL-based applications using locally-stored data. Thorough testing of these applications can be performed locally, without concern for either the server load or the possibility of damaging live data. Once an application has been tested and debugged locally, it may then be scaled to work with an external database server. If the developer has elected to use specific features of InterBase, these will perform identically when the application is deployed against an external InterBase server.

With PowerBuilder, because the Windows version of WATCH SQL lacks many of the features on the server versions of Watcom SQL, developers cannot experience the benefits of direct scaling that Delphi provides. When PowerBuilder accesses the Windows version of Watcom SQL, it must open a separate application window. Since this is a standard window, users can task-switch to the window or close it - either from the window itself or externally. This makes the Windows version of Watcom SQL unsuitable in a production environment.

The Local InterBase Server, however, loads as a DLL in Delphi and cannot be accessed by users. This, combined with its comprehensive feature set, make the Local InterBase Server ideal for applications running on notebook PCs. When a user returns to the office, the application can be linked to the server version of the database. Applications may therefore scale from notebooks up to Windows NT or Novell NetWare NLM servers, and all the way up to the largest Unix-based systems, without developers needing to change a single line of code. By offering this capability, Delphi represents the ultimate in database scalability.

## Conclusion

This paper has examined just a few of the many differences between Delphi and PowerBuilder. Both are clearly high end client/server development tools with excellent features. When examined from the perspectives of performance, scalability, reuse and RAD, Delphi clearly leads the way into the next generation of client/server development.

## Reference:

7/16/98 4:33:51 PM

Last Modified: 01-SEP-99

## Article #15685: REENTRANT CODE

Technical Notes Database

TN685D.txt REENTRANT CODE  
Category :Pascal  
Platform :All  
Product :Turbo Pascal 4.0+

### Description:

Q. How do I write reentrant code in Turbo Pascal?

A. If a routine follows these rules, it is reentrant:

1. All data is allocated on the stack.
2. The routine doesn't use any global variables.
3. The routine can be interrupted at any time without affecting the execution of the routine.
4. The routine doesn't call any other routines that are not reentrant (e.g., DOS I/O).

### Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15689: Delphi vs Visual Basic

Technical Information Database

TI689D.txt Delphi vs Visual Basic  
Category :Product Info  
Platform :All  
Product :Delphi 1.0

### Description:

An Evaluation of  
Rapid Application Development Tools for Windows:  
A Comparison Between  
Delphi and Visual Basic

### Table Of Contents

Introduction  
Windows Visual Development  
Performance  
Benchmarks  
Benchmark Results  
Rapid Application Development (RAD)  
Controls  
Control Icon Array  
Templates and Experts  
Object Placement  
Property Lists  
Code Window  
Debugging And Object Inspecting  
Component Reuse  
Shared Event Functions  
Reusable Functions and Libraries  
Components  
Programming Language  
OOP Design Methodology  
Database Scalability  
Moving Up to Delphi  
Conclusion

Introduction



With the growing demands on Windows applications developers to create increasingly complex applications in less time, the evolution of Rapid Application Development (RAD) tools has become a crucial focus of the development community. First-generation RAD tools for Windows included application frameworks (such as OWL and MFC), DLL-based class libraries, and VBXs (Visual Basic custom controls).

The release of Borland's Delphi heralds a new generation of RAD tools that combines the power of traditional 3GL compilers with the ease of use and development speed of a 4GL environment. This white paper will contrast the approach of the new technology used by Delphi with that of Microsoft's Visual Basic.

## Windows Visual Development

The earliest methodology for Windows applications programming was to code directly using calls to the Windows API. The API provided a crude mechanism for creating such items as menus and windows, leaving the developer with an enormous coding task even when creating a rudimentary application. These early programs were typically created with C compilers that were equally crude by today's standards. As a result, the expectations for what could reasonably be achieved by professional developers was severely limited by schedule and performance constraints. Early applications were also, for the most part, independent. In other words, they were generally not reliant on each other and did not share data or invoke other applications.

Appreciating the underlying potential of the Windows environment, the development community demanded better tools, to enhance their productivity and facilitate the creation of more sophisticated applications. As Windows evolved, inter-application communications using DDE and OLE was introduced, then more powerful application frameworks products such as Borland's OWL and Microsoft's MFC appeared, as well as third-party products such as zApp from Zinc and Island Systems' object-Menu.

These libraries were used to encapsulate the most common functions of Windows applications, as well as to leave room for expansion and customization by developers. Thus, a programmer could quickly create a window with a certain border style, make it modal and add a "Close" button, then invoke it with a single call. Further, the advent of C++ compilers for Windows allowed experienced developers to exploit the power of object-oriented

technology. Developers now had the means to create complex applications within acceptable schedules.

Object-oriented languages allowed developers to create classes and override specific virtual functions, providing a direct path to building custom libraries. This generation of tools still had two major shortcomings:

1. Although productivity had been significantly increased, schedules for complex applications development were still quite lengthy. For example, the common development scenario would proceed as follows:
  - a) some sample screens would be created in a prototype environment or a resource tool;
  - b) marketing would critique the screen design and modifications would be made;
  - c) the "final" screens would be integrated with code to complete the application;
  - d) any changes that were requested often became complicated and painful, since the code and the screens were so closely linked.
2. The expertise required made Windows programming the sole domain of the experienced developer. In other words, the extensive needs of the corporate community simply could not be met due to the requirement for significant programmer expertise.

Thus evolved the next stage of Windows development, characterized by 4GL visual design environments such as Microsoft Visual Basic and high-end products such as Powersoft's PowerBuilder. These environments provided a major step forward in user-friendly development, with high-level, reusable components called controls which introduced the concept of a "building block" approach to software development. Most of the application development effort could now occur within a visual design tool where the programmer would piece together an interface from a suite of available libraries. Customization of the components can be accomplished by modifying a corresponding collection of properties sheet. Any "work" to be done within the application is triggered via events that affect the interface components (mouse clicks, keyboard entries and so on). With these first-generation visual development tools, the specification of the actions to be taken on these events is defined using a Basic-like scripting language.

As proven by their broad popularity, these tools went a long way toward solving some of the problems of corporate developers. However, there remained a serious deficit in their capabilities, due to their reliance on the visual design process for creating the application and also their underlying interpreted languages.

As the demands of Windows applications buyers continued to grow, developers were stretching the limits of the existing technologies to create projects such as mission-critical client/server applications. Team development and software quality assurance issues were becoming prominent. Applications were being designed as a series of modules that would need to interact seamlessly, and capable of communicating with and invoking other applications. For example, a user may have a need to insert a graphic in the context of an application. The graphic would be found by accessing a database created by some other application potentially residing over a network on a remote system. All of this needed to be transparent to the application at hand, so that users need not be concerned with where the graphic came from or how it was created.

Specifically, the following issues thus became crucial to professional applications developers:

- \* Performance
- \* Rapid Application Development
- \* Component Reuse
- \* Database Scalability

The next logical step in this evolution is technology that combines the significantly enhanced productivity of modern RAD tools with the power and flexible architecture of proven 3GL compiled languages. The remainder of this paper will contrast how Delphi and Visual Basic address these four key criteria for a robust Windows development system.

It is assumed that readers of this paper are familiar with RAD design concepts, but a detailed knowledge of Visual Basic or Delphi is not necessary in order to understand this paper.

## Performance

Performance of deployed applications is a key issue in today's highly competitive software market. Particularly for large,

distributed client/server applications, any shortfalls in execution speed become far more apparent, due to higher overall system demands.

Delphi is based upon Object Pascal (a significant extension of the popular Borland Pascal 7.0) whereas Visual Basic uses Microsoft Basic as its underlying language. Delphi's performance is significantly better, simply because it generates compiled executable files, while Visual Basic produces semi-interpreted code. That is, Delphi is built around an optimizing native code compiler instead of the slower interpreted p-code used by products such as Visual Basic. This results in Delphi applications executing 10 to 20 times faster than interpreted code. Delphi's intelligent linker also enables segment optimization, thereby reducing executable file size by as much as 30 percent, which enables faster loading and additional performance gains.

Delphi can compile standalone executable files (.EXEs) as well as reusable Dynamic Linked Libraries (DLLs). For the ultimate in execution speed, Delphi also allows professional programmers to go one step further by writing in-line assembler code, for direct control of the microprocessor.

Other areas in which Delphi displays considerable performance gains over Visual Basic is in database connectivity. The database layer of Visual Basic is implemented via ODBC, as opposed to the more efficient Borland Database Engine used in Delphi (and other core Borland development tools). However, Delphi also supports links to data via ODBC drivers. The high-performance native SQL Links supplied with Delphi Client/Server also outperform comparable Visual Basic SQL connectivity options.

## Benchmarks

Delphi's superior performance over Visual Basic becomes immediately apparent when running a few simple benchmarks. Consider the following examples, where a database is filled with items of text representing lastname, firstname, phone and street information. The phone number field is filled with consecutive integers, then the database is re-read and filled with a global array of integers from the phone number field. Finally, the global array is sorted to become reverse-ordered using a comparatively slow bubble sort algorithm.

Similar code can be written in both Delphi and Visual Basic, with the stages of the benchmarks summarized in the following code fragments:

VB - Fill

-----

```
Sub btnFill_Click ()
Dim k As Integer
MaxArray = EdArraySize.Text
For k = 1 To MaxArray
    Data1.Recordset.AddNew
    Data1.Recordset("LastName") = "Smith " + Str(k)
    Data1.Recordset("FirstName") = "Joe " + Str(k)
    Data1.Recordset("Phone") = Str(k)
    Data1.Recordset.Update
Next k
Data1.Recordset.MoveLast
End Sub
```

VB - Read

-----

```
Sub btnSearch_Click ()
Dim k As Integer
Dim n As Integer
Dim s As String
Data1.Recordset.MoveFirst
For k = 1 To MaxArray
    s = edPhone.Text
    n = Val(s)
    Call AppendArray(k, n)
    Data1.Recordset.MoveNext
Next k
End Sub
```

VB - Sort

-----

```
Sub btnSort_Click ()
Dim j As Integer
Dim k As Integer
Dim tmp As Integer
For j = 1 To MaxArray - 1
    For k = 1 To MaxArray - j
        If GlobArray(k) < GlobArray(k + 1) Then
```

```

    ' Swap GlobArray[k+] with GlobArray[k] ...
    tmp = GlobArray(k + 1)
    GlobArray(k + 1) = GlobArray(k)
    GlobArray(k) = tmp
  End If
Next k
Next j
End Sub

```

Delphi - Fill

-----

```

procedure TForm1.Button4Click(Sender: TObject);
var
  k,err: integer;
  s: string;
begin
  val(edDBsize.Text,maxArray,err);
  for k:=1 to maxArray do
    with Table1 do
      begin
        str(k,s);
        Append;
        FieldByName('Lastname').AsString := 'NewGuy'+s;
        FieldByName('Firstname').AsString := 'Paul'+s;
        FieldByName('Phone').AsString := s;
        Post;
      end
    end;
end;

```

Delphi - Read

-----

```

procedure TForm1.btnSearchTestClick(Sender: TObject);
var
  s: string;
  n,err,k: integer;
begin
  val(edDBsize.Text,MaxArray,err);
  Table1.First;
  for k:=1 to MaxArray do
    begin
      s := DBedPhone.EditText;
      val(s,n,err);
      AppendArray(k,n);
      Table1.Next;
    end
  end;
end;

```

```
end;  
end;
```

## Delphi - Sort

-----

```
procedure TForm1.btnSortArrayClick(Sender: TObject);  
var  
  j,k,tmp: integer;  
begin  
  for j:=1 to MaxArray-1 do  
    for k:=1 to MaxArray-j do  
      if GlobArray[k] < GlobArray[k+1] then  
        begin  
          { Swap GlobArray[k+] with GlobArray[k] ... }  
          tmp := GlobArray[k+1];  
          GlobArray[k+1] := GlobArray[k];  
          GlobArray[k] := tmp;  
        end;  
      end;  
    end;  
  end;
```

## Benchmark Results

The following table shows the results for database tables ranging in size from 100 to 4000 records. The test stages Fill, Read and Sort correspond the code sections described on the previous pages.

(All benchmark times are in seconds.)

D = Delphi  
FL = Fill  
RD = Read  
X = x Sort  
ST = Sort

# items	D	VB	D	VB	D	X	VB	D	X Total	VB
	FL	FL	RD	RD	ST		ST	Total		Total
	--	--	--	--	--		--	-----		-----
100	2	2	30	1	0	0	0	2	1.5	3
1000	16	70	6	23	1	22	22	23	5	115
2000	33	141	12	46	4	21	84	49	5.5	271
3000	50	227	17	69	8	23.6	189	76	6.4	485
4000	67	297	23	77	15	19.6	294	106	6.3	668

As can be seen from the results, the resulting Delphi-generated code outperformed the Visual Basic routines, especially in code-bound portions such as the Sort stage, by about 20 times faster. Delphi's database access functionality was also shown to be about five times more efficient than that of the Visual Basic code.

### Rapid Application Development (RAD)

The other side of the performance issue relates to the speed of application development, which is crucial for programmers intent on ensuring the fastest time to market for their products. The RAD features of an environment are the key to establishing how easy it is for programmers to progress from initial design and prototyping through to final implementation and deployment.

A modern RAD environment provides developers with several elements that significantly speed the development process over the traditional sequential coding approach. These include:

- \* A visual design environment;
- \* High-level building block components (often called "controls");
- \* Contextual access to code segments directly, via objects. In other words, homing in on the specific code relating to a particular object.



Under Windows, the structure of an application is frequently molded around its graphical user interface (GUI), with the behavior of the application triggered by various Windows messages or events. The methodology for RAD flows according to the following outline:

- 1) The developer creates an empty window or form to contain the application's interface components;
- 2) The developer selects a component from a pool of available components, which are generally displayed as an array of icons. Components are then placed and sized on the form;
- 3) Relevant properties are set or adjusted for each component, according to the application's requirements;
- 4) Code is written and "attached" to all relevant events for each component;
- 5) The application is run within the development environment;
- 6) The developer can then continue to modify the form design or underlying code until the final working application is completed.

Both Visual Basic and Delphi subscribe to this general methodology, making the products appear deceptively similar. However, there are several key enhancements that Delphi adds to this process, including:

- \* More built-in controls Enhanced icon layout, via a fully-customizable, multi-page (tabbed) component palette;
- \* Extensive gallery of extensible project templates and experts;
- \* Enhanced object placement capabilities;
- \* Enhanced modification of property lists;
- \* Two-way, synchronized code window;
- \* Shared event functions;

\* Integrated graphical debugging and object inspection

## Controls

Visual Basic custom controls are referred to as VBXs, and a limited selection is supplied with Visual Basic itself. Additional controls are sold by third-party manufacturers, although these not only cost additional money but also extend the overall learning curve, due to variations in product styles. To utilize controls for a tabbed folder, notebook, database grid or 3D list box, for example, Visual Basic owners must obtain third-party VBXs. Some of the controls supplied with Visual Basic suffer from memory and other limitations, making it necessary to purchase third-party alternatives.

Delphi's Visual Component Library (VCL) is a comprehensive suite of high-performance controls that support all standard Windows functionality, along with additional features such as tabbed folders, notebooks, database grids and 3D list boxes. Delphi also supports third-party VBXs, providing access to a wide range of third-party components.

## Control Icon Array

The Visual Basic control display is an array of icons with pictorial representations that are not always intuitive. In other words, developers can be left searching for the Image Control, for example, amongst many other icons with a similar look. Developers must then place a control into the form to be certain as to its identity. The Visual Basic control icon array can quickly become unwieldy as additional third-party custom controls are acquired. Since these icons are organized as a configurable rectangle, developers working with a large set of VBX controls are forced to give up valuable screen space or sacrifice accessibility of some controls.

Delphi solves these component layout problems with several enhancements. Firstly, Delphi's component palette is organized with several tabbed notebook pages, displaying icons in a single-row, scrollable toolbar format. This keeps the display uncluttered, yet fully accessible. More importantly, Delphi's customization options allow full configuration of the grouping, placement and display of components, so that the environment can be fine-tuned to suit the working style of an individual or development team. To address the problem of obscure or

similar-looking icons, Delphi offers "fly-by help" showing the purpose of the control associated with the icon as the cursor is dragged over it.

## Templates and Experts

Delphi includes pre-built templates that make it easy to develop standard applications or complex components such as MDI windows, database forms, multi-pages dialog and dual list boxes. The architecture is fully extensible, allowing developers to easily register their own custom templates and experts into the gallery.

## Object Placement

Delphi facilitates visual design with features such as automatic object alignment, sizing and scaling, while Visual Basic supports placement only. Delphi's automatic alignment also speeds up the creation of aesthetic forms.

## Property Lists

A subtle yet significant distinction between the two development tools can be seen in the means of accessing property lists. Visual Basic users access a pull-down selection of options for a particular property via an entry bar at the top of the list, so that changing several property items, requires selection of the item, clicking on the entry bar to make the change, then clicking on the next item, and so forth. Delphi provides pull-down lists that can be accessed directly alongside the property value, making for more efficient and intuitive modifications.

## Code Window

Delphi's code editing window synchronizes all visual design representations with the underlying source code. In other words, as the application is constructed by dropping objects into a form, the corresponding bug-free code is generated simultaneously. There are no limitations, since the code is always accessible, and developers can instantly switch between the code editor and the visual design tools, allowing them to select the most efficient mode for each part of the project.

## Debugging and Object Inspecting

Visual Basic provides program debugging capabilities such as

variable watches and a call stack monitor. However, this functionality is limited in that it cannot break on a specific condition, and the call stack is modal, so it cannot be viewed during the entire debug session.

Delphi provides a full-featured debugger with conditional breakpoints and a modeless call stack viewer. The debug window and viewers can be saved from session to session, allowing developers to create a comfortable custom environment. Delphi also includes a powerful object browser similar to that used within Borland C++, which provides a comprehensive display of code objects and classes - including the capability to trace object lineage (inheritance, children) and virtual procedures.

## Component Reuse

One of the most significant advances in applications development methodology is the concept of creating an application from high-level components. By linking predefined building blocks, developers need only define the "glue" between objects that specifies the unique qualities of an application, with the potential for substantial productivity gains. Although Delphi and Visual Basic both provide various ways to reuse and share components and code, Delphi again delivers a cleaner and simpler solution.

The issue of reuse can be viewed in three areas:

- \* shared event functions
- \* reusable functions
- \* reusable building blocks (components)

## Shared Event Functions

A common problem encountered in Windows programs is how to share a function that is executed upon the occurrence of several Windows events. Although the implementation is similar in both Delphi and Visual Basic, the Delphi solution has some obvious advantages. In Visual Basic, shared functions must be placed in the local code file or in a global .BAS file if the function is to be shared. The problem with this is that the function is now global to the entire project. In contrast, Delphi allows the function to be placed in the local file or in a DCU (Pascal unit

file) which must be explicitly referenced only by the files that use it.

## Reusable Functions and Libraries

In Visual Basic, common functions or libraries are accomplished by use of a global .BAS file, which then makes the functions accessible to every file in the project. The disadvantage to this approach is that the shared functions must be global to every file. Alternatively, Visual Basic can take advantage of functions organized in a DLL, but DLLs (like VBXs) must be created by another development tool external to Visual Basic, which requires a different level of expertise and an additional learning curve. All libraries for Delphi can be created from within the Delphi environment. Pascal code is organized as units, and shared functions are accessible through a Pascal unit by simply referencing the "library" unit that contains the desired function. Delphi can also use and create high-performance Windows DLLs. Further, Delphi's underlying programming language allows developers to reuse and customize functionality within a class via subclassing (see further details of OOP methodology below).

## Components

VBXs can be developed for Visual Basic with functionality that is usable across different projects, but a significant disadvantage of VBXs is the complexity involved in creating them. There is a detailed set of restrictions associated with creation of a VBX such that they cannot be created within Visual Basic itself. Instead, the most common method to create a VBX is to use a C/C++ compiler to create a DLL and then put a VBX "wrapper" around it. The advantage of this is the speed of computation gained by using optimized C/C++ compiled code over the Visual Basic's interpreted technology. The disadvantage is that developers are forced to "switch gears" in order to work with the compiler, and the added complexity can lead to additional troubleshooting and debugging time.

Delphi components are more easily created. Unlike Visual Basic, where VBXs must be built using an external compiler, Delphi components are built within the Delphi development environment itself. This is an important distinction, because professional developers prefer to work with a consistent set of tools. Being able to use Delphi to create reusable components

becomes a major productivity enhancement, enabling more rapid development with the added benefits of reusability.

Additionally, since Delphi components are created with Delphi's optimizing native code compiler and linker, there is improved performance over traditional VBXs. One other considerable advantage of Delphi components is that developers can subclass the functionality of a component to create their own custom versions. If a specific VBX is insufficient for a Visual Basic user, the only alternative is to build (or purchase) an alternative VBX.

## Programming Language

An obvious difference between Visual Basic and Delphi is the underlying programming language. The use of Object Pascal within Delphi has several important repercussions:

- \* Pascal is a more powerful and structured language than Basic.
- \* Object Pascal is a true object-oriented programming language, providing the benefits of inheritance, encapsulation and polymorphism;
- \* Pascal is a compiled language, ensuring high-performance executables;
- \* The organization of files as DCUs provides a cleaner mechanism for creating libraries of reusable code (see Shared Event Code);
- \* Object Pascal utilizes the world's fastest commercial compiler technology;
- \* Object Pascal support in-line assembler code for maximum performance;

One final point of differentiation is that in Visual Basic, all code files must be specifically associated with a form, except for a global .BAS file. In other words, a function must be global to the entire project unless associated with a form. In Delphi, however, code files (and therefore classes and functions) can be disassociated with any form, allowing proper scoping of functions without any loss of functionality.

## OOP Design Methodology

The power and flexibility of an object-oriented design methodology is widely accepted as the best way to solve complex, real-world programming problems. Object-oriented design provides both a solid foundation and elegant architecture for an application. Some of the benefits of OOP are:

- \* Shorter development cycles;
- \* Code that is highly maintainable;
- \* Code that is easily shared with other modules or other projects;
- \* Facilitation of team programming and version control;
- \* By exploiting object inheritance and polymorphism, the coding process can become much simpler and the code itself significantly more coherent;
- \* Applications can incorporate several functions that are mostly similar but have certain distinct "personality traits".

Object Pascal is a structured, object-oriented programming language, providing full support for class architectures, inheritance, virtual functions and polymorphism. Visual Basic is not an object-oriented language. Note that although developers need not be familiar with object-oriented concepts to create programs using Delphi, professional programmers will appreciate the benefit of these capabilities.

## Database Scalability

A good RAD environment must address the pervasive issue of creating a database application, and Visual Basic and Delphi are no exceptions. In Visual Basic, developers can place a database component onto a form which can then have a property set that allows it to bidirectionally communicate with an ODBC-compatible database. The database component can be used as a crude mechanism to navigate through the database using arrows representing first, next, previous and last records. SQL queries can also be defined in code, to form a query snapshot into the database for viewing

or computation. Crystal Reports is shipped with Visual Basic, providing a report generating capability. Setup of the database structures, the associated forms, interaction between them and most of the navigation through the database must all be done explicitly via the visual design tools or within code.

Delphi includes extensive database support including the Borland Database Engine (BDE) for Paradox and dBASE access, and middleware layers that support local and remote SQL data access. The Borland database architecture provide developers with high-performance access to a variety of data sources including ODBC drivers. Delphi includes data access components and data-aware user interface components to provide a comprehensive database solution.

Delphi ships with several controls for data entry and display, including tables and grids. The grid control (TDBGrid Component) can be used to build a spreadsheet-style of application. A unique characteristic of the database grid control is that it can be linked to multiple database sources.

Delphi also includes wizards and experts that facilitate rapid design and implementation of databases and the corresponding user interface. The DataSet designer facility included with Delphi allows developers to rapidly create table or query data for database components. It is a simple matter to specify which set of fields from the database must be incorporated into the table or query.

When designing a database grid, an application often needs an editor to allow in-place modification of field data. Delphi's DBEdit provides a consolidated component to handle this task. Grid-aware, specialized versions of the control are available for labels, lists, combo boxes, images, memo (multi-line editors), check and radio buttons, lookup lists and lookup combo lists.

Delphi also features built-in support for queries and reports. A query control (TQuery Component) provides the ability to perform SQL queries in order to form the data set corresponding to the filtered elements of a database. If this data was extracted from dBASE or Paradox, you would also have the ability to modify, insert or delete records. By placing this component into a form that also contains the database component, developers can create a filtered, printable report based on some SQL or query into the data set. Delphi includes the award-winning ReportSmith report



writer for PC and SQL databases. ReportSmith provides an intuitive interface for report creation using live data at design time, and it supports queries, crosstabs, templates, calculations and unlimited report sizes.

## Moving Up to Delphi

Visual Basic developers who may be considering migrating their applications to Delphi, you may be concerned about the effort required to migrate existing Visual Basic applications in order to continue project development and maintenance within Delphi. Project migration is actually a fairly straightforward process. A conversion utility is available from EarthTrek, Inc (617) 273-0308 that performs most of the translation including project files, form files and code translation. The utility completes all of the possible automatic translation leaving some ambiguous language elements to be identified by a simple syntax check using the Delphi compiler. Many projects can be translated with virtually no effort. Others may require a few hours of post-work to complete.

## Conclusion

When examining the various RAD products in the marketplace, both Visual Basic and Delphi stand out as leading edge products. However, Delphi has clearly emerged as a next generation tool with its higher performance, highly facilitated visual design capability, extensive support for reusable components and database readiness. Delphi achieves its goals as a powerful application development system by combining a state-of-the-art visual design environment with the power, flexibility and reusability of a fully object-oriented language, the world's fastest compiler, and leading-edge database technology. Further, through the integration of Object Pascal, Delphi empowers developers with a full-featured programming environment without sacrificing rapid visual development thus allowing construction of sophisticated client-server applications in record time.

## Reference:

7/16/98 4:33:52 PM



## Article #15691: INTERRUPT HANDLER RULES

Technical Notes Database

TN691D.txt      INTERRUPT HANDLER RULES

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. What rules should I follow when writing an interrupt handler?

A. The following is a list of rules to keep in mind when writing an interrupt handler:

1. Use GetIntVec and SetIntVec to install/un-install interrupt handlers.
2. Use the interrupt keyword.
3. Be careful about reentrancy. Don't use any calls to DOS or to Turbo Pascal's overlay or heap management routines in your interrupt handler.
4. Interrupt procedures and functions must use the far call model (use the {\$F+} option).
5. Be proficient with the BIOS and Assembly language before attempting to write an interrupt handler.
6. Make sure your interrupt handler is not in an overlaid unit.
7. You will want to create an exit procedure to un-install the interrupt service routine.

Reference:

7/16/98 4:35:34 PM

Last Modified: 01-SEP-99

## Article #15695: Configuration of an ODBC Driver with BDE v3.5 and less

Technical Information Database

TI695D.txt - Configuration of an ODBC Driver with BDE v3.5 and less

Category :Delphi 1.x  
Platform :All-32Bit  
Product :

Description:

CONFIGURING AN ODBC DRIVER AND ALIAS FOR DELPHI

-----

CONTROL PANEL'S ODBC OPTION

Delphi installs an ODBC option to the Windows Control Panel program. The ODBC option indicates the available data sources (drivers) installed for use by ODBC. As you will find by selecting the ODBC option, a number of formats are installed with Delphi and are seen in the main window titled Data Sources. Additional formats may be supported by the drivers installed and can be configured by selecting the Add... button.

If a new driver is to be added or removed,

1. Select the Drivers... button from the Data Sources Window. From the drivers dialog, select the Add... button and provide the path where the ODBC driver will be found.
2. Return to the Data Sources Windows and include the possible data sources available through the new driver by selecting its Add... button.
3. To configure options available for a particular data source use the Setup... button. The function of the Setup... button will vary with each data format. Very often options like the working directory for the driver are configured in this area.

Online help is available for each dialog involved with the ODBC option.

## BDE CONFIGURATION UTILITY

After installing the ODBC driver, run the BDE Configuration utility to configure the database engine to use the new driver.

1. From the drivers page, select the New ODBC driver button.
2. A dialog titled Add ODBC driver will appear. The option for SQL link driver is what will distinguish the databases created using this ODBC driver.
3. Next select the default ODBC driver. Dropping down the list from the combobox will reveal the file types supported by ODBC drivers installed on the system.
4. Select the default data source for the ODBC driver. Having set the ODBC driver in step 3 above, the list of this combobox will have the data source names appropriate for use with the selected driver.
5. Select Ok.
6. Returning to the drivers page, select File/Save from the main menu to save this configuration.

## CREATING AN ALIAS IN THE DATABASE DESKTOP

While this can be done from the BDE Configuration utility, it is more convenient overall to create ODBC aliases from the Database Desktop.

1. From the File menu, select Aliases...
2. From the resulting Alias Manager dialog, select New.
3. Type the name for your new alias in the area labeled Database Alias.
4. Use the drop down list of the Driver Type combobox to select the driver appropriate for this alias. Paradox and dBase tables are considered STANDARD. If the ODBC driver was properly configured in the BDE Configuration utility it's name will appear in this list.

5. Additional options may appear depending upon the driver type you select.

6. When finished, select Keep New to store the new alias. Then select Ok. You will be prompted for whether or not to save the aliases to IDAPI.CFG. Select Okay.

The alias will now be usable from both the Database Desktop and Delphi.

Reference:

4/22/99 12:45:11 PM

Last Modified: 01-SEP-99

## Article #15697: FILE HANDLES USED BY TURBO PASCAL

Technical Notes Database

TN697D.txt      FILE HANDLES USED BY TURBO PASCAL  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 5.0 5.5

### Description:

Q. What file handles are used by Turbo Pascal?

A. Pascal uses a file handle for the overlays, reading the BGI driver, the Rename and Erase procedures. The other file handles are used by DOS: Standard Input, Standard Output, Standard Error, Standard Auxiliary, and Standard Printer. All except these five standard file handles are returned immediately.

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15701: Creating Database Aliases in Code

Technical Information Database

TI701D.txt      Creating Database Aliases in Code

Category      :Database Programming

Platform      :All

Product       :Delphi 1.0

### Description:

This Technical Information document will help step thru concepts regarding the creation and use of ALIASES within your Delphi Applications.

Typically, you use the BDE Configuration Utility BDECFG.EXE to create and configure aliases outside of Delphi. However, with the use of the TDatabase component, you have the ability to create and use this ALIAS within your application-- not pre-defined in the IDAPI.CFG.

The ability to create Aliases that are only available within your application is important. Aliases specify the location of database tables and connection parameters for database servers. Ultimately, you can gain the advantages of using ALIASES within your applications-- without having to worry about the existence of a configuration entry in the IDAPI.CFG when you deploy your application.

### Summary of Examples:

----- -- -----

#### Example #1:

Example #1 creates and configures an Alias to use STANDARD (.DB, .DBF) databases. The Alias is then used by a TTable component.

#### Example #2:

Example #2 creates and configures an Alias to use an INTERBASE database (.gdb). The Alias is then used by a TQuery component to join two tables of the database.

#### Example #3:

Example #3 creates and configures an Alias to use STANDARD (.DB, .DBF) databases. This example



demonstrates how user input can be used to configure the Alias during run-time.

Example #1: Use of a .DB or .DBF database (STANDARD)

1. Create a New Project.
2. Place the following components on the form:
  - TDatabase, TTable, TDataSource, TDBGrid, and TButton
3. Double-click on the TDatabase component or choose Database Editor from the TDatabase SpeedMenu to launch the Database Property editor.
4. Set the Database Name to 'MyNewAlias'. This name will serve as your ALIAS name used in the DatabaseName Property for dataset components such as TTable, TQuery, TStoredProc.
5. Select STANDARD as the Driver Name.
6. Click on the Defaults Button. This will automatically add a PATH= in the Parameter Overrides section.
7. Set the PATH= to C:\DELPHI\DEMOS\DATA (PATH=C:\DELPHI\DEMOS\DATA)
8. Click the OK button to close the Database Dialog.
9. Set the TTable DatabaseName Property to 'MyNewAlias'.
10. Set the TDataSource's DataSet Property to 'Table1'.
11. Set the DBGrid's DataSource Property to 'DataSource1'.
12. Place the following code inside of the TButton's OnClick event.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Table1.Tablename:= 'CUSTOMER';
    Table1.Active:= True;
end;
```

13. Run the application.

\*\*\* If you want an alternative way to steps 3 - 11, place the following code inside of the TButton's OnClick event.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Database1.DatabaseName:= 'MyNewAlias';
    Database1.DriverName:= 'STANDARD';
    Database1.Params.Clear;
```

```

Database1.Params.Add('PATH=C:\DELPHI\DEMOS\DATA');
Table1.DatabaseName:= 'MyNewAlias';
Table1.TableName:= 'CUSTOMER';
Table1.Active:= True;
DataSource1.DataSet:= Table1;
DBGrid1.DataSource:= DataSource1;
end;

```

\*\*\*\*\*

## Example #2: Use of a INTERBASE database

1. Create a New Project.
2. Place the following components on the form:
  - TDatabase, TQuery, TDataSource, TDBGrid, and TButton
3. Double-click on the TDatabase component or choose Database Editor from the TDatabase SpeedMenu to launch the Database Property editor.
4. Set the Database Name to 'MyNewAlias'. This name will serve as your ALIAS name used in the DatabaseName Property for dataset components such as TTable, TQuery, TStoredProc.
5. Select INTRBASE as the Driver Name.
6. Click on the Defaults Button. This will automatically add the following entries in the Parameter Overrides section.

```

SERVER NAME=IB_SERVER:/PATH/DATABASE.GDB
USER NAME=MYNAME
OPEN MODE=READ/WRITE
SCHEMA CACHE SIZE=8
LANGDRIVER=
SQLQRYMODE=
SQLPASSTHRU MODE=NOT SHARED
SCHEMA CACHE TIME=-1
PASSWORD=

```

7. Set the following parameters

```

SERVER NAME=C:\IBLOCAL\EXAMPLES\EMPLOYEE.GDB
USER NAME=SYSDBA
OPEN MODE=READ/WRITE
SCHEMA CACHE SIZE=8
LANGDRIVER=
SQLQRYMODE=
SQLPASSTHRU MODE=NOT SHARED
SCHEMA CACHE TIME=-1

```

PASSWORD=masterkey

8. Set the TDatabase LoginPrompt Property to 'False'. If you supply the PASSWORD in the Parameter Overrides section and set the LoginPrompt to 'False', you will not be prompted for the password when connecting to the database. WARNING: If an incorrect password is entered in the Parameter Overrides section and LoginPrompt is set to 'False', you are not prompted by the Password dialog to re-enter a valid password.
9. Click the OK button to close the Database Dialog.
10. Set the TQuery DatabaseName Property to 'MyNewAlias'.
11. Set the TDataSource's DataSet Property to 'Query1'.
12. Set the DBGrid's DataSource Property to 'DataSource1'.
13. Place the following code inside of the TButton's OnClick event.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Query1.SQL.Clear;
    Query1.SQL.ADD(
        'SELECT DISTINCT * FROM CUSTOMER C, SALES S
        WHERE (S.CUST_NO = C.CUST_NO)
        ORDER BY C.CUST_NO, C.CUSTOMER');
    Query1.Active:= True;
end;
```

14. Run the application.

### Example #3: User-defined Alias Configuration

This example brings up a input dialog and prompts the user to enter the directory to which the ALIAS is to be configured to.

The directory, servername, path, database name, and other necessary Alias parameters can be read into the application from use of an input dialog or .INI file.

1. Follow the steps (1-11) in Example #1.
2. Place the following code inside of the TButton's OnClick event.

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  NewString: string;
  ClickedOK: Boolean;
begin
  NewString := 'C:\';
  ClickedOK := InputQuery('Database Path',
    'Path: --> C:\DELPHI\DEMOS\DATA', NewString);
  if ClickedOK then
  begin
    Database1.DatabaseName:= 'MyNewAlias';
    Database1.DriverName:= 'STANDARD';
    Database1.Params.Clear;
    Database1.Params.Add('Path=' + NewString);
    Table1.DatabaseName:= 'MyNewAlias';
    Table1.TableName:= 'CUSTOMER';
    Table1.Active:= True;
    DataSource1.DataSet:= Table1;
    DBGrid1.DataSource:= DataSource1;
  end;
end;
```

3. Run the Application.

-----  
See Also:

Delphi On-line help -->  
    Database Properties Editor  
    TDatabase

Reference:

7/16/98 4:33:52 PM

Last Modified: 01-SEP-99

## Article #15703: FINDFIRST - THE EXISTENCE OF FILES

Technical Notes Database

TN703D.txt      FINDFIRST - THE EXISTENCE OF FILES

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 4.0+

Description:

Q. How can I check to see if a File Exists?

A. The following function will return whether or not  
a file exists:

Uses DOS;

```
function Exist( Name: PathStr ):boolean;
```

```
Var
```

```
  Search: SearchRec;
```

```
Begin
```

```
  FindFirst( Name, AnyFile and (Not Directory), Search );
```

```
  Exist := DosError = 0;
```

```
End;
```

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15707: Creating & Deleting TFields at run-time

Technical Information Database

TI707D.txt     Creating & Deleting TFields at run-time  
Category     :Database Programming  
Platform     :All  
Product      :Delphi All

### Description:

TField components (or more appropriately, descendants of the TField component of types corresponding to field types) can be created at design-time using the Fields Editor. The Fields Editor is invoked by double-clicking on the design icon for a TTable or TQuery component. But TField descendants can also be created and deleted at run-time.

Descendants of the TField component (such as TStringField, TIntegerField, etc.) are created by invoking the Create method for the type of TField descendant appropriate to the field in the data set. That is, the Create method for the TStringField descendant of TField would be called to create a TField descendant for a string-type field in the current data set. The Create method requires one parameter, that of the owner of the TField descendant, which is the containing TForm. After creating the TField descendant component, a number of key properties need to be set in order to connect it with the field in the data set. These are:

FieldName: the name of the field in the table.  
Name: a unique identifier for the TField descendant component.  
Index: the TField descendant component's position in the array of TFields (the Fields property of the TTable or TQuery with which the TField will be associated).  
DataSet: the TTable or TQuery with which the TField will be associated.

The code snippet below demonstrates creating a TStringField. The containing TForm is called Form1 (referred to here with the Self variable), the active data set is a TQuery named Query1, and the field for which the TStringField component is being created is a dBASE table field named CO\_NAME. This new TField descendant will be the second TField in the Fields array property of Query1. Note that the data set with which the new TField descendant will be associated (in this case, Query1) must be closed prior to adding the TField and then reopened afterwards.

```
procedure TForm1.Button2Click(Sender: TObject);
var
    T: TStringField;
begin
    Query1.Close;
    T := TStringField.Create(Self);
    T.FieldName := 'CO_NAME';
    T.Name := Query1.Name + T.FieldName;
    T.Index := Query1.FieldCount;
    T.DataSet := Query1;
    Query1.FieldDefs.Update;
    Query1.Open;
end;
```

The example above example creates a new TStringField named Query1CO\_NAME.

Deleting an existing TField descendant is merely a matter of invoking the Free method for that component. In the example below, the TForm's FindComponent method is used to return a pointer to the TStringField component named Query1CO\_NAME. The return value for the FindComponent will either be of type TComponent if successful or nil if unsuccessful. This return value can be used to determine whether the component actually exists prior to invoking its Free method.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TC: TComponent;
begin
  TC := FindComponent('Query1CO_NAME');
  if not (TC = nil) then begin
    Query1.Close;
    TC.Free;
    Query1.Open;
  end;
end;
```

As with creating a TField, if the data set associated with the TField descendant is currently active, it must be closed or deactivated prior to invoking this method.

Reference:

7/16/98 4:33:52 PM

Last Modified: 01-SEP-99



## Article #15709: SEEK - RANDOM-ACCESS FILES

Technical Notes Database

TN709D.txt      SEEK - RANDOM-ACCESS FILES  
Category      :Pascal  
Platform       :All  
Product       :Turbo Pascal 3.0+

### Description:

Q. How can I change an item in a random-access file using the SEEK procedure?

A. Here is the example code:

```
..  
SEEK(FileVar, RecordNum);  
READ(FileVar, RecordNum);  
{ Code to change the record }  
SEEK(FileVar, RecordNum);  
WRITE(FileVar, RecordNum);  
..
```

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15713: How to iterate through the fields of a table

Technical Information Database

TI713D.txt     How to iterate through the fields of a table  
Category     :Database Programming  
Platform     :All  
Product     :Delphi All

### Description:

Getting a list of the fields in a table at run-time can be as simple as a call to the `GetFieldNames` method of the `TTable`, `TQuery`, or `TStoredProc` component. The `GetFieldNames` method returns a list of the fields that comprise the structure of the data set in the form of a `TStrings` list, which may be inserted into such visual components as a `TListBox` through its `Items` property:

```
ListBox1.Clear;  
Table1.GetFieldNames(ListBox1.Items);
```

Of course, the `TStrings` list returned by the `GetFieldNames` method need not be used with a visual component. It could just as well serve as an array of field names stored entirely in memory, that can be used as a list or array.

But it is also possible to retrieve much more information about the fields in a table than just the names. Other descriptive attributes include field types and sizes. Retrieving these values is slightly more involved than the use of the `GetFieldNames`. Basically, this process involves iterating through the `FieldDefs` property of the `TTable`, `TQuery`, or `TStoredProc` component. The `FieldDefs` property is essentially an array of records, one record for each field in the structure. Each field record contains information about the field, including its name, type, and size. It is a relatively straightforward process to iterate

through  
this array of field descriptions, extracting information about  
individual  
fields.

There are a number of reasons why a program might need to query the  
structure of a table used in the application. One reason is a prelude  
to  
creating TField components at run-time that represent the fields in  
the  
table. The information gleaned from the structure of the table form  
the  
basis of the TField components to be created.

The example below demonstrates how to iterate through the fields  
available  
in a TTable or TQuery. The example extracts information about the  
available  
fields and displays the information in a TListBox, but the same  
methodology  
can be used to provide information necessary for the dynamic building  
of  
TField descendants. The example uses a TTable as the data set, but a  
TQuery  
can be used in the same manner as both TTable and TQuery components  
incorp-  
orate the Field-Defs property the same way.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  F: TFieldDef;
  D: String;
begin
  Table1.Active := True;
  ListBox1.Items.Clear;
  with Table1 do begin
    for i := 0 to FieldDefs.Count - 1 do begin
      F := FieldDefs.Items[i];
      case F.DataType of
        ftUnknown: D := 'Unknown';
        ftString: D := 'String';
        ftSmallint: D := 'SmallInt';
        ftInteger: D := 'Integer';
        ftWord: D := 'Word';
```

```
ftBoolean: D := 'Boolean';
ftFloat: D := 'Float';
ftCurrency: D := 'Currency';
ftBCD: D := 'BCD';
ftDate: D := 'Date';
ftTime: D := 'Time';
ftDateTime: D := 'DateTime';
ftBytes: D := 'Bytes';
ftVarBytes: D := '';
ftBlob: D := 'BLOB';
ftMemo: D := 'Memo';
ftGraphic: D := 'Graphic';
else
  D := '';
end;
ListBox1.Items.Add(F.Name + ', ' + D);
end;
end;
Table1.Active := False;
end;
```

Reference:

7/16/98 4:33:52 PM

Last Modified: 01-SEP-99

## Article #15715: READ - SKIPPING OVER A READLN OR READ

Technical Notes Database

TN715D.txt      READ - SKIPPING OVER A READLN OR READ

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 2.0+

### Description:

Q. Why is my program not waiting for input on a Read or Readln?

A. The program has a Read statement. When the user enters information and hits a carriage return the information and the carriage return are stored in the keyboard buffer. The Read statement reads everything until the carriage return, but not the carriage return. When the program executes the next Read or Readln it will see the carriage return in the keyboard buffer. It will then process the return and continue with the next instruction. To the user it will appear that the program has skipped over the Read or Readln. To fix this problem never use Read to read information from the keyboard, always use Readln.

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15719: How to match file date/time stamps.

Technical Information Database

TI719D.txt     How to match file date/time stamps.

Category     :General Programming

Platform     :All

Product     :Delphi   All

Description:

Q: "How can I write a function that sets the date of one file equal to the date of another file?"

A: No problem. Just use the following function, which takes two strings representing full DOS path/file names. The file who's date you wish to set is the second parameter, and the date you wish to set it to is given by the file in the first parameter.

```
procedure CopyFileDate(const Source, Dest: String);
var
  SourceHand, DestHand: word;
begin
  SourceHand := FileOpen(Source, fmOutput);      { open source file }
  DestHand := FileOpen(Dest, fmInput);           { open dest file }
  FileSetDate(DestHand, FileGetDate(SourceHand)); { get/set date }
  FileClose(SourceHand);                         { close source
file }
  FileClose(DestHand);                           { close dest file }
end;
```

Reference:

7/16/98 4:33:52 PM

Last Modified: 01-SEP-99

## Article #15721: RESET - READING AND WRITING TYPED UNTYPED FILES

Technical Notes Database

TN721D.txt     RESET - READING AND WRITING TYPED UNTYPED FILES  
Category     :Pascal  
Platform     :All  
Product     :Turbo Pascal 4.0+

### Description:

Q. How can I write to a binary file without destroying the information already in it?

A. You must use the Reset procedure. It opens the file for BOTH reading and writing. Furthermore, the RecordSize\*Count cannot exceed the total size of the buffer. Therefore, you may have to use

```
Reset(FileVariable, 1) { RecordSize = 1 }.
```

Now, you may use the Seek procedure, to go to a particular position in the file, and the BlockWrite procedure to write over some data. The following example demonstrates the use of BlockRead and BlockWrite:

```
program ReadWriteFile;  
var  
  FromF : file;  
  NumRead, NumWritten: Word;  
  Buf : array[1..2] of Char;  
  Ch : Char;  
begin  
  { Open input file and create it }  
  Assign(FromF, 'MyFile.Dat');  
  { Record size = 1 }  
  Buf[1] := 'a'; Buf[2] := 'b';  
  Ch := 'c';  
  Rewrite(FromF, 1);  
  BlockWrite(FromF, Ch, SizeOf(Ch), NumWritten);  
  Close(FromF);  
  Reset(FromF, 1);  
  { Open output file and write to it and read from it }  
  BlockWrite(FromF, Buf, SizeOf(Buf), NumWritten);
```

```
BlockRead(FromF,Ch,SizeOf(Ch),NumRead);  
Close(FromF);  
end.
```

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99



## Article #15725: Adding Graphics in Your Listboxes and Comboboxes

Technical Information Database

TI725D.txt Adding Graphics in Your Listboxes and Comboboxes

Category :VCL

Platform :All

Product :Delphi All

Description:

TI: Inserting graphics in Owner Drawn ListBoxes and ComboBoxes

The ability to place graphics inside ListBoxes and ComboBoxes can improve the look of your application and set your user interface apart from the others.

Q: How do I stick graphics in a Listbox or ComboBox???

Here is an step-by-step example.....

1. Create a form.
2. Place a ComboBox and Listbox component on your form.
3. Change the Style property of the ComboBox component to `csOwnerDrawVariable` and the Style property of the ListBox to `lbOwnerDrawVariable`.

An Owner-Draw TListBox or TComboBox allows you to display both objects (ex. graphics) and strings as the items. For this example, we are adding both a graphic object and a string.

4. Create 5 variables of type TBitmap in the Form's VAR section.
5. Create a Procedure for the Form's OnCreate event.
6. Create a Procedure for the ComboBox's OnDraw Event.
7. Create a Procedure for the ComboBox's OnMeasureItem.

8. Free the resources in the Form's OnClose Event.

```
{START OWNERDRW.PAS}
unit Ownerdrw;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure ComboBox1DrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure ComboBox1MeasureItem(Control: TWinControl; Index:
Integer;
      var Height: Integer);
    procedure ListBox1DrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure ListBox1MeasureItem(Control: TWinControl; Index:
Integer;
      var Height: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  TheBitmap1, TheBitmap2, TheBitmap3, TheBitmap4,
  TheBitmap5 : TBitmap;
implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  TheBitmap1 := TBitmap.Create;
```

```

TheBitmap1.LoadFromFile('C:\delphi\images\buttons\globe.bmp');
TheBitmap2 := TBitmap.Create;
TheBitmap2.LoadFromFile('C:\delphi\images\buttons\video.bmp');
TheBitmap3 := TBitmap.Create;
TheBitmap3.LoadFromFile('C:\delphi\images\buttons\gears.bmp');
TheBitmap4 := TBitmap.Create;
TheBitmap4.LoadFromFile('C:\delphi\images\buttons\key.bmp');
TheBitmap5 := TBitmap.Create;
TheBitmap5.LoadFromFile('C:\delphi\images\buttons\tools.bmp');
ComboBox1.Items.AddObject('Bitmap1: Globe', TheBitmap1);
ComboBox1.Items.AddObject('Bitmap2: Video', TheBitmap2);
ComboBox1.Items.AddObject('Bitmap3: Gears', TheBitmap3);
ComboBox1.Items.AddObject('Bitmap4: Key', TheBitmap4);
ComboBox1.Items.AddObject('Bitmap5: Tools', TheBitmap5);
ListBox1.Items.AddObject('Bitmap1: Globe', TheBitmap1);
ListBox1.Items.AddObject('Bitmap2: Video', TheBitmap2);
ListBox1.Items.AddObject('Bitmap3: Gears', TheBitmap3);
ListBox1.Items.AddObject('Bitmap4: Key', TheBitmap4);
ListBox1.Items.AddObject('Bitmap5: Tools', TheBitmap5);

```

```
end;
```

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin

```

```

    TheBitmap1.Free;
    TheBitmap2.Free;
    TheBitmap3.Free;
    TheBitmap4.Free;
    TheBitmap5.Free;

```

```
end;
```

```

procedure TForm1.ComboBox1DrawItem(Control: TWinControl; Index:
Integer;

```

```

    Rect: TRect; State: TOwnerDrawState);

```

```
var
```

```

    Bitmap: TBitmap;
    Offset: Integer;

```

```
begin
```

```

    with (Control as TComboBox).Canvas do

```

```

        begin

```

```

            FillRect(Rect);

```

```

            Bitmap := TBitmap(ComboBox1.Items.Objects[Index]);

```

```

            if Bitmap <> nil then

```

```

                begin

```

```

                    BrushCopy(Bounds(Rect.Left + 2, Rect.Top + 2, Bitmap.Width,

```

```

        Bitmap.Height), Bitmap, Bounds(0, 0, Bitmap.Width,
        Bitmap.Height), clRed);
    Offset := Bitmap.width + 8;
end;
{ display the text }
TextOut(Rect.Left + Offset, Rect.Top, Combobox1.Items[Index])
end;
end;

procedure TForm1.ComboBox1MeasureItem(Control: TWinControl; Index:
    Integer; var Height: Integer);

begin
    height:= 20;
end;

procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index:
Integer;
    Rect: TRect; State: TOwnerDrawState);
var
    Bitmap: TBitmap;
    Offset: Integer;
begin
    with (Control as TListBox).Canvas do
    begin
        FillRect(Rect);
        Bitmap := TBitmap(ListBox1.Items.Objects[Index]);
        if Bitmap <> nil then
        begin
            BrushCopy(Bounds(Rect.Left + 2, Rect.Top + 2, Bitmap.Width,
                Bitmap.Height), Bitmap, Bounds(0, 0, Bitmap.Width,
                Bitmap.Height), clRed);
            Offset := Bitmap.width + 8;
        end;
        { display the text }
        TextOut(Rect.Left + Offset, Rect.Top, Listbox1.Items[Index])
    end;
end;

procedure TForm1.ListBox1MeasureItem(Control: TWinControl; Index:
Integer;
    var Height: Integer);
begin
    height:= 20;
end;

```

```

end.
{END OWNERDRW.PAS}

{START OWNERDRW.DFM}
object Form1: TForm1
  Left = 211
  Top = 155
  Width = 435
  Height = 300
  Caption = 'Form1'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  OnClose = FormClose
  OnCreate = FormCreate
  TextHeight = 16
  object ComboBox1: TComboBox
    Left = 26
    Top = 30
    Width = 165
    Height = 22
    Style = csOwnerDrawVariable
    ItemHeight = 16
    TabOrder = 0
    OnDrawItem = ComboBox1DrawItem
    OnMeasureItem = ComboBox1MeasureItem
  end
  object ListBox1: TListBox
    Left = 216
    Top = 28
    Width = 151
    Height = 167
    ItemHeight = 16
    Style = lbOwnerDrawVariable
    TabOrder = 1
    OnDrawItem = ListBox1DrawItem
    OnMeasureItem = ListBox1MeasureItem
  end
end
end
{END OWNERDRW.DFM}

```

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15727: READING CHARACTERS DIRECTLY FROM SCREEN

Technical Notes Database

TN727D.txt      READING CHARACTERS DIRECTLY FROM SCREEN

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 4.0+

Description:

Q. How can I read characters directly from the screen in Turbo Pascal?

A. The operation can be done though it is not implemented directly. A useful function outlined below can be used to read characters directly from the screen. The program using this function must use the DOS unit and the CRT unit.

```
function GetChar(x,y: integer): char;
var
  r: Registers;
  tx,ty: Integer;
begin
  tx:=wherex;
  ty:=wherey;
  r.ah:=8;
  r.bh:=0;          {Current video page; 0 is default}
  GotoXy(x,y);
  intr(16,r);
  GetChar:=r.al;
  GotoXy(tx,ty);
end;
```

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99



## Article #15731: Using OnHint Events Among Multiple Forms

Technical Information Database

TI731D.txt Using OnHint Events Among Multiple Forms

Category :VCL

Platform :All

Product :Delphi All

Description:

Using OnHint Among Multiple Forms  
-----

Delphi's Online Help and Visual Component Library Reference describe an example for processing TApplication's OnHint event. The example shows how a panel can be used to display hints associated with other components. As the example sets the Application's OnHint method in the Form's OnCreate event, a program involving more than one form will have difficulty using this technique.

Moving the assignment of OnHint from the Form's OnCreate event to its OnActivate method will allow different forms involved in the application to treat Hints in their own way.

Here is an altered form of the source code presented in the Online Help and the VCL Reference.

type

```
TForm1 = class(TForm)
  Button1: TButton;
  Panel1: TPanel;
  Edit1: TEdit;
  procedure FormActivate(Sender: TObject);
private
  { Private declarations }
public
  procedure DisplayHint(Sender: TObject);
end;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.DisplayHint(Sender: TObject);  
begin  
    Panell1.Caption := Application.Hint;  
end;
```

```
procedure TForm1.FormActivate(Sender: TObject);  
begin  
    Application.OnHint := DisplayHint;  
end;
```

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15733: WRITING TO FILE 'CON'

Technical Notes Database

TN733D.txt      WRITING TO FILE 'CON'

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

### Description:

Q. Using 4.0+, can I still write to file 'Con' without changes?

A. The 'Con' file is not longer available in 4.0+. However, you can still write to the screen with a simple Write with no file variable. The file system has been completely redesigned to allow you to write your own text file device drivers. With these, you can implement a Pascal-like text-file interface to any device, such as serial ports, windowing systems, memory, etc.

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15737: Making the Enter key work like a Tab in a TDBGrid

Technical Information Database

TI737D.txt Making the Enter key work like a Tab in a TDBGrid  
Category :VCL  
Platform :All  
Product :Delphi All

### Description:

Code to make the key act as the tab key while inside a grid.

This code also includes the processing of the key for the entire application - including fields, etc. The grid part is handled in the ELSE portion of the code. The provided code does not mimic the behavior

of the key stepping down to the next record when it reaches the last column in the grid - it moves back to the first column - .

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
{ This is the event handler for the FORM's OnKeyPress event! }
{ You should also set the Form's KeyPreview property to True }
begin
  if Key = #13 then { if it's an enter
key }
    if not (ActiveControl is TDBGrid) then begin { if not on a
TDBGrid }
      Key := #0; { eat enter key }
      Perform(WM_NEXTDLGCTL, 0, 0); { move to next
control }
    end
    else if (ActiveControl is TDBGrid) then { if it is a
TDBGrid }
      with TDBGrid(ActiveControl) do
        if selectedindex < (fieldcount -1) then { increment the
field }
          selectedindex := selectedindex +1
        else
          selectedindex := 0;
    end;
end;
```

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15739: I/O BUFFER FOR TEXT FILES

Technical Notes Database

TN739D.txt I/O BUFFER FOR TEXT FILES

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. How can I allocate my own I/O buffer for a text file?

A. You can use the procedure SetTextBuf to allocate your own text file buffer either in the data segment or on the heap.

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15743: Moving to a tab by name on a TabSet

Technical Information Database

TI743D.txt     Moving to a tab by name on a TabSet  
Category     :VCL  
Platform     :All  
Product      :Delphi All

### Description:

Moving to a page by name on a TabSet.

Place a Tabset(TabSet1) and an Edit (Edit1) on your form. Change the Tabset's Tabs Property in the String List Editor to include 4 Tabs:

```
    Hello,  
    World,  
    Of,  
    Delphi,
```

Change Edit1's onChange event to:

```
procedure TForm1.Edit1Change(Sender: TObject);  
var  
    I : Integer;  
begin  
    for I:= 0 to tabset1.tabs.count-1 do  
        if edit1.text = tabset1.tabs[I] then  
            tabset1.tabindex:=I;  
end;
```

If You type any of the Tabs names in edit1 it will focus on the appropriate tab.

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15745: FILE AND RECORD LOCKING SUPPORT

Technical Notes Database

TN745D.txt      FILE AND RECORD LOCKING SUPPORT  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

- Q. Is there any support in Turbo Pascal 4.0+ for file and record locking?
- A. There's a standard variable in the System unit called FileMode, which you can use to assign an open mode for use in all subsequent Resets. There are no record-locking routines implemented in the standard version, but they are easily implemented through MsDos calls.

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99



## Article #15749: Calling a Delphi DLL from C

Technical Information Database

TI749D.txt      Calling a Delphi DLL from C  
Category        :General Programming  
Platform        :All  
Product         :Delphi 1.0

Description:  
Calling a DLL from your C Code.

First create a simple DLL in Delphi:

```
{ Begin DLL code }  
  
Library MinMax;  
  
Function Min(X, Y: Integer): Integer; export;  
begin  
    if X < Y then Min := X else Min := Y;  
end;  
  
Function Max(X, Y: Integer): Integer; export;  
begin  
    if X > Y then Max := X else Max := Y;  
end;  
  
Exports  
    Min index 1,  
    Max index 2;  
  
begin  
end.  
  
{ End DLL code }
```

Then to Call it from your C Code:

1. In your .DEF File add:

```
IMPORTS
```

Min =MINMAX.Min

Max =MINMAX.Max

2. In your C application, you must prototype the functions as:

```
int FAR PASCAL Min(int x, y);
```

```
int FAR PASCAL Min(int x, y);
```

3. Now you can call Min or Max anywhere in your application.

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15751: FILE MODE VARIABLES

Technical Notes Database

TN751D.txt      FILE MODE VARIABLES  
Category      :Pascal  
Platform        :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. If I set the file mode variable to be a certain mode and then open two files, will the file mode change to the default on the second file open?

A. No. The file mode will stay at the set value until you change it again.

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15755: Getting a query's Memo field as a string

Technical Information Database

TI755D.txt     Getting a query's Memo field as a string  
Category     :Database Programming  
Platform      :All  
Product       :Delphi 1.0

### Description:

Returning a Memo value to an Edit field without using a memo field.

Place a query object on your form (Query1)

Place a Edit object on your form (Edit1)

Place a Button object on your form (Button1)

Double-Click on the query and add the memo field.

(Biolife.db using notes field)

Set Query1's SQL property to: Select \* from Biolife

Set Query1's Active property to: True

Add the following code to Button1's OnClick event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  bs : TBlobStream;
  p  : array [0..50] of char;
begin
  FillChar(p, SizeOf(p), #0);
  bs:= TBlobStream.Create(Query1Notes, bmRead);
  try
    bs.Read(p, 50);
  finally
    bs.Free;
  end;
  Edit1.Text:=StrPas(p);
end;
```

### Reference:



## Article #15757: CLEARING THE KEYBOARD BUFFER

Technical Notes Database

TN757D.txt      CLEARING THE KEYBOARD BUFFER

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. How do I efficiently clear the keyboard buffer?

A. Use the following code:

```
MemW[$0:$041A] := MemW[$0:$041C];
```

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15761: How to encrypt a String

Technical Information Database

TI761D.txt - How to encrypt a String

Category : Turbo Pascal  
Platform : All  
Product :

### Description:

Here is a program that demonstrates routines for encryption and decryption of strings. Note: We claim no responsibility for the security of these functions.

```
{ Begin code }
```

```
program Crypt;
```

```
uses WinCRT;
```

```
const
```

```
  C1 = 52845;
```

```
  C2 = 22719;
```

```
function Encrypt(const S: String; Key: Word): String;
```

```
var
```

```
  I: byte;
```

```
begin
```

```
  Result[0] := S[0];
```

```
  for I := 1 to Length(S) do begin
```

```
    Result[I] := char(byte(S[I]) xor (Key shr 8));
```

```
    Key := (byte(Result[I]) + Key) * C1 + C2;
```

```
  end;
```

```
end;
```

```
function Decrypt(const S: String; Key: Word): String;
```

```
var
```

```
  I: byte;
```

```
begin
```

```
  Result[0] := S[0];
```

```
for I := 1 to Length(S) do begin
    Result[I] := char(byte(S[I]) xor (Key shr 8));
    Key := (byte(S[I]) + Key) * C1 + C2;
end;
end;
```

```
var
    S: string;
begin
    Write('>');
    ReadLn(S);
    S := Encrypt(S,12345);
    WriteLn(S);
    S := Decrypt(S,12345);
    WriteLn(S);
end.
```

Reference:

3/30/99 12:28:33 PM

Last Modified: 01-SEP-99



## Article #15763: I/O REDIRECTION \$P128 \$G128

Technical Notes Database

TN763D.txt I/O REDIRECTION \$P128 \$G128

Category :Pascal

Platform :All

Product :Turbo Pascal 3.0+

Description:

Q. How do you do I/O redirection?

How can I write to standard output?

In version 3.0, I used \$P128,G128. What do I do in Version 4.0+?

A. If you want to do DOS I/O redirection when running an .EXE file generated by Turbo Pascal 4.0+, DO NOT use the Crt unit. If you do, make sure you assign a text file variable to the standard DOS output device. For example:

```
Assign(Output, ''); { assign a text file variable }
                  { to a null file name           }
Rewrite(Output);  { do a rewrite here             }
```

Any Write statement that does not specify a file variable will be redirected to the DOS standard output file. You can also use Write(Output,...).

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15767: Undo in a Memo field

Technical Information Database

TI767D.txt Undo in a Memo field  
Category :VCL  
Platform :All  
Product :Delphi All

### Description:

Doing an Undo in a Memo Field:

If you have a pop-up menu in a TMemo, and put shortcuts on it for the Cut, Copy, Paste, then you can handle those events, and call CuttoClipboard, CopytoClipboard, etc.

However, if you put an Undo option onto your pop-up menu (normally Ctrl-Z) how do you instruct the TMemo to do the Undo? If the built-in undo is sufficient, you can get it easier than a Ctrl+Z:

```
Mem1.Perform(EM_UNDO, 0, 0);
```

To check whether undo is available so as to enable/disable an undo menu item:

```
Undo1.Enabled := Mem1.Perform(EM_CANUNDO, 0, 0) <> 0;
```

### Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15769: OPENING MORE THAN 15 FILES

Technical Notes Database

TN769D.txt      OPENING MORE THAN 15 FILES

Category       :Pascal

Platform       :All

Product        :Turbo Pascal  3.0+

Description:

Q. My program needs to open more than 15 files.  What do I do?

A. Normally, DOS allows you to open up to 15 files at any one time.  But, you may use an interrupt to open more files.

  Please consult TI174, Files Open Extend.

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15773: Getting the Line number in a memo Field

Technical Information Database

TI773D.txt Getting the Line number in a memo Field

Category :VCL

Platform :All

Product :Delphi All

### Description:

How do you figure out what line number you are currently on with a TMemo control?

The trick is to use the `em_LineFromChar` message. Try this:

```
procedure TMyForm.BitBtn1Click(Sender: TObject);
var
  iLine : Integer ;
begin
  iLine := Memol.Perform(em_LineFromChar, $FFFF, 0);
  { Note: First line is zero }
  messageDlg('Line Number: ' + IntToStr(iLine), mtInformation,
            [mbOK], 0 ) ;
end;
```

### Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15775: CTRL-ALT-DEL TRAP KEYBOARD ISR

Technical Notes Database

TN775D.txt CTRL-ALT-DEL TRAP KEYBOARD ISR

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. How do you trap Ctrl-Alt-Del in an application?

A. Write an ISR for Keyboard Interrupt \$9. If ~~key was~~  
~~pressed, check bits 2(Ctrl) and 3(Alt) of Toggle Controls at~~  
~~\$0:\$0417 for a Keypressed.~~

Reference:

~~7/16/98 4:35:35 PM~~

Last Modified: 01-SEP-99

## Article #15779: Loading Bitmaps Into dBASE And Paradox BLOB Fields

Technical Information Database

TI779D.txt Loading Bitmaps Into dBASE And Paradox BLOB Fields  
Category :Database Programming  
Platform :All  
Product :Delphi 1.0

### Description:

There are a number of ways to load a bitmap image into the BLOB field of a dBASE or Paradox table. Three of the easier methods involve 1) copying the data from the Windows clipboard into a TDBImage component connected to the BLOB field, 2) using the LoadFromFile method of the TBLOBField component, and 3) using the Assign method to copy an object of type TBitmap into the Picture property of a TDBImage.

The first method, copying the bitmap from the clipboard, is probably most handy when an application needs to add bitmaps to a table when the end-user is running the application. A TDBImage component is used to act as an interface between the BLOB field in the table and the image stored in the clipboard. The PasteFromClipboard method of the TDBImage component is invoked to copy the bitmap data from the clipboard into the TDBImage. When the record is posted, the image is stored into the BLOB field in the table.

Because the Windows clipboard can contain data in formats other than just bitmap, it is advisable to check the format prior to calling the CopyFromClipboard method. To do this, a TClipboard object is created and its Has-

Format method is used to determine if the data in the clipboard is indeed of bitmap format. Note that to use a TClipboard object, the Clipbrd unit must be included in the Uses section of the unit that will be creating the object.

Here is an example showing the contents of the clipboard being copied into a TDBImage component, if the contents of the clipboard are of bitmap format:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  C: TClipboard;
begin
  C := TClipboard.Create;
  try
    if Clipboard.HasFormat(CF_BITMAP) then
      DBImage1.PasteFromClipboard
    else
      ShowMessage('Clipboard does not contain a bitmap!');
  finally
    C.Free;
  end;
end;
```

The second method of filling a BLOB field with a bitmap involves loading the bitmap directly from a file on disk into the BLOB field. This method lends itself equally well to uses at run-time for the end-user as for the developer building an application's data.

This method uses the LoadFromFile method of the TBLOBField component, the Delphi representation of a dBASE for Windows Binary field or a Paradox for Windows Graphic field, either of which may be used to store bitmap data in a table.

The LoadFromFile method of the TBLOBField component requires a single parameter: the name of the bitmap file to load, which is of type String.

The value for this parameter may come from a number of sources from the end-user manually keying in a valid file name to the program providing a string to the contents of the FileName property of the TOpenDialog component.

Here is an example showing the use of the LoadFromFile method for a TBLOBField component named Table1Bitmap (a field called Bitmap in the table associated with a TTable component named Table1):

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Table1Bitmap.LoadFromFile(
        'c:\delphi\images\splash\16color\construc.bmp');
end;
```

The third method uses the Assign method to copy the contents of an object of type TBitmap into the Picture property of a TDBImage component. An object of type TBitmap might be the Bitmap property of the Picture object property of a TImage component or it may be a stand-alone TBitmap object.

As with the method copying the data from the clipboard into a TDBImage component, the bitmap data in the TDBImage component is saved into the BLOB field in the table when the record is successfully posted.

Here is an example using the Assign method. In this case, a stand-alone TBitmap object is used. To put a bitmap image into the TBitmap, the LoadFromFile method of the TBitmap component is called.

```
procedure TForm1.Button3Click(Sender: TObject);
var
    B: TBitmap;
begin
    B := TBitmap.Create;
    try
        B.LoadFromFile('c:\delphi\images\splash\16color\athena.bmp');
        DBImage1.Picture.Assign(B);
    finally
        B.Free;
    end;
```



end;

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15781: GRAPH DETECT VIDEO CARD AUTODETECT HARDWARE

Technical Notes Database

TN781D.txt GRAPH DETECT VIDEO CARD AUTODETECT HARDWARE

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. How can I detect what kind of video card my computer has using Turbo Pascal?

A. The Graph unit of Turbo Pascal has an autodetection scheme built-in which will allow you to detect what kind of graphics hardware is available at runtime. See the following example:

```
Program AutoDetectGraphics;
Uses Graph;
var
  gd,gm: integer;

Procedure InitializeGraphics;
begin
  gd:=Detect;
  InitGraph(gd,gm, '');
  OutTextXY(20,20,'Graphics Test!');
  readln;
  CloseGraph;
end; { InitializeGraphics }

Procedure DetectGraphics(var driver, mode: integer);
begin
  DetectGraph(driver,mode);
end; { DetectGraphics }

begin { MAIN }
  DetectGraphics(gd,gm);
  writeln('Detected: ');
  writeln('  GraphDriver = ',gd);
  writeln('  GraphMode    = ',gm);
  readln;
  InitializeGraphics;
```

end. { End of Program }

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15785: Printing the contents of a TMemo or TListbox

Technical Information Database

TI785D.txt     Printing the contents of a TMemo or TListbox

Category     :VCL

Platform     :All

Product      :Delphi   All

Description:

Q: How can I print all of the lines within a TMemo or TListbox component?

A: The function below accepts a TStrings object as a parameter and prints

out each string to the default printer. Because it uses a TStrings, the function will work with any type of component that contains a TStrings-type property, such as a TDBMemo or TOutline.

{ Begin code listing }

```
uses Printers;
```

```
procedure PrintStrings(Strings: TStrings);
```

```
var
```

```
  Prn: TextFile;
```

```
  i: word;
```

```
begin
```

```
  AssignPrn(Prn);
```

```
  try
```

```
    Rewrite(Prn);
```

```
    try
```

```
      for i := 0 to Strings.Count - 1 do
```

```
        writeln(Prn, Strings.Strings[i]);
```

```
    finally
```

```
      CloseFile(Prn);
```

```
    end;
```

```
  except
```

```
    on EInOutError do
```

```
      MessageDlg('Error Printing text.', mtError, [mbOk], 0);
```

```
  end;
```

```
end;
```

```
{ End code listing }
```

To print out the contents of a TMemo or TListbox, use the following code:

```
PrintStrings(Memo1.Lines);
```

or

```
PrintStrings(Listbox1.Items);
```

Reference:

7/16/98 4:33:53 PM

Last Modified: 01-SEP-99

## Article #15787: REVERSE VIDEO WITH GRAPH UNIT

Technical Notes Database

TN787D.txt REVERSE VIDEO WITH GRAPH UNIT

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. How do you reverse the background color with the foreground color with the Graph unit routines?

A. To create reverse video in graphics mode:

1. Set the background color: SetBkColor(6). The background color is now brown.
2. Set the palette so that the background and foreground colors won't be the same: SetPalette(6,0).
3. Set the foreground color: SetColor(6).

To restore, either:

1. Use GetPalette(P) before creating reverse video and restore with SetPalette(P), or
2. a) Set the background color: SetBkColor(0). The background color is back to black.  
b) Return the palette to its original state: SetPalette(6,20).

Note that these procedures work only in graphics mode.

Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15791: Extracting A Bitmap From A BLOB Field

Technical Information Database

TI791D.txt     Extracting A Bitmap From A BLOB Field

Category     :Database Programming

Platform     :All

Product     :Delphi All

### Description:

Extracting a bitmap from a dBASE or Paradox blob field -- without first saving the bitmap out to a file -- is a simple process of using the Assign method to store the contents of the BLOB field to an object of type TBitmap. A stand-alone TBitmap object or the Bitmap property of the Picture object property of a TImage component are examples of compatible destinations for this operation.

Here is an example demonstrating using the Assign method to copy a bitmap from a BLOB field into a TImage component.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Image1.Picture.Bitmap.Assign(Table1Bitmap);
end;
```

In this example, the TBLOBField object Table1Bitmap is a BLOB field in a dBASE table. This TBLOBField object was created using the Fields Editor.

If the Fields Editor is not used to create TFields for the fields in the table, the fields must be referenced using either the FieldByName method or the Fields property, both part of the TTable and TQuery components. In cases where one of those means is used to reference the BLOB field in a table, the field reference must be type-cast as a TBLOBField object

prior  
to using the Assign method. For example:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Image1.Picture.Bitmap.Assign(TBLOBField(Table1.Fields[1]));  
end;
```

A bitmap stored in a BLOB field may also be copied directly to a stand-alone TBitmap object. Here is an example showing the creation of a TBitmap object and storing into it a bitmap from a BLOB field.

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    B: TBitmap;  
begin  
    B := TBitmap.Create;  
    try  
        B.Assign(Table1.Bitmap);  
        Image1.Picture.Bitmap.Assign(B);  
    finally  
        B.Free;  
    end;  
end;
```

Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99



## Article #15793: INVERSE VIDEO TEXT

Technical Notes Database

TN793D.txt      INVERSE VIDEO TEXT  
Category        :Pascal  
Platform        :All  
Product         :Turbo Pascal 3.0+

### Description:

Q. How do I get inverse video in text mode using Turbo Pascal 3.0 and 4.0+?

A. You can create inverse video by reversing text and background colors before displaying text. For example:

```
Program InverseIt;  
{ Uses Crt; => Add if using TP 4.0+ }  
begin  
  TextBackground(White);  
  TextColor(Black);  
  writeln('This is inverse video!');  
end.
```

### Reference:

7/16/98 4:35:35 PM

Last Modified: 01-SEP-99

## Article #15797: Bitmaps And InterBase BLOB Fields

Technical Information Database

TI797D.txt      Bitmaps And InterBase BLOB Fields

Category      :Database Programming

Platform      :All

Product       :Delphi    All

### Description:

dBASE and Paradox tables provide a BLOB field to store binary data that,

when the stored data is of bitmap-format, work as-is with the TDBImage component to display images. In Database Desktop, these field types are

listed as Binary and Graphic (for dBASE and Paradox tables, respectively).

However, the process of storing bitmap images in InterBase BLOB fields and using the stored data with TDBImage components is not as straightforward.

InterBase tables do not have just one type of BLOB field. There are three

variants, or sub-types: type 0, type 1, and user-defined sub-types.

Types

0 and 1 are pre-defined types. Type 0 BLOB fields (the default type) are

for storing general binary data. Type 1 BLOB fields are for storing text

BLOB data. Either the pre-defined type 0 or a user-defined BLOB sub-type

will allow the automated retrieval of bitmap data from the BLOB field that

is to be displayed in a TDBImage component. Type 0 BLOB fields may be used

for storing bitmap-format data or raw binary data. Here is an example of

manually extracting bitmap data stored in a type 0 BLOB field (Table1-BLOBField) and displaying the data in a (non-data-aware) TImage component:

```
procedure TForm1.ExtractBtnClick(Sender: TObject);
```

```
begin
  Image1.Picture.Bitmap.Assign(Table1BLOBField);
end;
```

This manual method may be used or, more commonly, a data-aware control would be used so that the display of a given record's bitmap (in a BLOB field) will be automatically displayed. The TDBImage serves this purpose, and by setting the DataSource property to the TDataSource component associated with the underlying table and setting the DataField to the BLOB field containing the bitmap, the TDBImage component will display the image stored in the BLOB field and automatically load each record's BLOB field contents as the record pointer is changed.

The Database Desktop utility will allow the creation only of type 0 binary BLOB fields, no provision was made for user-defined BLOB field sub-types.

If it is desired that a user-defined BLOB sub-type be used to store the bitmap data, it would need to be created with an SQL statement. Typically this would be through the WISQL utility, but an appropriate SQL statement in a TQuery would suffice. Here is an SQL statement that creates a table with a user-defined BLOB field sub-type:

```
CREATE TABLE WITHBMP
(
  FILENAME CHAR(12),
  BITMAP    BLOB SUB_TYPE -1
)
```

Once a table with a compatible BLOB field is created, storing bitmap data to the BLOB field and displaying the bitmap images in a TDBImage component uses the same methods as would be used with dBASE or Paradox tables. There are a number of ways to load a bitmap image into a BLOB field.

Three of the easier methods involve 1) copying the data from the Windows clipboard into a TDBImage component connected to the BLOB field, 2) using the LoadFromFile method of the TBLOBField component, and 3) using the Assign method to copy an object of type TBitmap into the Picture property of a TDBImage.

The first method, copying the bitmap from the clipboard, is probably most handy when an application needs to add bitmaps to a table when the end-user is running the application. A TDBImage component is used to act as an interface between the BLOB field in the table and the image stored in the clipboard. The PasteFromClipboard method of the TDBImage component is invoked to copy the bitmap data from the clipboard into the TDBImage. When the record is posted, the image is stored into the BLOB field in the table.

Because the Windows clipboard can contain data in formats other than just bitmap, it is advisable to check the format prior to calling the CopyFromClipboard method. To do this, a TClipboard object is created and its HasFormat method is used to determine if the data in the clipboard is indeed of bitmap format. Note that to use a TClipboard object, the Clipbrd unit must be included in the Uses section of the unit that will be creating the object.

Here is an example showing the contents of the clipboard being copied into a TDBImage component, if the contents of the clipboard are of bitmap format:

```
procedure TForm1.Button1Click(Sender: TObject);  
var
```

```

    C: TClipboard;
begin
    C := TClipboard.Create;
    try
        if Clipboard.HasFormat(CF_BITMAP) then
            DBImage1.PasteFromClipboard
        else
            ShowMessage('Clipboard does not contain a bitmap!');
    finally
        C.Free;
    end;
end;

```

The second method of filling a BLOB field with a bitmap involves loading the bitmap directly from a file on disk into the BLOB field. This method lends itself equally well to uses at run-time for the end-user as for the developer building an application's data. This method uses the LoadFromFile method of the TBLOBField component, the Delphi representation of an InterBase BLOB field.

The LoadFromFile method of the TBLOBField component requires a single parameter: the name of the bitmap file to load, which is of type String.

The value for this parameter may come from a number of sources from the end-user manually keying in a valid file name to the program providing a string to the contents of the FileName property of the TOpenDialog component.

Here is an example showing the use of the LoadFromFile method for a TBLOBField component named Table1Bitmap (a field called Bitmap in the table associated with a TTable component named Table1):

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    Table1Bitmap.LoadFromFile(
        'c:\delphi\images\splash\16color\construc.bmp');
end;

```

The third method uses the Assign method to copy the contents of an object of type TBitmap into the Picture property of a TDBImage component. An object of type TBitmap might be the Bitmap property of the Picture object property of a TImage component or it may be a stand-alone TBitmap object. As with the method copying the data from the clipboard into a TDBImage component, the bitmap data in the TDBImage component is saved into the BLOB field in the table when the record is successfully posted.

Here is an example using the Assign method. In this case, a stand-alone TBitmap object is used as the source of the bitmap data. To put a bitmap image into the TBitmap, the LoadFromFile method of the TBitmap component is called.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  B: TBitmap;
begin
  B := TBitmap.Create;
  try
    B.LoadFromFile('c:\delphi\images\splash\16color\athena.bmp');
    DBImage1.Picture.Assign(B);
  finally
    B.Free;
  end;
end;
```

Going the opposite direction -- extracting a bitmap from an InterBase BLOB field (without first saving the bitmap out to a file) is a simple process of using the Assign method of the TBLOBField object to store the contents of the BLOB field to an object of type TBitmap. A stand-alone TBitmap object or the Bitmap property of the Picture object property of a TImage component are examples of compatible destinations for this operation.

Here is an example demonstrating using the Assign method to copy a

bitmap  
from a BLOB field into a TImage component (Table1Bitmap is the  
TBLOBfield  
for the BLOB field in the table).

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Image1.Picture.Bitmap.Assign(Table1Bitmap);  
end;
```

In this example, the TBLOBField object Table1Bitmap is a BLOB field  
in an  
InterBase table. This TBLOBField object was created using the Fields  
Editor. If the Fields Editor is not used to create TFields for the  
fields  
in the table, the fields must be referenced using either the  
FieldByName  
method or the Fields property, both part of the TTable and TQuery  
comp-  
onents. In cases where one of those means is used to reference the  
BLOB  
field in a table, the field reference must be type-cast as a  
TBLOBField  
object prior to using the Assign method. For example:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Image1.Picture.Bitmap.Assign(TBLOBField(Table1.Fields[1]));  
end;
```

A bitmap stored in a BLOB field may also be copied directly to a  
stand-  
alone TBitmap object. Here is an example showing the creation of a  
TBitmap object and storing into it a bitmap from a BLOB field.

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    B: TBitmap;  
begin  
    B := TBitmap.Create;  
    try  
        B.Assign(Table1Bitmap);  
        Image1.Picture.Bitmap.Assign(B);  
    finally  
        B.Free;
```

end;  
en

Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99



## Article #15799: HIDING THE CURSOR DOS INTERRUPTS

Technical Notes Database

TN799D.txt      HIDING THE CURSOR DOS INTERRUPTS

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 3.0+

Description:

Q. How can I hide the cursor using 3.0 or 4.0+?

A. Uses DOS Interrupt \$10 to hide the text cursor. For example:

```
Procedure SetCursor(On: boolean);
var
  Reg: Registers;      { type Registers is defined within }
                       { DOS unit of TP 4.0+ }
begin
  Reg.AX:= $0100;
  Reg.BX:= 0;
  if On then
    Reg.CX:= $0B0C      { $0607 if on color monitor }
  else
    Reg.CX:= $2020;
  Intr($10,Reg);
end; { SetCursor }
```

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15803: Control Font Styles

Technical Information Database

TI803D.txt Control Font Styles

Category :VCL

Platform :All

Product :Delphi All

Description:

Control Font Styles:

This code will change the font style of a Edit when selected. This code could be implemented to control font style on other objects.

With a Edit(Edit1) and a ListBox(ListBox1) on a form Add the following Items to the ListBox:

```
fsBold
fsItalic
fsUnderLine
fsStrikeOut
```

```
procedure TForm1.ListBox1Click(Sender: TObject);
var
  X : Integer;
type
  TLookUpRec = record
    Name: String;
    Data: TFontStyle;
  end;
const
  LookUpTable: array[1..4] of TLookUpRec =
    ((Name: 'fsBold'; Data: fsBold),
     (Name: 'fsItalic'; Data: fsItalic),
     (Name: 'fsUnderline'; Data: fsUnderline),
     (Name: 'fsStrikeOut'; Data: fsStrikeOut));
begin
  X := ListBox1.ItemIndex;
  Edit1.Text := ListBox1.Items[X];
  Edit1.Font.Style := [LookUpTable[ListBox1.ItemIndex+1].Data];
end;
```

Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99

## Article #15805: UNIT FILE FORMAT ERROR

Technical Notes Database

TN805D.txt      UNIT FILE FORMAT ERROR

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

### Description:

Q. What does the error, unit file format error, mean and how do I resolve it?

A. This error means that you are trying to link a unit that was compiled with a different version of the compiler than the one that you are currently using. If you have the source code for the unit, recompile source code. If you do not have the source code, then you must get the source code or use the appropriate version of the compiler. You cannot use the unit otherwise.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15809: Handling EDBEngineError Exceptions

Technical Information Database

TI809D.txt Handling EDBEngineError Exceptions

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

Information that describes the conditions of a database engine error can

be obtained for use by an application through the use of an EDBEngineError

exception. EDBEngineError exceptions are handled in an application through

the use of a try..except construct. When an EDBEngineError exception occurs, a EDBEngineError object would be created and various fields in that

EDBEngineError object would be used to programmatically determine what went wrong and thus what needs to be done to correct the situation.

Also,

more than one error message may be generated for a given exception.

This

requires iterating through the multiple error messages to get needed information.

The fields that are most pertinent to this context are:

**ErrorCount:** type Integer; indicates the number of errors that are in

the Errors property; counting begins at zero.

**Errors:** type TDBError; a set of record-like structures that contain information about each specific error generated; each record is accessed via an index number of type Integer.

**Errors.ErrorCode:** type DBIResult; indicating the BDE error code for the

error in the current Errors record.

Errors.Category: type Byte; category of the error referenced by the ErrorCode field.

Errors.SubCode: type Byte; subcode for the value of ErrorCode.

Errors.NativeError: type LongInt; remote error code returned from the server; if zero, the error is not a server error; SQL statement return codes appear in this field.

Errors.Message: type TMessageStr; if the error is a server error, the server message for the error in the current Errors record; if not a server error, a BDE error message.

In a try..except construct, the EDBEngineError object is created directly in the except section of the construct. Once created, fields may be accessed normally, or the object may be passed to another procedure for inspection of the errors. Passing the EDBEngineError object to a specialized procedure is preferred for an application to make the process more modular, reducing the amount of repeated code for parsing the object for error information. Alternately, a custom component could be created to serve this purpose, providing a set of functionality that is easily transported across applications. The example below only demonstrates creating the DBEngineError object, passing it to a procedure, and parsing the object to extract error information.

In a try..except construct, the DBEngineError can be created with syntax such as that below:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  if Edit1.Text > ' ' then begin
    Table1.FieldName('Number').AsInteger := StrToInt(Edit1.Text);
```

```

    try
        Table1.Post;
    except on E: EDBEngineError do
        ShowError(E);
    end;
end;
end;

```

In this procedure, an attempt is made to change the value of a field in a table and then call the Post method of the corresponding TTable component. Only the attempt to post the change is being trapped in the try..except construct. If an EDBEngineError occurs, the except section of the construct is executed, which creates the EDBEngineError object (E) and then passes it to the procedure ShowError. Note that only an EDBEngineError exception is being accounted for in this construct. In a real-world situation, this would likely be accompanied by checking for other types of exceptions.

The procedure ShowError takes the EDBEngineError, passed as a parameter, and queries the object for contained errors. In this example, information about the errors are displayed in a TMemo component. Alternately, the extracted values may never be displayed, but instead used as the basis for logic branching so the application can react to the errors. The first step in doing this is to establish the number of errors that actually occurred. This is the purpose of the ErrorCount property. This property supplies a value of type Integer that may be used to build a for loop to iterate through the errors contained in the object. Once the number of errors actually contained in the object is known, a loop can be used to visit each existing error (each represented by an Errors property record) and extract information about each error to be inserted into the TMemo component.

```
procedure TForm1.ShowError(AExc: EDBEngineError);
var
  i: Integer;
begin
  Mem1.Lines.Clear;
  Mem1.Lines.Add('Number of errors: ' + IntToStr(AExc.ErrorCount));
  Mem1.Lines.Add('');
  {Iterate through the Errors records}
  for i := 0 to AExc.ErrorCount - 1 do begin
    Mem1.Lines.Add('Message: ' + AExc.Errors[i].Message);
    Mem1.Lines.Add('  Category: ' +
      IntToStr(AExc.Errors[i].Category));
    Mem1.Lines.Add('  Error Code: ' +
      IntToStr(AExc.Errors[i].ErrorCode));
    Mem1.Lines.Add('  SubCode: ' +
      IntToStr(AExc.Errors[i].SubCode));
    Mem1.Lines.Add('  Native Error: ' +
      IntToStr(AExc.Errors[i].NativeError));
    Mem1.Lines.Add('');
  end;
end;
```

Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99



## Article #15814: How to handle text drawing in a TDBGrid

Technical Information Database

TI814D.txt     How to handle text drawing in a TDBGrid

Category     :Database Programming

Platform     :All

Product      :Delphi 1.0

### Description:

The following method can be used as the event handler for a TDBGrid.OnDrawDataCell event. This method demonstrates how to paint the text in one column a different color than the rest of the text in the grid.

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect:
TRect;
  Field: TField; State: TGridDrawState);
{ NOTE: DefaultDrawing property of Grid(s) must be set to False }
begin
  { if field name is "NAME" }
  if Field.FieldName = 'NAME' then
    { change font color to red }
    (Sender as TDBGrid).Canvas.Font.Color := clRed;
  { draw text in the grid }
  (Sender as TDBGrid).Canvas.TextRect(Rect, Rect.Left + 2, Rect.Top +
2,
                                     Field.AsString);
end;
```

### Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99

## Article #15816: "INTERNAL ERROR REPORT TO BORLAND"

Technical Notes Database

TN816D.txt "INTERNAL ERROR REPORT TO BORLAND"

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

Q. Why I am seeing "Internal Error Report to Borland"? What should I do?

A. You may receive this message under any of the following circumstances:

1. An incompatible display adapter is being used.
2. The copy of Turbo Pascal in memory has been overwritten by the program begin developed. (See Memory Corruption)
3. During a DOS shell from Turbo Pascal a Terminate and Stay Resident program was loaded into memory.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15820: Trouble running Delphi programs from Delphi

Technical Information Database

TI820D.txt - Trouble running Delphi programs from Delphi

Category :Install Issues

Platform :All Windows

Product :All32Bit,

Description:

### Delphi Integrated Debugger

-----

Delphi's integrated debugger is a feature of the IDE that allows you to execute, step, trace, and otherwise debug your Delphi applications. By default, Delphi's integrated debugger is enabled (via Options|Project|Integrated Debugging) on startup of Delphi.

The debugger handles certain interrupts and operating system debugger hooks in an effort to trap General Protection Faults and other types of exceptions. Other programs that attempt to handle exceptions in the same way will quite possibly conflict with the integrated debugger. DataSafe is one example of a program that conflicts with the integrated debugger. Classic symptoms include, Delphi not recognizing that you've stopped your program and getting kicked into DOS without warning at various stages.

To test for a debugger conflict, compiling the program from Delphi, and run it from File Manager. If it runs, then disable the integrated debugger, and run your application. If it still runs, then the problem is most likely a conflict between the exception-handling software and the integrated debugger.

There are two ways to fix this. The first is to remove the conflicting software. The other is to disable the integrated

debugger by un-selecting Options|Enviroment|Integrated Debugging.

Reference:

4/1/99 11:45:06 AM

Last Modified: 01-SEP-99

## Article #15822: MOVING FILES ACROSS DIRECTORIES

Technical Notes Database

TN822D.txt      MOVING FILES ACROSS DIRECTORIES

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 4.0+

Description:

Q. How can I move a file from one directory to another from a Turbo Pascal program?

A. If you are on a system equipped with a hard drive, you can simply use the

RENAME procedure. Assign a file variable to the file name as it currently

appears in the directory. Use the same file name when RENAME is called but

replace the directory path specified with that of directory where the file is to be placed.

This method will not work across drives. See the Copyfile example from the

description of BLOCKREAD in the Turbo Pascal Reference Lookup Chapter in the Reference Manual.

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15826: Searching your application's help file

Technical Information Database

TI826D.txt     Searching your application's help file

Category     :VCL

Platform     :All

Product      :Delphi   All

### Description:

The following code demonstrates how to bring up the WinHelp "Search" dialog for your application's help file. You can use TApplication's HelpCommand method to send the Help\_PartialKey command to the WinHelp system. The parameter for this command should be a PChar (cast to a longint to circumvent typechecking) that contains the string on which you'd like to search. The example below uses an empty string, which invokes "Search" dialog and leaves the edit control in the dialog empty.

```
procedure TForm1.SearchHelp;
var
  P: PChar;
begin
  Application.HelpFile := 'c:\delphi\bin\delphi.hlp';
  P := StrNew('');
  Application.HelpCommand(Help_PartialKey, longint(P));
  StrDispose(P);
end;
```

### Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99

## Article #15828: DOS - PARAMSTR AND PARAMCOUNT

Technical Notes Database

TN828D.txt      DOS - PARAMSTR AND PARAMCOUNT

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 4.0+

Description:

Q. How can I grab DOS command line parameters from within my program?

A. Use the ParamStr and ParamCount functions.

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15832: Searching Through Query Result Sets

Technical Information Database

TI832D.txt Searching Through Query Result Sets

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

The TQuery component does not offer the index-based search capabilities of the TTable component (FindKey, GotoKey, and GotoNearest). So how do you search within the result data set from a TQuery to find a row with a specific field value?

One way to search a query result set is a sequential search. This type of search starts at the first row in the data set and, in a While loop, sequentially compares the value of a field in the row with a search value.

One of two results are possible: a value will be found (success) or the end of the data set will be reached (failure). The problem with this way of searching the data set is that the further into the data set a row with a matching value is, the longer it takes to arrive at that row. And, a failed search takes longest of all because it must go all the way to the last row in the data set. If the data set being searched is a large one, this process may take a considerable amount of time.

Here is a function that will perform a sequential search of the result set from a TQuery:

```
function SeqSearch(AQuery: TQuery; AField, AValue: String): Boolean;  
begin
```



```

with AQuery do begin
    First;
    while (not Eof) and (not (FieldByName(AField).AsString =
AValue)) do
        Next;
    SeqSearch := not Eof;
end;
end;

```

This function takes three parameters:

1. AQuery: type TQuery; the TQuery component in which the search is to be executed.
2. AField: type String; the name of the field against which the search value will be compared.
3. AValue: type String; the value being searched for. If the field is of a data type other than String, this search value should be changed to the same data type.

The Boolean return value of this function indicates the success (True) or failure (False) of the search.

An alternative is using a bracketing approach. On a conceptual level, this method acts somewhat like a b-tree index. It is based on the given that for a row at a given point in the data set, the value of the field being searched compared to the search value will produce one of three possible conditions:

1. The field value will be greater than the search value, or...
2. The field value will be less than the search value, or...
3. The field value will be equal to the search value.

A bracketing search process uses this means of looking at the current row in respect to the search value and uses it to successively reduce the rows

to be search by half, until only one row remains. This search field value for this sole remaining row will either be a match to the search value (success) or it will not (failure, and no match exists in the data set).

Functionally, this process lumps the condition of the search field being less than or equal to the search value into a single condition. This leaves only two possible results for the comparison of the current search field value with the search value: less than/equal to or greater than. Initially, a range of numbers is established. The low end of the range is represented by an Integer, at the start of the search process set to 0 or one less than the first row in the data set. The far end of the range is also an Integer, with the value of the RecordCount property of the TQuery. The current row pointer is then moved to a point half way between the low and high ends of the range. The search field value at that row is then compared to the search value. If the field value is less than or equal to the search value, the row being sought must be in the lower half of the range of rows so the high end of the range is reduced to the current row position. If the field value is greater than the search value, the sought value must be in the higher half of the range and so the low end is raised to the current point. By repeating this process, the number of rows that are encompassed in the range are successively reduced by half. Eventually, only one row will remain.

Putting this into a modular, transportable function, the code would look like that below:

```
function Locate(AQuery: TQuery; AField, AValue: String): Boolean;  
var  
    Hi, Lo: Integer;  
begin
```

```

with AQuery do begin
  First;
  {Set high end of range of rows}
  Hi := RecordCount;
  {Set low end of range of rows}
  Lo := 0;
  {Move to point half way between high and low ends of range}
  MoveBy(RecordCount div 2);
  while (Hi - Lo) > 1 do begin
    {Search field greater than search value, value in first half}
    if (FieldByName(AField).AsString > AValue) then begin
      {Lower high end of range by half of total range}
      Hi := Hi - ((Hi - Lo) div 2);
      MoveBy(((Hi - Lo) div 2) * -1);
    end
    {Search field less than search value, value in far half}
    else begin
      {Raise low end of range by half of total range}
      Lo := Lo + ((Hi - Lo) div 2);
      MoveBy((Hi - Lo) div 2);
    end;
  end;
  {Fudge for odd numbered rows}
  if (FieldByName(AField).AsString > AValue) then Prior;
  Locate := (FieldByName(AField).AsString = AValue)
end;
end;

```

Because there will never be a difference of less than one between the low and high ends of the range of rows, a final fudge was added to allow the search to find the search value in odd numbered rows.

This function takes the same three three parameters as the SeqSearch function described earlier.

The return value of this function is of type Boolean, and reflects the success or failure of the search. As the search does move the row pointer, the effects of this movement on the implicit posting of changed data and on where the desired position of the row pointer should be after a failed search should be taken into account in the calling application. For

instance, a TBookmark pointer might be used to return the row pointer to where it was prior to a search if that search fails.

How is this process better than a sequential search? First, in bracketing the search value, only a fraction of the number of rows will be visited as would be the case in a sequential search. Unless the row with the value being sought is in the first 1,000 rows, this search method will be faster than a sequential search. Because this process always uses the same number of records, the search time will be consistent whether searching for the value in row 1,000 or row 90,000. This is in contrast with the sequential search that takes longer the farther into the data set the desired row is.

Can this method be used with any TQuery result set? No. Because of the way this method works in basing the direction of the search as either high or low, it depends on the row being ordered in a descending manner based on the field in which the search will be conducted. This means that it can only be used if the data set is naturally in a sequential order or an ORDER BY clause is used in the SQL statement for the TQuery. The size of the result set will also be a factor when deciding whether to perform a sequential or bracketing search. This process is most advantageous for speed when used with larger result sets. With smaller sets (1,000 or less rows), though, a sequential search will often be as fast or faster.

Reference:

7/16/98 4:33:54 PM

Last Modified: 01-SEP-99

## Article #15834: RETURNING ERROR LEVELS TO BATCH FILES

Technical Notes Database

TN834D.txt      RETURNING ERROR LEVELS TO BATCH FILES  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

- Q. How do I set the error level within a Turbo Pascal program and test for the error level inside a batch (.BAT) file?
- A. The DOS error level can only be checked in a .BAT file and only a line immediately following the application that is returning the DOS error level. For example, suppose in the program TEST we set the error level to 5.

```
Begin
  HALT(5);
End.
```

Then in a .BAT file we could have

```
TEST
IF ERROR LEVEL
  go do something ...
```

The IF ERROR LEVEL 4 will evaluate to true because 5 is greater than 4 and the "go do something..." part will be executed. Note that you must always check the highest possible error level first. The expression IF ERROR LEVEL 0 will ALWAYS evaluate to true.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15838: dBASE .DBF File Structure

Technical Information Database

TI838D.txt dBASE .DBF File Structure

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

Sometimes it is necessary to delve into a dBASE table outside the control of the Borland Database Engine (BDE). For instance, if the .DBT file (that contains memo data) for a given table is irretrievably lost, the file will not be usable because the byte in the file header indicates that there should be a corresponding memo file. This necessitates toggling this byte to indicate no such accompanying memo file. Or, you may just want to write your own data access routine.

Below are the file structures for dBASE table files. Represented are the file structures as used for various versions of dBASE: dBASE III PLUS 1.1, dBASE IV 2.0, dBASE 5.0 for DOS, and dBASE 5.0 for Windows.

\*\*\*\*\*  
The data file header structure for dBASE III PLUS table file.  
\*\*\*\*\*

The table file header:

=====

Byte	Contents	Description
0	1 byte	Valid dBASE III PLUS table file (03h without a memo (.DBT file; 83h with a memo).
1-3	3 bytes	Date of last update; in YYMMDD format.
4-7	32-bit number	Number of records in the table.
8-9	16-bit number	Number of bytes in the header.
10-11	16-bit number	Number of bytes in the record.
12-14	3 bytes	Reserved bytes.
15-27	13 bytes	Reserved for dBASE III PLUS on a LAN.
28-31	4 bytes	Reserved bytes.
32-n	32 bytes	Field descriptor array (the structure of this array is

each shown below)  
n+1 1 byte 0Dh stored as the field terminator.

n above is the last byte in the field descriptor array. The size of the array depends on the number of fields in the table file.

#### Table Field Descriptor Bytes

=====

Byte	Contents	Description
0-10	11 bytes	Field name in ASCII (zero-filled).
11	1 byte	Field type in ASCII (C, D, L, M, or N).
12-15	4 bytes	Field data address (address is set in memory; not useful on disk).
16	1 byte	Field length in binary.
17	1 byte	Field decimal count in binary.
18-19	2 bytes	Reserved for dBASE III PLUS on a LAN.
20	1 byte	Work area ID.
21-22	2 bytes	Reserved for dBASE III PLUS on a LAN.
23	1 byte	SET FIELDS flag.
24-31	1 byte	Reserved bytes.

#### Table Records

=====

The records follow the header in the table file. Data records are preceded by one byte, that is, a space (20h) if the record is not deleted, an asterisk (2Ah) if the record is deleted. Fields are packed into records without field separators or record terminators. The end of the file is marked by a single byte, with the end-of-file marker, an OEM code page character value of 26 (1Ah). You can input OEM code page data as indicated below.

#### Allowable Input for dBASE Data Types

=====

Data Type	Data Input
C (Character)	All OEM code page characters.
D (Date)	Numbers and a character to separate month, day, and year (stored internally as 8 digits in YYYYMMDD format).
N (Numeric)	- . 0 1 2 3 4 5 6 7 8 9
L (Logical)	? Y y N n T t F f (? when not initialized).
M (Memo)	All OEM code page characters (stored internally as 10 digits representing a .DBT block number).



## Binary, Memo, and OLE Fields And .DBT Files

=====

Memo fields store data in .DBT files consisting of blocks numbered sequentially (0, 1, 2, and so on). The size of these blocks are internally set to 512 bytes. The first block in the .DBT file, block 0, is the .DBT file header.

Memo field of each record in the .DBF file contains the number of the block (in OEM code page values) where the field's data actually begins. If a field contains no data, the .DBF file contains blanks (20h) rather than a number.

When data is changed in a field, the block numbers may also change and the number in the .DBF may be changed to reflect the new location.

This information is from the Using dBASE III PLUS manual, Appendix C.

\*\*\*\*\*  
The data file header structure for dBASE IV 2.0 table file.  
\*\*\*\*\*

### File Structure:

=====

Byte	Contents	Meaning
-----	-----	-----
0	1byte	Valid dBASE IV file; bits 0-2 indicate version number, bit 3 the presence of a dBASE IV memo file, bits 4-6 the presence of an SQL table, bit 7 the presence of any memo file (either dBASE III PLUS or dBASE IV).
1-3	3 bytes	Date of last update; formatted as YYMMDD.
4-7	32-bit number	Number of records in the file.
8-9	16-bit number	Number of bytes in the header.
10-11	16-bit number	Number of bytes in the record.
12-13	2 bytes	Reserved; fill with 0.
14	1 byte	Flag indicating incomplete transaction.
15	1 byte	Encryption flag.
16-27	12 bytes	Reserved for dBASE IV in a multi-user environment.
28	1 bytes	Production MDX file flag; 01H if there is an MDX, 00H if not.
29	1 byte	Language driver ID.
30-31	2 bytes	Reserved; fill with 0.
32-n*	32 bytes each	Field descriptor array (see below).
n + 1	1 byte	0DH as the field terminator.

\* n is the last byte in the field descriptor array. The size of the array depends on the number of fields in the database file.

The field descriptor array:

=====

Byte	Contents	Meaning
0-10	11 bytes	Field name in ASCII (zero-filled).
11	1 byte	Field type in ASCII (C, D, F, L, M, or N).
12-15	4 bytes	Reserved.
16	1 byte	Field length in binary.
17	1 byte	Field decimal count in binary.
18-19	2 bytes	Reserved.
20	1 byte	Work area ID.
21-30	10 bytes	Reserved.
31	1 byte	Production MDX field flag; 01H if field has an index tag in the production MDX file, 00H if not.

Database records:

=====

The records follow the header in the database file. Data records are preceded by one byte; that is, a space (20H) if the record is not deleted, an asterisk (2AH) if the record is deleted. Fields are packed into records without field separators or record terminators. The end of the file is marked by a single byte, with the end-of-file marker an ASCII 26 (1AH) character.

Allowable Input for dBASE Data Types:

=====

Data	Type	Data Input
C	(Character)	All OEM code page characters.
D	(Date)	Numbers and a character to separate month, day, and year (stored internally as 8 digits in YYYYMMDD format).
F	(Floating point binary numeric)	- . 0 1 2 3 4 5 6 7 8 9
N	(Binary coded decimal numeric)	- . 0 1 2 3 4 5 6 7 8 9
L	(Logical)	? Y y N n T t F f (? when not initialized).
M	(Memo)	All OEM code page characters (stored internally as 10 digits representing a .DBT block number).

## Memo Fields And .DBT Files

=====

Memo fields store data in .DBT files consisting of blocks numbered sequentially (0, 1, 2, and so on). SET BLOCKSIZE determines the size of each block. The first block in the .DBT file, block 0, is the .DBT file header.

Each memo field of each record in the .DBF file contains the number of the block (in OEM code page values) where the field's data actually begins. If a field contains no data, the .DBF file contains blanks (20h) rather than a number.

When data is changed in a field, the block numbers may also change and the number in the .DBF may be changed to reflect the new location.

This information is from the dBASE IV Language Reference manual, Appendix D.

\*\*\*\*\*

The data file header structure for dBASE 5.0 for DOS table file.

\*\*\*\*\*

The table file header:

=====

Byte	Contents	Description
-----	-----	-----
0	1 byte	Valid dBASE for Windows table file; bits 0-2 indicate version number; bit 3 indicates presence of a dBASE IV or dBASE for Windows memo file; bits 4-6 indicate the presence of a dBASE IV SQL table; bit 7 indicates the presence of any .DBT memo file (either a dBASE III PLUS type or a dBASE IV or dBASE for Windows memo file).
1-3	3 bytes	Date of last update; in YYMMDD format.
4-7	32-bit number	Number of records in the table.
8-9	16-bit number	Number of bytes in the header.
10-11	16-bit number	Number of bytes in the record.
12-13	2 bytes	Reserved; filled with zeros.
14	1 byte	Flag indicating incomplete dBASE transaction.
15	1 byte	Encryption flag.
16-27	12 bytes	Reserved for multi-user processing.
28	1 byte	Production MDX flag; 01h stored in this byte if a prod-

uction .MDX file exists for this table; 00h if no .MDX file exists.

29	1 byte	Language driver ID.
30-31	2 bytes	Reserved; filled with zeros.
32-n	32 bytes	Field descriptor array (the structure of this array is shown below)
	each	
n+1	1 byte	0Dh stored as the field terminator.

n above is the last byte in the field descriptor array. The size of the array depends on the number of fields in the table file.

#### Table Field Descriptor Bytes

=====

Byte	Contents	Description
0-10	11 bytes	Field name in ASCII (zero-filled).
11	1 byte	Field type in ASCII (B, C, D, F, G, L, M, or N).
12-15	4 bytes	Reserved.
16	1 byte	Field length in binary.
17	1 byte	Field decimal count in binary.
18-19	2 bytes	Reserved.
20	1 byte	Work area ID.
21-30	10 bytes	Reserved.
31	1 byte	Production .MDX field flag; 01h if field has an index tag in the production .MDX file; 00h if the field is not indexed.

#### Table Records

=====

The records follow the header in the table file. Data records are preceded by one byte, that is, a space (20h) if the record is not deleted, an asterisk (2Ah) if the record is deleted. Fields are packed into records without field separators or record terminators. The end of the file is marked by a single byte, with the end-of-file marker, an OEM code page character value of 26 (1Ah). You can input OEM code page data as indicated below.

#### Allowable Input for dBASE Data Types

=====

Data Type	Data Input
C (Character)	All OEM code page characters.
D (Date)	Numbers and a character to separate month, day, and year (stored internally as 8 digits in YYYYMMDD format).

```

F (Floating point binary numeric) - . 0 1 2 3 4 5 6 7 8 9
N (Numeric) - . 0 1 2 3 4 5 6 7 8 9
L (Logical) ? Y y N n T t F f (? when not initialized).
M (Memo) All OEM code page characters (stored internally as 10
digits representing a .DBT block number).

```

Memo Fields And .DBT Files

=====

Memo fields store data in .DBT files consisting of blocks numbered sequentially (0, 1, 2, and so on). SET BLOCKSIZE determines the size of each block. The first block in the .DBT file, block 0, is the .DBT file header.

Each memo field of each record in the .DBF file contains the number of the block (in OEM code page values) where the field's data actually begins. If a field contains no data, the .DBF file contains blanks (20h) rather than a number.

When data is changed in a field, the block numbers may also change and the number in the .DBF may be changed to reflect the new location.

Unlike dBASE III PLUS, if you delete text in a memo field, dBASE 5.0 for DOS may reuse the space from the deleted text when you input new text. dBASE III PLUS always appends new text to the end of the .DBT file. In dBASE III PLUS, the .DBT file size grows whenever new text is added, even if other text in the file is deleted.

This information is from the dBASE for DOS Language Reference manual, Appendix C.

```

*****
The data file header structure for dBASE 5.0 for Windows table file.
*****

```

The table file header:

=====

Byte	Contents	Description
0	1 byte	Valid dBASE for Windows table file; bits 0-2 indicate version number; bit 3 indicates presence of a dBASE IV or dBASE for Windows memo file; bits 4-6 indicate the presence of a dBASE IV SQL table; bit 7 indicates the presence of any .DBT memo file (either a dBASE III PLUS

type or a dBASE IV or dBASE for Windows memo file).

1-3	3 bytes	Date of last update; in YYMMDD format.
4-7	32-bit number	Number of records in the table.
8-9	16-bit number	Number of bytes in the header.
10-11	16-bit number	Number of bytes in the record.
12-13	2 bytes	Reserved; filled with zeros.
14	1 byte	Flag indicating incomplete dBASE IV transaction.
15	1 byte	dBASE IV encryption flag.
16-27	12 bytes	Reserved for multi-user processing.
28	1 byte	Production MDX flag; 01h stored in this byte if a production .MDX file exists for this table; 00h if no .MDX file exists.
29	1 byte	Language driver ID.
30-31	2 bytes	Reserved; filled with zeros.
32-n	32 bytes each	Field descriptor array (the structure of this array is shown below)
n+1	1 byte	0Dh stored as the field terminator.

n above is the last byte in the field descriptor array. The size of the array depends on the number of fields in the table file.

#### Table Field Descriptor Bytes

=====

Byte	Contents	Description
-----	-----	-----
0-10	11 bytes	Field name in ASCII (zero-filled).
11	1 byte	Field type in ASCII (B, C, D, F, G, L, M, or N).
12-15	4 bytes	Reserved.
16	1 byte	Field length in binary.
17	1 byte	Field decimal count in binary.
18-19	2 bytes	Reserved.
20	1 byte	Work area ID.
21-30	10 bytes	Reserved.
31	1 byte	Production .MDX field flag; 01h if field has an index tag in the production .MDX file; 00h if the field is not indexed.

#### Table Records

=====

The records follow the header in the table file. Data records are preceded by one byte, that is, a space (20h) if the record is not deleted, an asterisk (2Ah) if the record is deleted. Fields are packed into records

without field separators or record terminators. The end of the file is marked by a single byte, with the end-of-file marker, an OEM code page character value of 26 (1Ah). You can input OEM code page data as indicated below.

Allowable Input for dBASE Data Types

=====

Data Type	Data Input
B (Binary)	All OEM code page characters (stored internally as 10 digits representing a .DBT block number).
C (Character)	All OEM code page characters.
D (Date)	Numbers and a character to separate month, day, and year (stored internally as 8 digits in YYYYMMDD format).
G (General)	All OEM code page characters (stored internally as 10 digits or OLE) representing a .DBT block number).
N (Numeric)	- . 0 1 2 3 4 5 6 7 8 9
L (Logical)	? Y y N n T t F f (? when not initialized).
M (Memo)	All OEM code page characters (stored internally as 10 digits representing a .DBT block number).

Binary, Memo, and OLE Fields And .DBT Files

=====

Binary, memo, and OLE fields store data in .DBT files consisting of blocks numbered sequentially (0, 1, 2, and so on). SET BLOCKSIZE determines the size of each block. The first block in the .DBT file, block 0, is the .DBT file header.

Each binary, memo, or OLE field of each record in the .DBF file contains the number of the block (in OEM code page values) where the field's data actually begins. If a field contains no data, the .DBF file contains blanks (20h) rather than a number.

When data is changed in a field, the block numbers may also change and the number in the .DBF may be changed to reflect the new location.

Unlike dBASE III PLUS, if you delete text in a memo field (or binary and OLE fields), dBASE for Windows (unlike dBASE IV) may reuse the space from the deleted text when you input new text. dBASE III PLUS always appends new text to the end of the .DBT file. In dBASE III PLUS, the .DBT file size grows whenever new text is added, even if other text in the file is deleted.

This information is from the dBASE for Windows Language Reference manual, Appendix C.

Reference:

7/16/98 4:33:55 PM

Last Modified: 01-SEP-99



## Article #15840: FILE HANDLES EXTENDING

Technical Notes Database

TN840D.txt     FILE HANDLES EXTENDING  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal 4.0+

Description:

Q. How do I open more than 15 files at one time?

A. In DOS version 3.3 and above there is a function call to allocate memory

for additional file handles. This will allow you to open more than 15

files at one time as long as your Files line in your CONFIG.SYS is 5

greater than the number of files you would like to open.

Here is an example program:

```
{ $M $8000,0,$1000 } { Important do not forget }
```

```
Program Many_Files;
```

```
Uses Dos;
```

```
Var
```

```
Files:array[1..30] of file;
```

```
I:Integer;
```

```
Procedure Allow_More_Files( Max_Files:Byte );
```

```
Var
```

```
Regs:Registers;
```

```
Begin
```

```
With Regs do
```

```
Begin
```

```
AH := $67;
```

```
BX := Max_Files;
```

```
Intr( $21,Regs );
```

```
If Flags And FCarry <> 0
```

```
Then
```

```
Writeln( 'Dos error : ',AX);
```

```
End;
```

```
End;
```

```
Begin
```

```
Allow_More_Files( 30 );
For I:=1 to 30 do
  Begin
    Assign(Files[I], 'DAT.'+ Chr(I DIV 10+48) + Chr(I Mod 10
+48));
    Rewrite(Files[I]);
  End;
For I:=1 to 100 do
  Close( Files[I] );
End.
```

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15844: Passing a Variable to a ReportSmith Report

Technical Information Database

TI844D.txt - Passing a Variable to a ReportSmith Report

Category :Delphi 1.x  
Platform :All Windows  
Product :

### Description:

The following code demonstrates how to pass variables to a ReportSmith report.

Note: this technical information sheet assumes that you know how to create a report that includes report variables. For additional information see chapter four of the ReportSmith manual 'Creating Reports' that shipped with Delphi.

In this case we are assigning a value ('CA') to a string report variable named 'state'.

### ReportSmith Code:

This is the info in the ReportVariables Dialog box in ReportSmith. You can get to this dialog from the reportsmith menu by choosing Tools | Report Query and pushing the Report Variables button.

Name: state  
Type: string  
Title: state var  
Prompt: Enter your favorite state.  
Entry: type-in  
Report Variables: state ; Note this variable and the value it holds are  
  
both case sensitive when passed to  
ReportSmith.

### Delphi Code:

This code assumes that you have placed a TReport component on your form named 'Report1' and set the ReportName property to the name of the

report

that will be accepting the variable as defined above.

Place the following code in the OnClick Method of a pushbutton on your form. I use a pushbutton for simplicity, but this code could just as easily be triggered by any other event.

```
procedure TForm1.Button1Click(Sender: TObject);
var s: string;
begin
  s:='CA';
  Report1.InitialValues.Add('
state=<'+s+'>');
  Report1.run;
end;
```

Reference:

3/31/99 10:30:46 AM

Last Modified: 01-SEP-99

## Article #15846: DOS - PROCEDURE FLUSH

Technical Notes Database

TN846D.txt      DOS - PROCEDURE FLUSH  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. What does the Flush procedure do?

A. This procedure flushes Turbo Pascal's buffers to DOS's buffers.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15850: Extracting Index Data From A Table

Technical Information Database

TI850D.txt     Extracting Index Data From A Table  
Category     :Database Programming  
Platform     :All  
Product      :Delphi All

### Description:

Getting a list of the indexes associated with a table at run-time can be as simple as a call to the `GetIndexNames` method of the `TTable`, `TQuery`, or `TStoredProc` component. The `GetIndexNames` method returns a list of the indexes that are available for the data set in the form of a `TStrings` list, which may then be inserted into such visual components as a `TListBox` through its `Items` property:

```
ListBox1.Clear;  
Table1.GetIndexNames(ListBox1.Items);
```

Of course, the `TStrings` list returned by the `GetIndexNames` method need not be used with a visual component. It could just as well serve as an array of index names stored entirely in memory, that can be used as a list or array.

But it is also possible to retrieve much more information about the indexes for a table than just the names. Other descriptive attributes include the name of each index, the names of the fields that comprise each index, and the index options used when each index was created. Retrieving these values is slightly more involved than the use of the `GetIndexNames`. Basically, this process involves iterating through the `IndexDefs` property

of the TTable, TQuery, or TStoredProc component. The IndexDefs property is essentially an array of records, one record for each index for the table.

Each index record contains information about the index. It is a relatively straightforward process to iterate through this array of index descriptions, extracting information about individual indexes.

The IndexDefs property of the TTable component contains information about the indexes for the table used by the TTable, TQuery, or TStoredProc component in use. The IndexDefs property itself has various properties that allow for the extraction of information about specific indexes. The two properties in the IndexDefs object are:

Count: type Integer; available only at run-time and read-only; indicates the number of entries in the Items property (i.e., the number of indexes in the table).

Items: type TIndexDef; available only at run-time and read-only; an array of TIndexDef objects, one for each index in the table.

The Count property of the IndexDefs object is used as the basis for a loop program construct to iterate through the Items property entries to extract specific information about each index. Each IndexDef object contained in the Items property consists of a number of properties that provide various bits of information that describe each index. All of the properties of the IndexDef object are available only at run-time and are all read-only. These properties are:

Expression: type String; indicates the expression used for dBASE multi-field indexes.

Fields: type String; indicates the field or fields upon which the

index is based.

Name: type String; name of the index.

Options: type TIndexOptions; characteristics of the index  
(ixPrimary,  
ixUnique, etc.).

Before any index information (Count or Items) can be accessed, the Update method of the IndexDefs object must be called. This refreshes or initializes the IndexDef object's view of the set of indexes.

## Examples

=====

Here is a simple For loop based on the Count property of the IndexDefs object that extracts the name of each index (if any exist) for the table represented by the TTable component Table1:

```
procedure TForm1.ListBtnClick(Sender: TObject);
var
  i: Integer;
begin
  ListBox1.Items.Clear;
  with Table1 do begin
    if IndexDefs.Count > 0 then begin
      for i := 0 to IndexDefs.Count - 1 do
        ListBox1.Items.Add(IndexDefs.Items[i].Name)
      end;
    end;
  end;
end;
```

Below is an example showing how to extract information about indexes at run-time, plugging the extracted values into a TStringGrid (named SG1).

```
procedure TForm1.FormShow(Sender: TObject);
var
  i: Integer;
  S: String;
begin
  with Table1 do begin
    Open;
    {Refresh IndexDefs object}
```



```

IndexDefs.Update;
if IndexDefs.Count > 0 then begin
  {Set up columns and rows in grid to match IndexDefs items}
  SG1.ColCount := 4;
  SG1.RowCount := IndexDefs.Count + 1;
  {Set grid column labels to TIndexDef property names}
  SG1.Cells[0, 0] := 'Name';
  SG1.ColWidths[0] := 200;
  SG1.Cells[1, 0] := 'Fields';
  SG1.ColWidths[1] := 200;
  SG1.Cells[2, 0] := 'Expression';
  SG1.ColWidths[2] := 200;
  SG1.Cells[3, 0] := 'Options';
  SG1.ColWidths[3] := 300;
  {Loop through IndexDefs.Items}
  for i := 0 to IndexDefs.Count - 1 do begin
    {Fill grid cells for current row}
    SG1.Cells[0, i + 1] := IndexDefs.Items[i].Name;
    SG1.Cells[1, i + 1] := IndexDefs.Items[i].Fields;
    SG1.Cells[2, i + 1] := IndexDefs.Items[i].Expression;
    if ixPrimary in IndexDefs.Items[i].Options then
      S := 'ixPrimary, ';
    if ixUnique in IndexDefs.Items[i].Options then
      S := S + 'ixUnique, ';
    if ixDescending in IndexDefs.Items[i].Options then
      S := S + 'ixDescending, ';
    if ixCaseInsensitive in IndexDefs.Items[i].Options then
      S := S + 'ixCaseInsensitive, ';
    if ixExpression in IndexDefs.Items[i].Options then
      S := S + 'ixExpression, ';
    if S > ' ' then begin
      {Get rid of trailing ", "}
      System.Delete(S, Length(S) - 1, 2);
      SG1.Cells[3, i + 1] := S;
    end;
  end;
end;
end;
end;
end;

```

## Special Considerations

=====

There are idiosyncracies associated with extracting information about indexes for different table types that Delphi can access.

## dBASE Tables

-----

With dBASE indexes, which properties of Fields and Expression will be filled will depend on the type of index, simple (single-field) or complex (based on multiple fields or a dBASE expression). If the index is a simple one, the Fields property will contain the name of the field in the table on which the index is based and the Expression property will be blank. If the index is a complex one, the Expression property will show the expression on which the index is based (e.g., "Field1+Field2") and the Fields property will be blank.

## Paradox Tables

-----

With Paradox primary indexes, the Name property will be blank, the Fields property will contain the field(s) on which the index is based, and the Options property will contain ixPrimary. With secondary indexes, the Name property will contain the name of the secondary index, the Fields property will contain the field(s) on which the index is based, and the Options property may or may not have values.

The Fields property for indexes based on more than one field will show the field names separated by semi-colons. Indexes based on only a single field will show the name of only that one field in the Fields property.

## InterBase Tables

-----

For both index types, single- or multiple-field, the Expression property will be blank. For single-field indexes, the Fields property will contain the field on which the index is based. For multi-field indexes, the Fields

property will show all of the multiple fields that comprise the index, each separated by a semi-colon.

Indexes designated as PRIMARY when the CREATE TABLE command is issued will have "RDB\$PRIMARYn" in the Name property, where n is a number character uniquely identifying the primary index within the database metadata. Secondary indexes will show the actual name of the index.

Foreign key constraints also result in an index being created by the system. These indexes appear in the IndexDefs property, and will have the name "RDB\$FOREIGNn" where n is a number character that uniquely identifies the index within the database metadata.

The Fields property for indexes based on more than one field will show the field names separated by semi-colons. Indexes based on only a single field will show the name of only that one field in the Fields property.

Reference:

7/16/98 4:33:55 PM

Last Modified: 01-SEP-99

## Article #15852: DOS - ILLEGAL FILENAME; COMMAND.COM OVERWRITE

Technical Notes Database

TN852D.txt      DOS - ILLEGAL FILENAME; COMMAND.COM OVERWRITE  
Category      :Pascal  
Platform        :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. I load Turbo Pascal and I get the message, "Illegal filename press escape." What's wrong?

A. Apparently, you must have run an installer type of program that overwrote your copy of COMMAND.COM with its own version. You must recopy COMMAND.COM from your DOS master disk.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15856: Redistributing the 16 bit Borland Database Engine

Technical Information Database

TI856D.txt - Redistributing the 16 bit Borland Database Engine

Category :Delphi 1.x  
Platform :All-32Bit  
Product :

### Description:

Follow the following steps in order to deploy the redistributable BDE engine on a target machine:

- 1) Format two diskettes in the diskette drive of the target machine. Label the diskettes "Disk 1" and "Disk 2".
- 2) From the DELPHI CD, Copy the \REDIST\BDEINST\DISK1 directory to the diskette labeled "Disk 1" and \REDIST\BDEINST\DISK2 to the diskette "Disk 2".
- 3) Insert the diskette labeled "BDE Install 1" to a floppy drive (We'll use drive A: in this example) of the target machine.
- 4) Make sure there are no other programs running in Windows. From the Windows Program Manager select File|Run, enter "A:\DISK1\SETUP" in the space labeled "Command Line" and press "OK" to begin install of the Borland Database Engine on the target machine.
- 5) A dialog labeled "Database Engine Install" will appear briefly, then a dialog labeled "preparing to install...", and finally a background screen labeled "BDE Redistributable" will appear along with a dialog allowing you to Continue or Exit. Press "Continue".
- 6) A dialog "Borland Database Engine Location Settings" will appear and allow you to change the path for BDE programs and configuration file. Leave the default settings at and press "Continue".

- 7) The "Borland Database Engine Installation" dialog will appear and allow you to display previous dialogs, or execute the install. Press "Install".
- 8) Installation progress will display while files from the "Disk 1" diskette are processed.
- 9) The "BDE Redistributable Install Request" dialog will appear. Load the diskette labeled "Disk 2". Press "continue".
- 10) On completion, the "Borland Database Engine Installation Notification" dialog will inform you that BDE is installed. Press "Exit".
- 11) Exit Windows, remove any floppy disks from your drives and reboot the target machine.

If default settings were used, changes listed below will have taken place.

Two new directories, \IDAPI, and \IDAPI\LANGDRV will exist on the target machine. Note that the BDE Configuration Utility, BDECFG.EXE is located in the \IDAPI directory. Language drivers will now be located in \IDAPI\LANGDRV as \*.LD files. AUTOEXEC.BAT, CONFIG.SYS, AND SYSTEM.INI will not be affected.

WIN.INI in the \WINDOWS\SYSTEM directory will have new entries:

```
[IDAPI]
DLLPATH=C:\IDAPI
CONFIGFILE01=C:\IDAPI\IDAPI.CFG
```

```
[Borland Language Drivers]
LDPath=C:\IDAPI\LANGDRV
```

Reference:

4/22/99 12:27:23 PM



## Article #15858: DOS FILES - PROCEDURE FINDFIRST

Technical Notes Database

TN858D.txt      DOS FILES - PROCEDURE FINDFIRST

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 4.0+

Description:

Q. Can I use procedure FindFirst to search for files that have been erased?

A. The procedure FindFirst will not find erased files.

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99



## Article #15862: Cascading Deletes With Pdox Referential Integrity

Technical Information Database

TI862D.txt Cascading Deletes With Pdox Referential Integrity  
Category :Database Programming  
Platform :All  
Product :Delphi 1.0

### Description:

Paradox tables offer a Referential Integrity feature. This feature prevents adding records to a child table for which there is no matching record in the parent table. It will also cause the key field(s) in the child table to be changed when the corresponding key field(s) in the parent are changed (commonly referred to as a cascading update). These events occur automatically, requiring no intervention by a Delphi application using these tables. However, the Paradox Referential Integrity feature will not accommodate cascading deletes. That is, Delphi will not allow you to delete a record in the parent table while matching records exist in the child table. This would make "orphans" of the child records, losing referential integrity. Delphi raises an exception when an attempt is made to delete such a parent record.

To effect a cascading delete requires that the deletion of the matching child records be deleted programmatically -- before the parent record is deleted. In a Delphi application, this is done by interrupting the process of deleting the record in the parent table, deleting the matching records in the child table (if there are any), and then continuing with the deletion of the parent record.

A record in a table is deleted by a call to the Delete method of the

TTable component, which deletes the current record in the associated table. Interrupting the this process to first perform some other operations is a matter creating a procedure associated with the BeforeDelete event of the TTable. Any commands in a BeforeDelete event procedure are executed before the call actually goes out from the application to the Borland Database Engine (BDE) to physically remove the record from the table file.

To handle the deletion of one or more child records, in a BeforeDelete event procedure the Delete method for the TTable representing the child table is called in a loop. The loop is based on the condition of the record pointer in the table not being positioned at the end of the data set, as indicated by the Eof method of the TTable. This also accounts for there being no child records at all matching the parent record to be deleted: if there are no matching records, the record pointer will already be at the end of the data set, the loop condition will evaluate to False, and the Delete method in the loop never gets executed.

```
procedure TForm1.Table1BeforeDelete(DataSet: TDataset);
begin
  with Table2 do begin
    DisableControls;
    First;
    while not Eof do
      Delete;
    EnableControls;
  end;
end;
```

In the above example, the parent table is represented by the TTable component Table1 and the child by Table2. The DisableControls and EnableControls methods are used as a cosmetic measure to freeze any data-aware components that might be displaying data from Table2 while the records are being deleted. These two methods make the process visually appear smoother, but are only optional and not essential to this process. The

Next method need not be called within this loop. This is because the loop begins at the first record and, as each record is deleted, the record that previously followed the deleted record moves up in the data set, becoming both the first and the current record.

This example presumes that the parent and child tables are linked with a Master-Detail relationship, as is typical for tables for which such Referential Integrity is configured. Linking the tables in this manner results in only those records in the child table that match the current record in the parent table being available. All other records in the child table are made unavailable through the Master-Detail filtering. If the tables are not so linked, there are two additional considerations that must be accounted for when deleting the child records. The first is that a call to the First method may or may not put the record pointer on a record that matches the current record in the parent table. This necessitates using a search method to manually move the record pointer to a matching record. The second consideration affects the condition for the loop. Because records other than those matching the current record in the parent table will be accessible, the condition for the loop must check that each record is a matching record before attempting to delete it. This checking is in addition to querying the Eof method. Because the records will be ordered by this key field (from a primary or secondary index), all of the matching records will be contiguous. This leads to the given that, as soon as the first non-matching record is reached, it can be assumed that all matching records have been deleted. Thus, the previous example would be modified to:

```
procedure TForm1.Table1BeforeDelete(DataSet: TDataset);
begin
  with Table2 do begin
    DisableControls;
```

```
FindKey([Table1.Fields[0].AsString])
while (Fields[0].AsString = Table1.Fields[0].AsString) and
  (not Eof) do
  Delete;
  EnableControls;
end;
end;
```

In the above, it is the first field in the parent table (Table1) upon which the Referential Integrity is based, and the first field in the child table (Table2) against which matching is judged.

Reference:

7/16/98 4:33:55 PM

Last Modified: 01-SEP-99

## Article #15864: DOS - CHDIR PROCEDURE

Technical Notes Database

TN864D.txt      DOS - CHDIR PROCEDURE

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 4.0+

Description:

Q. How do you use procedure ChDir to go to the root directory?

A. Use ChDir('\');

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15868: dBASE Expression Indexes: A Primer

Technical Information Database

TI868D.txt      dBASE Expression Indexes: A Primer  
Category      :Database Programming  
Platform       :All  
Product       :Delphi All

### Description:

Indexes for dBASE tables may be based on the values from a single field, unmodified, or on an expression. Index expressions, unique to dBASE indexes, may be composed of multiple fields, modifications of field values, or combinations of these. The expression for a dBASE expression index is created by using dBASE functions and syntax to concatenate multiple fields or to perform the modifications of field values for fields included in the index expressions.

Two sections are included at the end of this technical article which describe the mechanics of creating dBASE expression indexes, one applicable to doing this in the Database Desktop utility and the other for including this capability in Delphi applications.

### Expression Indexes Based On Multiple Fields

=====

dBASE functions are available for use in Delphi or the Database Desktop for the express use in index expressions, and then only in conjunction with dBASE indexes. That is, you cannot use dBASE functions or syntax to build an index expression for a Paradox or Local InterBase Server (LIBS) table. Nor can dBASE functions be used in Delphi programming. They are only available for dBASE expression indexes. The dBASE functions and syntax that can be used for expression indexes are provided by the Borland Database Engine (BDE) Dynamic Linked Library (DLL) file IDDBAS01.DLL.

When creating a dBASE index that is to be based on the values from two or more fields in the table for which the index is being created, the two or more fields are concatenated (connected together) in a manner similar to how String type values are

concatenated in Delphi syntax: the "+" operator. For example, the expression needed to create an index that orders first on a LastName field and then on a FirstName field would be:

```
LastName + FirstName
```

Unlike in dBASE itself, such indexes that are based on multiple fields are limited to using just those fields in the one table. dBASE allows the creation of indexes based on multiple fields contained in different tables. This is possible only by having the other table open at the time the index is created or when the table containing the index is used.

With multi-field indexes for other table types (e.g., Paradox and InterBase), the multiple fields are delimited by the semi-colon (;), as in:

```
LastName;FirstName
```

In dBASE expression indexes that concatenate multiple fields, an actual expression is used:

```
LastName + FirstName
```

When creating index expressions that concatenate two or more fields, all of the fields included in the index expression must be of the same data type. Additionally, if they are to be concatenated instead of added together, the fields must all be of String type. For example, given two Integer type fields, Value1 and Value2, the index expression...

```
Value1 + Value2
```

...would not cause an error. But then, neither would it concatenate the two field values; it would add them together. Thus, if Value1 for a given record contained 4 and Value2 5, the resulting index node would be an Integer value of 9, not a String concatenation "45".

If fields to be included in an expression index are not of String type, they must be converted. Here are some dBASE functions to convert various data types to String for purposes of creating index expressions:

```
STR( [, [, ]])
```

Converts from either Float or Numeric dBASE types to Character (String)

DTOS()

Converts Date value to Character, format YYYYMMDD

MLINE(, )

Extracts a single line from a memo field as a Character value

Another consideration in creating indexes based on the concatenation of multiple field is the maximum allowable length of the index value. The value returned by an index expression may not exceed 100 characters. This is a limit on the length of the value returned by the expression, not on the length of the expression itself. For example, you cannot index on the concatenation of two fields that both have a length of 255 characters.

Expression Indexes Based On Modifications Of Field Values

=====

In addition to creating indexes based on the concatenation of two or more field values, it is also possible to construct an index that is based on a modification of a field value. Examples of this include indexing on just the first three characters of a String type field, on just the year and month from a Date field, indexing on a concatenation of a String and Date field such that the ordering of the String field is ascending but the Date descending, and even indexing on Boolean field values.

Creating indexes that are based on modifications of field values requires at least a working knowledge of dBASE functions and syntax -- because the process uses dBASE, and not Delphi, functions and syntax. The dBASE function SUBSTR() extracts a substring of a String value. The Delphi equivalent for this dBASE function is Copy. But, of these two functions that serve the same purpose, only SUBSTR() may be used in dBASE index expressions.

To use dBASE functions in dBASE index expressions, simply include the function wherever an index expression is called for, using the appropriate dBASE syntax for the function, along with a reference to the name(s) of the field(s) used in the function. For example, an index expression based on only the last three characters of a String type field called Code, that is 20 characters long, would be:



```
RIGHT(Code, 3)
```

Caution should be used in constructing dBASE index expressions that modify field values to ensure that the resulting expression would return a value of a consistent length for every record in the table. For instance, the dBASE TRIM() function removes the trailing blanks (ASCII decimal 32) from a String type value. If this were used in conjunction with concatenating two String fields where the field does not contain values of the same length for all records, the value resulting from the expression will not be the same for all records. Case in point, an index expression based on the concatenation of a LastName and a FirstName field, where the TRIM() function is applied to the LastName field:

```
TRIM(LastName) + FirstName
```

This expression would not return values of a consistent length for all records. If the LastName and FirstName fields contained the values...

```
LastName  FirstName
-----  -----
Smith     Jonas
Wesson    Nancy
```

...the result of applying the index expression above would be:

```
SmithJonas
WessonNancy
```

As can be seen, the length of the value for the first record would be 10 characters, while that for the second 11 characters. The index nodes created for this index expression would be based on the field values for the first record encountered. This would result in an index node 10 characters long being applied to the field values for all record. In this example, that would result in the truncation of the expression value for the second record to "WessonNanc". This would subsequently cause searches based on the full field value to fail.

The solution to this dilemma would be to not use the TRIM() function so that the full length of the LastName field, including padding from the trailing spaces, is used. In indexes that use the IIF() function to order by one field or another, based on the

evaluation of a logical expression in the IIF(), if the two fields are of different lengths, the shorter field would need to be padded with spaces to make it the same length as the longer field. For example, assuming an index that uses the IIF() function to index either on a Company or a Name field, based on the contents of Category field, and where the Company field is 40 characters long but the Name field is 25 characters long, the Name field would need to be padded with 15 spaces; accomplished with the dBASE function SPACE(). That index expression would then be:

```
IIF(Category = "B", Company, Name + SPACE(15))
```

#### Searches And dBASE Expression Indexes

=====

dBASE expression indexes are exceptions to the norm in how they are handled by Delphi and the BDE as opposed to how multiple field indexes for other table types are handled.

This puts such dBASE indexes into a separate class. Handling of such indexes by Delphi and the BDE is different than those for other table types. One of these differences is that not all index-based searching using Delphi syntax can be used with these dBASE expression indexes. The FindKey, FindNearest, and GotoKey methods of the TTable component cannot be used with expression indexes. If an attempt to use FindKey is made, this will result in the error message: "Field index out of range." If the GotoKey method is tried, this error message may occur or the record pointer may just not move (indicating the search value was not found). Only the GotoNearest method may be used with expression indexes. Even then, the GotoNearest method may not work with some index expressions. Experimentation will be needed to see whether the GotoNearest method will work with a given index expression.

#### Filtering With dBASE Expression Indexes

=====

As with index-based searches, dBASE expression indexes present some exceptions when using Delphi filtering.

With an expression index active, the SetRange method of the TTable component will produce the error: "Field index out of range." However, with the same expression index active, the SetRangeStart and SetRangeEnd methods will successfully filter

the data set.

For example, with an expression index concatenating a LastName and a FirstName field active, the code below using the FindKey method (intended to filter to just those records where the first character of the LastName field is "S") will fail with an error:

```
begin
  Table1.SetRange(['S'], ['Szzz'])
end;
```

Whereas, the code below, with the same expression index active and filtering on the same LastName field, will successfully filter the data and not incur an error:

```
begin
  with Table1 do begin
    SetRangeStart;
    FieldByName('LastName').AsString := 'S';
    SetRangeEnd;
    FieldByName('LastName').AsString := 'Szzz';
    ApplyRange;
  end;
end;
```

And, as is the case with index-based searches, with filtering, success of a filtering attempt will also be dependent on the index expression. The use of the SetRangeStart and SetRangeEnd methods in the preceding example worked with an index that simply concatenated two String type fields. But if the expression for the index was instead based conditionally on one or the other fields through use of the IIF() function, the same filtering routine would fail (although without an error).

### Some Handy dBASE Index Expressions

=====

Here are some handy dBASE index expressions. Some are readily apparent in the intended purpose, others are more arcane.

Character field ascending and Date field descending

-----

With a Character field called Name and a Date field OrdDate:

Name + STR(OrdDate - {12/31/3099}, 10, 0)

Character field ascending and Numeric (or Float) field descending

-----  
With a Character field called Company and a Numeric field Amount  
(the Amount field being 9 digits wide with two decimal places):

Company + STR(Amount - 999999.99, 9, 2)

Ordering by a Logical field

-----  
To have True values appear before False values for a Logical  
field called Paid:

IIF(Paid, "A", "Z")

Two Numeric (or Float) fields

-----  
Assuming two Numeric fields with widths of five and two decimal  
places, the first field named Price and the second Quantity:

STR(Price, 5, 2) + STR(Quantity, 5, 2)

Ordering by one field of two, depending on a logical condition

-----  
Assuming that if the Company field is empty, the record should be  
included in the sort order by the Name field (instead of an empty  
Company field).

IIF(Company = " ", Name, Company)

Ordering by the names of months in a Character field

-----  
Assuming a field containing the names of the months ("Jan," "Feb"  
etc.) to put the records in proper month order (field named M):

IIF(M="Jan", 1, IIF(M="Feb", 2, IIF(M="Mar", 3, IIF(M="Apr", 4,  
IIF(M="May", 5, IIF(M="Jun", 6, IIF(M="Jul", 7, IIF(M="Aug", 8,  
IIF(M="Sep", 9, IIF(M="Oct", 10, IIF(M="Nov", 11, 12))))))))))

(The above is a single expression line, broken into multiple lines here due to page width.)

Ordering by the first line of a memo field

-----

For a memo field named Notes:

```
MLINE(Notes, 1)
```

Ordering by the middle three characters in a nine character long field

-----

For a nine character long field called StockNo:

```
SUBSTR(StockNo, 4, 3)
```

Creating dBASE Expression Indexes In Database Desktop

=====

In the Database Desktop utility, indexes may be created for a table either during the process of creating a new table or by restructuring an existing table. In both cases, the Define Index dialog is used to create one or more indexes for the table used.

To get to the Create Index dialog while creating a new table, in the Create dBASE Table dialog (showing the structure), from the Table Properties listbox select "Indexes" and click the Define button.

To get to the Create Index dialog to create an index for an existing table, select Utilities|Restructure, select the table file from the Select File dialog, and in the Restructure dBASE Table dialog (showing the table structure) from the Table Properties listbox select "Indexes" and click the Define button.

Once in the Create Index dialog, expression indexes can be created by clicking the Expression Index button and entering the expression to be used in the Expression Index entry field. To assist in this process, you can double-click on a field name in the Field List listbox and that field name will be inserted into the Index Expression entry field at the current insertion point (caret position).

Once the index expression has been entered, click the OK button. Enter the name of the new index tag in the Index Tag Name entry field on the Save Index As dialog and click OK. (Remember, dBASE index tag names cannot exceed ten characters in length and must abide by the normal dBASE naming conventions.)

## Creating dBASE Expression Indexes In Delphi Applications

=====

dBASE indexes can be created programmatically in Delphi applications, either as a new table is being created (CreateTable method of the TTable component) or by adding an index to an existing table.

Another peculiarity of the dBASE expression index and the BDE is that the table must exist prior to creating an expression index. Thus, while single-field indexes may be created as the table is created by populating TIndexDef objects, this cannot be done with expression indexes. Expression indexes can only be added to a newly created table after the call is made to the CreateTable method, using the AddIndex method. The Options parameter of the AddIndex method must include the index option value ixExpression. This index option is unique to dBASE indexes, and should only be used with dBASE expression indexes. For example:

```
with Table1 do begin
  Active := False;
  DatabaseName := 'Delphi_Demos';
  TableName := 'CustInfo';
  TableType := ttdBASE;
  with FieldDefs do begin
    Clear;
    Add('LastName', ftString, 30, False);
    Add('FirstName', ftString, 20, False);
  end;
  CreateTable;
  AddIndex('FullName', 'LastName + FirstName', [ixExpression]);
end;
```

## Learning More About dBASE Functions And Syntax

=====

Only dBASE functions and syntax that apply to data manipulation can be used to construct a dBASE expression index. However, it is beyond the scope of this technical article to fully list and

describe all of these functions. To learn more about dBASE data manipulation functions, the user is advised to consult the dBASE Language Reference manual or one of the many third-party dBASE books.

Reference:

7/16/98 4:33:55 PM

Last Modified: 01-SEP-99

## Article #15870: GETINFO - FREE MEMORY IN DOS

Technical Notes Database

TN870D.txt GETINFO - FREE MEMORY IN DOS

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

- Q. How come the GetInfo menu shows that the amount of free memory is zero but MemAvail and MaxAvail indicate free memory?
- A. The menu refers to the memory recognized by DOS, whereas MemAvail and MaxAvail refer to the free memory internal to your program. When a program loads it may take away all of the available memory from DOS. This memory would be used inside your program for the heap, overlays, and the stack, for example.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99



## Article #15874: Removing the vertical scrollbar from a TDBGrid

Technical Information Database

TI874D.txt Removing the vertical scrollbar from a TDBGrid

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

In order to remove the vertical scrollbar from a TDBGrid component, you must override its Paint method. Inside the Paint method you must call the SetScrollRange API procedure to set the min and max scroll values to zero (this disables the scrollbar), and then call the inherited Paint. The code below is a unit containing a new component called TNoVertScrollDBGrid that does this. You can copy the code into a file called NEWGRID.PAS, and add it to the component library as a custom component.

```
unit Newgrid;

interface

uses
  WinTypes, WinProcs, Classes, DBGrids;

type
  TNoVertScrollDBGrid = class(TDBGrid)
  protected
    procedure Paint; override;
  end;

procedure Register;

implementation

procedure TNoVertScrollDBGrid.Paint;
begin
  SetScrollRange(Self.Handle, SB_VERT, 0, 0, False);
  inherited Paint;
end;
```

```
procedure Register;  
begin  
  RegisterComponents('Data Controls', [TNoVertScrollDBGrid]);  
end;  
  
end.
```

Reference:

7/16/98 4:33:56 PM

Last Modified: 01-SEP-99

## Article #15876: OOP - DATA AND CODE SEGMENT SIZES

Technical Notes Database

TN876D.txt OOP - DATA AND CODE SEGMENT SIZES

Category :Pascal

Platform :All

Product :Turbo Pascal 5.5

Description:

Q. When I select Compiler/GetInfo, why does the program's data and code segment sizes increase with version 5.5?

A. There is more overhead due to the object oriented features and enhanced overlay manager. For example, the System unit must interface to more variables.

Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15882: 4.0 TOOLBOX COMPATIBILITY

Technical Notes Database

TN882D.txt      4.0 TOOLBOX COMPATIBILITY  
Category      :Pascal  
Platform       :All  
Product       :Turbo Pascal 5.0 5.5

### Description:

Q. Will the toolboxes for 4.0 work with 5.0+?

A. Yes. All 4.0 versions of the toolboxes will work with Turbo Pascal 5.0+. In a few cases, minor changes to compiler directives are recommended. Refer to the Turbo Pascal README file for more information.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15886: InterBase BLOB Fields: A Primer

Technical Information Database

TI886D.txt InterBase BLOB Fields: A Primer

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

InterBase BLOB fields are not all the same. They actually consist in a variety of forms, or sub-types of the general BLOB type. Knowing which sub-type of BLOB field to use when is essential to creating database applications that incorporate InterBase BLOB fields. BLOB fields come in three varieties: sub-type 0 and sub-type 1 (the two predefined sub-types), and user-defined sub-types.

Sub-type 0 BLOB fields are the type created when a CREATE command is issued and a sub-type is not specified. For clarity in SQL syntax, though, it is possible to explicitly indicate that the BLOB field is to be of sub-type 0. This sub-type of BLOB field is for the storage of binary data. InterBase makes no analysis of the data stored, it just stores it in the BLOB field on a byte-for-byte basis. The most common intended use for BLOB fields in Windows applications is the storage of bitmap binary data, typically for display in a TDBImage component. Either the BLOB field sub-type 0 or a user-defined sub-type BLOB field will work for this purpose.

The second predefined sub-type is 1. This BLOB field sub-type is designed for the storage of text. Typically, this is the free-form memo or notes data displayed and edited with the TDBMemo component. This BLOB field sub-

type is better for storing text data than the VARCHAR field because, unlike with the VARCHAR field, there is no design-time limit placed on the storage capacity of the field.

In SQL syntax, the sub-type 1 BLOB field is created by following the BLOB field type keyword with the SUB\_TYPE keyword and the integer one:

```
CREATE TABLE WITHBLOB
(
  ID CHAR(3) NOT NULL PRIMARY KEY,
  MEMO BLOB SUB_TYPE 1,
  AMOUNT NUMERIC
)
```

Aside from the two predefined BLOB field sub-types, there are user-defined sub-types. User-defined sub-types are designated by a negative integer value in association with the SUB\_TYPE keyword. The actual integer value, as long as it is negative, is actually arbitrary and up to the discretion of the table creator. A designation of -1 is functionally the same as that of a -2. The only consideration when using user-defined sub-types is ensuring that the same type of binary data is stored for every row in the table for a BLOB field of a given user-defined sub-type. InterBase will not evaluate whether this criteria is met, and it is the responsibility of the application inserting the binary data to store the appropriate type of data. No error will occur from the InterBase side if an incorrect type of binary data is stored in a user-defined BLOB field sub-type, but an application can incur difficulties if it is expecting one type of data but encounters another.

A BLOB field of a user-defined sub-type is created with the SQL syntax such as that below:

```
CREATE TABLE IMAGE_DATA
```

```
(
  FILENAME CHAR(12) NOT NULL PRIMARY KEY,
  BITMAP BLOB SUB_TYPE -1,
  EXEs BLOB SUB_TYPE -2,
)
```

When using a table created with the above command, the field BITMAP would only be used to store one distinct type of binary data for all records. In this case, bitmap data. The field EXEs implies the storage of executable files loaded from disk. If an application using this table were to mistakenly store binary data that should have been in the EXEs field into the BITMAP field, InterBase would generate no errors, but the application would have extreme difficulties displaying a stored executable file in a TDBImage component.

#### InterBase BLOB fields and Delphi

-----

When defining TField objects for InterBase BLOB fields in Delphi, the various BLOB field sub-types are assigned TField derivative types as follows:

```
Sub-type 0:   TBlobField
Sub-type 1:   TMemofield
User-defined: TBlobField
```

Because both the predefined sub-type 0 and user-defined sub-types are recognized as TBlobField objects, care must be taken when designing an application to not mistake a field of one sub-type for that of another.

The only way to differentiate between a field of sub-type 0 from that of a user-defined type is by viewing the metadata information for the table, which cannot be done from within Delphi. The Local InterBase Server utility WISQL can be used to view table metadata.

#### InterBase BLOB fields and Database Desktop

-----

The Database Desktop utility that comes with Delphi (DBD) does not create user-defined subtypes. When using BLOB fields created in Database Desktop, use the "BLOB" field type for binary data, including bitmap data. This creates a BLOB field of the predefined sub-type 0.

The DBD also offers a BLOB field type TEXT BLOB. This is equivalent to the pre-defined subtype 1, and should be used where free-form text storage will be needed. While it is functionally equivalent to the pre-defined subtype 1 BLOB field, it will appear with a slightly different type designation if you view the metadata for the table in the WISQL utility.

Reference:

7/16/98 4:33:56 PM

Last Modified: 01-SEP-99



## Article #15888: UPGRADING 3.0 PROGRAMS WITH CHAIN FILES

Technical Notes Database

TN888D.txt UPGRADING 3.0 PROGRAMS WITH CHAIN FILES

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

Q. How do you go about upgrading version 3.0 programs with lots of chain files?

A. Chaining is not possible with .EXE files. Control can be passed to another program by use of the EXEC procedure in the DOS unit. You can also use 5.0+'s overlay manager to build very large programs.

### Reference:

7/16/98 4:35:36 PM

Last Modified: 01-SEP-99

## Article #15892: Using The ASCII Driver With Comma-delimited Files

Technical Information Database

TI892D.txt Using The ASCII Driver With Comma-delimited Files

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

Delphi (and the BDE) has the capability to use ASCII files to a limited

degree as tables. The ASCII driver has the capability to translate the data values in an ASCII fixed-length field or a comma-delimited file into

fields and values that can be displayed through a TTable component.

How

this translation of the ASCII file takes place depends on an accompanying

schema file. The schema file for an ASCII data file defines various attri-

buties necessary for parsing the ASCII data file into individual field values. The field definitions for an ASCII fixed-length field file is relatively straightforward, the offsets of various fields in the ASCII file being consistent across all rows in the file. However, for comma-delimited files, this process is slightly more complicated due to the fact that not all data values in such a file may be the same length for

all rows in the file. This article, then, concentrates on this more difficult task of reading data from comma-delimited, or varying-length field, files.

### The Schema File

=====

The schema file for an ASCII data file contains information that defines

both the file type (comma-delimited versus fixed-length field), as well as

defining the fields that are represented by the data values in each row of

the ASCII data file. (All of the settings used in a schema file are

case insensitive, so "ascii" is just as valid as "ASCII".) In order that a schema file be recognized as such, it must have the same filename as the ASCII data file for which it provides definitions, but with the filename extension .SCH (for SCHEMA). The attributes that describe the file are:

**File name:** Enclosed in square brackets, this setting specifies the name of the ASCII data file (sans the filename extension, which must be .TXT).

**Filetype:** Specifies whether the ASCII data file is structured as a fixed-length field file (use a setting of FIXED) or a comma-delimited file (with data values of potentially varying length (use a setting of VARYING)).

**Delimiter:** Specifies the character that surrounds String type data values (typically, the double quotation mark, ASCII decimal 34).

**Separator:** Specifies the character that is used to separate individual data values (typically, a comma). This character must be a visible character, i.e., cannot be a space (ASCII decimal 32).

**CharSet:** Specifies the language driver (use a setting of ASCII).

Following the file definition settings are field definitions, one for each data value on each row of the ASCII data file. These field definitions supply the information Delphi and the BDE will need to create a virtual field in memory to hold the data value, that virtual field's data type which will affect how the value is translated after being read from the ASCII file, and size and positioning attributes. The various settings that will appear in each field definition are:

Field: Virtual field name, will always be "Field"  
followed  
ord-  
the  
Field1,  
the second Field2, and so on.

Field name: Specifies the display name for the field, which  
Naming appears as the column header in a TDBGrid.  
for convention for ASCII table fields follows that  
Paradox tables.

Field type: Specifies the data type BDE is to use in  
translating the data value for each field and tells Delphi  
what type of virtual field to create.

Use the setting For values of type

```
-----  
CHAR          Character  
FLOAT         64-bit floating point  
NUMBER        16-bit integer  
BOOL          Boolean (T or F)  
LONGINT       32-bit long integer  
DATE          Date field.  
TIME          Time field.  
TIMESTAMP     Date + Time field.
```

(The actual format for date and time data values  
will be determined by the current setting in  
the BDE  
configuration, Date tab page.)

Data value length: Maximum length of a field's corresponding data  
values value. This setting determines the length of the  
virtual field that Delphi creates to receive  
read from the ASCII file.

Number of decimals: Applicable to FLOAT type fields; specifies the number of digit positions to the right of the decimal place to include in the virtual field definition.

Offset: Offset from the left that represents the starting position for the field in relation to all of the fields that precede it.

For example, the field definition below is for the first field in the ASCII table. It defines a String type data value with a name of "Text", a maximum data value length of three characters (and the field will appear as only three characters long in Delphi data-aware components such as the TDBGrid), no decimal places (a String data value will never have any decimal places), and an offset of zero (because it is the first field and there would not be any preceding fields).

```
Field1=Text,Char,3,00,00
```

Here is an example of a schema file with three fields, the first of String type and the second and third of type date. This schema file would be contained in a file named DATES.SCH to provide file and field definitions for an ASCII data file named DATES.TXT.

```
[DATES]
Filetype=VARYING
Delimiter="
Separator=,
CharSet=ascii
Field1=Text,Char,3,00,00
Field2=First Contact,Date,10,00,03
Field3=Second,Date,10,00,13
```

This schema defines a comma-delimited field where all String type data values can be recognized as being surrounded by the double quotation mark

and where distinct data values are separated by commas (excepting any commas that may appear within the specified delimiter, inside individual String data values). The character field has a length of three characters, no decimal places, and an offset of zero. The first date field has a length of 10, no decimals, and an offset of three. And the second date field has a length of 10, no decimals, and an offset of 13.

For reading ASCII comma-delimited files, the length and offset parameters for the field definitions do not apply to data values in the ASCII files (as is the case for fixed-length field files), but to the virtual fields, defined in the application, into which the values read will be placed. The length parameter will need to reflect the maximum length of the data value for each field -- not counting the delimiting quotation marks or the comma separators. This is most difficult to estimate for String type data values as the actual length of such a data value may vary greatly from row to row in the ASCII data file. The offset parameter for each field will not be the position of the data value in the ASCII file (as is the case for fixed-length field files), but the offset as represented by the cumulative length of all preceding fields (again, the defined fields in memory, not the data values in the ASCII file).

Here is a data file that would correspond to the schema file described above, in a file named DATES.TXT:

```
"A",08/01/1995,08/11/1995
"BB",08/02/1995,08/12/1995
"CCC",08/03/1995,08/13/1995
```

The maximum length of an actual data value in the first field is three ("CCC"). because this is the first field and there are no preceding fields, the offset for this field is zero. The length of this first field

(3) is used as the offset for the second field. The length of the second field, a date value, is 10, reflecting the maximum length of a data value for that field. The accumulated length of the first and second fields are then used as the offset for the third field (3 + 10 = 13) .

It is only when the proper length for the data values in the ASCII file are used and each fields length added to any preceding fields to produce offset values for succeeding fields that this process will correctly read the data. If data is misread because of improper length settings in the schema file, most values will suffer adverse translation effects, such as truncation of character data or numeric values being interpreted as zeros. Data will usually still be displayed, but no error should occur.

However, values that must be in a specific format in order to be translated into the appropriate data type will cause errors if the value read includes characters not valid in a date value. This would include a date data value which, when incorrectly read may contain extraneous characters from other surrounding fields. Such a condition will result in a data translation exception requiring an adjustment of the field length and offset settings in the schema file.

Reference:

7/16/98 4:33:56 PM

Last Modified: 01-SEP-99

## Article #15894: READ & READLN - DIFFERENCES BETWEEN 3.0 AND 4.0+

Technical Notes Database

TN894D.txt      READ & READLN - DIFFERENCES BETWEEN 3.0 AND 4.0+  
Category      :Pascal  
Platform      :All  
Product      :Turbo Pascal 3.0+

### Description:

Q. What are the changes between Read and Readln of 3.0 and 4.0+?

A. The problem which you have encountered is due to a modification of the READ procedure in Turbo Pascal version 4.0+. In version 4.0+, the READ procedure was altered to allow the creation of text file device drivers. Also, the Read procedure handles keyboard input in the same manner as if reading from a disk file. This change has the following effects:

- 1) The carriage return and line feed character combination generated when the Return key is pressed, is no longer cleared from the file input buffer. However, the cursor still advances to the beginning of the next line.
- 2) A Read(NumericVar) call no longer defaults the NumericVar's value. In version 3.0, a simple press of the Return key caused the program to retain the value of the NumericVar and processing continued. In version 4.0, a non white-space character input is required. Simply pressing the Return key will cause the cursor to advance to the next line and continue to wait for input.
- 3) Since the carriage return and line feed combination no longer are handled, they remain in the file input buffer until the next Read call. If the next call is of the type Read(CharacterVar), CharacterVar will return the carriage return character, ASCII 13. A subsequent Read(CharacterVar) call will return the line feed character, ASCII 10. For this reason a Read(Ch); call may appear to have no effect, if it follows Read procedure call.



4) A Read(StringVar) will have a different effect. With the ASCII 13, carriage return and 10, line feed characters as the next to be read in the keyboard buffer, the EOLN function returns true. When reading from a disk file a Read(FileVar, StringVar) returns a null string in StringVar. The same is true with keyboard input. For example the following code will create an infinite loop:

```
Read(AnyVar);  
Repeat  
  Read(StringVar);  
Until StringVar <> '';
```

I suggest that you replace all Read(from standard input) to Readln calls. Many developers have resolved this difference by implementing ReadChar, ReadString, ReadInt, and ReadReal procedures based on the ReadKey function.

Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15898: Determining Record Number In A dBASE Table

Technical Information Database

TI898D.txt Determining Record Number In A dBASE Table

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

dBASE tables employ a fairly static record numbering system. The record number for a given record reflects its physical position in the table file. These record number are not subject to change dependent on the filtering of data or index ordering. For instance, a record that is the first record stored in the .DBF file would be record number 1. It is possible that, through the ordering of an index, this record may be displayed as the last of 100 records. In such a case, its record would remain the same -- one -- and would not be changed to 100 to reflect its position in the index ordered data set. This is in contrast with Paradox tables, where there is a sequence number. The Paradox sequence number is like the dBASE record number except that it is much more fluid and the number for a given record will reflect its position relative to the data set. That is, a record may not always have the same sequence number given filtering of the data set to reduce the number of records or when an index is active that may change the displayed order of the record.

In database applications created with Delphi and the Borland Database Engine (BDE), there is no provision built into the stock data components for extracting or determining the record for dBASE tables. Such an operation is, however, possible by making a call from the application a BDE function.

There are a number of BDE functions that will return information about the current dBASE record, such as the record number. Basically, any function that fills a BDE pRECProps structure would suffice. Such BDE functions include DbtGetRecord, DbtGetNextRecord, and DbtGetPriorRecord. Of course, only the first of these functions really applies to retrieving information about the current record. The other two move the record pointer when invoked, similar in effect to the Next and Prior methods of the TTable or TQuery components.

The pRECProps structure consists of the fields:

iSeqNum: type LongInt; specifies the sequence number of the record (relative to the data set, including filtering and index ordering);

applicable if the table type supports sequence numbers (Paradox only).

iPhyRecNum: type LongInt; specifies the record number for the record;

applicable only when the table type supports physical record numbers (dBASE only).

bRecChanged: type Boolean; not currently used.

bSeqNumChanged: type Boolean; not currently used.

bDeleteFlag: type Boolean; indicates whether the record is deleted; applicable only for table types that support soft-deletes (dBASE only).

One of these BDE functions may be invoked in a Delphi application to fill this structure, from which the physical record number may be retrieved.

Below is an example of the DbtGetRecord function used for this purpose.

```
function RecNo(ATable: TTable): LongInt;
```

```

var
  R: RECProps;
  rslt: DbarResult;
  Error: array [0..255] of Char;
begin
  ATable.UpdateCursorPos;
  rslt := DbGetRecord(ATable.Handle, dbiNoLock, nil,
R);
  if rslt = DBIERR_NONE then
    Result := R.iPhyRecNum
  else begin
    DbGetErrorString(rslt, Error);
    ShowMessage(StrPas(Error));
    Result := -1;
  end;
end;

```

As with invoking any BDE function in a Delphi application, the BDE wrapper units DbTypes, and DbErrs, DbProcs must be included in the Uses section of the unit in which the BDE function will be invoked (the Uses section not shown here). To make this function more transportable, it does not reference the subject TTable component directly, but a reference to the TTable is passed as a parameter. If this function is used in a unit that does not reference the Delphi units DB and DBTables, they must be added so that references to the TTable component will be valid.

The UpdateCursorPos method of the TTable is called to ensure that the record number current in the TTable component is synchronized with that of the underlying table.

BDE functions do not in themselves cause an exception if they fail. Rather, they return a value of BDE type DbarResult that indicates the success or failure of the intended operation. This return value must then be retrieved and evaluated by the front-end application, and the appropriate action taken. A result other than DBIERR\_NONE indicates an

unsuccessful execution of the function. An extra step may be taken (as in the example above) to query the BDE to translate an error code into a readable message with the BDE function `DbiGetErrorString`. In this example, the return value from the invoking of `DbiGetRecord` is stored in the variable `rslt`, and then compared against `DBIERR_NONE` to determine the success of the function call.

If the call to `DbiGetRecord` succeeds, the physical record number from the `iPhyRecNum` field of the `pRECPProps` structure is stored to the variable `Result`, which is the function's return value. To indicate when the function has failed (i.e., the invoking of the `DbiGetRecord` function failed), a value of negative one is returned instead of the record number. This value is purely arbitrary, and any value of a compatible type may be used at the discretion of the programmer.

Reference:

7/16/98 4:33:56 PM

Last Modified: 01-SEP-99

## Article #15900: RUN-TIME ERROR CODE CHANGES FROM VERSION 3.0

Technical Notes Database

TN900D.txt      RUN-TIME ERROR CODE CHANGES FROM VERSION 3.0  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 3.0+

### Description:

- Q. Have Turbo Pascal 3.0 run-time error codes changed in Turbo Pascal 4.0+?
- A. Yes. Error codes have changed; refer to Appendix I in the Reference Guide. The Turbo3 unit contains a version 3.0 compatible IOResult function.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15904: Managing disk volume labels in Delphi

Technical Information Database

TI904D.txt     Managing disk volume labels in Delphi  
Category     :General Programming  
Platform     :All  
Product      :Delphi 1.0

### Description:

This document contains the source code for a unit that is useful for getting, setting, and deleting volume labels from a floppy or hard disk.

The code for getting a volume label uses the Delphi FindFirst function, and the code for setting and deleting volume labels involves calling DOS

interrupt 21h, functions 16h and 13h respectively. Since function 16h isn't supported by Windows, it must be called through DPMI interrupt 31h, function 300h.

```
{ *** BEGIN CODE FOR VOLLABEL UNIT *** }  
unit VolLabel;  
  
interface  
  
uses Classes, SysUtils, WinProcs;  
  
type  
  EInterruptError = class(Exception);  
  EDPMIError = class(EInterruptError);  
  Str11 = String[11];  
  
procedure SetVolumeLabel(NewLabel: Str11; Drive: Char);  
function GetVolumeLabel(Drive: Char): Str11;  
procedure DeleteVolumeLabel(Drv: Char);  
  
implementation  
  
type  
  PRealModeRegs = ^TRealModeRegs;
```

```

TRealModeRegs = record
  case Integer of
    0: (
      EDI, ESI, EBP, EXX, EBX, EDX, ECX, EAX: Longint;
      Flags, ES, DS, FS, GS, IP, CS, SP, SS: Word);
    1: (
      DI, DIH, SI, SIH, BP, BPH, XX, XXH: Word;
      case Integer of
        0: (
          BX, BXH, DX, DXH, CX, CXH, AX, AXH: Word);
        1: (
          BL, BH, BLH, BHH, DL, DH, DLH, DHH,
          CL, CH, CLH, CHH, AL, AH, ALH, AHH: Byte));
end;

```

```

PExtendedFCB = ^TExtendedFCB;
TExtendedFCB = Record
  ExtendedFCBflag : Byte;
  Reserved1       : array[1..5] of Byte;
  Attr           : Byte;
  DriveID        : Byte;
  FileName       : array[1..8] of Char;
  FileExt        : array[1..3] of Char;
  CurrentBlockNum : Word;
  RecordSize     : Word;
  FileSize       : LongInt;
  PackedDate     : Word;
  PackedTime     : Word;
  Reserved2      : array[1..8] of Byte;
  CurrentRecNum  : Byte;
  RandomRecNum   : LongInt;
end;

```

```

procedure RealModeInt(Int: Byte; var Regs: TRealModeRegs);
{ procedure invokes int 31h function 0300h to simulate a real mode }
{ interrupt from protected mode. }
var
  ErrorFlag: Boolean;
begin
  asm
    mov ErrorFlag, 0      { assume success }
    mov ax, 0300h        { function 300h }
    mov bl, Int          { real mode interrupt to execute }
    mov bh, 0            { required }
    mov cx, 0            { stack words to copy, assume zero }
  endasm
end;

```



```
les di, Regs          { es:di = Regs }
int 31h               { DPMI int 31h }
jnc
```

```
End                  { carry flag set on error }
```

```
Error:
    mov ErrorFlag, 1      { return false on error }
```

```
End:
    end;
    if ErrorFlag then
        raise EDPMIError.Create('Failed to execute DPMI interrupt');
    end;
```

```
function DriveLetterToNumber(DriveLet: Char): Byte;
{ function converts a character drive letter into its numerical
equiv. }
begin
    if DriveLet in ['a'..'z'] then
        DriveLet := Chr(Ord(DriveLet) - 32);
    if not (DriveLet in ['A'..'Z']) then
        raise EConvertError.CreateFmt('Cannot convert %s to drive number',
            [DriveLet]);

    Result := Ord(DriveLet) - 64;
end;
```

```
procedure PadVolumeLabel(var Name: Str11);
{ procedure pads Volume Label string with spaces }
var
    i: integer;
begin
    for i := Length(Name) + 1 to 11 do
        Name := Name + ' ';
    end;
```

```
function GetVolumeLabel(Drive: Char): Str11;
{ function returns volume label of a disk }
var
    SR: TSearchRec;
    DriveLetter: Char;
    SearchString: String[7];
    P: Byte;
```

```

begin
  SearchString := Drive + ':\*.*';
  { find vol label }
  if FindFirst(SearchString, faVolumeID, SR) = 0 then begin
    P := Pos('.', SR.Name);
    if P > 0 then begin
      Result := '          ';
      name }
      Move(SR.Name[1], Result[1], P - 1);
      Move(SR.Name[P + 1], Result[9], 3);
    end
    else begin
      Result := SR.Name;
      PadVolumeLabel(Result);
    end;
  end
  else
    Result := '';
  end;
end;

```

```

procedure DeleteVolumeLabel(Drv: Char);
{ procedure deletes volume label from given drive }
var
  CurName: Str11;
  FCB: TExtendedFCB;
  ErrorFlag: WordBool;
begin
  ErrorFlag := False;
  CurName := GetVolumeLabel(Drv);
  FillChar(FCB, SizeOf(FCB), 0);
  with FCB do begin
    ExtendedFCBflag := $FF;
    Attr := faVolumeID;
    DriveID := DriveLetterToNumber(Drv);
    Move(CurName[1], FileName, 8);
    Move(CurName[9], FileExt, 3);
  end;
  asm
    push ds
    mov ax, ss
    mov ds, ax
    lea dx, FCB
    mov ax, 1300h
    Call DOS3Call
    pop ds
  end

```

```
    cmp al, 00h                                { check for success }
    je
```

End

```
Error:                                { set flag on error }
    mov ErrorFlag, 1
```

```
End:
    end;
    if ErrorFlag then
        raise EInterruptError.Create('Failed to delete volume name');
    end;
```

```
procedure SetVolumeLabel(NewLabel: Str11; Drive: Char);
{ procedure sets volume label of a disk. Note that this procedure }
{ deletes the current label before setting the new one. This is }
{ required for the set function to work. }
```

```
var
    Regs: TRealModeRegs;
    FCB: PExtendedFCB;
    Buf: Longint;
begin
    PadVolumeLabel(NewLabel);
    if GetVolumeLabel(Drive) <> '' then          { if has label... }
        DeleteVolumeLabel(Drive);              { delete label }
    Buf := GlobalDOSAlloc(SizeOf(PExtendedFCB)); { allocate real
buffer }
    FCB := Ptr(LoWord(Buf), 0);
    FillChar(FCB^, SizeOf(FCB), 0);            { init FCB with
zeros }
    with FCB^ do begin
        ExtendedFCBflag := $FF;                { required }
        Attr := faVolumeID;                    { Volume ID
attribute }
        DriveID := DriveLetterToNumber(Drive); { Drive number }
        Move(NewLabel[1], FileName, 8);        { set new label }
        Move(NewLabel[9], FileExt, 3);
    end;
    FillChar(Regs, SizeOf(Regs), 0);
    with Regs do begin                          { SEGMENT of FCB }
        ds := HiWord(Buf);                    { offset = zero }
        dx := 0;
```

```
    ax := $1600;                                { function 16h }
end;
RealModeInt($21, Regs);                         { create file }
if (Regs.al <> 0) then                           { check for success }
    raise EInterruptError.Create('Failed to create volume label');
end;

end.
{ *** END CODE FOR VOLLABEL UNIT *** }
```

Reference:

7/16/98 4:33:56 PM

Last Modified: 01-SEP-99

## Article #15906: UPGRADING 3.0 TO 4.0+ PROBLEMS

Technical Notes Database

TN906D.txt UPGRADING 3.0 TO 4.0+ PROBLEMS

Category :Pascal

Platform :All

Product :Turbo Pascal 3.0+

### Description:

Q. What Turbo Pascal 3.0 code will cause the most problems converting to version 4.0+?

A. With our UPGRADE program (see appropriate Appendix in your manual), it's not very difficult to port your code to 4.0+. It depends a lot on the type of programs you write. The passing of parameters on the stack is done much more efficiently now, so changes will have to be made to inline machine code and assembly language. Most of the changes are optional: using new types, breaking your program into modules to take advantage of separate compilation. (The UPGRADE program has a special option to help you "unitize" your program too. It does all the "typing" for you.) Some stricter type-checking is performed in version 4.0+. For example, the Dos unit now defines the often-seen registers record type (AX, BX...); MsDos and Intr now take this type. In this case, you can typecast or change the type identifier and recompile.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15910: How to copy files in Delphi.

Technical Information Database

TI910D.txt     How to copy files in Delphi.

Category     :General Programming

Platform     :All

Product     :Delphi 1.0

Description:

Q: How do I copy a file?

A: Here are three ways:

{This way uses a File stream.}

```
Procedure FileCopy( Const sourcefilename, targetfilename: String );
```

```
Var
```

```
  S, T: TFileStream;
```

```
Begin
```

```
  S := TFileStream.Create( sourcefilename, fmOpenRead );
```

```
  try
```

```
    T := TFileStream.Create( targetfilename,  
                              fmOpenWrite or fmCreate );
```

```
    try
```

```
      T.CopyFrom(S, S.Size ) ;
```

```
    finally
```

```
      T.Free;
```

```
    end;
```

```
  finally
```

```
    S.Free;
```

```
  end;
```

```
End;
```

{This way uses memory blocks for read/write.}

```
procedure FileCopy(const FromFile, ToFile: string);
```

```
var
```

```
  FromF, ToF: file;
```

```
  NumRead, NumWritten: Word;
```

```
  Buf: array[1..2048] of Char;
```

```
begin
```

```
  AssignFile(FromF, FromFile);
```

```
  Reset(FromF, 1);                             { Record size = 1 }
```

```

AssignFile(ToF, ToFile);      { Open output file }
Rewrite(ToF, 1);             { Record size = 1 }
repeat
  BlockRead(FromF, Buf, SizeOf(Buf), NumRead);
  BlockWrite(ToF, Buf, NumRead, NumWritten);
until (NumRead = 0) or (NumWritten <> NumRead);
CloseFile(FromF);
CloseFile(ToF);
end;

```

{This one uses LZCopy, which USES LZExpand.}

```

procedure CopyFile(FromFile, ToFileName: string);

```

```

var

```

```

  FromFile, ToFile: File;

```

```

begin

```

```

  AssignFile(FromFile, FromFileName); { Assign FromFile to
  FromFileName }

```

```

  AssignFile(ToFile, ToFileName);    { Assign ToFile to ToFileName }

```

```

  Reset(FromFile);                  { Open file for input }

```

```

  try

```

```

    Rewrite(ToFile);                { Create file for output }

```

```

    try

```

```

      { copy the file and if a negative value is returned }

```

```

      { raise an exception }

```

```

      if LZCopy(TFileRec(FromFile).Handle, TFileRec(ToFile).Handle) <

```

```

0

```

```

        then

```

```

          raise EInOutError.Create('Error using LZCopy')

```

```

        finally

```

```

          CloseFile(ToFile); { Close ToFile }

```

```

        end;

```

```

    finally

```

```

      CloseFile(FromFile); { Close FromFile }

```

```

    end;

```

```

end;

```

Reference:

7/16/98 4:33:56 PM





## Article #15912: COMPILING VERSION 2.0 CODE WITH THE VERSION 4.0+ C

Technical Notes Database

TN912D.txt      COMPILING VERSION 2.0 CODE WITH THE VERSION 4.0+ C  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 2.0+

### Description:

- Q. Why do I get a stack overflow when compiling code with Turbo Pascal 4.0+ but not with version 2.0?
- A. Version 2.0 does not indicate a stack overflow error. Version 2.0 has a 64k stack size whereas version 4.0+ has a default stack size of 16K. Use the {\$M+} directive to increase the stack size.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15916: How to terminate all running applications

Technical Information Database

TI916D.txt     How to terminate all running applications

Category     :Windows API

Platform     :All

Product      :Delphi 1.0

Description:

Q: How do I terminate all running tasks?

A: Below is some code that will help if you want to terminate ALL tasks,  
no questions asked.

A word of caution, before you run this for the first time, make sure that you save it and anything else that may have some pending data.

```
procedure TForm1.ButtonKillAllClick(Sender: TObject);
var
  pTask      : PTaskEntry;
  Task       : Bool;
  ThisTask   : THANDLE;
begin
  GetMem (pTask, SizeOf (TTaskEntry));
  pTask^.dwSize := SizeOf (TTaskEntry);

  Task := TaskFirst (pTask);
  while Task do
  begin
    if pTask^.hInst = hInstance then
      ThisTask := pTask^.hTask
    else
      TerminateApp (pTask^.hTask, NO_UAE_BOX);
    Task := TaskNext (pTask);
  end;
  TerminateApp (ThisTask, NO_UAE_BOX);
end;
```

Reference:

7/16/98 4:33:57 PM

Last Modified: 01-SEP-99

## Article #15921: How to check to see if a drive is ready.

Technical Information Database

TI921D.txt     How to check to see if a drive is ready.

Category     :General Programming

Platform     :All

Product     :Delphi   All

### Description:

Q: How can I check to see if there is a disk in the "A" drive without an error message box telling you that it is not ready?

A: The following function accepts a drive letter as a parameter, and it will return a boolean value that indicates whether or not there is a disk in the drive.

```
function DiskInDrive(Drive: Char): Boolean;
var
  ErrorMode: word;
begin
  { make it upper case }
  if Drive in ['a'..'z'] then Dec(Drive, $20);
  { make sure it's a letter }
  if not (Drive in ['A'..'Z']) then
    raise EConvertError.Create('Not a valid drive ID');
  { turn off critical errors }
  ErrorMode := SetErrorMode(SEM_FailCriticalErrors);
  try
    { drive 1 = a, 2 = b, 3 = c, etc. }
    if DiskSize(Ord(Drive) - $40) = -1 then
      Result := False
    else
      Result := True;
  finally
    { restore old error mode }
    SetErrorMode(ErrorMode);
  end;
end;
```

Reference:

7/16/98 4:33:57 PM

Last Modified: 01-SEP-99

## Article #15926: How to do pointer arithmetic in Delphi.

Technical Information Database

TI926D.txt     How to do pointer arithmetic in Delphi.  
Category     :General Programming  
Platform     :All  
Product      :Delphi 1.0

### Description:

Q: How do I do pointer arithmetic in Delphi?

A: First a brief explanation of pointer arithmetic. When you are dealing with dynamic memory locations and all you have is a pointer to where it all begins, you want to have the ability to traverse that line of memory to be able to perform whatever functions you have in mind for that data. This can be accomplished by changing the place in memory where the pointer points. This is called pointer arithmetic.

The main idea that must be kept in mind when doing your pointer arithmetic is that you must increment the pointer's value by the correct amount. (The correct amount is determined by the size of the object receiving the pointer. e.g. char = 1 byte; integer = 2 bytes; double = 8 bytes; etc.) The Inc() and Dec() functions will alter the amount by the correct amount. (The compiler knows what the correct size is.)

For an example of the practical application of pointer arithmetic, download the BreakApart() TI2905.

If you are doing dynamic memory allocation, it is done like this:

```
uses WinCRT;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  MyArray: array[0..30] of char;
```

```
  b: ^char;
```

```
  i: integer;
```

```
begin
```

```
  StrCopy(MyArray, 'Lloyd is the greatest!'); {get something to point
```

```

to}
  b :=
MyArray; { assign the pointer to the memory location }
  for i := StrLen(MyArray) downto 0 do
  begin
    write(b^); { write out the char at the current pointer
location. }
    inc(b); { point to the next byte in memory }
  end;
end;

```

The following code demonstrates that the Inc() and Dec() functions will increment or decrement accordingly by size of the type the pointer points to:

```

var
  P1, P2 : ^LongInt;
  L : LongInt;
begin
  P1 :=
L; { assign both pointers to the same place }
  P2 :=
L;
  Inc(P2); { Increment one }

{ Here we get the difference between the offset values of the
two pointers. Since we originally pointed to the same place in
memory, the result will tell us how much of a change occurred
when we called Inc(). }

  L := Ofs(P2^) - Ofs(P1^); { L = 4; i.e. sizeof(longInt) }
end;

```

You can change the type to which P1 and P2 point to something other than a longint to see that the change is always the correct value (SizeOf (P1^)).

Reference:

7/16/98 4:33:57 PM





## Article #15928: DEBUGGER - DISPLAYING FUNCTIONS IN THE WATCH WINDO

Technical Notes Database

TN928D.txt      DEBUGGER - DISPLAYING FUNCTIONS IN THE WATCH WINDO  
Category      :Pascal  
Platform      :All  
Product      :Turbo Pascal 5.0 5.5

### Description:

Q. What built-in functions can you display in the watch window?

A. In the Turbo Pascal User's Guide in chapter 6 there is a table 6.3 that

lists all the built-in functions that you can place in the watch window.

They are:

Abs, Addr, Chr, Cseg, Dseg, Hi, IOResult, Length,  
Lo, Maxavail, Memavail, Odd, Ofs, Ord, Pred, Ptr,  
Round, Seg, Sizeof, Sptr, Sseg Succ, Swap, and  
Trunc.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15932: How to do bit-wise manipulation.

Technical Information Database

TI932D.txt     How to do bit-wise manipulation.

Category     :General Programming

Platform     :All

Product     :Delphi   All

Description:

Q:   How do I do bit-wise manipulation?

A:

```
{*****  
TheBit parameter is counted from 0..31  
*****}
```

```
unit Bitwise;
```

```
interface
```

```
function IsBitSet(const val: longint; const TheBit: byte): boolean;  
function BitOn(const val: longint; const TheBit: byte): LongInt;  
function BitOff(const val: longint; const TheBit: byte): LongInt;  
function BitToggle(const val: longint; const TheBit: byte): LongInt;
```

```
implementation
```

```
function IsBitSet(const val: longint; const TheBit: byte): boolean;  
begin  
  result := (val and (1 shl TheBit)) <> 0;  
end;
```

```
function BitOn(const val: longint; const TheBit: byte): LongInt;  
begin  
  result := val or (1 shl TheBit);  
end;
```

```
function BitOff(const val: longint; const TheBit: byte): LongInt;  
begin
```

```
    result := val and ((1 shl TheBit) xor $FFFFFFFF);  
end;
```

```
function BitToggle(const val: longint; const TheBit: byte): LongInt;  
begin
```

```
    result := val xor (1 shl TheBit);  
end;
```

```
end.
```

Reference:

7/16/98 4:33:57 PM

Last Modified: 01-SEP-99

## Article #15934: DEBUGGER - POINTERS CORRUPTING MEMORY?

Technical Notes Database

TN934D.txt      DEBUGGER - POINTERS CORRUPTING MEMORY?

Category      :Pascal

Platform      :All

Product       :Turbo Pascal    5.0 5.5

### Description:

Q. How can I tell if a pointer could be corrupting my code or data segments?

A. Normally, pointers point to areas located in Turbo Pascal's heap.

You can

determine if a pointer is pointing outside of the heap by comparing its

segment value to that of HEAPORG (a predefined pointer establishing the

beginning of normal heap space). This comparison is best done by placing a

Watch on both HEAPORG and any pointers that you suspicious of. If your

pointers have segment values less than that of HEAPORG when an assignment

is made to them then memory corruption is likely. To correct the situation,

ensure that all pointers are initialized with either the NEW or GETMEM

procedures.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15938: Basic Delphi DLL template

Technical Information Database

TI938D.txt     Basic Delphi DLL template  
Category     :General Programming  
Platform     :All  
Product      :Delphi 1.0

Description:  
DLL sample

Without units

First the DLL "framework" that you wanted, save as DLLFRAME.DPR:

```
{-----DLLFRAME.DPR-----}  
library Dllframe;  
  
uses WinTypes;  
  
function GetString : string ; export ;  
begin  
    Result := 'Hello from the DLL!' ;  
end;  
  
exports  
    GetString;  
  
begin  
end.  
{-----}
```

Now here's the calling program, save it as DLLCALL.DPR:

```
{-----DLLCALL.DPR-----}  
program Dllcall;  
  
uses  
    Dialogs;
```

```
{ $R *.RES }
```

```
function GetString : string ; far ; external 'DLLFRAME' ;
```

```
begin
```

```
  MessageDlg( GetString, mtInformation, [ mbOK ], 0 ) ;
```

```
end.
```

```
With units
```

```
Here's the calling program, save it as DLLCALL.DPR:
```

```
{-----DLLCALL.DPR-----}  
program Dllcall;
```

```
uses
```

```
  Dialogs;
```

```
{ $R *.RES }
```

```
function GetString : string ; far ; external 'MyDLL' ;
```

```
begin
```

```
  MessageDlg( GetString, mtInformation, [ mbOK ], 0 ) ;
```

```
end.
```

```
{-----}
```

```
The DLL "framework" that you wanted, save as DLLFRAME.DPR:
```

```
{-----DLLFRAME.DPR-----}  
library Dllframe;
```

```
uses DLLUnit;
```

```
exports
```

```
  GetString;
```

```
begin
```

```
end.
```

```
{-----}
```

```
The unit we will save as dllunit.pas:
```

```
{-----dllunit.pas-----}
```

```
unit DLLUnit;
interface

uses WinTypes;

function GetString: string; export;

implementation

function GetString: string;
begin
  GetString := 'Hello from the DLL!' ;
end ;

begin
end.
```

Reference:

7/16/98 4:33:57 PM

Last Modified: 01-SEP-99

## Article #15940: INVALID LINKER VERSION ERROR

Technical Notes Database

TN940D.txt      INVALID LINKER VERSION ERROR

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.0 5.5

### Description:

- Q. What does it mean when I get an "Invalid linker version" error using TD 1.0 to debug a program compiled with TP 5.5?
- A. You are using the wrong version of Turbo Debugger. You must have version 1.5 to successfully debug an object oriented Turbo Pascal program.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99



## Article #15944: Form display with different screen resolutions.

Technical Information Database

TI944D.txt Form display with different screen resolutions.

Category :VCL

Platform :All

Product :Delphi All

### Description:

When designing forms, it is sometimes helpful to write the code so that the screen and all of its objects are displayed at the same size no matter what the screen resolution is. Here is some code to show how that is done:

### implementation

const

```
ScreenWidth: LongInt = 800; {I designed my form in 800x600 mode.}
```

```
ScreenHeight: LongInt = 600;
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
  scaled := true;
```

```
  if (screen.width <> ScreenWidth) then
```

```
  begin
```

```
    height := longint(height) * longint(screen.height) div
```

```
ScreenHeight;
```

```
    width := longint(width) * longint(screen.width) div ScreenWidth;
```

```
    scaleBy(screen.width, ScreenWidth);
```

```
  end;
```

```
end;
```

Then, you will want to have something that checks to see that the font sizes are OK. You can iterate over each child control's font to adjust its size as necessary. This can be done as follows:

```
type
```

```
TFooClass = class(TControl); { needed to get at protected }  
           { font property }
```

```

var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do
    TFooClass(Controls[i]).Font.Size :=
      (NewFormWidth div OldFormWidth) *
      TFooClass(Controls[i]).Font.Size;
end;

```

Note: The following are issues to bear in mind when scaling Delphi applications (forms) on different screen resolutions:

- \* Decide early on in the form design stage whether you're going to allow the form to be scaled or not. The advantage of not scaling is that nothing changes at runtime. The disadvantage of not scaling is that nothing changes at runtime (your form may be far too small or too large to read on some systems if it is not scaled).

- \* If you're NOT going to scale the form, set Scaled to False.

- \* Otherwise, set the Form's Scaled property to True.

- \* Set AutoScroll to False. AutoScroll = True means 'don't change the form's frame size at runtime' which doesn't look good when the form's contents do change size.

- \* Set the form's font to a scaleable TrueType font, like Arial. MS San Serif is an ok alternate, but remember that it is still a bitmapped font. Only Arial will give you a font within a pixel of the desired height. NOTE: If the font used in an application is not installed on the target computer, then Windows will select an alternative font within the same font family to use instead. This font may not match the same size of the original font and may cause problems.

- \* Set the form's Position property to something other than poDesigned. poDesigned leaves the form where you left it at design time, which for me always winds up way off to the left on my 1280x1024 screen - and completely off the 640x480 screen.

- \* Don't crowd controls on the form - leave at least 4 pixels

between controls, so that a one pixel change in border locations (due to scaling) won't show up as ugly overlapping controls.

- \* For single line labels that are `alLeft` or `alRight` aligned, set `AutoSize` to `True`. Otherwise, set `AutoSize` to `False`.

- \* Make sure there is enough blank space in a label component to allow for font width changes - a blank space that is 25% of the length of the current string display length is a little too much, but safe. (You'll need at least 30% expansion space for string labels if you plan to translate your app into other languages) If `AutoSize` is `False`, make sure you actually set the label width appropriately. If `AutoSize` is `True`, make sure there is enough room for the label to grow on its own.

- \* In multi-line, word-wrapped labels, leave at least one line of blank space at the bottom. You'll need this to catch the overflow when the text wraps differently when the font width changes with scaling. Don't assume that because you're using large fonts, you don't have to allow for text overflow - somebody else's large fonts may be larger than yours!

- \* Be careful about opening a project in the IDE at different resolutions. The form's `PixelsPerInch` property will be modified as soon as the form is opened, and will be saved to the DFM if you save the project. It's best to test the app by running it standalone, and edit the form at only one resolution. Editing at varying resolutions and font sizes invites component drift and sizing problems.

- \* Speaking of component drift, don't rescale a form multiple times, at design time or a runtime. Each rescaling introduces roundoff errors which accumulate very quickly since coordinates are strictly integral. As fractional amounts are truncated off control's origins and sizes with each successive rescaling, the controls will appear to creep northwest and get smaller. If you want to allow your users to rescale the form any number of times, start with a freshly loaded/created form before each scaling, so that scaling errors do not accumulate.

- \* Don't change the `PixelsPerInch` property of the form, period.

- \* In general, it is not necessary to design forms at any particular resolution, but it is crucial that you review their

appearance at 640x480 with small fonts and large, and at a high-resolution with small fonts and large before releasing your app. This should be part of your regular system compatibility testing checklist.

\* Pay close attention to any components that are essentially single-line TMemos - things like TDBLookupCombo. The Windows multi-line edit control always shows only whole lines of text - if the control is too short for its font, a TMemo will show nothing at all (a TEdit will show clipped text). For such components, it's better to make them a few pixels too large than to be one pixel too small and show not text at all.

\* Keep in mind that all scaling is proportional to the difference in the font height between runtime and design time, NOT the pixel resolution or screen size. Remember also that the origins of your controls will be changed when the form is scaled - you can't very well make components bigger without also moving them over a bit.

Reference:

7/16/98 4:33:57 PM

Last Modified: 01-SEP-99

## Article #15946: DEBUGGER - STEPPING INTO A PROCEDURE

Technical Notes Database

TN946D.txt      DEBUGGER - STEPPING INTO A PROCEDURE

Category      :Pascal

Platform      :All

Product       :Turbo Pascal 5.0 5.5

### Description:

- Q. I am using the integrated source level debugger, and I cannot step into a procedure, using the F7 key. What is wrong?
- A. First, make sure that your program and units are compiled with debug and local symbol information, as well as integrated debugging turned on. Also, the program itself may be corrupting the debug information. You should step through the code in a smaller test program and try to narrow down the problem.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15950: How to keep the app iconized.

Technical Information Database

TI950D.txt     How to keep the app iconized.

Category     :VCL

Platform     :All

Product      :Delphi 1.0

Description:

iconized apps

Q: How do I keep the form in icon form when I run it?

A:

1. You must set WindowState to wsMinimized in the form's properties.
2. In the private section of the form object's declaration, put:

```
PROCEDURE WMQueryOpen(VAR Msg : TWMQueryOpen); message  
WM_QUERYOPEN;
```

3. In the implementation section, put this method:

```
PROCEDURE TForm1.WMQueryOpen(VAR Msg : TWMQueryOpen);  
begin  
    Msg.Result := 0;  
end;
```

That's it! The form will always remain iconic.

Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99

## Article #15952: IS THERE A DEBUGGER AVAILABLE FOR TP 5.0+?

Technical Notes Database

TN952D.txt IS THERE A DEBUGGER AVAILABLE FOR TP 5.0+?

Category :Pascal

Platform :All

Product :Turbo Pascal 5.0 5.5

Description:

Q. Is there a debugger available for use with TP 5.0+?

A. There is a built-in debugger in version 5.0+. In addition, you can use the Turbo Debugger on .EXE files generated by Turbo Pascal 5.0+. Finally, you can use any debugger that can process .MAP files (see the Options/Linker menu).

Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15956: How to use a custom cursor.

Technical Information Database

TI956D.txt      How to use a custom cursor.

Category        :VCL

Platform        :All

Product         :Delphi    All

Description:

Q: How do I use one of the cursor files in the c:\delphi\images\cursors?

A: Use the image editor to load the cursor into a RES file.  
The following example assumes that you saved the cursor in the RES file  
as "cursor\_1", and you save the RES file as MYFILE.RES.

```
(*** BEGIN CODE ***)
{$R c:\programs\delphi\MyFile.res}    { This is your RES file }

const PutTheCursorHere_Dude = 1;      { arbitrary positive number }

procedure stuff;
begin
  screen.cursors[PutTheCursorHere_Dude] := LoadCursor(hInstance,
                                                         PChar
('cursor_1'));
  screen.cursor := PutTheCursorHere_Dude;
end;
```

Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99



## Article #15958: ACCESSING MORE THAN 64K OF DATA

Technical Notes Database

TN958D.txt      ACCESSING MORE THAN 64K OF DATA

Category      :Pascal

Platform      :All

Product      :Turbo Pascal 4.0+

### Description:

Q. I need my program to work with more than 64K of data at a time.

How

can I use large data structures in my Turbo Pascal programs?

A. You would use heap space for your larger data structures. Instead of

declaring variables to be large arrays, records, or arrays of records,

declare these variables to be pointers to your data structures and reserve

space for them with procedures NEW or GETMEM. The space reserved for

this data will not reside in the normal data segment but rather in heap

space where each data structure may occupy up to 64K.

If you need even more memory, use typed files to store your data.

Storage is limited only by the capacity of your drive system. The SEEK procedure can be used to index into the file as though it were a large array out on disk. This technique is often the only option when memory consumption must be kept to a minimum. However,

accessing

data on disk is slower than accessing data in RAM memory.

### Reference:

7/16/98 4:35:37 PM

Last Modified: 01-SEP-99

## Article #15962: How to use a popup menu with a VBX.

Technical Information Database

TI962D.txt - How to use a popup menu with a VBX.

Category :Delphi 1.x  
Platform :All Windows  
Product :

### Description:

Q: I want to be able to right click on my VBX and have a popup menu display. When I use a popup menu for the form, it shows no matter where I right click. I want to just have it popup for right clicks on the vbx.

How do I trap for that?

A: Here it is:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
  Shift: TShiftState; X, Y: Integer);  
begin  
  if button = mbRight then  
    with (Sender AS TControl) do  
      with ClientToScreen(Point(X,Y)) do  
        begin  
          PopupMenu1.PopupComponent := TComponent(Sender);  
          PopupMenu1.Popup(X,Y);  
        end;  
      end;  
end;
```

Note: The form's PopupMenu property must be empty, or it will popup from everywhere. If you want the form to be the only place showing the popup, place this method on the form's OnMouseDown event. If you want the VBX to be the only place, then place it on the VBX's OnMouseDown event, etc.

Reference:

3/31/99 10:07:50 AM

Last Modified: 01-SEP-99

## Article #15964: NORMALIZING POINTERS

Technical Notes Database

TN964D.txt      NORMALIZING POINTERS  
Category      :Pascal  
Platform       :All  
Product        :Turbo Pascal 4.0+

### Description:

Q. How do I normalize a pointer in Turbo Pascal?

A. When you allocate a pointer, it is already normalized. So, if you're SURE that you won't be making the offset greater than 64K, you can skip the normalizing step.

```
PROCEDURE Normalize(VAR P);  
VAR pt : Pointer ABSOLUTE P;  
BEGIN  
  Pt := Ptr(seg(pt^) + (ofs(pt^) DIV 16), ofs(pt^) MOD 16);  
END;
```

You CAN perform pointer arithmetic, if you're desperate to do "C things" in Pascal. Just normalize the pointer (maximize segment, minimize offset), typecast it to a LongInt, and do your math. E.g.

```
Normalize(recptr);  
Inc(LongInt(recptr), SizeOf(rec));
```

where recptr is a pointer to a record, and rec is the record variable.

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #15968: How to set a max and min form size.

Technical Information Database

TI968D.txt     How to set a max and min form size.

Category     :VCL

Platform     :All

Product      :Delphi   All

### Description:

When you want to control how much your users can resize your form, you can control that by setting the MinMax values. (If you use the resize method to limit the size, it will work, but it won't look quite as good.)

Note: To make it so that the user cannot change the form's size at all, make the min and max sizes the same values.

This is an example of how to declare and use the `wm_GetMinMaxInfo` windows message in your applications.

```
unit MinMax;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
  private
```

```
    { Private declarations }
```

```
    procedure WMGetMinMaxInfo(var MSG: Tmessage); message
```

```
WM_GetMinMaxInfo;
```

```
  public
```

```
    { Public declarations }
```

```
  end;
```

```
var
```

```
  Form1: TForm1;
```

implementation

{ \$R \*.DFM }

```
procedure TForm1.WMGetMinMaxInfo(var MSG: Tmessage);
Begin
  inherited;
  with PMinMaxInfo(MSG.lparam)^ do
  begin
    with ptMinTrackSize do
    begin
      X := 300;
      Y := 150;
    end;
    with ptMaxTrackSize do
    begin
      X := 350;
      Y := 250;
    end;
  end;
end;

end;
```

Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99

## Article #15970: DATA SEGMENT TOO LARGE ERROR

Technical Notes Database

TN970D.txt DATA SEGMENT TOO LARGE ERROR

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

Q. What should I do when I get "Data segment too large" compiler error?

A. Allocate your data on the heap by using pointers.

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #15974: How to get the windows and DOS versions.

Technical Information Database

TI974D.txt     How to get the windows and DOS versions.

Category     :Windows API

Platform     :All

Product     :Delphi 1.0

Description:

Q: How can I get the Windows or DOS version numbers?

A: The API call GetVersion will do it, but the information is encrypted into a longint. Here is how to get and decrypt the information:

Type

```
TGetVer = record
```

```
  WinVer,
```

```
  WinRev,
```

```
  DosRev,
```

```
  DosVer: Byte;
```

```
end;
```

const

```
  VerStr = '%d.%d';
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  AllVersions: TGetVer;
```

```
begin
```

```
  AllVersions := TGetVer(GetVersion);
```

```
  Edit1.Text := Format(VerStr, [AllVersions.WinVer, AllVersions.  
WinRev]);
```

```
  Edit2.Text := Format(VerStr, [AllVersions.DOSVer, AllVersions.  
DOSRev]);
```

```
end;
```

Note1: The values that windows displays for the versions and the values that it returns through its API call are not always the same. e.g. The



workgroup version displays as 3.10 rather than 3.11.

Note2: Win32 applications should call GetVersionEx rather than GetVersion.

Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99

## Article #15976: FILE MODE LOCKING/UNLOCKING ON A NETWORK

Technical Notes Database

TN976D.txt FILE MODE LOCKING/UNLOCKING ON A NETWORK

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

Q. How can I access my data on a network when I have more than one user?

A. The routine in the following unit will help.

Using the fm\* constants you can set the FILEMODE byte to match the amount of sharing that you wish to take place.

```
example: FileMode := fmReadWr+fmDenyNone;
```

Using the Lock function you can lock a record or a series of records.

Lock returns True if the request to lock the record was successful. If

Lock returns False it can be assumed that the record was locked by someone

else and not available for editing at this time.

Locking Needs to take place anytime that a user will be allowed to change

the data. If your program in in a view mode you can safely leave the

record unlocked.

Unit NETio;

Interface

Const {File Open Mode Constants}

	{	I	SSS	R	AAA	I= Inheritance Flag	}		
fmRead	=	\$00;	{	_	___	0	000	S= Sharing Mode Field	}
fmWrite	=	\$01;	{	_	___	0	001	R= Reserved (must Be 0)	}
fmReadWr	=	\$02;	{	_	___	0	010	A= Access Field	}
fmCompat	=	\$00	{	_	000	0	___	}	
fmDenyReadWr	=	\$10	{	_	001	0	___	}	

```

fmDenyWrite   = $20 { _ 010 0 ____ }
fmDenyRead    = $30 { _ 011 0 ____ }
fmDenyNone    = $40 { _ 100 0 ____ }
fmNotInher    = $80 { 1 ____ 0 ____ }

```

Const

```

Lock    = 0;
UnLock  = 1;

```

```
Function Lock( Var FilVar; RecordNumber: longint;
```

```
Function LockRegion( Var FilVar; RecordNumber,NumRec: longint;
                    Lock_Unlock:byte ) : Word;
```

Implementation

```
Function LockRegion( Var FilVar; RecordNumber,NumRec: longint;
                    Lock_Unlock:byte ) : Word;
```

Var

```

R : registers;
ByteOffset : Longint;

```

begin

```
With FileRec(FilVar) do
```

```
Begin
```

```
ByteOffset := Recordnumber * RecSize;
```

```
NumRec      := NumRec          * RecSize;
```

```
With R do
```

```
Begin
```

```
AH := $5C;
```

```
AL := Lock_Unlock;
```

```
BX := Handle;
```

```
CX := (ByteOffset and $FFFF0000) shr 16;
```

```
DX := (ByteOffset and $0000FFFF);
```

```
SI := (NumRec      and $FFFF0000) shr 16;
```

```
DI := (NumRec      and $0000FFFF);
```

```
End;
```

```
Intr( $21, R );
```

```
if (R.Flags and 1) = 1 then
```

```
LockRegion := R.AX
```

```
Else
```

```
LockRegion := 0;
```

```
End;
```

end;

```
Function Locklock( Var FilVar; RecordNumber: longint;  
                  Lock_Unlock:byte ) : Word;  
Begin  
  Lock := LockRegion( FilVar, RecordNumber, 1, Lock_Unlock );  
End;  
END.
```

Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #15980: How to tell what kind of drive is used.

Technical Information Database

TI980D.txt    How to tell what kind of drive is used.  
Category    :Windows API  
Platform    :All  
Product     :Delphi All

### Description:

When dealing with multiple drives, it is helpful to know whether a drive is associated with a is attached to a letter (A, B, C, etc), and what its type is. This code uses the API GetDriveType function to do that.

```
function ShowDriveType(DriveLetter: char): string;
var
  i: word;
begin
  if DriveLetter in ['A'..'Z'] then {Make it lower case.}
    DriveLetter := chr(ord(DriveLetter) + $20);
  i := GetDriveType(ord(DriveLetter) - ord('a'));
  case i of
    DRIVE_REMOVABLE: result := 'floppy';
    DRIVE_FIXED: result := 'hard disk';
    DRIVE_REMOTE: result := 'network drive';
    else result := 'does not exist';
  end;
end;
```

### Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99

## Article #15982: VARIANT RECORD LONGINT TO BYTE AND WORD

Technical Notes Database

TN982D.txt     VARIANT RECORD LONGINT TO BYTE AND WORD  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal 4.0+

### Description:

Q. How can I quickly change a Longint into two Words or four Bytes?

A. With a variant record you can map a group of variables on the same memory and be able to view the variables in different ways. Here is an example:

```
Program Variant;
  Type
    Forms = Record
      Case Integer Of
        0:(L:LongInt);
        1:(W1,W2: Word);
        2:(B1,B2,B3,B4: Byte);
      End;
  Var
    Many:Forms;
  Begin
    With Many do
      Begin
        Many.L:=123456789;
        Writeln(' LongInt Value : ',L);
        Writeln(' Word Value      : ',W1,' ',W2);
        Writeln(' Byte Value       : ',B1,' ',B2,' ',B3,' ',B4);
      End
    End.
End.
```

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #15986: How to determine the current record number.

Technical Information Database

TI986D.txt     How to determine the current record number.

Category     :Database Programming

Platform     :All

Product      :Delphi   All

Description:

Q: "How can I determine the current record number for a dataset?"

A: If the dataset is based upon a Paradox or dBASE table then the record number can be determined with a couple of calls to the BDE (as shown below). The BDE doesn't support record numbering for datasets based upon SQL tables, so if your server supports record numbering you will need to refer to its documentation.

The following function is given as part of a whole unit and takes as its parameter any component derived from TDataset (i.e. TTable, TQuery, TStoredProc) and returns the current record number (greater than zero) if it is a Paradox or dBASE table. Otherwise, the function returns zero.

NOTE: for dBASE tables the record number returned is always the physical record number. So, if your dataset is a TQuery or you have a range set on your dataset then the number returned won't necessarily be relative to the dataset being viewed, rather it will be based on the record's physical position in the underlying dBASE table.

```
uses
  DB, DBTables, DbiProcs, DbiTypes, DbiErrs;

function GetRecordNumber(Dataset: TDataset): Longint;
var
  CursorProps: CurProps;
  RecordProps: RECProps;
begin
```

```

{ Return 0 if dataset is not Paradox or dBASE }
Result := 0;
with Dataset do
begin
  { Is the dataset active? }
  if State = dsInactive then
    raise EDatabaseError.Create('Cannot perform this operation '+
                                'on a closed dataset');

  { We need to make this call to grab the cursor's iSeqNums }
  Check(DbiGetCursorProps(Handle, CursorProps));

  { Synchronize the BDE cursor with the Dataset's cursor }
  UpdateCursorPos;

  { Fill RecordProps with the current record's properties }
  Check(DbiGetRecord(Handle, dbiNOLOCK, nil,
RecordProps));

  { What kind of dataset are we looking at? }
  case CursorProps.iSeqNums of
    0: Result := RecordProps.iPhyRecNum; { dBASE }
    1: Result := RecordProps.iSeqNum;    { Paradox }
  end;
end;
end;
end.

```

Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99



## Article #15988: POST-DECLARATION TYPECASTING

Technical Notes Database

TN988D.txt POST-DECLARATION TYPECASTING

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

Description:

Q. How can I change the type of a variable to another type?

A. The method is called type casting. Here is the syntax:

```
Type
```

```
  RecType = Record
```

```
    W : Word;
```

```
  End;
```

```
Var
```

```
  A : Array[1..2] of Byte;
```

```
  R : RecType;
```

```
Begin
```

```
  ...
```

```
  R := RecType( A ); { Convert the array into a record
```

```
type }
```

```
  ...
```

```
End.
```

Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #15992: How to automate logon for Paradox tables

Technical Information Database

TI992D.txt     How to automate logon for Paradox tables

Category     :Database Programming

Platform     :All

Product      :Delphi   All

Description:

Password automation

Q: I have a paradox table that uses a password. How do I make it so that the form that uses the table comes up without prompting the user for the password?

A: The table component's ACTIVE property must be set to FALSE (If it is active before you have added the password, you will be prompted). Then, put this code in the handler for the form's OnCreate event:

```
Session.AddPassword('My secret password');  
Table1.Active := True;
```

Once you close the table, you can remove the password with `RemovePassword('My secret password')`, or you can remove all current passwords with `RemoveAllPasswords`. (Note: This is for Paradox tables only.)

Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99

## Article #15994: STATIC VARIABLES LIKE C

Technical Notes Database

TN994D.txt      STATIC VARIABLES LIKE C  
Category      :Pascal  
Platform      :All  
Product       :Turbo Pascal  4.0+

### Description:

- Q. Is there anything similar to C's static variables in 4.0+?
- A. You can declare private global variables in the implementation part of a unit. Such variables are only visible within that unit. Like other global variables, they retain their values across calls. Typed constant declarations declared within a procedure or function also behave exactly like C's static variables. They are local in scope but since they are allocated in the data segment, they retain their values from call to call.

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #15998: packing a dBASE table

Technical Information Database

TI998D.txt     packing a dBASE table  
Category     :Database Programming  
Platform     :All  
Product      :Delphi All

### Description:

packing a dBASE table

Q: How do I pack a dBASE table?

A: To pack a dBASE table that has been opened with a TTable, use the BDE function DbiPackTable. There are two basic steps to do this:

1. Add the following units to your uses clause:

```
{ For Delphi 1.0: } DBITYPES, DBIPROCS and DBIERRS;  
{ For Delphi 2.0: } BDE;
```

2) Then call the DbiPackTable BDE function as follows:

```
Check(DbiPackTable(Table1.DbHandle, Table1.Handle, Nil, szDBASE,  
TRUE));
```

### Notes:

- \* The table must be opened in exclusive mode.
- \* Use the Check procedure when calling BDE API functions. Check will raise an exception if an error occurs on the BDE call.

### Reference:

7/16/98 4:33:58 PM

Last Modified: 01-SEP-99

## Article #16000: LARGEST GLOBAL DATA STRUCTURE

Technical Notes Database

TN1000D.txt    LARGEST GLOBAL DATA STRUCTURE

Category     :Pascal

Platform     :All

Product      :Turbo Pascal  4.0+

Description:

Q. What is the largest global data structure you can allocate?

A. The maximum size of a single variable that can be defined is  
65,521 bytes.

Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #16004: string manipulation routines

Technical Information Database

TI1004D.txt string manipulation routines

Category :Object Pascal

Platform :All

Product :Delphi All

Description:

Here are some standard string manipulation functions:

{To determine if the character is a digit.}

```
function IsDigit(ch: char): boolean;
```

```
begin
```

```
    Result := ch in ['0'..'9'];
```

```
end;
```

{To determine if the character is an uppercase letter.}

```
function IsUpper(ch: char): boolean;
```

```
begin
```

```
    Result := ch in ['A'..'Z'];
```

```
end;
```

{To determine if the character is an lowercase letter.}

```
function IsLower(ch: char): boolean;
```

```
begin
```

```
    Result := ch in ['a'..'z'];
```

```
end;
```

{Changes a character to an uppercase letter.}

```
function ToUpper(ch: char): char;
```

```
begin
```

```
    Result := chr(ord(ch) and $DF);
```

```
end;
```

{Changes a character to a lowercase letter.}

```
function ToLower(ch: char): char;
```

```
begin
```

```
    Result := chr(ord(ch) or $20);
```

```
end;
```

```

{ Capitalizes first letter of every word in s }
function Proper(const s: string): string;
var
  i: Integer;
  CapitalizeNextLetter: Boolean;
begin
  Result := LowerCase(s);
  CapitalizeNextLetter := True;
  for i := 1 to Length(Result) do
  begin
    if CapitalizeNextLetter and IsLower(Result[i]) then
      Result[i] := ToUpper(Result[i]);
    CapitalizeNextLetter := Result[i] = ' ';
  end;
end;

{ NOTE: The following functions are available in Delphi 2.0,
  but not in Delphi 1.0. }

{Supresses trailing blanks in a string.}
function TrimRight(const s: string): string;
var
  i: integer;
begin
  i := Length(s);
  while (I > 0) and (s[i] <= ' ') do Dec(i);
  Result := Copy(s, 1, i);
end;

{Removes the leading spaces from a string.}
function TrimLeft(const S: string): string;
var
  I, L: Integer;
begin
  L := Length(S);
  I := 1;
  while (I <= L) and (S[I] <= ' ') do Inc(I);
  Result := Copy(S, I, Maxint);
end;

{ Removes leading and trailing whitespace from s);
function Trim(const S: string): string;
var
  I, L: Integer;
begin

```

```
L := Length(S);  
I := 1;  
while (I <= L) and (S[I] <= ' ') do Inc(I);  
if I > L then Result := '' else  
begin  
    while S[L] <= ' ' do Dec(L);  
    Result := Copy(S, I, L - I + 1);  
end;  
end;
```

Reference:

7/16/98 4:33:59 PM

Last Modified: 01-SEP-99



## Article #16006: STRING SIZE LIMIT 255 CHARS?

Technical Notes Database

TN1006D.txt     STRING SIZE LIMIT 255 CHARS?

Category     :Pascal

Platform     :All

Product      :Turbo Pascal  4.0+

Description:

Q. Is the string size limit still 255 characters?

A. Yes. It's just like in 3.0; you can write your own routines to handle greater than 255 character strings. There may also be tools available through third party distributors such as Turbo Power Software.

Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #16010: WinExecAndWait

Technical Information Database

TI1010D.txt WinExecAndWait

Category :Windows API

Platform :All

Product :Delphi 1.0

Description:

Q: How do I execute a program and have my code wait until it is finished?

A:

```
uses Wintypes,WinProcs,Toolhelp,Classes,Forms;
```

```
Function WinExecAndWait(Path : string; Visibility : word) : word;
```

```
var
```

```
    InstanceID : THandle;
```

```
    PathLen : integer;
```

```
begin
```

```
    { inplace conversion of a String to a PChar }
```

```
    PathLen := Length(Path);
```

```
    Move(Path[1],Path[0],PathLen);
```

```
    Path[PathLen] := #0;
```

```
    { Try to run the application }
```

```
    InstanceID := WinExec(
```

```
Path,Visibility);
```

```
    if InstanceID < 32 then { a value less than 32 indicates an Exec  
error }
```

```
        WinExecAndWait := InstanceID
```

```
    else
```

```
    begin
```

```
        Repeat
```

```
            Application.ProcessMessages;
```

```
        until Application.Terminated or (GetModuleUsage(InstanceID) = 0);
```

```
        WinExecAndWait := 32;
```

```
    end;
```

```
end;
```

Reference :

7/16/98 4:33:59 PM

Last Modified: 01-SEP-99

## Article #16015: How to check for app already running.

Technical Information Database

TI1015D.txt How to check for app already running.

Category :Windows API

Platform :All

Product :Delphi 1.0

Description:

Q: How can I write my Delphi program to detect if there is already another copy running and exit if so?

A: Create a unit called PrevInst and add it to your uses clause. Here's the code:

```
unit PrevInst;

interface

uses
  WinTypes, WinProcs, SysUtils;

type
  PHWND = ^HWND;
  function EnumFunc(Wnd:HWND; TargetWindow:PHWND): bool; export;
  procedure GotoPreviousInstance;

implementation

function EnumFunc(Wnd:HWND; TargetWindow:PHWND): bool;
var
  ClassName : array[0..30] of char;
begin
  Result := true;
  if GetWindowWord(Wnd,GWW_HINSTANCE) = hPrevInst then
  begin
    GetClassName(Wnd,ClassName,30);
    if StrIComp(ClassName,'TApplication') = 0 then
    begin
      TargetWindow^ := Wnd;
      Result := false;
    end;
  end;
end;
```

```

    end;
end;
end;

procedure GotoPreviousInstance;
var
    PrevInstWnd : HWND;
begin
    PrevInstWnd := 0;
    EnumWindows(
EnumFunc, longint(
PrevInstWnd));
    if PrevInstWnd <> 0 then
        if IsIconic(PrevInstWnd) then
            ShowWindow(PrevInstWnd, SW_RESTORE)
        else
            BringWindowToTop(PrevInstWnd);
    end;

end.

```

And then make the main block of your \*.DPR file look something like this--

```

begin
    if hPrevInst <> 0 then
        GotoPreviousInstance
    else
        begin
            Application.CreateForm(MyForm, MyForm);
            Application.Run;
        end;
    end;
end.

```

Reference:

7/16/98 4:33:59 PM

Last Modified: 01-SEP-99

## Article #16017: Compatibility with MS-DOS GENERIC MACHINES

Technical Notes Database

TN1017D.txt      Compatibility with MS-DOS GENERIC MACHINES

Category        :Pascal

Platform        :All

Product         :Turbo Pascal 4.0+

### Description:

Q. Can Turbo Pascal run on generic MS-DOS machines?

A. TPC.EXE will run on generic machines when you use the /Q option. The System, Overlay, Dos, and Printer standard units will operate correctly on MS-DOS generic machines. Generated .EXE's are MS-DOS compatible as long as you don't use the special PC units (such as Crt, Graph, and Graph3).

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #16021: How to use a form several times

Technical Information Database

TI1021D.txt    How to use a form several times

Category     :General Programming

Platform     :All

Product     :Delphi   All

Description:

multiple forms

Q: I have a form that is a sort of template. I want to be able to create and show the same form several times (with different data in the fields). How do I use the same form several times?

A: You need to make modeless window by calling create and show for each form instance, like this:

```
with TMyForm.create(self) do show;
```

To demonstrate how to use and control these new forms, here is an example that changes the caption and name of each form that is created. You have access to it through the form's component array. This example uses an about box (named "box") as the other form. Also, there is a variable called "TheFormCount" that keeps track of how many times the form is instantiated.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with TBox.create(self) do begin
    Name := 'AboutBox_' + intToStr(TheFormCount);
    caption := 'About Box # ' + intToStr(TheFormCount);
    Show;
  end;
  inc(TheFormCount);
end;
```

These forms can be found and used by their name by means of the FindComponent method used something like this:

```
with Form1.FindComponent('AboutBox_' + IntToStr(Something)) as TForm
do
    DoSomethingHere;
```

Reference:

7/16/98 4:33:59 PM

Last Modified: 01-SEP-99



## Article #16023: SERIAL COMMUNICATION AUXINOUT UNIT

Technical Notes Database

TN1023D.txt SERIAL COMMUNICATION AUXINOUT UNIT

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

Q. I'm using the AuxInOut unit, and the machine that sends out the data tries to read incoming data. However, the incoming data is garbage. What do I do?

A. To read incoming data, use the Reset procedure before reading. Also, you should read from a buffer rather than directly from the serial port. Furthermore, the AuxInOut is not a good example to use as input buffering is not supported; you should use the TI407.ARC routines for this purpose.

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #16027: Manually Installing Delphi

Technical Information Database

TI1027D.txt - Manually Installing Delphi

Category :Delphi 1.x  
Platform :All Windows  
Product :

### Description:

To manually install Delphi you should first copy the entire run-image directory off of the CD-ROM onto your hard-drive. This is one of the main directories on the delphi CD-ROM. This includes copying all the files from the run-image\windows to your local \windows directory and the \windows\system files into your \windows\system directory, UNLESS you have NT. If you have NT you will want to do as above, but do not overwrite any existing files.

It is important that if you have another Borland DataBase product that you Z-copy the \idapi directory from the run-image ONTO the existing \idapi directory on your computer.

To manually install delphi you first need to add the following lines to your win.ini (this is all dependent on which drive/directory that you copied the run-image to):

```
[IDAPI]
DLLPATH=C:\IDAPI
CONFIGFILE01=C:\IDAPI\IDAPI.CFG
```

```
[Borland Language Drivers]
LDPath=C:\IDAPI\LANGDRV
```

```
[BWCC]
BitmapLibrary=BWCC.DLL
```

```
[Interbase]
RootDirectory=C:\IBLOCAL
```

```
[Paradox Engine]
UserName=PxEEngine
NetNamePath=C:\
RecBufs=64
MaxLocks=64
MaxFiles=64
SwapSize=64
```

```
[DDE Servers]
DBD=C:\DBD\DBD
```

```
[DBD]
WORKDIR=C:\DBD
PRIVDIR=C:\DBD\DBDPRIV
```

You will also want to add the following lines to your autoexec.bat file:

```
rem add the following if it they are not already present
rem you only need share if using windows 3.1
```

```
Share /f:4096 /l:40
path=%path%;c:\iblocal\bin;
```

These are once again dependent on the exact drive/directory that you copied the run-image to.

You will now want to download the file dlpgrp.zip from our download BBS or from the web site. This file will create the Icons and the Delphi Program Group.

Reference:

3/30/99 1:06:06 PM

Last Modified: 01-SEP-99

## Article #16029: ASYNC LOST CHARACTERS BROKEN SERIAL

Technical Notes Database

TN1029D.txt ASYNC LOST CHARACTERS BROKEN SERIAL

Category :Pascal

Platform :All

Product :Turbo Pascal 4.0+

### Description:

Q. The Async routines in the manual do not work. Why?

A. The Asynchronous communications routines in the manual do not provide buffered communication. In other words, if you are not actively reading the serial port you will only get the last character that arrives. For example, if the characters ABCD come across the line and then you read the serial port, you will get back D but not A, B, or C. TI407 provides buffered input from a serial port.

### Reference:

7/16/98 4:35:38 PM

Last Modified: 01-SEP-99

## Article #16033: Printing in the DOS IDE under Windows 95

Technical Information Database

TI1033D.txt - Printing in the DOS IDE under Windows 95

Category :Printing  
Platform :All-32Bit  
Product :

Description:

Printing from the DOS IDE in Windows 95

To print from the Borland Pascal IDE under Windows 95 can require more than just selecting print from the file menu. Follow this step by step procedure if you can print from Windows applications but cannot print from the DOS IDE.

- 1.Open the printers folder under the control panel sub-directory. You can get to the control panel by going from the start button to the settings then click control panel. Inside control panel double click the printers folder.
- 2.Click the icon for the printer you are going to print on.
- 3.On the file menu click properties.
- 4.Click on the details tab and then click on the Capture Printer Port button.
- 5.In the Device list, select the printer port that you want to capture.
- 6.Type the network path to the printer, and then click OK to save the selection.
- 7.In the Print To The Following Port dialog box, select the port you just mapped.

For a handy additional reference on the topic also see the Windows help file under "Capture".

Reference :

4/22/99 12:57:44 PM

Last Modified: 01-SEP-99

## Article #16039: Handling " system error cannot read cdrom.bmp"

Technical Information Database

TI1039D.txt Handling " system error cannot read cdrom.bmp"

Category :General Programming

Platform :All

Product :Pascal All

### Description:

If you receive the system error " system error cannot write to device CD-ROM" while writing to the file cdrom.bmp and you receive the options to cancel or to retry do the following.

1. Back up the autoexec.bat and the config.sys file in preparation for editing them.
2. Search the autoexec.bat for the lines "mcsex\d:cdrom". Change "cdrom" to "MyCDROM". Search the config.sys for a reference to "cdrom" and then make the same substitution in this file.
3. Be sure and save the changes when exiting the editor and then reboot the system.

This problem is a conflict between MSDOS and MSWINDOWS.

### Reference:

7/16/98 4:33:59 PM

Last Modified: 01-SEP-99

## Article #16045: Different colored characters in a string grid

Technical Information Database

TI1045D.txt Different colored characters in a string grid

Category :General Programming

Platform :All

Product :Delphi All

### Description:

This unit will show how to have text in a string grid where the characters are different colors.

```
unit Strgr;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs, Grids, StdCtrls, DB;
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    StringGrid1: TStringGrid;
```

```
    procedure StringGrid1DrawCell(Sender: TObject; Col, Row: Longint;  
      Rect: TRect; State: TGridDrawState);
```

```
  private
```

```
    { Private declarations }
```

```
  public
```

```
    { Public declarations }
```

```
end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject; Col, Row:  
Longint;
```

```
  Rect: TRect; State: TGridDrawState);
```



```
const
  CharOffset = 3;
begin
  with StringGrid1.canvas do
    begin
      font.color := clMaroon;
      textout(rect.left + CharOffset, rect.top + CharOffset, 'L');
      font.color := clNavy;
      textout(rect.left + CharOffset + TextWidth('L'),
        rect.top + CharOffset, 'loyd');
    end;
  end;
end.

end.
```

Reference:

7/16/98 4:33:59 PM

Last Modified: 01-SEP-99

## Article #16051: Returns the amount required to repay a debt.

Technical Information Database

TI1051D.txt Returns the amount required to repay a debt.

Category :General Programming

Platform :All

Product :Delphi All

Description:

PAYMENT()

Returns the periodic amount required to repay a debt.

```
function payment(princ, int, term: double): double;
var temp: double;
begin
  int := int / 100;
  temp := exp(ln(int + 1) * term);
  result := princ * ((int * temp) / (temp - 1));
end;
```

Syntax

PAYMENT( , , )

The original amount to be repaid over time.

The interest rate per period expressed as a positive decimal number. Specify the interest rate in the same time increment as the term. It is to be expressed as a percentage. The number is divided by 100 inside the function.

The number of payments. Specify the term in the same time increment as the interest.

## Description

Use PAYMENT( ) to calculate the periodic amount (payment) required to repay a loan or investment of amount in payments. PAYMENT( ) returns a numeric value based on a fixed interest rate compounding over a fixed length of time. If is positive, PAYMENT( ) returns a positive number. If is negative, PAYMENT( ) returns a negative number. Express the interest rate as a decimal. For example, if the annual interest rate is 9.5%, is 9.5 for payments made annually.

Express and in the same time increment. For example, if the payments are monthly, express the interest rate per month, and the number of payments in months. You would express an annual interest rate of 9.5%, for example, as 9.5/12, which is the 9.5% divided by 12 months. The formula used to calculate PAYMENT( ) is as follows:

$$\text{pmt} = \text{princ} * \frac{\text{int} * (1 + \text{int})^{\text{term}}}{(1 + \text{int})^{\text{term}} - 1}$$

where int = rate / 100 (as a percentage).

For the monthly payment required to repay a principal amount of \$16860.68 in five years, at 9% interest, the formula expressed as a dBASE expression looks like this:

```
MyVar := PAYMENT(16860.68, 9/12, 60)           {Returns 350.00}
```

## Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16057: How to click and move components at runtime.

Technical Information Database

TI1057D.txt    How to click and move components at runtime.

Category     :VCL

Platform     :All

Product      :Delphi   All

Description:

Q: How can I program a component, such as a TPanel, so that I can move it around with a click and drag of the mouse?

A: This code goes on the OnMouseDown event of the component in question (a TPanel in this case):

```
procedure TForm1.PanellMouseDown(Sender: TObject; Button:
TMouseButton;
    Shift: TShiftState; X, Y: Integer);
const
    SC_DragMove = $F012; { a magic number }
begin
    ReleaseCapture;
    panell.perform(WM_SysCommand, SC_DragMove, 0);
end;
```

Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16063: Delphi 1.02 Maintenance Release Information

Technical Information Database

TI1063D.txt - Delphi 1.02 Maintenance Release Information

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

THE DELPHI 1.02 MAINTENANCE RELEASE

-----

BRIEF OVERVIEW OF DELPHI 1.02

-----

Thank you for your interest in Borland's new RADical application development tool, Delphi. Below you will find information regarding the product's overview as well as the problems fixed with this release, but here, we would like to give you a brief overview of Delphi 1.02.

- o The Delphi 1.02 maintenance release is not a patch; it is a complete product.
- o This release is available only through Borland. It is not available via CompuServe, Borland's Web site, Borland's FTP site, or resellers.
- o This maintenance release includes the patch release of Delphi 1.01.
- o Delphi 1.02 is not Delphi32. Delphi32 will be released as a separate product.

To order the Delphi 1.02 maintenance release, please call our Order Desk at 800-453-3375 extension 1327.

NOTE: All printed and online documentation assume that you have installed Delphi using the default directory structure.

TABLE OF CONTENTS

-----

1. Product overview
2. Problems fixed in this release

3. Delphi Supplemental Documentation Set
4. Minimum system requirements

## 1. PRODUCT OVERVIEW

-----

Delphi's visual design environment lets you create sophisticated Windows applications faster than any other development tool. Because Delphi is built around and optimizing native code compiler, Delphi applications are up to 10-20 times faster than interpreted code.

Delphi integrates the Borland Database Engine, so you have instant support for dBase, Paradox, and ODBC local databases.

Delphi includes the Local InterBase Server so you can create standalone client/server applications with a high-performance ANSI SQL-92 compatible database server.

Delphi includes Borland's award-winning ReportSmith report writing tool, which allows programmers to prepare innovative reports using live data in all popular database formats.

Delphi Client/Server includes all of Delphi plus high performance native drivers for Oracle, Sybase, Microsoft SQL Server, Informix and InterBase remote servers with unlimited application deployment. You also get team development support, ReportSmith SQL, a Visual Query Builder, source code to the Visual Component Library (VCL), and the Local InterBase Server Deployment Kit.

Delphi includes an Open Tools API capability that allows you to extend the Delphi environment to include your own tools, experts, and so on. By using this API, you can seamlessly integrate these extensions into the Delphi environment. For details, refer to the file TOOLINTF.PAS located in the \DELPHI\DOC for Delphi or in the \DELPHI\SOURCE\VCL directory for Delphi Client/Server. Source code for an example expert is also located in the \DELPHI\DEMOS\EXPERTS directory.

## 2. PROBLEMS FIXED IN THIS RELEASE

-----

The purpose of this section is to provide a general list of problems that are fixed in the 1.02 Delphi release.

## ReportSmith

-----

- o Create master/detail reports to combine multiple reports in one using heterogenous data.
- o Choose to group, sort or summarize data locally or on a database server.
- o Access data using Borland's BDE drivers.
- o Include columns in a report for query-only or value-only.
- o Utilize an updated ReportBasic macro language.
- o Place page totals to display summary values for each page in a report.
- o Create data dictionaries to use with the PC or SQL versions of ReportSmith, to simplify the view of the data.

## Delphi

-----

- o Fixed problems in the Delphi online help system.

## Local InterBase

-----

- o Improved performance of index creation and SQL request involving sort operations on data that exceeds the database cache as defined during database create.
- o Improved I/O diagnostic and error messages from the server.
- o Enhanced the Local InterBase engine to handle complex requests that reference more than 500 columns causing internal buffers to exceed 64k.
- o Improved performance of Local InterBase requests that reference more than 500 columns.
- o Removed the Local InterBase internal requirement that databases have a page size of 1024. It is now possible to create and use a Database specifying any allowable InterBase page size.
- o Numeric overflow errors are now trapped by Local InterBase and reported correctly to the client application.
- o Fixed the incompatibility with Dashboard when using Dashboard, from Starfish Inc., and viewing users in the InterBase security database (Task|User Security) with the InterBase Server Manager tool.
- o Fixed cleanup problems encountered when exiting the Windows Interactive SQL tool while executing a database validate (Task|Database Validate) in the InterBase Server Manager.

## BDE

---

This version of the Borland Database Engine is the same version that is used with the new Visual dBASE 5.5.

- o Fixed TQuery and TTable so that they support Oracle synonyms. Synonyms can now be viewed by typing its name in the name table property.
- o Editing now supported of an opened Paradox table simultaneously in Delphi with referential integrity constraints.
- o Fixed the ability to support Stored Procedures that have string parameters.
- o Fixed problems relating to accessing tables on Lantastic 6.0 network.
- o Removed the nine parameter limitation for SQL queries.
- o Fixed problems relating to opening Access 2.0 files through ODBC.
- o Fixed DBD so that it is now able to open Watcom 4.0 tables.
- o Fixed 32K memory leak associated with live-query results sets and also for local queries involving joins.
- o Fixed problem moving batch from ASCII fixed-length files.
- o Now supports the SYBASE forcedindex feature.

### 3. DELPHI SUPPLEMENTAL DOCUMENTATION SET

-----  
The Supplemental Documentation Set includes the following:

- o Object Pascal Language Reference Guide  
(objlang.pdf is 1.3M.)
- o Delphi Visual Component Library  
Reference (vclref.pdf is 5.1M.)

In the United States, you can purchase hard copy versions of the Delphi Supplemental Documentation Set by calling the Borland order desk at 1-800-331-0877.

Internationally, you can purchase hard copy versions of the Delphi Supplemental Documentation Set by contacting the nearest Borland office.

### 4. MINIMUM SYSTEM REQUIREMENTS

-----  
Delphi requires Windows 3.1 or a 100% compatible operating system, an 80386 or newer processor (486 recommended), and 6Mb of system memory (Delphi Client/Server requires 8Mb, 12Mb or more is recommended for Client/Server development). A minimum installation requires approximately 30Mb of disk space (a full installation of Delphi Client/Server requires approximately



80Mb).

DCC.EXE, the DOS command-line compiler, requires at least 1Mb of extended memory.

Delphi has been tested under Windows 3.1, Windows for Workgroups 3.11, Windows NT 3.5, OS/2 Warp, and Windows 95.

Reference:

3/30/99 1:04:00 PM

Last Modified: 01-SEP-99

## Article #16069: Creating Dynamic Components at Runtime

Technical Information Database

TI1069D.txt     Creating Dynamic Components at Runtime

Category     :General Programming

Platform     :All

Product      :Delphi 1.0

### Description:

Using Delphi forms and components is simple. When coupled with the object inspector, controlling those objects requires little effort. Creating these types dynamically is also not difficult. This document is intended to give you some tips and hints on how to make dynamic component work for you.

(please note this term "dynamically" is subjective, behind the scenes,

Delphi creates all objects dynamically. The information presented herein

is for the programmer setting creation/properties/deletion the given type at run time)

All types (a form or a component) can be created dynamically. To do this, one needs to put a declaration in the VAR section of their code. This does not create an instance of the object, it creates a pointer. This pointer resides in the data segment (if the variable is declared globally) or the stack (if the variable is declared local to a procedure

or function). In order to instantiate this class, you must call the constructor. This will allocate memory in the computers global heap for the class instance. Trying to access the component before allocating memory will produce a general protection fault.

The Create() constructor is a class method descended from the TObject Class. Create() returns a pointer. This method may or may not take one

or more parameters. For most components (all objects which descend from

TComponent are referred to as components), the constructor takes one parameter, the "owner" of type TComponent.

When dynamically creating a component, setting the Owner to "Self" is the most common practice. If you are in one of a form's methods, "Self" refers to that form in that context. If the owner is a valid object, freeing that object will also free the "owned" component. Another common parameter is "Application". This might be used for a visual component that will not be displayed to the program's users. However, most components do not require that you set a specific owner, so it is not uncommon to set the owner to Nil. Keep in mind, though, that you will not be able to change the owner afterwards. If you do pass Nil to a component's constructor, you must remember to call that component's Free method when you are through using the component.

After creation, but before they can be displayed, windowed components (those descending from TWinControl) require the Parent property to be set.

At the time you set the Parent property, it's usually also a good time to set other properties of this components instance, including event handlers (ie, Width, Color, OnClick).

Event handlers are identical to those specified in the object inspector.

Simply set the component's property name for the event you want to handle

to name of the event handler method you want invoked. Example 1 below

would call the method called "myclick" whenever the button is clicked. Please note this method will be sent the appropriate parameters, and its incoming parameter list must be exact.

Example 1:

```
var
  b1 : TButton;
begin
  .
  .
  .
  b1 := TButton.Create(Self);
  with b1 do begin
    Left := 20;
    Top := 20;
    Width := 90;
    Height := 50;
    Caption := 'my button';
```

```

    Parent := Form1;
    OnClick := MyClick; { a procedure I defined somewhere else }
end;
.
.
.
end;

```

The next example demonstrates how to create a button at run time by clicking a predefined button. Note the different way the button has been created. Either way would work. Also note the buttons that are created are not freed in this code, they will be freed when the form is released.

```

unit Unit1;
interface

uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        procedure myClick(Sender: TObject);
    end;

var
    Form1: TForm1;

const
    i : integer = 0;

implementation
{$R *.DFM}

procedure TForm1.myClick(Sender: TObject);
begin
    with Sender as TButton do
        Self.Caption := ClassName + ' ' + Name;

```

```
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  with TButton.Create(self) do begin
    Left := 20;
    Top := 30 + i;
    Width := 120;
    Height := 40;
    Name := 'ThisButton' + IntToStr(i);
    Caption := 'There' + IntToStr(i);
    OnClick := MyClick; { a procedure I defined somewhere else }
    Parent := Form1;
  end; {end with}
  inc(i, 40);
end; {end button1.click}

end.
```

Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16075: Loading a Custom Cursor from a RES File

Technical Information Database

TI1075D.txt Loading a Custom Cursor from a RES File

Category :VCL

Platform :All

Product :Delphi All

Description:

Q: How do I use one of the cursor files in the  
c:\delphi\images\cursors?

A: Use the image editor to load the cursor into a RES file.  
The following example assumes that you saved the cursor in the  
RES file as "cursor\_1", and you save the RES file as MYFILE.RES.

```
(*** BEGIN CODE ***)
{$R c:\programs\delphi\MyFile.res}    { This is your RES file }

const PutTheCursorHere_Dude = 1;      { arbitrary positive number }

procedure stuff;
begin
    screen.cursors[PutTheCursorHere_Dude] := LoadCursor(hInstance,
                                                         PChar('cursor_1'));
    screen.cursor := PutTheCursorHere_Dude;
end;
```

Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16081: Loading Bitmaps and Cursors from RES Files

Technical Information Database

TI1081D.txt Loading Bitmaps and Cursors from RES Files

Category :VCL

Platform :All

Product :Delphi All

Description:

### Loading Bitmaps and Cursors from RES files

Bitmaps and cursors can be stored in a resource (RES) files and linked into your application's EXE file. RES files can be created with Delphi's Image Editor or Borland's Resource Workshop that comes with the Delphi RAD Pack. Bitmaps and cursors stored in RES files (after being bound into an EXE or DLL) can be retrieved by using the API functions LoadBitmap and LoadCursor, respectively.

#### Loading Bitmaps

-----

The LoadBitmap API call is defined as follows:

```
function LoadBitmap(Instance: THandle;  
                   BitmapName: PChar): HBitmap;
```

The first parameter is the instance handle of the module (EXE or DLL) that contains the RES file you wish to get a resource from. Delphi provides the instance handle of the EXE running in the global variable called Hinstance. For this example it is assumed that the module that you are trying to load the bitmap from is your application. However, the module could be another EXE or DLL file. The following example loads a bitmap called BITMAP\_1 from a RES file linked into the application's EXE:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Bmp: TBitmap;  
begin
```

```

Bmp := TBitmap.Create;
Bmp.Handle := LoadBitmap(HInstance, 'BITMAP_1');
Canvas.Draw(0, 0, Bmp);
Bmp.Free;
end;

```

There is one drawback to using the LoadBitmap API call though LoadBitmap is a Windows 3.0 API call and loads in bitmaps only as DDBs (Device Dependent Bitmaps). This can cause color palette problems when retrieving DIBs (Device Independent Bitmaps) from RES files. The code listed below can be used to retrieve DIBs from RES files. This code loads the bitmap as a generic resource, puts it into a stream, and then does a LoadFromStream call which causes Delphi to realize the color palette automatically.

```

procedure TForm1.Button1Click(Sender: TObject);
const
  BM = $4D42; {Bitmap type identifier}
var
  Bmp: TBitmap;
  BMF: TBitmapFileHeader;
  HResInfo: THandle;
  MemHandle: THandle;
  Stream: TMemoryStream;
  ResPtr: PByte;
  ResSize: Longint;
begin
  BMF.bfType := BM;
  {Find, Load, and Lock the Resource containing BITMAP_1}
  HResInfo := FindResource(HInstance, 'BITMAP_1', RT_Bitmap);
  MemHandle := LoadResource(HInstance, HResInfo);
  ResPtr := LockResource(MemHandle);

  {Create a Memory stream, set its size, write out the bitmap
  header, and finally write out the Bitmap }
  Stream := TMemoryStream.Create;
  ResSize := SizeofResource(HInstance, HResInfo);
  Stream.SetSize(ResSize + SizeOf(BMF));
  Stream.Write(BMF, SizeOf(BMF));
  Stream.Write(ResPtr^, ResSize);

  {Free the resource and reset the stream to offset 0}
  FreeResource(MemHandle);
  Stream.Seek(0, 0);

```



```
{Create the TBitmap and load the image from the MemoryStream}
Bmp := TBitmap.Create;
Bmp.LoadFromStream(Stream);
Canvas.Draw(0, 0, Bmp);
Bmp.Free;
Stream.Free;
end;
```

## Loading Cursors

-----

The LoadCursor API call is defined as follows:

```
function LoadCursor(Instance: THandle;
                    CursorName: PChar): HCursor;
```

The first parameter is the Instance variable of the module that contains the RES file. As above, this example assumes that the module that you are trying to load the cursor from is your application. The second parameter is the name of the cursor.

Under the interface section declare:

```
const
  crMyCursor = 5; {Other units can use this constant}
```

Next, add the following two lines of code to the form's OnCreate event as follows:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crMyCursor] := LoadCursor(HInstance, 'CURSOR_1');
  Cursor := crMyCursor;
end;
```

or you may want to change one of the standard Delphi cursors as follows (the Cursor constants can be found in the On-line Help under Cursors Property):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  {This example changes the SQL Hourglass cursor}
  Screen.Cursors[crSQLWait] := LoadCursor(HInstance, 'CURSOR_1');
end;
```

Note: Normally it is necessary to delete any cursor resources with the DeleteCursor, however, in Delphi this is not necessary because Delphi will delete the all cursors in the Cursors array.

Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16087: SQL: Embedded Spaces in Field/Column Names

Technical Information Database

TI1087D.txt SQL: Embedded Spaces in Field/Column Names

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

Implementing SQL with spaces or special characters in field/column names

Implementing SQL statements in Delphi's TQuery component (or the SQL query facilities of Database Desktop, Visual dBASE or Paradox for Windows) requires special syntax for any columns that contain spaces or special characters.

Using the Biolife.DB table of from Delphi's demo data to illustrate, and without the use of any special syntax requirements, a SQL Select statement might be formed as follows,

```
SELECT
  Species No,
  Category,
  Common_Name,
  Species Name,
  Length (cm),
  Length_In,
  Notes,
  Graphic
FROM
  BIOLIFE
```

While appearing normal, the space in the species number and name columns and the column expressing length in centimeters - as well as the parentheses present - cause syntax errors.

Two changes must be taken to correct the syntax of the above SQL statement. First, any columns containing spaces or special characters must be surrounded by single (apostrophe) or double quotes. Secondly, a table reference and a period must precede

the quoted column name. This second requirement is particularly important since a quoted string alone is interpreted as a string expression to be yielded as a column value. A properly formatted statement follows:

```
SELECT
  BIOLIFE."Species No",
  BIOLIFE."Category",
  BIOLIFE."Common_Name",
  BIOLIFE."Species Name",
  BIOLIFE."Length (cm)",
  BIOLIFE."Length_In",
  BIOLIFE."Notes",
  BIOLIFE."Graphic"
FROM
  "BIOLIFE.DB" BIOLIFE
```

The above example uses the table alias BIOLIFE as the table reference that precedes the column name. This reference may take the form of an alias name, the actual table name, or a quoted file name when using dBASE or Paradox tables. The following SQL statements would serve equally well.

Note: This SQL statement may be used provided that the necessary alias is already opened. In the case of the TQuery this means the alias is specified in the DatabaseName property.

```
SELECT
  BIOLIFE."Species No",
  BIOLIFE.Category,
  BIOLIFE.Common_Name,
  BIOLIFE."Species Name",
  BIOLIFE."Length (cm)",
  BIOLIFE.Length_In,
  BIOLIFE.Notes,
  BIOLIFE.Graphic
FROM
  BIOLIFE
```

If an alias is not available then the entire path to the table can be specified as in this example:

```
SELECT
  "C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Species No",
  "C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Category",
```

```
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Common_Name",  
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Species Name",  
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Length (cm)",  
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Length_In",  
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Notes",  
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"."Graphic"
```

```
FROM
```

```
"C:\DELPHI\DEMOS\DATA\BIOLIFE.DB"
```

Finally, two facilities that automatically handle this special formatting exist. The first is the Visual Query Builder that is a part of the Client/Server version of Delphi. The Visual Query Builder performs this formatting automatically as the query is built. The other facility is Database Desktop's Show SQL feature, available when creating or modifying a QBE-type query. After selecting Query|Show SQL from the main menu, the displayed SQL text may be cut and pasted where needed.

Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16093: Dynamically Allocating Arrays

Technical Information Database

TI1093D.txt - Dynamically Allocating Arrays

Category :General Programming  
Platform :All-32Bit  
Product :

Description:

{  
Dynamic Arrays

Is it possible to create a dynamically-sized array in Delphi?

Yes. First, you need to create an array type using the largest size you might possibly need. When creating a type, no memory is actually allocated. If you created a variable of that type, then the compiler will attempt to allocate the necessary memory for you. Instead, create a variable which is a pointer to that type. This causes the compiler to only allocate the four bytes needed for the pointer.

Before you can use the array, you need to allocate memory for it. By using `AllocMem`, you will be able to control exactly how many bytes are allocated. To determine the number of bytes you'll need to allocate, simply multiply the array size you want by the size of the individual array element. Keep in mind that the largest block that can be allocated at one time in a 16-bit environment is 64KB. The largest block that can be allocated at one time in a 32-bit environment is 4GB. To determine the maximum number of elements you can have in your particular array (in a 16-bit environment), divide 65,520 by the size of the individual element.

Example: `65520 div SizeOf(LongInt)`

Example of declaring an array type and pointer:

```
type  
  ElementType = LongInt;
```

```

const
  MaxArraySize = (65520 div SizeOf(ElementType));
  (* under a 16-bit environment *)

type
  MyArrayType = array[1..MaxArraySize] of ElementType;

var
  P: ^MyArrayType;

const
  ArraySizeIWant: Integer = 1500;

```

Then when you wish to allocate memory for the array, you could use the following procedure:

```

procedure AllocateArray;
begin
  if ArraySizeIWant <= MaxArraySize then
    P := AllocMem(ArraySizeIWant * SizeOf(LongInt));
end;

```

Remember to make sure that the value of ArraySizeIWant is less than or equal to MaxArraySize.

Here is a procedure that will loop through the array and set a value for each member:

```

procedure AssignValues;
var
  I: Integer;
begin
  for I := 1 to ArraySizeIWant do
    P^[I] := I;
end;

```

Keep in mind that you must do your own range checking. If you have allocated an array with five members and you try to assign a value to the sixth member of the array, you will not receive an error message. However, you will get memory corruption.

Remember that you must always free up any memory that you allocate. Here is an example of how to dispose of this array:

```

procedure DeallocateArray;

```

```
begin
  P := AllocMem(ArraySizeIWant * SizeOf(LongInt));
end;
```

Below is an example of a dynamic array:

```
}

unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  ElementType = Integer;

const
  MaxArraySize = (65520 div SizeOf(ElementType));
  { in a 16-bit environment }

type
  { Create the array type. Make sure that you set the range to
    be the largest number you would possibly need. }
  TDynamicArray = array[1..MaxArraySize] of ElementType;
  TForm1 = class(TForm)
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  { Create a variable of type pointer to your array type. }
  P: ^TDynamicArray;

const
  { This is a typed constant. They are actually static
```



variables that are initialized at runtime to the value taken from the source code. This means that you can use a typed constant just like you would use any other variable. Plus you get the added bonus of being able to automatically initialize its value. }

```
DynamicArraySizeNeeded: Integer = 10;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin  
  { Allocate memory for your array. Be very careful that you  
    allocate the amount that you need. If you try to write  
    beyond the amount that you've allocated, the compiler will  
    let you do it. You'll just get data corruption. }
```

```
  DynamicArraySizeNeeded := 500;
```

```
  P := AllocMem(DynamicArraySizeNeeded * SizeOf(Integer));
```

```
  { How to assign a value to the fifth member of the array. }
```

```
  P^[5] := 68;
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin  
  { Displaying the data. }
```

```
  Button1.Caption := IntToStr(P^[5]);
```

```
end;
```

```
procedure TForm1.FormDestroy(Sender: TObject);
```

```
begin  
  { Free the memory you allocated for the array. }
```

```
  FreeMem(P, DynamicArraySizeNeeded * SizeOf(Integer));
```

```
end;
```

```
end.
```

Reference:

4/2/99 2:03:27 PM

Last Modified: 01-SEP-99

## Article #16099: Resource Expert: What It Is and How to Install It

Technical Information Database

TI1099D.txt    Resource Expert: What It Is and How to Install It  
Category     :Product Info  
Platform     :All  
Product      :Delphi 1.0

### Description:

Resource Expert: what it is and how to install and use it.

### What is the Resource Expert

The Resource Expert is a Delphi expert add-on that is available as a part of the Delphi Rad Pack. The expert assists the programmer in porting existing projects to Delphi by converting dialog and menu resource scripts intended for use in traditional Windows applications. Dialog resources and their contents are converted to Delphi forms with the analogous controls converted to Delphi components.

### How to install the Resource Expert

The Resource Expert is installed via the Delphi Rad Pack's Resource Workshop 4.5 install procedure. Once installed, it is incorporated into the Delphi component library and is available as an option on the Delphi Help Menu or via the Experts page of the Forms Gallery dialog. Installation of the Resource Expert files may be installed from within the Windows environment or from the command line under Windows 95 or Windows NT.

To install the Resource Expert files from within Windows,

- 1) Begin the installation procedure for Borland Resource Workshop.
- 2) On the third dialog, entitled 'Resource Workshop - Resource Expert Options', ensure that the 'Install Resource Expert' check box is checked.

- 3) The 'Install to:' entry indicates the destination directory for the Resource Expert files, indicating C:\DELPHI\RCEXPRT by default. Change this entry as needed.
- 4) Proceed with the rest of the Resource Workshop installation process as normal.

To install the Resource Expert files from the command line, type the following commands,

- 1) MD C:\DELPHI\RCEXPRT
- 2) CD C:\DELPHI\RCEXPRT
- 3) E:\INSTALL\RW\UNPAQ -X E:\INSTALL\RW\RESEXP.PAK

Note: The last command above assumes that the E: drive is a CD-ROM drive containing the Rad Pack Installation CD.

Once the installation of the Resource Expert files is completed, the Delphi Component Library must be recompiled. To do this,

- 1) Load Delphi.
- 2) Select Options|Install Components.
- 3) Click the Add... button.
- 4) When the Add Module dialog appears, enter the full path name of the rcexpert.pas file or find the file via the Browse... button.
- 5) Finally, choose the OK button on the Install Components Dialog.

## How to use the Resource Expert

To convert a resource script, all source files normally required to compile the script must be present. This would include .RC, .MNU, or .DLG file(s) and any .H or .PAS include files they refer to. Resource scripts typically use WINDOWS.H and BWCC.H. These files are usually located in directories such as \BC4\INCLUDE or \BP7\UNITS. The Resource Expert supports the RC language extensions defined by Resource Workshop.

Again, the Resource Expert may be invoked via the Help|Resource Expert menu option or via the Experts page of the Forms Gallery dialog. The latter will appear if the 'Use on new form' check

box is checked on the Preferences page of the Environment Options dialog.

Once the Resource Expert has been invoked, click the 'Next' button to bypass the page that introduces the expert to the user. The second page of the expert allows the user to select the resource scripts to convert. A number of scripts may be chosen provided that they all reside in the same directory. The particular type of script to view (.RC, .DLG or .MNU) can be selected via the 'List Files of Type' combo box. After selecting the scripts to convert, click the 'Next' button again. The third page presents a single 'Include Path' edit box. Enter the list of directories containing .H, .INC, or .PAS include files used by the resource scripts, (if any). Each directory name should be separated by a semicolon. Again, click the 'Next' button to continue. On the fourth and final page of the expert, the 'Convert' button appears. Clicking it begins the actual conversion process. If the resources script contain many dialogs, the 'Show all forms' check box may be un-checked in order to speed the conversion process and to minimize impact on Windows system resources.

If a syntax error is encountered during the conversion process, the erroneous statement will be discarded and conversion will resume at the next statement or block. Errors will be noted in the log file ERRLOG.TXT and displayed in a Delphi editor window.

Once the conversion process is complete, separate forms for each dialog resource will have been created. For menu resources, a simple form containing the converted menu component will have been created. If a project was active before the conversion began, the converted forms are added to the project. Each form may now be used and modified as would any Delphi form.

Reference:

7/16/98 4:34:00 PM

Last Modified: 01-SEP-99

## Article #16105: Delphi Configuration Files

Technical Information Database

TI1105D.txt - Delphi Configuration Files

Category :Delphi 1.x  
Platform :All  
Product :

### Description:

Delphi employs a number of files for its configuration, some global to the Delphi environment, some project specific. Chief among all the configuration files is DELPHI.INI. DELPHI.INI resides in the Windows directory and contains most of the configurable items to be found within Delphi. Being the largest Delphi configuration file, DELPHI.INI contains settings to control the Delphi speed bar, component palette, component library, gallery, installed experts, editor, printing, tools menu and all the environment options found on the Environment Options dialog. This document will explore DELPHI.INI in depth.

DELPHI.CBT is a copy of DELPHI.INI (minus the ReportSmith section) that is installed along with Delphi and may serve as a sort of backup for restoring the original Delphi setup. It resides in the Windows directory along with DELPHI.INI. Below is the ReportSmith section and its one entry that should be placed in a DELPHI.INI created from DELPHI.CBT.

```
[ReportSmith]
ExePath=C:\RPTSMITH
```

Delphi also makes use of Desktop (.DSK) files. Desktop files, like most Delphi configuration files, are formatted in the same manner as .INI files, with section headers and individual settings in each section. The purpose of desktop files is to retain the appearance and content of the Delphi desktop between sessions or between projects. Each desktop file contains information regarding the presence and appearance of the Delphi main window, the Object Inspector, the Alignment Palette, the Project Manager, and the Watch, Breakpoint, CallStack, and

component list windows. Also kept in each desktop file is the number of editor windows open as well as the names, number and order of files open in each editor window.

If the 'Desktop files' check box (on the Preferences page of the Environment Options dialog) is checked, Delphi will automatically create desktop files for each project closed and saved. Each desktop file carries the same root name as the saved project file. If no project is active when Delphi exits, a default desktop file, DELPHI.DSK, is created. The last active project determines which desktop file Delphi loads at startup. Again, if no project was active when Delphi exited last, then DELPHI.DSK is loaded. While project specific desktop files reside in the same directory with the corresponding project, DELPHI.DSK resides in the \DELPHI\BIN directory. The PrivateDir setting in the Globals section of DELPHI.INI may be used to relocate DELPHI.DSK to a different location.

Option files (.OPT) are another INI-like file in which Delphi maintains values directly corresponding to those settings on the Compiler, Linker, and Directories/Conditionals pages of the Project Options dialog. Each of these pages has a corresponding section in the option file and each setting has a individual entry in that section. Each option file also retains the last parameter string entered via the Run Parameters dialog. An option file is created for each project saved. Like .DSK files, the root name of the .OPT file is the same as its corresponding project and reside in the same directory as that project.

A default option file, DEFPROJ.OPT, is created if the Default check box of the Project Options dialog is checked. The settings in DEFPROJ.OPT serve as the default project settings each time a new project is created.

Additionally, the Delphi command line compiler, DCC.EXE, supports the use of the configuration file DCC.CFG. DCC.CFG is a text file opened when the command line compiler starts and is used in addition to options entered on the command line. Command line options may be placed in DCC.CFG, each on a separate line. When DCC starts, it looks for DCC.CFG in the current directory. If it is not found there, the directory in which DCC.EXE resides is then searched. A sample DCC.CFG follows:

```
/b
/q
/v
/eC:\DELPHI\WORK
```

The above settings instruct the command line compiler to build all units (/b), compile without displaying file names and line numbers (/q), append debug information to the .EXE (/v), and place the compiled units and executable in the C:\DELPHI\WORK directory (/eC:\DELPHI\WORK). The contents of the installed DCC.CFG are included below to serve in restoring it should it be deleted or damaged.

```
/m
/cw
/rD:\DELPHI\LIB
/uD:\DELPHI\LIB
/id:\DELPHI\LIB
```

STDVCS.CFG is a file installed with the Client/Server of Delphi, but is only used in conjunction with the Version Control manager DLL. The contents of the installed STDVCS.CFG are included here to serve in restoring it should it be deleted or damaged.

```
NODELETEWORK WRITEPROTECT
NOCASE VCSID
```

```
COMMENTPREFIX .PAS = "{ "
COMMENTPREFIX .PRJ = "{ "
```

```
NOEXPANDKEYWORDS .FRM
NOEXPANDKEYWORDS .EXE
NOEXPANDKEYWORDS .DLL
NOEXPANDKEYWORDS .DOC
NOEXPANDKEYWORDS .ICO
NOEXPANDKEYWORDS .BMP
```

Lastly, MULTIHELP.INI is a file Delphi uses to provide context-sensitive help across multiple help files. This file should not be modified; doing so may cause the Delphi Help system to behave erratically. The contents of the installed MULTIHELP.INI are included here to serve in restoring it should it be deleted or damaged.

[Index Path]

DELPHI.HLP=C:\DELPHI\BIN  
WINAPI.HLP=C:\DELPHI\BIN  
CWG.HLP=C:\DELPHI\BIN  
CWH.HLP=C:\DELPHI\BIN  
LOCALSQL.HLP=C:\DELPHI\BIN  
VQB.HLP=C:\DELPHI\BIN  
SQLREF.HLP=C:\IBLOCAL\BIN  
WISQL.HLP=C:\IBLOCAL\BIN  
BDECFG.HLP=C:\IDAPI  
RPTSMITH.HLP=C:\RPTSMITH  
RS\_DD.HLP=C:\RPTSMITH  
SBL.HLP=C:\RPTSMITH  
RS\_RUN.HLP=C:\RPTSMITH  
DBD.HLP=C:\DBD

Note:

What follows below is a comprehensive dissection of the DELPHI.INI file. In order to save space, a few conventions were observed in the describing possible values for settings.

Where only one of a limited set of values is applicable, a pipe symbol is used to separate each of the possible value, e.g.:

MapFile=0|1|2|3

allows only the values 0, 1, 2, or 3

Where a single value within a range is applicable, the range of values is presented inside brackets with the minimum and maximum values separated by two periods, e.g.:

GridSizeX=[2..128]

permits any value between 2 and 128, inclusively.

=====

Section: [Globals] - The Globals section contains settings not included in other sections and that have an effect on Delphi as a whole. Items in the Globals section may be changed only by editing DELPHI.INI.



---

PrivateDir=

This item controls where Delphi both creates and locates the files DELPHI.DSK, DELPHI.DMT, DEFPROJ.OPT and STDVCS.CFG. The default location is the \DELPHI\BIN directory. If Delphi is run from a read-only directory (or from a CD-ROM) this item should be set to a writeable directory, either on a network or local drive. This item should contain a fully qualified path, including the drive letter. Example:

PrivateDir=J:\USERS\JSMITH ; Private network directory

HintColor=

This item controls the color of the fly-by hint window for the Delphi IDE. The value may be a decimal or hex constant, or one of the symbolic color constants defined in VCL (e.g. clCyan). Note that the text in the hint window is always painted using clWindowText. The default value is clYellow.

PropValueColor=

This item controls the color of the text in the right-hand (value) pane of the Object Inspector. The value may be a decimal or hex constant, or one of the symbolic color constants defined in VCL (e.g. clBlue). The default value is clWindowText.

Section: [Library] - The Library section contains entries for those settings found on the Library page of the Environment Options dialog (accessed via Options|Environment). The options in this section take effect when the Options|Rebuild Library menu option is chosen.

---

SearchPath=

Specifies search paths where the compiler can find the units

needed to build the component library. Path names should be listed consecutively, separated by a semicolon. This entry is changed via the 'Library Path' combo box. Example:

```
SearchPath=D:\DELPHI\LIB;d:\delphi\rcexpert
```

ComponentLibrary=

Specifies the name of the active component library. This item is changed via the Options|Open Library menu option. It may also be changed from the 'Library filename' edit of the Install Components dialog (accessed via Options|Install Components). Example:

```
ComponentLibrary=D:\DELPHI\BIN\REXPRT.DCL
```

SaveLibrarySource=0|1

Indicates whether Delphi saves the source code for the component library when installing new components or rebuilding it via Options|Rebuild Library. A setting of 1 causes the project source to be saved using the library file's root name with a .DPR extension. The default value is 0. This setting is changed via the 'Save library source code' check box.

MapFile=0|1|2|3

Determines the type of map file produced, if any, when the component library is rebuilt. The map file is placed in the same directory as the library, and it has a .MAP extension. The default value is 0. This setting is changed via the 'Map file' radio button group.

Option	Effect
0 - Off	Does not produce map file.
1 - Segments	Linker produces a map file that includes a list of segments, the program start address, and any warning or error messages produced during the link.
2 - Publics	Linker produces a map file that includes a list of segments, the program start address, any

warning or error messages produced during the link, and a list of alphabetically sorted public symbols.

- 3 - Detailed Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, a list of alphabetically sorted public symbols, and an additional detailed segment map. The detailed segment map includes the address, length in bytes, segment name, group, and module information.

LinkBuffer=0|1

Specifies the location of the link buffer. A setting of 1 causes Delphi to use available disk space for the link buffer; 0 causes the use of available memory. The default value is 0. This setting is changed via the 'Link Buffer' radio button group.

DebugInfo=0|1

Determines whether the component library file is compiled and linked with debug information. A setting of 1 causes the inclusion of debug information. The default setting is 0. The setting is changed via the 'Compile with debug info' check box.

Section: [Gallery] - The Gallery section controls the use and base location of the form and project galleries. It contains those settings found in the Gallery: group box on the Preferences page of the Environment Options dialog.

-----  
BaseDir=

Points to the directory where Delphi attempts to find Gallery files. To share a gallery directory with other users, set this item to point to a shared network directory. This item should contain a fully qualified path, including the drive letter. This entry may be changed only by editing DELPHI.INI.

Example:

```
BaseDir=D:\DELPHI\GALLERY
```

```
GalleryProjects=0|1
```

Indicates whether Delphi displays the Browse Gallery dialog box when the File|New Project menu option is chosen. A setting of 1 causes the Browse Gallery dialog box to display. The default setting is 0. The setting is changed via the 'Use on New Project' check box.

```
GalleryForms=0|1
```

Indicates whether Delphi displays the Browse Gallery dialog box when the File|New Form menu option is chosen. A setting of 0 prevents the Browse Gallery dialog box from displaying. The default setting is 1. The setting is changed via the 'Use on New Form' check box.

Section: [Experts] - The Experts section lists the Experts which Delphi will attempt to load and initialize upon startup. Any value may be used on the left of the equals sign, as the item name is not interpreted. Borland recommends using a combination of the vendor name and the product name. Example:

```
[Experts]
```

```
ComponentWare.CommExpert=c:\delphi\cware\commexpt.dll
```

```
CodeFast.TheExpert=c:\delphi\codefast\codefast.dll
```

Section: [ReportSmith] - The ReportSmith section contains just one entry which specifies the directory in which ReportSmith is installed.

---

```
ExePath=
```

ExePath indicates the location of RPTSMITH.EXE. This entry is

placed in DELPHI.INI at install time and may be changed only by editing DELPHI.INI. Example:

```
ExePath=D:\RPTSMITH
```

Section: [Session] - The Session section and its one entry identify the active project when Delphi was last closed.

---

Project=

Identifies the active project when Delphi was last closed. This setting is only meaningful if the DesktopFile setting in the AutoSave section is set to 1. This setting also serves to identify the project's desktop file (using a .DSK extension). This setting is updated automatically when Delphi exits.

Example:

```
Project=D:\DELPHI\WORK\MAILAPP.DPR
```

Section: [MainWindow] - The MainWindow section defines characteristics of the Delphi main window as they relate to the speedbar and component palette. The SpeedBar Layout section details the actual contents of the speedbar. Likewise, the .Palette section details the actual contents of the component palette.

---

Split=[-1..400]

Indicates the horizontal position if the vertical bar separating the speedbar and component palette. The default value is 183. This setting is changed by moving the split bar with the mouse.

SpeedHints=0|1

Determines whether hints are displayed as the mouse passes over buttons on the speedbar. A setting of 0 prevents the display

of speedbar hints. The default setting is 1. This setting is changed using the Show Hints menu option of the speedbar speedmenu.

PaletteHints=0|1

Determines whether hints are displayed as the mouse passes over buttons on the palette. A setting of 0 prevents the display of palette hints. The default setting is 1. This setting is changed using the Show Hints menu option of the palette speedmenu.

Speedbar=0|1

When set to 0, prevents the display of the speedbar. The default setting is 1. This setting is changed via the View|Speedbar menu option or via the Hide option of the speedbar speedmenu.

Palette=0|1

When set to 0, prevents the display of the component palette. The default setting is 1. This setting is changed via the View|Component Palette menu option or via the Hide option of the component palette speedmenu.

Section: [Speedbar Layout] - The Speedbar Layout details the specific contents of the speedbar. The contents of this section are changed via the Configure option of the speedbar speedmenu.

---

Count=[0..52]

Specifies the number of buttons on the speedbar. The default is 14.

Button[0..51]=n,x,y

This entry appears once for each button on the speedbar. Each button entry is uniquely numbered, the first being Button0. The number n identifies a unique pre-defined id code. The x value is a number specifying the horizontal position of the button on the speedbar. The y value is a number specifying the vertical position of the button on the speedbar. Below is a listing of the default speedbutton set and their corresponding menu options.

```

Button0=30001,4,2      ; File|Open Project...
Button1=30002,27,2    ; File|Save Project
Button2=30007,4,25    ; File|Open File...
Button3=30008,27,25   ; File|Save File
Button4=30009,50,2    ; File|Add File...
Button5=30010,50,25   ; File|Remove File...
Button6=30069,79,2    ; View|Units...
Button7=30070,102,2   ; View|Forms...
Button8=30068,79,25   ; View|Toggle Form/Unit
Button9=30004,102,25  ; File|New Form
Button10=30090,131,2  ; Run|Run
Button11=30093,154,2  ; Run|Program Pause
Button12=30092,131,25 ; Run|Trace Into
Button13=30091,154,25 ; Run|Step Over

```

Section: [Desktop] - The Desktop section contains a single entry that determines which desktop settings are saved when Delphi exits. This section and its one entry is only meaningful if the DesktopFile entry in the AutoSave section is 1.

-----  
SaveSymbols=0|1

Determines if browser symbol information is saved along with Desktop information when Delphi exits. This setting is changed via the 'Desktop contents:' radio button group box. The default setting is 1.

Option	Effect
-----	-----
0 - Desktop only	Saves directory information, open files in the editor, and open windows.
1 - Desktop and symbols	Saves desktop information and browser

symbol information from the last successful compile.

Section: [AutoSave] - The Autosave section determines which files and options are saved automatically when the current project is run or when Delphi exits. This section corresponds to the 'Autosave options:' group box of the Preferences page of the Environment Options Dialog.

---

EditorFiles=0|1

When set to 1, causes Delphi to save all modified files in the Code Editor when Run|Run, Run|Trace Into, Run|Step Over, or Run|Run To Cursor are chosen, or when Delphi exits. The default setting is 0. This setting is changed via the 'Editor files' check box on the Preferences page of the Environment Options Dialog.

DesktopFile=0|1

When set to 0, prevents Delphi from saving the arrangement of the desktop when a project is closed or when Delphi exits. The default setting is 1. This setting is changed via the 'Desktop' check box on the Preferences page of the Environment Options Dialog.

Note: Further discussion regarding desktop files are discussed below under Desktop (.DSK) files.

Section: [FormDesign] - The FormDesgin section contains those settings that control the appearance and behavior of a forms grid at design time. This section corresponds to the 'Form designer:' group box of the Preferences page of the Environment Options Dialog.

---

DisplayGrid=0|1

Determines the design time visibility of the dots that comprise



the form grid. A setting of 0 avoids grid display. The default setting is 1. This setting is changed via the 'Display grid' check box.

SnapToGrid=0|1

Indicates whether components are automatically aligned with the grid when components are moved with the mouse. A setting of 0 avoids grid alignment. The default setting is 1. This setting is changed via the 'Snap to grid' check box.

GridSizeX=[2..128]

Sets grid spacing in pixels along the x-axis. The default value is 8. This setting is changed via the 'Grid Size X' edit.

GridSizeY=[2..128]

Sets grid spacing in pixels along the y-axis. The default value is 8. This setting is changed via the 'Grid Size Y' edit.

DefaultFont=

This item controls the default font for new forms. The name of the font, the font size, and optionally the style of the font may be entered, each separated by commas. (Supported font styles are "bold" and "italic.") This setting may be changed only by editing DELPHI.INI. Example:

DefaultFont=MS Sans Serif, 8, bold, italic

Section: [Debugging] - The Debugging section contains those settings that control integrated debugging and the appearance of Delphi during project execution. This section corresponds to the 'Debugging:' group box of the Preferences page of the Environment Options Dialog.

---

IntegratedDebugging=0|1

Allows or prevents the uses of the Delphi Integrated Debugger. A setting of 0 prevents integrated debugging. The default setting is 1. This setting is changed via the 'Integrated Debugging' check box.

DebugMainBlock=0|1

When set to 1, causes the debugger to stop at the first unit initialization that contains debug information. The default setting is 0. This setting is changed via the 'Step program block' check box.

BreakOnExceptions=0|1

When set to 1, stops the application when an exception is encountered and displays the following the exception class, exception message and the location of the exception. When set to 0, exceptions do not stop the running application. The default setting is 1. This setting is changed via the 'Break on exception' check box.

MinimizeOnRun=0|1

When set to 1, minimizes Delphi when the current project is executed. The default is 0. This setting is changed via the 'Minimize on run' check box.

HideDesigners=0|1

When set to 1, hides designer windows, such as the Object Inspector and Form window, while the application is running. The default setting is 1. This setting is changed via the 'Hide designers on run' check box.

NoResetWarning=0|1

When set to 1, prevents Delphi from presenting a warning message when Program Reset is selected. The default setting is 0. This setting may be changed only by editing DELPHI.INI.

Section: [Compiling] - The compiling section contains a single entry that determines whether the user is presented with a dialog that reports compiler progress. This section corresponds to the 'Compiling:' group box of the Preferences page of the Environment Options Dialog.

---

ShowCompilerProgress=0|1

Specifies whether compilation progress is reported. A setting of 1 causes Delphi to display a window detailing compilation progress. The default setting is 0. This setting is changed via the 'Show compiler progress' check box.

Section: [Browser] - The Browser section contains settings that are found on the Browser page of the Environment Options dialog. These settings specify how ObjectBrowser functions and what symbol information is displayed.

---

Filters=

This setting determines which filters are active in the Object Browser. The value is the sum of the values listed below for each filter desired.

Value	Filter
-----	-----
2	Constants
4	Types
8	Variables
16	Functions and Procedures
32	Properties
128	Inherited
256	Virtuals only
1024	Private
2048	Protected

4096 Public  
8192 Published

The default setting is 15806, which activates all filters. Each filter corresponds to a check box in the 'Symbol filters:' group box. For example, the following setting activates the Properties, Public and Published filters:

Filters=12320 ; 8192 + 4096 + 32 = 12320

InitialView=1|2|3

InitialView determines the type of information the browser displays when first opened. The default setting is 2. This setting is changed via the 'Initial view:' radio button group box.

Value	Viewed
-----	-----
1	Units
2	Objects
3	Globals

Sort=0|1

When set to 1, causes Delphi to display symbols in alphabetical order by symbol name. When set to 0, symbols display in order of declaration. The default setting is 0. This setting is changed via the 'Sort always' check box.

QualifiedSymbols=0|1

When set to 1, causes Delphi to display the qualified identifier for a symbol. When set to 0, only the symbol name is displayed. The default setting is 0. This setting is changed via the 'Qualified symbols' check box.

CollapsedNodes=

Specifies which branches of the object tree hierarchy are collapsed when the ObjectBrowser is started. This entry is a

list of class names, separated by separated by semicolons.  
This setting is changed via the 'Collapse Nodes:' combo box.  
Example:

```
CollapsedNodes=Exception;TComponent
```

```
ShowHints=0|1
```

Determines whether hints are displayed as the mouse passes over filter buttons. A setting of 0 prevents the display of filter hints. The default setting is 1. This setting is changed using the Show Hints menu option of the ObjectBrowser speedmenu.

Section: [Custom Colors] - The Custom colors section lists up to sixteen user defined colors. Each color is specified as a six-digit hexadecimal RGB value. An unused color entry is indicated by the hexadecimal value FFFFFFFF. Entries in this section are created and updated via the Color dialog of any components Color property (accessed by double-clicking the entry area of the Color property).

---

```
Color[A..P]=
```

Specifies an individual RGB value for a user defined color.

Section: [Print Selection] - The Print Selection section contains those options that appear when the File|Print menu option is chosen. These settings correspond to the options displayed in the 'Options:' group box.

---

```
HeaderPage=0|1
```

When set to 1, Delphi includes the name of the file, current date, and page number at the top of each page. The default setting is 0. This setting is changed via the 'Header/page number' check box.

LineNumbers=0|1

When set to 1, Delphi places line numbers in the left margin of the printed output. The default setting is 0. This setting is changed via the 'Line numbers' check box.

SyntaxPrinting=0|1

When set to 1, Delphi uses bold, italic, and underline characters to indicate elements with syntax highlighting. When set to 0, Delphi uses no special formatting when printing. The default value is 1. This setting is changed via the 'Syntax print' check box.

UseColor=0|1

When set to 1, causes Delphi to print colors that match colors on screen. This option requires that the current printer support color. The default value is 0. This setting is changed via the 'Use Color' check box.

WrapLines=0|1

When set to 1, causes Delphi to use multiple lines to print characters beyond the page width. When set to 0, code lines are truncated and characters beyond the page width do not print. The default value is 0. This setting is changed via the 'wrap lines' check box.

LeftMargin=[0..79]

Specifies the number of character spaces used as a margin between the left edge of the page and the beginning of each line. The default value is 0. This setting is changed via the 'Left margin' edit.

Section: [Highlight] - The Highlight section contain those

settings that determine the syntax and context specific colors used in the Code Editor. The settings in this section are changed via the Editor Colors page of the Environment Options dialog.

---

ColorSpeedSetting=0|1|2|3

Determines which color scheme was last selected. Changing this setting directly does not affect the actual colors used for individual elements. The Color SpeedSetting combo box does not save color schemes; it only serves as a quick means of setting all color elements at once. The default setting is 0. The table below shows each value's corresponding speedsetting.

Value	SpeedSetting
0	Defaults
1	Classic
2	Twilight
3	Ocean

=

All the color entries correspond to a single color element. Each color element entry uses the following format:

=fRGB,bRGB,attr,deffore,defback,fcell,bcell

Value code	Meaning
fRGB	Foreground RGB value
bRGB	Background RGB value
attr	Text attribute; zero or more of B, I and U
deffore	Use default foreground color (1=yes, 0=no)
defback	Use default background color (1=yes, 0=no)
fcell	Foreground color grid cell number
bcell	Background color grid cell number

Section: [Editor] - This section describes the appearance and behavior of the Delphi Code Editor. Settings from both the Editor options and Editor display pages are detailed

here.

---

DefaultWidth=  
DefaultHeight=

These two items, if present, control the initial width and height of the Delphi Code Editor window. Delphi does not update these values, but it does read them each time a Code Editor is created. The default width is 406; the default height is 234. These settings may be changed only by editing DELPHI.INI.

FontName=  
FontSize=

These settings specify the name and size, respectively, of a mono-spaced font that the Code Editor uses to display text. Courier New is the default font, 10 the default size. These entries may be changed via the 'Editor font:' and 'Size:' combo boxes on the Editor display page.

BlockIndent=[1..16]

Specifies the number of spaces to indent a marked block. The default value is 1. This setting may be changed via the 'Block indent' combo box on the Editor display page.

UndoLimit=[0..]

Specifies the number of keystrokes that can be undone, which is limited by available memory. The default value is 32,767. This setting may be changed via the 'Undo limit:' combo box on the Editor Options page.

TabRack=

Determines the columns at which the cursor will move to each time the Tab key is pressed. Each successive tab stop must be separated by a space and must be larger than its predecessor. If only one number is specified, tab stops are spaced apart



evenly, using that number. If two numbers are specified then tab stops occur at the specified positions and at positions that mark the difference between the two values. The default tab stops are 9 and 17. This setting may be changed via the 'Tab stops:' combo box on the Editor Options page. Note: this option has no effect if the smart tabs setting is enabled.

`RightMargin=[0..1024]`

Specifies the right margin of the Code Editor. The default value is 80. The valid range is 0 to 1024. This setting may be changed via the 'Right margin:' combo box on the Editor display page.

`Extensions=`

#### Combo Box

Specifies file masks of those files that will display with syntax highlighting. Typically, only specific extensions are included. The default setting is '\*.PAS;\*.DPR;\*.DFM;\*.INC;\*.INT'. This setting may be changed via the 'Syntax extensions:' combo box on the Editor Options page. Example:

```
Extensions=*.PAS;*.DPR;*.SRC
```

`FindTextAtCursor=0|1`

When set to 1, causes Delphi to Place the text at the cursor into the 'Text To Find' combo box in the Find Text dialog box when the Search|Find menu option is chosen. When set to 0, the default setting, the search text must be typed in. This entry may be changed via the 'Find text at cursor' check box on the Editor Options page.

`BRIEFRegularExpressions=0|1`

When set to 1, permits the use of Brief-style regular expressions when searching for text. The default setting is 0. This entry may be changed via the 'BRIEF regular expressions' check box on the Editor Options page.

PreserveLineEnds=0|1

Determines whether end-of-line characters are changed to carriage return/line feed pairs or are preserved. When set to 0, Delphi converts end-of-line characters to carriage return/line feed pairs. The default value is 1. This entry may be changed via the 'Preserve Line Ends' check box on the Editor display page.

FullZoom=0|1

Determines whether the Code Editor fills the entire screen when maximized. When set to 0 (the default), the Code Editor does not cover the Delphi main window when maximized. A setting of 1 allows the Code Editor window to encompass the entire screen. This setting may be changed via the 'Zoom to full screen' check box on the Editor Display page.

DoubleClickLine=0|1

When set to 1, causes Delphi to highlight the whole line when the user double-clicks any character in the line. When set to 0 (the default), only the selected word is highlighted. This entry may be changed via the 'Double click line' check box on the Editor Options page.

BRIEFCursors=0|1

Determines whether Delphi uses BRIEF-style cursor shapes in the Code Editor. A setting of 1 causes Delphi to use Brief-style cursors. The default setting is 0. This setting may be changed via the 'BRIEF cursor shapes' check box on the Editor Display page.

ForceCutCopyEnabled=0|1

When set to 1, enables the Edit|Cut and Edit|Copy menu options, even when no text is selected. The default setting is 0. This entry may be changed via the 'Force cut and copy enabled' check

box on the Editor Options page.

KeyBindingSet=0|1|2|3

Determines which pre-defined key mapping set Delphi recognizes. The default setting is 0. This setting may be changed via the 'Keystroke mapping:' list box on the Editor Display page. The table below identifies the appropriate mapping for the desired value.

Value	Mapping
-----	-----
0	Default
1	Classic
2	Brief
3	Epsilon

Mode=

This setting determines the state of sixteen of the options available on the Editor Options page and two of the options on the Editor Display page. The value is the sum of the values listed below for each check box checked. Unless noted, all the options below correspond to a similarly named check box on the Editor Options page.

- 1 Insert mode - Inserts text at the cursor without overwriting existing text.
- 2 Auto indent mode - Positions the cursor under the first nonblank character of the preceding nonblank line when Enter is pressed.
- 4 Use tab character - Inserts tab character. If disabled, inserts space characters. This option and the Smart Tabs option are mutually exclusive. enabled, this option is off.
- 16 Backspace un-indent - Aligns the insertion point to the previous indentation level (out-dents it) when Backspace is pressed, if the cursor is on the first nonblank character of a line.
- 32 Keep trailing blanks - Saves trailing spaces and tabs present at the end of a line.
- 64 Optimal fill - Begins every auto-indented line with the minimum number of characters possible, using tabs

and spaces as necessary.

- 128 Cursor through tabs - Enables the arrow keys to move the cursor to the beginning of each tab.
- 256 Group undo - Undoes the last editing command as well as any subsequent editing commands of the same type when Alt+Backspace, Ctrl+Z is pressed or the Edit|Undo menu option is chosen.
- 512 Persistent blocks - Keeps marked blocks selected even when the cursor is moved, until a new block is selected.
- 1024 Overwrite blocks - Replaces a marked block of text with whatever is typed next. If Persistent Blocks is also selected, text entered is added to the currently selected block.
- 4096 Create backup file - Creates a backup file when source files are saved. This item is set via the 'Create backup file' check box on the Editor Display page.
- 8192 Use Syntax highlight - Enables syntax highlighting.
- 16384 Visible right margin - Enables the display of a line at the right margin of the Code Editor. This item is set via the 'Visible right margin' check box on the Editor Display page.
- 32768 Smart tabs - Tabs to the first non-whitespace character in the preceding line. This option and the Smart Tabs option are mutually exclusive.
- 131072 Cursor beyond EOF - Allows cursor positioning beyond the end-of-file.
- 262144 Undo after save - Allows retrieval of changes after a save.

EditorSpeedSetting=0|1|2|3

Determines which editor emulation scheme was last selected. Changing this setting directly does not affect the actual keystroke mapping or the editor options used. The Editor SpeedSetting combo box does not save emulation schemes; it only serves as a quick means of setting many editor options at once. The default setting is 0. The table below shows each value's corresponding speedsetting.

Value	SpeedSetting
-----	-----
0	Default keymapping

- 1 IDE classic
- 2 Brief emulation
- 3 Epsilon emulation

Section: [.Palette] - This section describes the content of the Component Palette. Each entry name in this section matches a single page name on the component palette. The value for each entry is a list of the component type names that appear on that page, each separated by a semicolon. This section appears once for each component library configured via the Palette page of the Environment Options dialog.

Section: [Transfer] - The Transfer section defines those items that appear on the Tools menu. Entries in this section are defined when using the Tool Properties dialog. The Tool Properties dialog is itself accessed via the Options|Tools menu option.

---

Count=

Specifies the number of items that should appear on the Tools menu. This item is changed by adding or removing programs from the Tools Options dialog.

Title#=

Path#=

WorkingDir#=

Params#=

These entries appear once each for every item on the Tools menu. Each item name is immediately followed by a number indicating its position in the Tools menu, zero being the first.

Title#            Specifies the text that actually appears on the Tools menu.

Path#            Specifies the full path to the program that the menu option will execute.

WorkingDir#      Determines the current directory when the program starts.

Params#            Specifies the parameters to pass to the program at startup.

Section: [Closed Files] - The Closed Files section lists the full path name of the last three closed project files. The files are listed in the order of most recently used first. Each entry takes the form

File\_#=.DPR,col1,row1,col2,row2

where # is either 0, 1 or 2. Col1 identifies the first visible column in the code editor, row1 the first visible row. Col2 is the cursor column, row2 the cursor row.

Section: [VBX] - The VBX section contains various settings that are available when installing a VBX into the Delphi Component Library.

---

VBXDir=

Contains the last location from which a VBX was installed. This value is saved automatically by Delphi upon installing a VBX.

UnitDir=

Specifies the last location in which Delphi placed a source unit for use with the previously installed VBX. This value is saved automatically by Delphi upon installing a VBX.

PalettePage=BVSP

This entry retains the last specified name of the component palette page onto which Delphi placed the most recently installed VBX. This value is saved automatically by Delphi upon installing a VBX.

Section: [Version Control]

---

VCSManager=

This item specifies the fully qualified path of a Version Control manager DLL. Delphi Client/Server, which includes team support, supplies a Version Control manager by the name STDVCS.DLL, located in the \BIN directory. Example:

VCSManager=d:\delphi\bin\stdvcs.dll

Section: [Resource Expert] - The Resource Expert section appears only if the Delphi Resource Expert is installed. This section has but one entry.

---

RCIncludePath=

Specifies the list of directories (separated by semicolons) that the expert should search to find any include files needed for resource file conversion. Example:

RCIncludePath=D:\DELPHI\WORK;D:\RESOURCE\INCLUDE

Section: [History\_##] - A number of history sections, each with a unique number following the underscore, reside in DELPHI.INI. Each history section corresponds directly to a particular combo box in a Delphi dialog. Each section contains at least one entry; the Count entry, indicating the number of history items in the section. Each actual history item is named by an H, followed by its order in the history list, H0 being first. The table below indicates to which combo box the particular section belongs. Only those histories saved by Delphi are listed.

Section	Combo box location
[History_0]	'Text to find', Find Text or Replace Text dialog
[History_1]	'Replace with', Replace Text dialog
[History_2]	'Output directory', Directory/conditionals page

of Project Options dialog  
[History\_3] 'Search path', Directory/conditionals page of  
Project Options dialog  
[History\_7] 'Conditionals', Directory/conditionals page of  
Project Options dialog  
[History\_8] 'Undo Limit', Editor options page of Environment  
Options dialog  
[History\_9] 'Right margin', Editor display page of  
Environment Options dialog  
[History\_10] 'Tab stops', Editor options page of Environment  
Options dialog  
[History\_11] 'Syntax extensions', Editor options page of  
Environment Options dialog  
[History\_12] 'Enter new line number', Go to Line Number dialog  
[History\_18] 'Block indent', Editor options page of  
Environment Options dialog  
[History\_20] 'File name', Open Project dialog  
[History\_23] 'File name', Install VBX file dialog  
[History\_25] 'File name', Unit file name dialog (under  
Install VBX)  
[History\_33] 'Collapse nodes', Browser page of Environment  
Options dialog  
[History\_34] 'Library path', Library page of Environment  
Options dialog  
[History\_35] 'File name', Open Library dialog  
[History\_36] 'File name', Save Project1 As dialog

Reference:

3/30/99 3:03:01 PM

Last Modified: 01-SEP-99



## Article #16111: BDE: Writing Buffer to Disk

Technical Information Database

TI11111D.txt BDE: Writing Buffer to Disk

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

Product: Delphi and the Borland Database Engine

Number: ??????

Versions: 1.x, 2.x

OS: WINDOWS 3.x, WINDOWS 95, WINDOWS NT

DATE: December 7, 1995

TITLE: Using DbUseIdleTime and DbSaveChanges.

### General:

=====

Changes made to a table are not written directly to disk until the table is closed. A power failure or system crash can result in a loss of data, and an inconvenience. To avoid this loss of data, two direct Database Engine calls can be made, both of which have the similar effects. These functions are DbUseIdleTime and DbSaveChanges.

DbSaveChanges(hDBICur):

=====

DbSaveChanges saves to disk all the updates that are in the buffer of the table associated with the cursor (hDBICur). It can be called at any point. For example, one may want to make save changes to disk every time a record is updated (add dbiProcs to uses clause):

```
procedure TForm1.Table1AfterPost(DataSet: TDataSet);
begin
  DbSaveChanges(Table1.handle);
end;
```

This way, one does not have to worry about losing data if a

power failure or system crash occurs after a record update.

DbiSaveChanges can also be used to make a temporary table (created by DbiCreateTempTable) permanent.

This function does NOT apply to SQL tables.

DbiUseIdleTime:

=====

DbiUseIdleTime can be called when the "Windows Message Queue" is empty. It allows the Database Engine to save "dirty buffers" to disk. In other words, it does what DbiSaveChanges does, but performs the operation on ALL the tables that have been updated. This operation however, will not necessarily occur after every record update, because it can only be executed when there is an idle period.

In Delphi, it can be used in this fashion (add dbiProcs to uses clause):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Application.OnIdle := UseIdle;
end;
```

```
procedure TForm1.UseIdle(Sender: TObject; var Done: Boolean);
begin
    DbiUseIdleTime;
end;
```

USAGE NOTES:

=====

Using both DbiUseIdleTime and DbiSaveChanges (after every record modification) is redundant and will result in unnecessary function calls. If the application is one that performs a great deal of record insertions or modifications in a small period of time, it is recommended that the client either call DbiUseIdleTime during an idle period, or call DbiSaveChanges after a group of updates.

If not very many updates are being performed on the table, the client may choose to call DbiSaveChanges after every post or

set up a timer and call DbUseIdleTime when a timer even is generated.

Reference:

7/16/98 4:34:02 PM

Last Modified: 01-SEP-99

## Article #16117: Creating and Using Parameterized Queries

Technical Information Database

TI1117D.txt Creating and Using Parameterized Queries

Category :Database Programming

Platform :All

Product :Delphi All

Description:

Q: How do I pass a variable to a query?

A: First, you must write a query that uses a variable.

```
Select Test."FName", Test."Salary Of Employee"
```

```
From Test
```

```
Where Test."Salary of Employee" > :val
```

Note: If you just write the field name as "Salary of Employee" you will get a Capability Not Supported error. It must be Test."Salary of Employee".

In this can the variable name is "val", but it can be whatever you want (of course). Then, you go to the TQuery's params property and set the "val" parameter to whatever the appropriate type is. In our example here we will call it an integer.

Next, you write the code that sets the parameter's value. We will be setting the value from a TEdit box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    Close;
    ParamByName('val').AsInteger := StrToInt(Edit1.Text);
    Open;
  end;
end;
```

Note: you may want to place this code in a try..except block as a safety precaution.

If you want to use a LIKE in your query, you can do it this way:

Note: This next section uses the customer table from the \delphi\demos\data directory. It can also be referenced by using the DBDEMOS alias.

SQL code within the TQuery.SQL property:

```
SELECT * FROM CUSTOMER
WHERE Company LIKE :CompanyName
```

Delphi code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Query1 do
  begin
    Close;
    ParamByName('CompanyName').AsString := Edit1.Text + '%';
    Open;
  end;
end;
```

An alternate way of referencing a parameter (other than ParamByName) is params[TheParameterNumber].

The way that this line:

```
ParamByName('CompanyName').AsString := Edit1.Text + '%';
```

can be alternately written is:

```
Params[0].AsString := Edit1.Text + '%';
```

The trick to the wildcard is in the concatenating of the percentage sign at the end of the parameter.

Reference :

7/16/98 4:34:02 PM

Last Modified: 01-SEP-99

## Article #16123: Working With Auto-increment Field Types

Technical Information Database

TI1123D.txt Working With Auto-increment Field Types

Category :Database Programming

Platform :All

Product :Delphi 1.0

### Description:

In Delphi applications, the use of tables containing fields that autoincrement, or change automatically in some manner unknown to the application, can be problematic. Paradox, InterBase, Sybase and Informix tables all provide means of inserting or updating field values automatically, without intervention from the front-end application. Not every table operation is affected by this mechanism, however. So, this document will attempt to provide a guideline for dealing with issues relating to the use of such field types in Paradox 5.0, Informix 5.x, MS/Sybase SQL Server 4.x, InterBase 4.0 and Local InterBase tables.

For each table type, a different mechanism provides this behind-the-scenes behavior. Paradox tables support an Autoincrement field type. When new records are added to such tables, the Borland Database Engine determines the highest current value in that column, adds one, and updates the new row with the new value.

For Informix tables, this behavior is provided by an Informix-specific field type called Serial. Serial columns differ from Paradox Autoincrement fields in that their values may be changed, while Autoincrement columns are read-only.

InterBase and MS/Sybase SQL Server tables do not support a special type for this kind of behavior, but may employ triggers to accomplish the same task. Triggers are specialized procedures that reside on the database server and automatically execute in response to events such as table inserts, updates and deletes. The use of tables with associated triggers can be particularly problematic, since triggers are capable of doing much more than just incrementing column values.

The three areas that are affected by these field types are simple inserts, batchmoves, and table linking.

## Handling Update and/or Append BatchMoves

---

### Paradox Tables

Since the Autoincrement field type is a read-only type, attempting to perform a batchmove operation with such a column in the destination table may cause an error. To circumvent this, the TBatchMove components Mappings property must be set to match source table fields to the target destination fields excluding the destination table's Autoincrement field.

### Informix Tables

Batch moving rows to Informix tables with Serial columns will not cause an error in and of itself. However, caution should be used since Serial columns are updateable and are often used as primary keys.

### InterBase Tables

#### MS/Sybase SQL Server Tables

Triggers on InterBase and SQL Server tables may catch any improper changes made to the table, but this depends strictly upon the checks placed in the trigger. Here again, caution should be used since trigger-updated columns are often used as primary keys.

## Linking Tables via MasterSource & MasterFields

---

### Paradox Tables

### Informix Tables

If the MasterFields and MasterSource properties are used to create linked tables in a master-detail relationship and one of the fields in the detail table is an Autoincrement or Serial field, then the matching field in the master table must be a Long Integer field or a Serial field. If the master table is not a Paradox table then the master table's key field may be any integer



type it supports.

InterBase Tables

MS/Sybase SQL Server Tables

Linking with these tables types presents no particular problems relating to trigger-modified fields. The only necessity is matching the appropriate column type between the two tables.

Simple Inserts/Updates

---

Paradox Tables

Since Paradox Autoincrement fields are read-only, they are not typically targeted for update when inserting new records. Therefore, the Required property for field components based on Autoincrement fields should always be set to False. This can be accomplished from within Delphi, using the Fields Editor to define field components at design time by double clicking on the TQuery or TTable component or at runtime with a statement similar to the following.

```
Table1.Fields[0].Required := False;
```

or

```
Table1.FieldName('Fieldname').Required := False;
```

Informix Tables

Although Informix Serial fields are updateable, if their autoincrement feature is to be used, then the Required property of field components based on them should be set to False. Do this in the same manner described for Paradox Tables.

InterBase Tables

MS/Sybase SQL Server Tables

Handling inserts on these trigger-modified table types requires a number of steps for smooth operation. These additional steps are

particularly necessary if inserts are accomplished via standard data-aware controls, such as DBEdits and DBMemos.

Inserting rows on trigger-modified InterBase and SQL Server tables may often yield the error message 'Record/Key Deleted'. This error message appears despite that the table is properly updated on the server. This will occur if:

1. The trigger updates the primary key. This is not only likely when a trigger is used, but is probably the most common reason for using a trigger.
- 2a. Other columns in the table have bound default values. This is accomplished with the DEFAULT clause at table creation in the case of InterBase. or with the sp\_bindefault stored procedure in SQL Server.

or

- 2b. Blob type fields are updated when a new row is inserted.

or

- 2b. Calculated fields are defined in an InterBase table.

The fundamental cause for this is that when the record (or identifying key) is changed at the server, the BDE no longer has means of specifically identifying the record for re-retrieval. That is, the record no longer appears as it did when it was posted, therefore the BDE assumes that the record has been deleted (or the key changed).

Firstly, the field components of trigger-modified fields must have their Required property set to False. Do this in the same manner described for Paradox Tables.

Secondly, to avoid the spurious error, order the table by an index that does not make use of fields updated by the trigger. This will also prevent the newly entered record from disappearing immediately after insertion.

Lastly, if requirement 1 above holds but neither 2a, 2b nor 2c hold, then code similar to the following should be used for the table component's AfterPost event handler.

```
procedure TForm1.Table1AfterPost(DataSet: TDataset);  
begin  
    Table1.Refresh  
end;
```

A Refresh of the table is necessary to re-retrieve the values changed by the server.

If criteria 2a, 2b or 2c cannot be avoided, then the table should be updated without using Delphi's data-aware controls. This can be accomplished using a TQuery component targeted at the same table. Once the query has posted the update, any table components using the same table should be Refreshed.

Reference:

7/16/98 4:34:02 PM

Last Modified: 01-SEP-99

## Article #16129: How to Populate a TDBComboBox Or TDBListBox

Technical Information Database

TI1129D.txt How to Populate a TDBComboBox Or TDBListBox

Category :Database Programming

Platform :All

Product :Delphi All

Description:

Filling dblistboxs and dbcomboboxs

Most of Delphi's data aware components will populate themselves after they are wired up to a open dataset. However DbListboxs and DbComboboxs do not display this characteristic. These two components are not for displaying your datasets, but filling them. Use of these components is straight forward. When you update your table, the value of the DbListbox or DbCombobox will be posted in the appropriate field.

Filling the DbCombobox or DbListbox the same as filling normal comboboxs or listboxes. The lines of text in a listbox or combobox are really a tstring list. The "Items" property of the given component holds this list. Use the "Add" method for adding items to a tstring. If you want to use data types other than strings they must be converted at run time. If your list has a blank line at the end, consider setting the "IntegralHeight" property to True.

Filling a DbListbox with 4 lines programmatically might look similar to this:

```
DbListbox1.items.add('line one');  
DbListbox1.items.add('line two');  
DbListbox1.items.add('line three');  
DbListbox1.items.add('line four');
```

Filling a DbListbox at design time requires using the object inspector. By double clicking on the components "Items" property, you can bring up the "String List Editor" and input the desired rows.

Unfortunately, if a combobox is filled this way, there is not default value. Setting a DbComboboxs "text" property will achieve this result. (the "text" property is not available in the object inspector, so it must be set programmatically). Setting the default value to the first value in the DbCombobox's list looks like this:

```
DbCombobox1.text := DbCombobox1.items[0];
```

Often it is useful to fill a DBListBox from a dataset. This can be done using loop:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with table2 do begin
    open;
    while not EOF do
    begin
      DBlistbox1.items.add(FieldByName('name').AsString);
    next;
    end;
  end;
end;
```

Reference:

7/16/98 4:34:02 PM

Last Modified: 01-SEP-99

## Article #16135: New Language Features in Delphi 2.0

Technical Information Database

TI1135D.txt - New Language Features in Delphi 2.0

Category :Delphi 2.x  
Platform :All Windows  
Product :Delphi2.x,

### Description:

New Language Features in Delphi 2.0 - 32

Delphi 2.0 defines several new data types that exploit the features available by Windows 95. Delphi 2.0 has also changed a few data types to take advantage of the 32-bit environment.

New data types include:

- Character type
- String type
- Variant type
- Currency type

Changed data types:

- Integer
- Cardinal

Additional Syntax:

- Unit finalization section

New Data Types

--- ----

### Character Type

Delphi 2.0 introduces new wide character types to support Unicode. Delphi 1.0 treated characters as 8-bit values of type Char.

These are the standard types that represent characters in Delphi 2.0.

`ANSIChar` - A standard 8-bit ANSI character, equivalent to the `Char` type in previous versions of Delphi.

`WideChar` - A 16-bit character, representing a Unicode character. If the high-order byte is zero, the low-order byte contains an ANSI character.

`Char` - By default, `Char` is equivalent to `ANSIChar`. `Char` works in the same way as the implementation-dependent Integer type, which is equivalent to `SmallInt` in 16-bit versions of Delphi and to `LongInt` in 32-bit versions of Delphi. In Delphi 2.0, `Char` defaults to an 8-bit value.

Character-pointer types:

Pointer type	Character type
<code>PANSIChar</code>	<code>ANSIChar</code>
<code>PWideChar</code>	<code>WideChar</code>
<code>PChar</code>	<code>Char</code>

The semantics of all the character-pointer types are identical. The only thing that varies is the size of the character pointed to.

String Type

Delphi 2.0 supports strings of nearly unlimited length in addition to the 255-character counted strings previously supported. A new compiler directive, `$H`, controls whether the reserved word "string" represents a short string or the new, long string. The default state of `$H`, is `$H+`, using long strings by default. All Delphi 2.0 components use the new long string type.

These are the new string types.

`ShortString` - A counted string with a maximum length of 255 characters. Equivalent to `String` in Delphi 1.0. Each element is of type `ANSIChar`.

AnsiString - A new-style string of variable length, also called a "long string." Each element is of type ANSIChar.

string - Either a short string or an ANSI string, depending on the value of the \$H compiler directive.

Here are the compatibility issues.

Although most string code works interchangeably between short strings and long strings, there are certain short-string operations that either won't work on long strings at all or which operate more efficiently when done a different way. The following table summarizes these changes.

Short String operation	Long string equivalent	Explanation
PString type	string	All long strings are dynamically allocated, so PString is redundant and requires more bookkeeping.
S[0] := L	SetLength(S,L) SetString(S,P,L)	Because long strings are dynamically allocated, you must call the SetLength procedure to allocate the appropriate amount of memory.
StrPCopy (Buffer, S)	StrPCopy(Buffer, PChar(S))	You can typecast long strings into null-terminated strings. The address of the long string is the address of its first character, and the long string is followed by a null.
S := StrPas(P)	S := P	Long strings can automatically copy from null-terminated strings.

Long strings cannot be passed to OpenString-type parameters or var short-string parameters.

Variant Type

Delphi 2.0 introduces variant types to give you the flexibility



to dynamically change the type of a variable. This is useful when implementing OLE automation or certain kinds of database operations where the parameter types on the server are unknown to your Delphi-built client application.

A variant type is a 16-byte structure that has type information embedded in it along with its value, which can represent a string, integer, or floating-point value. The compiler recognizes the standard type identifier Variant as the declaration of a variant.

In cases where the type of a variant is incompatible with the type needed to complete an operation, the variant will automatically promote its value to a compatible value, if possible. For instance, if a variant contains an integer and you assign it to a string, the variant converts its value into the string representing the integer number, which is then assigned to the string.

You can also assign a variant expression to a variable of a standard type or pass the variant as a parameter to a routine that expects a standard type as a parameter. Delphi coerces the variant value into a compatible type if necessary, and raises an exception if it cannot create a compatible value.

## Currency Type

Delphi 2.0 defines a new type called Currency, which is a floating-point type specifically designed to handle large values with great precision. Currency is assignment-compatible with all other floating-point types (and variant types), but is actually stored in a 64-bit integer value much like the Comp type.

Currency-type values have a four-decimal-place precision. That is, the floating-point value is stored in the integer format with the four least significant digits implicitly representing four decimal places.

## Changed Data Types

-----

The implementation-dependent types Integer and Cardinal are 32-bit values in Delphi 2.0, where they were 16-bit values in Delphi 1.0. To explicitly declare 16-bit integer data types,

use the SmallInt and Word types.

## Additional Syntax

-----

You can include an optional finalization section in a unit. Finalization is the counterpart of initialization, and takes place when the application shuts down. You can think of the finalization section as "exit code" for a unit. The finalization section corresponds to calls to ExitProc and AddExitProc in Delphi 1.0.

The finalization begins with the reserved word finalization. The finalization section must appear after the initialization section, but before the final end. statement.

Once execution enters an initialization section of a unit, the corresponding finalization section is guaranteed to execute when the application shuts down. Finalization sections must handle partially-initialized data properly, just as class destructors must.

Finalization sections execute in the opposite order that units were initialized. For example, if your application initializes units A, B, and C, in that order, it will finalize them in the order C, B, and A.

The outline for a unit therefore looks like this:

```
unit UnitName;
interface
{ uses clause; optional }
...
implementation
{ uses clause; optional }
...
initialization { optional }
...
finalization { optional }
...
end.
```

Reference:

3/30/99 1:01:56 PM

Last Modified: 01-SEP-99

## Article #16141: Preventing a Form from Resizing

Technical Information Database

TI1141D.txt Preventing a Form from Resizing

Category :VCL

Platform :All

Product :Delphi All

### Description:

In some cases, developers would want to create a regular window (Form) in Delphi that contains some of the characteristics of a dialog box. For example, they do not want to allow their users to resize the form at runtime due to user interface design issues. Other than creating the whole form as a dialog box, there is not a property or a method to handle this in a regular window in Delphi. But due to the solid connection between Delphi and the API layer, developers can accomplish this easily.

The following example demonstrates a way of handling the Windows message "WM\_GetMinMaxInfo" which allows the developer to restrict the size of windows (forms) at runtime to a specific value. In this case, it will be used to disable the functionality of sizing the window (form) at runtime.

Consider the following unit:

```
unit getminmax;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs;
```

```
type
```

```
TForm1 = class(TForm)
```

```
private
```

```
{ Private declarations }
```

```
procedure WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
```

```
message WM_GETMINMAXINFO;
```

```
procedure WMInitMenuPopup(var Msg: TWMInitMenuPopup);
```

```

        message WM_INITMENUPOPUP;
    procedure WMNCHitTest(var Msg: TWMNCHitTest);
        message WM_NCHitTest;
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
begin
    inherited;
    with Msg.MinMaxInfo^ do
    begin
        ptMinTrackSize.x:= form1.width;
        ptMaxTrackSize.x:= form1.width;
        ptMinTrackSize.y:= form1.height;
        ptMaxTrackSize.y:= form1.height;
    end;
end;

procedure TForm1.WMInitMenuPopup(var Msg: TWMInitMenuPopup);
begin
    inherited;
    if Msg.SystemMenu then
        EnableMenuItem(Msg.MenuPopup, SC_SIZE, MF_BYCOMMAND or
MF_GRAYED)
    end;
end;

procedure TForm1.WMNCHitTest(var Msg: TWMNCHitTest);
begin
    inherited;
    with Msg do
        if Result in [HTLEFT, HTRIGHT, HTBOTTOM, HTBOTTOMRIGHT,
HTBOTTOMLEFT, HTTOP, HTTOPRIGHT, HTTOPLEFT] then
            Result:= HTNOWHERE
        end;
end. { End of Unit}

```

A message handler for the windows message "WM\_GetMinMaxInfo" in

the code above was used to set the minimum and maximum TrackSize of the window to equal the width and height of the form at design time. That was actually enough to disable the resizing of the window (form), but the example went on to handle another couple of messages just to make the application look professional. The first message was the "WMInitMenuPopup" and that was to gray out the size option from the System Menu so that the application does not give the impression that this functionality is available. The second message was the "WMNCHitTest" and that was used to disable the change of the cursor icon whenever the mouse goes over one of the borders of the window (form) for the same reason which is not to give the impression that the resizing functionality is available.

Reference:

7/16/98 4:34:02 PM

Last Modified: 01-SEP-99

## Article #16147: SQL: Sorting on a Calculated Column

Technical Information Database

TI1147D.txt SQL: Sorting on a Calculated Column

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

At times, a given data schema will require that a data set will need to be ordered by the result of a calculation. In Delphi applications using SQL, this is possible, but the methodology varies slightly depending on the database type used.

For local SQL involving Paradox and dBASE tables, the calculated field would be given a name using the AS keyword. This allows the calculated field to be referenced for such purposes as setting a sort order with an ORDER BY clause in an SQL query. For example, using the sample table ITEMS.DB:

```
SELECT I."PARTNO", I."QTY", (I."QTY" * 100) AS TOTAL
FROM "ITEMS.DB" I
ORDER BY TOTAL
```

In this example, the calculated field is designated to be referred to as TOTAL, this column name then being available for the ORDER BY clause for this SQL statement.

The above method is not supported for InterBase. It is still possible, though, to sort on a calculated field in InterBase (IB) or the Local InterBase Server tables. Instead of using the name of the calculated field, an ordinal number representing the calculated field's position in field list is used in the ORDER BY clause. For example, using the sample table EMPLOYEE (in the EMPLOYEE.GDB database):

```
SELECT EMP_NO, SALARY, (SALARY / 12) AS MONTHLY
FROM EMPLOYEE
ORDER BY 3 DESCENDING
```

While IB or LIBS tables require this second method and cannot use the first method described, either of the two methods can be used with local

SQL. For example, using the SQL query for the Paradox table and adapting

it to use the relative position of the calculated field rather than the

name:

```
SELECT I."PARTNO", I."QTY", (I."QTY" * 100) AS TOTAL
FROM "ITEMS.DB" I
ORDER BY 3
```

Reference:

7/16/98 4:34:03 PM

Last Modified: 01-SEP-99



## Article #16153: SQL: Using the SUBSTRING Function

Technical Information Database

TI1153D.txt SQL: Using the SUBSTRING Function

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

The SQL function SUBSTRING can be used in Delphi applications that include local SQL queries, but is not supported for InterBase (IB) or the Local InterBase Server (LIBS) tables. What follows is the syntax for the SUBSTRING function, examples of its use in local SQL queries, and an alternative that will return the same results for IB/LIBS tables.

The syntax for the SUBSTRING function is:

```
SUBSTRING( FROM [, FOR ])
```

Where:

is the name of the column in the table from which the sub-string is to be extracted.

is the point in the column value from which the sub-string to be extracted will start.

is the length of the sub-string to be extracted.

Using these values, the use of the SUBSTRING function below would return the second, third, and fourth characters from a column named COMPANY:

```
SUBSTRING(COMPANY FROM 2 FOR 3)
```

The SUBSTRING function can be used either in the field list for a SELECT query or in the WHERE clause of a query to allow for comparing a value with a specific sub-set of a column. The SUBSTRING function can only

be

used with String type columns (the CHAR type in SQL parlance). Here is an example of the SUBSTRING function used in a columns list in a SELECT query (using the sample Paradox table CUSTOMER.DB):

```
SELECT (SUBSTRING(C."COMPANY" FROM 1 FOR 3)) AS SS
FROM "CUSTOMER.DB" C
```

This SQL query extracts the first three characters from the COMPANY column, returning them as the calculated column named SS. Now, an example

of the SUBSTRING function used in the WHERE clause of an SQL query (using the same sample table):

```
SELECT C."COMPANY"
FROM "CUSTOMER.DB" C
WHERE SUBSTRING(C."COMPANY" FROM 2 FOR 2) = "an"
```

This query returns all rows from the table where the second and third characters in the COMPANY column are "ar".

As the SUBSTRING function is not supported at all by IB or LIBS databases, it is not possible to have a sub-string operation in the column list of a query (exception: IB can do sub-strings via User-Defined Functions). But through use of the LIKE operator and the accompanying character substitution marker, it is possible to effect a sub-string in a WHERE clause. For example, using the sample table EMPLOYEE (in the EMPLOYEE.GDB database):

```
SELECT LAST_NAME, FIRST_NAME
FROM EMPLOYEE
WHERE LAST_NAME LIKE "_an%"
```

This SQL query would return all rows in the table where the second and third characters of the LAST\_NAME column are "an", similar to the previous example for the Paradox table. While IB and LIBS databases would require this method for performing sub-string comparisons in the WHERE clause of a query and cannot use the SUBSTRING function, Paradox and dBASE tables (i.e., local SQL) can use either method.

Reference :

7/16/98 4:34:03 PM

Last Modified: 01-SEP-99

## Article #16159: SQL: Summarizing a Calculated Column

Technical Information Database

TI1159D.txt SQL: Summarizing a Calculated Column

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

Occasionally in a Delphi application that uses SQL to access data, it becomes necessary to summarize calculated data. That is, to create a calculated column and apply the SUM function to it.

When performing this operation against SQL tables (such as those for the

Local InterBase Server), it is a simple matter of enclosing the calculation within the SUM function. For example, using the sample table

EMPLOYEE (in the EMPLOYEE.GDB database):

```
SELECT SUM(SALARY / 12)
FROM EMPLOYEE
```

This same methodology can also be used when the returned data set is to be

grouped by the value in another column with a GROUP BY clause:

```
SELECT EMP_NO, SUM(SALARY / 12)
FROM EMPLOYEE
GROUP BY EMP_NO
ORDER BY EMP_NO
```

While SQL databases support the summarization of calculated columns, local SQL will not. Other means would be needed to obtain the results, such as copying the results of a query with a calculated column to a temporary table (as with a TBatchMove component) and then using a TQuery component to summarize the data in the temporary table.

Reference:

7/16/98 4:34:03 PM

Last Modified: 01-SEP-99

## Article #16165: Managing Data Segment Size

Technical Information Database

TI1165D.txt - Managing Data Segment Size

Category :Delphi 1.x  
Platform :All-32Bit  
Product :

Description:  
Managing Data Segment Size

The "Data Segment too large" error occurs in Delphi 16 bit applications when the size of static data, stack and local heap are larger than the 64k application limit imposed by Windows. This document discusses how to identify and measure portions of your code that consume memory in the data segment, and how to manage this limited resource.

What does the data segment consist of?

-----  
Task header: 16 bytes of various Windows system information  
Static data: Contains global variables and typed constants.  
Stack: Stores local variables allocated by procedures and functions. The default stack size is 16k and can be modified in the Options|Project|Linker page.  
Local heap: Used by Windows for temporary storage and defaults to 8k. Do not set the local heap to 0. Windows may expand this area of memory if necessary.

How do I measure total data segment size?

-----  
To get the size of a 16 bit Delphi application static data, stack and local heap for a project, compile the project and select the Delphi menu item Compile|Information. The dialog will list the following, given a new project with one form:

```
Source compile: 12 lines
Code size:      128981 bytes
Data size:     3636 bytes
Stack size:    16384 bytes
Local Heap size: 8192 bytes
```

A Delphi application starts with the overhead of static data declared in units that provide the applications initial functionality. If the only global variable is the form name, the application will still consume at least 3,636 bytes. Adding a second form increases the datasize to only 3640 -- an increase of only the size of the global variable needed to declare a second form.

```
var
  Form2: TForm2; { 4 byte pointer }
```

The total data segment size, (the sum of static data, stack and local heap) is 28,212 bytes:

```
Data size:      3,636
Stack size:     16,384
Local Heap size: 8,192
-----
                28,212
```

What portions of my project increase data size?

- ```
-----
-- Variables declared within the interface and implementation
   sections.
-- Typed constants declared anywhere within the application.
   An example of a typed constant declaration:
```

```
const
  MyConst: integer = 100;
```

Units declared in the Uses clause and components contain code that may have global variables or typed constants. For instance, a TBitBtn adds 180 bytes when added to a project, where at least 132 bytes are accounted for by typed constants and global variables within the Buttons.Pas unit. Adding 10 more TBitBtns to the project does not increase the project size beyond the initial 180 byte increase.

The following sample unit includes comments describing memory

usage:

```
unit Test;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,  
  Controls, Forms, Dialogs, StdCtrls;
```

```
{ Functions used from the units may have global variables  
  and typed constants that will increase the size of the  
  data segment. }
```

```
type
```

```
  { Class objects are stored on the global heap, not the data  
    segment }
```

```
TForm1 = class(TForm)
```

```
  Label1: TLabel;
```

```
  Button1: TButton;
```

```
  procedure Button1Click(Sender: TObject);
```

```
public
```

```
  { MyInteger and MyString are stored  
    on the global heap. }
```

```
  MyInteger: Integer;
```

```
  MyString: String;
```

```
end;
```

```
const
```

```
{ MyConst is a typed constant and is stored in  
  static data portion of the data segment.
```

```
  Minimize the number of typed constants. }
```

```
MyConst: integer = 23;
```

```
var
```

```
{ Form1 is a global variable - stored in the static  
  data portion of the data segment. You should minimize  
  the number and size of global variables. Form1 is  
  pointer and uses only four bytes. }
```

```
Form1: TForm1;
```

```
{ MyGlobalString will consume 256 bytes even if the  
  string is only a few characters long. }
```

```
MyGlobalString: string;
```



implementation

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
{ MyLocal is a local variable and is not stored
  in the data segment. }
  MyLocal: String;
begin
  MyLocal := 'Test application';
  Label1.Caption := MyLocal
end;

end.
```

What is the impact of components on the data size?

-----  
Here is a list of components from the Standard, Additional, Data Access, and Data Control (partial list) pages of the component palette. The list shows the compile size after added a single instance of the component to a new project. In descending order by data size usage:

| Component  | App<br>in<br>bytes | Bytes<br>Over<br>3,636 |
|------------|--------------------|------------------------|
| table      | 4272               | 636                    |
| batchmove  | 4272               | 636                    |
| storedproc | 4258               | 622                    |
| query      | 4250               | 614                    |
| database   | 4036               | 400                    |
| datasource | 3886               | 250                    |
| outline    | 3838               | 202                    |
| bitbtn     | 3816               | 180                    |
| stringgrid | 3794               | 158                    |
| drawgrid   | 3790               | 154                    |
| maskedit   | 3762               | 126                    |
| memo       | 3750               | 114                    |
| report     | 3722               | 86                     |
| listbox    | 3704               | 68                     |
| edit       | 3694               | 58                     |

|                |      |    |
|----------------|------|----|
| tabset         | 3692 | 56 |
| combobox       | 3674 | 38 |
| scrollbar      | 3654 | 18 |
| button         | 3652 | 16 |
| checkbox       | 3652 | 16 |
| radiobutton    | 3652 | 16 |
| radiogroup     | 3652 | 16 |
| panel          | 3650 | 14 |
| label          | 3648 | 12 |
| speedbutton    | 3646 | 10 |
| header         | 3644 | 8  |
| scrollbox      | 3644 | 8  |
| notebook       | 3638 | 2  |
| menu           | 3636 | 0  |
| groupbox       | 3636 | 0  |
| tabbednotebook | 3636 | 0  |
| image          | 3636 | 0  |
| shape          | 3636 | 0  |

How do I manage data segment size?

- 
- 1) Avoid declaring global variables or typed constants, particularly large arrays. Instead, declare a type and a pointer to that type. Then use memory management routines such as Getmem to allocate memory from the global heap.  
This reduces the resource hit on the data segment to the 4 bytes used by the pointer variable. See code sample below.
  - 2) Be aware of the impact of components. See above.
  - 3) If you have a large number of strings, or arrays of strings, allocate these dynamically. Note: strings default to 255 in length -- declare as a specific size where possible: (i.e. MyShortString: string[20]).
  - 4) A TStringList object can be used to create and manipulate large numbers of strings.
  - 5) The Pchar type, "pointer to a string", can be used to dynamically create and manipulate character strings using little data segment space. See online help under "String-handling routines (Null-terminated)".
  - 6) Information on memory issues is available in Object Pascal Language Guide, OBJLANG.ZIP and can be downloaded from Comuserve, DELPHI forum, and the World Wide Web, [www.borland.com/TechInfo/delphi/whatsnew/dwnloads.html](http://www.borland.com/TechInfo/delphi/whatsnew/dwnloads.html).

## Alternative to global declaration of a large structure

---

Here's an example that will waste 32 bytes of data segment, followed by a second example using only 4 bytes, and accomplishes essentially the same task.

1)

```
{ Declaring TMyStructure causes no change
  in the data segment size. }
TMyStructure = record
  Name: String[10];
  Data: array[0..9] of Integer;
end;

var
  Form1: TForm1;

{ MyStructure declaration causes a 32 byte
  increase:
  Mystructure pointer = 1 byte
  Name = 11 bytes (10 chars + length byte)
  Data = 20 bytes (10 * 2 bytes per integer)
}
MyStructure: TMyStructure;
```

2)

```
{ Declaring TMyStructure causes no change
  in the data segment size. }
PMyStructure = ^TMyStructure;
TMyStructure = record
  Name: String[10];
  Data: array[0..9] of Integer;
end;

var
  Form1: TForm1;
  { MyDataPtr causes 4 byte increase
    for the pointer variable. }
  MyDataPtr: PMyStructure;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { Here, resources are taken from the heap. }
  New(MyDataPtr);
  MyDataPtr.Name := 'Fred';
  MyDataPtr.array[0] := 560;
  Dispose(MyDataPtr);
end;
```

You can also put a variable declaration within a class:

```
type
  TMyBigArray = array[1..100] of string

  TForm1 = class(TForm)
  public
    { This declaration has no impact on data segment size. }
    MyBigArray: TMyBigArray;
  end;

var
  { This declaration increases data segment by 25,600 bytes. }
  MyOtherBigArray: TMyBigArray;
```

Reference:

4/22/99 12:56:29 PM

Last Modified: 01-SEP-99

## Article #16171: Validating input in TEdit components

Technical Information Database

TI1171D.txt Validating input in TEdit components

Category :VCL

Platform :All

Product :Delphi All

### Description:

Q: How can I validate input in my TEdit components?

A: Assuming you're using regular TEdit components, during OnExit, you will see irregular behavior from controls if you attempt to change focus at that time.

The solution is to post a message to your form in the TEdit's OnExit event handler. This user-defined posted message will indicate that the coast is clear to begin validating input. Since posted messages are placed at the end of the message queue, this gives Windows the opportunity to complete the focus change before you attempt to change the focus back to another control.

Attached is a unit and text representation of a DFM (form) file which demonstrates this technique.

```
{ *** BEGIN CODE FOR UNIT5.PAS *** }
unit Unit5;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Mask;

const
  { User-defined message }
  um_ValidateInput = wm_User + 100;

type
  TForm5 = class(TForm)
```

```

    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Button1: TButton;
    procedure EditExit(Sender: TObject);
    procedure EditEnter(Sender: TObject);
private
    Refocusing: TObject;
    { User-defined message handler }
    procedure ValidateInput(var M: TMessage); message
um_ValidateInput;
end;

var
    Form5: TForm5;

implementation

{$R *.DFM}

procedure TForm5.ValidateInput(var M: TMessage);
var
    E : TEdit;
begin
    { The following line is my validation. I want to make sure }
    { the first character is a lower case alpha character. Note }
    { the typecast of lParam to a TEdit. }
    E := TEdit(M.lParam);
    if not (E.Text[1] in ['a'..'z']) then begin
        Refocusing := E;           { Avoid a loop      }
        ShowMessage('Bad input');  { Yell at the user }
        TEdit(E).SetFocus;        { Set focus back  }
    end;
end;

procedure TForm5.EditExit(Sender: TObject);
begin
    { Post a message to myself which indicates it's time to }
    { validate the input. Pass the TEdit instance (Self) as }
    { the message lParam. }
    if Refocusing = nil then
        PostMessage(Handle, um_ValidateInput, 0, longint(Sender));
end;

```

```
procedure TForm5.EditEnter(Sender: TObject);
begin
if Refocusing = Sender then
  Refocusing := nil;
end;

end.
{ *** END CODE FOR UNIT5.PAS *** }

{ *** BEGIN CODE FOR UNIT5.DFM *** }
object Form5: TForm5
  Left = 489
  Top = 303
  Width = 318
  Height = 205
  Caption = 'Form5'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 16
  object Edit1: TEdit
    Left = 32
    Top = 32
    Width = 121
    Height = 24
    TabOrder = 0
    Text = 'Edit1'
    OnEnter = EditEnter
    OnExit = EditExit
  end
  object Edit2: TEdit
    Left = 160
    Top = 32
    Width = 121
    Height = 24
    TabOrder = 1
    Text = 'Edit2'
    OnEnter = EditEnter
    OnExit = EditExit
  end
  object Edit3: TEdit
    Left = 32
    Top = 64
```

```
Width = 121
Height = 24
TabOrder = 2
Text = 'Edit3'
OnEnter = EditEnter
OnExit = EditExit
end
object Edit4: TEdit
Left = 160
Top = 64
Width = 121
Height = 24
TabOrder = 3
Text = 'Edit4'
OnEnter = EditEnter
OnExit = EditExit
end
object Button1: TButton
Left = 112
Top = 136
Width = 89
Height = 33
Caption = 'Button1'
TabOrder = 4
end
end
{ *** END CODE FOR UNIT5.DFM *** }
```

Reference:

7/16/98 4:34:03 PM

Last Modified: 01-SEP-99



## Article #16177: DDE: A simple example

Technical Information Database

TI1177D.txt - DDE: A simple example

Category :Delphi 1.x  
Platform :All Windows  
Product :

Description:

Q: How can I do DDE under Delphi using API calls ?

A: Its fairly easy to accomplish, following is an example of how to have a client program talk with a server program. Both are completely written in Delphi. In total there are 2 projects, 3 forms, and 3 units. This demo uses DDE ML API methods to handle the DDE requests.

The server must be running before the client will load. This demo program shows 3 different ways data can be moved between a client and a server.

1. The Client can 'POKE' data to the server.
2. The Server can automatically pass data to the Client and the Client will update a graph based on the results from the Server.
3. The Server's Data changes, then the Client will make a request to the Server for the new data, then update the graph.

\*\*\*\*\* How to handle the program. \*\*\*\*\*

Following are 8 files concatenated together. Each one has a { \*\*\* BEGIN CODE FOR FILENAME.EXT \*\*\* } CODE { \*\*\* END CODE FOR FILENAME.EXT \*\*\* } take each block of code BETWEEN THE { \*\*\* } lines and place in a file of the corresponding name, then compile and have fun !!!!

```

{ *** BEGIN CODE FOR DDEMLCLI.DPR *** }
program Ddemlcli;

uses
  Forms,
  Ddemlclu in 'DDEMLCLU.PAS' {Form1};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
{ *** END CODE FOR DDEMLCLI.DPR *** }

{ *** BEGIN CODE FOR DDEMLCLU.DFM *** }
object Form1: TForm1
  Left = 197
  Top = 95
  Width = 413
  Height = 287
  HorzScrollBar.Visible = False
  VertScrollBar.Visible = False
  Caption = 'DDEML Demo, Client Application'
  Font.Color = clWindowText
  Font.Height = -13
  Font.Name = 'System'
  Font.Style = []
  Menu = MainMenu1
  PixelsPerInch = 96
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  OnShow = FormShow
  TextHeight = 16
  object PaintBox1: TPaintBox
    Left = 0
    Top = 0
    Width = 405
    Height = 241
    Align = alClient
    Color = clWhite
    ParentColor = False
  end
end

```

```

    OnPaint = PaintBox1Paint
end
object MainMenu1: TMainMenu
  Top = 208
  object File1: TMenuItem
    Caption = '&File'
    object exit1: TMenuItem
      Caption = 'E&xit'
      OnClick = exit1Click
    end
  end
end
object DDE1: TMenuItem
  Caption = '&DDE'
  object RequestUpdate1: TMenuItem
    Caption = '&Request an Update'
    OnClick = RequestUpdate1Click
  end
  object AdviseofChanges1: TMenuItem
    Caption = '&Advise of Changes'
    OnClick = AdviseofChanges1Click
  end
  object N1: TMenuItem
    Caption = '-'
  end
  object PokeSomeData: TMenuItem
    Caption = '&Poke Some Data'
    OnClick = PokeSomeDataClick
  end
end
end
end
end
{ ***      END CODE FOR DDEMLCLU.DFM *** }

{ *** BEGIN CODE FOR DDEMLCLU.PAS *** }
{ **** }
{ }
{ Delphi 1.0 DDEML Demonstration Program }
{ Copyright (c) 1996 by Borland International }
{ }
{ **** }

{ This is a sample application demonstrating the use of the DDEML
APIs in
  a client application. It uses the DataEntry server application that
  is part of this demo in order to maintain a display of the entered

```

data  
as a bar graph.

You must run the server application first (in DDEMLSRV.PAS), and then run this client. If the server is not running, this application will fail trying to connect.

The interface to the server is defined by the list of names (Service, Topic, and Items) in the separate unit called DataEntry (DATAENTR.TPU).

The server makes the Items available in cf\_Text format; they are converted and stored locally as integers.

```
unit Ddemlclu;
```

```
interface
```

```
uses
```

```
SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
Forms, Dialogs, VBXCtrl, ExtCtrls, DDEML, Menus, StdCtrls;
```

```
const
```

```
NumValues = 3;
```

```
type
```

```
{ Data Structure which constitutes a sample }  
TDataSample = array [1..NumValues] of Integer;  
TDataString = array [0..20] of Char;      { Size of Item as text }
```

```
{ Main Form }  
TForm1 = class(TForm)  
  MainMenu: TMainMenu;  
  File1: TMenuItem;  
  exit1: TMenuItem;  
  DDE1: TMenuItem;  
  RequestUpdate1: TMenuItem;  
  AdviseofChanges1: TMenuItem;  
  PokeSomeData: TMenuItem;  
  N1: TMenuItem;
```

```

PaintBox1: TPaintBox;
procedure FormCreate(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure RequestUpdate1Click(Sender: TObject);
procedure FormShow(Sender: TObject);
procedure AdviseofChanges1Click(Sender: TObject);
procedure PokeSomeDataClick(Sender: TObject);

procedure Request(HConversation: HConv);
procedure exit1Click(Sender: TObject);
procedure PaintBox1Paint(Sender: TObject);

```

```

private
  { Private declarations }
public
  Inst: Longint;
  CallbackPtr: ^TCallback;
  ServiceHSz : HSz;
  TopicHSz   : HSz;
  ItemHSz    : array [1..NumValues] of HSz;
  ConvHdl    : HConv;

  DataSample : TDataSample;
end;

```

```
var Form1: TForm1;
```

```
implementation
```

```

const
  DataEntryName : PChar = 'DataEntry';
  DataTopicName : PChar = 'SampledData';
  DataItemNames : array [1..NumValues] of pChar = ('DataItem1',
  'DataItem2',
  'DataItem3');

```

```
{ $R *.DFM }
```

```
{ Local Function: Callback Procedure for DDEML }
```

```

function CallbackProc(CallType, Fmt: Word; Conv: HConv; hsz1, hsz2:
HSZ;
  Data: HDDEData; Data1, Data2: Longint): HDDEData; export;
begin
  CallbackProc := 0;    { See if proved otherwise }

```

```

case CallType of
  xtyp_Register:
    begin
      { Nothing ... Just return 0 }
    end;
  xtyp_Unregister:
    begin
      { Nothing ... Just return 0 }
    end;
  xtyp_xAct_Complete:
    begin
      { Nothing ... Just return 0 }
    end;
  xtyp_Request, Xtyp_AdvData:
    begin
      Form1.Request(Conv);
      CallbackProc := dde_FAck;
    end;
  xtyp_Disconnect:
    begin
      ShowMessage('Disconnected!');
      Form1.Close;
    end;
end;
end;

```

{ Posts a DDE request to obtain cf\_Text data from the server.

Requests

the data for all fields of the DataSample, and invalidates the window

to cause the new data to be displayed. Obtains the data from the Server synchronously, using DdeClientTransaction.

}

```
procedure TForm1.Request(HConversation: HConv);
```

```
var
```

```
  hDdeTemp : HDDEData;
```

```
  DataStr : TDataString;
```

```
  Err, I : Integer;
```

```
begin
```

```
  if HConversation <> 0 then begin
```

```
    for I := Low(ItemHSz) to High(ItemHSz) do begin
```

```
      hDdeTemp := DdeClientTransaction(nil, 0, HConversation, ItemHSz
```

```
[I],
```

```
      cf_Text, xtyp_Request, 0, nil);
```

```

    if hDdeTemp <> 0 then begin
        DdeGetData(hDdeTemp,
DataStr, SizeOf(DataStr), 0);
        Val(DataStr, DataSample[I], Err);
    end; { if }
end; { for }
Paintbox1.Refresh; { Redisplay the Screen }
end; { if }
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
var
    I : Integer;
{ Constructs an instance of the DDE Client Window. Constructs the
  window using the inherited constructor, then initializes the
  instance
  data.
}

```

```

begin
    Inst          := 0;          { Must be zero for first call to
DdeInitialize }
    CallbackPtr := nil;        { MakeProcInstance is called in
SetupWindow    }
    ConvHdl      := 0;
    ServiceHSz  := 0;
    TopicHSz    := 0;
    for I := Low(DataSample) to High(DataSample) do begin
        ItemHSz[I] := 0;
        DataSample[I] := 0;
    end;
end;

```

```

procedure TForm1.FormDestroy(Sender: TObject);
{ Destroys an instance of the Client window. Frees the DDE string
  handles, and frees the callback proc instance if they exist. Also
  calls DdeUninitialize to terminate the conversation. Then calls on
  the ancestral destructor to finish the job.
}

```

```

var I : Integer;
begin
    if ServiceHSz <> 0 then
        DdeFreeStringHandle(Inst, ServiceHSz);
    if TopicHSz <> 0 then
        DdeFreeStringHandle(Inst, TopicHSz);

```

```

for I := Low(ItemHSz) to High(ItemHSz) do
  if ItemHSz[I] <> 0 then
    DdeFreeStringHandle(Inst, ItemHSz[I]);

if Inst <> 0 then
  DdeUninitialize(Inst);    { Ignore the return value }

if CallbackPtr <> nil then
  FreeProcInstance(CallbackPtr);
end;

procedure TForm1.RequestUpdate1Click(Sender: TObject);
begin
{ Generate a DDE Request in response to the DDE | Request menu
selection.}
  Request(ConvHdl);
end;

procedure TForm1.FormShow(Sender: TObject);
{ Completes the initialization of the DDE Server Window.  Performs
those
actions which require a valid window.  Initializes the use of the
DDEML.
}
var
  I      : Integer;
  InitOK: Boolean;
begin
  CallbackPtr := MakeProcInstance(
CallbackProc, HInstance);

{ Initialize the DDE and setup the callback function.  If server is not
present, call will fail.
}
  if CallbackPtr <> nil then begin
    if DdeInitialize(Inst, TCallback(CallbackPtr), AppCmd_ClientOnly,
0) = dmlErr_No_Error then begin
      ServiceHSz:= DdeCreateStringHandle(Inst, DataEntryName,
cp_WinAnsi);
      TopicHSz  := DdeCreateStringHandle(Inst, DataTopicName,
cp_WinAnsi);
      InitOK := True;
{
  for I := Low(DataItemNames) to High(DataItemNames) do begin }
  for I := 1 to NumValues do begin
    ItemHSz[I]:= DdeCreateStringHandle(Inst, DataItemNames[I],

```



```

        cp_WinAnsi);
    InitOK := InitOK and (ItemHSz[I] <> 0);
end;

if (ServiceHSz <> 0) and (TopicHSz <> 0) and InitOK then begin
    ConvHdl := DdeConnect(Inst, ServiceHSz, TopicHSz, nil);
    if ConvHdl = 0 then begin
        ShowMessage('Can not start Conversation!');
        Close;
    end
end
else begin
    ShowMessage('Can not create Strings!');
    Close;
end
end
else begin
    ShowMessage('Can not Initialie!');
    Close;
end;
end;
end;
end;

```

```

procedure TForm1.AdviseofChanges1Click(Sender: TObject);
{ Toggles the state of the DDE Advise setting in response to the
  DDE | Advise menu selection. When this is selected, all three
  Items are set for Advising.
}
var
    I: Integer;
    TransType: Word;
    TempResult: Longint;
begin
    with TMenuItem(Sender) do begin
        Checked := not Checked;
        if Checked then
            TransType:= (xtyp_AdvStart or xtypf_AckReq)
        else
            TransType:= xtyp_AdvStop;
    end; { with }

    for I := Low(ItemHSz) to High(ItemHSz) do
        if DdeClientTransaction(nil, 0, ConvHdl, ItemHSz[I], cf_Text,
            TransType, 1000,
            TempResult) = 0 then

```

```
ShowMessage('Can not perform Advise Transaction');
```

```
if TransType and xtyp_AdvStart <> 0 then Request(ConvHdl);  
end;
```

```
procedure TForm1.PokeSomeDataClick(Sender: TObject);
```

```
{ Generates a DDE Poke transaction in response to the DDE | Poke  
  menu selection. Requests a value from the user that will be  
  poked into DataItem1 as an illustration of the Poke function.  
}
```

```
var
```

```
  DataStr: pChar;
```

```
  S: String;
```

```
begin
```

```
  S := '0';
```

```
  if InputQuery('PokeData', 'Enter Value to Poke', S) then begin
```

```
    S := S + #0;
```

```
    DataStr :=
```

```
S[1];
```

```
    DdeClientTransaction(DataStr, StrLen(DataStr) + 1, ConvHdl,  
      ItemHSz[1], cf_Text, xtyp_Poke, 1000, nil);
```

```
    Request(ConvHdl);
```

```
  end;
```

```
end;
```

```
procedure TForm1.exit1Click(Sender: TObject);
```

```
begin
```

```
  close;
```

```
end;
```

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
```

```
{ Repaints the window on request. Plots a graph of the current sales  
  volume.  
}
```

```
}
```

```
const
```

```
  LMargin = 30;    { Left Margin of graph }
```

```
var
```

```
  I,
```

```
  Norm: Integer;
```

```
  Wd: Integer;
```

```
  Step : Integer;
```

```
  ARect: TRect;
```

```
begin
```

```

Norm := 0;
for I := Low(DataSample) to High(DataSample) do begin
  if abs(DataSample[I]) > Norm then
    Norm := abs(DataSample[I]);
  end; { for }

if Norm = 0 then Norm := 1; { Just in case we have all zeros }

with TPaintBox(Sender).Canvas do begin
  { Paint Background }
  Brush.color := clWhite;
  FillRect(ClipRect);

  { Draw Axis }
  MoveTo(0, ClipRect.Bottom div 2);
  LineTo(ClipRect.Right, ClipRect.Bottom div 2);

  MoveTo(LMarg, 0);
  LineTo(LMarg, ClipRect.Bottom);

  { Print Left MArgin Text }
  TextOut(0,0, IntToStr(Norm));
  TextOut(0, ClipRect.Bottom div 2, '0');
  TextOut(0, ClipRect.Bottom + Font.Height, IntToStr(-Norm));

  TextOut(0, ClipRect.Bottom div 2, '0');
  TextOut(0, ClipRect.Bottom div 2, '0');
  TextOut(0, ClipRect.Bottom div 2, '0');
  { Print X Axis Text }

  { Now draw the bars based on that Normalized value. Compute
the
width of the bars so that all will fit in the window, and
compute an inter-bar space that is approximately 20% of the
width of a bar.
}
  {
  SelectObject(PaintDC, CreateSolidBrush(RGB(255, 0, 0)));
  SetBkMode(PaintDC, Transparent);
}

ARect := ClipRect;
Wd := (ARect.Right - LMarg) div NumValues;
Step := Wd div 5;
Wd := Wd - Step;
with ARect do begin
  Left := LMarg + (Step div 2);

```

```
Top := ClipRect.Bottom div 2;
end; { with }
```

```
{ Display Bars and X-Axis Text }
```

```
For i := Low(DataSample) to High(DataSample) do begin
  with ARect do begin
    Right := Left + Wd;
    Bottom := Top - Round((Top-5) * (DataSample[I] / Norm));
    end; { with }
    { Fill Bar }
    Brush.color := clFuchsia;
    FillRect(ARect);
    { Display Text - Horizontal Axis }
    Brush.color := clWhite;
    TextOut(ARect.Left, ClipRect.Bottom div 2 - Font.Height,
      StrPas(DataItemNames[i]));
    with ARect do
      Left := Left + Wd + Step;
    end; { for }
  end; { with }
end;
end. { *** END CODE FOR DDEMLCLU.PAS *** }
```

```
{ *** BEGIN CODE FOR DDEMLSVR.DPR *** }
program Ddemlsvr;
```

```
uses
```

```
  Forms,
  Ddesvru in 'DDESVRU.PAS' {Form1},
  Ddedlg in '\DELPHI\BIN\DDEDLG.PAS' {DataEntry};
```

```
{ $R *.RES }
```

```
begin
```

```
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TDataEntry, DataEntry);
  Application.Run;
```

```
end.
```

```
{ *** END CODE FOR DDEMLSVR.DPR *** }
```

```
{ *** BEGIN CODE FOR DDESVRU.DFM *** }
```

```
object Form1: TForm1
  Left = 712
```

```
Top = 98
Width = 307
Height = 162
Caption = 'DDEML Demo, Serve Application'
Color = clWhite
Font.Color = clWindowText
Font.Height = -13
Font.Name = 'System'
Font.Style = []
Menu = MainMenu1
PixelsPerInch = 96
OnCreate = FormCreate
OnDestroy = FormDestroy
OnShow = FormShow
TextHeight = 16
object Label1: TLabel
    Left = 0
    Top = 0
    Width = 99
    Height = 16
    Caption = 'Current Values:'
end
object Label2: TLabel
    Left = 16
    Top = 24
    Width = 74
    Height = 16
    Caption = 'Data Item1:'
end
object Label3: TLabel
    Left = 16
    Top = 40
    Width = 74
    Height = 16
    Caption = 'Data Item2:'
end
object Label4: TLabel
    Left = 16
    Top = 56
    Width = 74
    Height = 16
    Caption = 'Data Item3:'
end
object Label5: TLabel
    Left = 0
```

```
Top = 88
Width = 265
Height = 16
Caption = 'Select Data|Enter Data to change values.'
end
object Label6: TLabel
  Left = 96
  Top = 24
  Width = 8
  Height = 16
  Caption = '0'
end
object Label7: TLabel
  Left = 96
  Top = 40
  Width = 8
  Height = 16
  Caption = '0'
end
object Label8: TLabel
  Left = 96
  Top = 56
  Width = 8
  Height = 16
  Caption = '0'
end
object MainMenu1: TMainMenu
  Left = 352
  Top = 24
  object File1: TMenuItem
    Caption = '&File'
    object Exit1: TMenuItem
      Caption = '&Exit'
      OnClick = Exit1Click
    end
  end
  object Data1: TMenuItem
    Caption = '&Data'
    object EnterData1: TMenuItem
      Caption = '&Enter Data'
      OnClick = EnterData1Click
    end
  end
  object Clear1: TMenuItem
    Caption = '&Clear'
    OnClick = Clear1Click
  end
end
```

```

    end
  end
end
end
end
{ ***      END CODE FOR DDESVRU.DFM *** }

{ *** BEGIN CODE FOR DDESVRU.PAS *** }
{*****}
{
{      Delphi 1.0 DDEML Demonstration Program      }
{      Copyright (c) 1996 by Borland International  }
{
{*****}

{ This sample application uses the DDEML library in the server side
of a
  cooperative application.  This server is a simple data-entry
application
  which allows an operator to enter three data items, which are made
available through DDE to interested clients.

This server makes its service available under the following names:

    Service: 'DataEntry'
    Topic   : 'SampledData'
    Items   : 'DataItem1', 'DataItem2', 'DataItem3'

Conceivably, other topics under this service could be defined.
Things
  such as historical data, information about the sampling, and so on
might make useful topics.

You must run this server BEFORE running the client (DDEMLCLI.PAS),
or
  the client will fail the connection.

The interface to this server is defined by the list of names
(Service,
  Topic, and Items) in the separate unit called DataEntry (DATAENTR.
TPU).
  The server makes the Items available in cf_Text format; they can be
converted and stored locally as integers by the client.
}
unit Ddesvru;

```

```
interface
```

```
uses
```

```
SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
Forms, Dialogs, StdCtrls, Menus,
```

```
DDEML,      { DDE API }  
ShellApi;
```

```
const
```

```
NumValues = 3;  
DataItemNames : array [1..NumValues] of PChar = ('DataItem1',  
  'DataItem2',  
  'DataItem3');
```

```
type
```

```
TDataString = array [0..20] of Char;      { Size of Item as text }  
TDataSample = array [1..NumValues] of Integer;
```

```
{type
```

```
{ Data Structure which constitutes a sample }  
{ TDataSample = array [1..NumValues] of Integer;  
{ TDataString = array [0..20] of Char;      { Size of Item as text }
```

```
const
```

```
DataEntryName: PChar = 'DataEntry';  
DataTopicName: PChar = 'SampledData';
```

```
type
```

```
TForm1 = class(TForm)  
  MainMenu1: TMainMenu;  
  File1: TMenuItem;  
  Exit1: TMenuItem;  
  Data1: TMenuItem;  
  EnterData1: TMenuItem;  
  Clear1: TMenuItem;  
  Label1: TLabel;  
  Label2: TLabel;  
  Label3: TLabel;  
  Label4: TLabel;  
  Label5: TLabel;  
  Label6: TLabel;  
  Label7: TLabel;  
  Label8: TLabel;  
  procedure Exit1Click(Sender: TObject);
```



```

function MatchTopicAndService(Topic, Service: HSz): Boolean;
function MatchTopicAndItem(Topic, Item: HSz): Integer;
function WildConnect(Topic, Service: HSz; ClipFmt: Word):
HDDEData;
function AcceptPoke(Item: HSz; ClipFmt: Word;
  Data: HDDEData): Boolean;
function DataRequested(TransType: Word; ItemNum: Integer;
  ClipFmt: Word): HDDEData;
procedure FormCreate(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure FormShow(Sender: TObject);
procedure EnterData1Click(Sender: TObject);
procedure Clear1Click(Sender: TObject);

private
  Inst      : Longint;
  Callback  : TCallback;
  ServiceHSz : HSz;
  TopicHSz  : HSz;
  ItemHSz   : array [1..NumValues] of HSz;
  ConvHdl   : HConv;
  Advising  : array [1..NumValues] of Boolean;

  DataSample : TDataSample;

public
  { Public declarations }
end;

var
  Form1: TForm1;

implementation
uses DDEDlg; { DataEntry Form }

{$R *.DFM}

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;
{ Initialized globals }

const

```

```
DemoTitle    : PChar = 'DDEML Demo, Server Application';
```

```
MaxAdvisories = 100;
```

```
NumAdvLoops  : Integer = 0;
```

```
{ Local Function: Callback Procedure for DDEML }
```

```
{ This callback procedure responds to all transactions generated by  
the
```

```
DDEML. The target Window object is obtained from the stored global,  
and the appropriate methods within that objects are used to respond  
to the given transaction, as indicated by the CallType parameter.
```

```
}
```

```
function CallbackProc(CallType, Fmt: Word; Conv: HConv; HSz1, HSz2:  
HSZ;
```

```
Data: HDDEData; Data1, Data2: Longint): HDDEData; export;
```

```
var
```

```
ItemNum      : Integer;
```

```
begin
```

```
CallbackProc := 0;      { See if proved otherwise }
```

```
case CallType of
```

```
  xtyp_WildConnect:
```

```
    CallbackProc := Form1.WildConnect(HSz1, HSz2, Fmt);
```

```
  xtyp_Connect:
```

```
    if Conv = 0 then
```

```
      begin
```

```
        if Form1.MatchTopicAndService(HSz1, HSz2) then
```

```
          CallbackProc := 1;  { Connected! }
```

```
        end;
```

```
{ When a connection is confirmed, record the conversation handle as  
the
```

```
window's own.
```

```
}
```

```
  xtyp_Connect_Confirm:
```

```
    Form1.ConvHdl := Conv;
```

```
{ The client has requested data, either as a direct request or  
in response to an advisory. Return the current state of the  
data.
```

```
}
```

```

xtyp_AdvReq, xtyp_Request:
begin
  ItemNum := Form1.MatchTopicAndItem(HSz1, HSz2);
  if ItemNum > 0 then
    CallbackProc := Form1.DataRequested(CallType, ItemNum, Fmt);
  end;

```

```

{ Respond to Poke requests ... this demo only allows Pokes of
DataItem1.

```

```

  Return dde_FAck to acknowledge the receipt, 0 otherwise.
}

```

```

xtyp_Poke:
begin
  if Form1.AcceptPoke(HSz2, Fmt, Data) then
    CallbackProc := dde_FAck;
  end;

```

```

{ The client has requested the start of an advisory loop. Note
that we assume a "hot" loop. Set the Advising flag to indicate
the open loop, which will be checked whenever the data is changed.
}

```

```

xtyp_AdvStart:
begin
  ItemNum := Form1.MatchTopicAndItem(HSz1, HSz2);
  if ItemNum > 0 then begin
    if NumAdvLoops < MaxAdvisories then begin { Arbitrary
number }
      Inc(NumAdvLoops);
      Form1.Advising[ItemNum] := True;
      CallbackProc := 1;
    end;
  end;
end;

```

```

{ The client has requested the advisory loop to terminate.
}

```

```

xtyp_AdvStop:
begin
  ItemNum := Form1.MatchTopicAndItem(HSz1, HSz2);
  if ItemNum > 0 then
  begin
    if NumAdvLoops > 0 then
    begin
      Dec(NumAdvLoops);
      if NumAdvLoops = 0 then

```

```

        Form1.Advising[ItemNum] := False;
    CallbackProc := 1;
    end;
    end;
    end;
end; { Case CallType }

```

```
end;
```

```
{ Returns True if the given Topic and Service match those supported
  by this application.  False otherwise.
}
```

```
function TForm1.MatchTopicAndService(Topic, Service: HSz): Boolean;
begin

```

```

    Result := False;
    if DdeCmpStringHandles(TopicHSz, Topic) = 0 then
        if DdeCmpStringHandles(ServiceHSz, Service) = 0 then
            Result := True;

```

```
end;
```

```
{ Determines if the given Topic and Item match one supported by this
  application.  Returns the Item Number of the supported item (in the
  range 1..NumValues) if one is found, and zero if no match.
}
```

```
function TForm1.MatchTopicAndItem(Topic, Item: HSz): Integer;
```

```
var
```

```

    I : Integer;

```

```
begin
```

```

    Result := 0;
    if DdeCmpStringHandles(TopicHSz, Topic) = 0 then
        for I := 1 to NumValues do
            if DdeCmpStringHandles(ItemHSz[I], Item) = 0 then
                Result := I;

```

```
end;
```

```
{ Responds to wildcard connect requests.  These requests are generated
  whenever a client tries to connect to a server with either service
or
```

```

  topic name set to 0.  If a server detects a wild card match, it
  returns a handle to an array of THSZPair's containing the matching
  supported Service and Topic.
}
```

```
function TForm1.WildConnect(Topic, Service: HSz; ClipFmt: Word):
```

```

HDDData;
```

```
var
```

```

TempPairs: array [0..1] of THSZPair;
Matched   : Boolean;
begin
  TempPairs[0].hszSvc   := ServiceHSz;
  TempPairs[0].hszTopic:= TopicHSz;
  TempPairs[1].hszSvc   := 0;           { 0-terminate the list }
  TempPairs[1].hszTopic:= 0;

  Matched := False;

  if (Topic= 0) and (Service = 0) then
    Matched := True                    { Complete wildcard }
  else
    if (Topic = 0) and (DdeCmpStringHandles(Service, ServiceHSz) = 0)
then
      Matched := True
    else
      if (DdeCmpStringHandles(Topic, TopicHSz) = 0) and (Service = 0)
then
        Matched := True;

    if Matched then
      WildConnect := DdeCreateDataHandle(Inst,
TempPairs, SizeOf(TempPairs),
        0, 0, ClipFmt, 0)
    else
      WildConnect := 0;
end;

{ Accepts and acts upon Poke requests from the Client.  For this
demonstration, allows only the value of DataItem1 to be changed by a
Poke.
}
function TForm1.AcceptPoke(Item: HSz; ClipFmt: Word;
  Data: HDDEData): Boolean;
var
  DataStr   : TDataString;
  Err       : Integer;
  TempSample: Integer;
begin
  if (DdeCmpStringHandles(Item, ItemHSz[1]) = 0) and
    (ClipFmt = cf_Text) then
    begin
      DdeGetData(Data,
DataStr, SizeOf(DataStr), 0);

```

```
Val(DataStr, TempSample, Err);
```

```
if IntToStr(TempSample) <> Label6.Caption then begin  
    Label6.Caption := IntToStr(TempSample);  
    DataSample[1] := TempSample;  
    if Advising[1] then  
        DdePostAdvise(Inst, TopicHSz, ItemHSz[1]);  
    end;  
    AcceptPoke := True;  
end  
else  
    AcceptPoke := False;  
end;
```

{ Returns the data requested by the given TransType and ClipFmt values.

This could happen either in response to either an xtyp\_Request or an xtyp\_AdvReq. The ItemNum parameter indicates which of the supported items (in the range 1..NumValues) was requested (note that this method

assumes that the caller has already established validity and ID of the

requested item using MatchTopicAndItem). The corresponding data from

the DataSample instance variable is converted to text and returned.

}

```
function TForm1.DataRequested(TransType: Word; ItemNum: Integer;  
    ClipFmt: Word): HDDEData;  
var ItemStr: TDataString;    { Defined in DataEntry.TPU }
```

```
begin  
    if ClipFmt = cf_Text then  
        begin  
            Str(DataSample[ItemNum], ItemStr);  
            DataRequested := DdeCreateDataHandle(Inst,  
ItemStr,  
                StrLen(ItemStr) + 1, 0, ItemHSz[ItemNum], ClipFmt, 0);  
        end  
    else  
        DataRequested := 0;  
end;
```

{ Constructs an instance of the DDE Server Window. Calls on the inherited constructor, then sets up this objects own instance

```

    data.
}
procedure TForm1.FormCreate(Sender: TObject);
var I : Integer;
begin
    Inst      := 0;      { Must be zero for first call to
DdeInitialize }

    Callback := nil;    { MakeProcInstance is called in SetupWindow }

    for I := 1 to NumValues do begin
        DataSample[I] := 0;
        Advising[I]   := False;
    end; { for }

end;

{ Destroys an instance of the DDE Server Window. Checks to see if the
  Callback Proc Instance had been created, and frees it if so. Also
  calls DdeUninitialize to terminate the conversation. Then just
  calls
  on the ancestral destructor to finish.
}
procedure TForm1.FormDestroy(Sender: TObject);
var
    I : Integer;
begin
    if ServiceHSz <> 0 then
        DdeFreeStringHandle(Inst, ServiceHSz);
    if TopicHSz <> 0 then
        DdeFreeStringHandle(Inst, TopicHSz);
    for I := 1 to NumValues do
        if ItemHSz[I] <> 0 then
            DdeFreeStringHandle(Inst, ItemHSz[I]);

    if Inst <> 0 then
        DdeUninitialize(Inst);    { Ignore the return value }

    if
    Callback <> nil then
        FreeProcInstance(
    Callback);
end;

```

```

procedure TForm1.FormShow(Sender: TObject);
var
  I : Integer;
{ Completes the initialization of the DDE Server Window.  Initializes
  the use of the DDEML by registering the services provided by this
  application.  Recall that the actual names used to register are
  defined in a separate unit (DataEntry), so that they can be used
  by the client as well.
}
begin
  Callback:= MakeProcInstance(
  CallbackProc, HInstance);

  if DdeInitialize(Inst, Callback, 0, 0) = dmlErr_No_Error then begin
    ServiceHSz:= DdeCreateStringHandle(Inst, DataEntryName,
cp_WinAnsi);
    TopicHSz := DdeCreateStringHandle(Inst, DataTopicName,
cp_WinAnsi);
    for I := 1 to NumValues do
      ItemHSz[I] := DdeCreateStringHandle(Inst, DataItemNames[I],
      cp_WinAnsi);

    if DdeNameService(Inst, ServiceHSz, 0, dns_Register) = 0 then
      ShowMessage('Registration failed.');
```

end;

```

end;

procedure TForm1.EnterData1Click(Sender: TObject);
{ Activates the data-entry dialog, and updates the stored
  data when complete.
}
var
  I: Integer;

begin
  if DataEntry.ShowModal = mrOk then begin
    with DataEntry do begin
      Label6.Caption := S1;
      Label7.Caption := S2;
      Label8.Caption := S3;
      DataSample[1] := StrToInt(S1);
      DataSample[2] := StrToInt(S2);
      DataSample[3] := StrToInt(S3);
```



```
end; { with }
```

```
for I := 1 to NumValues do
```

```
  if Advising[I] then
```

```
    DdePostAdvise(Inst, TopicHSz, ItemHSz[I]);
```

```
  end; { if }
```

```
end;
```

```
procedure TForm1.Clear1Click(Sender: TObject);
```

```
{ Clears the current data.
```

```
}
```

```
var
```

```
  I: Integer;
```

```
begin
```

```
  for I := 1 to NumValues do begin
```

```
    DataSample[I] := 0;
```

```
    if Advising[I] then
```

```
      DdePostAdvise(Inst, TopicHSz, ItemHSz[I]);
```

```
  end;
```

```
  Label6.Caption := '0';
```

```
  Label7.Caption := '0';
```

```
  Label8.Caption := '0';
```

```
end;
```

```
end.
```

```
{ ***      END CODE FOR DDESVRU.PAS *** }
```

```
{ *** BEGIN CODE FOR DDEDLG.DFM *** }
```

```
object DataEntry: TDataEntry
```

```
  Left = 488
```

```
  Top = 132
```

```
  ActiveControl = OKBtn
```

```
  BorderStyle = bsDialog
```

```
  Caption = 'Data Entry'
```

```
  ClientHeight = 264
```

```
  ClientWidth = 199
```

```
  Font.Color = clBlack
```

```
  Font.Height = -11
```

```
  Font.Name = 'MS Sans Serif'
```

```
  Font.Style = [fsBold]
```

```
  PixelsPerInch = 96
```





```
    Spacing = -1
    IsControl = True
end
object Panel2: TPanel
    Left = 16
    Top = 88
    Width = 153
    Height = 49
    BevelInner = bvLowered
    BevelOuter = bvNone
    TabOrder = 1
    object Label1: TLabel
        Left = 24
        Top = 8
        Width = 5
        Height = 13
    end
    object Label2: TLabel
        Left = 8
        Top = 8
        Width = 48
        Height = 13
        Caption = 'Value 2:'
    end
    object Edit2: TEdit
        Left = 8
        Top = 24
        Width = 121
        Height = 20
        MaxLength = 10
        TabOrder = 0
        Text = '0'
    end
end
object Panel1: TPanel
    Left = 16
    Top = 16
    Width = 153
    Height = 49
    BevelInner = bvLowered
    BevelOuter = bvNone
    TabOrder = 0
    object Label4: TLabel
        Left = 8
        Top = 8
```

```

    Width = 48
    Height = 13
    Caption = 'Value 1:'
end
object Edit1: TEdit
    Left = 8
    Top = 24
    Width = 121
    Height = 20
    MaxLength = 10
    TabOrder = 0
    Text = '0'
end
end
object Panel3: TPanel
    Left = 16
    Top = 144
    Width = 153
    Height = 49
    BevelInner = bvLowered
    BevelOuter = bvNone
    TabOrder = 2
object Label6: TLabel
    Left = 8
    Top = 8
    Width = 48
    Height = 13
    Caption = 'Value 3:'
end
object Edit3: TEdit
    Left = 8
    Top = 24
    Width = 121
    Height = 20
    MaxLength = 10
    TabOrder = 0
    Text = '0'
end
end
end
end
{ ***      END CODE FOR DDEDLG.DFM *** }

{ *** BEGIN CODE FOR DDEDLG.PAS *** }
{ **** }
{ }
```

```
{ Delphi 1.0 DDEML Demonstration Program }
{ Copyright (c) 1996 by Borland International }
{ }
{*****}
```

```
{ This unit defines the interface to the DataEntry DDE
server (DDEMLSRV.PAS). It defines the Service, Topic,
and Item names supported by the Server, and also defines
a data structure which may be used by the Client to
hold the sampled data locally.
```

The Data Entry Server makes its data samples available in text (cf\_Text) form as three separate Topics. Clients may convert these into integer form for use with the data structure defined here.

```
}
unit Ddedlg;
```

```
interface
```

```
uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Buttons,
StdCtrls, Mask, ExtCtrls;
```

```
type
```

```
TDataEntry = class(TForm)
```

```
OKBtn: TBitBtn;
```

```
CancelBtn: TBitBtn;
```

```
Bevel1: TBevel;
```

```
Panel2: TPanel;
```

```
Label1: TLabel;
```

```
Label2: TLabel;
```

```
Panel1: TPanel;
```

```
Label4: TLabel;
```

```
Panel3: TPanel;
```

```
Label6: TLabel;
```

```
Edit1: TEdit;
```

```
Edit2: TEdit;
```

```
Edit3: TEdit;
```

```
procedure OKBtnClick(Sender: TObject);
```

```
procedure FormShow(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
public
```

```
S1, S2, S3: String;
```

```
{ Public declarations }
```

```
end;

var
  DataEntry: TDataEntry;

implementation

{$R *.DFM}

procedure TDataEntry.OKBtnClick(Sender: TObject);
begin
  S1 := Edit1.Text;
  S2 := Edit2.Text;
  S3 := Edit3.Text;
end;

procedure TDataEntry.FormShow(Sender: TObject);
begin
  Edit1.Text := '0';
  Edit2.Text := '0';
  Edit3.Text := '0';
  Edit1.SetFocus;
end;

end.
{ ***      END CODE FOR DDEDLG.PAS *** }
```

Reference:

4/1/99 4:32:55 PM

Last Modified: 01-SEP-99

## Article #16183: TDBGrid and Multi-Selecting Records

Technical Information Database

TI1183D.txt - TDBGrid and Multi-Selecting Records

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

When you add [dgMultiSelect] to the Options property of a DBGrid, you give yourself the ability to select multiple records within the grid.

The records you select are represented as bookmarks and are stored in the SelectedRows property.

The SelectedRows property is an object of type TBookmarkList. The properties and methods are described below.

```
// property SelectedRows: TBookmarkList read FBookmarks;

// TBookmarkList = class
// public

{* The Count property returns the number of currently
   selected items in the DBGrid *}
// property Count: Integer read GetCount;

{* The Clear method will free all the selected records
   within the DBGrid and set the Count property to 0 *}
// procedure Clear;

{* The Delete method will delete all the selected rows
   from the dataset *}
// procedure Delete;

{* The Find method determines whether a bookmark is
   in the selected list. *}
// function Find(const Item: TBookmarkStr;
```



```

//      var Index: Integer): Boolean;

{* The IndexOf method returns the index of the
   bookmark within the Items property. *}
// function IndexOf(const Item: TBookmarkStr): Integer;

{* The Refresh method returns a boolean value to notify
   whether any orphans were dropped (deleted) during the
   time the record has been selected in the grid. The
   refresh method can be used to update the selected list
   to minimize the possibility of accessing a deleted
   record. *}
// function Refresh: Boolean; True = orphans found

{* The CurrentRowSelected property returns a boolean
   value and determines whether the current row is
   selected or not. *}
// property CurrentRowSelected: Boolean
//      read GetCurrentRowSelected
//      write SetCurrentRowSelected;

{* The Items property is a TStringList of
   TBookmarkStr *}
// property Items[Index: Integer]: TBookmarkStr
//      read GetItem; default;

// end;

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  Grids, DBGrids, Db, DBTables, StdCtrls;

type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    btnCount: TButton;
    btnSelected: TButton;

```

```

btnClear: TButton;
btnDelete: TButton;
btnSelect: TButton;
btnGetBookmark: TButton;
btnFreeBookmark: TButton;
btnFind: TButton;
procedure btnCountClick(Sender: TObject);
procedure btnSelectedClick(Sender: TObject);
procedure btnClearClick(Sender: TObject);
procedure btnDeleteClick(Sender: TObject);
procedure btnSelectClick(Sender: TObject);
procedure btnGetBookmarkClick(Sender: TObject);
procedure btnFreeBookmarkClick(Sender: TObject);
procedure btnFindClick(Sender: TObject);
private
    Bookmark: TBookmark;
public
    { Public declarations }
end;

```

```

var
    Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

//Example of the Count property

```

```

procedure TForm1.btnCountClick(Sender: TObject);
begin
    if DBgrid1.SelectedRows.Count > 0 then
    begin
        ShowMessage(IntToStr(DBgrid1.SelectedRows.Count));
    end;
end;

```

```

//Example of the CurrentRowSelected property

```

```

procedure TForm1.btnSelectedClick(Sender: TObject);
begin
    if DBgrid1.SelectedRows.CurrentRowSelected then
        ShowMessage('Selected');
end;

```

```

//Example of the Clear Method

```

```

procedure TForm1.btnClearClick(Sender: TObject);

```

```

begin
    DBGrid1.SelectedRows.Clear;
end;
//Example of the Delete Method
procedure TForm1.btnDeleteClick(Sender: TObject);
begin
    DBGrid1.SelectedRows.Delete;
end;

{*
    This example iterates through the selected rows
    of the grid and displays the second field of
    the dataset.

    The Method DisableControls is used so that the
    DBGrid will not update when the dataset is changed.
    The last position of the dataset is saved as
    a TBookmark.

    The IndexOf method is called to check whether or
    not the bookmark is still existent.
    The decision of using the IndexOf method rather
    than the Refresh method should be determined by the
    specific application.
*}

procedure TForm1.btnSelectClick(Sender: TObject);
var
    x: Integer;
    TempBookmark: TBookmark;
begin
    DBGrid1.Datasource.Dataset.DisableControls;
    with DBGrid1.SelectedRows do
        if Count > 0 then
            begin
                TempBookmark := DBGrid1.Datasource.Dataset.GetBookmark;
                for x := 0 to Count - 1 do
                    begin
                        if IndexOf(Items[x]) > -1 then
                            begin
                                DBGrid1.Datasource.Dataset.Bookmark := Items[x];
                                ShowMessage(DBGrid1.Datasource.Dataset.Fields[1].AsString);
                            end;
                    end;
            end;
        end;
end;

```

```

    DBGrid1.Datasource.Dataset.GotoBookmark(TempBookmark);
    DBGrid1.Datasource.Dataset.FreeBookmark(TempBookmark);
end;
DBGrid1.Datasource.Dataset.EnableControls;
end;

{ *
This example allows you to set a bookmark and
and then search for the bookmarked record within
selected a record(s) within the DBGrid.
* }

//Sets a bookmark
procedure TForm1.btnGetBookmarkClick(Sender: TObject);
begin
    Bookmark := DBGrid1.Datasource.Dataset.GetBookmark;
end;
// Free the bookmark
procedure TForm1.btnFreeBookmarkClick(Sender: TObject);
begin
    if Assigned(Bookmark) then
    begin
        DBGrid1.Datasource.Dataset.FreeBookmark(Bookmark);
        Bookmark := nil;
    end;
end;

//Uses the Find method to locate the position of the
//bookmarked record within the selected list in the DBGrid
procedure TForm1.btnFindClick(Sender: TObject);
var
    Index: Integer;
begin
    if Assigned(Bookmark) then
    begin
        if DBGrid1.SelectedRows.Find(TBookMarkStr(Bookmark), Index) then
            ShowMessage('Index in SelectedRows: ' + IntToStr(Index));
        end;
    end;
end;

end.

{ Unit1.DFM Source }
object Form1: TForm1
    Left = 206

```

```
Top = 107
Width = 616
Height = 459
Caption = 'Form1'
Color = clBtnFace
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = False
PixelsPerInch = 96
TextHeight = 13
object DBGrid1: TDBGrid
  Left = 48
  Top = 112
  Width = 505
  Height = 281
  DataSource = DataSource1
  Options = [dgEditing, dgTitles, dgIndicator, dgColumnResize,
dgColLines, dgRowLines, dgTabs, dgConfirmDelete, dgCancelOnExit,
dgMultiSelect]
  TabOrder = 0
  TitleFont.Charset = DEFAULT_CHARSET
  TitleFont.Color = clWindowText
  TitleFont.Height = -11
  TitleFont.Name = 'MS Sans Serif'
  TitleFont.Style = []
end
object btnCount: TButton
  Left = 40
  Top = 32
  Width = 75
  Height = 25
  Caption = 'Count'
  TabOrder = 1
 OnClick = btnCountClick
end
object btnSelected: TButton
  Left = 144
  Top = 32
  Width = 75
  Height = 25
  Caption = 'Selected'
  TabOrder = 2
```

```
    OnClick = btnSelectedClick
end
object btnClear: TButton
    Left = 248
    Top = 32
    Width = 75
    Height = 25
    Caption = 'Clear'
    TabOrder = 3
    OnClick = btnClearClick
end
object btnDelete: TButton
    Left = 352
    Top = 32
    Width = 75
    Height = 25
    Caption = 'Delete'
    TabOrder = 4
    OnClick = btnDeleteClick
end
object btnSelect: TButton
    Left = 464
    Top = 32
    Width = 75
    Height = 25
    Caption = 'Select'
    TabOrder = 5
    OnClick = btnSelectClick
end
object btnGetBookmark: TButton
    Left = 144
    Top = 72
    Width = 75
    Height = 25
    Caption = 'GetBookmark'
    TabOrder = 6
    OnClick = btnGetBookmarkClick
end
object btnFreeBookmark: TButton
    Left = 248
    Top = 72
    Width = 81
    Height = 25
    Caption = 'FreeBookmark'
    TabOrder = 7
```

```
    OnClick = btnFreeBookmarkClick
end
object btnFind: TButton
    Left = 344
    Top = 72
    Width = 75
    Height = 25
    Caption = 'Find'
    TabOrder = 8
    OnClick = btnFindClick
end
object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'employee.db'
    Left = 384
    Top = 360
end
object DataSource1: TDataSource
    DataSet = Table1
    Left = 440
    Top = 376
end
end
```

Reference:

3/30/99 11:15:25 AM

Last Modified: 01-SEP-99

## Article #16189: Listing the field structures of a table.

Technical Information Database

TI1189D.txt - Listing the field structures of a table.

Category :Database Programming  
Platform :All  
Product :

Description:

Q: How can I list the field structures of a table?

A: This project demonstrates listing the field structure from a given table, using the Fields and IndexDefs arrays, and displaying them in a listbox.

There is a demo (dbbrowsr.dpr) approaches this task differently. You may want to compare the two versions of this code.

Note: This code works with 16 bit only.

```
procedure TForm1.Button1Click(Sender: TObject); const
MyFielddefs: array[ftUnknown..ftGraphic] of string [8] =
  ('Unknown', 'String', 'Smallint', 'Integer', 'Word',
  'Boolean', 'Float', 'Currency', 'BCD', 'Date',
  'Time', 'DateTime', 'Bytes', 'VarBytes', 'Blob',
  'Memo', 'Graphic');

var
  i, Indx: integer;
  Definition: string;
begin
  for i := 0 to Table1.FieldCount - 1 do begin
    Definition := Table1.Fields[i].DisplayLabel;
    Definition := Definition + ' ' +
      MyFieldDefs[Table1.Fields[i].DataType];
    Table1.IndexDefs.Update;
    if Table1.Fields[i].IsIndexField then begin
      Indx := Table1.IndexDefs.Indexof(Table1.Fields[i].Name);
      if Indx > -1 then
```



```

        if ixPrimary in Table1.IndexDefs[Indx].Options then
            Definition := Definition + ' (Primary)';
        end;
        Listbox1.Items.Add(Definition);
    end;
end;

```

The version above does not work with the 32 bit version as there are more field types that must now be taken into account. Here is a version that works with the 32 bit version:

```

procedure TForm1.Button1Click(Sender: TObject);
const
    MyFielddefs: array[ftUnknown..ftTypedBinary] of string [11] =
        ('Unknown', 'String', 'Smallint', 'Integer',
         'Word', 'Boolean', 'Float', 'Currency', 'BCD',
         'Date', 'Time', 'DateTime', 'Bytes', 'VarBytes',
         'AutoInc', 'Blob', 'Memo', 'Graphic', 'FmtMemo',
         'ParadoxOle', 'DBaseOle', 'TypedBinary');
var
    i, Indx: integer;
    Definition: string;
begin
    for i := 0 to Table1.FieldCount - 1 do begin
        Definition := Table1.Fields[i].DisplayLabel;
        Definition := Definition + ' ' +
            MyFieldDefs[Table1.Fields[i].DataType];
        Table1.IndexDefs.Update;
        if Table1.Fields[i].IsIndexField then begin
            Indx := Table1.IndexDefs.Indexof(Table1.Fields[i].Name);
            if Indx > -1 then
                if ixPrimary in Table1.IndexDefs[Indx].Options then
                    Definition := Definition + ' (Primary)';
                end;
            Listbox1.Items.Add(Definition);
        end;
    end;
end;

```

Reference:

3/31/99 10:29:10 AM

Last Modified: 01-SEP-99

## Article #16195: Showing deleted records in a dBASE table.

Technical Information Database

TI1195D.txt Showing deleted records in a dBASE table.

Category :Database Programming

Platform :All

Product :Delphi All

Description:

Q: How can I view dBASE records marked for deletion. That is, I want to view those records marked as "soft deletion"?

A: In a dBASE table, records are not removed from the table until the table is packed. Until that happens, records that are "deleted" are actually just marked as "to be" deleted. To show these existing but not displayed records, the following function, ShowDeleted(), makes use of a BDE API function, DbisetProp(), to show records marked for deletion. It is not necessary to close and re-open the table when using this function. ShowDeleted() takes a TTable and a boolean variable as parameters. The boolean parameter determines whether or not to show deleted records.

Example code follows:

```
-----  
  
unit Unit1;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls, ExtCtrls, DBCtrls, Grids, DBGrids,  
  DB, DBTables;  
  
type  
  TForm1 = class(TForm)  
    Table1: TTable;  
    DataSource1: TDataSource;  
    DBGrid1: TDBGrid;  
    DBNavigator1: TDBNavigator;
```

```
    CheckBox1: TCheckBox;  
    procedure CheckBox1Click(Sender: TObject);  
public  
    procedure ShowDeleted(Table: TTable; ShowDeleted: Boolean);  
end;
```

```
var  
    Form1: TForm1;
```

```
implementation
```

```
uses DBTYPES, DBIERRS, DBIPROCS;
```

```
{ $R *.DFM }
```

```
procedure TForm1.ShowDeleted(Table: TTable; ShowDeleted: Boolean);
```

```
var  
    rslt: DBIResult;  
    szErrMsg: DBIMSG;  
begin  
    Table.DisableControls;  
    try  
        Check(DbiSetProp(hDBIObj(Table.Handle), curSOFTDELETEON,  
            LongInt(ShowDeleted)));  
    finally  
        Table.EnableControls;  
    end;  
    Table.Refresh;  
end;
```

```
procedure TForm1.CheckBox1Click(Sender: TObject);
```

```
begin  
    ShowDeleted(Table1, CheckBox1.Checked);  
end;
```

```
end.
```

```
Reference:
```

```
7/16/98 4:34:05 PM
```

```
Last Modified: 01-SEP-99
```

## Article #16201: Determining a memo's number of lines showing.

Technical Information Database

TI1201D.txt Determining a memo's number of lines showing.

Category :General Programming

Platform :All

Product :Delphi All

Description:

Q: How can I tell how many lines a memo is capable of showing?

A: Here is the short and elegant way that works (most of the time):

```
function TForm1.MemoLinesShowing(memo: TMemo): integer;
var
  R: TRect;
begin
  Memo.Perform(EM_GETRECT, 0, Longint(
R));
  Result := (R.Bottom - R.Top) div Canvas.TextHeight('XXX');
end;
```

The problem with this code is that the TForm and the TMemo must both be using the same font. If the fonts are different, then the calculations are not accurate.

You have to retrieve the font height by selecting it into a device context. The reason you cannot use the font Height provided by Delphi is because Delphi caches the font information but doesn't actually select the font into the DC (canvas) until it is actually going to draw something. This occurs in the painting event of the memo.

To get around this problem, you can get the memo's device context using the Windows API and get the font information from the device context to calculate the text height.

The function below illustrates this process:

```
function TForm1.MemoLinesShowingLong(Memo: TMemo): integer;
```

```

Var
  Oldfont: HFont;    {the old font}
  DC: THandle;      {Device context handle}
  i: integer;       {loop variable}
  Tm: TTextMetric; {text metric structure}
  TheRect: TRect;

begin
  DC := GetDC(Memo.Handle); {Get the memo's device context}
  try
    {Select the memo's font}
    OldFont := SelectObject(DC, Memo.Font.Handle);
    try
      GetTextMetrics(DC, Tm); {Get the text metric info}
      Memo.Perform(EM_GETRECT, 0, longint(
TheRect));
      Result := (TheRect.Bottom - TheRect.Top) div
        (Tm.tmHeight + Tm.tmExternalLeading);
    finally
      SelectObject(DC, Oldfont); {Select the old font}
    end;
  finally
    ReleaseDC(Memo.Handle, DC); {Release the device context}
  end;
end;

```

Reference:

7/16/98 4:34:05 PM

Last Modified: 01-SEP-99

## Article #16203: Desqview Control Break Conflict

Technical Notes Database

TN1203D.txt Desqview Control Break Conflict

Category :Pascal

Platform :All

Product :Turbo Pascal 6.0

Description:

These setup options to Desqview will fix the conflict

In DesqView:

Main Setup - Set Shift States to YES.

Pif Setup - Advanced Options submenu - Keyboard Conflicts = 9

Reference:

7/16/98 4:35:42 PM

Last Modified: 01-SEP-99

## Article #16207: Delphi 2.0 Install Issues

Technical Information Database

TI1207D.txt - Delphi 2.0 Install Issues

Category :Delphi 2.x  
Platform :All-32Bit  
Product :Delphi2.x,

### Description:

Install Issues for Delphi 2.0

Last updated 3/13/96

- 1) Minimum System Requirements
- 2) a. Canceling the Delphi 2.0 install  
b. Canceling Delphi 2.0 Online Registration
- 3) Un-Installing
- 4) Re-installing Delphi 2.0 on Window's 95
- 5) Installing to systems with Paradox 7.0
- 6) Installing Delphi 1.0 and 2.0 on the same system
- 7) Errors During the Install

- 1) Minimum System Requirements

### Disk space:

a) OCX, DLL, and licensing files will require approx. 10 megs in your \windows\system directory.

b) Expansion of files during installation temporarily requires 10-15 megs in the drive that contains your \temp directory. For example, if you are installing Delphi 2.0 to drive D:, and your temporary directory path is C:\WINDOWS\TEMP, so that drive C will require 10 - 15 megs during file expansion.

### Minimum system requirements:

Processor



Desktop : intel 386dx based pc or higher with  
 math coprocessor  
 Developer: intel 486/25 based pc or higher  
 C/S Suite: intel 486/25 based pc or higher  
 MS windows '95 or NT 3.51  
 8 meg memory (12 recommended)  
 50 meg hard disk space  
 CD rom drive (3.5 floppy disks available seperately)  
 mouse or Windows pointing device

The chart below shows disk space used resulting from sample installs run on a Windows '95 system.

"Permanant Megs Used" represents the amount of disk used after a sample install to a stock Window's '95 system with no other applications running. These figures are approximate. Your disk cluster size may cause the file sizes to vary. Also, Window's '95 handling of swap file may vary according to your system configuration.

| Product      | Permanent<br>Megs<br>Used | \Borland | \Windows | (Swap<br>\System File) |
|--------------|---------------------------|----------|----------|------------------------|
| CSS compact  | 45                        | 39       | 6        | 9                      |
| CSS full     | 105                       | 98       | 7        | 24                     |
| Dev compact  | 43                        | 37       | 6        | 9                      |
| Dev full     | 104                       | 97       | 7        | 20                     |
| Desk compact | 32                        | 30       | 2        | 9                      |
| Desk full    | 61                        | 59       | 2        | 15                     |

- 2)
- a) Canceling the Delphi 2.0 install  
 Canceling before 10 - 15% may not clean the Window's '95 registry. This appears to be a limitation of the Install Shield engine.
- Canceling the install after 15% will cause Delphi 2.0 to be un-installed.

- b) Canceling Delphi 2.0 Online Registration  
 If you wish to register Delphi 2.0 at a later time, you may cancel the Registration Wizard. Canceling the Registration Wizard will not cause

an un-install. You may register Delphi 2.0 later from the "Delphi 2.0 Online Registration" icon located in the Borland Delphi 2.0 directory.

### 3) Un-Installing

Files created after the install of Delphi 2.0 and the directories that contain them, will not be deleted. For example, if you were to create a new project in \Borland\Delphi 2.0\Bin, the un-install would leave your project files and the \Borland\Delphi 2.0\Bin directory that contained them.

A general rule of thumb is that only files from the most recent install will be un-installed.

Files marked as shared, OCX's and DLL's for example, that have a "use" count of zero will cause a delete confirmation dialog to appear for each file. Microsoft standards for un-installing do not include a "Delete All" option at this stage.

### 4) Re-installing Delphi 2.0 on Window's 95

If an earlier installation of Delphi 2.0 already exists on your Window's 95 system, you must run the un-install program. This is necessary to cleanup the system registry. The un-install utility is located off the Start menu, Settings|Control Panel|Add/Remove Programs.

### 5) Installing to systems with Paradox 7.0

The necessary order to install both Delphi 2.0 and Paradox 7.0 is to

- a) install Paradox 7.0
- b) install Delphi 2.0.

If Delphi 2.0 already exists on the system, un-install Delphi 2.0, then do a) and b) listed above. If you un-install either Delphi 2.0 or Paradox 7.0, you will need to re-install the 32 bit IDAPI.

### 6) Installing Delphi 1.0 and 2.0 on the same

system

Although Delphi 1.0 and 2.0 will run on the same system, don't combine Delphi 1.0 and 2.0 into the same directory. Installing to the same directory will cause difficulty when the Delphi IDE attempts to locate the Help file.

## 7) Errors During the Install

Error 101

Error 112;

There is not enough room in the \Temp directory to expand files. Expansion of files during installation temporarily requires 10-15 megs in the drive that contains your \temp directory. For example, if you are installing Delphi 2.0 to drive D:, and your temporary directory path is C:\WINDOWS\TEMP, so that drive C will require 10 - 15 megs during file expansion.

"Cannot install Delphi, Local Interbase is Running"

Shut down Interbase from the system tray by right clicking the Interbase icon. The System Tray is the indentation located on right hand side of the task bar)

Registration Wizard dialog appears as a blank InstallShield could not find the install script. Cancel the dialog and run the Registration Wizard from the icon "Delphi 2.0 Online Registration".

Temporary directories remain after install to NT. On NT systems temporary directories named \ISTEMP0x may be left over. These directories will probably be empty, and may be deleted.

Reference:

3/30/99 12:37:32 PM

Last Modified: 01-SEP-99

## Article #16209: Using the 80X87 emulator in TPW.

Technical Notes Database

TN1209D.txt Using the 80X87 emulator in TPW.

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

Description:

Question:

Must TPW a the math coprocessor?

Answer:

Windows implements its own 80x87 instruction emulator as a DLL that is loaded when an 80x87 instruction is detected. The reason that the '87 code generation is an option is speed: emulators are always slower than the real hardware, and slower than using the 6 byte reals. If you need double, or extended reals, or Comp integers, then you must turn on '87 code generation, and Windows will adjust itself to compensate if there is no '87 hardware when the program is running. You do not need a coprocessor to compile or run the program.

Reference:

7/16/98 4:35:42 PM

Last Modified: 01-SEP-99

## Article #16213: How to Validate ISBNs

Technical Information Database

TI1213D.txt - How to Validate ISBNs

Category :Miscellaneous  
Platform :All-32Bit  
Product :All32Bit,

### Description:

ISBNs (or International Standard Book Numbers) are mystical code numbers that uniquely identify books. The purpose of this article is to remove the mystery surrounding the structure of the ISBN, allowing applications to perform data validation on entered candidate ISBNs.

ISBNs are composed of thirteen characters, limited to the number characters "0" through "9", the hyphen, and the letter "X". This thirteen-character code is divided into four parts, each separated by hyphens: group identifier, publisher identifier, book identification for the publisher, and the check digit. The first part (group identifier) is used to identify countries, geographical regions, languages, etc. The second part (publisher identifier) uniquely identifies the publisher. The third part (book identifier) uniquely identifies a given book within a publisher's collection. The fourth and final part (check digit) is used with the other digits in the code in an algorithm to derive a verifiable ISBN. The number of digits in the first three parts of an ISBN may contain a variable number of digits, but the check digit will always consist of a single character (between "0" and "9", or "X" for a value of 10) and the ISBN as a whole will always consists of thirteen characters (ten

numbers

plus the three hyphens dividing the four parts of the ISBN).

The ISBN 3-88053-002-5 breaks down into the parts:

Group: 3  
Publisher: 88053  
Book: 002  
Check Digit: 5

An ISBN can be verified to be a valid code using a simple mathematical algorithm. This algorithm takes each of the nine single digits from the

first three parts of the ISBN (sans the non-numeric hyphens), multiplies

each single digit by a number that is less than eleven the number of positions from the left each digit that is in the ISBN, adds together the

result of each multiplication plus the check digit, and then divides that

number by eleven. If that division by eleven results in no remainder (i.e., the number is modulo 11), the candidate ISBN is a valid ISBN.

For

example, using the previous sample ISBN 3-88053-002-5:

|                   |                                     |   |   |   |   |   |   |   |   |   |
|-------------------|-------------------------------------|---|---|---|---|---|---|---|---|---|
| ISBN:             | 3                                   | 8 | 8 | 0 | 5 | 3 | 0 | 0 | 2 | 5 |
| Digit Multiplier: | 10                                  | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| Product:          | 30+72+64+00+30+15+00+00+04+05 = 220 |   |   |   |   |   |   |   |   |   |

Since 220 is evenly divisible by eleven, this candidate ISBN is a valid

ISBN code.

This verification algorithm is easily translated into Pascal/Delphi code.

String manipulation functions and procedures are used to extract the check digit and the remainder of the ISBN from the String type value passed to a validation function. The check digit is converted to

Integer

type, which forms the start value of the aggregate variable onto which the

multiplication of each digit in the remainder of the ISBN (the single digits that comprise the first three parts of the ISBN) will be added. A

For loop is used to sequentially process each digit in the remainder,

ignoring the hyphens, multiplying each digit times its position in the ISBN remainder relative to the other digits in the remainder. The final value of this aggregate variable is then checked to see whether it is evenly divisible by eleven (indicating a valid ISBN) or not (indicating an invalid candidate ISBN).

Here is an example of this methodology applied in a Delphi function:

```
function IsISBN(ISBN: String): Boolean;
var
  Number, CheckDigit: String;
  CheckValue, CheckSum, Err: Integer;
  i, Cnt: Word;
begin
  {Get check digit}
  CheckDigit := Copy(ISBN, Length(ISBN), 1);
  {Get rest of ISBN, minus check digit and its hyphen}
  Number := Copy(ISBN, 1, Length(ISBN) - 2);
  {Length of ISBN remainder must be 11 and check digit between 9 and
  9 or
  X}
  if (Length(Number) = 11) and (Pos(CheckDigit, '0123456789X') > 0)
  then
    begin
      {Get numeric value for check digit}
      if (CheckDigit = 'X') then
        CheckSum := 10
      else
        Val(CheckDigit, CheckSum, Err);
      {Iterate through ISBN remainder, applying decode algorithm}
      Cnt := 1;
      for i := 1 to 12 do begin
        {Act only if current character is between "0" and "9" to exclude
        hyphens}
        if (Pos(Number[i], '0123456789') > 0) then begin
          Val(Number[i], CheckValue, Err);
          {Algorithm for each character in ISBN remainder, Cnt is the
          nth
          character so processed}
          CheckSum := CheckSum + CheckValue * (11 - Cnt);
          Inc(Cnt);
        end;
      end;
    end;
end;
```



```
{Verify final value is evenly divisible by 11}
if (Checksum MOD 11 = 0) then
    IsISBN := True
else
    IsISBN := False;
end
else
    IsISBN := False;
end;
```

This is a simplified example, kept simple to best demonstrate the algorithm to decode ISBNs. There are a number of additional features that would be desirable to add for use in a real-world application. For instance, this example function requires the candidate ISBN be passed as a Pascal String type value, with the hyphens dividing the four parts of the ISBN. Added functionality might accommodate evaluating candidate ISBNs entered without the hyphens. Another feature that might be added is checking that ensures three hyphens are properly included, as opposed to just thirteen number characters.

Reference:

4/22/99 4:00:25 PM

Last Modified: 01-SEP-99

## Article #16215: The \$G+ directive, 286 code generation, and Real M

Technical Notes Database

TN1215D.txt    The \$G+ directive, 286 code generation, and Real M  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal  1.0

Description:

Question:

If I use the \$G+ 286 code generation directive, how can I be sure that Windows is running in Real mode?

Answer:

The 286 code generation option allows the compiler to substitute more efficient 286 instructions in place of normal 8088 instructions for things like loops, jumps and memory operations. If you compile your program with \$G+, then you should call GetWinFlags early in the startup of your application - such as in your Application.Init. Check for the wf\_PtMode flag set. If it's not set, then you're not running in protected mode, and you should display a message box and exit your program. Setting the Application's status variable to a non-zero value should also work, although that would cause more code to be executed, implying greater risk of hitting a 286 instruction.

All the precompiled units in TPW will run in real mode and are not affected by the \$G+ directive. The \$G+ directive only affects code that you compile yourself.

Reference :

7/16/98 4:35:42 PM

Last Modified: 01-SEP-99

## Article #16219: Delphi 2.0 for Windows 95 & Windows NT Factsheet

Technical Information Database

TI1219D.txt - Delphi 2.0 for Windows 95 & Windows NT Factsheet

Category :Delphi 2.x  
Platform :All-32Bit  
Product :Delphi2.x,

Description:

Delphi 2.0  
for Windows 95 and Windows NT  
Factsheet

Delphi 2.0 is the only Rapid Application Development tool that combines the performance of the world's fastest optimizing 32-bit native-code compiler, the productivity of visual component-based design, and the flexibility of scalable database architecture in a robust object-oriented environment.

"The most exciting new programming tool. Delphi combines the intuitive ease of a visual programming tool with the power of a true compiler." -PC Magazine, January 1996

The fastest way to the fastest 32-bit applications  
-----

Now, whether you are a beginning programmer, professional developer, or client/server developer, there is a version of Delphi that is right for you. Introducing Delphi Desktop 2.0, Delphi Developer 2.0, and Delphi Client/Server Suite 2.0. Delphi 2.0 for Windows 95 and Windows NT combines the most intuitive visual design environment and scalable database tools with the world's fastest optimizing 32-bit native-code compiler.

The NEW! high-performance, optimizing 32-bit native-code compiler creates standalone executables that are up to 300 to 400% faster than 16-bit Delphi applications, and up to 15 to 50 times faster than applications built with p-code interpreters. Create fast, standalone, royalty-free EXEs with no runtime interpreter DLLs

required.

Only Delphi 2.0 brings the power of true 32-bit development to a visual environment. The 32-bit architecture eliminates all 16-bit memory restrictions, enabling you to build strings and data structures up to two gigabytes.

Delphi 2.0 includes FREE! Delphi 1.0 for 16-bit Windows 3.1 application development, making it a complete solution no matter where your Windows-based applications reside today. A simple recompile allows rapid migration of 16-bit applications to 32 bits, preserving your existing investment in code.

Reuse: The key to productivity

-----

The NEW! 32-bit Visual Component Library (VCL) includes more than 90 customizable, reusable components and a dozen NEW! Windows 95 components, so you can drag-and-drop your way to your first 32-bit application in no time. Everything from tree views and list boxes, to notebook tabs, grids, and multimedia controls, is right at your fingertips. You can easily modify and reuse Delphi components and OCX controls, saving time and boosting productivity.

FREE! Quick Reports components allow you to easily create, preview, and print simple embedded reports.

NEW! Visual Form Inheritance maximizes reuse, simplifies the implementation of standards, and makes code maintenance easier. Visual Form Inheritance lets you visually create new forms that inherit all properties from another form.

Create reusable DLLs easily

-----

Build DLLs in Delphi and use them with C++, Paradox,(R) Visual Basic, and other tools. And gain tighter integration between all of your Borland(R) C++ and Delphi applications with NEW! support for OBJ files and COM objects.

Raising the bar in Rapid Application Development

-----

The object-oriented architecture and Two-Way-Tools(TM) in Delphi

maximize productivity. Everything you do visually is automatically reflected in code.

The NEW! centralized Object Repository stores forms, Data Module Objects, and business rules for reuse. Any new application can inherit, use, or copy an existing structure so you can pick the architecture that best fits your development needs.

NEW! Data Module Objects act as your application's information core, allowing you to define data sources, store relationships, and apply business rules to your database. You can separate your business rules from application code for easier maintenance.

A comprehensive suite of visual design and debugging tools accelerate development so you move seamlessly from prototype to production. The NEW! multi-error compile with hints and warnings helps you resolve errors faster. Programmers can jump to and fix multiple problems in a single compile. For quick and easy debugging of even your largest applications, the new compiler even catches logic errors like empty loops, uninitialized pointers, and unused variables.

Unleash the power of Windows 95 and Windows NT

-----  
Delphi 2.0 provides complete access to Windows 95 and NT APIs including multi-threading, MAPI, and Unicode, so you can leverage the power of the 32-bit operating system.

NEW! OCX support lets you drag-and-drop OCX controls directly into your applications from the Delphi Component Palette. Delphi also lets you subclass OCXs for easy customization. You can also integrate OLE-enabled applications like Visual Basic, Excel, Borland C++, and Paradox, with NEW! OLE Automation controller and server support.

Drag-and-drop database development

-----  
Create high-performance, scalable database applications with integrated database support. NEW! data-aware controls let you manipulate live data at design time. Simply select the fields, and Delphi instantly builds the connections.

Let Delphi 2.0 do the database work for you with NEW! Fast

Filters and Smart Lookups. Fast Filters make viewing a specific subset of your data as fast and easy as changing a property. Smart Lookups automatically provide a dropdown box to seamlessly display data from multiple tables.

The NEW! Database Explorer allows you to visually browse and manage tables and aliases without leaving the Delphi IDE.

NEW! 32-bit Borland(R) Database Engine (BDE) includes a new query engine and takes advantage of 32-bit flat address space for increased performance. It provides direct access to Visual dBASE(R) and Paradox tables, and forms a solid foundation for all your database applications.

Delphi Developer: The choice for professional programmers  
-----

If you are building sophisticated, multiuser LAN or corporate database applications, you need Delphi Developer 2.0. It provides all the power of Delphi Desktop 2.0, plus a comprehensive set of professional programming tools.

Powerful programming tools  
-----

The VCL Source Code lets you easily modify native Delphi components, and build your own custom components. Also included are sample OCXs for graphing, charting, spreadsheet, and spell-checking.

Easily add your favorite custom development tools directly to the Delphi environment with the NEW! Open Tools API. Interact with CASE tools or link to your favorite editor--whatever you need to be your most productive.

Build sophisticated, professional reports using live data at design time with the NEW! 32-bit ReportSmith.(R)

When you're ready to distribute your applications, the integrated install utility, InstallShield(R) Express, instantly builds a professional installation program for even your most complex database applications.

Advanced database power  
-----

Quickly establish and maintain data integrity with the NEW! scalable Data Dictionary. It stores customized information about the data in your tables, extended field attributes like min, max, default values, display preferences, and edit masks.

Easily build detailed views of your data with the NEW! Multi-Object Grid. It provides codeless support for quick searching and automatic lookups, and lets you decide whether data appears in rows or individual panels. Panel views support multiple dropdowns, check boxes, and edit fields.

Leverage the power of the NEW! 32-bit BDE with complete access to the BDE API and help files. You also get complete ODBC connectivity.

#### Complete scalability

-----

The scalable NEW! 32-bit Local InterBase(R) Server is included for economical off-line development and testing, and for building standalone ANSI SQL-92 compliant applications.

#### Delphi Client/Server Suite: Gain the competitive advantage

-----

Delphi Client/Server Suite 2.0 gives you a complete suite of tools to build scalable, high-performance client and server applications for Windows 95, Windows NT, and Windows 3.1.

Deploy royalty-free applications across corporate database servers using the NEW! 32-bit SQL Links high-performance native drivers for InterBase, Oracle, Sybase, and MS SQL Server.

Visually browse and modify server-specific meta data including stored procedures, triggers, event alerters, and domains with the NEW! SQL Database Explorer. Gain optimum performance by testing, debugging, and tuning your SQL applications with the NEW! SQL Monitor. Build and test sophisticated multiuser client/server applications with NEW! InterBase NT database server (2-user license). Use the NEW! Visual Query Builder to visually create complex database queries and automatically generate bug-free ANSI SQL-92 commands.

NEW! Cached Updates increase server responsiveness by reducing



the amount of network traffic between client and server. Scale networked applications and move data between formats and host platforms with the NEW! Data Pump Expert. Manage complex team projects with sophisticated source code check-in, check-out, and version control using the NEW! integrated PVCS Version Manager.

Delphi Client/Server Suite 2.0 shortens every stage of the development cycle, allowing you to respond to changing business needs, and gain the competitive advantage.

Delphi 2.0 Specifications and system requirements

#### -----

#### Optimizing Native-Code Compiler

- \* Compiles at more than 350,000 lines per minute
- \* Create fast standalone EXEs with no runtime interpreter
- Dynamic Link Library (DLL)
- \* Applications run up to 15 to 50 times faster than interpreted p-code
- \* Create DLLs that work with C++, Visual dBASE, Paradox, Visual Basic, PowerBuilder, and others
- \* Full access to Windows 95 and NT APIs
- \* Optimized case statements, sets, 32-bit math operations, string and file routines, and more
- \* Smart Linker removes unused objects and code
- \* Automatic form linking defines uses clauses
- \* Advanced math library for extended statistical and financial analysis

#### Visual Component Library (VCL)

- \* More than 90 reusable components
- \* Support for the latest Microsoft systems technologies including OCX, OLE Automation controller and server, MAPI, and ODBC
- \* Standard components for menus, buttons, masked edit fields, panels, graphics, notebook tabs, grids, outlines, list boxes, and more
- \* "Live" design-time data access
- \* VCL Source Code available separately

#### Object Pascal Language

- \* Structured, object-oriented language
- \* Supports inheritance, polymorphism, and encapsulation
- \* Control over privacy with Public, Private, Protected, and Published reserved words

- \* Create components with properties and events
- \* Use inheritance to customize any object
- \* Runtime type information and object persistence
- \* Automatic, extensible exception handling
- \* Support for open arrays, user-defined types, objects, and pointers
- \* Support for delegation and class references

#### Borland Database Engine

- \* High-performance engine with native drivers for Visual dBASE, Paradox, and Local InterBase Server
- \* Fully scalable support for migrating applications from desktop to client/server
- \* Royalty-free deployment of database engine
- \* ODBC support for Access, Btrieve, Excel, DB2, AS/400, Ingres, HP ALBASE/SQL, and gateways like IBM DDCS/2, Micro Decisionware, and Sybase Net-gateway (available separately)
- \* Local InterBase is a high-performance ANSI SQL-92 conformant SQL database server

#### Documentation and Help

- \* More than 5Mb of online help
- \* Manuals: Getting Started, Object Pascal Language Guide, Delphi User's Guide, Component Writer's Guide, and Database Application Developer's Guide

#### Minimum System Requirements

- \* Intel 486-based PC or higher
- \* Microsoft Windows 95 or Windows NT
- \* 8Mb of extended memory or higher
- \* 50Mb hard disk space
- \* CD-ROM drive (3.5" disks available separately)

#### Networks Supported

- \* Any Microsoft Windows 95 or NT-compatible networks

The fastest way to the fastest Windows 95/NT applications

|                                                 | Delphi Desktop | Delphi Developer | Delphi C/S Suite |
|-------------------------------------------------|----------------|------------------|------------------|
| -----<br>Optimizing 32-bit native code compiler | X              | X                | X                |

|                                                                                                                      |   |   |   |
|----------------------------------------------------------------------------------------------------------------------|---|---|---|
| 32-bit Borland Database Engine (BDE)                                                                                 | X | X | X |
| Create reusable DLLs and royalty-free, standalone EXEs that are up to 15 to 50 times faster than p-code interpreters | X | X | X |
| Full Windows 95/NT support for OLE Automation controllers, servers, and OLE Controls (OCXs)                          | X | X | X |
| Complete access to Windows 95/NT APIs for multi-threading, Unicode, long filenames, MAPI, and more                   | X | X | X |
| Complete suite of Windows 95 common controls                                                                         | X | X | X |
| Integrated Development Environment                                                                                   | X | X | X |
| Object-oriented, fully extensible component architecture                                                             | X | X | X |
| Object Repository for storing and reusing forms, Data Modules, and DLLs                                              | X | X | X |
| Visual Form Inheritance to reduce coding and simplify maintenance                                                    | X | X | X |
| Visual Component Library (VCL) with more than 90 reusable components                                                 | X | X | X |
| Data-aware components for drag-and-drop database applications                                                        | X | X | X |
| Data Module Objects to separate business rules from application code                                                 | X | X | X |
| Database Explorer to visually browse and modify tables, aliases, and indexes                                         | X | X | X |

|                                                                           |   |   |   |
|---------------------------------------------------------------------------|---|---|---|
| FREE! Quick Reports to easily create, preview, and print embedded reports | X | X | X |
| FREE! 16-bit Delphi 1.0 for Windows 3.1 Deployment                        | X | X | X |
| Complete documentation including Object Pascal Language Guide             | X | X | X |
| VCL Source Code and manual to develop or customize components             |   | X | X |
| Scalable Data Dictionary to implement and maintain data integrity         |   | X | X |
| Complete ODBC connectivity                                                |   | X | X |
| Additional components like the Multi-Object Grid                          |   | X | X |
| 32-bit Local InterBase Server for off-line scalable SQL development       |   | X | X |
| 32-bit ReportSmith for sophisticated reporting                            |   | X | X |
| InstallShield Express for building professional installs                  |   | X | X |
| WinSight(TM) 32 for monitoring Windows messaging                          |   | X | X |
| BDE low-level API support and Help files                                  |   | X | X |
| Expanded Open Tools API to integrate your favorite tools                  |   | X | X |
| Team development interface (requires InterSolv PVCS)                      |   | X | X |
| Versatile sample OCXs for charting, graphics, spreadsheet, and            |   | X | X |

spell-checking

32-bit SQL Links native drivers, with unlimited deployment license for Oracle, Sybase, Informix, MS SQL Server, and InterBase X

InterBase NT (2-user license) for scalable multiuser SQL development X

SQL Database Explorer to visually manage server-specific meta data X

SQL Monitor to test, debug, and tune SQL applications X

Integrated InterSolv PVCS Version Manager for team development X

Visual Query Builder to generate bug-free SQL code X

Cached Updates to speed up server response time X

Data Pump Expert for rapid upsizing and application scaling X

Reference:

3/30/99 1:07:38 PM

Last Modified: 01-SEP-99

## Article #16221: TPW 1.0 Hot Sheet.

Technical Notes Database

TN1221D.txt TPW 1.0 Hot Sheet.

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

Description:

Turbo Pascal for Windows 1.0 Hot Sheet.

Feature Highlights

-----  
NEW! state-of-the-art Windows integrated development environment (IDE).

- Runs under Windows

- Multiple editor windows

- Full Mouse Support

- Supports TP6 type hot keys

- Multi-file editor that can edit files up to 1MB  
complete save and restore of desktop

NEW! ObjectWindow library - Built in support for Windows, Menus, Dialogs, Buttons, List boxes, Edit fields, Icons and more. All for use in your applications.

NEW! Full access to all Windows API functions and messages.

NEW! Fully Supports creation of DLLs.

NEW! Turbo Debugger for Windows

- Supports Windows messages

- Advanced breakpoints

- Reverse execution

- Automatic DLL debugging

- Object browser and inspectors

- Single and dual monitor support

INCLUDED! Whitewater Resource Toolkit - Visually create Dialogs,

Menus, Icons, Bitmaps and String resources.

INCLUDED! Resource Compiler

INCLUDED! Windows Help Compiler

Full featured inline assembler (BASM)

Private fields and methods in object declarations

Extended syntax directive (\$X) that lets you treat functions like procedures (and ignore function results)

286 code generation

Address references in typed constants

Far and near procedures directives (\$F)

Link in initialized data (\$L) from object (OBJ) files

Smart linker removes unused objects and code

complete math coprocessor emulation and support -  
8087/80287/80387

NEW! Turbo Help hypertext on-line help facilities, including references to all Windows API functions and messages.

Turbo Pascal for Windows includes everything you need to create Windows applications. It does not require the Microsoft Windows SDK.

## System Requirements

-----

IBM PC or PS/2 and all 100% compatibles

Microsoft Windows 3.0 or later

2Mb of memory

EGA, Hercules or VGA graphics

Mouse or other pointing device

Hard Disk (Requires 6.5 Meg for full product)

## Benchmarks

-----

Machine

Lines/Min

Compaq DeskPro 386/33            84,000

Documentation

-----

1. User's Guide
2. Programmer's Guide
3. Windows Reference Guide
4. Windows Programming Guide
5. Whitwater Resource Toolkit Users Guide
6. Turbo Debugger for Windows Users Guide
7. Help Compiler Reference Guide

Pricing Information

-----

Turbo Pascal for Windows    \$249.95

Special Offer Information

-----

Registered user of ANY BORLAND LANGUAGE PRODUCT \$99.95 \*\*\*

\*\*\*Turbo Pascal for Windows is exclusively for Windows Development and does not replace Turbo Pascal version 6.0. TP6 is the current product for standard DOS development. The offer is not an upgrade it is a limited special offer to our current Turbo Pascal Customers.

Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99



## Article #16225: Delphi Client/Server Suite 2.0 for Windows 95 & NT

Technical Information Database

TI1225D.txt - Delphi Client/Server Suite 2.0 for Windows 95 & NT

Category :Delphi 2.x  
Platform :All Windows  
Product :

### Description:

Delphi Client/Server Suite 2.0 for Windows 95 and NT  
Quick Info Guide

Delphi Client/Server Suite 2.0 provides you with all the tools you need to build scalable client and server applications while shortening every stage of the development cycle everything you need to boost productivity and gain a competitive advantage.

### Scalable client/server architecture

- \* Create fast, standalone, royalty-free scalable applications that are up to 15 to 50 times faster than p-code interpreters.
- \* Unleash the power of Windows 95 and Windows NT with complete support for OCXs, OLE Automation, multi-threading, Unicode, MAPI, and more.
- \* Increase productivity with more than 100 reusable components.
- \* Deploy royalty-free applications using the NEW! 32-bit SQL Links native drivers for Oracle, Sybase, MS SQL Server, and InterBase.(R)
- \* Visually manage server-specific meta data like stored procedures and triggers with the NEW! SQL Database Explorer.
- \* Test, debug, and tune SQL applications with the NEW! SQL Monitor.
- \* Increase server responsiveness with NEW! Cached Updates.
- \* Build and test multiuser SQL applications with the NEW! InterBase NT (2-user license).
- \* Use the NEW! Data Module Objects to separate business rules from application code, and scalable NEW! Data Dictionary to implement and maintain data integrity.
- \* Generate bug-free ANSI SQL-92 code with the Visual Query Builder.

- \* Manage complex team projects with the NEW! integrated PVCS Version Control.

-----  
Everything you need to gain a competitive advantage!

- \* High-performance, optimizing 32-bit native-code compiler
- \* Create reusable DLLs and royalty-free, standalone EXEs
- \* Full Windows 95 support for OCX controls, and OLE Automation controllers and servers
- \* Complete access to Windows 95 APIs for multi-threading, Unicode, MAPI, and more
- \* Complete suite of Windows 95 components
- \* 32-bit SQL Links native drivers, with unlimited deployment license for Oracle, Sybase, MS SQL Server, and InterBase
- \* InterBase NT (2-user lic.) for scalable multiuser SQL development
- \* SQL Database Explorer to visually manage server-specific meta data
- \* SQL Monitor to test, debug, and tune SQL applications
- \* Integrated InterSolv PVCS Version Control for team development
- \* Visual Query Builder to generate bug-free SQL code
- \* Cached Updates to speed up server response time
- \* Data Pump Expert for rapid upsizing and application scaling
- \* Object-oriented component architecture
- \* Object Repository for storing and reusing objects, business rules, and forms
- \* Visual Form Inheritance to reduce coding and simplify maintenance
- \* Integrated Development Environment
- \* Visual Component Library (VCL) with more than 100 reusable components
- \* VCL Source Code and manual to develop or customize components
- \* Sophisticated data-aware components
- \* 32-bit Borland(R) Database Engine with low-level API support and Help files
- \* Database Explorer to visually browse and modify tables and aliases
- \* Data Module Objects to separate business rules from application code for easier maintenance
- \* Scalable Data Dictionary to implement and maintain data integrity
- \* Complete ODBC connectivity
- \* 32-bit Local InterBase Server for off-line SQL development

- \* 32-bit ReportSmith(R) for sophisticated reporting
- \* InstallShield(R) Express for building professional installs
- \* WinSight(TM) to monitor Windows messages
- \* Expanded Open Tools API to integrate your favorite tools
- \* Complete documentation including Object Pascal Language Guide
- \* Includes 16-bit Delphi 1.0 for Windows 3.1 development

-----

## Delphi Facts

|                                                    | Delphi | PowerBuilder | Visual Basic |
|----------------------------------------------------|--------|--------------|--------------|
|                                                    | -----  | -----        | -----        |
| Optimizing 32-bit native-code compiler             | Y      | N            | N            |
| Create standalone EXEs and DLLs                    | Y      | N            | N            |
| Create multi-threaded Windows 95/NT applications   | Y      | N            | N            |
| Full support for OLE Automation and OCXs           | Y      | N            | Y            |
| Includes more than 100 reusable components         | Y      | N            | N            |
| Easily build or customize components               | Y      | N            | N            |
| High-performance Visual Form Inheritance           | Y      | N            | N            |
| Object Repository for forms and Data Modules       | Y      | N            | N            |
| Scalable Data Dictionary                           | Y      | N            | N            |
| Fully scalable client/server architecture          | Y      | Y            | N            |
| Fast learning curve for increased productivity     | Y      | N            | Y            |
| SQL Database Explorer to visually manage meta data | Y      | N            | N            |
| SQL Monitor to test, debug, and true SQL apps      | Y      | N            | N            |

-----

## Minimum System Requirements

- \* Intel 486/25-based PC or higher
- \* Microsoft Windows 95 or Windows NT 3.51
- \* 8Mb of memory (12Mb recommended)
- \* 50Mb hard disk space
- \* CD-ROM drive

\* Mouse or other Windows pointing device

## Borland Delphi Client/Server Suite support services

- \* Fast Fax for Detailed Information: 1-800-408-0001
- \* TechFax(TM) for Technical Information: 1-800-822-4269
- \* Connections Developer Program: 1-800-353-2211
- \* Free Install Support: (408) 461-9195
- \* Credit Card Advisor Line: 1-800-330-3372
- \* 900-Advisor Lines:
  - o Delphi and Delphi Client/Server: 1-900-555-1015
  - o Local InterBase Server: 1-900-555-1013
  - o ReportSmith: 1-900-555-1011
  - o For Technical Support Service contracts and information: 1-800-523-7070

Satisfaction guaranteed!

You can buy the Delphi Client/Server Suite with complete assurance. If for any reason you are not fully satisfied with your purchase, you can return it to Borland within 90 days. No questions asked!

Authorization required to sell these products. Call 1-800-408-0001 to request a client/server authorization form. Educational pricing is available through Borland Authorized Educational Resellers. Call 1-800-847-7797. Advisor Line charges: \$2.00 per minute, first minute free.

Borland's DELPHI products and services are not associated with or sponsored by Delphi Internet, an online service and Internet access provider.

Reference:

3/30/99 1:10:37 PM

Last Modified: 01-SEP-99

## Article #16227: Status line in a MDI Application.

Technical Notes Database

TN1227D.txt     Status line in a MDI Application.

Category     :Pascal

Platform     :All

Product      :Turbo Pascal  1.0

### Description:

To have a status line in an MDI application, override the windows MDI WMSize method like this:

```
TGDIDemoWindow.WMSize(var Message: TMessage);
begin
  TMDIWindow.WMSize(Message);
  {Calculate the new size of the client area}
  MoveWindow(ClientWnd^.HWindow, size of the new client area)
end;
```

This will keep the client window from obscuring your status line or other controls you might like to have in your window.

### Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16231: Performing database queries in a background thread

Technical Information Database

TI1231D.txt - Performing database queries in a background thread

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This document explains how to perform queries in a background thread by using the TThread class. For information on the general usage of the TThread class, please refer to the Borland documentation and to the online help. You should be aware of how to use Delphi's database components to understand the this TI's contents.

Two requirements must be met in order to perform a threaded query. First, the query to be threaded must be contained within its own session by using a separate TSession component. Therefore, you would place a TSession component on your form and assign it's name to the SessonName property of the TQuery component to be used in the thread. You must use a separate TSession component for each TQuery component to be used in a thread. If you are also using a TDataBase component, a separate TDataBase must be used for each threaded query as well. The second requirement is that the threaded TQuery component must not be connected to a TDataSource in the context of the thread in which it will be executed. This must be done in the context of the primary thread.

The code example below illustrates this process. This unit shows a form which contains two each of the following comopnents: TSession, TDatabase, TQuery, TDataSource and TDBGrid. These components have the following property settings:

```
Session1
    Active True;
    SessionName "Ses1"
```

```

DataBase1
    AliasName      "IBLOCAL"
    DatabaseName   "DB1"
    SessionName    "Ses1"

Query1
    DatabaseName   "DB1"
    SessionName    "Ses1"
    SQL.Strings    "Select * from employee"

DataSource1
    DataSet       ""

DBGrid1
    DataSource     DataSource1

Session2
    Active        True;
    SessionName    "Ses2"

DataBase2
    AliasName      "IBLOCAL"
    DatabaseName   "DB2"
    SessionName    "Ses2"

Query2
    DatabaseName   "DB2"
    SessionName    "Ses2"
    SQL.Strings    "Select * from customer"

DataSource2
    DataSet       ""

DBGrid1
    DataSource     DataSource2

```

Notice that the DataSet property for both TDataSource components do not refer to anything initially. This will be set at run-time as illustrated in the code.

```

unit Unit1;

interface

```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls, Grids, DBGrids, DB, DBTables;
```

```
type
```

```
TForm1 = class(TForm)
  Session1: TSession;
  Session2: TSession;
  Database1: TDatabase;
  Database2: TDatabase;
  Query1: TQuery;
  Query2: TQuery;
  DataSource1: TDataSource;
  DataSource2: TDataSource;
  DBGrid1: TDBGrid;
  DBGrid2: TDBGrid;
  GoBtn1: TButton;
  procedure GoBtn1Click(Sender: TObject);
end;
```

```
TQueryThread = class(TThread)
private
  FSession: TSession;
  FDatabase: TDatabase;
  FQuery: TQuery;
  FDataSource: TDataSource;
  FQueryException: Exception;
  procedure ConnectDataSource;
  procedure ShowQryError;
protected
  procedure Execute; override;
public
  constructor Create(Session: TSession; DataBase:
    TDatabase; Query: TQuery; DataSource: TDataSource);
    virtual;
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```



```

constructor TQueryThread.Create(Session: TSession; DataBase:
  TDatabase; Query: TQuery; Datasource: TDataSource);
begin
  inherited Create(True);    // Create thread in a
suspendend state
  FSession := Session;      // connect all private fields
  FDatabase := DataBase;
  FQuery := Query;
  FDataSource := Datasource;
  FreeOnTerminate := True;  // Have thread object free
itself when terminated
  Resume;                   // Resume thread execution
end;

procedure TQueryThread.Execute;
begin
  try
    { Run the query and connect the datasource to the TQuery
      component by calling ConnectDataSource from main
      thread (Synchronize used for this purpose)}
    FQuery.Open;
    Synchronize(ConnectDataSource);
  except
    { Capture exception, if one occurs, and handle it in the
      context of the main thread (Synchronize used for this
      purpose. }
    FQueryException := ExceptObject as Exception;
    Synchronize(ShowQryError);
  end;
end;

procedure TQueryThread.ConnectDataSource;
begin
  FDataSource.DataSet := FQuery; // Connect the DataSource
to the TQuery
end;

procedure TQueryThread.ShowQryError;
begin
  Application.ShowException(FQueryException); // Handle the
exception
end;

procedure RunBackgroundQuery(Session: TSession; DataBase:

```

```

TDataBase; Query: TQuery; DataSource: TDataSource);
begin
  { Create a TThread instance with the various parameters. }
  TQueryThread.Create(Session, Database, Query, DataSource);
end;

```

```

{$R *.DFM}

```

```

procedure TForm1.GoBtn1Click(Sender: TObject);
begin
  { Run two separate queries, each in their own thread }
  RunBackgroundQuery(Session1, DataBase1, Query1,
Datasource1);
  RunBackgroundQuery(Session2, DataBase2, Query2,
Datasource2);
end;

```

```

end.

```

The TForm1.GoBtn1Click method is an event handle for a button click event. This event handler calls the RunBackgroundQuery procedure twice, each time passing a different set of database components. RunBackgroundQuery creates a separate instance of the TQueryThread class, passing the various database components to its constructor which in turn assigns them to the appropriate TQueryThread private data fields.

The TQueryThread contains two user-defined procedures: ConnectDataSource and ShowQryError. ConnectDataSource connects FDataSource.DataSet to FQuery. However, it does this in the primary thread by using the TThread.Synchronize method. ShowQryError handles the exception in the context of the primary thread, again by using the Synchronize method. The Create constructor and Execute method are explained in the code's comments.

Reference:

3/30/99 11:18:31 AM

Last Modified: 01-SEP-99

## Article #16233: Changing Button Text

Technical Notes Database

TN1233D.txt Changing Button Text

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

Description:

Question:

How do you change the text on a button?

Answer:

To change the text on a button, send a WM\_SETTEXT message with the string for the lParam. For example:

```
SendMessage(btnWindow, WM_SETTEXT, 0, 'New Button Title');
```

Or if you are using OWL, call the SetCaption method of TButton.

Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16237: How to check a ComboBox without OnClick occurring.

Technical Information Database

TI1237D.txt How to check a ComboBox without OnClick occurring.

Category :VCL

Platform :All

Product :Delphi All

### Description:

When you modify the state of a TCheckBox by setting the value of its Checked property, the OnClick event of that TCheckBox is fired. For example, the following code:

```
CheckBox1.Checked := True;
```

causes CheckBox1.OnClick to execute.

Sometimes, however, you may want to check or uncheck the CheckBox in code without firing the OnClick event. You can do this by sending a BM\_SETCHECK message to the ComboBox. The WParam of this message can be 1 (meaning the box should be checked), or 0 (meaning the box should be unchecked). The LParam of this message is always 0.

With this in mind, the following procedure accepts as parameters a TCheckBox called CB and a Boolean parameter called CheckIt. When CheckIt is True, the box will be checked, and when it's False, the box will be unchecked:

```
procedure CheckNoClick(CB: TCheckBox; CheckIt: Boolean);
begin
  CB.Perform(BM_SETCHECK, Ord(CheckIt), 0);
end;
```

Using this procedure, the code for checking a TComboBox called ComboBox1 looks like this:

```
CheckNoClick(ComboBox1, True);
```

Reference:

7/16/98 4:34:06 PM

Last Modified: 01-SEP-99

## Article #16239: TDW popping in dual mode in TPW.

Technical Notes Database

TN1239D.txt TDW popping in dual mode in TPW.

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

Description:

Question:

I want to make TDW come up in dual mode from within the TPW.

Answer:

Add the following lines to the tpw.ini file:

```
[Debugger]
Exepath=
Switches=
```

where is the path to the TDW.EXE and  
are the options you normally want to use with TDW, this will do  
it. The  
command-line option for the dual display is -do.

Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16243: TForm.MDIChildren[] Array and Form Creation

Technical Information Database

TI1243D.txt TForm.MDIChildren[] Array and Form Creation

Category :VCL

Platform :All

Product :Delphi All

### Description:

The documentation of TForm.MDIChildren[] states that the index of the first-created MDI child is 0. This is incorrect -- the index of the most-recently-created MDI child is always 0, and the index of the first-created MDI child is always MDIChildCount - 1.

With this in mind, you can use the following code to iterate over the MDI child array from from the first-created to the last:

```
procedure TForm1.IterateOverMDIChildren;
var
  i: integer;
begin
  for i := MDIChildCount - 1 downto 0 do begin
    { do something with MDI child here }
  end;
end;
```

### Reference:

7/16/98 4:34:06 PM

Last Modified: 01-SEP-99



## Article #16245: PChars in TPW.

Technical Notes Database

TN1245D.txt PChars in TPW.  
Category :Pascal  
Platform :All  
Product :Turbo Pascal 1.0

### Description:

PChar is a special pointer type that has been added to the Pascal Windows language definition. A PChar is a pointer to an array of characters with a maximum size of 64K and terminated by a null character (#0). This is a C string, and comes with all the memory management hassles and pointer arithmetic advantages of C. The PChars and arrays of char are type compatible, and a Strings unit is devoted to C-string manipulation functions (all start with "STR"). When passing a string literal to a procedure, the compiler will figure out whether the string literal should be stored in c-string format or in pascal string format, or both.

### Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16251: WinCrt and Windows 3.0

Technical Notes Database

TN1251D.txt WinCrt and Windows 3.0

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

### Description:

WinCrt is an emulation of the Dos and Crt units. Basically, it provides a quick and dirty means of output to a Windows window. It's not intended to be a Dos to Windows porting service - WinCrt apps are not full Windows apps - just a Windows window with some text, key scanning loops, and scrollbars.

Note that writelns to the 'screen' will cause a runtime error in TPW unless the WinCrt unit is used. (WinCrt opens the standard Input and Output files, which are closed by default in TPW). The Readln, Keypressed, procedures, etc., are also only valid under the WinCrt umbrella. The text colors and text inverse are not supported. And since WinCrt takes control of the message loop and dispatch services of your windows app, WinCrt cannot be used in conjunction with OWL. You can call Windows API functions directly, however.

### Reference:

Last Modified: 01-SEP-99

## Article #16257: Turbo Pascal for Windows New and GetMem procedures

Technical Notes Database

TN1257D.txt Turbo Pascal for Windows New and GetMem procedures

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

### Description:

The way memory allocation works for new and getmem is as follows:

There are two system variables: HeapBlock and HeapLimit.

When memory is allocated for a size less than HeapBlock (Default is 1k)

then the memory will be suballocated in a block of size HeapBlock (default is 8k).

Allocation of blocks larger than HeapLimit will have there own block. All allocations will be in global memory.

You will not be able to change the way this works.

### Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16263: Exec from within a Turbo Pascal for Windows progra

Technical Notes Database

TN1263D.txt    Exec from within a Turbo Pascal for Windows progra  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal  1.0

Description:

Question:

I want to be able to EXEC a DOS or Windows program in Windows. How do I do that?

Answer:

Take a look at the WinExec procedure in your Windows Reference Guide.

You can use Exec on both WinApps and DOSApps.

Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16269: Local Heap and Global Heap

Technical Notes Database

TN1269D.txt Local Heap and Global Heap

Category :Pascal

Platform :All

Product :Turbo Pascal 1.0

### Description:

The Local Heap is very small, and not used by TPW, and is occasionally used by Windows functions (like edit controls).

The Global Heap is generally very big, and is where your pascal program's "heap" is stored. The GetMem, FreeMem, New, and Dispose procedures all operate on the Global Heap.

### Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16275: Turbo Pascal for Windows Strings literals in the D

Technical Notes Database

TN1275D.txt Turbo Pascal for Windows Strings literals in the D  
Category :Pascal  
Platform :All  
Product :Turbo Pascal 1.0

### Description:

Unlike the DOS version of Pascal, String Literals are stored in the data segment along with all global variables etc. If customer is running out of data segment space, suggest using pointers to strings instead of string literals.

clb 4/8/91

### Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16279: Redistributing Applications using the ISP

Technical Information Database

TI1279D.txt - Redistributing Applications using the ISP

Category :Delphi 2.x  
Platform :All Windows  
Product :Delphi2.x,

### Description:

Applications built with the Internet Solutions Pack can be redistributed using most third party installation programs, including the InstallShield Express utility that ships with Delphi 2.0 Client/Server and Developer. In order to deploy your application, any OCX controls you used in building the application need to be redistributed along with your executable, as well as any other necessary files (eg. dll's) your program requires.

In addition, the Internet controls are dependent upon NetMasters support dll's, and must be included with your program. The following list contains of all the OCX files and NetMasters DLLs that ship with the Internet Solutions Pack (and that may need to be redistributed):

#### OCX files

-----

HTML.OCX  
HTTPCT.OCX  
NNTPCT.OCX  
POPCT.OCX  
SMTPCT.OCX  
WINSCK.OCX  
FTPCT.OCX

#### NetMasters DLLs

-----

NMOCOD.DLL  
NMSCKN.DLL  
NMF'TPSN.DLL  
NMW3VWN.DLL



NMORENU.DLL

For a list of which files you actually need to include with your distribution, use the following dependency matrix:

|             | Winsock | FTP | NNTP | SMTP | POP3 | HTTP | HTML |
|-------------|---------|-----|------|------|------|------|------|
| NMSCKN.DLL  | X       | X   | X    | X    | X    | X    | X    |
| NMORENU.DLL |         | X   | X    | X    | X    | X    | X    |
| NMOCOD.DLL  |         | X   | X    | X    | X    | X    | X    |
| WINSCK.OCX  | X       |     |      |      |      |      |      |
| FTPCT.OCX   |         | X   |      |      |      |      |      |
| NMFTPSN.DLL |         | X   |      |      |      |      |      |
| NNTPCT.OCX  |         |     | X    |      |      |      |      |
| SMTPCT.OCX  |         |     |      | X    |      |      |      |
| POPCT.OCX   |         |     |      |      | X    |      |      |
| HTTPCT.OCX  |         |     |      |      |      | X    |      |
| HTML.OCX    |         |     |      |      |      |      | X    |
| NMW3VWN.DLL |         |     |      |      |      |      | X    |

Patches:

The following patches are available for download from the Borland Delphi 32 CompuServe Forum (BDELPHI32) and the Borland website ([www.borland.com](http://www.borland.com)).

NMPATCH.ZIP - Unzip this file and follow the included directions. This patch was supplied by NetMasters and will actually update the controls themselves.

ISP.DCU - Place in the Delphi 2.0\lib subdirectory and recompile

your project. This will prevent the error message about not being licensed.

To deploy your program using InstallShield Express, follow these steps:

- 1) Start up ISXpress, and build your Setup project as you normally would (if you are unfamiliar with building a Setup project with InstallShield Express, refer to the InstallShield online help).
- 2) In the Components & Groups portion of the setup, add your executable file into an appropriate group (eg. the Program Files group (default)).
- 3) Create a new group for the shared files (eg. "Shared Files"), and set the Destination Directory to . Then launch the Explorer and drag the necessary files (as determined by using the above matrix) to the newly created folder. These files should be located in your Windows\System directory on your hard drive (if not, you can find them on the Delphi 2.0 CD-ROM under \RUNIMAGE\DELPHI20\WINDOWS\SYSTEM32).
- 4) Finish the rest of the Setup procedure.

You are now ready to deploy your application. InstallShield will automatically register all OCX's and DLLs that have set as their destination directory.

Reference:

3/30/99 12:56:20 PM

Last Modified: 01-SEP-99

## Article #16281: Turbo Pascal for Windows "Can't find Debugger" err

Technical Notes Database

TN1281D.txt Turbo Pascal for Windows "Can't find Debugger" err  
Category :Pascal  
Platform :All  
Product :Turbo Pascal 1.0

### Description:

With Turbo Pascal for Windows you can get the Can't Find Debugger error message when there is not enough memory to load the debugger. There must be at least 250K free in order to load the debugger.

If you have checked that TDW is in the path etc., and this error is still displayed, check the memory free...

clb 4/8/91

### Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16285: How to Create a TDBGrid Lookup Field in 32-bit Delphi

Technical Information Database

TI1285D.txt - How to Create a TDBGrid Lookup Field in 32-bit Delphi

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

1. Drop 2-TTable's, 1-TDataSource and 1-TDBGrid on a form.
2. Connect Table1 to DataSource1 to DBGrid1
  - a. DataSource1.DataSet = Table1
  - b. DBGrid1.DataSource = DataSource1
3. Setup Table1
  - a. Table1.Database = DBDemos
  - b. Table1.TableName = Customer
  - c. Table1.Active = True
4. Setup Table2
  - a. Table2.Database = DBDemos
  - b. Table2.TableName = Orders
  - c. Table2.Active = True
5. Add all of the fields for Table1 by bringing up the Fields Editor:
  - a. Double click on Table1
  - b. Right click on Fields Editor
  - c. Add New Fields. Add all of them
6. Add a new field for Table1.
  - a. Right click on Fields Editor, and select New Field.
7. Specify the following parameters for the newly added field.
  - a. Name: Bob
  - b. Type: String
  - c. Size: 30
  - d. Select Lookup
  - e. Key Fields: CustNo - Field in Table1 to store value
  - f. DataSet: Table2 - Table lookup is being done on

g: LookUpKeys: CustNo - This Key gets copied to KeyField  
h: Result Field: OrderNo - Value to display to the user in the  
drop down box

8. Run the application

Reference:

3/30/99 12:17:42 PM

Last Modified: 01-SEP-99

## Article #16287: Timer Program Example

Technical Notes Database

TN1287D.txt Timer Program Example

Category :Pascal

Platform :All

Product :Turbo Pascal 5.5

Description:

Example of a timer program in Turbo Pascal 5.5.  
It uses the system tick to calculate the time.

Program Timer;

uses CRT;

```
var
  Starttime:longint;\
  SystemTimer: Longint absolute $0040:$006C
```

```
procedure StartTimer;
```

```
begin
  Starttime:=SystemTimer;
end;
```

```
function TimePassed:real;
begin
  Timepassed := (SystemTimer-Starttime)/18.206759747;
end;
```

```
begin
  clrscr;
  StartTimer;
  repeat
    gotoxy(10,10);
    write(TimePassed:15:10);
  until keypressed;
end.
```

11/29/90clb

Reference:

7/16/98 4:35:43 PM

Last Modified: 01-SEP-99

## Article #16297: Navigating a MultiselectedListbox

Technical Information Database

TI1297D.txt Navigating a MultiselectedListbox

Category :VCL

Platform :All

Product :Delphi All

Description:

Navigating a multiselectedListbox

This example shows a message for every element in a listbox that has been selected by the user.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Loop: Integer;
begin
  for Loop := 0 to Listbox1.Items.Count - 1 do begin
    if Listbox1.Selected[Loop] then
      ShowMessage(Listbox1.Items.Strings[Loop]);
  end;
end;
```

Reference:

7/16/98 4:34:08 PM

Last Modified: 01-SEP-99



## Article #16299: TP6 IDE Memory Documentation

Technical Notes Database

TN1299D.txt TP6 IDE Memory Documentation

Category :Pascal

Platform :All

Product :Turbo Pascal 6.0

Description:

Subject: TP6 IDE memory doc

Both the IDE and Turbo Pascal's itself are very configurable and there are several steps you can take to make more workspace available for compiling and debugging your programs. Some solutions are easy to implement, while others involve altering your code or turning off debug information selectively. Always start with the options that are painless and safe and then, if necessary, take progressively more radical steps in order to increase the IDE's capacity. Once you find a system configuration that provides you with enough capacity, you might want to permanently modify the your AUTOEXEC.BAT, CONFIG.SYS, and TURBO.TP, and TURBO.EXE files.

SCAN ITEM:

Changes you make in the Options|Environment|Startup dialog box are saved directly into TURBO.EXE and are permanent. Changes you make in other dialog settings inside the IDE can be saved in a TURBO.TP configuration file. For more information, refer to the IDE Reference Chapter xxxx.

Outside the IDE

-----

- x. Remove TSRs from memory. If you have Sidekick or Superkey loaded in memory or EMS, exit the IDE, remove them, and then reload TURBO.EXE.
- x. Modify CONFIG.SYS to remove unnecessary drivers (ANSI.SYS, disk caches, etc.). You can also reduce the number of files and buffers to: FILES = 20, BUFFERS = 20. Make sure these

changes are safe for other software you are using.

## Re-configuring Turbo Pascal

- 
- x. Set Compile|Destination to Disk.
- x. On the Options|Linker dialog box, set Link Buffer to Disk.
- x. Using the Options|Environment|Startup menu, try one or more of the following:

### SCAN ITEM:

There are command-line parameters that you can pass to TURBO.EXE at startup that correspond to all of the settings on the Options|Environment|Startup menu. Refer to P-xxx.

- x. If you have expanded memory on your system (EMS), make sure the Use e(x)panded memory option is enabled and make plenty of EMS available to Turbo Pascal (by reducing the amount of EMS being used by resident programs or drivers like RAM disks, Sidekick, etc.). The IDE can use at least 400K of EMS for overlays, extra buffers, and other system resources. All these will increase the workspace for your programs. (Making more than 400K EMS available will increase the IDE's performance, although it will not make more memory available to compile and debug your programs.)

### SCAN ITEM:

If you have EMS available, disabling this option will have no effect on IDE capacity.

- x. If you're not trying to debug a graphics program, make sure the (G)raphics screen save option is disabled. Like all startup options, you can enable this option on the command-line when you do need to debug a graphics program.
- x. Reduce the default (O)verlay and (W)indow heap sizes. Every Kbyte you subtract here yields another KByte for your program. If you have EMS available, reducing these heap sizes somewhat won't have much negative impact on the IDE's performance.
- x. Disable the Load (T)URBO.TPL option. TURBO.TPL contains the commonly used standard units and is loaded into memory

at startup to optimize linker performance. By disabling this option, you'll still be able to compile and debug programs, but you'll have to extract all the units from TURBO.TPL first (using the TPUMOVER utility. Refer to TPUMOVER.DOC on your distribution disk).

SCAN NOTE: If you don't have the IDE load TURBO.TPL, you won't be able to evaluate expressions using the Evaluate/Modify box unless a debugging session is active.

As an alternative, you can leave the Load (T)URBO.TPL option enabled and still reduce the size of TURBO.TPL by about 15K. Just extract all units from TURBO.TPL with the exception of SYSTEM.TPU. Then delete all units from TURBO.TPL with the exception of SYSTEM.TPU.

SCAN ITEM:

Make sure to leave the extracted units on disk and in your unit path (Options|Directories|Unit directory) so your programs can still make use of the Dos, Crt, Overlay and the Printer units.

- x. On a unit-by-unit basis, turn off debug information in those units that are already debugged. A common technique is to build a "test harness" around your code as you develop it. Once that code is implemented, tested and debugged, turn off symbol information in that unit by disabling the (D)ebug information switch on the Options|Compiler box and recompiling. You can also embed a {\$D-} in the unit itself. If you do so, it's a good idea to use conditional directives and defines to control enabling and disabling debug information in various units (refer to chap xxxx). If you proceed as described here and end up with debug information disabled everywhere in your program--and are still having capacity problems--consider modifying your code as described below.

SCAN ITEM:

Of course, if you're not debugging, you can greatly increase IDE capacity by disabling the Integrated debugging switch in the Options|Debugger box.

Modifying your source code

-----

Some of the following measures are easy to do and yield big

capacity gains. Others are more radical and you might want to use conditional directives (see P-xxxx) to turn them on or off.

- x. Overlay units in your program. This is very safe, flexible, and can dramatically increase the IDE workspace. Refer to chapter xxx in the Programmer's Guide for more information.
- x. Using the Options|Memory sizes menu, reduce the (S)tack size and (L)ow heap limit. Make sure there's enough stack for your program, especially if you've turned off stack checking as recommended below.
- x. Using the Options|Compiler box, try one or more of the following:
  - x. Disable (R)ange checking and (S)tack checking. (S)tack checking is on by default. Turn it off once your program is stable and you've determined its stack requirements.
  - x. Disable (E)mulation during debugging. Of course, only enable emulation and (8)087/80287 code generation if you are doing IEEE floating point. If you have a numeric coprocessor on your debugging machine, disable emulation while you're debugging non-floating point code.
- x. Reduce the number of symbols in the interface sections of units. Don't declare something in the interface section of a unit unless it's used by code outside the unit. Doing this is good, safe programming practice and will make more symbol space available during the compilation of large programs.

## Turbo Debugger and the IDE

-----

Turbo Pascal itself and the IDE both offer many ways for you to gain capacity by making adjustments to default settings. If you run out memory compiling or debugging your programs and have tried most of the painless ideas offered above, consider using the IDE to edit and compile your programs, and then using Turbo Debugger to debug them. If you have Turbo Debugger and want to use it to debug programs developed in the IDE, configure the IDE as follows:

- o Set Compile|Destination to disk.

- o In the Options|Debugger box, disable Integrated and enable Standalone debugging.

You can also use the command-line compiler, TPC.EXE, or the extended memory command-line compiler, TPCX.EXE to build massive programs (several megabytes in size). Then you can use TD, TD286 or TD386 to debug them.

11/27/90clb

Reference:

7/16/98 4:35:44 PM

Last Modified: 01-SEP-99

## Article #16303: Making your 32 bit Delphi applications "sing"

Technical Information Database

TI1303D.txt - Making your 32 bit Delphi applications "sing"

Category :Windows API  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This TI demonstrates how to make your 32 bit Delphi application "sing" by loading and playing a wave file four different ways:

- 1) Use the `sndPlaySound()` function to directly play a wave file.
- 2) Read the wave file into memory, then use the `sndPlaySound()` to play the wave file
- 3) Use `sndPlaySound` to directly play a wave file thats embedded in a resource file attached to your application.
- 4) Read a wave file thats embedded in a resource file attached to your application into memory, then use the `sndPlaySound()` to play the wave file.

To build the project you will need to:

- 1) Create a wave file called 'hello.wav' in the project's directory.
- 2) Create a text file called 'snddata.rc' in the project's directory.
- 3) Add the following line to the file 'snddata.rc':  
HELLO WAVE hello.wav
- 4) At a dos prompt, go to your project directory and compile the .rc file using the Borland Resource

compiler (brcc32.exe) by typing the path to brcc32.exe and giving 'snddata.rc' as a parameter.

Example:

```
bin\brcc32 snddata.rc
```

This will create the file 'snddata.res' that Delphi will link with your application's .exe file.

Final Note: Keep on Jamm'n!

===

```
unit PlaySnd1;
```

```
interface
```

```
uses
```

```
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
    Dialogs,  
    StdCtrls;
```

```
type
```

```
TForm1 = class(TForm)  
    PlaySndFromFile: TButton;  
    PlaySndFromMemory: TButton;  
    PlaySndbyLoadRes: TButton;  
    PlaySndFromRes: TButton;  
    procedure PlaySndFromFileClick(Sender: TObject);  
    procedure PlaySndFromMemoryClick(Sender: TObject);  
    procedure PlaySndFromResClick(Sender: TObject);  
    procedure PlaySndbyLoadResClick(Sender: TObject);  
private  
    { Private declarations }  
public  
    { Public declarations }  
end;
```

```
var
```

```
    Form1: TForm1;
```

```
implementation
```

```

{$R *.DFM}
{$R snddata.res}
uses MMSystem;

procedure TForm1.PlaySndFromFileClick(Sender: TObject);
begin
    sndPlaySound('hello.wav',
                SND_FILENAME or SND_SYNC);
end;

procedure TForm1.PlaySndFromMemoryClick(Sender: TObject);
var
    f: file;
    p: pointer;
    fs: integer;
begin
    AssignFile(f, 'hello.wav');
    Reset(f,1);
    fs := FileSize(f);
    GetMem(p, fs);
    BlockRead(f, p^, fs);
    CloseFile(f);
    sndPlaySound(p,
                SND_MEMORY or SND_SYNC);
    FreeMem(p, fs);
end;

procedure TForm1.PlaySndFromResClick(Sender: TObject);
begin
    PlaySound('HELLO',
            hInstance, SND_RESOURCE or SND_SYNC);
end;

procedure TForm1.PlaySndbyLoadResClick(Sender: TObject);
var
    h: THandle;
    p: pointer;
begin
    h := FindResource(hInstance, 'HELLO', 'WAVE');
    h := LoadResource(hInstance, h);
    p := LockResource(h);
    sndPlaySound(p, SND_MEMORY or SND_SYNC);
    UnlockResource(h);
    FreeResource(h);
end;

```



end;  
end.

Reference:

3/26/99 3:52:11 PM

Last Modified: 01-SEP-99

## Article #16305: TP 6.0 Carton Text

Technical Notes Database

TN1305D.txt    TP 6.0 Carton Text  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal  6.0

Description:  
Carton Text

Turbo Pascal with Objects  
Develop sophisticated applications easily with Turbo Vision

New Turbo Pascal 6.0  
Object-oriented programming with Turbo Vision

Turbo Pascal(R) now includes Turbo Vision, the first object-oriented application framework for MS-DOS. With Turbo Vision, your applications automatically inherit a multi-window, mouse-based user interface.

Turbo Vision is a complete library of objects including windows, pull down menus, dialog boxes and scrollbars--with built-in mouse support. It also includes collection and stream objects for data management. With Turbo Vision, even the most sophisticated applications are within reach. It's easy to learn but has power to spare for major applications. In fact, we used Turbo Vision to write Turbo Pascal's new Integrated Development Environment! So not only can you build better programs, you can do it faster than ever.

\*\* Screenshot (TP1)

Zoom through source code and eliminate bugs quickly with the new Integrated Development Environment.

Powerful Sample Applications

Turbo Vision comes with a complete set of applications including desktop tools such as a calculator, a calendar, a rolodex, a disk browser and more--all with complete source code. And since

they're written using Turbo Vision they're easy to add to your own programs or customize.

## \*\*Screenshot (TP2)

Learning Turbo Pascal is a snap. Copy, paste and compile over 200 sample programs from the Turbo Help hypertext system.

## Features:

- o Turbo Vision application framework
  - o Over 50 object types and 350 methods
  - o Windows, dialogs, menus, mouse support and more
  - o Completely extensible
- o New Integrated Development Environment
  - o Multi-file editor with macros
  - o Overlapping windows and mouse support
  - o Built-in support for Expanded Memory (EMS)
  - o Integrated source code debugging
  - o Turbo Help hypertext system
  - o Conditional breakpoints and CPU window
- o New inline assembler for low-level programming
- o Turbo Pascal 6.0 has all the power and speed you need
  - o Fast! Compiles 50,000 lines per minute\*
  - o Object-oriented programming now supports private methods and data
  - o Smart linker removes unused code, objects
  - o Fast overlays for large programs
  - o 8087 emulation and coprocessor support
  - o BGI graphics for fonts, shapes, charts and more
  - o Source code compatible with Turbo Pascal 5.x

## \*\* Mouse type:

\*On IBM PS/2 Model 80

Minimum system requirements" For the IBM PS/2(R) and IBM(R) family of personal computers and all 100% compatibles. PC-DOS (MS-DOS) 2.0 or later. 512K RAM minimum (256K for command line version). Two floppy disk drives. BGI graphics supports VGA, EGA, CGA, MCGA, Hercules, ATT400 and IBM 8514 monitors.

Turbo Pascal is a registered trademark of Borland International, Inc. Copyright (C) 1990 Borland International, Inc. All rights reserved.

\*\*\*\*\*

Turbo Pascal Professional  
Develop professional applications with Turbo Vision

New Turbo Pascal Professional 6.0

Includes:

- o Turbo Pascal(R) 6.0 with Turbo Vision
- o Protected-mode command-line compiler
- o Turbo Debugger(R)
- o Turbo Profiler TM
- o Turbo Assembler(R)

Turbo Pascal now includes Turbo Vision, the first object-oriented application framework for MS-DOS. With Turbo Vision, your applications automatically inherit a multi-window, mouse-based user interface.

Turbo Vision is a complete library of objects including windows, pull down menus, dialog boxes and scrollbars--with built-in mouse support. It also includes collection and stream objects for data management. With Turbo Vision, even the most sophisticated applications are within reach. In fact, we used Turbo Vision to write Turbo Pascal's new Integrated Development Environment! So not only can you build better programs, you can do it faster than ever.

\*\* Screenshot (TP1)

Zoom through multiple source code files with the new Integrated Development Environment.

Turbo-Driven Compiler

Turbo Pascal Professional now includes a protected mode command-line compiler that allows you to take full advantage of extended memory and create programs several megabytes in size.

Turbo Debugger--The Best Gets Better

Turbo Debugger 2.0 supports object-oriented programming in Turbo Pascal 6.0. It's the only debugger to support reverse

execution, which allows you to step backwards to find bugs you may have missed.

## \*\* Screenshot (TP PRO)

Use Turbo Debuggers' object browsers and inspectors to track down the toughest bugs.

## Turbo Profiler

The world's first interactive profiler helps you tune your programs for maximum speed. Turbo profiler measures the performance of your programs by showing you where time is spent in your routines, overlays, interrupts and file I/O.

## Turbo Assembler

The world's fastest 100% MASM-compatible assembler produces optimized code for the 8086/286/386/486.

## Turbo Pascal

- o New Turbo Vision application framework
- o New protected mode compiler for large programs
- o New built-in assembler (TASM and MASM compatible)
- o New integrated development environment with multiple windows, macros and mouse support
- o Object-oriented programming now supports private data and methods
- o Fast! Compiles 50,000 lines per minute\*
- o Includes smart linker, fast overlays, 8087 support, Turbo Help system and BGI graphics

## Turbo Debugger

- o Reverse execution
- o Object browsers and inspectors

## Turbo Profiler

- o Histograms of program execution
- o Track call history

## Turbo Assembler

- o Multi-pass optimizing assembler supporting 486

\*\* Mouse type:

\*On IBM PS/2 Model 80

Minimum system requirements" For the IBM PS/2(R) and IBM(R) family of personal computers and all 100% compatibles. PC-DOS (MS-DOS) 2.0 or later. 512K RAM minimum (256K for command line version). Two floppy disk drives. BGI graphics supports VGA, EGA, CGA, MCGA, Hercules, ATT400 and IBM 8514 monitors.

Turbo Pascal is a registered trademark of Borland International, Inc. Copyright (C) 1990 Borland International, Inc. All rights reserved.

\*\*\*\*\*

11/27/90clb

Reference:

7/16/98 4:35:44 PM

Last Modified: 01-SEP-99

## Article #16309: Obtaining the Physical Path of a Table

Technical Information Database

TI1309D.txt - Obtaining the Physical Path of a Table

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

When a Table is referenced through an alias, the physical path is not readily available. To obtain this path, use the `DbiGetDatabaseDesc` BDE function. This function takes the alias name and a pointer to a `DBDesc` structure. The `DBDesc` structure will be filled with the information pertaining to that alias. This structure is defined as:

```
pDBDesc = ^DBDesc;  
DBDesc = packed record          { A given Database Description }  
    szName          : DBINAME;   { Logical name (Or alias) }  
    szText          : DBINAME;   { Descriptive text }  
    szPhyName       : DBIPATH;   { Physical name/path }  
    szDbType        : DBINAME;   { Database type }  
end;
```

The physical name/path will be contained in the `szPhyName` field of the `DBDesc` structure.

Possible return values for the `DBIGetDatabaseDesc` function are:

|                                 |                                                                               |
|---------------------------------|-------------------------------------------------------------------------------|
| <code>DBIERR_NONE</code>        | The database description for <code>pszName</code> was retrieved successfully. |
| <code>DBIERR_OBJNOTFOUND</code> | The database named in <code>pszName</code> was not found.                     |

The code example below illustrates how to obtain the physical path name of a `TTable` component using the `DBDemos` alias:

```
var  
    vDBDesc: DBDesc;  
    DirTable: String;  
begin
```

```
    Check(DbiGetDatabaseDesc(PChar(Table1.DatabaseName),
vDBDesc));
    DirTable := Format('%s\s', [vDBDesc.szPhyName, Table1.TableName]);
    ShowMessage(DirTable);
end;
```

Reference:

3/30/99 11:24:02 AM

Last Modified: 01-SEP-99



## Article #16311: TP6 Benchmarks

Technical Notes Database

TN1311D.txt    TP6 Benchmarks  
Category     :Pascal  
Platform     :All  
Product      :Turbo Pascal  6.0

Description:

TP6 Benchmarks  
August 23, 1990

Here are the timings for a full build of TCALC (as shipped on TP55 disks) using TPC.EXE version 6.0:

| Machine               | Mhz | Disk Type | Seconds | Lines Per Minute |
|-----------------------|-----|-----------|---------|------------------|
| -----                 | --- | ----      | -----   | -----            |
| IBM PS/2 Model 60     | 8   | RAM       | 14.7    | 34,114           |
| Toshiba 5200          | 25  | Fixed     | 11.3    | 44,378           |
| Toshiba 5200          | 25  | RAM       | 5.2     | 96,438           |
| Compaq DeskPro 386/33 | 33  | Fixed     | 5.9     | 84,996 **        |
| Compaq DeskPro 386/33 | 33  | RAM       | 2.9     | 172,924          |

Note that all components reside on the disk type indicated (compiler, sources, TPUs and EXE that are output). Here's the DOS commandline that performs the compilation:

```
tpc /q /b tcalc
```

\*\* indicates the figure we will publish (85 K LPM)

-----

Reference:

7/16/98 4:35:44 PM



## Article #16315: Making accelerators work with a TPageControl

Technical Information Database

TI1315D.txt - Making accelerators work with a TPageControl

Category :VCL  
Platform :All-32Bit  
Product :All32Bit,

### Description:

The TPageControl found on the Win95 page of the Component Palette does not currently work with accelerators. However, it is possible to create a descendant of TPageControl which includes this feature.

The component shown in the code to follow is such a control. This TPageControl descendant captures the CM\_DIALOGCHAR message. This allows you to capture key strokes that may be accelerators for a given form. The CMDialogChar message handler uses the IsAccel function to determine if the captured key code refers to an accelerator on one of the TPageControls pages. If so, the appropriate page is made active and is given focus.

```
unit tapage;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComCtrls;

type

  TAPageControl = class(TPageControl)
  private
    procedure CMDialogChar(var Msg: TCMDialogChar); message
      CM_DIALOGCHAR;
  end;

procedure Register;
```

implementation

```
procedure TAPageControl.CMDDialogChar(var Msg: TCMDDialogChar);
var
  i: Integer;
  S: String;
begin
  if Enabled then
    for I := 0 to PageCount - 1 do
      if IsAccel(Msg.CharCode, Pages[i].Caption) and
        Pages[I].TabVisible then begin
        Msg.Result := 1;
        ActivePage := Pages[I];
        Change;
        Exit; // exit for loop.
      end;
    inherited;
  end;

procedure Register;
begin
  RegisterComponents('Test', [TAPageControl]);
end;

end.
```

Reference:

3/26/99 11:54:06 AM

Last Modified: 01-SEP-99

## Article #16317: TP6 Marketing QNAs

Technical Notes Database

TN1317D.txt TP6 Marketing QNAs

Category :Pascal

Platform :All

Product :Turbo Pascal 6.0

Description:

Turbo Pascal 6.0  
Questions and Answers  
Draft 3 - Oct 22, 1990

### I. General Questions

#### A. Who is the target audience for Turbo Pascal 6.0?

Turbo Pascal 6.0 is targeted to the hundreds of thousands of MS-DOS application programmers who want to create professional looking applications but don't have time to write the code from scratch. Turbo Pascal 6.0 with Turbo Vision brings sophisticated application development within reach.

#### B. What are the major new features of Turbo Pascal 6.0?

The most important features of Turbo Pascal 6.0 are the Turbo Vision object-oriented application framework and the new Integrated Development Environment (IDE). These new features will increase productivity of application programmers using Turbo Pascal. In addition, Turbo Pascal Professional now includes a Turbo Drive compiler that can compile much larger programs using extended memory.

#### C. What is new in the Integrated Development Environment (IDE)?

The new Integrated Development Environment includes multi-file, multi-window editing with full mouse support and a macro language for customization. You can also set conditional breakpoints and view a register window while debugging. The new IDE includes Turbo Help, a hyper text system with help on the entire system as well as over 200 sample programs that can be automatically cut and paste

into an editor and compiled.

D. What has changed in the compiler from version 5.5 to 6.0? In addition to Turbo Vision and the new Integrated Development Environment (IDE) several changes have been made to the Pascal compiler itself in Turbo Pascal 6.0 to give programmers a more powerful development system. We have added inline assembler so that you can mix Pascal and assembly language easily. Turbo Pascal 6.0 now also generate 80286 code to take advantage of instructions on 80286 and 80386 processors for faster performance. The compiler supports privacy in objects. This allows you to declare object methods and fields as private therefore providing better encapsulation. Also, there is an extended syntax compiler directive ( $\$X+$ ) that allows you to ignore the return value of a function call treating thereby using a function as a procedure call. Lastly, Turbo Pascal Professional includes the Turbo Drive compiler that allows you to compile very large programs by using extended memory.

E. Does Turbo Pascal 6.0 use VROOMM?

Turbo Pascal 6.0's automatic overlay manager is based on the same technology as VROOMM (Virtual Real-Time Object-Oriented Memory Manager) used in Quattro Pro, Paradox and C++. Future versions of Turbo Pascal may incorporate all of the VROOMM technology.

F. How large a program can you create with the Turbo Drive compiler?

The Turbo Drive command line compiler can use as much extended memory as is available. Therefore you can compile programs that are several megabytes in size without removing debug information. However, the programs you create still run in real mode and therefore use Turbo Pascal's automatic overlay manager to run in 640K.

G. What is Turbo Drive?

Turbo Drive is a core Borland technology used to create applications that break the 640K DOS barrier by using extended memory. Turbo Drive is used in Turbo Pascal Professional as well as Turbo Debugger 286 and 386 and Paradox 3.5.

## II. Turbo Vision Questions

### A. What is an application framework?

An application framework is the basic architecture of your program, already developed, with all the building blocks inter-linked. Because Turbo Vision is object-oriented your applications automatically inherit a powerful, extensible user interface. The application framework includes an object library with support for event handling, user interface management and data management.

Turbo Vision is a major step forward that brings sophisticated user interfaces, including mouse support, windows and context sensitive help, to DOS programmers, without requiring complex programming. The end result is that programmers can create better programs in less time. Another advantage is that the Turbo Vision architecture is much easier to extend since the framework of every application is already in place.

### B. What type of data management objects are built-in?

Turbo Pascal 6.0 includes collections objects as well as streams that allow you to create persistent objects. All of the objects included in Turbo Vision can be stored on disk automatically using streams. The supplied objects can be used as the basis for more sophisticated user-written objects. In the future, a version of the Paradox Engine will also be accessible from Turbo Pascal.

### C. How large is Turbo Vision?

Turbo Vision includes over 50 objects and 350 methods. Since Turbo Vision is written in Turbo Pascal with assembly language it compiles to about 40K for a typical application. The result is that Turbo Vision is not only productive, but extremely efficient.

### D. How will people learn to program with Turbo Vision?

We have included a tutorial as well as reference documentation on programming with Turbo Vision. The Turbo Help system includes detailed coverage of Turbo Vision. There are also many sample programs included on disk. Programmers new to object-oriented programming may wish to refer to many of the third party books that feature more detailed tutorials on Turbo Vision programming.

E. Are there any royalty fees or runtime charges with Turbo Vision?

No. Programs written using Turbo Vision can be distributed without royalty fees or runtime charges.

F. Are there other application frameworks for MS-DOS?

Turbo Vision is the first application framework designed for MS-DOS. Other systems, such as MacApp on the Macintosh, are available on graphical systems.

G. How does Turbo Vision effect third party vendors?

Turbo Vision provides a solid foundation and rich architecture for third party vendors to build on. We expect that there will be a large marketplace for high-level application objects including communications, electronic mail, databases, spreadsheets, specialized financial, scientific and engineering objects and so on. All of these can be built on top of Turbo Vision to allow programmers to concentrate on their specific application domain and not on what it takes to put it on the screen and fit it into their own application architectures. Turbo Vision is the key component that will allow this kind of extensibility.

H. Will Turbo Vision be moved to the Turbo C++ platform?

Borland is committed to providing application frameworks across a variety of languages and platforms.

I. Will Turbo Vision be available for Windows?

Although the architecture and user interface of a Turbo Vision program is similar to that of a Windows program, Turbo Vision has been optimized for creating character based applications. A similar approach could be used for Windows though it would have to be optimized for a graphical environment.

### III. Other Questions

A. Is the Paradox Engine compatible with Turbo Pascal?

We are committed to making the Paradox Engine compatible with Turbo Pascal. This will give Turbo Pascal programmers additional power for creating applications that have sophisticated data management requirements.



B. Does Turbo Pascal support Microsoft Windows or OS/2 development?

Turbo Pascal 6.0 does not currently support Windows or OS/2 development. We plan to support both platforms in future releases of the product and we have committed significant development resources toward this end. We will first deliver products for Windows and follow with products for OS/2. Both Windows and OS/2 are emerging platforms that are a key part of our Turbo language strategy. Programmer's using Turbo Vision today will be creating applications that have the same architecture as is found on Windows and Presentation Manager.

C. Are Turbo Pascal objects compatible with Turbo C++?

Currently Turbo Pascal objects are not compatible with C++. We are examining ways to improve compatibility between Turbo Pascal and Turbo C++ to allow programmers to use a common object format.

D. Does Turbo Pascal support multiple inheritance?

Turbo Pascal does not support multiple inheritance and thus an object can only inherit from one ancestor object. Although multiple inheritance adds flexibility, it also adds complexity for the programmer. Single inheritance provides the greatest amount of reusability with the least amount of additional complexity.

#### IV. Pricing and Availability Questions

A. When will Turbo Pascal 6.0 be available?

Turbo Pascal 6.0 is shipping immediately!

B. What is the retail pricing for Turbo Pascal 6.0?

Turbo Pascal \$149.95

Turbo Pascal Professional \$299.95

(includes Turbo Pascal, Turbo Drive Compiler, Turbo Debugger, Turbo Assembler, Turbo Profiler)

Educational Pricing (Scholar Program)

Turbo Pascal \$39.95

C. What is the price of the Runtime Library?

The Runtime Library, including source code to Turbo Vision, is \$199.95. Current owners of the Runtime Library can upgrade for \$99.95.

D. What is the upgrade path for current Turbo Pascal customers?

We are offering registered of any version of Turbo Pascal an upgrade price of \$69 for Turbo Pascal 6.0 and \$95 for Turbo Pascal Professional.

\* \* \*

11/27/90clb

Reference:

7/16/98 4:35:44 PM

Last Modified: 01-SEP-99

## Article #16321: How to dynamically create a TPageControl

Technical Information Database

TI1321D.txt - How to dynamically create a TPageControl

Category :VCL  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This document demonstrates how to dynamically add tab sheets (TTabSheet objects) to a Windows 9x/NT Page Control (TPageControl object). Both of these objects are declared in the ComCtrls unit. So make sure that ComCtrls is listed in the 'uses' clause on your form.

### HOW TO DYNAMICALLY CREATE A PAGE CONTROL

-----

Before we get into dynamically creating tab sheets, lets first discuss how to dynamically create a PageControl (if one isn't on the form already). This is done by calling TPageControl's Create constructor with an owner parameter of Self. The Create constructor returns a object reference of the newly created Page Control object and assigns it to the 'PageControl' variable. The second step is to set PageControl's Parent property to Self. The Parent property determines where the new PageControl is to be displayed; in this case its the form itself. Here's a code snippet that demonstrates this.

```
var  
    PageControl : TPageControl;  
  
PageControl := TPageControl.Create(Self);  
PageControl.Parent := Self;
```

Note: When the form gets destroyed the Page Control and it tab sheets will be destroyed also because they are owned by the form.

## HOW TO DYNAMICALLY CREATE A TAB SHEET

---

There are two basic steps to dynamically add a new page to a PageControl. The first is to dynamically create the TTabSheet as follows:

```
var
  TabSheet : TTabSheet;

TabSheet := TTabSheet.Create(Self);
```

Then we need to give it a caption as follows:

```
TabSheet.Caption := 'Tabsheet 1';
```

And finally, the most important piece is to tell the new tab sheet which Page Control it belongs to. This is done by assigning the TTabSheet's PageControl property a TPageControl reference variable like the one created above (PageControl). Here's a code snippet that demonstrates this.

```
TabSheet.PageControl := PageControl;
```

## HOW TO DYNAMICALLY ADD A CONTROL TO A TAB SHEET

---

The key to creating and placing any control on a tab sheet is to assign it's Parent property a TTabSheet class reference variable. Here is an example.

```
var
  Button : TButton;

Button := TButton.Create(Self);
Button.Caption := 'Button 1';
Button.Parent := TabSheet;
```

For more information on the TPageControl and TTabSheet objects refer to the on-line documentation as well as look at the ComCtrls.pas file in your ..\Delphi..\SOURCE\VCL directory.

## FULL SOURCE EXAMPLE

---

```
// This code is extracted from a form with a single button on it.
```

```
unit DynamicTabSheetsUnit;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls, Buttons;
```

```
type
```

```
  TForm1 = class(TForm)  
    Button1: TButton;  
    procedure Button1Click(Sender: TObject);  
    procedure TestMethod(Sender: TObject);  
  private  
    { Private declarations }  
  public  
    { Public declarations }  
  end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
uses ComCtrls;
```

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  PageControl : TPageControl;  
  TabSheet : TTabSheet;
```

```
begin
```

```
  // Create the PageControl  
  PageControl := TPageControl.Create(Self);  
  PageControl.Parent := Self;
```

```
  // Create 1st page and associate it with the PageControl  
  TabSheet := TTabSheet.Create(Self);  
  TabSheet.Caption := 'TabSheet 1';  
  TabSheet.PageControl := PageControl;
```

```
// Create the first page

with TButton.Create(Self) do
begin
    Caption := 'Button 1';
    OnClick := TestMethod; // Assign an event handle
    Parent := TabSheet;
end;

// Create 2nd page and associate it with the PageControl

TabSheet := TTabSheet.Create(Self);
TabSheet.Caption := ' Tabsheet 2';
TabSheet.PageControl := PageControl;
end;

procedure TForm1.TestMethod(Sender: TObject);
begin
    ShowMessage('Hello');
end;

end.
```

Reference:

3/26/99 11:29:14 AM

Last Modified: 01-SEP-99

## Article #16323: TP6.0 Hot Sheet

Technical Notes Database

TN1323D.txt TP6.0 Hot Sheet

Category :Pascal

Platform :All

Product :Turbo Pascal 6.0

Description:

TP6.0 Hot Sheet

November 7, 1990

### Feature Highlights

- o NEW! state-of-the-art integrated development environment (IDE)

- Multiple overlapping windows

- Mouse Support

- Multi-file editor that can edit files up to 1MB

- Enhanced debugging facilities

- NEW! Conditional breakpoints

- Step over and trace into

- Watch expressions

- Evaluate/modify

- Call stack

- NEW! Register viewer

- complete save and restore of desktop

- o NEW! Object Oriented application framework, Turbo Vision, for use in your applications.

- o NEW! full featured inline assembler (BASM)

- o NEW! private fields and methods in object declarations

- o extended syntax directive (\$X) that lets you treat functions like procedures (and ignore function results)

- o NEW! 286 code generation
- o address references in typed

constants

- o far and near procedures directives (\$F)
- o link in initialized data (\$L) from object files
- o NEW! heap manager is faster and reduces fragmentation
- o Built in Overlay Manager based on VROOM (Virtual Run-time Object-Oriented Memory Manager) technology, used in Turbo C++, Paradox and Quattro Pro. Supports EMS.
- o Borland Graphics Interface (BGI)  
Smart linker removes unused objects and code
- o complete math coprocessor emulation and support - 8087/80287/80387
- o enhanced hypertext online help facilities, with complete cut and paste example code for every library procedure and function.
- o Online Tutorial (TPTOUR)

### System Requirements

IBM PC or PS/2 and all 100% compatibles  
PC/MS-DOS 2.0 or later  
Dual Floppy drive (hard disk recommended)  
(5.5MB of total free disk space for TP6.0 Professional)

### Memory/Storage Requirements

|                         | Memory | Storage  |
|-------------------------|--------|----------|
| Turbo Pascal 6.0        |        |          |
| IDE                     | 512K   |          |
|                         |        | >- 3.5MB |
| Command-Line compiler   | 256K   |          |
| Protected-mode compiler | 1MB    | 260K     |



(Pro Only)

Memory      Storage

Turbo Debugger  
(Pro Only)

Std.            384K

>- 1.4MB

TD286

TD386            512K

Turbo Profiler  
(Pro Only)

Std.            384K

>- 512K

TF386

512K

Turbo Assembler  
(Pro Only)

256K

360K

#### Benchmarks

| Machine               | MHz | DiskType | Seconds | Lines/Min |
|-----------------------|-----|----------|---------|-----------|
| IBM PS/2 Mod 60       | 8   | RAM      | 14.7    | 34,114    |
| Toshiba 5200          | 20  | Fixed    | 11.3    | 44,378    |
| Toshiba 5200          | 20  | RAM      | 5.2     | 96,438    |
| Compaq DeskPro 386/33 | 33  | Fixed    | 5.9     | 84,996    |
| Compaq DeskPro 386/33 | 33  | RAM      | 2.9     | 172,924   |

#### Documentation

Turbo Pascal 6.0 includes:

User's Guide

Programmer's Guide

Turbo Vision Guide

Library Reference

In addition Turbo Pascal Professional includes:

Turbo Debugger User's Guide

Turbo Profiler User's Guide

Turbo Assembler User's Guide

Turbo Assembler Reference Guide

# Turbo Assembler Quick Reference

## Pricing Information

- o Turbo Pascal 6.0 \$149.95
- o (Scholar Program \$39.95)
- o Turbo Pascal Professional 6.0 \$299.95

NEW! Turbo Driven Protected-mode compiler

Turbo Debugger

2.0

Turbo Assembler 2.0

Turbo Profiler 1.0

- o Turbo Pascal RTL \$199.95

Full RTL Source code

NEW! Full Turbo Vision Source Code

## Upgrade Information

- o Turbo Pascal x.x to Turbo Pascal 6.0 \$69.95
- o Turbo Pascal x.x to Turbo Pascal Professional \$99.95\*\*\*
- o Turbo Pascal RTL x.x to Turbo Pascal 6.0 RTL \$99.95
- o Turbo Pascal 6.0 to Turbo Pascal Professional (Disk Only)  
\$30.00

\*\*\*Turbo Pascal Professional is the preferred upgrade path. Includes TPCX.EXE the protected mode command line compiler that is not available

with the standard edition. This is the suggested upgrade even if you currently own the Turbo Debugger, Assembler and Profiler.

11/27/90clb

Reference:

7/16/98 4:35:44 PM

Last Modified: 01-SEP-99

## Article #16327: BDE Callbacks to Provide Status on Operations

Technical Information Database

TI1327D.txt - BDE Callbacks to Provide Status on Operations

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This TI demonstrates using a BDE DbiParam function to drive a progress bar during a lengthy batch move operation under the 32-bit versions of Delphi.

Additional documentation on BDE function calls can be found in the file BDE32.HLP (located in the directory where the 32 bit IDAPI is installed).

When you install a callback function to the BDE, the BDE will "call back" to a function in your application, letting your application know when certain events take place, and in some cases, allow your application to return information back to the BDE.

The BDE defines several callback types that may be installed:

during large batch operations.  
requests a response from the caller.

This TI details installing a cbGENPROGRESS callback to drive a progress bar in your application.

To do this, first call the DbiParam() function to retrieve the handle to any callback that might already be installed (and it's parameters), and save this information to a data structure. Then install your callback, replacing any previous callback installed.

You will need to pass a pointer the data structure containing

the information to the previously installed callback to the BDE when you install your callback, so when your callback function is executed, you can call the original callback (if one is installed).

The BDE will call back to your application every so often, reporting either a text message containing the number of records it has processed or how far the batch move has progressed, expressed as an integer percentage. Your callback should be able to deal with either of these situations. If the percentage field of the callback structure is greater than -1, then the percentage is correct and you are free to update your progress bar. If the percentage reported is less than zero, the callback has received a text message in the szTMsg field containing a message that includes the number of records processed. In this case, you will need to parse the text message and convert the remaining string to an integer, then calculate the percentage done and update the progress bar.

Finally, when your batch move is complete, you will need to un-register your callback, and reinstall the previous callback function (if it exists).

The following code example is based on a form containing two tables, a batch move component, a progress bar, and a button.

----- Code Follows -----

```
unit Testbc1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids, DBGrids, DB, DBTables, ComCtrls;

type
  TForm1 = class(TForm)
    Table1: TTable;
    BatchMove1: TBatchMove;
    Table2: TTable;
    Button1: TButton;
    ProgressBar1: TProgressBar;
    procedure Button1Click(Sender: TObject);
```

```
private
  { Private declarations }
public
  { Public declarations }
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
uses Bde; {Dbi Types and Procs are located here}
```

```
{$R *.DFM}
```

```
{data structure type to save previous callback info}
```

```
type TDbiCbInfo = record
  ecbType      : CBType;
  iClientData  : longint;
  DataBuffLn   : word;
  DataBuff     : pCBPROGRESSDesc;
  DbiCbFn      : pointer;
```

```
end;
type PDbiCbInfo = ^TDbiCbInfo;
```

```
{Our callback function}
```

```
function DbiCbFn(ecbType      : CBType;
                 iClientData  : Longint;
                 CbInfo       : pointer): CBRTYPE stdcall;
```

```
var
  s : string;
begin
  {Check to see if the callback type is what we expect}
  if ecbType = cbGENPROGRESS then begin
    {if iPercentDone is less than zero then extract the number}
    {of records processed from szMsg parameter}
    if pCBPROGRESSDesc(CbInfo).iPercentDone < 0 then begin
      s := pCBPROGRESSDesc(CbInfo).szMsg;
      Delete(s, 1, Pos(':', s) + 1);
      {Calculate percentage done and set the progress bar}
      Form1.ProgressBar1.Position :=
        Round((StrToInt(s) / Form1.Table1.RecordCount) * 100);
    end else
      begin
```



```

DbiCbFn);

Form1.ProgressBar1.Position := 0;
BatchMove1.Execute;

{if a previous callback exists - reinstall it else}
{unregister our callback}
if OldDbiCbInfo.DbiCbFn <> nil then
    DbiRegisterCallBack(Table2.Handle,
                        cbGENPROGRESS,
                        OldDbiCbInfo.iClientData,
                        OldDbiCbInfo.DataBuffLn,
                        OldDbiCbInfo.DataBuff,
                        OldDbiCbInfo.DbiCbFn) else
    DbiRegisterCallBack(Table2.Handle,
                        cbGENPROGRESS,
                        longint(
OldDbiCbInfo),
                        SizeOf(cbDataBuff),
cbDataBuff,
                        nil);

{lets show our success!}
Table2.Open;

end;

end.

```

Reference:

3/30/99 11:26:29 AM

Last Modified: 01-SEP-99



## Article #16329: TP6 Fact Sheet (Draft 1.0)

Technical Notes Database

TN1329D.txt TP6 Fact Sheet (Draft 1.0)

Category :Pascal

Platform :All

Product :Turbo Pascal 6.0

Description:

Fact Sheet

TURBO

PASCAL 6.0

PROFESSIONAL

TURBO PASCAL 6.0 PROFESSIONAL:

PROGRAMMING WITH VISION

New Turbo Pascal 6.0 includes Turbo Vision, the first object-oriented application framework for MS-DOS. With Turbo Vision, your applications automatically inherit a multi-window, mouse-based user interface. Turbo Pascal 6.0 also includes a new Integrated Development Environment with multiple windows and mouse support. So not only can you build better programs, you can do it faster than ever.

Turbo Pascal 6.0 Professional features Turbo Pascal 6.0 together with Turbo Debugger, the standard tool for object-oriented debugging, and Turbo Profiler and Turbo Assembler. A protected-mode Turbo Pascal compiler that frees conventional memory and lets you compile extra-large applications is included in the Professional package to complete your inventory of professional programming tools.

TURBO PASCAL 6.0 PROFESSIONAL

NEW! Turbo Vision

NEW! Integrated Development Environment

NEW! Inline assembler

NEW! Conditional breakpoints

NEW! Protected-mode compiler

Object-oriented programming extensions include:

Inheritance

Static and virtual methods

Static and dynamic objects

Constructors and destructors

NEW! Private methods and data fields

Integrated debugger with object support

Turo help hypertext system with copy and paste

High speed--compiles more than 50,000 (80,000?) lines per minute

Enhanced smart linker removes unused objects and code

Complete math coprocessor emulation and support--8087/80287/80387

State-of-the-art overlay management

#### PACKAGED APPLICATIONS WITH TURBO VISION

Turbo Vision is the first object-oriented application framework for DOS. It provides Turbo Pascal programmers with a solid architecture on which to build all their applications. Just as the same steel-girder framework can be used to build a school, an apartment house, or a hospital, so Turbo Vision provides programmers with an inheritable object hierarchy suitable for packaging applications. Programmers no longer have to reinvent features; their programs inherit them.

Turbo Vision is a professional tool. To prove it, we wrote our new IDE using Turbo Vision (and we wrote it in record time!). With the new IDE, you can write your programs, compile, run, and debug them within a single integrated environment. And you can do

it all on the leading edge of technology--with the Turbo Vision application framework.

Turbo Pascal 6.0, with Turbo Vision, OOP, units, an enhanced smart linker, and an intelligent, unit-based overlay manager give you the ability to create large and complex programs easily.

#### INTEGRATED DEBUGGING--

##### THE LEADER OF THE PACK

With Turbo Pascal, you can debug your programs without leaving the comfort of the integrated environment. Examine objects and their associated methods. Step over or single-step through each line of your program, watching your variables as they change with each statement. Set conditional breakpoints. Watch and evaluate variables, data structures, and expressions. Assign new values to your variables and data structures while debugging. With this visibility into your code, you can find and fix errors with greater ease.

#### HELP!

Turbo Pascal's indexed and cross-referenced online Turbo Help hypertext system uses hypertext to give context-sensitive help on a comprehensive range of topics. There's an example for every standard Turbo Pascal procedure and function. If you see something you'd like to use, the IDE lets you copy it from Help into your program in the editor. A separate RAM-resident program offers you the identical help from anywhere in your system (not just inside the integrated environment), and only uses 25K of memory. (Have we dropped this?)

#### EDITING MADE EASY

Automatic indenting and outdenting, search and replace, and block operations (mark, move, copy and delete) all make editing your programs faster and easier. And Turbo Pascal automatically uses expanded memory (EMS), freeing up to 400K of real memory for compilation. If you prefer to use your own editor, a command-line version of Turbo Pascal is included in your package.

#### NEW! INLINE ASSEMBLER

Sometimes you need to drop down to assembly language for extra speed or control. Turbo Pascal 6.0 makes it easy to mix Pascal and assembly language code in your programs. The ability to mix Pascal and assembler gives programmers the edge in creating fast applications in less time.

## UNITS--PERFECT YOUR PROGRAM

### PIECE BY PIECE

Units are separate source code modules, written and compiled individually, that can be called by your program or other units. Since a single unit can be used by many different programs, you can get it right once and not bother with it again. With units, you can build your own library of flawless code to use over and over again.

## OVERLAY MANAGEMENT--

### BIGGER PROGRAMS, GREATER SPEED

Feeling cramped by the 640K barrier? Turbo Pascal's improved overlay manager allows you to write programs of unlimited size. It's based on Borland's VROOMM (Virtual Realtime Object-Oriented Memory Manager) technology, used in Quattro Pro and Paradox. Using the sophisticated "least recently used" (LRU) algorithm and EMS (if available), the overlay manager runs your overlaid programs fast. And all you have to do is to call an initialization routine and specify which units may be overlaid.

## MATH COPROCESSOR SUPPORT

Turbo Pascal offers full math coprocessor emulation and support. This lets you take advantage of the high-precision, high-speed math capabilities provided by the 8087, 80287, or 80387 math coprocessor chips when present, yet your programs will work even if one isn't there.

## GRAPHICS

The Borland Graphics Interface (BGI) lets you easily use graphics and animation in your programs. Turbo Pascal's complete graphics library has more than 70 functions that you can use to draw and fill shapes, create viewports, produce multiple-font text, and handle bit-level images. BGI supports a wide variety of graphics adapters: VGA, EGA, CGA, MCGA, Hercules, ATT400, IBM 8514, and PC3270.

## CUSTOMIZABLE

Borland knows that you want to make Turbo Pascal your very own. The IDE comes with a sophisticated macro language that allows you to customize your editor command keys. The IDE's pull-down menus allow you to modify default settings, change your screen colors, and more.

## SOLID SUPPORT

When you need help, Borland's experienced Technical Support Group

is there for you. Borland's Support Group has earned a reputation in the industry as being one of the best there is. Why? Because our customers are treated with courtesy and concern, and receive sound technical knowledge from professionals well-versed in the products they support.

#### THE RUNTIME LIBRARY

Professional programmers often wish they could change standard functions and procedures slightly to better work with the application at hand. As part of our open-architecture policy, Borland offers users the Turbo Pascal 6.0 runtime library, including the source for Turbo Vision, at moderate additional cost. The runtime library source provides you with a broad range of state-of-the-art functions, including graphics, I/O, math coprocessor emulation, sorting and searching, DOS interfacing, and overlay management. You can use these routines just as they are, or you can customize them to fit your exact needs. Either way, you save valuable time by incorporating proven, professional, customizable code in your programs.

#### TURBO PASCAL 6.0 PROFESSIONAL

INCLUDES THE POWER AND PERFORMANCE OF TURBO ASSEMBLER, TURBO DEBUGGER AND TURBO PROFILER

When you have Turbo Pascal 6.0 Professional, you have the right tools to undertake your most complicated applications. Turbo Pascal, Turbo Debugger, Turbo Profiler, and Turbo Assembler work beautifully together. You don't have to do anything special to move from one to another. Write your Turbo Pascal program, debug it with Turbo Debugger, find the bottlenecks with Turbo Profiler, then link in Turbo Assembler code for optimum speed.

#### TURBO DEBUGGER 2.0

Turbo Debugger 2.0 supports object-oriented debugging in Turbo Pascal, and has the breakthrough capabilities you need to find the toughest bugs:

Undo multiple instructions with reverse execution

Debug using Turbo Drive(TM) 286 protected mode, 386 virtual mode, and remote debugging to fix programs of any size

Browse your object hierarchy; inspect classes and objects; walk through structures and records; follow linked lists

Edit with multiple overlapping windows, mouse support, icons,

dialog boxes, and scroll bars in the new user interface

Examine your program in 13 different views, including breakpoints, watch, and CPU

Catch bugs with the powerful breakpoint facility (including hardware breakpoints)

Debug TSRs and device drivers

Debug using the built-in assembler

Record and playback keystrokes

Use in 25-, 43-, or 50-line mode

#### OBJECT DEBUGGING

In addition to letting you examine and debug data structures, Turbo Debugger offers a feature unique among debuggers: You can debug objects as well. This means you can browse through your all of your data structures and objects and view their contents, follow pointers, walk down linked lists, and examine methods. You can even change the value of structure and object members. Turbo Debugger allows you to debug object instances as easily as data structures. Turbo Debugger's hierarchy browser gives you a high-level view of your object hierarchy, its easy to find the source of bugs inherited by objects.

#### YOU ARE IN CONTROL!

Turbo Debugger can directly undo the execution of CPU instructions or source lines to help you to find the most difficult bugs. In addition, Turbo Debugger provides keystroke recording and playback, so you can find bugs triggered by long and involved sequences of user input.

#### DEBUG PROGRAMS OF ANY SIZE

286 protected mode: Load Turbo Debugger itself into extended memory and use only 70K of real memory.

386 virtual machine mode: Run Turbo Debugger entirely from extended memory, giving every all of conventional RAM to the program you're debugging.

Remote debugging: Run Turbo Debugger on one computer to debug

programs on another, using only 15K on the target system.

EMS support: Take advantage of EMS memory for symbol table caching, leaving more room for your program.

Two monitors on one machine: Run Turbo Debugger on one screen, your program on the other. This method is especially useful for debugging programs that contain graphics.

#### POWERFUL BREAKPOINTS GIVE YOU GREATER CONTROL OVER YOUR DEBUGGING SESSIONS

You can define Turbo Debugger's breakpoints in three ways: where the breakpoint occurs, the condition under which the break is triggered, and what happens when the break occurs. {Shall we drop this paragraph?}

#### ASSEMBLER AND SOURCE LEVEL POWER

Debug at the source code level using the module window. If you prefer, debug at the assembler level, by opening the CPU window. Open both windows to debug your program's source and machine code side by side.

#### TURBO PROFILER 1.0 BEYOND OPTIMIZATION

Turbo Profiler takes you beyond optimization by giving you the information you need to optimize your algorithms and data structures. An optimizing compiler generates the fastest possible executable code for the program you give it. Turbo Profiler goes one step further by helping you enhance the design of the program for maximum performance.

#### PROFILE THE EXECUTION OF YOUR PROGRAMS

Turbo Profiler gives you a profile of the execution of your program, graphically displaying where your program is spending its time. It interactively measures your program, allowing you to identify bottlenecks in your code and fine-tune the design for optimal performance. Turbo Profiler brings performance analysis to the mainstream of PC programming.

Turbo Profiler measures:

Where time is spent in your program

How many times a line is executed

How many times a routine is called and from where

What files are accessed most often and for how long

When and where resources such as (CPU, disk, keyboard, printer, and interrupts) are consumed

#### EASY TO USE

With its multiple overlapping windows, icons, mouse support, and context-sensitive online help, Turbo Profiler's intuitive user interface lets you become productive immediately, whether you've used a profiler before or not. Using easy-to-read histograms, Turbo Profiler makes it clear where your program is consuming time.

#### PROFILE PROGRAMS OF ANY SIZE

Using the same advanced technology as Turbo Debugger, Turbo Profiler can profile programs of any size by using a 386 or remote profiling.

#### MULTIPLE VIEWS

Turbo Profiler provides many ways to see into your program. The Execution Profile view tells you how many times and how often your program's modules, functions, and individual lines are executed. You can tell Turbo Profiler which lines, routines, and modules you are interested in from the Module view. The File Activity view shows you how many times and for how long each file is accessed, and how many bytes are read from or written to it. You can view DOS interrupt and overlay activities from the Interrupt and Overlay views. A total of nine different windows allow you to thoroughly inspect your program's performance.

#### TURBO ASSEMBLER 2.0

Multi-pass optimizing assembler with NOP squishing

MASM compatible and two to three times faster than MASM

Generates assembler code for 8086, 80286, 80386, and 80486

Ideal mode supports extended syntax and MASM

CALL statement allows inclusion of arguments and optional languages



Generates Microsoft Windows-compatible code from your DOS program  
(no macros!)

Assembles multiple files at the same time

Allows named local labels, nested structures and unions

#### BETTER THAN MASM

Turbo Assembler offers a superior professional alternative to Microsoft Assembler (MASM). Turbo Assembler is compatible with MASM 4.0, 5.0, and 5.1--even more so than MASM itself. MASM 5.1 can't claim full compatibility with earlier versions of MASM--Turbo Assembler can.

#### QUIRKS MODE

MASM has what can best be described as "quirks." Anomalies. Things that don't work the way you might expect. Some programmers have learned to work with them; their code needs special support. So Turbo Assembler has a Quirks mode that fully duplicates the functioning of MASM quirks. Programs written to use those quirks run under Turbo Assembler.

#### IDEAL MODE

When you're first learning assembly language, you can benefit from Turbo Assembler's Ideal mode, with simplified parsing methods, improved consistency, and stricter type-checking. Ideal mode makes learning Turbo Assembler painless for the beginner since it's simpler to learn. And for the expert, programs are easier to read and write. You can use Standard MASM mode, Quirks mode, and Ideal mode all in the same program, taking advantage of Turbo Assembler's new features while maintaining compatibility with your existing MASM code.

#### TECHNICAL SPECIFICATIONS

##### SYSTEM REQUIREMENTS

##### COMPUTER AND STORAGE REQUIREMENTS

For the IBM PS/2, the IBM family of personal computers, and all 100% compatibles. A hard disk with a minimum of \_\_\_\_\_MB free storage is required.

## OPERATING SYSTEM

PC-DOS (MS-DOS) 2.0 or later

## MEMORY AND STORAGE REQUIREMENTS

|                                           |      | Memory | Storage  |
|-------------------------------------------|------|--------|----------|
| Turbo Pascal 6.0                          |      |        |          |
| IDE                                       |      | 512K   |          |
|                                           |      |        | >- 3.5MB |
| Command-Line<br>compiler                  | 256K |        |          |
| Protected-<br>mode compiler<br>(Pro Only) | 1MB  | 260K   |          |
|                                           |      | Memory | Storage  |
| Turbo Debugger<br>(Pro Only)              |      |        |          |
| Std.                                      |      | 384K   |          |
|                                           |      |        | >- 1.4MB |
| TD286                                     |      |        |          |
| TD386                                     | 512K |        |          |
| Turbo Profiler<br>(Pro Only)              |      |        |          |
| Std.                                      |      | 384K   |          |
|                                           |      |        | >- 512K  |
| TF386                                     | 512K |        |          |
| Turbo Assembler<br>(Pro Only)             | 256K | 360K   |          |

## A FAST OBJECT LESSON

Object-oriented applications model the way we think, and application frameworks like Turbo Vision model the way we build. Objects and object hierarchies are easy to extend and easier to maintain. Objects contain both data and code, and Turbo Vision fulfils the potential of objects. As in a spreadsheet cell, the

value and the formula are together. Objects can inherit properties from other objects, and Turbo Vision takes maximum advantage of this feature. Your applications inherit characteristics of a prepackaged application framework. Your application can have mouse support, dialog boxes, and overlapping windows, all of which are inherited from Turbo Vision.

Objects are intelligent, too. Clicking the mouse on a radio button inside of a dialog box has a different result from mousing on a zoom box or scroll bar.

Turbo Pascal's object-oriented extensions are fast, efficient, and elegant. You get the full power of inheritance, data abstraction, and late binding. For maximum flexibility, you can allocate dynamic objects referenced through pointers. For maximum efficiency, you can declare static objects and object constants that are referenced directly. Static and virtual methods can both be used in the same program for maximum performance and extensibility. Constructors and destructors make creating, initializing, and disposing of dynamic objects simple and reliable. They also reduce errors, save programming time, and improve readability.

With Turbo Pascal Objects, your code is compact, flexible, and reusable. With Turbo Vision, your applications acquire a polished look without a lot of perspiration. What's more, your programs become easier to extend and easier to maintain. Object-oriented programming with Turbo Pascal and Turbo Vision is programming that makes sense.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Copyright (C) 1990 Borland International, Inc. All rights reserved.

#### CORPORATE HEADQUARTERS

1800 GREEN HILLS ROAD, P.O. BOX 660001  
SCOTTS VALLEY, CA 95066-0001, (408) 438-5300

#### BORLAND U.K.

UNIT 8 PAVILIONS, RUSCOMBE BUSINESS PARK  
TWYFORD, BERKSHIRE, RG10 9NN-ENGLAND

#### BORLAND FRANCE

B.P. NO. 6, 43 AVENUE DE L'EUROPE

78141 VELIZY VILLACOUBLAY CEDEX, FRANCE

11/27/90clb

Reference:

7/16/98 4:35:45 PM

Last Modified: 01-SEP-99

## Article #16333: Accessing Paradox Tables on CD or Read-Only Drive

Technical Information Database

TI1333D.txt Accessing Paradox Tables on CD or Read-Only Drive

Category :Database Programming

Platform :All

Product :Delphi All

### Description:

This Technical Information document will step through the concepts regarding accessing Paradox tables which are located on a CD-ROM or any read-only device.

The Paradox locking scheme requires the existence of a PDOXUSRS.LCK file to handle its locking logic. This file is generally created at run-time and resides in the directory which also contains the tables. However, with a CD-ROM there is not a way to create this file at run-time on the CD-ROM. The solution is simple, we create this file and put it on the CD-ROM when the CD is pressed. The following steps will give you a very simple utility program for creating the PDOXUSRS.LCK file which you will then copy to the CD-ROM image.

1. Starting with a blank project add the following components: TEdit, TButton and TDatabase.

2. In the OnClick event for the button use the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if ChkPath then
        Check(Db1AcqPersistTableLock(Database1.Handle,
            'PARADOX.DRO', 'PARADOX'));
end;
```

3. The ChkPath function is a user defined method of the form. It will simply check the path entered in the Edit box and make sure it exists.

Here is the function:

```

function TForm1.ChkPath : Boolean;
var
  s : array[0..100] of char;
begin
  If DirectoryExists(Edit1.Text) then begin
    DataBase1.DatabaseName:= 'TempDB';
    DataBase1.DriverName:= 'Standard';
    DataBase1.LoginPrompt:= false;
    DataBase1.Connected := False;
    DataBase1.Params.Add('Path=' + Edit1.Text);
    DataBase1.Connected := TRUE;
    Result := TRUE;
  end
  else begin
    StrPCopy(s,'Directory : ' + Edit1.text + ' Does Not Exist');
    Application.MessageBox(s, 'Error!', MB_ICONSTOP);
    Result := FALSE;
  end;
end;

{ Note: Don't forget to put the function header in the public section
  of the form.}

```

4. There is one more thing you need to add before compiling, in the Uses statement at the top of the unit add the following units:

```

Delphi 1.0: FileCtrl, DbiProcs, DbiTypes, DbiErrs.
Delphi 2.0: FileCtrl , BDE

```

When you have compiled and executed the utility program, it will create two files in the directory you specified. The two files created are: PDOXUSRS.LCK and PARADOX.LCK.

Note: The PARADOX.LCK file is only necessary when accessing Paradox for DOS tables so you can delete it.

5. The only thing left for you to do is copy the remaining file (PDOXUSRS.LCK) to the CD-ROM image. Of course your tables will be Read-Only.

Note: If you want to clean up this utility for future use, you can

change the text property of the Edit box to be some default directory and change the Caption property of the Button to be something more meaningful.

Here is the final version of the code:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DB, StdCtrls, FileCtrl,

  {$IFDEF WIN32}
    BDE;
  {$ELSE}
    DbiProcs, DbiTypes, DbiErrs;
  {$ENDIF }

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Database1: TDatabase;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    function ChkPath : Boolean;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function TForm1.ChkPath : Boolean;
var
  s : array[0..100] of char;
```

```

begin
  If DirectoryExists(Edit1.Text) then begin
    DataBasel.DatabaseName:= 'TempDB';
    DataBasel.DriverName:= 'Standard';
    DataBasel.LoginPrompt:= false;
    DataBasel.Connected := False;
    DataBasel.Params.Add('Path=' + Edit1.Text);
    DataBasel.Connected := TRUE;
    Result := TRUE;
  end
  else begin
    StrPCopy(s,'Directory : ' + Edit1.text + ' Does Not Exist');
    Application.MessageBox(s, 'Error!', MB_ICONSTOP);
    Result := FALSE;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if ChkPath then
    Check(DbAcqPersistTableLock(DataBasel.Handle,
      'PARADOX.DRO', 'PARADOX'));
end;

end.

```

Reference:

7/16/98 4:34:09 PM

Last Modified: 01-SEP-99



## Article #16339: Synchronize a DLL to an Open Dataset

Technical Information Database

TI1339D.txt Synchronize a DLL to an Open Dataset

Category :Database Programming

Platform :All

Product :Delphi 2.0

### Description:

Synchronize a DLL to an Open Dataset

This document demonstrates how to use Object Pascal to link a DLL dynamically, on-the-fly, into an active Database, thus, giving the Developer the ability to 'Modularize' features. (Whether at Run-Time or Design-Time)

The technique of linking a DLL Dynamically into an EXE is useful in many cases. Examples include, an Accounting package which offers several 'plug-in' modules (A/R, A/P, General Ledger, etc.) or a Point of Sale package with Current Stock, FIFO/LIFO Ordering, Vendor Tracking, etc., modules.

This TI will provide a solid example of how to do this with one dll, 'Editdll.dll', but will give the Developer the essentials leading to extensive modularation in projects.

### Prerequisites:

-----  
Familiarity with the TTable component, DLL usage, BDE API, and BDE hCursor knowledge. \*WIN API for Dynamical loading of any DLL's.

### SAMPLE APPLICATION

The following form, EditForm, is based on the COUNTRY table in the DBDEMO's directory. When the user presses the 'Edit' button or double-clicks a record (row), a dialog box appears, from 'EditDll.dll',

displaying record specific information. At this point, the DLL has synchronized itself not only with the dataset (& session) but also with the current record. This mean's, the user can now modify the same data EditForm is viewing! So, with no ado, let's delve into the sample code.

(For best results, simply cut and paste into appropriate files)

```
>Main Form Project<
```

```
-----
```

```
{ MAINDB.DPR }  
program maindb;
```

```
uses
```

```
    Forms,  
    mainform in 'mainform.pas' {DBMainForm};
```

```
{ $R *.RES }
```

```
begin
```

```
    Application.Initialize;  
    Application.CreateForm(TDBMainForm, DBMainForm);  
    Application.Run;
```

```
end.
```

```
>>
```

```
{ MAINFORM.PAS }  
unit mainform;
```

```
interface
```

```
uses
```

```
    SysUtils, Windows, Messages, Classes, Graphics, Controls,  
    StdCtrls, Forms, DBCtrls, DB, DBGrids, DBTables, Grids, ExtCtrls,  
    BDE;
```

```
type
```

```
TDBMainForm = class(TForm)  
    Table1Name: TStringField;  
    Table1Capital: TStringField;  
    Table1Continent: TStringField;  
    Table1Area: TFloatField;  
    Table1Population: TFloatField;
```

```

DBGrid1: TDBGrid;
DBNavigator: TDBNavigator;
Panel1: TPanel;
DataSource1: TDataSource;
Panel2: TPanel;
Table1: TTable;
EditButton: TButton;
procedure FormCreate(Sender: TObject);
procedure EditButtonClick(Sender: TObject);
procedure DBGrid1DbClick(Sender: TObject);
private
  { private declarations }
public
  { public declarations }
end;

```

```

var
  DBMainForm: TDBMainForm;

```

```

implementation

```

```

{$R *.DFM}

```

```

procedure TDBMainForm.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;

```

```

// {NOTES: DBHandle is a Handle to the Database & DSHandle is a
cursor
// to the record being viewed. Also, if the purpose is to
dynamically
// load a DLL at run-time, use the LoadLibrary, GetProcAddress, and
// FreeLibrary API calls in place of the implicit load calls at
startup.
// An Example of the APICalling convention is: }
// Type
// {For GetProcAddress}
// BDEDataSync =
//   function(const DBHandle: HDBIDB; const DSHandle: HDBICur):
Boolean;
//           stdcall;
// {For the trapping of DLL load error's}
// EDLLLoadError = class(Exception);
// var h: hwnd;

```

```

//      p: BDEDataSync;
//      LastError: DWord;
// begin
// UpdateCursorPos;
// Try
//   h := loadLibrary('EDITDLL.DLL');
//   {NOTE to Delphi 1.0 users: Whereas Win32 LoadLibrary returns a
//   NULL if the DLL load was unsuccessful, thus requiring a call to
//   GetLastError to find the error, Win16 LoadLibrary returns
//   an error value (less than HINSTANCE_ERROR) which can be checked
//   in the Win16API SDK as to the reason for the failure.}
//   if h = 0 then begin
//       LastError := GetLastError;
//       Raise EDLLLoadError.create(IntToStr(LastError) +
//                                   ': Unable to load DLL');
//       end;
//   try
//       p := getProcAddress(h, 'EditData');
//       if p(DBHandle, Handle) then Resync([]);
//   finally
//       freeLibrary(h);
//   end;
// Except
//   On E: EDLLLoadError do
//       MessageDLG(E.Message, mtInformation, [mbOk],0);
// end;
// end;
// {or}
function EditData(const DBHandle: HDBIDB; const DSHandle: HDBICur):
    Boolean; stdcall external 'EDITDLL.DLL' name 'EditData';

procedure TDBMainForm.EditButtonClick(Sender: TObject);
begin
    with Table1 do
        begin
            UpdateCursorPos;
// Call the EditData Procedure from EditDll.dll.
            if EditData(DBHandle, Handle) then Resync([]);
        end;
    end;
end;

procedure TDBMainForm.DBGrid1DblClick(Sender: TObject);
begin
    EditButton.Click;
end;

```

end.

>>

>EDIT DLL PROJECT<

-----

```
{ EDITDLL.DPR }  
library editdll;
```

```
uses  
  SysUtils,  
  Classes,  
  editform in 'editform.pas' {DBEditForm};
```

```
exports  
  EditData;
```

```
begin  
end.
```

>>

```
{ EDITFORM.PAS }  
unit editform;
```

```
interface
```

```
uses  
  SysUtils, Windows, Messages, Classes, Graphics, Controls,  
  StdCtrls, Forms, DBCtrls, DB, DBTables, Mask, ExtCtrls, BDE;
```

```
type  
  TTableClone = class;
```

```
TDBEditForm = class(TForm)  
  ScrollBox: TScrollBox;  
  Label1: TLabel;  
  EditName: TDBEdit;  
  Label2: TLabel;  
  EditCapital: TDBEdit;  
  Label3: TLabel;  
  EditContinent: TDBEdit;  
  Label4: TLabel;
```

```

    EditArea: TDBEdit;
    Label5: TLabel;
    EditPopulation: TDBEdit;
    DBNavigator: TDBNavigator;
    Panel1: TPanel;
    DataSource1: TDataSource;
    Panel2: TPanel;
    Database1: TDatabase;
    OKButton: TButton;
private
    TableClone: TTableClone;
end;

{ TTableClone }

TTableClone = class(TTable)
private
    SrcHandle: HDBICur;
protected
    function CreateHandle: HDBICur; override;
public
    procedure OpenClone(ASrcHandle: HDBICur);
end;

function EditData(const DBHandle: HDBIDB; const DSHandle: HDBICur):
    Boolean; stdcall;

var
    DBEditForm: TDBEditForm;

implementation

{$R *.DFM}

{ Exports }

function EditData(const DBHandle: HDBIDB; const DSHandle: HDBICur):
    Boolean; stdcall;

var
    DBEditForm: TDBEditForm;
begin
    DBEditForm := TDBEditForm.Create(Application);
    with DBEditForm do
        try
// Set the handle of the Database1 to that of the currently opened

```

```

database
  Databasel.Handle := DBHandle;
  TableClone := TTableClone.Create(DBEditForm);
  try
    TableClone.DatabaseName := 'DB1';
    DataSource1.DataSet := TableClone;
    TableClone.OpenClone(DSHandle);
    Result := (ShowModal = mrOK);
    if Result then
      begin
        TableClone.UpdateCursorPos;
        DbiSetToCursor(DSHandle, TableClone.Handle);
      end;
  finally
    TableClone.Free;
  end;
finally
  Free;
end;
end;

{ TTableClone }

procedure TTableClone.OpenClone(ASrcHandle: HDBICur);
begin
  SrcHandle := ASrcHandle;
  Open;
  DbiSetToCursor(Handle, SrcHandle);
  Resync([]);
end;

function TTableClone.CreateHandle: HDBICur;
begin
  Check(DbiCloneCursor(SrcHandle, False, False, Result));
end;

end.

>>

{ EDITFORM.DFM }
object DBEditForm: TDBEditForm
  Left = 201
  Top = 118
  Width = 354

```

```
Height = 289
ActiveControl = Panel1
Caption = 'DBEditForm'
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
Position = poScreenCenter
PixelsPerInch = 96
TextHeight = 13
object Panel1: TPanel
    Left = 0
    Top = 0
    Width = 346
    Height = 41
    Align = alTop
    TabOrder = 0
    object DBNavigator: TDBNavigator
        Left = 8
        Top = 8
        Width = 240
        Height = 25
        DataSource = DataSource1
        Ctl3D = False
        ParentCtl3D = False
        TabOrder = 0
    end
    object OKButton: TButton
        Left = 260
        Top = 8
        Width = 75
        Height = 25
        Caption = 'OK'
        Default = True
        ModalResult = 1
        TabOrder = 1
    end
end
object Panel2: TPanel
    Left = 0
    Top = 41
    Width = 346
    Height = 221
    Align = alClient
    BevelInner = bvLowered
```



```
BorderWidth = 4
Caption = 'Panel2'
TabOrder = 1
object ScrollBox: TScrollBox
  Left = 6
  Top = 6
  Width = 334
  Height = 209
  HorzScrollBar.Margin = 6
  HorzScrollBar.Range = 147
  VertScrollBar.Margin = 6
  VertScrollBar.Range = 198
  Align = alClient
  AutoScroll = False
  BorderStyle = bsNone
  TabOrder = 0
  object Label1: TLabel
    Left = 6
    Top = 6
    Width = 28
    Height = 13
    Caption = 'Name'
    FocusControl = EditName
  end
  object Label2: TLabel
    Left = 6
    Top = 44
    Width = 32
    Height = 13
    Caption = 'Capital'
    FocusControl = EditCapital
  end
  object Label3: TLabel
    Left = 6
    Top = 82
    Width = 45
    Height = 13
    Caption = 'Continent'
    FocusControl = EditContinent
  end
  object Label4: TLabel
    Left = 6
    Top = 120
    Width = 22
    Height = 13
```

```
    Caption = 'Area'
    FocusControl = EditArea
end
object Label5: TLabel
    Left = 6
    Top = 158
    Width = 50
    Height = 13
    Caption = 'Population'
    FocusControl = EditPopulation
end
object EditName: TDBEdit
    Left = 6
    Top = 21
    Width = 135
    Height = 21
    DataField = 'Name'
    DataSource = DataSource1
    MaxLength = 0
    TabOrder = 0
end
object EditCapital: TDBEdit
    Left = 6
    Top = 59
    Width = 135
    Height = 21
    DataField = 'Capital'
    DataSource = DataSource1
    MaxLength = 0
    TabOrder = 1
end
object EditContinent: TDBEdit
    Left = 6
    Top = 97
    Width = 135
    Height = 21
    DataField = 'Continent'
    DataSource = DataSource1
    MaxLength = 0
    TabOrder = 2
end
object EditArea: TDBEdit
    Left = 6
    Top = 135
    Width = 65
```

```
    Height = 21
    DataField = 'Area'
    DataSource = DataSource1
    MaxLength = 0
    TabOrder = 3
end
object EditPopulation: TDBEdit
    Left = 6
    Top = 173
    Width = 65
    Height = 21
    DataField = 'Population'
    DataSource = DataSource1
    MaxLength = 0
    TabOrder = 4
end
end
end
object DataSource1: TDataSource
    Left = 95
    Top = 177
end
object Database1: TDatabase
    DatabaseName = 'DB1'
    LoginPrompt = False
    SessionName = 'Default'
    Left = 128
    Top = 176
end
end
>>
```

Reference:

7/16/98 4:34:09 PM

Last Modified: 01-SEP-99

## Article #16341: Fix for BP7 GRAFFITI example.

Technical Notes Database

TN1341D.txt     Fix for BP7 GRAFFITI example.

Category       :OWL

Platform        :All

Product         :Pascal   All

### Description:

The problem is that BWCC seems to stick around in memory.

Add this line to the very end of the GRAFFITI.PAS file (right before "end.")

to decrement the usage count of BWCC:

```
FreeLibrary(GetModuleHandle('BWCC'));
```

### Reference:

7/16/98 4:35:45 PM

Last Modified: 01-SEP-99

## Article #16345: Clean-Boot 32 bit Delphi Installation

Technical Information Database

TI1345D.txt - Clean-Boot 32 bit Delphi Installation

Category :Install Issues  
Platform :All-32Bit  
Product :All32Bit,

Description:

-----  
HOW TO INSTALL DELPHI IN A CLEAN-BOOT ENVIRONMENT IN WINDOWS 9x  
-----

CONTENTS

-----  
Section 1: INTRODUCTION  
Section 2: WHAT IS SAFE MODE?  
Section 3: HOW TO BOOT-UP WINDOWS 95 IN SAFE MODE

-----  
Section 1: INTRODUCTION  
-----

This document demonstrates how to do a clean-boot installation for the desktop and Client/Server versions of Delphi 1.0, and the all other versions of Delphi in Windows 95.

-----  
Section 2: WHAT IS SAFE MODE?  
-----

An important feature that comes with Windows 95 is the "Safe Mode" option. "Safe Mode" is one of the several different ways of loading Windows 95. In "Safe Mode", Windows 95 does not load and use any of your device drivers installed in your system, including your video card, sound card, modem, monitor, mouse, joysticks, etc. Instead, Windows 95 loads up and use the default standard VGA mode in 16 colors and a standard mouse and keyboard driver. One of the advantage of using "Safe Mode" is to free up more conventional memorys and remove any device driver that can cause conflicts during the installation process.

-----  
SECTION 3: HOW TO BOOT-UP WINDOWS 95 IN SAFE MODE  
-----

1. When you boot-up your system the first time, press F8 when you see STARTING WINDOWS 95 on the screen. Usually the message display on the top-left hand corner.

2. If you press F8 before Windows 95 starts loading, you will see the Boot-Up Main Option as follow:

1. Normal
2. Logged
3. Safe Mode
4. Safe Mode With Network Support
5. Step-by-step Confirmation
6. Command Prompt Only
7. Safe Mode Command Prompt Only
8. Previous Version of MS-DOS

NOTE: Some of the options listed above might not be available on your system, for example, "Safe Mode With Network Support" if your system don't have network feature installed. It depends on your system configuration.

3. Select "Safe Mode" from the Main Option menu and Windows 95 will load the operating system.

4. To install Delphi, please refer to Delphi install.txt and readme.txt.

Reference:

4/1/99 11:45:24 AM

Last Modified: 01-SEP-99

## Article #16351: Creating a Wallpaper Using Delphi

Technical Information Database

TI1351D.txt    Creating a Wallpaper Using Delphi

Category     :General Programming

Platform     :All

Product      :Delphi    All

### Description:

To the wallpaper in Windows 95 you must use the Win32 API function SystemParametersInfo. SystemParametersInfo retrieves and sets system wide parameters including the wallpaper. The code below illustrates setting the wallpaper to the Athena bitmap.

---- Code Follows -----

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'c:\windows\athena.bmp';
  SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, PChar(s), 0)
end;
```

### Reference:

7/16/98 4:34:09 PM

Last Modified: 01-SEP-99

## Article #16357: Detecting Windows Shutdown

Technical Information Database

TI1357D.txt - Detecting Windows Shutdown

Category :VCL  
Platform :All  
Product :All32Bit,

### Description:

It is possible for you to detect when windows is being shutdown and to prevent this shutdown from within a running Delphi application.

The simple solution is to add an event handler to the main form's OnCloseQuery event. This event handler occurs as a result of a WM\_QUERYENDSESSION message which is sent to all running application in Windows when Windows is about to be shut down. The boolean CanClose parameter to this event can be set to True to allow Windows to shut down or, CanClose can be set to False to prevent Windows from shutting down.

The code below illustrates using this event.

----- Code Follows -----

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
    // Ask user if shutdown should occur.
    if MessageDlg('Are you sure?', mtConfirmation, mbYesNoCancel, 0) =
mrYes
    then CanClose := true    // Allow Windows to shut down.
    else CanClose := false; // Prevent Windows from shutting down.
end;
```

Reference:



Last Modified: 01-SEP-99

## Article #16363: Returning Default Cursor after Running Queries

Technical Information Database

TI1363D.txt Returning Default Cursor after Running Queries

Category :General Programming

Platform :All

Product :Delphi All

Description:

Q: Why does the Mouse cursor not change back to an arrow after I do a Query?

A: When performing an open on a query, Delphi will change the cursor for you, even in the middle of an event, such as a button click. The example below will cause the cursor to show as a SQL Hourglass Icon, after you close the showmessage dialog box. The Mouse will behave as if its in the arrow state.

```
// Add to a button click event, the Query used does not  
// matter I used  
// Select * from Customer (on IBLocal)
```

```
with query1 do begin  
  close;  
  open;  
  showmessage(IntToStr(RecordCount));  
end; // with
```

What appears to be happening is that when Delphi tries to change the Cursor back to the Arrow, a new form has already been shown (the showmessage dialog) and so its job is done as the cursor will automatically be set to the arrow on the showmessage dialog form.

To solve this problem, add a `Application.ProcessMessages` before showing anew form, this will clear all pending message commands from the message queue, and the Mouse Cursor will appear normal again.

```
// Add to a button click event, the Query used does
```

```
// not matter I used
// Select * from Customer (on IBLocal)

with query1 do begin
  close;
  open;
  application.ProcessMessages;          // Add This Line.
  showmessage(IntToStr(RecordCount));
end; // with
```

Reference:

7/16/98 4:34:09 PM

Last Modified: 01-SEP-99

## Article #16369: Dynamic creation and circularly linking forms

Technical Information Database

TI1369D.txt Dynamic creation and circularly linking forms

Category :General Programming

Platform :All

Product :Delphi All

### Description:

Question: How do I make a simple method to rotate between forms?  
How do I add my own return results to a ShowModal form?  
How do I instantiate forms at runtime?

Answer: The method required is quite simple to implement. For my example I used 3 forms, the MainForm, Form1, and Form2. I placed a button on the MainForm that will bring up Form1, then from that form you could rotate through any number of forms via buttons placed on those forms. For my example, only Form1 and Form2 can be flipped between.

step 1. Places these two lines in the interface section of this Form, which will be referred to as the main form

```
const
    mrNext = 100;
    mrPrevious = 101;
```

step 2. On the main form add a button and add the following block of code into it.

```
var
    MyForm: TForm;
    R, CurForm: Integer;
begin
    R := 0;
    CurForm := 1;
    while R <> mrCancel do begin
        Case CurForm of
            1: MyForm := TForm1.Create(Application);
            2: MyForm := TForm2.Create(Application);
        end;
```

```

try
  R := MyForm.ShowModal;
finally
  MyForm.Free;
end;
case R of
  MrNext : Inc(CurForm);
  MrPrevious : Dec(CurForm);
end;
  // these 2 lines will make sure we don't go out of
bounds
  if CurForm < 1 then CurForm := 2
  else if CurForm > 2 then CurForm := 1;
end; // while
end;

```

step 3. Add forms 1 and 2 (and any others you are going to have) to the uses statement for the MainForm.

step 4. On Form1 and Form2 add the MainForm to the uses (so they can see the constants.

step 5. On Form1, Form2 and all subsequent forms add 2 TBitBtn's, labeled Next and Previous. In the OnClick Events for these buttons

add the following line of code.

If it's a Next Button add : ModalResult := mrNext;

If it's a Previous Button add : ModalResult := mrPrevious;

Reference:

7/16/98 4:34:09 PM

Last Modified: 01-SEP-99

## Article #16375: Avoid using Resource Heap with Tabbed Notebooks

Technical Information Database

TI1375D.txt Avoid using Resource Heap with Tabbed Notebooks

Category :General Programming

Platform :All

Product :Delphi All

Description:

Intended Audience:

-----  
Anyone who wants to necessarily avoid using User Resource heap in conjunction with TTabbedNotebooks.

Prerequisites:

- 
- Familiarity with the TTabbedNotebook and TTimer components.
  - Understanding of Microsoft Windows Architecture; specifically User resources.

Purpose of the TI:

-----  
This document demonstrates how to use Object Pascal to help control the number of active handlesWindows, the User Resource heap specifically, keeps track of. Why does this matter? Simplifying things, Windows keeps track of every Focusable Control via a Handle. The issue is,Windows cannot simply maintain an inexhaustible amount of Windows Handles (4 byte pointers) and this is where this TI's sample code will help "ease" the resource load which might restrict the Delphi Developer.

The User DLL actually is the Library which allocates and maintains resources for all Windows and related data structures, including focusable object's and other unmentioned, but nonetheless important USER functions, under Windows. It is this USER DLL resource limitation\* which we are temporarily working around. An example of an added resource load is, every control we add to a form, takes up 4 bytes of the USER 64k.

With this in mind, what exactly are we doing? We will be

destroying\*\* the Handles which Windows is architecturely designed to remember. By destroying these Handles, thus freeing USER resource drain, it does not mean we need to recreate said objects again. On the contrary, currently built into the VCL is the ability to keep track of said objects, which are in fact pointers to structures. So, knowing the VCL will maintain a Handle and windows will recreate a new Handle as NEEDED, instead of maintaining one permanently, as designed, we can take control of the USER resources manually and "ease" the overall USER resource load.

This TI will demonstrate the freeing of USER Handle resources via Delphi's TTabbedNoteBook (specifically destroying Page Handles), Delphi's DestroyHandle (TWinControl procedure for removing USER handles), and the Windows API call LockWindowUpdate (Locking unwanted repaints).

The technique of freeing a TTabbedNoteBook Page Handle can be extended to anyTWinControl descendant. TWinControl is the ancestor class which introduces the creating/destroying of Windows Handles; CreateHandle & DestroyHandle.

\* 64K for Win3.1 & 64K for Win95 16-bit subsystem alone. For further information, contact Microsoft or look in the MSDN.

\*\* As a side effect of destroying said Handles, the TTabbedNotebook used in this TI will experience faster page movement.

#### SAMPLE CODE

The following attached events are direct excerpts from a Project with a TTimer, TTabbedNotebook (with multiple pages) and an assorted plethora of controls on each notebook page. (The later is to emphasize the benefits of adding the below code) The attached Event snippet's should reside in the OnTimer event of the TTimer control and the OnChange event of the TTabbedNotebook, respectively. With no further adou, let's try our new code:

```
-----  
...  
Implementation
```

```
Type TSurfaceWin = class(TWinControl);
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
{This code will update the Form caption with the percentages of
 free SYSTEM, GDI, &USER for Windows.}
caption := 'SYSTEM: ' +
           inttostr(getfreesystemresources(GFSR_SYSTEMRESOURCES)) +
           ' GDI: ' + inttostr(getfreesystemresources(GFSR_GDIRESOURCES)) +
           ' USER: ' + inttostr(getfreesystemresources(GFSR_USERRESOURCES));
end;
```

```
procedure TForm1.TabbedNotebook1Change(Sender: TObject; NewTab:
           Integer; var AllowChange: Boolean);
```

```
begin
```

```
{LockWindowUpdate prevents any drawing in a given window}
```

```
LockWindowUpdate(handle);
```

```
{The reason for TSurfaceWin is because the DestroyHandle call is
declared abstract in TWinControl, which means, only descendant
classes can surface this Procedure. The rest of the line is meant
to flag the current page of the TabbedNotebook and destroy its
handle as we move to another page. NOTE: Even though we destroy
the handle, Windows itself remembers the page object and will
reassign/create a new one when the tab is once more clicked to. }
```

```
TSurfaceWin(TabbedNotebook1.pages.objects[tabbedNotebook1.
           pageindex]).DestroyHandle;
```

```
{Release the Lock on the Form so any Form drawing can work}
```

```
LockWindowUpdate(0);
```

```
end;
```

Reference:

7/16/98 4:34:10 PM

Last Modified: 01-SEP-99



## Article #16377: Installing Acrobat Reader for Windows

Technical Notes Database

TN1377D.txt Installing Acrobat Reader for Windows

Category :General Programming

Platform :All

Product :Delphi All

Description:

ADOBE FAXYI

COMPREHENSIVE OVERVIEW:

INSTALLING ADOBE ACROBAT READER FOR WINDOWS

WHAT'S COVERED

Overview of the Installation Process

Files Installed by the Acrobat Reader

Description of Installed Files

Changes Made to the WIN.INI and SYSTEM.INI Files

Installing Acrobat Reader 2.0 when an Earlier Version of Acrobat Reader is Installed

Installing Acrobat Reader when a Version of ATM Earlier than 3.01 has been Installed

Default .INI File Listings Description of .INI File Functions and Settings

Installing Acrobat Reader Manually

This document is designed to help network administrators and support staff who want to predict how installing the Acrobat Reader might affect their networks and users. By understanding how the Acrobat Reader Installer works, it is possible to decide the best way to install the Acrobat Reader.

For information on obtaining Acrobat Reader 2.0, see Related Records.

OVERVIEW OF THE INSTALLATION PROCESS

When the Acrobat Reader Installer is launched in Windows, it partially decompresses itself in

the directory listed in the "Set Temp=" line in the AUTOEXEC.BAT file. Once the installer application is decompressed in this directory, it displays the license agreement. After select-ing "Accept" in the "Electronic End User License Agreement" dialog box, "Install" in the "Acrobat Installer" dialog box, and "OK" after entering your name and company, Acrobat Reader files are installed. If the "Set Temp=" line in the AUTOEXEC.BAT file points to an invalid drive or directory or there is insufficient free disk space on the drive specified, errors will occur during the Acrobat Reader installation.

The Acrobat Reader Installer installs the Acrobat Application, supporting libraries, on-line documentation, configuration (.INI) files, fonts, and Adobe Type Manager 3.01 (ATM) into pre-designated locations. The version of ATM included with Acrobat Reader isn't a complete version of ATM; it will only run under Windows 386-enhanced mode, does not include a control panel for adding or removing fonts, and, by default, is not available to other applications.

#### FILES INSTALLED BY THE ACROBAT READER

Acrobat reader installs the following files and directories (in quotes):

#### "ACROREAD"

```
ACROGRAF.DLL 117808 10-14-94 2:12a
ACROREAD.REG 686 [date/time of install]
ACROCORE.DLL 364640 10-14-94 2:12a
ACROREAD.EXE 299040 10-14-94 2:12a
VIEWLIB.DLL 405904 10-14-94 2:12a
"HELP"
  HELP_R.PDF 61575 10-14-94 2:12a
  README_R.TXT 11033 10-14-94 2:12a
```

NOTE: ACROREAD is the default installation name of the Acrobat Reader directory and can be renamed during the installation process. If installing under a networked version of Windows, the following files are installed into each user's personal WINDOWS

directory. Otherwise, these files are installed into the shared WINDOWS directory.

#### "WINDOWS"

```
ATM .INI 917 [date/time of install]
ACROREAD.INI 136 [date/time of install]
~ACROBAT.TMP 76 [date/time of install]
ADOBEACR.GRP 1632 [date/time of install]
```

An ATMFONTS.QLC file is created in the directory indicated by the "QLCDir=" line in the ATM.INI file. By default, the ATMFONTS.QLC file is installed in the same directory in which the ATM.INI file is located. The size of the file varies depending on how many fonts are installed.

If installing onto a system with an ATM.INI file that includes a "PFB\_Dir=" pointing to a valid directory, the following files are installed to that directory. If there is no ATM.INI file present on the system, or the directory listed in the PFB\_Dir= line is invalid, the files are installed in the root of the WINDOWS directory. If the SY\_\_\_\_.PFB or ZD\_\_\_\_.PFB files are present, they are not updated.

```
SY____ PFB 31586 [date/time of install]
ZD____ PFB 40858 [date/time of install]
ZX____ PFB 74870 [date/time of install]
ZY____ PFB 95283 [date/time of install]
```

If installing onto a system with an ATM.INI file that includes a "PFM\_Dir=" line pointing to a valid directory, the following files are installed to that directory. If there is no ATM.INI file present on the system, or the directory listed in the "PFM\_Dir=" line is invalid, the files are installed in the root of the WINDOWS directory. If the SY\_\_\_\_.PFM or ZD\_\_\_\_.PFM files are present, they are not updated.

```
SY____ PFM 672 [date/time of install]
ZD____ PFM 684 [date/time of install]
ZX____ PFM 683 [date/time of install]
ZX____ MMM 7188 [date/time of install]
ZY____ PFM 684 [date/time of install]
```

If Installing Acrobat Reader under a non-networked version of Windows, the following files are installed into the WINDOWS\SYSTEM directory. If installing Acrobat Reader under a networked version of Windows, the following files are installed in the root of the shared WINDOWS directory.

ATM32.DLL 372384 10-6-94 3:01a  
ATMSYS.DRV 26336 10-6-94 3:01a

When the file ATM32.DLL is first installed, it is installed with the name "ATM32.NDL." The next time windows is restarted the ATM driver (ATMSYS.DRV) will check for .NDL files. If any are found, the current .DLL (if any) will be deleted and the .NDL file will be renamed to take its place. If the DLL cannot be deleted, an error will occur.

If you don't have write access to the shared WINDOWS directory when installing Acrobat Reader, the files will not be installed and the error message "You are attempting a network node install, but certain components which must be installed by the network administrator are missing or are out of date. Please see your administrator" will be displayed followed by the message "The Installation is complete." After the unsuccessful installation Acrobat or Windows will fail to launch. If Acrobat Reader was installed without write access to the shared WINDOWS directory, restore the ATM.INI and SYSTEM.INI files to their original state prior to installing Acrobat Reader.

#### DESCRIPTION OF INSTALLED FILES

|              |                                                                                                                               |
|--------------|-------------------------------------------------------------------------------------------------------------------------------|
| ACROGRAF.DLL | Acrobat viewer support file.                                                                                                  |
| ACROCORE.DLL | Acrobat viewer support file.                                                                                                  |
| ACROREAD.EXE | Acrobat Reader program file.                                                                                                  |
| VIEWLIB.DLL  | Acrobat viewer support file.                                                                                                  |
| ACROREAD.REG | OLE registration information. Acrobat Reader can accept only DDE open/close messages. Reader is not an OLE-aware application. |

HELP\_R.PDF On-line documentation.

README\_R.TXT Installation directions, upgrade information, and license agreement.

ATM.INI Configuration information for the version of ATM installed with Reader.

ACROREAD.INI Configuration information for Reader. ~ACROBAT.TMP A temporary file used by the installer. It is deleted the first time Acrobat Reader is launched.

ADOBEACR.GRP Windows group where acrobat items are displayed.

ATMFonts.QLC ATM QuickLoad file. This file contains a list of installed fonts and font metrics of recently used fonts, which reduces Windows startup time. It is updated every time you exit Windows.

SY\_\_\_\_\_ PFB Symbol outline font.

ZD\_\_\_\_\_ PFB Zaph Dingbats outline font.

ZX\_\_\_\_\_ PFB Sans serif multiple master substitution outline font.

ZY\_\_\_\_\_ PFB Serif multiple master substitution outline font.

SY\_\_\_\_\_ PFM Symbol character metrics data.

ZD\_\_\_\_\_ PFM Zaph Dingbats character metrics data.

ZX\_\_\_\_\_ PFM Sans Serif multiple master substitution font metrics data.

ZX\_\_\_\_\_ MMM Sans serif multiple master metrics file.

ZY\_\_\_\_\_ PFM Serif multiple master substitution font metrics data.

ZY\_\_\_\_\_ MMM Serif multiple master metrics file.

ATM32.DLL Adobe Type Manager 3.01 program file for 386 Enhanced mode Windows.

ATMSYS.DRV Adobe Type Manager 3.01 system driver.

CHANGES MADE TO THE WIN.INI AND SYSTEM.INI FILES During the Acrobat Reader installation, the following line is added to the [extensions] section of the WIN.INI file:

```
PDF=C:\ACROREAD\ACROREAD.EXE ^.PDF
```

This allows the user to double-click on a file with a .PDF extension to launch Acrobat Reader and open the file.

The following line is added to the [Adobe Display] section:

```
UseGDI=1
```

NOTE: If there is no [Adobe Display] section heading, the installer adds it as well as the UseGDI=1" line. The number "1" in the "UseGDI=1" setting tells Acrobat Reader to use Windows features to accelerate screen redraw. However, with some video displays, this may cause problems. The setting can be changed to "0" to disable the "UseGDI=" setting if video compatibility problems arise.

The following lines are added to the users [boot] section of their SYSTEM.INI file if they do not already exist.

```
system.driv=atmsys.driv
atm.system.driv=system.driv
```

These lines load ATM when Windows is launched.

#### INSTALLING ACROBAT READER 2.0 WHEN AN EARLIER VERSION OF ACROBAT READER IS INSTALLED

The Acrobat Reader Installer does not remove obsolete Reader 1.0 components (e.g., help files, ReadMe files, etc.). When Acrobat Reader 2.0 is installed in a different directory from the one in which Acrobat Reader 1.0 is installed, neither application's performance is affected.

#### INSTALLING ACROBAT READER WHEN A VERSION OF ATM EARLIER THAN 3.01 HAS BEEN INSTALLED

When Acrobat Reader 2.0 is installed into a system with a full version of ATM earlier than version 3.01, the alert "The current version of Adobe Type Manager (ATM) on your system is not compatible with Adobe Acrobat 2.0. You may partially upgrade your version of ATM at this time, or you may call Adobe at 1-800-83-FONTS for a full upgrade." appears. Selecting "OK" to partially upgrade ATM allows ATM and Acrobat to run only in Windows 386 Enhanced Mode. If "Cancel" is selected, Acrobat will not run.

CHANGES MADE AFTER SELECTING "OK" TO PARTIALLY UPGRADE ATM - If the version of ATM being upgraded is 2.6, references to the substitution multiple master fonts

AdobeSansXMM and AdobeSerifMM are replaced by references to the newer AdobeSanMM and AdobeSerMM multiple master fonts.

- ATM sections new to ATM 3.01 are added to the current ATM.INI.
- Sections of .INI files concerning printer drivers are not updated.
- The ATM control panel application is not removed or updated. The older control panel should continue to function with ATM 3.01, but using a mismatched driver and control panel is not supported by Adobe. If problems are encountered, upgrade to full ATM by calling 1-800-83-FONTS.
- A new ATM32.DLL and ATMSYS.DRV are added to the WINDOWS\SYSTEM directory.

#### DEFAULT .INI FILE LISTINGS

Acrobat Reader has its own .INI file, ACROREAD.INI, which is first written to contain only the following information:

[Oem]

OEM1=For product information and online help,  
OEM2=please see the file HELP\_R.PDF in  
OEM3=the Acrobat(tm) Reader Help folder.

When Acrobat Reader is first launched, the following lines are added to the ACROREAD.INI file:

```
[AdobeViewer]
User=John Smith
MaxDoc=1
MaxApp=1
ShortMenu=0
ShowToolbar=1
MaxPageCacheZoom=131072
MinPageCacheTicks=12
MaxPageCacheBytes=4194304
FullScrolling=0
DefaultSplitterPos=140
ThreadIndicator=180
DrawMissingThumbs=0
ThumbViewScale=8192
ThumbViewTimeout=120
DefaultFitType=FitR
```

```
DefaultInheritZoom=0
MaxCosDocCache=200000
DefaultOverviewType=1
CaseSensitive=0
WholeWords=0
UseDIC=0
CustomColor0=16777215
CustomColor1=16777215
CustomColor2=16777215
CustomColor3=16777215
CustomColor4=16777215
CustomColor5=16777215
CustomColor6=16777215
CustomColor7=16777215
CustomColor8=16777215
CustomColor9=16777215
CustomColor10=16777215
CustomColor11=16777215
CustomColor12=16777215
CustomColor13=16777215
CustomColor14=16777215
CustomColor15=16777215
```

The first time "OK" is selected in the "General Preferences" dialog box under the Edit menu, the following lines are added to the ACROREAD.INI file:

```
DefaultMag=100
DefaultMagType=0
GreekThreshold=6
GreekText=1
Chameleons=0
ShowLargeDIBS=1
PageUnits=1
SkipDeleteWarnings=0
DisplayAboutDialog=1
UsePageCache=1
MaxThreadZoom=200
SkipOpenDialog=0
```

The first time "OK" is selected in the "Full Screen Preferences" dialog box under the Edit menu, the following lines are added to the ACROREAD.INI file:

```
FullScreenColor=0
```



```
DefaultSlideTimer=5
UseSlideTimer=0
ForceTBHidden=0
WrapSlideShowPages=1
```

## Description of .INI File Functions and Settings

Unless stated otherwise, lines with invalid values are ignored, and the default value, as shown in the ACROREAD.INI file above, is used.

NOTE: Some of the following .INI functions are described as "Reserved" to indicate the entries relate to a feature or plug-in that has not yet been implemented. "Reserved" .INI entries should not be altered in any way.

|                   |                                                                                                                                                                                                        |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [Oem]             | These lines appear on the splash screen when Reader starts.                                                                                                                                            |
| User              | The name typed by the user into the installer dialog box during installation. Appears on the splash screen when Reader starts.                                                                         |
| MaxDoc            | Controls default size of displayed document windows. If "MaxDoc=1," documents are Maximized. If "MaxDoc=0," documents are cascaded.                                                                    |
| MaxApp            | Sets the default size for the application window. If "MaxApp=1," the application window size is Maximized (full screen). If "MaxApp=0," application window size and position is Determined by Windows. |
| ShortMenu         | Ignored by Reader.                                                                                                                                                                                     |
| ShowToolBar       | If "ShowToolBar=1," Reader displays the toolbar, if "ShowToolBar=0," the toolbar is initially hidden. The setting can be changed by selecting "Show Toolbar" or "Hide Toolbar" from the Window menu.   |
| MaxPageCacheZoom  | Sets the maximum page magnification at which caching will occur.                                                                                                                                       |
| MinPageCacheTicks | Sets the time threshold for page caching in 1/12ths of seconds. If a page draws in less than this amount of time, the                                                                                  |

page will not be cached.

MaxPageCacheBytes            Sets the size of the page cache in bytes.

FullScrolling                If "FullScrolling=1," the scrolling region on each page is made larger, so that when using the elevator thumb or page down key, it is possible to scroll past the bottom of a page before it turns to the next page. This setting does not affect documents whose pages fit completely within the display. If "FullScrolling=0," the page is set for a normal scrolling region.

DefaultSplitterPos           The value, in pixels, which sets the horizontal position of the boarder which divides the document area and the overview area when book-marks or thumbnails are being displayed. Any value smaller than the minimum value of 100 is rounded up (clamped) to 100.

ThreadIndicator              Reserved

DrawMissingThumbs           Reserved

ThumbViewScale              Sets the display size of thumbnails. The default of 8192 is roughly 12% of actual size. 65535 is 100%. 2000, or about 3%, is the minimum size. Invalid values will be clamped.

ThumbViewTimeout            Reserved

DefaultFitType               Reserved

DefaultInheritZoom          Reserved

MaxCosDocCache              Reserved

DefaultOverviewType         Default pagemode if one is not specified in the PDF file.

| value | meaning                           |
|-------|-----------------------------------|
| 0     | viewer takes no action            |
| 1     | page is displayed                 |
| 2     | page and thumbnails are displayed |
| 3     | page and bookmarks are displayed  |
| 4     | full screen mode                  |

CaseSensitive                If "CaseSensitive=0," case-sensitive searching is disabled by default.

                              If "CaseSensitive=1," case-sensitive searching is enabled. This setting can be overridden in the "Find" dialog box, but for the current session

only (i.e., the ACROREAD.INI file is not updated).

WholeWords If "WholeWords=0," "Match Whole Word Only" searching is disabled by default. If "WholeWords=1," "Match Whole Word Only" is enabled. This setting can overridden in the "Find" dialog box, but for the current session only (i.e., the ACROREAD.INI file is not updated).

UseDIC Reserved

CustomColorx (where x is a number from 0-15) Reserved.

The remaining ACROREAD.INI value settings can be set in Acrobat Reader by selecting a new value in the "General Preferences" or "Full Screen Preferences" dialog boxes.

DefaultMag If the PDF being opened does not have a default magnification specified in it, the value listed here will be used when "DefaultMagType=0." Valid values are 12 to 800. Values outside this range will be clamped.

DefaultMagType Specifies the magnification type. "DefaultMagType=0" uses the DefaultMag value, "DefaultMagType=1" displays at "Fit Page" magnification, "DefaultMagType=2" displays at "Fit Width" magnification, "DefaultMagType=4" displays at "Fit Visible" magnification. If the "DefaultMagType=" number is invalid, Acrobat Reader assumes a value of 1.

GreekThreshold Minimum height of text, in pixels, before Reader will render character shapes. Text below this size will display as a colored bar. Ignored if GreekText is set to 0.

GreekText If "GreekText=1," Acrobat Reader will greek text below GreekThreshold pixels. If "GreekText=0," text is rendered no matter what its size.

Chameleons Font(s) to use when a font required by a PDF is not available. If

"Cha-meleons= 0," Both Serif and Sans Serif are used, if "Chameleons=1," only Sans Serif is used and if "Chameleons=2," only Serif is used.

ShowLargeDIBS If "ShowLargeDIBS=1," Acrobat Reader will draw all graphics. If "ShowLargeDIBS=0," Acrobat Reader will only draw graphics that consume less than 4K of space within the PDF before decompression.

PageUnits Default measurement system. Acrobat Reader uses points if "PageUnits=0," inches if "PageUnits=1," and millimeters if "PageUnits=2."

SkipDeleteWarnings Ignored by Reader.

DisplayAboutDialog The Acrobat Reader splash screen is displayed during startup if "DisplayAboutDialog=1," and is not displayed if "DisplayAboutDialog=0."

UsePageCache Page caching is enabled if "UsePageCache=1," and is disabled if "UsePageCache=0."

MaxThreadZoom Maximum magnification to use when viewing in fit visible magnification. Valid values are between 12 to 800. Invalid values will be clamped.

SkipOpenDialog If Acrobat is launched and no PDF is specified on the command line, Reader will display an open dialog if "SkipOpenDialog=0." If "SkipOpenDialog=1," a blank workspace is displayed.

FullScreenColor Color to use as a background when displaying in full screen mode. Uses standard Windows color numbering system.

DefaultSlideTimer Time, in seconds to wait between slides. Use "SlideTimer=" setting must be set to 1 to be used.

UseSlideTimer If "UseSlideTimer=1," enabling full screen mode activates an automatic slide show, switching slides at intervals specified by the "DefaultSlideTimer" setting.

ForceTBHidden Ignored.

WrapSlideShowPages When "UseSlideTimer=1 (on)," setting

"WrapSlideShowPages=" to a number other than 0 will cause the pages of the PDF to be looped endlessly. Set "WrapSlideShowPages=0" to disable looping and allow Acrobat Reader to exit from full screen mode when all pages have been displayed.

The Acrobat Reader Installer creates an ATM.INI containing the below entries if an ATM.INI does not already exist. If an ATM.INI file already exists, lines shown below, which do not exist in the current ATM.INI, are added.

```
[Settings]
Private=On
ATMIcon=off
TmpDir=C:\DOS
TmpCount=0
QLCDDir=C:\WINDOWS
ATM=off
FontCache=256
SynonymPSBegin=9
Version=3.01
[Aliases]
Helv=Helvetica
Tms Rmn=Times
Roman=Times
Modern=Helvetica
[Synonyms]
Helv=Helvetica
Tms Rmn=Times
[OldScreenFonts]
Helv=Yes
Tms Rmn=Yes
Courier=Yes
[ATMVersions]
pm5app=2.60
[DisablePSBD]
Acroexch=Yes
Acroread=Yes
XPress=Yes
Wpwin60=Yes
wpwprint=Yes
FLW2=Yes
```

[Mono]  
Courier=Yes  
LetterGothic=Yes  
PrestigeElite=Yes  
Orator=Yes

[Fonts]  
Symbol=C:\WINDOWS\PFM\SY\_\_\_\_\_.PFM,C:\WINDOWS\SY\_\_\_\_\_.PFB  
ZapfDingbats=C:\WINDOWS\PFM\ZD\_\_\_\_\_.PFM,C:\WINDOWS\ZD\_\_\_\_\_.

PFB

AdobeSanMM=C:\WINDOWS\PFM\ZX\_\_\_\_\_.PFM,C:\WINDOWS\ZX\_\_\_\_\_.

PFB

AdobeSerMM=C:\WINDOWS\PFM\ZY\_\_\_\_\_.PFM,C:\WINDOWS\ZY\_\_\_\_\_.

PFB

[MMFonts]  
AdobeSanMM=C:\WINDOWS\PFM\ZX\_\_\_\_\_.MMM,C:\WINDOWS\ZX\_\_\_\_\_.

PFB

AdobeSerMM=C:\WINDOWS\PFM\ZY\_\_\_\_\_.MMM,C:\WINDOWS\ZY\_\_\_\_\_.

NOTE: For more information on the ATM.INI file, see the file TECHREF.  
WRI included with most  
copies of ATM 3.0x. A few lines specific to the Acrobat installer are  
discussed here.

ATM                    If "ATM= on" then ATM is available to all  
applications. If "ATM=off," ATM is  
                      either completely or partially disabled, depending on  
how the "Private" line  
                      is set.

Private                When "ATM=" is set to off, "Private=on" makes ATM  
available to Acrobat, but no  
                      other applications. When "Private=off," ATM not be  
available to Acrobat Reader,  
                      preventing Reader from launching. This line is used  
only by Acrobat and will not  
                      be present in the ATM.INI unless Acrobat has been  
installed.

ATMIcon                If "ATMIcon=on," ATM displays its icon at startup, If  
"ATMIcon=off," it does not.

#### MANUALLY INSTALLING ACROBAT READER

It is recommended to install Acrobat Reader using the Acrobat Reader  
installer. However, in some  
cases a custom install is required.

Manually installing Acrobat Reader:

1. The ACROGRAF.DLL, ACROCORE.DLL, ACROREAD.EXE, VIEWLIB.DLL, and README\_R.TXT must all be installed into the same directory. Item 3 of the Adobe Acrobat Reader License agreement requires the agreement be distributed with the Acrobat Reader software. The license agreement can be found in the README\_R.TXT file. The HELP\_R.PDF is not required but is strongly recommended.

2. An ATM.INI must be present in the WINDOWS directory, and have the information necessary for Acrobat to locate all the fonts normally installed with Reader. Refer to the TECHREF.WRI file included with many copies of ATM to learn what lines of the ATM.INI file are required for a particular installation.

3. An ACROREAD.INI file is recommended, but not required. Acrobat will create one if it is not present.

4. The following printer fonts and font metric files must be installed and pointed to by the ATM.

INI file: SY\_\_\_\_\_ PFB, ZD\_\_\_\_\_ PFB, ZX\_\_\_\_\_ PFB, ZY\_\_\_\_\_ PFB,  
SY\_\_\_\_\_ PFM, ZD\_\_\_\_\_ PFM,  
ZX\_\_\_\_\_ PFM, ZX\_\_\_\_\_ MMM, ZY\_\_\_\_\_ PFM, and ZY\_\_\_\_\_ MMM.

5. ATM32.DLL and ATMSYS.DRV files for ATM 3.01 or later must be installed. Lines must be added to the SYSTEM.INI file so the driver loads with Windows.

NOTE: It is recommended to always try using the Adobe Installer instead of a manual installation to Acrobat Reader. Adobe Technical Support cannot help with manual installations of Acrobat Reader because there are too many variables in performing a correct custom installation.

Reference:

7/16/98 4:35:46 PM

Last Modified: 01-SEP-99



## Article #16381: The DocInput Object: Properties and Methods

Technical Information Database

TI1381D.txt - The DocInput Object: Properties and Methods

Category :Delphi 2.x  
Platform :All Windows  
Product :Delphi2.x,

### Description:

The DocInput Object is an object from the NetMasters Internet Solutions Pack deployed with Delphi 2.01. It describes the input information for a document being transferred to the control. All of the internet controls in the suite that can receive a doc use this object as a property. The DocInput Object has the following properties:

BytesTotal, BytesTransferred, DocLink, FileName, Headers, PushStreamMode, State and Suspended.

BytesTotal is the total byte count of the item to be transmitted. The default or initial value is zero. The data type is a Long. It is a runtime and read only property. This value is obtained from the header property content-length. This value is used by the control to determine the amount of information to be transferred. It is also accessible to you to manage the buffer that you will use to re-assemble the data after transfer.

The BytesTransferred property is a property given to you inside the OnDocInput event. It is a runtime read only property that is of type long. It is set to zero when a new transfer begins. It is updated at the beginning of the OnDocInput Event. This value will reflect the value of the last transfer when no other transfer is in progress. The BytesTransferred property can be used to show progress on a progress bar or to confirm that the actual amount transferred corresponds to that which is expected.

The DocLink property tells the receiving control that the

source of the document will not be sent via data streaming or via an input file. It references a DocOutput.DocLink property which becomes the source of data in the transfer. This property is a read/write property that is runtime only. The property is of type DocLink. It is a string type and the defaultvalue is ''. When the DocLink property is set to a value other than '', the FileName property is automatically set to ''. This property is used to specify a source that is an internet control that has a DocOutput.DocLink property set to correspond with it (i.e. they are used in pairs).

The FileName property is a read/write runtime only property that is of type string. Its default value is ''. It must be a valid filename. This property can be set by passing it as an argument to a DocInput object. If this property is set to a value other than '', then the DocLink property is set to ''.

The Headers property is a runtime readonly property. The "headers" is a collection of DocHeader items that define the doc being transferred. The contents of the headers property should be modified before calling the GetDoc method. Each DocHeader represents a MultiPurpose Internet Mail Extension(MIME). Mime is the mechanism for specifying and describing the format of Internet Message Bodies. (See rfc1341 for details). The headers used depend on the protocol used but two are common to all protocols:

1. content-type

content type indicates the MIME specification of the ensuing document. "text/plain" is an example of this.

2. content-length

content length indicates the size of the documents in bytes.

The state property is a runtime read only property of the enumerated type DocStateConstants. The default value is icDocNone. The state property is updated by the control itself each time the DocInput event is activated.

The suspended property is a runtime read only property that is of type boolean. It is set by calling the suspend method. If it is set to true transfer is suspended.

The PushStream property is a read/write, runtime only property that is of the type boolean. The default value is false. If the FileName or DocLink properties are set to values other than ''

then the PushStream property is not accessible.

The DocInput object has 4 methods:  
GetData, PushStream, SetData and Suspend.

The GetData method tells the DocInput object to retrieve the current block of data when the DocOutput event is activated. This method can only be called during the OnDocInput event and only when the State property is set to icDocData(3). When using the FileName or DocLink properties this method may be used to examine data during transfer.

The PushStream method should only be called when the PushStreamMode has been set to true and when data is available. PushStream sets the State property based on the next step of the document transfer and activates the DocInput event when appropriate. It then returns to wait for the next call to PushStream. SetData should be called before calling PushStream.

The SetData method specifies the next data buffer to be transferred when the DocInput event is activated. SetData is called during a DocInput event or before calling SendDoc. If it is used before calling SendDoc then it is an alternative to sending the InputData parameters to InputData. The type should be specified as a variant.

The Suspend method takes the form suspend(true) or suspend(false). If the method has been called true twice then it must be called false twice to resume transfer.

This is some example code on how to use the DocInput Object. The full program that this code is from can be seen in the demos sub directory of the Delphi 2.01 CD-Rom. The project name is SimpMail.dpr. This example is a great example of using the headers property of the object. The DocInput event also shows proper use and testing of the State property.

```
{Clear and repopulate MIME headers, using the component's  
DocInput property. A separate DocInput OLE object could also be  
used. See RFC1521/1522 for complete information on MIME types.}  
procedure TMainForm.CreateHeaders;  
begin  
  with SMTP1 do  
  begin  
    DocInput.Headers.Clear;
```

```

DocInput.Headers.Add('To', eTo.Text);
DocInput.Headers.Add('From', eHomeAddr.Text);
DocInput.Headers.Add('CC', eCC.Text);
DocInput.Headers.Add('Subject', eSubject.Text);
DocInput.Headers.Add('Message-Id', Format('%s_%s_%s',
    [Application.Title, DateTimeToStr(Now), eHomeAddr.Text]));
DocInput.Headers.Add('Content-Type',
    'TEXT/PLAIN charset=US-ASCII');
end;
end;

```

```

{Send a simple mail message}
procedure TMainForm.SendMessage;
begin
    CreateHeaders;
    with SMTP1 do
        SendDoc(NoParam, DocInput.Headers, reMessageText.Text, '',
            '');
end;

```

```

{Send a disk file. Leave SendDoc's InputData parameter blank and
specify a filename for InputFile to send the contents of a disk
file. You can use the DocInput event and GetData methods to do
custom encoding (Base64, UUEncode, etc.) }
procedure TMainForm.SendFile(FileName: string);
begin
    CreateHeaders;
    with SMTP1 do
        begin
            DocInput.FileName := FileName;
            SendDoc(NoParam, DocInput.Headers, NoParam,
                DocInput.FileName, '');
        end;
end;

```

```

{The DocInput event is called each time the DocInput state
changes during a mail transfer. DocInput holds all the
information about the current transfer, including the headers,
the number of bytes transferred, and the message data itself.
Although not shown in this example, you may call DocInput's
SetData method if DocInput.State = icDocData to encode the data
before each block is sent.}
procedure TMainForm.SMTP1DocInput(Sender: TObject;
    const DocInput: Variant);
begin

```

```

case DocInput.State of
  icDocBegin:
    SMTPStatus.SimpleText := 'Initiating document transfer';
  icDocHeaders:
    SMTPStatus.SimpleText := 'Sending headers';
  icDocData:
    if DocInput.BytesTotal > 0 then
      SMTPStatus.SimpleText :=
        Format('Sending data: %d of %d bytes (%d%%)',
          [Trunc(DocInput.BytesTransferred),
            Trunc(DocInput.BytesTotal),
              Trunc(DocInput.BytesTransferred/DocInput.BytesTotal*100)])
    else
      SMTPStatus.SimpleText := 'Sending...';
  icDocEnd:
    if SMTPError then
      SMTPStatus.SimpleText := 'Transfer aborted'
    else
      SMTPStatus.SimpleText := Format('Mail sent to %s
        (%d bytes data)',
        [eTo.Text, Trunc(DocInput.BytesTransferred)]);
end;
SMTPStatus.Update;
end;

```

Reference:

3/30/99 12:43:15 PM

Last Modified: 01-SEP-99

## Article #16392: Creating Class Properties

Technical Information Database

TI1392D.txt - Creating Class Properties

Category :Object Pascal  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This TI demonstrates how to add a class property to a new component like the Font property on most components.

The following example declares a class, TMyClassProp which contains several fields including an enumerated type field.

### Notes:

1. PropertyObject - must be of type TPersistent or a descendant class of TPersistent.
2. Must create an instance of this TMyClassProp in the Create of the component.

----- Unit Follows -----

```
unit SubClass;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs;
```

```
type
```

```
{ First Create a enumerated list of elements  
then create a set of these elements }
```

```
TEnumList = (Mom, Dad, Sibling, Sister, Brother);  
TEnum = set of TEnumList;
```

```
{ Here is the Class Object we want to use as a property in our  
Component.
```

It has 4 properties, a Boolean, Word, String, and the enumerated type. }

```
TMyClassProp = class(TPersistent)
  FBool: Boolean;
  FWord: Word;
  FString: String;
  FEnum: TEnum;

published
  property HaveCar: Boolean read FBool Write FBool;
  property Age: Word read FWord write FWord;
  property Name: String read FString write FString;
  property Relation: TEnum read FEnum write FEnum;
end; // TMyClassProp

{ Now create the component which will contain a property of type
  TMyClassProp.}
TMyComponent = class(TComponent)
  FEnum: TEnum;           { Enumerated type, just for fun }
  FSubClass: TMyClassProp; { The Class Property we want }
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
published
  property Relation: TEnum read FEnum write FEnum;
  property Relative: TMyClassProp read FSubClass write FSubClass;
  { Published declarations }
end;

procedure Register;

implementation

  { Override Create, to create an instance of the Class property.
    This is required. }

constructor TMyComponent.Create(AOwner: TComponent);
begin
  inherited;
  FSubClass := TMyClassProp.Create;
end;
```

```
{ Override Destroy to perform house cleaning. }
destructor TMyComponent.Destroy;
begin
  FSubClass.Free;
  inherited;
end;

procedure Register;
begin
  RegisterComponents('Samples', [TMyComponent]);
end;

end.
```

Reference:

4/1/99 4:35:00 PM

Last Modified: 01-SEP-99



## Article #16397: Optimizing Oracle Connections with Windows 95

Technical Information Database

TI1397D.txt - Optimizing Oracle Connections with Windows 95

Category :Database Programming  
Platform :All-32Bit  
Product :

Description:

Intended Audience:

-----

Anyone who wants to Improve Oracle IP Connection time under Windows 95.

Prerequisites:

-----

- Windows 95
- Oracle Client Software Installed & Setup to connect to a Oracle Server via TCP/IP.
- (Optional) Delphi 2.0 C/S Software to test result.

Purpose of the TI:

-----

To help speed up Connection time to Oracle under Windows 95. Under WinNT this is not an issue, therefore, this TI applies only to Windows 95. Performance times, as seen on some computer's, with & without this modification are:

Before : Win95 = 10-15 Seconds.  
          WinNT = 2-3     Seconds.

After : Win95 = 3-4     Seconds. (Big Improvement)

Problem: Windows 95 inherently searches for IPC addresses on some network nodes PRIOR to making a connection to a Oracle DNS whereas WinNT does not.

Solution: Modify the Oracle SQLNET.ORA file to disable said Windows 95 feature.

### Step-by-Step Solution:

-----  
1) Open SQLNET.ORA via Notepad or Write.  
(This file can be found in the \network\admin directory.  
Disregard any other occurrences of this file)

This files looks like this:

```
#####  
# Filename.....: sqlnet.ora  
# Node.....: local.world  
# Date.....: 24-MAY-94 13:23:20  
#####  
TRACE_LEVEL_CLIENT = OFF  
sqlnet.expire_time = 15  
names.default_domain = borland.world  
name.default_zone = borland.world
```

2) Add the following Parameter to the SQLNET.ORA file:

```
AUTOMATIC_IPC = OFF
```

Modified, the SQLNET.ORA file looks like this:

```
#####  
# Filename.....: sqlnet.ora  
# Node.....: local.world  
# Date.....: 24-MAY-94 13:23:20  
#####  
AUTOMATIC_IPC = OFF  
TRACE_LEVEL_CLIENT = OFF  
sqlnet.expire_time = 15  
names.default_domain = borland.world  
name.default_zone = borland.world
```

3) Save the file the new SQLNET.ORA file and voila! Anytime an Oracle Connection is initiated, performance with increase from 15 seconds to 3 seconds required. This improves Delphi's already blazing speed.

Reference:

4/22/99 1:05:44 PM

Last Modified: 01-SEP-99

## Article #16402: Connecting to a 32-bit Sybase server

Technical Information Database

TI1402D.txt - Connecting to a 32-bit Sybase server

Category :Database Programming  
Platform :All-32Bit  
Product :

### Description:

Connecting To Sybase 32-bit

This TI will instruct you on how to connect to a Sybase database via Borland's native 32-bit Sybase SQL Links packaged with Delphi 2.x. The Sybase client software should take up approximately 10+ megabytes of hard drive space.

### Steps To Connect:

1. Make sure that the SQL Links package is installed on your local drive. If you installed the full package of Delphi 2.x then it should be installed already.
2. Install the Sybase client software.
3. When prompted during installation to install either the 16 or 32 bit Sybase links, choose only 32-bit by checking the check box next to the 32-bit selection and making sure that the check box next to the 16-bit selection is blank.
4. After the client software is installed on your hard drive, you will be asked if you would like the install program to automatically update your AUTOEXEC.BAT file. You should choose YES.
5. When prompted to edit your SQL.INI file, choose yes.
6. In the "Input Server Name:" section, type in the alias of the server. Click on the 'Add' button to add this name to your "Server Entry:" list. Next, make sure that the "Service Type:" (It should be 'query'), the "Platform:" (it usually defaults to either NT, dos, or Win3), and the "Net-Library Driver:" (should default to either NLWNSCK or NLNWLINK) edit boxes are correct. Fill out the "Connection Information/Network Address:" by entering the network address of the server you will be connecting to in this edit box. click on the 'Add Service' button. You should now be able to Ping your server by clicking on the 'Ping' button. Exit and save the current settings.

7. Shutdown and Restart your machine.
8. Goto the Delphi program group and execute the Database Explorer.
9. In the Database explorer, make sure that you are on the Database tab. In the pull down menus, choose Object | New... A dialog box should appear with the name STANDARD next to a down arrow. Click the down arrow and select SYBASE from the list that is displayed.
10. There should now be an Alias for your Sybase connection called SYBASE1. Make sure that this name is highlighted. Click on the definitions tab in the Database Explorer. Under the "Server Name" section, type the name of one of the servers that you entered in your SQL.INI that you were able to Ping. In the "User Name" section type the name of a user with rights to the server specified in your "Server Name" section. Make sure that you know the password of the user name that you just specified.
11. Double click on the alias name (SYBASE1) and you should be prompted for a User Name and Password. The User Name should default to the name you specified in the "User Name" section of the Sybase Alias. Type in the password that corresponds to this User. Click on the OK button. You should now see a little green box around the icon next to the Sybase Alias (SYBASE1). This means that you are connected.

Testing your Connection within Delphi 2.x:

1. Drop a TDataSource, a TTable, and a TDBGrid on a blank form.
2. In the Object Inspector for the TDataSource, Set the DataSet to 'Table1'(no quotes).
3. In the Object Inspector for the TTable, set the DataBase Name to SYBASE1. Scroll down to the TableName property and double click on the edit box next to it. You should be prompted for a user name and a password. The user name that you entered in the Database Explorer for the Sybase Alias should already be displayed. Enter the corresponding password. Click the OK button.
4. You should now see a list of table names. Choose one of these names.
5. Click on the TDBGrid. Set the DataSource Property to DataSource1.
6. Set the Active property of the TTable to TRUE.
7. You should now see the data in the TDBGrid. When you run this application, you will be prompted for the username/password. Type in your password and click OK. You should now see the data in the grid.

Error Messages:

Unable to determine net-library error: This error means that the .DLL's needed are nowhere to be found. These files should be in your

\Sybase\DLL directory:

Libblk.dll  
Libcomn.dll  
Libcs.dll  
Libct.dll  
Libintl.dll  
Libsrv.dll  
Libsybdb.dll  
Libtcl.dll  
Mscvrt10.dll  
Nldecnet.dll  
Nlmsnmp.dll  
Nlnwadv.t.exe  
Nlnwlink.dll  
Nlwnsck.dll

Disclaimer: This document does not promise a connection, it only shows the best and fastest way to connect.

Reference:

4/22/99 12:54:21 PM

Last Modified: 01-SEP-99

## Article #16407: Hints on Overcoming Installation Problems

Technical Information Database

TI1407D.txt - Hints on Overcoming Installation Problems

Category :Delphi 2.x  
Platform :All-32Bit  
Product :Delphi2.x,

Description:

Delphi 2.0 Up and Running!

-----

This technical document is designed to assist you in reinstalling Delphi 2.0. If you require additional assistance see the borland web site for support options.

Pre-Installation Recommendations

-----

- \* Before reinstalling Delphi 2.0, run the uninstall option from the add/remove program icon in the Windows 95 control panel, or select the uninstall icon from the Delphi 2.0 program group under Windows NT.
  
- \* Do not run the uninstall from the Delphi 2.0 CD.
  
- \* Before uninstalling and reinstalling Delphi 2.0, shut down the any applications using the Borland Database Engine, and close the Local InterBase Server (if running) by selecting the Local InterBase icon (right click from the system tray in Windows 95) and choose "shutdown".

- \* Installing Delphi 2.0 on a Windows NT system requires Windows NT version 3.51 or later.
  
- \* Before installing Delphi 2.0, it is suggested that you install the latest service pack available for your Windows system. Service packs are available from the Microsoft Corporation. One good source is Microsoft's World Wide Web site on the Internet located at [www.microsoft.com](http://www.microsoft.com).
  
- \* If you already have a Delphi 1.0 installation on your system, you should install Delphi 2.0 to a different directory structure. The only directory that is recommended to be common to both versions of Delphi is the IDAPI directory.
  
- \* Be sure you have proper administration rights on the machine you are installing to.
  
- \* Installing Delphi 2.0 to a network server is not supported.
  
- \* To install Delphi 2.0 to a computer that does not have a CD-ROM drive, use a network connection or set up a direct cable connection using a parallel or serial cable, and copy the files in the install directory on the Delphi 2.0 CD to a temporary directory on the computer you wish to install to, then run the setup program from the temporary directory created on the computer without the CD-Rom. After successful installation, you may delete the files you copied to the temporary directory.
  
- \* If you are running stacker, rename VSTACKER.386 (located in your windows\system directory) to VSTACKER.\*\*\*. Restart Windows, and run the install program again.
  
- \* If you have disabled virtual memory, you need to re-enable it for the installation program allowing for at least 64MB of virtual memory.



- \* Long file names should be enabled on your operating system.
- \* If you have installed Paradox, delete all trailing lock files before installing.

## Installation Errors

-----

- \* If the online registration wizard does not complete your online registration, run the install program again, selecting cancel when you are presented with the choice of registering online. Please complete and mail the registration card included with your Delphi 2.0 product.
- \* Should you receive a blank dialog box during install, select cancel and the install should continue successfully.
- \* During the installation, should you receive one of the following error messages:

"out of disk space",  
"no temp var",  
"error 101"  
"error 102"

Add more disk space and/or make sure you have a "temp" environment variable defined and a temp directory created. There must be ample free space on the drive the temp directory is located on.

- \* During the installation, should you receive the error:

"Install Shield error filename -51"

Try the following:

- 1) Copy all the files (EXCEPT CTL3D32.DLL) from the

runimage\delphi20\windows\system32 directory on the Delphi 2.0 CD to a temporary directory on your hard disk.

- 2) Clear the read only attribute on all the files you copied to the temporary directory.
- 3) Copy the files to your windows\system directory, or windows\system32 on Windows NT systems.
- 4) Run setup again.

#### Other tips and techniques for a successful installation

-----

- \* Try running the install from your hard disk. To do this, uninstall Delphi 2.0, then simply copy the files from the install directory on the Delphi 2.0 CD to a temporary directory on any hard drive, then run the install program from that temporary directory. After successful installation, you may delete the files you copied to the temporary directory.
- \* Temporarily rename the win.ini file (located in your Windows directory) to win.in\$, reboot, and reinstall or rerun Delphi 2.0. If this helps, suspect any programs listed in the run, or load sections of win.ini, or any non standard print drivers.
- \* Boot with the standard video driver shipped with your Windows system.
- \* Check for read-only attributes on files in the Windows and windows\system directories.

#### Post Installation Issues

-----

- \* Should you receive the error message: "odbc is corrupt or not

installed correctly" or "BDECFG32.EXE Error" when you try to install a 32 bit ODBC driver to the BDE, you first need to install a 32 bit ODBC manager, available from InterSolv and Microsoft. One good source is Microsoft's World Wide Web site on the Internet located at [www.microsoft.com](http://www.microsoft.com).

- \* If you install Delphi 1.0 after installing Delphi 2.0, and Delphi 2.0 loads Delphi 1.0's help files, delete any references to the Delphi 1.0 help file in the WINHELP.INI file located in your Windows directory.

Reference:

4/22/99 12:55:36 PM

Last Modified: 01-SEP-99

## Article #16412: A Better Way To Print a Form

Technical Information Database

TI1412D.txt - A Better Way To Print a Form

Category :Printing  
Platform :All-32Bit  
Product :All32Bit,

Description:

The following TI details a better way to print the contents of a form, by getting the device independent bits in 256 colors from the form, and using those bits to print the form to the printer.

In addition, a check is made to see if the screen or printer is a palette device, and if so, palette handling for the device is enabled. If the screen device is a palette device, an additional step is taken to fill the bitmap's palette from the system palette, overcoming some buggy video drivers who don't fill the palette in.

Note: Since this code does a screen shot of the form, the form must be the topmost window and the whole form must be viewable when the form shot is made.

```
unit Prntit;  
  
interface  
  
uses  
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,  
    Controls, Forms, Dialogs, StdCtrls, ExtCtrls;  
  
type  
    TForm1 = class(TForm)  
        Button1: TButton;  
        Image1: TImage;
```

```

    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

uses Printers;

procedure TForm1.Button1Click(Sender: TObject);
var
    dc: HDC;
    isDcPalDevice : BOOL;
    MemDc :hdc;
    MemBitmap : hBitmap;
    OldMemBitmap : hBitmap;
    hDibHeader : Thandle;
    pDibHeader : pointer;
    hBits : Thandle;
    pBits : pointer;
    ScaleX : Double;
    ScaleY : Double;
    ppal : PLOGPALETTE;
    pal : hPalette;
    Oldpal : hPalette;
    i : integer;
begin
    {Get the screen dc}
    dc := GetDc(0);
    {Create a compatible dc}
    MemDc := CreateCompatibleDc(dc);
    {create a bitmap}
    MemBitmap := CreateCompatibleBitmap(Dc,
  form1.width,
  form1.height);
    {select the bitmap into the dc}

```

```

OldMemBitmap := SelectObject(MemDc, MemBitmap);

{Lets prepare to try a fixup for broken video drivers}
isDcPalDevice := false;
if GetDeviceCaps(dc, RASTERCAPS) and
    RC_PALETTE = RC_PALETTE then begin
    GetMem(pPal, sizeof(TLOGPALETTE) +
        (255 * sizeof(TPALETTEENTRY)));
    FillChar(pPal^, sizeof(TLOGPALETTE) +
        (255 * sizeof(TPALETTEENTRY)), #0);
    pPal^.palVersion := $300;
    pPal^.palNumEntries :=
        GetSystemPaletteEntries(dc,
            0,
            256,
            pPal^.palPalEntry);
    if pPal^.PalNumEntries <> 0 then begin
        pal := CreatePalette(pPal^);
        oldPal := SelectPalette(MemDc, Pal, false);
        isDcPalDevice := true
    end else
        FreeMem(pPal, sizeof(TLOGPALETTE) +
            (255 * sizeof(TPALETTEENTRY)));
end;

{copy from the screen to the memdc/bitmap}
BitBlt(MemDc,
    0, 0,
    form1.width, form1.height,
    Dc,
    form1.left, form1.top,
    SrcCopy);

if isDcPalDevice = true then begin
    SelectPalette(MemDc, OldPal, false);
    DeleteObject(Pal);
end;

{unselect the bitmap}
SelectObject(MemDc, OldMemBitmap);
{delete the memory dc}
DeleteDc(MemDc);
{Allocate memory for a DIB structure}
hDibHeader := GlobalAlloc(GHND,
    sizeof(TBITMAPINFO) +

```

```

        (sizeof(TRGBQUAD) * 256));
{get a pointer to the allocated memory}
pDibHeader := GlobalLock(hDibHeader);

{fill in the dib structure with info on the way we want the DIB}
FillChar(pDibHeader^,
        sizeof(TBITMAPINFO) + (sizeof(TRGBQUAD) * 256),
        #0);
PBITMAPINFOHEADER(pDibHeader)^.biSize :=
    sizeof(TBITMAPINFOHEADER);
PBITMAPINFOHEADER(pDibHeader)^.biPlanes := 1;
PBITMAPINFOHEADER(pDibHeader)^.biBitCount := 8;
PBITMAPINFOHEADER(pDibHeader)^.biWidth := form1.width;
PBITMAPINFOHEADER(pDibHeader)^.biHeight := form1.height;
PBITMAPINFOHEADER(pDibHeader)^.biCompression := BI_RGB;

{find out how much memory for the bits}
GetDIBits(dc,
        MemBitmap,
        0,
        form1.height,
        nil,
        TBitmapInfo(pDibHeader^),
        DIB_RGB_COLORS);

{Alloc memory for the bits}
hBits := GlobalAlloc(GHND,
        PBitmapInfoHeader(pDibHeader)^.BiSizeImage);
{Get a pointer to the bits}
pBits := GlobalLock(hBits);

{Call fn again, but this time give us the bits!}
GetDIBits(dc,
        MemBitmap,
        0,
        form1.height,
        pBits,
        PBitmapInfo(pDibHeader)^,
        DIB_RGB_COLORS);

{Lets try a fixup for broken video drivers}
if isDcPalDevice = true then begin
    for i := 0 to (pPal^.PalNumEntries - 1) do begin
        PBitmapInfo(pDibHeader)^.bmiColors[i].rgbRed :=
            pPal^.palPalEntry[i].peRed;
    end
end

```

```

    PBitmapInfo(pDibHeader)^.bmiColors[i].rgbGreen :=
        pPal^.palPalEntry[i].peGreen;
    PBitmapInfo(pDibHeader)^.bmiColors[i].rgbBlue :=
        pPal^.palPalEntry[i].peBlue;
end;
FreeMem(pPal, sizeof(TLOGPALETTE) +
        (255 * sizeof(TPALETTEENTRY)));
end;

{Release the screen dc}
ReleaseDc(0, dc);
{Delete the bitmap}
DeleteObject(MemBitmap);

{Start print job}
Printer.BeginDoc;

{Scale print size}
if Printer.PageWidth < Printer.PageHeight then begin
    ScaleX := Printer.PageWidth;
    ScaleY := Form1.Height * (Printer.PageWidth / Form1.Width);
end else begin
    ScaleX := Form1.Width * (Printer.PageHeight / Form1.Height);
    ScaleY := Printer.PageHeight;
end;

{Just incase the printer driver is a palette device}
isDcPalDevice := false;
if GetDeviceCaps(Printer.Canvas.Handle, RASTERCAPS) and
    RC_PALETTE = RC_PALETTE then begin
{Create palette from dib}
    GetMem(pPal, sizeof(TLOGPALETTE) +
        (255 * sizeof(TPALETTEENTRY)));
    FillChar(pPal^, sizeof(TLOGPALETTE) +
        (255 * sizeof(TPALETTEENTRY)), #0);
    pPal^.palVersion := $300;
    pPal^.palNumEntries := 256;
    for i := 0 to (pPal^.PalNumEntries - 1) do begin
        pPal^.palPalEntry[i].peRed :=
            PBitmapInfo(pDibHeader)^.bmiColors[i].rgbRed;
        pPal^.palPalEntry[i].peGreen :=
            PBitmapInfo(pDibHeader)^.bmiColors[i].rgbGreen;
        pPal^.palPalEntry[i].peBlue :=
            PBitmapInfo(pDibHeader)^.bmiColors[i].rgbBlue;
    end;
end;

```



```

end;
pal := CreatePalette(pPal^);
FreeMem(pPal, sizeof(TLOGPALETTE) +
        (255 * sizeof(TPALETTEENTRY)));
oldPal := SelectPalette(Printer.Canvas.Handle, Pal, false);
isDcPalDevice := true
end;

{send the bits to the printer}
StretchDibits(Printer.Canvas.Handle,
              0, 0,
              Round(scaleX), Round(scaleY),
              0, 0,
              Form1.Width, Form1.Height,
              pBits,
              PBitmapInfo(pDibHeader)^,
              DIB_RGB_COLORS,
              SRCCOPY);

{Just incase you printer driver is a palette device}
if isDcPalDevice = true then begin
    SelectPalette(Printer.Canvas.Handle, oldPal, false);
    DeleteObject(Pal);
end;

{Clean up allocated memory}
GlobalUnlock(hBits);
GlobalFree(hBits);
GlobalUnlock(hDibHeader);
GlobalFree(hDibHeader);

{End the print job}
Printer.EndDoc;

end;

```

Reference:

4/22/99 12:51:56 PM

Last Modified: 01-SEP-99

## Article #16417: Creating a DataAware Control for Browsing Data

Technical Information Database

TI1417D.txt Creating a DataAware Control for Browsing Data

Category :Database Programming

Platform :All

Product :Delphi All

Description:

OVERVIEW

This document describes minimal steps necessary to create a data-aware browsing component that displays data for a single field. The example component is a panel with DataSource and DataField properties similar to the TDBText component. See the Component Writer's Guide "Making a Control Data-Aware" for further examples.

WHO SHOULD USE THIS DOCUMENT

To get the best use of this document, you should be familiar with data aware control functionality and fundamental component creation tasks, such as

- deriving components from existing components
- overriding constructor and destructors
- creating new properties
- getting and setting property values
- assigning event handlers

BASIC STEPS TO CREATE A DATA-BROWSING COMPONENT

- Create or derive a component that allows the display, but not the entry of data. For instance, you could use a

TMemo with ReadOnly set to true. In the example outlined in this document, we'll use a TCustomPanel. The TCustomPanel will allow display, but not data entry.

- Add a data-link object to your component. This object manages communication between the component and the database table.
- Add DataField and DataSource properties to the component.
- Add methods to get and set the DataField and DataSource.
- Add a DataChange method the component to handle the data-link object's OnDataChange event.
- Override the component constructor to create the datalink and hook up the DataChange method.
- Override the component destructor to cleanup the datalink.

#### CREATING THE TDBPANEL

- Create or derive a component that allows the display, but not the entry of data. We'll be using a TCustomPanel as a starting point for this example.

Choose the appropriate menu option to create a new component (this will vary between editions of Delphi), and specify TDBPanel as the class name, and TCustomPanel as the Ancestor type. You may specify any palette page.

- Add DB and DBTables to your Uses clause.
- Add a data-link object to the components private section. This example will display data for a single field, so we will use a TFieldDataLink to provide the connection between our new component and a DataSource. Name the new data-link object FDataLink.

```
{ example }
private
    FDataLink: TFieldDataLink;
```

-- Add DataField and DataSource properties to the component. We will add supporting code for the get and set methods in following steps.

Note: Our new component will have DataField and DataSource properties and FDataLink will also have its own DataField and Datasource properties.

```
{ example }
published
  property DataField: string
    read  GetDataField
    write SetDataField;
  property DataSource: TDataSource
    read  GetDataSource
    write SetDataSource;
```

-- Add private methods to get and set the DataField and DataSource property values to and from the DataField and DataSource for FDataLink.

```
{ example }
private
  FDataLink: TFieldDataLink;
  function GetDataField: String;
  function GetDataSource: TDataSource;
  procedure SetDataField(Const Value: string);
  procedure SetDataSource(Value: TDataSource);
  .
  .
implementation
  .
  .
function TDBPanel.GetDataField: String;
begin
  Result := FDataLink.FieldName;
end;

function TDBPanel.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBPanel.SetDataField(Const Value: string);
begin
```

```

    FDataLink.FieldName := Value;
end;

procedure TDBPanel.SetDataSource(Value: TDataSource);
begin
    FDataLink.DataSource := Value;
end;

-- Add a private DataChange method to be assigned to the
-- datalink's OnDataChange event.  In the DataChange method
-- add code to display actual database field data provided
-- by the data-link object.  In this example, we assign
-- FDataLink's field value to the panel's caption.

{ example }
private
    .
    .
    procedure DataChange(Sender: TObject);

implementation
    .
    .
    procedure TDBPanel.DataChange(Sender: TObject);
begin
    if FDataLink.Field = nil then
        Caption := '';
    else
        Caption := FDataLink.Field.AsString;
end;

-- Override the component constructor Create method.  In the
-- implementation of Create, create the FDataLink object, and
-- assign the private DataChange method to FDataLink's
-- OnDataChange event.

{ example }
public
    constructor Create(AOwner: TComponent); override;
    .
    .
implementation
    .
    .

```

```

constructor TMyDBPanel.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    FDataLink := TFieldDataLink.Create;
    FDataLink.OnDataChange := DataChange;
end;

-- Override the component destructor Destroy method. In the
  implementation of Destroy, set OnDataChange to nil (avoids a
  GPF), and free FDataLink.

{ example }
public
    .
    .
    destructor Destroy; override;
    .
    .
implementation
    .
    .
destructor TDBPanel.Destroy;
begin
    FDataLink.OnDataChange := nil;
    FDataLink.Free;
    inherited Destroy;
end;

-- Save the unit and install the component (see the Users
  Guide, and the Component Writers Guide for more on
  saving units and installing components).

-- To test the functionality of the component, add a TTable,
  TDataSource, TDBNavigator and TDBPanel to a form. Set the
  TTable DatabaseName and TableName to 'DBDemos' and 'BioLife',
  and the Active property to True. Set the TDataSource
  Dataset property to Table1. Set the TDBNavigator and
  TDBPanel DataSource property to DataSource1. The TDBpanel
  DataField name should be set as 'Common_Name'. Run the
  application and use the navigator to move between records to
  demonstrate the TDBPanel's ability to detect the change in
  data and display the appropriate field value.

```

```

unit Mydbp;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, DB, DBTables;

type
  TDBPanel = class(TCustomPanel)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: String;
    function GetDataSource: TDataSource;
    procedure SetDataField(Const Value: string);
    procedure SetDataSource(Value: TDataSource);
    procedure DataChange(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property DataField: string
      read   GetDataField
      write  SetDataField;
    property DataSource: TDataSource
      read   GetDataSource
      write  SetDataSource;
  end;

  procedure Register;

implementation

  procedure Register;
  begin
    RegisterComponents('Samples', [TDBPanel]);
  end;

  function TDBPanel.GetDataField: String;
  begin
    Result := FDataLink.FieldName;
  end;

  function TDBPanel.GetDataSource: TDataSource;

```



```

begin
  Result := FDataLink.DataSource;
end;

procedure TDBPanel.SetDataField(Const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBPanel.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

procedure TDBPanel.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then
    Caption := ''
  else
    Caption := FDataLink.Field.AsString;
end;

constructor TDBPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
end;

destructor TDBPanel.Destroy;
begin
  FDataLink.Free;
  FDataLink.OnDataChange := nil;
  inherited Destroy;
end;

end.

```

Reference:

7/16/98 4:34:11 PM

Last Modified: 01-SEP-99

## Article #16422: Using InputBox, InputQuery, and ShowMessage

Technical Information Database

TI1422D.txt Using InputBox, InputQuery, and ShowMessage  
Category :General Programming  
Platform :All  
Product :Delphi 1.0

### Description:

This function will demonstrate 3 very powerful and useful procedures built into Delphi.

The InputBox and InputQuery both allow user input.

Use the InputBox function when it doesn't matter if the user chooses either the OK button or the Cancel button (or presses Esc) to exit the dialog box. When your application needs to know if the user chooses OK or Cancel (or presses Esc), use the InputQuery function.

The ShowMessage is another simple way of displaying a message to the user.

```
procedure TForm1.Button1Click(Sender: TObject);
var s, s1: string;
    b: boolean;
begin
  s := Trim(InputBox('New Password', 'Password', 'masterkey'));
  b := s <> '';
  s1 := s;

  if b then
    b := InputQuery('Confirm Password', 'Password', s1);

  if not b or (s1 <> s) then
    ShowMessage('Password Failed');
end;
```

Reference :

7/16/98 4:34:11 PM

Last Modified: 01-SEP-99

## Article #16427: Create a new file with the .wav extension.

Technical Information Database

TI1427D.txt Create a new file with the .wav extension.

Category :General Programming

Platform :All

Product :Delphi All

### Description:

This document describes the process for creating added functionality ,that many Delphi users have requested, to the TMediaPlayer. The new functionality is the ability to create a new file with the .wav format when recording. The procedure "SaveMedia" creates a record type that is passed to the MCISend command. There is an appexception that calls close media if any error occurs while attempting to open the specified file. The application consists two buttons. Button1 calls the OpenMedia and RecordMedia procedures in that order.The CloseMedia procedure is called whenever an exception is generated in this application. Button2 calls the StopMedia,SaveMedia, and CloseMedia procedures.

```
unit utestrec;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls,  
Forms, Dialogs, MPlayer, MMSystem, StdCtrls;
```

```
type
```

```
TForm1 = class(TForm)
```

```
Button1: TButton;
```

```
Button2: TButton;
```

```
procedure Button1Click(Sender: TObject);
```

```
procedure Button2Click(Sender: TObject);
```

```
procedure FormCreate(Sender: TObject);
```

```
procedure AppException(Sender: TObject; E: Exception);
```

```
private
```

```
FDeviceID: Word;
{ Private declarations }
public
  procedure OpenMedia;
  procedure RecordMedia;
  procedure StopMedia;
  procedure SaveMedia;
  procedure CloseMedia;
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
var
  MyError, Flags: Longint;

procedure TForm1.OpenMedia;
var
  MyOpenParms: TMCI_Open_Parms;
  MyPChar: PChar;
  TextLen: Longint;
begin
  Flags:=mci_Wait or mci_Open_Element or mci_Open_Type;
  with MyOpenParms do
    begin
      dwCallback:=Handle; // TForm1.Handle
      lpstrDeviceType:=PChar('WaveAudio');
      lpstrElementName:=PChar('');
    end;
  MyError:=mciSendCommand(0, mci_Open, Flags,
    Longint(
MyOpenParms));
  if MyError = 0 then
    FDeviceID:=MyOpenParms.wDeviceID;
  end;

procedure TForm1.RecordMedia;
var
  MyRecordParms: TMCI_Record_Parms;
  TextLen: Longint;
begin
```

```

Flags:=mci_Notify;
with MyRecordParms do
begin
    dwCallback:=Handle; // TForm1.Handle
    dwFrom:=0;
    dwTo:=10000;
end;
MyError:=mciSendCommand(FDeviceID, mci_Record, Flags,
Longint(
MyRecordParms));
end;

procedure TForm1.StopMedia;
var
    MyGenParms: TMCI_Generic_Parms;
begin
if FDeviceID <> 0 then
begin
    Flags:=mci_Wait;
    MyGenParms.dwCallback:=Handle; // TForm1.Handle
    MyError:=mciSendCommand(FDeviceID, mci_Stop, Flags,
    Longint(
MyGenParms));
end;
end;

procedure TForm1.SaveMedia;
type // not implemented by Delphi
    PMCI_Save_Parms = ^TMCI_Save_Parms;
    TMCI_Save_Parms = record
        dwCallback: DWord;
        lpstrFileName: PAnsiChar; // name of file to save
end;
var
    MySaveParms: TMCI_Save_Parms;
begin
if FDeviceID <> 0 then
begin
    // save the file...
    Flags:=mci_Save_File or mci_Wait;
    with MySaveParms do
        begin
            dwCallback:=Handle;
            lpstrFileName:=PChar('c:\message.wav');
        end;
end;
end;

```

```

        MyError:=mciSendCommand(FDeviceID, mci_Save, Flags,
        Longint(
MySaveParms));
    end;
end;

procedure TForm1.CloseMedia;
var
    MyGenParms: TMCI_Generic_Parms;
begin
    if FDeviceID <> 0 then
    begin
        Flags:=0;
        MyGenParms.dwCallback:=Handle; // TForm1.Handle
        MyError:=mciSendCommand(FDeviceID, mci_Close, Flags,
        Longint(
MyGenParms));
        if MyError = 0 then
            FDeviceID:=0;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    OpenMedia;
    RecordMedia;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    StopMedia;
    SaveMedia;
    CloseMedia;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Application.OnException := AppException;
end;

procedure TForm1.AppException(Sender: TObject; E: Exception);
begin
    CloseMedia;
end;

```



end.

Reference:

7/16/98 4:34:11 PM

Last Modified: 01-SEP-99

## Article #16432: Using MS Internet Explorer 3.0 OCX in Delphi 2

Technical Information Database

TI1432D.txt - Using MS Internet Explorer 3.0 OCX in Delphi 2

Category :Delphi 2.x  
Platform :All-32Bit  
Product :Delphi2.x,

Description:

Topic: Access Violation when using MS Internet Explorer 3.0 WebBrowser as an OCX in Delphi.

Problem: When you create an OCX wrapper class in Delphi to host the Internet Explorer 3.0 HTML viewer control (named TExplorer or TWebBrowser depending on the age of your IE installation) and use it in a Delphi app that calls the Navigate method of that OCX control, you'll get an access violation as well as possibly ruin your whole Win95 OLE session.

Reason: IE 3.0 calls the IOleClientSite.GetContainer method of Delphi's OCX wrapper implementation. Delphi returns an error code E\_NOTIMPL, but IE 3.0 only looks for error code E\_NOINTERFACE. IE 3.0 ignores all other error codes and plows ahead with using the bogus interface pointer, thus the access violation occurs.

Solution: In Delphi 2.0's OleCtrls.pas, modify method TOleClientSite.GetContainer to return E\_NOINTERFACE instead of E\_NOTIMPL as its function result. Note that this doesn't entirely solve the IE 3.0 error checking problem, but it at least placates it.

Important Note: Delphi Developer and Delphi C/S customers can make the change and recompile without affecting any other units. Delphi Desktop customers don't have the VCL source code, so they will need an updated DCU from Borland in order to fix it.

Special Thanks: Danny Thorpe

Reference :

3/30/99 4:12:37 PM

Last Modified: 01-SEP-99

## Article #16437: BDE and Database Desktop Locking Protocol

Technical Information Database

TI1437D.txt - BDE and Database Desktop Locking Protocol

Category :Database Programming

Platform :All Windows

Product :

Description:

Intended Audience

This information will be of benefit to anyone considering designing a database application using Delphi and the BDE.

Prerequisites

A basic knowledge or interest in Paradox locking protocols and table formats.

Purpose

To give users a better understanding of the table locking protocol.

Table and field types and features supported

Each major release of Paradox has implemented improvements to table structures since version 2.0. All Paradox table types from Paradox 1.0 through Paradox 3.5 are compatible with each other.

Paradox 4.0 adds a new data type to the table format: Binary Large Objects, commonly known as BLObs and new types of Secondary Indices. Paradox 4.0 supports two types of BLOb fields: Memo, and BLOb. Versions of Paradox prior to 4.0, or the Engine prior to 3.0, cannot read, write, or create this new table format. If you attempt to read or write a Paradox 4.0 table type in an earlier version of Paradox, it will return an error that the table is password protected.

Paradox 5.0 added several new data types to the table format: Long Integer, Time, TimeStamp, Logical, Autoincrement, BCD, Bytes. Paradox 7.0 added a descending secondary index.

Any table created or modified to include any of these newer features will default to the respective table level. The default table type created using the Database Desktop and the BDE (Borland Database Engine) is a Paradox 4.0. Although the default can be changed in the BDE configuration utility or the Database desktop configuration utility to default to Paradox 3, 4, 5 and 7 for the BDE.

Paradox 4.0 can read, write, and create Paradox table types that are compatible with Paradox 1.0 through Paradox 4.0. So a table created in Paradox 1.0 is compatible with Paradox 4.0. A table created in Engine 1.0 or 2.0 can be read by or written to by Paradox 4.0.

Paradox and the Engine do not change the table type when reading or writing. The table type is only changed when a Restructure is performed on the table.

### Paradox Locking Protocols

There are two different Paradox locking protocols: the protocol introduced with Paradox 2.0 and the protocol introduced with Paradox 4.0. These two protocols are not compatible with each other. The locking protocol has no bearing on which type of table a program can work with. There are a few programs that can support either locking protocol; however, these programs can only support one protocol at a time. We will only focus on the 4.0 locking protocol.

### Database Desktop/ Paradox 4.0 Locking Protocol

The Paradox 4.0 locking protocol is the only protocol available for Paradox 4.0 and the IDAPI Engine. The designation "Paradox 4.0 locking protocol" represents this style of locking.

### Directory Locks

Paradox 4.0 places a locking file, PDOXUSRS.LCK, in each directory where tables are being accessed. This locking file regulates concurrent access to files in the directory. The lock file references PDOXUSRS.NET, so every user must map to the data directory the same way. It also places an exclusive PARADOX.LCK file in the directory as well. It does this to prevent versions of Paradox or the Engine using the older locking system from inadvertently accessing tables.

### Working/Shareable Directories

When Paradox or Database Desktop needs to access tables in a

directory, they place a "Shareable" PDOXUSRS.LCK file in that directory and an "Exclusive" PARADOX.LCK file in that directory. This designation means that other Paradox 4.0 users can access tables in that directory. The exclusive PARADOX.LCK file is placed in this directory to keep the older, incompatible locking protocol from putting data at risk. In Paradox, this is known as a "Working" directory.

#### Private/Exclusive Directories

Paradox and Database Desktop also need a directory to store temporary files, such as the Answer table from a query. When Paradox or Paradox Runtime start, they also place an "Exclusive" PDOXUSRS.LCK file in a directory and an "Exclusive" PARADOX.LCK file in the same directory, designating that directory as the location for temporary files. This designation means that other Paradox users cannot access tables in that directory. In Paradox this is known as a "Private" directory.

#### Table Locks

Paradox 4.0 places each table lock in the directory locking file: PDOXUSRS.LCK. It no longer uses the separate table lock file of previous versions. For example, if three users are viewing the CUSTOMER.DB table and one user is restructuring the ORDERS.DB table, the PDOXUSRS.LCK file will have a shareable lock listed for each of those three users who are viewing the CUSTOMER.DB table, and an exclusive lock on ORDERS.DB for the user who is restructuring that table.

#### Paradox 4.0 Locking Protocol Concurrency

In a multi-user environment, the Paradox 4.0 locking protocol maintains concurrency, the simultaneous use of applications, through the PDOXUSRS.NET file. All users who want to share Paradox tables must map to the same PDOXUSRS.NET file in the same way using the same path, but not necessarily the same drive letter. Paradox places a PDOXUSRS.LCK and an exclusive PARADOX.LCK file in each directory where tables are being accessed preventing previous versions of Paradox from accessing files in the same directory. Each user who wants to share tables in that directory must map that directory the same way using the same drive letter and path. Paradox then places all of the locking information for that table in the PDOXUSRS.LCK file, reducing the number of files needed.

#### Network Control File

The Paradox network control file, PDOXUSRS.NET, serves as the

reference point for all lock files created by Paradox. The net file contains the users currently using the BDE and which table they're accessing. Each lock file references the network control file and contains information regarding the locks on the table and by which user, so each user must map to the same network control file in the same way, but not necessarily with the same drive letter.

For example, if you are using volume DATA on server SERVER\_1 and the network control file is in the directory \PDOXDATA each user must map \\SERVER\_1\DATA:\PDOXDATA the same way, however, each user should, but is not required to use the same drive letter. If the network you are using does not have volumes, then DATA would be a directory off the root of SERVER\_1.

If you are mapping \\SERVER\_1\DATA to the root of drive P: then each Paradox system would specify the location of PARADOX.NET as P:\PDOXDATA\. However, other users could map \\SERVER\_1\DATA to the root of drive O: and specify O:\PDOXDATA\ as the location of the network control file.

#### Configuring 16 bit Database Engine / IDAPI.CFG

The Database Engine configuration file holds the network specific information and the list of database aliases, as well as other information. You can configure IDAPI using the Database Engine configuration program, BDECFG.EXE, to set the location of the network control file. Also add, delete, modify database aliases including which driver or type of alias used and whether IDAPI will share local tables with other programs using the Paradox 4.0 locking protocol as well as some other specifics regarding the tables and how data is displayed.

#### Local Settings 16 bit

The WIN.INI file holds the locations of the IDAPI.CFG file, the Database Desktop "Working" directory, and the Database Desktop "Private" directory. You can use any text editor to change these designations in the WIN.INI file. The location of the IDAPI.CFG file is CONFIGFILE= or CONFIGFILE01= in the [IDAPI] group.

The locations of the Database Desktop "Working" and "Private" directories are in the [DBD] group as WORKDIR=, and PRIVDIR=.

#### Configuring 32 bit Database Engine / IDAPI32.CFG

The BDE configuration file holds the same information as the Database Engine configuration file. Use the BDE Configuration, BDECFG32.EXE, to configure IDAPI32.CFG. Optionally you can store the information in the registry or in both the registry and IDAPI32.CFG.

#### Local Settings 32 bit

The registry holds the locations of the IDAPI32.CFG, the "Working" directory, and the "Private" directory. The location of the IDAPI32.CFG file is stored in HKEY\_LOCAL\_MACHINE\Software\Borland\Database Engine. The value CONFIGFILE01 holds the data containing .

The location of the BDE "Working" and "Private" directories are stored in

HKEY\_CURRENT\_USER\Software\Borland\DBD\7.0\Configuration\WorkDir and HKEY\_CURRENT\_USER\Software\Borland\DBD\7.0\Configuration\PrivDir respectively. Each directory default value stores the data containing

.

#### Accessing a Paradox Table

The BDE will first try to access the PDOXUSRS.NET file. If a PDOXUSRS.NET file is not found, Paradox will create a new PDOXUSRS.NET file and continue with the startup procedure. If the PDOXUSRS.NET file is found but the owner of this net file used a different path, i.e. mapped to the server differently, an exception of "Multiple net files in use" will be raised and the BDE will shutdown. After the net is successfully opened an exclusive lock, PARADOX.LCK, is placed in the temporary, private, directory. If it fails to place the lock the BDE will shut down. This can fail if some other user has an exclusive lock in this directory or the lock files are using different net files. After it secures a directory for private use it will place a shareable PARADOX.LCK in the working directory and now Initialization is complete.

Reference:

3/31/99 10:21:50 AM

Last Modified: 01-SEP-99



## Article #16442: Getting a record member char array into a memo.

Technical Information Database

TI1442D.txt - Getting a record member char array into a memo.

Category :Delphi 1.x  
Platform :All Windows  
Product :

### Description:

Handling large strings with the 16 bit Delphi product can be difficult. Especially when the strings are part of a record structure and you would like to flow them into a TMemo. This document shows how to create a record structure that has a 1000 character member and still write it out from a memo then read it back into a memo. The main thrust of the method is to use the GetTextBuf method of the memo. The record structure used is just a string and the array of 1000 chars, but it could be much more complex.

```
unit URcrdIO;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls,dbtables;
```

```
type
```

```
  TForm1 = class(TForm)  
    Button1: TButton;  
    Memo1: TMemo;  
    Button2: TButton;  
    procedure Button1Click(Sender: TObject);  
    procedure Button2Click(Sender: TObject);
```

```
  private
```

```
    { Private declarations }
```

```
  public
```

```
    { Public declarations }
```

```
  end;
```

```
type
```

```
TMyRec = record
  MyArray : array [1..1000] of char;
  mystr : string;
end;
```

```
var
  Form1: TForm1;
  MyRec : TMyRec;
  mylist : TStringlist;
  PMyChar : PChar;
  myfile : file;
  mb : TStream;
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  assignfile(myfile, 'c:\testblob.txt');
  rewrite(myfile,1);
  fillchar(MyRec.MyArray,sizeof(MyRec.MyArray),#0);
  pmychar:=
MyRec.MyArray;
  StrPCopy(pmychar,memo1.text);
  Blockwrite(MyFile,MyRec,SizeOf(MyRec));
  closefile(MyFile);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
  assignfile(myfile, 'c:\testblob.txt');
  reset(myfile,1);
  fillchar(MyRec.MyArray,sizeof(MyRec.MyArray),#0);

  Blockread(MyFile,MyRec,SizeOf(MyRec));
  pmychar:=
MyRec.MyArray;
  Memo1.SetTextBuf(pmychar);
```

```
end;
```

```
end.
```

```
Reference:
```

3/30/99 1:21:06 PM

Last Modified: 01-SEP-99

## Article #16447: How to Get the Most Out of DBDEMOS

Technical Information Database

TI1447D.txt - How to Get the Most Out of DBDEMOS

Category :Delphi 2.x  
Platform :All Windows  
Product :Delphi2.x,

Description:

OVERVIEW:

To most Delphi Developers, DBDEMOS is just an alias, but this TI will show that DBDEMOS is more than just sample tables. Included with Delphi 2.0 is the Delphi Demo Explorer where you can easily choose to view a wide range of sample applications, based on concept or by example. Also included in this TI is a directory listing of where to find various other examples and demonstrations included with the Delphi 2.0 CD.

TARGET AUDIENCE:

This TI is aimed at all developers, from beginning to master programmers. Although beginners will be able to run though these examples in order to grasp fundamental Delphi concepts, the master programmers can use this as a reference sheet in order to gain more experience and to research further in any particular area.

DELPHI 2.0 DEMO EXPLORER

-----  
The Demo Explorer allows you to browse through a database of all Delphi demonstration programs. As outlined in `\info\Borland\Demoexpl\Readdx.Txt` on the Delphi 2.0 CD, follow the directions below to install the Delphi 2.0 Demo Explorer:

1. Close Delphi 2.0 if it is running.
2. Create a `\DemoExpl` directory off of the directory in which you installed Delphi 2.0.  
(i.e., `c:\Program Files\Borland\Delphi 2.0\DemoExpl`).

3. Copy all of the files from the \info\borland\demoexpl directory of the Delphi 2.0 CD-ROM to the directory you created in step 1.
4. Use the Run option from the Start menu to run the REGEDIT application. Find:  
  \HKEY\_CURRENT\_USER\Software\Borland\Delphi\2.0\Experts  
key by expanding the tree view in the left pane of the Registry Editor window. If you have no experts key then you must create one.
5. Highlight the Experts key in the left pane, and right click on the right pane of the Registry Editor window. From the local menu, select New|String value, enter "DemoExplorer" as the new value, and press the Enter key.
6. Right click on DemoExplorer in the right pane, and select Modify from the local menu. This will invoke the Edit String dialog. In this dialog, enter the full path from step 1 followed by "\demoexpl.dll". For example:  
  c:\Program Files\Borland\Delphi 2.0\DemoExpl\demoexpl.dll.
7. Demo Explorer is now installed. Run Delphi 2.0, and you can invoke Demo Explorer by selecting "Explore Sample Applications" from the Help menu.

#### WHERE TO FIND DB DEMOS, DATA AND RELATED FILES

-----

1. \Demos\DB contains 17 directories of demonstration database related projects.
2. \Doc
  - a) \Dataedit
  - b) \DBCAl
3. \Demos\Data contains Interbase, Paradox, dBase tables, indexes, validation, memos, bitmaps.
4. \Iblocal\Examples contains Interbase tables Employee.GDB and INTlemp.GDB for Delphi 1.0.
5. \IntrBase\Examples) contains Interbase tables Employee.GDB and INTlemp.GDB for Delphi 2.0.
6. \Source\Samples
  - a) Ibctrls.pas
  - b) IBEvents.pas
  - c) IBProc32.pas

#### DELPHI 1.0 EXAMPLES

-----

1. Animals:  
  Simplest possible example using data aware controls

and a dBase table. See also FishFact.

2. BDEDLL:  
Demonstrates use of the BDE within a DLL for Delphi 1.0. If the customer is having trouble with a DLL that uses BDE they may check the comments in these two projects. Note: If "DB" appears in the uses clause of a unit, they are using BDE. This is because DB.PAS contains an initialization section that automatically starts the Borland Database Engine. For example, if a data aware control is dropped on the form, DB will be added to the uses clause, and when the application is run, BDE will start, even if the data aware control is removed at a later time.
3. CalcFlds:  
Demonstrates assigning values to calculated fields where looking into a secondary table is necessary.
4. Datalist:  
Populates list boxes with list of available databases, tables, fields and indexes for a given session.
5. DBAware1:  
Demos the use of TDBEdit, TDBComboBox, and TDBListbox.
6. DBAware2:  
Demos the use of TDBLookupCombo and TDBLookupList. The use of TDBLookupCombo and TDBLookupList are commonly misunderstood - this project shows these controls being used for their intended use.
7. FishFact:  
Simplest possible example using data aware controls and a Paradox table. See also Animals.
8. Format:  
Though not specifically a database application, this project demonstrates the use of the Format function to manipulate the display of floating point numbers.
9. InsQuery:  
Demonstrates inserting and deleting records from a server table using SQL statements containing parameters.
10. LinkQry:  
Displays two grids of master and detail records where the relationship between the master and child tables is performed through a parameterized query.
11. MastApp:  
A complete order entry system with functionality to add, modify, browse and report on orders, customers, parts, and items. Also contains application help, and comments within the main unit on upsizing from the default Paradox to Interbase.
12. QJoin:  
An example of joining two tables using SQL statements.

- Displays customer and order records where customer numbers match.
13. Range:  
Demonstrates the use of SetRangeStart, SetRangeEnd, ApplyRange, and CancelRange.
  14. Search:  
Demonstrates a single search for customer number executed from a button click.
  15. Stocks:  
Stock charting application including Market Browser, Industry Charts, Customer Details, Customer Charts, and reporting. The projects demonstrates dynamically population of combo boxes used in turn for dynamically created SQL statements and Chart FX.
  16. Tools:  
Review the Readme.txt file in this directory for more information on setting up these projects. Here's an excerpt from the read me:
    - \* These demos show examples of:
    - \* An example database MDI application (DBBROWSE.DPR)
    - \* An example database popup application (SQLMON.DPR)
    - \* An example of how to derive new database components for the Borland Database Engine system tables (BDETABLES.PAS)
    - \* An example of how to use the database error stack available in the VCL exception class EDbEngineError. This exception is raised when a Borland Database Engine error occurs (DBEXCEPT.\*)
  17. TwoForms:  
Shows data aware controls on two forms kept in sync by accessing a common dataset control (a TTable) located on the first form.

## DELPHI 2.0 EXAMPLES

-----

1. BdQuery:  
Demonstrates how to execute a query in a background thread.
2. CachedUp:  
This example demonstrates how cached updates can be used with live data and in conjunction with the UpdateSQL component for non-live data. See About.txt in this same directory for more information.
3. CsDemos:  
Demonstrates Client/Server concepts, including triggers, stored procedures executed from both TStoredProc and TQuery, and transactions.
4. CtrlGrid:

A Stock browser showing detail data in a TDBCtrlGrid.

5. DbErrors:

This example represents a sampling of the way that you might approach trapping a number of database errors. See comments in the data module of this project.

6. Filter:

Customer/Orders browser allowing a number of query and table based filters. A second form allows building and executing filters over the customer table at runtime.

7. Find:

Demonstrates record location and incremental searches on filtered and un filtered tables. Includes examples of FindNearest, Locate, and GoToCurrent usage.

8. GdsDemo:

Order entry application for "Global Diving Supply" with grid and single record views.. This project allows you to experiment with the effects of form inheritance. The project contains two sets of both grid and single record forms which allows you to change the "GDS standard" form and watch the effects on the inherited form. The project also demonstrates techniques for calculating fields, record filtering, and locating records that match the filter, using FindFirst, FindNext, FindPrior, or FindLast.

9. IBDemo:

Demonstrates listing, registering, and generating InterBase events. The project also includes the TIbEventAlerter component from the samples component palette page.

10. Lookup:

Example of "lookup" TFields that allow for drop down combos within a TDBGrid. This feature currently available in Delphi 2.0.

11. NavMDI:

Master and detail data displayed on separate MDI forms, and navigator displayed in the MDI parent.

12. NavSDI:

Navigator, master and detail each displayed on separate forms.

13. NdxBuild:

Utility to rebuild indexes for one or all tables within a given alias. Contains code using the BDE API (the dbiRegenIndexes function ), application level exception handler, and listing of alias for a given session.

14. QBFDemo:

This example shows a way to provide users with the ability to define their own queries. If your query returns records, then a Results Viewer displays those results; otherwise,



you'll receive a message indicating that no records matched the search.

Reference:

3/30/99 2:56:03 PM

Last Modified: 01-SEP-99

## Article #16452: Exposing the OnClick of the DBGrid control

Technical Information Database

TI1452D.txt - Exposing the OnClick of the DBGrid control

Category :VCL  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Many people want to use the OnClick of the TDBGrid. The TDBGrid has no such event. This document tells how to surface the OnClick event for the TDBGrid. The general technique applied here can be used to surface other properties for other objects. If you know that an ancestor could do it this is how to make the descendant do it. One of the powerful things done here is the addition of the csClickEvents to the ControlStyle set property of the control. This allows the control when typecast as THack to receive and correctly process the click message from windows. The assignment of the OnClick for some other control to the OnClick of the DBGrid1 gives the ability to access and use the OnClick event of a control that has no such event.

This is a hack. There are reasons that dbgrid does not surface the click event. Use this code at your own risk.

```
unit Udbgclk;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs,
  StdCtrls, Grids, DBGrids, DBTables, DB;

type
  thack = class(tcontrol);
```

```
TForm1 = class(TForm)
  DBGrid1: TDBGrid;
  Button1: TButton;
  DataSource1: TDataSource;
  Table1: TTable;
  procedure Button1Click(Sender: TObject);
  procedure FormClick(Sender: TObject);
```

```
private
  { Private declarations }
public
  { Public declarations }
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  THack(dbgrid1).controlstyle :=
    THack(dbgrid1).controlstyle + [csClickEvents];
  THack(dbgrid1).OnClick := Form1.OnClick;
end;
```

```
procedure TForm1.FormClick(Sender: TObject);
begin
  messagebeep(0);
  application.processmessages;
end;
```

```
end.
```

```
Reference:
```

```
3/30/99 3:43:30 PM
```

```
Last Modified: 01-SEP-99
```

## Article #16457: Getting runtime properties at runtime

Technical Information Database

TI1457D.txt Getting runtime properties at runtime

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

Object Property Information at Runtime.

-----

You may need to know at runtime what properties are available for a particular component at runtime. The list can be obtained by a call to `GetPropList`. The types, functions and procedures, including `GetPropList`, that allow access to this property information reside in the VCL source file `TYPINFO.PAS`.

### GetPropList Parameters

-----

```
function GetPropList(TypeInfo: PTypeInfo; TypeKinds: TTypeKinds;
  PropList: PPropList): Integer;
```

The first parameter for `GetPropList` is of type `PTypeInfo`, and is part of the RTTI (Run Time Type Information) available for any object. The record structure defined:

```
PTypeInfo = ^PTypeInfo;
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
  Kind: TTypeKind;
  Name: ShortString;
  {TypeData: TTypeData}
end;
```

The `TTypeInfo` record can be accessed through the objects `ClassInfo` property. For example, if you were getting the property list of a `TButton`, the call might look, so far, like this:

```
GetPropList(Button1.ClassInfo, ....
```

The second parameter, of type TTypeKinds, is a set type that acts as a filter for the kinds of properties to include in the list. There are a number of valid entries that could be included in the set (see TYPEINFO.PAS), but tkProperties covers the majority. Now our call to GetPropList would look like:

```
GetPropList(Button1.ClassInfo, tkProperties ....
```

The last parameter, PPropList is an array of PPropInfo and is defined in TYPEINFO.PAS:

```
PPropList = ^TPropList;  
TPropList = array[0..16379] of PPropInfo;
```

Now the call might read:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
  PropList: PPropList;  
begin  
  PropList := AllocMem(SizeOf(PropList^));  
  GetPropList(TButton.ClassInfo, tkProperties + [tkMethod],  
    PropList);  
  .  
  .  
  .
```

Getting Additional Information from the TTypeInfo Record

---

The example at the end of this document lists not just the property name, but it's type. The name of the property type resides in an additional set of structures. Let's take a second look at the TPropInfo record. Notice that it contains a PTypeInfo that points ultimately to a TTypeInfo record. TTypeInfo contains the class name of the property.

```
PPropInfo = ^TPropInfo;  
TPropInfo = packed record  
  PropType: PTypeInfo;  
  GetProc: Pointer;
```

```

SetProc: Pointer;
StoredProc: Pointer;
Index: Integer;
Default: Longint;
NameIndex: SmallInt;
Name: ShortString;
end;

```

```

PTypeInfo = ^PTypeInfo;
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
    {TypeData: TTypeData}
end;

```

#### Example code

-----

The example below shows how to set up the call to `GetPropList`, and how to access the array elements. `TForm` will be referenced in this example instead of `TButton`, but you can substitute other values in the `GetPropList` call. The visible result will be to fill the list with the property name and type of the `TForm` properties.

This project requires a `TListBox`. Enter the code below in the forms `OnCreate` event handler.

```

uses TypInfo;

procedure TForm1.FormCreate(Sender: TObject);
var
    PropList: PPropList;
    i: integer;
begin
    PropList := AllocMem(SizeOf(PropList^));
    i := 0;
    try
        GetPropList(TForm.ClassInfo, tkProperties + [tkMethod], PropList);
        while (PropList^[i] <> Nil) and (i < High(PropList^)) do
            begin
                ListBox1.Items.Add(PropList^[i].Name + ': ' +
                    PropList^[i].PropType^.Name);
            end;
    end;
end;

```

```
    Inc(i);  
    end;  
finally  
    FreeMem(PropList);  
end;  
end;
```

Reference:

7/16/98 4:34:12 PM

Last Modified: 01-SEP-99

## Article #16462: Search and replace in strings: a task made easy

Technical Information Database

TI1462D.txt Search and replace in strings: a task made easy

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

Search and replace in strings: a task made easy.

An often overlooked, yet strong feature of Delphi is its history. This product has its roots, and shares some code with Turbo Pascal version 1 released in 1983. Since then, every version of Turbo Pascal added new features, standard functions and procedures to the language. Under its current incarnation, Delphi continues to add features and syntax.

Doing search and replace on strings has been made trivial because of these 3 functions: Pos(), Delete(), and Insert(). Pos() takes two parameters, a pattern search string, and a string to find the pattern in - it returns the location of the string, or 0 if it does not exist. Delete() takes three parameters, the string to delete from, location of where to start deleting, and how much to delete. Similarly, Insert() takes three parameters too. The string that will be inserted, the string to insert into, the location to insert.

Many class properties use strings to store values, so one can use this method on any of them. For instance, the searching and replacing of an entire TMemo component might look like this:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  i : integer;
  s1 : string;
  SearchStr : string;
  NewStr : string;
```



```
    place : integer;
begin
  SearchStr := 'line';
  NewStr := 'OneEye';
  for i := 0 to Mem1.Lines.Count -1 do begin
    s1 := Mem1.Lines[i];
    Repeat
      Place := pos(SearchStr, s1);
      if place > 0 then begin
        Delete(s1, Place, Length(SearchStr));
        Insert(NewStr, s1, Place);
        Mem1.Lines[i] := s1;
      end; //if-then
    until place = 0;
  end; //for-loop
end;
```

Reference:

7/16/98 4:34:12 PM

Last Modified: 01-SEP-99

## Article #16467: Dynamically creating a TTable & fields at runtime

Technical Information Database

TI1467D.txt - Dynamically creating a TTable & fields at runtime

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Delphi allows rapid addition and configuration of database elements to a Delphi project within the design environment, but there are situations where information needed to create and configure objects is not known at design time. For instance, you may want to add the ability to add columns of calculated values (using formulas of the users own creation) to an application at runtime. So without the benefit of the design environment, Object Inspector, and TFields editor, how do you create and configure TFields and other data related components programmatically?

The following example demonstrates dynamically creating a TTable, a database table based off the TTable, TFieldDefs, TFields, calculated fields, and attaches an event handler to the OnCalc event.

To begin, select New Application from the File menu. The entire project will be built on a blank form, with all other components created on-the-fly.

In the interface section of your forms unit, add an OnCalcFields\ event handler, and a TaxAmount field to the form declaration, as shown below.

Later we will create a TTable and hook this handler to the TTable's OnCalcFields event so that each record read fires the OnCalcFields event and in turn executes our TaxAmountCalc procedure.

type

```
 TForm1 = class(TForm)
  procedure TaxAmountCalc(DataSet: TDataset);
```

```
private
    TaxAmount: TFloatField;
end;
```

In the implementation section add the OnCalc event handler as shown below.

```
procedure TForm1.TaxAmountCalc(DataSet: TDataset);
begin
    Dataset['TaxAmount'] := Dataset['ItemsTotal'] *
        (Dataset['TaxRate'] / 100);
end;
```

Create a OnCreate event handler for the form as shown below (for more information on working with event handlers see the Delphi Users Guide, Chapter 4 "Working With Code").

```
procedure TForm1.FormCreate(Sender: TObject);
var
    MyTable: TTable;
    MyDataSource: TDataSource;
    MyGrid: TDBGrid;
begin
    { Create the TTable component -- the underlying
      database table is created later. }
    MyTable := TTable.Create(Self);
    with MyTable do
    begin
        { Specify an underlying database and table.
          Note: Test.DB doesn't exist yet. }
        DatabaseName := 'DBDemos';
        TableName := 'Test.DB';

        { Assign TaxAmountCalc as the event handler to
          use when the OnCalcFields event fires for
          MyTable. }
        OnCalcFields := TaxAmountCalc;

        { Create and add field definitions to the TTable's
          FieldDefs array, then create a TField using the
```

```

    field definition information. }
with FieldDefs do
begin
    Add('ItemsTotal', ftCurrency, 0, false);
    FieldDefs[0].CreateField(MyTable);
    Add('TaxRate', ftFloat, 0, false);
    FieldDefs[1].CreateField(MyTable);
    TFloatField(Fields[1]).DisplayFormat := '##.0%';

{ Create a calculated TField, assign properties,
and add to MyTable's field definitions array. }
TaxAmount := TFloatField.Create(MyTable);
with TaxAmount do
begin
    FieldName := 'TaxAmount';
    Calculated := True;
    Currency := True;
    DataSet := MyTable;
    Name := MyTable.Name + FieldName;
    MyTable.FieldDefs.Add(Name, ftFloat, 0, false);
end;
end;

{ Create the new database table using MyTable as
a basis. }
MyTable.CreateTable;
end;

{ Create a TDataSource component and assign
to MyTable. }
MyDataSource := TDataSource.Create(Self);
MyDataSource.DataSet := MyTable;

{ Create a data aware grid, display on the
form, and assign MyDataSource to access
MyTable's data. }
MyGrid := TDBGrid.Create(Self);
with MyGrid do
begin
    Parent := Self;
    Align := alClient;
    DataSource := MyDataSource;
end;

{ Start your engines! }

```

```
MyTable.Active := True;  
Caption := 'New table ' + MyTable.TableName;  
end;
```

The following is the full source for the project.

```
unit gridcalc;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, Grids, DBGrids, ExtCtrls, DBCtrls, DB,  
  DBTables, StdCtrls;  
  
type  
  TForm1 = class(TForm)  
    procedure FormCreate(Sender: TObject);  
    procedure TaxAmountCalc(DataSet: TDataSet);  
  private  
    TaxAmount: TFloatField;  
  end;  
  
var  
  Form1: TForm1;  
  
implementation  
  
{ $R *.DFM }  
  
procedure TForm1.TaxAmountCalc(DataSet: TDataSet);  
begin  
  Dataset['TaxAmount'] := Dataset['ItemsTotal'] *  
    (Dataset['TaxRate'] / 100);  
end;  
  
procedure TForm1.FormCreate(Sender: TObject);  
var  
  MyTable: TTable;  
  MyDataSource: TDataSource;  
  MyGrid: TDBGrid;  
begin
```

```
MyTable := TTable.Create(Self);
```

```
with MyTable do
```

```
begin
```

```
  DatabaseName := 'DBDemos';
```

```
  TableName := 'Test.DB';
```

```
  OnCalcFields := TaxAmountCalc;
```

```
with FieldDefs do
```

```
begin
```

```
  Add('ItemsTotal', ftCurrency, 0, false);
```

```
  FieldDefs[0].CreateField(MyTable);
```

```
  Add('TaxRate', ftFloat, 0, false);
```

```
  FieldDefs[1].CreateField(MyTable);
```

```
  TFloatField(Fields[1]).DisplayFormat := '##.0%';
```

```
  TaxAmount := TFloatField.Create(MyTable);
```

```
with TaxAmount do
```

```
begin
```

```
  FieldName := 'TaxAmount';
```

```
  Calculated := True;
```

```
  Currency := True;
```

```
  DataSet := MyTable;
```

```
  Name := MyTable.Name + FieldName;
```

```
  MyTable.FieldDefs.Add(Name, ftFloat, 0, false);
```

```
end;
```

```
end;
```

```
MyTable.CreateTable;
```

```
end;
```

```
MyDataSource := TDataSource.Create(Self);
```

```
MyDataSource.DataSet := MyTable;
```

```
MyGrid := TDBGrid.Create(Self);
```

```
with MyGrid do
```

```
begin
```

```
  Parent := Self;
```

```
  Align := alClient;
```

```
  DataSource := MyDataSource;
```

```
end;
```

```
MyTable.Active := True;
```

```
Caption := 'New table ' + MyTable.TableName;
```

```
end;
```

end.

Reference:

3/31/99 10:20:34 AM

Last Modified: 01-SEP-99

## Article #16472: Activation and Use of the CPUWindow in the IDE

Technical Information Database

TI1472D.txt - Activation and Use of the CPUWindow in the IDE

Category :Delphi 2.x  
Platform :All Windows  
Product :Delphi2.x,

### Description:

Warning: The CPU window function is not fully tested and may fail in some cases. If you are having problems with the debugger or your program while using it you should disable it and this may solve your problem. Generally leave it off unless you have a specific need for it.

Delphi 2 has a built in feature, that is off by default, called the CPU window or DisassemblyView. It is easy to use and can be useful in debugging as well as in comparing code for optimization.

To activate this feature, run REGEDIT and edit the registry in the following way.

Go to HKEY\_CURRENT\_USER\Software\Borland\Delphi\2.0\Debugging. Once there add a new string key called "ENABLECPU". The value of the new key will be the string "1". That is all it takes. Now in thge Delphi IDE Select View|CPUWindow.This should bring up the new window.

Now to use this powerful feature to do comparative analysis on 2 snippets that do the same work with different source code use the following procedure.

Create 2 identical event handlers. Inside of each event handler place one of the snippets to be compared. Place a breakpoint on the first line within each handler. Run the application and activate the events. Compare the assembly code for each method. Is one shorter? If so this will execute faster.

Cases worthy of this analysis are lines of code that



will execute repeatedly, code that must be optimized for real time application, or code in applications that have to be as fast as possible for whatever reason.

A great example of code that accomplishes the same thing but produces different performance is the "with object do" construct. Many times the source code will be longer using the "with object do" construct but the unassembled code will be shorter. Many times you set properties in a series for dynamically created objects. The code:

```
with TObject.create do
begin
  property1 := ;
  property2 := ;
  property3 := ;
end;
```

executes faster than:

```
MyObj := TObject.create;
MyObj.Property1 := ;
MyObj.Property2 := ;
MyObj.Property3 := ;
```

Reference:

3/30/99 2:51:21 PM

Last Modified: 01-SEP-99

## Article #16477: Passing Multidimensional Arrays as Parameters

Technical Information Database

TI1477D.txt - Passing Multidimensional Arrays as Parameters

Category :Object Pascal  
Platform :All-32Bit  
Product :

### Description:

Passing an array to a procedure or function is straight forward and behaves as expected. However, passing a multi-dimensional array to a function or procedure is not handled in the same way. Consider MyArray to be defined as:

```
var  
MyArray: array[1..3, 1..5] of double;
```

And you want to pass it to a procedure called DoSomething() defined as:

```
procedure DoSomething(MyArray: array of double);  
begin  
    showmessage( floattostr( MyArray[1 , 1]));  
end;
```

One might think a simple statement like `DoSomething(MyArray);` would do the trick. Unfortunately, this is not the case. The statement `DoSomething(MyArray);` will not compile. The compiler sees the two data structures involved as different types - so it will not allow the statement. The `DoSomething()` procedure is expecting an array of doubles, but the example is passing a multi-dimensional array.

Delphi handles multi-dimensional arrays as user defined type, so there is no syntax to tell a procedure that its parameter(s) are multi-dimensional arrays - without declaring a type. Creating a type, and using

this type as the parameter is the correct method to pass a multi-dimensional array. We could just pass a pointer to the array, but inside the function, we need to typecast that pointer. What type to cast it as is the next issue. You would have to have the type defined, or declared identically in 2 places. This method just doesn't make sense.

By defining a type, we can change the process to this:

```
type
  TMyArray = array[1..3, 1..5] of double;

var
  MyArray : TMyArray;

procedure DoSomething(MyArray: TMyArray);
begin
  showmessage( floattostr( MyArray[1 , 1]));
end;
```

Now the actual call looks as we expected:

```
DoSomething(MyArray);
```

If you want to use the method of passing a pointer, your function might look like this:

```
type
  PMyArray = ^TMyArray;
  TMyArray = array[1..3, 1..5] of double;

var
  MyArray : TMyArray;

procedure DoSomething(MyArray: PMyArray);
begin
  showmessage ( floattostr( (MyArray[2,3]) ));
end;
```

Note, under 32 bit version Delphi, you do not need to dereference the MyArray variable inside DoSomething(). Under older versions you might have to refer to MyArray as MyArray^.

If you want to pass just a generic pointer, you may not

be able to use it directly. You can declare a local variable, and use it. Again, you may need to cast the local variable for older versions of PASCAL. Of course this method does offer more flexibility in data usage.

```
procedure DoSomething(MyArray: pointer);  
var  
    t : ^TMyArray;  
begin  
    t := MyArray;  
    ShowMessage ( FloatToStr( t[2,3]) );  
end;
```

Regardless, both calls that use a pointer method, will look something like:

```
MyArray[2, 3] := 5.6;  
DoSomething(  
MyArray);
```

Reference:

4/1/99 4:35:29 PM

Last Modified: 01-SEP-99

## Article #16482: Implementing Drag and Drop Functionality

Technical Information Database

TI1482D.txt     Implementing Drag and Drop Functionality

Category     :Application Interop

Platform     :All

Product     :Delphi All

### Description:

This document describes the process of creating a drag and drop functionality that could be applied to text items and graphical objects. Regular text indicates the procedural steps to execute to achieve the functionality. Text in the parentheses is explanation and background information.

1. Place the two edit boxes onto your form. {One will serve as the source for the text that you will drag and drop. The other will serve as the destination you will drop the text onto.}
2. Select the first edit box and name it SourceEdit by entering that name in the name property in the Object Inspector.
3. Select the second edit box and name it SenderEdit. {The names source and sender are not necessary. They are suggested because "source" and "sender" are default variables given to work with in the OnDragOver and OnDragDrop event procedure code blocks. The source variable is where the drag operation began, and the sender variable is the control that was dropped onto. See the help topics related to messages for further information on this and related topics.}
4. Click the SourceEdit component so it is selected and go to the Object Inspector. Set its DragMode property to dmAutomatic.
5. Select the SenderEdit and go to the events page of the Object Inspector and double click on the OnDragOver event.
6. Type exactly what is in the quotes here where the cursor is between the begin and end statements: "Accept := True;".

```
{Notice that the Accept variable is supplied for you by  
default inside of the OnDragOver event procedure code block.}
```

7. Now go to the events page of the Object Inspector and double click on the OnDragDrop event.

8. Type exactly what is in the quotes here where the cursor is between the begin and end statements:

```
" SenderEdit.Text := SourceEdit.Text ".
```

Now if you run the application you will be able to click down on the SourceEdit component and drag and drop onto the SenderEdit component. When you drop onto the SenderEdit it changes its text to whatever was in the SourceEdit.

Reference:

7/16/98 4:34:13 PM

Last Modified: 01-SEP-99

## Article #16487: Trapping Windows Messages in Delphi

Technical Information Database

TI1487D.txt Trapping Windows Messages in Delphi

Category :General Programming

Platform :All

Product :Delphi All

### Description:

While Delphi provides many ways to trap the incoming messages to VCL controls, you may require a quick and effective method of "stealing" and replacing the windows procedure of a given window. The method presented here will effectively trap all Windows messages for any window or VCL control that has a Window handle property.

### Background:

For every window that is created, a structure is created by the system that holds information about the window. The information contained in this structure includes, among other things, the window's parent, instance information, and the address of the window's main window procedure. It is through this procedure that all Windows messages are sent for that window.

The Microsoft Windows API provides functions to both retrieve and set the values contained in this structure, making it possible to retrieve the address of a window's main procedure and reset that address to point to a user installed function.

To replace the windows procedure, we will use the API function `SetWindowsLong()`, and pass in the handle of the window we wish to work with, the constant `GWL_WNDPROC` (telling the function we wish to replace the windows procedure), and the address of our function we wish to replace it with.

When we call the `SetWindowsLong()` function, it will return the previous value we are replacing. We will want to remember that value so we can call the original procedure for all the messages we do not care to trap. We will also want to reinstall the old procedure when we are done, so that message handling returns to normal.

In our example code, we will trap the WM\_VSCROLL message of a TDBGrid component, giving us an indication that the user has scrolled the vertical scroll bar. Since all messages for the control are passed first through our procedure, we can effectively trap and do processing for any event before it is ever received by the component.

For a list of other notification messages you may be interested in trapping, you may search the Delphi's Windows help file for messages starting with WM\_.

Also note that the code has been written to compile under both sixteen and thirty-two bit versions of Delphi.

```
unit WinProcl;  
  
interface  
  
uses  
{ $IFDEF WIN32 }  
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
  Dialogs, Grids, DBGrids, DB, DBTables;  
{ $ELSE }  
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,  
  Controls, Forms, Dialogs, DB, DBTables, Grids, DBGrids;  
{ $ENDIF }  
  
type  
  TForm1 = class(TForm)  
    DBGrid1: TDBGrid;  
    Table1: TTable;  
    DataSource1: TDataSource;  
    procedure FormCreate(Sender: TObject);  
    procedure FormDestroy(Sender: TObject);  
  private  
    { Private declarations }  
  public  
    { Public declarations }  
  end;  
  
var  
  Form1: TForm1;
```



implementation

```
{ $R *.DFM }
```

type

```
{ $IFDEF WIN32 }
```

```
  WParameter = LongInt;
```

```
{ $ELSE }
```

```
  WParameter = Word;
```

```
{ $ENDIF }
```

```
  LParameter = LongInt;
```

```
{ Declare a variable to hold the window procedure we are replacing }
```

```
var
```

```
  OldWindowProc : Pointer;
```

```
function NewWindowProc(WindowHandle : hWnd;
```

```
                        TheMessage   : WParameter;
```

```
                        ParamW       : WParameter;
```

```
                        ParamL       : LParameter) : LongInt
```

```
{ $IFDEF WIN32 } stdcall; { $ELSE } ; export; { $ENDIF }
```

```
begin
```

```
{ Process the message of your choice here }
```

```
  if TheMessage = WM_VSCROLL then begin
```

```
    ShowMessage('The vertical scrollbar is scrolling!');
```

```
  end;
```

```
{ Exit here and return zero if you want }
```

```
{ to stop further processing of the message }
```

```
{ Call the old Window procedure to }
```

```
{ allow processing of the message. }
```

```
  NewWindowProc := CallWindowProc(OldWindowProc,
```

```
                                WindowHandle,
```

```
                                TheMessage,
```

```
                                ParamW,
```

```
                                ParamL);
```

```
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
{ Set the new window procedure for the control }
```

```
{ and remember the old window procedure. }
```

```

OldWindowProc := Pointer(SetWindowLong(DbGrid1.Handle,
                                     GWL_WNDPROC,
                                     LongInt(
NewWindowProc)));
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
{ Set the window procedure back }
{ to the old window procedure.  }
  SetWindowLong(DbGrid1.Handle,
                GWL_WNDPROC,
                LongInt(OldWindowProc));
end;

end.

(*
{ The program's main source file }
program WinProc;

uses
  Forms,
  WinProcl in 'WinProcl.pas' {Form1};

{$R *.RES}

begin
{$IFDEF WIN32}
  Application.Initialize;
{$ENDIF}
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
*)

{ end of ti }

```

Reference:

7/16/98 4:34:13 PM

Last Modified: 01-SEP-99

## Article #16497: Direct Commands to Printer - Passthrough/Escape

Technical Information Database

TI1497D.txt - Direct Commands to Printer - Passthrough/Escape

Category :Printing  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Although Delphi's TPrinter unit makes it easy to interface to a given printer, there are times when you may need to drop down to the printers level and send device specific escape sequences. Under 16-bit versions of Windows, this was as easy as opening the printer port, but under Windows NT, directly accessing the hardware is illegal. One solution is to use the Windows "PASSTHROUGH" escape to send an escape sequence directly to the printer. In order to use the "PASSTHROUGH" escape, it must be supported by the printer driver. Be forewarned that not all printer drivers will support this feature.

It's worth noting that the "PASSTHROUGH" escape is documented as obsolete for thirty-two bit applications. It should be a number of years before this escape goes by the way, since it is used in many commercial applications.

The example code presented is not targeted to any specific printer model. You will need to know the correct escape sequences to send to the printer you are interfacing to. Note that you must still call the BeginDoc and EndDoc methods of TPrinter. During the BeginDoc call, the printer driver initializes the printer as necessary, and during the EndDoc call, the printer driver will uninitialized the printer and eject the page. When you do make your escape call, the printer may be set for the current windows mapping mode if the printer supports scaling internally. Technically, you should not do anything that would cause the printer memory to be reset, or eject a page with an escape sequence. In other words, try to leave the printer in the same state it was in when you made your escape. This is more important on intellegent

printers (Postscript printers), and not important at all on a standard TTY line printer, where you can do just about anything you like, including ejecting pages.

Code Example:

You will need to declare a structure to hold the buffer you are sending. The structure of the buffer is defined as a word containing the length of the buffer, followed by the buffer containing the data.

Before making the escape call to pass the data, we will use the escape "QUERYESCSUPPORT" to determine if the "PASSTHROUGH" escape is supported by the print driver.

Finally, be aware that your data will be inserted directly into the printers data stream. On some printer models (Postscript), you may need to add a space to the start and end of your data to separate your data from the printer drivers data.

(Postscript is a Registered Trademark of Adobe Systems Incorporated)

```
unit Escl;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```

{ add the printers unit }
uses
    Printers;

{$R *.DFM}

{ declare the "PASSTHROUGH" structure }
type TPrnBuffRec = record
    BuffLength : word;
    Buffer : array [0..255] of char;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Buff : TPrnBuffRec;
    TestInt : integer;
    s : string;
begin
    { Test to see if the "PASSTHROUGH" escape is supported }
    TestInt := PASSTHROUGH;
    if Escape(Printer.Handle,
        QUERYYESCSUPPORT,
        sizeof(TestInt),
        @TestInt,
        nil) > 0 then begin

        { Start the printout }
        Printer.BeginDoc;

        { Make a string to passthrough }
        s := ' A Test String ';

        { Copy the string to the buffer }
        StrPCopy(Buff.Buffer, s);

        { Set the buffer length }
        Buff.BuffLength := StrLen(Buff.Buffer);

        { Make the escape }
        Escape(Printer.Canvas.Handle,
            PASSTHROUGH,
            0,
            @Buff,

```

```
nil);
```

```
{ End the printout }
```

```
Printer.EndDoc;
```

```
end;
```

```
end;
```

```
end.
```

Reference:

4/22/99 12:48:27 PM

Last Modified: 01-DEC-99

## Article #16502: Creating a form based on a string

Technical Information Database

TI1502D.txt    Creating a form based on a string  
Category     :Application Interop  
Platform     :All  
Product      :Delphi All

### Description:

Creating a form based on a string.

### OVERVIEW

This document demonstrates how to instantiate a Delphi form based on a string which specifies the name of the type. Sample code is given.

### WHO SHOULD USE THIS DOCUMENT

Anyone with a basic familiarity with Delphi programming. Applies to any version of Delphi.

### INSTANTIATING A FORM BASED ON A STRING

To be able to instantiate forms based on the string representing the names of their type, you must first register the type with Delphi. This is accomplished with the function "RegisterClass".

RegisterClass is prototyped as follows:

```
procedure RegisterClass(AClass: TPersistentClass);
```

AClass is a class of TPersistent. In other words, the class you are registering must be descended at some point from TPersistent. Since all Delphi controls, including forms, fit this requirement, we have will have no problem. This could not be used, for instance, to register classes descended directly from TObject.

Once the class has been registered, you can find a pointer to the type by passing a string to FindClass. This will



return a class reference, which you can use to create the form.

An example is in order:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  b : TForm;
  f : TFormClass;
begin
  f := TFormClass(findClass('Tform2'));
  b := f.create(self);
  b.show;
end;
```

This will create the TForm2 type that we registered with RegisterClass.

#### WORKING SAMPLE PROGRAM

Create a new project, and then add 4 more forms, for a total of 5. You can populate them with controls if you like, but for this example, it is not important.

On the first form, put down an edit control, and a pushbutton. Take all forms, but the main form, out of the AutoCreate List.

Finally, paste the following code over the code in unit1, this will give you the ability to type the Class NAME into the Edit, and it will create that form for you.

```
unit Unit1;

interface

uses
  Unit2, Unit3, Unit4, Unit5, Windows, Messages,
  SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
```

```
TForm1 = class(TForm)
  Edit1: TEdit;
  Button1: TButton;
  procedure FormCreate(Sender: TObject);
  procedure Button1Click(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;
```

```
var
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RegisterClass(Tform2);
  RegisterClass(Tform3);
  RegisterClass(Tform4);
  RegisterClass(Tform5);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f : TFormclass;
begin
  f := tformclass(findClass(edit1.text));
  with f.create(self) do
    show;
end;
```

Reference:

7/16/98 4:34:14 PM

Last Modified: 01-SEP-99

## Article #16507: Finding the color depth of a canvas

Technical Information Database

TI1507D.txt Finding the color depth of a canvas

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

When programming graphics in Windows, it is often desirable to know the number of colors available for a given canvas. This information is available through the `GetDeviceCaps()` function.

The color depth of a canvas is calculated by multiplying the number of bits per pixel that is required for a given canvas by the number of planes the canvas uses.

You can find the color depth of a canvas in by using the following code:

```
TotalNumBitsPerPixel :=  
    GetDeviceCaps(Canvas.Handle, BITSPIXEL) *  
    GetDeviceCaps(Canvas.Handle, PLANES)
```

Will give you the total number of bits used to color a pixel.

Return values of:

1 = 2 colors (monochrome)

4 = 16 colors

8 = 256 colors

15 = 32,768 colors

16 = 65,536 colors

24 = 16,777,216 colors

Reference:

7/16/98 4:34:14 PM

Last Modified: 01-SEP-99

## Article #16512: Using FindFirst and the WIN\_32\_FIND\_DATA structure

Technical Information Database

TI1512D.txt Using FindFirst and the WIN\_32\_FIND\_DATA structure  
Category :Application Interop  
Platform :All  
Product :Delphi All

### Description:

Here is how to access the WIN\_32\_FIND\_DATA structure of a TSearchRec using the FindFirst(), FindNext(), and FindClose() functions.

In this example, we will show how to display both the long and short filename for each file found. Notice in the example that the unit name is fully qualified for the FindFirst(), FindNext(), and FindClose() functions, since there is more than one implementation for these function names. The implementations that we are calling are the Delphi wrapper functions to the WinApi functions.

```
var
  sr : TSearchRec;
  R : integer;
begin
  R := Sysutils.FindFirst('C:\*.*', faAnyFile, sr);
  while R = 0 do
    begin
      Mem1.Lines.Add(sr.FindData.cFileName);
      if sr.FindData.cAlternateFileName <> '' then
        Mem1.Lines.Add(sr.FindData.cAlternateFileName) else
        Mem1.Lines.Add(sr.FindData.cFileName);
      R := Sysutils.FindNext(sr);
    end;
  Sysutils.FindClose(sr);
end;
```

Reference:

7/16/98 4:34:14 PM

Last Modified: 01-SEP-99

## Article #16517: Setting the pixels per inch property of TPrinter

Technical Information Database

TI1517D.txt - Setting the pixels per inch property of TPrinter

Category :Printing  
Platform :All-32Bit  
Product :

### Description:

When changing printers, be aware that font sizes may not always scale properly. To ensure proper scaling, set the PixelsPerInch property of the font after changing the printer index property. Be sure not to make the change until you have started the print job.

Here are two examples:

```
*****  
uses Printers;
```

```
var  
    MyFile: TextFile;  
begin  
    Printer.PrinterIndex := 2;  
    AssignPrn(MyFile);  
    Rewrite(MyFile);  
    Printer.Canvas.Font.Name := 'Courier New';  
    Printer.Canvas.Font.Style := [fsBold];  
    Printer.Canvas.Font.PixelsPerInch:=  
        GetDeviceCaps(Printer.Canvas.Handle, LOGPIXELSY);  
    Writeln(MyFile, 'Print this text');  
    System.CloseFile(MyFile);  
end;
```

```
*****  
  
uses Printers;
```

```
begin
  Printer.PrinterIndex := 2;
  Printer.BeginDoc;
  Printer.Canvas.Font.Name := 'Courier New';
  Printer.Canvas.Font.Style := [fsBold];
  Printer.Canvas.Font.PixelsPerInch:=
    GetDeviceCaps(Printer.Canvas.Handle, LOGPIXELSY);
  Printer.Canvas.Textout(10, 10, 'Print this text');
  Printer.EndDoc;
end;
```

Reference:

4/1/99 4:36:42 PM

Last Modified: 01-SEP-99



## Article #16522: How to use a string table resource

Technical Information Database

TI1522D.txt    How to use a string table resource

Category     :Application Interop

Platform     :All

Product      :Delphi   All

### Description:

Stringtable resources are a very useful tool when your application must store a large number of strings for use at runtime. While you may be tempted to directly embed strings into your executable, using a stringtable resource offers two advantages: 1) The strings contained in the stringtable do not consume memory until they are loaded by your application. 2) Stringtables are easily edited, providing an easy path to internationally localized versions of your application.

Stringtables are compiled into a ".res" file that is attached to your application's exe file at build time. Even after you distribute your application, the stringtable contained in your application's exe file can be edited with a resource editor. My favorite resource editor is Borland's Resource Workshop that ships with the RAD pack. It can produce and edit both 16 and 32 bit resources that are self contained, standalone, or embedded in a .exe or .dll in full WYSIWYG fashion.

It's worth noting that all versions of Delphi ship with the Borland Resource Command Line Compiler (BRCC.EXE and BRCC32.EXE), and can be found in Delphi's Bin directory.

For our example, we will build an internationalized application that displays two buttons. The buttons will have captions for "Yes" and "No" presented in English, Spanish, and Swedish.

It's worth noting that if you want to build international applications using Delphi, you should take a look at Borland's Delphi Translation Suite and Language Pack software. These packages can make porting your application to international

markets a snap!

Example:

We first must create a text file containing our string resources in the applications build directory. You may name the file anything you wish, so long as it has the file extension ".rc" and the filename without the extension is not the same as any unit or project filename. This is very important, as Delphi also will create a number of resource files for your project automatically.

Here is the contents of the .rc file for our example. It contains the words "Yes" and "No" in English, Spanish, and Swedish:

```
STRINGTABLE
{
  1, "&Yes"
  2, "&No"
  17, "&Si"
  18, "&No"
  33, "&Ja"
  34, "&Nej"
}
```

The file starts with the key word stringtable denoting that a string table resource will follow. Enclosed in the curly braces are the strings. Each string is listed by its index identifier, followed by the actual string data in quotes. Each string may contain up to 255 characters. If you need to use a non-standard character, insert the character as a backslash character followed by the octal number of the character you wish to insert. The only exception is when you want to embed a backslash character, you will need to use two backslashes. Here are two examples:

```
1, "A two\012line string"
2, "c:\\Borland\\Delphi"
```

The Index numbers that you use are not important to the resource compiler. You should keep in mind that string

tables are loaded into memory in 16 string segments.

To compile the .rc file to a .res file that can be linked with your application, simply type on the dos command line the full path to the resource compiler, and the full path to the name of the .rc file to compile. Here is an example:

```
c:\Delphi\Bin\brcc32.exe c:\Delphi\strtbl32.rc
```

When the compiler is finished, you should have a new file with the same name as the .rc file you've compiled, only with an extension of ".res".

You can link the resource file with your application simply by adding the following statement to your application's code, substituting the name of your resource file:

```
{ $R ResFileName.RES }
```

Once the .res file is linked to your program, you can load the resource from any module, even if you specified the \$R directive in the implementation section of a different unit.

Here is an example of using the Windows API function LoadString(), to load the third string contained in a string resource into a character array:

```
if LoadString(hInstance,  
             3,  
             a,  
             sizeof(a)) <> 0 then ....
```

In this example, the LoadString() function accepts the hInstance of the module containing the resource, the string index to load, the address of the character array to load the string to, and the size of the character array. The LoadString function returns the number of characters that were actually loaded not including the null terminator. Be aware that this can differ from the number of bytes loaded when using unicode.

Here is a complete example of creating an international

application with Borland's Delphi. The application is compatible with both 16 and 32 bit versions of Delphi.

To do this, you will need to create two identical .rc files, one for the 16 bit version, and the other for the 32 bit version, since the resources needed for each platform are different. In this example. we will create one file named STRTBL16.rc and another called STRTBL32.rc. Compile the STRTBL16.rc file using the BRCC.exe compiler found in Delphi 1.0's bin directory, and compile STRTBL32.rc using the BRCC32.exe compiler found in Delphi 2.0's bin directory.

We have taken into account the language that Windows is currently using at runtime. The method for getting this information differs under 16 and 32 bit Windows. To make the code more consistant, we have borrowed the language constants from the Windows.pas file used in 32 bit versions of Delphi.

```
{ $IFDEF WIN32 }
    { $R STRTBL32.RES }
{ $ELSE }
    { $R STRTBL16.RES }
    const LANG_ENGLISH = $09;
    const LANG_SPANISH = $0a;
    const LANG_SWEDISH = $1d;
{ $ENDIF }

function GetLanguage : word;
{ $IFDEF WIN32 }
{ $ELSE }
    var
        s : string;
        i : integer;
{ $ENDIF }
begin
{ $IFDEF WIN32 }
    GetLanguage := GetUserDefaultLangID and $3ff;
{ $ELSE }
    s[0] := Char(GetProfileString('intl',
                                'sLanguage',
```

```

                                'none',
s[1],
                                sizeof(s)-2));
    for i := 1 to length(s) do
        s[i] := UpCase(s[i]);
    if s = 'ENU' then GetLanguage := LANG_ENGLISH else
    if s = 'ESN' then GetLanguage := LANG_SPANISH else
    if s = 'SVE' then GetLanguage := LANG_SWEDISH else
        GetLanguage := LANG_ENGLISH;
{$ENDIF}
end;

procedure TForm1.FormCreate(Sender: TObject);
var
    a : array[0..255] of char;
    StrTblofs : integer;
begin
    {Get the current language and stringtable offset}
    case GetLanguage of
        LANG_ENGLISH : StrTblofs := 0;
        LANG_SPANISH : StrTblofs := 16;
        LANG_SWEDISH : StrTblofs := 32;
    else
        StrTblofs := 0;
    end;

    {Load language dependent "Yes" and set the button caption}
    if LoadString(hInstance,
        StrTblofs + 1,
a,
        sizeof(a)) <> 0 then
        Button1.Caption := StrPas(a);

    {Load language dependent "No" and set the button caption}
    if LoadString(hInstance,
        StrTblofs + 2,
a,
        sizeof(a)) <> 0 then
        Button2.Caption := StrPas(a);
end;

```

Reference :

7/16/98 4:34:14 PM

Last Modified: 01-SEP-99

## Article #16532: TRichEdit Printing in Delphi 2 & Windows NT 4.0

Technical Information Database

TI1532D.txt - TRichEdit Printing in Delphi 2 & Windows NT 4.0

Category :Delphi 2.x  
Platform :All Windows  
Product :Delphi2.x,

### Description:

TRichEdit Printing Under Windows NT 4.0

This document provides a Delphi Unit that will solve the "divide by zero" error that occurs when printing from a TRichEdit control under Windows NT 4.0. To use this Unit simply include it in the USES clause of the Unit that you want to print from. Instead of calling the RichEdit.Print() method to print, call the PrintRichEdit() procedure and pass it the TRichEdit control that you want to print as a parameter.

If you own the VCL source code you can make the changes shown below to the TCustomRichEdit.Print method in the COMCTRLS.PAS file.

```
unit PrtRichU;

interface

uses
  ComCtrls;

procedure PrintRichEdit(const Caption: string;
                       const RichEdt: TRichEdit);

implementation

uses
  Windows, RichEdit, Printers;

procedure PrintRichEdit(const Caption: string;
                       const RichEdt: TRichEdit);

var
```

```

Range: TFormatRange;
LastChar, MaxLen, LogX, LogY, OldMap: Integer;
begin
  FillChar(Range, SizeOf(TFormatRange), 0);
  with Printer, Range do
  begin
    BeginDoc;
    hdc := Handle;
    hdcTarget := hdc;
    LogX := GetDeviceCaps(Handle, LOGPIXELSX);
    LogY := GetDeviceCaps(Handle, LOGPIXELSY);
    if IsRectEmpty(RichEdt.PageRect) then
    begin
      rc.right := PageWidth * 1440 div LogX;
      rc.bottom := PageHeight * 1440 div LogY;
    end
    else begin
      rc.left := RichEdt.PageRect.Left * 1440 div LogX;
      rc.top := RichEdt.PageRect.Top * 1440 div LogY;
      rc.right := RichEdt.PageRect.Right * 1440 div LogX;
      rc.bottom := RichEdt.PageRect.Bottom * 1440 div LogY;
    end;
    rcPage := rc;
    Title := Caption;
    LastChar := 0;
    MaxLen := RichEdt.GetTextLen;
    chrg.cpMax := -1;
    OldMap := SetMapMode(hdc, MM_TEXT);
    SendMessage(RichEdt.Handle, EM_FORMATRANGE, 0, 0);
    try
      repeat
        chrg.cpMin := LastChar;
        LastChar := SendMessage(RichEdt.Handle, EM_FORMATRANGE, 1,
          Longint(
Range));
        if (LastChar < MaxLen) and (LastChar <> -1) then NewPage;
      until (LastChar >= MaxLen) or (LastChar = -1);
    EndDoc;
  finally
    SendMessage(RichEdt.Handle, EM_FORMATRANGE, 0, 0);
    SetMapMode(hdc, OldMap);
  end;
end;
end;
end;

```



end.

Reference:

3/30/99 2:49:11 PM

Last Modified: 01-SEP-99

## Article #16537: How to use a user defined resource.

Technical Information Database

TI1537D.txt How to use a user defined resource.

Category :General Programming

Platform :All

Product :Delphi All

Description:

(\*

The following demonstrates linking a user defined resource.

Resources are compiled into a ".res" file that is attached to your application's .exe file at build time. Even after you distribute your application, the resources contained in your application's .exe file can be edited with a resource editor. The resource editor in Borland's Resource Workshop, that ships with the RAD pack, can produce and edit both 16 and 32 bit resources that are self contained, standalone, or embedded in a .exe or .dll in full WYSIWYG fashion.

It's worth noting that all versions of Delphi ship with the Borland Resource Command Line Compiler (BRCC.EXE and BRCC32.EXE), and can be found in Delphi's Bin directory.

We first must create a text file containing our resource definitions in the application's build directory. You may name the file anything you wish, so long as it has the file extension ".rc" and the filename without the extension is not the same as any unit or project filename. This is very important, as Delphi also will create a number of resource files for your project automatically.

Here is the contents of the .rc file for our example:

```
MYUSERDATA MYDATATYPE TEST.TXT
```

MYUSERDATA is the resource name, MYDATATYPE is the resource type

and TEST.TXT is the name of the file continuing the user data we will link in.

We must next create the TEST.TXT file, using any ASCII editor (notepad will do fine). For our example, the file needs to contain one single line:

```
Hello!
```

To compile the .rc file to a .res file that can be linked with your application, simply type on the dos command line the full path to the resource compiler, and the full path to the name of the .rc file to compile. Here is an example:

```
c:\Delphi\Bin\brcc32.exe c:\Delphi\MYRES.RC
```

When the compiler is finished, you should have a new file with the same name as the .rc file you've compiled, only with an extension of ".res".

You can link the resource file with your application simply by adding the following statement to your application's code, substituting the name of your resource file:

```
{ $R ResFileName.RES }
```

where ResFileName.RES is the actual name of the compiled .res file.

Once the .res file is linked to your program, you can load the resource from any module, even if you specified the \$R directive in a different unit.

To actually use the resource, you must make a few Windows API calls. First you will call the FindResource() function, passing the instance handle of your application, the name of the resource to load, and the resource type. If FindResource is successful, the function will return a handle to the unloaded resource.

Next, you can call the SizeOfResource() function to find the aligned size of the resource, if you want to know the actual size of the loaded resources memory block. Be forewarned that this not the actual size of the data, but rather the size of the memory block that will be allocated when you load the resource. If you want to know the actual size of the data, you will need to embed that

information in the data itself.

To load the resource, we will call the LoadResource() function, passing the handle returned to us when we called FindResource(). This function will return a handle to the loaded resource that we can pass to the LockResource() function to retrieve a pointer to the resource that we use to actually work with the data.

In our example, we will loop through the data, adding the data to a string, until we find the embedded '!' character, signaling that the end of the data has been found.

Finally, we will free the resource by first calling UnLockResource() and then FreeResource(), passing the handle of the loaded resource.

Example Code:

\*)

implementation

```
{ $R *.DFM }
```

```
{ $R MYRES.RES }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  hRes      : THandle; {handle to the resource}  
  pRes      : pointer; {pointer to the resource}  
  ResSize   : longint; {aligned size of the resource}  
  i         : integer; {counting variable}
```

```
{ $IFDEF WIN32 }
```

```
  s         : shortstring; {a string to play with}
```

```
{ $ELSE }
```

```
  s         : string;      {a string to play with}
```

```
{ $ENDIF }
```

```
begin
```

```
  {find the resource}
```

```
  hRes := FindResource(hInstance,  
                      'MYUSERDATA',  
                      'MYDATATYPE');
```

```
  if hRes = 0 then begin
```

```
    ShowMessage('Could not find the resource');
```

```
    exit;
```

```
  end;
```

```

{get the aligned size of the resource}
ResSize := SizeOfResource(hInstance, hRes);
if ResSize = 0 then begin
    ShowMessage('Nothing to load - size = 0');
    Exit;
end;
{load the resource}
hRes := LoadResource(hInstance, hRes);
if hRes = 0 then begin
    ShowMessage('Resource Load Failure');
    Exit;
end;
{Get a pointer to the resource}
pRes := LockResource(hRes);
if pRes = nil then begin
    ShowMessage('Resource Lock Failure');
    FreeResource(hRes);
    Exit;
end;

{convert the resource pointer to a string}
s := '';
i := 0;
while pChar(pRes)[i] <> '!' do begin
    s := s + pChar(pRes)[i];
    inc(i);
end;

{prove it works}
ShowMessage(s);

{unlock and free the resource}
UnlockResource(hRes);
FreeResource(hRes);
end;

end.

```

Reference:

7/16/98 4:34:15 PM

Last Modified: 01-SEP-99

## Article #16542: A better way to do pointer arithmetic

Technical Information Database

TI1542D.txt A better way to do pointer arithmetic

Category :Application Interop

Platform :All

Product :Delphi All

Description:

(\*

This document demonstrates a technique for incrementing pointers under Borland Pascal 7.0, Turbo Pascal for Windows, and Delphi.

Send it any pointer, and an offset of the pointer you really want (yes, huge pointers > 64 are supported) and it will return a pointer to that location. Unlike most other huge pointer code, this snippet will take into account the fact that you may be sending a pointer that does not have a zero based offset.

Be aware that you cannot access memory that crosses a 64k boundary in Win16. If you need to access a record that spans a 64K boundary, you must first grab the portion of the record that exists before the boundary, then grab the second portion that exists above the boundary, and piece them together.

Note the function is designed to be upwardly portable to Win32 where there is no need to do pointer manipulation since Win32 uses a flat (non-segmented) memory model where there are no 64k boundaries. While using the function call to get a pointer under Win32 is not exactly efficient, it will make porting legacy 16 bit code to Win32 a breeze!

\*)

type

PtrRec = record

Lo : Word;

Hi : Word;

end;

PHugeByteArray = ^THugeByteArray;

```

THugeByteArray = array[0..0] of Byte;

function GetBigPointer(lp : pointer;
                      Offset : LongInt) : Pointer;

begin
  {$IFDEF WIN32}
    GetBigPointer :=
PHugeByteArray(lp)^[Offset];
  {$ELSE}
    Offset := Offset + PTRREC(lp).Lo;
    GetBigPointer := Ptr(PtrRec(lp).Hi + PtrRec(Offset).Hi *
SelectorInc,
                          PtrRec(Offset).Lo);
  {$ENDIF}
end;

{Lets test it!}

procedure TForm1.Button1Click(Sender: TObject);
var
  h   : THandle; {handle to the memory block}
  p   : pointer; {pointer to the memory block}
  p2  : pointer; {pointer for testing}
  p3  : pointer; {pointer for testing}
begin
  {allocate two hundred thousand bytes of memory and zero out}
  h := GlobalAlloc(GHND, 200000);

  {get a pointer to the allocated memory}
  p := GlobalLock(h);

  {get a pointer to the byte at index 75000}
  p2 := GetBigPointer(p, 75000);

  {get a pointer to the byte 80000 bytes from p2}
  p3 := GetBigPointer(p2, 80000);

  {verify the byte p3 points to is zero}
  Mem1.Lines.Add(IntToStr(pByte(p3)^));

  {change the value of the byte p3 points to}
  pByte(p3)^ := 10;

```



```
{verify the change}  
    Mem1.Lines.Add(IntToStr(pByte(p3)^));
```

```
{free the memory}  
    GlobalUnlock(h);  
    GlobalFree(h);
```

```
end;
```

```
end.
```

Reference:

7/16/98 4:34:15 PM

Last Modified: 01-SEP-99

## Article #16547: Assuring Proper Font Scaling When Printing

Technical Information Database

TI1547D.txt Assuring Proper Font Scaling When Printing

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

When changing printers, be aware that font sizes may not always scale properly. To ensure proper scaling, set the PixelsPerInch property of the font after changing the printer index property. Be sure not to make the change until you have started the print job.

Here are two examples:

\*\*\*\*\*

```
uses Printers;
```

```
var
```

```
    MyFile: TextFile;
```

```
begin
```

```
    Printer.PrinterIndex := 2;
```

```
    AssignPrn(MyFile);
```

```
    Rewrite(MyFile);
```

```
    Printer.Canvas.Font.Name := 'Courier New';
```

```
    Printer.Canvas.Font.Style := [fsBold];
```

```
    Printer.Canvas.Font.PixelsPerInch:=
```

```
        GetDeviceCaps(Printer.Canvas.Handle, LOGPIXELSY);
```

```
    Writeln(MyFile, 'Print this text');
```

```
    System.CloseFile(MyFile);
```

```
end;
```

\*\*\*\*\*

```
uses Printers;
```

```
begin
```

```
Printer.PrinterIndex := 2;  
Printer.BeginDoc;  
Printer.Canvas.Font.Name := 'Courier New';  
Printer.Canvas.Font.Style := [fsBold];  
Printer.Canvas.Font.PixelsPerInch:=  
    GetDeviceCaps(Printer.Canvas.Handle, LOGPIXELSY);  
Printer.Canvas.Textout(10, 10, 'Print this text');  
Printer.EndDoc;  
end;
```

Reference:

7/16/98 4:34:15 PM

Last Modified: 01-SEP-99

## Article #16552: Delphi 3 file types with descriptions

Technical Information Database

TI1552D.txt - Delphi 3 file types with descriptions

Category :Delphi 3.x  
Platform :All-32Bit  
Product :

Description:

The .CAB File Format

This is the file format that Delphi now offers its users for web deployment. The cabinet format is an efficient way to package multiple files. The cabinet format has two key features: multiple files can be stored in a single cabinet (.cab file) and data compression is performed across file boundaries, which significantly improves the compression ratio. Cabinet file construction can be designed around the number of files to be compressed and the expected patterns for gaining access to them (sequential, random, all at once, or a few at a time). Delphi does not take advantage file compression across file boudaries.

The .LIC File Format

There really is no .lic file format per se. These files are generally just text files that contain a key string or two.

The .INF File Format

All inf files are made up of sections and items. Each named section contains its associated items. All inf files start with the header section. After the header, sections can be laid out in any order desired. Each header is as follows [HeaderName]. This is followed by the items: ItemA = ItemDetail. For detailed information on this topic please see the Device Information File Reference.

The .dpr file format.

The .dpr file is the central file to a delphi project. It serves as

the primary entry point for the executable. The dpr contains the references to the other files in the project and links forms with their associated units. This file should be edited with care as changes in it can prevent your project from loading. This file is critical for loading or moving(copying) the project.

The .pas file format.

This is a standard text file that can be edited in a text editor. Edit this file carefully as it may result in the loss of some advantages of the two way tool. For example pasting code for a button into the type declaration for a form does not result in the corresponding entry into the .dfm file. All pas files in a project are critical for rebuilding the project.

The .dfm file format.

This file contains the details of the objects contained in a form. It can be view as text by right clicking on the form and selecting view as text from the pop-up menu, or it can be converted to text and back using the convert.exe found in the bin directory. Caution should be used in altering this file as changes to it could prevent the IDE from being able to load the form. This file is critical to moving or rebuilding the project.

The .DOF File Format

This text file contains the current settings for project options, such as compiler and linker settings, directories, conditional directives, and command-line parameters. These settings can be customized on a project-by-project basis.

The .DSK File Format

This text file stores information about the state of your project, such as which windows are open and what position they are in. Like the .DOF file, this file can be customized project-by-project.

The .DPK File Format

This file contains the source code for a package (analogous to the .DPR

in a standard Delphi project) Like the .DPR file the .DPK file is a plain text file that can be edited (with caution) using a standard editor. One of the primary reasons you may need to do this would be if you were using the command line compiler.

#### The .DCP File Format

This binary image file consists of the actual compiled package. Symbol information and additional header information required by the IDE are all contained within the .DCP file. The IDE must have access to this file in order to build a project.

#### The .DPL File Format

This is the actual executable runtime package. This file is a Windows DLL with Delphi-specific features integrated into it. This file is essential for deployment of an application that uses a package.

#### The .DCI File Format

This text file contains both standard and user-defined code templates for use within the IDE. The file can be edited with a standard text editor or through the IDE. As with any text data file used by Delphi, modifying the file directly is discouraged.

#### The .DCT File Format

This proprietary binary file contains the user-defined component template information. This file is not meant to be edited by any means other than through the IDE. Since this file is proprietary the format and compatibility with future versions of Delphi may likely change.

#### The .TLB File Format

The .TLB file is a proprietary binary type library file. This file provides a way for identifying what types of objects and interfaces are available on an ActiveX server. Like a unit or a header file the .TLB serves as a repository for necessary symbol information

for an application. Since this file is proprietary the format and compatibility with future versions of Delphi may likely change.

#### The .DRO File Format

This text file contains information about the object repository. Each entry in this file contains specific information about each available item in the object repository. Even though this file is a standard text file it is not recommended that you edit it by hand. The repository should only be modified from the Tools|Repository menu in the IDE.

#### The .RES File Format

This standard binary windows-format resource file includes information about an application. By default Delphi creates a new .RES file every time a project is compiled into an application.

#### The .DB File Format

Files with this extension are standard Paradox files.

#### The .DBF File Format

Files with this extension are standard dBASE files.

#### The .GDB File Format

Files with this extension are standard Interbase files.

#### The .DMT File Format

This proprietary binary file contains the shipped and user-defined menu templates information. This file is not meant to be edited by any means other than through the IDE. Since this file is proprietary the format and compatibility with future versions of Delphi may likely change.

#### The .DBI File Format

This text file contains initialization information for

the Database Explorer. This file is not meant to be edited by any means other than through the Database Explorer.

#### The .DEM File Format

This text file contains some standard country-specific formats for a TMaskEdit component. As with any text data file used by Delphi, modifying the file directly is discouraged.

#### The .OCX File Format

An .OCX file is a specialized DLL which contains all or some associated functions of an ActiveX control. The OCX file can be thought of as a "wrapper" which contains the object and it's means of communications with other objects and servers.

Reference:

4/22/99 12:52:35 PM

Last Modified: 01-SEP-99



## Article #16557: Moving Projects Between Machines or Directories

Technical Information Database

TI1557D.txt Moving Projects Between Machines or Directories

Category :General Programming

Platform :All

Product :Delphi All

Description:

Moving Projects Between Machines

The main files involved in a project are the DPR, DFM, PAS, and RES. These are the main files in any project. There are other files associated with any project. This document focuses on these because they are the minimum files you need to move a project.

Moving a project between machines is easy if you follow a few simple rules. First keep all the files associated with the project in the same directory. This allows for quick movement all in one operation. To move the files copy the entire directory to a new temporary directory. Now delete all the files except those with these extensions DPR, PAS, DFM, and RES. All the other files will be recreated during the first build. Having a single directory eliminates questions as to which version of a dcu or pas file is going into the build. If third party components are in the project you should install those on to the Delphi Component Palette of the target machine before attempting to open the project on that machine.

If the project files were not all in one directory you may end up editing the DPR file to provide the appropriate new path information on the target machine. The text involved is in the uses clause of the DPR or

main project file. You may also choose to preserve the DSK files or the DOF files if you have a specific need to preserve your desktop settings or your project options settings respectively. The options file can be important in cases where you have specific unit aliases, you have specified directories under project options, use packages, version info, run parameters, and version info. The dsk file is less critical and generally not moved with a project.

The res file should be moved with the project in cases where your project uses custom cursors, icons, string resources, bitmaps, etc. These are resources included by you with the compiler directive to be linked into the project and become part of the executable. The sting resources are used for internationalization or localization. Other custom resource are visual enhancement of the project user interface.

For more detailed information on the specific file types referred to in this document see the TI on file types, TI 3213.

Reference:

7/16/98 4:34:16 PM

Last Modified: 01-SEP-99

## Article #16562: An example of drag and drop between DBGrids

Technical Information Database

TI1562D.txt - An example of drag and drop between DBGrids

Category :General Programming  
Platform :All Windows  
Product :All32Bit,

### Description:

Title: An example of drag and drop between DBGrids

This sample component and sample project demonstrates an easy way of enabling drag and drop of an arbitrary field in one data aware grid onto an arbitrary field in another data aware grid.

1. Launch Delphi x.xx (the code will work in 1 and 2 as well with some minor changes).
2. Do a File|New|Unit. Take the MyDBGrid unit (below) and paste it in the newly created unit. Do a File|Save As. Save the unit as MyDBGrid.pas.
3. Do a Component|Install Component. Switch to the Info New Package tab. Put MyDBGrid.pas in the Unit file name box. Call the package MyPackage.dpk. Hit Yes when Delphi tells you that the package will be built and installed. Hit OK when Delphi tells you that VCLxx.DPL is needed. The package will now be rebuilt and installed. You will now find the TMyDBGrid component on your Samples tab on your component palette. Close the package editor and save the package.
4. Do a File|New Application. Right click on the form (Form1) and select View As Text. Take the GridU1 form source (below) and paste it in Form1. Right click on the form and select View As Form. This may take a few moments since it's opening up the tables for you. Take the GridU1 unit (below) and paste it in the unit (Unit1).
5. Do a File|Save Project As. Save the unit as GridU1.pas. Save the project as GridProj.dpr.

6. Now, run the project and enjoy the dragging and dropping of fields inbetween or with the two grids.

```
-----  
The MyDBGGrid unit  
-----
```

```
unit MyDBGGrid;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
    Dialogs, Grids, DBGrids;  
  
type  
    TMyDBGGrid = class(TDBGGrid)  
    private  
        { Private declarations }  
        FOnMouseDown: TMouseEvent;  
    protected  
        { Protected declarations }  
        procedure MouseDown(Button: TMouseButton; Shift: TShiftState;  
            X, Y: Integer); override;  
    published  
        { Published declarations }  
        property Row;  
        property OnMouseDown read FOnMouseDown write FOnMouseDown;  
    end;  
  
procedure Register;  
  
implementation  
  
procedure TMyDBGGrid.MouseDown(Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Assigned(FOnMouseDown) then  
        FOnMouseDown(Self, Button, Shift, X, Y);  
    inherited MouseDown(Button, Shift, X, Y);  
end;  
  
procedure Register;  
begin  
    RegisterComponents('Samples', [TMyDBGGrid]);  
end;
```

```

end;

end.

-----
The GridU1 unit
-----

unit GridU1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Db, DBTables, Grids, DBGrids, MyDBGrid, StdCtrls;

type
  TForm1 = class(TForm)
    MyDBGrid1: TMyDBGrid;
    Table1: TTable;
    DataSource1: TDataSource;
    Table2: TTable;
    DataSource2: TDataSource;
    MyDBGrid2: TMyDBGrid;
    procedure MyDBGrid1MouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure MyDBGrid1DragOver(Sender, Source: TObject;
      X, Y: Integer; State: TDragState; var Accept: Boolean);
    procedure MyDBGrid1DragDrop(Sender, Source: TObject;
      X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

var
  SGC : TGridCoord;

```

```

procedure TForm1.MyDBGrid1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  DG : TMyDBGrid;
begin
  DG := Sender as TMyDBGrid;
  SGC := DG.MouseCoord(X,Y);
  if (SGC.X > 0) and (SGC.Y > 0) then
    (Sender as TMyDBGrid).BeginDrag(False);
end;

procedure TForm1.MyDBGrid1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  GC : TGridCoord;
begin
  GC := (Sender as TMyDBGrid).MouseCoord(X,Y);
  Accept := Source is TMyDBGrid and (GC.X > 0) and (GC.Y > 0);
end;

procedure TForm1.MyDBGrid1DragDrop(Sender, Source: TObject;
  X, Y: Integer);
var
  DG      : TMyDBGrid;
  GC      : TGridCoord;
  CurRow  : Integer;
begin
  DG := Sender as TMyDBGrid;
  GC := DG.MouseCoord(X,Y);
  with DG.DataSource.DataSet do begin
    with (Source as TMyDBGrid).DataSource.DataSet do
      Caption := 'You dragged "'+Fields[SGC.X-1].AsString+'";
    DisableControls;
    CurRow := DG.Row;
    MoveBy(GC.Y-CurRow);
    Caption := Caption+' to "'+Fields[GC.X-1].AsString+'";
    MoveBy(CurRow-GC.Y);
    EnableControls;
  end;
end;

end.

```

-----

## The GridUI form

-----

```
object Form1: TForm1
  Left = 200
  Top = 108
  Width = 544
  Height = 437
  Caption = 'Form1'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  PixelsPerInch = 96
  TextHeight = 13
object MyDBGrid1: TMyDBGrid
  Left = 8
  Top = 8
  Width = 521
  Height = 193
  DataSource = DataSource1
  Row = 1
  TabOrder = 0
  TitleFont.Charset = DEFAULT_CHARSET
  TitleFont.Color = clWindowText
  TitleFont.Height = -11
  TitleFont.Name = 'MS Sans Serif'
  TitleFont.Style = []
  OnDragDrop = MyDBGrid1DragDrop
  OnDragOver = MyDBGrid1DragOver
  OnMouseDown = MyDBGrid1MouseDown
end
object MyDBGrid2: TMyDBGrid
  Left = 7
  Top = 208
  Width = 521
  Height = 193
  DataSource = DataSource2
  Row = 1
  TabOrder = 1
  TitleFont.Charset = DEFAULT_CHARSET
  TitleFont.Color = clWindowText
  TitleFont.Height = -11
  TitleFont.Name = 'MS Sans Serif'
```

```
TitleFont.Style = []
OnDragDrop = MyDBGrid1DragDrop
OnDragOver = MyDBGrid1DragOver
OnMouseDown = MyDBGrid1MouseDown
end
object Table1: TTable
  Active = True
  DatabaseName = 'DBDEMOS'
  TableName = 'ORDERS'
  Left = 104
  Top = 48
end
object DataSource1: TDataSource
  DataSet = Table1
  Left = 136
  Top = 48
end
object Table2: TTable
  Active = True
  DatabaseName = 'DBDEMOS'
  TableName = 'CUSTOMER'
  Left = 104
  Top = 240
end
object DataSource2: TDataSource
  DataSet = Table2
  Left = 136
  Top = 240
end
end
```

Reference:

3/30/99 4:11:01 PM

Last Modified: 01-SEP-99



## Article #16567: Looping Through the Controls and Components Arrays

Technical Information Database

TI1567D.txt    Looping Through the Controls and Components Arrays

Category     :General Programming

Platform     :All

Product      :Delphi   All

### Description:

There are 2 important properties in Delphi that everyone using the product should know about. The components array is a property of all components that own other components. The controls array is a property of all `TWinControls` that are parents of `TWinControls`. These properties are so important because they allow you to iterate through a group of components without knowing the names or knowing what type of controls or components they are. This could be important for example if you wanted to disable or enable all the controls on a form or within a container control (such as a panel). Proper code for this iterative process is shown below for both cases.

1. Iteration through the components array of the form:

```
begin
  for i:= 0 to componentcount - 1 do
  begin
    {enter the code to act on each member of the array here}
  end;
end;
```

2. Iteration through the controls array of the form:

```
begin
  for i:= 0 to controlcount - 1 do
  begin
```

```
    {enter the code to act on each member of the array here}  
end;  
end;
```

The variable 'i' is an integer declared for iteration one array member at a time and must be declared by the user. The variables controlcount and componentcount are additional properties of TWinControl and TComponent respectively. These properties are often used in conjunction with iteration through the corresponding array properties.

Reference:

7/16/98 4:34:16 PM

Last Modified: 01-SEP-99

## Article #16572: Adding ODBC Drivers in Delphi with the BDE Administrator

Technical Information Database

TI1572D.txt - Adding ODBC Drivers in Delphi with the BDE Administrator

Category :General Programming  
Platform :All-32Bit  
Product :Delphi3.x, Delphi4.x,

### Description:

Adding ODBC Drivers in 32 bit Delphi with BDE Administrator

The minimum requirements necessary to install an ODBC driver in Delphi 3.0 are as follows:

- Microsoft ODBC Manger
- Windows 95 or NT
- Delphi Developer or Delphi Client/Server edition
- Vendor-provided ODBC driver (already installed on your system)

When using Delphi 3.0 there are now two common methods to add ODBC drivers to the BDE. The first step to using any of the methods is to first obtain and install the vendor-supplied ODBC driver onto your system. Once this first essential step is completed the next steps are fairly simple.

In the left panel on the screen is a list of drivers and data sources that have been previously configured for use with BDE applications.

### Method A:

1. First start the BDE Administrator from the Windows Start Menu (it will be in the Delphi 3.0 folder.)
2. Now select Object|ODBC administrator from the main menu. (this will bring up a list of currently installed drivers.)
3. Choose Add and then select the ODBC driver you would like to create a data source for, then click on OK.

4. Next fill in the appropriate information for your driver. (A minimal configuration will require the Data Source Name field. You will also need to fill in at least one other field that is a location specifier for the data. This could be a path in the case of Paradox or dBase tables or the Server field in the case of configuring an Interbase ODBC driver. Some non-exhaustive examples include; if you are using Interbase you would select a path to a .GDB file, if using Paradox or dBASE files you would specify the data directory containing your tables or if you were using Oracle you would specify the entry as it appears in your TNSNAMES.ORA file. Once this is done you have created a virtual driver and will be able to access your database files through the datasource your created.)

#### Method B:

1. First start the BDE Administrator from the Windows Start Menu (it will be in the Delphi 3.0 folder.)
2. Click on the database tab, then right click inside of the left panel.
3. Select New from the popup menu and then select the type of ODBC driver you would like to add and click OK.
4. Again right click in the database panel and select Apply from the popup menu.
5. Now you must select a valid ODBC DSN (Data Source Name) from the definition panel on the right and select apply.

Both of these methods result in the ability to hook a TDataset in Delphi to live data.

You may have noticed some new options from the Object|Options menu, these options allow you to select different configuration modes to view. It's advisable to have all check boxes in the Select Configuration Modes to View panel selected. When all the check boxes are selected you will be provided with an extensive list of all drivers and aliases available for your use. Without checking the ?virtual? setting you will not be able to ?see? the drivers that you have added through the MS ODBC manager but not explicitly through the BDE (as per method 2).

Reference:

4/22/99 12:46:56 PM

Last Modified: 01-SEP-99

## Article #16577: Delphi/400: Activating your License Key

Technical Information Database

TI1577D.txt - Delphi/400: Activating your License Key

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Description:

This document describes the steps to obtain and apply a permanent security key for ClientObjects/400 in the Delphi/400 package. Also refer to the instructions contained in the pamphlet "Additional Information for Delphi/400 Users" that ships with Delphi/400.

Delphi/400 software is installed to both an AS/400 and a PC. When the AS/400 part of the installation is complete, a temporary automatic key of two months(60 days) will be created. After this time functionality will be unavailable, and attempting to access the AS/400 through the supplied components will generate a protection error.

To obtain a permanent key, fill in the registration card that ships with Delphi/400 and follow the mailing directions on the card. Be sure to include your AS/400 model and serial number (these numbers are used to compute a unique key). The model and serial number can be obtained by running the following AS/400 commands:

```
DSPSYSVAL(QMODEL)  
DSPSYSVAL(QSRLNBR)
```

When you receive your permanent key, activate the key by entering the following commands at an AS/400 terminal.

```
ADDLIBLE CO400  
CALL CO400INS
```

The screen that appears allows you to enter your permanent key.

Reference :

4/22/99 12:43:20 PM

Last Modified: 01-SEP-99

## Article #16582: Displaying System Resources in Win 9x and NT 4

Technical Information Database

TI1582D.txt - Displaying System Resources in Win 9x and NT 4

Category :Application Interop  
Platform :All-32Bit  
Product :All32Bit,

Description:

### Displaying System Resources

This document describes the displaying of total system resources or the amount of available system resources for Windows 9x or Windows NT 4.0. It includes the structure of the TMemoryStatus structure, the function GlobalMemoryStatus that populates this structure as defined in Windows.pas, and the Windows API header description. It also includes a code example of populating a memo with this information in Delphi.

TMemoryStatus source can be found in Windows.PAS which is in the Source\RTL\WIN directory if you have the Client\Server Suite of Delphi.

Structure of TMemoryStatus:

```
TMemoryStatus = record
  dwLength: DWORD;
  dwMemoryLoad: DWORD;
  dwTotalPhys: DWORD;
  dwAvailPhys: DWORD;
  dwTotalPageFile: DWORD;
  dwAvailPageFile: DWORD;
  dwTotalVirtual: DWORD;
  dwAvailVirtual: DWORD;
```

Function called to populate TMemoryStatus:

```
procedure GlobalMemoryStatus(var lpBuffer: TMemoryStatus); stdcall;
```

WINAPI help for said function:



```
VOID GlobalMemoryStatus(  
    // pointer to the memory status structure  
    LPMEMORYSTATUS lpBuffer  
);
```

Code for populating a TMemo with Information about system resources:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
    Dialogs, StdCtrls;
```

```
type
```

```
    TForm1 = class(TForm)  
        Button1: TButton;  
        Mem1: TMemo;  
        procedure Button1Click(Sender: TObject);
```

```
    private
```

```
        { Private declarations }
```

```
    public
```

```
        { Public declarations }
```

```
    end;
```

```
var
```

```
    Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    MemoryStatus: TMemoryStatus;
```

```
begin
```

```
    Mem1.Lines.Clear;
```

```
    MemoryStatus.dwLength := SizeOf(MemoryStatus);
```

```
    GlobalMemoryStatus(MemoryStatus);
```

```

with MemoryStatus do
begin
// Size of MemoryStatus record
    Mem1.Lines.Add(IntToStr(dwLength) +
        ' Size of 'MemoryStatus' record');
// Per-Cent of Memory in use by your system
    Mem1.Lines.Add(IntToStr(dwMemoryLoad) +
        '% memory in use');
// The amount of Total Physical memory allocated to your system.
    Mem1.Lines.Add(IntToStr(dwTotalPhys) +
        ' Total Physical Memory in bytes');
// The amount available of physical memory in your system.
    Mem1.Lines.Add(IntToStr(dwAvailPhys) +
        ' Available Physical Memory in bytes');
// The amount of Total Bytes allocated to your page file.
    Mem1.Lines.Add(IntToStr(dwTotalPageFile) +
        ' Total Bytes of Paging File');
// The amount of available bytes in your page file.
    Mem1.Lines.Add(IntToStr(dwAvailPageFile) +
        ' Available bytes in paging file');
// The amount of Total bytes allocated to this program
// (generally 2 gigabytes of virtual space).
    Mem1.Lines.Add(IntToStr(dwTotalVirtual) +
        ' User Bytes of Address space');
// The amount of available bytes that is left to your program to use.
    Mem1.Lines.Add(IntToStr(dwAvailVirtual) +
        ' Available User bytes of address space');
end; // with
end; // procedure

end.

```

Sample Output of what is contained in Mem1.Lines:

```

32: Size of 'MemoryStatus' record in bytes
76%: memory in use
33054720: Total Physical Memory in bytes
499712: Available Physical Memory in bytes
53608448: Total Bytes of Paging File
36372480: Available bytes in paging file
2143289344: User Bytes of Address space
2135556096: Available User bytes of address space

```

Reference:

3/31/99 10:23:13 AM

Last Modified: 01-SEP-99

## Article #16587: Edit Controls that Align Under NT 4

Technical Information Database

TI1587D.txt - Edit Controls that Align Under NT 4

Category :VCL  
Platform :Windows NT  
Product :All32Bit,

### Description:

Creating an edit control that allows alignment under NT4.

### OVERVIEW

This document demonstrates how to create a descendant of TEdit that allows alignment of the text.

### WHO SHOULD USE THIS DOCUMENT

Anyone with a basic familiarity with Delphi programming.  
Applies to any version of Delphi.

### CREATING JUSTAEDIT

One of the features the TMemo surfaces that a TEdit does not is the ability to justify the text. This functionality exists via the Alignment property. Great! So why doesn't a TEdit have this functionality, or perhaps a more pertinent question is how do we add it?

This property, unfortunately can not be surfaced from an ancestor, because the functionality does not exist in any ancestor. So it needs to be implemented via the window style flags. ES\_LEFT, ES\_RIGHT, and ES\_CENTER specify the text alignment for edit controls. These flags do not have any bearing on single line edit controls...unless running NT 4, Service pack 3, and that is the reason the functionality does not exist in the native control. This functionality is now implemented in NT 4.0 but will have no effect in Win 3.1 and Windows 95.

WORKING CODE

unit JusEdit;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs,  
StdCtrls;

type

TJustaEdit = class(TEdit)

private

{ Private declarations }  
fAlignment : TAlignment;

protected

{ Protected declarations }  
procedure SetAlignment(Value: TAlignment);

public

{ Public declarations }  
procedure createParams(var Params : TCreateParams); override;

published

{ Published declarations }  
property Alignment: TAlignment read FAlignment write SetAlignment  
default taLeftJustify;

end;

procedure Register;

implementation

procedure TJustaEdit.CreateParams(var Params : TCreateParams);

var

x : Longint;

begin

inherited CreateParams(Params);

case fAlignment of

tarightjustify: x := es\_right;

taleftjustify : x := es\_left;

tacenter : x := es\_center;

end;

params.style := params.style or x;

end;

```
procedure TJustaEdit.SetAlignment;  
begin  
    if FAlignment <> Value then  
    begin  
        FAlignment := Value;  
        RecreateWnd;  
    end;  
end;  
procedure Register;  
begin  
    RegisterComponents('Samples', [TJustaEdit]);  
end;  
  
end.
```

Reference:

4/1/99 4:39:06 PM

Last Modified: 01-SEP-99

## Article #16592: How do I map a network drive in Windows NT or '95?

Technical Information Database

TI1592D.txt How do I map a network drive in Windows NT or '95?

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

How do I map a network drive in Windows NT or '95?

You can use the WNetAddConnection2 API call. The prototype for the API call is in Windows.Pas.

```
function WNetAddConnection2W(var lpNetResource: TNetResourceW;  
    lpPassword, lpUserName: PWideChar;  
    dwFlags: DWORD): DWORD; stdcall;
```

To make the call you will need to fill a lpNetResource structure with a minimum set of parameters, shown in the example below. You pass this structure as the first parameter to the call, the password, user name, and a flag that indicates whether this mapping should be persistent every time the machine is logged onto. For more info on the API itself, see Window's Programmers Reference help (find the function in Windows.pas, place your text cursor over the function call, and hit F1 to bring up help).

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    NRW: TNetResource;  
begin  
    with NRW do  
    begin  
        dwType := RESOURCETYPE_ANY;  
        lpLocalName := 'X:'; // map to this driver letter  
        lpRemoteName := 'MyServerMyDirectory';  
        // Must be filled in. If an empty string is used,  
        // it will use the lpRemoteName.  
        lpProvider := '';  
    end;  
    WNetAddConnection2(NRW, 'MyPassword', 'MyUserName',
```

```
CONNECT_UPDATE_PROFILE);  
end;
```

Reference:

7/16/98 4:34:16 PM

Last Modified: 05-OCT-99



## Article #16597: Adding shortcuts to Win95/WinNT4 Desktop/StartMenu

Technical Information Database

TI1597D.txt - Adding shortcuts to Win95/WinNT4 Desktop/StartMenu

Category :Application Interop  
Platform :All-32Bit  
Product :

Description:

Title: Adding shortcuts to the Win95/WinNT40 desktop or start menu

This sample project demonstrates an easy way to add shortcuts to your Windows 95 or Windows NT 4.0 desktop or start menu.

1. Launch Delphi.
2. In a new project, drop a TButton on the form (make sure it's called Button1). Then double click on Button1. Now you can go ahead and directly replace the code for Unit1 with the code for Unit1 below.

The program will set up a shortcut either (see the code) on the desktop or on the start menu. The shortcut will be called FooBar and it will open up your AUTOEXEC.BAT in NOTEPAD when executed.

It will read the value of the "Desktop" and "Start Menu" strings from the registry key named (under HKEY\_CURRENT\_USER):

```
Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
```

```
-----  
The Unit1 unit  
-----
```

```
unit Unit1;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
```

```
Dialogs, StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
uses
```

```
  ShlObj, ActiveX, ComObj, Registry;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
  MyObject   : IUnknown;
  MySLink    : IShellLink;
  MyPFile    : IPersistFile;
  FileName   : String;
  Directory  : String;
  WFileName  : WideString;
  MyReg      : TRegIniFile;
```

```
begin
```

```
  MyObject := CreateComObject(CLSID_ShellLink);
  MySLink := MyObject as IShellLink;
  MyPFile := MyObject as IPersistFile;
  FileName := 'NOTEPAD.EXE';
  with MySLink do begin
    SetArguments('C:\AUTOEXEC.BAT');
    SetPath(PChar(FileName));
    SetWorkingDirectory(PChar(ExtractFilePath(FileName)));
  end;
  MyReg := TRegIniFile.Create(
    'Software\Microsoft\Windows\CurrentVersion\Explorer');
```

```
// Use the next line of code to put the shortcut on your desktop
```

```
Directory := MyReg.ReadString('Shell Folders','Desktop','');

// Use the next three lines to put the shortcut on your start menu
// Directory := MyReg.ReadString('Shell Folders','Start Menu','')+
//           '\Whoa!';
// CreateDir(Directory);

WFileName := Directory+'\FooBar.lnk';
MyPFile.Save(PWChar(WFileName),False);
MyReg.Free;
end;

end.
```

Reference:

4/1/99 4:39:45 PM

Last Modified: 01-SEP-99

## Article #16602: Minimizing Application When a Form Minimizes

Technical Information Database

TI1602D.txt Minimizing Application When a Form Minimizes

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

How do I minimize the application when I minimize a secondary form? I want to replicate the behavior that occurs when I minimize the main form.

You can use the same logic found in `\source\vcl\forms.pas` that `TCustomForm` (in Delphi 3) uses when the form receives a system message to minimize. The form traps `WM_SYSCOMMAND` messages, checks to see if the command is to minimize the form, checks that the form is the `MainForm`, then performs the application object's `Minimize` method. If the message doesn't specify `SC_MINIMIZE` then we call `inherited` so that default processing can occur. Note: failure to include the call to `inherited` will probably cause bad things to happen.

```
.  
.br/>procedure WMSysCommand(var Message: TWMSysCommand);  
    message WM_SYSCOMMAND;  
.br/>.br/>procedure TCustomForm.WMSysCommand(var Message: TWMSysCommand);  
begin  
    if (Message.CmdType and $FFF0 = SC_MINIMIZE) and  
        (Application.MainForm = Self) then  
        Application.Minimize  
    else  
        inherited;  
end;
```

To get this same behavior you can move this code to your form's unit, and remove the logic that checks that the form is a `MainForm`.

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button2: TButton;
  private
    procedure WMSysCommand(var Message: TWMSysCommand);
      message WM_SYSCOMMAND;
  end;

var
  Form1: TForm1;

implementation

uses Unit2;

{$R *.DFM}

procedure TForm1.WMSysCommand(var Message: TWMSysCommand);
begin
  if (Message.CmdType and $FFF0 = SC_MINIMIZE) then
    Application.Minimize
  else
    inherited;
end;

end.

```

Reference:

7/16/98 4:34:17 PM

Last Modified: 01-SEP-99

## Article #16607: Graying Out Enabled/Disabled Data Aware Controls

Technical Information Database

TI1607D.txt - Graying Out Enabled/Disabled Data Aware Controls

Category :VCL  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Graying out enabled/disabled data aware controls

Most data aware components are capable of visually showing that they are disabled (by changing the text color to gray) or enabled (by setting the color to a user-defined windows text color). Some data aware controls such as TDBGrid, TDBRichEdit (in Delphi 3.0) and also TDBEdit (when connected to a numeric or date field) do not display this behavior.

The code below uses RTTI (Run Time Type Information) to extract property information and use that information to set the font color to gray if the control is disabled. If the control is enabled, the text color is set to the standard windows text color.

What follows is the step by step creation of a simple example which consists of a TForm with a TButton and a TDBRichEdit that demonstrates this behavior.

1. Select File|New Application from the Delphi menu bar.
2. Drop a TDataSource, a TTable, a TButton and a TDBEdit onto the form.
3. Set the DatabaseName property of the table to 'DBDEMOS'.
4. Set the TableName property of the table to 'ORDERS.DB'.
5. Set the DataSet property of the datasource to 'Table1'.
6. Set the DataSource property of the DBEdit to 'DataSource1'.
7. Set the DataField property of the DBEdit to 'CustNo'.
8. Set the Active property of the DBEdit to 'False'.

9. Add 'TypInfo' to the uses clause of the form.

Below is the actual procedure to put in the implementation section of your unit:

```
// This procedure will either set the text color of a
// dataware control to gray or the user defined color
// constant in clInfoText.

procedure SetDBControlColor(aControl: TControl);
var
  FontPropInfo: PPropInfo;
begin
  // Check to see if the control is a dataware control
  if (GetPropInfo(aControl.ClassInfo, 'DataSource') = nil) then exit
  else
    begin
  // Extract the front property
      FontPropInfo:= GetPropInfo(aControl.ClassInfo, 'Font');
  // Check if the control is enabled/disabled
      if (aControl.Enabled = false) then
  // If disabled, set the font color to gray
          TFont(GetOrdProp(aControl, FontPropInfo)).Color:=
clGrayText
        else
  // If enabled, set the font color to clInfoText
          TFont(GetOrdProp(aControl, FontPropInfo)).Color:=
clInfoText;
        end;
    end;
end;
```

The code for the buttonclick event handler should contain:

```
// This code will cycle through the Controls array and call
// SetDbControlColor for each control on your form
// making sure the font text color is set to what it
// should be.
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
  // Loop through the control array
  for i:= 0 to ControlCount-1 do
    SetDBControlColor(Controls[i]);
```



end;

Reference:

3/31/99 10:27:12 AM

Last Modified: 01-SEP-99

## Article #16612: Exposing a multi string object in COM

Technical Information Database

TI1612D.txt - Exposing a multi string object in COM

Category :ActiveX/OLE/COM/ActiveForm  
Platform :All  
Product :All32Bit,

### Description:

This document is intended for those using OLE/COM and want to expose a multi string object similar to Delphi's TStrings object. The IStrings interface for COM parallels the functionality of Delphi's TStrings object. For example, in a case where you have an automation client that contains a TStrings descendant, and you would like the automation server to process the strings, the automation server can employ an implementation of an IStrings interface to store and manipulate these strings.

The example project below contains the elements and steps needed to implement an IStrings interface in a COM automation server, and to interact with the interface from a COM automation client. You'll find the complete unit source listings next, then finally a discussion of some of the design choices made in this project.

The example project contains these units:

Project1\_TLB: A Pascal wrapper for the type library containing the interface definition.

Unit1: The interface implementation containing storage for interface properties and code to implement interface methods.

Unit2: The main form for the automation server. This unit is not strictly necessary, but provides feedback to know when methods have been successfully called

AutCli: The automation client that obtains a reference to the interface and uses the interface methods.

The general steps involved are listed below. You can compare each of these steps with the unit code that follows.

1) Create a type library and add an interface called IStr, with a single property called Items of type IStrings. See Developer's Guide, chapter 42 for more information on working with type libraries and creating interfaces in the type library editor.

2) In the previous step, adding an interface using the type library editor will cause a Pascal wrapper unit to be produced (in this example, the unit is called Unit1). Unit1 will contain shell implementations of get and set methods of the Items property. In this step you add implementation code to make the get and set methods functional. Also you need to add a method to create storage for your Items property, and to clean up that storage when the interface is no longer needed. Unit1 uses Unit2. Unit2 contains a form, memo and status bar to display status of each implementation method, for diagnostic purposes.

3) Create Unit2 containing a form with TMemo and TStatusBar. The form is used to reflect activity in Unit1.pas. Though this step is not strictly necessary, it helps visualize what's happening between the automation client and server.

4) Create an automation client. In this case the unit is called AutCli and contains a memo component and two buttons that assign the Memo's TStrings data to the IStr interface's Items property.

```
{-----}
unit Project1_TLB;

{ This file contains pascal declarations imported from a type library.
  This file will be written during each import or refresh of the type
  library editor. Changes to this file will be discarded during the
  refresh process. }

{ Project1 Library }
{ Version 1.0 }

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
```

```

const
  LIBID_Project1: TGUID = '{E6F9F3B6-FD3C-11D0-908F-00C04FC291A4}';

const

{ Component class GUIDs }
  Class_IStr: TGUID = '{E6F9F3B8-FD3C-11D0-908F-00C04FC291A4}';

type

{ Forward declarations: Interfaces }
  IIStr = interface;
  IIStrDisp = dispinterface;

{ Forward declarations: CoClasses }
  IStr = IIStr;

{ Dispatch interface for IStr Object }

  IIStr = interface(IDispatch)
    ['{E6F9F3B7-FD3C-11D0-908F-00C04FC291A4}']
    function Get_Items: IStrings; safecall;
    procedure Set_Items(const Value: IStrings); safecall;
    property Items: IStrings read Get_Items write Set_Items;
  end;

{ DispInterface declaration for Dual Interface IIStr }

  IIStrDisp = dispinterface
    ['{E6F9F3B7-FD3C-11D0-908F-00C04FC291A4}']
    property Items: IStrings dispid 1;
  end;

{ IStrObject }

  CoIStr = class
    class function Create: IIStr;
    class function CreateRemote(const MachineName: string): IIStr;
  end;

implementation

uses ComObj;

```

```

class function CoIStr.Create: IIStr;
begin
    Result := CreateComObject(Class_IStr) as IIStr;
end;

class function CoIStr.CreateRemote(const MachineName: string): IIStr;
begin
    Result := CreateRemoteComObject(MachineName, Class_IStr) as IIStr;
end;

end.
{-----}
unit Unit1;

interface

uses

ComObj, Project1_TLB, StdVCL, Classes, AxCtrls, Unit2;

type
    TIStr = class(TAutoObject, IIStr)
    private
        FItems: TStringList;
    public
        destructor Destroy; override;
        procedure Initialize; override;
        function Get_Items: TStringList; safecall;
        procedure Set_Items(const Value: TStringList); safecall;
    end;

implementation

uses ComServ, Dialogs;

procedure TIStr.Initialize;
begin
    Inherited Initialize;
    FItems := TStringList.Create;
end;

destructor TIStr.Destroy;
begin
    FItems.Free;

```

```

end;

function TIStr.Get_Items: IStrings;
begin
  Form1.Memo1.Lines.Assign(FItems);
  GetOleStrings(FItems, Result);
  Form1.StatusBar1.SimpleText := 'TIStr.Get_Items: IStrings';
end;

procedure TIStr.Set_Items(const Value: IStrings);
begin
  SetOleStrings(FItems, Value);
  Form1.Memo1.Lines.Assign(FItems);
  Form1.StatusBar1.SimpleText := 'TIStr.Set_Items(const Value:
IStrings)';
end;

initialization
  TAutoObjectFactory.Create(ComServer, TIStr, Class_IStr,
ciMultiInstance);
end.
{-----}
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  StdCtrls, ComCtrls, ToolWin, StdVCL, AxCtrls;

type
  TForm1 = class(TForm)
    StatusBar1: TStatusBar;
    Memo1: TMemo;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.

```

```

{-----}
unit AutCli;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  StdCtrls, ComCtrls, ToolWin, StdVCL, AxCtrls, Project1_TLB;

type
  TForm1 = class(TForm)
    ToolBar1: TToolBar;
    tbLinestoIStrings: TToolButton;
    tbIStringsToLines: TToolButton;
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure tbLinestoIStringsClick(Sender: TObject);
    procedure tbIStringsToLinesClick(Sender: TObject);
  public
    MyIString: IString;
  end;

var
  Form1: TForm1;

implementation

uses ComObj;

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyIString := CoIString.Create;
end;

procedure TForm1.tbLinestoIStringsClick(Sender: TObject);
var
  TempIStrings: IStrings;
begin
  GetOleStrings(Memo1.Lines, TempIStrings);
  MyIString.Set_Items(TempIStrings);
end;

```

```

procedure TForm1.tbIStringsToLinesClick(Sender: TObject);
var
  TempISttrs: IStrings;
begin
  TempISttrs := MyISttr.Get_Items;
  SetOleStrings(Memol.Lines, MyISttr.Items);
end;

end.
{-----}

```

So why was Unit1, the interface implementation created? An Ole interface, such as IStrings, can be considered a contract that properties and functions will be defined in a given format and that the functions will be implemented as defined (see Developer's Guide, Chapter 36, "An Overview of COM" for more information. The fact that you define an interface does not provide implementation. For example, to make a defined IStrings interface useable you need to provide storage for the strings and mechanism to add and retrieve strings. Any additional functionality, defined in the interface, such as a Clear or IndexOf method, must also be implemented.

You may notice in Unit1 that we do not override the Create constructor as the point to create an instance of the FItems TStrings property. Instead we override the Inialize method. The reason for this can be found in the hierarchy of components starting with our TISTR object.

```

TComObject
  TTypedComObject
    TAutoObject
      TISTR

```

In this series of objects, only TComObject has a Create method, and that method IS NOT VIRTUAL. The Create method calls the TComObject.CreateFromFactory and it in turn calls the Initilize method which IS virtual. You can see more detail of the mechanics of this hierarchy in the \source\rtl\sys\comobj.pas unit.

Reference:



Last Modified: 01-SEP-99

## Article #16617: Getting Version Information From Your Program

Technical Information Database

TI1617D.txt Getting Version Information From Your Program

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

This sample project demonstrates how to get at the version info in executables.

1. Launch Delphi 3.
2. In a new project, drop a TMemo and a TButton on the form (make sure they're called Mem01 and Button1). Double click on Button1. Now you can go ahead and directly replace the code for the Button1Click procedure (see below).

-----  
The Button1Click procedure  
-----

```
procedure TForm1.Button1Click(Sender: TObject);
const
  InfoNum = 10;
  InfoStr : array [1..InfoNum] of String =
    ('CompanyName', 'FileDescription', 'FileVersion', 'InternalName',
     'LegalCopyright', 'LegalTrademarks', 'OriginalFilename',
     'ProductName', 'ProductVersion', 'Comments');
var
  S          : String;
  n, Len, i  : Integer;
  Buf        : PChar;
  Value      : PChar;
begin
  S := Application.ExeName;
  n := GetFileVersionInfoSize(PChar(S),n);
  if n > 0 then begin
    Buf := AllocMem(n);
    Mem01.Lines.Add('FileVersionInfoSize='+IntToStr(n));
```

```
GetFileVersionInfo(PChar(S),0,n,Buf);
for i:=1 to InfoNum do
  if VerQueryValue(Buf,PChar('StringFileInfo\040904E4\'+'
                        InfoStr[i]),Pointer(Value),Len) then
    Mem1.Lines.Add(InfoStr[i]+'='+Value);
  FreeMem(Buf,n);
end else
  Mem1.Lines.Add('No FileVersionInfo found');
end;
```

Reference:

7/16/98 4:34:17 PM

Last Modified: 01-SEP-99

## Article #16627: Adding an IPersistPropertyBag to Active Controls

Technical Information Database

FAQ: TI1627D - Adding an IPersistPropertyBag to Active Controls

Category: ActiveX/OLE/COM/ActiveForm

Platform: All-32Bit

Product: All32Bit,

Description:

This document will explain how to add an IPersistPropertyBag interface to an ActiveX control. This will allow setting the properties of an ActiveX control through the use of HTML PARAM tags. Adding the IPersistPropertyBag interface to an ActiveX control will also allow it to have its properties set through the use of tools like the ActiveX Control Pad.

Adding the IPersistPropertyBag interface to an ActiveX control is a simple process. All that is necessary is to add the interface to an object's class definition and implement the interface's three methods. The following example will step you through this process using a simple ActiveX control based on TButton. In the example, for the sake of simplicity, this example will only implement this functionality for the "Caption" property. It is a simple task to extrapolate this example to provide full functionality for all properties available in a control.

Begin by using the ActiveX Control Wizard to create a ActiveX control based on TButton.

Choose File|New and select the ActiveX tab from the New Item Dialog. Then select ActiveX Control. In the next dialog box choose TButton for the VCL Class Name. The rest of the defaults in the dialog box are fine so don't change anything else. Let Delphi generate the base code for your control.

The next step is to add the IPersistPropertyBag interface to the class definition. Do this by modifying the first line of the type declaration from...

```
type
  TButtonX = class(TActiveXControl, IButtonX)
```

to...

```
type
  TButtonX = class(TActiveXControl, IButtonX, IPersistPropertyBag)
```

Now the IPersistPropertyBag interface has been added to the type declaration. Next declare the necessary methods by adding the following to the protected section.

```
function IPersistPropertyBag.InitNew = PersistPropBagInitNew;
function IPersistPropertyBag.Load = PersistPropBagLoad;
function IPersistPropertyBag.Save = PersistPropBagSave;
function PersistPropBagInitNew: HRESULT; stdcall;
function PersistPropBagLoad(const pPropBag: IPropertyBag;
  const pErrorLog: IErrorLog): HRESULT; stdcall;
function PersistPropBagSave(const pPropBag: IPropertyBag;
  fClearDirty: BOOL; fSaveAllProperties: BOOL): HRESULT;
  stdcall;
```

Then, of course, implement those functions...

```
// -- implement PersistPropBagInitNew

function TButtonX.PersistPropBagInitNew: HRESULT;
begin
  Result := S_OK;
end;

// -- implement PersistPropBagLoad

function TButtonX.PersistPropBagLoad(const pPropBag:
  IPropertyBag; const pErrorLog: IErrorLog): HRESULT; stdcall;
var
  v: OleVariant;
begin
```

```
if pPropBag.Read('Caption', v, pErrorLog) = S_OK then
    FDelphiControl.Caption := v;
Result := S_OK;
end;

// -- implement PersistPropBagSave

function TButtonX.PersistPropBagSave(const pPropBag:
    IPropertyBag; fClearDirty: BOOL; fSaveAllProperties: BOOL)
    : HRESULT; stdcall;
var
    v: OleVariant;
begin
    v:= FDelphiControl.Caption;
    pPropBag.Write('Caption', v);
    Result := S_OK;
end;
```

With that code added the Control is finished. Go ahead and build the ActiveX control and deploy it. Do this through the use of the Web Delpoy Wizard. Just set up the necessary options through Project|Web Delpoyment Options and deploy through Project|Web Deploy.

Either you or the Web Deployment Wizard will create an HTML page which will contain an OBJECT tag something like the following:



This page should work fine. However, now you have the ability to set the buttons caption through HTML by adding a PARAM tag. Your new OBJECT tag will look like the following:



The caption of the button will now say "Hello". In this example only the caption will be accessable through this method. To expose other properties, follow this example and change the name of the property you wish to use.

Reference :

8/5/99 12:49:48 PM

Last Modified: 01-SEP-99

## Article #16632: Exposing a Font Property in an Automation Server

Technical Information Database

TI1632D.txt - Exposing a Font Property in an Automation Server

Category :ActiveX/OLE/COM/ActiveForm  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This document is intended for those using OLE/COM and want to expose a font object similar to Delphi's TFont in their automation server. The IFontDisp interface for COM parallels the functionality of Delphi's TFont. For example, in a case where you have an automation client that contains an object with a font property and you would like the automation server method that acts on the the font's attributes (such as font name or toggling of bold or italics). The automation server can employ an implementation of an IFontDisp interface to store and manipulate the font.

The example project below contains the elements and steps needed to implement an IFontDisp interface in a COM automation server, and to interact with the interface from a COM automation client. You'll find the complete unit source listings next, then finally a discussion of some of the design choices made in this project.

The example project contains these units:

Project1\_TLB: A Pascal wrapper for the type library containing the interface definition.

Unit1: The interface implementation containing storage for interface properties and code to implement interface methods.

Unit2: The main form for the automation server. This unit is not strictly necessary, but provides feedback to know when methods have been successfully called

FontCli: The automation client that obtains a reference to the



interface and uses the interface methods.

The general steps involved are listed below. You can compare each of these steps with the unit code that follows.

1) Use File|New|ActiveX|Automation Object and within the Automation Object Wizard supply MyFontServer as the class name. Provide a single property called MyFont of type IFontDisp. See Developer's Guide, chapter 42 for more information on working with type libraries and creating interfaces in the type library editor.

2) In the previous step, adding an interface using the type library editor will cause a Pascal wrapper unit to be produced (in this example, the unit is called Unit1). Unit1 will contain shell implementations of get and set methods of the MyFont property. In this step you provide storage for the MyFont property, in the form of FFont (a TFont), and add implementation code to make the get and set methods functional.

Unit1 uses Unit2. Unit2 contains a form, memo and status bar to display status of each implementation method, for diagnostic purposes.

3) Create Unit2 containing a form with TMemo and TStatusBar. The form is used to reflect activity in Unit1.pas. Though this step is not strictly necessary, it helps visualize what's happening between the automation client and server.

4) Create an automation client. In this case the unit is called FontCli and contains a label that reflects the current font and a button to set MyFont on the server.

```
{-----}
unit Project1_TLB;

{ This file contains pascal declarations imported from a type library.
  This file will be written during each import or refresh of the type
  library editor. Changes to this file will be discarded during the
  refresh process. }

{ Project1 Library }
{ Version 1.0 }

interface
```

```
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
```

```
const
```

```
LIBID_Project1: TGUID = '{29C7AC94-0807-11D1-B2BA-0020AFF2F575}';
```

```
const
```

```
{ Component class GUIDs }
```

```
Class_MyFontServer: TGUID = '{29C7AC96-0807-11D1-B2BA-0020AFF2F575}';
```

```
type
```

```
{ Forward declarations: Interfaces }
```

```
IMyFontServer = interface;
```

```
IMyFontServerDisp = dispinterface;
```

```
{ Forward declarations: CoClasses }
```

```
MyFontServer = IMyFontServer;
```

```
{ Dispatch interface for MyFontServer Object }
```

```
IMyFontServer = interface(IDispatch)
```

```
['{29C7AC95-0807-11D1-B2BA-0020AFF2F575}']
```

```
function Get_MyFont: IFontDisp; safecall;
```

```
procedure Set_MyFont(const Value: IFontDisp); safecall;
```

```
property MyFont: IFontDisp read Get_MyFont write Set_MyFont;
```

```
end;
```

```
{ DispInterface declaration for Dual Interface IMyFontServer }
```

```
IMyFontServerDisp = dispinterface
```

```
['{29C7AC95-0807-11D1-B2BA-0020AFF2F575}']
```

```
property MyFont: IFontDisp dispid 1;
```

```
end;
```

```
{ MyFontServerObject }
```

```
CoMyFontServer = class
```

```
class function Create: IMyFontServer;
```

```
class function CreateRemote(const MachineName: string):
```

```
IMyFontServer;
```

```
end;
```

```

implementation

uses ComObj;

class function CoMyFontServer.Create: IMyFontServer;
begin
    Result := CreateComObject(Class_MyFontServer) as IMyFontServer;
end;

class function CoMyFontServer.CreateRemote(const MachineName: string):
    IMyFontServer;
begin
    Result := CreateRemoteComObject(MachineName, Class_MyFontServer) as
    IMyFontServer;
end;

end.
{-----}
unit Unit1;

interface

uses
    ComObj, Project1_TLB, ActiveX, Graphics;

type
    TMyFontServer = class(TAutoObject, IMyFontServer)
    private
        FFont: TFont;
    public
        procedure Initialize; override;
        destructor Destroy; override;
        function Get_MyFont: IFontDisp; safecall;
        procedure Set_MyFont(const Value: IFontDisp); safecall;
    end;

implementation

uses ComServ, AxCtrls, Unit2;

procedure TMyFontServer.Initialize;
begin

```

```

    inherited Initialize;
    FFont := TFont.Create;
end;

destructor TMyFontServer.Destroy;
begin
    FFont.Free;
    inherited Destroy;
end;

function TMyFontServer.Get_MyFont: IFontDisp;
begin
    FFont.Assign(Form2.Label1.Font);
    GetOleFont(FFont, Result);
end;

procedure TMyFontServer.Set_MyFont(const Value: IFontDisp);
begin
    SetOleFont(FFont, Value);
    Form2.Label1.Font.Assign(FFont);
end;

initialization
    TAutoObjectFactory.Create(ComServer, TMyFontServer,
        Class_MyFontServer,
            ciMultiInstance);
end.
{-----}
unit Unit2;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs,
    StdCtrls;

type
    TForm2 = class(TForm)
        Label1: TLabel;
    end;

var
    Form2: TForm2;

```

implementation

{ \$R \*.DFM }

end.

{-----}

unit FontClil;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
Dialogs,  
StdCtrls, StdVCL, Project1\_TLB;

type

TForm1 = class(TForm)  
    Button1: TButton;  
    Label1: TLabel;  
    FontDialog1: TFontDialog;  
    procedure Button1Click(Sender: TObject);  
    procedure FormCreate(Sender: TObject);  
public  
    MyFontServer: IMyFontServer;  
end;

var

Form1: TForm1;

implementation

uses ActiveX, AxCtrls;

{ \$R \*.DFM }

procedure TForm1.Button1Click(Sender: TObject);

var

Temp: IFontDisp;

begin

    if (FontDialog1.Execute) then

    begin

        Label1.Font.Assign(FontDialog1.Font);

        GetOleFont(Label1.Font, Temp);

        MyFontServer.Set\_MyFont(Temp);

    end;

```
end;  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    MyFontServer := CoMyFontServer.Create;  
end;  
  
end.
```

{-----}

So why was Unit1, the interface implementation created? An Ole interface, such as IFontDisp, can be considered a contract that properties and functions will be defined in a given format and that the functions will be implemented as defined (see Developer's Guide, Chapter 36, "An Overview of COM" for more information. The fact that you define an interface does not provide implementation. For example, to make a defined IFontDisp interface useable you need to provide storage for the font and mechanism to add and retrieve font information attributes such as font name, italics, size, etc.

Note:

GetOleFont and SetOleFont are defined in AxCtrls.pas.  
IFontDisp is defined in ActiveX.pas

Reference:

3/31/99 1:07:55 PM

Last Modified: 01-SEP-99

## Article #16637: Resolving Focus to a DBGrid After Focus Transfers

Technical Information Database

TI1637D.txt - Resolving Focus to a DBGrid After Focus Transfers

Category :VCL  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This TI shows how to resolve the inability of a DBGrid to receive focus after clicking on some other control in the parent form when the grid is on a MDI child form.

Applies to all versions of Delphi

Apparently the DBGrid has some focus control problems when on an MDI child form. This problem can be counteracted by descending from TDBGrid, and handling the mousedown messages that indicate that focus should be going back to the control. The following component fixes the problem. Note, this was written generically for all versions of Delphi. You may have problems in Delphi 2 or 3 if you do not have unit aliasing set up correctly, in which case you will want to replace "winprocs" and "wintypes" with "windows."

```
-----  
  
unit FixedDBGrid;  
  
interface  
  
uses  
    Winprocs, wintypes, Messages, SysUtils, Classes, Graphics,  
    Controls, Forms, Dialogs, Grids, DBGrids;  
  
type  
    TFixedDBGrid = class(TDBGrid)  
    private  
        { Private declarations }  
    end;  
end;
```

```
protected
  { Protected declarations }
public
  { Public declarations }
  procedure WMRButtonDown(var Message: TWMRButtonDown); message
    WM_RBUTTONDOWN;
  procedure WMLButtonDown(var Message: TWMLButtonDown); message
    WM_LBUTTONDOWN;
published
  { Published declarations }
end;
```

```
procedure Register;
```

```
implementation
```

```
procedure TFixedDBGrid.WMRButtonDown(var Message: TWMRButtonDown);
begin
  winprocs.SetFocus(handle); {remember that winprocs in Delphi 1
only!}
  inherited;
end;
procedure TFixedDBGrid.WMLButtonDown(var Message: TWMLButtonDown);
begin
  winprocs.SetFocus(handle); {remember that winprocs in Delphi 1
only!}
  inherited;
end;
procedure Register;
begin
  RegisterComponents('Samples', [TFixedDBGrid]);
end;

end.
```

Reference:

4/1/99 4:40:37 PM

Last Modified: 01-SEP-99



## Article #16642: Importing or "Wrapping" DLL Function Calls

Technical Information Database

TI1642D.txt Importing or "Wrapping" DLL Function Calls

Category :Application Interop

Platform :All

Product :Delphi All

### Description:

Importing or "wrapping" dll function calls

Two methods exist for importing or loading functions from a Dynamic Link Library(DLL). The first method (which is discussed extensively in this document) is called "implicit" loading. Implicit loading involves loading the DLL statically at startup and accessing the functions through an Object PASCAL interface. This method should be used when an application is totally dependent upon the loading of a DLL for proper functioning. The other method available is referred to as "explicit" loading because the DLL is loaded dynamically on demand. This method requires a little more coding and should be used if the application needs to run even if the DLL fails to load properly.

What is a "wrapped" function call?

A wrapped function or set of functions consists of an entry into the interface section and an entry into the implementation section (along with associated constants or types) which corresponds to a function or set of functions to be imported from a DLL. A wrapper is simply a declaration in a PASCAL unit that provides an entry Point into a DLL. With Delphi, this wrapper is represented by a unit file containing object pascal code. The Delphi development team has already conveniently wrapped many standard Windows controls and functions for you. On occasion it may become necessary to create wrappers for dll function calls that are not already wrapped in Delphi.

The first and often the most difficult step in this process is locating information about the function(s). One of the best sources for locating documentation of the function(s) in question is the World Wide Web. An extensive search beginning with the MSDN and extending to the numerous available search engines will often reveal some pertinent information. Search C++ header files in a product like Borland C++ or MS Visual C++ to get the structure of the function calls. Type conversion and calling conventions are generally able to be resolved when converting between C++ and PASCAL. A good resource for compatibility between Delphi and C++ can be found on the Borland web site at:  
"http://www.borland.com/delphi/papers/brick.html".

After locating an example or documentation of the DLL in question the next step is to create a new unit file. The interface of the unit will contain the types and constants that are specific to the function calls of the DLL along with the function headers. These function headers are the Object PASCAL interface that is being provided for other Delphi applications to call the DLL function. Once the interface section of the unit is complete the next section is the implementation. The implementation section of the unit contains declarations of the imported external functions. These headers are not identical to those found in the interface section of the unit (these contain the actual function identifiers plus other important implementation information). For a more thorough treatment of this subject see the help topic "DLLs:accessing procedures and functions" in the Delphi 3 help file.

For example, say there exists a function called BOB in a DLL called 'BLOGGE.DLL'. (detailed below are the steps required to implicitly load the DLL)

- 1) WWW Research on the function BOB reveals that it returns a boolean and takes a word and a boolean as it's arguments.
- 2) Create a new unit file named 'UseBob.pas' via Delphi (File|New and choose unit)
- 3) The following line of code would go in the interface section of the new unit:

```
function BOB(Fire: Word; Dances: Boolean): Boolean; stdcall;
```

4) The following line of code would go in the implementation section of the new unit:

```
function BOB; external 'BLODGE';
```

5) Save the unit, name it 'UseBob.pas'.

6) It is also necessary to make sure that UseBob.pas is in the same directory of the current project or that it resides in a directory that is in the Delphi search path.

7) Add the 'UseBob' unit to a new project's uses clause. The function BOB can be called from this new project just like any other standard function.

8) At runtime BLODGE.DLL must be in the path of the current process's environment.

The steps required to explicitly load the 'BLODGE.DLL' are slightly different and involve a bit of coding. As before a knowledge of the function/procedure arguments is necessary (and a result type if it's a function).

What follows is a unit that implements the BOB function call on a button click event:

```
unit UDLLTest;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls;
```

```
type
```

```
  TForm1 = class(TForm)  
    Button1: TButton;  
    procedure Button1Click(Sender: TObject);
```

```
  private
```

```
    { Private declarations }
```

```
  public
```

```
    { Public declarations }
```

```
end;
```

```

{ Here's a type which points to our bob function }
TBOB = function(Fire: Word; Dances: Boolean): Boolean; stdcall;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
    BOB: TBOB;
    hDLLInst: THandle;
    IsAlive,IsDancing: Boolean;
    Years: Word;
begin
    { Load a handle to our BLODGE.DLL }
    hDLLInst:= LoadLibrary('BLODGE.DLL');
    { If the load was unsuccessful raise a custom exception }
    if (hDLLInst <= 0) then
        raise exception.create('[LoadLibrary Fail]');
    { Try to load the address of the BOB function }
    try
BOB:= GetProcAddress(hDLLInst,'BOB');
        if not assigned(BOB) then
            raise exception.Create('[GetProcAddress Fail]');
        Years:= 25;
        IsDancing:= True;
    { Now we can execute the BOB function }
        IsAlive:= BOB(Years,IsDancing);
    finally
        { Free the handle to the DLL }
        FreeLibrary(hDLLInst);
    end;
end;
end.

```

JG

Reference:

7/16/98 4:34:18 PM

Last Modified: 01-SEP-99

## Article #16647: Implementing TCollection

Technical Information Database

TI1647D.txt - Implementing TCollection

Category :ActiveX/OLE/COM/ActiveForm  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This document is intended for those needing to descend from a class that manages an array of lightweight persistent objects of the same type. The class that best accomplishes this is TCollection and TCollectionItem. For example, TCollection is used to manage Panels in a TStatusBar, Columns in a TDBGrid, or Constraints in a TTable.

This document begins with a discussion of the expected behavior of TCollection descendants, followed by a listing of the minimal steps necessary to implement a TCollection descendant, a listing of the component source, and finally some notes on design decisions and ideas for expansion of your TCollection descendant component.

### General Discussion

-----  
To become familiar with the default behavior of TCollection, try adding a TStatusBar component to a form, click the ellipses of the Panels property, press the Add button of the "Editing Panels". This last step adds a TStatusPanel to the editor. Click on on the TStatusPanel item in the editor and notice the change in the object inspector. Instead of seeing TStatusBar now you will see StatusBar1.Panels[0] reflected in the Object Inspector.

There are three major players involved with collections. A collection item (TCollectionItem) descendant, a TCollection that manages the list of TCollectionItems, and a component that contains the TCollection as one of it's properties. In our above example of TStatusBar, TStatusBar contains a descendant of TCollection called TPanels and TPanels manages a list of TCollectionItem descendants called TPanel. Notice that each TCollectionItem contains one or

more properties; for instance, TPanel contains Alignment, Bevel, Style, Text, and Width properties. This list changes depending on the definition of your TCollectionItem descendant.

### Creating a Minimal TCollection Implementation

-----

In a new unit you must first define three new descendant classes from TCollectionItem, TCollection and a TComponent.

```
TMyCollectionItem = class(TCollectionItem)
```

```
TMyCollection = class(TCollection)
```

```
TMyComponent = class(TComponent)
```

To make TMyCollectionItem functional, you need to define one or more properties to contain information to be tracked by the collection mechanism. The example defines a Text and a MoreStuff integer property. You will also need to override the GetDisplayName method to supply the string shown for each item in the collection property editor:

```
TMyCollectionItem = class(TCollectionItem)
private
  FText: string;
  FMoreStuff: LongInt;
  function GetDisplayName: string; override;
  procedure SetText(const Value: string);
  procedure SetMoreStuff(const Value: LongInt);
published
  property Text: string read FText write SetText;
  property MoreStuff: LongInt
    read FMoreStuff write SetMoreStuff;
end;
```

Next, define the TCollection descendant. This class will keep track of the component the collection belongs to, override the GetOwner method to accommodate streaming, and manage an array of the previously defined TCollectionItem descendants.

You will need to define a new static constructor. The parameter passed in this constructor is the reference to the component that contains the collection. Also in the constructor you need to populate the ItemClass property with the class of your TCollection item descendant. Note: ItemClass returns the class

(descended from TCollectionItem) to which the items in the collection belong.

```
TMyCollection = class(TCollection)
  private
    FMyComponent: TMyComponent;
    function GetItem(Index: Integer): TMyCollectionItem;
    procedure SetItem(Index: Integer; Value: TMyCollectionItem);
  protected
    function GetOwner: TPersistent; override;
  public
    constructor Create(MyComponent: TMyComponent);
    function Add: TMyCollectionItem;
    property Items[Index: Integer]: TMyCollectionItem
      read GetItem write SetItem; default;
end;
```

Finally, define the component that will contain the collection. The component will contain a property descended from the TCollection type defined previously. The TCollection property will need a private field, an access method to the private field, and storage allocated in the constructor and freed in the destructor.

Note: See The Developers Guide for more information on creating custom components.

```
TMyComponent = class(TComponent)
  private
    FItems: TMyCollection;
    procedure SetItems(Value: TMyCollection);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Items: TMyCollection
      read FItems write SetItems;
end;
```

Complete Unit Listing

```
-----
unit Collec1;

interface
```



```
// Note: TCollection and TCollectionItem are defined in Classes.Pas.
```

```
uses Classes;
```

```
type
```

```
  TMyComponent = class;
```

```
  TMyCollectionItem = class(TCollectionItem)
```

```
  private
```

```
    FText: string;
```

```
    FMoreStuff: LongInt;
```

```
    function GetDisplayName: string; override;
```

```
    procedure SetText(const Value: string);
```

```
    procedure SetMoreStuff(const Value: LongInt);
```

```
  public
```

```
  published
```

```
    property Text: string read FText write SetText;
```

```
    property MoreStuff: LongInt read FMoreStuff write SetMoreStuff;
```

```
  end;
```

```
  TMyCollection = class(TCollection)
```

```
  private
```

```
    FMyComponent: TMyComponent;
```

```
    function GetItem(Index: Integer): TMyCollectionItem;
```

```
    procedure SetItem(Index: Integer; Value: TMyCollectionItem);
```

```
  protected
```

```
    function GetOwner: TPersistent; override;
```

```
  public
```

```
    constructor Create(MyComponent: TMyComponent);
```

```
    function Add: TMyCollectionItem;
```

```
    property Items[Index: Integer]: TMyCollectionItem
```

```
      read GetItem write SetItem; default;
```

```
  end;
```

```
  TMyComponent = class(TComponent)
```

```
  private
```

```
    FItems: TMyCollection;
```

```
    procedure SetItems(Value: TMyCollection);
```

```
  public
```

```
    constructor Create(AOwner: TComponent); override;
```

```
    destructor Destroy; override;
```

```
  published
```

```
    property Items: TMyCollection read FItems write SetItems;
```

```

end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Sample', [TMyComponent]);
end;

{ TMyCollectionItem }

// Note: Inherited default behavior of GetDisplayName is to
// return the classname.

function TMyCollectionItem.GetDisplayName: string;
begin
  Result := Text;
  if Result = '' then Result := inherited GetDisplayName;
end;

procedure TMyCollectionItem.SetText(const Value: string);
begin
  if FText <> Value then
    FText := Value;
end;

procedure TMyCollectionItem.SetMoreStuff(const Value: LongInt);
begin
  if FMoreStuff <> Value then
    FMoreStuff := Value;
end;

{ TMyCollection }

constructor TMyCollection.Create(MyComponent: TMyComponent);
begin
  inherited Create(TMyCollectionItem);
  FMyComponent := MyComponent;
end;

```

```

function TMyCollection.Add: TMyCollectionItem;
begin
    Result := TMyCollectionItem(inherited Add);
end;

function TMyCollection.GetItem(Index: Integer): TMyCollectionItem;
begin
    Result := TMyCollectionItem(inherited GetItem(Index));
end;

procedure TMyCollection.SetItem(Index: Integer;
    Value: TMyCollectionItem);
begin
    inherited SetItem(Index, Value);
end;

// Note: You must override GetOwner in Delphi 3.x to get
// correct streaming behavior.
function TMyCollection.GetOwner: TPersistent;
begin
    Result := FMyComponent;
end;

{ TMyComponent }

constructor TMyComponent.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    FItems := TMyCollection.Create(Self);
end;

destructor TMyComponent.Destroy;
begin
    FItems.Free;
    inherited Destroy;
end;

procedure TMyComponent.SetItems(Value: TMyCollection);
begin
    FItems.Assign(Value);
end;

end.
{-----}

```

## Notes

-----

In this minimal example we didn't override the Assign method for the TCollectionItem, but this method should have further support. Here's an example of how you might implement Assign in the above project:

```
procedure TMyCollectionItem.Assign(Source: TPersistent);
begin
  if Source is TMyCollectionItem then
  begin
    Text := TMyCollectionItem(Source).Text;
    MoreStuff := TMyCollectionItem(Source).MoreStuff;
    Exit;
  end;
  inherited Assign(Source);
end;
```

Also not included in the above project is the logic needed to notify the TCollection class when one of it's contained items has changed. This could be particularly important in a visual control such as TStatusBar. TCollection supplies a virtual Update method for handling this behavior. See TStatusBar or THeaderControl in \source\vcl\commctrl.pas for further examples.

Reference:

4/2/99 3:00:45 PM

Last Modified: 01-SEP-99

## Article #16652: Delphi Titles from Leading Book Publishers

Technical Information Database

TI1652D.txt - Delphi Titles from Leading Book Publishers

Category :Delphi 1.x  
Platform :All  
Product :

Description:

BORLAND'S DELPHI IS  
SUPPORTED BY THESE TITLES  
FROM LEADING BOOK PUBLISHERS  
April 1996

Delphi Version 2.0

-----

The following is a list of book titles for Delphi 2.0 that are currently available or will be available in the future. For more information, contact the publisher.

- \* Delphi 2 Developer's Guide; X. Pacheco/S. Teixeira; Borland Press/Sams Publishing; ISBN: 0-672-30914-9  
Phone: 800-428-5331; Fax: 800-448-3804
- \* Borland's Official No-Nonsense Guide to Delphi 2; M. Manning; Borland Press/Sams Publishing; ISBN: 0-672-30871-1;  
Phone: 800-428-5331; Fax: 800-448-3804
- \* Teach Yourself Delphi 2 in 21 Days; D. Osier; Borland Press/Sams Publishing; ISBN: 0-672-30863-0;  
Phone: 800-428-5331; Fax: 800-448-3804
- \* Database Developer's Guide with Delphi 2; K. Henderson; Borland Press/Sams Publishing; ISBN: 0-672-30862-2;  
Phone: 800-428-5331; Fax: 800-448-3804
- \* Delphi 2 Unleashed, Second Edition; C. Calvert; Borland Press/Sams Publishing; ISBN: 0-672-30858-4  
Phone: 800-428-5331; Fax: 800-448-3804

The New Delphi 2 Programming Explorer; J. Duntemann/J. Mischel/D. Taylor; Coriolis Group; ISBN: 1-883577-72-1;  
Phone: 800-410-0192; Fax: 602-483-0193

Delphi 2 Nuts and Bolts, 2nd Edition; G. Cornell;

Osborne/McGraw-Hill; ISBN: 0-07-88203-3

Phone: 800-227-0900; Fax: 510-549-6603

Special Edition Using Delphi, 2nd Edition; J. Matcho; Que;

ISBN: 0-7897-0591-5; Phone: 800-428-5331; Fax: 800-882-8583

Delphi 2 By Example, 2nd Edition; B. Watson; Que;

ISBN: 0-7897-0592-3; Phone: 800-428-5331; Fax: 800-882-8583

Building Delphi 2.0 Database Applications; P. Kimmel; Que; ISBN:

0-7897-0492-7; Phone: 800-428-5331; Fax: 800-882-8583

Mastering Delphi for Windows 95; M. Cantu; Sybex;

ISBN: 0-7821-1860-7; Phone: 880-227-2346; Fax: 510-523-2373

Revolutionary Guide to Delphi 2 Programming; B. Long/B. Swart/et

al; Wrox Press Ltd.; ISBN: 1-874416-67-2;

Phone: 312-465-3559; Fax: 312-465-4063

\* Part of the Borland Press Series

Delphi Version 1.0

-----

\* Delphi Developer's Guide; X. Pacheco/S. Teixeira; Borland  
Press/Sams Publishing; ISBN: 0-672-30704-9

\* Teach Yourself Database Programming with Delphi in 21 Days;  
Nathan and Uri Gurewich; Borland Press/Sams Publishing;  
ISBN: 0-672-30851-7

Delphi Programming Unleashed; C. Calvert; Sams Publishing; ISBN:  
0-672-30499-6

The Delphi Programmer Explorer; J. Duntemann/J. Mischel/D.  
Taylor; Coriolis Group; ISBN: 1-883577-25-X

Delphi Programming for Dummies; N. Rubenking; IDG Books;  
ISBN: 1-56884-200-7

Foundations of Delphi Programming; T. Swan; IDG Books; ISBN:  
1-56884-347-X

Delphi: A Developer's Guide; V. Kellen/B. Todd; MIS Press; ISBN:  
1-55851-455-4

Teach Yourself Delphi; D. Hall; MIS Press; ISBN: 1-55828-390-0

Delphi Nuts and Bolts; G. Cornell/T. Strain; Osborne/McGraw-Hill;  
ISBN: 0-07-882-136-3

Using Delphi, Spec. Ed; J. Matcho/M. Andrew/et al; Que; ISBN:  
1-56529-823-3

Mastering Delphi; M. Cantu; Sybex; ISBN: 0-7821-1739-2

Borland Delphi How-To; G. Frerking/W. Niddery/N. Wallace; Waite  
Group Press; ISBN: 1-57169-019-0

Developing Windows Applications Using Delphi; P. Penrod; John  
Wiley and Sons; ISBN: 0-471-11017-5

Delphi By Example; B. Watson; Que; ISBN: 1-56529-757-1

Teach Yourself Delphi in 21 Days; A. Wozniewicz; Sams Publishing;  
ISBN: 0-672-30470-8

Instant Delphi; D. Jewell; Wrox Press Ltd.; ISBN: 1-874416-57-5

Delphi for Windows Power Toolkit; H. Davis; Ventanna; ISBN:  
1-56604-292-S

\* Part of the Borland Press Series

Reference:

3/30/99 1:17:25 PM

Last Modified: 01-SEP-99

## Article #16657: Pascal Titles from Leading Publishers

BORLAND'S PASCAL AND TURBO PASCAL  
ARE SUPPORTED BY THESE TITLES  
FROM LEADING PUBLISHERS  
April 1996

Borland Pascal 7.0 with Objects  
-----

Borland Pascal from Square One; Jeff Duntemann; Bantam Electronic  
Publishing; ISBN 0-553-37243-2

Borland Pascal 7.0 Programming for Windows; Tom Swan;  
BantamElectronic Publishing; ISBN 0-553-37244-0

A Programmer's Guide to Turbo Vision; Ertl/Machholz/Golgath;  
Addison Wesley; ISBN 0-201-62401-X

Borland Pascal Problem Solver; Brian Long; Addison-Wesley;  
ISBN0-201-59383-1

Object-Oriented Programming in Pascal: A Graphical Approach;  
Conner/Niguidula/VanDam; Addison Wesley; ISBN 0-201-62883

Borland Pascal with Objects 7.0; Jose de Jesus/Abad Godog; M&T  
Books; ISBN 1-558-28247-5

Programming with Turbo Vision; Blake Watson; M&T Books

Borland Pascal Developer's Guide; Ed Mitchell/Namir Shammas; Que;  
ISBN 0-88022-862-8

Borland Pascal Programs for Scientists and Engineers Alan R.  
Miller; Sybex; ISBN 0-7821-1150-5

Programming in Borland Pascal; Scott Palmer; Sybex;  
ISBN 0-7821-1151-3

Borland Pascal 7 Insider; Paul Cilwa; John Wiley & Sons, Inc.  
ISBN 0-471-59894-1

Borland Pascal Step-By-Step; Keith Weiskamp; John Wiley & Sons;  
ISBN  
0-471-30429-8

Pascal and the Paradox Engine; Robert Signore/Michael  
Vernick/Michael O. Stegman; Windcrest/McGraw-Hill;  
ISBN 0-8306-4325-7

Object-Oriented Programming with Borland Pascal 7; Namir  
Shammas; Sams; ISBN: 0-672-3033-7

Turbo Pascal 7.0



-----  
Turbo Pascal 4th Edition; Elliott Koffman with Bruce Maxim;  
Addison-Wesley; ISBN 0-20-155811-4  
Turbo Pascal: An Introduction to the Art and Science of  
Programming 4th Edition; Walter Savitch; Benjamin/Cummings;  
ISBN 0-8053-0418-5  
Programming with Turbo Vision; Blake Watson; M&T Books  
Turbo Pascal 7 Complete Reference; Stephen K. O'Brien/Steve  
Nameroff; Osborne/McGraw-Hill; ISBN 0-07-881793-5  
Turbo Pascal 7 Handbook; Werner Feibel; Osborne/McGraw-Hill; ISBN  
0-07-881799-4  
Turbo Pascal Programming 101; Charles Calvert; Sams; ISBN  
0-672-30285-3  
Object Oriented Programming in Turbo Pascal 7.0; Namir Shamma;  
Sams; ISBN 0-672-30285-3  
Turbo Pascal Memory Management Techniques; Len Dorfman/Neuberger;  
Windcrest/McGraw-Hill; ISBN 0-8306-4059-2  
The Beginner's Guide to Turbo Pascal; Oleg Perminov; Wrox Press  
Ltd.; ISBN: 1-874416-30-3  
The Revolutionary Guide to Turbo Pascal; Yuri Borodich, et al;  
Wrox Press Ltd.; ISBN: 1-874416-11-7

#### Turbo Pascal for Windows

-----

Turbo Pascal for Windows 3.0 Programming; Tom Swan; Bantam  
Electronic Publishing; ISBN 0-553-35293-8  
Turbo Pascal for Windows; Gunther Faeber/Martin Pauly; Abacus  
Using Turbo Pascal for Windows; Paul Perry; Que;  
ISBN 0-88022-806-7  
Turbo Pascal for Windows Bible; Gary Entsminger; Howard W. Sams &  
Co.; ISBN 0-672-30212-8  
Programmer's Introduction to Turbo Pascal for Windows Scott D.  
Palmer; Sybex; ISBN 0-7821-1022-3  
Write TSRs Now with Borland's Turbo Assembler, Turbo C/C++ &  
Turbo Pascal; Jim Ras; Wordware Publishing, Inc.  
PC Magazine's Turbo Pascal for Windows Tips and Techniques; Neil  
Rubenking; Ziff-Davis; ISBN 1-56276-035-1

#### Turbo Pascal Version 6.0

-----

Turbo Pascal System Programming; Michael Tischer; Abacus; ISBN  
1-55755-124-3  
Turbo Vision Toolbox; Abacus

Programmer's Guide to Turbo Vision; Ertl/Machholz/Golgath;  
Addison-Wesley

Graphics Programming in Turbo Pascal 6; Ben Ezzell; Addison-  
Wesley; ISBN 0-201-58119-1

Turbo Pascal Problem Solving & Programming Design; Elliott  
Koffman; Addison-Wesley

Using Turbo Pascal 6.0; Ben Ezzell; Addison-Wesley ISBN;  
0-201-5774-7

Turbo Pascal: Introduction to Art Science of Programming; Walter  
J. Savitch; Benjamin/Cummings

Hands-On Turbo Pascal; Larry Goldstein; Brady Books;  
ISBN 0-13-629635-1

Effective Programming with Turbo Pascal 6.0; Seymour Pollock;  
Brooks-Cole Publishing

Programming Fundamentals Using Turbo Pascal; Thomas M. Boger;  
W.C. Brown

Data Structures Using Turbo Pascal; Thomas A. Boger; W.C. Brown

Turbo Pascal Third Edition; Nell Dale/Chip Weems; D.C. Heath

Problem Solving & Program Implementation Using Turbo Pascal:  
Versions 4.0-6.0; Rick Mercer; Franklin, Beedle & Associates

Turbo Pascal Theory and Practice of Good Programming; Gary W.  
Martin; HBJ (Saunders); ISBN 0-15-592375-7

Mastering Turbo Pascal 6.0 Third Edition; Tom Swan; Hayden/Howard  
W. Sams; ISBN 0-8104-6287-7

Introduction to Computer Science with Turbo Pascal; William I.  
Salmon; Irwin

A Guide to Programming in Turbo Pascal 2nd Edition (Versions 4.0  
through 6.0); Tim Corica/William Freitas/Bruce Presley;  
Lawrenceville Press

Advanced Graphics Programming in Turbo Pascal; Roger  
Stevens/Christopher Watkins; M&T Books; ISBN 1-55851-132-6

Clean Coding in Turbo Pascal 6 with Turbo Vision; Amrik Dhillon;  
M&T Books; ISBN 1-55851-228-4

Fractal Programming in Turbo Pascal; Roger T. Stevens; M&T Books;  
ISBN 1-55851-107-5

Turbo Pascal 6.0: The Nuts & Bolts of Program Construction; John  
DiElsi; McGraw-Hill; ISBN 0-07-557790-9

Oh! Pascal! Turbo Pascal 6.0; Doug Cooper; W.W. Norton & Company;  
ISBN 0-393-96077-3 (3.50"); ISBN 0-393-96249-0 (5.25")

Turbo Pascal 6 Disk Tutor 2nd Edition; Werner Feibel;  
Osborne/McGraw-Hill; ISBN 0-07-881738-2

Turbo Pascal 6: The Complete Reference; Stephen O'Brien;  
Osborne/McGraw-Hill; ISBN 0-07-881703-X

Using Turbo Pascal 4-6; Julien Hennefeld; PWS Kent

Object-Oriented Programming with Turbo Pascal and Object

Professional; Ken Knecht; Prentice Hall Professional  
Publishing; ISBN 0-13-630310-2

Turbo Pascal 6.0 By Example; Greg Perry; Que; ISBN 0-88022-908-X  
Using Turbo Pascal 6.0 2nd Edition; Michael Yester/Namir Shammass;  
Que; ISBN 0-88022-700-1

Turbo Pascal 6 Object-Oriented Programming; Namir Shammass; Howard  
W. Sams; ISBN 0-672-30221-7

Turbo Pascal 6.0 Programming for the PC; Douglas Hergert; Howard  
W. Sams & Co.; ISBN 0-672-22811-4

Object-Oriented Programming with Turbo Pascal; Keith Weiskamp;  
Scott Foresman Professional Books

Mastering Turbo Pascal 6.0; Scott Palmer; Sybex; ISBN  
0-89588-675-8

Building Turbo Pascal Libraries, Data Entry Tools; Jeremy G.  
Soybel; Tab Books; ISBN 0-8306-7734-8

Introduction to Programming Principles Using Turbo Pascal; R.  
Foley; Van Nostrand Reinhold

Turbo Pascal 6.0; Richard Mojena; Wadsworth

Turbo Pascal How To: The Definitive Object-Oriented Problem  
Solver; Gary Syck; The Waite Group; ISBN 1-878739-04-2

Data Structure in Engineering with Turbo Pascal (supplement); Bob  
Crawford; West Educational Publishing

Power Graphics Using Turbo Pascal; Keith Weiskamp/Loren Heiny;  
John Wiley & Sons, Inc.; ISBN 0-471-54736-0

Turbo Pascal Self-Teaching Guide; Keith Weiskamp; John Wiley &  
Sons, Inc.; ISBN 0-471-54492-2

Graphics Programming with Turbo Pascal;  
Tanik/Slepoway/Townsend/Cheng; Wordware Publishing, Inc.

Illustrated Turbo Pascal 6; Paul Schlieve; Wordware Publishing,  
Inc.; ISBN 1-55622-202-5

PC Magazine Turbo Pascal 6.0 Techniques and Utilities; Neil  
Rubenking; Ziff-Davis Books; ISBN 1-56276-010-6

Turbo Assembler/Turbo Debugger  
-----

Mastering Turbo Assembler, 2nd Edition; Tom Swan; Sams  
Publishing; ISBN: 0-672-30526-7

Using Turbo Debugger and Tools; Ben Ezzell; Addison-Wesley

The Waite Group's Turbo Assembler Bible; Gary Syck; Sams

Mastering Turbo Debugger 2.0; Tom Swan; Sams

Macro Magic with Turbo Assembler; Jim Mischel; John Wiley & Sons,  
Inc.

Illustrated Turbo Debugger 3.0; Hari Venkateswaran/Marat M.  
Tanik; Wordware Publishing, Inc.

Learn Turbo Assembler Programming in a Day; Stephen K.  
Cunningham; Wordware Publishing, Inc.

Write TSRs Now with Borland's Turbo Assembler, Turbo C/C++ and  
Turbo Pascal; Jim Ras; Wordware Publishing, Inc.

Last Modified: 01-SEP-99

## Article #17596: Calling a Delphi DLL from your BCB Code.

Technical Information Database

TI2596C.txt Calling a Delphi DLL from your BCB Code.

Category :General

Platform :All

Product :C++Builder All

### Description:

Calling a Delphi DLL from your BCB Code.

First create a simple DLL in Delphi:

```
{ Begin DLL code }
Library MinMaxDll;
uses Windows;
Function Min(X, Y: Integer): Integer; export;
begin
    if X < Y then Min := X else Min := Y;
end;
Function Max(X, Y: Integer): Integer; export;
begin
    if X > Y then Max := X else Max := Y;
end;
procedure Del_MessageBox;export;
begin
    MessageBeep(0);
    MessageBox(0, 'Delphi Message Box', 'MinMaxDll', MB_OK or
MB_TASKMODAL);
end;
Exports
    Min name '@Min',
    Max name '@Max',
    Del_MessageBox name '@Del_MessageBox';
begin
end.
{ End DLL code }
```

Then to Call it from your BCB Code:

1. Create and insert an import library first using implib:

```
implib MinMaxDLL.lib MinMaxDLL.dll
```

Then add the import library to your BCB project, Project|Add to Project.

2. In your BCB application, you must prototype the functions as:

```
extern "C" {
    int __fastcall Min(int, int);
    void __fastcall Del_MessageBox(void);
    int __fastcall Max(int, int);
}
```

3. Now you can call Min or Max anywhere in your application.

```
Min();
Del_MessageBox();
Max();
```

Important Note:

If you are exporting other than fastcall, please read the following. Make sure you are exporting the functions in the Export Section of your

Delphi dll, and prototype the functions in the format listed in the commented fields below.

Exports {Exports section in Delphi dll}

```
Min name '@Min', { BCB prototype: extern "C"{int __fastcall Min(int, int); }
```

```
Max name '_Max', { BCB prototype: extern "C"{int __cdecl Max(int, int); }
```

```
Del_Message;      { BCB prototype: extern "C"{void __stdcall
Del_Message(); }
```

Reference:

7/2/98 10:41:01 AM

Last Modified: 01-SEP-99

## Article #17624: Assembly linked with Pascal - can't trace through

Technical Notes Database

TN2624C.txt Assembly linked with Pascal - can't trace through

Category :OTHER

Platform :All

Product :TD 1.xx

Description:

QUESTION:

Why can't I trace through my external assembly language routines which have been linked with my TP program?

ANSWER:

The Pascal linker does not pull through the symbolic debug information which is necessary to do source level debugging. Therefore, you will not be able to see your TASM source code. However, if you pull up the CPU window, you can trace through the disassembled ASM code. You will not have the benefit of variable names and labels, but you will be able to see the flow of execution.

Reference:

7/2/98 10:43:01 AM

Last Modified: 01-SEP-99

## Article #17795: Removing the Vertical Scroll Bar from a TDBGrid

Technical Information Database

TI2795C.txt Removing the Vertical Scroll Bar from a TDBGrid

Category :Database Issues

Platform :Win95/NT

Product : C++Builder3.x

### Description:

In order to remove the vertical scrollbar from a TDBGrid component, you must override its Paint method. Inside the Paint method you must call the SetScrollRange API procedure to set the min and max scroll values to zero (this disables the scrollbar), and then call the inherited Paint. The code below is a unit containing a new component called TNoVertScrollDBGrid that does this. You can copy the code into a file called NoScroll.cpp, and add it to the component library as a custom component.

```
-----  
// -----  
// In the header file:  
// -----  
class TNoVertScrollDBGrid : public TDBGrid  
{  
protected: void __fastcall Paint();  
};  
  
// -----  
// In the .cpp file:  
// -----  
  
//-----  
  
void __fastcall TNoVertScrollDBGrid::Paint()  
{  
    SetScrollRange(Handle, SB_VERT, 0, 0, false);  
    TDBGrid::Paint();  
}  
  
//-----  
namespace NoScroll  
{  
    void __fastcall Register(TObject *Sender)  
    {  
        TComponentClass classes[1] = {__classid(TNoVertScrollDBGrid)};  
        RegisterComponents("Data Controls", classes, 0);  
    }  
}
```



```
}  
}
```

Reference:  
None

12/30/99

Last Modified: 01-SEP-99

## Article #17801: Writing a Buffer Disk

Technical Information Database

TI2801C.txt :Writing a Buffer Disk  
Category :Database Issues  
Platform :Win95/NT  
Product :

Description:

General:

=====

Changes made to a table are not written directly to disk until the table is closed. A power failure or system crash can result in a loss of data, and an inconvenience. To avoid this loss of data, two direct Database Engine calls can be made, both of which have the similar effects. These functions are `DbiUseIdleTime` and `DbiSaveChanges`.

`DbiSaveChanges(hDBICur):`

=====

`DbiSaveChanges` saves to disk all the updates that are in the buffer of the table associated with the cursor (`hDBICur`). It can be called at any point. For example, one may want to make save changes to disk every time a record is updated:

```
void __fastcall TForm1::Table1AfterPost(TDataSet *DataSet)
{
    DbiSaveChanges(Table1->Handle);
}
```

This way, one does not have to worry about losing data if a power failure or system crash occurs after a record update.

`DbiSaveChanges` can also be used to make a temporary table (created by `DbiCreateTempTable`) permanent.

This function does NOT apply to SQL tables.

DbiUseIdleTime:

=====

DbiUseIdleTime can be called when the "Windows Message Queue" is empty. It allows the Database Engine to save "dirty buffers" to disk. In other words, it does what DbiSaveChanges does, but performs the operation on ALL the tables that have been updated. This operation however, will not necessarily occur after every record update, because it can only be executed when there is an idle period.

In C++ Builder, it can be used in this fashion:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Application->OnIdle = UseIdle;
}
```

```
void __fastcall TForm1::UseIdle(TObject *Sender, bool &Done)
{
    DbiUseIdleTime;
}
```

USAGE NOTES:

=====

Using both DbiUseIdleTime and DbiSaveChanges (after every record modification) is redundant and will result in unnecessary function calls. If the application is one that performs a great deal of record insertions or modifications in a small period of time, it is recommended that the client either call DbiUseIdleTime during an idle period, or call DbiSaveChanges after a group of updates.

If not very many updates are being performed on the table, the client may choose to call DbiSaveChanges after every post or set up a timer and call DbiUseIdleTime when a timer event is generated.

Reference:

None

2/16/99 6:39:15 PM

Last Modified: 01-SEP-99

## Article #17805: TI2805C: Showing Deleted Records in a dBase Table

Technical Information Database

TI2805C.txt :Showing Deleted Records in a dBase Table  
Category :Database Issues  
Platform :Win95/NT  
Product :All-CBuilder,

### Description:

Q: How can I view dBASE records marked for deletion. That is, I want to view those records marked as "soft deletion"?

A: In a dBASE table, records are not removed from the table until the table is packed. Until that happens, records that are "deleted" are actually just marked as "to be" deleted. To show these existing but not displayed records, the following function, ShowDeleted(), makes use of a BDE API function, DbiSetProp(), to show records marked for deletion. It is not necessary to close and re-open the table when using this function. ShowDeleted() takes a TTable and a boolean variable as parameters. The boolean parameter determines whether or not to show deleted records.

```
//In your header file:
```

```
//-----
```

```
private://private declarations  
void ShowDeleted(TTable*, bool);
```

```
//In your .cpp file:
```

```
//-----
```

```
//-----
```

```
void TForm2::ShowDeleted(TTable *Table1, bool ShowDeleted)  
{  
    DBIResult rslt;  
    DBIMSG szErrMsg;  
  
    Table1->DisableControls();  
    try {  
        Check(DbiSetProp((hDBIObj)(Table1->Handle),  
                        curSOFTDELETEON, int(ShowDeleted)));  
    }  
    __finally {  
        Table1->EnableControls();  
    }  
}
```

```
}
Table1->Refresh();
}

void __fastcall TForm2::CheckBox1Click(TObject *Sender)
{
    ShowDeleted(Table1, CheckBox1->Checked);
}
//-----
```

Reference:

None

2/16/99 6:44:55 PM

Last Modified: 27-SEP-99

## Article #18045: Searching on a secondary index in Engine 2.0

Technical Notes Database

TN3045C.txt Searching on a secondary index in Engine 2.0

Category :OTHER

Platform :All

Product :BDE 2.0

### Description:

Searching on a secondary index in Engine 2.0

When you open a table on index 0 (primary index), and you perform a search

on a field which possesses a secondary index, that secondary index WILL be

used. However, there are some tricky problems you may run into.

Although searching using SEARCHFIRST or SEARCHNEXT with PXSrchFld will search based upon the ordering of the indexed field, end of table will be

determined based upon the primary index (ie., physical ordering).

Therefore, it is safe to do a SEARCHFIRST on this field but a SEARCHNEXT may

return PXERR\_ENDOFTABLE even if all records matching the search criterion

have not been found.

What is the solution, then, if you are using SEARCHNEXT on a field search?

Open the table on the secondary index. This means that if you are doing

some of your searches on the primary index and others on an indexed field

you will want to open the same table twice - once on the primary index and

once on the secondary index.

### Reference:

7/2/98 10:43:13 AM

Last Modified: 01-SEP-99



## Article #18107: FAQ document 5

Technical Notes Database

TN3107C.txt      FAQ document 5

Category        :Win32 API

Platform        :All

Product         :BCW 4.00-

Description:

BCPPWIN SECTION 11 - Win32 API

-----  
This document contains information that is most often provided to users

of this section. There is a listing (when available) of common Technical

Information Documents and example files that can be downloaded from the libraries. There is also a listing of the most frequently asked questions and their answers.

TECHNICAL INFORMATION DOCUMENTS AND EXAMPLE FILES

-----  
Currently no listing available.

FREQUENTLY ASKED QUESTIONS AND ANSWERS

-----  
Q:    How do I create a Win32 application in the IDE?

A:    Via the Speed Menu, go to Target Expert. Under Target Type, set Platform to Win32.

Q:    What are the command-line tools for creating Win32 applications?

A:    BCC32, TLINK32, and BRC32 are the primary Win32 tools. You can use implib, impdef, and tdump can be used for both Win32 and Win16.

Q:    Why does the IDE not work in NT?

A:    The Windows 3.1 emulation provided by NT does not fully support all the features of the IDE. The Borland development team is working vigorously to resolve this problem. The Borland command-line tools as well as TD32 work under NT.

Q:    Why do I get the error "Cannot run multiple instances of a DLL under WIN32s"?

A:    Due to stability issues, Borland dll's do not support multiple instances under Win32s. This is not an issue

under Win32, i.e. NT.

Q: My stack is not expanding past 1Mb. I have set my stack size much larger in my .def file. What is wrong?

A: It is necessary to turn on the compiler option Check Stack Overflow to enable stack expansion.

Q: Can I utilize my 16bit dll in a Win32 app?

A: Yes, if you are creating an application for Win32s. See the section under Universal Thunk in the Windows help file win32s.hlp. Windows NT does not support mixing of 16bit and 32bit code in the same process.

Q: After creating a program using stdin or stdout (i.e. printf, cout, etc. ) I try to run it under Win32s. Nothing appears. What's wrong?

A: There are two possible problems. First, if you are trying to create a console application, Win32s does not support console apps. Second, if you are creating a GUI application, Borland does not supply an EasyWin for Win32. Check the forum libraries for third-party implementations of EasyWin for Win32.

Reference:

7/2/98 10:43:18 AM

Last Modified: 01-SEP-99

## Article #18289: Comparing connections to an AS/400 between Delphi/400 and ODBC

Technical Information Database

TI3289D.txt - Comparing connections to an AS/400 between Delphi/400 and ODBC

Category :AS400  
Platform :All-32Bit  
Product :DelphiAS400,

### Description:

This document describes how Delphi/400 compares with ODBC in regards to connections to an AS/400.

### Reference:

Reading records

For sequential access a TQuery with ODBC is faster than Delphi/400. But if you compare a TTable Delphi/400 is faster. If you want to improve performance with a TTable, TQuery or TFile400, you need to play with the "buffer count". Buffer count is the number of data base records Delphi/400 sends in only one communication record. The optimal value of "buffer count" may be determined by using the following formula:

$$\text{"Buffer count"} * \text{Record length} < 32 \text{ Kbytes}$$

It is best if this result can (or nearly) be divisible by the max frame length of your communication card; parameter MAXFRAME of the LIND of your AS/400 line description.

For Direct or Keyed access, Delphi/400 is much faster than ODBC. On large tables, executing Query1.First and then executing Query1.Last is significantly slower with ODBC. Similarly, scrolling a DBGrid to the bottom

is significantly slower with ODBC.

Below are examples how to code "buffer count" programmatically. In all cases buffercount is an integer and should be set after the Object (Database, Table, File) is opened:

Example 1

```
query1.DatabaseName := 'idco400';  
dbisetprop(hDBIObj(databasel.handle), $404000E, buffercount);
```

Example 2

```
table1.Open;  
dbisetprop(hDBIObj(table1.handle), $4050009, buffercount);
```

Example 3

```
file4001.Open;  
file4001.BufferCount := buffercount;
```

## 2. INSERTing records via a TTable

Our tests show executing the following code is approximately 6 times faster when using Delphi/400 vs. ODBC.

Var

```
  i : integer;  
  Starttime : DWORD;
```

Begin

```
  Starttime := gettickcount( );  
  databasel.connected :=true;  
  table1.active:=true;  
  for i=0 to 100 do  
  begin  
    table1.insert;  
    table1.fieldbyname('FLD1').asstring := 'my string';  
    .....  
  
    Table1.post;  
  end;  
  Table1.active:=false;  
  Databasel.connected:=false;  
  edit1.text :=floattostr( (gettickcount() - Starttime ) / 1000);
```

end;

3. Running AS/400 commands in Delphi application.

- With Delphi/400, you can run an AS/400 command and wait for return parameters. Using ODBC you must use Stored Procedures which do not have as much flexibility due to all data types not being supported.

Many thanks to Van Vinh Pham from TCIS whom is the source of technical information in this document.

4/1/99 4:29:32 PM

Last Modified: 01-SEP-99

## Article #19020: TURBO PROFILER SOURCE CODE VIEWING

Technical Notes Database

TN4020C.txt TURBO PROFILER SOURCE CODE VIEWING

Category :Pascal

Platform :All

Product :TPRF 1.0

### Description:

Q. When I load a program in the Profiler I don't see my source code?

A. Your program does not have the necessary debugging information.

Rebuild your program with {\$D+,L+} and stand-alone debugging ON.

### Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99

## Article #19021: TP 5.0/TASM - ARG DIRECTIVE SYMBOLIC REFERENCES

Technical Notes Database

TN4021C.txt TP 5.0/TASM - ARG DIRECTIVE SYMBOLIC REFERENCES

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. I am using the Arg directive to create symbolic references to variables on the stack. I am not getting the correct values I should in the Assembler routine. What am I doing wrong?

A. It is easier if remember a few points about the Arg directive. First, the Arg directive always adjusts for a near or far call. If you have a proc definition like:

```
myproc proc far
```

the Arg directive will take into account the four (4) byte return address on the stack. If your definition looks like:

```
myproc proc near
```

the Arg directive will adjust for a two (2) byte return address on the stack. The second thing to remember is that the Arg directive always assumes that you will push BP onto the stack first and thus, will add two bytes to the size of the return address. For a far return the total will be six bytes and for a near return the total will be four bytes. The third point to remember about the Arg directive itself is that the leftmost symbol in the Arg list will be nearest the top of the stack. Finally, remember that for each subsequent symbol to the right, a number equal to the number of bytes on the stack occupied by the symbols to the left will be added to the BP register.

Consider the following examples:

```
myproc proc near
  arg first:word, second:word
  push bp
  mov  bp,sp
```

The distance is near, a two byte return address, and the BP is assumed to pushed onto the stack. Thus, any reference to first is equivalent to [BP + 04H]. Since first is a word it occupies two bytes on the stack. Thus second evaluates to [BP + 04H + 02H] or [BP + 06H].

```
myproc proc far
arg first:word, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 06H]. Since first is a word, it occupies two bytes on the stack. Thus, second evaluates to [BP + 06H + 02H] or [BP + 08H].

```
myproc proc far
arg first:near ptr word, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus any reference to first is equivalent to [BP + 06H]. Since first is a near pointer it occupies two bytes on the stack. Thus second evaluates to [BP + 06H + 02H] or [BP + 08H].

```
myproc proc far
arg first:byte, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 06H]. Although first is a byte value it occupies two bytes on the stack. Thus, second evaluates to [BP + 06H + 02H] or [BP + 08H].

```
myproc proc far
arg first:byte:1, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 06H]. Although first is a byte value it occupies two bytes on the stack. To force it to use only one byte we must use the :1 after the word byte. Second evaluates to {BP + 06H + 01H} or [BP + 07H]. Note that the type of variables must match the types of those that appear in the high level language call.

Reference:

7/2/98 10:45:32 AM



Last Modified: 01-SEP-99

## Article #19024: TASM - UNDEFINED SYMBOL WITH TURBO C

Technical Notes Database

TN4024C.txt TASM - UNDEFINED SYMBOL WITH TURBO C

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

- Q. When I try to link a Turbo C module with a TASM generated .OBJ file, the linker reports "undefined symbols".
- A. Assemble your TASM file with the /ML option. Turbo C symbols are by default, case sensitive. Turbo Assembler on the other hand, converts all symbols to uppercase by default. Therefore, make sure that the symbols case is exactly the same in the two modules, and assemble with the /ML option.

### Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99

## Article #19025: TASM - LOAD AND EXECUTE OVERLAY

Technical Notes Database

TN4025C.txt TASM - LOAD AND EXECUTE OVERLAY

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

- Q. Why does Turbo Debugger not step through overlaid portions of a program?
- A. Overlays are generally accomplished through DOS interrupt 21, function 4BH. Pressing F7 while on any interrupt in Turbo Debugger causes the computer to execute everything included in the interrupt before returning control to TD. Because the overlay is executed inside of an interrupt, TD is not given control until the overlay has finished executing, and the INT 21H is returned from. The workaround is to do an instruction trace into the interrupt. You will have to trace through the DOS code which loads the overlay and transfers control to it. Also, since you are doing CPU tracing, you will not have the benefit of source level debugging.

### Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99

**Article #19026: TD - AT&T 6300 clock speeds up after TD is loaded.**

Technical Notes Database

TN4026C.txt TD - AT&T 6300 clock speeds up after TD is loaded.

Category :Pascal

Platform :All

Product :TD 1.x 2.0

Description:

Q. I have an AT&T 6300 and after loading TD my clock speeds up.

A. Run TDINST and set the NMI option to NO. Make sure to save the configuration both to TD.EXE, and to a config file. Erase all other TDCONFIG.TD files in all directories before doing this.

Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99

## Article #19027: TD - 8087 MATH COPROCESSOR NOT RECOGNIZED

Technical Notes Database

TN4027C.txt TD - 8087 MATH COPROCESSOR NOT RECOGNIZED

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. Why does TD not recognize the presence of my 8087 in the 8087 window.

A. Check to make sure that the coprocessor dipswitch is set properly on the motherboard, or that the system CMOS ram setup indicates that a 8087 is present.

### Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99

## Article #19028: TD - EMS VERSIONS TD

Technical Notes Database

TN4028C.txt TD - EMS VERSIONS TD

Category :Pascal

Platform :All

Product :TD 1.x 2.0

Description:

Q. What types of EMS board/drivers work with TD?

A. The EMS board and driver must both support at least the EMS version 3.2 specification.

Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99

## Article #19029: TA - FAR JUMP OR FAR CALL TO AN ADDRESS

Technical Notes Database

TN4029C.txt TA - FAR JUMP OR FAR CALL TO AN ADDRESS

Category :Pascal

Platform :All

Product :TD 1.x 2.0

Description:

Q. How can I JMP or CALL and address that is not a defined procedure or label?

A. You have to define a label as DD (double). The following code shows how.

```
.DATA
My_Struc STRUC          ; Make a structure to contain the
    OFFSETVALUE  DW      ; segment and offset values.
    SEGMENTVALUE DW      ; Remember reverse order addressing
My_Struc ENDS
SmallUnion UNION        ; Make a union - the DPTR is so that
    DPTR         DD      ; TASM will make a far call. The
    ADDR         My_Struc ; ADDR is to simplify putting your
SmallUnion ENDS        ; segment-offset address in.
JumpLocation SmallUnion ; Declare the union to be used
.CODE
...
...
MOV  JumpLocation.ADDR.OFFSETVALUE , 06000H ; Use your offset
MOV  JumpLocation.ADDR.SEGMENTVALUE , 0F000H ; Use your segment
CALL JumpLocation.DPTR
...
```

The structure within a union allows easy loading of the values. An alternative, shorter method would be to use:

```
.DATA
FarLabel DD
.CODE
...
MOV  WORD PTR [FarLabel] , 06000H
MOV  WORD PTR [FarLabel+2] , 0F000H
CALL FarLabel
```

ALSO, in IDEAL mode, you can make an absolute jump like this:  
ideal

```
codeseg
start:
    ...
    jmp     far 7c00h:5
    ...
end start
```

Reference:

7/2/98 10:45:32 AM

Last Modified: 01-SEP-99



## Article #19030: TASM - EQU in masm mode vs. in ideal mode

Technical Notes Database

TN4030C.txt TASM - EQU in masm mode vs. in ideal mode

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. When I use EQU and the location counter \$ to get the length in bytes of a data item in ideal mode the assembler always returns a "Can't subtract dissimilar relative quantities" error message. The manual states that this error message is generated when an expression subtracts two addresses that cannot be subtracted from each other. This could happen if the

addresses

are in two different segments. It works fine in masm mode.

My code looks like this:

```
IDEAL
MODEL SMALL
DATASEG
    firstbyte      db      1
    secondbyte     db      2
    thirdbyte      db      3
    ;The following generates an error!
    totalbytes     EQU     $ - firstbyte
MASM
    ;In masm mode it works!
    thisworks      EQU     $ - firstbyte
IDEAL
CODESEG
start:
    mov ax, @data
    mov ds, ax
    ;The error is generated here!
    mov al, totalbytes
    mov al, thisworks
    mov ax, 4c00h
    int 21h
    end start
```

A. In masm mode an EQU can be treated as a string or a number. In

IDEAL mode an EQU is always treated as a text substitution. Thus, in masm mode the number \$ - firstbyte is substituted for this works when the assembler first encounters the expression. In IDEAL mode the string \$ - firstbyte is substituted when the symbol totalbytes appears in the instruction mov al, totalbytes. The variable totalbytes is in the segment defined by dataseg, but the location counter \$ refers to an offset in the segment defined by codeseg, the current segment when the text substitution takes place. This causes an error to be generated. Always remember that in ideal mode text equates with EQU take place when the constant name appears. This is referred to as late binding. If you must create a numeric equate use the = operator.

Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19031: TLINK or TC

Technical Notes Database

TN4031C.txt TLINK or TC  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

- Q. The linker returns a fixxup overflow message. I am trying to reference an external symbol.
- A. Make sure that your external symbols are declared outside of any segment.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19032: TA - PROGRAM WILL ASSEMBLE UNDER MASM & NOT TASM

Technical Notes Database

TN4032C.txt TA - PROGRAM WILL ASSEMBLE UNDER MASM & NOT TASM

Category :Pascal

Platform :All

Product :TD 1.x 2.0

Description:

Q. I have a program that assembles fine under masm but will not assemble under tasm.

A. Insert keyword MASM51 and the keyword QUIRKS at the top of your source file and try again.

Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19033: TP 5.0/TD - ORDER OF ARG DIRECTIVE OPPERANDS

Technical Notes Database

TN4033C.txt TP 5.0/TD - ORDER OF ARG DIRECTIVE OPPERANDS

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. In what order should the operands to the Arg directive appear? Should they be the same as the order in the calling procedure or in reverse?

A. The answer depends upon the language that you're using and whether or not you're using the .model memorymodel, language option.

Case 1: You are using the full segment definitions or you are using the .model directive without specifying a language.

The leftmost symbol in the Arg list is always nearest the top of the stack (TOS). Thus, if your proc definition is

```
myproc proc far
    arg first:word, second:word
```

first will evaluate to [BP + 06H], the size of the four byte far return address plus the size of the two byte BP register pushed onto the stack. The symbol, second, will evaluate to [BP + 08H], two bytes further from TOS.

Again, the key point to remember is that the leftmost symbol is nearest TOS.

If you are using Pascal or Basic, arguments are pushed from left to right onto the stack. This is called Pascal calling syntax or Pascal calling conventions. In this case, the last argument in the list is nearest TOS. For example:

```
{ $F+ }
procedure myproc(one:integer; two:integer); external;
{ $F- }
.
.
.
procedure myproc(first:integer; second:integer);
.
.
```

The variable, first, is pushed onto the stack followed by second. Thus, second is nearest TOS. In this case, you would want to set up your Arg list as follows:

```
myproc proc far
  arg second:word, first:word
  push bp
  mov  bp,sp
```

The symbol, second, would evaluate to [BP + 06H], the symbol nearest TOS, and first would evaluate to [BP + 08H].

To get the correct results with Basic or Pascal while using the simplified segment directives or .model without the language option, you must reverse the order of the arguments as they appear in the high level language calling routine's parameter list.

If you are calling the Assembler routine from a C module and you are NOT using Pascal calling conventions, then the arguments are pushed onto the stack from right to left. Thus, the leftmost argument is nearest TOS and there is no need to reverse the order to the symbols in the Arg list.

If you wish to use Pascal calling conventions in Turbo C, use the -p option for TCC or the Pascal keyword. Pascal calling syntax can also be enabled from within TC by setting the /Options/Compiler/Code generation/Calling convention to Pascal.

Case 2: You are using the .model memorymodel, language directive.

All you need to do is put the Arg list symbols in the same order as they appear in the calling routine's parameter list. This is the easiest method, by far. Remember to include the comma after the memory model in the .model directive and to specify the language. If you are using Turbo Pascal, just use .model TPascal.

Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19034: TD METAWARE HIGH C TDCONVRT

Technical Notes Database

TN4034C.txt TD METAWARE HIGH C TDCONVRT  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

- Q. How come my Metaware High C program does not show up in the module window? I've TDCONVRT'ed it.
- A. Metaware does not operate exactly like Microsoft C. Basically, leading underbars are not placed on variable names. When TD looks for the symbol "\_main", it can't find it, so it comes up in the CPU window.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19035: TP 5.0/TASM - RETURN FAILED CALLING ASSEMBLY ROUT

Technical Notes Database

TN4035C.txt TP 5.0/TASM - RETURN FAILED CALLING ASSEMBLY ROUT  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

- Q. When I call my Assembler routine from Turbo Pascal, everything works OK. However, it does not seem to return properly. Why?
- A. Any routine declared in the main program or in the implementation section of a unit, is by default a near call. Your Assembly language routine, assembled to an .OBJ file, is brought into the Pascal module as if it were declared and defined in that module. Thus, by default, any call to that Assembly language routine is a near call. If you have defined your Assembly language routine as a proc far, the Assembler will generate a corresponding RETF. Thus, a near call is made, 4 bytes are popped off of the stack when the RETF is executed, and control will be returned to the wrong code segment. If you want that call to be a far call in your Pascal module, be sure to bracket the function or procedure declaration with {\$F+} and {\$F-}. Always make sure that the distance declarations and definitions agree.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99



## Article #19036: TA - SIZE OF STRUC IN IDEAL MODE

Technical Notes Database

TN4036C.txt TA - SIZE OF STRUC IN IDEAL MODE

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. In masm mode I can get the size of a struc without actually defining a variable of that type. Why do I get zero when I attempt the same thing in ideal mode?

A. In IDEAL mode, SIZE returns the total number of bytes allocated for the data item. If no variable of that type has been defined then no storage is allocated. In other words, use it on an instance of a structure, rather than on the tag name.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19037: TA 1.0 - TED.ASM; FORWARD SEGMENT REFERENCE PROBL

Technical Notes Database

TN4037C.txt    TA 1.0    - TED.ASM; FORWARD SEGMENT REFERENCE PROBL  
Category     :Pascal  
Platform     :All  
Product      :TD  1.x 2.0

### Description:

- Q.    I downloaded a file called Ted.asm that assembles with MASM but not with TASM.
- A.    The problem is with a forward segment reference.  Simply insert the following lines of code at the beginning of your program.
- ```
CSEG SEGMENT
CSEG ENDS
FILE_SEG SEGMENT
FILE_SEG ENDS
```
- Alternatively, use TASM 1.01 or later.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19038: TA - HOW TO WRITE A TSR

Technical Notes Database

TN4038C.txt TA - HOW TO WRITE A TSR

Category :Pascal

Platform :All

Product :TD 1.x 2.0

Description:

Q. How do you write a TSR?

A. See the file Tesserract on our CompuServe forum at BPROGB.

This file is on the CLMFOR and is supported there.

Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19039: TA - TYPE in IDEAL MODE

Technical Notes Database

TN4039C.txt TA - TYPE in IDEAL MODE

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. Why does TYPE in IDEAL mode not behave as documented in the TASM reference manual?

A. In IDEAL mode TYPE acts as a type cast and can be used in an expression with PTR to force the operand to have a specified type. For example, consider the following:

```
ideal
```

```
bvarbyte db ?
```

```
bvarword dw ?
```

```
mov al, [TYPE bvarbyte PTR bvarword]
```

```
; the above expression is the same as
```

```
mov al, [BYTE PTR bvarword]
```

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19040: TA - CREATING A .COM FILE TLINK /V

Technical Notes Database

TN4040C.txt TA - CREATING A .COM FILE TLINK /V

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. How do I create a .COM file from my .ASM file? Whenever I try to create the .COM with TLINK /t I always get the message invalid entry point. I can create the .COM file with EXE2BIN.

A. In order to create a .COM file with TLINK /t you must specify an entry point at offset 100h in the file with both the ORG and END directives. The following code fragment demonstrates the required conditions. Also, a .COM file may not exceed 64K in size, may not have any segment-relative fixups, may not define a stack segment, and must have a starting address equal to 0:100h. When an extension other than .COM is used for the executable file (.BIN, for example), the starting address may be either 0:0 or 0:100h.

```
.model tiny
```

```
.code
```

```
ORG 100H
```

```
start:          ;this will be the entry point at 100H
```

```
.
```

```
.
```

```
.
```

```
END start      ;specify the entry point with END
```

```
or
```

```
mycode SEGMENT 'CODE'
```

```
    assume cs:mycode,ds:mycode,ss:mycode
```

```
    ORG 100H
```

```
    start:
```

```
    .
```

```
    .
```

```
    .
```

```
mycode ends
```

```
    end start
```

Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19041: TA - DEBUG .COM FILE

Technical Notes Database

TN4041C.txt TA - DEBUG .COM FILE

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. How do you debug a .COM file? I can debug my program as an exe file but not as a com file.

A. Due to the structure of a .COM file it cannot contain debug information. When you specify a file to be debugged the Turbo Debugger first looks for an exe file and then for a .COM file. The debugger will look for a symbol table in a file of the same name with a .TDS extension. You can create such a file with the utility TDSTRIP. By using the -c (create .COM file) and -s(put symbol table in .TDS file) options you can create a com file and symbol table file in one step.

The line

```
TDSTRIP -c -s exefilename
```

creates exefilename.com and exefilename.tds. Note that the -c and -s must be separated by a space.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19042: TA - NONFUNCTIONAL CONDITIONAL BREAKPOINT

Technical Notes Database

TN4042C.txt TA - NONFUNCTIONAL CONDITIONAL BREAKPOINT  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

- Q. How do I get my conditional breakpoint to work? I know that the condition is true but it still will not break.
- A. If when using the conditional breakpoints and expression true condition the break does not occur, check to be sure that the condition is really true by evaluating the expression in the evaluate window. For example, the following will not break correctly.

```
.data
  nptr dw ?
.code
start:
    mov ax,@data
    mov ds,ax
    mov [nptr], 0fffH
    mov dx,5           ;set a conditional break here
;break if the expression ds:[WORD PTR nptr] EQ 0fffh
;this break will fail
;if you evaluate this expression in the evaluate
;window it will evaluate to false. This particular
;expression requires parenthesis around the ds:expression
;to bind the segment override to the memory referencing
;expression
;The correct syntax is (ds:[WORD PTR nptr]) EQ 0fffh
The key here is to use the evaluate window to see how the
debugger is parsing the expression.
```

### Reference:





## Article #19043: TD - LOCATION POINTER SOURCE CODE

Technical Notes Database

TN4043C.txt TD - LOCATION POINTER SOURCE CODE

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

- Q. Why does the location pointer appear to be pointing to the wrong location in the source file?
- A. If your source code has been modified since the last compile the debugger may be indicating the wrong source location. Observe the status line for the module viewer. If the word modified appears you need to recompile your source code. It is a good idea to use a makefile and to periodically run make to be sure your program is up to date. See the documentation in the Turbo Assembler Reference Guide for the make utility.

### Reference:

7/2/98 10:45:33 AM

Last Modified: 01-SEP-99

## Article #19044: TD - NOT ENOUGH ENVIRONMENT SPACE

Technical Notes Database

TN4044C.txt TD - NOT ENOUGH ENVIRONMENT SPACE  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

- Q. Why when loading TD386 do I get a "not enough environment space" message?
- A. You need to allocate more environment space in your config.sys file with the -eXXXX option of TDH386.sys.  
device = tdh386.sys -e2000  
The above line allocates 2000 bytes of environment strings.  
The default is 1024.

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19045: TD - HANGING ON NMI

Technical Notes Database

TN4045C.txt    TD - HANGING ON NMI  
Category    :Pascal  
Platform    :All  
Product     :TD 1.x 2.0

### Description:

- Q. Why does the debugger give me an initial startup message, and then hang? Sometimes I get a message concerning the NMI.
- A. By default the debugger intercepts the NMI in order to allow one to break into the debugger with a breakout switch. When the breakout switch is activated an NMI is generated and the debugger regains control. If your hardware is using the NMI then there will be a compatibility problem as the debugger does not chain on the NMI. Run Tdinst and set NMI Intercept to NO in the Options menu. Be sure to save the changes to your TD.EXE by selecting SAVE/MODIFY TD.EXE. You must then erase any tdconfig.td files as they will have the NMI Intercept set to YES and will override the setting in TD.EXE.

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19046: TD - HANGING ON IBM PC

Technical Notes Database

TN4046C.txt    TD   - HANGING ON IBM PC  
Category     :Pascal  
Platform     :All  
Product      :TD  1.x 2.0

### Description:

- Q. Why does the debugger hang when loading on my old IBM PC, XT, or compatible?
- A. Check to make sure that the dipswitch that corresponds to the math coprocessor is correctly set. If it is not, the debugger will try to execute a floating point instruction and will hang while loading. Try flipping dipswitch number 1 (when numbered starting from 0) on the motherboard. If you have the setting for the coprocessor in CMOS RAM run the setup program to see if it indicates the presence of a npx. If so, change the switch setting. Alternatively, upgrade to Turbo Debugger 1.5.

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19047: TD - 43/50 LINE MODE VIDEO 7 VEGA DELUX

Technical Notes Database

TN4047C.txt    TD   - 43/50 LINE MODE VIDEO 7 VEGA DELUX  
Category     :Pascal  
Platform     :All  
Product      :TD  1.x 2.0

### Description:

- Q.  When I use TD in 43/50 line mode with my video 7 vega deluxe the bottom line contains garbage.
- A.  Use the /Q option in your config.sys as follows:  
      device = vegabios /Q /m(monitor type 1,2,3,or 4)

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19048: TASM , TLINK & use with EPROMS & intel hex files

Technical Notes Database

TN4048C.txt    TASM , TLINK & use with EPROMS & intel hex files  
Category     :Pascal  
Platform     :All  
Product      :TD 1.x 2.0

### Description:

Q. How can I download the output of the assembler to an EPROM?  
Can I get an Intel hex file from the assembler and linker?  
A. You will not be able to output an intel hex file from TASM  
and TLINK. Nor will you be able to download the .exe file  
to an eprom. The .exe file contains fixxups that must be  
resolved be the DOS loader. To download the .exe file to  
and EPROM burner you must first process the .exe file with  
a locater. Here is the address and phone number of one  
company that provides a locater.

Paradigm Systems  
PO Box 152  
Milford, MA 01757  
(800) 537-5043

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19049: TD - ROMABLE CODE

Technical Notes Database

TN4049C.txt    TD   - ROMABLE CODE  
Category     :Pascal  
Platform     :All  
Product      :TD  1.x 2.0

### Description:

Q.  Can I use TDremote to debug ROMable code?  Can I use TDremote to debug an application running on a non-DOS platform?  Can I use TDremote to debug my embedded controller?

A.     Contact:  
  
          Paradigm Systems Inc.  
          (800)537-5043

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99



## Article #19050: TD - TDREMOTE TIMEOUT

Technical Notes Database

TN4050C.txt TD - TDREMOTE TIMEOUT

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. Why do I get a timeout when running tdremote?

A. If your application runs too long without tdremote sending any information to the debugger, TD will issue a timeout message. Try reducing the BAUD rate.

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19051: TD - EXIT QUIT NULL POINTER ASSIGNMENT

Technical Notes Database

TN4051C.txt TD - EXIT QUIT NULL POINTER ASSIGNMENT

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. Sometimes when I quit the debugger while running tdremote I get a null pointer assignment error message.

A. This message is coming from within TD and can be safely ignored.

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19052: TD - CPU VIEWER

Technical Notes Database

TN4052C.txt TD - CPU VIEWER

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. Why does the debugger always come up in the CPU viewer?

A. Debugger always comes up in the CPU viewer.

This can occur in several different situations.

1)

The user may be able to open up the module viewer and view the source code but is unable to make the debugger come up viewing the source.

If the module viewer can be opened with no error message then problem is merely an incorrect path in the config file, an incorrect source directory saved to td.exe or tdconfig.td with tdinst, a /l (do not execute startup code when loading) command line option inadvertently saved to the config file when the user saved the options from within td, or the Display/Beginning Display option was set to Assembler from within tdinst and either saved to tdconfig.td or to TD.EXE.

2)

If there is an error such as

"Source file Source.ext not found. Press ESC."

when trying to open the module viewer.

In this case there are two possible causes, either the file does not exist or the Options/Path For Source is set incorrectly. Have the user set the path from within the debugger and then open the module viewer. If that works then they need to create a tdconfig.td or some other config file by either saving the current options or by running TDINST. It makes sense to have a tdconfig.td in each working directory with a correct path for those applications.

3) The user has saved the current options to a configuration file while the CPU window is open.

Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19053: TD - HARDWARE DEBUGGERS

Technical Notes Database

TN4053C.txt TD - HARDWARE DEBUGGERS  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

Q. Are there any hardware debuggers available for td?

A. Yes, below is a list of hardware debuggers that support td.

Trapper Board

Purart Inc

113 Drinkwater Road

PO Box 189

Hampton Falls, New Hampshire 03844

Tel 603 772-9907

Fax 603 772-2902

contact Elaine Pura or J.Scott Benson

mci: Elaine Pura or JS Benson

price: \$199

### Reference:

7/2/98 10:45:34 AM

Last Modified: 01-SEP-99

## Article #19054: TD - TD386 CANT LOAD VIRTUAL MODE

Technical Notes Database

TN4054C.txt TD - TD386 CANT LOAD VIRTUAL MODE

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

- Q. Why do I get a message that I can't load td386 because the processor is already in virtual mode?
- A. There cannot be a program running in either virtual or protected mode when td386 is loaded. It has to be the only program running in virtual mode. There is probably a memory manager that runs in virtual or protected mode in the system. Examples of this type of program include, but are not limited to, 386^MAX, CEMM.SYS, and any of a number of drivers that allow, among other things, extended memory to emulate expanded memory. TD386 has a built in ems simulator that will allow td use a subset of the ems calls for its symbol table. This simulator is for td only. If the application must make ems calls, the user must have an ems driver that does not run in virtual or protected mode.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19055: TD - TD and HP Vectra, unexpected interrupt 06

Technical Notes Database

TN4055C.txt    TD - TD and HP Vectra, unexpected interrupt 06  
Category     :Pascal  
Platform     :All  
Product      :TD 1.x 2.0

### Description:

- Q.    When loading TD on an HP Vectra an unexpected interrupt 06 is generated.
- A.    We have contacted HP regarding this matter and they have offered a solution. Part of the HP/HIL code uses an instruction that is illegal in virtual 8086 mode. To word around this, run the HP Setup program. At the first menu do not choose any of the selections. Instead type "YADA" . You should now receive an 8 option screen. Option 2 hp/hil defaults to on if this is set to off the unexpected interrupts 06 will go away.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19056: TD - TD386 unexpected interrupt 06

Technical Notes Database

TN4056C.txt TD - TD386 unexpected interrupt 06

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. After running the program for awhile, an unexpected interrupt 06 occurs.

A. Interrupt 06 is caused in virtual mode when the CPU encounters and illegal instruction. Most probably some portion of the code has been overwritten by an errant pointer, an array bounds error, or a stack overflow. If you can locate the instruction that

TD386

is trying to execute, you can set a changed memory breakpoint to find

where in your program the code is being overwritten.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99



## Article #19057: TD - UNEXPECTED INTERRUPT 0D

Technical Notes Database

TN4057C.txt TD - UNEXPECTED INTERRUPT 0D

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. What is unexpected interrupt 0D?

A. TD386, unexpected interrupt 0d

Interrupt 0d is the general protection exception and is generated by any protection violation that does not generate some other exception. See the above question for a more complete description of the problem. Common causes of this problem are network boards and certain hard disk controllers.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19058: TD - TD386 UNEXPECTED INTERRUPT 0A

Technical Notes Database

TN4058C.txt TD - TD386 UNEXPECTED INTERRUPT 0A

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. Why do I get unexpected interrupt 0A as soon as I call TD386?

A. This indicates that there is a hardware or device driver incompatibility. Interrupt 0A is being generated as a result of this hardware/software. In the virtual 8086 mode of the 80386,

Interrupt 0A is generated by the CPU when an invalid task state segment is encountered. TD386 is unable to determine whether the CPU generated the error, or if it was generated by the hardware. Therefore, to prevent making the situation worse, TD386

terminates

itself, and prints the "UNEXPECTED INTERRUPT" message.

If you have a Microsoft Bus Mouse make sure that you have set the jumpers correctly. The jumper should be set to number 5 for an AT type machine, including any 80386 machine with an AT bus. If the jumper is incorrectly set you will receive this error message. Try changing the jumpers until the error goes away.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19059: TD - ILLEGAL EXPRESSION HEX NUMBERS

Technical Notes Database

TN4059C.txt TD - ILLEGAL EXPRESSION HEX NUMBERS

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

- Q. Why does the debugger give me an error when I enter a hex number in one of the prompt boxes?
- A. If a user reports a message like "Illegal expression", "illegal syntax", "extra input after expression ", it could be because they are trying to enter a number that ends with a "b" or "d" like 0ff0b or 0AED. If a number ending in a "b" or "d" is entered at any prompt in the debugger, it is assumed that the user is trying to override the default radix or base. A number ending in b or d is assumed to be a binary or decimal number respectively. The user need only append a "H" to the end of the number, as in 0ff0bH or 0AEDH, if they are using assembler syntax or source language when debugging an assembler program. If a 'C' program is being debugged, it is important to prefix all hex numbers with the '0x' prefix. In general, when a message of this type is reported, one should examine the current syntax in the OPTIONS/LANGUAGE menu selection and make sure that the syntax being used corresponds to the language in the menu selection.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19060: TLINK - TLINK /V

Technical Notes Database

TN4060C.txt TLINK - TLINK /V  
Category :Pascal  
Platform :All  
Product :TD 1.x 2.0

### Description:

Q. Why does TLINK not accept the /v option?

A. TLINK and /v option

If tlink does not accept /v option it could be because an old version of tlink is in the path before the new version.

For example, the new version could be in the TD directory and the old version in the TC directory.

```
PATH=c:\TC;c:\debug\TD
```

The following command would use the old version of TLINK.

```
C:\TC\WORK>tlink /v myprog
```

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19061: TD - NULL POINTER ASSIGNMENT.

Technical Notes Database

TN4061C.txt TD - NULL POINTER ASSIGNMENT.

Category :Pascal

Platform :All

Product :TD 1.x 2.0

### Description:

Q. How do I find a null pointer assignment in my Turbo C program?

A. Open a watch on

(char \* ) 0,s

(char \*) 4,s

This is the location of the Borland message. It is also the area that will be overwritten if there is a null pointer assignment. The user will be able to watch the Borland message while stepping through the code. When the null pointer assignment occurs the message in the watch window will change. The same thing can be done by opening a dump viewer and doing a CTRL-G (GOTO) to 0000.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19062: TASM 1.0 - ARG DIRECTIVE SYMBOLIC REFERENCES - THO

Technical Notes Database

TN4062C.txt    TASM 1.0 - ARG DIRECTIVE SYMBOLIC REFERENCES - THO  
Category     :Pascal  
Platform     :All  
Product      :TD 1.x 2.0

### Description:

Q. I am using the Arg directive to create symbolic references to variables on the stack. I am not getting the correct values I should in the Assembler routine. What am I doing wrong?

A. It is easier if remember a few points about the Arg directive. First, the Arg directive always adjusts for a near or far call. If you have a proc definition like:

```
myproc proc far
```

the Arg directive will take into account the four (4) byte return address on the stack. If your definition looks like:

```
myproc proc near
```

the Arg directive will adjust for a two (2) byte return address on the stack. The second thing to remember is that the Arg directive always assumes that you will push BP onto the stack first and thus, will add two bytes to the size of the return address. For a far return the total will be six bytes and for a near return the total will be four bytes. The third point to remember about the Arg directive itself is that the leftmost symbol in the Arg list will be nearest the top of the stack. Finally, remember that for each subsequent symbol to the right, a number equal to the number of bytes on the stack occupied by the symbols to the left will be added to the BP register.

Consider the following examples:

```
myproc proc near  
arg first:word, second:word  
push bp  
mov bp,sp
```

The distance is near, a two byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 04H]. Since first is a word it occupies two bytes on the stack. Thus second evaluates to [BP + 04H + 02H] or [BP + 06H].

```
myproc proc far
arg first:word, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 06H]. Since first is a word, it occupies two bytes on the stack. Thus, second evaluates to [BP + 06H + 02H] or [BP + 08H].

```
myproc proc far
arg first:near ptr word, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus any reference to first is equivalent to [BP + 06H]. Since first is a near pointer it occupies two bytes on the stack. Thus second evaluates to [BP + 06H + 02H] or [BP + 08H].

```
myproc proc far
arg first:byte, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 06H]. Although first is a byte value it occupies two bytes on the stack. Thus, second evaluates to [BP + 06H + 02H] or [BP + 08H].

```
myproc proc far
arg first:byte:1, second:word
push bp
mov bp,sp
```

The distance is far, a four byte return address, and the BP is assumed to be pushed onto the stack. Thus, any reference to first is equivalent to [BP + 06H]. Although first is a byte value it occupies two bytes on the stack. To force it to use only one byte we must use the :1 after the word byte. Second evaluates to {BP + 06H + 01H} or [BP + 07H]. Note that the type of variables must match the types of those that appear in the high level language call.

Reference:

Last Modified: 01-SEP-99



## Article #19063: TD386 and DOS 5.0 Compatibility

Technical Notes Database

TN4063C.txt TD386 and DOS 5.0 Compatibility

Category :Pascal

Platform :All

Product :TD 2.01

### Description:

TD386 ( & TF386 ) will hang on startup if you use DOS 5.0, and have the command:

```
DOS=HIGH
```

in the CONFIG.SYS file.

This does not happen if DOS is not loaded HIGH.

The machine is hanging inside of the DOS program loader code.

When doing the equivalent to DOS=HIGH with DR-DOS 5.0, this problem does not occur.

All of the "new" commands to load device drivers & TSR's high in DOS 5.0 are incompatible with TD386, as they use V8086 mode ( ala QEMM & 386MAX). Note that loading device drivers & TSR's high is different from DOS=HIGH , which loads the DOS kernel into the XMS high memory area, and which does not use V8086 mode.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19064: IMPORT.PAS name fields empty

Technical Notes Database

TN4064C.txt    IMPORT.PAS name fields empty  
Category     :Pascal  
Platform     :All  
Product      :BDE 2.0

### Description:

After Initializing the fields in the InitFields function in import.pas, we forget to return True (OK). This results in empty fields. To fix, add the line

```
InitFields := True;
```

to the end of the function.

IMPORT.PAS is an example program for the TP for DOS PX Engine.

### Reference:

7/2/98 10:45:35 AM

Last Modified: 01-SEP-99

## Article #19301: How to do a "sounds like" query in Interbase

Technical Information Database

TI4301C.txt :How to do a "sounds like" query in Interbase

Category :Database Issues

Platform :Win95/NT

Product :All-CBuilder,

### Description:

Imagine you could run a query on an Interbase table that would return a set of values that `_sound_` like your query. It is possible through the magic of UDFs (user defined functions) and an algorithm called SoundEx. Using a very simple algorithm in a DLL, you can declare a function that assigns a numeric code based on the way a word "sounds". This code can be compared against your query string to return an accurate (based on the simplicity of the algorithm) set.

----How it works----

SoundEx classifies words for how they sound by assigning a short numeric code based on a group of letters.

Take the first letter.

Then, for each of the subsequent letters, score a code for the group it belongs to. There are 6 groups :

Group	Letters
1	BFPV
2	CGJKQSXZ
3	DT
4	L
5	MN
6	R

Ignore all other letters and characters (vowels, h, punctuation, etc.) If a code has already occurred, ignore it. Continue until you have 4 characters,

if there are less, add zeroes until you have 4.

Thus

Smith becomes S5300

Smitt becomes S5300

Smythe becomes S5300

And they all sound the same or similar.

-----Implementation-----

In order to get Interbase to use the Soundex function we must do the following:

Write the function in C++ or Delphi.

Encapsulate the function in a DLL.

Declare the function and DLL to interbase as a UDF.

Create a column on our table to keep track of the SoundEx codes.

Add a trigger to update the data.

The Delphi function was originally written by John Midwinter and was implemented to

return not the SoundEx 4 character code, but a number in a Byte (SmallInt).

He did this by realising that, though the natural Soundex Code has 4 characters,

it only has  $26 * 6 * 6 * 6$  (=5,616) possible values which can be held by a small int.

And therefore chose to store it as a number rather than a character string. Below

I have provided both Delphi and C++Builder source code.

-----C++ Function-----

```
extern "C"
```

```
int __declspec(dllexport) SoundBts(char *Name)
```

```
{
```

```
    unsigned char SoundExTable[59] =
```

```
    //A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ / ] ^ _ ' ,
```

```
    {0,1,2,3,0,1,2,0,0,2,2,4,5,5,0,1,2,6,2,3,0,1,0,2,0,2,0,0,0,0,0,0,
```

```
    //a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
    0,1,2,3,0,1,2,0,0,2,2,4,5,5,0,1,2,6,2,3,0,1,0,2,0,2};
```

```
    //This table is used to keep track of what number goes with what
```

```
letter.
```

```
    Byte i,s,SO,x;
```

```
    int Multiple, Result;
```

```
    int len;
```

```
    if (Name != "")
```

```
    {
```

```

Result = (short)UpCase(Name[0]); //initialize to first character
SO = 0; //initialize last char to 0
Multiple = 26; //26 letters in the alphabet
len = (StrLen(Name)); //length of word
for (i=0; i 64) && (s < 123)) //if it is a letter
    {
        x = SoundExTable[((int)s - 65)]; //get the SoundEx code
        if ((x > 0) && (x != SO))
            {
                Result += (x * Multiple);
                if (Multiple == (26 * 6 * 6 * 6))
                    break; //we have a sufficient code
                Multiple *= 6;
                SO = x;
            }
    }
}
else return 0;
return Result; //Finished SoundEx code
}

```

-----  
-----Delphi Function by John Midwinter-----  
library SoundBytes;

```

uses
    SysUtils;

function SoundBts(Name : PChar) : SmallInt; cdecl;
Const
{This table gives the SoundEX SCORE for all characters Upper and Lower
Case
hence no need to convert. This is faster than doing an UpCase on the
whole input string
The 5 NON Chars in middle are just given 0}

SoundExTable : Array[65..122] Of Byte
//A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ / ] ^ _ '
=(0,1,2,3,0,1,2,0,0,2,2,4,5,5,0,1,2,6,2,3,0,1,0,2,0,2,0,0,0,0,0,0,
//a b c d e f g h i j k l m n o p q r s t u v w x y z
    0,1,2,3,0,1,2,0,0,2,2,4,5,5,0,1,2,6,2,3,0,1,0,2,0,2);

Var
    i, l, s, SO, x : Byte;
    Multiple : Word;

begin

```

```

If Name > '' //do nothing if nothing is
passed
then begin
    Result := Ord(UpCase(Name[0])); //initialise to first character
    SO := 0; //initialise last char as 0
    Multiple := 26; //initialise to 26 char of
alphabet
    l := Pred(StrLen(Name)); //get into var to save
repeating function
    For i := 1 to l do //for each char of input str
    begin
        s := Ord(name[i]); // *
        If (s > 64) and (s < 123) //see notes * below
        then begin
            x := SoundExTable[s]; //get soundex value
            If (x > 0) //it is a scoring char
            AND (x <> SO) //is different from previous
char
            then begin
                Result := Result + (x * Multiple); //avoid use of POW as it
needs maths unit
                If (Multiple = 26 * 6 * 6) //we have done enough (NB
compiles to a const
                then break; //We have done, so leave loop
                Multiple := Multiple * 6;
                SO := x; //save for next round
            end; // of if a scoring char
        end; //of if in range of SoundEx
table
    end; //of for loop
    end else result := 0;
end; //of function SoundBts

exports
    SoundBts;
begin
end.

```

-----

Putting the function into Interbase.

For this example, I have included an SQL script. The table we are updating is called People.

-----soundex.sql-----

```
CONNECT MyDatabase.gdb USER SYSDBA PASSWORD mypass ;
```

```
DECLARE EXTERNAL FUNCTION SoundBytes Char(30)
```

```

RETURNS INTEGER BY VALUE           //function declaration
ENTRY_POINT "SoundBts"             //function name (DLL entry point)
MODULE_NAME "soundex.dll" ;        //name and path of the dll

CREATE DOMAIN SoundBytesDomain      //here we create a domain for the
type                                //that the code column will hold
INTEGER                             //that the code column will hold
DEFAULT 0
NOT NULL;

ALTER TABLE PEOPLE                //add a column to hold the SoundEx
codes
ADD SOUNDBYTES INTEGER NOT NULL;

UPDATE PEOPLE                      //get the SoundEx codes into
SOUNDBYTES                          //from the NAME column
SET SOUNDBYTES = SoundBytes(NAME);

CREATE INDEX SoundByteIndex        //put an index on the table
ON PEOPLE (SOUNDBYTES);

SET TERM !! ;                      //create a trigger to update the
table with
CREATE TRIGGER SoundBytes_Update   //new soundex codes
BEFORE UPDATE AS
BEGIN
    NEW.SOUNDBYTES = SoundBytes(NEW.NAME);
END !!
SET TERM ; !!

EXIT ;

```

-----Using the function in your application-----  
There are several ways to utilize this function. You would use it in pseudo-SQL like this:

```
select * from MY_TABLE where MY_FIELD sounds like MY_QUERY
```

One of the ways is to have the server execute the SoundBts function using a select

statement:

```
select NAME from PEOPLE where
    SOUNDBYTES = SoundBytes("John")
```

There can be problems with this method, however - it can GPF because of a DLL path discrepancy.

Another way is to generate the SoundEx code within your app by importing the function from the DLL. This is quicker and you simply have to pass the number, rather than executing the UDF on the server.

The code might look like this:

```
//after importing the function, use it
SoundExCode = SoundBts("MySearchString");

//using a TQuery connected to a TDatabase attached to the Interbase
database:
//the SQL property of the query looks like this:
Select Name from People where
  soundbytes = :SXCode

//then all you need to do is this:
```

```
Query1->Prepare();
Query1->ParamByName("SXCode")->AsInteger = SoundExCode;
Query1->Open();
```

-----

This is the magic of SoundEx and Interbase UDFs. This is very simple, and the function can be fine-tuned to get closer matches (say, if your table had a whole lot of similar-sounding names).

-----FIN-----

Reference:  
None

2/23/99 1:59:49 PM

Last Modified: 01-SEP-99



## Article #19335: How do I create a simple MIDAS Client Server application?

Technical Information Database

TI4335C.txt - How do I create a simple MIDAS Client Server application?

Category :Database Issues  
Platform :Win95/NT  
Product :C++Builder3.x,

### Description:

How do I create a simple MIDAS Client Server application?

### Reference:

Creating the Server

1. File | New Application
2. File | New | Remote Data Module
3. Type in the name. Mine is "ExampleDataModule".
4. Put down whatever you want to send to the client as data. You can use TQuery or TTable. I'm going to use TTable exporting one of the example databases we ship with C++Builder3.  
Properties set are:  

```
Table1->DatabaseName = BCDEMOS  
Table1->TableName = animals.dbf
```
5. Right click on Table1, choose "Export Table1 from Data Module". (If you go to View | Type library, you will see that Table1 is exported as an IProvider.)
6. Now build and run the server. After running it you can close it down. It just needed to get registered. (If you put the server on a remote system then it will have to be registered with regsvr32.exe on both systems)

## Creating the Client

1. Go to Project | Project Manager and click the New button. Choose new Application.
2. Drop down a TMIDASConnection. (If you wanted to use DCOM and run the server remotely then you would have to set the computer name that the server is registered and running on.)
3. Set the property (Notice that the server has been registered.):

```
MIDASConnection->ServerName = Server.ExampleDataModule
```

4. Drop down a TClientDataSet on your form. Set the properties:

```
ClientDataSet1->RemoteServer = MIDASConnection;  
ClientDataSet1->ProviderName = Table1;
```

(Notice that Table1 has been exported) Set Active to true:

```
ClientDataSet1->Active = true;
```

5. Now do normal database stuff... I put down a TDataSource and a TDBGrid to view the data.

3/16/99 8:02:56 AM

Last Modified: 01-SEP-99

## Article #19523: Working with cookies

Technical Information Database

TI4523D.txt - Working with cookies

Category :Internet/WEB  
Platform :All-32Bit  
Product :All32Bit,

### Description:

The purpose of this document is not to explain what a cookie is or what it is used for. Instead this document assumes that you are familiar with the use of cookies and want to know how to work with them when developing web server applications with Delphi 3.

The TWebRequest and TWebResponse objects provided in Delphi 3 both have properties which allow for the easy use of cookies. TWebRequest has the Cookie and CookieFields properties to allow a web server application to read the cookie header sent as part of a HTTP request. The TWebResponse object has the Cookies property which allows a web server application to place a cookie on the client machine through the cookie header of the HTTP response. This is usually set through the use of the SetCookieField method.

When a server responds to a HTTP request it sends a document with a header and a content section. Delphi provides the ability to add a cookie header through the TWebResponse.Cookies property. This is best set through the use of the SetCookieField method. The following TWebActionItem demonstrates the use of the SetCookieField method to return a cookie to the requesting browser. This example uses localhost as the domain. You would use this for testing and replace this string with your domain name. The third parameter is a slash. This means that this cookie will be sent along with all requests from the browser while at this domain. For further explanations of these parameters see the aforementioned RFC 2109.

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  MyCookies: TStringList;
begin
  MyCookies := TStringList.Create;
  with MyCookies do begin
    Add('Name=Frank Borland');
    Add('Address=100 Borland Way');
    Add('Product=Delphi');
  end;
  with Response do begin
    SetCookieField(MyCookies, 'localhost', '/', (Now + 1), False);
    Content := 'Cookie planted';
  end;
end;

```

When a request is made of the HTTP server the client browser issuing the request will package up all of the applicable cookie name value pairs and include them in the HTTP requests cookie header. Delphi 3 will make these available to a web server application in two ways. The first way is as a string through the Cookie property of the TWebRequest parameter of a TWebActionItem. It is also available as a TStringList property called CookieFields. CookieFields is the parsed contents of the Cookie header of an HTTP request message.

The following are TWebActionItems which extract the name value pairs and return them to the client in the form of a bare bones HTML page. The first example returns the cookie as a single string, while the second example returns each name value pair on a separate line.

```

procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '' + Request.Cookie +
  '
'; end; procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean); var i: integer; begin Response.
Content := ''; for i := 0 to (Request.CookieFields.Count - 1) do begin Response.Content := Response.
Content + '

```

' + Request.CookieFields[i] + '

'; end; Response.Content := Response.Content + "; end; //end TI Reference: None 4/1/99 11:45:44 AM

Last Modified: 01-SEP-99

## Article #19524: Handling Winsock errors

Technical Information Database

TI4524D.txt - Handling Winsock errors

Category :Windows API  
Platform :All-32Bit  
Product :All32Bit,

### Description:

For any of the following exception handling methods to work, the VCL must in some way become aware that an error condition exists. If a call to the Winsock does not return, or does not provide information to the TCustomWinSocket descendant that called it, then there is no mechanism to handle the condition. OnError exception handler

One method for trapping exception conditions in a descendant of TCustomWinSocket is to use an OnError exception handler. This method will only handle a limited set of conditions because the mechanism provided by the Winsock for event notification only reacts to a limited list of conditions. To be notified of an exception condition within the Winsock, TCustomWinSocket registers user message CM\_SocketMessage to be sent to the component, and the CMSocketMessage message handler raises an exception. The message is registered with the Winsock by an API call to WSASyncSelect. WSASyncSelect is a request for event notification of socket read, writes, connect, close, and accept events. If the exception condition is not read, write, connect, close or accept, or if the CM\_SocketMessage is not sent by the Winsock for any reason, the error handler will not fire.

### Usage:

```
procedure TChatForm.ClientSocketError(Sender: TObject;  
    Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;  
    var ErrorCode: Integer);  
const  
    ErrorEvents: array[eeGeneral..eeAccept] of string = (  
        'eeGeneral',  
        'eeSend',
```

```

    'eeReceive',
    'eeConnect',
    'eeDisconnect',
    'eeAccept'
);
begin
ListBox1.Items.Add('ClientSocketError.   TErrorEvent: ' +
    ErrorEvents[ErrorEvent] + '   ErrorCode: ' + IntToStr(ErrorCode));
    ErrorCode := 0; // don't raise an exception
end;

```

Definition:

```

procedure TCustomWinSocket.CMSocketMessage(var Message:
TCMSocketMessage);

function CheckError: Boolean;
var
    ErrorEvent: TErrorEvent;
    ErrorCode: Integer;
begin
    if Message.SelectError <> 0 then
    begin
        Result := False;
        ErrorCode := Message.SelectError;
        case Message.SelectEvent of
            FD_CONNECT: ErrorEvent := eeConnect;
            FD_CLOSE: ErrorEvent := eeDisconnect;
            FD_READ: ErrorEvent := eeReceive;
            FD_WRITE: ErrorEvent := eeSend;
            FD_ACCEPT: ErrorEvent := eeAccept;
        else
            ErrorEvent := eeGeneral;
        end;
        Error(Self, ErrorEvent, ErrorCode);
        if ErrorCode <> 0 then
            raise ESocketError.CreateFmt(sASyncSocketError, [ErrorCode]);
        end else Result := True;
    end;
end;

begin
with Message do
    if CheckError then
        case SelectEvent of

```

```

    FD_CONNECT: Connect(Socket);
    FD_CLOSE: Disconnect(Socket);
    FD_READ: Read(Socket);
    FD_WRITE: Write(Socket);
    FD_ACCEPT: Accept(Socket);
end;

```

```
end;
```

## Object Pascal Exception Handling

You can also wrap a specific call in a try..except block or setting an application level exception handler. For this to work, the component must in some way become aware of the exception condition and an exception must be raised for the exception to be trapped here.

### Example of Application Exception Handler:

```

TChatForm = class(TForm)
.
.
    public
        procedure AppException(Sender: TObject; E: Exception);
    end;
.
.
implementation
.
.
    procedure TChatForm.AppException(Sender: TObject; E: Exception);
    begin
        ListBox1.Items.Add('AppException: ' + E.Message);
    end;

    procedure TChatForm.FormCreate(Sender: TObject);
    begin
        Application.OnException := AppException;
    end;

```

### Example of Try..Except block:

```

with ClientSocket do
    try
        Active := True;

```



```

except
  on E: Exception do
    ListBox1.Items.Add('Try..except during open: ' + E.Message);
  end;
end;

```

### SocketErrorProc

For calls that use the CheckSocketResult function to check the result returned by WSAGetLastError, errors can be handled in a programmer defined function by setting the SocketErrorProc.

### Usage:

#### Interface

```

.
.
procedure MySocketError(ErrorCode: Integer);

```

#### implementation

```

.
.
procedure MySocketError(ErrorCode: Integer);
begin
  ShowMessage('MySocketError: ' + IntToStr(ErrorCode));
end;

```

```

procedure TChatForm.FormCreate(Sender: TObject);
begin
  SocketErrorProc := MySocketError;
end;

```

### Defined:

```

function CheckSocketResult(ResultCode: Integer; const Op: string):
  Integer;
begin
  if ResultCode <> 0 then
  begin
    Result := WSAGetLastError;
    if Result <> WSAEWOULDBLOCK then
      if Assigned(SocketErrorProc) then
        SocketErrorProc(Result)
      else raise ESocketError.CreateFmt(sWindowsSocketError,
        [SysErrorMessage(Result), Result, Op]);
  end;
end;

```

```
end else Result := 0;  
end;
```

Help Text for SocketErrorProc:

Unit ScktComp

SocketErrorProc handles error messages that are received from a Windows socket connection.

```
threadvar SocketErrorProc: procedure (ErrorCode: Integer);
```

Assign a value to SocketErrorProc to handle error messages from Windows socket API calls before the socket component raises an exception. Setting SocketErrorProc prevents the socket component from raising an exception. SocketErrorProc is a thread-local variable. It only handles errors that arise from the Windows socket API calls made within a single execution thread.

Reference:

None

4/22/99 12:31:30 PM

Last Modified: 01-SEP-99

## Article #19528: Converting a BMP to a JPEG and vice-versa under Delphi 3

Technical Information Database

TI4528D.txt - Converting a BMP to a JPEG and vice-versa under Delphi 3

Category :Delphi 3.x  
Platform :All-32Bit  
Product :Delphi3.x,

Description:

Q: How do I convert a BMP to a JPEG and vice versa?

A: This little project demonstrates how to do it using the JPEG components that ship with Delphi 3 (in the INFO\EXTRAS\JPEG directory). Just start a brand new project in Delphi 3. Drop two TButtons on the form. Then double click on the first button. Then double click on the second button. (This hooks the event handlers up). Finally, replace the code inbetween "{\$R \*.DFM}" and "end." in Unit1 with this code:

```
uses
    JPEG;

procedure TForm1.Button1Click(Sender: TObject);
var
    MyJPEG : TJPEGImage;
    MyBMP   : TBitmap;
begin
    { Convert a BMP to a JPEG }

    MyBMP := TBitmap.Create;
    with MyBMP do
        try
            LoadFromFile('YourBmpHere.BMP');
            MyJPEG := TJPEGImage.Create;
            with MyJPEG do begin
                Assign(MyBMP);
```

```
        SaveToFile('YourJpegHere.JPEG');
        Free;
    end;
finally
    Free;
end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    MyJPEG : TJPEGImage;
    MyBMP   : TBitmap;
begin
    { Convert a JPEG to a BMP }

    MyJPEG := TJPEGImage.Create;
    with MyJPEG do begin
        LoadFromFile('YourJpegHere.JPEG');
        MyBMP := TBitmap.Create;
        with MyBMP do begin
            Width := MyJPEG.Width;
            Height := MyJPEG.Height;
            Canvas.Draw(0,0,MyJPEG);
            SaveToFile('YourBmpHere.BMP');
            Free;
        end;
        Free;
    end;
end;
end;
```

Reference:

None

3/31/99 1:08:27 PM

Last Modified: 01-SEP-99

## Article #19531: Installing JPEG Components that Ship with Delphi 3

Technical Information Database

TI4531D.txt - Installing JPEG Components that Ship with Delphi 3

Category :Delphi 3.x  
Platform :All-32Bit  
Product :All32Bit,

### Description:

#### Step 1 - Copying the JPEG files to your hard drive

Insert your Delphi 3 CD in your CD-ROM drive.

In the Explorer, browse to the INFO\EXTRAS directory.

To copy the JPEG folder, drag and drop it from the Explorer to another folder, e.g. C:\MyStuff.

All the JPEG files should now be in C:\MyStuff\JPEG.

To make thing easier for us, now copy all the files from the C:\MyStuff\JPEG\OBJ folder to the C:\MyStuff\JPEG folder.

Do this by, in the Explorer, selecting all the files in the C:\MyStuff\JPEG\OBJ. When all files are selected, right click and choose Copy. Then browse to C:\MyStuff\JPEG and choose Paste.

#### Step 2 - Installing the JPEG components

Launch Delphi 3.0

From Delphi's main menu, select Component | Install Component

You'll should now see the "Install Component" dialog.

Hit the Browse button next to the Unit file name box.

In the "Unit file name" dialog, browse to C:\MyStuff\JPEG.

Select the file named JPEG.PAS and hit Open.

The Unit file name box should now be filled in properly.

Make sure the Package file name is set to the Delphi User's Components package, which by default is in this location:

C:\Program Files\Borland\Delphi 3\Lib\dclusr30.dpk

Hit OK.

You'll now see a "Confirm" dialog, saying that the package will be installed.

Hit Yes.

You should now see an "Information" dialog, saying that the

package is installed.

Exit the package editor and make sure you save the changes to the package.

### Step 3 - Using the JPEG component

Simply drop down a TImage and load a JPEG into the Picture property. Also, make sure the JPEG unit is in the USES clause, so that the JPEG stuff is registered at run-time.

Reference:

4/1/99 11:43:31 AM

Last Modified: 01-SEP-99

## Article #19532: Extracting a JPEG Resource from an EXE

Technical Information Database

TI4532D.txt - Extracting a JPEG Resource from an EXE

Category :VCL  
Platform :All-32Bit  
Product :Delphi3.x,

### Description:

Pick any JPEG you want to link into your project and copy it to your project directory. Now, in the same directory, create an RC file (resource script). Call it MYRES.RC and have it contain this line:

```
MYJPEG RCDATA YourJpegHere.JPEG
```

Now, compile this RC file. At the command line, say:

```
BRCC32 MYRES
```

This'll create MYRES.RES which you can link into your project.

Now, start a brand new project in Delphi 3. Drop a TButton and a TImage on the form. Then double click on the button. Then replace the code inbetween "{\$R \*.DFM}" and "end." in Unit1 with this code:

```
{ $R MyRes.RES }  
  
uses  
    JPEG;  
  
const  
    ResName = 'MYJPEG';  
  
procedure TForm1.Button1Click(Sender: TObject);  
var  
    MyRes : Integer;  
    MyResP : Pointer;  
    MyResS : Integer;  
    MyMS : TMemoryStream;  
    OldDir : String;
```

```

begin
  Caption := 'Please wait...';
  Application.ProcessMessages;
  GetDir(0,OldDir);
  ChDir(ExtractFilePath(Application.ExeName));
  MyRes := FindResource(HInstance,PChar(ResName),RT_RCDATA);
  if MyRes <> 0 then begin
    MyResS := SizeOfResource(HInstance,MyRes);
    MyRes := LoadResource(HInstance,MyRes);
    if MyRes <> 0 then begin
      MyResP := LockResource(MyRes);
      if MyResP <> nil then begin
        MyMS := TMemoryStream.Create;
        with MyMS do begin
          Write(MyResP^,MyResS);
          Seek(0,soFromBeginning);
          Imager1.Picture.Graphic := TJPEGImage.Create;
          Imager1.Picture.Graphic.LoadFromStream(MyMS);
          Free;
        end;
        UnLockResource(MyRes);
        with Imager1.Picture do
          Caption := 'Cool JPEG! ('+
            IntToStr(Width)+'x'+IntToStr(Height)+')';
        end else
          Caption := 'LockResource failed. '+
            SysErrorMessage(GetLastError);
        FreeResource(MyRes);
      end else
        Caption := 'LoadResource failed. '+
          SysErrorMessage(GetLastError);
      end else
        Caption := 'FindResource failed. '+SysErrorMessage(GetLastError);
      ChDir(OldDir);
    end;
  end;

```

Reference:

None

3/31/99 1:09:14 PM

Last Modified: 01-SEP-99



## Article #19534: Creating a 32-Bit Screen Saver in 32-bit Delphi

Technical Information Database

TI4534D.txt - Creating a 32-Bit Screen Saver in 32-bit Delphi

Category :Miscellaneous  
Platform :All-32Bit  
Product :All32Bit,

### Description:

This TI shows how you can write a 32-bit screen saver in 32 bit Delphi.

The screen saver contains support for preview mode (the little monitor in Display Properties | Screen Saver), as well as password protection and a configuration dialog.

First, a brief overview of what a screen saver is. For a more complete overview, consult the MSDN (Microsoft Developers Network), books and articles on the subject. There are also web sites with screen saver information and source code.

A screen saver is just an executable that has an extension of SCR instead of EXE. In Delphi 3, you can set this using the \$E compiler directive.

A screen saver can be launched in several ways:

When the screen saver timeout happens

By going to the Screen Saver tab in Display Properties (preview)

By configuring it

By previewing it (full screen)

By changing the screen saver password (Win95)

The screen saver is launched with different parameters depending on how it's launched:

When the screen saver timeout happens, it's launched with ParamStr(1) containing either '/S', '-S', or

just 'S'.

When you go to the Screen Saver tab in Display Properties the screen saver is supposed to preview itself in the little monitor. ParamStr(1) will contain '/P', '-P', or just 'P'. At the same time, ParamStr(2) will contain the window handle for the little monitor window.

When you configure the screen saver it's launched either with no parameters at all, or ParamStr(1) will contain '/C', '-C', or just 'C'.

When previewing the screen saver in full screen mode, it's launched just as if the screen saver timeout happened. ParamStr(1) will contain either '/S', '-S', or just 'S'.

When you change the screen saver password (Win95) ParamStr(1) will contain either '/A', '-A', or just 'A'.

A screen saver has to make sure it's not launched several times. In this screen saver this is accomplished by way of a semaphore (see Simple.dpr below).

A couple of things to look out for when it comes to the little preview window:

You have to wait until the window is visible  
You have to kill the previewing when the window is made invisible  
You'll see how both of these things are handled in Simple.dpr below.

As you know a screen saver has to respond to mouse events and key presses. When you don't have a password, it should simply shut down. When you have a password set, it should ask for the password.  
You'll see this as part of the SSave unit (see SSave.pas below).

One final note before we create the screen saver:  
Debugging a screen saver can be very tricky, so make sure you save your code before you run the screen saver in any way... If it locks up, you will most likely have to reboot, or at least kill Delphi 3 using the Task Manager...

OK, now let's go ahead and create the screen saver!

1. Create a new folder, e.g. C:\Foo. Launch Delphi 3, and start a brand new application. From the Project Manager, delete Unit1 and Form1 from the project. Do a File | Save Project As, and save the project as Simple.dpr in the newly created folder.
2. Do a File | New | Form. Select Unit1 in the Code Editor. Do a File | Save As, and save the new form as SSetup.pas.
3. Do a File | New | Form. Select Unit2 in the Code Editor. Do a File | Save As, and save the new form as SSave.pas.
4. Do a File | New | Unit. Select Unit3 in the Code Editor. Do a File | Save As, and save the new unit as Globals.pas.
5. Do a File | New | Unit. Select Unit3 in the Code Editor. Do a File | Save As, and save the new unit as CodeSpot.pas.
6. Select the form SSetup. Right click on the form and select View As Text. Replace all the text in the editor with the code for SSetup.dfm below. Right click and select View As Form. Now go to the unit SSetup.pas in the editor and replace all the code with the code for SSetup.pas below.
7. Select the form SSave. Right click on the form and select View As Text. Replace all the text in the editor with the code for SSave.dfm below. Right click and select View As Form. Now go to the unit SSave.pas in the editor and replace all the code with the code for SSave.pas below.
8. Select the unit Globals.pas. Replace all the code with the code for Globals.pas below.
9. Select the unit CodeSpot.pas. Replace all the code with the code for CodeSpot.pas below.
10. Do a View | Project Source. Replace all the code with the code for Simple.dpr below.

11.Do a Project | Build All.

12.Copy the compiled screen saver Simple.Scr into your system directory (Something like C:\WinNT\System32 or C:\Win95\System). You can right click on Simple.Scr in the Explorer and select Install.

13.Have lots of fun with your new screen saver project!

```
SSetup.pas
```

```
*****
```

```
unit Ssetup;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls, ExtCtrls, Buttons, Spin, ExtDlgs;
```

```
type
```

```
  TSetup = class(TForm)
```

```
    Ball1Box: TGroupBox;
```

```
    Label3: TLabel;
```

```
    xPos1: TSpinEdit;
```

```
    yPos1: TSpinEdit;
```

```
    xVel1: TSpinEdit;
```

```
    yVel1: TSpinEdit;
```

```
    Label5: TLabel;
```

```
    Size1: TSpinEdit;
```

```
    Label7: TLabel;
```

```
    Label4: TLabel;
```

```
    Label8: TLabel;
```

```
    Random1: TCheckBox;
```

```
    OKButton: TBitBtn;
```

```
    CancelButton: TBitBtn;
```

```
    TestButton: TBitBtn;
```

```
    procedure TestButtonClick(Sender: TObject);
```

```
    procedure FormCreate(Sender: TObject);
```

```
    procedure FormActivate(Sender: TObject);
```

```
    procedure OKButtonClick(Sender: TObject);
```

```
    procedure CancelButtonClick(Sender: TObject);
```

```
    procedure Random1Click(Sender: TObject);
```

```
    procedure Size1Change(Sender: TObject);
```

```
private
  { Private declarations }
  Loading : Boolean;
public
  { Public declarations }
end;
```

```
var
  Setup: TSetup;
```

```
implementation
```

```
uses
  SSave, Globals;
```

```
{ $R *.DFM }
```

```
procedure TSetup.TestButtonClick(Sender: TObject);
begin
```

```
  DefRandom := Random1.Checked;
  DefSize := Size1.Value;
  DefPosX := xPos1.Value;
  DefPosY := yPos1.Value;
  DefVelX := xVel1.Value;
  DefVelY := yVel1.Value;
```

```
  TestMode := True;
  Scrn := TScrn.Create(Application);
  Scrn.LoadingApp := True;
  Scrn.Left := -1000;
  Scrn.Top := -1000;
  Scrn.Width := 0;
  Scrn.Height := 0;
  Scrn.ShowModal;
  Scrn.Free;
  SetFocus;
  TestMode := False;
```

```
end;
```

```
procedure TSetup.FormCreate(Sender: TObject);
```

```
begin
  Loading := True;
end;
```

```
procedure TSetup.FormActivate(Sender: TObject);
```

```

begin
  if Loading then begin
    Loading := False;

    ReadIniFile;

    Size1.Value := DefSize;
    xPos1.Value := DefPosX;
    yPos1.Value := DefPosY;
    xVel1.Value := DefVelX;
    yVel1.Value := DefVelY;

    Random1.Checked := DefRandom;

    xPos1.MinValue := (DefSize*SpotSize div 2)+1;
    xPos1.MaxValue := Screen.Width-(DefSize*SpotSize div 2);
    yPos1.MinValue := (DefSize*SpotSize div 2)+1;
    yPos1.MaxValue := Screen.Height-(DefSize*SpotSize div 2);
  end;
end;

```

```

procedure TSetup.OKButtonClick(Sender: TObject);

```

```

begin
  DefRandom := Random1.Checked;
  DefSize := Size1.Value;
  DefPosX := xPos1.Value;
  DefPosY := yPos1.Value;
  DefVelX := xVel1.Value;
  DefVelY := yVel1.Value;

```

```

  WriteIniFile;
  Close;
end;

```

```

procedure TSetup.CancelButtonClick(Sender: TObject);

```

```

begin
  Close;
end;

```

```

procedure TSetup.Random1Click(Sender: TObject);

```

```

var
  NewColor : TColor;
begin
  NewColor := clWindow;
  with Random1 do begin

```

```
if Checked then
  NewColor := clBtnFace;

  DefRandom := Checked;

  Size1.Enabled := not Checked;
  xPos1.Enabled := not Checked;
  yPos1.Enabled := not Checked;
  xVel1.Enabled := not Checked;
  yVel1.Enabled := not Checked;
end;
```

```
Size1.Color := NewColor;
xPos1.Color := NewColor;
yPos1.Color := NewColor;
xVel1.Color := NewColor;
yVel1.Color := NewColor;
end;
```

```
procedure TSetup.Size1Change(Sender: TObject);
begin
  xPos1.MinValue := (Size1.Value*SpotSize div 2)+1;
  xPos1.MaxValue := Screen.Width-(Size1.Value*SpotSize div 2);
  yPos1.MinValue := (Size1.Value*SpotSize div 2)+1;
  yPos1.MaxValue := Screen.Height-(Size1.Value*SpotSize div 2);

  xPos1.Value := xPos1.Value;
  yPos1.Value := yPos1.Value;
end;
```

end.

```
*****
SSetup.dfm
*****
```

```
object Setup: TSetup
  Left = 260
  Top = 188
  BorderIcons = []
  BorderStyle = bsDialog
  Caption = 'Simple Saver Setup'
  ClientHeight = 145
  ClientWidth = 345
```

```
Font.Charset = DEFAULT_CHARSET
Font.Color = clBlack
Font.Height = -12
Font.Name = 'Arial'
Font.Style = []
Position = poScreenCenter
ShowHint = True
OnActivate = FormActivate
OnCreate = FormCreate
PixelsPerInch = 96
TextHeight = 15
object Ball1Box: TGroupBox
  Left = 8
  Top = 8
  Width = 329
  Height = 89
  Caption = 'Settings'
  TabOrder = 0
  object Label3: TLabel
    Left = 72
    Top = 40
    Width = 30
    Height = 15
    Caption = 'x-pos'
  end
  object Label5: TLabel
    Left = 8
    Top = 40
    Width = 23
    Height = 15
    Caption = 'Size'
  end
  object Label7: TLabel
    Left = 136
    Top = 40
    Width = 30
    Height = 15
    Caption = 'y-pos'
  end
  object Label4: TLabel
    Left = 200
    Top = 40
    Width = 24
    Height = 15
    Caption = 'x-vel'
```



```
end
object Label8: TLabel
  Left = 264
  Top = 40
  Width = 24
  Height = 15
  Caption = 'y-vel'
end
object xPos1: TSpinEdit
  Left = 72
  Top = 56
  Width = 57
  Height = 24
  MaxLength = 4
  MaxValue = 9999
  MinValue = 0
  TabOrder = 2
  Value = 0
end
object yPos1: TSpinEdit
  Left = 136
  Top = 56
  Width = 57
  Height = 24
  MaxLength = 4
  MaxValue = 9999
  MinValue = 0
  TabOrder = 3
  Value = 0
end
object xVel1: TSpinEdit
  Left = 200
  Top = 56
  Width = 57
  Height = 24
  MaxLength = 4
  MaxValue = 10
  MinValue = -10
  TabOrder = 4
  Value = 0
end
object yVel1: TSpinEdit
  Left = 264
  Top = 56
  Width = 57
```

```
    Height = 24
    MaxLength = 4
    MaxValue = 10
    MinValue = -10
    TabOrder = 5
    Value = 0
end
object Size1: TSpinEdit
    Left = 8
    Top = 56
    Width = 57
    Height = 24
    MaxLength = 4
    MaxValue = 4
    MinValue = 1
    TabOrder = 1
    Value = 1
    OnChange = Size1Change
end
object Random1: TCheckBox
    Left = 8
    Top = 16
    Width = 97
    Height = 17
    Caption = 'Randomize'
    TabOrder = 0
    OnClick = Random1Click
end
end
object OKButton: TBitBtn
    Left = 8
    Top = 104
    Width = 73
    Height = 33
    Caption = 'Ok'
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clBlack
    Font.Height = -12
    Font.Name = 'Arial'
    Font.Style = []
    ParentFont = False
    TabOrder = 1
    OnClick = OKButtonClick
    Kind = bkOK
end
```

```
object CancelButton: TBitBtn
  Left = 136
  Top = 104
  Width = 73
  Height = 33
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clBlack
  Font.Height = -12
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 2
  OnClick = CancelButtonClick
  Kind = bkCancel
```

```
end
```

```
object TestButton: TBitBtn
  Left = 264
  Top = 104
  Width = 73
  Height = 33
  Caption = 'Test'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clBlack
  Font.Height = -12
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 3
  OnClick = TestButtonClick
```

```
Glyph.Data = {
  76010000424D760100000000000000760000002800000020000000100000000100
  0400000000000000100000000000000000000000100000001000000000000000
  8000008000000008080008000000080008000808000007F7F7F00BFBFBF000000
  FF0000FF000000FFFF00FF000000FF00FF00FFFF0000FFFFFFF00333333000000
  0333333FFFF77777773F330000077777770333777773FFFFFF733077777000000
  03337F3F3F777777733F0797A770003333007F737337773F3377077777778803
  30807F333333337FF73707888887880007707F3FFFF333777F37070000878807
  07807F777733337F7F3707888887880808807F333333337F7F37077777778800
  08807F333FFF337773F7088800088803308073FF777FFF733737300008000033
  33003777737777333377333080333333333333F737333333333300803333333
  33333773733333333333088033333333333373F7F33333333333308033333333
  33333737333333333333330333333333333333733333333333333}
```

```
NumGlyphs = 2
```

```
end
```

```
end
```

\*\*\*\*\*

Ssave.pas

\*\*\*\*\*

unit Ssave;

interface

uses WinTypes, WinProcs, Graphics, Forms, Messages, Classes, Controls,  
ExtCtrls, StdCtrls, SysUtils;

type

TScrn = class(TForm)

Image1: TImage;

procedure FormKeyDown(Sender: TObject; var Key: Word;  
Shift: TShiftState);

procedure FormCreate(Sender: TObject);

procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,  
Y: Integer);

procedure FormActivate(Sender: TObject);

procedure FormMouseDown(Sender: TObject; Button: TMouseButton;  
Shift: TShiftState; X, Y: Integer);

procedure FormClose(Sender: TObject; var Action: TCloseAction);

private

{ Private declarations }

Mouse : TPoint;

procedure StartSaver(var WinMsg : TMessage); message WM\_USER+1;

procedure StopSaver(var WinMsg : TMessage); message WM\_USER+2;

procedure GetPassword;

procedure Trigger(Sender : TObject; var Done : Boolean);

public

{ Public declarations }

LoadingApp : Boolean;

end;

var

Scrn : TScrn;

DesktopBitmap : TBitmap;

implementation

uses

CodeSpot, Globals, Registry;

```

const
  IgnoreCount : Integer = 0;

{$R *.DFM}

procedure CursorOff;
begin
  ShowCursor(False);
end;

procedure CursorOn;
begin
  ShowCursor(True);
end;

procedure TScrn.StartSaver(var WinMsg : TMessage);
begin
  DrawSpot;
end;

procedure TScrn.StopSaver(var WinMsg : TMessage);
begin
  GetPassword;
end;

procedure TScrn.GetPassword;
var
  MyMod      : THandle;
  PwdFunc    : function (Parent : THandle) : Boolean; stdcall;
  SysDir     : String;
  NewLen     : Integer;
  MyReg      : TRegistry;
  OkToClose  : Boolean;
begin
  if (SSMode <> ssRun) or TestMode then begin
    Close;
    Exit;
  end;

  IgnoreCount := 5;
  OkToClose := False;
  MyReg := TRegistry.Create;
  MyReg.RootKey := HKEY_CURRENT_USER;
  if MyReg.OpenKey('Control Panel\Desktop',False) then begin
    try

```

```

try
  ShowCursor(True);
  if MyReg.ReadInteger('ScreenSaveUsePassword') <> 0 then begin
    SetLength(SysDir,MAX_PATH);
    NewLen := GetSystemDirectory(PChar(SysDir),MAX_PATH);
    SetLength(SysDir,NewLen);
    if (Length(SysDir) > 0) and (SysDir[Length(SysDir)] <> '\')
then
    SysDir := SysDir+'\';
    MyMod := LoadLibrary(PChar(SysDir+'PASSWORD.CPL'));
    if MyMod = 0 then
      OkToClose := True
    else begin
      PwdFunc := GetProcAddress(MyMod,'VerifyScreenSavePwd');
      if PwdFunc(Handle) then
        OkToClose := True;
      FreeLibrary(MyMod);
    end;
  end
  else
    OkToClose := True;
  finally
    ShowCursor(False);
  end;
except
  OkToClose := True;
end;
end
else
  OkToClose := True;

MyReg.Free;

if OkToClose then
  Close;
end;

procedure TScrn.Trigger(Sender : TObject; var Done : Boolean);
begin
  PostMessage(Handle,WM_USER+1,0,0);
end;

procedure TScrn.FormKeyDown(Sender: TObject; var Key: Word; Shift:
TShiftState);
begin

```

```

    GetPassword;
end;

procedure TScrn.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
Y: Integer);
begin
    if IgnoreCount > 0 then begin
        Dec(IgnoreCount);
        Exit;
    end;

    if (Mouse.X = -1) and (Mouse.Y = -1) then begin
        Mouse.X := X;
        Mouse.Y := Y;
    end
    else
        if (Abs(X-Mouse.X) > 2) and (Abs(Y-Mouse.Y) > 2) then begin
            Mouse.X := X;
            Mouse.Y := Y;
            GetPassword;
        end;
    end;

end;

procedure TScrn.FormCreate(Sender: TObject);
begin
    LoadingApp := True;
end;

procedure TScrn.FormActivate(Sender: TObject);
var
    Dummy : Boolean;
begin
    if LoadingApp then begin
        LoadingApp := False;
        Scrn.Color := clBlack;
        Scrn.Top := 0;
        Scrn.Left := 0;
        Scrn.Width := Screen.Width;
        Scrn.Height := Screen.Height;
        InitSpot;
        Mouse.X := -1;
        Mouse.Y := -1;
        Application.OnIdle := Trigger;
        SetWindowPos(Handle,HWND_TOPMOST,0,0,0,0,SWP_NOSIZE + SWP_NOMOVE);
        SystemParametersInfo(SPI_SCREENSAVER_RUNNING,1,@Dummy,0);
    end;
end;

```

```

    CursorOff;
    Scrn.Visible := True;
    SetCapture(Scrn.Handle);
end;
end;

procedure TScrn.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);

begin
    GetPassword;
end;

procedure TScrn.FormClose(Sender: TObject; var Action: TCloseAction);
var
    Dummy : Boolean;
begin
    SystemParametersInfo(SPI_SCREENSAVERRUNNING, 0, @Dummy, 0);
    Application.OnIdle := nil;
    ReleaseCapture;
    CursorOn;
end;

end.

```

```

*****
SSave.dfm
*****

```

```

object Scrn: TScrn
    Left = 314
    Top = 376
    HorzScrollBar.Visible = False
    BorderIcons = [biSystemMenu]
    BorderStyle = bsNone
    ClientHeight = 130
    ClientWidth = 457
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -13
    Font.Name = 'System'
    Font.Style = []
    OnActivate = FormActivate
    OnClose = FormClose
    OnCreate = FormCreate
    OnKeyDown = FormKeyDown

```



```
OnMouseDown = FormMouseDown
OnMouseMove = FormMouseMove
PixelsPerInch = 96
TextHeight = 16
object Image1: TImage
    Left = 0
    Top = 0
    Width = 457
    Height = 130
    Align = alClient
    Visible = False
end
end
```

```
*****
Globals.pas
*****
```

```
unit Globals;
```

```
interface
```

```
type
```

```
TSSMode = (ssSetPwd,ssPreview,ssConfig,ssRun);
```

```
const
```

```
SSMode      : TSSMode = ssRun;
```

```
TestMode    : Boolean = False;
```

```
Section     = 'Screen Saver.Simple Screen Saver';
```

```
SpotSize    = 50;
```

```
DefSize     : Integer = 2;
```

```
DefPosX     : Integer = 51;
```

```
DefPosY     : Integer = 51;
```

```
DefVelX     : Integer = 1;
```

```
DefVelY     : Integer = 1;
```

```
DefRandom   : Boolean = True;
```

```
procedure ReadIniFile;
```

```
procedure WriteIniFile;
```

implementation

uses

  IniFiles;

procedure ReadIniFile;

var

  IniFile : TIniFile;

begin

  IniFile := TIniFile.Create('CONTROL.INI');

  DefSize := IniFile.ReadInteger(Section, 'Size1', DefSize);

  DefPosX := IniFile.ReadInteger(Section, 'PosX1', DefPosX);

  DefPosY := IniFile.ReadInteger(Section, 'PosY1', DefPosY);

  DefVelX := IniFile.ReadInteger(Section, 'VelX1', DefVelX);

  DefVely := IniFile.ReadInteger(Section, 'Vely1', DefVely);

  DefRandom := IniFile.ReadBool(Section, 'Rand1', DefRandom);

  IniFile.Free;

end;

procedure WriteIniFile;

var

  IniFile : TIniFile;

begin

  IniFile := TIniFile.Create('CONTROL.INI');

  IniFile.WriteInteger(Section, 'Size1', DefSize);

  IniFile.WriteInteger(Section, 'PosX1', DefPosX);

  IniFile.WriteInteger(Section, 'PosY1', DefPosY);

  IniFile.WriteInteger(Section, 'VelX1', DefVelX);

  IniFile.WriteInteger(Section, 'Vely1', DefVely);

  IniFile.WriteBool(Section, 'Rand1', DefRandom);

  IniFile.Free;

end;

end.

\*\*\*\*\*

CodeSpot.pas

\*\*\*\*\*

```

unit Codespot;

interface

uses
  WinTypes, WinProcs, Graphics, Forms, Controls, Classes, Sysutils,
  Dialogs;

var
  zx, zy  : Integer;
  cx, cy,
  vx, vy,
  d       : Real;
  Picture : HBitmap;

procedure InitSpot;
procedure DrawSpot;

implementation

uses
  SSave, Globals;

procedure InitSpot;
begin
  Randomize;

  if not TestMode then
    ReadIniFile;

  zx := Screen.Width;
  zy := Screen.Height;

  d := (Random(4)+1)*SpotSize;
  cx := Random((zx div 2)-Round(d)-1)+1;
  cy := Random(zy-Round(d)-1)+1;
  vx := Random(2)+1;
  vy := Random(2)+1;
  if Random(2) = 0 then
    vx := -vx;
  if Random(2) = 0 then
    vy := -vy;

  if not DefRandom then begin
    d := DefSize*SpotSize;

```

```

    cx := DefPosX-d/2;
    cy := DefPosY-d/2;
    vx := DefVelX;
    vy := DefVelY;
end;

Scrn.Imagel.Picture.Bitmap := DesktopBitmap;
Picture := Scrn.Imagel.Picture.Bitmap.Handle;
end;

procedure DrawSpot;
var
    WinDC, MemDC : HDC;
    Rgn1, Rgn3    : HRgn;
begin
    WinDC := GetDC(Scrn.Handle);
    MemDC := CreateCompatibleDC(WinDC);

    SelectObject(MemDC,Picture);

    if ((cx+vx <= 0) or (cx+d+vx >= zx)) then
        vx := -vx;

    if ((cy+vy <= 0) or (cy+d+vy >= zy)) then
        vy := -vy;

    cx := cx+vx;
    cy := cy+vy;

    Rgn3 := CreateRectRgn(0,0,zx,zy);
    Rgn1 := CreateEllipticRgn(Round(cx),Round(cy),
                             Round(cx+d),Round(cy+d));

    CombineRgn(Rgn3,Rgn3,Rgn1,RGN_DIFF);
    FillRgn(WinDC,Rgn3,GetStockObject(BLACK_BRUSH));

    SelectObject(WinDC,Rgn1);
    BitBlt(WinDC,0,0,zx,zy,MemDC,0,0,SRCCOPY);

    DeleteObject(Rgn3);
    DeleteObject(Rgn1);

    DeleteDC(MemDC);
    ReleaseDC(Scrn.Handle,WinDC);
end;

```

end.

\*\*\*\*\*

Simple.dpr

\*\*\*\*\*

program Simple;

uses

Forms,  
SysUtils,  
Windows,  
Graphics,  
Classes,  
Ssave in 'SSave.pas' {Scrn},  
Codespot in 'CodeSpot.pas',  
Ssetup in 'SSetup.pas' {Setup},  
Globals in 'Globals.pas';

{ \$E SCR }  
{ \$R \*.RES }

var

MySem : THandle;  
Arg1,  
Arg2 : String;  
DemoWnd : HWnd;  
MyRect : TRect;  
MyCanvas : TCanvas;  
x, y,  
dx, dy : Integer;  
MyBkgBitmap,  
InMemBitmap : TBitmap;  
ScrWidth,  
ScrHeight : Integer;  
SysDir : String;  
NewLen : Integer;  
MyMod : THandle;  
PwdFunc : function (a : PChar; ParentHandle : THandle; b, c :  
Integer) :  
Integer; stdcall;

begin

Arg1 := UpperCase(ParamStr(1));

```
Arg2 := UpperCase(ParamStr(2));
```

```
if (Copy(Arg1,1,2) = '/A') or (Copy(Arg1,1,2) = '-A') or  
    (Copy(Arg1,1,1) = 'A') then  
    SSMode := ssSetPwd;
```

```
if (Copy(Arg1,1,2) = '/P') or (Copy(Arg1,1,2) = '-P') or  
    (Copy(Arg1,1,1) = 'P') then  
    SSMode := ssPreview;
```

```
if (Copy(Arg1,1,2) = '/C') or (Copy(Arg1,1,2) = '-C') or  
    (Copy(Arg1,1,1) = 'C') or (Arg1 = '') then  
    SSMode := ssConfig;
```

```
if SSMode = ssSetPwd then begin  
    SetLength(SysDir,MAX_PATH);  
    NewLen := GetSystemDirectory(PChar(SysDir),MAX_PATH);  
    SetLength(SysDir,NewLen);  
    if (Length(SysDir) > 0) and (SysDir[Length(SysDir)] <> '\') then  
        SysDir := SysDir+'\';  
    MyMod := LoadLibrary(PChar(SysDir+'MPR.DLL'));  
    if MyMod <> 0 then begin  
        PwdFunc := GetProcAddress(MyMod,'PwdChangePasswordA');  
        if Assigned(PwdFunc) then  
            PwdFunc('SCRSAVE',StrToInt(Arg2),0,0);  
        FreeLibrary(MyMod);  
    end;  
    Halt;  
end;
```

```
MySem := CreateSemaphore(nil,0,1,'SimpleSaverSemaphore');  
if ((MySem <> 0) and (GetLastError = ERROR_ALREADY_EXISTS)) then  
begin  
    CloseHandle(MySem);  
    Halt;  
end;
```

```
Application.Initialize;
```

```
if SSMode = ssPreview then begin  
    DemoWnd := StrToInt(Arg2);  
    while not IsWindowVisible(DemoWnd) do  
        Application.ProcessMessages;  
    GetWindowRect(DemoWnd,MyRect);  
    ScrWidth := MyRect.Right-MyRect.Left+1;
```

```

ScrHeight := MyRect.Bottom-MyRect.Top+1;
MyRect := Rect(0,0,ScrWidth-1,ScrHeight-1);
MyCanvas := TCanvas.Create;
MyCanvas.Handle := GetDC(DemoWnd);
MyCanvas.Pen.Color := clWhite;
x := (ScrWidth div 2)-16;
y := (ScrHeight div 2)-16;
dx := 1;
dy := 1;
MyBkgBitmap := TBitmap.Create;
with MyBkgBitmap do begin
  Width := ScrWidth;
  Height := ScrHeight;
end;
MyBkgBitmap.Canvas.FillRect(Rect(0,0,ScrWidth-1,ScrHeight-1));
InMemBitmap := TBitmap.Create;
with InMemBitmap do begin
  Width := ScrWidth;
  Height := ScrHeight;
end;
while IsWindowVisible(DemoWnd) do begin
  InMemBitmap.Canvas.CopyRect(MyRect,MyBkgBitmap.Canvas,MyRect);
  InMemBitmap.Canvas.Draw(x,y,Application.Icon);
  MyCanvas.CopyRect(MyRect,InMemBitmap.Canvas,MyRect);
  Sleep(10);
  Application.ProcessMessages;
  if (x = 0) or (x = (ScrWidth-33)) then
    dx := -dx;
  if (y = 0) or (y = (ScrHeight-33)) then
    dy := -dy;
  x := x+dx;
  y := y+dy;
end;
MyBkgBitmap.Free;
InMemBitmap.Free;
MyCanvas.Free;
CloseHandle(MySem);
Halt;
end;

DesktopBitmap := TBitmap.Create;
with DesktopBitmap do begin
  Width := Screen.Width;
  Height := Screen.Height;
end;

```

```
BitBlt(DesktopBitmap.Canvas.Handle,0,0,Screen.Width,Screen.Height,  
GetDC(GetDesktopWindow),0,0,SrcCopy);
```

```
if SSMODE = ssConfig then begin  
  Application.CreateForm(TSetup, Setup);  
end else  
  Application.CreateForm(TScrn, Scrn);
```

```
Application.Run;
```

```
DesktopBitmap.Free;  
CloseHandle(MySem);
```

```
end.
```

Reference:

None

4/22/99 4:08:08 PM

Last Modified: 01-SEP-99



## Article #19538: Surfacing an Event on an ActiveForm to a Container

Technical Information Database

TI4538D.txt - Surfacing an Event on an ActiveForm to a Container

Category :ActiveX/OLE/COM/ActiveForm

Platform :All-32Bit

Product :All32Bit,

### Description:

Let's create an ActiveForm with a button on it. Now we want to surface the OnClick event of this button to any container using our ActiveForm. What we need to do is this:

Start a brand new ActiveForm project, by selecting File | New... | ActiveX | ActiveForm. Change the ActiveX Name, Implementation Unit, and Project Name if you want. For instance, TestX, TestImpl.pas, and Test.dpr respectively.

Drop a TButton on the TestX form. Double click on the button and make the event handle say:

```
procedure TTestX.Button1Click(Sender: TObject);
var
  OldCaption : String;
begin
  with Button1 do begin
    OldCaption := Caption;
    Caption := 'Auch!';
    Sleep(1000);
    Caption := OldCaption;
  end;
end;
```

This is just to demonstrate that we can handle the button's click event in both the ActiveForm and in the container. What we just implemented was of course the ActiveForm side. Now, let's take care of the container side...

Do a View | Type Library. In ITestXEvents we need to add a new event just below OnPaint. Let's call it OnButton1Click. When you

hit Refresh you should see that the ID (dispid) of this event is set to 9.

Now go back to the OnClick event handler of Button1 on the ActiveForm. Add one line to make it look like this:

```
procedure TTestX.Button1Click(Sender: TObject);
var
  OldCaption : String;
begin
  with Button1 do begin
    OldCaption := Caption;
    Caption := 'Auch!';
    Sleep(1000);
    Caption := OldCaption;
  end;
  if FEvents <> nil then FEvents.OnButton1Click;
end;
```

This will make the container see things after you've handled the OnClick event on the ActiveForm side. If you want to let the container do its thing first, just put the line at the top instead.

Save the ActiveForm project somewhere on disk, e.g. in C:\Test.

Do a Project | Build All.

Do a Run | Register ActiveX Server.

Do a Component | Import ActiveX Control. Find Test Library in the list of registered controls. Select it, and hit Install. Put it in dclusr30.dpk (the default). After hitting OK a couple of times the control should pop up on your ActiveX tab on the component palette. Close the package editor and save the changes to the package.

Do a File | New Application. Drop down a TestX (our ActiveForm) on the form. Double click on the OnButton1Click event handler on the Events tab in the Object Inspector and make it say:

```
procedure TForm1.TestX1Button1Click(Sender: TObject);
begin
  ShowMessage('You hit me!');
end;
```

Run the application, and hit Button1 in the TestX component. You should see the Caption change for a second and then you should see your 'You hit me!' message popping up on the screen.

Reference:

None

4/2/99 2:54:13 PM

Last Modified: 01-SEP-99

## Article #19540: Drag and Drop Selected Text in between Memo Components

Technical Information Database

TI4540D.txt - Drag and Drop Selected Text in between Memo Components

Category :VCL  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Without getting too deep into component creation, here's a very simple way of accomplishing drag and drop of selected text.

Create a new component (TMyMemo) descending off of TMemo.

Make the type declaration look something like this:

```
type
  TMyMemo = class(TMemo)
  private
    FLastSelStart : Integer;
    FLastSelLength : Integer;
    procedure WMLButtonDown(var Message: TWMLButtonDown);
      message WM_LBUTTONDOWN;
  published
    property LastSelStart : Integer read FLastSelStart
      write FLastSelStart;
    property LastSelLength : Integer read FLastSelLength
      write FLastSelLength;
  end;
```

Make the implementation of WMLButtonDown look like this:

```
procedure TMyMemo.WMLButtonDown(var Message: TWMLButtonDown);
var
  Ch : Integer;
begin
  if SelLength > 0 then begin
    Ch := LoWord(Perform(EM_CHARFROMPOS, 0,
      MakeLParam(Message.XPos, Message.YPos)));
    LastSelStart := SelStart;
```

```

    LastSelLength := SelLength;
    if (Ch >= SelStart) and (Ch <= SelStart+SelLength-1) then
        BeginDrag(True)
    else
        inherited;
    end
else
    inherited;
end;

```

Now, install this component into a package, start a brand new project in Delphi 3 and drop two TMyMemos down.

Make them both have an OnDragOver event handler looking like this:

```

procedure TForm1.MyMemo1DragOver(Sender, Source: TObject; X, Y:
Integer;
    State: TDragState; var Accept: Boolean);
begin
    Accept := Source is TMyMemo;
end;

```

Make them both have an OnDragDrop event handler looking like this:

```

procedure TForm1.MyMemo1DragDrop(Sender, Source: TObject;
                                X, Y: Integer);
var
    Dst, Src : TMyMemo;
    Ch       : Integer;
    Temp     : String;
begin
    Dst := Sender as TMyMemo;
    Src := Source as TMyMemo;
    Ch := LoWord(Dst.Perform(EM_CHARFROMPOS, 0, MakeLParam(X, Y)));

    if (Src = Dst) and (Ch >= Src.LastSelStart) and
        (Ch <= Src.LastSelStart+Src.LastSelLength-1) then
        Exit;

    Dst.Text := Copy(Dst.Text, 1, Ch)+Src.SelText+
                Copy(Dst.Text, Ch+1, Length(Dst.Text)-Ch);
    Temp := Src.Text;
    Delete(Temp, Src.LastSelStart+1, Src.LastSelLength);
    Src.Text := Temp;
end;

```

Launch the app and put some text into the memos, and watch what happens as you drag and drop inbetween the two memos, or even drag and drop text from one location in one memo component to another location in the same memo component.

Reference:

4/1/99 11:38:22 AM

Last Modified: 01-SEP-99

## Article #19546: Description of Multi-Byte Character Support in Delphi, BDE

Technical Information Database

TI4546D.txt - Description of Multi-Byte Character Support in Delphi, BDE

Category :Miscellaneous  
Platform :All-32Bit  
Product :Delphi3.x, Delphi4.x,

### Description:

This TI describes the variations in the Multi-Byte Character Support (MBCS) in Borland Products that use the BDE. The BDE 4.0 and above are MBCS enabled and can handle all Asian languages i.e. Simplified Chinese(Mainland), Traditional Chinese(Tainwan), Korean, and Japanese. All Borland products that ship with the BDE 4.0 and above are enabled to handle MBCS. Paradox on the other hand is not a Borland product any more and entering MBCS into tables may give you incomplete or garbage entries.

BDE 4.0 and above do not have any localised runtime messages. For European languages many of these runtime messages are localised and can be gotten through the Borland Delphi Language Pack,(not to be confused with the Translation Suite). For Asian languages the only one that has localised for 4.0 and above is the Taiwanese(Traditional) Delphi Solution Pack sold exclusively in Taiwan.

The Japanese BDE 4.5 is actually identical to the BDE 4.5 that ships world-wide except that it contains localised Japanese runtime messages and some Japanese specific sorting and locale info. The Japanese BDE 4.51 has a special resource file(IDR2011.DLL) that contains this. It also has the BDE Administrator that is adapted to Input Method Editors for the language.

### Reference:

None

4/22/99 12:30:53 PM

Last Modified: 01-SEP-99

## Article #19548: Creating Lookup Fields at Runtime

Technical Information Database

TI4548D.txt - Creating Lookup Fields at Runtime

Category :Database Programming  
Platform :All-32Bit  
Product :All32Bit,

### Description:

Creating a lookup field at runtime differs in only a few ways from creating any standard TField object. (for more explicit information on this topic and other related topics see the TI entitled "Dynamically creating a TTable & fields at runtime".) Three properties of a field that must be set in order to function as a Lookup field are LookUpDataset, LookUpKeyFields and LookUpResultField. The sample code below demonstrates how to create a Lookup field at runtime using the two DBDEMOS tables 'customer.DB' and 'orders.DB'.

Start a new application (File|New Application).

Drop two TTables, a TButton, a TDBLookupComboBox and a TDataSource on a form.

Set the two TTables 'DatabaseName' property to 'DBDEMOS'.

Set Table1's 'TableName' property to 'CUSTOMER.DB' and Table2's 'TableName' property to 'ORDERS.DB'.

Right click on the TTable object for Table2 and launch the field editor. Add the 'CustNo' field object (right click on the fields editor and choose 'Add Fields', then select 'CustNo' from the list).

Copy and paste the code below into the code editor.

Example:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
Forms, Classes, Controls, StdCtrls, Db, DBTables, DBCtrls;
```



```

type
  TForm1 = class(TForm)
    Table1: TTable;    // DBDemos customer table
    Table2: TTable;    // DBDemos orders table
    Button1: TButton;
    DBLookupComboBox1: TDBLookupComboBox;
    DataSource1: TDataSource;
    Table2CustNo: TFloatField; // CustNo key field object used
                                // for Lookup
    procedure Button1Click(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  with TStringField.Create(Table2) do begin
    FieldName := 'MyLookup';
    FieldKind:= fkLookup;
    DataSet := Table2;
    Name := DataSet.Name + FieldName;
    KeyFields:= 'CustNo';
    LookUpDataset:= Table1;
    LookUpKeyFields:= 'CustNo';
    LookUpResultField:= 'Company';
    DbLookupCombobox1.DataField:= FieldName;
    DataSource1.DataSet:= DataSet;
    Table2.FieldDefs.Add(Name, ftString, 20, false);
  end;
  DbLookupCombobox1.DataSource:= DataSource1;
  Table1.Active:= True;
  Table2.Active:= True;
end;

end.

```

Reference:

4/22/99 12:29:17 PM

Last Modified: 01-SEP-99

## Article #19549: ShortCut Keys For Delphi

Technical Information Database

TI4549D.txt - ShortCut Keys For Delphi

Category :IDE  
Platform :All-32Bit  
Product :Delphi3.x, Delphi4.x,

Description:

Object Inspector ShortCut Keys

Ctrl+DwnArr

Makes the Object ComboBox drop Down and gives it focus.

Ctrl+Tab

Switches Between Events and Properties.

Ctrl+Enter

Executes a dialog associated with a property, makes an event handler procedure.

Alt + DwnArr

If the property is a drop down list, it will drop down this list.

### IDE ShortCutKeys

F1

Displays context-sensitive Help

F4

Run|Go to Cursor

F5

Run|Toggle Breakpoint

F7

Run|Trace Into

F8

Run|Step Over

F9

Run|Run

F11

View|Object Inspector

F12	View Toggle Form/Unit
Ctrl+F2	Run Program Reset
Ctrl+F3	View Call Stack
Ctrl+F4	Close the Focused Page in the Delphi Source Editor
Ctrl+F5	Executes the Watch List and the Watch Properties dialogs
Ctrl+F7	Executes the Evaluate Modify Dialog
Ctrl+F9	Compile application
Ctrl+F12	Executes the 'View Unit' Dialog
Ctrl+Spc	Brings up the Code Completion Window.
Ctrl+Enter	When Done in the Source Code Editor, it executes the Open Dialog Box and populates the file name with the word that the cursor was nearest to with an extension of .pas. It also opens up in the current directory.
Ctrl+Home	Top of Source File
Ctrl+End	End of Source File
Ctrl+Del or Del	Delete a component from the form
Ctrl+C	Copy Selected block to the Clipboard
Ctrl+E	Incremental Word search within a Source file. Searches one letter at a time. Also populates the word search edit box.
Ctrl+F	Full word search. this searches the Source file for a specific word.
Ctrl+J	Code Completion Window
Ctrl+K+E	Changes a word to lowercase

Ctrl+K+F	Capitalize word to the Left of cursor
Ctrl+K+N	Changes a block to uppercase
Ctrl+K+O	Changes a block to lowercase
Ctrl+K+P	Executes the Print Dialog Box
Ctrl+K+R	Reads a block from a file
Ctrl+K+T	Highlight word to the left of cursor
Ctrl+K+W	Writes a selected block to a file
Ctrl+O+G	Search Go to line number
Ctrl+O+O	Inserts compiler options and directives
Ctrl+O+U	Toggles Upper/Lower case of highlighted text
Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+R	Search and Replace a word
Ctrl+S	Save a Source file
Ctrl+T	Delete a word at a time
Ctrl+V	Paste from the Clipboard
Ctrl+X	Cut Selected block to the Clipboard
Ctrl+Y	Delete a line of UNHIGHLIGHTED Source Code
Ctrl+Z	Undo Changes
Ctrl+0	Goes to bookmark 0
Ctrl+1	Goes to bookmark 1
Ctrl+2	Goes to bookmark 2
Ctrl+3	Goes to bookmark 3

Ctrl+4  
Goes to bookmark 4

Ctrl+5  
Goes to bookmark 5

Ctrl+6  
Goes to bookmark 6

Ctrl+7  
Goes to bookmark 7

Ctrl+8  
Goes to bookmark 8

Ctrl+9  
Goes to bookmark 9

Ctrl+Shift+Spc  
Brings up the Code Parameters Window.

Ctrl+Shift+G  
Creates and displays a GUID

Ctrl+Shift+I  
Indents a highlighted block of text

Ctrl+Shift+P  
Plays back a key macro

Ctrl+Shift+R  
Records a key macro

Ctrl+Shift+U  
OutDents a Highlighted block of text

Ctrl+Shift+Y  
Deletes to the End of the Line.

Ctrl+Shift+Z  
Redo Changes

Ctrl+Shift+0  
Sets or removes bookmark 0

Ctrl+Shift+1  
Sets or removes bookmark 1

Ctrl+Shift+2  
Sets or removes bookmark 2

Ctrl+Shift+3  
Sets or removes bookmark 3

Ctrl+Shift+4  
Sets or removes bookmark 4

Ctrl+Shift+5  
Sets or removes bookmark 5

Ctrl+Shift+6  
Sets or removes bookmark 6

Ctrl+Shift+7  
Sets or removes bookmark 7

Ctrl+Shift+8

	Sets or removes bookmark 8
Ctrl+Shift+9	Sets or removes bookmark 9
Alt+F4	Closes Focused Window
Alt+F6	Switches between the last 2 objects that had focus in the IDE (i.e. the Form Designer and the Object Inspector or the Form and the Main Delphi Menu, or the Object Inspector and the Main Delphi Menu, etc..)
Alt+F10	Executes a Right Click in the Source Code Editor
Alt+F12	View Form as Text
Alt+Spc	Current Windows' System Menu
Alt+0	Executes the Window List dialog
Shift+F7	Trace to next Source Line
Shift+F12	Executes the 'View Form' Dialog
Shift+Alt+Arw	Highlights a column of text.

### Highlighting a column of text using the mouse

Hold down Alt+LeftMouseButton. Drag around to get a Highlighted Column...

### Moving Components in Design Time

Highlight the desired component:

Shift+DownArrow	Increases the Components Height property.
Shift+UpArrow	Decreases the Components Height property.
Shift+RightArrow	Increases the Components Width property.
Shift+LeftArrow	

Decreases the Components Width property.  
Ctrl+DownArrow  
Increases the Components Top property.  
Ctrl+UpArrow  
Decreases the Components Top property.  
Ctrl+RightArrow  
Increases the Components Left property.  
Ctrl+LeftArrow  
Decreases the Components Left property.  
Ctrl+Shift+DownArrow  
Increases the Components Top property by a grid snap  
space.  
Ctrl+Shift+UpArrow  
Decreases the Components Top property by a grid snap  
space.  
Ctrl+Shift+RightArrow  
Increases the Components Left property by a grid snap  
space.  
Ctrl+Shift+LeftArrow  
Decreases the Components Left property by a grid snap  
space.

### Delphi's TLB Editor

Tip: If you Create an Interface, Method, Property, etc, you can Ctrl  
+C (copy) it  
to the windows Clipboard and then when you want to create a new one,  
you can  
Ctrl+V (paste) it from the windows clipboard... The Type Library  
Editor will  
create a new object of whatever type that you copied to the clipboard.

### Delphi's Package Editor

Ctrl+S  
Save Package.  
Ctrl+B  
Compile Package.  
Ctrl+I  
Install Package.  
Ins  
Add Unit to package or add a required package to another  
packages Requires  
Page.



Del Remove Package or Unit from a package.

Enter View Required Unit/Package that is currently highlighted.

## SQL Explorer

F2 Rename an Alias.

F6 Switch between the Databases/Dictionary Tab and the Definition Tab.

Ctrl+A Apply Changes.

Ctrl+D Delete an Alias.

Ctrl+M Rename an Alias.

Ctrl+N Create a New Database Alias.

Ctrl+P Previous View.

Ctrl+R Refresh.

Ctrl+S Copy an Alias and its settings to a new alias. (Save Foo as NewName)...

Alt+F10 Executes a Right Click over the selected Item.

Alt+BckSpC Cancel.

ArrowKeys Use the Arrow Keys to navigate through the Alias List.

Pressing the Right Arrow Key while an Alias is highlighted will attempt to connect to the associated Database.

Reference:

None

4/22/99 12:32:37 PM

Last Modified: 01-SEP-99

## Article #19550: Installing and Using Delphi 3 Under NT4

Technical Information Database

TI4550D.txt - Installing and Using Delphi 3 Under NT4

Category :Install Issues  
Platform :Windows NT  
Product :Delphi3.x,

### Description:

This document applies to people who wish to install Delphi 3 on an NT4 system and have multiple users use that Delphi 3 installation on that machine. For instance, you might want to install Delphi 3 as USER1, and at a later time, USER2 wants to use Delphi 3 on the same machine, and not have to log in as USER1.

It should be noted that Delphi 3 was never designed to do this, and Borland makes no claims that it will work for all users on all machines. If this document fails in your case, go with the suggestion that always applies - install Delphi 3 to the same location as both USER1 and USER2. (Of course, you'll still need all the rights mentioned in this document even in the latter case.)

There's no need to be logged in as Administrator on NT4 in order to install Delphi 3 on the system.

You will however need rights to do a lot of things, including:

- Writing shared DLLs and OCXs to NT4's System32 directory (usually C:\WinNT\System32)
- Writing registry entries to HKEY\_LOCAL\_MACHINE
- Writing registry entries to HKEY\_CURRENT\_USER
- Writing registry entries to HKEY\_CLASSES\_ROOT (part of registering OCXs)

If you're missing one of these rights, the installation will fail.

If you install as USER1, and USER2 wants to use Delphi 3, there are a few things USER2 will need:

- Access to C:\WinNT\System32
- Access to HKEY\_LOCAL\_MACHINE, specifically the Software\Borland registry keys
- Access to HKEY\_CLASSES\_ROOT
- Access to all Delphi 3 directories and subdirectories

On top of this you need to do this:

- Copy (or move) the "Borland Delphi 3" group that was put in USER1's Start Menu to the USER2's Start Menu (or to All Users).
- Export all the Borland keys from USER1's registry under HKEY\_CURRENT\_USER\Software\Borland and import them to USER2's HKEY\_CURRENT\_USER\Software\Borland.

Reference:

None

4/1/99 11:47:18 AM

Last Modified: 01-SEP-99

## Article #19555: Why a Network Install of Delphi 3 Isn't Supported

Technical Information Database

TI4555D.txt - Why a Network Install of Delphi 3 Isn't Supported

Category :Install Issues  
Platform :All-32Bit  
Product :Delphi3.x,

### Description:

Delphi 3 was never designed to be installed on a network, and Borland makes no claims that a network install of Delphi 3 will work.

It may work for some users if they from all machines install Delphi 3 to the same location on the network. They will also need a lot of rights, including:

- Writing shared DLLs and OCXs to NT4's System32 directory (usually C:\WinNT\System32)
- Writing registry entries to HKEY\_LOCAL\_MACHINE
- Writing registry entries to HKEY\_CURRENT\_USER
- Writing registry entries to HKEY\_CLASSES\_ROOT (part of registering OCXs)

Delphi 3 expects all shared DLLs, packages and OCXs to be locally installed.

Furthermore, Delphi 3 expects all registry entries to be local.

As you can see, many problems may arise when trying to install Delphi 3 on a network. Possibly the most important one is the network software and hardware itself. Unforeseen problems may occur with any network configuration.

### Reference:

None

4/1/99 11:47:37 AM

Last Modified: 01-SEP-99

## Article #19611: How to debug ISAPI dll's on Windows 98 with Personal Web Server

Technical Information Database

FAQ: TI4611D - How to debug ISAPI dll's on Windows 98 with Personal Web Server

Category: Internet/WEB

Platform: Windows 98

Product: All32Bit,

Description:

This document describes how to debug ISAPI dll's under Win98 using Microsoft Personal Web Server (PWS). The directories referenced in this document assume a default installation of PWS. You may have to change the directories if you did not do a default installation.

1. First, create a shortcut to PWS.EXE. By default, it is located at:

```
"C:\WINDOWS\SYSTEM\inetsrv\pws.exe"
```

Right click on the new shortcut, and change the target to be:

```
"C:\WINDOWS\SYSTEM\inetsrv\pws.exe /stop"
```

This shortcut will allow you to stop PWS and unload any dll's from memory that are currently being used.

2. In your Delphi ISAPI dll project, select "Run->Parameters" and set the host application to be:

```
"C:\WINDOWS\SYSTEM\inetsrv\inetinfo.exe"
```

and the parameters to be: "-e w3svc".

3. In your Delphi ISAPI dll project, go to "Project->Options..." and on the "Directories/Conditionals" tab, set the "Output Directory" to be "c:\inetpub\scripts" (or whatever your scripts directory is).

Now to actually be able to debug your dll, you must first make sure PWS is not running. Right click on the PWS icon in the system tray, and click "Stop Service". The PWS icon should now have a red X on it. Now, double click the shortcut that was made in step one to completely unload

PWS. It may take a couple of seconds, but the PWS icon should disappear completely.

You will now be able to set breakpoints in your ISAPI dll project. Running your project will start PWS and place the PWS icon in your system tray. Right click on and select "Start Service". Access your ISAPI dll in a web browser by typing in `http://localhost/scripts/Project1.dll`. Delphi should now say "Running..." in the caption and will stop on breakpoints in your DLL code.

After you have finished debugging, to return to Delphi you must unload the dll. Right click on the PWS icon in the system tray and click "Stop Service". Then, double click on the shortcut you made in step 1. It may take a second, but Delphi will pop back up.

On a side note, I don't recommended using "Run->Program Reset (Ctrl-F2)"

while debugging ISAPI dll's under PWS. Quite often, this will cause PWS to crash, and it will require a restart of your computer to allow it to run again.

Reference:

None

6/21/99 2:09:21 PM

Last Modified: 01-SEP-99



## Article #19646: How to register my program's file type with the Window's registry.

Technical Information Database

FAQ: TI4646D - How to register my program's file type with the Window's registry.

Category: Windows API

Platform: All-32Bit

Product: All32Bit,

Description:

There are several different ways to do this. One way would be to create a .reg file that contains the necessary registry keys, and then merge it with the registry. Another way is to do it with code using TRegistry. This TI describes how to do it with code but it could easily be extended to just a .reg file if need be.

```
////////////////////////////////////
```

```
uses Registry, ShlObj;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
const
```

```
  cMyExt = '.abc';
```

```
  cMyFileType = 'Project1.FileType';
```

```
var
```

```
  Reg: TRegistry;
```

```
begin
```

```
  Reg := TRegistry.Create;
```

```
  try
```

```
    // Set the root key to HKEY_CLASSES_ROOT
```

```
    Reg.RootKey := HKEY_CLASSES_ROOT;
```

```
    // Now open the key, with the possibility to create  
    // the key if it doesn't exist.
```

```
    Reg.OpenKey(cMyExt, True);
```

```
    // Write my file type to it.
```

```
    // This adds HKEY_CLASSES_ROOT\.abc\(\Default) = 'Project1.
```

```
FileType'
```

```
    Reg.WriteString('', cMyFileType);
```

```
    Reg.CloseKey;
```

```
    // Now create an association for that file type
```

```

Reg.OpenKey(cMyFileType, True);
// This adds HKEY_CLASSES_ROOT\Project1.FileType\Default
//   = 'Project1 File'
// This is what you see in the file type description for
// the a file's properties.
Reg.WriteString('', 'Project1 File');
Reg.CloseKey;
// Now write the default icon for my file type
// This adds HKEY_CLASSES_ROOT\Project1.FileType\DefaultIcon
//   \Default = 'Application Dir\Project1.exe,0'
Reg.OpenKey(cMyFileType + '\DefaultIcon', True);
Reg.WriteString('', Application.ExeName + ',0');
Reg.CloseKey;
// Now write the open action in explorer
Reg.OpenKey(cMyFileType + '\Shell\Open', True);
Reg.WriteString('', '&Open');
Reg.CloseKey;
// Write what application to open it with
// This adds HKEY_CLASSES_ROOT\Project1.FileType\Shell\Open
\Command
//   (Default) = '"Application Dir\Project1.exe" "%1"'
// Your application must scan the command line parameters
// to see what file was passed to it.
Reg.OpenKey(cMyFileType + '\Shell\Open\Command', True);
Reg.WriteString('', '"' + Application.ExeName + '" "%1"');
Reg.CloseKey;
// Finally, we want the Windows Explorer to realize we added
// our file type by using the SHChangeNotify API.
SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, nil, nil);
finally
    Reg.Free;
end;
end;
////////////////////

```

Reference:

None

6/21/99 2:10:19 PM

Last Modified: 01-SEP-99

## Article #20193: Changing BDE Driver Settings Programmatically

This document illustrates how to change BDE driver settings programmatically (settings found under “Configuration->Drivers” in the BDE Administrator only). You may need to change BDE driver settings programmatically if, for example, you want to specify the DLL32 setting for the Oracle or MSSAccess driver (to specify oracle 7 or 8, MSAccess 95 or 97), or you want to increase the value of Max DBProcesses for the MSSQL driver.

Note: If you need to change the parameters for anything that can be found on the “Database” tab in BDE administrator, then you should use Tdatabase.Params (not covered in this document).

A BDE driver can be configured at runtime using Tsession.ModifyDriver. The following code changes the Max DBProcesses driver parameter for the MSSQL driver.

Start a new application, drop a Tsession on the form, set autosessionname to true, and add the following code into the OnCreate event handler for the form:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  DriverParams: TStringList;
begin
  DriverParams := TStringList.Create;
  try
    { add the parameters that you want to change only. The default for
      this param is 31. }
    DriverParams.Add('MAX DBPROCESSES=50');

    { call modify driver with name of driver and list of params }
    session1.ModifyDriver('MSSQL',DriverParams);

    { call this if you want to save the changes in the BDE for all
      applications to use, will only take affect after all BDE applications
      have been unloaded on the system. }
    session1.SaveConfigFile;
  finally
    DriverParams.Free;
  end;
end;
```

Be sure to read the documentation for Tsession.ModifyDriver in the online help files-> especially about Tsession.ConfigMode and the scope of ModifyDriver.

## Article #20245: MS SQL Server 7 Stored Procedure PRINT Statements with ADO

MS SQL Server stored procedures use a number of ways of communicating information back to the user. One of these is the PRINT statement. This statement is typically used for informational type messages. If the intention is to raise an exception in the calling application, it is preferable to use the RAISERROR statement.

Many users want to use the PRINT statement, however, and it has not been possible to access informational type messages with the BDE - only RAISERROR was supported, and then only those errors of sufficient severity to raise an exception.

With ADO it is possible to access PRINT statement messages. ADO considers these messages to be errors of severity 0 - too low to raise an exception - but nevertheless the messages are written to ADO's errors collection, which, in Delphi, is a property of the TADOConnection component.

Another fact worth pointing out is that Print statement messages are only placed in the Errors collection if the ADO command is executed with the eoExecuteNoRecords Option set to True.

How do we retrieve this message in practice?

Consider the following Stored Procedure:

```
create PROCEDURE sp_show_test (@param1 char(30)) as
begin
  If (@param1 = "Some Value")
    Select * from dbo.authors
  else
  begin
    print "Illegal value : "+@param1
    return 25
  end
end
```

This will return a result set if the input parameter is set to "Some Value", otherwise it will issue the "Illegal Value" message and return a user defined return code of 25.

How do we deal with this in Delphi? If we just use a TADODataset component, we risk getting an exception raised when a result set is not returned by the stored procedure. It is therefore better to use a TADOCommand component AND a TADODataset. The Execute method of a TADOCommand can return an ADO \_RecordSet object, which we can assign to the RecordSet property of the TADODataset. If the stored procedure does not return a result set, the \_RecordSet is still returned but in a closed state, which we can test for.

Here is a simple Delphi unit that allows us to see the Print statement's message. In addition to the three ADO components listed above this code requires a TEdit and TButton, and 'ADOInt' should be added to your uses clause:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyReturn : Integer;
  looper : Integer;
  MyRecordSet : _RecordSet;
begin
  ADOCommand1.Prepared := True;
```

```

ADOCCommand1.Parameters.ParamByName('@param1').Value := Edit1.Text;
MyRecordSet := ADOCCommand1.Execute;
if MyRecordSet.State = adStateOpen then
begin
    ADODataSet1.Recordset := MyRecordSet;
    ADODataSet1.Open;
end
else
begin
    ADOCCommand1.ExecuteOptions := [eoExecuteNoRecords];
    ADOCCommand1.Execute;
    MyReturn := ADOCCommand1.Parameters.ParamByName('RETURN_VALUE').Value;
    for looper := 0 to ADOCConnection1.Errors.Count - 1 do
    begin
        ShowMessage(Format('Stored Procedure %s Failed with Return Code %d and Message
"%s"',
                            [ADOCCommand1.CommandText, MyReturn,
                            TrimRight(ADOCConnection1.Errors[looper].Description)]));
    end;
end;
end;

```

As mentioned, when we execute the procedure the first time, we are expecting a result set to be returned. This will mean that the Errors collection will not be filled. If the result set has not been returned, we must execute the procedure a second time with the eoExecuteNoRecords option set.

Last Modified: 17-DEC-99

## Article #20377: Maintaining State in Web Applications

# Advanced Web Applications

## Maintaining State in Web Applications

By Corbin Dunn

There are several ways to maintain state in a web application. Take a look at DavidI's document about maintaining state (<http://community.borland.com/article/0,1410,10265,00.html>) for some pro's and con's of each method. In this document, I will describe ways to use each method and provide an example.

### Maintaining state with cookies

**The first way**, and one of the most common ways, is to maintain state with cookies. This document was written with Delphi 5 in mind, but it should work fine in previous versions.

Cookies are stored in a file on your client machine and can be set and retrieved by a Web Application program with the help of a web browser. Cookies are sent as part of the HTML request/response header so it is transparent to the user. A cookie has a Name and Value pair, an expiration date, a domain name that the cookie is valid for, and a set of URLs that the cookie is valid for. In addition, you can set the Secure property of a cookie to only allow it to be sent and received via a secure connection (typically, HTTPS).

### Delphi Example

This best way to learn is to start up Delphi and create a new Web Application (this requires the Enterprise version of Delphi). In the new web module, add a single action and make it the default action (set Default to True). Now, add the following code in the OnAction event:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
  strVisitNum: string;
begin
  // This is the main "entry" action.
  // This example will count the number of times a person
  // has been to this page using a cookie.

  strVisitNum := '1'; // Initialize the visitor number to 1

  // First check to see if there is a cookie present
```

```

if Request.CookieFields.Count > 0 then
begin
    // See if the 'Visit Number' cookie is present
    strVisitNum := Request.CookieFields.Values['Visit Number'];
    if strVisitNum <> '' then
        begin
            try
                // Try to convert it to an integer and inc it
                strVisitNum := IntToStr(StrToInt(strVisitNum) + 1);
            except // Eat exceptions (conversion exceptions)
            end;
        end
    end;

    // Set the 'Visit Number' cookie
    with Response.Cookies.Add do
        begin
            // You should set the domain to your domain to
            // get only cookies sent back to that domain (yours).
            // I'm leaving it commented out because in testing
            // you usually reference your machine by http://machinename/scripts
            // with no domain, and you can test this app by just compiling.
            // Domain := 'inprise.com';

            // Expire (quit sending the cookie) in one month from now
            Expires := IncMonth(Now, 1);
            // Limit the cookie to the path to this DLL
            Path := Request.URL; // Such as '/scripts/StateKeeper.dll'
            Secure := False;
            // The most important parts of the cookie are the Name and Value
            Name := 'Visit Number';
            Value := strVisitNum;
        end;

        Response.Content := '<html><body>Welcome!<br>You have visited ' +
            strVisitNum + ' times!</body></html>';
    end;

```

When receiving cookies, the main thing that you will be looking at is `Request.CookieFields.Values['Cookie Name'];`. If the cookie is not present, the returned string will be blank.

When setting a cookie, the most important things to set are the Name and Value properties.

## Maintaining state with Hidden Fields

In this next example, we will see how to save state with forms using the POST method and hidden fields.

Conceptually, you can send a custom form to the user based on previous selections. You can also maintain state with hidden fields. This example does both. It allows the user to select an option, and based on what option they selected, it will set it as selected in the resulting page. In addition, it keeps track of the number of times they have gone through that form by saving a counter in a hidden field.

### Delphi Example

To build this example, simply add another action in your web module and change the PathInfo to be any string (I chose '/hidden'). Then, access the page with: <http://MyMachineName/scripts/DllName.dll/hidden> (or something similar based on the names you used).

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  nOptionSelected: Integer;
  strOptionString: string;
  strVisitNum: string;
begin
  // Hidden field example.

  // The first thing we do is check to see what method this
  // action was invoked by. If it was from a GET method,
  // we will return a form for the user to use and send
  // a selected option.
  // In addition, we will maintain "state" of how many visits
  // the user has made to the page with a hidden field.

  if Request.MethodType = mtGet then
  begin
    // Send a form back as the response so the user can
    // invoke this same action with it (with a POST method)
    Response.Content := '<html><body>' + #10#13 +
      '<form action="' +
      Request.URL + Request.PathInfo +
      '" method="post">' +
      'Test saving state! Select '
      +
      'something for me to remember: ' +
      '<select name="selection">' +
      '<option value="1">First Option' +
```



```
'<option value="2">Second Option' +  
'<option value="3">Third Option' +  
'<option value="4">Forth Option' +  
'</select>' +  
// Notice the hidden field below to  
// keep track of the number of "visits"  
'<input type="hidden" ' +  
'name="Visit Number" value="1">' +  
'<input type="submit">' +  
'</form></body></html>';
```

```
end  
else if Request.MethodType = mtPost then  
begin  
    // On a post, data is sent to us in the ContentFields  
    // string list.  
    try  
        nOptionSelected :=  
            StrToInt(Request.ContentFields.Values['selection']);  
    except // Catch conversion exceptions  
        nOptionSelected := -1;  
    end;  
    // Set the selected option based on which one was sent  
    // in the post data.  
    case nOptionSelected of  
        1: strOptionString := '<option value="1" selected>First Option' +  
            '<option value="2">Second Option' +  
            '<option value="3">Third Option' +  
            '<option value="4">Forth Option';  
        2: strOptionString := '<option value="1">First Option' +  
            '<option value="2" selected>Second Option' +  
            '<option value="3">Third Option' +  
            '<option value="4">Forth Option';  
        3: strOptionString := '<option value="1">First Option' +  
            '<option value="2">Second Option' +  
            '<option value="3" selected>Third Option' +  
            '<option value="4">Forth Option';  
        4: strOptionString := '<option value="1">First Option' +  
            '<option value="2">Second Option' +  
            '<option value="3">Third Option' +  
            '<option value="4" selected>Forth Option';  
    else // None selected (some error occurred if this is executed)  
        strOptionString := '<option value="1">First Option' +  
            '<option value="2">Second Option' +  
            '<option value="3">Third Option' +
```

```

                                '<option value="4">Forth Option';
end;

// Now try to inc the Visit Number
try
    // Try to convert it to an integer and inc it
    strVisitNum := Request.ContentFields.Values['Visit Number'];
    strVisitNum := IntToStr(StrToInt(strVisitNum) + 1);
except // Eat exceptions (conversion exceptions)
    strVisitNum := '1';
end;

Response.Content := '<html><body>' + #10#13 +
                    '<form action="' +
                    Request.URL + Request.PathInfo +
                    '" method="post">' +
                    'You selected option number ' +
                    Request.ContentFields.Values['selection'] +
                    ' and this is vist number ' +
                    strVisitNum + '<br>' +
                    'You can select another: ' +
                    '<select name="selection">' +
                    strOptionString +
                    '</select>' +
                    // Add the hidden field back in
                    '<input type="hidden" ' +
                    'name="Visit Number" value="' +
                    strVisitNum + '>' +
                    '<input type="submit">' +
                    '</form></body></html>';

end
else // An "Unsupported" MethodType
    Response.Content := '<html><body>Method not supported</body></html>';
end;

```

One of the key things to notice here is the line:

```
'<input type="hidden" name="Visit Number" value="' + strVisitNum + '>'
```

that adds the hidden field to keep track of the number of visits.

This example is probably one of the more elegant ways of saving state. You could easily extend it to database browsing. Instead of saving the "Visit Number" you could save the current record (or range) that the user was viewing.

The third way to maintain state is with "Fat URLs" like the following which shows how a search on HotBot can be saved: <http://hotbot.lycos.com/?MT=Corbin%27s+treehouse>

You can easily implement this in Delphi, and because of its simplicity it is one of the more common ways of saving state that is easily 'bookmarkable.'

The first thing we are going to do is take a closer look at a GET request (aka, Fat URL). Typically, they will look something like:

<http://www.mydomain.com/scripts/myIsapi.dll/PathInfo?name1=value1&name2=value2>

The first part (<http://www.mydomain.com/scripts/myIsapi.dll/PathInfo>) contains the usual information; the domain name, followed by the path to the ISAPI dll (or CGI), and then the PathInfo that specifies which action to invoke. Immediately after that is a **question mark** that signifies the start of the query data, or Fat URL. It is then followed by **Name=Value** pairs separated by ampersands.

In Delphi, the way you access these values is with the `Request.QueryFields.Values[ 'Name' ]` object which returns the value associated with a given Name.

Something to be aware of is that the web browser has to URL encode the data sent via the GET method. If you use a form to send the data with a GET method (as which is done in my example below), then you don't have to worry about anything. If you want to construct a Fat URL in code, you have to be sure to URL encode any possible Name and Value pairs. In Delphi, with the NetMaster's TNMURL component, this is a piece of cake. Here is an example:

```
function Encode(const str: string): string;
begin
    NMURL1.InputString := str;
    Result := NMURL1.Encode;
end;

var
    strFatURL: string;
begin
    strFatURL := 'http://MyMachine/scripts/MyIsapi.dll/Path?' +
        Encode('Query Name Is')
        + '=' +
        Encode('Corbin''s Tree House')
        + '&' +
        Encode('&Another Name')
        + '=' +
        Encode('Here"s The Result Value');
end;
```

Notice the small helper function "Encode" that encodes a given string with the TNMURL component and returns the encoded string. After execution, strFatURL contains: <http://MyMachine/scripts/MyIsapi.dll/Path?Query+Name+Is=Corbin%27s+Tree+House+%26Another+Name=Here%22s+The+Result+Value> which is a properly URL

encoded.

## Delphi Example

Now it is time to add some code to the Delphi example we have been working on. Add another action to your web module and give it a new PathInfo string, such as '/get' (since GET is the method which a Fat URL is invoked by).

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;  
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
var  
    strVisitNum: string;  
begin  
    // Fat URL example (using the GET request method).  
  
    // When a web browser sends a Fat URL to the web server, you have  
    // to access the data sent via the Request.QueryFields, instead  
    // of the Request.ContentFields that are filled in with the POST  
    // method.  
  
    // First, check to see if the URL has any data sent in the Query  
    if Request.QueryFields.Count <= 0 then  
    begin  
        // Set the response to an HTML form with the GET method  
        Response.Content :=  
            '<form action="' +  
            Request.URL + Request.PathInfo +  
            '" method="get">' +  
            'Test saving state with Fat URL's!<br>' +  
            '<input type="submit" name="counter" value="1">' +  
            '</form></body></html>';  
    end  
    else  
    begin  
        // Inc the counter and send back the count number  
        try  
            strVisitNum := Request.QueryFields.Values['counter'];  
            strVisitNum := IntToStr(StrToInt(strVisitNum) + 1);  
        except // Eat exceptions (conversion exceptions)  
            strVisitNum := '1';  
        end;  
        Response.Content :=  
            '<form action="' +  
            Request.URL + Request.PathInfo +  
            '" method="get">' +  
            'Test saving state with Fat URL's!<br>' +  
            '<input type="submit" name="counter" value="' +
```

```
strVisitNum + '>' +  
'</form></body></html>';  
end;  
end;
```

You will probably notice that the URL becomes something like: <http://MyMachine/scripts/MyIsapi.dll/get?counter=23>, which means the user can very easily change the URL and the underlying request that is sent to the Web Application when a Fat URL is sent with the GET method. Of course, it is possible to do this with a POST method too, but it is more difficult because the user has to create an HTML form that sends the custom data to a Web Application.

This is just something to keep in mind. If you try to put [counter=1.123](http://MyMachine/scripts/MyIsapi.dll/get?counter=1.123) in the URL, and didn't catch the conversion exception like I do, your application would show an exception to the end user (which isn't very pretty).

You should now have a good understanding of how to save state in a Web Application.

**[Download the complete source code for this web application.](#)**

Last Modified: 17-DEC-99

## Advanced Web Applications

### Streaming Images from Web Applications

By Corbin Dunn

You may have had an instance when you wanted to stream a custom image out to a web page based on what the user selected. This document describes several different ways to store and send images to a web browser.

[Download this whole project.](#)

#### Sending a jpeg image from the hard drive

**This tutorial** was created with Delphi 5, but it should work fine with previous versions. You will need the Enterprise (or Client/Server) version of Delphi to create Web Applications.

#### Delphi Example

Create a new Web Application (File->New->Web Server Application, ISAPI DLL) and add a new Action. Set the PathInfo to `/motorcycle.jpg` and in the OnAction event add the following code:

```
procedure TWebModMain.WebModMainwbactnGetJpegAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);

  // Helper function to get the current directory the DLL is in
function GetDllDir: string;
begin
  SetLength(Result, MAX_PATH+1); // Add 1 for the null character
  GetModuleFileName(hInstance, PChar(Result), MAX_PATH+1);
  SetLength(Result, Length(PChar(Result)));
  Result := ExtractFilePath(Result);
end;

var
  FileStream: TFileStream;
begin
  // This demonstrates returning a Jpeg file from a file on the hard drive.
  try
    FileStream := TFileStream.Create(GetDllDir + 'motorcycle.jpg', fmOpenRead);
    // Note that the file 'motorycle.jpg' must be in the same
    // directory as this DLL.
    FileStream.Position := 0; // Go to the start of the stream
    Response.ContentStream := FileStream;
    Response.ContentType := 'image/jpeg';
    Response.SendResponse;
    // Notice that the stream is not freed (it shouldn't be!)
  except // Catch all exceptions and show a custom message
    on E: Exception do
```

```

Response.Content := '<html><body>An error occurred ' +
  E.ClassName + ': ' + E.Message + '</body></html>';
end;
end;

```

---

The first thing you will see is the helper function `GetDllDir` that simply get's the current directory the DLL resides in. You will have to put the file '[motorcycle.jpg](#)' in the same directory as your ISAPI dll for this to work correctly.

Something to notice is how the whole function is wrapped in a `try..except` so that I can show a custom error message if something goes wrong. The `TFileStream.Create` can potentially raise an exception if the file passed to it doesn't exist or is locked.

Next, the `Response.ContentStream` is set to the `FileStream` we just created. Once you do this note that you **should not free the stream**. It is taken care of for you in the `TWebResponse` destructor.

Then the `Response.ContentType` is set based on what we are sending (a Jpeg in this case), and the `Response` is sent with `Response.SendResponse`.

One thing which you may want to do is send other file types. A problem you may encounter is what to set the `Response.ContentType` to. This can easily be solved in code by looking up the `ContentType` in the registry:

```

// MimeType is a string
Reg := TRegistry.Create;
try
  Reg.RootKey := HKEY_CLASSES_ROOT;
  if Reg.OpenKey(ExtractFileExt(FileName), False) then
    begin
      if Reg.ValueExists('Content Type') then
        MimeType := Reg.ReadString('Content Type');
      Reg.CloseKey;
    end;
  finally
    Reg.Free;
  end;

```

Of course, if the `ContentType` wasn't found, you will probably want to default to something such as 'text/plain' (plain text).

Another thing which you may be interested in doing is setting the "suggested file name" when a user goes to download a file by sending back a response other than an image. This can easily be done by adding a custom `Content-Disposition` header field:

```

Response.CustomHeaders.Add('Content-Disposition: ' +
  'filename="Suggested Name.ext"');

```

If you would like to know more about what you can do with the `Content-Disposition` header, take a look at the RFC: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec19.html>

At one point I wanted to create an ISAPI dll that was completely self contained with its own images. All I had to output was a simple checked or unchecked image, so I decided a simple way to solve this problem was to put the GIF images in the resource portion of the DLL.

The first thing I did was to create a text file named "web\_gifs.rc". Inside it, I had the lines:

```
CHECKED GIF "checked.gif"  
UNCHECKED GIF "unchecked.gif"
```

The two gif files were in the same directory as the "web\_gifs.rc" file, and I compiled it into a resource file (.res) with a command line of:

```
brcc32 web_gifs.rc
```

For ease of use, I included a simple batch file that does this for you. I then added two new Actions to my project: one with a PathInfo of '/unchecked.gif' and another with a PathInfo of '/checked.gif'. I then created an OnAction event, and hooked them both to the same event with the following code in the event:

---

```
// Link the Gif's into the DLL resource section  
{$R web_gifs.RES}
```

```
procedure TWebModMain.WebModMainwbactnGetGifAction(Sender: TObject;  
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
var  
  ImageType: string;  
begin  
  try  
    // Two Web Actions are hooked up to this single OnAction.  
    // Based on which PathInfo is in the Action, we send the  
    // correct Gif image.  
    if (Sender as TWebActionItem).PathInfo = '/unchecked.gif' then  
      ImageType := 'UNCHECKED' // Resource name in .RES file  
    else  
      ImageType := 'CHECKED';  
    Response.ContentStream := TResourceStream.Create(hInstance, ImageType, 'GIF');  
    Response.ContentType := 'image/gif';  
    Response.SendResponse;  
  except  
    on E: Exception do  
      Response.Content := '<html><body>An error occurred ' +  
        E.ClassName + ': ' + E.Message + '</body></html>';  
  end;  
end;
```

---

Based on the previous example, this one should be very easy to understand.



The next challenge was to dynamically create a custom image at run-time and return it to the user. Conceptually, the way this will be done is to create a TBitmap, draw on that Bitmap's Canvas, and then save that Bitmap to a TJpegImage. The JpegImage can then be streamed out to the web browser as before.

---

### Delphi Example

I created another WebAction and set the PathInfo to '/custom.jpg'. The OnAction event is as follows:

```
procedure TWebModMain.WebModMainwbactnGetOnFlyAction(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Bitmap: TBitmap; // The unit Graphics must be in your uses clause
  JpegImage: TJpegImage; // The unit JPEG must be in your uses clause
  MemoryStream: TMemoryStream;
  I: Integer;
begin
  // This demonstrates creating a custom image on the fly,
  // converting it from a Bitmap to a Jpeg, and sending the
  // Jpeg to the browser.
  try
    Bitmap := TBitmap.Create;
    try
      // Draw some custom stuff on the bitmap
      Bitmap.Width := 300;
      Bitmap.Height := 300;
      for I := 0 to Bitmap.Height - 1 do
        begin
          // Add a little color
          Bitmap.Canvas.Pen.Color := Trunc(I/Bitmap.Height*255);
          Bitmap.Canvas.MoveTo(0, I);
          Bitmap.Canvas.LineTo(Bitmap.Width - 1, I);
        end;
      // Set the background mode to be transparent when writing text.
      SetBkMode(Bitmap.Canvas.Handle, TRANSPARENT);

      // Write a custom message based on some of the Request properties
      Bitmap.Canvas.Font.Color := clWhite;
      Bitmap.Canvas.TextOut(5, 5, 'Hello Host ' + Request.RemoteHost +
        ' (' + Request.RemoteAddr + ')');
      Bitmap.Canvas.TextOut(5, 30, 'You are using ' + Request.UserAgent);
      Bitmap.Canvas.TextOut(5, 55, 'The date/time is ' + DateTimeToStr(Now));

      JpegImage := TJPEGImage.Create;
      try
        // Assign the Bitmap to the Jpeg
        JpegImage.Assign(Bitmap);
        MemoryStream := TMemoryStream.Create;
        JpegImage.SaveToStream(MemoryStream);
        MemoryStream.Position := 0;
        Response.ContentStream := MemoryStream;
        Response.ContentType := 'image/jpeg';
        Response.SendResponse;
```

```

    finally
        JpegImage.Free;
    end;
finally
    Bitmap.Free;
end;
except
    on E: Exception do
        Response.Content := '<html><body>An error occurred ' +
            E.ClassName + ': ' + E.Message + '</body></html>';
    end;
end;

```

---

A new thing to notice in this example is the saving the Jpeg to a TMemoryStream. The Jpeg and Bitmap can safely be freed, but the MemoryStream is automatically freed for you in the TWebResponse's destructor.

## Putting it all together

When you want to display any of the images, all you have to do is type in the path to the DLL and add the right PathInfo. More commonly, you will probably embed it into an <IMG> tag, such as: 

The last thing to do in this example is to put it all together. I created one more action and set the Default property to True. In the OnAction event I added the following code:

```

procedure TWebModMain.WebModMainwbactnDefaultAction(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
    // Default action, display several different images
    // that were streamed out by the ISAPI dll
    Response.Content := '<html><body>' + #10#13 +
        '<h2>Below are some images sent out from this Web Application</h2>' +
        '<b>Here is a Gif that was streamed out from a resource inside ' +
        'the Dll: checked  and unchecked: ' +
        '<br><br>' +
        'Here is a Jpeg that was loaded from a file:<br>' +
        '<br><br>' +
        'Here is a custom image created on the fly:<br>' +
        '' +
        '</b></body></html>';
end;

```

## Article #20385: How can I install Delphi on a Micron computer?

### How can I install Delphi on a Micron computer?

It seems that the proprietary OS that is shipped with Micron conflicts with Delphi as well as some of the hardware drivers. Compiled here are a number of solutions that seem to resolve the issue.

1. Uninstall/Reinstall: Make sure that any directories and/or registry keys that refer to Delphi have been removed during the uninstall process. Before reinstalling, disable all software running in the background i.e. anti-virus applications. If using Win98, it is a good idea to turn down the Hardware Acceleration to none before reinstalling. If using Win95 boot into safe-mode when reinstalling, and for WinNT 4.x boot into VGA-mode.
2. Hard drive install: Copy the Runimage and Install directories from the Delphi CD to the root of your hard drive. Next, copy all the loose files at the root of the CD to the root of your hard drive. Make sure that the Runimage and Install directories are the same size as the ones on the CD. After everything is copied over, but before you install, read the above Uninstall/Reinstall section.
3. If you are using a proprietary OS from Micron, and reinstalling a non-proprietary OS is out of the question, make sure you have all the latest updates. Especially for the CD-ROM, video card drivers and system files. The Micron web-site has a number of drivers, however you should go to the video card/CD-ROM vendors sites first. You may need to go to the Microsoft site also. Once you have all the updates and drivers you may need to delete the old drivers and reboot so the OS auto-detects the latest drivers for you.
4. Network: If you have access to other non-Micron computers over a network you may need to install from one of those machines. There are a few ways of doing this. Using the non-Micron's CD-ROM, run the setup.exe from the non-micron computer and choose a directory on the Micron computer to install into. Or Map a drive to the non-micron's CD-ROM and install from there. Or use the same procedure as a hard drive install using the non-micron's hard drive. Then map a drive to where you copied over the files and install from there.
5. Reinstalling a non-proprietary version of your OS. This is a worst case last effort scenario, but it has been a solution when none of the above worked.



## Article #20450: Adding TextCompletion to a TComboBox

### Adding text completion to a TComboBox

The Netscape Communicator location box, The Windows 98 'Run' dialog, and other programs, have implemented a very user friendly feature known commonly as text completion. This document describes how to add similar functionality to a TComboBox. The most elegant and reusable way to add this functionality is by descending from TComboBox and overriding the ComboWndProc to handle the WM\_KEYUP message. By adding a new property 'TextCompletion', the functionality can be toggled to act like a regular TComboBox. Below is the component unit that implements text completion in a TComboBox. This unit can be installed as is.

```
unit CompletingComboBox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TCompletingComboBox = class(TComboBox)
  private
    FTextCompletion: Boolean;
    function GetTextCompletion: Boolean;
    procedure SetTextCompletion(const Value: Boolean);
  protected
    // override the WndProc() so that we can trap KeyUp events.
    procedure ComboWndProc(var Message: TMessage; ComboWnd: HWND;
      ComboProc: Pointer); override;
  public
    { Public declarations }
  published
    property TextCompletion: Boolean read GetTextCompletion write SetTextCompletion;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Standard', [TCompletingComboBox]);
end;

{ TCompletingComboBox }

function TCompletingComboBox.GetTextCompletion: Boolean;
begin
  Result := fTextCompletion;
end;
```

```

procedure TCompletingComboBox.SetTextCompletion(const Value: Boolean);
begin
    fTextCompletion := Value;
end;

procedure TCompletingComboBox.ComboWndProc(var Message: TMessage; ComboWnd: HWND;
    ComboProc: Pointer);
var
    rc, len: Integer;
begin
    inherited;
    case Message.Msg of
        WM_KEYUP:
            begin
                // test to see if its a character that should not be processed.
                if (Message.WParam <> 8) and (Message.WParam <> VK_DELETE) and
                    (Message.WParam <> VK_SHIFT) and (fTextCompletion = True) then
                    begin
                        // Use CB_FINDSTRING to locate the string in the Items property
                        rc := Perform(CB_FINDSTRING, -1, Integer(PChar(Caption)));
                        // if its in there then add the new string to the Text and
                        // select the portion that wasn't typed in by the user
                        if rc <> CB_ERR then
                            begin
                                // store the length of the current string
                                len := Length(Text);

                                // set the new string
                                ItemIndex := rc;

                                // highlight the rest of the text that was added.
                                SelStart := len;
                                SelLength := Length(Text) - len;

                                // return 0 to signify that the message has been handled.
                                Message.Result := 0;
                            end;
                        end;
                    end;
            end; // case
    end;
end.

```

Last Modified: 11-JAN-00

## Advanced Web Applications

### Redirecting a request to another URL or HTML page

By Corbin Dunn

[Download the entire Delphi 5 project for this document.](#)

At some point, you may want to redirect a user request to another URL or HTML page on your server. This is quite easy to do with the following code inside your `OnAction` event:

```
Response.SendRedirect('http://www.borland.com');
```

One question that you may ask is exactly what this does. The easiest way to find out is to look at the full headers that the server returns:

```
HTTP/1.1 302 Object Moved
Location: http://www.borland.com
Server: Microsoft-IIS/5.0
Content-Type: text/html
Content-Length: 145
```

```
<head><title>Document Moved</title></head>
<body><h1>Object Moved</h1>
This document may be found
<a HREF="http://www.borland.com">here</a></body>
```

But how did I produce this output? It is easy to "spy" on the headers returned by a webserver by simulating a GET request just like a web browser would make:

### Simulating an HTTP GET request

Click on the Start button, and go to Run. Type in:

```
telnet support.microsoft.com 80
```

This will open up a connection to support.microsoft.com on the standard HTTP port (80), and you will see a blank telnet window.

Carefully type the following in the telnet window, noting the case:

```
GET / HTTP/1.0
```

Note that what you are typing will not be echoed back to you, so be sure to type carefully.

After you type in the request, press return twice to send the request.

Effectively, a break down what you are doing is this:

- **GET** - This is the request type. GET is the most common, and is used for getting standard web pages and images. POST is another common request type used for posting data to a web server.
- **/** - The next thing you type is a slash. This is the relative document that you want on the server. "/" means the default document. You could have easily typed something else, such as "/a\_directory/a\_file.html".
- **HTTP/1.0** - This is the HTTP version that your "browser" supports. 1.0 is the lowest, and will always work. 1.1 is also becoming a standard, but some servers may not support HTTP/1.1. You can look at the first line of the response to see what version they do support.
- **Two returns** - Two returns are used to send the request. Two are needed because some requests may span multiple lines, and the server waits for a blank line followed by another return before executing the request.

After you press the two returns, you should get some content back, such as the response headers and requested file.

The thing that we are interested in looking at is the response code:

```
HTTP/1.1 302 Object Moved
```

The first part tells us what version of HTTP that the server supports (1.1 in this case). The second part is the response code, and the third part is a server defined response message for that code.

To find out exactly what each response code means, and when you should give it, you can look at the HTTP RFC (Request For Comments): <http://www.w3.org/Protocols/rfc2068/rfc2068>

From that document, we can see that 302 means:

```
302 Found
```

```
The requested resource resides temporarily under a different URI.  
Since the redirection might be altered on occasion, the client SHOULD
```



continue to use the Request-URI for future requests. This response is only cacheable if indicated by a Cache-Control or Expires header field.

...

In other words, you should use a response of 302 when you want to redirect the user to another URL, but don't want them to have to change the link that they came from. All the proper headers are set up for you when you use:

```
Response.SendRedirect('http://www.borland.com');
```

If you want to inform the web browser that the URL has been permanently moved, you need to send a response code 301:

### 301 Moved Permanently

The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs. Clients with link editing capabilities ought to automatically re-link references to the Request-URI to one or more of the new references returned by the server, where possible. This response is cacheable unless indicated otherwise.

The new permanent URI SHOULD be given by the Location field in the response. ...

The reason for doing this is because some search engines will see a 301 response, and will update their database with the new link (since it was moved permanently, not temporarily like a 302 indicates).

Now, Delphi doesn't have an automatic way of sending a 301 Response, so we have to manually do it:

#### Sending a custom response and code

```
// This action is going to send the web browser a 301 code indicating  
// a permanent move to a new URL.  
Response.ReasonString := 'Moved Permanently';  
Response.StatusCode := 301;  
// Location is the permanent new location; add it as a custom header  
Response.CustomHeaders.Values['Location'] :=  
    'http://www.borland.com';  
Handled := True;
```

That's it! You should be able to easily extend this to send any custom response code and message.

**More related Advanced Web Application documents:**

- **Maintaining State in Web Applications**
- **Streaming Images from Web Applications**
- **Writing a fancy CGI hit counter in Delphi 5**

Last Modified: 19-JAN-00

## Article #20901: Debugging an ISAPI DLL with Delphi under NT 4.0 for IIS version 4

# How to debug an ISAPI DLL with Delphi under NT 4.0 with IIS version 4

1. Make sure that the user name you use to do all your developing with for WinNT, has Administrator rights. You need to add some settings to the user, and in order to do so the user must have Administrator rights.
2. Run the User manager for the user that is going to do the debugging. Start | Programs | Administrative Tools (Common) | User Manager. On the Menu bar select Policies | User Rights. Make sure the Show Advanced User Rights box is checked. Hit the down arrow and scroll through the listing of the different Rights, you should see what you need to add.

Give this user the following permissions:

Log on as a Service  
Act as part of the operating system  
Generate security audits

3. You will need IIS to run as a process instead of a service. To do this run the registry file labeled: Regproc.reg. This file will make the necessary changes to you registry that makes IIS run as a process. The other registry file: Regserv.reg, will undo the changes made by the previous registry file.

Run as a Process: Regproc.reg (to be able to debug run this one)

Run as a Service: Regserv.reg (this will undo the previous reg setting)

4. Now stop these services and set them to manual: WWW publishing service, FTP Service and IIS Admin.
5. Restart your computer.
6. Open Delphi and from the menu bar, select Project | Options -> Linker and select Use Include remote debugger symbols option.
7. Now in Delphi, open the DLL project and use the following parameters:

Run: C:\Winnt\system32\inetsrv\inetinfo.exe  
Parameters: -e w3svc

Last Modified: 01-MAR-00

## Article #21772: Deploying a Delphi/VisiBroker Application

This article discusses the process of deploying a Delphi application that uses CORBA. Often there is confusion over what part Delphi plays versus what part VisiBroker plays, and exactly what is needed from each. This document will attempt to clarify this.

Deploying a Delphi/CORBA app is not an exact science. Much depends on what components you are using, and what files already exist on the deployment machine. For the purpose of this article it is assumed you are deploying to a machine which contains Windows NT and nothing else. It is also assumed that you know how to create a Delphi CORBA application, and how to deploy a basic Delphi application without CORBA. When deploying a Delphi application you must use a Borland certified deployment utility such as InstallShield Express. Using this utility helps insure that the Delphi files your application needs are included in your deployment.

### Background Information

VisiBroker for Delphi uses the VisiBroker C++ ORB. The C++ ORB has been encapsulated with Delphi code. This code is contained in the library orbpas50.dll. This approach provides the Delphi community with a fast and stable industry-standard ORB and avoids the need to develop and maintain a new ORB from scratch.

Applications built with VisiBroker for Delphi require both the VisiBroker for Delphi runtime and the VisiBroker for C++ runtime. Orbpas50.dll is a Delphi wrapper around the C++ VisiBroker run time DLL (orb\_br.dll). The CORBA objects are created in the C++ ORB in the same way as they would be for a pure C++ client and a proxy for the object is maintained on the Delphi side. When a method is invoked on the proxy from a Delphi executable the proxy in turn invokes the same method on the C++ object.

### Deployment

One of the mysteries surrounding Delphi/VisiBroker deployment is who is responsible for providing what? VisiBroker states that the components necessary to deploy a VisiBroker application are a subset of their full development kit. The software associated with a deployment license is already a part of the development kit. That's fine if you are purchasing VisiBroker separately as a product, but what if you have purchased Delphi 5 Enterprise which comes with all the VisiBroker stuff you need to create and deploy an application? In this case you must still purchase a VisiBroker license, but not the development kit. They do not provide much information in their documentation on Visibroker redistribution, partly because there is no general answer. Deployment questions depend on the application and what you need to deploy. So even though you are paying for the VisiBroker license you will be getting everything else you need from Delphi.

Your deployment needs to include whatever is required to run your application. For a client, this usually involves the application executables plus the orb library (orb\_br.dll plus orbpas50.dll and cp3245mt.dll – See table below). For a server, it includes the server applications, orb library, and

optionally an osagent, OAD, and IREP servers. Osagent, OAD and IREP do not necessarily need to be running on the server, but can be running anywhere on the network where they are visible to the application. If an Object Activation Daemon (OAD) is installed then the implementation repository should be initialized using the oadutil tool. Oadutil and irep require a java vm also (this will already exist on a machine with Delphi 5). Details of OAD are not covered in this document.

VisiBroker can be installed from the Delphi CD by doing a custom install or the necessary files can be included with your deployment (see Environment below for registry settings). The default location for these files on your development machine is C:\Program Files\Borland\vbroker. For development and testing purposes a VisiBroker license is not needed. For your "real world" applications you will need to acquire a VisiBroker deployment license from Inprise. VisiBroker is licensed using a server-based licensing scheme so developing VisiBroker for Delphi applications that integrate with existing applications for which you already have a license may not require additional licenses. Consult your VisiBroker license for complete details. Also note that if you have a multi-tier MIDAS application you will also need to purchase MIDAS licenses for each MIDAS server you deploy [[more info](#)].

## Deployment summary

File Name	Client	Server	Default location (Delphi 5 installation)	Comments
orbpas50.dll	X	X	C:\Program Files\Borland\Delphi5\Bin	Delphi 5 CORBA layer.
orbpas33.dll	X	X	C:\Program Files\Borland\Delphi5\Bin	This file is not required for all deployments. It comes with the <u>Visibroker 3.3 for Delphi update</u> . Provides additional functionality and works in conjunction with existing DLLs.
cp3245mt.dll	X	X	C:\Program Files\Borland\Delphi5\Bin	C++ Builder Runtime Library.
borlndmm.dll	X	X	C:\Program Files\Borland\Delphi5\Bin	Used by cp3245mt.dll and orbpas50.dll.
orb_br.dll	X	X	C:\Program Files\Borland\vbroker\Bin.	VisiBroker runtime Library.
OSAGENT		X	C:\Program Files\Borland\vbroker\Bin	One copy needs to be running per network.
IREP		X	C:\Program Files\Borland\vbroker\Bin	One per network – optional.
OAD		X	C:\Program Files\Borland\vbroker\Bin	One per network – optional.

## Environment

Here is environment information that should be used in your deployment. This is what the installer does to the environment and to the Windows Registry during installation of VisiBroker for Delphi.

The following environment variable is modified (if the user approves):

PATH=<vbroker installation root>bin;%path%

### Windows Registry Settings

HKEY\_LOCAL\_MACHINE\SOFTWARE\Visigenic Software Inc\VisiBroker3.0

OSAGENT\_PORT=14000 (default)

Pathname=<vbroker installation root>

vbc="" (used during uninstall)

vbj="" (used during uninstall)

VBROKER\_ADM=<vbroker installation root>adm

VisiBroker Developer for Java 3.0 key=

### NT Services - OSAgent

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\services\osagent

DisplayName "VisiBroker Smart Agent"

ImagePath "<vbroker installation root>\bin\osagent.exe"

ObjectName "Domain Nameusername"

### NT Services - OAD

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\services\load

Display Name "VisiBroker Activation Daemon"

ImagePath "<vbroker installation root>\bin\load.exe"

ObjectName "Domain Nameusername"

The NT Service installation is done using dlls( register.dll ) packaged with the installer.

The Services are installed with SERVICE\_ALL\_ACCESS, SERVICE\_INTERACTIVE\_PROCESS | SERVICE\_WIN32\_OWN\_PROCESS, SERVICE\_DEMAND\_START and SERVICE\_ERROR\_CRITICAL flags to CreateService.

The services run under LocalSystem account.

## Troubleshooting

Error "NO\_IMPLEMENT"

This means the client app cannot find the server and could be caused by a multitude of reasons.

Run OSFIND.EXE. If you don't see your server listed you are not set up correctly.

Other options:

- \* Is the DOS path correct
- \* Are all DLLs present on correct machines (see table)?

Last Modified: 01-MAR-01



**Article #21967: Oracle 7 and Multiple Threads.**

**Question:**

Why am I experiencing delays when connecting to an Oracle database?

**Answer:**

There are problems with BDE 5.X (native Oracle driver) and OCI versions before 7.3.4. Upgrade to at least 7.3.4 (client).

Last Modified: 19-JUN-00

## Article #22507: Problems comparing floating point numbers.

Because of the way floating point numbers are represented by the computer there can be inconsistencies when comparing two numbers that appear to be identical. Unlike integers, IEEE floating point numbers are only approximates, not exact numbers. The need to convert the numbers to a form the computer can store in binary leads to minor precision or round-off deviations. For example 1.3 may really be represented as 1.299999999999.

There may particularly be problems when using conversion functions such as `StrToFloat`, `TextToFloat` and `StrToCurr`. Therefore you should never use `=` or `<>` to compare two floating point numbers. Instead, subtract the two numbers and compare them against a very small number.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f1, f2 : double;
begin
  f1 := 1.3000000;
  f2 := 1.2999999;
  if abs(f1-f2) < 0.00001
  then ShowMessage('f1 and f2 are equal')
  else ShowMessage('f1 and f2 are not equal');
end;
```

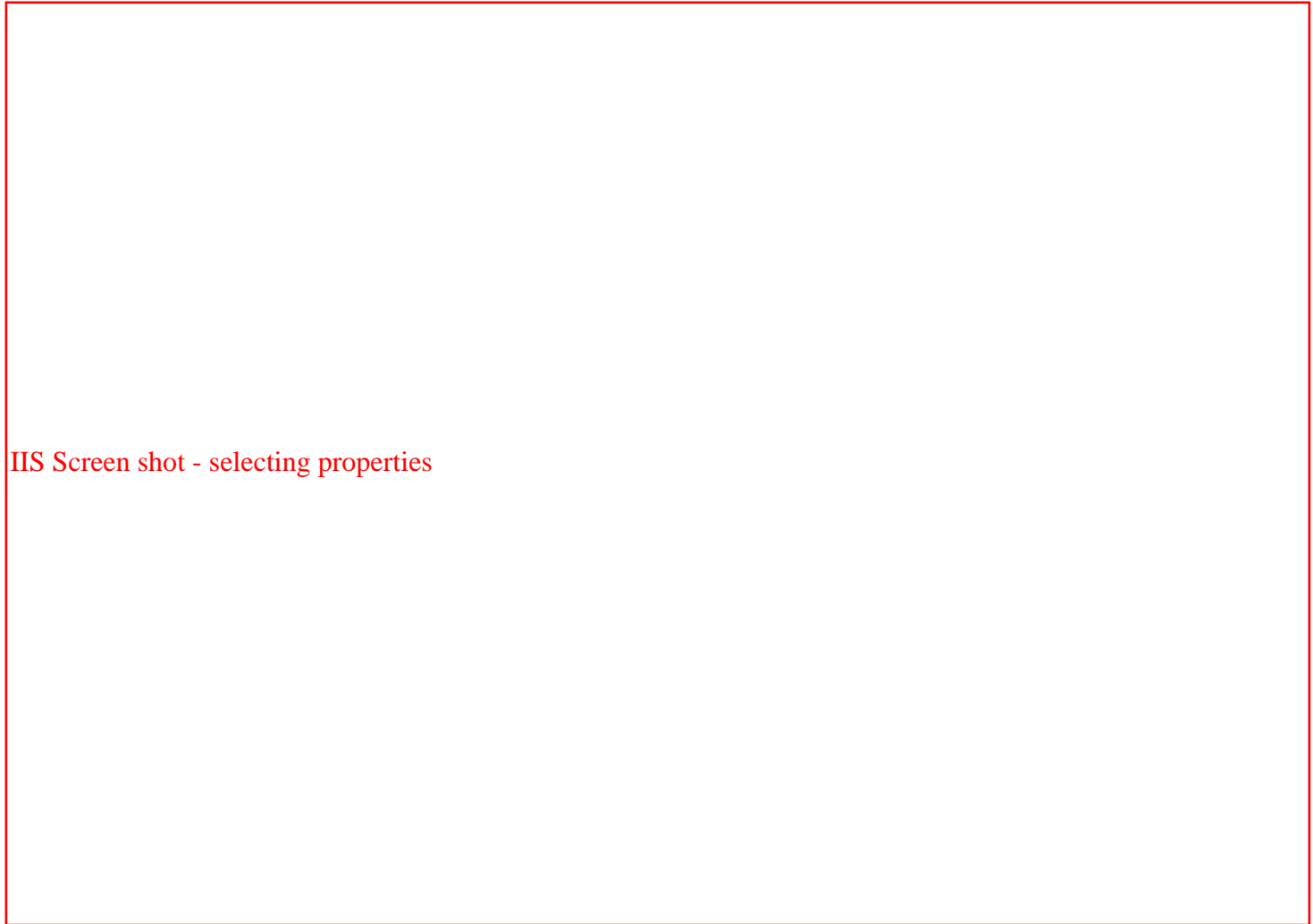
Last Modified: 02-OCT-00

## Article #23024: How to debug ISAPI DLL's on Windows 2000 with IIS5

### How to debug ISAPI DLL's on Windows 2000 with IIS 5

by Corbin Dunn

1. In the "Administrative Tools" portion of the "Control Panel", start the "Internet Services Manager." Right click on your website and select "Properties":



2. On the "Home Directory" tab, set "Application Protection" to "Low (IIS Process)":



## IIS Process

This causes IIS load ISAPI DLL's inside of its process, allowing you to debug them. Close this dialog, but not the IIS Manager program.

3. Expand your website and right click on the "Scripts" virtual directory (or, whatever directory your ISAPI DLL's will be put into). Again, set the "Application Protection" to "Low (IIS Process)" for the virtual directory (similar to step 2).

**Important - if you don't do this you will probably not be able to load your ISAPI DLL.** You can now close the IIS Manager.

4. In the "Administrative Tools" portion of the "Control Panel" select click on the "Services" and for each of the following services, make sure that they are "Stopped" and that the "Startup Type" is set to Manual:

- Simple Mail Transport Protocol (SMTP)
- World Wide Web Publishing Service
- FTP Publishing Service
- IIS Admin Service

Close the Services Window.

**If after you reboot the IIS Admin Service has started, then try making the Startup Type be Disabled - then reboot and make sure that it has not started. If it has, you won't be able to debug.**

5. Run "Local Security Policy" from the "Administrative Tools" folder. Expand "Local Policies", and then click on "User Rights Assignment". In the right hand panel, you should see "Act as part of the operating system". Double click on it, and select "Add..." to add a user to the list. Add your current user to the list that you intend to debug with (for example, I added the user cdunn, which is my main login to my Windows 2000 machine). Close this window after adding the user.

Set OS stuff

6. Reboot to make the previous step take effect.

7. Merge the file IISProcess.reg to merge the proper registry settings that make IIS run as a process on your system (so you can debug). Important: If you installed IIS into somewhere other than the default directory, you will have to modify this file. The default location is: C:\WinNT\system32\inetsrv". Use IISService.reg to set the registry settings back as a service.

*NOTE:* These files are on code central as a zip file, but, if you download them individually, you will have to rename the downloaded file to the right name. It may prompt you to name it "getfile.reg", but rename this to "IISService.reg" or "IISProcess.reg".

---

### **In Delphi:**

1. In your ISAPI DLL project, go to the "Project Options" and on the "Directories/Conditionals" tab, set the "Output directory" to be "C:\Inetpub\Scripts" or whatever directory you put your ISAPI dll's in. Also, make sure you have all debugging information turned on.

2. Select Run->Parameters from the IDE menu. Set the Host Application to be: "c:\winnt\system32\inetsrv\inetinfo.exe" (without the quotes - assuming this is where you installed IIS). Set the Parameters to be "-e w3svc" (without the quotes).

3. You should now be able to select "Run" and have Delphi start in debug mode. Put a breakpoint somewhere in your source code. Now, using a web browser, browse to your ISAPI DLL (such as http://localhost/scripts/project1.dll), and it should stop on the breakpoint.

4. Use Ctrl-F2 to stop the web server and end a debugging session.

---

### **Trouble shooting**

If you get an error about: *"A required privilege is not held by the client."* then you probably haven't done step 3, or you are trying to debug in a scripts directory that hasn't had step 3 done on it. Or, you have not rebooted at some point.

If the debugger starts and stops right away, then you probably have inetinfo.exe running. Run the Task Manager and look

for inetinfo.exe running. Try to kill it. If you can't it is currently running as a service. Try shutting it down in the services. If it is stuck on "Starting" the service, then run IISService.reg and reboot. After rebooting, stop the service. Then run IISProcess.reg and try debugging.

When releasing your ISAPI DLL application, be sure to undo what you did in these steps to not hinder performance on your DLL.

Last Modified: 04-APR-01

## **Article #25920: Writing Clean and Safe UDFs in Delphi**

This document is a PDF "Portable Document Format" file. You will need Adobe Acrobat Reader to view it.

Last Modified: 27-OCT-00

## Article #26048: Problems using dcomcnfg to configure ActiveX library objects (DLLs).

Issues relating to configuring DCOM for ActiveX Library objects.

### Description of problem:

After successfully building and registering the server object DLL (e.g. ActiveX Library with Automation object), the Client application produces a 'Class not found' error message when attempting a remote DCOM connection using CreateRemote(). Server applications created as executables (e.g. Application with Automation object) do not appear to have this problem.

The crux of this issue is that, in order to make DCOM work, the COM object must be properly configured using an external utility. Unfortunately, the default utility for configuring DCOM (dcomcnfg.exe) does not seem to *reliably* locate COM library objects (DLLs), but is only able to find COM executable object (EXE). Furthermore, even when it does display the library, there seems to be no way from within the utility to configure the object to make it work. The reason for this is because the dcomcnfg utility does not allow specification of a surrogate process to host the DLL object for the Inproc Server.

The workaround is to download and install OLEview (free):

<http://www.microsoft.com/com/resources/oleview.asp>

Now, the critical step is specifying that the implementation utilize a surrogate process for hosting the DLL. This may be done in the following way:

1. Register the DLL using regsvr32
2. Run OLEview.
3. In the left panel, expand 'All Objects,' and select your server object.
4. In the right panel, select the 'Implementation' tab.
5. Then select the sub-tab 'Inproc Server'
6. Check 'Use Surrogate Process'

This may need to be done on each computer that will be hosting the server. You will also need to register the DLL on each system before you can use the Client application. Finally, if you receive errors regarding access rights, you may need to specify 'Launch Permissions' and 'Access Permissions' in OLEview.

Borland Developer Support

Last Modified: 13-OCT-00

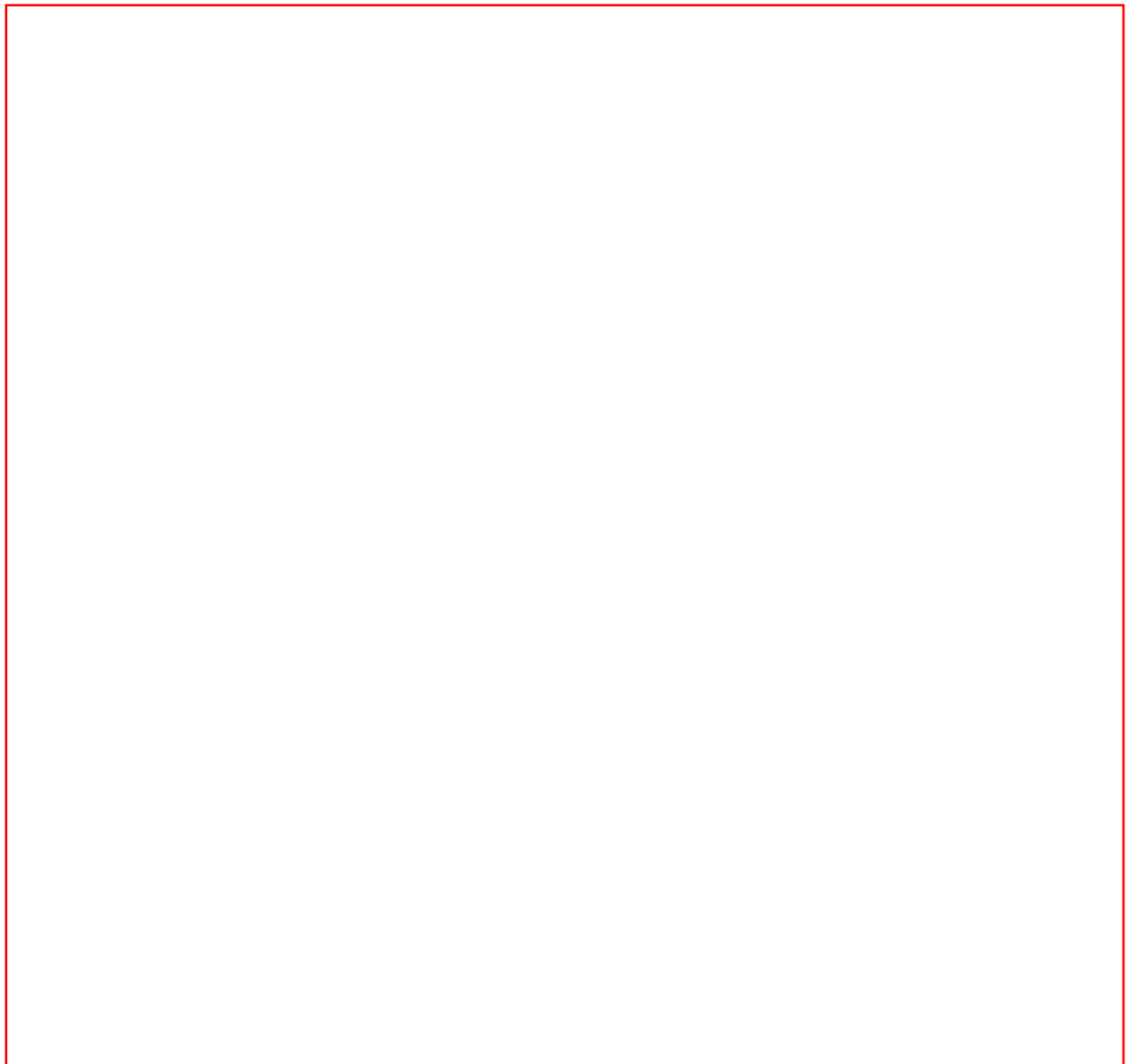


**Article #26277: Creating transparent windows**

**Creating Transparent Windows**  
*And moving around the children on that window*  
**by Corbin Dunn**

[Download the sample source code and project for this article.](#)

You may have seen some windows that have transparent portions, and wonder how you can do that in Delphi. It is quite easy, and there are several different ways to achieve the affect such as below:



The first way you could go about this is by adding the `WS_EX_TRANSPARENT` style to your window. This will work, but it will not have all the desired effects, and Microsoft recommends using the

WS\_EX\_TRANSPARENT style for modal windows that have a short life.

The second way of doing this is with Windows 2000's new Layered Windows feature. However, doing this will limit your application to only run on Windows 2000.

The third approach is the one chosen in this article. You can use the Windows API call SetWindowRgn to specify the exact portion of the window that you want seen. One of the tough parts is creating a region that contains all child windows that you want seen. The solution is to go through all visible Controls on the form, and create one gigantic region that combines all these small regions. Here is how to do this:

```
procedure TForm1.SetRegions;
var
  I: Integer;
  RgnAll, RgnCtrl: HRGN;
begin
  RgnAll := 0;
  for I := 0 to ControlCount - 1 do
  begin
    with Controls[I] do
    begin
      if Visible then
      begin
        // Create a region for each given visible control
        RgnCtrl := CreateRectRgn(Left, Top, Left + Width, Top +
Height);
        // Combine the region with all previous ones, if available
        if (RgnCtrl <> 0) and (RgnAll <> 0) then
        begin
          CombineRgn(RgnAll, RgnAll, RgnCtrl, RGN_OR);
          DeleteObject(RgnCtrl);
        end
        else
          RgnAll := RgnCtrl; // This is the first region being created
        end;
      end;
    end;
  // Now, set the RgnAll as what we see for the Window
  if RgnAll <> 0 then
  begin
    (*
    From SetWindowRgn in the help file:
    "After a successful call to SetWindowRgn,
    the operating system owns the region specified
    by the region handle hRgn. The operating system
    does not make a copy of the region. Thus, you
```

*should not make any further function calls with this region handle. In particular, do not close this region handle."*

*So don't call DeleteObject on RgnAll after using it for SetWindowRgn (thanks to Richard Albury for pointing this out!) A previous version of this article made this mistake.*

```
*)  
SetWindowRgn(Handle, RgnAll, True);  
end;  
end;
```

Note how I call DeleteObject after I was finished using the region. Not doing this will cause resource leaks on Windows. I use the API CreateRectRgn, but if you have a different shape you can use CreatePolygonRgn to customize the exact shape of the Window or Control.

One problem that you may encounter deals with moving controls on the form. If you programatically move the control (such as in an OnMouseMove event), the area behind the control will be out of the view of the control. Also, the control may not be properly painted while it is being moved. There is a simple solution for this: call SetRegions again to update the new visible area of the form, and force a repaint of the control with a call to Control.Repaint, such as below:

```
procedure TForm1.GenericMouseMove(Sender: TObject; Shift:  
TShiftState; X,  
Y: Integer);  
begin  
  // If the control wasn't moved, then exit right away  
  if (X - LastX = 0) and (Y - LastY = 0) then Exit;  
  // Move the control  
  with (Sender as TControl) do  
    begin  
      Left := Left + (X - LastX);  
      Top := Top + (Y - LastY);  
    end;  
    SetRegions;  
    (Sender as TControl).Repaint;  
end;
```

You should now be able to add transparent windows to your application. [Download the sample source code and project.](#)

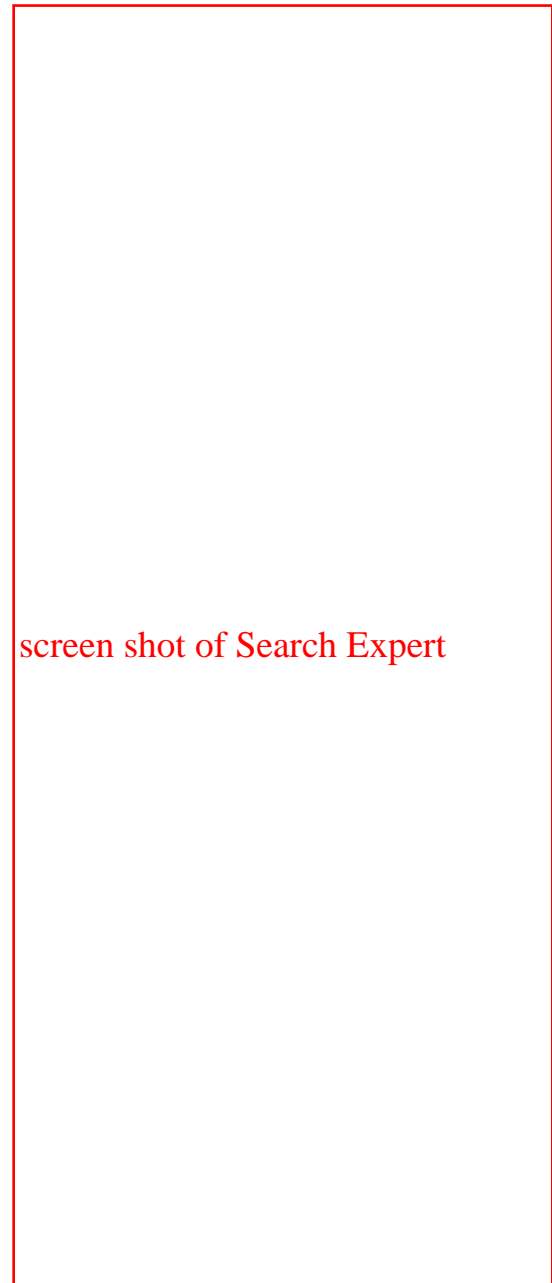
## Article #26418: Creating a Web Search Wizard using the OpenTools API.

### Creating a Web Search Wizard for Delphi

By [Corbin Dunn](#)

Delphi Developer Support

Allen Bauer has written three articles for the OpenTools API with the [most recent](#) giving us some ideas on how to create a dockable IDE form. One thing that he didn't go into was how to get an expert or wizard to remember its position and state. This article goes through how I created a Web Search Expert for Delphi 5 that remembers its current position and state in the Desktop Speedsetting.



First of all, [download the complete project](#) so we are on the same playing field. I strongly recommend reading [Allen Bauer's article](#) to see how I started the project, but this isn't necessary. The key in his article is how to derive from an internal IDE form by adding DsnIDE50.dcp to the requires list of the package. After you have downloaded the source, open up the package SearchWiz.dpk.

Now, take a look at the unit SearchWizMnu. This is your normal registration unit that adds a menu item

named "Web search..." to the Delphi IDE's Help menu. One of the new things to notice is in the Register procedure I call RegisterFieldAddress passing two parameters: a string that is the SearchWizForm's ini file section (which is 'SearchWizard' in this case), and the actual address of the SearchWizForm variable:

```
procedure Register;  
begin  
    if Assigned(RegisterFieldAddress) then  
        RegisterFieldAddress(CIniSection, @SearchWizForm);  
    ...  
end;
```

Whenever a desktop is loaded, the IDE checks to see if the SearchWizForm is to be created and shown. To see how this is done, open up a .dst file. These files contain your desktop speedsettings and are located in your Delphi5\bin directory. If you don't have any, then save your desktop as something. As an example, I have a speed setting named Corbin's, so I opened up Corbin's.dst. Looking into it, I see the ini file section for my search expert (you won't see this until you have installed the Search Wizard):

```
[SearchWizard]  
Create=1  
Visible=1  
State=0  
Left=260  
Top=116  
Width=213  
Height=481  
MaxLeft=-1  
MaxTop=-1  
ClientWidth=205  
ClientHeight=457  
TBDockHeight=481  
LRDockWidth=140  
Dockable=1  
Last Query=Corbin's treehouse  
Last Search Engine=Borland Community  
Use Same Window=1  
WebBrowser Showing=1
```

Whenever I start Delphi it sees Create=1 in the ini section of my SearchWizard and automatically creates an instance of the Wizard in the SearchWizForm variable (if it is nil - this way it doesn't create multiple instances of the same form). But how does Delphi know what class type to create? Well, that is what the next line is for:

```
RegisterDesktopFormClass(TSearchWizForm, CIniSection, CIniSection);
```

```
RegisterPackageWizard(TSearchWizard.Create);
```

The call to RegisterDesktopFormClass set's TSearchWizForm as the class type to create when it reads the CIniSection and determines it should create it. RegisterPackageWizard is simply used to register my Wizard with the IDE. Finally, notice how in the finalization section of the unit I call UnRegisterFieldAddress and free my SearchWizForm variable.

### **finalization**

```
if Assigned(UnRegisterFieldAddress) then
  UnRegisterFieldAddress(@SearchWizForm);

if SearchWizForm <> nil then
begin
  SearchWizForm.Free;
  SearchWizForm := nil;
end;
```

If we don't do this, then you may get access violations and make the IDE unstable. All these functions are in DeskUtil.pas, which is inside of the IDE. If you want to see the stub for this file, [download](#) Allen's example.

Next, we want to open up the main form unit, SearchWizFrm.pas, but it uses some internal IDE forms that I got from Allen's example, and if we open up SearchWizFrm.pas it will complain about not finding the base classes. The trick here is to open up DeskForm.pas, then DockForm.pas, and finally SearchWizFrm.pas. To get my Search Wizard to remember its position and data, I overrode LoadWindowState and SaveWindowState (both of which are in TDesktopForm). In SaveWindowState we are passed a TMemIniFile from which I write my settings to. By first calling inherited SaveWindowState, TDesktopForm saves the current position and visible properties.

```
procedure TSearchWizForm.SaveWindowState(Desktop: TMemIniFile;
  isProject: Boolean);
var
  I: Integer;
begin
  inherited SaveWindowState(Desktop, IsProject);
  if SaveStateNecessary and (Desktop <> nil) then
  begin
    Desktop.WriteString(CIniSection, CIniLastSearchQuery, cmbxQuery.
Text);
    with cmbxEngines do
      Desktop.WriteString(CIniSection, CIniLastSearchEngine, Items
[ItemIndex]);
    Desktop.EraseSection(CIniSavedQueries);
    if IsProject then
```

```

    for I := 0 to cmbxQuery.Items.Count - 1 do
        Desktop.WriteString(CIniSavedQueries, IntToStr(I), cmbxQuery.
Items[I]);
        Desktop.WriteBool(CIniSection, CIniUseSameWindow, FUseSameWindow);
        Desktop.WriteBool(CIniSection, CIniWebBrowserShowing,
FWebBrowserShowing);
    end;
end;

```

Notice the variable IsProject that tells if this is a project's desktop settings (.dsk file) that is being saved. Finally, LoadWindowState does the reverse, loading in all my custom information (such as saved search engines and queries):

```

procedure TSearchWizForm.LoadWindowState(Desktop: TMemIniFile);
var
    TempList: TStringList;
    I: Integer;
begin
    inherited LoadWindowState(Desktop);

    if Desktop <> nil then
    begin
        cmbxQuery.Text := Desktop.ReadString(CIniSection,
CIniLastSearchQuery, '');
        if FSearchEngineList.IndexOfName(Desktop.ReadString(CIniSection,
CIniLastSearchEngine, '')) >= 0 then
            cmbxEngines.ItemIndex :=
                FSearchEngineList.IndexOfName(Desktop.ReadString(CIniSection,
CIniLastSearchEngine, ''));
        ...
    end;

```

After all this code was in place, one of the biggest problems that I had was getting my Search Wizard to show up when it was left open from a previous desktop. I could see my IniSection in the a project's .dsk file, but it wasn't getting shown (.dsk files are your project's current desktop) This is because I have a custom SpeedSetting that I use, and the speed setting was being set before my Search Wizard ever existed. So, after hitting "View->Desktops->Save Desktop" and saving my desktop, the Search Wizard would open when I started Delphi (or switch to that desktop).

The next thing that you will want to do is install this Wizard into the IDE. It should be as simple as clicking "Install Package" in the Package Editor window. You will notice that the usual Project->Compile (F9) option will not work because a lot of the internal code is running in the IDE. This is normal, and also makes debugging really difficult. The easiest way to do it is to place ShowMessage calls inside of function calls to see where your program is at. Or, you could log things to a file.

Once again, feel free to download the source code for this project. Open up SearchWiz.dpk and click "Install Package". Then, select "Help->Web Search..." and save your desktop if you want it to remember the wizard being open!

If you find any problems with the source code, please let me know! I also wouldn't mind getting any updates you make to enhance the Wizard...

Last Modified: 14-DEC-00



## Article #26531: Automating Internet Explorer to POST Form Data

### Automating Internet Explorer to POST Form Data

By [Corbin Dunn](#)

Delphi Developer Support

This document is based on an example from the MSDN: [HOWTO: Automate Internet Explorer to POST Form Data](#). It shows the Delphi way of POSTing form data.

It is easy to automate Internet Explorer (IE) to send data using the POST method to any web server. This example illustrates automation of Internet Explorer (TWebBrowser) while it is embedded inside of the application, but it could easily be not shown and hidden from the user.

Internet Explorer versions 3.0, 3.01, and 3.02 do not support posting data to an HTTP server via the Navigate method. WinInet and URL Monikers are two possible alternatives. For more information on WinInet and Monikers, visit these articles on the MSDN: [Q165298 - HOWTO: Simulate a Form POST Request Using WinInet](#) and [Q165800 - SAMPLE: Using a URL Moniker to POST Data](#).

To test the example below, the following Active Server Page (ASP) should be saved to the file navpost.asp in a directory on Windows NT 4 or 2000 machine running at least Internet Information Server (IIS) version 3.0. The directory should be recognized by IIS as a virtual root with execute permission:

```
<HTML>
<%
cFlavor = Request("Flavor")
cName = Request("FName")
%>
<BODY>
Hello, <% =cName %>. <br>
One scoop of <% =cFlavor %> coming right up!
</BODY>
</HTML>
```

1. Create a new application with Delphi
2. Add two TLabel's, a TEdit, TComboBox, TWebBrowser and a TButton to the form with names and a layout similar to what is shown below:

## Form Screen Shot

In the TComboBox, add several flavors to the Items.

3. Under the implementation section of the form, add "uses HTTPApp;". HTTPApp contains the HTTPEncode function used to url-encode strings.

4. Double click on the TButton and add the following code to the OnClick event handler (I named my button btnSubmit):

```
procedure TForm1.btnSubmitClick(Sender: TObject);  
var  
    EncodedDataString: string;  
    PostData: OleVariant;  
    Headers: OleVariant;  
    I: Integer;  
begin  
    // First, create a URL encoded string of the data  
    EncodedDataString := 'FName=' + HTTPEncode(edtFirstName.Text) + '&' +  
        'Flavor=' + HttpEncode(cmbxFlavor.Text);  
    // The PostData OleVariant needs to be an array of bytes as large  
    // as the string (minus the NULL terminator)  
    PostData := VarArrayCreate([0, Length(EncodedDataString) - 1],  
varByte);  
    // Now, move the Ordinal value of the character into the PostData  
array  
    for I := 1 to Length(EncodedDataString) do
```

```
    postData[I-1] := Ord(EncodedDataString[I]);
    Headers := 'Content-Type: application/x-www-form-urlencoded' +
#10#13;
    // Finally, we just navigate to the URL. Note that you may have to
modify
    // the path to your ASP page's location.
    WebBrowser1.Navigate('http://localhost/scripts/navpost.asp',
EmptyParam,
    EmptyParam, postData, Headers);
end;
```

You should now be able to run the application and see the results.

[Download the complete source from CodeCentral](#)

Last Modified: 04-JAN-01

## Article #26574: Automating Internet Explorer to Find All Links on a Web Page

### Automating Internet Explorer to Find All Links on a Web Page

By [Corbin Dunn](#)

Delphi Developer Support

You may be faced in a situation where you need to find all links on a given web page. This document describes how to find all the links by automating Internet Explorer (IE). It was written with IE 5, but the concepts should work with IE 4 too.

[Download the complete source for this project from Borland CodeCentral](#)

*Note: MSHTML\_TLB.pas is not included with the above project because it is so large and one can import it by themselves. Read this document for how to import it.*

1. First, create a new application in Delphi. I saved Form1 as MainFrm.pas, and the application as FindingLinks.dpr.
2. To get the most power out of automating IE, first import the MSHTML type library. To do this, select Project->Import Type Library:



You should now see "Microsoft HTML Object Library (Version 4.0)" listed:

## Importing the TLB

Your version may be different if you have a different version of Internet Explorer (I have 5.0). If you don't see it listed, search on your computer for **mshtml.tlb**. After you find this file, click on the "Add..." button and select **mshtml.tlb**. If you don't find **mshtml.tlb**, then you may not have IE installed, or your version is outdated (so, update IE before continuing).

Finally, select "Microsoft HTML Object Library" and then click on the "Create Unit" button. It will take quite a while to create the unit because the type library is **very** large. My MSHTML\_TLB file that it created is 241,899 lines long! This should give you an idea about much you can automate Internet Explorer.

3. Once you have control of Delphi, go back to Form1. Create a layout similar to this:

## Form Layout

with these components:

<i>Component Class</i>	<i>Name</i>	<i>Caption or Text</i>
TLabel	lblURL	URL:
TEdit	edtURL	http://www.borland. com
TButton	btnFindLinks	Find All Links
TListBox	lstbxLinks	<i>n/a</i>

4. Add **OleCtrls**, **SHDocVw**, and **OleServer** to the **uses** list of the Form1's interface section. This allows us to create an instance of TInternetExplorer, which wraps the Internet Explorer ActiveX Object. We don't want to show the Internet Explorer control in this application. If you do want to, then place a TWebBrowser control on the form, as it will work in the same way and allow you to see the resulting web page.
5. Add the following to the **private** section of your form:

```
FInternetExplorer: TInternetExplorer;  
procedure WebBrowserDocumentComplete(Sender: TObject; var  
pDisp: OleVariant;  
    var URL: OleVariant);
```

Press Ctrl-Shift-C to complete the class declaration.

## 6. Add

```
uses MSHTML_TLB, ComObj;
```

under the implementation section of Form1.

## 7. Double click on Form1 to go to the OnCreate event of the form. Add the lines:

```
FInternetExplorer := TInternetExplorer.Create(Self);  
FInternetExplorer.OnDocumentComplete :=  
WebBrowserDocumentComplete;
```

## 8. For **TForm1.WebBrowserDocumentComplete** add the following code:

```
procedure TForm1.WebBrowserDocumentComplete(Sender: TObject;  
  var pDisp: OleVariant; var URL: OleVariant);  
var  
  Doc: IHTMLDocument2;  
  ElementCollection: IHTMLCollection;  
  HtmlElement: IHTMLElement;  
  I: Integer;  
  AnchorString: string;  
begin  
  lstbxLinks.Clear;  
  // We will process the document at this time. Trying to do  
  // so earlier won't work because it hasn't fully loaded.  
  Doc := FInternetExplorer.Document as IHTMLDocument2;  
  if Doc = nil then  
    raise Exception.Create('Couldn't convert the ' +  
      'FInternetExplorer.Document to an IHTMLDocument2');  
  // First, grab all the elements on the web page  
  ElementCollection := Doc.all;  
  for I := 0 to ElementCollection.length - 1 do  
  begin  
    // Get the current element  
    HtmlElement := ElementCollection.item(I, '') as IHTMLElement;  
    // Next, check to see if it is a link (tagName will be A).  
    // You could easily find other tags (such as TABLE, FONT,  
    etc.)  
    if HtmlElement.tagName = 'A' then  
    begin  
      // Now grab the innerText for this particular link. The  
      innerText is  
      // all text that is inside of the particular tag. For  
      example,
```

```

// this would give us "Go To Borland" from the HTML:
// <a href="http://www.borland.com"><b>Go To Borland</b></
a>.
// If you want "<b>Go To Borland</b>" use innerHTML.
AnchorString := HtmlElement.innerHTML;
if AnchorString = '' then
    AnchorString := '(Empty Name)';
// We know that the element is an IHTMLAnchorElement since
the tagName
// is 'A'.
AnchorString := AnchorString + ' - ' +
    (HtmlElement as IHTMLAnchorElement).href;
lstbxLinks.Items.Add(AnchorString);
end;
end;
end;

```

9. Next, double click on TButton **btnFindLinks** and add the following code in the **OnClick** event:

```

// Simply browse to the given page
FInternetExplorer.Navigate(edtURL.Text, EmptyParam, EmptyParam,
    EmptyParam, EmptyParam);

```

10. Compile the application and run it. It will probably take a while to compile because it has to compile the large MSHTML\_TLB.pas file. You will probably get a lot of warnings about the MSHTML\_TLB.pas file which you can ignore. You should then be able to find all links on a given web page by clicking the button. If you get a Variant exception when running in debug mode, you can ignore it (it doesn't effect anything).

[Download the complete source for this project from Borland CodeCentral](#)

*Note: MSHTML\_TLB.pas is not included with the above project because it is so large and one can import it by themselves.*

Last Modified: 22-JAN-01



## Article #26652: Random Images Screen Saver - a complete screen saver example

### Random Images Screen Saver in Delphi

By [Corbin Dunn](#)

Delphi Developer Support

There are some older documents on our site that describe how to go about writing a 32-bit screen saver with Delphi. I decided to update them with a completely new example that addresses several issues in some other screen savers.

First you will want to [download the complete source from CodeCentral](#) so you can follow along.

If you are new to creating a screen saver, there are certain things you have to do. First, you have to process command line switches and start your screen saver in the right "mode" depending on what was passed to it. For example, if you are passed /c as a parameter, you should start in the "Configure" mode. Inside my sample source code, I have a **ScreenSaverUtils** unit that takes care of the command line processing and some other utility routines. Inside the interface notice the global variables and function:

```
type
  TScreenSaverMode = (ssPassword, ssPreview, ssConfigure, ssRun);
var
  ScreenSaverMode: TScreenSaverMode;
  ParentSaverHandle: THandle = 0;

procedure SetScreenSaverMode;
```

Calling **SetScreenSaverMode** as the first thing in the screen savers .dpr file will cause ScreenSaverMode and optionally ParentSaverHandle (if present) to be properly set. I can then use it in the .dpr file to create the screen saver form, or the configuration form:

```
begin
  // First, process the Params passed to see
  // what mode the screen saver is to run in
  // This set's the global variable's ScreenSaverMode
  // and ParentHandle. If the mode was to set the password,
  // a call to this function will never return (it calls Halt).
  SetScreenSaverMode;

  Randomize;
  Application.Initialize;
  Application.Title := 'Random Images Screen Saver';
```

```

// Remove the start bar button
SetWindowLong(Application.Handle, GWL_EXSTYLE, WS_EX_TOOLWINDOW);
if ScreenSaverMode = ssConfigure then
begin
    // Create the configure form to configure the screen saver
    Application.CreateForm(TConfigureForm, ConfigureForm);
end
else
begin
    // We are either doing a preview or actually running the screen
saver
    Application.CreateForm(TMainForm, MainForm);
end;
Application.Run;
end.

```

**ConfigureForm** configures some registry settings for the screen saver. When the user selects "Settings..." from inside of the Display Properties of the control panel the configuration form will be shown. Some screen saver's don't modally show their configuration form, so it is possible to configure two screen savers at the same time. To correct this problem, I had to use the **ParentSaverHandle**, which should be properly set up after a call to **SetScreenSaverMode**. Inside of **TConfigureForm.FormCreate** I have:

```

if ParentSaverHandle <> 0 then
begin
    EnableWindow(GetParent(ParentSaverHandle), False);
end;

```

**ParentSaverHandle** is the handle to the little mini-desktop window that we are to draw a little "preview" on in the Display Properties dialog box. The parent of this little window is the actual Display Properties dialog box. The call to **GetParent** retrieves the Display Properties dialog box, and **EnableWindow** actually disables the window (making the configuration form modal). Notice the corresponding code in **TConfigureForm.FormDestroy** that re-enables the Display Properties dialog and brings it to the front:

```

if ParentSaverHandle <> 0 then
begin
    // The parent of the handle passed to us is
    // the main Properties dialog box
    EnableWindow(GetParent(ParentSaverHandle), True);
    SetForegroundWindow(GetParent(ParentSaverHandle));
end;

```

The next thing to take a look at is the main screen saver form: **MainForm**, inside of **MainFrm.pas**. This

form is created when the **ScreenSaverMode** is **ssRun** or **ssPreview**. When it is **ssRun**, we want the screen saver to:

1. Run in full screen mode
2. Hide the mouse
3. Close when the mouse is moved or a key is pressed
4. Prompt for a password on Windows 95/98/Me

To make it run in full screen mode, I set the **BorderStyle** property of **MainForm** to be **bsNone** and the **FormStyle** to be **fsStayOnTop**. Then, in the **OnCreate** event I have something close to this:

```
Top := 0;
Left := 0;
Width := Screen.Width;
Height := Screen.Height;
if ScreenSaverMode = ssRun then
begin
    SystemParametersInfo(SPI_SCREENSAVER_RUNNING, 1, @Dummy, 0);
    SetCapture(Self.Handle);
    ShowCursor(False);
end
```

Setting the width and height based on the Screen's properties makes it take up the full screen. The call to **SystemParametersInfo** is a crude way of telling Windows that the screen saver is running. **SetCapture** allows us to get all mouse movements, and **ShowCursor(False)**; hides the cursor. The **OnDestroy** event simply undoes all these things.

To test for key's being pressed, I simply call **Close** in the **OnKeyDown** event. Mouse movements are a little trickier because we want to have a certain tolerance to mouse movement. Take a look at **TMainForm.FormMouseMove** for how to do this.

Next, we have to prompt for a password, if this option is turned on (in Windows 95/98/Me only - Windows NT and 2000 do it for you). In the **OnCloseQuery** event I simply call the function **PromptIfPasswordNeeded** from **ScreenSaverUtils.pas**. If the function returns True, then **CanClose** is set to True and the screen saver will exit.

The actual drawing for the screen saver is done by timer events. The code is rather simple, and is left for you to examine yourself. Basically, you just draw whatever you want to the **MainForm**'s **Canvas**.

The next bit of trouble is to create the little preview in the Display Properties dialog box. If the **ScreenSaverMode** is **ssPreview**, then the **OnCreate** event of the **MainForm** set's the Application to not show the **MainForm** (with **Application.ShowMainForm := False**) and then calls **TMainForm.DoPreviewDrawing**. Here is this function:

```

procedure TMainForm.DoPreviewDrawing;
var
  Rect: TRect;
  Bitmap: TBitmap;
  Canvas: TCanvas;
begin
  FreeMutex; // Free the mutex so that the screen saver will actually
run in preview
  Canvas := TCanvas.Create;
  try
    Canvas.Handle := GetDC(ParentSaverHandle);
    try
      GetWindowRect(ParentSaverHandle, Rect);
      ...
      while IsWindowVisible(ParentSaverHandle) do
        begin
          // Drawing to Canvas.Handle will create your preview
          ...
        end;
      finally
        ReleaseDC(ParentSaverHandle, Canvas.Handle);
      end;
    finally
      Canvas.Free;
    end;
  Application.Terminate; // Done with the preview, so quit
end;

```

First, notice the call to **FreeMutex**, which is inside of the unit **SingleInstance**. This unit allows only one instance of the screen saver to run at any time. If we don't do this, then every screen saver timeout period, Windows will start a new instance of your screen saver, even if it is already running! The call to **FreeMutex** allows the Preview button to work correctly inside of the Display Settings dialog box (otherwise, timing issues will not allow it to show).

We get the Device Context (DC) from the little preview window that was passed to the Application via parameters. Setting the **TCanvas**'s Handle allows us to easily draw to it with a familiar Delphi class. Once the **ParentSaverHandle** window is not visible anymore, we terminate the application. The code to do the actual drawing is left out for simplicity.

Inside of the Display Properties dialog box you may have noticed that some screen savers specify a long name for the screen saver. In particular, it is different from just the executable's name. Microsoft claims that you can set this by creating a string table resource with an ID of 1 a value of whatever string you want. The included text file **delphi\_string\_table.rc** can be compiled into a resource (.res) file with the included **create\_res.bat** (you may have to modify the path information if you installed Delphi 5 into a

different location). Inside of the project's source I then have {\$R delphi\_string\_table.RES} to link in the resource. However, this doesn't always work. To make it work on Windows 2000/NT I discovered that the screen saver's executable name MUST be less than 8 characters long, AND all lowercase. I have no idea why this is required, but it seems to work.

Finally, you should be able to compile the source and you will have a nice little sample screen saver.

[Download the complete source from CodeCentral](#)

Last Modified: 18-JAN-01

## Article #26693: Using blocking sockets inside of a thread, and simulating HTTP requests

Using blocking sockets inside of a thread, and simulating HTTP requests

By [Corbin Dunn](#)

Delphi Developer Support

This article covers how to send an HTTP request to a web server, and work with the response. All of the data transfer is done with a blocking socket inside of a thread.

[Download this example from Borland CodeCentral](#)

You may have a couple questions at this point. Such as, **what is a blocking socket and why would I use it?**

A blocking function call is one that doesn't return until the function is totally complete. In other words, all reading and writing occurs synchronously. For sockets, this means that a call to the **recv** function will not return until all the data has been received. If a socket is set to non-blocking, a call to **recv** will return very quickly, returning the data that is ready at that time of the call. **Delphi's TClientSocket** is by default a non-blocking socket because the **ClientType** property set to **ctNonBlocking**. With the non-blocking socket you use the **OnRead** and **OnWrite** events to read and write to the socket. Basically, Windows sends your application a message when data is ready to read or write to the socket. If you set the **ClientType** to **ctBlocking**, then your application will freeze until the data has been read. The way to prevent this from happening is to put the **TClientSocket** in a thread.

If you go to <http://www.census.gov/ftp/pub/genealogy/names/> you will notice that there are a couple of text files named [dist.male.first](#) and [dist.female.first](#). They contain a list of the most frequent names in the 1990 Census. This example will send an HTTP request to [www.census.gov](http://www.census.gov) and get on of those two text files. It will then process the data and place it into a TListView. You can even search for your name.

Here is a screen shot of the example we will be working on:

If you want to just dig apart the code yourself, then [download it now](#). Otherwise, here are some step by step instructions on how I created the example.

1. First create a new application in Delphi. Rename the form to "**MainForm**" and save it in **MainFrm.pas**. Save the project as "**YourName.dpr**."
2. Next, add **ScktComp** to the uses list of **MainFrm.pas**.
3. We need to create a TThread that does the blocking socket work. Under the **type** keyword, but above the **MainForm** class, add:

```
TSocketThread = class(TThread)
private
    FClientSocket: TClientSocket;
    FData: string;
    FLastException: Exception;
    FGender: string;
    procedure SetGender(const Value: string);
protected
    procedure Execute; override;
    procedure HandleThreadException;
    procedure ThreadDone;
public
```

```

    constructor Create;
    destructor Destroy; override;
    property Gender: string read FGender write SetGender;
end;

```

Press Ctrl-Shift-C to complete the class.

4. In the **implementation** section, add the following code for your constructor and destructor:

```

constructor TSocketThread.Create;
begin
    inherited Create(True); // Initially suspended
    FreeOnTerminate := True;
    FClientSocket := TClientSocket.Create(nil);
    FClientSocket.Host := 'www.census.gov';
    FClientSocket.Port := 80; // HTTP port
    FClientSocket.ClientType := ctBlocking; // Blocking, so it is
in a thread
end;

destructor TSocketThread.Destroy;
begin
    FClientSocket.Free;
    inherited;
end;

```

The constructor will simply create the **TClientSocket** that will connect to [www.census.gov](http://www.census.gov) on the HTTP port (80), with a blocking socket. The **destructor** cleans up the allocated memory.

5. Now, standard TThread programming means that we will put our threaded code inside of the **Execute** procedure. To use a blocking socket, we have to use the **TWinSocketStream** class. The **TWinSocketStream** allows us to easily write to the socket and allow a timeout in case something goes wrong. Our **Execute** procedure will simply connect to the server, create a **TWinSocketStream**, and then read data until zero bytes are returned from a read, or the socket is closed. Zero bytes will be returned from a read when there is no more data available to read. Also, the HTTP port will be closed by the server when no more data is available. Here is the code:

```

procedure TSocketThread.Execute;
var
    Buffer: array[0..1024-1] of Char;
    SockStream: TWinSocketStream;
    RequestString: string;
begin
    try
        FClientSocket.Active := True;

```



```

FData := '';
FillChar(Buffer, SizeOf(Buffer), #0);

// Create the socket stream with a 1 minute timeout
SockStream := TWinSocketStream.Create(FClientSocket.Socket,
60000);
try
    RequestString := Format('GET /ftp/pub/genealogy/names/dist.
%s.first HTTP/1.0'#13#10#13#10,
    [FGender]);
    // Send the HTTP request
    SockStream.Write(RequestString[1], Length(RequestString));
    // Read the response in 1024 chunks
    while (SockStream.Read(Buffer, SizeOf(Buffer)) <> 0) do
    begin
        FData := FData + Buffer;
        FillChar(Buffer, SizeOf(Buffer), #0);
        // Check for termination of the thread or closure of the
socket
        if Terminated or not FClientSocket.Active then
            Exit;
        end;
    finally
        SockStream.Free;
        if FClientSocket.Active then
            FClientSocket.Active := False;
        end;
    except
        on E: Exception do
        begin
            if not(ExceptObject is EAbort) then
            begin
                FLastException := E;
                Synchronize(HandleThreadException);
            end;
        end;
    end;
    // Tell the main form that we are done
    Synchronize(ThreadDone);
end;

procedure TSocketThread.HandleThreadException;
begin
    Application.ShowException(FLastException);
end;

```

First of all, notice how I wrap all the operations in a **try..except** block. Any exceptions are handled via a call to **Synchronize** to prevent any multi-threaded problems from happening. For more information about how and why to do this read my article [How to handle exceptions in TThread objects.](#)

You will probably also notice my use of the **FGender** private data field. It contains a string of either "male" or "female" so that we can access the right file on the server depending on what the user selected. It is set via the **Gender** property's **write** procedure:

```

procedure TSocketThread.SetGender(const Value: string);
begin
    if (Value = 'male') or (Value = 'female') then
        FGender := Value
    else
        raise Exception.CreateFmt('Unknown gender (%s) to search
for',
            [Value]);
end;

```

6. The last thing that the **TSocketThread**'s **Execute** procedure does is call **ThreadDone** inside of **Synchronize** to make it thread safe. The code is as follows:

```

procedure TSocketThread.ThreadDone;
begin
    // The VCL is not thread safe. Accessing MainForm from inside
    // a thread would be bad, unless if it is inside a procedure
    // that was called via Synchronize.
    MainForm.ProcessData(FData);
end;

```

7. **ProcessData** is a function that we will add to the **MainForm**.
8. Now, the next thing to do is to set up **MainForm** to use the thread class that we just created. First, lay out some components on the form to make it look like the screen shot above.

**Here are the components and names that I used:**

<i>Component Class</i>	<i>Name</i>	<i>Caption or Text</i>
TRadioButton	rdbtnMaleNames	Male First Names
TRadioButton	rdbtnFemaleNames	Female First Names

TButton	btnGetData	Get Data From census. gov
TLabel	lblStatus	lblStatus (set at runtime)
TLabel	lblSearch	Search:
TEdit	edtSearch	CORBIN
TButton	btnFindIt	Find It
TListView	lstvwNames	n/a
Set the <b>ViewStyle</b> property for <b>lstvwNames</b> to be <b>vsReport</b> . Double click on <b>lstvwNames</b> and add four <b>columns</b> : Name, Percent of Population, Total Percent, and Rank.		

9. Double click on **btnGetData** and add the following code to create an instance of our **TSocketThread**:

```

procedure TMainForm.btnGetDataClick(Sender: TObject);
begin
    btnGetData.Enabled := False;
    // Create a thread
    with TSocketThread.Create do
        begin
            if rdbtnMaleNames.Checked then
                Gender := 'male'
            else
                Gender := 'female';
            Resume;
        end;
    end;

```

Remember that **rdbtnMaleNames** is the name of the **TRadioButton** on the form.

10. The next thing we need to do is process the data returned. The thread will call **MainForm.ProcessData(FData)** after it has received all its data. We need to add this procedure to the **MainForm**'s class interface. Since the **TSocketThread** is declared in the same unit, we can put the **ProcessData** procedure in the private section of **TMainForm**. Also add a **TStringList** to store some extra data later on:

```

private
    FNamesList: TStringList;
    procedure ProcessData(Data: string);

```

Press Ctrl-Shift-C to complete the class.

11. HTTP responses will have a header that we don't really care about. The start of data is immediately following two carriage-return newline pairs (ASCII #13#10). I simply use **Pos** to find this location and start processing the data after it:

```
procedure TMainForm.ProcessData(Data: string);  
var  
    DataStart: Integer;  
    CurrentPos: PChar;  
    Name: string;  
    ListItem: TListItem;  
  
    function ReadNextToken: string;  
    begin  
        Result := '';  
        // All tokens are delimited by whitespace.  
        // First, pass over any whitespace at the start.  
        while (CurrentPos^ in [#10, #13, ' ']) do  
            Inc(CurrentPos);  
  
        // Read in characters until we hit some more whitespace  
        while not (CurrentPos^ in [' ', #0, #10, #13]) do  
            begin  
                Result := Result + CurrentPos^;  
                Inc(CurrentPos);  
            end;  
        end;  
  
begin  
    // This is a very simple parser.  
  
    // First, we must skip the response header (it will have 2  
CRLF pairs before  
// the start of the data).  
  
    DataStart := Pos(#13#10#13#10, Data);  
    if DataStart = 0 then  
        raise Exception.Create('Could not find the data in the HTTP  
response');  
  
    // lstvwNames is the name of the TListView for displaying the  
data.  
    lstvwNames.Items.BeginUpdate;  
    try  
        lstvwNames.Items.Clear;
```

```

    // FNameList is a TStringList used to keep a sorted list of
the names.
    // This way, we can do a quick locate on it.
    FNameList.Clear;

    // Start past the 2 CRLF's (4 characters) with a pointer to
that character
    CurrentPos := PChar(@Data[DataStart + 4]);
    while (CurrentPos <> nil) and (CurrentPos^ <> #0) do
    begin
        Name := ReadNextToken;
        if Name <> '' then
        begin
            // Create a list item
            ListItem := lstvwNames.Items.Add;
            ListItem.Caption := Name;
            ListItem.SubItems.Add(ReadNextToken);
            ListItem.SubItems.Add(ReadNextToken);
            ListItem.SubItems.Add(ReadNextToken);
            // Add the name to the internal list of names so we can
easily
            // search for a name.
            FNameList.Add(Name);
            // Keep a pointer to the item.
            FNameList.Objects[FNameList.Count - 1] :=
ListItem;
            lblStatus.Caption := 'Processing the data...';
            Application.ProcessMessages;
        end
        else
            Break;
        end;
    finally
        lstvwNames.Items.EndUpdate;
    end;
    // Sort the internal list of names
    FNameList.Sort;
    lblStatus.Caption := 'Done';
    btnGetData.Enabled := True;
end;

```

12. You may have noticed that we use **FNameList** without ever creating it. So, in the **OnCreate** and **OnDestroy** events for **TMainForm** add the following code:

```

procedure TMainForm.FormCreate(Sender: TObject);
begin
    lblStatus.Caption := '';
    FNamesList := TStringList.Create;
end;

```

```

procedure TMainForm.FormDestroy(Sender: TObject);
begin
    FNamesList.Free;
end;

```

13. The last thing to do is to allow the user to search for their name. Double click on **btnFindIt** and add the following code:

```

procedure TMainForm.btnFindItClick(Sender: TObject);
var
    Index: Integer;
begin
    Index := FNamesList.IndexOf(UpperCase(edtSearch.Text));
    if (Index >= 0) then
        begin
            lstvwNames.Selected := TListItem(FNamesList.Objects[Index]);
            lstvwNames.Selected.MakeVisible(False);
            lstvwNames.SetFocus;
        end
    else
        raise Exception.Create('Could not find ' + edtSearch.Text);
end;

```

Now the reason for having **FNamesList** becomes apparent!

14. You should be able to compile the application and have it work. If not, then just **download the sample project**.

That's it! If I made any errors, please [email me](#).

Last Modified: 24-JAN-01

## Article #26736: I want to be able to use multiple aliases with a single TDatabase component.

### DESCRIPTION:

This Document will cover the following topics:

- The creation and deletion of dBase and Paradox BDE aliases in Delphi
- Using multiple aliases with one Database component
- Saving changes to the IDAPI.CFG file without using BDE Admin.

### ADDING ALIASES USING THE TSESSION COMPONENT:

Every Database application in Delphi has a default TSession component called SESSION. This Component manages all of your application's current database connections. You can also use it to add BDE aliases to the current session. This example creates a new dBase alias:

```
procedure TForm1.CreateNewAlias;
begin
  with Session do
  begin
    if not IsAlias('Deebase') then
      AddStandardAlias('DeeBase', 'c:\program files\Common files
\borland shared\data',
        'DBASE')
    else
      ShowMessage('Alias name already exists!');
  end;
end;
```

AddStandardAlias takes three string parameters. They are the name of the new alias, the path to the directory where the database resides, and the name of the driver to be used. IsAlias is a Boolean function that checks for the existence of a given Alias name.

### USING YOUR ALIAS:

Once you have created your new alias it can be used by your datasets. It is a simple matter to switch between several different aliases using one DataSet component. It requires only a reassignment of the alias.

```
procedure TForm1.OpenTable;
begin
  with Table1 do
  begin
    DatabaseName := 'DeeBase';
```

```
    TableName := 'clients.dbf';
    Active := True;
end;
end;
```

**Note:** *You do not use a full path to a file when creating a Paradox or dBase Alias. The actual file is reconciled later when a specific table is specified.*

### **SAVING YOUR ALIASES:**

Up until now, you have probably been using the BDE Administrator to change you ISAPI.CFG file. Now you can use your applications session component to do the same. Once you have created the session you want, you can permanently add them to the BDE configuration file by calling Session.SaveConfigFile.

### **DELETING ALIASES:**

Deleting BDE aliases is as simple as calling Session.DeleteAlias. Another call to SaveConfigFile will make this permanent. Use this to clean up alias that you don't want to persist after your application terminates.

### **WHOLE SOURCE EXAMPLE:**

The following is the main unit in a simple application that switches between and Paradox and dBase table while displaying the information to a TDBGrid.

```
-----Source-----
//AliasUnit.pas
unit AliasUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  StdCtrls, Grids, DBGrids, Db, DBTables, Menus;

type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    Table1: TTable;
    Datasel1: TDatabase;
    DBGrid1: TDBGrid;
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
    Label2: TLabel;
  end;
end;
```



```

MainMenu1: TMainMenu;
File1: TMenuItem;
Exit1: TMenuItem;
procedure FormCreate(Sender: TObject);
procedure ParadoxButtonClick(Sender: TObject);
procedure DBaseButtonClick(Sender: TObject);
procedure FormClose(Sender: TObject; var Action: TCloseAction);
procedure Exit1Click(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;

```

```

var
  Form1: TForm1;

```

```

implementation

```

```

{$R *.DFM}

```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  //Create the two session aliases
  with Session do
  begin
    if not IsAlias('Pdox') then
AddStandardAlias('PDox','c:\program files\common files\borland shared
\data','Paradox');
    if not IsAlias('Deebase') then
      AddStandardAlias('deeBase','c:\program files\common files
\borland shared\data','DBASE');
    end;
  end;
end;

```

```

procedure TForm1.ParadoxButtonClick(Sender: TObject);
begin
  //hook up the db to the new alias
  with Databasel do
  begin
    Connected := False;
    AliasName := 'PDox';
  end;
  //setup and activate table
  with Table1 do

```

```

begin
    TableName := 'country.db';
    Active := True;
end;
end;

procedure TForm1.DBaseButtonClick(Sender: TObject);
begin
    //hook up the db to the new alias
    with Database1 do
    begin
        Connected := False;
        AliasName := 'deebase';
    end;
    //setup and activate table
    with Table1 do
    begin
        TableName := 'clients.dbf';
        Active := True;
    end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    with Session do
    begin
        DeleteAlias('PDox');
        DeleteAlias('deeBase');
        SaveConfigFile;
    end;
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
    Close;
end;

end.

```

Last Modified: 05-FEB-01

## Article #26876: Use the Windows API to generate a list of available Network Resources.

### QUESTION:

How can I get a list of the computer names from Network Neighborhood?

### ANSWER:

Windows provides a set of API functions that allow enumeration of Network Resources. There are three basic steps to follow:

The first step is to call *WNetOpenEnum()*:

```
DWORD WNetOpenEnum(  
    DWORD dwScope,           // scope of enumeration  
    DWORD dwType,            // resource types to list  
    DWORD dwUsage,           // resource usage to list  
    LPNETRESOURCE lpNetResource, // resource structure  
    LPHANDLE lpEnum          // enumeration handle buffer  
);
```

The parameters passed into *WNetOpenEnum* determine what type of enumeration you will get. In this example *dwScope* is set to *RESOURCE\_CONTEXT*. This should limit *WNetEnumResource()* to finding only Network resources within your workgroup, which should be the list of computers retrieved from "Computers Near Me". Setting *dwType* to *RESOURCE\_DISK* keeps *WNetResource()* from returning any printers, etc. Check the on line help for more documentation on *WNetOpenEnum()*.

Once *WNetOpenEnum()* has been called the next step is to to enumerate the handle passed into *WNetOpenEnum()*, *lpEnum*. This is done through a call to *WNetEnumResource()*:

```
DWORD WNetEnumResource(  
    HANDLE hEnum,           // handle to enumeration  
    LPDWORD lpcCount,       // entries to list  
    LPVOID lpBuffer,        // buffer  
    LPDWORD lpBufferSize    // buffer size  
);
```

*WNetEnumResource()* also takes a pointer to an array of Network Resources, *lpBuffer*, which will be stuffed chuck full of yummy network goodness.

Finally, a call to *WNetCloseEnum()* should be made to close the resource handle:

```
DWORD WNetCloseEnum(  
    HANDLE hEnum // handle to enumeration  
);
```

Here is an example that I ported from Microsoft's Web site [Seen Here](#). I have changed the example to get the computers in the current workgroup, *RESOURCE\_CONTEXT*, instead of grabbing all available global resources.

In my example I use an array of TNetResource as opposed to allocating space for a pointer. I also used *FillChar()* to initialize the array. These are the main differences from my example and Microsoft's.

```
function EnumerateFunc( hwnd: HWND; hdc: HDC ; lptr: PNetResource ): Boolean;
const
    cbBuffer : DWORD          = 16384;          // 16K is a good size
var
    hEnum, dwResult, dwResultEnum : DWORD;
    lptrLocal : array
        [0..16384 div SizeOf(TNetResource)] of TNetResource;    // pointer to
enumerated structures
    i : integer;
    cEntries : Longint;
begin
    centries := -1;                                // enumerate all possible entries

    // Call the WNetOpenEnum function to begin the enumeration.
    dwResult := WNetOpenEnum(
        RESOURCE_CONTEXT, // Enumerate currently connected
resources.
        RESOURCETYPE_DISK, // all resources
        0, // enumerate all resources
        lptr, // NULL first time the function is
called
        hEnum // handle to the resource
    );

    if (dwResult <> NO_ERROR) then
    begin
        // Process errors with an application-defined error handler
        Result := False;
        Exit;
    end;

    // Initialize the buffer.
    FillChar( lptrLocal, cbBuffer, 0 );

    // Call the WNetEnumResource function to continue
    // the enumeration.
    dwResultEnum := WNetEnumResource(hEnum, // resource handle
        DWORD(cEntries), // defined locally as -1
        @lptrLocal, // LPNETRESOURCE
        cbBuffer); // buffer size

    // This is just printing
    for i := 0 to cEntries - 1 do
    begin
```

```
// loop through each structure and
// get remote name of resource... lpnrLocal[i].lpRemoteName)
end;

// Call WNetCloseEnum to end the enumeration.
dwResult := WNetCloseEnum(hEnum);

if(dwResult <> NO_ERROR) then
begin
    // Process errors... some user defined function here
    Result := False;
end
else
    Result := True;
end;
```

Last Modified: 12-FEB-01

## Article #26977: Finding the Windows task bar using the SHAppBarMessage API call

### Finding the Windows task bar

By [Corbin Dunn](#)

Delphi Developer Support

You may have come across a need to find the Windows task bar. This can be a simple task with Delphi. The Windows API call **SHAppBarMessage** allows you to get information about the task bar (also called the app bar by some). Visit [here on the MSDN](#) to find out more information about what you can do with this call. This document concentrates on passing it a message of ABM\_GETTASKBARPOS.

If you would like to follow along, you can create a new Application, or you can [download the complete source](#) from Borland's CodeCentral.

The first thing you may be wondering is what unit the API call is in. On the MSDN one can find out that it is declared in shellapi.h. In Delphi, it is in ShellApi.pas, so add **ShellApi** to your **uses** list. We can then code up a simple function that finds its position and bounding rectangle:

```
// FindTaskBar returns the Task Bar's position, and fills in  
// ARect with the current bounding rectangle.  
function FindTaskBar(var ARect: TRect): Integer;  
var  
    AppData: TAppBarData;  
begin  
    // 'Shell_TrayWnd' is the name of the task bar's window  
    AppData.Hwnd := FindWindow('Shell_TrayWnd', nil);  
    if AppData.Hwnd = 0 then  
        RaiseLastWin32Error;  
    AppData.cbSize := SizeOf(TAppBarData);  
    // SHAppBarMessage will return False (0) when an error  
    // happens.  
    if SHAppBarMessage(ABM_GETTASKBARPOS, AppData) = 0 then  
        raise Exception.Create('SHAppBarMessage returned false when  
trying ' +  
        'to find the Task Bar's position');  
    // Otherwise, we had success, so fill in the results.  
    Result := AppData.uEdge;  
    ARect := AppData.rc;  
end;
```

Once you have added this code to your application, you can use the function. Add a TLabel and a

TButton to your form. Then, in the button click, add the following code:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Rect: TRect;  
    DestLeft: Integer;  
    DestTop: Integer;  
begin  
    DestLeft := Left;  
    DestTop := Top;  
    case FindTaskBar(Rect) of  
        ABE_BOTTOM:  
            begin  
                DestLeft := Trunc((Screen.Width - Width) / 2.0);  
                DestTop := Rect.Top - Height;  
            end;  
        ABE_LEFT:  
            begin  
                DestTop := Trunc((Screen.Height - Height) / 2.0);  
                DestLeft := Rect.Right;  
            end;  
        ABE_RIGHT:  
            begin  
                DestTop := Trunc((Screen.Height - Height) / 2.0);  
                DestLeft := Rect.Left - Width;  
            end;  
        ABE_TOP:  
            begin  
                DestLeft := Trunc((Screen.Width - Width) / 2.0);  
                DestTop := Rect.Bottom;  
            end;  
    end;  
    Labell.Caption := Format('Found at Top: %d Left: %d Bottom: %d  
Right: %d)',  
        [Rect.Top, Rect.Left, Rect.Bottom, Rect.Right]);  
    // Move us to the task bar  
    while (Left <> DestLeft) or (Top <> DestTop) do  
    begin  
        if Left < DestLeft then  
            Left := Left + 1  
        else if Left <> DestLeft then  
            Left := Left - 1;  
  
        if Top < DestTop then  
            Top := Top + 1
```

```
else if Top <> DestTop then
```

```
    Top := Top - 1;
```

```
    Application.ProcessMessages;
```

```
end;
```

```
end;
```

That's it! You should now be able to find the task bars current position.

**[Download the complete source from Borland's CodeCentral.](#)**

Last Modified: 05-MAR-01



## Article #27126: Writing COM Automation Events

### WHAT IS AN AUTOMATION EVENT?

The typical COM Client/Server model that you have probably worked with before allows the client to call the server through a supported interface. This is fine when a client calls the server to perform an action or retrieve data, but what about when the server wants to ask things of the client? Incoming interfaces are just that: incoming. There is no way for the server to talk to any of its clients. This is where the Server Events model comes into play. A Server that supports Events not only responds to calls from the client, but can also report status and make its own requests of the client. For Example: A client makes a request for the server to download a file. Instead of waiting for the server to finish this download before continuing (as with the previous model), the client can go about another task. When the server is done, it can fire an Event that lets the client know it is finished, thus allowing the client to respond accordingly.

### HOW DO THEY DO IT?

The way in which a client is called by the server is not too much different in concept from the reverse. The server defines and fires events while the client is responsible for connecting to and implementing events. This is accomplished through an *events interface* or *outgoing interface* defined in the server object's type library. Now before we really get started, it's time for some terminology.

*Connection Point*: An entity describing the access to an events interface.

*Event Source*: An object that defines an outgoing interface and fires events, typically the Automation Server.

*Event Sink*: An object that implements an event interface and responds to events, typically the client.

*Advise*: Linking a sink to a connection point so that the sink's methods can be accessed by the source.

These are the main pieces of the Events model. If I were to sum this article up in one sentence it would be this: Simply put, an event sink will connect to a source via a connection point, thereby allowing the source to fire events implemented by the sink. My Goal now is to step through a simple server and client. The client will have a button that will call a server method. This method will simply fire and event that the client will catch and report via a memo control.

### LET'S WRITE THE SERVER

Any Automation server that wants to communicate using events needs to define an outgoing interface, and must implement an incoming interface for finding and attaching to those interfaces. This incoming interface is `IConnectionPointContainer`. The client will use this interface to find or enumerate the connection points supported by the sink with the `FindConnectionPoint` and `EnumConnectionPoints` methods. An `IConnectionPoint` connection point will be returned, from which the client can then call `Advise()` to advise its sink to the connection point. Both of these

interfaces are defined in `ACTIVEX.PAS`. You don't have to do it manually thankfully. The Automation Wizard will do all this for you.

So let's begin. Open a New Application and drop a `TMemo` on the form. Now go to **New|Automation Object** to start the Automation Wizard. Here you are asked for a `coClass` name. This example will use `SimpleEventServer`. Before clicking OK, check the box marked **Generate Event Support Code**. This is the implementation for your `IConnectionPointContainer` related things as described in the above paragraph (POOF!!! Delphi Magic!!!). The result will be a unit containing your `TSimpleServer` object and its `coClass` definitions. Delphi will also generate a Type Library that includes the typical dual incoming interfaces `ISimpleEventServer` and `ISimpleEventServerDisp`, plus one you have not seen before, your outgoing events interface `ISimpleEventServerEvents`.

`ISimpleEventServer` now needs to expose a method for our client to call. Open the Type Library Editor (if it is not open already) and add a method `CallServer()`. Now we need to define an event for the server to fire, so we will add the method `EventFired()` to the `ISimpleEventServerEvents`. With this done, click on the Refresh Implementation button and then go back to the source unit for your server. You'll notice that a method has been added. When the client calls the server, the server will inform the user via its memo, then fire the `EventFired` event. Implement it as follows:

```
procedure CallServer; safecall;
begin
  Form1.Memo1.Lines.Add('I have been called by a client');
  if FEvents <> nil then
  begin
    FEvents.EventFired;
    Form1.Memo1.Lines.Add('I am firing an Event');
  end;
end;
```

*NOTE: FEvents is our outgoing interface. Notice that we check it before we fire the event off it. This ensures that a client is actually listening. If it has not been advised to a client sink, it will return NIL.*

With that squared away, the Server is complete! Build and run it once to register it with the system.

## THE CLIENT

Start a new application and add a `TMemo` and a button. Now go to the Unit source and add the TLB unit of your server to the `USES` clause so we can have access to those types and methods. Your main form object will need fields to hold the interfaces to the Server object and the event sink. Declare them in the private field as follows:

```

...
private
    { Private declarations }
    FServer: ISimpleEventServer;
    FEventSink: IUnknown;
    FConnectionToken: integer;
...

```

Once this groundwork is complete, we can start on the only difficult task in COM events: The implementation of the Event Sink. It starts by defining the Event Sink Object, which is an automation object, and therefore must implement IDispatch. The job of the Event Sink is to delegate calls to itself by the server through its `Invoke()` method. It then calls the local implementation for the event. To enforce a level of separation here, we will leave these implementations in the main unit and hold a reference to the main form in the sink object to call them off of. Here is the Event Sink Definition:

```

TEventSink = class(TInterfacedObject, IUnknown, IDispatch)
private
    FController: TForm2;
    {IUnknown methods}
    function QueryInterface(const IID: TGUID; out Obj): HRESULT;
stdcall;
    {IDispatch}
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
HRESULT; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
        NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT;
stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID:
Integer;
        Flags: Word; var Params; VarResult, ExcepInfo, ArgErr:
Pointer): HRESULT; stdcall;
public
    constructor Create(Controller: TForm2);
end;

```

Most of these methods do not need to be implemented. Making them simply return `S_OK` will do fine, for all except `QueryInterface()` and `Invoke()`. These methods are used by the server to obtain interfaces and call your event handlers.

```

function TEventSink.QueryInterface(const IID: TGUID; out Obj): HRESULT;
stdcall;
begin
    if GetInterFace(IID, Obj) then

```

```

    Result := S_OK
else if IsEqualIID(IID, ISimpleEventServerEvents) then
    Result := QueryInterface(IDispatch, Obj)
else
    Result := E_NOINTERFACE;
end;

```

This method first takes care its own IDispatch and IUnknown, then it recurses to get the outgoing interface if being queried for ISimpleEventServerEvents.

```

function TEventSink.Invoke(DispID: integer; const IID: TGUID;
LocaleID: integer; Flags: Word; var Params; VarResult, ExcepInfo,
ArgErr: Pointer): HRESULT;
begin
    Result := S_OK;
    case DispID of
        1: FController.OnEventFired;
    end;
end;

```

The case statement above would, of course, have more statements if we had more events, but this interface and sink only support one. Note that Invoke ( ) calls our handler through a local reference to the main form object where our event handler resides. The Event Sink constructor should handle setting this up:

```

constructor TEventSink.Create(Controller: TForm2);
begin
    inherited Create;
    FController := Controller;
end;

```

With the methods and objects in place, all that is left is to connect sink to source! We will do this in the Client's OnCreate event handler:

```

procedure TForm2.FormCreate(Sender: TObject);
begin
    FServer := CoSimpleEventServer.Create;
    FEventSink := TEventSink.Create(form2);
    InterfaceConnect(FServer, ISimpleEventServerEvents, FEventSink,
FconnectionToken);
end;

```

*NOTE: Remember the FEventSink is an IUnknown, and we are receiving and interface from the*

*TEventSink constructor, not a standard reference. This will allow us to advise a connection point to it.*

First, we create our server using its coClass. Then we create an instance of our Event sink. As I said earlier, the `InterfaceConnect()` method is easier to use, but you lose some functionality and understanding. You can use it here by passing in your server's interface, the IID of the events interface you are querying for, the `IUnknown` of your newly created event sink, and an integer representing the connection. You **MUST** hold onto this integer if you wish to properly unadvise the connection. You can disconnect by simply calling `InterfaceDisconnect()` using the token integer you received from your `InterfaceConnect()` call.

Now compile and run the client. The server should start and that's it! You are now connected and listening for events! Click the client's button to test your event.

## **SOURCE UNITS EXAMPLES**

Below are the units Server Object and Client Object units. Ommited is the Main form of the server.

\*\*\*\*\*

```
-----source-----  
  
//SrvUnit2.pas  
//Contains the Automation Server Object  
unit SrvUnit2;  
  
interface  
  
uses  
    ComObj, ActiveX, AxCtrls, Classes, SrvEvent_TLB, StdVcl, Srvunit1;  
  
type  
    TSimpleEventServer = class(TAutoObject, IConnectionPointContainer,  
ISimpleEventServer)  
    private  
        { Private declarations }  
        FConnectionPoints: TConnectionPoints;  
        FConnectionPoint: TConnectionPoint;  
        FEvents: ISimpleEventServerEvents;  
        { note: FEvents maintains a *single* event sink. For access to  
more  
        than one event sink, use FConnectionPoint.SinkList, and iterate  
        through the list of sinks. }  
    public  
        procedure Initialize; override;
```

```

protected
  { Protected declarations }
  property ConnectionPoints: TConnectionPoints read
FConnectionPoints
  implements IConnectionPointContainer;
  procedure EventSinkChanged(const EventSink: IUnknown); override;
  procedure CallServer; safecall;
end;

implementation

uses ComServ;

procedure TSimpleEventServer.EventSinkChanged(const EventSink:
IUnknown);
begin
  FEvents := EventSink as ISimpleEventServerEvents;
end;

procedure TSimpleEventServer.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoint := FConnectionPoints.CreateConnectionPoint(
      AutoFactory.EventIID, ckSingle, EventConnect)
  else FConnectionPoint := nil;
end;

procedure TSimpleEventServer.CallServer;
begin
  Form1.Mem1.Lines.Add('I have been called by a client');
  if FEvents <> nil then
    begin
      FEvents.EventFired;
      Form1.Mem1.Lines.Add('I am firing an Event');
    end;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TSimpleEventServer,
Class_SimpleEventServer,
  ciMultiInstance, tmApartment);
end.

```

```

//clientunit.pas
unit ClientUnit;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, SrvEvent_TLB, ActiveX, ComObj, StdCtrls;

type
  TForm2 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
    FServer: ISimpleEventServer;
    FEventSink: IUnknown;

    FConnectionToken: integer;
  public
    { Public declarations }
    procedure OnEventFired;
  end;

  TEventSink = class(TInterfacedObject, IUnknown, IDispatch)
  private
    FController: TForm2;
    {IUnknown methods}
    function QueryInterface(const IID: TGUID; out Obj): HRESULT;
  stdcall;
    {IDispatch}
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
  HRESULT; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT;
  stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID:

```

```

Integer;
    Flags: Word; var Params; VarResult, ExcepInfo, ArgErr:
Pointer): HRESULT; stdcall;
    public
        constructor Create(Controller: TForm2);
    end;

var
    Form2: TForm2;

implementation

{$R *.dfm}

procedure TForm2.OnEventFired;
begin
    Mem1.Lines.Add('I have recieved an event');
end;

constructor TEventSink.Create(Controller: TForm2);
begin
    inherited Create;
    FController := Controller;
end;

function TEventSink.Invoke(DispID: integer; const IID: TGUID;
LocaleID: integer; Flags: Word; var Params; VarResult, ExcepInfo,
ArgErr: Pointer): HRESULT;
begin
    Result := S_OK;
    case DispID of
        1: FController.OnEventFired;
    end;
end;

function TEventSink.QueryInterface(const IID: TGUID; out Obj): HRESULT;
stdcall;
begin
    if GetInterFace(IID, Obj) then
        Result := S_OK
    else if IsEqualIID(IID, ISimpleEventServerEvents) then
        Result := QueryInterface(IDispatch, Obj)
    else
        Result := E_NOINTERFACE;
    end;
end;

```



```

function TEventSink.GetTypeInfoCount(out Count: Integer): HRESULT;
begin
    Result := S_OK;
end;
function TEventSink.GetTypeInfo(Index, LocaleID: Integer; out
TypeInfo): HRESULT;
begin
    Result := S_OK;
end;
function TEventSink.GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT;
begin
    Result := S_OK;
end;

procedure TForm2.FormCreate(Sender: TObject);
begin
    FServer := CoSimpleEventServer.Create;
    FEventSink := TEventSink.Create(form2);
    InterfaceConnect(FServer, ISimpleEventServerEvents, FEventSink,
FConnectionToken);
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
    Mem1.Lines.Add('I am calling the Server');
    FServer.CallServer;
end;

procedure TForm2.FormDestroy(Sender: TObject);
begin
    InterfaceDisconnect(FServer, ISimpleEventServer, FConnectionToken);
    FServer := nil;
    FEventSink := nil;
end;

end.

```

Last Modified: 29-MAR-01

## Article #27132: Creating a file browser for Linux using Kylix

### Creating a file browser for Linux using Kylix

By [Corbin Dunn](#)

Delphi Developer Support

One of the first things that I wanted to do with Kylix was create a simple Linux file browser such as this:



I am only going to cover the general outline of how I made this project, so [download the complete source](#) from CodeCentral. I realize that it is a zip file, and Linux users like tar.gz'd files, but CodeCentral stores everything as zips. The [KDE Archiver](#) program allows you to easily unzip files on Linux.

I started the project by putting down a TTreeView on the form and set its Align property to allLeft. The TTreeView will hold the current directory tree. Next, I put down a TSplitter, so the user could resize the components at run time. Then, I put down a TListView, which will hold the files for the currently selected directory.

The first problem is finding directories. I could find all directories on the entire system when the application starts, but this would be very time consuming. Plus, the user may not expand all directories. So, I need to keep track of whether or not a user has opened up a given directory. TTreeNode's have a Data property that allows the programmer to store whatever information they want in it. I decided to create a new TTreeFolderData class that stores information about a given TTreeNode:

```
TTreeFolderData = class
private
  FPath: string;
  FHasSubFolders: Boolean;
  FOpened: Boolean;
public
  constructor Create(const Path: string);
  destructor Destroy; override;
  property Path: string read FPath write FPath;
  property HasSubFolders: Boolean read FHasSubFolders write FHasSubFolders;
  property Opened: Boolean read FOpened write FOpened;
end;
```

In my Form's OnCreate event, I call a LoadTreeDirectory procedure that fills in the root TTreeNode of the TTreeView:

```
procedure TMainForm.LoadTreeDirectory;
var
  RootNode: TTreeNode;
  TreeFolderData: TTreeFolderData;
begin
  trvwLocal.Items.BeginUpdate; // trvwLocal is my TTreeView on the left
  try
    trvwLocal.Items.Clear;
    RootNode := trvwLocal.Items.Add(nil, 'Root');
    RootNode.ImageIndex := 0;
    TreeFolderData := TTreeFolderData.Create(PathDelim);
    TreeFolderData.Opened := True;
    RootNode.Data := TreeFolderData;
    AddSubFolders(PathDelim, RootNode);
    RootNode.Expand(False);
  finally
    trvwLocal.Items.EndUpdate;
  end;
end;
```

The **AddSubFolders** procedure simply takes a given path and node, and finds all folders that exist in it. I simply use our standard FindFirst and FindNext procedures, looking for faDirectory's:

```
procedure TMainForm.AddSubFolders(const Directory: string;
  ParentNode: TTreeNode);
var
  SearchRec: TSearchRec;
  Attributes: Integer;
```

```

NewNode: TTreeNode;
TreeFolderData: TTreeFolderData;
begin
  trvwLocal.Items.BeginUpdate;
  try
    // First, delete any subfolders from the parent node.
    if ParentNode <> nil then
      ParentNode.DeleteChildren;

    Attributes := faAnyFile;
    if FindFirst(IncludeTrailingPathDelimiter(Directory) + '*',
      Attributes, SearchRec) = 0 then
    begin
      repeat
        if (SearchRec.Attr and faDirectory) > 0 then
          if (SearchRec.Name <> '.') and (SearchRec.Name <> '..') then
            begin
              NewNode := trvwLocal.Items.AddChild(ParentNode, SearchRec.Name);
              NewNode.ImageIndex := 0;
              TreeFolderData := TTreeFolderData.Create(
                IncludeTrailingPathDelimiter(Directory) + SearchRec.Name);
              NewNode.Data := TreeFolderData;

              if HasSubFolder(TreeFolderData.Path) then
                begin
                  TreeFolderData.HasSubFolders := True;
                  // Add a fake child so the + appears
                  trvwLocal.Items.AddChild(NewNode, '+').ImageIndex := 0;
                end
              end;
            until FindNext(SearchRec) <> 0;
            FindClose(SearchRec);
          end;
        finally
          trvwLocal.Items.EndUpdate;
        end;
      end;
    end;
end;

```

One thing to notice is the call to my **HasSubFolder** function. By checking to see if a given folder has a subfolder, I can add a fake child node so the + appears in the TTreeView. To dynamically fill in the sub folders when the user expands the plus, I have the following code in the TTreeView OnExpanded event:

```

procedure TMainForm.trvwLocalExpanded(Sender: TObject; Node: TTreeNode);
var
  TreeFolderData: TTreeFolderData;
begin
  TreeFolderData := TTreeFolderData(Node.Data);
  if not TreeFolderData.Opened then
    begin
      TreeFolderData.Opened := True;
    end;
end;

```

```

    AddSubFolders(TreeFolderData.Path, Node);
end;
Node.ImageIndex := 1;
end;

```

Finally, when a directory is selected in the TTreeView, I need to update the TListView on the right with the selected directories contents. In the TTreeView's OnChange event I use the TTreeFolderData to find what path the current node contains, and then fill in the TListView with my FindFiles event:

```

procedure TMainForm.trvwLocalChange(Sender: TObject; Node: TTreeNode);
begin
    if Node <> nil then
        FindFiles(TTreeFolderData(Node.Data).Path);
end;

procedure TMainForm.FindFiles(const Directory: string);
var
    SearchRec: TSearchRec;
    Attributes: Integer;
    Pixmap: QPixmapH;
begin
    // Find files for the local folder to be added to the list view
    lstvwLocal.Items.BeginUpdate;
    try
        lstvwLocal.Items.Clear;
        Attributes := faAnyFile;
        if FindFirst(IncludeTrailingPathDelimiter(Directory) + '*',
            Attributes, SearchRec) = 0 then
            begin
                repeat
                    if (SearchRec.Attr and faDirectory) = 0 then
                        begin
                            with lstvwLocal.Items.Add do
                                begin
                                    Caption := SearchRec.Name;
                                    ImageIndex := 0;
                                    SubItems.Add(IntToStr(SearchRec.Size));
                                    SubItems.Add(DateTimeToStr(FileDateToDateTime(SearchRec.Time)));
                                    // Work around for a bug in setting the image to nothing
                                    // Search the community site for information on how to modify
                                    // the VCL to have this fix.
                                    Pixmap := QPixmap_create;
                                    QListViewItem_setPixmap(Handle, 1, Pixmap);
                                    QListViewItem_setPixmap(Handle, 2, Pixmap);
                                    QPixmap_destroy(Pixmap);
                                end;
                            end;
                        until FindNext(SearchRec) <> 0;
                        FindClose(SearchRec);
                    end;
                end;
            end;

```

```
finally  
    lstvwLocal.Items.EndUpdate;  
end;  
end;
```

That should pretty much be it! There are some other minor things I am leaving out, so [download the complete source](#) from CodeCentral and dig into it.

Last Modified: 29-MAR-01

## Article #27182: CLX styles demo for Kylix

### Clx styles demo for Kylix

By [Corbin Dunn](#)

Delphi Developer Support

Qt styles are what a lot of people know as "Themes". You can easily control the entire look of your application by providing your own custom drawing events, or using one of the built in default styles. This demo gives a complete example of how to do custom drawing and change the theme.

**First, download the complete source from Borland CodeCentral.** CodeCentral only supports zip files, and under Linux I recommend using [KDE's Archiver](#) to open the zip. In this article I will only cover some of the highlights of styles.

First, take a look at the **Default Styles** menu item. This allows you to easily select any of the built in Qt styles, such as the QtSGI theme seen below:



The code to set a theme is quite simple:

```
procedure TMainForm.mnuQtSgiClick(Sender: TObject);  
begin  
    Application.Style.DefaultStyle := dsQtSGI;  
    (Sender as TMenuItem).Checked := True;  
    RefreshChildren(True);  
end;
```

The **RefreshChildren** function is one that I wrote. It removes any custom bitmaps that the other options may have set. Control-click **dsQtSGI** to see all the other predefined themes.

Next, take a look at the ability to set a background bitmap for controls:



It is pretty easy to set a bitmap for all controls using the **Application.Palette**. Calling `Application.Palette.SetBitmap` takes three parameters. The first is the color group. I only set the `cgActive` color group, and the screen shot is not active, so some controls are not painted with the bitmap. A `TColorRole` is the next parameter which tells the Palette what role the bitmap should be applied to (such as the foreground, background, shadows, etc.). Finally, the third parameter is the bitmap you want to assign to it. I have a generic `TMenuItem` `OnClick` event that



does this for me. Also note how I manually set some bitmaps to controls. You could give each control a different bitmap, if you like.

```
procedure TMainForm.mnuSetBitmapsClick(Sender: TObject);  
var  
    ColorRole: TColorRole;  
    Bitmap: TBitmap;  
begin  
    if Sender = mnuLiteWood then  
        Bitmap := FBitLiteWood // These bitmaps are loaded in the FormCreate  
    else if Sender = mnuDarkWood then  
        Bitmap := FBitDarkWood  
    else if Sender = mnuShinyWood then  
        Bitmap := FBitShinyWood  
    else  
        Bitmap := FBitLiteGranite;  
  
    Application.Palette.BeginUpdate;  
    try  
        for ColorRole := crForeground to crHighlightText do  
            Application.Palette.SetBitmap(cgActive, ColorRole, Bitmap);  
    finally  
        Application.Palette.EndUpdate;  
    end;  
    // Manually assign bitmaps to some controls that don't do it  
    // automatically.  
    chkbxNormal.Bitmap := Bitmap;  
    chkbxkTriState.Bitmap := Bitmap;  
    TrackBar1.Bitmap := Bitmap;  
    RefreshChildren(False);  
end;
```

The next thing to take a look at is the **Application.Style** events. Take a look at all the available events in QStyle.pas. Doing drawing in these events controls the entire look and feel of your application. For example, when the **Draw Menu Items** menu item is clicked, I set the events to some of my own procedures:

```
procedure TMainForm.mnuDrawMenuItemsClick(Sender: TObject);  
begin  
    mnuDrawMenuItems.Checked := not mnuDrawMenuItems.Checked;  
    if mnuDrawMenuItems.Checked then  
        begin  
            Application.Style.DrawMenuFrame := DrawMenuFrame;  
            Application.Style.BeforeDrawMenuItem := BeforeDrawMenuItem;  
        end  
    else  
        begin  
            Application.Style.DrawMenuFrame := nil;  
            Application.Style.BeforeDrawMenuItem := nil;  
        end  
    end;
```

```
end;  
end;
```

The other menu items in **Custom Styles** have similar code.

The actual implementation is pretty easy. All you have to do is write code using the standard TCanvas and TFont:

```
procedure TMainForm.DrawMenuFrame(Sender: TObject; Canvas: TCanvas;  
  const R: TRect; LineWidth: Integer; var DefaultDraw: Boolean);
```

```
begin  
  Canvas.Pen.Color := clRed;  
  Canvas.Rectangle(R);
```

```
end;
```

```
procedure TMainForm.BeforeDrawMenuItem(Sender, Source: TObject;  
  Canvas: TCanvas; Highlighted, Enabled: Boolean; const Rect: TRect;  
  Checkable: Boolean; CheckMaxWidth, LabelWidth: Integer;  
  var DefaultDraw: Boolean);
```

```
var  
  Item: TMenuItem;  
  TempRect: TRect;  
  Text: string;  
  TextLeft: Integer;  
  TextTop: Integer;  
  AmpPos: Integer;
```

```
begin  
  DefaultDraw := False;  
  Item := Source as TMenuItem;
```

```
if Highlighted then
```

```
begin  
  Canvas.Brush.Color := clBlack;  
  Canvas.Font.Color := clWhite;
```

```
end
```

```
else
```

```
begin  
  Canvas.Brush.Color := clWhite;  
  Canvas.Font.Color := clBlack;
```

```
end;
```

```
Canvas.FillRect(Rect);
```

```
Canvas.Pen.Style := psSolid;  
TempRect := GetNewRect(Rect);  
Frame3d(Canvas, TempRect, clBlack, clGray, 1);
```

```
TextTop := Rect.Top + (Rect.Bottom - Rect.Top - Canvas.TextHeight('X'))  
div 2;  
if Item.Checked then
```

```

Canvas.Draw(Rect.Left, TextTop, FBitMenuCheck);

AmpPos := Pos('&', Item.Caption);
if AmpPos <> 0 then
begin
  // Output the part before the &
  Text := Copy(Item.Caption, 1, AmpPos-1);
  TextLeft := Rect.Left + FBitMenuCheck.Width;
  Canvas.TextOut(TextLeft, TextTop, Text);
  TextLeft := TextLeft + Canvas.TextWidth(Text);
  // Output the "shortcut character bolded
  Canvas.Font.Style := Canvas.Font.Style + [fsBold];
  Canvas.TextOut(TextLeft, TextTop, Item.Caption[AmpPos+1]);
  TextLeft := TextLeft + Canvas.TextWidth(Item.Caption[AmpPos+1]);
  // Output the rest of the text past the shortcut character
  Canvas.Font.Style := Canvas.Font.Style - [fsBold];
  Text := Copy(Item.Caption, AmpPos + 2, Length(Item.Caption) - AmpPos -
1);
  Canvas.TextOut(TextLeft, TextTop, Text);
end
else
  Canvas.TextOut(Rect.Left + FBitMenuCheck.Width, TextTop, Item.Caption);

  if Item.Count > 0 then
    Canvas.Draw(Rect.Right - FBitArrowBmp.Width - 2, Rect.Top + 2,
FBitArrowBmp);
end;

```

For more examples of custom drawing, [download the complete source](#) from Borland CodeCentral.

Last Modified: 13-APR-01

## Article #27363: Using queued components in Delphi

Queued Components are feature of COM+ based on Microsoft Message Queuing Services (MSMQ). They provide an easy way to invoke and execute components asynchronously. Processing can occur without regard to the availability or accessibility of either the sender or the receiver.

In COM terms, a queue is a storage area that saves messages for later retrieval. Queuing provides a mechanism for connectionless communication. That is, the sender and receiver are not connected directly and communicate only through queues. Queuing provides a way to hold the information until the receiver is ready to obtain it. The sender and receiver are indirectly communicating so that each can operate independently, unaffected by the other. Here is a simple example of creating and using a queued component in Delphi 6.

First we will **create a server**. All methods in your interfaces must use IN parameters only. Methods can not have return values. This is because there is no "real-time" connection between the client and server. Since you don't know exactly when the call will be processed the client cannot sit waiting for a response from the server.

In Delphi 6 go to File | New | Other | ActiveX and open a new ActiveX Library. Next go back to this same menu option and add an Automation Object. Select a name for your server object. I used QdComponent. Leave the other options at the default and click OK. You should now see the type library editor. Add a new method here by clicking the interface IQdComponent and then clicking the green icon at the top for adding a new method. Call it Task1 and add a parameter called SendMessage. Make this parameter of type BSTR (WideString in Delphi). If you have your type library options set to show Pascal code (in Environment options) then this is what the text for your method should look like:

```
procedure Task1(TaskMessage: WideString) [dispid $00000001]; safecall;
```

Save your project. I called my project MyQdServer and my implementation unit (Unit1 by default) I named QdServer. Fill in the implementation code for the procedure TQdComponent.Task1 in unit QdServer. My component will simply display a message that is sent via the MessageString parameter. Compile, save all and close. Here is what the server unit looks like:

```
unit QdServer;  
{ $WARN SYMBOL_PLATFORM OFF }  
interface  
uses  
  ComObj, ActiveX, MyQdServer_TLB, StdVcl, Dialogs;  
type  
  TQdComponent = class(TAutoObject, IQdComponent)  
  protected  
    procedure Task1(const TaskMessage: WideString); safecall;
```

```

    { Protected declarations }
end;
implementation
uses ComServ;
procedure TQdComponent.Task1(const TaskMessage: WideString);
begin
    ShowMessage(TaskMessage)
end;
initialization
    TAutoObjectFactory.Create(ComServer, TQdComponent,
Class_QdComponent,
    ciMultiInstance, tmApartment);
end.

```

### Installing the server

In the control panel go to the Component Services tool. Expand the tree in order to see "COM+ Application". Right click on this folder and choose New | Application. Choose "Create an empty application" and follow the prompts. When you are done find your new application listed below, right-click on it and go to properties. Go to the Queuing tab and check both check boxes - Queued and Listen. Listen means that any incoming messages on this queue will be processed immediately when it is active.

Under your new application right click on Components and choose New | Component. Choose "Install new component(s)" and follow the prompts to find and install your DLL. Next expand the tree to show the interfaces and right click on IQdComponent to show the properties. Click the Queuing tab and check "Queued". If you see an error message Stating that MSMQ is not running then you probably do not have it installed. It is not installed by default and can be installed by going to Add/Remove programs if you have your OS CDs.

### Creating a Client

Create a new Delphi Application. Add MyQdServer\_TLB to your uses clause. Add a TButton and a TEdit. You will need to create your object when your application starts. To do this you will use a Moniker which specifies the component information. The syntax for this is shown in the FormShow event handler below. This moniker can also contain *Server.InterfaceName* instead of the GUID/CLSID.

```

unit Unit1;
interface
uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
    Dialogs, StdCtrls, ComObj, ActiveX, MyQdServer_TLB;
type
    TForm1 = class(TForm)
        Edit1: TEdit;
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    end;

```

```

    procedure FormShow(Sender: TObject);
private
    { Private declarations }
    QInterface : IQdComponent;
public
    { Public declarations }
end;
var
    Form1: TForm1;
implementation
{$R *.dfm}
Function NewCoGetObject(pazName: PWideChar; pBindOptions: PBindOpts;
    const iid: TIID; out ppv): HRESULT; stdcall; external 'ole32.
dll'
    name 'CoGetObject';
procedure TForm1.Button1Click(Sender: TObject);
begin
    QInterface.Task1(Edit1.Text);
    Edit1.Clear;
end;
procedure TForm1.FormShow(Sender: TObject);
var
    QMoniker : PWideChar;
begin
    QMoniker := 'Queue:/new:{A7ECB8CA-9BBB-4BAB-B018-33210095AE34}';
    OleCheck(NewCoGetObject(QMoniker, nil, IQdComponent, QInterface));
end;
end.

```

Run the client and send a text string. You will notice the message does not appear because you have not started the server. There are a few ways to start the server. The simplest is to go back to the component services in control panel, find your server, right click on it and choose 'Start'. After you start the server the message should appear.

Last Modified: 25-JUN-01

## Article #27366: Calculating Business Days on Oracle

There are quite a number of examples already on the web for calculating business days on Oracle. Here are some links to what I found:

<http://www.arrowsent.com/oratip/tip6.htm>

<http://www.orsweb.com/memberarea/downloads/source/87.htm>

<http://dco-proxima.dco.pima.edu/oracle/tfts/PRE00136.HTM>

However, none of these did exactly what I needed for a job I was doing. What's different about my environment is the need to consider multiple geographic regions and to consider work hours and that different regions might have different work hours.

Here are the deciding factors for how this function decides what to do with timestamp after hours (both weekends and during the week)

- If the start date is after the end of the work day, and before midnight, it is advanced to the beginning of next work day.
- If the start date is before the beginning of the day (and therefore after midnight), it is advanced to the beginning of the current work day.
- If the start date is on a Saturday, it is advanced to the beginning of work day on the following Monday.
- If the start date is on a Sunday, it is advanced to the beginning of work day on the following Monday.
- If the end date is after the end of the work day, and before midnight, it is moved to the end of the day.
- If the start date is before the beginning of the day (and therefore after midnight), it is advanced to the beginning of the current work day.
- If the start date is on a Friday or Saturday it is moved to end of the previous Friday.

This function does not consider holidays, but could be modified to do so if you had a holiday table.

```
CREATE OR REPLACE FUNCTION bdays(start_date IN DATE, end_date IN
DATE, region IN CHAR)
RETURN NUMBER
IS
    retval NUMBER(15,7);
    new_start_date date;
    new_end_date date;
    bdaystart number(15,15);
    bdayend number(15,15);
BEGIN
    new_start_date := start_date;
```

```

new_end_date := end_date;
-- set defaults for business day start and end. Can be overridden per
region
bdaystart := 7/24;
bdayend := 17/24;
if region='Europe' then
    new_start_date := new_start_date + 9/24;
    new_end_date := new_end_date + 9/24;
    bdaystart := 9/24;
    bdayend := 18.5/24;
end if;
if region='Asia-Pac' then
    new_start_date := new_start_date + 15/24;
    new_end_date := new_end_date + 15/24;
end if;
--Start After end of day, make start be start of next day
if new_start_date-trunc(new_start_date)>bdayend then
    new_start_date := TRUNC(new_start_date+1)+bdaystart;
end if;
--Start before start of day, make start be start of same day
if new_start_date-trunc(new_start_date) < bdaystart then
    new_start_date := TRUNC(new_start_date) + bdaystart;
end if;
--Start Saturday, make start be Monday start of day
if to_char(new_start_date,'D')=7 THEN
    new_start_date := TRUNC(new_start_date+2)+bdaystart;
END IF;
--Start Sunday, make start be Monday start of day
if to_char(new_start_date,'D')=1 THEN
    new_start_date := TRUNC(new_start_date+1)+bdaystart;
END IF;
-- end after end of day, make end be end of day same day
if new_end_date-trunc(new_end_date) > bdayend then
    new_end_date := trunc(new_end_date) + bdayend;
end if;
-- end before start of day, make end be start of day the same day
if new_end_date-trunc(new_end_date) < bdaystart then
    new_end_date := trunc(new_end_date) + bdaystart;
end if;
--end on Saturday, make it be the end of the day on Friday
if to_char(new_end_date,'D')=7 then
    new_end_date := trunc(new_end_date-1) + bdayend;
end if;
--end on Sunday, make it be the end of the day on Friday
if to_char(new_end_date,'D')=1 then

```



```
    new_end_date := trunc(new_end_date-2) + bdayend;
end if;
--factor out weekend days
retval := new_end_date - new_start_date -
((TRUNC(new_end_date,'D') - TRUNC(new_start_date,'D'))/7)*2;
-- if holidays were to be calculated, the calculation would go here
-- if end is during nonbusiness hours, difference could be negative
if retval < 0 then
    retval := 0;
end if;
RETURN(retval);
END;
```

Last Modified: 04-JUN-01

## Article #27462: Streaming Bitmaps and other Binary Data to BLOB Fields

If you are reading this article you have probably discovered that reading and writing to and from BLOB fields is not as straight forward as their counterparts, integer and character. The wonderful implicit assignments to integer and character fields just don't exist. Don't despair. Accessing data in a BLOB is not as simple as other field types, but is by no means difficult.

### TBLOBStream

The main difference in accessing BLOB data is the use of streams for the actual data transfer. Once you have a reference to your Dataset's BLOB field, you open a stream to it by passing that reference to TBLOBStream's Create method. TBLOBStream descends from TStream and encapsulates the operations that allow your applications to read from or write to Binary Large Object(BLOB) fields.

There are a few things that you should take into consideration when creating your BLOB streams(s). One is that you should make sure to create a new instance of TBLOBStream for every field you plan to read from and free it when you are done. That other is that even though TBLOBStream lets you create instances of it to use with any kind of DataSet, it is best to instead call the particular DataSet's CreateBLOBStream method. It works exactly like TBLOBStream.Create, but it instantiates an object specific for that Dataset(TTable,TQuery). My examples will uses this method.

### A Simple Example: Streaming Bitmaps

While BLOBs can hold many different kinds of data, including MEMOs and... well just about anything you want, my example will focus on the streaming of bitmap files. These concepts hold true for all types of data.

### Using the BDE

First, start a new Delphi application and drop on your database components for your connection. This will include a TDataBase and TQuery. Hook your Query to your Database object and set it's RequestLive property to true, since we will be inserting data. Now we will write your bitmap image to you database field. First we will want a Bitmap to work with. After declaring a TBitmap, create it and load a file with your favorite picture into it.

```
MyFavoriteBitmap := TBitmap.Create;  
MyFavoriteBitmap.LoadFromFile('drive:pathMyFavoriteBitmap.bmp');
```

Once you have a Bitmap ready and waiting, you can go ahead and create your BLOB stream. To do this, you will need to get a reference to your blobfield from the query. If you know the name for the field, TDataSet.FindByName is the easiest. Now Pass this reference into CreateBLOBStream along with the BlobStreamMode(bmRead,bmWrite,bmReadWrite), which will return reference to a TStream. We are going to use this to actually stream the file using TBitmap's SaveToStream method.

```

procedure SavePictureToDatabase;
var
  BlobField: TField;
  BS: TStream;
begin
  with Query1 do
    begin
      Insert;
      BlobField := FieldByName('picturefield');
      BS := CreateBlobStream(BlobField,bmWrite);
      Bitmap.SavetoStream(BS);
      Post;
    end;
  end;

```

**Using dbExpress** Those of you using the new dbExpress components will find the process is identical with a few minor exceptions. If you are using TSQLQuery or TSQLTable, you are enjoying their extremely fast execution for your reporting applications. But since they do not cache their data, there is no way for you to edit any records, let alone your BLOBS. Don't forget that you must use TSQLClientDataSet instead of TSQLDataSet's to achieve this functionality. In code, the difference is trivial for writing your picture to a database.

```

procedure SavePictureToDatabase;
var
  BlobField: TField;
  BS: TStream;
begin
  with SQLClientDataSet1 do
    begin
      Insert;
      BlobField := FieldByName('picturefield');
      BS := CreateBlobStream(BlobField,bmWrite);
      Bitmap.SavetoStream(BS);
      Post;
    end;
  end;

```

When you are ready to read your bitmap back out, use the built in TBitmap method and you are set.

```

Blobfield := FieldbyName('picturefield');
BS := CreateBlobStream(BlobField,bmReadWrite);
MyFavoriteBitmap.LoadFromStream(BS);

```

**Other types of binary data**

When moving in large chunks of data that do not have convenient loading and saving methods like TBitmap, you must employ the TStream methods (Read, ReadBuffer, Write, WriteBuffer) manually. You can even stream components and their properties using WriteComponent and ReadComponent, were you so inclined.

```
var
  MyComponent: TMyComponent //any TComponent
begin
  MyComponent := TMyComponent.Create;
  MyComponent.aProperty := aValue;
  BS := CreateBlobStream(Blobfield,bmReadWrite);
  BS.WriteComponent(MyComponent);
end;
```

There you have it. A quick intro to using BLOBs to store binary data. I think the moral of the story is quite clear: Don't take candy from strangers', kids.

Last Modified: 03-JUL-01

## Article #27508: How can I change a folders Icon in Delphi?

### QUESTION:

How change the appearance of a folder icon?

### ANSWER:

The process of changing a folder's icon is pretty easy and there are a couple of ways that you can do it. In this article I'll show you how you can do it programmatically. I am only going to cover the general outline of how I made this project, so you can [download the complete source](#) from CodeCentral.



The first step is to create a Desktop.ini file and place it in the directory that you want to have a custom icon. In my project I used a TIniFile and pass it the path of the directory to which I am going to change the Icon.

Next you will want to start writing out <key>=<value> pairs to the .ini file. Before doing this you may want to add the option to use custom Folder.htt templates. If this is the case then you will want to write out the following <key>=<value> pairs (don't mind the GUID looking things, don't change them either):

```
[ExtShellFolderViews]
Default={5984FFE0-28D4-11CF-AE66-08002B2E1262}
{5984FFE0-28D4-11CF-AE66-08002B2E1262}={5984FFE0-28D4-11CF-AE66-08002B2E1262}
[ {5984FFE0-28D4-11CF-AE66-08002B2E1262} ]
PersistMoniker=file://Folder.htt
```

In Delphi the code would look something like this:

```
with iniFile do
begin
  //Next three lines are window specific and shouldn't be changed..
  WriteString('ExtShellFolderViews',
'Default', '{5984FFE0-28D4-11CF-AE66-08002B2E1262}' );
  WriteString('ExtShellFolderViews', '{5984FFE0-28D4-11CF-AE66-08002B2E1262}',
' {5984FFE0-28D4-11CF-AE66-08002B2E1262}' );
  WriteString('{5984FFE0-28D4-11CF-AE66-08002B2E1262}', 'PersistMoniker', 'file://
Folder.htt');
  UpdateFile; //For Windows 98/95
end;
```

If you don't want to worry about Folder.htt templates then ignore the code above. The important part is to specify the icon and its index. In the Desktop.ini file the <key>=<value> pairs should look something like this:

```
[.ShellClassInfo]
IconFile=C:\LocationofFolder.ico
IconIndex=0
InfoTip=Delphi is the coolest IDE ever!
```

The IconFile value should be the path to a .dll, .ico, or .exe. If the icon resides within a dll then IconIndex value should match its entry within the dll. In Delphi the code to do this is as follows:

```
with iniFile do
begin
  //These are the lines that will change the icon
  WriteString('.ShellClassInfo', 'IconFile', editIconPath.Text);
  WriteString('.ShellClassInfo', 'IconIndex', editIconIndex.Text);
  WriteString('.ShellClassInfo', 'InfoTip', 'Use Delphi because it
rocks!');
  UpdateFile;
end;
```

Now that the Desktop.ini file has been created you will need to change the attributes of the folder and apply the system flag. I found that I had to set the system flag for both the folder and the folders parent in order to view the icon correctly. (This way if a subfolder had a custom icon, the icon would show up in the left hand pain of Windows Explorer.) To set the file attribute I used *SetFileAttribute()*. Here is an example of how to do it in Delphi:

```
//Next we need to set the system attribute for the folder and its parent
SetFileAttributes(PChar(edFolderPath.Text), FILE_ATTRIBUTE_SYSTEM);
if Length(edFolderPath.Text) > 3 then //If not at the root of a disk...
begin
  //LastChar is a funtion that I wrote which returns an index of the last occurance
  of symbol in a string...
  //This way I can easily get the parent Directory path... unless the parent is the
  root of the disk...
  tempDir := Copy( edFolderPath.Text, 1,LastChar(edFolderPath.Text, '\')-1);
  SetFileAttributes(PChar(tempDir), FILE_ATTRIBUTE_SYSTEM);
end;
```

Last Modified: 13-JUL-01

## Article #27879: Using a TreeView in conjunction with a Master Detail relationship.

You can download a sample project from Code Central [here](#). Using a TTreeView to illustrate the relationship of master-detail query can greatly add to the user interface of you application. Not only does this process add to the UI, but it is also easy to do!

Lets say there are two tables on a form, the first table is linked to DBDemos->Customer and the other table is linked to DBDemos->Orders. (The shared key is on "CustNo")

### Filling the TreeView with the Master Information

We'd like the TreeView to list all of the Customer Company Names, so in the main forms **OnShow()** method we can iterate through the master dataset and fill the TreeView. Here is the code for the Form's OnShow event.

```
procedure TMainFrm.FormShow(Sender: TObject);
var
  tnCompany: TTreeNode;
begin
  tbCustomer.Active := True;
  tbOrders.Active := True;
  tbCustomer.DisableControls;
  tbCustomer.First;
  while not tbCustomer.Eof do
  begin
    tnCompany := TreeView1.Items.Add(nil, tbCustomer.Fields[1].AsString);
    //tnCompany.Data := ...Could set the pointer to point to something relevant to
the field...

    //When user clicks on it we will try to expand and fill with orders...
    //This gives the appearance that there are sub items to add...
    TreeView1.Items.AddChild(tnCompany, ' ');
    tbCustomer.Next;
  end;
  tbCustomer.First;
  tbCustomer.EnableControls;
end;
```

You'll notice that no assignment to the Node's **Data** property was made. The Data pointer can be used to store custom data, which in this case could have been used to hold record information. To do this one could create a custom class, or record, and copy the data into an instance of the class/record, and then point the Data property at the newly created instance. You'll also notice that we add a child node underneath the current node. This is done to give the appearance that there is additional information to be gotten from the current node.

### Adding Detail Information to Parent Nodes.



When the user clicks on the plus sign of a particular customer in the TreeView, we would like an event to grab the orders for that customer. In this example the TreeView's **OnExpanding()** is the perfect event to use. The idea is to locate the record in the master dataset and then iterate through the detail's dataset adding child nodes in the TreeView for each new record we encounter. Here is the code for the **OnExpanding** event.

```

procedure TForm1.TreeView1Expanding(Sender: TObject; Node: TTreeNode;
  var AllowExpansion: Boolean);
var
  oldCursor: TCursor;
  bkmrkCust, bkmrkOrd: TBookmark; //needed to restore original state.
begin
  //Save state..
  oldCursor := Screen.Cursor;
  bkmrkCust := tbCustomer.GetBookmark;
  bkmrkOrd := tbOrders.GetBookmark;

  //Begin query
  Screen.Cursor := crSQLWait;
  if tbCustomer.Locate('Company',VarArrayOf([Node.Text]),[loCaseInsensitive]) then
  begin
    // Note this doesn't take car of the times when a user clicks on the same node
more that once....
    //need to add logic to determine whether or not a node has been filled with
data...
    //this could be done with the data property
    Node.Item[0].Delete; //Remove the temporary subitem created earlier...

    while not tbOrders.Eof do
    begin
      //Whatever node we are at we can now add child nodes... until the end of the
dataset is reached
      TreeView1.Items.AddChild(node,tbOrders.Fields[0].AsString);
      tbOrders.Next;
    end;
  end
  else
    Node.Item[0].Delete; //In case there were no items for this particular customer

  //Return Datasets to original state
  tbCustomer.GotoBookmark(bkmrkCust);
  tbOrders.GotoBookmark(bkmrkOrd);

  //free up resources
  tbCustomer.FreeBookmark(bkmrkCust);
  tbOrders.FreeBookmark(bkmrkOrd);

  Screen.Cursor := oldCursor;
  TreeView1.Invalidate; //Be sure to repaint the TreeView when done...
end;

```

Notice that there is no logic currently in place to determine if the node has already been expanded or not, which will cause the child list to grow each time it is expanded. Again we could use the result from **AddChild()** to store record specific data in the node's Data property.



## Article #27953: Retrieving a list of installed Applications on Windows

Here is a method for retrieving a list of installed applications on a particular machine running a Windows OS.

1. Start up Delphi.
2. Select File | New Application.
3. Add Registry to the uses of your new Unit.
4. Place a TListBox (ListBox1) component on your form.
5. Place a TButton (Button1) in your form.
6. Place the following code in the OnClick event of the Button1:

```
procedure TForm1.Button1Click(Sender: TObject);  
const  
    REGKEYAPPS = '\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall';  
  
var  
    reg      : TRegistry;  
    List1    : TStringList;  
    List2    : TStringList;  
    i, n     : integer;  
  
begin  
    reg      := TRegistry.Create;  
    List1    := TStringList.Create;  
    List2    := TStringList.Create;  
  
    {Load all the subkeys}  
    with reg do  
    begin  
        RootKey := HKEY_LOCAL_MACHINE;  
        OpenKey(REGKEYAPPS, false);  
        GetKeyNames(List1);  
    end;  
    {Load all the Value Names}  
    for i := 0 to List1.Count -1 do
```

**begin**

```
reg.OpenKey(REGKEYAPPS + ' ' + List1.Strings[i],false);  
reg.GetValueNames(List2);
```

```
{We will show only if there is 'DisplayName'}
```

```
n := List2.IndexOf('DisplayName');
```

```
if (n <> -1) and (List2.IndexOf('UninstallString') <> -1) then
```

```
begin
```

```
ListBox1.Items.Add((reg.ReadString(List2.Strings[n])));
```

```
end;
```

```
end;
```

```
List.Free;
```

```
List2.Free;
```

```
reg.CloseKey;
```

```
reg.Destroy;
```

```
end;
```

Last Modified: 01-NOV-01

## Article #28258: Working with TMultiReadExclusiveWriteSynchronizer

### Working with TMultiReadExclusiveWriteSynchronizer

By [Justin Swett](#)

Delphi Developer Support

There will undoubtedly come a time when the need for developing a multi threaded application is a necessary. In some cases the use of a TMultiReadExclusiveWriteSynchronizer (TMREWS for short) will improve the performance of your threads, and therefore improve the performance of your application. (As a parenthetical side note, don't you just love saying "TMultiReadExclusiveWriteSynchronizer!") This article will cover the basics of using a TMREWS and point out some not so obvious pitfalls to avoid. You can [download](#) the complete project here.

#### When to use a TMultiReadExclusiveWriteSynchronizer

As the name of the component implies, a TMREWS should be used when there is much reading and little writing to be done. In addition, the READ operations should be kept to a MINIMUM otherwise your write threads could be left waiting indefinitely. If there are going to be equal amounts of READING and WRITING then you might want to consider using critical sections instead.

#### Creating a Thread Class that uses the TMREWS

One of the first things to do is declare a TMREWS. If you downloaded the complete project, you'll see a global lock declared inside the implementation section of the thread unit. It doesn't matter where the lock is declared so long as the following criteria are met:

- Threads can access the lock.
- The lock has been instantiated.  
(Use initialization section and finalization for this)
- Threads can access memory that you are trying to read/write to.

Once the lock has been declared you can create your thread class. You will need to override the Execute procedure, which is where you will be using TMREWS lock. Here is an example of an execute procedure using the TMREWS:

```
procedure TWorkerThread.Execute;  
begin  
  { Place thread code here }  
  while not Terminated do  
  begin  
    //Updates form that thread is "sleeping"  
    Synchronize(Sleeping);  
    //Sleep thread momentarily, for illustrative purposes  
    Pause;  
  
    //Update form that thread is "waiting"
```

```

Synchronize(Waiting);

//check to make sure thread hasn't been terminated
if Terminated then exit;

//if reader then
if (fThreadType = ttRead) and (not Terminated) then
begin
    globalLock.BeginRead;
    //wrap endRead in try finally... otherwise deadlock could occur
    try
        Synchronize(Reading);    //Still important to synchronize VCL calls
        Pause;                    //represents thread work being done
        Synchronize(DoneReading);
    finally
        globalLock.EndRead;
    end;
end
//if writer then
else if (fThreadType = ttWrite) and (not Terminated) then
begin
    globalLock.BeginWrite;
    //wrap in a try finally to prevent deadlock
    try
        Synchronize(Writing);
        Pause; // represents data manipulation
        Synchronize(DoneWriting);
    finally
        globalLock.EndWrite;
    end
else
    exit;
end;
end;

```

Notice that the work done after Begin'X' is wrapped in a try finally. This ensures that that the appropriate End'X' will get called if any exceptions are raised . If an exception is raised then other threads might not be able obtain locks. If there is no chance of an exception being raised, then removing the try finally for performance reasons would be acceptable.

Make sure any memory being referenced by your threads is sandwiched with calls to the TMREWS BeginREAD/WRITE and EndREAD/WRITE, otherwise your work of making your data thread safe is forfeited.

## **"Lock it up Chris, I'm about to..."**

TMREWS allows you to promote a READ lock to a WRITE lock if the same thread were to make such a request. Although this can be advantageous it can also cause two threads to lock up on each other. For example, two instances of a thread type are instantiated with an Execute method which promotes a read lock to a write lock. If the first thread calls BeginWrite after the second thread has called BeginWrite then the first thread is blocked. The second thread will at some point attempt to call BeginWrite, which will also get blocked. Now the two threads are waiting for the other to release the lock.

Another situation will arise where several Read threads are running and perhaps just a few Write threads. Whenever a Read thread calls BeginRead a Read lock is granted... so long as there isn't a Write lock currently out. If a Write thread were to call BeginWrite while a Read thread had the lock it could potentially be wait indefinitely, while other read threads obtain Read locks. Requests for locks are not queued, so this isn't a bug and is in fact as designed.

## **Summary**

Using the TMultiReadExclusiveWriteSynchronizer (just had to say it one more time) is fairly simple. To use them successfully you should call BeginRead/Write matched with the appropriate EndRead/Write. You should also check you read locks out for as little time as possible if you want your write threads to complete their execution.

Last Modified: 30-JAN-02

## Article #28325: Encrypting and Decrypting strings in Delphi.

### QUESTION:

How can I encrypt and decrypt string values?

### ANSWER:

There are a number of ways to do this, below is a code sample that demonstrates one of the ways. Start with a form, and drop three Edit components, and two buttons. Add the EnDeCrypt function to the TForm1 class declaration in the type portion of the Unit. Omitting the TForm. section of the declaration. Then add the function in the implementation section of the Unit as shown below. Now double click each of the buttons and add the code as shown below. Run the project, in the Edit1 text area type in a word and click Button1. Now click Button2, you should see your word encrypted in Edit2 text and decrypted in Edit3 text. This should give you a basic idea on how Encryption works.

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Button2: TButton;
    Edit3: TEdit;
    procedure Button1Click(Sender: TObject);
    function EnDeCrypt(const Value : String) : String;
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

function TForm1.EnDeCrypt(const Value : String) : String;
var
  CharIndex : integer;
```

```
begin
  Result := Value;
  for CharIndex := 1 to Length(Value) do
    Result[CharIndex] := chr(not(ord(Value[CharIndex])));
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit2.Text := EnDeCrypt(Edit1.Text);
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit3.Text := EnDeCrypt(Edit2.Text);
end;
```

Last Modified: 05-FEB-02

## Article #28416: How to debug a COM+ component in Delphi.

Assuming you have already created a COM+ component in Delphi, these are steps for debugging:

1. In Delphi, go into the menu option Project | Options | Linker and setting the Include remote debug symbols option.
2. Go into the menu option Run | Parameters.  
On the Local tab, under Host Application enter **C:\WINNT\system32\dllhost.exe**  
Under Parameters enter **/ProcessID:{GUID}**  
where *GUID* is the Application ID for your application.

This GUID can be obtained by going into Control Panel | Administrative Tools | Component Services  
Locate the COM+ application under Component Services | Computer and open its properties.  
On the first page below, the description box, is the Application ID.

After you have entered these parameters you can run the app (F9) to begin debugging.

Last Modified: 28-FEB-02



## Article #28452: How to control user input into a TEdit component.

### QUESTION:

How can I control user input to a TEdit component?

### ANSWER:

There are a number of ways to do this. Below the code samples explore restricting user input to only allow numbers to be added into the TEdit component. You can extend the functionality to include other keys, incorporate other restrictions. The first sample only allows positive integers to be entered into the TEdit component. Add the code below to the OnKeyPress event handler.

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);  
begin  
  if not (Key in [#8, '0'..'9']) then  
  begin //Backspace is #8.  
    ShowMessage('Please enter the corect input');  
    Key := #0; //Pass in #0 to disregard the key after it's use.  
  end;  
end;
```

If you want to add more functionality, you could also allow users to enter Decimal numbers. Below is the code sample that does this.

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);  
begin  
  if not (Key in [#8, '0'..'9', DecimalSeparator]) then  
  begin  
    ShowMessage('Please enter a number. Wrong input: ' + Key);  
    Key := #0;  
  end//End first if.  
  else  
  if (Key = DecimalSeparator) and (Pos(Key, Edit1.Text) > 0) then  
  begin  
    ShowMessage('Please enter a number. Wrong input: ' + Key);  
    Key := #0;  
  end//End second if.  
  
end;
```

Now you can add more functionality, by allowing the user to add Negative numbers. Below is a code sample that does this.

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);  
begin  
if not (Key in [#8, '0'..'9', '-', DecimalSeparator]) then
```

```
begin
  ShowMessage('Wrong input' + Key);
  Key := #0;
end;//End first if.
else
  if ((Key = DecimalSeparator) or (Key = '-')) and (Pos(Key, Edit1.Text) > 0)
then
  begin
    ShowMessage('Wrong input ' + Key);
    Key := #0;
  end;//End second if.
  else
    if (Key = '-') and (Edit1.SelStart <> 0) then
    begin
      ShowMessage('Please add at the beginning of the number: ' + Key);
      Key := #0;
    end;//End third if.
  end;
```

Last Modified: 05-MAR-02

## Article #28502: How to use a parameterized query with a Remote Data Service.

There is a demo that comes with Delphi which uses TRDSCConnection to connect to the DBDEMOS database using ADO. We are going to modify the server part of this by adding a Command object, as well as few other objects. We will be using Command.Execute instead of Result.Open. It is possible to execute a query without using a Command object by passing a query string to the Execute method of a Connection object or to the Open method of a Recordset object, but a Command object is required when you want to use query parameters.

Below is the code used to create a parameterized query using an ADO Command object. To test out this code just open the RDS demo: C:\Program Files\Borland\Delphi6\Demos\Ado\Rds. Replace the code for function TRDSAppServer.Get\_Employee: \_Recordset; with the code below.

```
function TRDSAppServer.Get_Employee: _Recordset;
var
  ConnStr: WideString;
  CMD : _command;
  vRecsAffected, vParams: OleVariant;
  aConnection: _Connection;
begin
  ConnStr := 'FILE NAME=' + DataLinkDir + 'DBDEMOS.UDL';
  aConnection := CoConnection.Create;
  aConnection.ConnectionString := ConnStr;
  aConnection.Open(ConnStr, '', '', -1);
  CMD := CoCommand.Create;
  CMD.Set_ActiveConnection(aConnection);
  CMD.CommandType := adCmdText;
  CMD.CommandText := 'SELECT * FROM Employee WHERE EmpNo = :EmpNo';
  Result := CoRecordSet.Create;
  Result.CursorLocation := adUseClient;
  Result.CursorType := adOpenForwardOnly;
  Result.LockType := adLockOptimistic;
  vRecsAffected := -1;
  vParams := CMD.CreateParameter('EmpNo', adInteger, adParamInput, 0,
4);
  Result := CMD.Execute(vRecsAffected, vParams, adCmdText);
end;
```

You can use code completion in Delphi to find the meaning of each of the parameters - just hold the cursor over the procedure name. The key to using a query with ADO, whether you are using the result set's open method or the Command object, is the command option adCmdText. Stored procedures can

also be used to execute queries. In this case calling open with the command option of adCmdStoredProc will work, and will not require the Command object.

When running this program you may get the error "Dataset does not support bookmarks, which are required for multi-record data controls". As far as I can tell, this is a problem with the OLE provider, not with Delphi. To avoid this do not use a DBGrid. Use other data-aware components instead (just drag the fields from the fields editor onto the form).

For more information on ADO Commands see the [ADO API Reference](#) on the Microsoft web site.

For more about RDS go [here](#)

.

Last Modified: 14-MAR-02

## Article #28614: Configuring NSAPI servers to handle ISAPI DLL requests

For ISAPI DLLs to work with an NSAPI server such as Netscape server and IPlanet, a MIME type must be assigned to the ISAPIs. Borland provides an ISAPI adapter DLL named ISAPITER.DLL located in your C++Builder or Delphi directory under `\bin\NetscpXX` (where X is the server version). The server must be set to use this DLL to handle requests for ISAPI DLLs. The following steps accomplish this task:

1. Copy ISAPITER.DLL from the Delphi or C++Builder `bin\netscpXX` directory to `C:\Netscape\Server\Nsapi\Examples` (actual directories may vary based on server versions and installation locations).
2. In the server's OBJ.CONF file, find this line:

```
fn=load-types mime-types=mime.types
```

and insert the following line after it:

```
Init funcs="handle-isapi,check-isapi,log-isapi" fn="load-modules"  
shlib="c:/netscape/server/nsapi/examples/ISAPIter.dll"
```

3. In OBJ.CONF, find the section:

```
<Object name=default>
```

Within this section, find the line reading:

```
NameTrans fn=document-root root="C:/Netscape/Server/docs"
```

and insert this line directly above it:

```
NameTrans from="/scripts" fn="pfx2dir" dir="C:/Netscape/Server/  
docs/scripts"  
name="isapi"
```

4. At the end of OBJ.CONF insert:

```
<Object name="isapi">  
  PathCheck fn="check-isapi"  
  ObjectType fn="force-type" type="magnus-internal/isapi"  
  Service fn="handle-isapi"  
</Object>
```

5. Append the following line to the end of the server's MIME.TYPES file:

```
type=magnus-internal/isapi exts=dll
```

Last Modified: 12-APR-02

## Article #28663: How to Surface Errors in WebSnap and Still Retain Field Values.

by,  
[Justin Swett](#)

A common task in web applications is validating data whenever a user submits something to your site. In most cases the web application does some kind of processing on the data and either rejects the data, or hums along merrily.

In cases where the user enters invalid data it is often required to relate this problem back to the user. In some cases some of the data is correct and doesn't need to be reentered. From a user's standpoint it makes sense to retain field values that don't need to be reentered.

WebSnap offers a way to validate the data and surface errors when they occur, as well as retain field values. This article will cover the steps necessary to perform such a task in Delphi. A sample program can be downloaded [here](#).

**Some knowledge of WebSnap would be advisable before reading this document. It is assumed that you know how to create AdapterFields and AdapterActions as well as how to incorporate them in the server script through the Web Page Editor.**

You WebSnap application should have a WebModule with an AdapterPageProducer and Adapter on it. The Adapter will need at least one AdapterField and one AdapterAction. Once you have created an AdapterForm you should "Add Component" (to the AdapterForm) and add an AdapterErrorList. To validate the data we will put code in the Adapter's OnBeforeExecute event and add errors to the AdapterErrorList. This event fires before the AdapterAction's Execute method fires and provides a way to prevent the Execute method from executing all together. Below is a example of some code that would be executed in the OnBeforeExecute event:

```
procedure TSomeForm.Adapter1BeforeExecuteAction(Sender, Action: TObject;
  Params: TStrings; var Handled: Boolean);
begin
  //BeforeExecute is called each time an Action gets executed.
  //for this reason we have to check which action we are in and make sure it is the
one we
  //want to validate.
  if TAdapterAction(Action).ActionName = 'adpActSubmit' then
  begin
    if (adpFldSomeInt.ActionValue <> nil) and (adpFldSomeInt.ActionValue.ValueCount
  > 0) then
    begin
      try
        //Simple check to see if field can be converted to an integer...
        StrToInt( adpFldSomeInt.ActionValue.Values[0] );
      except on E: Exception do
        begin
          //if there is an exception raised during the conversion then we can
surface this
          //through our Adapter.Errors property
          Adapter1.Errors.AddError(
            Format('%s is not an integer', [adpFldSomeInt.ActionValue.Values
  [0]]));
          //Next we would like to reflect the data that the user entered
          Adapter1.EchoActionFieldValues := True;
          //Finally stop the chain of events here to prevent the actual action from
executing...
          Handled := True;
        end
      end
    end
  end
```

```
        end;
    end;
end
else
begin
    Adapter1.Errors.AddError('Please enter an integer!');
    Adapter1.EchoActionFieldValues := True;
    Handled := True;
end;
end;
end;
```

In the example we tested the ActionValue of the adapter field to see if it was a valid integer. If for some reason the value couldn't be converted to an integer then an exception would have been raised. When we handle the exception we can add strings to the Adapter's Errors property and notify the user of their mistake(s).

The next step is to preserve the Field Values, which is done with the Adapter's EchoActionFieldValues property. Setting this property to True will display the values that were initially entered on the form the next time the user sees it.

Finally, the Handled property must be set to True if an error occurs. If this variable is set to False then the AdapterAction's Execute method will be called. If this variable is set to True the chain of events stops after OnBeforeExecute has finished. Validation can be done from within the Execute method, however you will not be able to surface any errors. You'll also notice in the Adapter's OnBeforeExecute that a parameter named Action was passed in. This variable can be cast to a TAdapterAction and the ActionName can be retrieved. This is useful since OnBeforeExecute will fire for every Action and you may only want to perform certain checks for certain actions.

You will now be able to create web forms and validate the data.

Last Modified: 10-MAY-02



## Article #28936: How to change the default printer from Delphi.

Changing the default printer in Windows 2000 requires changing the registry key HKEY\_CURRENT\_USER\Software\Microsoft\Windows NT\CurrentVersion\Windows\Device. This can be done from Delphi using Windows API calls.

Below is an entire unit you can copy. Start a new application and add 2 TButtons, 1 TLabel and 1 TListBox. Save and compile. Then replace all of the code in your Unit1 with the code below. Then be sure to double click each button to set the event handler.

**unit** Unit1;

**interface**

**uses**

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

**const**

MAXPRINTERBUFFER = 8000;

MAXPRINTERNAME = 500;

MAXPRINTERINFO = 50;

**type**

TPrinterBuffer = **array**[0..MAXPRINTERBUFFER - 1] **of** char;

TForm1 = **class**(TForm)

Listbox1: TListBox;

Button2: TButton;

Button1: TButton;

Label1: TLabel;

**procedure** Button2Click(Sender: TObject);

**procedure** GetPrinterNames;

**function** ParseNames(**const** namebuffer: TPrinterBuffer; var startPos: integer): **string**;

**function** SetPrinter(**const** PrinterName : **String**) : boolean;

**procedure** Button1Click(Sender: TObject);

**private**

{ *Private declarations* }

**public**

{ *Public declarations* }

**end**;

**var**

Form1: TForm1;

```
printerNames: TStringList;  
defaultPrinter: integer;
```

## implementation

```
{$R *.DFM}
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
var
```

```
  x : integer;
```

```
begin
```

```
  try
```

```
  for x := 0 to printerNames.Count -1 do begin
```

```
    if ListBox1.Selected[x] then begin
```

```
      if (SetPrinter(ListBox1.Items.Strings[x]))
```

```
      then label1.Caption := 'Printer set to ' + ListBox1.Items.Strings[x]
```

```
      else label1.Caption := 'Printer not set';
```

```
    end;
```

```
  end;
```

```
  except
```

```
    label1.Caption := 'An error occured while setting the printer';
```

```
  end;
```

```
end;
```

```
procedure TForm1.GetPrinterNames;
```

```
var
```

```
  buffer: TPrinterBuffer;
```

```
  currPos: integer;
```

```
  printerName: string;
```

```
begin
```

```
  printerNames.Free;
```

```
  printerNames := TStringList.Create;
```

```
  if GetProfileString(PChar('PrinterPorts'), nil, '', buffer, MAXPRINTERBUFFER) > 0 then
```

```
  begin
```

```
    currPos := 0;
```

```
    while (true) do
```

```
      begin
```

```
        printerName := ParseNames(buffer, currPos);
```

```
        if printerName <> '' then
```

```
          printerNames.Add(printerName)
```

```
      else
```

```
        break;
```

```
      end;
```

```
  end;
```

```
end;
```

```

function TForm1.ParseNames(const namebuffer: TPrinterBuffer;
var startPos: integer): string;
var
  i, j, NameLength: integer;
  str: string;
begin
  result := "";
  if (startPos > High(namebuffer)) or (namebuffer[startPos] = Chr(0))
  then
    exit;
  for i := startPos to High(namebuffer) do begin
    if namebuffer[i] = Chr(0)
    then begin
      nameLength := i - startPos;
      SetLength(str, nameLength);
      for j := 0 to nameLength - 1 do
        str[j+1] := namebuffer[startPos + j];
      result := str;
      startPos := i + 1;
      break;
    end;
  end;
end;

```

```

function TForm1.SetPrinter(const PrinterName: String): boolean;
var
  s2 : string;
  dum1 : Pchar;
  xx, qq : integer;
const
  cs1 : pchar = 'Windows';
  cs2 : pchar = 'Device';
  cs3 : pchar = 'Devices';
  cs4 : pchar = #0;

```

```

begin
  xx := 254;
  GetMem( dum1, xx);
  Result := False;
  try
    qq := GetProfileString( cs3, pchar( printerName ), #0, dum1, xx);
    if (qq > 0) and (trim( strpas( dum1 )) <> "")
    then begin
      s2 := PrinterName + ',' + strpas( dum1 );
    end;
  except
  end;

```

```

while GetProfileString( cs1, cs2, cs4, dum1, xx) > 0 do
  WriteProfileString( cs1, cs2, #0);
WriteProfileString( cs1, cs2, pchar( s2 ));
case Win32Platform of
  VER_PLATFORM_WIN32_NT :
    // SendMessage( HWND_BROADCAST, WM_WININICHANGE, 0, LongInt(cs1));
    // VER_PLATFORM_WIN32_WINDOWS :
    // SendMessage( HWND_BROADCAST, WM_SETTINGCHANGE, 0, LongInt(cs1));
  end;
  Result := True;
end;
finally
  FreeMem( dum1 );
end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  GetPrinterNames;
  Listbox1.Items.AddStrings(PrinterNames);
end;

end.

```

This sample is for Windows 2000 or NT but you should be able to adapt this code to your needs. The call to SendMessage notifies any other apps that are running that the default printer has changed. Sometimes this can cause the app to "freeze" because it is waiting for a reply from each app (which it may never get) so the code is commented out here. You can determine whether you need it or not.

Last Modified: 01-AUG-02

## Article #28956: How to programmatically open and close the CD-Rom drive.

### Question

### How can you use Delphi to open and close the CD-Rom drive?

### ANSWER:

Start Delphi and select File | New | Application. In Unit1.pas add MMSystem to the uses. Drop two buttons on your form. Name the first button **btnOpen** and name the second button **btnClose**. Change the caption of btnOpen to **Open** and the caption of btnClose to **Close**. Add the following code to the btnOpen OnClick event:

```
procedure TForm1.btnOpenClick(Sender: TObject);  
begin  
    mciSendString('Set cdaudio door open wait', nil, 0, 0);  
end;
```

Now add the following code to the btnClose OnClick event:

```
procedure TForm1.btnCloseClick(Sender: TObject);  
begin  
    mciSendString('Set cdaudio door closed wait', nil, 0, 0);  
end;
```

The code I just demonstrated will open up the default CD-Rom drive. What if you have a CD-Rom drive along with a CD-Burner? Here is an example of how you would do that. In the btnOpen OnClick event, place the following code:

```
procedure TForm1.btnOpenClick(Sender: TObject);  
begin  
    // Open drive X: (X being the letter of the drive you wish to open)  
    mciSendString('open cdaudio!X: alias driveX', nil, 0, 0);  
    mciSendString('set driveX door open wait', nil, 0, 0);  
end;
```

Now go ahead and add the following to the btnClose OnClick Event:

```
procedure TForm1.btnCloseClick(Sender: TObject);  
begin  
    // Close drive X:  
    mciSendString('set driveX door closed wait', nil, 0, 0);  
end;
```

Last Modified: 15-AUG-02

## Article #29028: Revamp Your Main Menu with Delphi 7

Revamp your applications using Delphi 7

By

[Justin Swett](#)

Borland Developer Support

Delphi 7 introduces the ability to add Windows XP styling to your ActionToolBar and ActionMainMenuBar through the ActionManager. This document will demonstrate how to use **ActionMainMenuBar** to create a XP styled menu with sub menu's.

### Enable XP Style

The first step is to create a New Application (File | New | Application )and drop an **ActionManager** on the Form. Next you should set the *Style* property of the ActionManager to *XP Style*.

### Creating Menu's and Sub Menu's

With the ActionManager on the Form we can now drop a **ActionMainMenuBar** and begin making our menu system. ActionMainMenuBar is located on the Additional tab of the component palette. The ActionManager comes with Standard Actions, which are useful for many applications, however, the Standard Actions are not fine tuned enough for the needs of every application. In order to make a customized menu system you will need to create a new Category. The Standard Actions serve as an example since they are grouped by category, such as actions listed in the category of File, or Edit.

Lets create a menu for the File category and then add a sub menu called *Some Category*. Double click on the ActionManager component to bring up the default editor. Next Select *New Standard Action*, and choose the *File* category and all of its Actions, then click "OK".

First create a new action from the ActionManager editor.

ListView of the Standard Actions for the ActionManager

Now we can drag and drop the *File* category from the ActionManager's Editor to our ActionMainMenuBar, and Delphi will automatically create a menu for us.

Next lets create a sub menu. First we will want to create a new Category, which can be done by adding a new Action in the ActionManager editor. In the ActionManager editor create a *New Action* and give it a caption, a name, and some event code. Next you can select the category that the action will belong to, if the Category doesn't exist then you can manually type in the Category and it will be automatically created:



Create a New Action and set it's Category Property. If the category doesn't exist then type in a new value for it.

Go ahead and repeat the process to create some more Actions for this Category. Once the Category is complete you can drag and drop it from the **ActionManager** to the **ActionMainMenuBar**. To make a sub menu you can continue the drag process over the existing *File* menu, which will expand the *File* menu, and then drop it somewhere within its contents. The end result is show below, however the resulting colors can be changed by using a **XPColorMap** and setting the *ColorMap* of the

**ActionMainMenuBar.**

An XPColorMap can be used to alter the menu's appearance.